**Introduction**

**Framework Overview**

So which framework should we choose? Let's go over a little overview of kind of the current frameworks that are out there and available for us and compare and contrast them a little bit to what Spring MVC has to offer. **Struts** used to be the standard, **Struts 1**. It was eventually very outdated. And then **Struts 2** was a significant change all for the better than the original Struts but different enough that a lot of people has lost their loyalty for Struts, started looking for other frameworks to use. **Tapestry** was and still is a very decent framework but is governed by one person and someone at the WIM or whatever project he was working on. Tapestry was one of the first heavily object-oriented or POJO-based web frameworks. **Wicket** is a lightweight framework that sells itself on being easier to configure than some of the other frameworks that are out there. And then **GWT or Google Web Toolkit** is a very rich user experience framework with a steeper learning curve than a lot of the other frameworks that are out there. **JSF** from Oracle, which actually is the only true standard framework for the web out there is, J2EE based as well and it's a very rich user experience. It comes with libraries like ice faces and richfaces and other third-party components for integrating into JSF.

**Spring MVC -** So Spring MVC is a very unobtrusive framework. And notice how I don't say lightweight. It is lightweight in a sense that there's very little overhead in running it but it's a very capable, heavy duty framework. It pulls a lot of the best practices from all of these frameworks we just mentioned and combines it into Spring MVC. Plus, it just makes sense that Spring MVC integrates very nicely with Spring and all the other capabilities that Spring has available for us to use.

**What is Spring MVC?**

Spring MVC doesn't try to niche itself for one particular type of web application.

* It can be RESTful-based.
* It can be JSP-based.
* We can use other view technologies like FreeMarker or Velocity.
* You can use it for headless applications.
* You can also use it as a remoting framework.
* And all of these are capabilities that are built into Spring MVC.

**What really is it?**

It's a web framework built around the principles of Spring. So interface-driven design, it's very POJO-based, it can be unit tested very easily actually, and it's all very lightweight, very unobtrusive. It's also based on a Dispatcher Servlet of Front Controller Pattern. If you've ever looked at the J2EE blueprints or some of the other historical documents, it's not really historical, but some of the other kind of object-oriented design literature about design patterns. This is based off of a front controller pattern. It is MVC, hence, in the name, which stands for Model-View-Controller. Spring MVC, you can just take the little pieces that you want and just use those. It's also built from the shortcomings of Struts 1. Now, that wasn't their entire intention with Spring MVC but they saw what they didn't like in Struts and built around that in Spring MVC. There's support for things like themes, localization, RESTful services which is actually one of my favorite parts of Spring MVC is how easily you can dump RESTful data out of Spring MVC. And it's annotation-based configuration. You don't even have to use interfaces and their code if you don't want to. There's things that you can get by following their convention over their configuration. So if we just mark some things up with annotations, that works very nicely. And it, of course, has seamless integration with other Spring Beans, Services, those types of things. You know, the whole dependency injection module works very well with all of Spring especially Spring MVC.

**The History of Releases**

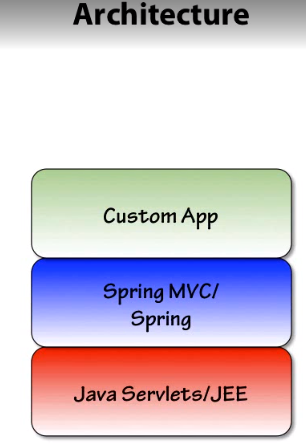
So let's take a minute and go through the history of Spring and Spring MVC as a company. In 2003, they released their first milestone of the Spring framework. This focus was simply just on interface-driven development and overcoming the complexities of J2EE development. In 2004, they did their 1. 0 release of the Spring framework and started incorporating things like Spring MVC at this time and focusing on other areas and how to integrate common tools like Hibernate and. NET into their framework. In 2006, they won the Jolt Productivity Award. This was version 1. 26. Things like SCG or Spring Security were also being integrated into the framework and becoming a more holistic approach to software development using Springs tools. In 2006, they released 2. 0. And even though this was a major milestone release, we went to 2. 0 from 1. 26. Up until this point, everything was backwards compatible. In fact, most things have always been backwards compatible than forwards compatible with Spring. In 2. 5m, there was heavy focus on annotation-driven development.



Now, this was a significant release for Spring MVC because now all of our configuration was done through annotations. Prior to this, development was very XML heavy. In fact, Spring got a little bit of luck for being such a-- you are now XML developer. If you were a Spring developer, you're an XML developer rather than being a Java developer. As of 2. 5, everything went very lightweight, very unobtrusive, all annotation-based, and this just carries on in 3. 0 and 3. 1 which is due to be released any day now. As of 3. 0, you can actually do your-- you can do configuration using Java. You didn't have to use XML. A lot of people still prefer the XML over Java just being a newer thing, that just makes sense though.

**Architecture**

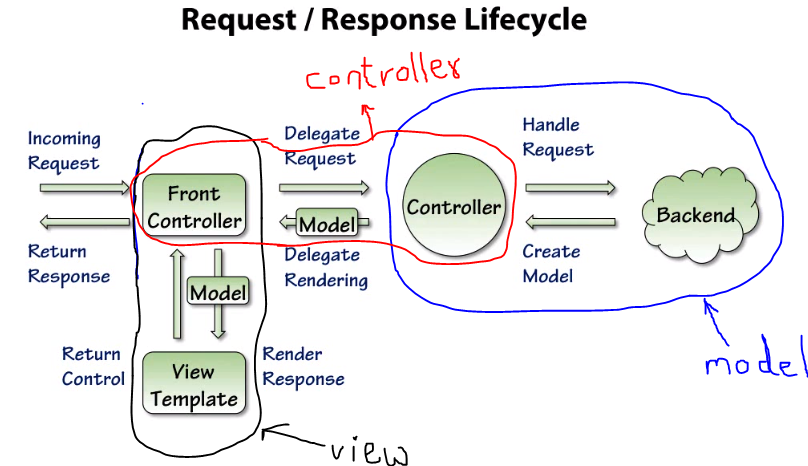
So I wanted to spend a second and just talk about how Spring MVC has built on top of the Java Servlet API and the standards that they're in J-- as I mentioned earlier, we are implemented using a Servlet Front Controller. So we have Java Servlets and the Servlet API that's the bottom of our stock or the basis of what we're building things on. Spring MVC is built using those technologies. So, really, when it boils right down to it, under the hood, Spring MVC and Struts and all the other frameworks we're looking at, really are just Servlets or anything but it's got design patterns and things wrapped around it to make it easier. We write our application on top of Spring MVC or using Spring MVC to build this entire stock. Now, this has a few things for us. One, we have a standard way of doing things. So if somebody else has worked on a Spring MVC app before, they can come in and use and help write our app because we're using Spring MVC and know that-- what's going on without necessarily knowing the business functions or business logic of our application.



So they can quickly come in and see what our app is doing and have an overall understanding of what we're trying to do inside of our application because we're not using custom frameworks. We're building our app on top of Spring MVC.

**Request/Response Lifecycle**

So the request/response life cycle is actually a tricky one when you look at it, the big scope of what's going on here. We have an incoming request that hits our Servlet Front Controller, just our Dispatcher Servlet inside of Spring. So that hands off the request, that delegates our request over to one of our controllers that we've set up. So, the dispatcher is the router that comes in and says, "What controller is going to handle it? I'm going to delegate this request over to this controller over here. " That controller just routes traffic to where it's supposed to go. So it says, "I'm going to handle that request and had it over at the back end. "



Now, our back end could be composed of web services or a database or multiple databases or any number of things that we're gathering data from. And in turn, what that does is it hands back a model to us. And 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 do the rendering. " Now, the rendering is separate from the business logic. So we have this Model-View-Controller design pattern. Everything is doing its own specified functionality here. So, coming back from that controller, we have a model now or data that we're going to try and represent the UI. So we go back to our Front Controller and it says, "Oh, well, who's going to handle this? Who's going to render this response? Well, let's pass our model down to some view template. " For this course, we're going to use JSP but we could be using FreeMarker or Velocity or any number of different technologies that integrate nicely with Spring MVC. Once it's gone down to our view template, our JSP page, we're going to return control back to our 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 our model gets created; our view, so a representing view; and our controller. So that is where our Model-View-Controller is coming into play with our entire application.

**Vocabulary**

So let's talk about some of the vocabulary that we have associated with Spring MVC just to clear some things up 'cause we're going to use these terms throughout the rest of this course. We have a **DispatcherServlet**. The **DispatcherServlet** is our entry or our configuration point for the application. It also happens to be our **Front Controller** as we talked about in our request/response life cycle. The next thing we have is our controller. Now, our **controllers** are simple POJOs. They-- it's a command pattern that handles a request and determines which view to route to and what model we should get and handles that information and hands it back and forth. Next thing we have is a **RequestMapping.** The **RequestMapping** is our URL and the request type that our method is tied to. So if we accept posts or we accept gets or puts or deletes, those types of HTML protocol requests, then the RequestMapping can say, "I only accept gets, I only accept posts. " We also have a **ViewResolver.** The **ViewResolver** is what we're going to use to locate our JSP pages or whatever view technology you may be using. For this course, we're going to be using JSP pages. We configure a ViewResolver and that's how **Spring MVC** knows when we ask for the hello page, where to actually find the hello page at. Then we have our **Servlet-config**. The **Servlet-config** is just a configuration file per **DispatcherServlet**. So this is where Spring knows to configure things and wire our application up. Don't worry, we're going to go into a lot more detail about what the Servlet-config is. And then two other terms, a **POJO**, if you haven't heard it before, it's just a Plain Old Java Object, generally has no arguments constructor and private member variables with-- exposed through getters and setters. And then a **bean** which-- a **bean** really isn't just unique to Spring but when I'm talking about a Spring bean, it's a Spring-configured POJO. So it's a POJO that we've instantiated through Spring, actually in our Servlet-config and it's wired up to be used inside of our framework.

**Building**

Introduction

we're going to talk a little bit about the prerequisites that you'll need in order to get started and get our environment setup so that we can start developing using the Spring MVC framework.

Prerequisites

There are a few prerequisites that you'll need to have in order to be successful in this course and follow along in this module. The first thing that you'll need is Java. I am using Java 7 in this class, but anything Java 5 or greater should work. I'm assuming that you already familiar with how to install Java so I won't be covering that. We're also going to use Maven to download our dependencies in this course. I am using Maven 3, it has a standard directory structure that we want to use inside of our App. The next thing that you'll need is an IDE. And finally, we're going to deploy our App to Tomcat. We will walk through the steps of how to deploy our application inside of our IDE and how to download Tomcat and configure it for our IDE.

**What are we going to build?**

So what are we going to build, you ask. We're going to be building an application that is an actual app that I wrote for an internal challenge at my company a few years back. The goal of this application will simply be to track the minutes that you have exercised in a day. We use this application to track an individual's minutes of exercise each day. We're going to help individuals reach their goals and report back on the number of minutes that they have exercised by notifying that their goal has been reached. In this module, we're going to be just getting the basics of our application running and not be implementing any of these features quite yet. We just want to make sure that we can actually show a web page that's working through Spring MVC, basically, a hello world to start with.

**Springsource Downloads**

Although, we're going to use Maven to get Spring MVC, I wanted to show you where you could download it from if you chose to not use Maven. So, the Spring Framework is bundled in kind of an interesting way. You can download it from springsource. org/spring-framework, and the problem is you can't just download the web resources. You have to download all of the Spring code, the compiled class as the jars. There's 21 different jars that it downloads. So when you download Spring, you're going to be getting these whole pile of all the pieces of Spring that they have enabled. And the reason they do that is just so that you know that these versions work well together. But you may not need any of the testing framework or the transaction API or I just-- I copied this little snippet of what the download jars look like just to show you, you know, some of the various stuff that gets downloaded with it, portlet API. You may not be doing anything with the portlet. So, it's a little confusing when you go to download Spring from their website not using Maven that it's going to download you 21 different jars, most of which you may not ever use.

Downloading Spring

So, let's look at where you can manually download the Spring Framework from. If you go to springsource. org which you can see, I just put in my URL bar here, my address bar, it'll pull up the Spring source web page. Now at the bottom and it's a little bit confusing because there-- it doesn't say anywhere on this page where you can download it, if you scroll to the bottom of the page here, you'll see Projects. And we're dealing with the Spring Framework. Now, notice there's Spring Web Flow over here and Spring Web Services, that's actually not the same thing. Web flow is a framework built on top of web MVC. So, we want to click on Spring Framework and then we want to click the download button here. And the text here just describes what the Spring Framework is and a little bit of history of it now. So, we want to click download and it's going to take us to a download page. Now, it may take you to a registration page before or that type of thing just to try and gather some information from you which is actually good to register for their page because it tells you of upcoming releases or updates and they really don't spam you that much. So, go ahead and go to the community downloads here, and the latest release that was actually just released couple of days ago is the 3. 2. 0 release. So I'm going to click on that, oops, I'm going to click on that and have it download. And I've actually already got it downloaded here to I'm going to go ahead and open up folder up. Now, we can see inside here I've got the spring-framework-3. 2. 0 release. If I open this up, you'll see there's the libs and schemas, the schemas for the XML definition. We'll talk more about that later, and then just the javadocs. I click in libs, you'll see all those jar files that I was talking about and the different releases and source code, javadocs, all that stuff for it. Now really I may be just want one or two things out of here like spring context or spring beans. You know, we know that we want the spring-web stuff down here, so it kind of just make a whole lot of sense and it doesn't really make it any easier because you don't know what you need for you application outside of this stuff. So, let's go ahead and look at how we do this with Maven now.

Maven Overview

So let's look at this Maven way. Using Maven we only need three dependencies. Those dependencies very simply are spring-webmvc, the servlet-api and jstl. We actually don't even need the sevlet-api or jstl because Tomcat's going to provide that for us. But we would like them in our code so that we can have out IDE compiled and get IntelliSense help for us and all those n ice things that our IDE will provide for us. So we're going to include those and I mark them as provided. The snippet of code that would do this and we're going to go through the complete example of this would look just like this. There's the dependency section, and inside there we have our dependency for spring webmvc and then we have our dependency for the Java servlet-api and jstl, and you'll notice that both those have a scope of provided on them.

