Course Overview

Course Overview

Hi everyone. My name is Bryan Hansen, and welcome to my course, Building Applications Using Spring JDBC. I'm the Director of Development at Software Technology Group and now a seasoned author on Pluralsight. Interacting with the database has always been harder than it should be in Java. Spring and the Spring JDBC framework give us a great alternative and make it quite easy to access our data. In this course, we're going to set up a database in a framework to fully utilize Spring JDBC for accessing our data tier. Some of the major topics we will cover are all of the CRUD functions of data access and retrieval, as well as batch operations and exception handling and transactions. By the end of this course, we will know how to effectively use Spring JDBC in our application. Before beginning this course, you should be familiar with Java and an IDE such as Spring STS or Eclipse. I hope you'll join me on this journey of learning how to Build Applications Using Spring JDBC, at Pluralsight.

Introduction

Version Check

Introduction

With Hibernate, JPA, JDBC and other persistence frameworks available in Java, are you wondering why you would use Spring JDBC? Maybe you know why you want to use it, but aren't sure where to start. Hello, I'm Bryan Hansen and welcome to my course on Building Applications Using Spring JDBC on Pluralsight. In this course we are going to be learning about integrating a data layer using Spring JDBC. We will discuss why you would want to use Spring JDBC over some of the alternatives and we will deep dive into configuration and various CRUD functions, as well as cover how to handle exceptions gracefully while using the Spring JDBC framework.

What Is Spring JDBC?

Spring JDBC is a framework for working with the standard JDBC API that Java provides. JDBC by itself is very error prone and not really user friendly. Spring JDBC is also pattern-based, relying heavily on the template method pattern where you just need to focus on the portions of the code that deal with the business logic. Spring JDBC also builds on top of Spring and its dependency injection libraries. Templates that we create can be injected into other classes very easily. It's also an ORM, much like Hibernate and JPA. Using Spring JDBC still encourages the use of an object. I would argue that one of the best reasons, though, for using it is that it works very well with existing database structures where many of the other ORM tools struggle in this regard. Let's look at some of the particular problems that Spring JDBC solves.

The Problem

What problem is Spring JDBC trying to solve? Whenever I'm going to introduce another framework or tool into my code base, I want to look and see what this is actually buying me. These are the big questions I ask, what is it trying to solve for me? Well Spring JDBC fixes quite a few problems over standard JDBC and is simpler to work with than an ORM like Hibernate and JPA. The biggest thing it fixes, and for me this is reason enough to use it, is the reduced complexity of my code base. Spring JDBC code is much simpler than standard JDBC. And although not a full featured ORM like Hibernate, it still encourages the use of object oriented programming in its design. Since it's written using the Spring framework, configuration information is injected into our application, making our application more portable. And overall I like the simplicity it brings and allows you to focus on the business needs and not the extraneous code of closing and opening connections or dealing with various JDBC problems.

Business Focus

I used this exact same example in my Spring fundamentals course, but let's talk about business focus for a minute. If you've ever done much database development in Java, you've used JDBC. And JDBC at first glance doesn't look that bad, but then you really start picking it apart and it's really not pretty. Look at all this extra stuff we have in our code, just to do a simple select statement. We have connections and prepared statements and result sets, and then we have this try catch finally block and we have a driver manager. We have all of this stuff in there and look at that great big finally block of if rs = null, if statement = null, if connection = null, and then we also have these empty curly braces just catching exceptions. There's a big bunch of just code in here when really all of our business cares about is this single line right here where we say select whatever fields are in the CAR table where this ID = whatever, and then we only grab the other line that our business really cares about, and that is the actual string, and store it in a Java object and return it. Now this may look like a great selling point for Hibernate instead of Spring JDBC, but I'm actually going to show you that Spring JDBC handles this example really well. So all these places that we're assigning things and handling stuff and closing all these connections can easily be handled by Spring JDBC for us.

The Solution

So the solution we're looking to get out of Spring JDBC is that we can remove configuration code or lookup code and developers can focus on the business needs. Like I said, that's a big one for me is that the business doesn't care that I have a try catch block or how I handle this exception, they care that when we ask for a CAR we get that CAR back. Our code can also focus on testing. So on that previous slide, it really wasn't that testable as far as I have things hard coded in there, I have a driver manager that I'm getting a connection, where's that connection coming from and that type of information. Spring JDBC is built on top of Spring so we also indirectly have the power of Spring using dependency injection. If we want to annotate our code rather than having so much configuration code in there, Spring can help us with this as well. So make things easier to test, make things focused on what the business needs, and remove the configuration lookup code through doing annotations or XML based development. Let's look at that JDBC code again and compare it with what it could be.

Business Focus Revisited

Let's look at that business focus slide again. That JDBC code that we looked at earlier has all of this getConnection and DriverManager, prepareStatement information, that huge finally block that essentially does nothing but close connections for us, and now here's what our code could look like using Spring JdbcTemplate. Notice that nowhere in here is it mentioned that we have a connection or a result set or a prepared statement, DriverManager, it doesn't talk about closing stuff or how we handle any of that extra code surrounding us doing that select statement. We have our basic find method and return statement, and that's it. This is a great example of the template method pattern and how Spring JdbcTemplate utilizes that template method pattern to interact with our code This may scare some of you that there is a black box with Spring and Spring JdbcTemplate, and I hope to alleviate some of those concerns, but look at how much smaller and more concise this code is, and it really is just doing what we want to do. There's no reason we should have to rewrite what's on the left for every query we need to make on the right.

Summary

In this short module we talked about what Spring JDBC is, the problems that we're trying to solve with it, and what the solution is, what we're essentially getting out of using it. We also took a before and after look at the business focus. Business focus is what you are going to see me drive towards in all of the examples that we do, how this enables us to get what the business wants done faster. Let's get the project set up now and dive into using Spring JDBC.

Setting up the JDBC Project

Introduction

In this module we are going to walk though the environment installation and get everything ready for the remainder of the course. By the end of this module we will have installed a database, created a schema, and configured our IDE and project for development.

Prerequisites

There are a few prerequisites you'll need to do the examples in this course. I am using Java 11. Java 11 is the most recent long-term supported, or LTS, version. There have been other releases since then, but 11 is the most current LTS. With Spring 5, you're required to use Java 11 or later, but most, if not all, developers are now using it for Java development. I am assuming that 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. I've been asked about Spring Boot and I like it for a lot of situations, the feedback that we often receive though is it's too much of a black box for developers learning new technologies. Once you know how the underlying pieces work, it's a great tool. Spring no longer offers up a way, though, to just download the JARs and you're forced to use Maven or Gradle to download those dependencies. There is a Pluralsight course on Maven. If you have more questions about that, you can go research and follow it. The next thing you'll need is an IDE. I'm using Spring STS, which is just a flavor of Eclipse. In fact, it's just rebranded Eclipse. If you're already using Eclipse, you can go ahead and just keep that and it'll work just fine, but I like Spring STS because it has a lot of the common plugins installed and you don't have very many, if any at all, configuration issues. The steps for Eclipse and Spring STS should be nearly identical, if not the same. For our web container, we're going to use Tomcat 9. At the time of writing this course, 10 is still a release candidate, so 9 is the stable version. And lastly for our database, we're going to use MySQL. MySQL has great tooling and is pretty user friendly.

Demo Outline

For our next few demos we're going to walk through and show you how to get the source code in your IDE. There are samples in the exercise files and samples in GitHub. They are identical and if you're not familiar with GitHub, you can go ahead and download the exercise files and we'll talk more about that. The app that we're going to be building is a microservice based application that is fairly a headless app with some client code to show you how to make those calls. We have chose this approach in case you might be wiring up your application with a UI like Angular or React to talk to microservices. And then lastly for our demo we're going to walk through the database setup and show you how to get MySQL and a schema created.

