Course Overview
Course Overview

Hi everyone. My name is Richard Seroter, and welcome to my course, Developing Java Microservices with Spring Cloud. I'm a director of product management at Google, a blogger and author, and long-time Pluralsight trainer. I've had the privilege of spending years working with the Spring engineering team when I worked at Pivotal Software, now VMware, and I want to bring to you the many things I learned. In this course, we're going to learn about the key projects in Spring Cloud that make it easier to build scalable, efficient, maintainable microservices. Some of the major things we're going to cover include externalizing your configuration with Spring Cloud Config, building serverless-ready functions with Spring Cloud Function, authorizing access to microservices with Spring Security, and building traceable services using Spring Cloud Sleuth. By the end of this course, you'll know many of the techniques and Spring projects that help you build great microservices and you'll have a great list of areas you can explore further. Before beginning this course, you should be familiar with Java and Spring Boot. We're going to have some fun here. I hope you'll join me on this journey to learn all about building modern microservices with this Developing Java Microservices with Spring Cloud course, here at Pluralsight.

Introduction to Microservices, Spring Boot, and Spring Cloud

Version Check

Introduction

Hey everyone. My name is Richard Seroter, and welcome to this course about developing Java-based microservices using Spring Cloud. In this module, we're going to discuss some of the core concepts of microservices, what Spring Cloud and Spring Boot are all about, and what you need to know to get the most out of this course. I'm looking forward to this. We're going to have some fun together. We'll talk mainly about why are microservices architectures popular and what's that all about, what are some of the core characteristics you should really understand about microservices, what is Spring Cloud and what is Spring Boot and how did those come to play when you're trying to build a great architecture, and what are some of the goals and prerequisites for this course to make sure you get the most out of it?

What Are Microservices and Why Are They Popular?

I could probably spend an entire course just listing out different people's definitions of microservices, so let's not do that. Instead, I'll just use Adrian Cockcroft's definition here, he's a VP at Amazon, and he said a microservices architecture is basically loosely coupled service-oriented architecture with bounded context. Basically, this idea of how do I develop a single app as a set of small services. Each one runs in its own process, they communicate in a very lightweight manner with each other. These services are often built around a business capability, and they're independently deployable using typically deployment automation tools. So why are these architectures popular? Why are people embracing this more and more for more traditional designs? But first off, this is a way to

deliver faster change. When I have a monolithic system, sometimes change cycles are connected together. I have few batched releases because I'm updating the whole system at once and therefore I don't do it all the time, all these components are very tightly coupled. With microservices, I want to be able to ship more often, and I can ship more independently because each piece stands alone. I may also do this to get some better availability in my system. I don't want a single sort of bad thing to take down my entire application and have to take hours and hours to get online. So in some cases, when I have more of these distributed components, I can actually tolerate failure better, I have a better pattern around detecting and responding to some of the service interruptions. There could be downsides. I can be less available with microservices if I don't design it well. But the idea is if I can split up some of the responsibilities, I have a chance to make each component available. I think one of the underrated aspects of microservices is the ability to do more fine-grained scaling. This idea, if I have a monolithic system, I only have a few scaling vectors. I can scale the database, maybe the app tier, things like that, but if only one thing needed to be scaled, I'm kind of stuck scaling the entire system. With microservices, I could scale services individually. Here's one service that's overloaded, let's give it more capacity, and I'm not unnecessarily scaling another piece that doesn't need it. So I can be more cost effective and surgical on how I scale my system. Then finally, this is fairly compatible with a DevOps mindset. It can be called one of the first post-DevOps architectures. It's oriented around customers and business value, a service, integrated teams delivering value. So it's a fairly modern architecture to match modern teams.

Core Characteristics of Microservices

All right, let's talk about some of the core characteristics of a microservice. First, these are components exposed to services. They have some sort of service endpoint. They take a business logic component, data processing, and they're exposed to the service that others can consume. They're often tied to a specific domain. The idea of a microservice is often what is the sort of thing this thing is responsible for? How can it be tightly bound to a specific domain; it's not trying to do everything. It's just doing one thing well, so it's typically tied to a single domain, has a single responsibility. Now you also see loose coupling here. How can I make sure that I'm not accidentally sharing fundamental information between these services? I'm communicating through service interfaces. And so if the implementation changes, heck, the database changes beneath the service, technically that shouldn't break anything. So, if I have good loose coupling and I'm really just exchanging contracts and messages between services, I'm not bound to the implementation details of any given service. Another core part of a microservices architecture is it's built to tolerate failure. It assumes that failure happens. Things go wrong, services go offline, databases become unavailable, network partitions happen. How do I tolerate that? How do I have good fallback? How do I make sure I can't take down the whole system if one service goes offline? So you think about this up front with microservices. One of the differences, I think, from the original service-oriented architecture days and SOA of 10, 15 years ago is that now we talk about the idea of continuous delivery. That wasn't something I was talking about in 2008 or 2005. It was still valuable, but the tools have matured. So now in a microservices architecture, we think about, how do I keep delivering this service? It's not just domain-specific services; it's the ability to constantly deliver those to production. Also, the evolution of thinking about independent teams running these microservices. This service might be really important and

has a dedicated product team that comes up with requirements, tests it, delivers it, maintains it. So when you think of a microservices architecture, these are often things that come to mind for me, plenty of other things as well, but these are kind of the important things as that we think about a bounded service, trying to solve a specific thing well that tolerates failure, we ship it often, and a team is responsible for it. Now, lots of questions you can have about a microservices architecture. It's not for every scenario; it's not for every team. But as you do think about what you want to be able to answer when someone asks you, you might have questions about how do I find my service if these things are really dynamic, coming and going, scaling, changing? I probably can't rely on fixed IP addresses and ports anymore. So what do I do with that? How am I supposed to ship these changes continuously? Should every app be turned into a set of microservices? I think the answer's clearly no. Certain things should be monoliths. That makes a ton of sense. You might just have a simple system that I don't need what microservices do. It would be overengineering to do it. Often microservices is a response to an organizational challenge. Teams are having struggles shipping software because they're too coupled, and so I actually do microservices to optimize my organization, sometimes even more than my software, so not always the right answer. Can I do this if my team isn't arranged in a product format, if I'm not doing DevOps, if I'm not doing CD? Technically the answer's yes. I don't have to have a product team run the microservice. How do I maintain consistency at scale with configurations? If I'm elastic with all these services, should I be putting configuration somewhere different so I can make changes to this expanding and contracting fleet of services? Is there a single stack, is there a single product stack that you should be using, programming language, frameworks? I think the answer's no. You can do this in all sorts of programming languages and frameworks and things like that. There's no one stack that helps you build microservices. What's the right way to secure these, if we're moving into a more modern, zero-trust model, where I'm not just implicitly trusting every service that might be calling me? How do I do modern security? Across platforms? Might be a Linux and Windows service, might be a different environment, might be in different physical locations. How do I think about authentication and authorization? Is the monolithic app just simpler? Absolutely. Sometimes it is. Your first version of your software should probably be a monolith. Makes a ton of sense, because you have no idea what the seams are to turn into microservices yet, or you might be overengineering it, planning for resilience or throughput or velocity that you don't even need. So in many cases, absolutely, you should be starting with monoliths. And how do I troubleshoot problems? All of a sudden, now there's all sorts of components, all sorts of systems. I don't know where something went wrong. How do I actually figure out where my performance problem is or what's throwing an error? You have to be thinking about what is your toolchain here? And then finally, how do I keep a poor-performing service from taking everything down? So maybe I like this idea of distributed services, but what if everyone's depending on this one service and that thing becomes slow or goes offline? Don't I have a cascading problem throughout my architecture? So this is a software engineering approach. It requires forethought, commitment to the skills development and things like that. So I'm glad you're taking this course. This is a journey; this isn't just a simple thing to do.

What Is Spring Cloud?

Okay, so microservices can be hard. There can be a lot of tricky things to figure out. What's going to make that easier? So this is where Spring Cloud comes in. I think this is an interesting

set of infrastructure, if you will, for helping you build and manage microservices. So what is it? Well, this was released a number of years ago, and it implements a lot of common distributed system patterns. That's what's kind of exciting here is you're seeing a number of patterns implemented into a framework, so you don't have to manually figure this stuff out. So it gives you a lot of best practices, kind of an opinionated way, for the right ways to do a lot of things with microservices. It's included a lot of originally Netflix open source technology. Some of that has gone away over time. But you see a lot of industry-standard technology for service discovery, a number of things. So it's not just a Spring-only project. It's also pulled in a number of interesting things from the community. This is all open source. There's nothing open core or proprietary about anything in Spring Cloud. You and I can use this for free, add it to our projects, deploy it. It's a great open-source project that's very healthy, continually updated, a lot of people participate. Now of course, this is optimized for Spring Boot apps, and we'll talk about Spring Boot in a moment, but this is really for Java and Spring apps. You can use this with other languages and frameworks, but really it caters mostly to Spring Boot. And again, the power here is I can run this in a public cloud, I can run this at my data center, I could run this at the edge, I can run this on different operating systems. So you get a lot of power here by having this sort of framework that's designed for building very resilient, scalable, cloud-ready applications. Now what's in Spring Cloud? A lot of stuff. So we're not going to see every project in this course because this would be a very, very long course. But there's a number of really important things, from how do I manage configuration? How do I still use some of the Netflix technology? What do I do for service discovery using console or modern security? How do I do tracing or implement functions in a streamlined way? How do I actually process data? How do I do API gateway work, and then how do I talk to individual clouds? Spring Cloud has even a number of other projects I haven't listed here. Our course is going to focus really on these four areas of configuration, security, tracing, and building functions, as we think of developing our microservices. In other courses, you'll see more about the actual coordination of microservices. But this one's about how do we build these microservices? There's a big catalog of things. There's other tools for testing, things like Spring Cloud Contract. Really, really rich portfolio, a lot of really exciting things in here.

What Is Spring Boot?

So let's back up and talk about Spring Boot. What's Spring Boot about because it's a huge part of how Spring Cloud works. First, the idea of Spring Boot is how do you get an opinionated runtime for Spring and the Spring Framework, which has been around almost two decades. As you think about a robust Java-based framework for helping you build applications, Spring's been terrific. Spring Boot came out around 2014 as a way to do convention, not configuration. How do you have a bunch of opinionated defaults that get activated? Now what's powerful though is that those default opinions can be overridden. So it's opinionated out of the box to help you get started really, really quickly, but you can easily make changes to those and override any of those defaults. What's great here is it handles so much of that boilerplate setup, setting up the web server, figuring out configuration, again, activating connections, the third-party systems like messaging systems. So a lot of really powerful things that you're not stuck doing when you start up an application and figuring out dependencies, all that sort of thing. And that dependency management is great, I'm able to just list out my dependencies, and Boot figures out which

versions are compliant with the framework version I'm using. So again, a lot of time I'm saving from not figuring out which versions work with which thing, Spring Boot takes care of that for me. What's also convenient, if you want it, is the ability to embed the app server in a fully executable JAR file. So I could run this JAR file anywhere, I don't have to worry about hosting this in a classic Java app server, instead, I'm getting that full, self-contained experience. What's also really helpful is that you can use actuator endpoints to actually get health metrics for your application really easily. Again, a lot of things that just get lit up automatically. So by simply including a dependency, I immediately light up dozens of health and informational endpoints in my application. So a really, really great way to just turn applications into more fully manageable, rich, best practices following systems without doing a whole lot of work.

Demo: Building a Spring Boot App

Let's actually see Spring Boot in action. First we're going to check out the Spring Initializr site, a really great way to build your Spring applications to start with. Next, we're going to be using Visual Studio Code throughout this course, so we'll just take a look at that and look at it for Spring development. We'll create a brand-new Spring Boot project. We'll edit some property files and just mess around with the code a little bit. We'll add a simple REST endpoint. We'll start and then run the application. And then finally, we'll browse some of those actuator endpoints. You can see the free things lit up automatically for your application. Really, the point of this exercise is to just get a little comfortable with Spring Boot, make sure that our environment works okay, and just kind of get some simple glimpse at what a microservice kind of looks like. Let's jump in. Here we are in the Spring Initializr site. I'm here at start.spring.io, and this is a site where I can use to generate my Spring Cloud projects or Spring Boot projects. Now I'm going to run that application here in Visual Studio Code. Now I've added a number of extensions here that make Spring development a little bit easier. I've installed some language support for Java to make it easier for it to intelligently know what my Java code is and offer suggestions, I've added support for Maven, I've added a set of Spring Boot extension components, some tools for some validation and even integration with the Spring Initializr from right here within Visual Studio Code, and so just a number of tools to make my Java development a little bit easier here in this free editor. Now back on the Spring Initializr, let's go ahead and create a new application here. So first off, I'm using a Maven project. I'm going to pick the Java language and the latest Spring Boot when I'm recording this course. Let's change the group name to com.pluralsight, and let's give the application artifact the name of setup since this is our first exercise here in the course. Now I want to add some dependencies. What kind of dependencies do I want in my Spring Boot application? Let's add one for the web, so I can build web applications, and one for actuator. When doing web, I have a choice of regular Spring web or the more reactive style. We'll start off with the regular one. And I can add those endpoints. And now I have everything I need for my application. Now if you're curious and just want to poke around what would be generated, you can click the Explore button here and actually see what you're about to download, including looking at the POM file, looking at the code, things like that. And you can also generate a link here that you can imagine even having a default Spring Boot project that you might stick on your intranet or something where people could click the link and come to a screen that's already preloaded. In our case, though, we're going to go ahead and generate the application and stick it into the folder associated with this course. So if you

downloaded the code that came with the course, you'll see that every single module has a folder and a before and after folder. We'll go ahead and drop this into the before folder. So if I go back onto my drive now, I now have a setup folder here in that before folder, and I'll delete the old ZIP. And here I have this sample project downloaded. So let me go into Visual Studio Code and open this folder. So I can open my POM file, look at my dependencies, and I can see I have my dependencies on actuator, web, and then I'm good to go. I'm going to open my application properties files and set a couple of properties. I'm going to set this Spring application name, which port I want to use, and light up all of those actuator endpoints. Now let's go into the code class and actually add a simple REST controller and expose the service as a REST endpoint. I'm going to come into my class, add another class in here to indicate our controller, and then I'll add an operation to it. At this point, I have everything I need to actually have a REST endpoint. I can simply start this application, and I should be able to browse this REST endpoint. Within Visual Studio Code, it's easy to get a terminal that's pointing directly at this folder, and I can start this up with a Maven command. Running that command compiles our apps, starts it up, and it should start on the port 8080 that we indicated. I'm going to now jump into Postman, a tool I can use to test APIs. From within here, I can call any endpoint, use any HTTP verb, which is really handy. So I'm going to do a GET request to my greeting endpoint, and when I send that request, sure enough, I get my greeting back. Let's also try out some of those actuator endpoints. If I hit the root actuator endpoint, it shows me all of the ones that are available, so I can browse which beans have been activated, I can look at the health information, I can look at some basic info, configuration properties, environment variables, all kinds of amazing stuff that just immediately gets lit up by my application. So, for example, health. It shows that the status of this is up. Great! If I want to look at the beans enabled, I can see how they've been scoped, what type they are, and what sort of beans are available to my application really handy for troubleshooting. I can look at environment variables and things that we loaded up, like the server port. And then finally I can look at mappings here too and I can actually see which services I've exposed. So that's a quick look at building a brand-new Spring Boot application, exposing a REST endpoint, and even browsing some of the actuator endpoints

Core Application and Prerequisites for This Course

Now let's talk about whether we're going to build together? Let's go ahead and pretend that we're working on a set of systems to support paid roads for cars, toll roads and highways, things like that. In this course, we're going to pull rates for driving in a given place, we'll show how to process different information and functions, we'll authorize some access to sensitive information, and they we'll analyze some driver behavior. All of this will come together in ways that we're going to flex a lot of different parts of Spring Cloud. All right, so what are some goals for this course I hope you have and that I have for you? I want you to understand some modern microservices patterns, thinking about the things we've talked about already in this module, thinking about the ways microservices work, the way they interact, the way you should process information with them. I want you to explore how to build systems that depend on Spring Cloud. We're not going to use all of Spring Cloud, but I want you to be comfortable with Spring Boot and Spring Cloud throughout this. So we're going to mess around with a lot of different components to give you a real sense of, frankly, where to keep exploring. Because my goal is to help you, really, learn how to configure and extend Spring Cloud and then inspire you to keep digging in. Every single thing that we talk

about in this course could be its own course. They're very dense topics. So I'm going to focus on a lot of the highlights. We're going to build a number of things to show you how they work, and hopefully I inspire you to keep going and learn even more about these projects. Now, what should you know to be successful with this course? Definitely a few things. You should have some knowledge of Java and some object-oriented programming thinking. I'm not going to do an intro to Java here. I'm going to assume that you understand the basic syntax, you understand some of the concepts around build systems like Maven, that you understand how beans work, at least enough of the basics, where, when I show you something, you're not completely baffled because I showed you a core Java concept. So don't have to be an expert by any stretch, but you should be at least comfortable with some of the basic ideas. You should also know a bit of the Spring framework. Again, this isn't an intro to Spring. There's some amazing courses at Pluralsight they can do that for you. So I'm going to assume a number of things that you do know. Again, should be easy to follow along. Even if you haven't done much Spring before, you're going to be in great shape, but you should have some familiarity with what it's about as a dependency injection framework. You'll need some Java-friendly development environment for coding. I'm going to be using Visual Studio Code here. You could use IntelliJ, you could use Eclipse, you could use all sorts of things. But I'm going to suggest you use Visual Studio Code for this, just so we're doing the same thing together. And I'm going to assume you have some access to Docker. So we're going to be using some container images to run RabbitMQ, to run some other components, so I'm going to assume you have Docker on your desktop; you're able to run container images. We'll do some other things just purely locally, but there are some systems like RabbitMQ that I want to pull in through a Docker image to make setup much easier. So what do you need on your workstation? Let's review. I'm going to suggest you have Visual Studio Code with some of those Java and Spring extensions I showed you. I'm going to assume that you have the Maven build manager on your system. You might use Gradle, there might be other things you're familiar with, but I'm going to be using Maven a lot here. Go ahead and load Postman. It's a great API testing tool. If you have something else, it's fine. Postman makes it easy. I mentioned you'll need to have Docker for running containerized apps. Again, we'll be using things like RabbitMQ. You don't have to really know that system very much, but you will need to know Docker. And you'll also want to have a GitHub account, as we're going to be loading configurations into a GitHub repo, and I want you to have access to that, as well as in the security module, we're going to mess around with a GitHub account. So, all those things, if you have those in place, you're going to be in great shape for this course.