Demo: Maven Configuration

So let's walk through setting up our project using spring STS and utilizing Maven to download our jars for us. First off, I'd like to download everything to my C dev directory, just a directory I created, and I separate my tools and my workspace out that way. So in my tools folder, I have spring STS expanded, and I'm just going to go ahead and open it up. Now it should prompt you for a workspace and I keep my workspace right alongside my dev directory, so I have-- or my tools directory so I have C:devworkspace. Go ahead and have it open that up. Now, we're going to utilize a plug-in that's already configured in spring STS called m2eclipse. If you're using straight Eclipse, you'll want to install this plug-in, it just makes things a lot easier. So I want to come up here and go to File, New, then I'm going to go down to other, all right. So you'll see there's a lot of default there that are available for me but I want to use the Maven plug-in. So if I go down here, you'll see there is a new Maven Project and I specifically want to choose Maven Project. I double click on that. I'm going to accept all the defaults on this screen, just use the default workspace location, and I click next. Now, there's a bunch of archetypes already created for me inside of a Maven, inside of the Maven repository. And an archetype is just a directory structure that follows the format that I want to use and does some default things for me. So to filter it down to find just the web projects, I'm going to type web in here, and you'll notice that there is this Maven-archetype-webapp which is really just a simple webapp, and that's the one we're going to go ahead and use. So I m going to choose this org. apache. Maven. archetypes Maven-archetype-webapp, and click next, and it's already tried to guess what my group Id is. I've com. Pluralsight, that will work just fine. And I'm going to substitute in my artifacts ideas, my application name here. So let's call this FitnessTracker. Then I can click finish and it will go out. You'll notice that down here in the corner, it's downloading a bunch of stuff and unpackaging some resources. So it's going out and pulling down those templates and substituting values for me. And you see now, I've got a directory structure that's all set up. So it has my source directories, it has some basic Maven Dependencies, I don't really have anything defined yet so it just shows that junit dependency, then it shows my source, main, webapp, WEB-INF directory with my XML folder in here. We're not going to add anything in there right now, I just wanted to show you that the basicWeb Project structure is already intact for us. Let's go ahead and double click on our pom file and open it up. And you'll notice it has our values that we just entered in there, all set up for us now. Let's go to the dependencies tab right here, and let's add our three dependencies that we need. And notice right now our Maven Dependencies directory only has junit in there and that's just 'cause that's the default one that it comes set up with. Let's go ahead and add a dependency and we want to do org. springframework and our Artifact Id is spring-webmvc. For our version, we'll do 3. 2. 0. RELEASE, and the scope of compile is fine. Let's go ahead and hit OK on that one. Let's add another one and we'll do javax. servlet and we want to do this servlet-api and 2. 5 for our version. Now for this one, we're going to choose a scope of provided, and provided means don't package it with my WAR file. We don't want to package these with our WAR file because they're already provided Tomcat for us. So let's go ahead and hit OK and then we have one more and that's jstl. So we'll do javax. servlet and jstl for our Artifact Id and 1. 2 for our jstl version, and also set that to a scope of provided. I'll hit OK on that, and then we want to save and you'll notice that for a second down there, I already have them downloaded, it went out and downloaded all this jar files for me. Now, you may be asking yourself, "Hey, wait second. We only added three dependencies. " This is why I like to use Maven. To use Spring MVC, I also have to have the spring-context jar and the spring-aop jar, as well as spring-core. It also needs commons-logging and some other jar files out there as well, the aopalliance and a few other things. So this gives me all the dependencies I need using Maven through what's called transdependencies. Like I said, don't worry yourself too much about this. If you have questions about it, you can go view the Maven course that I have on Pluralsight that will answer, if not, all of these questions for you.

Spring Configuration Parts

So there are four basic parts that we need to have configured to utilize spring MVC in our application. We need to configure our web. xml, configure out servlet-config. xml, our servlet-config, add a controller which is just a simple Java class that's annotated and then add a view. And our view could be JSP, it could be Freemarker, Velocity. We're going to use JSP for this class, which is most commonly used but the other templating frameworks are used quite often as well.

Where to get Tomcat

For this application, we're going to deploy it Tomcat. Tomcat is the most common or popular web container for Java development. Don't be confused though, Tomcat is just a web container. It doesn't have app server functionality like some of the other bigger, more supported vendors that are out there, Oracle, WebSphere, those types of tools. For most web applications though Tomcat works just fine. So we'll set up our Tomcat server and then our FitnessTracker. war will actually reside on top of that. This is what ends up serving up our application for us.

Tomcat Installation

When setting up Tomcat inside of Spring STS or Eclipse, there's an option to have it download automatically for you. I personally don't like to use this option because it always downloads a version that's quite a few releases older than the most current one, and I want to use the current one because of bug fixes and other things that they've improved upon performance possibly and those types of things so. Go ahead and open up a web browser, like I have here and you can go to tomcat. apache. org and once that comes up, you notice that there's the download section over on the left-hand side here. I want to go ahead and use tomcat 7. 0 and it'll pull up a mirror. We specifically want to grab the zip file. So I'm just going to start that downloading. Now, you notice there's other versions here. There's the 32-bit and 64-bit Windows installer. For development, I don't generally use those because they try to set things up as a service or manage it through Windows and we're going to be doing all of this right from within our IDE. So let's go ahead and show the folder that this downloaded it to. And you'll see that it's just a basic zip file. So we can go ahead and open our C dev tools folder where I like to expand all my development tools, too. And I can go inside of here and grab that package content and drop it right in my tools directory. Once that's done copying over, we can go back inside of spring STS and there's this new Servers tab. Now, if you don't have this Servers tab, maybe yours wasn't set up to open up the same way. You can just go to Window, Show View, servers and that will bring it back up. So let's click on the new server wizard and you'll notice that there is Apache Tomcat. I'm going to go ahead and choose Apache Tomcat 7 and click next and this is where we need to browse to our installation directory. Notice a minute ago I was talking about how it downloaded a version that's quite a bit older. I think we downloaded 7. 0. 22 or. 24 so it's significantly older than one that we downloaded. Let's go ahead and click browse and I'm going to go to my C dev tools directory and apache. And you just want-- you just want to select the top folder, apache-tomcat-7. 0. 34-- oh, it's 34. They've even had a few more releases since I started writing this course. Let's go ahead and select that and click next. And you'll notice that it actually has the option for us to add our project to Tomcat already. So since we used Maven and it recognize that as a web application, we can go ahead and select that and say, "Yep, go ahead and deploy that application to Tomcat when we start up our app. " Let's click finish, and you'll notice it shows our Tomcat server down here and it added FitnessTracker to it. Now one thing I might note that when you create a server down here, it will create a Servers folder up here. And that Servers folder has your Tomcat configuration in there. We don't need probably go anything-- go into any of these files for this class specifically. But this is where you configure some database stuff or those types of things inside your server. xml. A little bit more of an advanced topic, we'll probably cover it later.

Programming Standards

The last section we need to talk about before completely building our basic application is standards. So what are the standards that we kind of loosely adhere to or tightly adhere to for development and why should we do these? So for views, there's a common practice inside of Java to put your jsp pages or any of your view technology under your WEB-INF directory. So we have our folder structure I've put out here in front of you, source, main, webapp, WEB-INF, and then jsp pages. Now, the reason for doing this is often confusing to people. There's a handful or reasons. The first reason is security. I don't want people to know specifically what my file name is. So, I have to serve out to them whatever file that I want. The second reason is so that I can control their user experience. So I don't want people to directly navigate to us. So let's say I don't even care that they know that our filename is addTime. jsp. If I force them to go through my controller and through my routing, I can control their user experience, insuring that their session was set up correctly or that they are logged in, all those types of things. There's other reasons too, bookmarking, deep linking, things that are called breadcrumbs. But for what we care about, those are the two main reasons, security and just controlling their user experience. Now, how we do this is we use what's called an InternalResourceViewResolver. One thing you'll learn about Spring is they're notorious for having very long names for their classes. This InternalResourceViewResolver is what says, "Hey, when I ask for this jsp page, I'm going to go find it in this directory and serve it up to them". So, essentially addTime. jsp is the same thing as this URL right here. Now, the next section, we're going to talk about is controllers. Controllers are Annotation Based. They are a simple podio that you just add the @Controller annotation on top off. They're named whatever you want but typically named around your business domain. So if we're dealing with time, or we're dealing with people or like if we're dealing like with the hotel application, we'd have a hotel controller or those types of things. Path is set using an annotation. So we're going to set our path using the RequestMapping annotation. Don't worry about memorizing all of this. This is just to kind of what your appetite a little bit so that you'll understand why we're talking about this here in the next couple of demos. And a good example of this is our TimeController. java. So for adding time, deleting time, what our goals are for that, that would be in our TimeController. java file. Some examples of the URLs associated with our Controller could be time, addTime, updateTime, those types of things.

Upgrading the schema in the web.xml

So now, one last change, let's make before we start editing our application or rolling out our Hello World application, I don't care for how the archetype used, created the web. xml. It actually created the web. xml with an old DTD in here, and now, we're using schemas instead of DTDs and these are the 2. 3 specification, and they just did that to be backwards compatible with everything, but we're using a 2. 5 server web-apps, so let's go ahead and use that. So, I've actually got one here that I'm just going to paste in. And this will actually be available in the course file so you can actually just copy it in from your course file as well. I mean you could type this in if you wanted to but it's quite lengthy and kind of error prone. So just-- you look for this inside of our exercise files for this module or just search for 2. 5 servlet declaration, the web. xml servlet spec and copy the schema definition in there.

Configuring the web.xml

So, now that we have our web. xml ready to start making the changes to integrate Spring MVC into it, let's go ahead and get started. First, we need to create a servlet and the servlet is the Spring MVC dispatcher servlet. So we're going to go ahead and use the servlet element there, and as part of that, we want a servlet-name. So our servlet-name could be fitTrackerServlet, anything we really want to name it. But we have to be very careful and we need to know what that name is because we're going to use it in a servlet mapping here in just a minute. Next thing we need is the servlet-class. And the servlet-class, little bit of a long one here is org. springframework. web. servlet. DispatcherServlet. And that does have to be spelled correctly, so make sure you got that right because that will cause you an error. Now, I should tell you that by default, you don't have to have this next section in, but everybody recommends you do it, myself included, and that is specifying the exact location of your configuration file, your servlet-config. xml. By default, it will create one for you called-excuse me, fitTrackerServlet. xml or fitTrackerServletconfig. xml. It won't create it for you, that's just what it will look for. We want to specify exactly where it is and what we want it named just to be clear and it just takes two seconds anyways. So we're go ahead and add an init-param. And inside that, we're going to have a param name. The param name is contextconfigLocation in CamelCase exactly how I have it spelled there. And the param value is-- we're going to use is WEB-INF and it has to be all caps for the WEB-INF portion, config and we're going to just call ours servlet-config. xml, okay? Now, that defines our servlet but we don't have any servlet mapping set up just yet. So let's go ahead and set up our servlet-mapping here. The servlet-mapping is what routes traffic through. So even though we have the servlet defined, let's-- oops, I will carry it away there, do the servlet-mapping. And this name, remember how I told you we need to make sure we remembered our name up there, it's exactly the same as this name up here. So that's the key between the two. Usually, I try to not have any spaces in servlet-name or anything like that, just to help alleviate any error I might have. The last section we need is to add our url-mapping to the end of our url-pattern to our servlet-mapping. This is simply just what file extensions or what path we're going to accept for our Dispatcher servlet. So for now, let's just do \*. html, little later we're going to do some different ones here so I don't want to confuse you yet by adding too many of them upfront. Let's go ahead and save that. And now, our web. xml is complete.

Namespaces in Context Files

So when configuring our servlet-config, specifically our beam configuration or beam definitions, there's the notion of Namespaces. Now, I'm not going to dive in too much here, but just more or less point out what they are and what they're doing. So the top of a beam definition file may look like this. At first glance, it looks really complicated. I've got a lot of things going on here. Really all it says is I got a set of Namespaces that have some defined functionality with them. And this is an example of us using them. So, I've defined these Namespaces and here I am using them. And specifically, we'll show you an example here. I've got the MVC Namespace defined, you see by that little red circle there. And here's where I'm using it down below. And what this means is that I'm asking for specific functionality out of that MVC schema, and it's going to go through and replace this code with more complex definition. Spring did this in an effort to simplify their configuration files, because it was always copy and paste Boilerplate code. When anybody did annotation driven, they had the same exact annotation driven code. So, to make it simpler, they just you define a Namespace and then you can just drop this one tag in your code and you're done.

Configuring the servlet-config.xml

So now that we've configured our web. xml, let's create our servlet-config. Now, if you remember inside of our web. xml, we told it we're going to put it underneath our WEB-INF/config/servlet-config location. So, let's right click on the WEB-INF directory and say New Folder, we'll call it config, and make you've done it on the WEB-INF directory and not webapp or it won't put in the right place. So, WEB-INF config, it should look like something similar to this in your address bar up here, and then let's click Finish. Now, once we've done that, since I'm in Spring STS and it's almost the same thing in eclipse, I'm going to right click on config and say New, and I'm going to go to Other. Now, in Eclipse you would just do file but I'm-- since I'm in STS, I'm going to use SpringBeamConfigurationFile and I'm going to name this servlet-config. xml. Now, really all these bought me since I'm in Spring, is that it added this Namespace up here for me. As I showed you in the previous example, there's just some namespaces that we're going to use inside of our application, that just help make things a little easier for us. In fact, if we go down to our namespaces here, on this tab down below, you can see that there's a handful of namespaces I can add to my application. So, I'm going to click context and I'm going to click MVC as well and it just says "Hey, we're going to add these definitions in there, just making it easier for us to navigate down below here. So, now that we've got these-- now that we have these allotted in here, I can save my application and go back to my source code and you can see it just added a few more namespace definitions up here, nothing too complex. And the sample code for the exercise, we'll have this in there as well if that's not making a whole lot of sense to you. Now, I'm going to add my two pieces I need to annotate and start building my controller. So let's add MVC and we want to do annotation-driven, all right. And since we're not adding anything to it, I can just back up there and put that forward slash in and it will automatically get rid of the trailing-- closing element. And I'm going to also add the contextcomponent scanner. Now, the mvc annotation-driven just means "Hey, we're going to use annotations to configure our application which we're going to. Now, the next one I'm going to do is context and I'm going to do component-scan. Now, component-scan says "Hey, where should we start looking for annotations at? " Well, if you remember, when we created out project, we decided we're going to name our package, now you could have named it something different. But we created ours and called it com. Pluralsight and we're going to put all of our controllers underneath controller package. Now, we haven't created any of those yet, so there's nothing that I tying to just yet but we're going to. So, let's go ahead and close that off and we're done. That's all we need right now for our configuration. The-- let's go ahead and save that file. That's what I was doing when it popped up the Windows Explorer there. I'm going to save this file and now we're ready to start building Java code underneath this package structure.

Creating the Controller

So let's start creating our controllers now. Since we haven't added any Java code in the application yet, I actually don't have a source folder. So let's go ahead and right click on FitnessTracker and say New Folder and do src/main/java. Now, if yours didn't automatically put it up here as a source folder, then right click on it and say BuildPath, Use a Source Folder, all right. So now mine's up there with the rest of the my source code folders. You can tell by the little icon on the upper right hand corner that it's a source folder. Now, let's right click in there and say New Class. Now, for our Class, let's just call this the HelloController. And for our package, let's call it com. Pluralsight. controller. We want to put all of our controllers in a controller package. And we don't ever want to use the default package, just can cause problems long term. Now, the nice thing about is Spring is everything is podio. So, I don't need to implement an interface, I don't need to extend the class. There are some things you can do that have some convenience methods in them. But for example, this works just fine. So let's go ahead and click Finish, and there's our controller. Now, clearly we have to add a few things to it, for it to become useful for us. So, let's go ahead and mark it as a controller. So, we'll going to do @Controller. And if I do control spacebar at the end of it, it'll automatically do the imports for me. Now, let's add a method, and the method can actually be named anything we want, it doesn't necessarily tied to what we're trying to do. It uses the annotation for the RequestMapping for that. So, let's just do a method called public String sayHello, and we're going to pass in the model object, control spacebar, and that will give us back our model object. And we can close off our method. Now, let's go ahead and say return, let's call it "hello, " all right? So we have the basic architecture in place for our controller now. But what we're going to do-- we're going to do two things here. We're going to add a Requestmapping which is how we tie this to the URL. So we want to say RequestMapping and then we want to give this RequestMapping a value, and again just control spacebar, value equals. And let's tie this to /greeting, all right? Now, that's what says what our URL is going to be and what method it's going to tie to. So, http://localhost8080/fitnesstracker/greetings is going to tie to this page. And we're going to walk through that example but that's just tying that whole picture together for you. Now, the other we want to do is want to do is we want shove some things in the model and have that displayed in our UI. So we're going to model. addAttribute and we're going to say, greeting and we can just put in here "Hello World. " And that's all the pieces we need in place from our controller. This is a full functioning complete controller.