Exercise Files

Because not everyone is comfortable with GitHub and downloading source code that way, I'm going to first talk about that you can now download all of the exercise files right from Pluralsight. Pluralsight subscriptions now include access to everyone that was previously only available to premium subscribers. You can download the exercise files by visiting the page for this curse and clicking the link for Exercise files. Setup instructions are included in the download and you won't need to work with GitHub. But if you prefer GitHub and want to track your own changes as you go through it, I will show you an alternative and download the course with GitHub if you're already comfortable with that approach and possibly want to go through tracking your own revisions as you go through the changes of this course.

Github

As we just mentioned, I was going to show you two ways to get the project and the first way was just going through and downloading the exercise files from the Pluralsight course Exercise Files section and the second method is downloading from GitHub. I will let you choose which approach you want to take, but the end result will be the same. To set up the project from GitHub, you first need to fork the project to your own repository. Second, I have made a shortened URL for you to access the project, that will take you to the fully built out project on GitHub, and once you've done that we'll walk through the setup of the project. Let's dive into that demo now.

Github Demo

To get started downloading the project from GitHub, I've gone ahead and navigated in my browser to github.com. In the previous slide I gave you a shortened URL, and this is the place that it resides at, github.com/bh5k/ride_tracker. Now, you will want to go ahead and fork this to your own repository so you can track your changes with it, and then from there go ahead and clone it. So I'm going to grab the clone URL and I'm going to go ahead and go to the GitHub client and add a new repository. Now because I have this synced over to my repository, it's here available for me to already clone. If you don't, you're going to have to go ahead and enter the URL. So I'm going to click ride_tracker and I'm going to tell us the directory that I want this to go to. I like to put my projects under C:\dev\workspace, so I'm going to navigate to that, and I'm going to go ahead and click OK. Now it has my project all synced in there. We can minimize this, and now I can open up the Spring STS instance. As this loads, it's going to ask us for that C:\dev\workspace again. And now when I come in my project, it will go ahead and be able to import that project from my file system. There's a bunch of different ways you can do this. I like using the external source tool for GitHub that shows you the changes that you have in there. You can do this directly from inside of Spring STS, but I can now go ahead and say Import, Existing Project into Workspace through Maven, so an existing Maven project, and it will ask us for the root directory. We're going to look for our C:\dev\workspace\ride_tracker, and we'll see that it finds our POM and asks us if we want to go ahead and import that. We'll click Yes. As this goes through, yours may take a minute the very first time it does this because it will have to download some libraries and things that your Maven library and repository hasn't been updated with. But once you have this set up, you should have a src/main/java directory, src/main/resources directory, a src/test/java directory, as well as a src/main/webapp and WEB-INF directory for configurations. From here, we can go ahead and create our server. Now, all of this I have covered in previous courses in my Spring MVC course and Spring Fundamentals course, so I'm not going through all the step by step of it because you should be familiar with that at this point in this course. Let's go ahead and add a new server, and I'm going to choose a Tomcat 8 Server, click Next, and add my ride_tracker application to this and click Finish. From this point, our application should be ready to be deployed and ran. I can go ahead and click the Start server button. We can verify that everything inside of here looks good. I'm going to make this full screen for a minute to see. I don't see any errors in here, so it looks like it's deployed correctly. And I have created a small test under src/test/java for this RestController. Inside of here, this just does a basic call that's currently not connected to our database to verify that our localhost is returning a get on our RidesController. So it's going to go through and pull this back and pull back any rides we have, and right now I just have some hard coded information inside of here. So let's go ahead and run this as a unit test. We can just right-click on it and say Run As and JUnit Test should be one of the options that comes up for you. Once I do this, you'll go through and see that it's loading up this runtime JAR the first time, and our output should come back green just as it did. If this is the first time you've ran your application and it times out for whatever reason, you may need to run this unit test again because it does have a timeout built in there of 1 full second. And my console will show that it returned the ride named Corner Canyon. We'll dive into what this functionality is, but our application is now downloaded from GitHub, imported into our local Git repo, and imported into our application as a Maven project now. So everything should be configured here. Now let's set up our database and get things tied to this so we can start building out our connection pool, as well as our first instances of our create statements.

MySQL

Now that we have the source code configured, we're going to install the MySQL database and then the MySQL Workbench. There are a lot of databases to choose from, and for this course we chose MySQL. MySQL is available on every cloud hosting service that I've ever used and is a great, full-featured database. This course is not a how-to on MySQL though, but if you don't have experience with it, don't worry, we will cover everything that we need to so that you can be successful while using it.

MySQL Install Demo

To install MySQL, you're going to want to open up a browser and navigate to www.mysql.com. You're going to see a landing page like this. The first thing we want to do is go to the Downloads tab and scroll to the bottom of there past their commercial offerings to the GPL edition, or the Community edition. I'm going to select that and go to the MySQL Community Server. There's various offerings that they have, such as their clustering capabilities and fabric and router. We just want the MySQL Community Server. Now there's a few options or ways that you can download MySQL, and we're going to use the MySQL Installer that's web-based. You'll see options to download the complete bundle. We're just going to choose this MySQL Installer, and it will take us to two options, the 1.7 MB version and the 386 MB version. If you're choosing the web version, you know you're going to be connected, which is what I'm going to do, then look for the smaller download. It may change in size with a future release, but right now that's 1.7 MB for me. I'm going to click Download, and it's going to try to get me to sign up for a free account or log in if I have an existing account. You can do this if you want updates and notifications or access to their forums, but I'm just going to go ahead and say no thanks and start my download. This will go ahead and download that web installer, which since it's small it's a relatively fast install. We're going to go ahead and choose Run, and when this comes up it will give us the option to install just the custom installation that we want. The developer default isn't bad, but the custom installation will allow us to choose just the couple of pieces that we care about. So I'm going to choose Custom and Next, and then I'm going to navigate down to the SQL instance of just the server, so MySQL Server 5.7.17, and add that, and then I'm going to add the MySQL Connector and only the MySQL Connector/J. And then underneath Applications, we want the Workbench as well. So I'm going to choose the MySQL Workbench. So those three files, the MySQL Server 5.7, MySQL Workbench 6.3, and the MySQL Connector/J 5.1, and go ahead and click Next. It will go through and download these. It'll take it a second. MySQL Server is the largest of the downloads so it'll probably the last one to finish, but once this is done we can then go ahead and start installing these and configuring them to run inside of our application. The configuration for these will also prompt you if you have a previous version installed so that you don't potentially write over any databases that you may have out there. Once MySQL is done installing, we'll click Next and it will ask us if we want to configure this instance, which we do. So we're going to click Next. Now, I'm going to choose the defaults here. The configuration type is Development Machine and the port number is 3306. If you do not want to run on that port for some reason, make sure you write it down because when we create our data source in the next demo, you will need to know that port. So I'm going to leave it as the default of 3306 and click Next, and then I'm going to add a root password of just password. Since this is a development machine and it doesn't have any sensitive data on there, I'm just going to use password and password. I know it's weak. It's fine because there's nothing sensitive in there. I'll click Next. Then it will allow me to choose this to be ran as a Windows service and to start that service at system startup, which I'm going to click Next on that as well. I don't want to use this as a document store, so I'm going to leave that unchecked, and click Next and Execute. Now that this is completed I'll click Finish, and it will allow me to go through and start the MySQL Workbench after setup. We'll click Finish. And this will go through and launch a local instance for me, and if you've used the MySQL Workbench in the past, this has changed quite a bit recently. It has this Local instance, I'm going to click on it, and it will bring me into the Workbench. From here, I can go over in the Schemas navigator and right-click and say Create Schema, and I'm going to create one called ride_tracker. You can choose different collations. If you are aware of what those collations are, then you already know what you're looking for, but you can choose if you want UTF-8 or some other character set. I'm going to leave it as the server default and click Apply, and it will create that schema for me. Click Finish. And for now, there's no tables inside of here. I can click on all these, but they're all empty. So I have ride_tracker and Tables, Views, Stored Procedures, Functions, but there's nothing in those. If you used a different name other than ride_tracker, just like that port, you'll want to write that down because as we go through and configure our JDBC URL in the next demo, you're going to need that information as well.