Summary

All right, so in this module, we talked about the popularity of microservices architectures. It's happening because we want to ship faster, we want to ship more code, we want to make sure we're able to get better feedback from customers. So it often comes down to speed, stability, other reasons we care about microservices. And really, this is all about having more discrete units of compute specifically tied to domains, often continuously delivered and fault tolerant. So some of these main characteristics of microservices, call things whatever you want, we don't have to care about the names. But the idea is we're building more resilient, smaller bits of code that we can ship more often. Spring Cloud and Spring Boot make it easier to build microservices. No doubt about it, and I hope you feel that way when you're done with this course. But this will make it much easier, as we talked about, to have some of

this automation and scaffolding set up for you as you build your applications and you're not stuck figuring out how to turn all these things on. If you know some Java, if you know how to use Docker, you're going to be in great shape. We're going to have a lot of fun messing around with technology, trying out a number of new things. I think you'll learn a ton, and I can't wait to get going. See you in the next module.

Simplifying Environment Management with Centralized Configuration

Introduction

Hey there. My name's Richard Seroter. Welcome to this next module in the course about building Java microservices with Spring Cloud. In the last module, we talked about the value and challenges with microservices, and here we're going to look at the topic of configuration. First, we'll talk about the role of configuration in microservices. Why does it matter? What's the problem we're trying to solve? What are some of the challenges with the status quo, maybe how you've been doing things now and something that you could upgrade? I'll describe the Spring Cloud Config project, what is it all about? How does it work? We'll talk about creating a configuration server and what that looks like. And then we'll look at how do you consume configurations in apps? So we're going to play around with different types of configurations, get you some comfort and experience with the system and how it works.

The Role of Configuration in Microservices

What role does configuration management play in a microservices architecture? Let's look at it. Why do you care about configuration? First, this is one way to remove environmental settings from your compiled code. These could be things that are specific to your dev environment, maybe it's connection string to test or to production, and so when you're putting that into your code, whether that's in a config file or things like that, it's in essence part of your code. To make a change, you have to redeploy that application package because the configuration is stored within there. So, one of the goals is trying to get some of those settings from each environment out of the code and have something where your code base is uniform whether you're running in dev or test. Another valuable thing here is the ability to change runtime behavior. Think of logging levels, feature toggles, so configuration helps services change their behavior pretty easily and quickly. That's possible if you use a configuration store that supports even refreshing those values easily. So, this is a great way to change behavior without having to redeploy your app. What's also really powerful here is enabling consistency across your services. If you have a microservices architecture that scales really quickly and expands and contracts, then the question is how do I know that each instance has the right configuration? Maybe that instance I just stood up uses a new one and there's old ones, and I'm out of sync. How do I fix that? In a good architecture like this I need some confidence that everything is uniform. You might use configuration as a caching mechanism to reduce some of the load on your database. You could store certain configuration data in the database, of course, but there also might be reference data (especially if it doesn't change very often) that you'd like to stick in a configuration server and cache it that way.

Problems with the Status Quo

So what is the problem with the status quo? How come the things you're doing right now aren't working in the microservices architecture? Well, let's talk about some of the things that can happen. First, if you're using local configuration files, they can fall out of sync. If I'm packaging those configs in the app itself, A, that could be a security risk, and B, it could force a full deployment just to make a change. I might have different versions out there depending on how things are deployed, so if I have local configuration, that's an issue. You might use environment variables for configuration values. On their own, it's okay, they can be useful to do that, but there's no real history of those. Maybe if it's created as part of a deploy process, you have a history of what environment variables you have set, but it's not really a version control thing, it's not really a configuration store, it's just a very convenient tool, but it's not really a great mechanism for really having a robust configuration story. Another challenge is when configuration changes require a restart of your app. In some cases, you actually have to reboot the service to see any of the configuration changes. It's not always great if I have a system that I don't want to be rebooting all the time. There's also a challenge of sensitive information. So if you have different values for dev, test, performance, production, you could expose that information to people who shouldn't see it. So how do I make sure that I've got configurations that can be kept secure the entire way through? And then finally, there could be some inconsistent usage across teams. Microservices are often associated with a particular org, but at the same time, as things change and each team may do things their own way, you could have a security and management challenge because everyone's handling configurations differently. So how do I actually get some uniformity with some security and make sure I have something fairly dynamic for a very flexible set of microservices? That's what we'd like to do.

About Spring Cloud Config, and Creating a Config Server

Spring Cloud Config could be an answer to some of that. In a nutshell, Spring Cloud Config gives you HTTP access to Git or file-based configurations. It's a way to actually put configurations in a remote store. You're externalizing the configuration in this config server, and applications use a client to pull those configurations and use them locally. Let's talk a little more about what this really is. So when we create the configuration server, first, you're choosing some of your sources. This could be local files, this could be a local or online Git repo, technically some others as well. So first you're going to figure out where do I want to store these configurations? Then I'm going to add some of the configuration files. This is multiple formats accepted, so I might use YAML, I might use JSON, I might use just flat text files. I have a lot of different ways to store configuration data. Then I'm going to build my Spring project. It's really easy to stand up a configuration server. We'll do this here in a few moments. It's one simple annotation and some application property settings, and you're good to go, and you have then a config server that's staying in sync with the upstream config source. And then finally, you have some options to secure that configuration. You might secure access to the config store itself with some role-based access, and you might encrypt the values being stored in the configuration. We'll do all of those things. Now how do you decide your source? That is an important choice. How do I want to store? Do I want local files, do I want to use a Git-based repository? Well, if we're using local files, I'm just pointing to the file system or somewhere even within my project, I can actually pick multiple search

locations, so I can find things that might be sitting in different directories. I don't really have an audit trail here. It's on my local file system. So if I make changes, I don't really know what happened before that, what was the previous version? It does support labeling, so have a different value then that I can add to help me pinpoint exactly which configurations I'd like to load, so it does have that feature. For the file system, the label usually corresponds to the folder that's holding it. You can have placeholders in the URI. So again, as you're trying to do some more dynamic lookups, that works. And you see the native profile here, so that's what you're specifically setting in your configuration. It's looking at the native profile for Spring. That's how it knows to pull the local thing. But this is really for dev/test. Unless you're using a shared reliable file system, which might be the case, then you'd want to use this really just for testing some things out. This isn't really the production-grade option. The production-grade option would point to a Git repo, local or hosted. I can pick multiple repositories, multiple ways to search. We're actually going to do that together. Because it's in a Git repository, I have a full change history. I can see what this configuration value was before, and maybe if it changed to something bad, I can easily roll back. That's very powerful. It also supports labeling. So, again, I have a way that can match to the branch and things like that. I can set a placeholder so I can do some dynamic lookups. I can apply multiple profiles here. Again, we'll do this together, but I have a lot more flexibility here. And I may use a local Git repo for dev/test, and I'm going to use some sort of hosted service for production. So, when you're looking at the two, using a Git-base, and honestly, this supports databases and other destinations as well, you do want to pick something that's highly available and gives you an audit trail. So what are those other repository back ends that you have available to you? Well, you do have a few. So you can store configurations in any JDBC-compliant data store. So if I add Spring JDBC to my classpath, I can pull configurations from a properties table that has a specific set of columns. I can use Redis as well, very popular database. If you add spring-data-redis to the classpath, you can retrieve properties from a hash. You can even use Amazon S3, their object store. just add a dependency to AWS Java SDK, and you can pull from buckets. And it also supports HashiCorp Vault. So if you want to use your secret store to also stash configuration data, you have that option as well. Let's talk a little more about the configuration files themselves. What is holding your configurations? So I mentioned there is native support for YAML, for JSON, and property files. So by default, you could even switch between these via the URL. So you have some flexibility to even store it in one and retrieve it in another. And honestly you can serve out any sort of text file. So whatever you have, you want to serve out XML, you want to serve out other configuration file types, you can do that. The file name, and we'll see this, contains the application name, kind of the Spring app name you want to pull, the profile, and the label name. So these things can map to different things which help you actually store a lot of different configurations and pull the right ones based on which application wants to retrieve it. So again, gives you a lot of ways to pull values from a shared repository. Now, what's important here is that all the matching files are returned. It's a bit of a union. So if it matches three or four different files, you're going to get the combination there. Nesting configurations work as well. I could have folders within folders that have different configurations. It doesn't have to be one flat list. So again, different ways I can store this and mess around with it, and we'll actually do some of this together so you can get a real good sense of it. So what do you need to do to create the config server itself? Well, let's talk about that Spring project. First, you use the Spring Initializr to generate the project or any sort of environment that you'd like to to create a Spring Boot project. You set your POM dependencies here for a Maven project on Spring Cloud Config server and pick the actuator as well. You add the

@EnableConfigServer annotation to the class. And finally, you'll add some application properties to your config server project. You'll set up the port, you'll give it a name, and you'll pick a profile, like native profile, if you were pulling from the local disk.

Demo: Creating a Config Server for Local Files

Let's build something, see how this works a bit. We'll start by creating a new Spring project. We'll then annotate the main class to turn this into a configuration server/ We'll set a handful of application properties, just to make sure this is all wired up correctly. We're going to add some local configuration files, so we're not going to use a Git repo yet. We'll just do some local files and see how it works. We'll go ahead and run that application, and then we'll use a handful of queries to just kind of see what comes back. We're back here at the Spring Initializr, where we are going to create a brand-new Spring Cloud Config server. First off, let's go ahead and set up our artifact name. We're going to call this configserver, and we just need to add a few dependencies here. We're going to choose Config Server and Actuator. (Clicking) Now we'll go ahead and save this down to our downloaded code package. We'll stick this in the before folder. All right, now we've download our application. Let's go ahead and open that in Visual Studio Code. Excellent. So now let's go ahead and annotate our main class and turn this into a configuration server. So let's open up the main class file. All we have to do here is provide an @EnableConfigServer annotation. Great, and as the extent of the code, we actually have to write to light this up, which is pretty impressive. Now we want to create a config directory. Under the resources directory, we're going to add a few configuration files to this directly. So let's go ahead and add the new folder under resources, and we're going to add three different config files here, all very simple. So let's call the first one app1.properties, and the only configuration we're going to put in here is, let's go ahead and put a greeting into each one. Let's copy that file twice so we have app2 and app3.properties. (Typing) We'll say bonjour for the app2.properties, and for app3 we'll say hola. Great, now we have three different property files representing three different applications. This could also be different environments, if we want it as well, but let's keep it like this. Now let's open up our application properties file, the primary one, for this Spring Boot app. Here I just want to set a couple of things. I'm going to set the server port, and I'm going to set the profile to native so it looks for the local files. Terrific. So that's all I need to do to actually build a config server. Right now I have local files, I have all the sort of settings, and I'm telling it to look locally, thanks to setting the active profile to native. Let's go ahead and open up a new terminal and start up this application. I'm going to run the simple Maven command, run the Spring Boot activity here. Great, that started up. And now let's switch over the Postman, and we're going to run a couple queries against this configuration server. So we're going to call it on 8888, the local port, app1, and the default profile here. And as you can see, it pulled back hello. If I switch this to app2, I get back the greeting associated with app2. And as you expect, if I try app3, I get the value there. So I have a config server serving up configuration values that are Spring Boot ready and able to be consumed there.

Working with Config Server URIs

In that demonstration, we looked at using the local file system for our configuration values, which was fine. We got to prove something out. But let's talk a little more about using

a Git integration. First off, if you see the config here, I'm doing a YAML format versus a properties format, so it's a little more readable. In this case, you can see we call out the location of the main Git repo on the URI property. There's also the ability to have a pattern to search any of the subdirectories, which is great. So in this case, I'm searching for any directory that starts with the word station. Make sense? Now I can point to alternate repos, as I mentioned, so I might say, hey, look, if the pattern matches something else, go ahead and drop down to this repository instead. So you have a lot of flexibility on how you query the server. You point out, of course, the location of that alternate repo, and you're good to go. So a lot of properties here. But let's look at some specifics now. So let's think about the endpoints. So if I had, let's say, a GitHub account at the top, based on my user, and remember, we're working in a toll situation where we're processing tolls at the system we're building here, for example. And let's say I'm looking for rates. Well, I might have the application name, /application, that's required. I might have the profile, default profile, could be whatever I label it, and then a label, and this is optional. I don't need that. So let's say that I have a main branch, I have station1 for toll station 1, I have toll station number 2. And under that first one, I've got rates for the development environment, the QA environment, and then the sort of catch all. For station2, I just have the dev environment and the catch-all environment. So let's look at a few scenarios, saying, what if I queried different things, what would come back? So if I queried github.com/, whatever my user is, tolls as the repo, rates, and I ask for the s1rates, s1rates/default. What that would return is two things. First it would give me whatever's in that root application property. Then it would give me what's the base s1 rate, because it doesn't match anything else there. That's the default value, that's what would come back. I would get whatever's in there. What if I asked for s1rates/dev? Well, here I would get the base application properties, I would get the s1rates.properties, the catch-all, and I would also get the -dev properties, so I would get a kind of a combo of those three. What if I asked for s2rates/qa? Well, there is no qa profile, so it would fall back to just the base s2rates.properties file plus the topmost application properties. What if I asked for s3rates/default? There's not even an s3 rates. What's going on there? I would just get that top-level application properties back. So based on how I query the application name, the profile, the label, I get either the base results, I get some sort of union or combination of things. But it's important to know this so you don't get surprised when you get certain values back. You want to make sure you know and expect what you're getting.

Demo: Creating a Config Server that Uses GitHub

we get to some of the fun stuff. So let's go ahead and create a config server, use GitHub as a source for my property files and YAML files, and do some pattern matching. So first we'll create a GitHub repo with all the files that we need. We'll then create a new Spring project. We're going to annotate that main class so it's a config server again. Then we're going to set the Git URL in the application.yaml file associated with that config server, we're going to run that Spring Boot application, and then we're going to experiment with some of the search paths and queries and get a better sense of how does this all behave with a real Git-backed back end. Let's jump into the code. Okay, so we're in my GitHub repo environment. So go to your GitHub account, and what we want to do is create a brand-new repository. I'm calling it pluralsight-spring-cloudconfig-wa, for Washington, where I live, tolls-2. Now I'm doing that because I did a previous version of this course, I don't want to break folks, so I'm creating a copy of that directory. You don't need the -2 at the end, it's

completely up to you. I'm going to go ahead and create that repository. And I'm going to open that in GitHub Desktop and create a local clone, and I'm going to go put that clone into the before directory for our application code that you downloaded. And I'll go ahead and clone it. Now next, what I want to do is copy the files from the before folder into the cloned repo. So in the file system, let's go to the before folder, and here's the one we just cloned. Let's go ahead and grab the files I put into the github-configs directory for you. It's in the tolls folder, and we're going to take all of those and we're going to put them into the cloned repository. Now when I do that and go back to GitHub Desktop, you can see that it's pulling the application properties, I have some s1rates-dev, qa, a base file, all those sorts of things. Let's go ahead and commit those into our directory, and then let's make sure to publish that branch. And to prove that that worked, let's go back to the web version. And if I refresh my directory, I now see that I have the application property, station1 and station2, here in this directory. So now I have a Git repo ready to go with all the configuration values that I need. Now let's create a brand-new Spring starter project. So I'm going to go back to the Spring Initializr, and now we want to create a configserver-git. Keep the exact same dependencies, Config Server and Actuator, and click Finish and let's download this. I'm going to save it to the before folder again. Now let's open that project in Visual Studio Code. Excellent. So now let's go ahead and get rid of that application properties file. We're going to do a YAML instead to make it a little more readable with a lot of configuration values. I'm going to delete this file, add a new file. Now let's turn this into a config server first. Let's go back to our main class and enable configserver outstanding. So now let's go to our YAML file and set up all the configurations to point to our GitHub repo. I'm going to set the server port to 8888 again, and now let's add to Git reference. Here is going to be the URI of our GitHub account. I'm going to copy that URL and put that right there. And that's technically all that we need to get started, so let's go ahead and save that, open up a new terminal, and let's run this server. All right, our server is now running. Now let's do a quick query. I'm going to query s1rates, the default profile, and the main branch. So we do see that we get some values back here. I'm getting back a rate, a toll start time, a toll stop time, how many lanes. So interesting, so I'm getting some value, but this isn't actually pulling from station1. So if I change this value to s2, we're going to see the same values. Now why is that? It's because there's no search path, so the only thing it's returning is the default application properties because it doesn't know to check the folders underneath called station1 and station2, and things like that. So let's go fix that. I'm going to stop the app, and let's add a little bit of a search path so it knows where to look. So at the same place as the URI, let's add search paths. And I want to look for anything starting with station. Notice that I have all the files under directories by station name, so I want to make sure we're looking under those and pulling those configs. So let's start up the server again. Great! Let's go back to Postman, and now let's search for s2 again. Okay, so now I'm seeing things are coming from station2. I also see them getting their root application properties. If I search for s1 for station1, I can also see that I get a few values back, which is pretty cool. Now if I search for dev in the profile, I'm going to get even more back. So I get the dev rates, I just get the s1 rates, and I get the generic top-level application properties. In the last part of this demonstration, we want to add an additional repo that we can query based on that query path. So if someone asks for different values, we can actually point it in another repository. So let's go back into GitHub and create a new repo, and this one will be called pluralsight-spring-cloudconfig wa-tolls-perf because we want to pull values from our performance test environment. And once again I'll call this -2 to not conflict with one I already have for the previous version of this course. I created the repository. Let's go ahead and set this up with GitHub Desktop, or however you

want to clone it locally. I'm going to drop this into the same before folder that we used before. We'll clone that in there. Now let's copy the files again into that from the other directory called perf so we have some performance environment configuration values. So here's the perf directory we just cloned from the github-configs directory. Let's grab the perf configs, copy those into the perf directory, come back to GitHub Desktop. We want to now commit these changes and publish that branch. Terrific! So now we can go back in our code and have an alternate repo to look for if someone hits /perf as their pattern. Instead, we can point to a different repository. Let's go make the changes to our application.yaml file. So here at this same level let's add a repos directory. We're going to call this one perf. We're going to give it a pattern. So what should you be looking for to trigger this repository? So anything with /perf coming over will come here. Let's plug in the name of our new repository we just created. And let's also not forget to add the search paths, so those to look in the subdirectories. Outstanding! So let's save that. I'm going to stop the server and restart it. Okay, it's running again. So let's go back into Postman. And here we were querying for s1 rates default. What happens if we hit perf? Ah, so what you see, look at our property sources. This one's coming back from the perf directory we just created, as well as the application property file and the specific one for this app. That's pretty cool. We also now have s3 rates, which we didn't have in our original directory, /perf. And that also pulls from that directory, giving us the files we're looking for. This still doesn't break our first version, so I can still search for s1/default, not perf. And I'm getting that, so one config server is pointing to multiple GitHub repos, pulling the right configurations that match the query path and the search path I have, which is pretty neat. So this config server can be pretty robust pointing to lots of places to serve up config values.