Create JSP and Configuration

So now we that have our controller finished, we have two more pieces to put in. We need to put our JSP page and the mapping for our JSP page to our controller. So, looking at our controller that we just finished up here, you noticed that we-- we had this model parameter that we added here. Model has an attribute in there with a key of greeting and a value of "Hello World. " And model is just simply a hash map, basic map structure, it does a little bit more than that, but that's all you need to know for what it does for now. Keep that in mind when we create our JSP page. I'm going to reference back to greeting here. As you recall when we're talking about some of the prerequisite stuff or just some of the standard that we're going to code to, we'd like to put all of our JSP pages under the WEB-INF directory. So, let's right click on src/main/webapp WEB-INF and say New Folder and let's do JSP, click Finish. Now, let's right click on JSP and do Other and chose Web JSP file. Now, be careful here because by default, it chooses the wrong directory for you. We do not want it in this directory here, instead we want it in our src/main/webapp/WEB-INF/jsp directory. I've made that mistake multiple times. So, make sure you've got that in src/main/webapp/WEB-INF/jsp and let's call this hello. jsp, all right? Click Finish. Now, the beauty of Spring and JSP and doing it this way is that, by default we have the standard EL language available to us to extract values out of our request parameters. So what we can do is we can add an h1 element down here, standardized http tag. And we can do dollar sign, curly brace, greeting, okay? Let's go ahead and save that. Now, you remember greeting is what I was just reminding you about and the hello controller. Let me just put this two side by side here you can see them. Hello controller, we have this model attribute here, greeting. And inside of our hello. jsp page, we have greeting there. Those are what tied those two together, that's how it knows what to grab out of that model to display on the page, okay? The last piece that we need is to tell our controller which JSP page it should display with this. So, if we open up our servlet-config, src/main/webapp/WEB-INF config, servelet-config, I actually still have mine open here, you probably do as well, we need to add another bean in here. Now, I'm going to show you two ways to configure this bean. I like the more condensed version but it is a little bit of black magic so I'm going to show you both ways. The longhand way is to do bean and to do class and we're going to do org. springframework. web. -- ah, let see, servlet. They went for the longest class names here. Servlet and we want to go view,. view. InternalResourceViewResolver. Now, the longhand way of doing this is to come down here and now specify this bean property. So we come down here and say property and we can do the property for prefix and I give that a value of WEB INF, WEB-INF/jsp/ and closes that off. Or I can do-- and then I also have to do the suffix and this is just saying where is my JSP page, where my JSP page is located at and what suffix am I looking for them with? Oops, give me value. And our suffix should be. jsp so with our JSP extension and so JSP. So that's the longhand way, okay? Nothing wrong with that. It works just fine. I'm going to go ahead and comment this out and show you the shorthand way which really isn't that much shorter but just more and more examples and things that you see, codes, snippets are done the shorthand way. The shorthand way, I'm going to go down o my namespaces tab here and I'm going to click P for the property namespace and go back to my source and do bean and we have to suffer through that long class name again. We'll do org. springframework. No, we can just copy this up here. ( Pause ) Copy. Paste. And the shorthand way is I can do p colon and we can do a prefix. And on the same line, we could put all this information, WEB-INF/jsp/ and do p colon suffix and for that we'll have our. jsp extension. And then we can close that off. So, really, you know, is it a whole lot better, you know, that's debatable. It's just more and more of what you're starting to see people use. It means the same thing both ways. This is just shorthand for calling the getters and setters for prefix and suffix.

Run the Application

So let's run our application now. We've just done all those work to get everything up and running and configured and built and all the pieces in place. Let's run it. Now we've got everything wired up, our server should be configured down here and our web app should be added to it already down here. So you should see FitnessTracker under your server tab down here. And if you accidentally close this tab, let me show just like I did there. You can go up to Window, Show View, Servers and that should pop that back up and if you'd expand it, you should see FitnessTracker underneath there. If by chance you don't have FitnessTracker underneath there, you can re add it by just right clicking on it and say Add and Remove and going to FitnessTracker and adding that back over. All right, so everything should be added there. Let's go ahead and start our application up. Just click that green arrow and you should see it load up. There shouldn't be any errors over here. So I want to show you this console because if you really pay attention to the output from spring, it can help you a lot. There is no error messages. There is a warning. It has nothing to do with that so we're good to go there. So let's go ahead and pull up a web page and we'll go to loaclhost8080 and I actually have the URL in here because I've ran it before. And we want to go localhost8080 FitnessTracker/greeting. html. If I enter that, it should take a second to compile at the first time and you should see Hello World. So, everything should run fine. If it didn't, make sure you look at your console output and see if there is any error messages that didn't have the, you know, popped up or happen down here. You'll notice when Spring initialize and loaded everything, it will tell you what your URLs are mapped to and what beans it's created and those types of things down here. So, pretty informative console if you pay attention to it.

Recap and Walkthrough

Let's do a quick recap of what we did in this module 'cause there was a lot of things that we did. So, we started off by configuring our web. xml. In the web. xml, we went through and added the servlet, the dispatcher servlet and told it where our configuration location was at and we mapped all of our incoming requests to. html. We're going to play more with this later and do some different things with JSON and some other responses so this won't be the last time we look at that. The next thing we did is we created our servlet config. xml. Now, we did a quite of few things in this servlet config. We added some namespaces which really were just convenience methods for us. We also told it that we wanted to do annotation driven and set up our context component scanner. Now, a context component scanner tells it where to start scanning and for what classes to scan for those annotations. So, the first one tells us that we're doing annotation driven. The second one tells us where those annotations are at. And then the last thing that we did in this file is we created an internal resource view resolver. And that not just says, "Hey, where are our JSP pages at? " So when our controller says, "Hey, I need a JSP page, where is it located at? " This is what tells it where that file is located at. Now, the other thing we did in here is we created our controller. So we added our source folders, source main java and we created a controller under com. Pluralsight. controller. And our HelloController is pretty simple. This is the beauty of Spring MVC. I have a controller annotation and a request mapping annotation. And a request mapping annotation just says what URL am I going to tie to? So we told it to tie to greeting. Then inside of our method and our method could be named whatever it wants, we added our model object. Now, if we add that as an incoming parameter, this model object here, you can see the hover focus going towards it, that will tell it to populate that and return to this part of our request response lifecycle. Inside of our model we added an attribute for greeting that's going to tie to our JSP page. And then our return, we told it to go to the hello JSP page. So this name right here is the exact ties, the key to our JSP pages. And if we go back out our servlet config tied to our JSP directory that we have created here and specifically our hello. jsp and this was just a plain JSP page and there is our greeting that we're pulling out of our model right there.

Architecture

Introduction

Hello! This is Bryan Hansen from Pluralsight. And in this module, we're going to discuss architecture. We're going to talk a little bit about the architecture of Spring MVC and also about how Spring MVC fits into the overall framework of Spring.

Architecture

Software architecture has been around since long before Spring and Java even. Architecture and design patterns are important though because it gives us a common vocabulary that we can talk about features of framework. And if you some background in software development, you can know what a framework or an application is doing without knowing exactly how it's doing it. So, 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.

MVC Design Pattern

So here is a standard MVC or Model View Controller design pattern that you may or may not have seen before. It's pretty common pattern now and most developers have at least seen it or heard of it. A request comes in 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 database off of the user's actions. Now, this graphic is sometimes confusing though because it stemmed from rich client applications that were more event-based or using a design pattern called observer, observable, or a subscribe-listener type design pattern. Not really applicable to a lot of software development with do with the web. The pattern in a concept is still sound but it's-- we usually don't have our model updating our view because we've gone to a page at that point. Now, a more accurate description or a kind of life cycle that we may be faced with web development is this graphic here. This graphic is realistically more like the life cycle that we see in out web application. Our view can access our model but it usually does so through a controller. We'll look more lightweight approaches in the AJAX module later in this course. But the summary is, is we usually still make a request even if it is a lightweight request through our controller to access our model. We don't have our view typically going directly against our database or our model.

N Tier Applications

So, you'll often hear people talk about applications in terms of a tiered architecture or a 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 is assigned to do. 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-- data access is going to also be contained in its own separate layer. Now, there's another term called reusable layers 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 architectured in the correct tier. Now, that's a little bit of difference between the separation of concerns and the reusable layer. Separation of concerns, it's not about reuse, it's just about having things in the right layer so that I can re-architect them later, where reusable layers means that I have a purpose and a point that I'm trying to drive of-- 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. And then another term Maintenance or Refactoring might be a better way to put it. That's the ability to change things without having those changes 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 of a change I made in Hibernate. No, I'm not talking about, you know, a business need but more of I want to change how a table is structured or things 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 is outside the focus of this course.

Application Layers

Let's talk about the layers or tiers of a 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. So, first we have a data model that we would access using something like Hibernate or JDBC or Spring JDBC. And this layer represents the data or model of our application. Next, we have a controller that interprets the user's request 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-- for our module we're going to be using JSP, but our view could be FreeMarker or Velocity just to name some of the common templating tools. 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 this. So, is the model talking about our database or the model talking about what are framework is going represent just to our JSP page? It's actually referring to what we're going to representing in our JSP page but we always have to get that information from somewhere. So, that's where our various components come in that we're going to talk about next.

Components

So we've talked about tiers and we've talked about layers and we've started to allude to components inside of our application now. So, how do we represent these tiers with Spring and Spring MVC? And that's with these three components. We have our Controller, our Service and our Repository. We've discussed controllers already. They just route where we are going 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 probably start if we're accessing more than one database table as well. And then there is Repositories. The Repository tier is also sometimes referred to as a DAO or Data Access Object. And they usually have a one-to-one mapping with our database table.

Controllers

Controllers, as we've mentioned, handle our incoming request and 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 handed over to the business logic. This works with our service and repository tier for business logic and data gathering. And it's annotated with @Controller. There are some convenience classes that you can extend but you either have to wire them up or annotate them with controller still. It also should be noted that it handles exceptions and routes views accordingly based off of whether or not we had an exception or we've got the correct information.

Services

The service tier is annotated with the @Service annotation and it describes the verbs or actions of our system. So this is what we're trying to do inside of our application. It is without a doubt where our business logic should reside. In fact it should all be contained here. It shouldn't bleed over into our repository tier. And it ensures that the business object is in a valid state. So this is where all of our state management should be handled as far as "I've got a valid object. It's, you know, it meets the confines of our business objects or of our business requirements. And it's also where our transactions should begin. So, for doing two phase commits, or where we know where there's a chance we might have to roll back or access web services, those types of things. This is where we want our transactions to begin. It often has the same methods as the Repository but a different focus. So, 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.

Repositories

And lastly, the Repository tier is annotated with @Repository. It also describes the nouns of our system. So you can see where the focus is different from Service versus Repository. The Service tier describes the verbs or actions that we wanted do in the system, and the Repository tier describes the nouns or the data of the system. It's focused on persisting and interacting what the database or basic CRUD functions. It's also typically a one-to-one mapping with an object. So, I may have an address and I have an address repository. I'd have a customer 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 of your database design. So, you may break things in a multiple tables like a person in an employee table, but you may only have an employee object inside your application.

Summary

So just to quickly recap what we covered in this module, we talked about Software Architecture and how it's a much needed thing inside our industry to convey information and vocabulary about what we're trying to do. We talked about MVC and how the traditional MVC design pattern isn't necessarily very reflective of what we're doing inside of web applications. We've also talked about N-Tier architectures and why it's beneficial for us, and that lead us into a deeper discussion about the various components of our application 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 backend and where our business logic should reside. And even though there are more of Spring Proper rather than Spring MVC, we should always use those within Spring MVC to help with our model view controller design pattern.

Controllers

Introduction

Bryan Hansen: Hello. This is Bryan Hansen from Pluralsight. In this module, we're going to be looking at controllers in Spring MVC. Controllers are really the heart and soul of Spring MVC. They are a gateway or a 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 are not and how we can use them in order to build a successful Spring MVC application.

Controller Responsibilities

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 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 and RESTful services, though, 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 action or a request, and then the view, or the actual web page, is just a result of doing some action. So let's look a little more at the actual architecture that makes up a controller. So we've already seen some of this before, and we've actually even created a controller when we wrote our Hello World type application. So here's what's happening as we're going through a request from the user. So 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. It can theoretically be a web service or it could be our database or it could just be our business logic tier, and it's going to produce some output. Now we can think of that as our model for our UI, and it will eventually return that model back to our UI on our response. So as we come out of our web service, we're going to hand that back to our controller. Then our controller's 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 our user. So to kind of recap what are the responsibilities of a controller, well, it's to interpret user input and transport that input into a model. So 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 also will determine the correct view based off of our logic. One other key point that never really seems to come up to play as far as the duties of the controller is it also interprets exceptions for the business logic and service tier and how to handle and navigate 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 to this error page rather than requesting that you add this information or correct this information on the screen. So its duties also are to interpret those errors from our middle tier.

Controller Annotations

So controllers in Spring MVC are very lightweight, and they actually don't even require you to implement an interface. So if you remember our controller definition from our Hello World action that we did earlier, Hello World controller, 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 we needed a request mapping annotation. And the request mapping annotation tells Spring which method is going to handle which request. Now before we move on, though, I wanted 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, but it works just fine and it isn't deprecated. More and more people, though, are choosing this 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 the same application of doing one, and since I'm going to show you some things with RESTful services later, I chose to just show you this example. There are other ways of doing it, though, like I mention.

