Course Overview
Course Overview

Hey, everyone. My name is Brian Hansen, and welcome to my course, Spring Data JPA using Hibernate. I am the CEO of Complete Programmer and a director at Software Technology Group. Hibernate and JPA are one of the most advanced time-saving frameworks to be introduced into Java development. In this updated course, we're going to cover configuring Hibernate, utilizing Spring Data JPA, using Docker to run our database, various JPA tips and tricks, and all of our configuration has been migrated from XML to use Java configuration with annotations. By the end of this course, you'll know all of the fundamentals of Spring Data JPA using Hibernate as its provider. Before beginning the course, you should be familiar with Spring and Spring MVC some. I hope you'll join me on this journey to learn Spring Data JPA and Hibernate using the Spring Data with JPA Hibernate course, here at Pluralsight.

Spring, the Java Persistence API (JPA), and Hibernate

Version Check

Course Overview

Hello. This is Brian Hansen from Pluralsight, and welcome to my course on developing applications using Spring and the Java Persistence API, or commonly referred to as JPA. We don't assume that you have any knowledge of Spring or JPA, but experience with Java and relational databases will help accelerate your learning in this course. We're going to be learning about developing with JPA, and we will configure our application using Spring. We'll also be diving fairly deep with both of these technologies, and there are a lot of code samples that we'll walk through in this course. If you haven't used Spring MVC before, I would recommend that you take my Spring MVC course that's here on Pluralsight prior to doing this course, because we will be using the Spring MVC application we created in that course for the interface to our application.

What Is Spring?

The Spring Framework started out as an Inversion of Control container, and although I won't be covering all the details of Spring in this course, I'm going to cover enough to get you comfortable with understanding what Spring is doing for us inside of our application while we learn more about JPA and Hibernate. Spring was conceived to reduce or replace some of the complex configuration of the Java Enterprise Edition, and later it was built around using Java without EJBs. This is an important point because JPA was extracted out of the Enterprise API because people wanted to use JPA without EJBs. So, what is Spring? It is a framework originally built around reducing the complexities of enterprise Java development and later also providing enterprise development without EJBs. Spring can essentially be used with or without EJBs, and this is an important point because Spring enabled us to do enterprise development without an application server. This is one of the reasons that Tomcat and some of the other lightweight containers have taken the Java development space by storm. It is easy to use and lightweight, and until Spring, you were either using enterprise features with

an application server, and you had a more complex software development lifecycle, or you were doing rich client applications. Now, Spring enables us to switch over to using a lightweight container like Tomcat, which is easier and faster to develop applications with. Spring is POJO based, and POJO, as you remember, stands for Plain Old Java Object. All of the objects that we're using can be used and wired without Spring. The only reason I bring this point up is because some people often think there's some magic or black box that they don't understand about what's going on behind the scenes with the Spring Framework. It is still built around simple POJOs. We've talked about it being lightweight now, but we should also point out that Spring was built out of the frustrations of J2EE and the older APIs. They looked a lot at the blueprints and things like that and said, what's a better way for us to use these tools and make our code more testable? Spring also heavily uses AOP, or aspect-oriented programming, to apply things like transactions to your code without you needing to go through and manually or explicitly define them inside your code. We will use a AOP, although you won't realize that it's in your code just through some of the simple annotations, and your application will then automatically have transactions. Spring is also built around best practices and design patterns, which is one of the most appealing parts of Spring to me. A lot of developers used to wade through the JEE blueprints and implement singletons or factories, facades and abstract factories. You can still do that, but all of these are built into the Spring Framework as well. So if you have existing patterns that you've already developed, Spring will work well with these. But Spring also does this itself inside the application and inherently encourages better design in your code. So these patterns, singleton, factory, and abstract factory, if you've ever gone through and built some of the older style methods of making your database transparent to your application, you've done this, and you've implemented a concrete factory. But Spring also makes a lot of use of the template method pattern, as well as callbacks and those types of things. But lastly, it's heavily annotation based, and some developers used to shy away from Spring because of its XML configuration. With the more recent versions of Spring, all configuration is annotation based, and more recently, configuration is all done within Java code, thus eliminating all XML, if desired.

History

We're going to take a minute and just walk through a little bit of the history of Spring. Spring actually started out as a bunch of developers that were consulting and were tired of some of the shortcomings or painful pieces of developing with JEE. So this group of people got together and put together the Spring Framework, which actually used to be called Interface 21 and decided they wanted to roll this out as an enterprise framework available for the public to use. Their first milestone release was in 2003. Now, not really a big deal, but it's when things kind of got rolling for them. Now, in 2004, they had their 1.0 release, and one of the things you're going to see as we step through the years is that Spring had a pretty consistent, very stable release cycle. As part of their release cycle, they've always tried to keep it backwards and forwards compatible. In fact, all of the applications that I've ever written since 2003, I don't know of one thing that I've had break from backwards compatibility. I have actually also integrated some key dates in here with Hibernate in our timeline and notably started with Hibernate 3.0, which was released in 2005. Where it starts getting interesting though is in 2009 when JPA 2.0 was released and then again in 2013 when JPA 2.1 was released. All this time, Spring 2.0 and 3.0 were subsequently released in that

timeline. Hibernate 4.3 was released in 2013, as well as to support the new features of JPA 2.1, and then Spring 4.0 and 5.0 were both released after that, integrating newer JVMs and continued long-term support. Let's talk more about JPA though, since that really is the focus of this course.

What Is JPA?

JPA stands for the Java Persistence API. And much like Spring's roots, JPA was designed to make things easier. JPA focused on an object-relational mapping, or commonly referred to as just an ORM design principle. Originally, it was part of the EJB specification, but was later extracted out to be available just on its own. So, what is JPA? Well, there's a little debate on whether or not JPA started off as Hibernate because there were some other frameworks that had an influence into that design as well. But for our course, we're going to be using Hibernate for our implementation anyway. It is an ORM, or an object-relational mapping tool, and this is how we map our object-oriented language to our relational database. Much like Spring, it is POJO based and has a focus that's around being built on the object-oriented development. So if you've used things like JDBC or some of the other Java persistence frameworks, they have a tendency to focus more on the database side of things. Not saying that they're more powerful for that, but they tend to make you have some bad object-oriented development practices, where JPA and the JPA specification really focus on good OO design. You can use XML configuration for POJOs much like Spring, or you can do the annotation-based configuration that we'll be doing in this course. Since it is like Spring, everything is going to be built around these best practices and how we want to design our code, keeping a better OO design inside of our application. And lastly, we also have pluggable persistence providers. Now, this is kind of an interesting point. I've had people ask me before, why do I care if I can swap out a persistence provider? And I actually have on three different projects swapped out the entire persistence provider that I was using from one framework to another because it worked better with the database that I was using. So for this example, we are going to be using Hibernate, but there's TopLink or EclipseLink. And I won't say that they're better or worse than Hibernate, but they've been around, and they actually kind of solve some different problems. So that we can have these swappable persistence providers is actually a good thing and keeps everybody in check.

Current Release

Continuing on our timeline of current versions that we're going to be using, we had looked earlier at JPA, and JPA 2.2 was released in 2017. I'll honestly admit that releases for JPA versions have slowed down a lot, and some of that has to do with us tackling a lot of the problems that we see inside of ORM tools. Hibernate 5.4 was released in 2018, and we will be using a new version of this release, just a minor dot revision. And finally, Spring 5.3 was released and has all of the current integrations baked into it. So this is a pretty good snapshot of the current stack that we will be using with possibly some minor dot revisions in there as well.

The Problem

Let's talk about the problem that JPA and Hibernate solve and why we want to use them. Whenever I'm interviewing Java developers, I always ask them why should I use Hibernate or why should I use JPA? And a lot of times they don't understand what the problem is that we're trying to solve. Let's take a second to talk about what that problem is. One of these things is that developers don't always make good DBAs. A developer doesn't always understand some of the reasonings behind why we have foreign key constraints the way that we do or how to best represent what they're trying to do in a relational database. Relational databases are two dimensional where object-oriented languages are three dimensional. Also, the data model doesn't always line up with that object model because of that two-dimensional versus three-dimensional representation. I'm talking about it from the object-oriented side. What if we have an existing database that we're trying to map to? Getting these two to line up isn't always easy. Another problem is configuration. Although configuration is a lot better with JPA, it can still be better, and that's where Spring steps in. Things like transactions, testing, data source injection and configuration, these are all things that we're going to solve with both Spring and JPA inside of our sample application that we're going to be writing. The main point I want to talk about for a minute though is business focus. The business doesn't care if I'm using Spring or JPA. They care that I'm focusing on solving their problems. Let's look at some sample code to show you exactly what I'm talking about. This method that you have in front of you is an old-school JDBC query, and yes, you'll notice it's really wordy. In fact, I had to shrink it down just to fit it all on this screen. I've got to get a connection, I've got to get a prepared statement, I've got my result set, I'm getting all the stuff, populating it, binding it, looping through it, everything going on, and then finally I've got this catch statement at the bottom where over half of my code is making sure that I've closed the result sets, and I've closed the statement, and I've closed the connection, because if I don't do all of that I'm going to end up having memory leaks and the database will run out of connections. Believe me, if you've developed in Java code very long and you've done JDBC for very long, at some point in your career you forgot that finally block or one piece of that finally block and you've ran out of connections on your database. So really, what is the business focus here? They care about two things, that I got this query done and I shoved it in that object. So there's 35 to 45 lines of code there that they don't care how we do it. They just care that I'm grabbing the car based off the Id and getting this object back and returning it. It's all the application wants. What does it mean to get a car by Id? Well, just that, it doesn't mean that I have a finally block and all this other stuff that's going on. Our focus is really just on using a tool rather than getting the business what they need back, and you can see it's very error prone. If I forget one line in here, the application will deploy and it will run, it'll just chew up resources and finally run into and out of resource exception or out of memory or heap space or perm gen or the number of database connections, all problems that you can experience in a full-scale application if you're just using plain old JDBC. Let's talk about the solution though and how JPA fixes this.

The Solution

The solution is obviously JPA. It removes a lot of the boilerplate code, helps developers build objects and bridge that gap between our relational database and our object-oriented code, Spring will handle all of our configuration and our transactions behind the scene, our

code can just focus on testing and testability if we want to, we're not going to cover a lot of testing in this course, but we'll show you how some of it works just so you've seen it and you'll understand how you can build your code so that it's more easily unit tested. Transactions are also transparent to the developer, so when your code stops and starts a transaction, and now it's not just muddled throughout your code, you can just write your code how you normally would and apply transactions to it. You also get the option to do annotation-based development, and for me this is a big one because development is a lot faster and a lot easier using annotations. As we go through our samples, you'll see us generate tables in the database and you'll be amazed at how easy it is to generate a data model. So, now I talked before about the business focus, let's look at that example again. Here's our JDBC example that we had before and all the problems that it has with it. Using JPA and Spring, all of this code ends up turning into this method here where I say I want to get a car by its Id, and literally that's all the code for it. I'm still going to have my car object, just like I had to have on the other side, but this is all I need to do to execute this query. It handles the transactions, it handles all the JDBC or SQL associated with it, and then we'll see as we go through these examples how it just wraps this all up into our object for us, but look at how much simpler this code is and our focus is just on getting back what the business needs.

Summary

In this module, we talked about what is Spring and a little bit of the history of Spring. It's a very stable framework, has a pretty consistent release cycle, and it's always been very backwards and forwards compatible. We also talked about what is JPA and kind of the roots of JPA, how it was influenced from tools like Hibernate, TopLink, JDO, and became a 1.0 specification that was later extracted out of EJBs because people wanted to use it without using Enterprise JavaBeans. We also looked at the problem of what our business focus is and all that JDBC code that really has nothing to do with us getting this query back, and those are hoops we have to jump through to find that information. But really that's one line of query and getting that car object back is what our business cares about and what JPA and Hibernate will help us do inside of our Spring application. Let's get rolling on our application though and look at what we need to do to download the sample app and do a quick walkthrough of our Spring MVC application now.

Download and Walkthrough of the Spring MVC Java App

Introduction

Hello. This is Bryan Hansen from Pluralsight. In this module, we will configure and install the scaffold application that we're going to build upon throughout the rest of this course. We're going to cover the prerequisites for the course, as well as walk through the scaffold application and verify that your environment is up and running.

Scaffold Application

What are we going to build, you ask? Well, we're going to continue building on the conference application that we've used in our other Spring courses I've authored here on Pluralsight. The UI is mostly built out for us in code that we will download, and we will be wiring up the back end. Before we do that though, let's look to see what we have the prerequisites listed for this application.