Consuming Configurations from Spring Boot Apps

So we experimented with some configurations. Let's talk about other ways you consume configurations. You probably won't be doing them all from Postman, so what does this look like? Now, you may use Spring applications that take advantage of this config server as a property source. So it's loading its properties from this config server and loading those dynamically into your application. That's going to be a very common way that you do this. Properties from the config server will override any of those defined locally. Now when it loads these values from your client application, it's going to be looking at your Spring application name, the profile you set up, and then a label, optionally. Those things will determine then which value gets pulled back. Now of course, as you define this in the code, you're going to have value properties defined with annotations, which will then load these in from the config server. It's fairly easy to do. What's kind of neat too is, because this is just HTTP and it's just served up that way, I could consume this from a Node.js app, a .NET app, a Python app, a Go app. I'm really just querying it that way. It's going to be easiest with Spring Boot because of some nice integrations. But these are just served up over HTTP, so anybody could take advantage, which is pretty great.

Demo: Consuming Configurations from a Spring Boot App

To prove some of this out, let's have another fun exercise here. We want to consume configuration values from our code, so we're going to create a brand-new Spring project

again that will serve as our client application. We're going to add an application property file and set some properties so that we can pull the right name, profile, things like that. We'll add a controller then that pulls in the values from our config server. We're going to return values that come from those property values and show that on a page, and we can experiment with some of the names and profiles and see how that changes based on the dynamic configuration being loaded here. Let's jump into the code. All right, let's get started building our consumer application that will pull these configurations in our client application. I'm going to be back on the Spring Initializr here, and we're going to create yet another application. I'm going to call this one configclient. Now, if we look at our dependencies, what we want to depend on is the Web, Thymeleaf for the templating engine, the Actuator, and the Config Client. With that done, I'm going to now generate this project and save it back in the before folder, here in this module's code directory. And here we go, here's our config-client. I'm going to go ahead and open this now in Visual Studio Code. Our project is open, so let's go ahead and add some application properties here. Now do notice when I expand the resources directory, I do now have a templates directory because I added a templating engine in Thymeleaf. So we're going to add a UX to this as well, but for now let's go ahead and jump into the application properties. So I want to set the Spring application name to s1rates. That's going to map, of course, to what our config server's looking for for its app name. I'm going to set default as the active profile, and I'm going to set a property here that means I don't need the bootstrap file anymore like I did in previous versions of Spring Cloud Config. Here I'm saying where is the config server? It's at localhost 8888. I am saying it's optional, so if it was offline, the application could still start up, or you could remove that and make it a hard dependency. All right, now let's add a new controller class that will serve as the controller for this web application and also load in the properties that it needs from the config server. I want to indicate that this is a controller being used by our web application. So first off, let's add some attributes. Let's add some properties, if you will, to this class, and we'll refer to the things that will come back from our config server. So first, we have a rate that comes back from our config server. I'm going to annotate that with an @Value annotation, and the value that specifically comes back, if you recall, is named rate. I'll do the same for lanecount, and then I'll add ones for the tollstart. Now let's add the operation that will respond to the web request that we send in. And then we want to return the name of a view that's actually going to display all this. We haven't created that yet, so let's do that now. Here in templates, let's create a brand-new file called rateview.html. You can copy all of the HTML code from the after folder for the downloads. You don't have to type it all in yourself. I'm going to paste that in here. And all this does is it shows the values that come back, your rate, the number of lanes, and the start time. I'm going to save that, I'm going to go start that config server and they config client right after that. Here I'm on my configserver Git project that we created earlier. I'm going to start that back up again. And before I start that up, let me go back to the application properties file for a moment. Yep, and I realize I didn't yet add the label, so let's add the label there; otherwise, it won't know what to pull. And recall that this refers to the name of the branch, in this case, the main branch. And now before we start this up, let's make sure that we add the proper annotation to expose this as a proper operation on our controller. So you should be able to hit the /rate endpoint and pull back this view. Now let's start up this application. Now if we go back through the logs, you'll see that it's pulling the environment for s1rates, profile=[default], label=main, so it was able to successfully connect to that. Now let's go hit the page in the browser, and what comes back is, I see, is your rate is 191, number of lanes is 1, toll start time is 5:00. So let's go ahead and jump back into Postman and prove that these numbers are the ones that should come back. So here,

when I do my query for s1rates/default/profile/main branch, I do see that the default value should be 191, and 1 for the lane count. And because this value doesn't have one for toll start, it inherits it from the parent. So that's great. That worked correctly. Now if we go back to our code and change the profile to qa, save that and run that again, we get some different values back so it's pulling the qa profile. So in this case, we can see we can change some of these values, things get inherited overall, and all of this works great because the client's pulling these properties in dynamically when it starts up and connects to the config server. At no point in my code am I hardcoding any of these values. Instead, I'm pulling them from the environment once I start up.

Applying Access Security to Configurations

So we've successfully seen what it means to build a client application. What does security look like here? What are the ways you might have some security components? So first off, this can be fully integrated with Spring Security, and we'll look at that in a later module here in this course, but Spring Security offers a ton of options for securing access to applications, authorization, a number of awesome things; this works great with that. By default, when you turn this on, you get HTTP Basic support, but you can add things like OAuth2 and more, to make sure you get the right level of access. This is easily configured with your application property files, YAML files; you can turn this on, add the right sort of credentials, things like that. You can even make this unique per profile, giving you some choices for what is the right security for each environment. And there's probably more to this. When you look at a truly secure architecture for your configuration, you're probably also doing some network security things, potentially limiting access via firewalls. You might have an API gateway in front of some of these things. So this is part of a story, as you think of access controls, but you're probably going layer more here, because it's a pretty sensitive service. If I'm running all my configurations here, I want to make sure this stays safe.

Demo: Applying Access Security to Configurations

In this upcoming demonstration, we're going to do a fairly simple example of how do I access this securely? We're going to add a dependency to our project, our server project, the config server itself, to make sure that it's actually putting a layer of authentication in front of the config server. We're going to try that out and prove that access is denied. We can add some Basic auth credentials to our property files; it's not a best practice, but we're just doing a demonstration here. Then we'll call the API using valid credentials. We'll update the client app with credentials as well so it can call it. So we'll kind of see a full, fairly basic, but example of what does it mean to put some sort of credential layer in front of a config server. So we're back here in the configserver-git project that we've been creating so far; this is the one that talks to GitHub and pulls configurations. We want to add some security here. So what I'm going to do from Visual Studio Code is at a new starter, a new Spring Boot starter for Spring Security. With all the right plugins, I can right-click the POM file and choose Add Starters. I can choose Spring Security here, choose to select that, and then proceed, so it actually adds it, and then see that in my POM file and save this. Now if I do nothing else, this actually just adds Basic authentication. Let's prove that. I'm going to start up the application. If I go back through the logs, what you see here, it says, Using generated security password, and it gives

me a value. So it generates a password that I would be using for Basic auth. Now, if I go back to Postman and try to call this as is, we're going to hit that same s1rates/default value that came back before with no problem. Now when I call this back, I'm getting a 401 Unauthorized. I expect that; I don't have any credentials here. Instead, let's go ahead and add authorization, choose Basic auth. The password will be the value from those logs, and I'll paste that in here. And now when I make the request, I get the values back like I did, and that's great. Now maybe I don't love having user at some weird generated password. Let's go ahead and actually update our code to provide a username and password. I'm going to open up the application.yml file, and I'm going to add a section for Spring Security. And I've added a simple entry here: name pluralsight; password, pluralsight. Now this is comically insecure. Please never do this, but this is a good example of just showing a nice way to define the password here in the config server itself. This is the credential that the server now expects for you to be able to call it. Let's start up the project again. This time, you don't see the auto-generated password there anymore. Let's go back to Postman. If I use those previous credentials, it doesn't work, I'm unauthorized. If I use the Pluralsight, Pluralsight credentials, now it works. I've defined that password in the config server as config itself, and now I'm able to consume that. Finally, to do this from the client application, if I try to run this right now, it won't work, because this now has no access. So let's go ahead and add some credentials here. So I've provided a cloud.config username and password. I'm going to start up the application. It looks like it was able to authenticate successfully, but let's try it in the browser. We'll do a refresh here, and it still works. So now I have an authenticated client, a server demanding some credentials, and I've added a layer of security to my config server.

Encrypting and Decrypting Configurations

Now authentication gives me one level of security, that's terrific. But what about the values themselves in the configuration? You can imagine, a lot of these values I don't want in plain text sitting in GitHub, sitting in any of my Git repos. I don't want it just sitting in clear text. So I can make sure that the property value stored in my configuration are not plain text, I can actually encrypt them at rest in the repository. There are symmetric or asymmetric key options. So it could be a shared symmetric key or an RSA key pair for asymmetric options. The asymmetric choice is a bit superior in terms of security, but it's often more convenient to do the symmetric key. Now what's kind of cool here is the config server offers encrypt and decrypt endpoints. So you can use these endpoints to encrypt a value, or then on the opposite side, decrypt that value. And what I also like is you have a choice here, I can decrypt on the server side and then send it back over the wire in clear text even though in the repository itself it is still encrypted, or I can send it over the wire encrypted and it's only decrypted at the client side. Both of these are good options. There's reasons you might want to do both, but I like that you as a developer have a choice. I can either make sure I'm decrypting it at the last responsible moment on the client side, or I might have a reason to do that on the server side. Both are really easy to do, and we're going to do that together here in just a moment.

Demo: Encrypting and Decrypting Configurations

All right, this is a fun demo. We're going to do some encrypting, decrypting of values here. We're going to add an encryption key to our property file. We're going to then generate an encrypted value and add that to the properties files that were checking in. I'm going to retrieve that configuration via the API and see what that looks like. We're going to first test the client application with server-side decrypted value, so it's going to come over the wire in the clear being decrypted by the config server, and then we're going to flip that. We're going to actually update the server to require client-side decryption. And then we'll change the client to do the decryption itself. So this will be fun, let's jump in and mess around a little bit with how do we keep the values in the configuration itself secure. So we're back in our CONFIGSERVER-GIT project, and what we want to do is add an encryption key to our local configuration for the config server. Now I'm going to follow a terrible practice here, you should be generating a proper key, of course, for your value, I'm just going to do something really silly and simple, we're going to do the letters of the English alphabet. Okay, so I've got a simple dumb encryption key that I can use in my application now. Now what's also interesting though is that starting in Spring 2 and up, some of the security capabilities are looking at some of the cross-site scripting things and trying to protect you by default. So I can't hit that encryption endpoint with our secured server unless we override some of that. So I'm going to add a new class to this project that doesn't override, and so we're not looking for that CSRF capability baked in. So I'm going to add a new file. I'm going to call it WebSecurityConfig, I'm going to use the EnableWebSecurity annotation, and now this class I'm going to have it extend the WebSecurityConfig adapter. Then I'm going to define this configure operation, which will then again turn off the CSRF capability. And there we go. So all we need is that. This will automatically get injected and allow us to call that encryption endpoint now. So with that done, let's go ahead and start up the server again. We want to actually then encrypt the value, put that into our configuration, and consume it. I'm going to switch over to Postman with that running. Let's create a new request, we're going to POST it. The body is just going to be plain text. Let's assume we're going to encrypt a connection string. Maybe this is a database connection string, we want to store this in a property file, but we don't want this to just be sitting in a GitHub repo somewhere. So when I click send, what I get back is an encrypted value using the encryption key that we stored in our configuration file associated with our server. Under our tolls project, we want to add this to our application property file. We add the cipher designation to indicate that this is a secured value and paste in the value that got encrypted by the server. We now see the GitHub desktop picked up that change. We're going to go check that in and then push that. And now what's stored in GitHub is the secured value. Now if we query our config server the s1rates, we see we get this back. Now by default, this is doing server-side decryption. So what you see here is in the clear value, it says connection string, like this is the clear value, right? This is not what's sitting in GitHub right now. In GitHub, we can see my value here is still the cipher, but this got server-side decrypted, which is kind of cool. So let's go update our client application. So in our client application, let's go update our controller to now look for the connection string too. We've added it to the model to send back to the web page. Now let's finally update our web page, let's open up rateview.html. And we'll add that the encrypted value is connstring, so we'll see this come back on the page. Let's save all our files. I'm going to go back to the application.properties file, and also switch this back to default. Restarting the page and we see, look, the encrypted value comes back. Here is the full value decrypted again server side. We saw that when we pinged that in Postman, so it sent it back to the client in the clear. Let's

go ahead and do decryption on the client instead. So let's go back to the server, and we want to change the server configuration to say encryption enabled is false. So what this means is it will not do decryption when it reaches the server. Let's prove that. Let's start it up again, and what should come back from the config server now is this still encrypted value. And sure enough, we get the cipher value. Great. So now, even the server has no idea what this value is, or at least it's not sending it back to you in the clear. So let's go update our client now to decrypt it client side. Here we just need to add the encryption key, same value that was on the server that we used to encrypt the value in the first place, we need that client side. And that's all we have to do here. Let's start up the application one more time, and what should come back is now a client-side decrypted value because what we got back from the server was encrypted. And sure enough, what's still here is the in the clear value. So this got decrypted client side. Again, we saw both ways, both are valid, you could use either one, and either way, Spring Cloud Config Server does a nice job of taking care of that for you.

Advanced Settings and Property Refresh

Before wrapping up, I want to point out a few advanced settings that you should be aware of, specifically around property refresh. First, you can configure clients to fail fast if it can't access the config server. This can be useful if I don't want the service to even be able to start if it can't connect to the config server. So during that build process, when it's trying to load everything, if you can't access it, that's great, you can actually prevent that from starting up. Likewise, you can actually configure some retry ability. So if you want to make sure that, hey, maybe I should retry a few times because there could just be a quick blip, you can configure that fairly easily. And then you can also do a refresh, and this is a powerful feature, because let's say I have my application up and running, everything's great, but, hey, I know something changed in the configuration, we added a new flag, we changed a setting, do I have to cycle the app or can the app actually get refreshed live while it's running and pull in the latest values? The answer is yes. So we're going to do that in the last exercise here in the module, we're actually going to change these values live and see those come in without having to restart the application.

Demo: Refreshing Configurations

This last demo shows off how you can refresh things live. First, we're going to add a RefreshScope to the controller, we're going to make sure our applications are started, we're going to change a property directly in GitHub, we're going to then trigger a client refresh, and we're going to see that new value get applied to the application without requiring a restart. Again, pretty powerful capability here we're going to try out, and this really gives you a lot of flexibility if you want to make a lot of changes and not cycle all your app instances. Let's jump in. This exercise starts in our client application. What we want to do is add a RefreshScope and configure this so we can refresh the clients without having to restart them and they'll pull the latest configuration values. So first, for the Controller class, we're going to annotate this as a RefreshScope. This gives us the power to refresh its properties live. What happens here is the Spring Boot Actuator actually adds a refresh endpoint to the application. This endpoints map to /refresh. When I do a post to that, it actually refreshes any beans which are annotated with this RefreshScope. So I can use that to refresh the properties that were initialized with

values first provided by the Config Server, and it'll go pull those again. Now let's go to our properties file. Let's ensure that we've enabled the refresh endpoint. Okay, that's pretty easy. So let's go ahead and start our server and then our client. Okay, our server is up and running. Now our client is starting up, and so it's initially pulling the values it gets from the Config Server. And when I refresh the application, you see I see the 1.91, 1 lane. Let's go ahead and change that rate number and see what happens. Here is where the value is in GitHub. Let's go ahead and just change it live in GitHub. We'll live dangerously here. We're going to raise the rate to 2.91. We'll commit that, and now the value sitting in GitHub is 2.91, but my application is still showing 1.91 as the value. So let's go to Postman, and we want to do a POST to our client application. So 8080 is where our client application is. We're going to hit the actuator endpoint and refresh. That should then tell me what values got changed, and it's showing me the rate value changed. So if I go back to the web page, I now see it's 2.91. So I didn't have to restart my app, I just simply triggered that, refreshed all the property values, and got back the latest. That's pretty cool stuff.

Summary

The Spring Cloud Config is a pretty cool little project. In this module, we talked about the role of configuration microservices. Hopefully here by the end of this one you really see why there's so much value in having an externalized configuration that you can version control, update, secure, change without changing your client application. Because the status quo makes this tricky, right? If I'm just embedding my configs, or I'm trying to keep all this in sync locally with each instance, that's tough as I scale in a more elastic infrastructure. Spring Cloud Config gives me a ton of ways to actually get a lot of different types of property sources and configure that server in a lot of robust ways from a security perspective, encrypted values, and more. And we saw lots of ways I could consume the configurations from code directly via API, and maybe you would use this along with your PHP apps, your JavaScript apps, your .NET apps and use this as a single config store. So, powerful project. Definitely look at introducing this as you're building microservices with Java and Spring Cloud. Thanks for joining me in this module. Jump into the next one as we get further and further into developing modern microservices.

Offloading Asynchronous Activities with Lightweight, Short-lived Functions

Introduction

Hey there, my name's Richard Seroter. Welcome to this next module in this ongoing course about building Java microservices with Spring Cloud. In the last module, we looked at the value of a centralized configuration store. And in this module, we're going to talk about building short-lived microservices. First, we're going to talk about the role of asynchronous processing in a microservices architecture, and, as we're building new microservices, why async matters. We'll discuss a few problems that we can have with the status quo, how we might be doing things today. I want to briefly explain what is serverless computing all about because that plays a part in how we think about developing these types of microservices. I'll introduce you to Spring Cloud Function and what it does. We're going to

have some fun building some functions and seeing how these things work in the different types of interfaces. We'll understand each type of interface as we're building functions and when they come into play for each microservice. And then finally, we'll look at how do you deploy functions, especially even to the cloud? I'll do a brief demonstration of that as well. What does it look like to deploy one of these? How do you do that? And then finally, we'll wrap up.

The Role of Asynchronous Processing in Microservices

So what is asynchronous processing all about? How do we think about this? Well, if we contrast this traditional synchronous processing, service A calls service B, and service A calls and waits and blocks on that waiting for service B to give a response, and then service A continues processing. That's okay, but as you have more distributed systems, you want to reduce some of the dependencies between services. What if service B is down? What if service B is overloaded? Those sorts of things that maybe service A should be able to continue processing or just be able to wait for a response later. So how do I think about processing things not in a sequential synchronous manner, but instead almost a fire-and-forget model where I can have a callback or get a response later? So when I do asynchronous processing, first, I'm reducing some of that dependency. Not just a simple synchronous HTTP call, but instead, I'm doing things that aren't blocking. Now one reason this can be good is it supports these low-latency and high-throughput scenarios meaning I can scale easier with these independent components. I might scale my front-end listener differently than my back-end processor. And so this helps me process things better. I can be more efficient with CPU, as I'm not sitting here just blocking waiting for calls. I can be smarter about processing more and more things. And then finally, this actually helps you move towards a more event-driven computing model. So if your action needs to trigger some action in another service, I'm not just doing that in the request-response cycle. I'm simply publishing my request or some data, and then some other process picks it up and does something with it. So you start to build these more event-driven systems that respond to things happening versus just having these sort of request-response push commands. So we'll see more of this in a future course when I show you Spring Cloud Stream, but that stream-based processing plays a big part in how functions work as well. So, all of these sorts of things, when we think of asynchronous processing, we're trying to reduce dependency, we're trying to have more high-performing apps, and we're starting to move to more event-driven architectures where we don't have a bunch of tight connections between our services.