Configuration

Now that we have our database downloaded and installed and the Workbench, we've also downloaded the code sample from either GitHub or the exercise files, we can now go through and configure the rest of our application. We're going to start with adding a couple of Maven dependencies, the spring-jdbc JAR and the mysql-connector-java JAR. From there, we're going to go through and edit the jdbc-config.xml by adding a dataSource and the JdbcTemplate. And then finally, we can go through and update our RideRepositoryImpl to include an autowired reference to our JdbcTemplate that we created in our jdbc-config.

JdbcTemplate Configuration Demo

Now that we have downloaded our application from either GitHub or from the exercise files underneath the URL for this course, we can open up our pom.xml file and select the Dependencies tab at the bottom. We need to add two dependencies inside of our application here. Let's give it a group ID of org.springframework and an artifact ID of spring-jdbc. The version should match your spring-webmvc version of 5.2.7.RELEASE. And click Save on that or OK, depending on if you're on a Windows machine or an OS X or Linux box, and that will automatically download that JAR for you. We're going to add one more dependency inside of here. We can click the Add button again and give it a group ID of mysql and an artifact ID of mysql-connector-java and a version of 6.0.5. Click OK. Both of those resources should now be downloaded automatically for you, and you can double check that by looking at your Maven Dependencies and see that we have the connectors and JDBC pieces inside of our application. It will take it a minute to download if you haven't looked at yours yet. That's why it's always a good idea to check this. And so now we have all of those features inside of our dependencies. We can now begin configuring the XML to talk to our database. I'm going to close the pom.xml file and even minimize the Maven Dependencies portion and open up the

src/main/resources jdbc-config file. I'm going to create two XML beans inside of here. I'm going to make this full screen so you can see it a little bit better. And it's a lot of typing. You can grab yours out of the XML section inside of the exercise files or out of GitHub, or you can type it in as well. I'm going to type it in and and explain to you what these two beans are. The first bean is a dataSource, and that's what ties us to our DriverManagerDataSource or our database connection. They're kind of synonymous for one another. We give an ID of dataSource and the class is case
sensitive, org.springframework.jdbc.datasource.DriverManagerDataSource. And it has four properties inside of there, the class name of com.mysql.jdbc.driver, the URL to our database. If you look at the end of that URL, localhost:3306/ride_tracker, that is the schema we created in the previous demo. It has a username and password of root and password. And then on to the second bean. It's called jdbcTemplate, and this is the bean we're going to inject in our code and use inside of our code from here on out. And it has a property by name of dataSource where we pass in the reference of dataSource. That is the dataSource we just created just up above here. So, those two beans reference one another. Let's walk through our application though and finally run a test. We can exit out of full screen mode on our jdbc-config, and when you downloaded the source code you had a couple of packages inside of here. We've already started going into them. We've got our controller, model, repository, and service packages. We start with our controller. We're going to see that we have our RideController, and this is the entry point for our application. It's where our URL is going to go and hit. And this has injected into it a RideService, and that RideService is where our business logic should reside. So we have an interface and an implementation. Let's open up the implementation, and you'll also see inside of here that there is a repository that gets injected inside of here. If we open up our repository, we're going to see the same thing, the interface and the implementation. And inside of our implementation, this is where we will put the SQL to interact with our database, but we can start by injecting that template that we just created. So I'm going to add an @Autowired private JdbcTemplate, Jdbc camelCase'd template object inside of here. And I'll save this, we're not going to use this just yet, but it's set up to talk to our database. We literally just need to start writing our JdbcTemplate queries here, and that's what we're going to do in the next module, but let's run a unit test against this. So, everything's up and running here. We can make sure our server is started, and mine is up and running, and we can go to our src/test/java RestControllerTest, and if we right-click on this and say Run As, JUnit Test, which should be one of your options, you can see that it will go through and run and it executes and comes back to the screen and the Ride name: Corner Canyon is there, which is exactly what's being returned from our ride repository. So we just did a bunch of stuff there. Let's look at that really quick. Let's kind of recap what we've gone over. So, we started off by adding some dependencies to our POM and we had our Spring JDBC dependency, as well as our MySQL connector. From there, we went to our jdbc-config and added a dataSource and our jdbcTemplate beans. We walked through our controller, which has the RideService autowired into it, the service, which has the repository wired into it, and then we added our JdbcTemplate to our rideRepository as well. And then we went ahead and executed our unit test, and that went through all of the tiers of our application, it went clear through our web service and our web front end, clear to the back end and back out to our unit test and made that roundtrip. Successful roundtrip, whether it was localhost or hosted on a remote server, we actually made that complete roundtrip lifecycle.

Summary

In this module we set up everything to build out the rest of the course. We downloaded the source and got the project set up, we showed an alternative to download it from GitHub and walked through setup using that approach. We downloaded MySQL and MySQL Workbench and finally we created the remaining configuration by adding dependencies, adding our jdbcTemplate configuration and injecting the template that we created. The next step is going to be creating records in the database, and this will be the start of the CRUD functions for our application.

Creating Records in the Database

Introduction

In this module we are going to walk through creating records in the database. We are going to create a unit test, modify our controller service and repository tier, and then use the JDBC template API to insert records into the database.

CRUD

In this module we are focusing on the C of the CRUD functions or the Create. There are a couple of ways in which we can create records using Spring JDBC. We can execute an Insert statement using the Update method of the JdbcTemplate. The JdbcTemplate looks just like a database insert that you would execute from a SQL Workbench or another SQL editor, or another option is the SimpleJdbcInsert. Using the simple insert we don't actually write SQL, but rather setup the call. It is a little more code, but it acts a lot more like an ORM. We will look at both approaches and you can see which one you prefer.

Demo Outline

This demo is going to be broken into four parts. We're going to start off by creating the database table in our MySQL database. Then we're going to create a unit test and a method that the test will call in our controller. Next we're going to create the call in our service tier and finally we're going to create the call in our repository where we will interact with the database tier using Spring JDBC. I did the demo this way on purpose so that you could refer back to these examples independently of one another in the future. Specifically you can just view the video where we create the database table or the actual repository call by itself. Let's look at creating our table now.

Create Database Table

To create the database table, we are going to use the MySQL Workbench Table Editor. All the editor really does, though, is help us through a wizard approach create the script similar to what I have here. Let's open up my SQL Workbench now and begin creating our database table.

