Course Overview
Course Overview

Hi, everyone. My name is Bryan Hansen, and welcome to my course, Spring MVC Fundamentals. I'm the director of software development at Software Technology Group and the CEO of Complete Programmer. Spring MVC is a full-featured yet lightweight Java framework that has taken the web development space by storm. Spring's integration is excellent with just about any other Java tooling, and Spring MVC is no exception. In this course we're going to cover the fundamentals of Spring MVC. We will cover all the major parts of Spring MVC, including container and containerless deployments, both standard controllers and REST controllers, JSP pages and the Spring alternative, Thymeleaf, as well as consuming Spring MVC from client-side JavaScript. By the end of this course, you should feel comfortable developing with Spring MVC. Before beginning the course, you should be familiar with Java, and although not necessary, some Maven experience will help. From here, you should feel comfortable diving into Spring with courses on Spring security, Spring data JPA using Hibernate, or having a better understanding of architectural design through the design patterns in Java courses here on Pluralsight. I hope you'll join me on this journey to learn Spring MVC with the Spring MVC Fundamentals Course, here at Pluralsight.

What Is Spring MVC?

Version Check

Introduction

Just as Spring took the Java development space by storm, Spring MVC was equally as successful in the web space. When I originally wrote this course for Pluralsight, Spring MVC was just as popular as any of the other frameworks that were out there. But since then, it's become the standard for Java web development. Let's talk more about why and how it's used today because it's use has changed quite a bit over the years. The first release of this course was centered around strictly front-end development. Most Java web apps at the time were written using JavaServer Pages for the front end and had a very simple controller-based architecture. Later, people started looking for an alternative to SOAP services, and out came the REST service support in Spring MVC. REST support early on was quite simple, and Spring previously had a stance that they didn't want to try and support all of the features of REST and just the two most common ones, GET and POST. It has since become very full featured and easy to even expose endpoints from your data or model tier if you want. As JavaScript-heavy front ends took over, it was only a matter of time that single-page applications, or SPAs, started to dominate the front-end space. Spring MVC's full-fledged REST support made it a great service layer for SPA apps. Throughout this course, we're going to look at all three of these scenarios and how to use them inside of your applications.

Course Update

Spring's architecture is a traditional model-view-controller design pattern. Since it does follow this methodology, it's adaptable to many scenarios in which we can develop apps. We have a

module for each of the parts of this pattern throughout the course, but to give you a brief introduction, the view can be a simple REST service or JSP page and can handle more complex scenario such as compiling those JSPs and even work with various third-party templating tools and frameworks. The controller is the same regardless of what the front end or back end that we're using is. It's all written using Spring beans and configured just like any Spring bean. The back end, much like the front end, can be swapped out with various persistent strategies, JPA, Hibernate, JDBC, even non-relational databases. Spring has support for them. One thing that is different now than when we originally released this course is that all of this can be figured with or without XML. Upon the initial release of this course, there was a lot of debate between existing frameworks out there and which to use. Simplifying this course with the update, we opted out of discussing these comparisons of those various frameworks. One thing that has changed, though, is the number of apps now being completely self contained or apps being developed without XML at all. Specifically, the lack of web.XML or with later servlet specifications, any XML at all. Showing the difference of this approach will be the main focus in the update of this course, while showing you that model-view-controller design pattern.

Request / Response Lifecycle

The request/response life cycle is actually a tricky one when you look at it. The big scope of everything that's going on here seems a little bit overwhelming at first. We have an incoming request that hits our front controller, that's just our dispatcher servlet inside of Spring, and that hands off the request and delegates our request over to one of the controllers that we've set up. So the dispatcher is just the router that comes in and says what controller's going to handle it? I'm going to delegate this request over to that controller over there. The controller just routes traffic to where it's supposed to go. So it says, I'm going to handle that request and hand it over to the back end. Now our back end could be composed of web services or a database or multiple databases, any number of things that we're gathering data from. And in turn what that does is it hands back a model object to us. So it creates that model and hands it back. The model is basically just our data, what we're trying to represent on our screen. Once that gets back to the controller, the controller says okay, I'm going to let somebody else now do the rendering. And the rendering is separate from the business logic. So we have this model-view-controller design pattern, and everything is doing its own specified functionality here. So coming back to the controller, we have our model now, or our data that we're going to try and represent the UI with. So we go back to our front controller and it says, oh, well, now who's going to handle this? Who's going to render this response? Let's pass our model down to some view template. For this course we're going to use JSP, but we also are going to look at examples using Thymeleaf. There's many other different technologies that integrate nicely with Spring MVC as well that you could use. Once it's gone to our view template, our JSP page, we're going to return control back to the front controller and then return back our response to the browser. Now this whole thing is kind of divided up into three different parts. We have our model, so where we get our model created and the data associated with it, our views, so our representing view, and our controller. That's where our model-view-controller is coming into play with our entire application.

Vocabulary

Before we get too far into our application, let's get some simple vocabulary out of the way. First, we're going to start with a SPA or a single page application. You've probably heard of that before, but if you haven't, most people refer to a single page application as a SPA. The next piece is a DispatcherServlet. This is our entry point into our application, and it's really where a Spring MVC web app begins its configuration. We always start with the dispatcher servlet. Then we have a controller. The controller is actually an implementation of the command pattern handler. If you really want to understand architecture and understand the design behind this, you can go look up the command pattern design pattern and read more about it, but the controllers are always an implementation of a command pattern handler. The next piece that we have is a request mapping. Everything inside of Spring MVC is associated with a request mapping, and that's the URL, and the type of request that we're making to our application. Next, we have view resolvers. View resolvers are used to locate the view and to serve it back up. Now, a view resolver can be used to find a template or a JSP page or HTML page, or serve up a RESTful service. Then we'll talk about our servlet-config. Now this used to be a web.xml file coupled with a servlet-config.xml file, and now we're going to talk about how all of this is now configured using Java configuration. And then there is, of course, POJOs, plain old Java objects. Everything inside of Spring, all Spring beans are just simple POJOs, which is just a basic class that follows the bean specification of a no-arguments constructor with getters and setters named appropriately, and that is the definition of a Spring bean as well; it's just a Spring configured POJO. So when you talk about a Spring bean, it is just a POJO that's been configured and implemented inside of Spring. Let's look at the application that we're going to be building throughout this course now.

What Are We Going to Build?

We're going to build a conference application that has a registration object that we do some validation on. We're going to show you how to build the controllers associated with those and what that code looks like. It's where it's handling all the errors for it and navigation. We're going to solve complex problems using the Post/Redirect/Get pattern and just show you basic internationalization throughout the application. So things will be internationalized where it'll work in either English or Spanish. And you can add other languages to that resource bundle. We'll also dive into how to handle the dependencies that are in there and show you how to start your application using Spring Boot and move out of that into a standard Java configuration using the Boot starter and then overriding things through customizations inside of your app. It really goes into all of the details without getting too cumbersome in your project about how and what we should do and what Spring will do for us by convention and what we want to override through configuration.

Summary

To quickly recap what we learned in this module, we looked at what Spring can do for us, and that is your front-end technologies, RESTful services. Even if you're building a spy application, Spring MVC has got you covered. We went through the model-view-controller design pattern and talked about how Spring is really a good, clean, pure implementation of that as far as a web project is concerned. We also talked about that there's not going to be a focus of XML in

this course. The new standard is to have everything configured in Java. It's great for existing applications to have XML, but the industry has kind of chosen a Java-first, XML-second approach, and that's our focus in this course. And we picked apart the complex request response lifecycle and showed you how these pieces all tie into that model-view-controller design pattern. We briefly covered some vocabulary, and then we showed you a quick glimpse of what we're building. Now, let's go ahead and dive into creating our first application with Spring.

Creating Your First Spring MVC Application

Introduction

It may or may not be your first Spring MVC application, but in this module, we're going to cover how to build an application. There is more than one approach to using Spring, but I often see people pick one and never understand the benefit of the other approach. I'm speaking specifically about using Spring standalone or using Spring Boot to initialize your app. We're actually going to cover both in this module, but before we do, let's look at the prerequisites for getting started regardless of which approach we use. I'm going to use Java 11 in this course, and we picked Java 11 because it's the current release with specific LTS or long-term support. Java 12 has been released and 13 is on the cusp of being released, but those versions don't have long-term support for Java. I'm assuming that you already know how to install Java so we won't be covering that in this course, and I'm also going to use Maven in this course as well. Spring no longer offers up a way to download just the jars and forces you to use Maven or Gradle to download those dependencies. If you are a little bit confused or concerned about the use of Maven, there is a Pluralsight course on Maven. If you have more questions about that, you can research and follow it. The next thing you'll need is an IDE. This course has now been updated to use IntelliJ. We got feedback that the majority of our users wanted to see and use examples with IntelliJ, so we've changed the course to use it. And finally, Tomcat. A change from the previous release of this course though is that we're going to show you both of the approaches of deploying as a WAR and as a self-contained app with its own packaged server, basically, Java 11, the latest version of Maven, and IntelliJ A, and we will show both with and without Tomcat deployments. Speaking of IntelliJ, I'm going to be using the ultimate edition. The ultimate edition is actually a paid version, but there are a couple ways to go about this. First, it is free for students. Second, there is a trial with it. And third, a lot of students I speak with actually have it paid for by their company. If all else fails, all the examples we are going through will work with Spring STS. If you need help configuring them with a different IDE, reach out on the forum, and we'll try to help you there.

Getting Spring

One of the reasons for re-releasing this course was because Spring quit offering a direct download for the compiled jars. Sure, there are ways to obtain it, but nothing is simple as just clicking a link on their project page. With some of my older courses, people had asked for it just to focus on one technology and not use supporting tools such as Maven. Well, like it or

not, Spring wants you to download their tools using Maven. It will be simple for this course, though, and candidly, any project of moderate complexity should be using Maven or Gradle to manage its dependencies. The Maven repo has the source, Javadocs, and binaries all available for download. One of the main reasons Spring wants you to use Maven is because of the transitive dependencies that are required to run projects. We're going to start off by setting up our app to use Spring Boot and then add in the features that Spring Boot doesn't offer out of the basic configuration. Let's get our app configured and use Maven to download those defenses that our app will require now.