Demo: Build Controller

I've got the fitness tracker project open now, and we're going to start off by creating a new controller that will allow us to enter our minutes. We're not going to complete this entire controller life cycle just yet, but rather, we're just going to get enough of it working to display our result page. If you remember, what we had done so far is we had created a Hello World controller that just displayed a greeting. And let's go ahead and view that and just kind of review what we've done. So let's open up our code here and I'll show you the Hello World controller that we built. Here it is. Pretty straightforward, had our controller annotation and had our request mapping annotation mapping on there. And this linked it to where our application URL was tied to greeting. Now if you remember, I very specifically did a different return string than what my greeting type was, meaning that the URL is going to link to greeting, but we're going to return the hello JSP. As we open our Source, Main, Web App Config -- oh, excuse me -- WEB-INF/JSP directory here, you'll see that Hello JSP page that that's returning in and showing here. Let's go ahead and start up our server just to show that this is still working and everything's functioning fine. I'm going to go down to my Servers tab here and click that Start button. And if you've lost this tab, if you accidentally closed it, you can go up to Window, Show View, Servers, and that'll open it back up here. So let's start that back up. It should start up pretty quick, it does. I'm going to ahead and open up our browser and pull up Local Host, and there's our Fitness Tracker. And I've got a default index page here. Let's go to our greeting. HTML. So our greeting. HTML page returns Hello World just like we would have expected it to. If you remember, our Hello page here has just got that greeting that we're pulling back out of our model object. So this right here is just using JSTL and the EL expression language to pull this greeting here, this string here and display it on our JSP page. So now that we've done a little recap, what we're going to do is we're going to go through and make it so that we're going to have an act -- a controller that will interpret our application's request called the Minutes Controller. When we go to this controller, we want to go ahead and display a page. This is going to allow us to enter our minutes exercised and show you what the total is already. So let's go ahead and create our controller first. I'm going to come up here and I'm going to -- if I click on my package like you can see I've done right here, I'm going to right click there and say "new class" and it will automatically fill in my package for me. If I click somewhere else in the application, I've got to put this information in, but since I right clicked on my controller packager here, it'll automatically fill that information in for me. I'm going to go ahead and type the name in here of Minutes Controller. And that's it. I don't have to add any super class, I don't have to add an interface or anything like that. It automatically -- it's just a POJO, and we will annotate it from there. So now we've got our class here. Let's go ahead and add our at controller. And I can use Control Spacebar here and it will bring up my IntelliSense for me, at controller. So this tells Spring that hey, I am a controller and I want you to find me and I want me to use me. Now if you remember from our configuration earlier under Source, Main, Web App, WEB-INF Config and our Servlet Config, we had this context component scanner right here that tells it to start looking for controllers underneath that directory, underneath that package. So that's how it's going to pick this up because this code is in this package and we have that controller annotation. All right. From there, let's go ahead and create a method, and since this is just a POJO, it doesn't have to be named anything, you know, specific. So let's do Public String Add Minutes. And for now, we're just going to leave the signature empty and we'll return Add Minutes. Just like that. All right. We need one more piece. We need to use our request mapping that we had talked about earlier with like on our Hello Controller where we had mapped this here, that request mapping. So let's go ahead and do At Request Mapping. And this just says what URL am I going to intercept this with, what's going to tell it to drive -- what's going to tell Spring MVC to drive to this method. So let's go ahead and add the value parameter here and the attribute and we're going to do Add Minutes. All right. So pretty straightforward. If we add -- you know, deployed this on our server right now, we would -- we could go ahead and type the URL -- nope, we need to have a slash in there. We could type the URL in there, Add Minutes, and it would try and display this Add Minutes. JSP page. In fact, we would get an error. I'll show you that real quick just so you can see it. We do Add Minutes. HTML and we get a 404 because there is no page for that to return to. And that's actually what this error message will tell you down here is that there's no mapping found for this HTP request. So it tried to display that page for us, but there's -- we don't have a page for it yet. So let's go ahead and create a new JSP page down here. We'll go to New, Other and we'll do JSP file. Oh, ahead and make sure you get the right directory. I don't know why this is choosing a subdirectory for me. It's done this to me a couple of times. We'll do WEB-INF/JSP and we want to do Add Minutes. Okay. So make sure you have Source, Main, Web App, WEB-INF/JSP. selected. Your address bar here should look just like mine and then the file name should be Add Minutes. JSP. Let's go ahead and click Finish. And we've got a very basic page here. We can change the title, Add Minutes Page whatever. We can throw an H1 down here and just show that we've got, you know, something on the page. Add Minutes Exercised. And let's save that. Let's go ahead and restart our server. You may not need to if you've got it set to automatically redeploy, to automatically publish. Well, let's go ahead and restart our server just to make sure we don't have anything hung up there with deployment. And now that it's done, go ahead and open up our page again. And let's hit that Add Minutes. HTML and there's our page. So to walk there what happened here, when we go to this Add Minutes. HTML, it wants to go through our web. XML, and our web. XML says hey, send everything that is of asterisk. HTML to our dispatcher servlet. Well, our dispatcher servlet is configured using the servlet config. Our servlet config says hey, I want to scan for all of my controllers at this location, I want to be annotation-driven and I want to look for my controllers in this location here. And since our controller is annotative to at controller, it says oh, okay, I'm going to handle that. And since we put a request mapping on here of Add Minutes, it says I'm the method that's going to handle that. Now if you look down here in our console, you can see when our server started up that we have mappings for greetings and mappings for Add Minutes to our servlet. So that means it picked it up, our Spring MVC configuration picked it up. That's one way to tell that your annotations and stuff are working right because it actually says hey, I've got a mapping for this file here. And then it returns our view. Now, if don't remember where the view comes from, we're going to talk about this a lot more later. But in our servlet config, there is this internal resource viewer here that says hey, look for my JSP pages in my WEB-INF directory right here and look for a suffix of. JSP. So basically, I'm going to look for Add Minutes. JSP, this guy right here, underneath my Source, Main, Web App, WEB-INF/JSP directory. Now, once that's all done, it will just go ahead and return that JSP page and display our information here. ( Silence )

Parameter Passing

To be able to pass data from our JSP page to our controller, we're going to use some tag libraries provided to us by Spring MVC. We could actually just use standard HTML input tags, but Spring provides us a tag library making it easier to interact with our controllers. So historically, we've always just passed HTTP parameters that have gone on the 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 -- the tags that Spring provides to us take these values from our input page and either 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 take, or just be able to grab them off the request. So we can always pass in the HTTP servlet request into our method and access them those ways. And I'll actually walk there an example showing you all of those. The model can actually be a bit confusing as you'll notice that's in our controller box here, our controller object here. The model is used for both gets and posts to our web page. So when I'm retrieving data and setting data to it, so that incoming and outgoing data often confuses people.

Model Attribute

The model attribute annotation 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 all of this is 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. So we'll use the model attribute when we want to do an HTTP get to get data back. So say I'm going to get a drop-down of like states or, you know, drop-down list type data, gender for a male/female drop-down lists or those types of things that might be driven by our database. We also use it posts so when we want to send data back, so if we have a form that we're filling out, and that's actually what we're going to use here in a demo to send our minutes exercised back to our controller. Like I mentioned, it works with POJOs 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. And it can also be validated with a binding result. We're going to cover that in a later module, but just to point out, you can have this data validated to too as part of your life cycle.

Demo: Post Data to Controller

So this is where we left off from our previous demo. We have an Add Minutes Exercised screen here that we haven't put any form data onto yet. So let's go ahead and open up our project and go to our JSP page and start adding some of those form elements on there. So here I've got my Fitness Tracker application open, and I've got my Add Minutes. JSP page. And right now, it's straight up HTML. There's nothing too interesting going on here. Let's go ahead and add the tag library for Spring and specifically for its forms. Now we are going to go into these tags deeper in a later module, so don't worry about covering all the details here. Let's go ahead and add a tag library, Reset At, we're going to do Tag Lib and I want to give it a prefix -- oops -- prefix Equals to Forms. And -- actually, I just want to call it Form. And I want to give it a URI of HTTP:// Dub, dub dub. springframework. org/tags/form. Now, just so you know, it doesn't -- Java doesn't go out and download those tag libraries from this URL every time. If you're not very familiar with tag libraries, you can actually look some up out of JARs on our class pass. So this is just really an index into our JARs for -- to know to go download that. And we've had those downloaded. They're actually over here in our Maven Dependencies. But just -- that's where it's pulling that data from. Let's go ahead and go down here to our body section and go down to H1. Let's add a forms tag down here. So you notice, we have all these forms now available to us, these elements because we've added this tag library up there. So let's go ahead and use Form, Form. Now what this is is now I have Spring form instead of a standard HTML form that's going to allow us to bind an object to our data that we're submitting back to our controller. So inside of our form, let's just add an old school table, nothing fancy here. And add row, let's actually add two rows, and let's put some data in there. So in this first box, let's just go ahead -- this first column, let's just add a TD and we're just going to do Minutes Exercised for Today. Close that TD off and then let's add another TD and we'll do a form input. But we're going to actually use Spring's form input rather than just an HTML form input. And we're going to put a name in here of Minutes. Now this name is going to tie to an object. I will tell you right now that this page is going to break and we're going to go through and fix it. So this is binding to an object with the path of minutes on it. Let's go ahead and close that. Now let's go down to our other column down here and do a TD. Let's give it a call span of 2. And we'll just do a regular input, just a regular HTML input here and we'll do Type of Submit. And the value of -- oh, I don't know -- Enter Exercise. Doesn't really matter. Close that off, we'll close our TD off and we're good. So, as I mentioned, this page is going to break. I'll tell you right now it's going to break. But I want to show you what the error message is that you're familiar with seeing it. It's a mistake people often make. So now this is restarted. Let's go ahead and pull up my browser. I'm just going to refresh this page and you'll see that it's saying hey, Line 17 there's an error. It even shows us what's Line 17 and hey, surprise, surprise, it's that minutes path. And it tells us -- if I scroll down here a little bit -- in neither binding result nor plain target object for being named, command is available. What it's saying is it doesn't what beam to tie that minutes attribute to, that -- this guy right here. So let's go ahead and fix that. Let's go into our code and specifically let's look at our controller here. Well, our controller is just Plain Jane simple right now, and we have no objects for it to bind to. So let's go ahead and add another object. So let's go ahead and come out here to our Source directory, so I'm in Source Main Java and I right click and say New. I'm going to create a new package first because I want to put all of my objects under the model package. So. com,. Pluralsight,. model and I'm going to click Finish. I've created my new package here. And I'm going to right click on that and say New Class. Now the beauty of Spring MVC can and is generally POJOs, so I'm just going to create an object called Exercise, and I don't have to have a super class, I don't have any interfaces, nothing like that. I can click Finish, here's my exercise object. Inside there, let's add that variable, that number variable called Minutes. Go ahead and save that. We want to right click on it and generate getters and setters. So right click, go to Source and Generate Getters and Setters, hit OK. So now we've got this POJO. Okay, great, but how does this exercise object get bound to that Add Minutes. JSP page? Well, we've got to do that model attribute that we had mentioned in some of the previous slides. So we're going to open our method here and we're going to put our annotations on that say Model Attribute. And we're going to, inside those parentheses, give it the name of Exercise. Okay. And then we're going to say Exercise, Exercise and save that. Oh, we need to import it, so I'm just going to do Control, Shift, O and it will import that for me. And just so we know, when we hit it for right now, let's go ahead and do a system. out. println and we can just do whatever we want here. Exercise and Add Exercise. getminutes. So we'll grab our minutes that are exercised, all right? Now those are bound, we actually need to tell our form back here what object it's binding to. So if we come in here, we can put Command Name equals Exercise. And this name here is the same name as right there. All right, so this model attribute here is bound to this command name right here. So let's go ahead and save that. And it says everything's reloaded. Let's go back out to our page here and refresh it. Now our page pulls up, everything looks fine. Let's go ahead and enter our minutes exercised for today. We've only done 35 minutes today, hit Enter. And right now, the page is resubmitting back to itself. If I pull my page back up, though, I can see here that when it displayed it the first time, it was 0, but when I resubmitted back into it, it hit -- it entered in 35, so you can see that our code is being executed to go through and calls our system. outprintln for our number of minutes exercised.

Summary

It's just a quick recap of this module. We went through and talked about what the duties of a controller are and the various annotations associated with controller, specifically, the controller and request mapping annotation and the model attribute. We also did a little bit of kind of a brief introduction to the tag libraries and the various tags that are available for binding data from your JSP page back to your controller.

Views

Introduction

Bryan Hansen: Hello. This is Bryan Hansen from Pluralsight. In this module, we're going to be looking at views and resolving views in Spring MVC. Views are how we display the results of what our request was to our middle tier using Spring MVC. In a later module, we are going to also discuss how restful service results are a type of a view but not one that necessarily has a UI component to it.

View Types

Views are what we see from Spring MVC. They are the V in MVC. In our application, we are using JSP pages but we could have just as easily used Velocity or Freemarker or another templating tool. We can even use JSF for our view technology as Spring MVC has an API to replace the model portion of JSF. A lot of people don't look at JSF in using it in this matter but it actually makes it so it integrates with Spring quite well. There are some complex pieces of selecting a view, though, and we are going to walk through that process and how the correct view is resolved in our application.

View Conventions

Although not required, there's a convention of placing JSP 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. You can see I've got a snapshot of our application here and if you look underneath our web INF directory, we have our JSP directory and our JSP file is located underneath that. We can actually add more folders here as well and have subdirectories in there based off of our application architecture. This also enables us to secure our application easier because we are controlling all access to it. Rather than to allow people to go directly to a page, we make them navigate through our framework. To be able to find a page in the subdirectory, we use an internal resource view resolver to find that page based off of the name that we return.

Resolving a View

In our first demo, we configured an internal resource view resolver and have used that in all of our controller sense. We simply return a string and that is used by the view resolver to find the JSP page named and in the location specified with our view 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 view resolver with the data that it needed in the view. We can also have multiple view resolvers as well as multiple view types in our application.

Various ViewResolvers

These are some of the various view resolvers that are provided by Spring and you can also create your own custom view resolver since they just extend the view resolver interface. Some of these view resolvers are for using templating tools like Freemarker or Velocity for your UI. It just configures where it should be looking for those templates and then executes the binding and writes the output to them. The resource bundle view is used for internationalization purposes and the tiles view is used for tiles templated layout architecture. If you've ever used Struts 1. 0 or 1. 2, the earlier Struts versions, we use tiles a lot in that for layout and it carried over into some of the newer frameworks like Spring MVC as well. The last one I'm going to mention here is the content negotiating view resolver. It is used when returning multiple content types from the same application. For example, say we are going to return XML, JASON and JSP or HTML from the same application. We are actually going to walk through an example of the content negotiating view resolver in the AJAX model later in this course. We can also have multiple view resolvers configured in the app at the same time. You just need to order which one gets fired first.

Demo: Chaining

So I wanted to take a second before we dive too deep into our demo and just kind of review what we've done so far. So I've opened up our minutes controller here and you can see that it's just our basic structure that we've got. I return a string. That string is tied to our view resolver and goes ahead and navigates as to what page that we're looking for. So right now, it's saying that I'm looking for the add minute string and you can see underneath our web INF JSP directory that I have an add minutes dot JSP page. Now it knows to look there based off of our internal resource reviewer that's configured in our ServletConfig dot XML. So if you open up source, main, web app, web INF, config, ServletConfig, you can see our internal resource review resolver that we have configured here. Now you can see the prefix where it says look underneath the web INF JSP directory and that we're looking for a suffix of dot JSP. So we're looking for JSP files. Now some people get hung up on this, especially because we're returning a name of a JSP page out of our -- out of our controller. They think it's a little too tightly bound. But it's actually not the case. If I was using one of the free marker view resolvers or velocity view resolvers that we saw in some of the previous slides, it would simply be the same name add minutes right here and underneath our JSP directory it probably wouldn't be a JSP directory. It would have maybe a FTL for the Freemarker template library. I have an FTL directory over here. And underneath here we would have a new file that would be add minutes dot FTL. And I'm not going to walk through a whole configuration of Freemarker just to show you that it works. It does work. It's exactly the same thing as JSP and that's diving into a different view technology. But you can see that this string right here really doesn't mean that it's this exact file, but it's a key to that file. And by following that convention we can find our location. Now let's go ahead and fire up our app just to make sure that what we have is still working and walk through that whole process. So my app started up now. Everything looks fine. I'm going to go ahead and pull up our page here. We're going to go to local host. 8080 fitness tracker and let's go to add minutes dot HTML. And our page pulls up. Now, right now, this page isn't super exciting. We can go ahead and enter in that we did 13 minutes worth of workout today and you can see right here in our console that, yep, it said we did 13 minutes and it walked clear through our page here. Now I want to show you two types of chaining that you can do. We can do a basic chaining -- let's go ahead and copy this. And we're going to say add more minutes. We need to change our URL that it's looking for up above here. Right here. And change our method signature. And let's just change our comment so that we know that we actually hit this. Go ahead and save this. Now right now we have two methods that are just called different names that are doing the same thing. I realize this example is a little contrived but this is the main point that I want to show you here. So let's do forward, colon, add more minutes, just like this guy right here. Okay, go ahead and save it. Now what do you think is going to happen for a minute here? I'm going to go ahead and bounce my server and we'll pull this page up. Okay, the server's restarted. Let's go ahead and get back in the page here. Let's change this to 14 -- we worked 14 minutes today -- I worked out for 14 minutes. In our exercise -- oh, it broke. But where did it break? It actually went through and it said that we exercised to start for 14 minutes but it went to forward it broke. Well the reason why it gave us a 404 error is that it doesn't know what add more minutes is because it needs to be add more minutes dot HTML. Now I purposely made this break to show you an example here. This is going outside of our framework and coming back in. So if I go look at our web dot XML, when we set up our web dot XML, it is looking for a request of dot HTML so our app actually knows nothing to do with that because our old request was just going to add more minutes. It wasn't going to add more minutes dot HTML. So I've gone ahead and added dot HTML to that, saved it. My server restarted. Now let's go ahead and back up and hit submit again and see what happens. And you can see that we exercised for 14 and we were exercising for 14. So it went in one method outside of our framework and came back in and hit that method again. So by including this forward colon on the front of it, it actually bypassed our view resolver that first time and went back through as a request and then went to our view resolver on this time. Now we can also do redirect. And to redirect we just simply do that. Now what do you think our value is going to be down here? Let's go ahead and bounce that server real quick. It was actually redeploying as I went to do it but I feel a little bit more safer doing it on my own. And let's change this to 16 minutes. And notice we're still on the add minutes HTML. So let's hit enter exercise here and it went through as 16 and then came back as zero. And that's because on the redirect, it had closed our request and created a new request. So we went through once with our original request, it finished that request and came in as another request and so it had lost its exercise information. So the redirect was doing what it said it should do. And why would this be useful to you? A couple of reasons. If you've ever been on a website where you were using a checkout and it said make sure they don't hit the back button or those types of things so that the credit card didn't get charged twice, this is one of those mechanisms you can use to help mitigate that risk by stopping them from backing up because it's already done a redirect. They're actually a URL sitting on a different page on a different browser -- not a different browser but a different location -- that helps guarantee that there -- it helps -- doesn't guarantee but helps make sure that they don't resubmit that page as easily.