Create Database Table Demo

To create our database table, let's go ahead and open up MySQL Workbench. I wanted to do it from a fresh start-up, so in case you'd forgot how to navigate to that workbench. I'm going to go down to my Start button and click MySQL Workbench. And it will pull up this workspace desktop and inside of here you'll have one called local instance that doesn't really look like a button, but you can click on it and it will actually launch you into that workbench. Once we're inside of the workbench, just for good measure and in case you hadn't set yours before, right click on your ride_tracker schema and say set as default, and you'll know it's default if the name of that schema is bolded, such as mine under ride_tracker. From here we can right click on the Table item underneath ride_tracker and say Create Table. And this is going to bring up this editor for us. First let's start by naming our Table Name ride. And then underneath here we can start adding columns. I'm going to add three columns. The first one is id, and we'll go ahead and say that this is a primary key and it's auto incrementing, so they don't do a great job of telling you what these checkboxes are without hovering over them. So we're going to check the primary key column, the not null column, and the auto incrementing field or AI column. The next column we want to add is the name of our ride. So we'll call this name and it's defaulting to a VARCHAR, so we can go ahead and edit the size of that and say that we want to do a VARCHAR to a size 100 and we'll also select that as being not null. Then we're going to add a third column and we'll call this one duration, it's just the duration of what our ride is, and we'll make this one an INT. We can use the dropdown, choose INT, and then we'll also set that one as not null. Now that we have everything defined in here, we can go ahead and click the Apply button and it will create this schema for us, and really all this does is take that schema and execute it behind the scenes, but we can go ahead and click the Apply button right here. You'll see a successful message saying that that script was ran and it was applied to our database. We can go ahead and click Finish now. Now we can close that Table editor and you'll notice underneath here that we have a ride table and you can go inspect the columns if you want to. You can also go ahead and do a select statement here using a select * if you want, but suffice it to say your table was now created inside your MySQL instance. Now let's go ahead and start writing the code to do our unit test and call our controller.

Create Test

Using the RestTemplate in Spring, it is quite easy for us to build a test that calls our RESTful service in our controller. This test on this slide will create a ride object and then call our controller, passing the ride in. The template automatically handles the object by issuing a put call. Let's open up our IDE and create both the test and the endpoint for our test to call.

Create Test and Controller Demo

To create our test, I want to go ahead and open up the unit test that we have underneath our src/test/java directory called RestControllerTest. Currently we have our GetRides test that we had in here from before. I'm going to go ahead and just add another one inside of here. So, I'm going to start off by just grabbing this code and copying it in here and paste it for us to have a starting point for our testCreateRide method. I want to walk you through the RestTemplate for doing a put to here so you can see the object that we're passing in so you more conceptually understand the whole test that we're trying to write. So the first thing I'm going to do is, just like our other one, create an instance of the RestTemplate here. The difference this time is that we're going to create an instance of a ride. So we're going to say Ride ride = new Ride, and we're just going to use the default no arguments constructor, and then we're going to go ahead and say ride.setName, and we'll call this the Bobsled Trail Ride. And then we also want to set a duration, so we'll say ride.setDuration, and we'll say that it took us 35 minutes to do. So I just put a 35 in there. Now, we can call the RestTemplate's put method. So we'll say restTemplate and we just want to put, and we're going to enter in our URL. So we'll put in a string here and say http://localhost:8080/ride_tracker. Now if you named your project something different, you'll want to change this context, of course, and then we'll call the ride controller. Then the next thing we're going to do is pass in our ride object and the put will automatically call that for us. Now we currently don't have this endpoint in there, we need to go ahead and modify our controller to do so. So I'm going to go ahead and open up our RideController. And right now it's quite simple, we just go ahead and have our get for all of the rides in there. I'm going to paste in this code that I have written previously just so you don't need to watch me type all of this in, and we'll walk through what it does. So, I have a request mapping for /ride and a method type of PUT, and we go ahead and we call this and it takes a request body in, which is the object that we passed in from our test. Currently this just returns null, but it's going to eventually call our service and return an object. So let's save this. To test this, we can go ahead and start our server back up, make sure that there's no deployment errors there, everything looks fine, and we can go into our test that we've now created and right-click on testCreateRide and say Run As, JUnit Test, and we should see that the server then gets the call for deploying it and we have our unit test that also gets ran. And we can check that by looking at the tab for the unit test and see that it came back green. Let's go ahead and now write the code to talk to our service tier that will eventually call our repository now.

Modify Service and Repository Demo

To flesh out the rest of our code through the various tiers, I'm going to start with first going to our RideController. Inside of our RideController we're currently returning null. I want to go ahead and say rideService.createRide and pass in our ride object. Now you're going to get a red squiggly line because this method currently does not exist. So let's go ahead and open up our RideService interface. You may want to stop your server if it's going to continue to try and redeploy that context. I've stopped mine. I'm going to go into my interface now and say Ride createRide Ride, and pass in that instance. Now I want to mention again, because I get asked this quite frequently, and this is why I walked through these steps, do I have to have this service tier? The answer is no, you do not have to. We could go right from our RideController and go right to our ride repository. We could honestly even pass in our JdbcTemplate to our controller. Do I recommend that? No. You want those multiple tiers inside of your application so that we have some flexibility, as well as coding to an interface as you'll see in a more complex example we're going to do later in this course. Let's go into our RideService. We have our interface now created for Ride createRide, and you'll see you have a red x under your RideServiceImpl. We're going to do the same thing and go into our

RideServiceImpl and add an @Override and create this method. We'll just say public Ride createRide and pass in that ride instance. And we're going to do this exact same thing for now. And you're saying, why would I want to just do the same thing over and over? Because we're going to expand this example later. We'll say return rideRepository.createRide, and it's going to error out on us because we don't have that method currently created. We can save this and then go ahead and open up our RideRepository and create the same method signature there that we did in our RideService. And when we save this, it will transfer that error to our RideRepositoryImpl. Now inside of here we're going to do the same thing, but this is where we're going to finally go through and begin editing our database where we're going to incorporate the JdbcTemplate that you can see we've created right above. So let's go ahead and say @Override public Ride createRide, pass in that ride instance again. And in here, for now we're just going to return null. We'll go through all the JdbcTemplate features here in just a second. But we should be able to go back and start our application server up and now go to our unit test and right-click Run As, Unit Test and start this, look at our test results, and you'll see that we have a green bar still. So we went ahead and created the test that called our endpoint in our controller of the createRide that's going to go ahead and interact with our service tier through the interface and then through the implementation, which ultimately ends up calling the RideRepository for this createRide method that we're now going to go ahead and utilize the JdbcTemplate in.

JdbcTemplate Insert Demo

We are now to a point where we can modify our RideRepository to utilize Spring JdbcTemplate. Let's start off by giving ourselves a little whitespace, and I'm going to leave that return null down below because we're going to change that out later with a select statement. I'm always caught off guard a little bit by Spring JdbcTemplate in that we use the update method to insert, update, and delete records. The only thing we don't utilize it for is reading records back. So we're going to go ahead and create some basic SQL inside of here and say insert into ride, and we're going to pass in name and duration, and then we'll say values ?,?, and this uses the standard prepared statement approach of substituting in values into those question mark fields. And we'll pass in var args here by saying , ride.getName and , ride.getDuration, and now we've actually created everything that we need to to insert a record into our database. If our SQL is all correct and our object gets populated correctly, everything is here to insert that record into our database. Let's go ahead and test that now. I'm going to save this code, I'm going to exit full screen. Make sure your server is started up. Once that's up and running, I'm going to go ahead and go to my RestControllerTest and we'll right-click on that createRide method, Run as, Unit Test. Once this completes, we come back with our green bar, we should have a record into our database. Now we can test that by going to our MySQL Workbench and saying select * from Ride and execute that, and we should see that we have the Bobsled Trail Ride in there with a duration of 35. If you want to test that again, we can go back to our IDE, go into that unit test and change this name to something else. So we'll say the Willow Trail Ride, and it was a duration of 30, save that. We can right-click on that test, Run As, Unit Test. Once that's finished running, we can go back to our database, execute that statement again, and you will see that Willow Trail Ride with a duration of 30 is in there. Now if your IDs don't look exactly the same as mine, that's because it's an auto-incrementing field and if you've ran it a few times and deleted a few records, that auto-incrementing field will continue to increase regardless of whether you've deleted those rows out. So, on mine you'll see that the Willow Trail Ride has an ID of 4, and that's just because I ran this a couple of times testing it. So, now you have an end-to-end example of our code that will call our database. Really from beginning to end we have our unit test here that calls the RestTemplate and does a put with an object of the Willow Trail Ride with a duration of 30 that goes through our controller and our service tier and our repository until it finally gets to our createRide method where we then pass in our values into this prepared statement using the Spring JdbcTemplate of insert into ride, name, duration, these two values, and this is just using var arg syntax to say ride.getName and ride.getDuration, and it inserts those in there. Very slick, very clean, very succinct code. I am going to show you how to use the simple JdbcTemplate, but I will tell you up front that this is the approach I like to use. This is the smallest amount of code and it feels like you're just in the database and you can get done exactly what you need to. The other approach does feel a little bit more like an ORM and maybe that's appealing to you, but this is my default and my first choice when I'm doing this type of work.