Problems with the Status Quo

Let's talk about a few of the problems we come across with the asynchronous status quo. First, sometimes you're consuming resources when services aren't in use. So sometimes when I think about, well, every system has to be online, that means I have to have capacity pre-provisioned and ready to go, sometimes that means I have a bunch of underutilized infrastructure because I have services up and running not doing anything. I also may have some of these services baked into some of my more traditional monolithic applications, which means if I want to make a change or I want to make a deployment change, I have to deploy the whole monolith, which means I'm not going to do it as often and it's going to be more

complicated. This sort of model also can be tricky when things spike, so when I do have a spike in consumption, I'm often stuck scaling everything at once versus having very dynamic components that can quickly and elastically scale up to new demand and then scale back down. Now, in some of these services too, you have the routing logic intermixed with business logic. Sometimes the business logic really doesn't depend on the transport channel. This could be coming in via a queue, this could be coming via HTTP requests, it could be all sorts of things. The function code itself, the logic, doesn't really care how it came in. So sometimes when I have too much coupling and now I want to add a new channel and say, hey, this has been a web request, but now I actually want to respond to a message coming in from Apache Kafka, I have to rewrite my app. That's a whole new app versus maybe a newer model that says, how do I decouple the transportation, the routing, from the actual business logic?

What Exactly Is 'Serverless' Computing?

Some of these challenges have been solved by this model of serverless computing, which has been around for the last 5, 6, 7, 8 years when Amazon shipped their Lambda service a number of years back, and this was an idea of how do I have this sort of computing model or I have these managed services that scale to zero. This is a term used by the analyst firm RedMonk, and I like that. This idea of I have services that have no infrastructure that I have to manage, and they are very elastic. Right, the serverless model means I have no servers to manage. Now, we most typically associate this with Function as a Service, this idea of the compute that actually scales to zero. Now there's other services out there like storage, like messaging, databases, that actually also kind of follow that serverless model, but we typically do associate this with Function as a Service, this idea of I'm writing business logic that's able to be executed on demand, but when it's not being called, it's not doing anything. I actually don't even pay for it when it's not responding to requests, very different than a traditional synchronous services architecture where I have a lot of things up and running, everything pre-provisioned, ready to go, not a lot of elasticity there. And so really the hallmark of this sort of service and this sort of architecture is the idea of very scalable elastic services that are able to quickly meet demand, go back down to zero, respond to that. Now, the other hallmark of these, of course, if that's going to work, if you're going to have services that quickly scale from 1 instance to 100, then that has to have a very quick startup routine. It can't have a three minute warmup of cache and do a bunch of other stuff, it has to be a very fast-starting service, very quickly to shut down as well, and these typically run for short periods. You are seeing service providers in the public cloud offering these sort of services now with up to a minute, up to a few minutes of response time. It can actually hold the thread for that long, even for 60 minutes, but for the most part we think of serverless computing as fairly short-lived, stateless functions, where once that function goes away, anything that was in that instance disappears as well.

What Is Spring Cloud Function, and Integration With Spring Projects?

Now in that context, what we're going to talk about today is Spring Cloud Function. This is a fairly cool little project that maybe you don't hear a ton about. This actually helps you build logic into functions that separates the logic from the implementation, meaning that I can use

this with web-based requests, messaging, even batch scenarios. So this is a really important project that underpins a number of things in Spring Cloud. It makes it easy to build these modern sort of services that use functional interfaces. However, let's talk more about how does this fit into the Spring ecosystem, because this has a lot of interconnections across the Spring framework. So, first off, to be clear, Spring Cloud Function apps are powered by Spring Boot. These are just Boot apps. They just happen to be fairly unique Boot apps that are able to be spun up based on functional requests and be connected to a variety of back ends. This does use Project Reactor for reactive APIs. Again, you have a few ways you can use these sort of functions. We're going to try a bunch of different styles here in this module. You'll see how this works for reactive and even non-reactive functions. But it is used across other projects in the Spring ecosystem. It works with Spring Batch, works with Spring Integration, works with Spring Cloud Stream, works with Spring Cloud Data Flow. So a number of things this works in, and the sort of functional model was more and more prevalent in Spring Boot and in Spring Cloud, so this is a good thing to learn. Even if you're not writing a ton of functions, you're probably going to see this behavior in some of the other Spring projects, so it's a fun thing to learn.

Creating a Function

We'll talk now about how a function works, how do you create one, and how does its logic works. So first off, to create a function, we add a handful of dependent packages, really just things like Spring Cloud Function web, as we're building web applications, we'll do that together. You could also end up adding additional binders and middleware components, but we'll start with the web-based experience. Then you choose a functional interface. There's three main ones: supplier, consumer or function. We'll talk about when you use each one, but these are different interfaces based on whether I'm receiving data, sending back data or receiving data and sending back data. Then, of course, hopefully the reason you're using something like Spring Cloud Function in the first place is you're trying to focus on business logic. This is the real part of it. You should be spending most of your time building the actual logic of the function, not doing a bunch of the boilerplate stuff that Spring Boot and Spring Cloud and Spring Cloud Function take care of. And then finally, you annotate your appropriate business logic and you mark your function with a bean annotation so it's added to the function catalog and can be invoked based on sort of web requests or messaging requests and things like that. So it's a fairly lightweight experience. There's not a lot of machinery you have to set up. It's a fairly smart model, where it's detecting your functions and able to invoke those, and you don't have to do a whole ton as a developer to build these more serverless-based type applications. Let's talk a little more, though, about that business logic. That's the heart of any of your functions. So, first, again, these are Spring Boot apps, access to auto-configuration, more, if I want to use Spring Data, if I want to do other things that are part of Spring Boot, use Spring Cloud Config, all those sorts of things. This is just a Boot app. Now, the function can be treated as stateless. It can depend on where you're running. You might run somewhere where you have a scale to one environment, or you might be running this on your own infrastructure and you always keep the function running. But traditionally we think about a serverless computer, we think about these sort of function workloads as stateless, meaning that that instance, that process, that request, and maybe it uses some local data cache and it writes something to disk temporarily, but I should assume that instance goes away. So I'm not storing anything there solely. I'm not running my

database there. I'm not uploading files there, and then keeping them there, I'm putting them into an object storage. So it's sometimes safe to assume the data is going to disappear, although you can, again, persist it based on where you actually deploy this to. And functions are a little unique. This isn't just a traditional application that always takes in stuff or always sends stuff back. Instead, you may accept parameters and return values depending on your interface. You might say I don't return anything from this. All I do is accept data and process it, and the caller gets to move on once they publish it. Maybe you're invoking this on a schedule and so you don't expect a response to come back, all sorts of things. So it's going to depend on the interface and the use case, as we'll see in a moment.

Demo: Creating Functions

Alright, let's have some fun playing with functions. We're going to just do something simple to get started here. Remember, in this course and in this module we're building something around toll stations and toll roads and paved roads. And so let's assume for this simple function we just need something that responds to requests saying, hey, give me information about toll stations. So we'll start by building a brand-new Spring Boot project. We're going to then define the functional interface that makes the most sense here and annotate that as a bean. We'll add some business logic that simply retrieves data for a given toll station. And then we're going to make that a web request, and you'll see how simple it is to separate the business logic from the sort of transport channel in one basic interface, and then in subsequent examples we'll use some more of the other functional interfaces, get a little more sophisticated. But let's start simple now and understand how this works. We are back here in the Spring Initializer, as we want to build an application to process toll information, in this case, return some toll station data, and we want to do this as a function. So I'm going to call this application web-function, and we simply need one dependency added. We're choosing the function dependency. Let's generate this application, save it in the before folder associated with this particular module of the course. (Clicking) So now we have that available for opening in Visual Studio Code. Let's open that in Code. (Clicking) Let's open up the pom file here, and you see by default it added spring-cloud-function-context. That's not bad, but what we want to do is actually treat this function like a web service, and so in this case let's change that starter to spring-cloud-starter-function-web. In this case, this will make it automatically expose the function endpoints as web endpoints, which is pretty cool. Awesome, so now let's go and create a new class that represents toll station information. And let's assume this TollStation just has three simple parameters: a station ID, which mile marker it sits at on the highway, and how many stalls it has, or how many places does it have for someone to drive through. And for each one of these we'll do some getters and setters, and we'll set up one constructor that takes in all of these parameters. And, again, Visual Studio Code with the right plugins makes this fairly easy. Okay, so now I have a constructor that takes them all in and I have getters and setters for all the properties, let's go ahead and save that class. Now back in the main class, let's open that up, and what we want here, we're not going to call a database, a proper course here might be showing you Spring Data and pulling database information, but let's keep things easy for this and just have some in-memory information. So, specifically, let's create a list of tollStations and we'll load that up in the constructor. Let's create a new constructor. We'll instantiate that variable, and now let's add a few tollStations to it. Alright, I've added three tollStations, gave it a few identifiers. Remember, that second number is the mile marker, where on the highway do you

find that station. And then finally, how many stalls does it have, or how many places does it have for folks to drive through. Our final activity is to add the actual function. Let's have a function that retrieves the station that is requested. Here, this is going to be a type Function, you see using java.util.function, basic Java stuff. It's going to take in a string which represents the station ID, and it's going to return a TollStation object. Alright, so now we just need to return the value. First let's log the request, and now let's actually return the TollStation that matches the request ID. Alright, so I have a request that turns my list into a stream, does a filter, so I'm looking for any of the toll records in that list that match the ID coming in. I'm going to find any of those, and then if I don't find anything I'll just return null. Now, what do I have to do to make Spring Cloud Function recognize this thing as something that can be invoked? This is really simple. All I have to do is annotate this with a bean and then it adds it to its function catalog and invokes it upon request. I'm going to start up a New Terminal here in this application folder. Let's go ahead and start our function, and because we used spring-cloud-function-web, you see it actually starts up a web server on port 8080. Let's go switch over to Postman and make an invocation. Let's go ahead and POST to the endpoint. Now you'll notice the name in the URL corresponds to the name of the function. I can override that, but for now I'd like that to be the same. And when I send in that request, look, I get the response back. I get back the toll station that corresponds with that request ID, so I invoke that function. It responded to it, and because it is a web server, it's actually still sitting there running right now, but that's kind of cool. So let me go also do this with a GET request. GET requests have no body, so I would put this in the URL, and that works as well. So I can do this as a POST or a GET and that functional interface handles both of those. If I only have one function in this application and I just want that invoked by default, I can actually set a default value. So let's stop the application, let's jump into the application.properties file, and set the default value. So when I do that, now I could literally just hit localhost:8080 and it would invoke this function, because I'm telling it what the default one is. Let's start up the application again. Let's go back to Postman, and I can still invoke this via the URL with a discrete name for the function. Let's pick a different value here. And that still works, but look, what happens if I actually get rid of the rest of the URL and do a POST? And that works as well. So I could just post to the root URL, everything works because I set a default function that should be invoked when that request comes in. So kind of a cool way, again, to define that functional logic very separate from the web request, but still have it be fairly smart about knowing which function to call.

Choosing a Functional Interface

All right. That was fairly straightforward. Functions are cool. That's a neat demo that shows just how the functional interfaces work, but let's learn a little more here. What are the functional interfaces you can choose between? Supplier is terrific. It just provides output, no input. So it's good for an endpoint that provides data, but doesn't actually expect any input. Again, supplier can be reactive. I could return a flux or a mono or things like that or imperative. That's also fine. So this is a good choice if I just want to return some data. Consumer is pretty good, especially for asynchronous endpoints that take an input the opposite end, but they don't really expect any output. So I'm just going to publish data to this endpoint. Here's some data. I don't care what you do with it. Let's move on. That's great. And then function is good if I have an input and an output. So if I do have a request response sort of interface, that's still okay. As much as we talk about asynchronous

processing, and supplier and consumer are terrific for that, there are cases like we just had where there's a request response interaction. I want to send you some information, I want you to give me something back. So all three of these are valid. These are the three choices you have. Let's see this a little more in-depth. So if I look at the supplier interface and I do an imperative model, I'm not doing anything reactive, look at this example. This would respond to an HTTP GET request. The top option simply returns a single string value. The second option returns a list, so I can also send a list of values back. Either of these are valid functions. They both use the supplier interface, sending back one or many responses. Now if I look at that supplier interface in a reactive style, I can return a flux, and I would return a flux if that downstream provider wanted to subscribe to that data, it's an ongoing stream. In this case, I'm still sending three responses back. I'm sending them all back as a flux. If I were a streaming listener, not just a web request listener, I could be processing this data as it's coming. In a web request, typically this means all of these still come back at one time in the HTTP response. Now let's talk about consumers. If I have that consumer interface, look at the first option. I'm receiving a string, writing it out. In the second option, I'm receiving a list of strings and I'm looping through and writing out the values I get in there. In this case, this responds to a POST request. I send in an HTTP POST with one or multiple sets of data, and I can process it, but it doesn't send anything back. In the reactive style, I might be receiving a flux in. A whole data stream might be coming in or it might be responding to an HTTP POST request with a bunch of data. In this case, again, I'm processing a set of data, not sending anything back. Now, we saw the functional interface in our demonstration here. I could receive a string, send back a string. I could receive in a list of values, return a single value, as you see in the second option. This would reply as we saw to both an HTTP POST or an HTTP GET request. And this can work reactively as well. I might take in a data stream and return a mono. I might take in a single value and return a flux. So I can deal with reactive types as well on both ends of this, and this responds again to a POST or a GET request being able to send in and return streams of data.

Demo: Using Each Functional Interface

Since we now know a little bit more about these functional interfaces, let's go ahead and do a more deep dive demo. Let's actually add some functions to the existing application we built earlier, and I want to process more than just receiving requests for toll stations; I actually want to process some of the toll payment information. We'll experiment with some of the different functional interfaces so I can show you how to observe how data is processed based on whether you're a supplier or a consumer or a function. So let's jump into that and get some more experience and get some more comfort here. We are back here in Visual Studio Code in the web function that we built earlier. And so let's start by building a new class to represent a toll record, an actual record of a car driving by, us recording that, and we want to process that toll record. Maybe we want to charge them or debit their account. So let's first create a new class called TollRecord. And let's again just add three simple properties here. We'd like to have a station ID, the license plate, how do I identify the car, and the timestamp, when did it happen. And let's again add some getters and setters. And let's also add a constructor that takes in all three of these values. Terrific. So now I have a data class that represents the record that processes for each toll. Let's save that and go back to our main class. And the first thing we're going to do is let's add a new bean. This one is going to be a consumer function. This one just takes in the record, just takes in a new toll record,

doesn't return anything back. Let's say this is happening at an actual toll station. It doesn't want to process the response. It doesn't care. It's just telling the corporate system that a new car drove past. So let's use the Consumer interface. It consumes a TollRecord. And in this case, all I want to do is print out the value I received to prove I received it. All right. So we have a consumer that simply returns a value that prints out what we received from the license plate. Let's make sure we mark this as a bean so it's seen as a function. Excellent. Now let's start this up. Once again, this function app starts as a web application, so let's switch to Postman and invoke this. It's a consumer interface, so we want to use a POST. Remember that the function name is also part of the URL. And so I've got a valid JSON request with the station ID, the license plate, and the timestamp. And mess around with the syntax. Sometimes you'll have different casing depending on your variable names, but this should work. Let's click Send. You'll notice I get back an Accepted here. Because of the consumer interface, I'm not really expecting a response. Let's look at the logs. And you can see in the logs I did receive the toll for that license plate we just sent in. It handled that request and printed that out. That's pretty cool. Now let's build the same type of function, but let's do that in a reactive style. So if I write this as a function, remember, I still don't really want to return anything back, so I'm just going to send a mono back that's empty. Now in this case, I also have to return a mono because that's what's expected. So let's return an empty one. So I would really do this if I was doing everything reactive. In essence, the consumer interface I showed you earlier is simpler, but I just wanted to demonstrate this. Let's annotate this with a bean, save it, and start it up again. Let's go back to Postman. And we just want to provide some different values. Once again, it's accepted because I didn't actually return a response. And in my logs, I see it still printed out the license plate. So this is a more reactive style. The final consumer interface I want to show you is a flux. So let's say I receive a batch of records. I get a bunch of toll records at once. Maybe this toll station saves up the last few and then sends them to me. So in this case, I'm taking in a Flux of TollRecords. And in this case, because it's a flux, I have to subscribe on it till it actually starts processing data. So in this case, I send in a whole set of toll records. I'm going to then subscribe on those and print out each one as I subscribe and I loop through the data. We'll once again mark that as a bean so it's seen as a function that can be invoked. Let's start up our application. I'm going to switch back over the Postman. So I've pasted in a set of records here, a couple of station IDs, license plates, and timestamps. Let's go ahead and send that in. I get a 202 Accepted. Let's look at the logs. And sure enough, I see both records come back in. That's pretty cool. So I was able to send in a flux of records and process those. And then finally, let's add one more. Let's use the supplier interface. So let's return all the toll stations we know. Remember the supplier doesn't take anything in, it returns data. So let's go ahead and just send back all the toll stations. In this case, we're using the supplier interface. We'll return a flux of toll stations. In this case, what we want to return is a flux, and we're just going to turn that list of toll stations into a flux. Let's annotate that with a bean, save that, start up our application, and now when I do a GET of getTollStations, I should get them all back. Sure enough, I do. Here's our three toll stations that we added to our list. So we saw the function interfaces, the various ways to use consumer interfaces, both imperatively and reactively, as well as supplier interface.