Demo: Project Setup

To get started with setting up our project, I've gone ahead and opened up a browser and gone to start.spring.io, and it pulls up this page. We're going to start off by first selecting the type of project. We want a Maven project, and the language we're going to choose is Java. And the current release of Spring Boot is 2.2.4, which I have chosen here. And then I'm going to change the project metadata to the group of com.pluralsight. And the demo name, I'm going to change that artifact to now be conference, since we're building a conference app, as we saw in that earlier demo in the previous module. I'm going to name the artifact conference, and this has some other things that will change in the pom for us by doing so. I'm going to select to drop down those options. You'll see the name again. Should be the same as the artifact. We can leave the description the same, or if you want to put something more significant there, I'm fine with that. The reason, though, that mainly I chose to expand the options is to make sure that the packaging type is currently Jar. I'm going to show you what happens when we change that to a War later. But I want to specifically choose Java 11. So you'll notice that they have versions 13, 11, and 8 in here. We want to choose 11, and as I mentioned in the prerequisites section, that's because it's the current version that has long-term support. Now, before we get too anxious and skip out of here, we want to go down here to this Search dependencies to add and choose Web, or search for web, and click the plus sign. And when you've done this, you should see it expand over to the right saying Spring Web, Build Web, including RESTful applications using Spring MVC. And it includes the bundled or embedded Tomcat container, which is what we want to start with as well. Now that we have this in here, let's go ahead and click the Generate, and it will download that for us, and we can now go over and expand that in our file space. When it downloads that, it'll actually name the ZIP file the same name as what we had called our artifact in the start.spring.io initializer. I'm going to double-click on this and just expand it. And now we can open up IntelliJ and have it import that project in. When we first open IntelliJ, we're presented with this splash screen asking us if we want to create our project, import it, just open it, or get it from Version Control. Since we used start.spring.io, we want to import our project because it was already created for us. So let's select Import Project. You want to choose the home directory of that application. I store mine underneath dev/workspace/spring-mvc/conference, so I'm going to choose conference. And you'll notice there's the src folder here and the pom.xml and some Maven command stuff. Go ahead and choose Open. And then it will ask us if we want to finish importing that from Eclipse, Flash Builder, Gradle, or Maven. We want to choose Maven and click Finish. And it will go ahead and load up the application. Now, The first time you do this, it's going to resolve some dependencies and spend a minute scanning all those files, indexing them, bringing them all into your workspace. Behind the scenes it's actually going out and grabbing everything from the Maven repo. And you'll see the little progress indicator

in the lower-right portion of your screen is it goes ahead and does this for you. It's downloading a bunch of stuff. And if you haven't used Maven before, it may take it a minute. So now we should have a project structure in here. And inside of our project structure, we should have our src/main/java app, and underneath that it should contain our package structure and our actual conference application. We can open it up and see that it's just a basic configured Spring Boot application. And then outside of here, we also have our pom file and we have our resources directory. Right now there's really nothing contained inside of here. If we go back to this conference application and right-click on main, we can run this application. It will bring up our build window for us and tell us that it's building all of our code and everything else, and then it will fire up the server. And we're up and running now. So now our actual app is deployed and running on 8080. Let's pull up our browser and see what it looks like the first time we hit it. The first time that you pull up your app in a web page, you're going to get this nice, ugly Whitelabel Error Page, basically making you think you've done something wrong. And really what it is, we don't have an index.html page or anything else configured in our application yet. So the basic project that gets downloaded and the structure that gets supplied for your app has no greeting page. Let's switch back over to our IDE and fix that now so we have at least something nice to greet us as we go into the app. We need to do a couple of things to add that index page to our application. First, let's go over to our src/main/resources/static directory, right-click on it, and say New, HTML File. And we can select HTML5 file. It should already be selected for us. We'll just go ahead and give it a name of index.html. Click Enter. This will give us a standard. page. I'm going to change the title to Index. And inside the body, I'm just going to add an H1 so that we have a nice big greeting or hello for us to land on here. So we can save that. Now the other thing that we need to do is, we actually need to restart our server, and this seems a little odd. Where things were all self-contained in a JAR, that file hasn't been repackaged up and moved over to the server yet. Once we have deployed files and have a reference to them, we can change them without always redeploying them. But if it's never been deployed to that server, it won't work. So I'm going to go ahead and click that redeploy button and have it restart our application for us and switch back over to our browser. Now let's see what this page looks like when we refresh it. Hey, we have our Hello, just like we expected it to. So a quick recap of what we did there. We went out to start.spring.io and downloaded a bundled-up Spring Boot application for us, expanded that to our file server. Then we went ahead and imported that into our IDE, deployed it, launched the application, seeing that it had an error page for us, went back in under src/main/resources/static, added our index.html, and then came back to reverify that it displayed like we thought it should. Pretty simple, but there are a few steps to do, and if it's new to you, it may seem a little overwhelming. And we're going to break that down through the next few examples.

Demo: Tomcat Configuration

Our previous example displayed static pages as a fully self-contained Spring Boot application, and that works very well, and it works fine if you want to just go ahead and use RESTful services too. But deploying JSP pages, which a lot of you may be watching this course to migrate and existing application or maintain an existing application and have to use JSP pages, they actually still work great, but Spring Boot self-contained JARs don't do well with them. In fact, Spring will caution you in their documentation to not deploy your application as a fully self-contained JAR and as a Spring Boot application. So let's go ahead now and look at

converting that to a packaged WAR and deploying that on a Tomcat server. To get an instance of Tomcat for us to use for our project, I'm going to go ahead and open up a browser and navigate to tomcat.apache.org, and then we can go down to the Download section on the left here. And if you're not sure which version you want to use, there is a link that you can navigate to and read about all the different versions. But for Java 11 and what we're doing, we want to use Tomcat 9. So I'm going to click on that, and scroll down a little bit, and click the Core zip version, and it will download for me. And let's expand that ZIP file. I like to do that in my DevTools directory, and here's my downloaded ZIP file. As I mentioned, I like to have all of my tools under a DevTools directory where I've got multiple versions of Maven, and Tomcat, and those types of things, any of the things I'm messing around with on projects, I like to put underneath here. You can see I expanded our apache-tomcat-9.0.30 version here. And we'll want to remember where this path is because we have to configure this inside of IntelliJ. We have to tell it the home path to this Apache Tomcat instance. So remember where that path is. Let's open up our IDE and get that all configured. To begin with, we want to stop that server if yours is still running. So our self-contained server, I'm going to go ahead and click Stop on it. Then I want to open up our settings and preferences, so I want to go to File and Preferences. I'm on a Mac, so mine's underneath the IntelliJ application name. Yours may be under Windows Preferences if you're on a Windows machine. I want to go to Build, Execution, Deployment, Application Servers, and you'll notice that we don't have anything configured here yet. So I'm going to click the plus sign and say that I want to choose a Tomcat server, and this is where it's asking for that home directory like we spoke about earlier. So I want to navigate to that folder structure, which mine was under DevTools, and I had Apache Tomcat 9.0, and click Open. So you see where I had mine into that Tools directory, apache-tomcat-9.0.30, and it also put that as the base directory, which is correct. That's what we wanted that to have. Those two should mirror each other. There are cases where they don't, not in this course though. For what we're doing, this is how we want it to look. I'm going to choose OK, and it has all of that environment set up, and select OK again. Now, we need to change our project to be packaged as a WAR. So if we open up our pom.xml file, right now, there isn't a package type associated in here. So there's a lot of stuff going on in this Maven POM filer. There's the parent, there's the group, and artifact, and version ID, and our name of our project, our Java version, and then dependencies that were added inside of here, but we don't see any packaging type inheritance, and it's because it's defaulting to what the parents says that we are. We want to go underneath version here, and you see it on line 13 in my application, and choose package, and the packaging type is a war now, and we'll save this. And you can see that Maven, I've purposely left it turned off on mine to show you, is asking if we want to import these changes, and I want to choose Yes, and you can enable it to auto import it as well. I want to have it import these changes, and now it's changed the project type of the structure of what we had. Before, it was packaged up as a self-executable JAR, and now it's a WAR, so it should be available to our Tomcat server to deploy. We need to make one other change to our POM file. I'm going to scroll down, and I've already copied and pasted in the code because I didn't want to have to have you watch me just type it all in. I'm going to add a dependency, and I've just commented mine out here. I'm going to uncomment it. Below the org.springframework.boot, artifactId of spring-boot-starter-web, and above the spring-boot-starter-test, you see on line 29 through 33, I've got this spring-boot-starter-tomcat plugin in here, and this is to remove the dependencies on the internal packaged application and convert this over to where we now have our own standalone Tomcat-enabled WAR, and so we have a WAR that will work for Tomcat with all of its dependencies all packaged up. Let's save this, and now we can go edit

our runtime configuration. Click on Add Configurations up in your toolbar up here, and you'll see we currently don't have a configuration for Tomcat. We added the server, but we haven't created a configuration, so let's click the plus sign, scroll down, and select Tomcat Server and local. If you don't see this, you probably skipped the step earlier where we added that server in when we downloaded it. So let's choose Local and open this up. To start with, I want to select the correct JRE. So make sure if you have multiple JREs installed, you choose the right one. This is the one that I want. For deployments, choose the Deployments tab and click the plus sign, choose artifact, and choose the conference:war. Click OK. I want to change the Application context to just /conference, and let's switch back over to our server, make sure everything looks good here, and it is. The URL is localhost:8080/conference, though it should be the same from our deployment in our server, and click OK. There's one other small change we need to make. I'm going to exit full screen mode here. Static resources aren't served from src, main, resources, static in standard web applications. They're usually out of a src, main, web app directory. So if I right-click and say New, Directory, and choose webapp, now, notice, I started that off of src, main, so src, main, and enter in the new directory web app, and hit Enter, I now have a directory to start hosting up static resources out of. And to begin with, I'm just going to copy and paste that index.html that we had under resources, static and paste that into of the webapp directory. Click OK. Now we have everything ready and configured to run our application. If your configuration says Tomcat 9.0.30, like mine does or whatever version you downloaded, go ahead and click the plus sign next to it, and it should start up your server for you. And you'll see the build status in the lower-right-hand corner, and then your server pop up. And once its loads, it's automatically configured to bring up your application and launch that Index page. So it did it faster than I could even walk you through it, and you'll notice our application pulled up, and it's now localhost:8080/conference. Before, we were just at root. Tomcat, by default, does it by the name of your WAR file. So our application is now localhost:8080/conference. There's ways to change that. You can go put the final build artifact name in there, but this is great. This gets us developing and gets us doing everything we want to do, and our app is now up. We did a bunch of stuff there. Let's walk through that real quick. We started off by opening up our POM file and changing the packaging type to war and then adding this dependency for the spring-boots-starter-tomcat. We also downloaded a Tomcat instance and added a configuration for it. We see our configurations here. I changed the URL to be /conference, and in the deployment, I changed the Application context to be conference. We need to make sure those two line up. Back on our Server tab, if you hate that it launches a browser every time it starts up, you can uncheck this Open browser After launch checkbox right here in the center of the application configuration section, and that can become a little bit annoying after a while. I like it because it gets me up and running and lets me see what I want to faster. But if you do hate that, you can go ahead and disable that there. Then we went ahead and created a src, main, webapp directory, and copied over that index.html you can see on the left-hand pane there, and launched our application, which pulled up our index.html page. So quite a few steps, Tomcat downloaded and expanded, installed, configured the runtime configuration, and then switched it over to a WAR and added a plugin for it, not that bad when you break down the steps. It just maybe seems like a lot if you're not used to all of that.

Spring MVC Configuration

A major change that was just beginning in the previous release of this course was the migration from XML in your web application. It started first by just not having XML in your Spring code, and then other standards pushed for it as well. More recent serverless specification have called for not having a web.xml at all. XML is often thought to be more complicated and, at a minimum, is more error-prone for copy/paste errors. Not using XML though can fill a little like a blackbox. And what I mean by that is you can start using it, but not understand what's going on behind the scenes. This often leaves developers confused about what's going on in their application and how it's working and, furthermore, left wondering what to do when something breaks. The configuration of Spring MVC though, regardless of whether we're using XML or not, is broken down into four basic areas. First, we have a POM file to download any of our dependencies, and that's what we created using the start.spring.io starter initializer. We then have a configuration section, whether that is XML or Java configuration or using web.XML. There's going to be a configuration section next, so don't worry about that. The next section is our Java files, our controller, possibly our configuration, our model tier. That's all going to be done in Java code. And the last section is our view. It doesn't matter if we are doing REST services, Java Server Pages, FreeMarker templates, we're actually going to look at Thymeleaf in this course. As a side note, you can use all of these in your application at the same time, but we're going to focus on them one at a time and see how this helps us navigate through our app. Let's look and see how this was configured in our application now.