Prerequisites

The prerequisites are similar to other courses I've produced here regarding Spring and some of the other libraries surrounding it. We're going to still be using the Java 11 LTS version. Java 11 is the current version for long-term support, and after that we don't have guaranteed long-term support. So yes, Java 12, 13, and 14 have already been released, they're just not as stable as 11 is. They work, and in most cases all the examples we'll be doing should work there as well. This is just the one that's guaranteed by them to continue working. I'm assuming you already know how to install Java, so we won't be covering that in this course. I'm also going to use Maven in this course. Spring no longer offers up a way to download just the JARs that you use for Spring. You have to use Maven or Gradle to download these dependencies. There is a Pluralsight course on Maven that I've actually authored here on Pluralsight. If you have more questions about that, you can go follow that course and learn how to use all the ins and outs of Maven, but it's pretty simple. If you aren't super familiar with it, we'll show you everything you need to know inside of the course here. The next thing you'll need is an IDE. This course has now been updated to use IntelliJ. We got feedback that the majority of our users wanted to see and use the examples with IntelliJ, so we have changed the course to use it. I will be using an Enterprise version of IntelliJ. They do have a 30-day trial, as well as Student licenses available. If you do want to use Spring STS, which I do like as well, I will try and answer any questions you might have on the forum for this course. And then Tomcat, the app will run under Spring Boot or Tomcat. We're also going to use Docker to run the database, but we have a whole section dedicated to just that. So let's recap here. We're going to use Java 11 because of its LTS support, we have moved to IntelliJ, but all of the examples will work with Spring STS, and just the most recent version of Maven, as that's how Spring requires you to download those JARs anyway. Not required, but I do recommend having complete the Maven Fundamentals, Spring Fundamentals, and Spring MVC Fundamentals courses. This course does build upon those, so it will answer a lot of questions you might have. Let's look at how we download that source code now.

Downloading the Scaffold App

Let's take a second to walk through the steps of downloading our scaffold application. First, you can download the app and slides right here on the course web page under the Exercise files heading. It's definitely a good option. I have also hosted the application up on GitHub just to make it so that I can update the files that are in there and that you can get the most recent version. Don't worry though if you've never used Git or GitHub, you actually don't need to have Git installed for this class, and you don't really need to have any prior working knowledge of Git either. We'll do a demo now of how to get that and get our environment set up.

Download App Demo

Let's show you both ways in which you can go ahead and download the source code for our course here. The first is going to Pluralsight's website where you're already viewing this video from, and you'll notice on the main page for this course there is an Exercise files tab as one of the menu items that you can choose under the course header. And if you click on Exercise files, it will go ahead and give you the option to download the exercise files here. And it's a ZIP file, it's got a before and after for each module, it contains all the slides and everything else. I won't go into any more detail here because it's really straightforward and already has instructions on how to do that. So that's the first way you can go ahead and grab the source code. The second way is to go to GitHub, and I've pulled up another browser window and typed in the URL github.com/bh5k/spring-jpa. So you can see right there in my address bar. And if you actually look at the header of my source code, you can see it typed out there as well for bh5k/spring-jpa, and this is just a repository I have, and that will take you to the main page for this course and the branches and all of the source code that's associated with it is available here. Now if you're already familiar with Git and GitHub, you already know that you can fork this and pull it into your own personal repository and go ahead and start developing on it from there. But if you're not, don't worry about it. We're not going to go into that level of detail. If you know that, go ahead and execute it that way. If not, you can just go ahead and click the download Code button here and choose to download a ZIP file. It will download really fast because it doesn't contain any binaries, just the source code for this application. Navigating to the directory where your browser downloaded that zip file, you'll see that you have a spring-jpa-main.zip, or whatever your download was, maybe you grabbed the exercise files, but the setup is pretty much the same from here on out. Double-click that and expand this folder. I'm going to start off by renaming this to conference. That's the name that our application is going to have inside of here. And I'm going to copy that into my dev/workspace directory. So, yours may be c:/dev or you may be working on a Mac or a Linux-based system where it's just /dev, /workspace, wherever your workspaces at. I'm going to copy that folder into my workspace. And typically that's enough, but I do have other projects, so I'm going to create a new folder, call this spring-jpa, and copy that conference folder into spring-jpa. And inside of here, I'll now see that I've got a couple of hidden files, my .gitattributes, .gitignore, and then some setup information. But really the two that I care about are our pom.xml and our source file, which are both contained here now. And we can go ahead and fire up our IDE to load this project in. Launching my IDE for the first time to import this, it'll bring up this dialog asking me what I want to do, if I want to create a new project, import it, open or get one from version control. I'm going to go ahead and say Import Project, and I'm going to point that to the directory that we just created. So, dev/workspace/spring-jpa/conference and just choose that directory and select Open, and it will bring up a dialog where I can select Maven and say Import project from external model, and choose Finish, and your IDE will pull up with the application all configured, installed, and ready to go. We've got all of the pieces inside of here set up. I'm going to walk through what the actual application does, but for now it's all installed with Maven configured, your library is downloaded, and everything ready to start development.

What Is in the App?

Now that we have our scaffold application downloaded, what's in it? What does it do? Well, it's a conference application. It's the same one that we've used and standardized across a lot of our Spring applications here on Pluralsight. It has a Registration section and Users, and we gather all of that information. It happens to be the same application we add security to in my Spring Security course here on Pluralsight. It captures posts, gets, does all of the things that we would expect inside of a standard web application, but it does not store any data yet. That's the whole goal of this course is to tie that into the back end where we show you all the annotations and configuration to store that data. So to show how it works though, let's walk through this architecture a little bit deeper so you kind of have a nice recap of what's inside of this app.

Scaffold Application Walkthrough

The first time you pull up your project, there are a couple of things that are already defaulted for you, one of which you don't want to use, and that is the deployment configuration that already exists from Spring Boot. If we go over to our Configuration drop-down, you'll see that we have an Edit Configurations option or a ConferenceApplication option. The configuration that's already created for us is a Spring Boot instance, and it uses tc Server, which is a modified Tomcat instance. I'm going to be honest with you, you don't want to use this because it breaks a lot of things. It's meant to work with some of their tools. I have found it to be more problematic than just creating a Tomcat instance here, so I'm going to actually not use that and add a Tomcat Server, and we want to create a local instance, and I already have Tomcat downloaded on my laptop. If you have not downloaded one, you need to select one or configure one from this dialog. And so you can go through here, hit Configure, point it to a downloaded instance, choose whichever version you downloaded. I am using 9.0.30. I would keep it within the same major release as me just to guarantee that it works. They do have some experimental versions, some alpha versions that are out there that are later than this, but 9.0 is working fine with all the examples that we have for this course, and it's what I will continue to use. Now, before we change anything else, I want to point out that there is a warning on the bottom of our page already saying that no artifacts are marked for deployment. You can click that Fix link or you can go up to the Deployment tab and click the plus button and say that we want to do an artifact. We have two options inside of here, we have the conference:war and the conference:war exploded. I'm going to just choose conference:war. And the application context that it is at the bottom of this is the URL to our application for development. So if you don't want it to say conference_war, you need to change this now. You can back it off to just /conference. I'm going to go ahead and do that and click Apply and then OK, and now our configuration is ready to launch. So if I go ahead and click on that and hit the Play button, it will take it a second to load the first time, but you'll see our application deploys, fires up Spring, and then it will bring up our browser for us. Our app's default configuration is to pull up the index or greeting page, and so here we are at the root of our application, and if we click Add Registration, it's going to pull up the registration page and we can put in whatever we want and hit Add Registration, and it just cycles back to this page. It's not storing anything, there's nothing else happening, and if we back up to the Add User, the same thing happens here as well. It's going to pull down some default information because we're not pulling anything from that database yet. So, let's go

ahead and start implementing that now. To point out some key files though before we dive right into the code, first of all, this application was all configured using Java configuration, so it's launched through the @SpringBootApplication annotation and a main method inside of our ConferenceApplication class. The AppConfig is all set up inside of here and uses this ConferenceConfig file to wire together our application. This is where we're going to be making almost all of the changes inside of our application for adding in Hibernate and JPA. It's all annotation-driven. If we open up any of our controllers, you'll see that we have the @Controller, we have our model stuff wired up the same way, there's all of our registration and user inside of here, and we're going to add and flesh out the details of our service tier and our repository tier doing this same fashion. So we'll implement on top of these two model classes we have here and tie into this controller. And, one last thing to mention is the application.properties. This is where we will go through and wire in our database configuration to go ahead and connect to that database. So, really, those key files there, that's what the bulk of our configuration for our Spring MVC app connecting to a database will take place.

Spring MVC Recap

The key points we mentioned from our Spring MVC app were that it's all configured through Java. There's actually no XML in our app. There's not even a web.xml in our application. It's launched through the AppConfig, or our conference ApplicationConfig.java file, and everything is annotation-driven, our requests from our web app to how we've configured our Spring tiers. And lastly, our application.properties contains all the pertinent information that we'll add to connect to our database, any pooling information, that type of stuff, will all be handled. So those are the four main files that we're going to focus on for a lot of this class.

Summary

To recap what we learned in this module, we downloaded the sample app and we showed you that you could do that through the exercise files or by using GitHub and forking that to your own repository or just simply downloading the source code from that repository I have listed there that's public. We did a Spring MVC setup and recap of how to configure your application to use Tomcat and walk through that, and we demoed that app to show you what that basic functionality was inside of it and what the project structure and pertinent files look like. Let's go ahead and dive into this next section now.

Architecture Walkthrough of a Spring JPA / Hibernate Application

Introduction

Welcome back. This is Bryan Hansen from Pluralsight, and in this module, we're going to discuss architecture. We're specifically going to talk about the role of Spring in configuring JPA with regards to our architecture.

Architecture

In the Spring MVC course, I spoke a lot about Spring and how it tied into the MVC framework and how Spring MVC is a model-view-controller framework, but it really is just dealing with the UI. We didn't tie a database into our application or anything like that. The next couple of slides I actually took from the Spring MVC course, but I'm going to talk about them in a different light. As I mentioned in the Spring MVC course, that software architecture has been around a long time, long since before Spring and Java were around. Architecture and design patterns give us a way to communicate what we're trying to do and to also implement some of the best practices so that we don't keep recreating the same errors or making the same mistakes. For us, having a framework like Spring is all about repurposing these best practices and not having to reinvent the wheel each time.

MVC Design Pattern

There is a standard MVC, or model-view-controller, design pattern that you may or may not have seen before. It's a pretty common pattern 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 our controller. The controller can change the model, or not, and then it will select the view based off of our action. From here, the model can update the view with the database off of the user's actions. Now this graphic is sometimes confusing, though, because it's stemmed from rich client applications that were more event-based or using a design pattern called the observer observable or subscribe listener type design patterns, not really applicable to a lot of web software development. The pattern and concept is still sound, but we usually don't have our model updating our view because we've gone back to a page at that point. This is a more accurate description of that kind of lifecycle that we maybe faced with in web development. This graphic is realistically more like the lifecycle that we will see in our web application. Our view can access our model, but it's usually done so through the controller. We'll look more at the lightweight approaches in the AJAX module later in this course, but the summary is we usually still make a request, even if it's a lightweight request, through our controller to access our model. We don't have our view typically going directly against our database or our model.

Tiers

You'll often hear people talk about applications in terms of a tiered or an n tier architecture or layers. We try to build our applications and tiers more now in enterprise languages, and there are great reasons to do so. A few of these are separation of concerns. This just means that each layer is only concerned with the task that it's assigned to do. Our presentation layer is only going to have presentation type things in it, and our business logic is going to be contained in a layer, and data access is going to also be contained in its own separate layer. Now there's another term called reusable layers 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 own appropriate tier that I'm not duplicating that logic. If you ever notice yourself copying and pasting code in your application, it's probably not architected in the correct manner. Now, that's a little bit of

a difference between the separation of concerns and the reusable layer. Separation of concerns is not about reuse, it's just about having things in the right layer so that I can rearchitect them later where reusable layers means that I have a purpose and a point that I'm trying to drive. I can expose the same data in multiple ways without having to worry about it, or I can replace my presentation tier without having to rearchitect my entire application. And another term is 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. For example, 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, then it probably isn't architected correctly. If done correctly, we can change our code and not have to retest everything, but rather just the pieces that we changed. This leads into a much larger discussion about unit testing, but that's outside the focus of this course. Since we're talking about layers or tiers of a Spring MVC application, and this is important because of how we annotate our components, which we're going to actually talk about here next, let's first talk about the 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 requests and selects the appropriate view based off of what we've requested or what information we got back from our data model. And then we have our view. For our case, we're going to be using for our model just simple JSPs, but we could also use FreeMarker, Velocity or Web Services to just simply return data back from our requesting controller. Now, one thing that's often misunderstood by people is that Spring MVC has nothing to do with regards to our database, but it has a model associated with it. So, is this model talking about our database or the model talking about what the framework is going to represent to our JSP page? It's actually referring to the view, what we're going to represent to our JSP or our web service, but we always have to get that information for somewhere. So, that's where our various components come in and what we're going to talk about next.

Components

We talked about tiers and layers, and we've even started to allude to components inside of our application now, so how do we represent these tiers with Spring and Spring MVC? We do that using these three components. We have our controller, service, and our repository. We've already discussed controllers already, they just route where we're going and interpret the user's request, and then there's the service, and that's where our business logic goes. It should also be noted that it's where our transactions will likely start if we're accessing more than one database table as well. And then there's repositories. The repository tier is also sometimes referred to as a data access object, or DAO, and they usually have a one-to-one mapping with our database table.