Deploying Your Functions

Let's talk now about how do you deploy functions. Now, we saw a little bit of that as we tested things out. So you can embed that function in a standalone web application, as we've been doing. All right, we started this thing up, it ran in its own JAR file, everything was great. So you can just have these functions automatically exported as HTTP endpoints and then be able to run this as a web server. That's kind of a cool way to do it. You might also embed this in a standalone streaming application. You can have an application that responds to requests in RabbitMQ or Apache Kafka, handles that data, and does something with it. We're going to do that in a subsequent course. So that's another way you can do it. You could have a web application or you could have something a little more dynamic handling things from a message broker. There also is a model where you can import these different functions as package functions in a JAR file. So I could have a JAR file that actually then loads these things in dynamically. That can be pretty powerful. If I want to be flexible about the functions I'm calling, I can actually point to where my JAR file is containing my functions and load that in at runtime. So there's some kind of cool ways you can do things with functions. Probably you'll do this a lot as a standalone application, however. Now one of the options for deploying this is to serverless platforms. That's probably even where your mind goes when you think about serverless is you think of the public cloud. And so every major public cloud has these sort of function-as-a-service type services that make it easy to run this sort of standalone compute. What's kind of cool is that you have different adapters from the cloud providers, whether that's Amazon, Microsoft, Google, Alibaba, others, to make it easy to translate Spring Cloud Function into something that runs in each one of those function-as-a-service platforms. What's that adapter do? Well, it helps with some of the entry points. What is that first thing that platform should call that executes your function. It also does some things that might be unique to each platform API, but can hide you from that. So you just use that adapter and you're good to go; you don't have to know the nuances of each platform. Now, what's important is if I am using a public cloud function as a service, I don't want big bloated services, I want small, tight services that can start up quickly, shut down quickly. I don't want to do a lot of state management because right now it's kind of difficult in most public cloud function-as-a-service providers to do a lot of stateful processing. Each one of these goes away, each instance goes away at some period of time, so they're not for durable, stateful things. So when I am using a public cloud FaaS, I am thinking more and more about stateless, small, fast services.

Demo: Deploying a Function to Google Cloud Functions

In this last demonstration, I actually want to show you what it looks like to deploy a function to a public cloud Function as a Service. So this might be one you watch versus follow along. I'd love it if you follow along, but I'm going to make some assumptions here that you have a Google Cloud account, they're free, you can spin one up. And I'm just going to use that to deploy a function to Google Cloud Functions. Similar process to deploy the Lambda, Azure Functions, and others. But I'm going to show you what it means to deploy a function to the cloud and then how to test that function. So you'll see how the Google Cloud adapter works. It's going to be similar for other function providers as well, but I want you get a sense of how this behaves. Let's jump in. So in the before folder for this exercise, you'll see a project called cloud-function. You can open that up. Let's look at the POM file first. You'll see that

we're using spring-cloud-starter-function-web just like before. You'll also see that I'm using spring-cloud-function-adapter-gcp. It's going to be similar versions for other clouds as well. And I've also added some dependencies further on for the build process just to make sure it knows how to build things properly. The function itself, very, very simple. It's using the function interface, taking in a string, your name, and returning a string, giving you a greeting. Now it's interesting, and I've logged it here, the entry point for this function from the FaaS platform is a specific class in the JAR file, in the adapter. And so in this case, that adapter then invokes this function. So it's important to recognize the sort of entry point for this function is different. It has to figure out, okay, now which function should I call? So it always would be this regardless of what the name of my function here is. The only other thing I did, which is maybe a little quirky, I added a META-INF folder here, added a MANIFEST.MF file here, and I'm simply telling the function application where is my main class. So when that launcher kicks off and my entry point kicks off, it knows where to look to say where in the world are these functions at. So this is simply the connective tissue I've added here. I'm simply pointing to the class holding our functional bean. So here in the Google Cloud Platform console, I have no functions available right now, I'm going to deploy this all from the command line. I could also do this in the user interface, but right now there's no functions. I want to go deploy one and I want to invoke that function on demand, and when it's not doing anything, I don't want to pay for it. Deploying a function is just one command. I'm using gcloud function deploy, giving it a name of pluralsight-gcp-http. I'm telling it that entry point. So here's that launcher. I want this to run on Java 11. I'm going to trigger this via HTTP. I'm telling it where the source is in the target/deploy folder, and I want the memory to be about 512 MB of RAM. And before I run that, let's go ahead and just package this just to be safe because I do want to make sure as it's looking in that folder for target/deploy, I want to make sure that I've built that. Sometimes it can build it for you, I just want to be safe here, into a package. Now I have my function in the deploy folder. I just like to play it safe. Now let's run that gcloud function, which is actually going to deploy my workload. So again, it's going to look at our JAR file, it's going to deploy that to Spring Cloud Function, it's going to use that entry point. I'm going to allow unauthenticated invocations. Again, this is a really powerful way to run your Spring Boot applications. I could run this in these platforms, and I'm only paying when it's being executed, and if I have to scale up to 100 instances quickly and then back to 2, I'm only paying temporarily for that time, and it happens very, very quickly. So these are kind of a cool way to rethink how do I host my Java applications into a set of microservices, all potentially executed as a set of functions. If I refresh in the Google Cloud Function console here, I can actually see that it is starting to deploy my function in the US Central region. All those values I provided, including the entry point, including the runtime, the memory allocation all starting. It's already deployed. I can grab the trigger. So this is the URL. And sure enough, when I get back, Greetings, Pluralsight and cloud user Richard. So a really, really simple way to run this function, see the result, see what happens, and now once this thing is done in a few moments, that instance disappears and I'm not paying for anything anymore. So functions are a cool platform in the public cloud. Using Spring Cloud Function gives me a great way to consume that.

Summary

We did a lot in this module so far. Hope you enjoyed this. We talked about asynchronous processing, the role of this, what happens in microservices as I think about decoupling my

source service, my destination service. I have an easier scale, easier performance. The status quo with a lot of coupled services can make it difficult to scale, can make it difficult to do continuous updates, and so I like this idea of a more event-driven system. Serverless computing gives me these managed services that scale to 0, that scale quickly, that have very small sort of code bases that are very tight and high performing. We saw how Spring Cloud Function lets me build modern systems where I decouple the business logic from the transport mechanism. We created a number of functions using the function interface, the consumer interface, the supplier interface, both with reactive style taking in things like fluxes, as well as more imperative and just processing single values or lists. Then we talked about deploying functions. Maybe you put this in a web application, in a JAR file, maybe a streaming application. And then we even deploy it to a public cloud Function as a Service. So lots of things you can do with function platforms and Spring Cloud Function as you're building microservices with Spring Cloud. Thanks for joining me in this module. Jump into the next one as we keep learning about building great microservices.

Securing Your Microservices with a Declarative Model

Introduction

Hey everybody. My name is Richard Seroter. Welcome to this next module in a course about building Java microservices with Spring Cloud. In the last module, we talked about short-lived microservices and functions, and in this one, we're going to talk about securing your services using OAuth 2.0. This will be an action-packed module. We're going to start by talking about the role of security in a microservices architecture. We're going to talk about the problem with the status quo, and maybe how we've been securing services in more monolithic systems and trying to translate that to microservices. I'll explain what OAuth 2.0 is all about. I'll explain how Spring supports this as an authorization mechanism. We'll talk about different grant types in this module. So we'll talk about the authorization code grant. I'll show you some ways to stand up authorization servers. This will be fun. I'll talk about another type of grant, the resource owner password credential type. I'll discuss the commonly used client credential grant type. We'll talk about a handful of more advanced configuration options, and then we'll wrap up.

The Role of Security in Microservices

The topic of security is such a broad one. There is so many different areas we could cover here. So as we do think about the role of security as we're tackling microservices, what are some things to think about? Well, first and a really big one is user authentication and authorization. How do we figure out who is who? If we do a sort of 0 trust model where no component inherently trusts another, then I have to think about ways to constantly ensure that the user making the request is the right one and that they're allowed to do what they're doing. So how do I manage these credentials, check the access of the requester? It's a really important thing to solve. Now, single sign-on and token management are actually really important to complement this because I'm potentially chaining together a lot of services. So there is some idea of single sign-on so I'm not authenticating the user every

single time on every single request and slowing down. Instead, I can be passing a security token that I could validate at different stages so these things kind of go hand in hand. Another big piece of security is the data security. There is security in transit. How is it going between services over the wire and how is it at rest? What happens when I'm storing data? How am I keeping that secure? And this isn't just for data passed in and out of services, but also configuration data. How am I keeping configuration secure? We talked about that in an earlier module as well. That is another big part of this story. And then finally, increasingly interoperability matters here. A hallmark of a good microservices architecture is that teams are shipping at different times, maybe even using different programming languages, platforms. So any authorization authentication sort of scheme should work across platforms and I should be able to use this regardless of what type of service it is. I can't just assume everything looks the same. So a modern story needs to accommodate lots of different platforms.

Problems with the Status Quo

Let's talk a little bit about the status quo because the classic model of securing maybe a more monolithic system, hey, the traffic comes in through a load balancer, traffic gets sent to app servers, everything's high-trust within those known boundaries. That's not bad, but that's a different model from a more microservices, distributed systems architecture where I have things all over the place, different traffic routes, different servers, different environments. So first, one of the problems with the status quo sometimes is credentials embedded in applications. These are closely coupled to the app. Sometimes the credentials or even the authorization store are baked into the app itself, and that's not always reflected in the API tier. So, sometimes the credentials and the application are way too closely coupled. You also may end up with unnecessary permissions. In a classic monolithic model, I may authenticate users who now can do many different activities, and I've got this broad permission surface versus really going to a least privilege model that says what is the least amount of privilege you need to execute this operation. What's also important now is how do you differentiate users and machines? I don't want just a typical service account model because then it can be hard to audit that, so how do I think about having users calling things and representing themselves or having machines call things and represent themselves still with maybe short-lived credentials? Now the status quo for many organizations that I've seen is not always being optimized for a lot of different diverse clients, even nowadays, whether it's mobile apps, different browsers, different programming languages, different clouds. How do you think about all the different places your application could run, your services could run, and how do you make sure you actually work across those? The status quo isn't really ready for a low-trust stateless server model with a lot of changing dynamic permissions. I think we need a better way, so let's talk about that.

What Is Spring Security?

So, one solution to this, especially for the Java and the Spring developer, is Spring Security. And I put Spring Cloud Security there in parentheses because there is a Spring Cloud Security project. Now originally, this was a robust project that did a lot of OAuth and credential things for microservices, but a lot of that has now rolled into Spring Security

proper, which is what we're going to be using here with our microservices. Spring Security has this robust support for all sorts of security features. We're going to focus primarily on authorization. And these projects are going to help you whether you have API gateways, whether you have services on different machines, all those sorts of things. It's really a mechanism for doing token-based security between components, so we're going to flex a lot of this, and OAuth 2 is an important part of that.

What Is OAuth 2.0 All About?

What is OAuth 2 really about? Basically, it's a protocol for conveying authorization. It's built to be the standard here, so it's a protocol for conveying authorization decisions using tokens. It offers a standard way of obtaining and validating tokens. The old way was about securing requests at the edge. And then once you got into the system, you were kind of in. But the better way is a user authenticates on an authorization service, which maps that user session to a token. Any further API calls to any sort of resources provide that token. Then those services can recognize that value, look it up, and see if that person can do what they want to do. Now what's interesting about OAuth, it provides a few authorization flows for different types of clients. How do I authenticate users? How do I support service account-type scenarios? How do I support scenarios with direct credentials? So, it's got some different flows for how do you obtain that token? What's important is that you have limited access to user accounts. This protocol let's third-party apps grant limited access to these services, either on behalf of a resource owner or allowing some third-party app to also get access on its behalf. But access is requested by a client. It could be a website, mobile app, or whatever and then getting some credentials that it can use and some permissions it can use downstream. What's important here is it separates the idea of a user and a client. They're separate entities. Instead, I'm saying I am authorizing this app, this client, to perform these actions on my behalf. And we'll see that in action here in just a few moments. What's interesting here is the access token carries more than just identity. Actually, a lot of the point of OAuth is you should not be collecting user credentials in your application itself. So, you're not just passing sort of basic identity. You've got metadata about roles you have access to, other things like that. There's actually no fixed token schema interestingly in OAuth, so there could be a lot of different data that is passed along in that token. But what's really important here, this is not an authentication scheme. The user still has to be authenticated to get a token. How they get authenticated is outside of OAuth 2. How a token is validated is also outside the spec. So, really, this is about the process and the protocol for conveying and validating authorization, but there's some room to move between different providers. All right, let's go into a little more detail here. Who are the actors in an OAuth 2 scenario? What do you have to think about mentally? First, there's the resource owner. This is some entity capable of granting access to some protected resource. When that resource owner is a person, it's referred to as an end user. Now this is generally you. You're often the resource owner. Now you also have a resource server. This is the server hosting some sort of protected resource or data. It's capable of accepting and responding to protected resource requests using access tokens. So, a resource server could be hosting a REST endpoint that returns customer data. You as the resource owner are then getting access to that. Now the client is some application that wants to access that user's account. They're representing you. It's making a protected resource request on behalf of you, the resource owner, and with your authorization. And then finally, we have an authorization server. This is a server issuing access tokens to the

client after successfully authenticating that resource owner and getting their authorization. This could be the same as an authentication server. It could be part of the same component. They don't have to be physically separated, and it often is. But this is really the server responsible for issuing those access tokens and validating them. What are some other terms you should know? Well, I've used the term access token already a couple times here. Access tokens are really credentials used to access protected resources. It's a string often representing an authorization issued to that client. These tokens represents some sort of specific scope you have access to or some duration of access, how long is this good for, granted by that resource owner. This is often passed around as headers in these requests to the resource servers. It's got a limited lifetime. Even if someone happened to hijack that, they wouldn't be able to use it for very long because it expires. That's where refresh tokens come in. This token is issued with the access token. But unlike the latter, it's not sending each request from the client to the resource server. It merely serves to be sent back to the authorization server to renew the access token when it's expired. Now you have the idea of a scope. This is a permission, something you want to perform access control with. And how do you get this into the token? Well, when there's a match between what the app offers and what the user has permission to, you're set, and the scope might be read access to a particular dataset. You might have a scope that allows you to create a sort of record in the system. Then, you get to the client ID and secret. Who's the client? Once your application is registered, that service will issue some sort of client credentials in the form of a client identifier and client secret. We're going to use this a fair bit. The authorization server then issues this registered client, that identifier, and this is some unique string that represents the registration information from the client. That client identifier is not a secret. That's not something that's a private piece of data. It's actually exposed to the resource owner and must not be used actually has credentials. The client secret is required, and that is secret. That is something that you want to keep safe. You also may come across the term OpenID Connect. This is an authentication protocol based on OAuth 2. So, this lets developers authenticate their users across websites and apps without having to own and manage password files. So, the key with OAuth a lot is getting out of the business of dealing with passwords directly in your application and being able to offload that. And finally, you'll see JSON Web Tokens, or JWTs. This is actually used for a lot of different things, but it's commonly used in authentication flows providing encrypted content. Really, this is a token format. We'll use this a fair bit. You'll see us use JWTs. It's somewhat transparent because the Spring framework just kind of bakes that in.

How Spring Supports OAuth 2.0

Now Spring has fairly good support for OAuth 2.0, so all those concepts I just explained. Sometimes, honestly, this can feel overwhelming. It took me a while, frankly, to really understand how all of this worked. It seems like there's a lot of moving parts. I'm a developer, I just want to secure my app. Do I have to understand so many of these things? The answer is mostly no, because Spring has done a really good job of baking some of these things in so you're not stuck doing a lot of this yourself. So you get some very, very good auto-configuration for clients and resource servers, a lot of things baked in for you here. So, really, you're just doing some things and behind the scenes there's some good stuff happening, but you don't have to do a ton yourself to use this. Instead, as a developer, you can just do good practices without a ton of work. What's also nice is Spring supports a number

of those OAuth flows, so you can do client credentials, you can do the resource owner password one, you can do all kinds of things so you can really match nicely to what OAuth supports generically, which means that if you do have non-Spring clients using the same authorization server, everything should work really well. This works really well with both the RestTemplate and the WebClient, so when you are issuing requests to downstream services, a lot of the time you're getting these headers added transparently. So being able to get that token injected into your WebClient requests to a protected service, that's just going to happen for you with some very simple stuff, which is really convenient. And as you can imagine, there's a lot of extensibility points here, you can change token shapes, you can do all sorts of things, but Spring offers a really, really rich set of things you can do from a security perspective, specifically with OAuth, which is really great.

The Abstract OAuth Flow

How about we start visualizing this. What does this really look like? So the parties I mentioned earlier, you have a client, you have a resource owner, probably you. You have an authorization server that issues tokens. You have the resource server that actually has the protected resource you're trying to access that rest endpoint, whatever it will be. So generically, what does this flow look like? Well, the client makes an authorization request to the resource owner saying, hey, I want to be able to do something. The resource owner grants you access to that and says yes client application, you can represent me to retrieve this resource. That client then sends that authorization grant to the authorization server who then returns an access token. This access token then actually gives me scopes and gives me permissions to, if you will, authorization to the resource. The client then sends that access token along with their request to the resource server. That resource server then validates that token and sends back the protected resource. So this is a very simple flow. In real life, this happens in seconds. But what you can see is you're really just giving permission, using that permission to now get a token, using that token to make requests.

OAuth 2.0 Grant Type: Authorization Code

I showed you a bit of a generic flow, but let's look at a very specific one. So the authorization code grant, very popular, you're probably going to do this one a lot. You use this one to get access tokens and refresh tokens for confidential clients. This is a redirection base flow. We're actually going to do this in a moment in an exercise, but what does this look like? So you still have the user, let's say a web application, the client, an authorization server, and then finally, a secure API, some sort of thing I'm accessing in the client application. So first, the user would log into the app, then that web application is going to request an authorization code. That authorization server then redirects the user to log into the system and provide an authorization prompt saying, hey do you give permission to this web application? So again, in this case, the web app never sees your credentials. The user then authenticates themselves, provides consent saying, yep, this web app can act as me. Now the web app gets the authorization code back. So again that whole middle flow didn't involve the web application at all. Now that web application sends that authorization code, the client ID they have, the client secret they have, the authorization server validates all that, make sure it has everything it's supposed to, it then returns a token to that web application, which it can now use to represent

the user. Then the web application is just simply going to request whatever secure data it's trying to access putting the access token in the header. That secure API is going to validate that token, make sure it's still good, it hasn't expired, it's not requesting scopes it doesn't have, and then finally, going to return some secure data. Again, all of this happens in a matter of seconds, but this is an important flow to understand and really grok what's happening between these components.