Demo: Resolving Static Files

So I wanted to take a minute and show you one more example of resolving views in Spring MVC, specifically static views. So everything we've talked about up to this point has been dynamic views and how to resolve JSP pages and that type of thing. But what if you have PDF files or images inside your application that you just want to host up statically? Do we create another controller for those? I've seen some pretty creative ideas about how people solve those but there's actually a very handy, very easy to use method inside of Spring MVC for displaying static data. So let's go ahead and comment out what we were doing here. And we just simply can highlight that and hit control forward slash and we can do a block comment on it. And let's get rid of our redirect. And we're back to where we were. So let's go ahead and open up our ServletConfig. And we're going to add what's called a resources tag in here. So if we go MVC, you can see the resources right down here and we'll create a location which is just where it's located inside of our app called PDFs. Let's host up a PDF. And then the mapping for that -- let's say for it to look for any PDFs underneath this directory and it can recur. So that double asterisks means it can look for any file and recurs from there. Now we can also have multiple mappings in here but I'm just going to show you the default to get it up and running. Now one thing that we have to keep in mind is that our web dot XML needs to also point to PDFs. Right now, we've told it to only send requests for asterisks dot HTML through our app. So let's go ahead and copy that and paste that and change this to any PDF underneath our application. So we're going to host up all PDFs underneath our PDF directory. And we don't have a PDF directory yet. So let's go over here to source, main, web app, right click and say new folder. And we're going to type PDFs. Click finish. And I have a PDF on my desktop here that I just grabbed for this example. You can use any one you want. And there's one I just grabbed, hello world PDF. And we want to drag that into its location and click copy files and you can see it's underneath there. So source, main, web app, PDFs, hello world dot PDF and we should be ready to go now. So let's go ahead and bounce our server and if we look at our console output we can see that it said that we are going to look for PDFs. Now I put a -- it's telling me I have a suspicious URL because I opened it up to allow any request to come through here so it's saying, hey you want to take a look at this. But I wanted to keep it fairly open and easy for you for a basic example. And we can come over here to our application and we want to go to PDFs slash hello world dot PDF. If we hit enter, you can see it returns our PDF now. So this bypassed all of our going into our controller and all that and actually sets up a controller behind the scenes. It's a file resolver controller that just goes out and looks for files on your path and uses this location information here to find that file and serve it back up for you without having to go through and configure a view resolver and to set up a controller. All of that information's done for you. Really nothing more to that but, honestly, a very useful tool. I've had to do this in almost every application I've ever written. There's some static data or maybe images or that type of thing that I want to host up to my app and I can do it very easily with basically one line of XML configurations.

Summary

So let's quickly recap what we covered in this module. We learned that Spring MVC can host up different view technologies for us such as Freemarker or Velocity or JSF. Also that we're going to be looking at a demo of how to use JSON for your output and use multiple view resolvers. We also talked about JSP location and that the internal resource view resolver looks underneath our web INF directory for security purposes. And we looked at the various view resolvers that are available to us. There's a bunch of them. For the most part we used the internal resource view resolver, one of the templated tool view resolvers and then we did a quick demo on static views and viewing the static data through the XML configuration of the resources tag.

Tags

Introduction

Hello. This is Bryan Hansen from Pluralsight. In this module we're going to be looking at Tag libraries in Spring MVC. Tag libraries are used to make interacting with data easier in our pages. The Spring MVC tag libraries are written on the java standard tag library specification and are included as part of the spring web jar.

Tag Types

There are 2 tag libraries in Spring MVC, spring and spring-form. The spring tag library provides some tags for evaluating errors, setting themes, and outputting internationalized messages. The spring-form tag library is based around processing form data. Most of the form tags are a mirror of the html form tags but have binding for working with a backing object as well as validating its data.

Spring Tags

These are the tags that are part of the spring. tld. The spring tags are more about working with your application rather than processing information like the form tags are. Tags like hasBindErrors is used to see if we have returned errors from the controller or from our backing tier. The url tag is used to invoke a url and builds the path accordingly escaping characters that would normally break a request. Some of these characters might be things like spaces or special characters like the ampersand sign. Things like that that it needs to build into our url for us. So these tags will effectively escape that for us in our application.

Demo: spring:message Tag

So let's walk through using one of the spring tags, specifically the one for messages. The messages tag is used for externalizing strings out of your jsp pages or your free marker template or velocity template as well. It works for all of those. Why would we want to do this? Well, if we're ever internationalizing our pages or want to be able to edit things without having to go in and touch the jsp pages all the time, I want to pull those strings out. So go ahead and open up our application. I've got the addMinutes. jsp page here. And you can see that I've got our text here for addMinutes or for our Minutes Exercise for Today. Well, let's go ahead and pull that text out. I'm going to just go ahead and copy it right now. And go over here to source/main/resources and right click Create New File. So we're going to create a properties file called messages dot properties. Now this is just a standard java properties file that's commonly used for internationalization. I'm going to go ahead and say, goal dot text in here equals, and I'm going to paste in that Minutes Exercise for Today. Now the way that -- the way that maven works and the way that our build's set up is that anything that's inside of our source/main/resources folder will automatically get added to our class path. So if I go dive down into my target directory, you'll see a messages dot property that's getting included in my word file. So no magic taking place there. It just copies those files over for us. So now let's go back to our jsp page since we've got our messages dot properties created and I'm going to grab this goal dot text while I'm here, just copy that. Go back to our addMinutes page. We need to add a tag library for the spring tag. We've got the form tag from our earlier demo. Let's go ahead and say, percent at and we're going to do taglib prefix equals spring. Now you can use whatever prefix you want. These are just the common prefixes that are associated with these tags. Let's see uri equals and we're going to do http colon forward slash forward slash www dot springframework dot org slash tags. And that's our tag library definition. Now let's go ahead and replace this addMinutes -- this Minutes Exercise for Today with the spring tag. So we're going to go "spring" and you see now we get out context sensitive help for our spring tag libraries. We're going to go down here and select springmessage. Now one of the attributes of springmessage is code which really is a key for the string that I want to pull out of that messages dot properties file. So that's going to be that goal dot text that we copied into our buffer earlier. And let's go ahead and close that tag off. Now we have our goal text in there and we have our messages dot properties. We need one other piece for this to work though. We need to create a bean that we can use for our resource bundles to look up our resource bundles. We're going to do that in our servlet-config. So let's go ahead and open up our servlet-config. You can find it. It's under our source main webapp WEB-INF config and there's servlet-config. Another way that you can find that is to do Control Shift R and open a resource. And I want to open that resource and just type "servlet-config" and you can see my files here and I can choose it. So I've got my servlet-config here. Go ahead and just give us a little white space. It can go anywhere in this file but I usually leave my mvc and my context definitions up at the top of the file just by habit. And I'll create a bean. And inside this bean, I'm going to give it an ID of messageSource. Now our messageSource is what's going to be referenced by our spring messages tags so it does need to be named messageSource for this. And I'm going to give it the class of "org dot springframework" and I can use context sensitive help here. Org dot springframework -- oops. I copied over the rest of my files here. Let's go back in here. "dot springframework dot context dot support dot ResourceBundLeMessageSource. " So let me make this full screen so you can see the full url there. And this has a property associated with it of "basename". So I choose basename there and I choose "messages" here. Now the basename, here messages, ties to what we named our properties file up here. So you see "message dot properties". That messages there is referring to the name of this properties file here, messages dot properties. Not super crucial where we're not dealing with other languages yet. We're going to here -- I'm going to show you more advanced example in a little bit that ties those two together. So now we've got our resource bundle to find. Go ahead and save this file. I'm going to double click to get back out of that. Now I'm going to start my server up here. Okay. Our server's started. And let's go down here and pull up our page. Go to localhost FitnessTracker. And let's go to addMinutes dot html. And our page displays. Okay. Let's really prove that it did actually work. Because this was the text that we had in there before, we can go out here, open up our messages dot properties and change this to Minutes Exercise for the Day Today. It's a little redundant but just to make sure that you see that it's working. Now eclipse is set up to autopublish. You'll see I waited a second and it's finished reloading. In fact, it says right here that it's "reloading the context with name FitnessTracker. " So it's done. I could have bounced my server manually over here. You'll see me go back and forth from time to time doing that. But it says it's reloaded that context. Let me come out here to our jsp page. Click Refresh and you'll see our text is now changed. So now we've started pulling back those messages out of a file and it's not just hard coded inside our application.

Interceptors

Interceptors really aren't part of the tag libraries but they're commonly used with tag libraries. Interceptors are registered as part of the request lifecycle and usually involved with intercepting data that comes from our jsp page and is going to our controller. They have the ability to pre-handle and post-handle our web requests. So they can intercept things on the way to our controller and on the way back from our controller. They have callback methods that are used to override or change values. It's not typical that you have to create your own interceptor. They're a little bit more of a edge case, although it's nice to have the capability to do so. There are some canned interceptors though that are used quite frequently inside your application. And one of those is commonly used for Locale Changing and that's exactly what this example here is is a internationalization locale changer. So this bean here is used when you want to intercept a request to change the locale that our web application is using for things like our messages bundle.

Demo: Interceptors

So I have the addMinutes page just as we left it from our previous example where we had externalized this string and put it in a resource bundle. Now let's go ahead and open up our application again and take a look at it. Usually we externalize things when we want to internationalize them. Let's do just that using an interceptor. I want to take and give ourselves some white space in our addMinutes. jsp page here. And put in "Language" and colon. I'm just going to put a simple href on here. "href equals" I'm going to do question mark Language equals en. Now what this is going to do is just build a link on our page that attaches language en to the end of the url we're currently on. And we're going to say this is English. English. And we're going to put a pipe here and just do a href equals, and we'll do dollar sign Language equals es for Spanish. Let's close this off and put Spanish in there. Now if we look at our page which we can come out here and refresh. Now we'll see English and Spanish on here. Right now they're not going to do anything; all right. But it added the link on there. You can see that it's appending that on the end of our url. Now let's go inside of our application here. And we need to do a couple of things. So right now we have our default messages dot properties. We need to go ahead and create the Spanish messages dot properties so what we're going to do is right click on source/main/resources and say New, File. And we're going to do messages es dot properties. Now make sure that, one, you've got the right location and, two, that it's messages. It's got to be spelled exactly the same as the other one but with underscore es on the end of it. I've made this mistake a few times myself where I've forgotten this. Let's click Finish, and paste that in there. Now in here we need to put our Spanish translation. So I'm going to grab our text here, copy that. And I'm going to cheat and go out to Google, and google translate and just paste our text in there, and there it is in Spanish. Who says I can't speak Spanish? Copy that. Go into our messages es properties and paste that translation in there, and save it. So now I've got our translated properties file. Now we need to add our interceptor. So let's go into our servlet-config. I'm going to make this full screen so you can see everything. We have our messageSource that we had created in our previous examples just to load those ResourceBundLes and pull that text out of ResourceBundLes. Now we need to add two things. We need to add a bean that is just a locale resolver. And this bean is simply a -- is simply used by spring to know which current selected locale we have. So I'm just going to call this LocaleResolver. And the class for this is -- oops. Org dot springframework dot web dot servlet dot i18n dot SessionLocaleResolver. And we're going to set it as the default locale is English. Okay. So that's what our LocaleResolver looks like. It's just this one line here and that just says that, hey, go ahead and look in our session for the currently selected locale that we have. Now the next thing we need to do is register an interceptor. And to register that interceptor they actually created a nice way of doing this inside of spring 3 that they created an xml element called mvc interceptors and the namespace, and the element and the namespace mvc. And inside of this we need to create another bean. And this bean is simply a class and it is org dot springframework -- oh, framework dot web dot servlet dot i18n dot LocaleChangeInterceptor. And this has a parameter associated with it and that is paramName. And that paramName is language. Now you may not remember language here is the paramName we told it to tie to here. So this language here is what this is intercepting right here. All right. So there's the two pieces that we had to add to the servlet-config. We had to register an interceptor and we had to set up a locale resolver so it would look and know where to look for what locale we're currently using. So now let's escape back out of that and go ahead and start our server up. Make sure you've saved all your changes. Wait for this to restart. Everything looks good there. And let's pull our page up. Exit out of that. Refresh it. There's English. There's our Spanish translation. Back to English; back to Spanish. And that's an interceptor that's intercepting the language that we're passing through and binding that to our page.

Form Tags

So these are some of the spring-form tags. The form tags are all about working with html forms. Their names typically mirror the html options that they are meant to be representing. So why do we want to use these over the standard html elements? That's a good question. These are meant to bind to objects from our model as well as carry error information on required fills. They also take a CSS class for changing their display from normal to error status. And they're also javascriptive attributes and other things that you can tie to as well. Some of the events like on focus, on click, key press events. Those types of things. So really it's about how we want to style our page, where we want to get the data from, is there an error associated with this. So there's really some nice convenience methods that will help reduce the amount of code that we have in our jsp pages.