Demo: Greeting Controller

As I mentioned earlier, we had our four basic categories inside of our application. We've done a bunch with our POM file already to begin with, and we really don't need to add any more dependencies in there right now, so we've got one of the four out of the way. The next one we'll go to is the view, and our views are usually served up from underneath our src main webapp directory in a traditional JSP application. So I'm going to right-click on our webapp directory and say, New, Directory, WEB-INF, hit Enter. And underneath that, I want to create another directory and call this jsp, Enter, and I want to create a new file underneath here and call it greeting.jsp. And there are some wizards and stuff you can use to create this. This is a simple page. I'm actually going to open up our index.html page, grab the body of this all out of here, and paste this into my greeting.jsp page. I'm going to change the index to greeting, and we're going to do something a little different now. I'll close that index.html just to make sure that we don't make a mistake there. I'm going to go to the h1 tag that we have up there and change that to ${message}, and save that. We're going to pull this out of a request that we're going to make to a controller. So now we have our view taken care of. We've created a JSP page that's going to go pull these attributes back out. Let's go ahead and change the configuration of our application. It's now a Spring Boot application. Let's make it so it launches our DispatcherServlet how we're normally used to in a Spring MVC application. If I go to my ConferenceApplication class, notice on line 8, it is our configuration class. We're going to continue to take advantage of that, but we're going to make this extends SpringBootServletInitializer. And what this will do is launch our application with the desired configurations we're used to using inside of a web app. So it's going to set up an internal view resource resolver. We're going to talk about all of that stuff later. It's going to resolve JSP

pages, set up JSTL for us, basically launch stuff the way we want to. We'll break all of these pieces down. We've just got to get something out there for you to use so that we can talk about it. So let's save this. There's nothing else to do here, and then open up our application.properties file, and that's underneath src main resources. If you're using the ultimate version of IntelliJ or Spring STS, you actually get some context-sensitive help inside of here. So I'm going to type spring.mvc.view.prefix. And we want to put in that directory we just created, WEB-INF/jsp/, and don't forget that trailing slash. And then we want to do spring.mvc.view.suffix, and we want to have .jsp, and also don't forget the .jsp on that. So save this. Now this is the other part of our configuration that tells it where to one, look for JSP pages, and two, what the extension for those are. If we were using other templated tools or when we get into the Thymeleaf section, this will be a different configuration. So now that this is all set up, we can move to the Java portion of our application or where we start doing things inside of our Java code. Let's go up to our src main java directory and click on our com.pluralsight.conference package, right-click and say New, Java Class. And the way IntelliJ works, we want to type in controller., so it's going to add a new package for us. I'm going to call this the GreetingController, and hit Enter. It will pull this class up for us. Now we're going to go through all of these pieces and explain what all of this does, but for now, we want to type @Controller and have it select that and import that class for us, and we can put a method signature down here that does what we want to do inside of our applications. We're going to start off by saying @GetMapping. And when we request the URL of greeting, we want it to go through and call this method. So we'll say public String greeting. This method name could be anything. I just chose greeting to keep it consistent, but it could be foo, it could be bar, it could be greeting, it could be say hello. It doesn't matter what the method name is. The GetMapping on line 9 matters. The method name does not. So, now we could pass in some parameters here, and we want to pass in a map, and that map we want to be a String Object pair, and we're going to give it the parameter name of model. Now inside of here, this is how we can pass attributes back through our model. So we'll say model.put. And for our key, because it's just a simple map, we'll say message. Remember, that was the variable name we had in our JSP page. And for the object, we're going to put a string, and I'm going to put in Hello Bryan. Feel free to replace that with your own name. And from here, we're going to return the string greeting. Now, what this does is when we ask for the URL/greeting. It's going to call this method, put this message into the model, and then that return line on line 14's going to go out and look for a JSP page named greeting. Those are the two variables we put in our application.properties. If this finds everything and executes correctly, it will then return that JSP page with these values rendered for us, so those four files we just worked with. Let's save this. I'll make this full screen so you can see everything. It actually fit in that one window. But this is everything that we needed to create our controller and return that value for us. Now that we have this in place, let's go ahead and start up our server. When this launches, it will take us to our index page that we have created. And then we can type in our greeting URL. So now we can go here and type in greeting and hit Enter. It'll say Hello Bryan. So we got all those pieces in place. To reiterate our packaging and the way we talked about our four areas, we had our Java code, which contained our controllers, our greeting controller, some configuration in our properties, our actual webapp configuration, which we're going to expand on all of this as we build out the course, and our JSP pages that contained our view. Pretty easy when we broke it all down. Now that we've gone through all of that configuration work, it literally just allows us to put in little pieces of our application and build it out as we go.

Summary

This was a bigger module. We went through all of the prerequisites that we needed to build your application. We got Spring and configured it in our application. We went through all of those configuration pieces in both Spring Boot, and talked about WARs versus contained JARs with Spring Boot, and showed you how to do the Spring MVC configuration as a standalone, as well as configuring Tomcat and showing how to set up the build and run configurations in your environment. So a lot of stuff covered. All of these concepts, if you don't really understand them yet, that's not a problem. We're going to go through and build upon all of those as we go throughout this course. In fact, the next section we're going to dive deeper into JSPs and adding more functionality into your application to help solidify these points.

Understanding the Structure of Spring MVC Applications

Introduction

Understanding the architecture of Spring MVC will actually help you write applications faster and troubleshoot problems more effectively. We're not going to deep dive into an architecture discussion, but let's cover a basic understanding so you know how to use Spring MVC more effectively. This is a short module that leads us through the structure of the rest of the modules. Software architecture has been around long before since Spring and Java. Architecture and design patterns are important, though, because it gives us a common vocabulary that we can talk about the features of a framework. And if you have some background and software development, you can know what a framework or an application is doing without knowing exactly how it's doing it just by the terminology that we use. We can discuss parts of an application based around the architecture or design patterns that it's using, and everybody can be on the same page without even having developed on that application before.

Model-View-Controller

Here is a standard MVC or model view controller design pattern that you may or may not have seen before. It's a pretty common pattern. In fact, I would say that most developers have at least seen it or at least heard of it. A request comes through the view based off of some user event and then is interpreted by the controller. The controller can change the model or not. And then it will select the view based off of our action. From here, the model can update the view with the database off of the user's actions. This graphic is sometimes confusing though because it's stemmed from rich client applications that were more event based or using a design pattern called observer observable or a subscriber listener type design pattern. Not really applicable to a lot of the software development we do with the web. This pattern in concept is still very sound, but we usually don't have our model updating our view because it's gone to a page at that point, and it's disconnected from our back end. This graphic though is a more accurate description of the kind of life cycle that we may be faced with in web development. The graphic is realistically more like the life cycle that we see in our web application. Our view can access our model, but it's usually done so through a controller. We'll look more at lightweight approaches in the Ajax module later in this course.

The summary is we usually make a request, even if it's a lightweight request, through our controller to access our model. We don't have our view typically going directly against our database or model.

Application Layers

You'll often hear people talk about applications in terms of a tiered architecture, or an n-tier architecture, or layers. We try to build our applications in tiers more now in enterprise languages, and there are great reasons to do so. A few of these are separation of concerns. This just means that each layer is only concerned with the task that it's assigned to. So our presentation layer is only going to have presentation type things in it, and our business logic is going to be contained in a layer, and data access is going to also be contained in its own separate layer. Now there's another term called reusable layers, and that sounds very similar to separation of concerns, and in some ways it is, but its focus is different. If I have business logic in my presentation tier, I can't easily change my presentation tier or expose a web service, for example, without recreating some of that logic. If I have extracted it into its appropriate tier, then I'm not duplicating that logic. So if you ever notice yourself copying and pasting code in your application, it's probably not in the correct tier or it's not architected in the correct tier or way. Now, that's a little bit of a difference between the separation of concerns and the reusable layer. Separation of concerns is not about reuse. It's just about having things in the right layer, so I can re-architect them later. Where reusable layers means that I have a purpose and a point that I'm trying to drive. I can expose the same data in multiple ways without having to worry about it, or I can replace my presentation tier without having to re-architect my application. Another term is maintenance, or refactoring, that might be a better way to put it. The ability to change things without having those things ripple through all of our code. So if I have Hibernate in my application, and I have to change something in my UI based off the change I made in Hibernate, no, I'm not talking about a business need, but more of a I want to change how a table is structured or something like that, I'm not separating those into the right tiers correctly. If this is done correctly, we can change our code and not have to retest everything, but rather just the pieces that we changed. This leads into a much larger discussion about unit testing, but that's outside the focus of this course. Talking about those layers of our Spring MVC application, and this is important because of how we annotate our components, which we're going to talk about here in just a second, we first have a data model that we would be accessing using something like Hibernate, or JPA, Spring JDBC. This layer represents the data or model of our application. Next we have a controller that interprets the user's requests and selects the appropriate view based off of what we've requested or what information we got back from our data model. And then we have our view. For our class, we're going to be using the JSPs, and we'll also look at Thymeleaf, as well as exposing some things through web services. Now, one thing that's often misunderstood by people is that Spring MVC has nothing to do with regards to our database, but it has a model associated with it. So is the model talking about our database or the model talking about what our framework is going to represent to just our JSP page? It's actually the latter. It's referring to what we're going to represent in our JSP page, or our Thymeleaf page, or our web service. We always have to get that information from somewhere. So that's where our various components come in and what we're going to talk about next.

Spring Components

We've spoken about tiers, and we've spoken about layers, and we've started to allude to components inside of our application. So how do we represent these tiers with Spring and Spring MVC? That's with these three components. We have our controller, our service, and our repository. We've discussed controllers already a little bit. They just route where we're going to and interpret the user's request. The service is where our business logic goes. It should also be noted that it's where our transactions will most likely start as well, if we're accessing more than one database table. And then there is repositories. The repository tier is also sometimes referred to as the DAO, or data access object, and they usually have a one-to-one mapping with our database table. So looking at controllers, as we've mentioned, they handle our incoming requests in building the response. I can't emphasize strong enough that business logic should not be handled here. This is also where our request and our response object should stop as well. We shouldn't hand those off to separate tiers. It should grab information from the request and the response and hand that over to the business logic. This works with our service and repository tier for the business logic and data gathering, and it's annotated using the @Controller annotation. There are some convenience classes that you extend, but you either have to wire them up or annotate them with the controller still. It also should be noted that this is where we handle exceptions and route views accordingly, based off of whether or not you had an exception or if we've got the correct information. The service tier is annotated with the @Service annotation, and it describes the verbs, or actions, of our system. It's where our business logic should reside. In fact, it should all be contained here. It shouldn't bleed over into our repository tier. Another role of the service tier is to ensure that the business object is in a valid state. This is where all of our state management should be handled, confirming that we've got a valid object passed in from a request. It meets the standards of our business objects or our business requirements. Also, this is where our transactions should begin. If you're doing two-phase commits or there's a chance we might have to roll back or access web services, those types of things, this is where we want those transactions to begin. It often has the same methods as a repository, but a different focus. We may have a method in here that says, find user by last name, and we may have that same method in our repository, but what we do if we don't find the user or how many people will return or what state we might return those objects in is controlled by the service tier, where the repository tier is just going to go get that data. And lastly, the repository tier. The repository tier is annotated with @Repository, and it also describes the nouns of our systems. Where the service here described the verbs, the repository tier is focused on the nouns. You can see where the focus is different from the service versus the repository. The service tier describes those actions that we want to do, and the repository describes the data that we're going to interact with. It's focused on persisting and interacting with the database or basic CRUD functions. It's also typically a one-to-one mapping with an object. You may have an address and have an address repository. You would have a customer object and a customer repository. It's also often a one-to-one mapping with the database table, but that's not always the case based off your design. You make break things into multiple tables like a person and an employee table, but you may only have an employee object inside of your application. That's really more up to the ORM tool that you're using.

Summary

Just to quickly recap what we covered in this module, we talked about software architecture and how it's a much needed thing in our industry to convey information and vocabulary about what we're trying to do. We talked about the model-view-controller design pattern and how the traditional model-view-controller design pattern isn't necessarily very reflective of what we're doing inside of a web application. We also talked about N-tier applications and layers and applications and why it's beneficial for us, and that leads us into a deeper discussion about the various components of our app and how we interact with those components, specifically using our controller, service, and repository objects. Those three pieces are what really builds up our access to our back end and where our business logic should reside. And even though there are more Spring proper than Spring MVC, we should always use those within Spring MVC to help our model-view-controller design pattern, which Spring MVC participates in.