SimpleJdbcInsert Demo

To show you the SimpleJdbcInsert, I am actually going to comment out the tdbcTemplate example that we had done earlier because I'm going to revert our code back to that after we're done. If you are using GitHub, maybe you want to branch this example and do it in a separate branch and you can save it and look at it later or revert back to it, whatever you choose, just for this example I'm going to comment it out. Now there are a few reasons that I want to show you the SimpleJdbcInsert. It's an alternative, and it acts like an ORM more than a straight database call. So we're going to start off by creating a new instance of the new SimpleJdbcInsert, and this takes a jdbcTemplate in here. So it's built upon the JdbcTemplate. Now there's a lot more set up with this, but if you're going to use this in multiple places, you can create one SimpleJdbcInsert and set some of these values up per repository. You don't have to do this in every method. I'm going to go ahead and now create our list of columns. So you have to tell this what to work with as far as the columns in your database. So it's a little more verbose upfront with some of the configuration. So we'll say columns and then we're going to make a new ArrayList, and then we want to go ahead and add what our column names are in here. So we'll say columns.add, and we're just going to pass a string in here of name, and then we want to do columns.add, pass another string in here for duration. So these are the columns in our database. Just like we had created in our simple select statement up above, we're going to now map this object to that database. The next thing that we need to do is we need to set our table name. So we'll say insert.setTableName, and our table name is ride, and insert.setColumns. So we'll set our column names in there and we just called our columns. Now, we're finally down to where we can start putting the data in here. We want to create a map, and this is going to go through a simple map instance with a string and an object for the key and value pair. We'll call this data = new HashMap, and inside of here we can start putting our data in there. So we'll say data.put, and for the key it's going to be our column name. So for the name we're going to put in the value of ride.getName, and for the duration we'll say data.put and we'll give it a name of duration and the parameter we're going to pass in is ride.getDuration. And we're almost ready to run this. We've got to do one more thing. One of the benefits of using the SimpleJdbcInsert is that we can return the key that was created automatically from the database. So if you'll remember when we created that table we had an id column and it was an autoincrementing field. In one call we can get back that ID. So we can say insert.setGeneratedKeyName and we can pass in that it was the column name id. Now we're ready to say Number key = insert.executeAndReturnKey and pass in that data that we had just created. Just to show that the key is getting a return, I'm going to go ahead and throw a System.out.println for a test value here until we replace that return null and pass in the key value to it and see what happens. Now if your server is still running, it should automatically redeploy, but if it doesn't, you may want to restart your server. Once that's up and running, we can go ahead and open up our unit test and go back to our CreateRide unit test, and I'm going to change this to another trail ride name that I like called Round Valley, one of my favorite rides here locally, and change it to 38 minutes and save that just so we have another data point getting added to our database. Right-click and say Run As, JUnit Test. Once this completes, we should see that our testCreateRide completed successfully. If we switch back to our console, you'll see that ID number 5 was printed out on that System.out.println, and we can verify that by going to our workbench and running that execute statement again for the select *, and ID number 5 was in fact created with the Round Valley Ride with a duration of 38 minutes. So we've looked at that alternative way of using that SimpleJdbcInsert, which isn't bad, it's just not my favorite because of the amount of prep work that you have to do to get it ready to use. We have to define the SimpleJdbcInsert, we have to list the columns, set the table, we have to bind all of our data to a hash map and pass it in, but I do like the option of getting that key back in one call using the executeAndReturnKey approach. So there's a few tradeoffs, and as I mentioned also you can create one SimpleJdbcInsert and utilize that throughout your repository. So maybe you only do this in the setup of the object one time and use it throughout all of your methods. I am going to uncomment our SimpleJdbcTemplate update from before and comment out the code that we just did using the SimpleJdbcInsert so that my method is just using the JdbcTemplate and not the SimpleJdbcInsert because the other methods we're going to use just use the JdbcTemplate approach.

Summary

In this module, we created a unit test and modified our controller, we stubbed out our service and repository tier, and then finally inserted a record using the JdbcTemplate and then again using the SimpleJdbcInsert. The next step is going to be reading records from the database so that we aren't returning that null from our create method in the repository tier.

Reading Records from the Database

Introduction

In this module we're going to walk through reading records from the database. In previous examples we had a Create method that did an insert into the database. We're now going to make that return an object using the Read methods in Spring JDBC.

CRUD

We are focusing on the R of the CRUD functions, the Read. There are a couple of ways in which we can read records using Spring JDBC. We can execute a read statement using the JdbcTemplate and one of the mapping techniques, such as the primarily used RowMapper. Similar to the Create methods that used a SimpleJdbcInsert, we will also look at an alternative option with the SimpleJdbcCall. Using the SimpleJdbcCall we will still need to use a RowMapper to map the result sets to objects. Let's look at both approaches.

Demo Outline

This demo is going to be broken into multiple parts. We are going to start off with the easier read call to remove the hard coded getRides call from our application. Then we're going to work on the read functions of our CreateRecord call by modifying the test that we have for the Create to look for an object rather than just a successful call. We will also use a RowMapper to map our return statements from the database. Then we will create a standalone Read method that we can utilize in our Create call to return the newly created object from our database. To select all records from the database we need to use a query method to issue the select statement and then we use a RowMapper to map the object to the result set. It might not be clear to you at first glance, but you can see in this example by these two points that we are using an anonymous inner class to do this. It's not uncommon for you to just implement an anonymous inner class to map these result sets. Let's make these changes to our code now.