Controller

Controllers, as we've mentioned, handle our incoming request and building the response. I can't emphasize strongly enough that business logic should not be handled in the controller. This is also where our request and our response object should stop as well. We shouldn't hand those off to separate tiers. It should grab information from the request and the response and hand it over to the business logic. This works with our service and repository

tier for business logic and data gathering, and it's also annotated with the @Controller annotation. There are some convenience classes that you can extend, but you either have to wire them up or annotate them with controllers still. It should also be noted that this also handles exceptions and routes views accordingly based off of whether or not we had an exception or we've gotten the correct information.

Service

The service tier is annotated with the @Service annotation, and it describes the verbs or actions of our system. It is where our business logic should reside. In fact, it should all be contained here. It shouldn't bleed over into our repository tier. Another role of the service tier is to ensure that business logic is in a valid state. This is where all of our state management should be handled, confirming that we've got a valid object passed from a request. It meets the standards of our business objects or of our business requirements. Also, this is where our transactions should begin. If you are doing two-phase commits or there's a chance we might have to roll back or access web services, those types of things, this is where we want all of our transactions to begin. It often has the same methods as the repository, but a different focus. We may have a method in here that says find user by last name, and we may have that same method in our repository, but what we do if we don't find a user or how many people will be returned or what state we might return these objects in is controlled by the service tier where the repository tier is just going to get that data.

Repository

And lastly, the repository tier. It is annotated with @Repository. It also describes the nouns of our system. You can see where the focus is different from service versus repository. The service tier describes the verbs or actions that we wanted to do in the system, and the repository tier describes the nouns the data of the system. It's focused on persisting and interacting with the database or basic CRUD functions. It's also typically a one-to-one mapping with an object to a database table. You may have an address and have an address repository. You would have a customer object and a customer repository. It's also often a one-to-one mapping of those to the database table, but that's not always the case based off your database design. You may break things into multiple tables, like a person in an employee table, but you may only have an employee object inside of your application. That is more up to the ORM tool that you're using and how you've structured that data.

Summary

Let's recap what we covered in this short module about software architecture. It's a much needed thing inside of 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 reflective of what we're doing inside of our web applications. We also talked about n-tier architectures and why it's beneficial for us, and that leads us into a deeper discussion about the various components inside 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 back end and where our business logic

should reside. And even though there are more of Spring proper than Spring MVC, we should always use those within the Spring MVC app to help with our model-view-controller design pattern. Let's dive into some deeper configuration and start using this inside of our sample out now.

Recap of Spring Concepts that Are Used in This Course

Introduction

Welcome back. This is Bryan Hansen from Pluralsight. And in this module, we're going to discuss Spring. We're going to do a little intro or recap of Spring, but just enough to convey the principles that we will use for configuring our application.

Overview of Spring

Spring is all about making things work better together. It was designed from the ground up to help overcome difficult configuration problems common in software development. For JPA, we use Spring to configure the transactions, data source, and overall JPA configuration. With everything configured through Spring, we can then use dependency injection and inject resources into the rest of our application. Spring really helps tie everything together in a nice, non-hard coded way. Our controllers from our Spring MVC app can now seamlessly tie into our database and JPA objects without having to go in and hard code or look up references from within our application. Now I'm going to remind you, though, we are not going to cover Spring in its entirety because that's in other courses I have published here on Pluralsight. We're going to just recap the principles that we need to use for our application

Types of Injection

There are two types of injection in Spring. Setter injection is the most common, and this is usually the case because classes have setters to manipulate variables, but it might not have the constructor that we want. As you may have guessed, the other type of injection is constructor injection. Constructor injection will help enforce a contract within the code that was intended by the creator of the class. Either way, the code will work the same regardless of which method we choose or which option is available.

Setter Injection

Setter injection using the Java configuration approach is really is as simple as a method call. Some of the mystery of injection just goes away. A lot of developers I talk to are concerned with the black box fail of dependency injection or inversion of control, and with the Java configuration approach, it's more visible as to what's actually going on. With XML, there's a lot of wondering what's wired up and what's calling in, who and how this auto wiring is all taking place, and that just kind of goes away with the Java configuration approach. Setter injection is simply a matter of calling a setter on a bean. We're going to define a bean using

the @Bean annotation, and you can see here that we've got our getCustomerService method that returns a bean of type customerService, or a bean named customerService. And notice, though, that we have the setter injection in here. As we're building our beans, we're going to call this setCustomerRepository method on our customerService instance. To do that, we need to have a method called getCustomerRepository in our configuration that returns the customerRepository bean. This is going to wire the customerRepository inside of our customerService. Let me say that again. We're not going to call an instance or create, we're going to call and get an instance of the bean from the Spring configuration file. So when we define a bean such as our customerRepository here that returns an instance of customerRepository, you can see that those methods now line up. I'm going to call getCustomerRepository that's registered as a Spring bean, and that's going to return an instance for our setCustomerRepository instance on our service instance. Take note, Spring is still doing a lot of heavy lifting behind the scenes when a bean is registered, such as these beans are all by default a singleton and will only execute the method the first time it's called. It's a very key point because if I didn't have this set up this way, it would create a new bean every time we call this. Since it is in Spring, it's going to register this is as a singleton and only return one instance. Let's look at what the code would look like not using Spring since it's really the same code.

Setter Pojo

To use our setter-injected Java code, all of our code can be configured just the same as a plain old Java object by having a typical run method like here, a main method where we can create a repository, then we can create a service, and lastly, we set that repository into that service. Conceptually, this is the exact same thing Spring is doing behind the scenes, but also adding a lot of inherited functionality. You can see, though, the Java configuration is just the same as if we were doing it with Spring or without, and you can test code the same way.

XML and Spring Configuration

The focus of this course has since been moved to complete Java configuration, but to show the XML equivalent, this is what it would look like. You may maintain existing applications that still have XML in there, and the XML configuration actually still works just fine, so I wanted to show you the two so you could compare and contrast them. We have a bean for the customerRepository, we have a bean for the customerService, and the property for wiring the repository into the service. That's where our setter injection is occurring. We created that bean and then injected it through that property into our service instance.

Autowiring Setters

Instead of XML, and even just plain Java configuration, we can utilize auto wiring in our code. Auto wiring would still require us to have a bean defined as our customer repository, but we would then just mark the method in our service as being auto wired with that bean. This is actually the methodology we will use inside our application, and it looks very similar to the Java configuration with a slight twist. We define a bean as the customer repository, and then inside of our application we mark the setter for auto wiring in the

bean. This also incorporates the use of a component scanner. So instead of having everything all in one configuration file in the individual services, we can annotate and configure them as needed. It does move some of your configuration code around, but once you start using this approach, you'll find you generally like it a lot better.

Constructor Injection

Having seen setter injection now, constructor injection almost seems simpler. We have the same objects defined, but now we just move the call to get the repository to the constructor where we are creating the service. You can see inside of here we have our beans already defined, and then we just go ahead and call that instance to get it injected in there. So, you just simply put the constructor with arguments in the order that you want them to be injected and it will automatically call and instantiate those beans and fill that object for you. Let's look at that same configuration and how we used to do that with XML.

Spring XML Config Constructor Injection

The only difference to do XML constructor injection is we've now moved from a property to a constructor-arg element. It takes an index which specifies the position of the object we're trying to inject. Again, mostly showing this for historical purposes, you likely wouldn't be using XML configuration in a new project.

Autowiring Constructors

Auto wiring constructors is very similar to auto wiring setters. Instead of the XML configuration, just like setters, we have our customerRepository bean that we have configured, and note that the code is truncated for that definition and now we just use the @Autowired annotation to a constructor that takes the argument passed in instead of a setter. It really is almost identical as just moving the @Autowired annotation up to that constructor.

Context Files

So we've seen that Spring can be configured in multiple ways. There's the XML configuration, there's annotation, and then there's also pure Java config. Just to be clear, you can have annotations with XML and you can have them with the Java config. It doesn't mean that annotation-based is Java configuration-based, that's actually just annotation-based. We are going to be using the Java configuration with annotation approach. What also used to be done with a lot of XML, the web.xml, loader listener, all of that configuration is now all contained in our AppConfig, which is a very welcome change and makes the configuration in our app a lot less, and that's actually where we'll start off in the next module, integrating this all in to our application.

Summary

So in this module, we covered setter injection and constructor injection, we looked at code to see that it was all in basic POJOs under the hood, and we looked at doing both methods using XML configuration, as well as auto wiring using annotations. And lastly, we covered briefly the topic of context files, and it still has to be initiated or started through a context file, except now we're doing everything using Java configuration, that's kind of a historical approach. In the next module, we're going to walk through getting all of this configured and into our application so you can see these principles we've talked about and now put them to use.

Configuration of Spring and JPA for Development

Introduction

Welcome back. This is Bryan Hansen from Pluralsight. And in this module, we're going to discuss and walk through configuration. We're going to walk through all of the configuration to get Spring and JPA set up to develop the rest of our application. This module will have a lot of demos and configuration aspects to it. I did this so that we could get it all out of the way and dive deeper on just the specific features of JPA and Hibernate in the rest of the course.

Configuration Problems

Configuration is often a painful task. It can honestly be some of the most complex and time-consuming parts of development. And Spring aims to make this easier, but there are still a lot of moving pieces. It should also be noted that Spring isn't just about making things easier, but oftentimes making things more flexible as well. So there can be a trade-off for ease versus flexibility. We're going to walk through each piece of the configuration to get the development environment up and running. It's often the hardest part for developers when learning new technologies such as JPA.

Database Overview

Let's start off by talking about the database for a minute. JPA is an abstraction layer for our database. We can use almost any relational database that we have a dialect for. And if you don't know what dialects are, we're going to talk about these more in greater detail later in the JPA configuration section and some of the more detailed things that we can do with those dialects because it's a pretty advanced topic. For our course, we're going to use MySQL. We're actually going to walk through installing that in just a bit. But since this is a re-release of an old course, a new change is that we're going to use Docker to install the database and keep it contained. Docker has become the industry standard. And if you're not using it, you should be. It actually made this course simpler, and that was one of the deciding factors of why we used it. If for some reason you don't want to use Docker, you can install it natively, and all of the examples will work just the same. But I would encourage you to give Docker a shot if you haven't.

docker-compose.yml

All you need to do to run Docker is to have the Docker Desktop app installed. And for that, you just go to docker.com and download the version specific to your operating system. From there, we're going to add this docker-compose file to our application, and it will set up everything. In this case, we're going to have a database, and you can tell that by the db: that sets up the container for this. Its image type is of mysql:5.7, and the container name is conference. Next, we're going to bind the ports that are exposed on our operating system to this port inside of the container, which is 3306. From there, we'll tell it where it's going to store the data, which is underneath our project in a .data directory /db and then for our environment as well. We're going to say that the MYSQL_ROOT_PASSWORD is pass, and the database name is conference. And we have all of our stuff set up and configured right here. So this will download that instance for us, store it, bind the ports, and set up the username and password and database name for us. Let's add this to our application now, and we can start up our Docker instance.

Docker Download Demo

So I mentioned it in the previous slide, but to get Docker, you literally just go to docker.com, choose Products, and select Docker Desktop. From here, it'll bring you to a page where you can choose to download for the specific operating system that you're on. I am on a Mac, so I'll download the one for Mac, which I've actually already done, and just run through the install on it. Nothing special there. I do create an account and sign into that account on Docker just because it'll allow me to store and select images and do some other things. Really nothing promotional that comes out of it. But that's all you've got to do to get Docker installed on your machine so that we can start using the containers that we want to.

docker-compose.yml Demo

With Docker Desktop installed and running, I went ahead and switched back to our IDE, and I closed all of the open tabs we had earlier and shut down the tomcat instance that was running. And I'm going to right-click on our project and say New, File. We want to name this file docker-compose.yml. This is a YAML file. And yes, you do want to name your file the exact same thing as this. It does follow convention over configuration. So name your file this, and we'll hit Enter. And it will ask if I want to add this to Git. If you're using Git and have it set up that way, it's going to continually ask if you want to import those files. So I will just cancel this and do it in the desktop IDE that I've got installed locally. We can start editing this. So I'm going to begin with typing in version, and this does have some context-sensitive help. A lot of your IDEs do. If you don't, it doesn't matter because this is literally just a text file. A YAML file is just a text file. So we're going to add networks inside of here, and we're just going to use the default network. From here, we want to start utilizing these services and specifically the database service. And we're going to begin with an image, and the image we want is mysql. And this is name and case-sensitive, so I would follow the exact conventions I have typed in here. The container name, we're just going to use conference. And for the ports, we're actually going to use the default ports that they have. You could change this to something else, but you have to get it configured correctly with your OS and with the port that's inside of the container. We'll just use the defaults for now. And this is for your local instance, so it

shouldn't be affecting anything there. We'll have volumes, and this is probably the one that's the most error-prone for you typing incorrectly. We're going to say, store our data for this underneath our project in a .data directory and under a folder named /db. And then for our environment, we'll say use var/lib/mysql. And our last environment entry, and this is the environment of our container, I'm going to say MYSQL_ROOT_PASSWORD, and we will use pass there. And for the database, we'll say MYSQL_DATABASE, and this is the database name. We will say conference as well there and just save this. I'm going to switch over to our terminal, and we can actually run this docker-compose file.