Creating Controllers in Spring MVC

Introduction

We've already created one basic controller, but let's dive into some more of the complex functionality of controllers. Controllers are really the heart and soul of Spring MVC. They're the gateway or proxy into everything else that happens in our Spring MVC application. It's really crucial to have a great understanding of exactly what controllers are, what they're not, and how we can use them in order to build a successful Spring MVC application. So what is a controller? Spring MVC is very similar to other MVC frameworks in the sense that there is a separation of duties. This is different from older approaches where we may have just had logic and JSP pages and passed information from one page to another page or a framework of just servlets and building the UI out of string buffers, writing content at the output stream. In Spring MVC, the controller is the central concept or part of the framework. This may not make much sense to you with how you've written applications historically because we used to not think of controllers in the sense of verbs or actions. With the rise of RESTful applications, though, and RESTful services, it makes more sense to begin to think of controllers in the sense of verbs or what I can do within an application. You really need to think of controllers as choosing what to do based off of a user's actions or requests, and then the view or the actual web page is just a result of doing some action. Let's look a little more at the architecture of what makes up a controller.

What Is a Controller?

We've already seen some of this before, and, in fact, this image was the same image we used in our architecture discussion in the previous module, but I grayed out the other tiers to just focus on what is a controller. And we've even created a controller when we wrote our greeting controller in our first step of our application. But here's what's happening as we're going through a request from the user. The request comes in, and that's going to go to a particular controller based off of what the user's request was, and it's going to interact with some business logic. Now that business logic is going to produce some data, and it can

theoretically be a web service or it could be our database or it could just be our business logic tier. That's going to produce some output. We can think of that as our model or our UI, and we'll eventually return that model back to our UI on our response. As we come out of our web service or request, we're going to hand that back to our controller, and then our controller is going to go in and decide which view is appropriate based off of what information was returned or what happened out of our request, and we're going to hand that back to the user. The controller can almost be thought of as a traffic cop, hence the logo defining a traffic light. The responsibilities of a controller are to interpret user input and transport that input into a model. To take that information that's going to come back from our business logic from our service tier and build that into a model to hand back to our UI. It is the gateway to our business logic, and will determine the correct views based off of that logic. One other key point that never really seems to come up to play as far as the duties of a controller is, is that it also interprets exceptions for the business logic and service tier, and how to handle and navigate those correctly from there. Now, it could be a business logic exception in the sense that you haven't given me a valid date, or it could be, hey, the database is down. I'm going to send you this error page rather than requesting that you add this information or correct this information on the screen. So its duties are to also interpret those errors, and handle and navigate from our middle tier.

@Controller

Controllers in Spring MVC are very lightweight. They don't even actually require you to implement an interface. So if you remember, our controller definition from our Hello World action that we did earlier, our greeting controller was very basic, no interface, no class that we had to extend, we really only needed two parts. We needed a Controller annotation that told Spring MVC that this is a controller and it should be included in the available controllers for it to route information to, and that we needed to have a request mapping associated with it. And the request mapping, which in our case, is a GetMapping, tells Spring which method is going to handle which request. Now before we move on, though, I want to mention a few things. Spring does have some older concrete classes that you can extend, and your URL will map the class based off the controller name. It's a little bit of an older approach, in fact, it's kind of a more odd approach now, but it works just fine, and you may run into it out there. More and more people, though, are choosing the controller request-based mapping approach as they've worked with more and more RESTful services. It doesn't make a lot of sense to have two different ways inside your application. Since I'm going to show you some of the things with RESTful services later, I chose to just show you this approach in this example. There are other ways of doing it, though, like I mentioned.

Demo: Registration Controller

Let's start adding some of our conference registration functionality to our application. We've gone through and looked at controllers, looked at all the pieces. Let's just start adding some functionality to our app. I have closed all the open files that I had and shut down my server. I would advise you to do the same thing so that we're in the same space and same state if you don't have any problems with that. I'm going to start by first opening up our webapp folder, and we created this index.html file underneath webapp. We don't need that anymore

because we have our application configured correctly and hosting up Spring MVC files so it will utilize that src, main, resources one that we kept a copy of. I'm going to go ahead and delete this and click OK. And then what I want to do is I'm going to open up our jsp folder, and I'm going to right-click and create a new JSP file. And I'm going to call this registration.jsp. And it will create the standard template that we've already been using. I'm going to change the title to say Registration. And for now, I'm going to do the same thing we did before and add in an h1 just to make it so we don't have a blank page when we land here. And I'll just say Registration here as well. Now I want to go to that static HTML page we have for our index underneath src, main, resources, index.html. I'm going to open that page up. I told you when I set up our environment that I like it to go ahead and keep pulling that index page up. And there's a little trick as you're getting started with Spring development and IntelliJ and all these tools that I like to do. I like to leave this basic index.html page or a development page in here. You can keep one of these in your application, and it doesn't need to be named index.html. You can point your run configuration to a different page. But I want to go in here, and I'm going to add href. And I want to point this to greeting, and I want to close that off, and I'm just going to say Greeting here. Make sure you don't take the default of the /greeting because that will take you to the root of the application, and it won't function correctly. So I'm going to save that. I'm going to add another one here and do the same thing. I'm going to say a href. I'm going to do registration, and I want to close that element off and save that as well. Now I've got these two anchor tags in here, and what happens is that it launches our application. I'm just going to be able to click on these links really easily to navigate to different pieces of our application. Let's launch our app now and see what it does. I'm going to go down to our Run console and click the start for the run configuration, or you can do it from up above in our Tomcat configuration. And when this gets done building, it's going to pull up our web page and navigate us to our index page. Now you'll see we have a Greeting and a Registration link here. When I click on the Greeting link, it'll navigate me to the greeting page. It's exactly what we expected it to do. If I back up and go to the Registration, it's going to get an error. We don't have that mapping set up. It's because we haven't created a controller to point to that. Let's do that now. Switching back over to our IDE, let's go ahead and open up our src, main, java controller package. And I want to right-click on that and say New, Java Class. I'm going to call this RegistrationController and hit Enter on that. And we're going to go through the exact same operations we did before. We're going to annotate this as a Controller. We want to add an annotation for @GetMapping. We want to tie this mapping to registration. And then we want to add a method in here of public String getRegistration. And I specifically named it that because we're doing a Get on the registration URL. Honestly, it's just my naming convention. There's nothing significant to it, nothing Spring related. It's just my own personal preference. And we're going to pass in the Map<String, Object for our model to pass attributes back and forth. And inside of here, we can now just return the string registration, and that is going to do an internal lookup to our registration.jsp page. Let's go ahead and restart our server, let that pull up again, and check to make sure that all of those pieces are wired up how we think they should be. Our app will now display the registration page when we've linked to it by clicking on that registration page, and that's what we expect to see. It's now serving up our page through our controller. But hold on. How is it doing all of this? That's great that we've added a controller in here and we've added some JSP pages and they've somehow wired themselves together. What's really going on inside of here? If we open up our GreetingController that we created and our RegistrationController, they look the same. There's nothing telling it where to route here or what we're doing, any of that type of stuff. And there's nothing inside of this JSP page that's telling us anything significant. Well, it

all starts with our ConferenceApplication that is annotated as a @SpringBootApplication, and it extends SpringBootServletInitializer. This tells the application server to go ahead and create a dispatcher servlet and start serving up things. And as part of that annotation of @SpringBootApplication, it goes and starts looking for our controller and things with our annotation such as @Controller and @GetMapping, and our GreetingController that has the same thing, @Controller and @GetMapping. It starts navigating through all of that stuff. So this underneath the scenes here is creating a dispatcher servlet. If you've gone through this course before or you're looking at older JSPs that have been written or maintained through a Spring MVC application, it's always centered around a dispatcher servlet. If we look at our run configuration and you go scroll through this, you'll see inside of here it is calling our dispatcher servlet. It tells us that it's initializing that dispatcher servlet. Well, we didn't create that. Spring did for us by extending that SpringBootServletInitializer. It got rid of a lot of the boilerplate code that we have done in the past by just configuring that for us. Now if we open up our project, how do we interact with that though? It's our application.properties. This is where we did our InternalResourceViewResolver. We told it to go look under this directory and look for .jsp pages. So when we return that string from any of our controllers, our RegistrationController, for example, it is handling every web request through our dispatcher servlet. That's why the GetMapping for registration ties to it the way that it does. And when we tell it to return that registration page, it goes and looks for that registration.jsp page. Let's dissect this a little bit further in the next few examples.

@ModelAttribute

To be able to pass data from our request, whether that's a RESTful service or from a JSP page to our controller, we're going to use some libraries provided to us by Spring MVC. We could actually just use the standard HTML input tags, but Spring provides us a library making it easier to interact with our controllers. Historically, we've always just passed HTTP parameters that have gone on an HTTP request, and those parameters can then be accessed much similar to like a hash map where I just grab a parameter based off of its name so every HTML element would be named. Well, there's tags that Spring provides us to take these values from our input page and make them available to our controller through the model hash map, bind them to a specified object, which is actually the approach we're going to do, or just be able to grab them off the request. So we could always pass in the servlet request into our method name and access it that way. I'll actually walk through the example showing you all of those. The model can actually be a bit confusing, as you'll notice in our controller box that our model for accessing parameters and returning data is actually the same model and view object. We'll walk through that, though, so it's not quite as confusing. The binding object attribute approach that I just spoke about is used whenever we want to send data to our controller or retrieve data from our controller that's bound to an object or represented by an object. The nice thing about Spring MVC is that it's all done with basic POJOs and not using a class that's only specific to our UI like in some of the other frameworks such as Struts or WebWork or Stripes. So we'll use the @ModelAttribute when we want to do an HTTP Get to get back data. So say I'm going to get a drop-down like a list of states or, you know, a drop-down type list of data, gender, or those types of things. We want to grab that stuff that might be driven by our database. Well, we also use that same thing for Post. So it's the same object. It's bound the same way, and it's described the same way. We have a form that we're filling

out, and that's actually the example we're going to do here in our demo in a couple of minutes is the object that we send down through our Get is the same object we'll bind to on our Post. I mentioned it works with POJO, so it's very simple. And I don't have something that's specifically bound to just my UI tier. I can use it anywhere in my code. These objects can also be validated with a binding result. We're going to cover that in its own separate module later, but just to point it out so you're aware of it, you can have this data validated as part of your lifecycle.