Demo: Adding Goal Functionality

So now that we have our application all internationalized, well, we've got it set up to internationalize our strings anyways because we haven't done them for all of our fields in our application but you get the idea. So let's go ahead and tie up some loose ends of our application to make it more of a true application. So we've kind of cobbled pieces together so far to show functionality to work. So let's make this act more like an application. One of the things we haven't added is the functionality for a goal, and adding our goal for the day and how much we want to achieve in that day. Let's switch back over to eclipse and open this up. We're going to add a page called addGoal dot jsp that's going to be very similar to our addMinutes. jsp page here. And we're going to use it to review a little bit of the tags and really this whole exercise we're going to do here is going to be somewhat of a review of where we're at up to this point. So let's go ahead and right click on our application and say New, Other, and we want a jsp page. And make sure you select the correct directory. It's defaulting to the wrong one. Let's go to source, main, webapp, WEB-INF, jsp. Let's change our jsp page down here to addGoal dot jsp. Okay. We've got our addGoal page here. Now let's just add some text in here for Add Goal. We could go through and internationalize all these and use our message dot properties. But just for the sake of time, I'm not going to go through and do that in this exercise. Now let's add a -- a tag library up here for the spring-form tag library. So let's do, percent at taglib, and do, prefix equals to form, and uri equals http colon forward slash forward slash www dot springframework dot org forward slash tags and forward slash form. Okay. Now let's go down to our body here and create a form tag. And in the form tag, we want to do form -- now if you remember from our earlier discussions that form takes a commandName and really what that is is a backing object. So we're going to put "goal" in here now. In theory this should tie to a goal object which we have not created yet. We're going to create that here in a minute. So let's create a table here. And just some basic html real quick. ( Pause ) And put some tds inside of here. ( Pause ) Just to get a basic column structure set up. ( Pause ) Okay. Now inside of here -- let's also set the colspan to 2 really quick. Okay. So in this first td, let's just put in Enter Minutes. And in this next one, let's do a form input. Now if you remember from some of the discussions of tags earlier, this form input is almost the same thing as the html for just doing a basic input tag but these are tied to fields on this object. So this path that I'm putting in here, minutes, saying that there's a method inside of goal named minutes. So there's on goal, there's a getter and setter named minutes. So that's what those 2 are looking for. So now let's go down to this td and just create a simple input and make it a type, submit, and for the value, let's make that equal to Enter Goal Minutes. So there's our basic jsp page. Now we need a controller and a goal object for this to submit to. So let's start off with a goal object. Let's right click on source/main/java and say -- actually you know what? Let's click on model because this will put it in the right package structure for us. Right click on com. Pluralsight. model and say New, Class. And notice up above here that it puts it in the right package for us and let's just create Goal. And inside of Goal we're going to create a new field here. We're going to do private int minutes. And let's right click on here and say Source, Generate Getters and Setters, Select All, OK. Go ahead and save that. Now the next thing we need is a controller. And let's right click on controller and do New, Class. And do GoalController. Click Finish here. I'm going to make this full screen because we're going to do quite a few things in here. Now I'm going to introduce 2 new concepts to you in this controller. So first, let's add our controller annotation. And we can hit Control Spacebar, it will bring up the box for us to import the import. Come down here and create a method, public String addGoal and we're going to do something a little different this time. We're going to pass in the Model object and it's a type org dot springframework dot ui. All right. Now what this does is spring will pass in the backing model to your method and since it's passed by value -- passed by reference instead of passed by value, this is a pointer back to our actual object. So it's not going to be -- it's not just going to go away at the end of this method because it's going to be absorbed back into the framework as part of the interceptor as part of the requestLifecycle. So this object is available for me to use -- it will actually return data back to our jsp page for us. And I'll show you how this works here in just a second. So let's do model dot addAttribute. And we're going to pass in a goal. And let's create -- let's just create a new Goal here. Okay? Now if we return this, addGoal, this right now is almost identical to what we had going on in our addMinutes page. But we need to hit Control Shift O to bring in that Goal import -- I forgot to do that above there. So if you do Control Spacebar when you're adding an object, it will import it. Or if you do Control Shift O, it will say organize import so it will import all of your -- it will clean up all of your imports, get rid of unused ones, and get rid of the dot asterisks and that type of thing. And so now we have our addGoal method here. Well, we need to add our request mapping. Add RequestMapping so that it knows what method is going to handle this url. We're going to do something different on this one though. We're going to tie in the value just like we had used on the old MinutesController. In fact, I'll pull these side by side just so you can see them. So here's my MinutesController from before where we were adding a value in there. But now I'm going to come over here in my GoalController and I'm going to also put value equals addGoal. But I'm also going to put method in there. Method equals RequestMethod dot GET. All right. So what that says that this method is just going to handle http requests of GET so the http request GET is all that this method is going to handle. So if I try to post to it, it's actually going to error out. And the reason we want to do that is as we're setting up our page, we only want certain things to go to this method. And I'll show you exactly what I'm talking about. So we've got our addGoal. jsp and our GoalController here. Let's go ahead and fire that up and see what it looks like. ( Pause ) There we go. It was stuck in full screen mode for some reason. So let's bounce our server. Okay. And our server's up. Now let's go ahead and pull up our page here. And we want to go to addGoal dot html. So our page pulls up fine. If I enter in 30 minutes here, Enter Goal -- oh, we posted to it. So it failed on us. Now why would we want to do this? Let's get rid of the GET here for just a second. Go back to how we'd done it on the controller. And we'll clean up our imports. Save it. I'll bounce my server. Now it's restarted. Let's go ahead and resubmit to this page. And if I enter in 20 here, it's not doing anything. It's just killing the minutes that I have entered because I am creating a new Goal every time. Well, that's good and bad. We haven't talked about how we want to save things. So what if I want to store something in my session? There's a really easy way to do this inside of Spring MVC. And I come up here and add an annotation for at SessionAttributes, and I just tell it which attribute I want to store my session so we're going to put goal in there. And this goal right here ties exactly to this goal right here so these 2 goals is saying, hey, when I put this in my model, I want it to be stored with a scope of session. Okay. Now let's go ahead and add our method equal to RequestMethod back in here, RequestMethodGET, dot GET and save this. But before we go back and run this and see the lifecycle that takes place, now we've got -- we want to store this object in the session. Well, we don't want to sit there and keep overwriting ourself so let's add a method to post to. At RequestMapping, and we're going to give it the same value. So its value is going to be equal to addGoal but the method is going to be of type post. So RequestMethod -- RequestMethod dot POST. And we're going to say public String updateGoal. And in this one, we're going to do a little -- we're going to do something a little different. We're going to set up kind of how we did the addMinutes before in our MinuteController and we're going to say at ModelAttribute -- attribute. And we're going to say that we want the goal object -- oops, I need to have double quotes around that. The goal object, Goal goal. And now we have our goal in here. So if we do a system dot out dot print in here, we'll be able to see goal dot getMinutes. Not the best debugging but you'll get the idea. It's updated. We'll save that. Put a semi colon on the end here. Now where are we going to return this to though? We have to return a string. Where do we want this view to go to? Well, we don't want to just keep submitting back to ourself. We want to eventually get some navigation built into our application so actually let's start doing that now. Let's do a redirect, and let's go to our addMinutes dot html. So once we've added our goal, once we've requested the page -- so the first request is coming in to display the page and it comes in with a goal initialized. And maybe we want to -- maybe we want to change this even to say -- let set a default goal of 10 minutes. So let's cut this out of here and say Goal goal equals new Goal; and we'll say goal dot set minutes. We'll put 10 minutes in there and then we'll come out here and set the goal object there. So we'll come out. We're going to set a default of 10 -- a 10 minute goal and we're going to send it to the addGoal page. Once they go from the old addGoal page and they submit back to this page, it's going to update their goal and then we want to redirect them to the addMinutes page. So let's go ahead and save this and bounce our server here. Give it a second to start back up. And as soon as it started up, go back in here, let's reload our page, just for giggles I should come back, session attribute goal -- ah, it was trying to resubmit my page. So you saw that error there. It was trying to resubmit my page because I hadn't done it from the goal. So when I click the url again, you see we've got our 10 minutes here. So let's update that to, say our goal for the day to exercise is 45 minutes. Let's go ahead and enter that goal and down here in our system out print menu, you should see that our minutes updated to 45 and so our minutes updated printed out right there 45 and then it directed us to our addMinutes page. Well just to verify that it saved that in the session, let's go to our addMinutes dot jsp page and let's do down to the bottom of it and let's just add a h1. And we can use e, l to grab things out of our session. So let's do a dollar sign, curly brace and just do goal dot minutes. And let's get a little prettier name. Our goal for the day is that. Save that. And let's go ahead and resubmit our page. ( Pause ) And you can see our goal is 45 minutes. So it's pulled that value out of our session and redisplayed it for us there. All in all pretty easy to tie these together. You see the redirect out of our page so, you know, our addGoal -- the first time we request it, it comes into our goalController and it returns us to addGoal which goes back and displays this page, binds us to this object, we submit back up to that so we actually do a post back up to our controller and it will bind us to the goal object that we created. Once it's bound and it's updated everything, it redirects us back to addMinutes dot html. There's our addMinutes. jsp page and we're ready to start adding in our minutes for the day. And so now you can see that we've got our goalMinutes and it's pulling that out of our session.

Summary

So just to recap what we did in this module. We went through and looked at the Spring Tags and saw how they are useful for things like the message resource bundle and internationalization and that type of stuff. We didn't go into the error handling yet because that's another module we're actually going to go through next. We saw the Form Tags and kind of reviewed what we had done with the form tags earlier in our basic example. And we also configured an interceptor. Now the interceptor doesn't necessarily tie to the tags but we commonly use the tags in the internationalization to tie in with our interceptors so it was a good point to introduce that. And then we completely set up the internationalization request response lifecycle so that our application is all internationalized. We had a couple extra demos in there kind of tying everything in there and getting our controllers for our goals and that type of stuff and introduce how to also store things in the session as well.

Validation

Introduction

Bryan Hansen: Hello, this is Bryan Hansen from Pluralsight. In this module we're going to be looking at Validation and Spring MVC. Using the tag libraries and the model object in Spring MVC, you can easily validate data being passed to your controller in a protomatic way.

Validation

There are basically two types of validation that your app needs to be concerned about; constraint validation and business logic validation. Constraint validation is when you are expecting a field like first name or last name to not be null. Fields like password and re-enter password must match and have letters and numbers, those types of requirements. The other type of validation is when we are working with business logic. Null fields aren't necessarily business logic, but rather constraints and these are what Spring MVC has built-in mechanisms to help us validate against. The other type of validation on business logic should really be handled in your service tier and not the controller. This type of validation is guaranteeing things like an entity is valid in the database or a user name isn't already taken in our LDAP server or our database or whatever other security store we might be using.

Tags and Handling Errors

So when we talked about tags earlier we really didn't talk about error handling with tags. To recap the tags, all form tags have an error class for setting a specific color, font or combination of both for displaying errors. There is also a specific errors tag for displaying errors based on a key and you can see in this example I have here, we are using the errors tag to look at the path first name, which the path first name is actually the member variable inside of our object that we're bound to right now to display any errors that might have occurred on first name. So, here we have the error tags, it's tied to our object and anything with first name -- and you can see we've done the same thing with last name below as well.

Validator Interface

There are currently two ways to handle validation in Spring MVC. The first way is to use the validator interface and the validation utils helper class. This is an older way of doing things and is used with the simple form controller and not really dealing with annotations very much. So there's a validator interface that we have to go implement and we have to code all of this logic by our self in our implementation of this interface. Now there is this helper class called validation utils that is set out to help us go through and look for whether a field is empty or if it just contains white space, or things like that, if these fields match. Say we're comparing a password field or an email field to make sure that they've entered the email twice correctly; it can compare these types of things and it does this using the binding result class. So it's actually a pretty simple mechanism and you can actually still use it in your service tier to do things outside of Spring MVC. It does use the simple form controller, which is kind of a pre-annotation way of setting up controllers and it's kind of a narrow focus controller. The simple form controller is just dealing with forms and kind of a single post to that form. So I can't do multiple things inside this controller. Not really commonly used anymore, especially not with the ease of annotation. So I wanted to point this out because you will see examples of validator and the validator interface out there, but those are falling short now to the new JSR specification that's out there for validation.

JSR-303

JSR-303; JSR is a specification that has been established to do validation. This is a Java standard and not a Spring standard or a Spring API. Spring MVC just integrates with JSR-303 for its validation process and reporting errors. The nice thing about this is we can use this in any application or across multiple tiers. It's annotation based and can be extended to use our own custom validation rules as well, so we can use the JSR specification to build our own JSR-303 compliant rules and have those implemented inside of our actions as well. The reference and limitation is actually called Hibernate Validation. This is often confusing to people because the hibernate validator has nothing to do with Hibernate, the ORM tool. So you know, typically when we hear hibernate we're thinking database and JPA, and object relational mapping, and binding data to a database. Hibernate validation, it kind of started as a part of the ORM tool but they separated out to a completely, its own separate specification, has nothing to do with ORM other than the hibernate ORM tool can use the JSR-303 defined objects and rules to validate on your code going against the back end, but it has nothing to do with Hibernate. Since it is annotation based that means it works with POJO, so all of our stuff is done on POJO. So there's no weird interfaces that we have to implement or things that we have to bind to, to get these rules to be applied.