Read All Demo

As you may recall from our first demo, in our RestController we had a method to retrieve all of the rides from our system. So we had this testGetRides that would eventually get a list of rides returned from that response. It went end to end in our application clear to our RideRepositoryImpl that if you scroll to the bottom of this, you'll see that getRides method that has our hard coded array list in there. Before we modify this, let's open up our Ride object in our model and add one member variable and a getter and setter for it in here. Let's add a private Integer id in here, and if we scroll down, give ourselves some whitespace, we can right-click and say Source, Generate Getters and Setters, and create the getter and setter for that id field. We can save this. Now let's go back to our RideRepositoryImpl and we're going to eliminate that hard coded information in here. And I'm just going to eliminate everything but the return statement, and we'll say List an import and java util list of type Ride, and let's name this rides that will match our return statement that we left there, and we can say jdbcTemplate.query. Now we want to add a basic query here where we can just put some standard SQL where we can say select * from ride. And that will actually suffice for what we want to do with this select all. Now, we're going to bind that using that anonymous inner class that we talked about by saying new RowMapper, Ctrl+spacebar to import that, and give it a type of Ride. And if you haven't used the syntax of anonymous inner class before, it looks a little confusing at first, but it's not bad. Start with open/close parenthesis and then open/close curly brace and close off that line. Now, you will get an error at this point because it's telling you that we have to add unimplemented methods, so I'm going to go in between those curly braces and just give myself some whitespace to work with. So I just took that curly brace, closing parenthesis, semicolon down to a new line, and then I can hit Ctrl+spacebar and it will ask for the mapRow declaration. So I can go ahead and select that one, and you'll notice we've gotten rid of all of our errors at this point. If you are still seeing errors, you've possibly missed a step. From here, I can now implement this mapRow method, and this is just a template method pattern. If you're not familiar with this approach, I encourage you to go out and look at what a template method pattern is in one of the design patterns courses out here on Pluralsight. But suffice it to say, we can go ahead and now create our ride object, say Ride ride = new Ride, and we can begin filling this Ride object up with the result set that was returned from our query. So we can say ride.setId and use an rs from the result set .getInt and we can just give it a column name, which this one is going to be id. We'll do the same thing for the other member variables here. We'll say ride.setName, and we want to use an rs.getString for the name, and this column is named name, and then we'll do a ride.setDuration, and we'll do an rs.getInt again on this and pass in duration. And then make sure you don't miss this last step of changing that return null, which it defaults to to return ride. This is all we had to do to change this to now use our database. A couple things are going on here. Since this is a template method pattern, all we have to define is our mapRow method, which takes these items from our result set and enables us to store them in a Ride object. Behind the scenes, it adds this to an array list and returns that out of our query to our list of rides, and then we can return our rides array list at the bottom of this method. Let's save this and get out of full screen and start our server back up if yours is stopped. Once this starts up, we can go ahead and go back to our RestControllerTest and go to our testGetRides, right-click, and say Run As, JUnit Test, and this should return all of our lists, so our test was green, and we can see all of our rides returned from that list of our Bobsled Trail Ride, our Willow Trail Ride, and our Round Valley Trail Ride. Those were the three rides that we have stored in our database. So again, really simple. All we had to do was go through and we added that id to our Ride object, which was just a basic integer ID, and that's so we could return it to our UI, and then changed our getRides from being hard coded to use that query and RowMapper to map that object and return that list of rights.

Modify Test

In our CreateRide test and controller, we used a put method since in the rest definitions it is acceptable for both creating and updating records. In the RestTemplate API though, we must use a post if we want to return an object. Let's modify our test and controller to use and accept a post, and then we will be able to return our newly created Ride object.

Modify Test Demo

I'm going to stop my server just so it doesn't continue to auto deploy my code while we make our changes to our test and controller. Then I'm going to open up our RestController and scroll down to our TestCreate method where you can see we have this line of restTemplate.put for our object to this URL. I'm going to change this to say ride = restTemplate.postForObject. And the syntax for this method signature is a little bit different than the other in that we say that we want to post for a specific object and then we pass the type of that object in, so I have to say Ride.class. Now we have everything we need in here to accept an object and return that back rather than get a null response back for this ride. Now we can go to our controller, and inside of our controller change our request method type from a PUT to a POST. Now everything is in place for us to now go ahead and modify our CreateRide. Let's look at what we're going to do to change that to return that object for this create method.

RowMapper

In our select all, we used an anonymous inner class for the RowMapper. Although this is a common use, you oftentimes will have the same RowMapper in a repository or across repositories that you will want to utilize. Let's refactor our select all to use an external RowMapper because we are going to also use it in our RowMapper for our getSingleRide.

Externalize RowMapper Demo

To create our RowMapper I'm going to first exit full screen and make sure that my server has stopped. Then I'm going to right click on our com. Pluralsight. repository and say New Package. These don't have to be in their own package, but to reduce clutter I like to put a util package underneath my repository, so that'll be com. Pluralsight. repository. util and we'll click Finish. Now from here we can go ahead and say New Class and we can put in a name of RideRowMapper and we can add an interface. And we want to use the RowMapper interface and make sure you choose the one from org. springframework. jdbc. core. You'll oftentimes have two show up, as you can see here on my screen, one for swing and one for jdbc. core. We want to use the jdbc. core. Let's click OK and click Finish and now it's going to ask us what type we want in here. Let's change that to Ride. And rather than type all of this out, we can actually go into our RideRepositoryImpl and copy this out of the code that we had just created. So we have our @Override method that we can copy in here and that's just in our getRides, and go back to our RideRowMapper and paste that in. It should automatically do all of our imports for us and you can see it looks just like the anonymous inner class that we did. Now we can go back and change our RideRepository to use this RowMapper. So we can come down here and after that comma delete everything but that closing parenthesis and semicolon and say new RideRowMapper, and since it is in a separate package you will have to import it, and it should add the parenthesis on the end, if not you'll need to do that, and then save that and you'll see that we have our open close parenthesis on there that has now changed our implementation from an anonymous inner class to an external class for that RideRowMapper. Now we can exit full screen, start our server up, once that starts ups we can go back to our RideControllerTest, scroll down to our GetRides, right click on it, and say Run As Unit Test again and see that our unit test came back successful and we can verify that by looking at the results and that we have our green code. Now let's go ahead and work on changing our create ride to now return an object rather than just what we had originally defined not coming from the database.

Modify Create Ride

I debated on changing the CreateRide when we were in the create module to be able to return the object that we had created the database out of that function, but opted to do it in this read module because if you don't care about it being returned, this is a lot of extra work that you may or may not need. So if you care about reading that as you return it, I've opted to do it here. To be able to retrieve the record that we just created from the database, specifically the ID, we need to modify our create statement to utilize a keyholder. This is not a shortcoming of Spring JDBC, but rather JDBC itself. Spring has a decent workaround though, and let's implement this in our code now and return an object rather than null from our method.

Create Ride Read Demo

To change our CreateRide method, I'm going to first start by making sure that my server has stopped so it doesn't try to auto deploy our code while we're making changes to it, and then I'm going to go ahead and go to my RideRepository. Inside of our RideRepositoryImpl, you may remember we have all of this commented out code. I'm going to go ahead and remove that commented out code and paste it into a text file to save it for later because we're going to use it later and I don't want to have to go through and retype all of that. So I'll paste that into this notes.txt that I have in my project, and it can just live out there while we make changes to our RideRepository. So now inside of our RideRepository, I'm actually going to also comment out this jdbcTemplate.update because we're going to compare what we had before to what we have now and also be able to utilize that same insert statement. We want to start off by creating a KeyHolder and the KeyHolder is what stores the key that gets retrieved from our database while we do an insert. And we want to use a Generated KeyHolder, and a Generated KeyHolder is because we have an autoincrementing field that's automatically generated for us. The next thing that we want to do is also use a udbcTemplate update again, and that will enable us to use the PreparedStatementCreator. So, we'll say new PreparedStatementCreator, and if I use Ctrl+spacebar to choose this type, it says it's an anonymous inner type and it will automatically add our method that it needs to be generated through that IntelliSense. Now I can come down inside of here and get rid of that TODO, and inside of here I want to say PreparedStatement and this is just a prepared statement from java.sql, we'll say ps = con.prepareStatement, and here's where we get to utilize that same string that we had up above here. That's why I didn't just delete this out of here. We'll copy that and we will paste that in here. And now we're going to do something a little bit different. Instead of us putting in the var args like we had before in here, we can actually say comma new string, and it's an array, and we'll pass in an array syntax of id, and this is referring to the column in the database that we're going to return as a generated key. So yours should look something like this. It should say new String [] {} with an id in there for that array syntax. Now we can populate that string and say ps.setString, we're just going to fill those values in our PreparedStatement. And we'll say at position 1 we want to do ride.getName and we'll do ps.setInt, and say at position 2 we want to do ride.getDuration, and then make sure you change that return null to say return ps for our PreparedStatement. So now that we have our PreparedStatement set up, again we added that array syntax on to the end of it, we want to go ahead and now say , keyHolder on the end of that anonymous inner class, and this is what tells our insert statement to store that generated ID back in this keyHolder. Now, we can go down here and say Number id = keyHolder.getKey, and it will store that key into a number for us. And we can replace this return statement with a method that we're going to create. It currently does not exist. We'll say getRide id.intValue and say that now we can go below this method and create a new method, say public Ride getRide, and it's going to take an Integer id, and we'll just say Ride ride = jdbcTemplate.queryForObject. And this is a new query statement that we haven't used out of