Demo: Passing Parameters

This is where we left off our registration page. After just performing a GET to display the page. Let's now turn this into a form where we can use it to turn a POST back to our controller. Going back to our IDE, I'm going open up that registration.jsp page, and the first thing we need to do is add that spring tag library up at the top of it. And the syntax for that's a little bit odd. So we've got to do a %@, and it's going to ask us if we want to enter a tag library, which we do. And we need to give it a prefix. The prefix want to give this is form. And you can name this whatever you want. We can do a URI and choose springframework.org/tags/form. So now we've got our tag in here, and that's going to help us auto complete stuff inside of our body. I'm going to go below that h1 that we created and create a form using the form tag library that we just imported. And inside of here, I'm going to do a couple of things, and I'm not going to bore you with all the typing details. I'm going to throw in a table, yes, just a basic HTML table, and we'll do something fancy here with CSS and other stuff later. But inside of here, we're going to do a tr, and we're going to give this a td, and this is where it starts to get a little bit fun. I'm going to give this a name, and in the next td, I want to give this a form input of name. And what I mean by this is I'm going to input this form tag here, and the path of this is name. Now, you might be wondering why it's called a path, and it's because it refers to the binding object for this page. It's actually tied to a backing object that we're going to sync up with the model attribute. Don't worry, we'll show you here in a second what that means exactly. I'm going to create another tr. I going to do another td here. I'm going to give that a call span of 2. And then inside of that, I'm going to just do a regular HTML input, and this input we're going to do a type ="submit" and give that a value of Add Registration. Now, we can save this. And I'll be honest with you, the first time we run this, it's going to break. And I'm going to make this break on purpose because I want to show you the error that it displays. So when you run into this, because you will at some point, you'll know how to fix this. So I'm going to restart my server, have it pull up that page for us again and deploy everything, and then we'll have references to this new JSP page. Now, we can go to our Registration link, and when I click on it, it'll break, and it gives us this error. Now, one thing that Spring Boot does that I don't really necessarily care for, is it hides some of that stack trace. It's not an actual problem because I can see where the error is at. It tells us Name: 14, td: 15 and 16. It doesn't know what this form:input path="name" is on line 16, and it's telling you it doesn't know what to do with Name. It's because it doesn't have the field in a backing object to tie that to. Let's fix that now. Fixing that binding error is actually pretty easy. We only need to do a couple of things inside of our application. So I'm going to go ahead and click on com.pluralisight.conference right below src, main, java, not controller. I'm going to right-click and say New, Java Class, and inside of here, I want to type model.registration. And what this will do is create a new package and class at the same time, and you'll see that our controllers and our model on the left are separate. Inside of here,

I'm going to just do a private String name, and then below that, I want to right-click and generate our getters and setters, and click OK. And now we've created that registration object. I want to go to our RegistrationController now. Inside of here, we're going to get rid of that map that we have for that basic model object. I'm going to replace that with @ModelAttribute, and we name this ModelAttribute so we can reference it in our page, and I'm going to call this registration, and then we tie it to an actual object. So inside of here, I'll say Registration, and I'll call the instance of that registration as well, and now we have our object bound by our model to this ModelAttribute. One last thing we need to do now is go back over to our registration.jsp page, go up to our form, and we want to add the ModelAttribute here of registration and save that. Now, if you've looked at old code or you have old code that you're upgrading to a Spring 5 MVC implementation, this used to be called commandName. That will no longer work. It's been deprecated and removed. It will break, and you'll be left scratching your head, wondering why. It's now called ModelAttribute, which I like because the name is now consistent. It used to be inconsistent. So on line 9 here, this ModelAttribute that we just added to our form tag ties to our RegistrationController on line 14 where I say ModelAttribute registration. Those objects are bound together now when we display our page. I'm going to save that, pull up our server, and redeploy our server. And now when I click on our Registration link, it should pull our page up, and it does. I will tell you, we're not done yet though, and that's probably not what you wanted to hear. If I enter in a a name here, so I just put Bryan inside of here and click Add Registration, it's going to break because we haven't supported POSTs yet. We did a GET mapping, but we haven't done a POST mapping. Really easily, let's switch back over to our IDE. On line 13 in our RegistrationController, you can see that we're only supporting GET mappings with this method. There is a way to make it to where it will do both a GET and a POST in one method. It's honestly not a good to use case though. I'm going to go ahead and copy that, and paste it down to a new line, and do POST, and then I want to change the method name to addRegistration, save that, make sure you've imported everything. And now, we should be ready to bring that back up to our page. I'm going to go ahead and add a System.out.println here, just so we could see the value that's in registration as it comes through. I'm going to say registration, and then add just a getter on that registration for the Name, and just dump that out to the console. So I'll save that. Let's restart our server, have it pull that up, and now when we click on that Registration link, it'll display that page. We can enter our value in here, I'm going to put in Bryan, click Add Registration, and it will direct us back to the page because we don't have anywhere else for it to go right now. But if I go back over to our IDE, we can see in the logs at the bottom of here that it printed out Registration: Bryan Hansen. So it went full cycle. It went through our GET mapping, displayed the page, went through our POST, printed out what values we had there, we could do this in a debug mode and toggle for a breakpoint there to stop us and see what that value is, but this was quick and dirty and got the application up and running to where you could see those values and the full lifecycle of our GET and our POST inside of that object. We also just created a simple registration that we bound to, and then we created those forms inside of our JSP pages that went out and bound it to that object. We also have validation in there. We haven't done anything with that yet. That's in a separate module. But everything is set up. It's full featured. We've got our dispatcher servlet, our GETs, our POSTs, our controllers, our model objects that are backing our forms. We have all the pieces of our application set up for us now.

Summary

We went through a lot of stuff in this module. We talked about the duties of controllers and the various annotations associated with them, specifically the @Controller annotation, the @GetMapping, @ModelAttribute, and the @PostMapping so that we could do the full lifecycle of our controller. One thing we haven't covered yet, though, is what's really going on with how we find those JSP pages inside of our application. We went ahead and made some modifications to our application.properties, but it really didn't help you in understanding how we resolve those views. We're going to dive into that in the next module and hopefully clear up some of that mystery that goes on with how we find those pages inside of our application.

Creating Views in Spring MVC Applications

Introduction

We've already started creating some views in our application, but how we resolve them and how they're intricately integrated into our application is a little bit more of a mystery still. We're going to break that down a little further in this module, and also discuss how RESTful service types are a type of view, but not necessarily one that has a UI component. Views are what we see from Spring MVC. They are the V in the MVC. In our application, we're going to look at a few view technologies. JSP pages are becoming dated, but they still work great. We're also going to look at templating tools such as Thymeleaf. We could use JSF for a view technology, as Spring MVC has an API to replace the model portion of JSF, but a lot of people don't use JSF anymore in this manner, and it actually makes it so that it integrates with Spring well, but it's just not very useful. Not many people are developing apps using JSF that way. Most new apps have moved to a single-page application, and Spring MVC is a great way to expose rough services to be consumed with your SPA as well.

Demo: View Configuration

Although not required, there's a convention of placing view pages under the WEB-INF directory so that they cannot be deep-linked or bookmarked to. This way all requests must be directed through our application, and we can guarantee the user experience. We already have this set up in our application. Let's look at what that configuration did for us, though, and how it was tied to our application as far as deployments were concerned. When we converted our application from a self-contained executable JAR file to a WAR file that we deployed on Tomcat, we created a src main webapp directory, and you can see that in our project structure over here on the left. Underneath that, we created a WEB-INF directory, a JSP directory, and that JSP directory contains the JSP files that we have created so far. Now, why would we do this? As I mentioned, it's to make it so that people can't deep link to our application. And if I switch over to my browser, we have our page that we've set up links to, but if you remember when we added these links, they were a reference to the service call. So we were doing a call to the controller to route us back to the page that we wanted. What happens if we come over here and we type in registration.jsp. It won't serve that up, and we don't want it to serve that up. That's by design. So putting those pages underneath our WEB-INF folder and a JSP folder makes it so that they have to be served up internally and turned back to our end user, thus

making it to where we control the state of how they flow through the application, and they can't bookmark into things that we don't want them to. It's a lot better designed for us. All we had to do was just move that under that WEB-INF folder.

Demo: View Resolver

In our first demo, we configured an InternalResourceViewResolver, and didn't even realize it. We've used that in all of our controllers since then. We simply return a string, and that is what's used by the ViewResolver to find the JSP page named and in the location specified with our resolver. We can also return a view object that wraps a string that is the view name as well. Our controller can build a model if necessary and return that to our ViewResolver with data that is needed in the view. We can also have multiple ViewResolvers, as well as multiple view types in our application. Let's look closer at that ViewResolver we've been using and not really realizing that we had created it. As we saw earlier in our controllers, I'm going to open up our RegistrationController, we're just returning a string of a name. On line 22 here we have return registration, and that's referring to our registration JSP. You may or may not recall that we configured two lines inside of our application.properties, the spring.mvc.view.prefix and the spring.mvc.view.suffix, and this tells it where those files are located and what extension to look for. So really it's taking that name that we have in our RegistrationController and pre-appending the prefix and appending the suffix to that file to serve it back to our UI, so that's how it knows where to find this. Now how did it know how to do that? Well, in our ConferenceApplication configuration, we have this @SpringBootApplication, and we extend that SpringBootServletInitializer, and this configures it for us. But let's show you how to override that. Let's create a new class, and to do so we're going to put it alongside the ConferenceApplication. We're going to right-click and say New Java Class. We're going to call this ConferenceConfig. You can honestly name it anything you want, it doesn't matter. And we're going to define a couple of things inside of here. First of all, we're going to designate this as a Configuration class, so we're going to annotate that with the @Configuration annotation. Then inside of here we want to create a Bean. We're going to create @Bean, and this Bean we're going to make a public ViewResolver, and this will just return an instance of viewResolver. It can be named anything we want. Inside of here, we're going to create an InternalResourceViewResolver, and this bean is nothing more than what's already being defined for us in that application.properties. So I'm going to show you the same code side by side here before we comment it out. I want to say bean.setPrefix, and we want to add in here WEB-INF/jsp/, and then we want bean.setSuffix. We want to add quotes inside of there and say .jsp as well. Then we want to set the order, and this just tells it if we have multiple ViewResolvers, which one we want to fire in which order, and whichever one is first successful is what it will return. And then we want to return the bean at the end. Save this. This is all of the configuration that's done for us behind the scenes with that ConferenceApplication annotation being SpringBootApplication. So this main class that would launch our application if we had it self-contained, it still gets read off the class path, and the SpringBootApplication has a component scanner in there that tells it to go look for any other class labeled as a configuration, and inside those configurations load up the beans. Now, these two lines, line 14 and 15, are the exact same lines inside of our application.properties. So I can now comment out line 1 and line 2, and save that, and you'll see that line 14 and 15 will take place of that. I'm going to restart my server, and now we can test that our ViewResolver is doing what it should be. I'm going to click on that

registration link, and it did, it served those pages up for us. So looking back at our editor again, line 14 and 15, and really this whole method of 12 through 17, sets up the ViewResolver that's already preconfigured for us. And now we can override things and do various changes to this that aren't really exposed through those properties like we have in the application.properties. We started using that to begin with, we can convert it over to this. We are going to add other ViewResolvers inside that config as we do more things to our code. So we'll leave this here, and it's a good base for what we want to move forward with.

Resolving a View

Views are resolved in Spring MVC by the controller building a model that it passes to a view resolver, which then will determine the correct view that we want to display and choose the appropriate view based off of that request. And there are various view resolvers that are provided by Spring. You can create your own custom view resolver, since they just extend the ViewResolver interface. Some of these resolvers are used for templating tools like Thymeleaf or FreeMarker. It just configures where it should be looking for those templates and then executes the bindings and writes the output. The ResourceBundleViewResolver is used for internationalization purposes, and it's used for things like templated layouts. In a later example in this course, we're going to look at XML content negotiating and how to respond with JSON responses versus HTML or JSP responses, and we'll walk through one of those examples.

Resolve Static Files

There are actually a lot of reasons why we would want to host up static files from with inside of our Spring MVC application. We may want to implement security on them, we want to make sure they're logged in, we could possibly do some caching stuff. Regardless, it's really easy for us to do. It's very similar to what we just did with the view resolver, except we're going to add a resource handler. We do have to implement one interface, and we'll walk through that, but then we just add it to the registry of the Resource Handler Registry, and we say that, hey, any request to files, I want you to go ahead and look in this directory, and we've created a WEB-INF/pdf/ directory, and we can host up static files from there that are PDFs. Let's walk through this configuration in our code now. To host up static PDFs, I'm going to start by creating a PDF directory alongside of our JSP directory. I'm going to right-click on WEB-INF and say New Directory, pdf, hit Enter. Then I have a sample PDF I just grabbed off of my file system, and I encourage you to do the same, and just drag that PDF into your application. It's going to ask you if you want to copy that in, and we'll say yes. Then we want to implement an interface. Now we have this configuration class set up, but we're going to implement this interface and use it to override some functionality. So I'm going to say implements WebMvcConfigure, and then I want to add in here a resource handler. So I'm going to add in the addResourceHandler's method that overrides the implementing basic functionality that's currently there, and say registry.addResourceHandler. I'm going to put in a path of /files/**. So it's going to look for anything in that path, and then I'm going to specify the location of where we just created that directory structure, /WEB-INF/pdf/. So that resource location needs to match exactly what you named your folder underneath WEB-INF. If you added an s on the end or something like that, it's a common mistake I've seen people make. It needs to be

identical. Let's save that, restart our application server, and see if that will host up our PDF for us. Now that we are at our browser, we can go up to our address bar and type in files/ps.pdf, because you'll remember in that resource handler, that was the path that we wanted it to resolve to, and hit Enter, and it will serve up our PDF. And it works just like we thought it did. Let's look at that configuration in that configure again one more time. You can see on line 15 where we told it to go to /file/**, and on line 16 where we told it to locate those files under /WEB-INF/pdf/. That's all it took for us to resolve static files hosted from inside of our application.