Start Container Demo

Inside of our terminal, you can see that I am at the root of our project. I just did a print working directory for us there. And from here, we can actually execute that docker-compose file. So we'll say docker-compose. You leave off the .yml because we're actually invoking the application here and say up -d. And this will go through and create that instance and start it for the first time. So it's actually up and running now, and we can verify that by switching over to our Docker Desktop instance to see which containers are running.

Docker Desktop Demo

In my Docker Desktop, you can see any of the containers you currently have in there and that our conference container is running. So if I expand that, inside of here, you can tell that there is our container. And if I click into it, you can see the log files that tells you it was created, it was running, which port it's running on. All of that information is displayed, and it literally did all of that from that docker-compose file. So just to clarify how much simpler that was, once I got Docker Desktop installed, I literally went out, put a docker-compose file inside of our application with seven or eight lines inside of it that went out, downloaded the instance for us, set up the username and password and the port mappings, and we're up and running. We have our database set up for our application, ready to start development against.

Docker Tips

Just to give you a cheat sheet of some of the Docker commands we're going to run in this course, there is the docker-compose up -d, which is used to start up our Docker container and our image, docker-compose down shuts it down, docker ps will show you the process that's currently running, and the last two, docker inspect with pid and pid grep IPAddress, will let us inspect pieces of an instance that are currently running and if we need to do anything specific with those processes. We've got our YAML file created. We're start adding in the JPA jars to our application.

Adding Jars with Maven

The previous release of this course required us to import five different jars into it. Since we're now using Spring Boot, we can just include the spring-boot-starter-data-jpa jar that's listed here. A couple of things to note though, we don't have a version on this code. So if you've

used Maven before or it's maybe not so common for you to recognize it, there is not a version element inside of here. And that's because we inherited from the parent version inside of our pom.xml. Secondly, the Hibernate jars, you'll notice, aren't being imported here anymore. So we just have the data-jpa starter jar. Well, it pulls all of the other libraries we need in through transitive dependencies. If that's foreign to you, don't worry about it. But if you want to dig deeper, I suggest going and watching my Maven Fundamentals course here, on Pluralsight, and it will explain that greater detail to you. Let's add the spring-boot-starter-data-jpa dependency to our pom.xml inside of our application now.

Adding Jars Demo

Picking up where we left off in our IDE, let's open up our pom.xml. And we can give ourselves a little bit of space just somewhere between spring-boot-starter web and spring-boot-starter-tomcat. And let's add a dependency inside of here. And the groupId is going to be org.springframework.boot, and the artifactId is going to be spring-boot-starter-data-jpa. And when we save this, it's going to go ahead and download those dependencies in the background. I've already done it once before on my machine, so it did it quite fast. It was already cached. It may take it a minute or so for yours to download them. And a couple of things to note about this. Again, as I mentioned in the slides, there is no version here, and that's because it grabs this version from our spring-boot-starter parent version up above. So doing dependency management inside of our POM file, it grabs all of those based off of the version of the parent that we're using. If you want to verify that the libraries were, in fact, downloaded, you can expand the External Libraries section and see inside of here that we've got org.hibernate-core. And we've got 5.4.12.Final for the version of the parent that I'm using. You've also got the various JPA libraries, the Spring Data libraries inside of here. So everything looks like it got downloaded correctly for what we're trying to include inside of our application. Now let's continue configuring the rest of the database connection.

persistence.xml

A very pleasant upgrade to this course is that Spring has enabled it to where we don't need a persistence.xml file any longer. Historically, this is where we would configure database connections, some various transaction things, the persistency unit that we were dealing with, things of that nature. But we can do all of this without this file anymore. This was actually the last piece of XML that our application had in it, and we no longer need it. I leave this slide in here just to show you that if you are looking at older examples or you're troubleshooting something, this is kind of a legacy piece. If you're just doing JPA by itself, you still need one. But if you're using Spring, we no longer need this file. And that's a much nicer upgrade to not have yet another place to go hunting and pecking through, looking for some other configuration attributes. So no longer need this XML inside of our app.

Entity Manager Factory

And Spring is continuing to make life even easier for you as a developer. In the previous version of this course, we needed to create an EntityManagerFactory. It would have looked

very similar to this code above. But now Spring defaults to Hibernate as its EntityManagerFactory so you can omit creating one. Like the persistence.xml, I want to mention it in case you're looking at legacy examples, as well as to point out the declarative nature of this older code where Spring simply now just encourages convention over configuration. If you need to create an EntityManagerFactory, you can. But if the common defaults work for you, you can just simply begin coding. I also want to point out look at how we had to create the EntityManagerFactory and then begin our transactions, actually persist our code, then remember to commit it, close it, and close the factory. Very error-prone things that are more indicative of a library rather than a framework where Spring, as a framework, it takes care of all of this for us behind the scenes.

Adding MySql Dependency

I opted to separate adding the Spring Data dependency and the MySQL dependency into separate exercises, and this was to clearly differentiate that the JDBC driver has nothing to do with the Spring libraries. We are going to use the parent POM to pull in the correct version of this transitive dependency. But if you weren't using MySQL, this is also where you would choose something else like Postgres or Hypersonic, H2, Derby, some of the Oracle drivers or SQL Server drivers. So completely separate. We have our JPA libraries and then the JDBC driver that talks to the database that we are utilizing. So let's add this library in now.

MySQL Maven Demo

Inside of our IDE, let's go below the spring-boot-starter-data-jpa dependency that we just added and add another dependency inside of here. The groupId is going to be mysql, and the artifactId is the mysql-connector-java. Again, no version because it's going to inherit that from the Spring Boot parent starter jar, even though this is on MySQL jar, because it's using dependency management. So when we save this, it'll automatically download the correct version of that library for the package that we have inside of our app. And now we have all of those dependencies inside of here and can begin configuring our data source and the other resources.

application.properties Demo

With all of our libraries now in place and all the other configuration, we can begin to set up our data source, and it's actually quite simple. If we go to src, main, resources, we have our application.properties file inside of here. And we can begin by configuring using the Spring properties inside of here. So we can just say spring.datasource.url and type in here jdbc:mysql://localhost:3306/conference, and this should be the database that we created in our docker-compose file. And then we want to create spring.datasource.username, and we're just going to use root for this and spring.datasource.password=pass. And if you didn't use the same values that I did in your docker-compose, you'll want to change those to whatever values you put inside of there. Then finally, we'll say spring.datasource.driver-class-name, and we will put inside of here com.mysql.cj.jdbc.Driver, and we'll also specify the dialect for our database. So we'll say spring.jpa.database-platform. We want to set that equal to

org.hibernate.dialect.MySQL5InnoDBDialect. Make sure that you get MySQL5InnoDBDialect. If you just do the MySQLDialect, it actually won't generate tables correctly in the next module. So make sure you've got that exact dialect and then save this. And you'll notice that inside of here, all of my name value pairs are blue-green with the exception of the driver, and that's because it looks the driver up off the class path. If you have any red text here right now, you need to fix what it is pointing to because it's got something incorrect. Our URL, username, password, and dialect are all correct based off of the container that we set up.

Transactional

Transactions are enabled by default for us, and I wanted to point that out that they're there and are implicit since we are just using Hibernate. You can see in this example here that we just have to annotate our method with Transactional, and it will wrap it with a transaction. We will do this throughout our code, but we used to have to explicitly set up the configuration, and now it's just implied since we are using the standard convention over configuration.

Tips and Tricks

We're almost all the way set up in our app now to where we can go ahead and start creating our entities. But I want to sneak in a couple of tips and tricks that have helped me to learn Hibernate and JPA quicker. First is to tune the logging level for Hibernate to debug. This will generate quite a bit of logs, but it is really helpful to see what queries are being executed. Next is to tell JPA to show the actual SQL and to format that SQL Let's add this to our project, and then we're ready to start coding the business logic.

Logging Configuration

Since our properties file is actually a java.properties file and not just a Spring properties file, we can go ahead and change these parameters right inside of our application.properties. So I'm going to go ahead and set logging.level.org.hibernate.SQL=DEBUG, and you can turn that up and down to the various logging levels for whatever you want. Once you've maybe got the basis of your application set up, you'll switch it to warn or even error. Then I'm going to turn on spring.jpa.show-sql=true and spring.jpa.properties.hibernate.format_sql=true. These are really JPA properties, not Spring properties, but Spring does have an interface to those to tune those. So now that we have all these, we can save this. And when we fire up our application for the first time in the next module, it will produce those SQL log statements for us. It'll show the actual SQL that's being executed, and it'll be formatted to a human-readable format.

Summary

We covered a lot of random things in this module. We started off by looking at our database and how that's going to integrate into our application. We actually installed that database using Docker. Quickly pointed out some Docker tips if you're not familiar with it, maybe

haven't used it before. Talked about all of the legacy stuff that we could omit because Spring is following the convention-over-configuration approach. Added jars for JPA and MySQL to our application. Discussed that transactions were actually implicit, which is a key point because you used to have to explicitly include them. Then we wrapped that all up with some tips and tricks that help you learn JPA and Hibernate quicker. We are now ready to dive into the meat of our course now, which is creating our first entity and going through the repositories and all of the tiers of our application so we really can just focus on business logic now.

Overview of JPA and Creating Your First Entity

Introduction

Welcome back. This is Bryan Hansen from Pluralsight, and in this module we're going to set up our first entity and also create the database. I'm going to use some of the defaults that are currently just there by convention over configuration, and then show you how to override those for your own preferences in your data model.

Creating Your First Entity

Let's dive right into annotating an entity. The @Entity annotation is the simplest tagging to mark something as being tied to a database. For now, we're going to just accept the defaults, but we will improve upon them later. It can't stand alone though. With just this tagging, we also need to add the @Id annotation to signify which field is the primary key and also which type it is by using the @GeneratedValue. Let's switch over to our IDE and add these fields to our user object now.

@Entity Demo

I've closed all the files we had open and shut down our server from the previous demo so we're starting with a clean IDE. Let's open up src/main/java and go down into our model package and open up User. User is a pretty basic class right now, we're going to continue to add stuff to it throughout the rest of this course. We can begin by going above the class markings and say @Entity and choose the javax.persistence entity. And you'll notice that we already have a red error, and if you hover over it, it will tell you that it doesn't have a primary key designated in it. So, let's go down to above the rest of the member variables and say @Id, and choose the javax.persistence annotation and @GeneratedValue. And we need to give it a field that it can tie to, so we want to create a member variable of private Long, and we'll just name it id. We can go down below the rest of the variables and right-click and say Generate, Getter and Setter, select id, and save this. And we've got everything set up for this class to now tie to a user table, but we haven't created that table yet. Let's talk about how we do that.

Database Creation

Database creation is actually an interesting yet easy thing to do with JPA and Hibernate. By adding these two lines to the end of our application.properties, we can tell Hibernate to create the database and its tables upon startup. There are four different values that we can use here, create, which is used to create the database if it doesn't already exist and any respective tables; update, which will just look for changes upon the existing structure that exists there, but it will not remove columns if you have modified your entity to no longer have that column; create-drop, which will create and drop the schema each time the app is redeployed, and this is probably the most useful early on in your development, bear in mind it does delete any data that you have stored in your app right now; and then lastly, validate. Validate seems like it would be the least used when in fact this is what you should likely use once your app is stable to alert you to any changes. Technically there's a fifth one too, which is simply just none, but it does nothing, and you may choose that after your app has gone through QA and any other testing that you have to switch it to none. I do like validate. One thing between validate and none is that validate can slow your app deployment if you have a very, very large database as it will go through and verify the structure of all of those entities.

Database Creation Demo

Switching back to our IDE. If we exit full screen out of User.java and go back over to src/main/resources/application.properties, go to the end of our file and add in spring.jpa.generate-ddl=true and spring.jpa.hibernate-ddl-auto=create, and let's save this, and now what it will do is it'll go through when the application starts up, it will read our entities and look for the values inside of here and map them back to a database table. And it's going to do its best to guess what these types are and what they should be mapped to in the database. So it's going to look at the Long id, the String firstname, the string lastname, and the int age and map that to a user table for us.

Verify Database Demo

Let's begin by opening up our Docker Desktop. And if you're running Docker natively, you can just open up the admin app for that as well. But looking to see that your database is, in fact, running. And you can see that mine is running here on port 3306, and I just pulled up the log files. And sure enough, it is started up and is running. One common mistake I'll see from a lot of people is when they have pulled a new version down from GitHub or they've restarted their server for whatever reason, an update on their laptop or whatever, it will stop their database, and they can't figure out what's going on when they go to run their database with that create script. So make sure it's running. It looks like ours is up and everything is ready to go from that end. Now let's switch over and look at the MySQL admin to see that those tables are currently there.