Demo: JSR-303 Validation

So when we finished up our last demo we had added this add goal functionality in there that allowed us to set our goal for the day for how much we wanted to exercise and as it stands right now there's no validation in here, so I can come in and actually add negative one for my goal and submit it and it will allow me to enter in a negative range. Well, clearly we don't want to allow that to happen. So let's go ahead and open up our application, we're going to add JSR-303 validation to our app. So, we open up our palm file -- let me go ahead and just who you where that is to make sure you -- in case you didn't already have it open. At the bottom of our project we can open up our palm file here (typing) and go over to the palm. xml, tab down here, scroll the bottom and we need to add a dependency. So we're going to add a dependence for Maven to download our hibernate validator. We want to add a group Id, org. hibernate and we want to add an artifact Id, (typing) of hibernate dash validator. Last, we need to add a version of 4. 2. 0. Final and it has to be spelled just like this because it is case sensitive. So the F does have to be capitalized. Now, just to make sure that this change does get imported into our application I am going to right click on Fitness Tracker and go down to Maven, and say update project and it will have Fitness Tracker selected there and I say, okay. And I can scroll down in my Maven dependencies here and -- so in my application there's a folder called Maven dependencies -- and I can see that validation API 100 GAjar was included and hibernate validator 4. 2. 0. Final was included. These are the jars that get imported by adding this dependency. In fact, the validation API is a transitive dependency and you'll notice in its path there that its Java X validation, so this is where it is a Java specification. So we can go ahead and close the palm. xml, that actually is all we'll need to do in that file. And, let's go to our goal now. So if I go up to our com Pluralsight model goal, let's add range on minutes. I can do at range -- if I hit control spacebar it will automatically do the import for me or I can do control shift O and it will organize my imports, either way will work. And I can do min equals 1 and max equals 120, and I can save that. And that's actually what I need to do, that's all I need to do to put a range on this field and have it have a range a min of 1 and a max of 120 assigned to this field. So let's go to our controller and we need to do a couple of things inside of our controller to say hey, you know, I want my goal to be a valid goal that comes in. Now, if you remember the last demo we did we added a method (typing) of request method. get on here saying that this method here will only accept a get request. And that's when we're setting up the page. So, kind of our initialization method or however you want to look at it, but that's when we're requesting it, so we actually don't care if we have a valid goal there because we're creating the goal and returning that to be displayed on the page. Remember we put a default value of 10 in here. This method down here is where we're posting back because it takes request method type of post that we want a valid one submitted back to us. We want to do two things here; we want to say at valid and I want to hit control spacebar (typing) and import that. Give it some space there. Now, that will actually check to see if the goal that I am setting in is valid or not, but we do need to have something to check the result of that binding. So I come over here to the end of our signature and we put a comma and do binding result -- oops I don't know why it did that for me -- result, result shift O and save that. Now, this object, this object we named result will actually contain the response from validation off of this guy here. So when this method gets called and it goes to see if its valid, this object will contain whether or not it is valid and has any errors. So to show you how this works, let's go ahead and just put system. outprintln here and do result has errors (typing), and do plus result. haserrors. Semi colon there. Let's go ahead and start this up though your server may already be running -- I'm going to bounce mine just to be sure and everything looks good there. And let's go ahead and go to our page here and go to add goal. And let's hit negative two and enter that for our goal. It still let us enter in negative two. Well, why? That seems a little strange. We go back in our code, you'll see that it did a system out println and said it did have errors in there. The thing is we're not doing anything with the errors yet, so we need to process it. We're still saying that whether or not you have errors, go ahead and send them to the add minutes page. So we actually need to come down here and put just a little bit of logic in our controller and it's not business logic, its routing or navigation logic. So remember, our controller is responsible for selecting the correct view based off what's happened. We want to do if result. haserrors and return add goal. So we want to send them back to add goal if we don't have a valid page -- a valid object that they've submitted into. We want to submit them back to their -- to the page that they came from. So let's go ahead and -- it says its reloaded it so we should be fine. And let's go ahead and go back to our page here, and we're on our add goal, and if we hit enter goal it's just resubmitting the page, but now we haven't done anything that shows us what our error message was or that we had an error. We know we have an error because we put the logic in there to navigate us back to this page, but let's go ahead and add something in there first and now see what error message we had or why -- you know, why things were broken. So we're going to go over to the add goal page, and inside of here we're going to add a couple of things. We're going to go up in the head tag first, so let's go right below title and head, and add a style section. And it is CSS, but we can clean this out here, it'll work just fine. And let's do an error class, so we'll say. error (typing), and we'll give it a color of, oh let's do pound FF0000. And then let's do an error block -- and these are classes that the form tags are going to use here in a minute will all use. Let's give it a color of pound 000 and let's give it a background color of, oh let's go with pound FFEEEE and a border of 3 pix solid and let's make its color FF0000, and let's get some padding (typing) and we'll do that 8 pix, and last, let's give it a margin and we'll give it 16 pix. So whatever, just some standard error kind of output and you'll get to see what this displays and you can tweak these to whatever you want. Let's go ahead and save that. and you could put these in a global CSS that you just import in this. It doesn't have to be defined per page or anything like that. Now, let's go down to our form section and we're going to add an error, the errors tag that we've seen in a couple of pages now. But we're going to add one at the top of our form. So let's say form errors and we'll put in here path equals asterisks. Now, this does a couple of things. By putting asterisks in here it's saying I want all of the errors. So if there's any error, I want to display this section right here. And we're going to give this a CSS class of error block and that's the one that we just defined right up here. So error block and we're going to do an element equals div. So let's close that off. If I put the closing tag there it automatically gets rid of the trailing element. And now let's go down and we're going to change a little bit of, a little bit of html down here. So let's just copy this line real quick, saves us some typing. Paste that down here and do form -- we're going to do errors here, path is still going to stay minutes, but then we're going to do a CSS class equal to error. And this was the first CSS class that we built up here, so we have error and error blocks. We're going to use error down here. And let's go down and change this to a call span of three because we added this extra, this extra column here. So let's save that, let's bounce our server down here (silence) and lets go back to our page now. So if we refresh this -- let's cancel. If I hit submit on this it should send me back to my control and there's an error in which you get an error message back (silence). Expected session attribute goal, that's because I had data in here already. Let's go ahead and just hit enter on this and it should refresh my page. Yeah, it did. So, that error that you just got was because it resubmitted the page. Let's go ahead and enter negative five in our goal -- and you can see our error up here. And you see there's an error message that says must be between 1 and 20. Well, that's not really a great error message. Let's clean that up a little bit. Let's go back over to the clips here, to Spring SCS and open up our messages. properties. And if I come down here and do -- see how this says range here, we're using range and it's on the field minutes -- if I do it's -- it's in the class goals. We have a range in class goal on the field minutes. I can come inside of here and say range. goal. minutes, so range was the validation, goal was the object that we were in and minutes was the field that we were worried about. And I can say equals goal must be in the range 1 and 120. Save this, go ahead and bounce that again. Now let's come back over to our page and lets hit enter again on it and again I've -- you think I'd learn to just quit refreshing it. Let's hit negative five here, enter, and its updated our error message. Goal must be in the range of 1 and 120. And you'll notice it's also dumped it out here along the side of it because we have that text setup in our, in our JSP page that is taking and grabbing that error message and displaying it here as well. So, that form errors tag is what is dumping this message out right here. We can even do one more thing if we really want to be that blatant about it. The input tag also has a CSS error class and we can come in here and say that it also displays error and save that (typing). Let's grab this page again and do negative 12, whatever, and you'll see that it makes this text red as well. So we can put as much focus to this field as we want as far as defining it or maybe you might feel that this is a little too intrusive over here on the side, or maybe redundant with this up here. That we don't need these two and just change the text to red. So, I can do quite a few interesting things as far as calling their attention to it and it's all basically free inside of my application by just adding these form tags. And it should also go without saying that this message that we put in here for the range, I can do the same thing and put it in Spanish over in the Spanish field if we were, if we wanted to go through and translate this we'd have the same error message in Spanish. So it can be all of your error messages are internationalized and we get it by basically going through our object, adding a range on here and going in our controller and saying that's valid. And we do want to grab the binding result so that we can put some logic about where the page should go based on those errors. Just to show you that it does work too -- I won't hold you in here. We can go through and say our goal is 60 and enter, and it will take us to the correct page.

Summary

So in this module we talked about validation and how do we use tags to display validation errors, and we mentioned the old method of doing this using the validator interface. We didn't go over an example of it because it is kind of an outdated methodology, but it's one worth pointing out because you will see a lot of older examples out there prior to Spring 3. 0 that are using it. We also went over JSR-303 and the new hibernate validator, and we did a longer demo of adding validation, input validation to our add goal page using the JSR-303 specification. ( Silence )

Ajax

Introduction

Hello this is Bryan Hanson from Pluralsight. In this module we're going to be looking at Ajax and producing RESTful services using Spring MVC. Spring MVC has the ability to produce different content based off of our request type. If done correctly the same application can produce REST without having to change anything else to be utilized within our application. Rather than do a large demo we're going to do multiple small demos in this module just to help break it up and learn the smaller pieces as we go.

REST

Restful services have taken the SOA world by storm. They're easier to work with than soap and are more easily consumable across applications. Since they are built on top of the http specification and can produce JSON they are a great match for JavaScript Mashups. The Spring MVC framework allows us to return JSON just as easily as html by simply using a different view resolver. This makes it even nicer because we don't have to use a different framework like jersey or axis or cxf just because we want to expose a restful service. the rest services of Spring MVC are not as feature rich as jersey but support most of the features that Jersey has and don't requires a separate tool or framework.

Verbs

Restful services are all about what we can do against a domain object. The verbs or actions are the interface that we can interact with against this domain object. When talking about the domain or data base we commonly refer to CRUD or CRUD functions. CRUD stands for create, read, update and delete. So those are the four functions that we typically do against a data base. REST was built with the same notion in mind. Post is the same as create. We do a post when we want to create an object through a RESTFUL service. GET is synonymous with read. Parameters can be passed on a URL or part of the path of the url. Put is supposed to be used with updates but is usually overridden and sent using a post instead. Basically post is used for both create and update. Delete is obviously used for deleting.

ContentNegotiatingViewResolver

The content negotiating view resolver is what makes it possible to return various content types from our application based off of the request. The content negotiating view resolver uses the except header that is sent using http headers to determine the content that it will host up. It can also look at the file extension that is sent as well. This can be combined with other view resolvers, they just need to have an order that is associated with them. So we already have created an internal resource view resolver with us that's used to host up our jfp pages. We can also return xml and JSON as part of this view order, just as long as it's ordered before our internal resource view resolver. There are a few additional jars that we will need to have added to our project for the various output types that we want within our application. We will be using Jackson to martial our data to JSON.

Demo Maven

Just to quickly recap where we left off from our last demo, we had added the GO functionality to our application to have it validate and add things to our session, just to show us storing something in our session. We pull up our add-go page here, local host, pull an 8080 FitNesse tracker, addgo. html and we can enter in some time, we have this default value of 10 minutes in there. Click enter go and you can see our go for the day is 10 minutes. So we can go ahead and in our exercise here and it's -- everything is saving, it's doing what it's supposed to be doing. You can see from our output here that we had our goal entered and our time entered here and we're good to go. Let's go ahead and add our maven dependencies for doing the content negotiating view stuff. So let's stop our server here, just so it's not trying to read a play as we're in the middle of adding things to our pomfile and scroll down to the bottom of our app, double click on our pom that X amount. And I'm going to show you how to add dependencies, I'm going to show you the old way first and then I'm going to show you another easier way if you have the M2Eclips plug in or you're using SpringSTS. I'm going to double click here and open up our pom just to full screen so you can see it. Go to the bottom of our dependencies here and make sure you stay within the dependencies tag but below the last dependency and I'm going to click add dependency and I need to give it a group ID and I'm going to put org. codehaus. jackson and then I want to give it an artifact ID and I want to put in here Jackson-mapper-asl and the last thing I need is the version and our version we're just going to use 1. 4. 1. Now, I've looked these all up before hand. There's various websites. I've got a maven tutorial out there on Pluralsight if you want to go to that maven course you can see how I do all of this stuff in there but I've looked up these dependencies before hand about what we need. So we've got the codehaus. jackson artifact in our application now. Let me show you another way you can add these too. Down here in this dependencies tab and click on this and say add and it will bring up this nice UI for me and I can enter in my group ID, I'm going to do com. fotworks. xtreme and for the artifact I'm going to do xtreme and the version 1. 3. 1 and this will add it in there exactly the same as we were doing it manually. So there's our xtreme plug in and if I switch back over to the source view you can see its right there below us. So just two ways that you can easily add these things in there. There's one other dependency that we need to have in there too. I'll just enter it the old way here. Dependency and group ID and that is org. springframwork and the artifact ID is spring-oxm. And that stands for object xml mapper. So it's a tool that xtreme uses when it's marshaling stuff out to xml. Now we're not actually going to be doing xml in our demos but I wanted to set this up so you can see both of them. It's only these two more lines and then we're ready to go. And for the version we want this to be typically the same as our springframework version that we're using up here, in fact I'll just copy and paste it from here to there. They're not always in synch but usually they are. So there's all of our dependencies. Now we can check to make sure that Eclipse or SpringSTS is set up for us by looking at our Maven dependencies over here and seeing if the ones that we just added in were included. And there is spring-oxm, there's xtreme and you can see jacksonmapper and it's pulled in the cores its transient dependencies as well. If for some reason yours didn't upload, every now and then you need to come up here and right click on your projects and go to Maven and say update project and just make sure it's selected and hit okay and that will make sure that its got all the dependencies downloaded in your class path up to date. And that's all we need to do to get those correct dependencies in there for configuring this.

Demo: ContentNegotiatingViewResolver Configuration

So now that we have our dependencies configured, let's go ahead and open up our servlet Config. I've already got mine open here but in case you have forgotten where it's at you need to go into source, main, webap, webinf, config and double click on servlet Config. I'm going to double click and open mine up to full screen so you can see this and we're going to do a little bit of typing here, this is one of the things that I don't like about this view and there's quite a few things that you have to get in place to get this to work right. First I'm going to start off by going down to our internal resource view to resolver here and right now we have a order on our bean name viewresolver and our internal resource viewresolver is zero and one. I'm actually going to switch the internal resources viewresolver to two. Now there's a bug, they don't claim it's a bug but there was a configuration setting that your internal resources viewresolver had to be the last viewresolver that gets called. It's in being voted to be fixed but until it is I'm just going to make sure that your demo works by switching that to two. Now, let's go up here and go above our MVC interceptors and give ourself some white space. There is a lot of typing in this one so please bear with me on this. Let's go ahead and create a bean and inside of our bean definition we're going to create a class and the class is going to be org. springframework and I'm going to use my tab completion here, my control spacebar, my intellisense. web. servelet. view. content negotiatingviewresolver. Okay, and inside this viewresolver we have some properties we need to set. The first property we're going to set is the order. And we're going to set this to the value of one. Now if you remember just below there we set, we changed the internal resource viewresolver to be 2 so the first one is the bean name viewresolver and the second one will be our content negotiating viewresolver and the last one will be our internal resource viewresolver. The next property we're going to add is a content negotiating manager so we'll do property name and we'll have content negotiating manager and this is one reason that I do like SpringSTS is because you get this intellisense with it. Let's do a bean inside of here and the content negotiating manager just says hey, what types are we going to accept into our application and so we're going to create a new bean, and the bean type is of type content negotiating manager so we're going to do class equals org. springframework. web. accept. contentnegotiationmanager and that has a constructor-arg, now this is just constructor injection, nothing real magic going on here, and that has a bean inside of it. I warned you this had some typing. So class equals org. springframework. web. accept. pathextentionscontact negotiationstrategy. So this is just what tells it which path types we're going to accept in here. And we're going to give this a constructor-arg and inside of there it will take a map. Our maps, we have entries for the key equal to JSON so we're going to say we're going to accept JSON requests and the value of -- and this is the actual http header type. So application/JSON. And we can close that off and get rid of that trailing element there and we're going to do entry and since we're in here, even though we're not going to use it for our demo we'll just configure it just so you have it. We'll do xml and the value equals application/xml. So now we're set up to handle JSON and xml. Now there is one other thing that we do want to change in here, and that is that we want to add another property down below all this so we're still inside this bean but we're below our property here, just so you can see that correctly. And I'm going to come down here and add a new property and this property is called default views. So the default views are basically what we want to use, what framework do we want to use based off the view type that's coming through so inside of here we have a list and we're going to give this a bean as well and it has a class equal to org. springframework. web. servelet. view. jsonmappingjxon jsonview, excuse me. And then we can close this off and if you actually stop now that's every you need to do JSON. I've come this far, I'm just going to go ahead and throw the xml stuff in here as well so let's do bean, we're going to give this class equals 2 org close it off, org. springframework. web. servelet. view. xml. marshallingview and that bean has a constructor argument and this is why some people don't like xml and spring. Constructor arg and bean the class associated with this is org. springframework. oxm. xtreme. Marshaller and inside of that there's a property and the property name is autodetectannotation and the value is equal to true. If I close that off it will get rid of that trailing element. Now this is all the configuration I need. Sorry, there was a lot of typing there but this right here is what sets us up to be able to handle those views based off of JSON requests or xml requests and there's other types of requests you can do in here too but these are the two that we're focused on now. The next demo we do we'll walk through and actually show you that this is working correctly based off of our request type.