Summary

In this module, we looked at some of the deeper configuration of Spring MVC and how it resolved views and did various things inside of our application. We looked specifically at the views and view resolvers, and how if you followed that convention of just adding those two lines in the application.properties, it could find files for you. We then rode our own custom view resolver to show you how it was implementing that functionality, and then we showed you how to create your own handler to serve up static resources. Specifically, we did an example with PDFs inside of our application, which wasn't super complex, but did show you the power of having your own custom path that went to a different place on your file system. This way you wouldn't have to expose files off of your file system, but rather could host them up through your app and even wrap security around them if you wanted to. Let's now dive into some of the deeper configuration things that we can do with JSP pages in the next module.

Using Java Server Pages with Spring MVC View

Introduction

We have already been creating JSPs throughout our application, but there are a few features and techniques that we haven't done yet that are very useful while developing applications using JSP pages. We're going to cover interceptors and internationalizing our application, talk about the other interceptors that are out there that are common, and then we will discuss a common architectural problem and how to solve that using the Post/Redirect/Get pattern.

I18N and Interceptors

If you weren't aware, I18N stands for internationalization. It literally means there's 18 characters before the beginning I and the trailing N of the word internationalization. To this point in our course, we haven't been internationalizing our pages to accept other languages. Everything has just been hardcoded. This same code internationalized that we have here, where we've got firstName and lastName and the Save Changes text just hardcoded in English would look like this. Notice that we have removed any of those references to just plain English language in here and are now grabbing them from a resource bundle. FirstName, lastName, and the Save Changes text are all pulled from that bundle. Even

notice that the case with the Submit button is changing the value of that actual button and not just the label on the page. To do this, though, we need to introduce a new concept, and that is interceptors. Interceptors are used to allow us to intercept calls to our server and perform special tasks on a call. Common uses for interceptors are logging. We can change the log level on an application without having to redeploy it. Not useful to end users but very useful for debugging a specific problem in an environment without having to restart the application. Another common use is security. When we want to intercept each call or direct them to a page, make them log in and then redirect them back to that page that they were trying to access, it's a very common use of an interceptor. Security is outside the scope of this course, but there is a separate course that is focusing on the use of Spring Security in your applications here on Pluralsight. As we mentioned earlier, I18N internationalization of sites and catching their request to change locales is a very common use of interceptors. And, lastly, performance monitoring. Similar to logging, this is not something we're going to display to the end users, but we may want to intercept each call to perform some metric gathering on a request. Let's add internationalization support to our application now.

Demo: Adding Internationalization

To begin adding internationalization to our application, we want to start by opening up our ConferenceConfig, and we're actually going to add three beans inside of here. I'll make this full screen so it's a little easier for you to see. The first thing we're going to add is a SessionLocaleResolver, and this will just use the LocaleResolver interface, but the SessionLocaleResolver's what ties our current session to a locale. And this bean is just going to take and create an instance of the LocaleResolver and put it out in our Spring registry for us. I've gone ahead and pasted this in. You can see we start off with a default locale of Locale.US. You can change that to your specified region if you want, but this is what I'm going to start with. The next bean that we're going to add in here is the LocaleChangeInterceptor, and the LocaleChangeInterceptor just looks for a parameter, either through a hidden element or on our URL string as a query parameter, however we want to pass that in, but it looks for that to see if it should intercept that change. And you can see on line 35 here that we are looking for the param name of Lang, L-a-n-g. Now the next thing we need to add actually isn't a bean. It's going to be a feature that we override from the WebMvcConfigure, and it's the interceptor we're going to add. So we just created this LocaleChangeInterceptor. Now we're going to register it, and I like to keep all the things that we've overridden at the top where we're going to be making those changes to that WebMvcConfigure and the beans defined down below. Really, it's just personal preference. It doesn't matter. Anywhere you put this in this file will be fine. Those are the three things we had to add to the ConferenceConfig. Now we want to go ahead and create two properties files, and I'm going to do that by right-clicking on our src, main, resources directory and say New, File. And for the name, I'm going to type in messages.properties, and the name matters here. So notice its messages, plural. I'm going to hit Enter. And the first thing I'm going to put inside of here is a comment. I'm going to do labels, and this is just for personal preference. And then give it to the name-value pair of Name. And then I'm going to throw another comment in here for buttons, and the name-value pair of save.changes=Save Changes, and save it. Now we want to add another properties file out here. I'm going to right-click on our src, main, resources directory again and say New, File, and the name of this one's going to be messages_es.properties. And this is specifically for the Spanish

translations. So in here, I'm going to do the same thing, except I'm going to cheat. I'm going to just grab this text that we just typed out, copy it, and paste it inside of here. I'm going to change it to the Spanish translation of Nombre. And we'll change Save Changes, cambios. There we go. And we have our two properties files, and the locale that we pass in with that change interceptor is going to go look for the appropriate properties file and pull that translation out of that file. Now let's open up our registration.jsp page, and this is pretty simple, but we have to add two pieces inside of here. First, we want to start by adding another tag library at the top of our page. We'll say %@ for the tag library, and we're going to give this a prefix of spring. And for the URI, we want to choose the http://www.springframework.org/tags and save that. And now we can come down to this text for Name, and we'll change that to be the spring:message tag. And the code="name". And then make sure you close that tag off. So we can save that, and you can go through and change all of the fields as we go. I'm going to start my server up just to show you that this is working the way that we expect it to work. And now it's launched our browser for us. I can click on the registration link, and to begin with, it shows up with Name in English. But if I had a query parameter on the end of here and say language=es for Spanish and hit Enter, notice that it will translate Name to Nombre, and it will also ask us if it wants us to translate this page for us automatically. So you can see that we are now using that internationalization. And you can go through and change the title of the page, the buttons to say Submit changes or Save changes or to add registration, whatever translations you want to go through and add. You can see we have all the pieces now set up to do internationalization inside of our app.

Post-Redirect-Get

The Post/Redirect/Get pattern, or PRG, is a technique used to help eliminate form resubmission. The user's POST to the controller does whatever intended action was requested, but before returning the view back to the user, it does an internal redirect and issues a GET back to itself to display the page to the user. This makes sure that all the variables of state have been cleared, so a back button then resubmitted is disabled or stopped. Let's implement this in our app now. We've already done all of the work that we need to for us to implement the Post/Redirect/Get pattern because we've set up our GetMapping and our PostMapping separate. A lot of examples out there show these being combined. All we actually needed to do is go to line 22 now and put in redirect: and save that, and this will tell the ViewResolver to go ahead and do a complete redirect and a GET back to registration, and it will clear that form out for us. Now, yes, you can still back up, but it will be a new, completely separate POST. It won't be reposting the same information, if that makes sense. So let's save this, start or restart our server depending on what state you left yours in. And now we can go to our registration link, and inside of here, if I put in my information and click Add Registration, notice that it has submitted it and returned us back here and cleared it out. So we've done the whole GET, and then a Post/Redirect/Get back to ourselves, and it's cleared this form out. And that just helps us have that information in a safer state to not resubmit or overwrite the existing page that's there.

Summary

Although this is a shorter module, we went over a couple of advanced techniques. We went over interceptors, and we implemented internationalization inside of our application using some resource bundles, specifically a Message.Properties properties file and the according file for the Spanish translation. And then we went ahead and implemented a post-redirect-get pattern, a PRG, inside our controller using the redirect preappend to the page we want of view. Both of these are great solutions to complex problems that you'll often face inside of your application, and they don't just work for JSPs. You could actually implement these same things if you're using Thymeleaf or any other view technology integrated with Spring MVC.

Using Thymeleaf in Spring MVC Views

Introduction

Not everybody wants to use JSP pages and would rather have a lightweight view framework instead. If you aren't a serious JavaScript developer, but still want to use a lightweight HTML5 for an end, you can use a templating framework. Thymeleaf has some of the best integration with Spring of any of the templating frameworks out there. Thymeleaf was somewhat adopted as the lightweight HTML5 framework by a Spring development community and just sort of fit a gap that existed there. It is quite easy to use and only requires minor setup inside of our application. Although there are other templating tools that work with Spring, Thymeleaf remains to be one of the easier full-featured ones out there to implement. And if you are using Spring Boot and self-contained jars standalone, it's actually the version that they recommend in the templating tool that they encourage you to have inside of your application over anything else. I will say before we go any further, though, I would make a backup of your source directory because we are going to break the JSP functionality in this demo. I'm going to reverse it at the end, but if you want to save yourself a headache, just copy your source directory and your POM file, and when we're done with this, write back over the top of it, and you'll be good to go. You could also download the exercise files and look at these examples from there as well if you don't want to change your current project.

Demo: Thymeleaf Configuration

To add Thymeleaf to our project, it's actually quite easy. We just need to add another Maven dependency. So we're going to build a dependency element. We're going to add in a groupId of org.thymeleaf, an artifactId of thymeleaf-spring5 because we are using spring5, and then the version of 3.0.11.RELEASE, as that is the current release at the time of recording this course. Let's go ahead and add that into our POM now and verify that it's downloaded it to our project. Adding that to our POM is as simple as opening up the pom.xml file, and I'll make mine full-screen so you can see it a little bit better. I'm going to go underneath our spring-boot-starter-tomcat, but above the spring-boot-starter-test. And yes, order does matter in your dependencies. It will load those onto your class path in the order in which you have them in this file. So, I want to go ahead and add that in there, I've just got org.thymeleaf, thymeleaf-spring5, and version 3.0.11.RELEASE. I'm going to save that, and when I save

it, it'll automatically download that from the Maven repo. And I can verify that it's in my project by scrolling down to the external dependencies of my project structure, looking at the very bottom, and you can see that I've got org.thymeleaf:thymeleaf:3.0.11 and org.thymeleaf:thymeleaf-spring5:3.0.11.RELEASE, so I've got everything in there for the Spring versions that I'm using and the Thymeleaf dependencies that I need. I'm going to close this back over and get it ready for us to add the rest of the code in our project. Let's look at what it takes to add the resolvers in here now.

Template Resolver

A template resolver is not the same as a view resolver. When first looking at the template resolver, it actually feels just like a view resolver that we've seen quite a few times in this course now. But this configuration in here is actually where to locate the template files and how they should be resolved. The view resolver, which we will configure later, just sets the order and references this template resolver we'll use. Let's configure the template resolver in our project now and then we'll configure the view resolver in a minute. We need to configure a couple of beans for our application to resolve those template files. I'm going to do that inside of our ConferenceConfig. So as I open that up, I'm going to just scroll to the bottom and go below our ViewResolver, but before our closing curly brace, and I'm going to add in this templateResolver. To do so, I want to go ahead and import all of these resources that I have. And you can see we created an instance of a Spring templateResolver just named the method templateResolver. And inside of there, we have a SpringResourceTemplateResolver. We set the applicationContext. Notice that's still red. We're going to fix that. And then we set the prefix of WEB-INF/views and templateResolver suffix of .html. Now to fix that applicationContext, I'm going to scroll up to the top. Now we haven't done this yet in our application, but it's really common to use an applicationContext inside of our file. So I'm going to import an annotation for @Autowired, and I'm going to create an instance of the application context for it to be autowired into our application, and that's all I have to do. I am going to make that private just so that nothing else in the package utilizes it because we'll want autowire and inject that wherever we're going to use it. Now if we scroll back down, that is purple instead of red, and the colors may not show up super good for you, but we've resolved that dependency. So we have our templateResolver in here now. Now we can go configure a template engine that will utilize this resolver.