JdbcTemplate just yet. And inside of here we can do a select * from ride where id = ?, and the syntax is a little bit different here. We're going to use that externalized RowMapper that we created. So we'll say new Ride RowMapper and then we'll just pass in our ID as a var arg and then return our ride at the end of it. So we'll say return ride, and save that. Now all the pieces are in place with that PreparedStatementCreator and the KeyHolder to now query our database and get the object that we just created and return that back to our UI. So let's exit full screen, start up our server. Now that that's running, we can go into our RideControllerTest and I'm going to change this to another name just so we can see that it did in fact execute against our database and our ride is going to return our Ride object down below here, we can go ahead and right-click on testCreateRide and say Run As, Unit Test, and once this runs, we can see that it created a ride and our unit test was of course successful, but now we can come up and test our GetRides and see that it did in fact insert that into the database. So let's right-click on GetRides and say Run As, JUnit Test, and you can see that the fourth one that had entered in down there is the Sagebrush Trail. So now we have changed our RideRepository to utilize that PreparedStatementCreator to get that key back. Again, I didn't show this in the create method because it's a little bit more complex and you really only care about it if you want to retrieve that object that you just created out of the database with its ID, and this is the only way that you can grab that ID using this approach. Let's look at an alternative approach that you may find a little bit simpler

Read SimpleJdbcInsert Demo

If you didn't like using that KeyHolder to obtain the ID of our newly inserted record, there is one other option, and that is to use the SimpleJdbcInsert that we had earlier. I'll let you decide which one you like better. I'm comfortable with anonymous inner classes, but a lot of people aren't. First, let's start off by stopping our server so it's not trying to redeploy as we make these changes. And when we go into our RideRepository, that keyHolder that we just created down to the number ID, let's just comment that out. You can compare on your own as to which one you like better. I'm going to go into the notes.txt that I had mentioned earlier that we pasted in and grab that code now. We'll come back over and paste that in below and uncomment it, and you can see here that we have a SimpleJdbcInsert, it's exactly what we had written before, and we had to set up our GeneratedKeyName and map our columns, map our data, call set on it, but it does have that one helper function of executeAndReturnKey that we can see down here. Now we can save that, go back to our RestController and change this to another name, we'll say the Yellow Fork Trail, and save it. We'll start our server back up. Now we can right-click on our testCreateRide, Run As, JUnit Test, and see that it was successful. Run our testGetRides again and see that it now has our Yellow Fork Trail down there as well. So, not a lot of code changes saved as far as one over the other, but you do eliminate that anonymous inner class inside of your code and you're just dealing with these maps to swap data in and out. You don't have any SQL, it acts more like an ORM, so it's really which is more beneficial to you for what you would like to do. If your inserts are always going to return this and you like the ORM syntax, maybe you'll go with this approach. If you don't mind that anonymous inner class, you have this approach as well. Both work well, it's all a matter of which is easier for you. In terms of efficiency, they're both prepared statements so they get both compiled and ran with the same efficiency.

Summary

In this module we removed the hard coded reference to our Select All request and made it return our table data. We modified our unit test and our controller to use a post so that our Create could now return an object. We converted our anonymous inner class to an externalized RowMapper to be shared across multiple requests and finally we showed how to read an object after a create using the PreparedStatementCreator and then the SimpleJdbcInsert. Next we're going to look at updates in our application.

Updating Records in the Database

Introduction

In this module we're going to walk through updating records in the database. In the last module we created a method to retrieve a single record, but we didn't expose it to the UI and are going to do that in this module and use it to retrieve records to update. We're also going to show batch update functionality.

CRUD

We are focusing on the U of the CRUD functions, the Update. In this scenario, SimpleJdbcCall doesn't offer us much, so we will just use the JdbcTemplate for our updates. Oftentimes when updating records we will also want to update multiple record and we can use the batchUpdate options to help with this as well. Rather than writing a complicated SQL statement the batchUpdate will manipulate multiple rows in one call. We will update a single record as well as multiple records in this module.

Demo Outline

As with our other demos, this demo will be broken into multiple parts. We're going to start off with the recall to retrieve a single record from the database. We created the method in the previous module we were going to use for this call, but we haven't exposed it through our repository and service tiers to our UI yet. Then we're going to work on an update function to modify a single record. And then the last part of the demo, we will utilize the batch update functions to modify multiple roles with one call.

Select One

In the previous module we created this getRide method in our repository tier, but we didn't expose it through our service tier to our front end. For this next demo we're going to add a call to the service tier and controller as well as add a unit test exposing it clear through all of the tiers to our front end.

Select One Demo

Our previous demo ended up with us creating this getRide method, but you'll notice that we haven't implemented the @Override annotation and exposed that through our other tiers. So let's start off by changing the getRide method by adding in @Override, and this will tell us that we want to expose this to other tiers. Now we're going to open up our interface, go to the RideRepository, and add just the signature here of Ride getRide. We're going to use an integer for the lookup, and we'll save this. And now we need to expose this through our service tier. The service tier, we're going to do the same thing. We want to expose a method called public Ride getRide that takes an integer and calls that method that we just created in our RideRepository. So we're going to say RideRepository.getRide and pass in that ID. And likewise, we'll want to go into our RideService interface and expose that through our interface as well. So we'll say Ride getRide and make that take the integer as well. Now that we're at this point, our controller can now expose this to our test or our UI tier, so we can open up our RideController, and I'm going to go ahead and paste in some code here that I don't go through and type this all out. But I'll walk through everything it does. So I'm going to grab this code that I've already written and go into our RideController and paste this at the bottom. Now, this one's a little bit different than what we have done in other controller methods. We're going to actually look for a path variable. So we're going to say RequestMapping value ride/{id}, and this tells it it's a variable that we're going to grab. And you can see down below here in our method signature, we say @PathVariable value id, and it will automatically convert it into an integer, passing that into the rideService.getRide that we just created. Let's go ahead and now add that to our unit test. So we can open up our RideControllerTest, RestController, scroll to the bottom, and we can create a method to get that individual ride. And just like the controller, I have some code that I've already written, but we'll walk through what it does. We'll grab this test and paste this in at the bottom, and this is a little bit simpler. You can see here we've said testGetRide and we've created an instance of the RestTemplate, but we can now just grab this for one object. So restTemplate.getForObject enables us to call this URL and you can see I'm putting on a parameter at the end of that URL of 1. So I want to get that first object back, and that's referring to the ID in which is stored in the database, and it will return that to a Ride object. I don't have to use a parameterized list or anything like that, it will just grab that individual object back and we can then run the test on it to see what is being returned. So we'll print out the name and it should be for this example in my database, the Bobsled Trail. So let's go ahead and save this, start up our server. Once that's up and running, we can go into our test, right-click, Run As, Unit Test, and we should have a green bar which we've got and our console already exposed that we had Ride name: Bobsled Trail Ride. So it's pretty simple. All we did was go through our RideRepository and expose that through our interface and add that @Override. We did the same thing in our service tier. We exposed that through the interface, as well as made it available through our RideServiceImpl. And then in our RideController, we created that parameterized return where it would take a URL parameter, assign that to a path variable, call rideService and return that object back out. And now we're to a point that inside of our tests, since we can grab this single object back, we can now use that for our updates. Rather than trying to just falsify the information, I opted to grab that object back and manipulate it. Let's look at what we're going to do for our update method now

Update One

Now that we have a method to retrieve a single ride we can use it to retrieve a ride and then change something and submit it back to the database using an update statement. Let's build a test for this and then build out the pieces through our controller service and repository tiers so that we can update our records in our database.