Demo: Authorization Code Grant Type

Okay, how about I stop talking about stuff and start showing you some stuff. So let's actually build something. Let's use the authorization code flow and GitHub is both an authentication and authorization store. So we're going to build a toll reporting site. Let's say we have a website that you can hit the home page, but as soon as you want to view data of reports, then I'm going to authenticate and authorize you. We're going to add Spring Security with the OAuth2 login. We're then going to authenticate and authorize ourselves via GitHub. We're going to watch some of the redirects in the browser so we can see things moving around automatically for us. And then finally, we'll choose which pages to protect and configure that appropriately. This will be fun. Let's go jump in. Okay, we're here on the Spring Initializr site and I'm going to call this application secureui. Let's add a few dependencies. We're going to add a dependency on web, WebFlux, OAuth2 client, and Thymeleaf so we have some UI components. And we want to make sure we pick the OAuth2 client, not the resource server at this point. Awesome. Let's generate the application and save it in the before folder associated with this module. Terrific. Now let's open this up in Visual Studio Code. Excellent. So we have this open and I'm going to first go into the POM file and comment out the OAuth client only because it immediately forces security on the pages and we want to turn that off just to kind of first get our website up and running. Alright, so we've disabled that for now. Let's go ahead and add a new controller called ReportController and this won't do much to start with, it just really needs to return the pages for a page called home and a page called report. So if you hit the root, we'll load the home page, and if you hit the report path, we'll return a page called report. At the moment. I'm taking in a Model m because we will start adding some attributes to that once we call a downstream service. Now let's add the UI templates for the home and report pages. We put these in the templates folder. The HTML for the home page you can find in the after folder associated with the secure UI so we don't have to all type this out together. It's pretty basic, it just simply loads a page, it says this is the home page. Awesome. Now let's grab the code for the report page from the after folder for this module and paste that in here. In this Html, we have a table that shows results. We haven't called this downstream service yet, so let's go ahead and comment out this table. Terrific. The last thing we need to do to just get this up and running is let's go set the server report to 8080 in the application.properties file. I'm now going to open up a terminal and start this application, and remember, it'll have no security turned on. And I'll go to the browser and open up localhost 8080. You should see the Secure App Home Page. Let's hit this /report URL and that says Secure App Reports Page. There is no security on here, both of those should just come up perfectly fine. Let's go ahead and stop the application. Let's turn the OAuth security back on. And now we can start doing some OAuth things. So let's go to GitHub and you should be logged into your account. Go to your Settings, click on Developer Settings. Here we see GitHub apps and OAuth apps. We want to create a brand new OAuth app. So what this does is you create a client application reference really that has access to the GitHub API. Let's call

this pluralsight-app. We'll list out the home page here, we'll set the callback URL, this is a specifically formed one that Spring is looking for here, it supports GitHub natively, so we're going to put in localhost 8080 where our website is, login, oauth2/code/github. This is when it, once it authenticates you, it sends it back to here in your browser and then that application page knows how to process that token. Then we'll click Register. Alright, so now we need a client secret that we can put in our code that shows that I have access to this client. I will copy that value and it will have deleted it before you take this course so don't try to use mine. Now let's go back to our code. In our application.property file, we really don't have to do much here to make this work. Let's add a couple of different configurations, one that gives the client ID and one that gives the secret. For client secret, I'm going to paste in my secret value. Let's go back to the browser to get my client id. Alright, so this is what I need for my local application to at least access that particular client once I've authenticated myself. Now let's go back into code. We need a new SecurityConfiguration class. I'm creating this class called SecurityConfig.java and what we want to do here is actually set up a configuration that says, hey, how do I run authentication here and what do I do for authorization? So we want to set up the rules here to require authentication and tell it to do more. So let's jump in. Let's actually create this class. Let's extend WebSecurityConfigureAdapter. So we're going to override the basic configure operation here and what we want to do is authorizeRequests, require authentication. So here, we want to protect the reports pages, but we want to open up the root, we want to permit all access to that and then any request beyond that is authenticated using an OAuth2 login flow, which kicks off the process with GitHub. Cool. So let's save this and run this and we should actually see the flow happen. I'm going to log out of GitHub to force myself to log in so you can see what the real process looks like. And then I'm going to open up the Chrome Developer Tools, whatever you have in your browser, but this lets us see some of the flow and the transition between pages. So I just hit the root localhost 8080, nothing happened, it just returned localhost as we expected because we permitted access to the home page with no extra authentication or authorization. What happens when I hit the report page? Okay, so it sent me to GitHub. Now away from my browser, look at the URL github.com/login so my account is not any part of my web application, nothing in my web app is collecting credentials, it's redirected me, look at the flow. It redirected me to GitHub to an authorization endpoint and it's going to be looking, it has a client ID that I've provided and it's going to provide a token back if I have authenticated correctly. I'll plug in my credentials. My account requires two-factor authentication because I make good choices and now look what happened. So the authorization server has asked me to authorize this client application, this Pluralsight app to act on my behalf. I'm going to go ahead and authenticate that and then it will redirect me to localhost. And look at that, then it sent me to the report page passing the proper code back that now my application process and spring process and said, yep, this looks good and it showed me the report page. So you can see the different routing that happened there, which is pretty neat. So a lot of this happened without me doing much, right. We updated some client IDs and secrets. We did one WebSecurityConfig class to make sure we required authorization and authentication, and then a lot of this was handled for me automatically.

Options for Authorization Servers

In that example, we saw GitHub acting as an authorization server and authentication server. That's fine for a simple demonstration, but in real life, you're going to do something a little more sophisticated. So, you might be looking at a managed service from someone like Google, Ping Identity, Okta, so a number of providers you might get as a secure provider that's providing authentication and authorization using OAuth endpoints. If you have something on-premises, you might be using something like Microsoft's Active Directory, which can serve up OAuth endpoints, things like that, so you have some choices there. There's also some open-source alternatives, and we're going to mess with Keycloak as one option. You can also use the new Spring Authorization Server. This is an emerging product, but this would also give you the ability to run your own authorization server. So, we're going to mess with something like Keycloak, which gives us authentication and authorization just to see with an open-source server might do there for us. And remember, the point of these servers is that they are issuing tokens, and they are also being able to validate tokens so that when a resource server gets that request and looks at the header and sees a token, it can check with the authorization server saying, hey, is this still a good token? Does this token actually generate the right value and the right scopes to call this class? So, it can actually then use that token to make a request back to the authorization server, seeing if that token is valid if that user can do what they're trying to do.

Demo: Standing Up and Configuring an Authorization Server

Okay, I think you'll like this demonstration. There's a lot going on in this, but we're going to walk through this step by step together, and I think when you're done some light bulbs go off on how this process really works. So we're going to stand up an open source Keycloak instance on our machine. We're going to create the realm, the client, the users necessary. We're going to configure Keycloak as necessary, the minimum configuration. We won't go nuts here, but we're going to make sure we do enough to really see how this works. We're then going to go back to our secure UI and point to Keycloak instead of GitHub and kind of do a more realistic scenario. And throughout this whole thing, we'll kind of see how things work and what it really looks like to have scopes and permissions and authorization and tokens to kind of see how this all flexes out in my client application. All right, let's get started with our custom authorization server based on Keycloak. So if you go to keycloak.org, you can start with this with a number of ways. You can run this on Kubernetes or in Docker. I'm going to run Keycloak on OpenJDK, which gives me a standalone JAR file I can execute. So from here, I'm simply going to download the ZIP file containing the Keycloak server. I'm going to drop this into the before folder. All right, and now I'm going to open a terminal in this directory so I can actually start this up. All right, from here, I'm going to switch to the bin directory, and to make sure it doesn't start on port 8080, we'll have it start on port 8180. So I'm going to run one command to kick up the standalone .sh file to kick off this script. This will just take a moment to start up the web server. We can log in, create some administrative accounts, create the sort of things we need to then have OAuth 2 authorization against this server. All right, to log in, we can go to localhost 8180/auth. From here, because this is a new server, we'll have to create an administrative user. I'm going to create one called admin with the password pass@word1. Once that user is created, we can log in to the Administrative Console, and we're prompted to log in with the credentials we just

created. Now from here, we first off creative a realm. This is kind of an area that's going to have all the different clients and roles and scopes. There's a default one, but let's create a new one called pluralsight. So here in the top left you see the option to Add realm. I'm going to create one called pluralsight. And we can see there's some clients that come with this. We're going to create a new client here representing our toll application. I'll click the Create button and create this called toll-app. So each client would have its own settings and things like that, so our toll application, our front-end application acts as a client here. Now one main value we need to add here is the redirect URL. So what happens after I've authorized the user? Where do I send the browser back to? This will look similar to what we did with GitHub. We're going to send this back to localhost:8080/login/oauth2/code and then toll-app, the name of this application, and click Save. Now we're doing authentication here too, so we actually want to have some roles and some users. So let's go ahead and create a role. I'm going to click Add Role. I'll very uncreatively call this users. And then we'll create a brand-new user. I'll click the Add user button, and we'll name this richard. You could name it something else, or you could flatter me by using richard here. Now we want to go to the Credentials tab and give this user a password. I'm going to use the same password here, pass@word1. Check this box so I don't have to change it on the first login. We'll treat this as the permanent password. Then I'll click Set Password. Next, I'm going to go to the Role Mappings, and I'm going to set that user's role that we created. I'm going to add this user to that role. Finally, I want to add a new scope. Remember, we're using this to get permissions to read report information, so let's create a new scope called reader, and we'll make sure that's included in the Token Scope and click the Save button. Terrific. So now we have a client, we have a role, we've got a new user. Let's actually prove we set this up correctly. So I'm going to go into Postman, and we're going to do a POST request, and we're going to post to localhost 8180/auth/realms/pluralsight, the one we created, protocol/openid-connect/token. We're going to log in to this endpoint and get a token back. For the body, we'll do form-urlencoded. The first value is client_id, that's the client we added, username richard or whatever you created, the password we set up, and finally, the grant_type. This is one of the grants available called password. This is not a good one because this actually then requires the client to send password in there, and the whole point of OAuth is you shouldn't be dealing with the actual credentials locally, you should be offloading that, but this is a good one to test. I click Send, and look what I get. I get back an access token, an expiration time, a refresh token, how long the refresh token has. So this is great. This is something I can use. I can see some values come back there. Now let's go back into our application code of our secureui application and update it. Instead of using GitHub, it should be using this Keycloak client. So before I do that, let's go back to the Keycloak application, let's go into Clients, our toll-app, and we need to get a client secret, right? We have a client ID, that's toll-app, but where is the secret at? Well, here, if I switch the Access Type, I switch that to confidential and click Save, and now I have a client secret that I can use in my application. So now I have all the values I need. I'm going to copy that value because we're going to need that. Now let's go back to our application code. So I'm back in the secureui app we just built in the previous exercise. Let's go to our application.properties. I don't want to use the GitHub values anymore, so let's comment those out. Instead, now I want to use the ones that refer to Keycloak, spring.security.oauth2, and we want to set the issuer URI. This is where I go to actually get the token. And this is the value we just used in Postman, so localhost:8180/auth/realms and pluralsight, the realm we created. Next, we want to set the redirect URI. So where am I sending the redirection? Before, we set the valid ones, and we set the client thing, so what will it accept? Now we actually need to provide it. So here, again, we're sending to localhost:8080/login/oauth2/code/toll-app. That's

where it should send it back. Then we have to provide some credentials here. So we're going to set the client ID. What kind of access do I have to this client? We're connecting to the toll-app client. And then we need to provide our secret, and this is the value, again, we got when we added the Credentials tab to the client overview. The last thing we need to do is say which kind of grant type am I dealing with. So the one we talked about earlier, the authorization code grant, and the value here is authorization_code. Finally, let's ask for which scopes we want to have. And if you remember, the scopes we have available to us include things like profile and roles and reader, so let's ask for the profile and reader scopes. All right, let's try to start this up, and that did start up successfully, which is a good sign. Let's go back to the web browser. I'm going to open up the developer tools in Chrome so, again, I can see any of the redirects. And remember, I didn't have to change anything in my code here. I switched from GitHub to this, making no code changes. So if I refresh here, nothing changes. That's great. Let's go ahead and switch to the report's endpoint. Okay, interesting. So what it tells me here is that my user doesn't have permissions to the reader client scope, which is fascinating. So let's go back to Keycloak. So what I suspect happened here is that our clients scope wasn't actually connected to the client. And I look at Client Scopes. Look, reader isn't associated with this particular client. So I'm requesting a scope I actually don't have permission to, so that's actually the right behavior. Let's actually go add that. Now let's try to sign in again to the report page, and now it's asking me to log in. That's awesome. So what I want to use now is the user we set up. This is richard, pass@word1. So what happened here is it authenticated myself, then I requested the token, got back the token, sent it back to the client, and now I'm viewing the Secure Apps Report Page. So I did authentication with the Keycloak server, and then I did authorization by getting a token with the scopes I had requested.

Creating a Resource Server and Routing Tokens

Now, I've shown you so far how I have kind of a client UI requesting the token, consuming that token right there in the app. But what happens when I want to pass that token to downstream services? How do I start to chain together this authorization between different services? And so here again, instead of using the old Spring Security OAuth components, the Spring Cloud Security pieces where I would be adding annotations and things like that for OAuth, instead I'm using the Spring Security domain-specific language. I'm coding some things up to pass that token downstream. We'll see what that looks like. And then what's interesting is the resource server. Let's say I have a REST endpoint serving up data. That sort of service then also has a WebSecurityConfigurerAdapter class, which is able to take in that token, tell me what scopes it's looking for. So, when it receives a token, it can validate that token gives you the correct permission. So, I've got the resource server validating tokens, I've got the upstream service collecting that original token and passing it along in headers, and then I'm going to set a few properties to make sure that the right certificates are in place, things like that. Again, it's fairly simple. Don't be overwhelmed by it. I'll show you just a couple of properties we have to do. But it's important to know this flow because you're probably not going to process everything in the front end. You're probably going to be sending tokens between services, through gateways, and so I want to make sure you understand that.

Demo: Creating a Resource Server and Routing Tokens

All right, I think you'll like this exercise. Now, we're going to take advantage of resource servers and passing tokens around. We're going to call one service from another service and send that token. So, we're going to create a brand new resource server with a REST endpoint that returns toll data. I'm going to configure a class that extends that proper component so that I can read the scope, require authentication, and activate this server as a resource server. I'm going to add some application properties to validate that JWT token. And I'm going to update the client UI with a loaded up WebClient bean, so all the calls to the downstream resource server have the right sort of headers added so that it's sending that OAuth token with the request. So, once you finish this exercise, you'll feel a little more comfortable knowing how to pass tokens between services and validate them on the other side. All right, let's jump in and build a resource server. This will be a REST endpoint that's going to serve up data, but we want it to be secured. We want to validate that you have a valid OAuth token with the right scopes before we return that data to you. So, on the Spring Initializer, I'm going to create an artifact called resourceserver. And the dependencies will be Web and OAuth 2 Resource Server. Let's go ahead and generate that. There we go. Let's open that up in Visual Studio Code. All right, and the first thing we want to do in this new project is create a data object representing toll data that we're going to be returning from this controller. We'll call it TollData.java. And again, we'll keep it simple. Three properties, a recordId, the licensePlate, and the timestamp. We'll add an empty constructor, and then we'll also add one that takes in all of the different values. And we'll add some getters and setters for each one of these and call it done. Now let's add a controller called TollController. We'll annotate that as a RestController. We'll give it a member variable that's a List of TollData, and we'll set up a default constructor that loads this up. We should be pulling this from a database, but let's keep it simple. We'll just have a local array with toll information. All right, so we've instantiated that. Now, let's actually at an operation that's going to return the toll data. All right, and we've annotated this with a request mapping so that when the page wants to display report information, it's going to pull the toll data so it can show it on the page as a report. Now, let's jump into the application properties. We'll set this to start up on port 8081 so we don't have a conflict with our web application. Now, let's set a couple of the OAuth 2 Resource Server settings. Specifically, I want the issuer URI for the JSON Web Token, the JWT. So, how do I look up where the token comes from? And then I also want one where I'm looking at the JWK. I'm looking at the certificate pieces to validate the certs. Those are the only two things I need here. And the values here are just like before. I'm setting the auth realm of Pluralsight again. And then this time, I'm setting the auth realm to Pluralsight and also checking the certs. Again, this would be used by the framework to validate the certificate that's been signing your tokens. And that's the only properties we need to set. Now the last thing we need to do is set up a security configuration class that secures any request to this REST endpoint and looks for very specific scopes. So, let's add a new class called SecurityConfig. We'll annotate this as a configuration class so it gets loaded up properly. Once again. we're extending the WebSecurityConfigurerAdapter. We're going to override that configure operation. And again, some of this is going to look familiar to what we did in an earlier exercise when we had to authorize the request on the website. We want to validate any HTTP request GETs to the api/tollddata endpoint. I'm going to look for that reader scope. Every request has to be authenticated. We're going to mark this as a resource server, so it configures it all up to be a resource server. And we're going to make sure we've got it all set up to process JSON web tokens. Great. So, this resource server is now done. It's

doing everything it needs to request permissions for anybody who wants to call its endpoints. It's going to make sure you have a token. It's got the right configuration to point to our authorization server. And it, of course, returns data, which is the whole point of the service is as a REST endpoint, it's going to send some toll data back. So, finally, let's go back to our client UI, and now let's pass the token down. So back here, since we're actually now going to be working with toll data, let's also add another class here called TollData with the exact same shape. It's got the same three properties, recordId, licensePlate, and timestamp. We'll add getters and setters for each and then add an empty default constructor, as well as one that accepts all the values that we already defined on all the properties. All right, now we can process the toll data information that comes back from our REST endpoint. Now, go to the SecurityConfiguration class that we set up earlier. Remember, this is the same one that now says, hey, the local default URL, the homepage, you can permit all. But then any request below that has to be authenticated. Here, we want to make an additional configuration here. We want to set up a bean that actually does the web client configuration. So, we get this loaded web client that actually has the tokens ready to go in the headers. Now, this might look a little complicated. Again, all we really want is a registered bean that can be annotated and added with all the different things it needs. So, what we have here, we're setting up a filter function for OAuth 2. We're setting this as an authorized client, and we're returning an instance of the WebClient builder that has all the OAuth configuration ready to go. So, then, once I have this, I'm going to save this. And back in my controller, I can autowire an instance of our web client. So, this is going to be the one now because I have the bean there. I'm going to autowire really to that bean that has everything it needs for OAuth. And now, I can use this to make a call to my downstream service and not have to do anything else myself to mess with headers or things like that. Spring takes care of that for me. So, now in this service where we're going to load the report, let's actually call the downstream service. So, what am I doing here? Well, first, you see I'm expecting back a Flux when I call my web endpoint. So, what am I calling? I'm calling localhost 8081 API TollData, the new resource server I just did. I'm going to retrieve that, turn the body of that into a Flux using the TollData class. Then, I'm going to turn that response into a list and block and wait for that response to come back, add that to the model that then can be used on the web page. Finally, let's go to the report.html page and remove the comment blocks here because now I'm going to take that toll data and show the recordId, licensePlate, and timestamp coming back. Let's go ahead and start our resource server. And you want to make sure your Keycloak environment is running as well because clearly we're going to look that up for authorization. Let's start up our secure UI, which now is going to call this downstream service, putting the token in the header so that it's authorizing itself, getting back the toll data and showing it on the page. I'm going to go to the browser and refresh this page from when we used this before. Hey, and sure enough, I see some values come back. So, that's pretty cool. So, in this case, I'm able to call a secure resource server, have it all wired up mostly by Spring with some simple configuration on my side to pass that token down stream into different services.