Template Engine

We didn't have to create an engine of any sort when we were using JSP Pages, so this is a little bit unique to Thymeleaf. We have to create a Spring templateEngine that will process the pages and substitute in the model values from Spring into our pages to be displayed. Notice that this code makes a call to that templateResolver method that we just created in the previous example. Let's go ahead and add this into our ConferenceConfig file. Now the templateEngine is just another bean and can actually go right below the templateResolver that we just created. I'm going to paste this code in here and import it as well, and it stands alone on its own. It doesn't have a bunch of other references other than that call to the templateResolver. So on line 72 you can see where it's calling the templateResolver that we just created on line 61. We have this templateEngine that is configured just for Spring, and it

also has enabled the Spring Expression Language compiler, the EL compiler, and that just makes it so we can use the shorthand syntax of accessing Spring variables and passing them in. That's it. That's all we have to add for the templateResolver. Now we can add the actual ViewResolver for those templates that the controller can navigate things to.

View Resolver

And finally, we have a ViewResolver. The ViewResolver is a little bit different because the TemplateResolver looked up the actual template. The ViewResolver just takes whichever template was loaded and returns that based off the name, so they kind of work in conjunction. I will tell you now, though, that the viewResolver.setOrder that you can see in the middle of our method here, that has to be before the JSP page in this example for it to work. So we're going to change the order of our JSP viewResolver to 1, and this to 0. If not, it'll look for JSP's name this way. It is possible to configure them to work together, but it's kind of outside the scope of this course, and specifically this example. Let's add our ViewResolver, and our template page, and any other configuration we need to to run the rest of this demo now. To add the viewResolver for the Thymeleaf pages, I'm going to go ahead and give myself a little bit of whitespace below the ViewResolver we have for JSP pages, and I'm going to paste that code in there. I've already got it written. The thymeleafResolver, as we mentioned earlier, is real simple because it calls the TemplateEngine. One thing I am going to change, though, is you'll notice how line 57 and line 65 in my code are both set to an order of 0, I want to change that setOrder on line 57 to 1. This will make it to where the thymeleafResolver pulls our page up first. And as I mentioned earlier, and I'm going to mention again, we are currently breaking the JSP functionality of our application to show you the thymeleafResolver. It's outside the scope of this course to have both Thymeleaf and JSPs working together in the same application, and it's really not a common occurrence. Most people don't do it that way. They usually choose one or the other. Now that we have that set up, I want to go out to our WEB-INF folder and create the views directory that's being called by our templateResolver. So I'm going to right-click on WEB-INF and say New, Directory, and I'm going to type this as views. Inside of here, I want to create a new HTML page. So I'm going to say New, HTML 5, and I'm going to name this thyme.html, and hit Enter. It comes up with a very basic HTML page for us. To make this a Thymeleaf page, I'm actually going to just replace the contents inside of here and talk through the two pieces that we have. We have a HTML namespace on line 2 that's really an xml namespace that HTML is utilizing of thymeleaf.org, and we can now use those elements that that namespace defines as we have on line 8 where we're calling the th:text with the message in there. So this will pull out a message that we haven't yet defined. You'll notice that it's currently read on line 8 because it doesn't know where that's coming from, and that's because we have nothing to route to this page yet. Let's go up to our controllers, and specifically our greeting controller, and open it up. And let's add in a request to actually load this page and display that element that it's stopping on telling us that there's nothing to find calling it. I'm going to actually copy that greeting that we already have and paste that down below here, and change this to work for the thyme page that we just created. Call it thyme, and we're going to return to that thyme.html page that we just made, and let's just change this to Hello Thymeleaf. Save that. Now we have all of the pieces together in our application to run this. Let's exit full screen, start up our server, and go see what this now looks like. Here's our browser page, just like we would expect it to show up. The difference is now we can come up here and type in thyme,

and it will go through and pull that page out for us and show us the Hello Thymeleaf that we just added into that message property. It works pretty well. I will tell you, though, that if we back up right now and hit our greeting link, it's going to break. We're going to get that 500 error. So I'm going to back out the changes that we made for Thymeleaf. I wanted to show you how to integrate it, and you can see it, you can use everything that we have already taught you with JSPs works with Thymeleaf pages, but the two just aren't configured to run alongside of one another. But you can see how Thymeleaf works, how it's really lightweight, it's really centered around HTML 5, and it doesn't have some of the nuances that JSP pages have in there.

Summary

In this module, we talked about using Thymeleaf and how it is a templating tool, and we integrated the dependencies for it inside of our application. Then we configured the template resolver and the template engine and the view resolver, and how that's a little bit different using templating versus just straight JSP pages, that the view resolver doesn't do much; the template resolver is what's doing a lot of the heavy lifting for us. As a reminder, I am going to back out these changes for Thymeleaf, because we did break our JSP pages. So set that bean over to a priority of 0. That should get you enough to get it running and not have to remove everything, but if you want to remove all the Thymeleaf stuff, you're more than welcome to, and it'll just clean up your project for you, like I'm going to do.

Validating Objects in Spring MVC Applications

Introduction

Validation is always an interesting subject in applications and doesn't ever seem to be really handled cleanly. When I first recorded this course, the spec for this was JSR 303. Since then, they've made subsequent JSR releases to work with newer JVMs. JSR 303 was the original specification for Bean Validation and, although it worked this past release, its target was really for JSE 5. It works quite well and it has since its initial release, but took a little time to gain adoption. In keeping up with the Java specification releases, JSR 349 targeted Java 7. It cleaned up ambiguities with Beans spec 1.0, and 349 is version 1.1 of the specification. JSR 380 was released as of Java 8 and focuses on the modularity changes that were introduced in Java 9 and later. It is version 2.0 of the Bean Validation spec, and there are a few major changes with 380. That is what we're going to focus on in this module, and you can now utilize type Lists and other things such as Java Optionals in the 2.0 spec. If that doesn't quite make sense, don't worry, we're going to walk through some examples and it should by the end of this module.

Validator Interface

There is another approach that we're not going to show, but I want to spend a minute talking to you about it, and that is the Validator interface. It was the approach that we used before

the JSR 303 and subsequent releases came out. It works fine, just as it did 10 years ago when it was released, but it's very programmatic. You have to manually type out every line of code that you want to have it execute. It is not deprecated, and you can in fact still use it in modern Spring applications today. I personally try to avoid it because business logic tends to creep into these validators. The separation of concerns that we spoke about in an earlier module becomes a little hazy in these Validator classes that implement the Validator interface. For this reason, I tend to avoid the Validator interface and implementing classes and just used the JSR. More advanced validation I move into my own custom class inside the service tier and handle it appropriately there. Let's move on to how we configure Bean validation

Demo: Validator Configuration

The bean validator reference implementation is actually an instance of hibernate-validator. To configure it for spring, we just need to add it to our class path and begin validating objects. It doesn't matter if you are using Hibernate the ORM tool, the validator is actually just the reference implementation and has nothing to do with databases. Let's add the hibernate-validator dependency to our project. We want to open up our POM file, and I'm going to put that below our Tomcat starter, but above our starter test, and I'm just going to add that code in there. We want to have a group ID of org.hibernate.validator, an artifact ID of hibernate-validator, and a version of 6.1.1.Final. When you first add this in here and save it, it is going to take it a minute to download. So if you get a red error for a minute, realize it just has to resolve that dependency and download it to your local repository. Once you've saved that, though, just as we've done with other dependencies, you should be able to look in your external libraries and look for org-hibernator, and specifically the hibernate-validator, and you can see that it's downloaded there. We've got it in our window pane on the left, and it's specifically version 6.1.1.Final. Let's go back and see how we start using this now.

Demo: Validation

Validation in our project requires us using the @Valid annotation before our model, essentially saying that we want this model object to be valid upon entering this method. In the method signature, you can see where we have annotated this using @Valid. The other thing to note though is the reference passed in of the BindingResult. It contains any errors and a flag to notify you that there was a verification error so we can direct that to the correct view. From here, we can direct back to the appropriate page, in this case the form which it would have been submitted from. Inside our form element, we can add a form:errors tag that will only show when an error is present. So we're going to change the controller signature in our controller, and we can add an errors element inside of our JSP page, all by adding specific attributes on our model object. Let's implement this in our current project and see how this works out for us. To begin adding validation, we first want to start off with which object we want validate. I'm going to open up our model, Registration class, and let's make it to where we don't enter in a name that's just empty. So I want to go above that member variable and say @NotEmpty. And this will guarantee that when we ask for validation on it, it doesn't allow an empty or blank or spaces. There's a lot of characteristics around it that you can look up, and there's different options we could use, but we'll just signify @NotEmpty for now. The

next thing we want to do is open up our RegistrationController, and there's a couple of changes we need to make inside of here, but they're still quite minor. As we mentioned on the previous slide, we want to add Valid to that method signature, saying that this model attribute we want to be valid. And since it is a registration object, that NotEmpty that we put in there automatically signifies that we want it to be a valid registration object when validated. The next thing we want to do is add a binding result to this. So we're going to say BindingResult and give that a name of result. And you can see in here that it just goes through the method signature passed by reference to grab any errors that we have in here and display them for us. Now if we have an error, we can easily check for it and change our navigation based off of it by saying if result.hasErrors, go here instead. So we can return back to the registration form without doing the post redirect get that we had set up down below. And just to show you this is working the way that we think it should, let's add a system.out.println in here just to show you that there are some errors that we're going to return from this. There were errors. And we'll save this. And this is actually all we have to do for our code. Now to display those errors, we want to add some functionality to our JSP page, but this will validate our object, and we can handle it from there. We've got all that functionality built into it now. Let's open up our JSP page and customize that page to show those errors back to the user. I want to start off by adding in a little bit of CSS. And yes, in your project, you'll probably have a CSS file that contains all this. But just for the sake of this example, we're going to add it at the top of this page. I want to go below title, but still within the head tag. I'm going to add this style element and inside of there two classes, an error and an errorblock. And then I want to go below the form tag that we created earlier, and we're going to do another form tag, but this time going to do form:errors, and the path on this and the CSS class is errorblock, and the element is div. We can close that off and save it. So we've added our error code, what it's going to look like, and then the block to actually display that to the page. Let's exit full screen now and start our server up, and now we can go to our registration link. When I click on it, it'll pull the page up. If I enter a name in here, it would go through cleanly. But without it, let's just see what we get. We get our error message back saying it must not be empty. Notice that this is the default message. It's all lowercase. It is my simple HTML just displaying this red error block. But we can customize this message. Let's make sure it still works if we enter in a name though. I'll add in Bryan in here and hit Add Registration, and it works. We've got our post redirect get still working, and we have our error message still working, so great. Let's customize that a little bit so that we've got a little bit more user friendly error message in there.