MySQL Workbench Download

I didn't include a video of me installing the MySQL Workbench because there's a lot of tools out there that you can use. If you don't want to use MySQL Workbench, you could use MySQL

Admin or SQuirreL or some of the other database administration tools. I don't mind the MySQL Workbench, so I just went ahead and went to the mysql.com/products/workbench URL and click the Download Now link and installed that app, and that's what I'll be using for the next demo.

MySQL Workbench Demo

After installing MySQL Workbench, the first screen that pulls up is this MySQL Connections page. I'm going to click the plus sign. And inside of here, it's almost already set up for me. I'm going to give it a name and just call it conference. And it has a username of root. It's telling me it's going to store it in the keychain. And I can choose the default schema of conference as well, although it's not created yet, and I'll click OK. Then once I click on it, it will ask me to enter the password, and if you feel remember, we put the password as pass in our docker-compose file and choose to save that password since this is on my local machine and click OK. We're actually in the schema, connected to our Docker image that's running. If I click on Tables, there's nothing there, and so there's nothing to be expanded and viewed upon yet. But that's what we're about to change.

Run the App and Verify Tables Demo

Switching back over to our IDE, without changing anything else because we have our properties set in our application.properties, and our user.java has now been annotated, we can click the Play button or the Run button next to our Tomcat configuration for our conference WAR. And this will go through and actually create those tables in our database for us. If you watch these log files down below, I'm going to leave this window open more so you can see it, it'll go through and evaluate the database schema and see that it has not created those yet. So all of those logging statements that we put in application.properties and show sql, all of that stuff. You'll notice it says in here drop table if it exists, drop this other table if it exists, so it creates a sequence and the user table. Create those values inside of there. Establish any foreign and primary key constraints that it needs to. And it's creating those with the engine=InnoDB, which is the transactional engine. You can do a non-transactional engine for MySQL that's used for reporting. So you really want to make sure that does say InnoDB, which we specified that in our application.properties. But that is a common mistake I see people make. Switch back over to our MySQL Workbench. And if we come inside of our schema and refresh it, you'll notice that we now have our user table inside of here and our columns inside of that table, our id, age, firstname, and lastname.

Summary

This was a short module, but it was intended to be. I wanted to show you how easy it was to create an entity and have that generate your database table. We also wired up the schemas so that when we started up our app, it created our schema for us. And then we downloaded a workbench or admin tool to verify what was going on inside of our database and that it did, in fact, create those tables for us. Now I will point out all of the stuff we did was just utilizing the defaults. In fact, you may have not caught it, but it created a Hibernate sequence table inside of there, which is kind of an odd default that Hibernate does for any primary key values

inside of our application. We're going to look in the next module, and it's a big module, of how we dive in and override all of those defaults. And we're actually going to go through and use almost every annotation inside of JPA to show you its functionality in one fashion or another. So this is really the core of what we're going to cover in the entire app, and it is a very big module.

JPA Annotations and How to Use Them

Introduction

Welcome back. This is Bryan Hansen from Pluralsight. And in this module, we're going to discuss JPA annotations. This will be a larger module with a lot of demos covering a lot of the JPA annotations and walking through practical applications of how to use them.

Annotations

Annotations are a very powerful method of exposing functionality while staying within the domain. What I mean by this is that I can mark up the object that I am in with the functionality that I want it to have. Annotations aren't the only way to develop with JPA. We can use orm.xml files or straight Hibernate hbm.xml files and bind classes to tables that way as well. But the industry trend has currently moved towards developing through annotations.

Entity Annotations

We've seen some of the basic Entity annotations so far in the previous modules, so let's describe in more detail what they mean exactly. An entity declares the object as an entity and how the database should be aware of it. Basically, it declares to JPA, hey, pay attention to me. I want you to know who I am. The Table annotation describes more specific details about the database, for example, the name or the schema or how we want to access that by uppercase or lowercase, things like that nature. The Id is the identifier attribute for a very simple primary key type. You can do an Id class for a compound primary key, but it's kind of out of the scope of what we're trying to cover in this module. Then there's GeneratedValue. A GeneratedValue is used in conjunction with Id. And there are four options for us to choose for GeneratedValue. There is IDENTITY, and IDENTITY is used to specify an identity column in the database. These can be a little problematic because the ID isn't available to return back to the user until after the transaction has committed. It can also be a little slower as they cannot preallocate IDs for inserts. The auto-incrementing field in MySQL is an example of an identity column. Although they can be a little problematic as far as some of the features that you're looking for, they are simpler to use as far as just wiring your bean up and saying go. Let's just persist something now. The next option is AUTO, and AUTO defaults to IDENTITY if available. If there's auto-incrementing fields on the database vendor, it will automatically choose that one. But it will fall over to whatever is available if that's not available in that implementation. Databases like Oracle don't have auto-incrementing fields. You have to use a sequence, and so it's not going to default because that's not an option for Oracle. Speaking of

sequence, one of the GeneratedValue choices is SEQUENCE, and SEQUENCE works with a sequence if that database supports it. Now MySQL doesn't support sequences, so this wouldn't be a valid option for that. But Oracle and Db2 or some of the other database vendors that are out there have sequences, and you just plug in the sequence using @SEQUENCE generator tag annotation, and that will tie to the underlying implementation. The most portable of the options is TABLE. TABLE will work with all implementations of the database. It's a little bit simple, but it works anywhere. It just uses an identity table and column to ensure uniqueness, and you also have to tie this in using the TABLE generator annotation.

Entity Annotations Demo

Last time that we ran our application, we did not use the Table annotation, and you can see over here on the left side, and sorry there is actually no way to zoom in on the MySQL Workbench, that it created the table user, just that. Just user. It didn't specify users, which a more traditional or common naming convention for tables is somewhat plural. A lot of times you'll have your database table named users where your object is user, and that's because we can have a collection of the user objects, but the table is all of the users in our system, if that makes sense. So let's switch over to our IDE and annotate the user object to specify that we want the users table. Users is actually a reserved table name inside of the MySQL space. And so we could name our table the same, but we're going to have some collisions as for things that they use for security on the back end. So we'll actually add our Table annotation, and we'll just name ours conf_users and save this. And when we start up our server, it will go ahead and create that table in our database schema for us. Let's switch over to our MySQL Workbench and see what it created. Refreshing our schema, we can see that it has now created our table with conf_users, but we also have our user table hanging out there. Why, you might ask. Well, one thing that is a little bit misleading in the JPA and Hibernate documentation is that the create and create drop hbm2ddl auto generation doesn't drop your entire schema. It only drops tables that are currently mapped to objects. So to be honest, the simplest way for us to fix this right now is just to right-click on user and say drop table and drop it now. And now everything is cleaned up inside of our schema how we would expect it to be. The hbm2ddl auto stuff works great. It just has a few little caveats that, if you're not aware of, make it a little more difficult.

Uppercase Name

Another update requested of this re-release was how to specify naming to force uppercase. It's not an uncommon practice to have table names inside your database all uppercase. And in Hibernate, we can use the physical naming strategy standard implementation, and this just goes in our application.properties, and it will force uppercase in our table and column names. Let's add that to our application now and drop and recreate our database.

Uppercase Name Demo

To force that uppercase name, we can just go ahead and add the spring.jpa.hibernate.naming.physical-strategy=org.hibernate.boot.model.naming.PhysicalNamingStrategyStandardImpl in our application.properties. And once we save this, inside of

our user object, we can come in here and say conf_users, and this will force that to an uppercase naming convention. Let's restart our server and switch over to our MySQL Workbench. Refreshing our schema, we can see that our tables are now named how we thought they would be, conf_users, and it has updated everything. And since it was named the same as the lowercase version, it did update that to uppercase because it could alter that table that it was tied to. As we mentioned before, when we were switching complete table names, it didn't know how to bind that other one anymore, and so it just left it there, where this one will update it to the correct naming conventions.

Default Columns

In our last demo, we had columns that we hadn't specified a name for yet, but they were still mapped. It was pretty subtle, but when we created our users table, all of the other fields inside that object were automatically mapped to our database. JPA will look for the name and type of the column that the object is and make assumptions about what to map it as. We don't have to accept those defaults that are created for us though. We can use the @Column annotation, and it will allow us to override names and column mappings and specific attributes like that. Some of these only apply to the data type that we're using, such as decimal or numeric type in the database. And they actually won't do anything if you tie them to something like a string, for instance. Precision, scale, length all pertain to certain types of objects. Some of these are also just a way of applying foreign key constraints. For example, unique is just guaranteeing a unique value in that column through a constraint, and another example is updatable, whether or not we can or can't change that value in the column.

Column Names Demo

Opening our user object back up, I'll go ahead and give ourselves some space between our Long id and our String firstname and specify @Column from javax.persistence, and we'll give this a name = FIRST_NAME. And we can do the same thing with lastname. We'll say @Column, and we will give it a name = LAST_NAME, and that will get us through the example of what we're trying to show that we can have different columns named for specific member variables inside of our entity here. I do want to also point out that you can also use this Column annotation with the private Long id, the primary key associated with our object. So if you want to name your primary key something other than id, maybe USER_ID or CONF_USERS_ID, you can use the Column annotation on it as well. Let's save this and restart our server. And once it redeploys, we'll switch over to our MySQL Workbench. Let's refresh our schema. And once we expand that, we can see that we have CONF_USERS and our Columns are FIRST_NAME and LAST_NAME. So it went ahead and updated and altered the table how we thought it would, which was great. That's actually the functionality we're looking for, and you can see how we can override column names by default. I do want to go ahead and change our ddl update to create-drop. As we're doing a lot of these changes, it's often an inconvenience that something gets stuck because we've changed a value. So I'm going to switch back over to my IDE and change that right now. Inside of our application.properties, we left this as create. We'll change it to create-drop. Now I will remind you that as we start adding data into these tables, this will drop that data. But as we're making a lot of changes, you'll notice that from

time to time something may get stuck. You'll change a column that didn't exist or did exist, and you've removed it. Now it's kind of confused as to the state that it should be in. By changing this to create-drop, and if we stop our server before we make those changes, it should drop those tables, and things should not get stuck for us. So I'm going to go ahead and change mine to create-drop, stop the server, and it should go ahead and drop all of that stuff for us. And as we bring it back up next time, it'll keep recreating that exactly how we would like.

How We Use It

So we have all these pieces we've started putting together, but how do we use them all in one application? We've gone through and overridden IDs and column names, tables, got all these things set up. How do we actually start combining them all with Spring? We've looked at both halves, Spring and JPA. Well, let's put them all together in our application now. I'm going to start with the PersistenceContext. And this annotation is used to inject our EntityManager, which is how we persist things through Hibernate to our database. It gets our EntityManager for us. The next thing we need is our service tier. Now I'll be honest. You can do all of this without creating your service tier. But architecturally, this is how it's intended to be used. And this is where we put our business logic at. It also happens to be where we start our transactions from as well. And then next we have our Repository annotation or our repository tier. This is a one-to-one mapping of our model objects to our database. Trust me, this is a lot to learn or bite off all at once, but it will all make sense as we start cobbling the pieces together in this next couple of demos. Lastly, we need transactional, and that is the beginning of our transaction. And that should start at our service tier. Although if we're just doing an update against one table, it is implied that it's in a transaction because it's just one commit. It's when it spans multiple tables that you need to start and stop a transaction. Let's begin by creating our service tier now.

Service Demo

To create our service tier, I'm going to start off by minimizing and making sure my server is shut down and then right-click on our src, main, java, com.pluralsight.conference package and say New, Java Class. And inside of here, I'm going to give it the name service.UserServiceImpl. And inside of here, we can create our method. Do public User save and pass in User. Let's choose the com.pluralsight.conference.model.User. And for now, we'll return null. We'll save that. Now if we right-click and say Refactor and Extract Interface, we can create the user service interface, and it will be in the package com.pluralsight.conference.service. And let's have that also include the save method in our interface. Hit Refactor. It'll ask us if we want to make sure that that is, in fact, included and replace any concrete uses of that with the interface, which we are good. We also want to annotate this as a service, and it's the org.springframework.stereotype service annotation. We'll save that. Now opening up our controller, we go over to our UserController. And inside of here, we can now auto wire the UserService inside of here. And in our postUser method that we have down below, we can actually call our userService.save and pass in that user. We have now wired up our service tier to take our object that's coming from our post

and call our save method that's auto wired in to our ServiceImpl. We're not done yet though. We still need to do the repository on the back end.