Demo: Adding Activity

Now that we have our viewresolver set up we need to actually have something to return to our view to test it. So let's get out of that full screen and let's go up and go to our model and right click on model and let's create a new class. So if I right click on model it automatically puts it in the right package for me which is com. Pluralsight. model. Let's create an activity. All right. Since this is spring and spring MVC and everything is a basic POJO, I don't have to extend any classes or any weird interfaces to get this to be exposed for us. So I want to go on here and just create a private member variable string and we'll just call it desk for a description. And then we want to right click in here and say source generate getters and setters. Select all and hit okay. And so we got a basic object now. So now let's go ahead and go to our controller, our minute's controller and actually host something up with this. So here's our minutes controller from before and I'm going to make this full screen so we've got something to view here. Let's create a new method down here and we're actually going to expose some activities to choose in a drop-down on our page. So let's do at request mapping and we want to give it a value equal to slash activities. Now remember in talking about REST everything about REST is based off of one of the CRUD functions. Since we're reading we want to use a get method associated with this. We're going to do method equals request method. get and you can do control shift O and it will automatically import your imports for your -- your imports for your organizer imports for you. You want to do public at response body. Now this is a new one, you haven't seen me use this one yet. Response body says, hey, what you return is the actual content that I want to expose. So I'm saying I am returning the contents. I'm going to do at responsibility, hit control spacebar and that will automatically import it for me. And I want to do list, activity and hit control shift o, that will import list in activity for me. And do find all activities. And we're just going to make this open parens, and closed parens so there's now arguments that it was taken in here. Control shift o can do java util list and we'll make this, we'll create a new list an activity and activities equals new array list. Let's go ahead and close that off. And let's create an activity. So do activity run equals new activity and for run. set description, well, let's just add run in there. And then we'll do activities. add run. And let's just copy this real quick. And add a couple more, we'll do let's do bike and we'll do one more for swim. So we're training for our triathlon, we're going to run, bike and swim. Hopefully not in that order. So there's all of our activities now. And then we just simply need to return activities out of our application. So we should be all good now, shouldn't have any red errors or anything like that. Make sure that you're doing, and I mentioned it a couple times but do control shift o to organize your imports and make sure you've imported java util list instead of java. awt list. Everything should be good there, no compilation errors. Let's go ahead and start our server up, and you remember our activities here is based off of the url/activities so if I try to run this right now it's going to break and I'll tell you why. Because we don't have anything saying that it will allow it to pull through. So activities. json, if I come back it's going to give me a 404 and the reason for that is our Web, that xml is not configured yet. So let's take a minute and let's talk about the Web. xml.

web.xml

So the dispatcher servelet that needs to be configured to allow the various request types through. We first started out with html and just allowing html through our application. We used the internal resource, viewresolver to look up our jsp pages. The next thing we allowed through were pdfs and we used the MVC resources all on that which is underneath the covers as using the B named viewresolver and a few other resolvers that it creates for us. Just to expose those static pdf files. We could have done the same thing with images or any static content for that matter. The next thing we need to do though is configure it to let JSON through. And we did that with the content negotiating viewresolver and it's the same for xml. To allow xml content through we have also configured the content negotiating viewresolver. So now we need to go through and add to our servlet mapping to let these other file types come through.

Demo: web.xml

So let's open up our web. xml and we can do this two ways. We can go down and navigate to the web. xml and our source main webap directory and Web inf where you can also do control shift r and that will bring up the open resource dialog and we'll type web. xml really quick and that pulls up our application right there for us. So just double click on web. xml double click on the tab and it will open up full screen for us. You see where we had html configured before and we also have pdf configured. Let's add one for JSON down here. Let's copy that, just go to the bottom and say asterisk. json and save that. So we're going to allow requests for JSON to now come through our application. And if I put it as full screen mode, just double tap on that tab again, and I'm going to restart my server just to make sure those changes took place. Sometimes it doesn't recognize it on xml files. Go down to my browser again and we have our 404 air from our last demo that we did here if you didn't run through this, you local host;8080/fitnessetracker/activities. json and that activities is what we named our method here, our value for our request mapping and set up our controller and our minutes controller. If I go back here and go ahead and hit enter it should pull back my list of activities for us in JSON format and you can see that it does. So here it is in JavaScript object notation there's my description for my first object is run, my second one is bike and my third one is swim. So we've got our objects being returned by a JSON now for us and are available to our UI.

jQuery

Jquery is one of, if not the most popular JavaScript frameworks out there. There are thousands of plug-ins available for Jquery, they have a plug-in site devoted to developers just submitting back open-source plug-ins for people to use. The core framework of Jquery itself can easily consume JSON post and create html on the fly which what we're going to use it for in this simple demo that we're going to illustrate here. Jquery is just a file that that we drop into our project so we don't download it using Maven like we do jarfiles, we just need to go to the Jquery website, download it and there's a few types. There's a minimalist version where they've condensed all the white space and it's not very easily human readable but it's a smaller download. We just drop that right into our application.

Demo: jQuery

So now that we have this JSON response being returned from our activities rest call let's go ahead and download Jquery to do something with this result set. And we're actually going to add a drop down to our add minutes jsp page, just to show how we can consume this JSON response. Let's go ahead and open up a browser window and we can just search on Jquery and it will open up Jquery. com and in fact we want to go to download Jquery and there's two ways we can download this. You can download this in production view or development view. Let's go ahead and select development view just because it's more readable. It's not the minimal version and you can actually see what it's downloading and doing. So if I click download here you can see it says, in human readable form. I want to right click on it and click save as and just save that to my desktop. Once that's done we can come back in our application over here and we just want to drag that right in and clips will ask us to copy it. And we want to drag it to our source main Web out directory. Don't go down into your Web inf like we do with the jsp pages. We just want to place that right in webap, it will ask us to copy this files and we'll say yes and you can see there's our Jquery-183. js file. Now what we want to do is open up our ad minutes page and that's in our Web inf jsp directory. And let's go ahead and change a few things here with structure. So let's add another column here. I'm just going to do a simple htmltd and we'll close it off. And in here we want to do a form element using the spring form tag library of form select. And we're going to call this activity and we're going to give it ID equal to activities. Now we're going to use this ID because we're going to dynamically build this list on the fly. We're going to basically do a simple mash up and there's little bit of html that's involved with this. We also need to come down here and change our call span to three just so that the buttons line up right but it will work either way. Let's go up to the top of our application and go into our head section and go below title and we want to add a script element. So the type is text JavaScript and the source is equal to Jquery and that's the Jquery that we just added down here so Jquery-183. js- 1. 8. 3. Js make sure you've typed it the same as it appears down here and go ahead and save that. Now right now we aren't doing anything with that object and the example I'm going to show you may seem a little bit contrived because we have a very simple data set that we can just return simply with our bean. But doing this we can cash some things, we can do some interesting things with our -- if we were using like hibernate in the hcash on the back end to have this list returned not as part of our model object. So let's create another script tag here, that's text JavaScript and then inside of here we're going to start doing some Jquery. So let's go ahead and grab the document, we're going to say dollar sign document. ready so basically when the document is ready to start being manipulated, when the dom is available, go ahead and open that up and make sure you're inside those parens, and we want to type function and we're going to do a basic function that's going to build our options for us on the fly. So let's do dollar sign. getjson and we can take now, and grab our URL that we had created on the fly. So, well, not the fly that we turned our JSON response I should say. So this URL right here. So we're just going to just mash this up. Just grab this guy, control C, go back into our application and paste that in there. And we want to come to the end of that string, do a comma, and do an open brace, and do ajax true. We're going to need that single text, true and there is our closing brace there. Want to do a comma and we want to do a function on the data. So we're going to pass the data that comes back from that result into our -- into this function here so our data that's coming back is going to get passed into this function. We want to create a var html and say equals and we're actually going to put some html in here that we're going to manipulate on the fly. So we're going to do option, value, equals and we'll go double quotes, close that and we're going to put a default option value in here. So we'll say, please select one and close our option off. And there's our default option that's available and then we want to put a var and we can name these whatever we want in length equals data. length so we're going to take the length of that result set that came back which has our three elements in there and we're just going to create a simple four loop on those. Four var I equals zero, I less than length increment that, I plus plus and let's go ahead and start building our html. So I'm going to say html plus equals option, I guess we need to open that option element. Option value equals and this is where we can start dynamically manipulating this html on the fly. So we're going to say plus data at position I. description. So I want to grab the description and that's what the value is going to be that we sent back to the server. So we'll add the plus sign, single quote, double quote, and we're ready to close that off and then just on this next line here we're going to do the actual displayed value and we're using the same for the label as we are the return value. So we're going to do just the same data I here and get rid of that space, just so it doesn't mess us up. Data I. description here. And we want to close our option element, close option, and a semicolon there and then at the end of our four loop we just need to close our option off, so we'll say html and plus equals sign close option, just to make sure we get the last one in there. And a semi-colon and we're almost done, we need to so a dollar sign activities that's an element that we created down below and say pound activities which goes through and selects the element out of the dom and say, hey, replace the html that you have for that with this html that I'm going to give you right now and I want to close my activities off and get rid of this extra white space for no reason. And close that guy off. So there's all of our, oh, that should be a dollar sign, not an ap sign. Dollar sign, make sure you caught that there. There's s a -- I had put an application sign there before, should be a dollar sign. No errors, you shouldn't see and red errors down the side here. Now if we run this it should go through and pull our activities down to our page but there's one little thing that we have to add first. We put a path here on our form select and that is telling us that we need to, inside of our exercise object, have an activity inside there. so if we go into our exercise object up here, open it up, we want to add a private string activity and we want to right click on it and say source generate getters and sitters, and hit okay and now we have something for that path to bind to. So inside of our add minutes here we just had that path that's on our exercise object so we're going to submit an activity object back to our page. Let's go ahead and save that, it says it reloaded but let's just go ahead and bounce that just to make sure. And let's start from the beginning. So let's go to our application here and go to add go. html and that should pull our page up and say hey, let's -- and what is our goal for the day and we'll say our goal is 40 minutes. And go ahead and put enter goal and it should display our jsp page and here's our rsg page with our exercise minutes for the day here and we should have a drop down that has our three values and there they are, run, bike and swim. I can choose one of those and say that we exercise for 30 minutes today and click enter exercise. If I go in here you can see the enter exercise was displayed but let's go to our add minutes controller and just so we can see that something is actually occurring here we'll go ahead and copy this system at print one and say activity and we'll do the exercise. get activity, save that, it will reload that context and let's go ahead and enter that again. We'll choose bike and enter. And then on here we can see that we exercised for 30 minutes and our activity was bike. Now in this method we would typically now go out and call our data base or do something more than just display a system. print with it and probably navigate to another page, but there is the working end to end example of Jquery being pulled through our jsp page here. Let's do one more thing here just to show you another tag. We have this hard coded url in there and obviously we don't want a hard coded url in there especially at a local host, 8080. We can go ahead and use the spring url tag which is just one of the tags that we had seen earlier in the demonstration and do spring colon url value equals and we're going to do double quotes here and let's just delete this back to activities. json double quote, close that element off and if I save that go ahead and redisplay my page it came through and replaced that with my url here. So there's just another one of those spring tags that's in there so we don't have a hard coded URL in there but I wanted you to be able to see where that URL was pulling from the first time we ran it. That's why I didn't just show you this one to begin with.

Extra pieces to build Out Your Application

So now we have all the basic pieces that we need to do a Spring MVC application. And now we're going to spend a couple minutes just to show you a few things that aren't necessarily pieces of Spring MVC but really what you do if you're going to make a full fledged application using Spring MVC and all along we've been talking about not putting business logic in our controllers and we've done a really good job about cleaning that up but we have some hard coded things in here like our activity list that we're returning and we're not doing anything with our object once it gets back. Although I'm not going to wire up a data base I want to show you the pieces based off of the architectural module that we looked at earlier that you would want to typically add to your application. So let's go ahead and go over here quickly and I'm going to stop our server just to keep it from redeploying while we write some of this code. I'm going to create a new package. Create a new package and I'm going to call this com. pluralsite. service so it's our service terr is where our business logic should go. And inside of here I might create say an exercise service. So let's right click down here and say new class and do exercise service impl and everybody has a different naming convention whether they use I for the interface or impl for the implementation class. We'll just use impl for the implementation class here. And I'm going to go in and annotate this with at service. And inside of our service we want to name these and I'm just going to call this exercise service. And we can save that. You can put a method out here that's maybe find activities, find all activities. So let's maybe first start by pulling this activities list out of our controller and putting that into our class here, so into our service class. So let's cut that out of there and we'll leave it broken for just a second. And do public list activity and we'll do find all activities. Open, close, and we'll just paste this guy in here and we'll return our activities. Save that. Now so we've moved this data out of here and we're actually are probably going to move this again just to show you another example. But let's right click and go to refractor and say extract interface. And we'll just call this the exercise service and we want to expose that find all activities, we can uncheck these use extracted interface, we don't care about the method comments. I'm just cleaning up some basic interface stuff so it doesn't cloud things up for us. Activities, let's see here. Sure you can, you're out there. Okay, so now that we've saved that we have our exercise service interface, you can see our interface here. Now what we can do is we can go into our minute's controller and get this from our service. Now, I'm going to auto-wire these I'm going to do everything using dependency injection, come up here and say at auto-wired and I'm going to do the private exercise service and we'll do the exercise service just for the variable name and save that. Now we do need to import that so you can do control spacebar and select yes. Now there's one thing we need to do, well, there's two things we need to do before we can use it, we need to clean this up down here. So we can just return exercise service. find all activities and save that. For that to be auto-wired, if you recall, when we're using our controllers, right now those are being auto-wired, they're being looked up automatically out of our servelet Config. So I go down to my servelet config and open it up and you can see that we have this context component scanner here and right now it's looking for everything under controller. Let's actually just back that up to com. Pluralsight and say I want you to load anything under the com. Pluralsight package. So it's going to -- we're just going to broaden its scope. Some people will break these up into multiple configuration files where I have my service loaded in one configuration file and my controllers loaded in this servlet config and then my repository is loaded in another one. Just for the sake of this example I'm going to just broaden the scope of this search so everything just comes from com. Pluralsight. Now that we've got that working, let's maybe clean up our imports here, I just did control shift o in there and start our server to make sure that we haven't broke anything. That alls started up fine. And let's just, for the sake of the example start from the beginning again, I had go. html and that looks good. We're going to say 45 minutes, enter and it's bringing us to our page here. and we got, we've done five minutes of biking today and in our exercise. So now our esc is working and everything is coming back valid there. We could even, if we wanted to, with our validation say, you know, let's bind it to where the exercise that they selected cannot be null or those types of things. We never wired up the validation onto exercise itself but we can easy enough by just opening up the exercise object. Come over here and much like we did in activity, open up activity, or excuse me goal. We had the range on there, we could go inside of exercise and say that this range can be the same as the goal, copy that off there. And paste that in there. We could also do a at not know at this one and say that we have to have a valid activity that's coming through there. and we can go into our controller as well and pull that up. Now the one difference here, is if we start adding the validation on this we have to break up like we did for the at goal for the goal controller. Remember inside the goal controller we had it broken up based off of our request type, so we had the method get and the method post and do our validation stuff like that. So I just wanted to show you more of what this takes to be a true application. Some of the navigation stuff, how we incorporate our service tier, so now we pulling stuff out of our class and our controller is really just focused on doing controlling things and doesn't have this boiler plate data in there and you can see it really shouldn't get that big, it really shouldn't get bloated. There shouldn't be business logic in there. That's the type of stuff we want to push to our service tier. So just to show you those few pieces. We would also create a repository tier as well and that's what would interact with our data base.

Summary

In this module we covered everything that we needed to do to expose Restful services from our SpringMVC application. We used Maven to download the dependencies we needed for Jackson and the Spring object xml marshaller. We also went through and configured the content negotiating viewresolver which does have a lot of configuration that goes with it but it's kind of a crucial piece because it's doing some pretty important things. And we also went through and exposed JSON and html and looked at the various types through our servelet, our dispatcher servelet inside of our webdataxml. The last thing we did was add a Jquery to our application and we went out and downloaded it and showed that you just had to drop it into our ide, there's no special downloads or anything and we looked at the human readable form of Jquery just to show you that you can see what the framework is doing and we built a mash-up on the fly. Now for something with a simple as three results we wouldn't probably normally do that but if I was pulling back thousands of results or doing an ajax type ahead, you can see how this would be a valuable tool for us to build that html on the fly. We had our demos to go through and we had an extra demo just showing what it would take to round out this application to tie it to a data base. So we covered all the pieces that Spring MVC needed but just to show you some of the extra things that your application would do to make it a full fledged application.