OAuth 2.0 Grant Type: Resource Owner Password Credentials

Thus far, we've been working with the authorization code grant type. We've been asking the resource owner for their credentials and their authorization, which then generated a token that the client app used to call something downstream. But there is different flow types, as I

mentioned in the beginning, one of those is the resource owner password credentials. Now frankly, you should almost never use this one. This is one where you'd only use this when there is a lot of trust between the resource owner and the client. You're more or less sharing the credentials directly, so here is what the flow looks like. The user would log into an app, that web app then authenticates with the username and password that you just gave the web app. Again, the whole point of OAuth is arguably that web app never sees your credentials, it's only routing you to something which is then authenticating you and generating tokens. In this case, the web app does see, even potentially store your credentials. The authorization server would validate the username and password. It would return to token directly. The web app then can request data using that token. Of course, the secure API that we're trying to call here would validate the token. If it's valid, it would return the data. This is one that you might be doing more with local testing again or if you have a ton of trust between the client and the actual resource server, but in most cases, you should be avoiding this one.

OAuth 2.0 Grant Type: Client Credentials

Now unlike the resource owner password credential flow that we just saw, you should use the client credential flow. This is one where the client itself is the resource owner. There is no authorization to obtain from some end user, there is no end user at play here. The end user doesn't have to give authorization to access the server, it's not that same sort of taking of credentials from a user. In this case, the web application authenticates itself with the client ID and client secret. You're going to validate that ID and secret and any scopes you've requested, a token comes back to the web application or web service. You then call whatever secure resource you're trying to access on the resource server. That token is validated by the resource server, and then ideally, you get some data back. So this is kind of the service account model where the client credential grant type kind of gives you an application a way to access its own service account. You're probably going to use this in most cases, first of all, again, when you have service-to-service communication, and when the client credentials can be used as an authorization grant, and the scope is fairly limited to whatever protected resources are identified with that particular client. We'll see an example in the moment. You can see how this plays out. But this model and the authorization code grant should really be your two preferences.

Demo: Client Credentials Grant Type

This is our last demonstration of this module, and I think it connects some of the pieces here, so I think you'll enjoy this one. We're going to create a new client service that returns toll data. So, right now, we've been using a UI that returns information. What happens if we're doing service to service? So, we'll define a new client in Keycloak with a unique client ID and secret that would be unique to this service. We'll configure that new client itself not to require security to call it, but it does need to send a token downstream. So that service, when it talks to other services, has to have a secure transmission, but calling that upstream service does not, just to kind of show how some of these pieces work together. And then finally, we'll set up the properties to use the client_credentials grant type. We've been using authorization_code so far. I want you to see how this one works. All right, let's jump in. We are

back at one of probably your favorite sites on the internet, start.spring.io, and we're creating a new application. Let's call this clientservice. And for dependencies, let's use Web, Reactive Web, and OAuth 2 client. All right, this will be the application that calls that resource server we created earlier. Let's go ahead and generate this project and put it in the same directory, the before folder, as the rest of our projects here. Terrific. Now, let's open this in Visual Studio Code. Excellent. So, we've got this project open. Let's go back to the Keycloak server, and we want to set up a new client that represents this service, so we could have its own permissions, roles, users, whatever we want. We could have certain things tied to this client. Let's go jump into Keycloak. All right, so let's create a new client here that's not toll-app. Instead, we want a new one that we would be having its separate credentials for, things like that. Let's call this toll-service. And we'll set which valid redirect URIs are available here. We'll use port 8083 for this new toll service we're standing up, and we're going to send it back to that endpoint with a code of toll-service based on this client ID. Next, we want to switch this to confidential access type, so again we can get that client ID and client secret. Now we see credentials up here at the top, so we'll be able to use this secret value. And then finally, for this specific flow, we need to enable this. Service Accounts Enabled setting actually lets you use the client credential grant type. Let's make sure to save that. And then finally, let's make sure we add Reader to the valid client scopes so that if we have a client scope we expect on the server like we do on our resource server that looks for reader, we want to make sure that this client is allowed to request that. Okay, this is all we need in Keycloak. Let's go back to our new application, our toll-service application. Like the other ones we've built in the last exercise, we need a TollData class that holds the toll information. You can steal the configuration from the project we just built earlier where we have those three properties, recordId, licensePlate, and timestamp. So, we just need the same data we created earlier. Let's save that. Let's add a TollController class, which somebody would then call that endpoint to retrieve some toll data. And this will look a lot like the one we created earlier, so let's have an autowired web client, which we're going to create here in a few moments. Let's have an operation that just returns a single piece of toll data. We will call this getLatestToll. This one will return a mono, a single result, in a reactive fashion. And I've got a simple endpoint that just responds to requests at the root. And then similar to the last one, we'll go ahead and call the downstream service. The same service we called before from our client UI, now we're calling from a service. So just like before, we're using the WebClient to get the resource server. We're going to turn that into a flux. We're going to turn the results into a list. And then to make life interesting, we'll just use a single result. We'll just grab the first one we have and return that back. Now let's create the SecurityConfig class, like we've been doing so far. All right, you're probably a pro at setting these up so far, so let's go ahead and enable web security on this one to light that up. We'll mark this for configuration just to be safe. We're also going to now extend WebSecurityConfigurerAdapter again. Once again, we'll override the configure operation. Now distinctly this time, I'm actually enabling all access to this service. So, remember, the last ones we were forcing authorization, things like that. This time, I'm not asking the caller for anything. This service will operate with a service account that lets it called downstream. But this service itself, anyone can call it. That's what I'm allowing here. And now I'm adding a big gnarly looking class. It only really does a couple of core things here. What this does is it loads up the client configuration so that it can run that sort of client configuration flow. Specifically, it's letting me build up the clientCredentials OAuth2AuthorizedClientProvider. And then, it's using that. So, it's going to read those values, build that up, take values from configuration, make sure that it passes those forward. And then finally, I'm going to set up a bean that's actually using some of

those OAuth2AuthorizedClientManagers that we just created earlier. And in this case, what it's going to do, again, look for that Keycloak client registration and build up a WebClient that, again, it's all set up with the OAuth 2 stuff. So, I don't have to do any fancy stuff when I'm actually using the WebClient. It's all been preconfigured in autoconfigured when the application starts up. And now I'm going to jump into the application properties file. First, I'm going to make sure this starts up on port 8083. We set that up back in Keycloak as well. It's expecting that. Because I've been a little bit lazy and put some of the same objects that relate to each other in the same class files, I'm going to allow some circular references here. And now we get to the interesting stuff. So, let's go ahead and refer to our client ID and client secret. So, we know the client ID was toll-service. If we go back to Keycloak, it's toll-service, and then we need this secret. So really, those act as the credentials. Now we have to say that we want to register and expect that we need the reader scope. We're using the Keycloak client, and we're going to use the scope of reader. That's what we're requesting here. And now, for the first time, we're using the client credentials grant type. And once again as well, we're providing the token URI. Where do I go get tokens, how do I validate stuff, all those sorts of things. Where do I go find that? That's it. So I can save this. Let's make sure our resource server is up and running. And before I started this up, let me jump back into the TollData class. I think I got a little lazy. I need to make sure I have a public empty constructor as well so that this can be started up. Now with all of this, let's go ahead and start this app up. All right, we'll start that up. And now when we call our client UI, hopefully just at localhost 8083, no credentials required for me the caller, then it will use its client credentials to put that in the header to make the downstream call with an appropriate token. And what do we see there? Look, it returned the first license plate that it got back in the result. So, while this service had no authorization authentication, the service itself had some kind of service account credentials it used to get an appropriate token, put that in the header, use that to call my resource server, get back the secure result, and show it to me. So, we tied together hopefully a few concepts there for you, and you can see how some of these flows work, even in these scenarios where maybe one part of this doesn't have authorization and authentication and another part does.

Advanced Configuration Options

Now Spring Security should and does have fully dedicated Pluralsight courses. There is a ton here. At most, I've given you a taste of certain pieces to at least tease you with what's available, but there is a ton here and a lot of advanced configurations, even just for OAuth 2. So you can plug in a variety of providers. You saw that GitHub is supported out of the box. There's other one supported out of the box. Even Keycloak has its own kind of plugin components you can use specific to that. I wanted to show you how to use that more generically, but there's a really nice plugin model here. So as you're working with different OAuth 2 providers, you get a nice experience in Spring. And I showed you a lot of autoconfiguration, plenty of things you can override as well. You can change where the redirection endpoints go when someone logs in. You can change the shape of tokens, how things get decoded, so a lot of control over how the OAuth 2 behavior looks. You saw how we created a few beans to do things like switching to client credential types versus authorization code. So a lot of things you have control over. You can do a very simple flow. You can do very complicated flow. A lot of things you can override here. So look deeply into this. Hopefully this module inspired you to play with this a little more, and hopefully I made this a little less scary.

Summary

Alright, that was a big module. A lot of stuff was going on in there. We talked about the wide range of things that come into play with security in microservices, as you think about data security, transport security, authorization, authentication, single sign-on, big wide surface area there. We zeroed in primarily on authorization here. We talked about the problem with the status quo as you start moving more towards zero trust, not trusting everything by default, and instead requiring that each service might be validating the authorization of the caller. That requires new thinking, and frankly, new frameworks. So we talked how OAuth 2.0 is trying to solve some of this, getting applications out of the business of doing authentication and authorization, helping you offload that, and also giving you a clean way to have a token-based model for issuing tokens for permissions, validating those tokens. We saw that Spring has some pretty robust support for OAuth 2.0. An easy grant to get started with is that authorization code grant, usually when there's a client involved that can request permissions from the resource owner. We looked at the different authorization server options and got a little deeper in Keycloak. Lots of ways you can do this, but hopefully that helped you understand some of these flows. We then looked at the resource owner password credential. Again, be cautious with that one. That's not one you should use very much, as it requires the resource owner to actually give the password to the web application which you're trying to avoid with OAuth. Then we saw that the client credential grant type is a little better here, as instead I'm treating this more like a service account and just using the client ID and secret to authenticate. Now we talked a little bit about advanced configuration options, so again, hopefully this inspires you to explore this topic more. Honestly, security isn't always the most exciting topic to myself as a developer, and maybe you as well. You have to do it, but it's not the most fun thing. But I actually enjoyed this module, I think there was a lot of cool stuff to see how this works in Spring and in Spring Boot. Hopefully this inspired you to play with this a little more. See you in the next module.

Chasing Down Performance Issues Using Distributed Tracing

Introduction

Hey, everybody. My name is Richard Seroter, and welcome to this next module in a course about building Java microservices with Spring Cloud. In the last module, we talked about security in microservices, and in this module we'll look at how do you chase down latency issues in services that you've built. Specifically, we'll talk about the role of tracing in microservices. What is the problem we're trying to solve? And what are some of the issues with the status quo? How do we chase down these problems today? We'll talk about Spring Cloud Sleuth. Again, this is an interesting project, so as you're building microservices, this can play a big role. We'll look at how you set up and use the Zipkin project. We'll create some traces, we'll customize some sampling, we'll actually create our own spans, and then we'll wrap up.

The Role of Tracing in Microservices

So the microservices architecture, now we don't have one place we go for our application. Our application might be distributed amongst different services running in different places. And so as we think about tracing in microservices, well, this helps me locate what component is not behaving correctly, why is this slow, why is something going wrong. This is harder to do in a distributed system, so tracing can help me find out where latency is happening or uncover the source of faults. What's also really important here is starting to observe end-to-end latency. The performance of one service doesn't tell the story for your customer. Your customer doesn't really care if one service is slow. What they care about is their experience. How was their end-to-end experience? How was it working with their entire transaction they're trying to do? You need to see where are the hotspots in the entire flow. And then you also want to be able to understand the actual emergent behavior, like what's actually happening here. When you have a composable architecture, you might have paths that weren't originally designed. Maybe this service calls this service you didn't even realize that. People create apps out of services all the time. So you may have call patterns to services you didn't anticipate, and therefore, can't see. And so you're looking for tools that are giving you a live view of exactly what is the transaction, what is the actual call graph amongst all of your services.

Problems with the Status Quo

What are some challenges with what you might be doing today that could make distributed tracing a solution that makes a lot of sense for you? Well, first it's about instrumenting all the different communication paths. Today it might be tough, because traffic isn't just going over HTTP. How are you capturing performance information for traffic through message busses, things like that? Do you have what's necessary to catch all the communication between the services? Today's asynchronous, multi-threaded, multi-server services can make instrumentation kind of tough. And what about extensibility? Could devs attach their own events? So are you getting enough logs, enough information from all the different components? And then it can be tough. How do you actually collect all those logs in a centralized place so that you can correlate them, connect the dots? Oh, this service talked to this service, and therefore, see what's going on. Have you standardized on what you've collected and where you put it? And then, are you seeing the bigger picture? Today, again, this can be tricky if I can't even visualize this. How do I see a complex request tree? How do I see that call graph? And can I see outliers? And so sometimes the status quo, again, caters towards more centralized monolithic systems. As I move to more distributed architectures, these are problems that are going to emerge.

What Is Spring Cloud Sleuth?

So, the answer here for Spring Cloud developers is Spring Cloud Sleuth. There's some automatic instrumentation of all your different communication channels that can be fed then for later analysis. The goal is to understand our architecture, troubleshoot these production latency problems. That means I need near real-time data collection of all my timing information. I need identifiers that can connect which services are all part of one single call graph. All of this is part of what Spring Cloud Sleuth has to offer. Let's define a few terms,

shall we? Let's make sure we're talking the same language when we talk about Sleuth. So first, there's an idea of a span. This is an individual operation that took place. It's a basic unit of work. I have some timestamped events that start and stop. A trace is the end-to-end latency graph. It's made up of spans. So, a trace contains many spans. It puts those spans into context. Now there's also annotation. So, in a span, I'm going to have things like, hey, the client made a request, the client sentiment. I'll have a ServerReceived annotation, a ServerSent annotation, a ClientReceived annotation. So, I'm seeing all these different steps that happened for that given span. When did that request come, when did I get it, when did I send a response, those sorts of things. And then finally, I have a tracer. These run inside your production apps. They help create the spans. There's specific tracers included with Spring Cloud Sleuth. And they're made to kind of manage the volume of things. They're sampling the information to send downstream. So, they really kind of handle that proxy between this and then whatever engine you're going to send things to eventually.

Anatomy of a Trace

I've talked a lot about visualization so far in this module. Let me actually show you something. In this case, let's look at a trace end to end, and let's assume I have four services, two back-end services, 3 and 4, Service 2, which is doing some aggregation, and let's say that Service 1 is actually the client-facing one that's serving up a request. All four of these play a part in one total request from the client. So when that request comes in, it has no trace ID. There's no span ID yet. This is brand new. Remember, a span ID is really associated with a unit of work. So now I have a trace assigned, which should be uniform through the entire request, but I have a unique span ID for this unit of work. That service then makes a request. This unit of work now has a new span ID in a client-sent event or annotation. I've also marked when that server received that event. Again, this is all automatic. This is all traced automatically by Spring Cloud Sleuth. Now I have a new unit of work because Service 2 is going to call Service 3, so I have a new span ID, same trace ID. I make that request. I've got this new unit of work of the actual request and it being received. That service then does some work, so it generates a span. Now I'm going to send the response. I'm really closing the event from Span ID D. That response comes across with a client received event. I now make the final request. That's another new span ID, same trace ID. I have a client sent server received combo. That service does some work, generates a span. Now I'm going to send the response back. All part of the Trace ID X. Same thing, same span ID for that combo. Now that work done on Service 2 to get all those replies is closing. Same with Span B. I'm sending a response back to the original service. Trace ID X, Span ID B, getting that response back. Now I'm closing really Span A and sending a response back to the customer. All of this had the same trace ID, so I'm able to actually pull all this together and visualize this. This gives you some sense. There's a lot of spans generated throughout the process each time there's a unit of work. You can see them open and close based on the request and response.

What Is Instrumented, and How to Add Sleuth to a Project

Now I want to take a moment to call out what's actually automatically instrumented here. Some of the power of Sleuth is it's not just handling web requests; it's handling a number of things because your distributed microservices architecture may have a lot of

different types of components. You might be sending things through Spring Cloud Gateway. And so we're going to talk about this in another course, but if I have a gateway, an API gateway sitting in front of my services, that should be participating in this process so I can see how its request's doing in the calls. Is it passing the right headers along? Things like that. So this is automatically instrumented. Spring Cloud CircuitBreaker as well. So as I'm handling circuit breaker scenarios where services are failing over to another service, things like that, I want to see that in my call graph; I want to know that's happening. This does fully support WebFlux in the more reactive model, so I'm able to process things there and see what's happening. That's automatically instrumented. I don't have to do anything. What's great too when you think of the WebClient, the RestTemplate, whether its synchronous or asynchronous, all of that's instrumented for you. Nothing you have to do there. You're just going to start collecting spans and traces. If you're working with Spring Integration, Spring Cloud Stream, Spring Cloud Function, you also get some instrumentation there, which is really handy when you're not just doing web requests. And as you have operations in your code, things that are marked as async, things that are marked as scheduled, this all works great as well. So you have a lot of robust things covered for you, automatically instrumented, so in most cases, there are very few things that would slip outside of this. And Spring Cloud Sleuth is extensible, so it's really easy to add your own spans, introduce things to your own traces in your code if you have to, and we'll do that in a later exercise. What does it mean to add Sleuth to your project? Gosh, it's really simple. All you have to do really is one starter dependency, and in this case spring-cloud-starter-sleuth brings in what you need, at least to get Sleuth going.