Repository Demo

Creating the repository is almost identical to what we just did with the service with a few slight changes. So let's right-click on com.pluralsight.conference again. We'll say New, Java Class, and we'll call this repository.UserRepositoryImpl. Hit Enter. And then we'll create almost the same method in here. We're going to replace a couple things in here in another demo. We'll say public User save User user. And then we'll return that user instance as well and save this. The next thing we want to do inside of here though is annotate this as a repository instead of a service. And then we're going to right-click and refactor this as well and say Extract Interface. This will also be in the repository package. We will call this the UserRepository, not UserRepositoryImpl. We'll also opt to include that saveUser method and say Refactor, Yes, and click OK. And now we can go back to our UserServiceImpl and auto wire inside of here our UserRepository. And our UserRepository now can be used to call save on that object. So we'll say UserRepository.save user. We have all of our pieces wired up for the user repository and the user service. One thing we don't have yet though, inside of our UserRepository, is our PersistenceContext. To add that, we can just use the annotation @PersistenceContext, and we'll create a private instance of the EntityManager and store it to the entityManager variable because that's really what the PersistenceContexxt is under the annotation is our EntityManager. Now that we're in sight of our save method, we can just say entityManager.persist user. And since we are doing a persist or a save instead of just read-only data, we do need to wrap this whole thing with a transaction. So let's switch back to our UserServiceImpl, go to our save method, and we will add an annotation for @Transactional, and we will choose the one from the javax.transaction annotations and save this. And our entire code base is now wrapped in a transaction from the beginning of this method till we exit back out of this method. So our save will be persisted wrapped in a transaction. If we try and run it without this now, it'll actually give us an error. If you omit this step, you'll probably see that error.

Registration End-to-end Demo

To test our application, I've made it even easier by going out and just opening up Postman to do a post back against my server. I'm not even going through the user interface yet. So you can see inside of here, I've got a URL of localhost:8080/conference/user. And inside of there, I am putting in the params, firstname equals Bob, lastname equals Builder, age equals 44, and it does rewrite my URL to use those params. And when I click Send, it'll go through and send that to my server. Let's switch over to our IDE and see what the output was on our console. Opening up our logs, we can see that it executed that SQL like we expected it to, and it's got all the debug information associated with it. Let's verify what actually went out to our database. In my MySQL Workbench query window, if I just enter in select * from CONF_USERS and then run the query of the SQL underneath the cursor, you can see that it pulls up with Bob the Builder, and I've ran it a couple of times so I've got that entry in there more than once. If you get an error that there's no schema selected, you need to right-click on your schema and say Set as Default Schema, and then you can run it. That catches me off guard

from time to time. But we have actually wired up all of our application. Let's walk through all those pieces again. We had our Postman where we did a post against our URL, went with our firstname parameter, lastname, and age, clicked Send, which in turn goes to our controller. Our controller has a post mapping where we injected in the userService that we created, which was auto wired up above. The userService was annotated as a service and had auto wired into it the userRepository. And the userRepository had the PersistenceContext injected into it. We had to annotate our UserServiceImpl with Transactional for everything to be allowed to persist. So you always start your transactions at your service tier, and it carries through all your repositories. The reason for that is if we had two, three, four repositories injected into our service tier, that transaction could start and span across those multiple repositories.

Recap

Let's do the same thing we just did to our user object, UserController, UserService, and UserRepository to our registration. We'll go through and annotate our registration model, inject resources into our RegistrationController, create our registration service and repository, and be able to use those for that web interface. We start out by going to our Registration and annotate that as an Entity and then add our Table annotation. We can give this a name = REGISTRATION. We'll want to add an ID in here. We'll say @Id and @GeneratedValue. We can do a private Long id. Once we go below our name, we can right-click on that and say Generate and select Getter and Setter for our ID. Click OK. We'll save this. Now we can go ahead and start creating our service and repository. So let's go out to our service package and right-click on it and say New, Java Class. We'll call this RegistrationServiceImpl. Inside of here, just for the sake of showing the example, I call this method public Registration addRegistration because it is the business logic tier. It's not just about saving. There may be some business logic wrapped around adding the registration. We'll pass into that the registration. And for now, we will just return that registration instance. We're going to swap that functionality out, but we want to annotate this class as a service. And we know we're going to end up wrapping this in a transaction because we're going to save this to the database. So we'll say @Transactional. And then we can right-click in here and say Refactor, Extract Interface. I'll call this the RegistrationService and select that addRegistration method so that it gets extracted into that interface. We'll choose Refactor, and it's going to ask us to double-check if there's anything we want to change. Everything looks fine. Going to our RegistrationController, we can now auto wire in our RegistrationController the RegistrationService, and I'll save that. Going down to our addRegistration method, we can add an else on our method here. If there's no errors, we'll go ahead and persist to the database. So we'll say registrationService.addRegistration, and we'll pass in our registration and save that. And now we got to go out and add our registration repository. We'll say New, Class after we've right-clicked on our repository package. And this will be the RegistrationRepositoryImpl. And very similar to what we did in our UserRepositoryImpl, we are going to auto wire in our PersistenceContext or inject it. So we'll say our PersistenceContext. We'll do private, it actually is auto wiring behind the scenes, EntityManager, and our method inside of here will be public Registration. We will call this one save because at this point we are saving the registration. There shouldn't be any business logic that's inside of here, just persistence logic. So we'll say Registration registration, and we'll do EntityManager.persist and pass in our registration object and save

that, and we can return that instance as well. We're not modifying anything on it, but that's a good habit to get into is to return that object in case you have updated anything in there. We want to annotate our class as a Repository. And everything inside of here is set now. We'll do the same thing as we did before. We'll right-click, Refactor, extract our interface. Call it the RegistrationRepository. We'll choose to have that method included in that interface. Hit Refactor. Everything is fine there. Going back to our RegistrationService, we can now auto wire in our RegistrationRepository. And we'll choose the interface and go down to our return statement, and we will say registrationRepository.save and pass in our registration and save that. We're now ready to fire up our server, and it will go out and create those tables for us, and we can go run our web application as we do that. After starting up our server, we can go ahead and pull up localhost:8080/conference, click Add Registration, and put whatever name you want inside of here. I'll do Bryan and say Add Registration. And the page is set to refresh back to itself. If I go over to our MySQL Workbench, I can go ahead and type in select * from REGISTRATION, put my cursor over it, and select to run the SQL underneath my keyboard cursor. And we'll see that it has, in fact, put my registration in there. I ran it once or twice. You may have done so as well. So you see I've got Bryan 1 and Bryan 2 in there. That's just the names that I passed in there. So now we've got both our app wired up. We've got the web services wired up for Postman to go into. We have registration and users wired up. We can actually go through and now start looking at some of the more complex annotations of pulling back the various JPA entities based off of things like join types and more concrete examples like that.

Join Types

So far, everything we have seen and have been dealing with has been simple primitive types. How do we deal with objects or lists other than simple primitives? Since JPA is all about dealing with objects, we need to be able to bind collections of objects to the database. There are essentially four types of annotations that we can use to join objects and collections to one another. There is a OneToOne, a OneToMany, a ManyToOne, and a ManyToMany. These can be used in various configurations, such as unidirectional or bidirectional. And then we also have cascading features associated with them. The most common is typically the OneToMany and ManyToOne joins, and the most complex is definitely the ManyToMany.

@OneToMany

OneToMany is the most common of the joined type annotations. It's used to define a one-to-many relationship between an object and a list of objects. In our application, we will have a registration and a collection or a list of courses tied to that registration. We just need a OneToMany annotation on our registration class tying to our course list. We can then specify a ManyToOne on the course back to the registration. And this is all tied together using the mappedBy from the OneToMany onto the object on the opposing side. Courses will belong to a registration, or this registration has a list or many of these courses.

@OneToMany Demo

I've closed out all of the open tabs that I had just so we can focus on doing this one-to-many mapping. Let's begin by right-clicking on model and creating a new Java class, and its name is going to be Course. We're going to annotate this just like we did our registration. We're going to give it the @Entity annotation, @Table. We'll name this COURSE, all uppercase. Inside of our class, we'll also give it an @Id and @GeneratedValue. We'll have a member variable private Long id. And then let's give it a couple of other fields just to make it more meaningful. We'll have inside of here an @Column, and we'll name this name, which will just be the name of the course, and it will be a private String name. And then another column, and we'll name this one DESCRIPTION. And this is a bug I often see people introduce into their code base. They will name this field DESC, which in some database queries stands for descending. So if you're going to sort the results on something, you might sort it descending. Then we'll make this a string as well. And since we're already in this class, we'll add the many-to-one mapping back to our registration list here. This will make more sense in a minute, but to save us from switching back and forth, we'll do it while we're here. We'll say private Registration registration. We'll save that. Give ourselves some white space at the bottom and Generate, Getters and Setters. Select all the fields. Click OK and save that. Now we have our basic course object filled out here, and we did half of our one-to-many join. We did our many-to-one, which is our join back to our object. Let's go back to our Registration object now. And inside of here, below name, but above our getter for the ID, let's add an OneToMany annotation, and it is mapped by registration. And we're going to set cascade using the enum CascadeType.ALL. And then this is on a private List of Courses. So we'll pass in the object Course, and we'll name it Courses. And this is equal to a new ArrayList, and we'll save it. And let's go below our get id and right-click and say Generate, Getter and Setter. We'll choose Courses. Hit OK. And we've actually specified our one-to-many join and the many-to-one back to it. One key thing to point out here is that the mappedBy = registration, that string should tie to the name of the member variable on the course object. So you'll see we have registration over here. We have registration here. So that's saying that on the courses, our course object has a field named registration that we have getters and setters for. Tie that to that object, and that's how we keep that list, that one-to-many mapping in check.

@ManyToOne Demo

Let's add the code necessary to get our courses tied to our registration. We're really close to having this done, but there's a few things were going to tweak to make it actually easier in the long run. First of all, let's change our GeneratedValue that goes with our Id on our registration to a different strategy. So we're going to go into the GeneratedValue(strategy = GenerationType, and we want to select IDENTITY. And what this is going to do is use the auto-incrementing IDs in MySQL to automatically populate our objects. This is a nicer strategy for when we're cascading things. The other thing we were doing was working just fine. This is better for what we're trying to cascade saves across multiple objects and one-to-many mapping such as we're doing now. Let's do the same thing over on Course. Let's open it up and change the GeneratedValue here. I'll say strategy, and the identity here will be GenerationType.IDENTITY. There's an enum for that. So everything is saved there. Now let's go into our RegistrationServiceImpl, and I want to go here because this is where your business

logic should reside. So let's say that everybody who registers for our conference should attend an introductory course. So what we're going to do is we're still going to leave our registrationRepository setting here, but what we're going to do is get rid of that return for now and say registration is equal to that object that gets returned from the registrationRepository.save. And if you'll remember, I very subtly mentioned earlier that I like to always return the object. This is why. We're going to actually use that object here now. So let's get our return registration back inside of here, and let's go create a repository layer for our course. So we'll right-click on Repository and say New, Java Class. And this will be the CourseRepositoryImpl. And this is going to be just like the other ones that we did. You're starting to see a little bit of pattern here. I'll say Repository. We'll go inside of here. We'll inject our PersistenceContext, store that to an EntityManager, and then we'll create a method that is public, returns Course, and we'll call this save. It's going to take Course as a parameter, and then we will use our entityManager to persist it. And then finally, we will return our course object that we just created. So now that we have this, let's right-click and refactor and extract an interface. We'll call this CourseRepository and include that save method, Refactor, and have it look for any places that it can change. There are none. Everything's okay there. Go back to our RegistrationService. And now that we're in our RegistrationService, we can actually auto wire our new repository that we just created in here. So we'll say @Autowired private CourseRepositoryImpl, except we want to choose the interface, not the implementation. Say CourseRepository and save that. And now since we're just going to create this course for anybody that comes along, we could actually retrieve one out of the database, but this will work for our example. We'll say Course course = to new Course, and we'll just set the name on that to Intro, and we'll say course.setDescription. Every attendee the must complete the intro. And then we will now say course.setRegistration, and this is the registration we saved up above, and that will pass in registration. Now we can use the repository we just created. So I'll courseRepository.save, pass in our course, and those will now be bound. I want to point out if we go look at our RegistrationRepositoryImplementation, we are using a pass-by-reference here. Registration gets altered when it gets persisted and handed back to us to when we use it, over in our service, it will have the ID stored in there. So let's fire up our server. Our web page will pull up. We'll add a registration. And you'll see we'll have this course now bound to our registration. Now that our server has started up, I can go into our app and click on Add Registration, type in my name and Add Registration and now switch over to our MySQL Workbench, and this should all be synced up. First, I can click on our registration and run that select * from REGISTRATION and see that I've got a couple of entries in there. I've ran it a few times. And then I can see that I've got ID 1 and ID 2 since I ran it twice. If I click on select * from COURSE and run it, I can see that I have associated those correctly. I've got ID 1 or Course 1. It says that every attendee must complete the intro, and intro is the name, and it's tied back to the registration 1. And the same thing, the second one is tied back to registration 2. So I have all of my associations correct, and the one-to-many and the many-to-one mapping all synced up across my objects. Now if we deleted our registration since we have that cascade set to ALL, it would actually delete the course from this table as well. So we have everything synced up, our cascading set up, our generated strategy for our IDs synced up correctly. Everything is now wired up for this one-to-many, many-to-one join.