Custom Error Messages

Those last error messages were a little bit generic and didn't really fit our application very good. Let's customize that a little bit. Let's start off by going into our JSP page and adding a field-specific validation error. So I'm going to add another td in here and a form:errors path="name", and that's referring to the element or the variable inside of our registration object called name, and a CSS class of error, that's the CSS class we defined up above. I need to change this table to be three columns now because we have now added that third column on there. And let's save this. Now all this is going to do is just going to display the error message that we're returning. It's the same error message that we already have in our class Registration. Registration, if you remember on line 7 here, is just saying not empty. But how do we customize that error? Well, it's actually really cool. So I'm going to create another

comment in here and just say validations, and from here I'm going to take the type of validation that we're doing, which is a not empty on line 7. I'm going to say NotEmpty., and it goes to the actual instance name. So we are commonly referring to this as Registration. If you look at our controller, we're returning an instance of the registration object under the ModelAttribute registration. So we want to go back to our message.properties and say registration.name. So NotEmpty on the registration object on the member variable name=Name can not be empty, please fix. You can obviously name it whatever you want. And if you're going to do this in Spanish, you'll want to do the same thing in that message's properties as well. It'll work if we just stay in English. But just so you don't forget, I'm just going to change this to say in Spanish. You can do the translation later. That's all we had to do though. Our messages.properties, you do the validation that you're trying to achieve, the name of that instance, and the variable instance that you're trying to tie that to, so it does it all off of a name value-key pair, the actual string that we want to fix, and now we can run it. So let's restart our server here. Now when we go into our registration, we can click on it. If it's not there, now we have Name can not be empty, please fix. So it will show you the specific error, and we showed it at the whole page level and the individual field level. You can play with that and add validations to all the fields you have in there and do it in a programmatic fashion. It works really well, and this is all internationalized. So we've internationalized our application. We've internationalized our error messages. Everything is working, and the validation is really easy to do. So you can see how this is a really handy way for you to validate stuff. By the way, this bean validation just doesn't work for UI. You can use this anywhere in your application. I see a lot of people do programmatic things where they go try and apply this validation on their own. You can use the validation utils inside of that bean validator to run that across your object, so there's no reason for you to code it yourself. You can pass an object in and ask if it's valid and have that return. That's all Spring's doing behind the scenes. Works really well.

Summary

This module covered Bean Validation, which I think is one of the coolest, yet underutilized features inside of any of the UI tools I've seen. We looked at and learned the history of 303, 349, and 380, which it's just adapted and grown with the JVM releases. We talked about the validation interface, and it's still a viable option. I chose to not show it because I don't think you should be using it, but I want you to know that it's out there, and I want you to know why we addressed it. I don't think you should use it. It often gets too much business logic inside of those classes and those implementations that you do. We showed you how to add the dependency for the JSR, and we specifically used 380 and that it is an instance of Hibernate Validator, which often catches people off guard because they think they're dealing with the ORM, which they're not. It's just the Bean Validation implementation. Then we showed the configuration and actually showed a more advanced version of it where we customize the error messages. Let's go ahead and dive into some RESTful services using Spring MVC now.

Using Client-side JavaScript in Spring MVC Applications

Introduction

JavaScript frameworks like Angular, React, Vue, etc are changing faster than ever. Although a simple example with Angular would be nice, it would be out of date faster than we could publish this course. What has been very consistent though is jQuery. Before we dive into JavaScript though, let's look at the Spring configuration we need to serve up some RESTful services. Specifically, let's look at RestControllers. The RestController annotation is the heart of serving up RESTful services. Looking at it though, it looks very similar to other controllers that we've already created. In fact, we use the same annotation for the GetMapping that we'd do in a standard controller. In this particular example, we're going to call the User URL and get back either a default or pass in the name that we would like to retrieve. Obviously in a larger example, we would pass in a name and retrieve the user from a database. Let's add this RestController to our project now to retrieve a user.

Demo: @RestController

To begin adding that RestController, let's first start by giving us a model object to return off of our request. I want to go ahead and open up our src, main, java, com.pluralsight.conference package. And go to the model, and right-click on the model, and say New, Java Class. And inside of here, I want to just do a User Class. Let's hit Enter. Now we give it a couple of member variables. I'm going to do private String firstname, private String lastname, and a private int age. Now let's give ourselves some whitespace and right-click and say Generate. We want to do the Getters and Setters. Select all three and say OK. Let's save this. Now we've got a model object we can pass back and forth with our controller. Let's right-click on our controller and say New, Java Class. We'll call this UserController. I want to start off by saying @RestController. And what this does is it makes sure that every call that goes in and out of here looks at the content type and the except headers to see how and what it should return. And to save yourself the typing, you can go ahead and open up that RegistrationController and copy that if you want, or we can code it in by hand. I'm actually going to paste a Get in here and change a few things. First, I'm going to start by importing the classes that we need, and the User object specifically wants us to select that particular one, and our request parameters. Now we have all the pieces we need inside of here, and let's walk through what this example's doing. The return type of this method getUser is our user object type. Now, in the previous examples we were doing with our registration controller, you can see here on line 16, we were returning a string. Let me expand that code so you can see it. On line 17, we were returning the actual string registration, which referred to our GSP page with our InternalResourceViewResolver. Well, we don't do that with a RestController. We return the actual body to our caller. So we're going to have a request come in, and the response is going to be the actual body of the response. Now there's a couple other things in here. You'll notice that we have a @RequestParam that will pull those parameters off the URL as we make requests to it. And right now I just have one of them. We can add the other ones in here and say @RequestParam, lastname, and we can grab the age if we want as well. Now we can set those values in our actual objects. We'll say user.setLastname and user.setAge and save this. We have all the pieces we need for this to

now return that body back containing the user object that we've created. Now obviously, we usually wouldn't be just taking parameters that come in and storing them in an object and returning them. This is where we would have a service auto-injected into our business logic tier or our model that would go ahead and retrieve this out of a database. So lines 15, 17, 18, 19, all of this would be a database call to our model tier returning something from a repository. But for this example, it shows you how we're grabbing those request parameters and storing them in an object and returning them back out to you. Let's save this. Going to start my server up. And now, we don't have a page set up for this. We're just going to request that actual instance back. So we can say user and hit Enter, and you'll see we get JSON text back, and there are defaults that are already in there for us. But we can override that with query parameters, so we can say firstname is equal to Dan. And when we run that, it will return that as Dan. You can do others out there. We'll say lastname equals to Frandson. And we can override the age while we're at it. Just standard query parameters. We'll say age equals 33. Notice that it handled the strings appropriately, the integer appropriately. We are reading in those query params, storing it in the object, and returned it without us having to use any view resolvers. So it goes ahead and just by default accepts that JSON response back and returns it to you as a client.

Postman

Obviously, our browser worked fine to do a simple test to grab that URL and see that the values were changing, but I wanted to show you Postman. I've just downloaded their basic version for free. I don't have the paid version of it, but it's a great tool for testing web services, and you can see that it'll actually store history of it. You can create a test suite out of here if you'd like. But to run a simple test in here, we can go ahead and enter our URL, localhost:8080/conference/user, and this is of type GET. I can Send on this, and you'll see that it retrieves the firstname Bryan, lastname Hansen, age 43 in here. Works great. We can also do posts in here. So if we come in here and set a POST and create the Body, we can add in these key valued pairs inside of here saying that I want to pass in a key of firstname and a value of Bryan, a key of lastname and a value of Hansen, and a key of age and a value of 43, and it will handle all of the quotes, and strings, and manipulation, and converting it to integer objects for us automatically. If I call this right now, it's going to fail, though, and give us an error back because we haven't created this method yet. So let me hit Send on this, and you'll see we get a 405 error saying that the request method, POST, is not currently supported. Let's add that into our code now.

Demo: @PostMapping

Adding support for the POST was almost easier than doing the Git. We just need to come into our controller and below our Git mapping, but before our closing curly brace, you want to add a PostMapping, and inside of here we want to have it handled the same way with the /user request, and then we want to just start putting our signature in here. So we're going to say public, and it's going to take an object type of User, and we'll name it user. Instead of our method, we can return our user instance. We can manipulate things on there if we want, but for just some debugging, let's add a System.out.println here on User firstname, getFirstname and save this. Now, we have everything in here to actually handle a POST. Now, from the

previous module, you could go ahead and add validation in here and other stuff to handle that. We've already seen those pieces in action. We don't need to make this example more convoluted, but those same binding results and valid attributes will still work inside of RESTful services. Let's restart our server, and I'm going to go over the Postman. Now in Postman, with this POST set up where I've just chose the type of POST and clicked on the Body tab, added in my elements, my key of first name, value of last name, key of last name, value of Hansen, age, all of the name value key-pairs inside of here, if I click Send, you'll notice that it returned our object back to us, and that's because it successfully went through that. Let me switch back over to the IDE, and you'll be able to see that System.out.println in our console. You can see our user first name Bryan at the bottom of our console output because we went clear through that life cycle. That's really all we had to do to add a GET and a POST to our code. You can easily implement the other methods as well. A PUT and a DELETE, they have the same URL mappings, you can go ahead and put them in here, and it will accept the except headers that you send in. So if it's a PUT, POST, GET, or DELETE, it will handle it accordingly to that request. Makes it really easy. Let's see how we'd do this and now intercept this data with jQuery on the client-side.

Demo: jQuery Integration

As we mentioned before, JQuery has stayed pretty tried and true for accessing data, especially for some RESTful services. You can see in this example here we're going against a URL. It's the URL we actually just created for localhost:8080/conference/user. And then we can grab the body, the data, and grab the first name out of that and append that into our document last name and age and manipulate the data in our DOM with that. Let's go ahead and add some JQuery into our project now. It's pretty simple. We're going to add a little JQuery snippet and then add a user.html page that just uses JavaScript to interact with our RESTful service. To start let's add that user.js file inside of our static directory. So I'm going to right-click on src, main, resources, static here and say New, File, and I'll just do user.js and hit Enter. Then I'm going to right-click on static again and create another file, user.html. And these two files are just going to have some simple HTML and some simple JavaScript. In fact, it was the exact JavaScript we just showed on the previous slide, to access our URL. I'm going to paste in this JavaScript. I'll make it full screen so that you can see it. It's a little bit of coding. If you want, you can grab these out of the examples and just paste it in yourself. It's underneath the course, main directory, and there's an exercise files link in there that you can grab that information and just paste it in. And I'm going to do the same thing for the HTML. And there are a couple things to note in here. The script is using a CDN, or a Content Delivery Network, to pull in our JavaScript on line 5. And then we're referencing that script that we just created on line 6. And we have a couple of elements down here, firstname, lastname, and age, that are being referenced in our JavaScript file over here. So when we grab that data, we go ahead and manipulate those elements with that data from firstname, lastname, age on this request. So when the page loads, it calls this AJAX function when the page is ready, and then it takes the results of that and appends them into those elements in our HTML page. Let's start our server up. And we haven't created our helpful little link in here. We should, but all we have to do is go up to our URL and type in user.html and hit Enter. And you'll notice it pulled back Bryan, Hansen, and age 43. So it went out, did our request from a REST controller, and pulled that information back in. Really easy to do. You can see how valuable these RESTful services are, and we're not using anything like JSP pages in

this example or Thymeleaf. We're just going out and pulling these RESTful services into our app using simple JavaScript, just JQuery. I don't even have to have anything as advanced as Angular or React or Vue. It's a simple AJAX call. Very powerful and a great way to access data and have this inside of your just basic HTML.

Summary

This module was all about Ajax, Ajax calls, pulling back, RESTful services from our server. We did this implementing a RESTController. We showed you how to do the GET and the POST, and honestly, the PUT and the DELETE are identical. It's really no different, so there's not a lot of sense in going through those exercises. We then showed you a quick example using Postman and how powerful it can be to use that to not only GET but have canned examples of POSTs back to the server. It's a great way for you to test your web services and run through that pretty quickly. And then we did a final example using JQuery and showed you exactly how easy it was to use an Ajax call in your HTML to pull back that JSON data from your URL. And this is the end of this course. I want to thank you for staying with us this long and completing everything. I would recommend looking at some of the continuation courses after this. I have a Spring Data JPA with Hibernate course, Spring Security, I have other courses on RESTful services, and a great series on design patterns that, regardless of using Spring, if you're doing any type of Java development, I would recommend you look into those. Thank you, and I appreciate you sticking with us.