Demo: Adding Spring Cloud Sleuth to a Project

We'll do some fun little exercises here in this module. We're going to actually start learning Sleuth right now. So what we're going to do is we're going to take some pre-built services, I'll save you some time, but we're going to add some Spring Cloud Sleuth to them so we start generating traces and spans. We're going to update our properties file just to make sure we're exposing those. We'll see those in the logs. And we'll test the services and see the output. To get started, all we're really going to be doing is generating the traces,.Then well, in the subsequent exercise, actually start consuming them. But the first step is, let's make sure our code is instrumented with Sleuth. In the before folder of this module's Spring Cloud Sleuth directory, you'll see 3 folders, dataservice1, dataservice2, and customerservice. Let's open up each one of those in Visual Studio Code, and we'll quick run them, just so you can see what they do. We'll go ahead and run this application, and that's just started up, and what you'll see this does is it respond to a request for customer details. So you'll pass in a customer ID, get back some details. You'll see that I'm adding some latency automatically to this service somewhat randomly, so it'll be slower in some cases and faster than others. I'm now going to go to Postman and do a GET on 8081/customer/501, one of the IDs I automatically loaded into an array, contactdetails. And I get back a customer record. Great, that service works. We'll close that. Let's open up dataservice2. This service then gets the vehicle details associated with that customer. Let's start this up. I'm going to switch back to Postman. This first service was on 8081. This service is now on 8082/customers/501/vehicledetails. And sure enough, we get back our car information, terrific. And now let's open up the customerservice. This is the aggregate service that calls both of those data services. As you can see, it has data objects for VehicleDetails and CustomerDetails. This web client is calling the ContactDetails endpoint

and the VehicleDetails endpoint and combining that all into a single new customer details object and returning that. Let's start this one up. Great, and for this one to work, the other two must be running, so let's go ahead and start up dataservice1 and dataservice2. And we'll go back to Postman. This one is on port 8080, same customer ID we've been using so far. And we get back a combo record that includes the contactName, the carType, the licensePlate, so I'm combining the ContactDetails and the carDetails. Great, so everything works terrifically. Let's go ahead and stop all the services and add Sleuth to them. So on dataservice1, let's add Sleuth to our dependencies. I'm going to right-click the POM file if I'm in Visual Studio Code and add a new starter and save that. That's all I have to do to enable Sleuth to start with. To make sure I have everything I need, I'm going to jump into the application properties file, though, and just make sure I have a nice, friendly name attached to this. And I also want to kick up my logging level so I see all the different Sleuth logs directly in my console here once I start up my application. So I'll set the logging level to DEBUG. I'll do the same for dataservice2. We'll add the new POM dependency, we'll set the spring application name and logging level, and then finally, we'll do the same for the customerservice. And now finally, because the customerservice is going to be calling these downstream services, I need it to be adding things to its headers. The other two services aren't calling other services so nothing has to change there, but this first one needs to start sending that trace ID across so that they all have the same value. So here in the CustomerController you can see that right now I'm creating a new web client on every request. It's fine if I'm doing a call this way, but now I actually want to create a bean that is going to load up that client with everything it needs for Sleuth. So I'm going to open up the main class associated with this project and add a new bean. Now there's nothing fancy in this. I don't have to do any extra extension thing. I just really need to create a bean so that then Spring Cloud Sleuth can do all the appropriate activation and autoconfiguration of that. I'll save that, jump back into our controller, and now add an @Autowired property. And now it'll use that being that we've added that's been autoconfigured. In my code then, I don't want to use this one anymore. I want to use that autowired one that's all set up. So now just making this call, Spring will automatically set the headers for traces. So let's go ahead and start all three services up and observe the log files. Let me start up dataservice1. Great, we started that up. Now, let me make a quick call to that directly. Let me go back to Postman and let me call that first service again. Remember, this is the service we added some variable latency to. Let me switch back to Visual Studio Code. And now what you see here in the info, you see the name of my service, dataservice, you see a trace ID and a span ID. Now again, this isn't coming from anywhere but me directly calling it, so these values are the same. But we're seeing this information logged now. Before it got started, of course, there was nothing there. Awesome, so that works great. Now let's start dataservice2 and customerservice. All right, they're all started up again. You can see we're logging extra information. Now I see customerservice, no trace, no span. So what I want to do is call the customerservice, and what I should see is that the downstream calls of dataservice1 and 2 should have different span IDs, of course, but they should have the same trace ID. Great, let's go back to the customerservice. Great, we see customerservice has a span ID and trace ID of 7af, in my example here. Should be the same because it's the same service. Let's see if our downstream services have that same trace ID. Let's go to dataservice1. Sure enough, it does look at that has the exact same trace ID, dataservice2 as well. They have different span IDs because they're different units of work to call dataservice2 and dataservice1 and things like that, but I have the same trace ID. So you can imagine I'll be able to visualize these all the same because I have one connective tissue between every service call.

Visualizing Latency with Zipkin

Now I'm not going to ask you to visualize latency by studying log files, I'm not that mean to you, so instead, how do we visualize latency, how do we see the information besides just looking at log files? Well, one good answer is Zipkin? This was originally created by Twitter years ago as a way to actually surface up some of this tracing information and let you actually see what's going on. And it collects the timing data. So the tracers are actually helping collect all of this information you're seeing in the log files and then present that information. So the tracing systems themselves take data the tracers collected and show it. It puts those spans into a tree, gives you some query tools, some visualization tools for analytics, things like that. Zipkin itself is a distributed tracing system. You have collectors, you have storage, a web inquiry server, and a user interface. What's great here is it shows lots of different dependencies. So I can see which services are calling which services, I get a visual sense of where this is happening, and I can see if there's a service that takes a long time, there's a server erroring out, things like that. So visualizing that latency for spans. And what's great here is Zipkin does support many different sorts of integrations. So I'm visualizing latency not just for Spring Boot applications, but Zipkin works with JavaScript, Ruby, C#, Go, PHP, and others. So as soon as you close a span, things can get sent to Zipkin automatically, which is great. This does work with both streaming and HttpChannels. So all of this is terrific. I get a great way to visualize all of this. How do I use this? Well, to be clear, in old versions of Spring Cloud Sleuth with Zipkin, you would actually stand up your own Zipkin server with annotations. That doesn't work the same way anymore. Now you use an out-of-the-box Zipkin server somewhere. But what you do want to do, let's say you're using the HttpChannel, is you simply add spring-cloud-sleuth-zipkin to your dependency list, and now your sleuth application knows how to send all these spans to a remote Zipkin server. The Zipkin server is great. It can have some in-memory storage, of course, for demonstration purposes, but you can point this to a MySQL instance, it can work with Elasticsearch or Cassandra, things like that. But you don't have to deal with that as a developer. You're just going to assume there's a Zipkin server out there that can handle all of your spans and show all your traces and let you analyze that.

Demo: Standing Up a Zipkin Server for Your Apps

Alright, let's visualize some latency, let's see what's going on here. So we're going to start by actually getting Zipkin up and running. First, we're going to download a prepackaged Zipkin server. This is free and easy to do. We're going to start up that server, and then we're going to update our services to actually send the spans to Zipkin. So, we're going to do a very simple exercise here and then pull up the dashboard and make sure it works, but we're just going to simply get Zipkin up and running, which takes a moment, and then we're going to update our services to point to the Zipkin server. Let's go into it. Alright, I'm back in the browser and I'm at zipkin.io, and I'm looking at the Quickstart page, and so there's a lot of ways you can download a Zipkin server directly. I could use Docker, I could do a lot of different things, I'm going to go ahead and include the latest Java release, which gives me a self-contained jar file. We're going to drop that into the directory for spring-cloud-sleuth. Great, now I'm going to open up a terminal pointing to that folder. Now I'm going to start up this Zipkin server with a really simple command, and actually to make my life easier, how about I just rename this jar to Zipkin, and now I can just start this with a java -jar command, and we now have the server

up and running, the default port is 9411. Technically, I don't have to change anything in my code to point to that because it looks we're there by default, we will set that by default. But let's jump back into dataservice1, and now let's add our dependency on Zipkin. I'm going to add a starter to my pom file, I'm going to pick the Zipkin Client, and you see it added spring-cloud-sleuth-zipkin. I'm going to save that, let's go into our application properties, and again, this is optional because it knows where to look for the default one, but let's manually set where the base URL of our Zipkin server is. Excellent. Now let's do the exact same things in dataservice2 and customerservice, we're going to add that pom dependency, and then we're going to update the base URL. And then finally, let's update our customerservice. Perfect. Now let's start each service up and make one call to it. We'll switch over to Postman, make a request, we should still see that information in the logs, and then let's simply open up the Zipkin dashboard at localhost:9411/zipkin. A simple query shows this sees the customerservice, so we did indeed send some traces over, so we have Zipkin set up correctly and configured.

Visualizing and Querying Traces in Zipkin

In that exercise, we got Zipkin up and running, which was terrific, but let's talk a little bit more about how do I visualize and query the traces in Zipkin. So first off, I can view the dependencies, I can see what's going on in a given service, and it'll show me based on the trace information which services depend on which. It can determine that automatically by looking at the call history. I can also look at a very specific trace and view the details. What are the services that have participated? What was the performance like? What is some of the metadata associated with that? I can do some queries, I might query and say, hey, show me every service of this type, show me every service that took this long. That means that I can look at this information, look at durations and latency, query by that, look at different criteria. So the Zipkin interface is fairly straightforward, but that built-in dashboard does give me some flexibility there, and if I'm using another service that implements Zipkin, I might get an even richer experience, but the basic experience still gives me a nice way to figure out what are my dependencies and what happened with that trace.

Demo: Visualizing and Querying Traces in Zipkin

Now that we have a Zipkin server up and running, now let's get a little deeper into this. So we're going to view the dependencies between our services using that dashboard, we're going to analyze the details of some of our traces, we're going to filter by time duration. There can be times where I would say, look, there's a lot of requests coming into the service. I'm not browsing each of them. Just show me the instances where it took, let's say, more than 5 seconds. Okay, now I want to drill down and see what was the problem, what was going on here? Now I have some idea of how to pinpoint the problem, not just looking at everything. So we'll look for some long traces. So we're picking up where we left off with our last exercise. Just in case you don't have it running anymore, let's make sure you're Zipkin server is running. That service should be running. Let's go back to Visual Studio Code. Your dataservice1, dataservice2, and customerservice should all be running as well. Now let's go to Postman and issue some requests. Remember, we have variable latency, so we want to issue a few requests so we get some different types of data. All right, that's great. Now let's go

back into the Zipkin dashboard, and first, let's look at the Dependencies button here at the top. I can pick a time period, and it's showing me which services depend on which. You see the customerservice calls the dataservice, and customerservice calls dataservice2. This can be determined automatically based on the call graph, and you can see it actually knows the number of requests over this time period. It's sending little bubbles over there, which is kind of neat. All right, let's go back to Find a trace. I can run a query; I can see all of these different calls. It shows all the services it's recognized so far, so I could pick customerservice. Now you're seeing here it's automatically sorted by duration, fastest service and slowest service up here at the top. This one took 3 seconds; this one took 21 ms. If I click the down arrow, I can see the trace ID and which services participated. If I click the SHOW button, now I get a deeper view. I can actually see the latency, I can see the spans, I can see the path that was called, the controller class, the actual method that was called, the source IP, I can see things like the client start and server start and server finish, all the different annotations we talked about. Now let's look for long traces. We only have a few results here, so it was easy to eyeball this, but let's actually do a query. In this top box, if you click the plus, if I look for a minimum duration of 4 seconds, nothing comes back. Nothing took that long. If I look for a minimum duration of 3 seconds, I see one offender. So I'm able to do some queries with a few values here based on tags, maxDuration, and serviceName, so a few ways I can easily drill down into my service set, look at the details, and start to figure out what's going wrong.

Working with Sampling and Skip Patterns

Let's talk a little bit about sampling. This is what the tracer does, in this case, the Zipkin tracer, and how many spans does it actually send in process. By default, Sleuth is exporting about 10 spans per second with the Zipkin tracer, so, there is no default that I can see on the actual percentage that's not a default value, you can set it from anywhere from 0 to 100%. So you can set the probability and say, hey, I only want 10% of traces to come through, I don't care how many there are, I only want 10%, I don't want to flood my system, I'm getting a million requests a day, I don't need all those traces. Sleuth also then limits that export to 10 spans per second, so if you do get flooded and get 1000 requests per second, you're not drowning your downstream tracing system. So you have ability to set all of these values. You could up your per second value to 1000, you can switch your probability to 10%, you have a lot of control over this. But the idea behind sampling is I want to prevent overloading the system by constantly tracing just some, not all requests, and you want to be careful really with any sort of rate above 100 traces per second, because that could overload your tracing system, so typically, you don't want to be able to go above that, so you might want to set that property just to be safe. What's also kind of cool is you can create your own custom sampler, it's easy to do that in code. You can also create skip patterns and say, hey look, I don't want to trace every endpoint, there might be certain service endpoints, actuator endpoints, there could be certain operations that you don't really care about, you can do all sorts of things and actually define a skip pattern so that it doesn't trace any of those requests.

Demo: Experimenting with Samplers and Skip Patterns

Let's mess around with sampling just a little bit. So we're going to mess around with some of the percentages, only sending in 10% of traces. See what happens there. We're going to add a quick operation to our controller, and then also define a skip pattern because we don't want to see any of that noise in our tracing system, and then we'll view some of the logs in Zipkin results as we make these changes. All right, we're back in our environment. I started and stopped my Zipkin server to erase all the traces. By default, it's using an in-memory database, so that's a great, easy way to destroy everything. I'm going to go back to my CustomerService and stop that. And I want to set some properties here. So let's only take 10% of the traces to get exported into Zipkin. So you see for sampler, I have probability, I have rate, things like that. Let's set the probability, 0.1, which is 10%, 1.0 is 100%. So if I now start this up, to be really clear, I'm still tracing everything. So if I make 10 requests, you will still see 10 requests in my log, but the key is the tracer is only exporting 10%. So let's go make five calls, for instance. If I look in the logs, I still see all of these requests. If I go to Zipkin, I still haven't seen anything. Let's do a few more. All right, so I've done 10 total requests, and I finally have something in there. So I'm only sampling a little bit of this. So my tracing system isn't getting overwhelmed even though I'm actually still collecting all of that trace information. I'm not exporting all of those. Awesome. Now let's try out a skip pattern. Let's go back to the CustomerService and stop that. And then our Controller, let's add a new operation, and it's just a silly operation called getStatus. Let's say this service has some sort of status endpoint that we want to invoke because we don't like to use actuators for some reason. Let's just add this operation to it. Now, by default, this will get traced, right? Any request into this would get traced, it would get exported. We don't want that. So we'll switch this back to 1.0, and now let's add a skip-pattern. So this skip-pattern says which URLs do I want to skip? I want to skip the servicestatus endpoint. So I can call that endpoint. And, again, I still see requests going to servicestatus. Here, I'm doing a GET request. I'm still getting it, but if I go back into the Zipkin server, even with 100% of my traces I'm not getting that back now. So that skip pattern gives me a way to make sure I'm not accidentally sending more than I want to to the tracing system. It gives me some control over what goes over there. So I can use the sampler percentage, I can use the sampling rate, and I can use skip patterns to control what goes out to my Zipkin server.

Manually Creating Spans

The last thing I wanted to talk about is, how do you manually create spans? We're seeing these things get automatically instrumented and created, thanks to Sleuth, but what if you're running some operations in your code and you want to actually purposely designate this as span? Maybe you're making a database call and you want to trace the time between if you're calling an external system in some other way. How do you kind of inject yourself into this process? So you can absolutely create brand-new spans yourself in code, fairly easy to do. And you're going to do this if you want to track any actions that aren't automatically instrumented. And, of course, you can continue existing ones. Maybe this is the next daisy-chaining part of a span, so you can actually continue a span in another operation. I can explicitly associate it with a parent. Maybe there's another thread getting spun up, or this or that. I can say this is your parent span. And I can add metadata too. I could have additional

properties, tags, things that I would use to query or analyze this information, so I have some nice control over this.

Demo: Manually Creating Spans

In our last exercise of this module, we're going to actually add a span to a data query service and we'll pretend we're doing a data query service. We're going to include some tags on that span. We'll, of course, then call that microservice and want to see that new span in Zipkin, including our metadata. So we'll do some simple stuff here, but I want you to see how to add your own spans to your code. Alright, let's manually add a span to dataservice1. I'm in dataservice1, I've stopped it if it was running from before. Let's go ahead and open up our controller, and the first thing we want to do is auto wire the Sleuth Tracer into the controller. So now I have access to the wired up Tracer, nothing else I have to do there, which is pretty cool. You'll see here that we added some arbitrary latency. Let's actually pretend that this was a database call. So I'm going to go wrap this up in a span call. So first, let's actually create a new span, and we'll use the Sleuth one, we'll call this dbSpan, and we'll give it a name, DBLookup. So now we're going to wrap this up in a try finally block, and so what we're doing, we're going to try getting this Tracer started up using this new span we just created. These are arbitrary values that can help us with searching or just understanding what happened, and we can also add an event, so maybe when this query is done, we want to actually log an event. Now interestingly, this doesn't show up in the Zipkin UI, but it does show up in the exported data that I'll show you, and we always want a finally block because we want to clean up our spans and end our span. That's all we need. So let's start up dataservice1, dataservice2 should already be running, customerservice should already be running, we're going to call the primary service. Alright, let's jump back into the Zipkin dashboard, I stopped and started it so I have no traces again. We have a couple of slow ones here, so let's go ahead and show it, and look what I have, I have a new span called dbLookup, that's exactly the one we created. Here's that tag we added, sql-database, which is terrific as well. So you can see we actually added our own thing here. So when I look at this, why is this useful? Well I can say okay, dataservice1 is slow, but wait a second, maybe it's because this operation is slow, the whole service is fine, but this one operation in there is killing me. So, all of a sudden I'm getting some additional visibility into what actually happened in that operation, I can't just blame the whole operation, in this case I can say, oh it's the dblookup that took over 3 seconds of the whole thing, that's where I should go find the problem. And if I do download the JSON here, and open that file, you will see that it does actually add that additional event information we put in here, and there it is, "value": "db lookup complete!" So if you want to use the raw data or you have a system that actually can process all the information that comes across in the Tracer, you'll actually see those custom events as well. So here we actually added some spans to our data query, we added some tags, some events, we called it, and we saw that show up in Zipkin.

Summary

Distributed tracing is one of those underrated but super important parts of actually developing modern microservices. It's when we think about how do we actually see what's happening in this set of services running all over the place getting called

synchronously, asynchronously, and so forth. Tracing is huge, because currently you might not have everything instrumented, you might not be able to actually see the dependencies between your services or see some of the unexpected relationships. Spring Cloud Sleuth is pretty cool, it actually instruments a lot of your code and also enables you to export that to something like Zipkin, so I can visualize, I can query, I can analyze, which again, if you have a problem, if your system is down, you don't want to waste time looking at everything, you want to know exactly where to go. So you need tools that help you pinpoint where exactly is the problem. Distributed tracing is a powerful tool for that, and it is fairly customizable, so you can add new traces and spans, you can do whatever you need to do to get your system online quickly. Well I hope you enjoyed this module and learning about this. I hope you enjoyed this course. Developing microservices with Spring Cloud is a rich, fun topic as we do configuration, as we build functions, as we do security, do tracing, this all makes building these services easier, makes them more maintainable, and helps us build an architecture that's both powerful for our customers, maintainable for us, and exciting to change in the future. Thanks for joining me in this course. Please give me some feedback at @rseroter on Twitter, add some comments here in the Pluralsight discussion, and I appreciate you watching.