Fetch Types

When using an annotation like OneToMany or one of the other join types, we can choose at what time we want to fetch that data. There are two types of fetch. There is a lazy, which queries the database when that property is called. So when I call getCourses on the registration object, it's going to then go out and populate that collection. It will wait until we actually call that getter to query the database. And then there is eager. Eager queries the database when that object was originally created, and this is one of those that does vary by implementation. Hibernate does limit you to only having two eagerly fetched collections on an object. Usually not a big deal, but it is different between implementations. So let's look at what this means for our registration object that we just created. The OneToMany annotation we had mapped by registration had a cascade type all. If we tack on fetch = fetchType.lazy, which is the default, it will wait until we call getCourses on our registration object before it queries the database. Up until now, we haven't had an example to read from the database, so we don't have any code out there yet that will demonstrate this. We're going to change that now.

JPQL

Up until now, everything we have been doing has just been simple inserts. We need to do a read in order to demonstrate the various fetch types. We will use JPQL, which is the Java persistence query language, to work with the data. I believe that JPQL syntax is the hardest part for people to learn because it's not SQL. So when dealing with JPA and the different persistence APIs, Hibernate and that, the hardest part seems to be understanding that I'm working with objects and not working with SQL. People often know what they want to do in SQL and are still trying to figure out how to do that in JPQL. So JPQL is centered around objects, and this is an example of a query using JPQL. That same query written in SQL would look like a select * from Registration. So it looks very similar. Except since we're dealing with the object, we say Select r, and you can use whatever variable name you want there. But it is selecting from the object. Registration is referring to the Java object, and it's spelled accordingly. If you don't have that uppercase R lowercase e-g-i-s-t-r-a-t-i-o-n, it will look for a differently named object. So it has to be tied to the objects were trying to query on. Let's add this to our application now.

JPQL Demo

I've gone ahead and closed all of the open files that we had so we can just focus on a couple of specific things here. I decided to incorporate two smaller examples into one here, thus demonstrating the fetch types, but also creating another RESTful service so you didn't have to write a bunch of JSP and HTML code to illustrate this demo. I'm going to start out by opening up Registration, and the first thing we're going to do is go down here and add a fetch type in here. So we'll say fetch and FetchType because there's an enum for this is LAZY. Now before we leave out of here though, I do want to add one other annotation. And this is just because we chose to do this as a RESTful service. I want to say @JsonManagedReference. And all this does, very simple, is it just says, hey, when you go to export this as JSON, don't cascade and keep trying to repopulate the whole tree. Grab it at one level and return. If that doesn't make any sense to you, don't worry about it. Just know that if you omit this, you will get an

error. The next thing we need to do is open up Course, and on our many-to-one above that, we need to do an @JsonBackReference and save it. That's all we had to do to these objects. Easy enough. Now to create that RESTful service, I want to point out two things here because I don't want you to be caught off guard when you go to use this. We try to make these courses toward it's real world and real-life scenario. If you open up our UserController, you'll notice that it's annotated at the top as a RestController. That means that every method you call inside of here returns the body as a response body. If you look at our RegistrationController, it's just a standard controller. It's not a big deal, but it doesn't annotate every return type as a response body. Now as we go to create this, it'll make more sense. I'm actually going to make this easier on myself and copy this method and paste it. Change this URL to registrations and getRegistrations. This is where it starts to change. I'm going to say @ResponseBody, and it's going to return a list of Registration objects. Then we can get rid of this ModelAttribute in here because we're not trying to return that anymore. And we'll return this line down. And the next thing we want to do is say List Registration registrations is equal to, and we haven't created this method yet, registrationService.findAll. And we'll get a red error for this. We'll change our return type to registrations. So that ResponseBody annotation just said, hey, whatever I return, I want you to turn that into JSON, and that's why we had to add that ResponseBody annotation is because this is a standard controller and add that Json reference and BackManage reference in our two objects. Now let's create this method inside of our registrationService. And if I hover over it and choose the quickfix of Create method findAll in the RegistrationService, I can save this. When I open up my RegistrationServiceImpl, it'll ask me if I want to create that entity there, and I'll say Yes. Click OK. And we're going to pass this right through to our repository. So we'll say return registrationRepository.findAll. And we haven't created this method yet either. Now why are we duplicating these methods? Well, because there may be some business logic that we want to incorporate in here. And again, that's where we would do this is in our service tier. That business logic may be things like pagination or sorting or a select number of rows that we return per user, or we implement some security. All of that stuff would happen inside of this method here in our service tier, not in our repository. So let's save this, and then we'll choose the quickfix on this error and do the same exact thing. We'll say Create method findall in that RegistrationRepository. So we'll save this, open up our RegistrationRepository, and this is where we're going to go through and create that query using JPQL. So we'll say Implement methods, OK, and this, we'll leave it returning null for a second. We'll say List Registration registrations = entityManager.createQuery. And inside of here, we'll say Select r from Registration r .getResultList. And don't forget our semicolon on the end there. Now again, I can't stress enough that that Select r from Registration is referring to the object. And that's why it has to be spelled out exactly like our class name, Registration. Those two are exactly the same. So there's our select statement. We'll change this return to return registrations, and our code is actually done. We're going to go through and grab this. Now why is this a unique example? Well, a couple of things. One, we're going to use a RESTful service to do the get on it. It's currently going to do this lazy. It is also going to show how you cascade a list inside of an object through a RESTful service because that's an error that catches a lot of people off guard. And we're using JPQL. We've got all of that rolled up into one example here. So let's start our server, and then I'm going to switch over to Postman. And I've gone ahead and deleted our entire history that I had in Postman. I'm going to create a new request here. And for the URL, I'm going to do localhost:8080/conference/registrations and click the Send link. And you can see it returned our object for us, and the parent object is our registration, and inside of that is our course. And it displayed all of that. So that was all done going out to our

query. If we switch back to our IDE, we can see the SQL that was ran here, and it looks like it was around more than once. It actually wasn't. It's because we have the debug statements turned up to show all that. But you can see where it grabs our registration. And then, as we call or cascade through our object, it calls the lazy load to also display the courses when we see that. If you want to mess around with that some more, you can go into your registration object and change this to eager and run it again to see that it's pulled the very first time it comes back from the database. So you can put some log statements in and walk through all of that, but you can see the difference between lazy and eager. And we grabbed all of that using a RESTful service call that just went out and did a get.

Projection

It's not often that we want to go through like our report and display every filled and every element to the end user. In fact, some people will subscribe to the notion of never showing an ID to the end user for them to see. There's a really great way using a technique called projection that we can select the fields that we want to and build an object based off of that request. We can do complex joins and things like that. So we're going to actually do two things here in one example to show you how this works. Projection is a great way to present objects to the UI. We have an object that represents exactly what we want to show to the UI. Objects are added using JPQL syntax. So we go through and build our query using JPQL, and I'll show you how that works and what that looks like here in just a second. And then projection objects can be JPA entities, but they don't have to be a JPA entity. This is the other reason I like this, and I'll talk through that design issue here with you in a second. You can do this using a constructor for the projection. So what happens is your JPQL ends up looking like this code up above here. You have a Select new, and I've created an object that we don't even have here yet called RegistrationReport. That takes three parameters in it, three arguments. We've got name, the course name, and the description. You can also look and see that we're selecting from registration and course where the id equal to the r .registration id in the course object. So I've joined these two together and done that query. Got the information back. This is actually a lot more efficient query than what we were doing before where we had a collection of registrations and then a collection of course objects that we were going down into. If I had three or four or five registrations, it was going to run through that query five, six, or seven times. And they actually call that an N + 1 select problem. Using projection, I can go get what I need in one query and one join in the database. It's a lot more efficient database operation, and it's only showing the fields I want to to the UI.

Projection Demo

I've gone ahead and shut down my server and closed all the open files that we had just so we can focus on implementing projection inside of our application. Let's right-click on our model package, so com.pluralsight.conference, model, and create a new Java class. We're going to call this RegistrationReport and hit Enter. I'm going to add three member variables at the top here. We'll say private String name, private String courseName, and the last one we'll do private String courseDescription. And then we're going to give ourselves some white space below those, and we can right-click and say Generate, Constructor. We'll choose all three of those. Hit OK. And then we will right-click below that and go to Generate again and

choose Getter and Setter and choose all three of them again. So we've created our basic object, just a simple POJO for our RegistrationReport that contains our name, courseName, and courseDescription and the constructor we're going to use for projection. So let's save this. Let's open up our RegistrationController, and I'm going to grab our registrations method that we created in the previous demo and just copy it and paste this. I'm going to change the GetMapping to registration-reports and change this to a list of RegistrationReports. We'll call this getRegistrationReports. And the same thing down here, we'll change the list type and the return. And finally, we'll change the method that it calls to findAllReports. We haven't created this yet. We're about to. So I'll save this class, and let's do the quickfix on it and go to Create method findAllReports, and it'll take us to our RegistrationService interface. I can save this. And when I open up our RegistrationServiceImpl, it'll tell us that we're missing a method. And if we do the hot fix for it, we can say Implement methods, choose that, click OK. It'll give us a space at the bottom here. We're going to do the same exact thing. And again, I realize that we're copying and pasting this for the course. You could put some query logic inside of here for pagination or those types of things, filtering. This will work for the example that we have though. Say findAllReports. And if we do the hot fix on it, say Create method and save it and open up our RegistrationRepositoryImpl, we're going to run into the same exact thing of it telling us that we need to implement this method. So we'll choose that, and I'm going to grab this method that we had here up above and use the body of it to help us create this. So we're going to return a list of RegistrationReports, RegistrationReports. Because we're starting to scroll off the screen here, I'm going to create a string and just call it jpql. Set that equal to this and grab the body out of this method here, jpql. And we'll define what that is up above here. So we're going to set up the class name where we call Select new. And we do have to fully qualify the package in here. That's one thing that is a little bit confusing. So we'll say com.pluralsight.conference.model.RegistrationReport. And from here, we actually get to specify in the constructor arguments. So we're going to say r.name, c.name, c.description. And then let's return that down to a new line. We're going to pull all of that from registration r, Course c where r.id = c.registration.id. And then let's put our semicolon on the end of that. I'm going to give you a second to look and see how I have that typed in. And notice that at the end of each of these lines, I do have some white space. So there is a space at the end of r.name, c.name, description in between that and that closing parenthesis and the same at the end of our from clause because if not, it would be all bunched together and actually wouldn't query correctly. So make sure you have that typed in there. You can also opt to grab this query out of the exercise files or the GitHub repo where I've got this all typed out for you as well, just in case you're having any problems with it. So I'm going to say this. I'm going to fire up my server. And as this loads, I'm going to open up Postman and do a query on that new URL that we typed in here. Okay, picking up where we left off in Postman, I'm going to change this to registration-reports and just hit the Send button. And you'll notice the structure that we had before where we were pulling back a collection and then it had a collection inside of it, that's going to flatten out. So when I click Send on this now, you'll notice that we're just one layer deep. So it's a lot more performant of a query. It just gets back the fields that we want if we wanted to leave the course name out or the description out. Notice the IDs aren't showing up here, any of those things. So it allowed us to create a much more specific object for what we wanted to return. I should also note to you that this is a common technique used for searching. So one thing that people often don't realize in web applications is that your search is an object, and that's how you can save search history for people as well. So

searching oftentimes is a JPA entity, and the report back or the results back can be done very well with projection.

Named Queries

There's one last part we want to talk about before wrapping up some of the things that we've done with JPA annotations, and that's named queries. We've kind of gone through and shown you all of the best practices while going through and introducing you to these annotations. You'd rather learn it the right way than go through and learn how to do it and then find out there's a much better way after the fact. With named queries, we can clean up some of the ad-hoc SQL that's in our repository fields. One of the nice things about this is we can use named parameters and things like that as well. So it's a lot cleaner than ad-hoc JPQL or what looks like SQL inside of our objects. It's not required, but it focuses on the domain. So we can put these named queries wherever we want, but usually we store them in our domain objects. And we can use named parameters. If we're going to be substituting anything in query strings or search parameters, things of that nature, we can have named parameters that we're doing those substitution is with. It's a lot cleaner approach. How this works is we take and define our named query at the top of our object. In this example I've got here, we're in the registration object. And we take that JPQL out of our repository and add it as a named query inside of our registration object.

Named Queries Demo

To convert our object over to using a named query, let's start by creating a couple of static final strings inside of our registration object. So at the top of the class, but below the class designation, we'll say public static final String, and we'll call this first one REGISTRATION_REPORT and set that equal to registrationReport. And the name of this actually doesn't matter, but it does have to be unique. You can't use registrationReport for another named query. It's tied to this specific one. And then the second string you want to do is the actual JPQL that we're going to create. So we'll say public static final String, and we'll call this one REGISTRATION_REPORT_JPQL. We'll set that equal to a string. We're going to go grab that right now. So let's grab that code from our RegistrationRepositoryImpl and scroll to the bottom, and we're going to cut this JPQL out of here. So we'll grab that. I'm going to switch back to my registration object quick so I don't lose that out of my buffer and paste that in here and get rid of that jpql string. And just so it's more readable, I'm going to bring this down to the next line. So we've got our string inside of here and moved that out of our object that was in our repository. We're going to fix that here in a second because it's broken right now. But first, let's create our named query. So we'll add an annotation up at the top here that's NamedQueries, the plural version. And we want to add a curly brace inside of there because we do an array of these. We'll then have @NamedQuery. And inside the NamedQuery, there's two parameters. There's name, and the name is going to be Registration.REGISTRATION_REPORT. And the query is going to be Registration.REGISTRATION_REPORT_JPQL. We can save that. So we've got our named query created inside of here. We have the NamedQueries array. If you have more than one, you just comma-separate these and do another @NamedQuery annotation. We have our name, which is REGISTRATION_REPORT and then the actual JPQL inside of here. Now switching back over to

our repository, we go inside of here and just create a named query and change the name equal to Registration.REGISTRATION_REPORT and save that. Bring it down to another line so you guys can see it better. That's what we had to do to switch this over to a named query. It is compiled. It's checked and validated at the time of us creating our object so we can actually find errors earlier than if we just try and run it and have it fail. Let's fire up our server. And once this starts up, I'll switch over to Postman. And now I can just rerun the same query that we had before for registration reports, and it switched over to a named query. Really easy to do. At first glance, it may not seem like it's buying you a whole bunch, except all of these can now be passed around through static names rather than us making them up on the fly. And it keeps everything contained in our domain object. That's beneficial for one reason that I'm going to show you here. Inside our domain object, I can see the fields that we're doing our query on. So looking at this, I know that I have registration.name, I have registration courses, and I can walk through and tie that to the actual fields inside the object. So it takes a little getting used to. As I mentioned at the beginning of this, JPQL is working with objects, not working with SQL. And so you have to think of this in terms of the object graph.

Summary

So this was a big module. We covered a lot of things in here and had a lot of demos. We went over the most common and default annotations using JPA. We looked at how to override the defaults and what they implement for you. We went through the service and repository tier from Spring, and we also looked at joins and using one-to-many and many-to-one. We also demoed fetch types, eager versus lazy, and showed you some of the limits or exceptions you might face with that. And then we went over projection, one of the most powerful, but often overlooked usages of JPA in my opinion. And lastly, we cleaned up some of our code using named queries. We have one more module left, and we're going to dive into the actual Spring Data portion of using JPA and some of their convenience interfaces that they have and how that can reduce even more of the boilerplate code that we have inside of our applications. So some of that copy and pasting we've been doing, you're going to see a lot of that go away in this next module.

How to Configure and Use Spring Data JPA

Introduction

Welcome back. This is Bryan Hansen from Pluralsight, and in this module we are going to discuss Spring Data JPA.

Create vs. Update

In the last module, we had a lot of boilerplate copy and paste type code. Spring Data JPA is a framework to help eliminate boilerplate code with regards to the DAO layer. Spring Data JPA

has taken concepts learned from Ruby on Rails or Groovy on Grails and other rapid application development tools and frameworks to help simplify development.

Registration Repository Demo

In the previous module, we used the persist method of the EntityManager to create records in the database, but we never did an update. The persist method is used only for inserts, though, and we use a merge method instead for updating a record. When we created our method we purposely called it save because we were going to override this method to handle creates, as well as updates. Now, this is one of those things that we have to do in all of our repository classes, our implementations, that just is copy and paste code. Now, some people get a little creative and create an abstract parent class and move this code up into that parent class, well let's show you what that looks like. This is an example of what our goal method will look like after updating it to support creates, as well as updates. A few things to notice about this. First, we will look to see if it has an Id, so the getId equal to null, and we can determine from that whether or not it is a create versus update. Second, one other thing that's a little confusing is we don't have to flush because the merge is handled differently. Third, and lastly, the merge does return an updated object where the persist is void. This is because the problem I mentioned earlier about the various Id generated value methods when defining what Id type we have on our entity. Some of the types of Id, specifically the Identity column, aren't chosen until after the object has been saved. So this snippet of code, this image here, shows what we would update our goal repository Impl to for the save method.

Registration Service Demo

The next piece that we want to update is to open up our RegistrationServiceImpl, and we're going to do almost the same thing. And, I am making a really deep example or a harder example out of this to prove a point, and that's why we're going to use Spring Data JPA. The actual Spring Data portion is what we've been dealing with, but the JPA combined with Spring Data we haven't delved into yet. But this example is going to help illustrate that and show you why we want to upgrade these things. We're going to do the same thing here. We're going to wrap this course with an if statement. Say if(registration.getId() ==null, then let's go ahead and add that course to it and save that, if not, we're going to just skip that. We don't need to add that default attendee course to the registration on an update, we're only going to do that as part of the creation portion of this. So, illustrating where we're putting our business logic at inside the ServiceImpl, we have overridden our save method, and there is a tiny bit of business logic in there that's going to get replaced with Spring Data JPA. Let's save this now though and we'll move on to the UI portion.

Registration Controller

For the UI portion, let's open up our RegistrationController, and we'll go to the bottom of here and we'll create a new PostMapping. And the URL that we want to hit with that is registration/update. We'll follow the CRUD conventions of a RESTful service, and we're going to add in here public@ResponseBody and we will return the Registration object. We'll call this

method updateRegistration. We can pass in here an @Valid @ModelAttribute, and the ModelAttribute we'll name registration, and we'll tie that to a registration object. And I'm going to give myself a little space down here to break that on the next line. So we'll say Registration registration. And for error handling, we're not going to do anything with it for this basic example, but we could catch any errors in a binding result and, if we needed to, work on those. We have everything tied in here. Now we can go ahead and call our registrationService, and we'll still call addRegistration and pass in our registration object. And now we have this new method in here that we'll call our same addRegistration, but our addRegistration got smart and can determine whether or not we are creating a new one or updating an existing one and then passing that through to our repository tier, which also got a little bit of intelligence around there doing the same thing. Let's start our server up in case you've shut yours down and let that automatically deploy to our server, and I'm going to switch over to Postman to run this example.

Postman Registration Call Demo

We've used the conference/registrations URL already to retrieve all of the listed registrations in our system. Let's go ahead and call that again using a simple GET, and you'll see it returns a course and the associated registration with those courses from our query here. Now that we know we have a course with an Id of 1 and the name of Bob, I want to do a POST on a new URL. That's the one we just created. So I'm going to switch this to POST and go to registration/update. I'm going to put in a name value pair down here, and the first one's going to be id and the value is going to be 1. And the key is the name, and it can be to whatever you want to change it to. I had Bob in here, maybe I'll change it back to Bryan. Let's click Send. And it says that it went through clear to our database. Let's go back to our previous example here, which using Postman I already have it stored over here, I can click on this and call registrations again, and you'll see it switched it to Bryan. So all we had to do there was type in a new POST and go to registration/update, and you can actually put everything on the URL there. I didn't put it in the body, I just did it as request parameters. You can do it a bunch of different ways, but this works for this basic example, but it shows that we're updating rather than creating a new one each time. And that's the example I wanted to illustrate with this. Now we're going to talk about using Spring Data JPA to clean up a lot of this boilerplate code that we've been copying and pasting throughout our application.

Spring Data JPA

Spring Data is a wrapper for JPA. You actually need to know JPA before you can start using Spring Data JPA. It essentially replaces our repository tier unless you need some custom functionality. It's extremely powerful and eliminates a lot of our boilerplate code. That's one of the things you'll start seeing more and more common as we build more of these repository tier objects, and it's the same if you're using Hibernate. This isn't a problem that's unique to JPA. If you're using standalone Hibernate or any of these frameworks, the same copy and paste code smell, as it were, exists across all these repository tiers or DAO objects. It's the same pattern. Spring Data will help clean up a lot of this, and it can be extended for additional complex functionality. Sometimes when people look at it, they go well that's great

for simple cases, but it doesn't do anything for complex functionality. No, it can be extended very easily, and it actually isn't very complex to do so.

Maven Dependencies

We've actually already added this dependency to our project, but I want to point out that this is all you had to do to begin using Spring Data JPA. We have the spring-data-jpa-boot-starter jar added to our project, and it's just added through Maven. The other nice thing is that any incompatible transitive dependencies are already excluded by using their version. So if we add this dependency to our project using their dependency management, it's handled any of the inconsistencies or problems that we wouldn't want to import into our project. Everything is set up, and we're ready to go from the downloading or inclusion of library aspect.

UserRepository

Now that we've briefly chatted about some of the neat functionality that we get out of Spring Data JPA, let's actually see it do something. The first thing we're going to do is delete our implementation class for the UserRepository. The implementation class is only required if we need to override functionality. Our interface actually becomes our fully functional user interface. We're going to change our interface to extend the JPA interface. And then it just needs to know the class it services and its ID type. This is what our UserRepository interface becomes now, nothing more. It extends JpaRepository, the class that it's representing as User, and the ID type as Long.

UserRepository with Spring Data Demo

Let's update our UserRepository to now be a JpaRepository. To do so, we just open up the interface and say that it extends JpaRepository, and it's used to service the User class, and its id is a Long. From here, we can actually delete this method because it's going to inherit methods from the JpaRepository interface. The other thing that we can do is go in our UserRepositoryImpl, right-click, and delete it. Click OK. And our application already knows that this is a JpaRepository because we have extended that interface. But I am going to give it the annotation of @Repository up here, and this is how you also would override the name if you want to give it some other registered bean name. This will work for us. We can save this. If you open up your UserServiceImpl, you can see that we don't have any compile errors because that method save is already implemented for us in the JpaRepository interface that we extended. That's it. We have no other functionality here. We will add functionality to one once to show you how to customize this. But the default CRUD implementations are already here for us.

RegistrationRepository

The RegistrationRepository is very similar to the UserRepository in what we need to do to make it available to use Spring Data JPA. One thing to note though is in the previous version of this course, there was no support for projections or using named queries the way that we

have inside of our RegistrationRepository right now. Both projections and named queries have since been added to this functionality, and that's why I did this object next is because we're going to add all of that functionality in there. All the customize stuff we did, we're going to convert over to Spring Data JPA.

RegistrationRepository with Spring Data Demo

Let's open up our RegistrationRepository. We'll start off by extending the JpaRepository interface, and that's going to use the registration object that it represents, and its id is Long as well. And we'll save this functionality. We can already get rid of this save method because we already have that through inheritance on the JpaRepository interface. And we actually have the findAll method as well. What we don't have is this findAllReports. Let's convert this over to using that named query that we created now and replicate that functionality.

NamedQueries with Spring Data Demo

Switching this over to use the named query and the projections that the named query is utilizing is actually easier than you may think. There is one little caveat though. We have to go to our registration object and change the name of our named query. You'll remember we used this public static final String to name our named query in this definition of above. All we have to do is come in here and specify the domain. So we'll say Registration.registrationReport. Now since we're in this RegistrationRepository interface, that literally becomes the name of the method now, RegistrationReport. We'll save this, and that's actually our named query. So the rest of the functionality that we had inside of our RegistrationRepository, the save, findAll, and findAllReports can actually all go away. I'm going to delete the implementation, click OK, and our RegistrationRepository has a call from the RegistrationService that will now just be registrationReport, and we can save that. We have all of our code switched over to now using the JpaRepository interfaces, and this is the extent of what our repository tier looks like now. Let's fire up our server and see what this looks like from the user interface side of it.

Spring Data JPA Recap

Switching back over to Postman, if we run our getter on the registrations URL again, you can see that it still works correctly, returns that one course that we expected to be in there tied to that one registration object. If we go to our request for the registrations reports that used the named query and the projection and click Send on that, you can see that we get the projection model back of just the name, courseName, and courseDescription without the id, but this was also using that named query. We can verify that our post is working correctly as well by going back in here, and we can change that back over to Bob or whatever you want to and click Send. If we switch back over to our registrations and get those again, it's updated. So all of that functionality is working now how we expected it to. Looking back at our IDE, our UserRepository is empty. All of the basic CRUD functions and the basic searching functions are available to us by just extending this interface. Same thing with our RegistrationRepository. That logic surrounding whether the ID was there or not and how to do those things, even the use of the named query, we named our method the name of the

named query. We just had to specify the domain. You remember, if we look at our registration, we had to say registration., which just said registration needs this domain assigned to this named query. We have cleaned up all of that code, eliminated two classes, and condensed it all in one, still using injection to auto wire it and deployed our entire app without having to do anything more than add one line to one interface inside of our repository tier.

Summary

In this module, we finished up our configuration by enabling Spring JPA and going through and having all of the configuration set up to run that in conjunction with Hibernate . We finished out are examples of the update to round out all of that development, as well as converted our UserRepository and our RegistrationRepository to fully utilize the JpaRepository interface provided to us by Spring Data JPA. This actually concludes our course, and I want to thank you, and I hope you appreciated this update. And I thank you for sticking with us clear through. Thank you.