Update One Demo

To implement our update method, let's start with shutting down our server so that it doesn't continue to try to deploy our changes as we save files. I'm going to open up our RestControllerTest and just go to the bottom. I'm actually going to copy this testGetRide that we did and change it. Much of the code will be the same. I'm going to call this testUpdateRide, and we're going to leave the same getForObject in here that we had before and just turn around and call that for our save. So we'll say ride, and rather than hard code a value in here, I'm going to actually update one of the values that gets returned. So I'm going to say ride.setDuration and call ride.getDuration +1 in here. So basically taking whatever value gets returned, updating that by one, and setting it back in that object. Now we can turn around and use that same RestTemplate that we already have and say restTemplate.put, and I'm going to take the URL that we have from up above here and copy that down below. So we're going to call that URL with a put and then pass in our ride object. So that's what that entire method looks like now. You can leave the System.out.println at the bottom, but it's going to just be outoutting to the screen what you have already retrieved from before, not what necessarily gets updated. Now we can go ahead and open up our RideController and go to the bottom just like we did before and I'm going to do the same thing, copy that getRide that we have and change that to be our updateRide. And inside of here, I'm going to just cut that down to /ride, change the RequestMethod to PUT, and now change the method to updateRide. And we don't need a path variable anymore and rather just the object that we're going to pass in. So we'll say @RequestBody and we'll just passing the object Ride and name it ride. And now we can call our rideService.updateRide. And this method won't exist yet, so it will give us an error. So we'll say updateRide. And that's everything that we need inside of our controller. Now there's two ways you can go about this. I'm going to use the helper method by clicking on the red X that you should have over here as well and say Create method updateRide in type RideService. So when I double-click on that, it's going to add this signature to our RideService interface and subsequently give us that red X inside of our RideServiceImpl. Going into our RideServiceImpl, I can go to the bottom and hit Ctrl+spacebar and it will ask me to update the ride, or add our updateRide method in there, and inside of here we can do the same thing. Very quickly say return rideRepository.updateRide and pass in the ride object. And just like before, this method signature currently doesn't exist. So I can click that red X again and say Create method updateRide in type RideRepository. Once I say this, it will transfer that red X to our RideRepositoryImpl. Don't worry, in the next module, we're going to use the service tier for more than just a passthrough, so you won't be wondering why we're just creating all of these passthrough methods. Now let's open up our RideRepositoryImpl and likewise go to the bottom, hit Ctrl+spacebar, and it will ask us if we want to add that updateRide method. So we can double-click on that. Now we have all the pieces in place to issue our update statement using the JdbcTemplate. I'm going to go ahead and say jdbcTemplate.update, and this is pretty straightforward SQL, so we're just going to now say update ride set name = ?, and this is using the PreparedStatement syntax, say duration = ? where id = ?, and now we can close that SQL off and put in a comma, and I'm going to carry this down to the next line so that you can see it easier, say ride.getName, ride.getDuration, and ride.getId. Since this is an update, we have that ID stored in our object. And now, we'll just return ride from here. Now we can save that, and all of the pieces are in place to now run our test. So I'm going to exit full screen mode and start our server up. Everything looks like it started up fine there. I'm going to switch to our test. Now I'm going to right-click on testUpdateRide and say Run As, Unit Test, and we can see that it ran, but let's verify that it did actually run against the values in our database. So I can switch to MySQL Workbench and just issue a select statement of select * from Ride where id =1, and it says the duration is 37. If I switch back and run that test again, say Run As, JUnit Test. It ran again. If I issue that select statement again, it should say 38, and it does. So, recapping all those pieces, we went to our test and created this testUpdateRide where we retrieved an object based off of its ID and then set a duration, just took the current value that it is, added 1 to it, set that back in that object, and then called a put with that object, which in turn calls our RideController, it takes in that object, does an update right into our service tier on it, which went through the interface and then to the implementation, which passed it to our repository tier, which again is just the signature going to our update statement where we used basic PreparedStatement syntax of question marks and passed in those values through var args to issue that update statement. Pretty straightforward, not too complex. Let's look at doing the same thing using batch updates.

Batch Update

Up until this point we have been working with only a single row in the database and that includes both creating and updating records. Spring JDBC has a convenient way, using a batch update, to pass in parameters and have it execute across multiple rows. To demonstrate this functionality we're going to add a column for ride_date to our database and then use the batch functionality to update it.

Batch Update Demo

To execute the batch functionality, I'm going to start off by adding a column to our database table that's going to give us something to update with our batch statement. So I'm going to switch to our MySQL Workbench and I'm going to execute this SQL statement to alter the table and add a ride_date column to the table after the duration column. So I'm going to select that field and execute the statement underneath that cursor. You can see that it executed green down below in our log statement, but I'm going to double check that by saying select * from ride and execute that statement, and we can see that we have a column, ride_date, that has all null values inside of that table. So it added our column, but we didn't populate that with anything. Let's switch back over to our IDE and go into our RestControllerTest. From here, we're going to execute a call to trigger that batch statement. Now I will tell you that passing an array list of variables across a RESTful call is outside the scope of this course, and we're just going to shortcut that by grabbing our get from up above and using that to call our URL. So let's go up and grab our getRide and take a copy of that and paste it down below. And I'm going to call this testBatchUpdate. And from here, we can go through and say restTemplate.getForObject, and we're not going to store this to anything because we really don't care that it grabs an object, we're just going to use it to call the URL batch. So I'm going to change that URL to batch. And say that it just returns an Object.class, and save that. So, our testBatchUpdate method looks just like that. Very simple, I'm going to call this URL, we really just want to get it. We could even trigger that from just a browser call and have it issue the statement as well. Now we can switch over to our RideController and we're going to create a batch request down here. I'll grab our getRide from up above again and paste that down below. And the same thing, I'm going to change that URL to batch, I'm going to leave the request method type as GET, the response body is going to just be Object because we don't care about it, we're actually going to throw it away, and we're going to change the name to batch and just give it an empty set of arguments that we call in there. Now inside of here, I'm going to just change the body of this method to return null. And, then we can start by implementing our service calls that we're going to execute in our service tier. So I want to start off by saying rideService.batch, and we're going to now create a method inside of our service tier that does a couple of things. So let's save this method, click the red X to tell it to create a method batch in type RideService. It will add that to our interface. Once we save this, it will now tell us that we have a problem in our RideServiceImpl. I can switch over to our RideServiceImpl, hit Ctrl+spacebar, and it will ask us for a batch statement. Now is where we can start utilizing our services tier. Inside of here, I am going to now create a method to retrieve a list of rides, and we're actually going to use the method that we created up above. So we'll say Ride rides = rideRepository.getRides. Now this will return our list of rides for us from up above, just like we had used to return all of the rides to our test. From here, I'm going to create a name value pair, and this is where it might get a little interesting for you. So I'm going to create a list of object arrays, and I'm just going to call this pairs, and set that equal to a new array list. And save that. Now from here, I'm going to execute a foreach statement, and this is going to iterate over the Ride objects from our rides array list. So we'll save this. Now inside of here, we're going to create a name value pair, and the order of this does matter. So we're going to say Object [] tmp =, and we're just going to use the array syntax and say new Date and then ride.getId. Close that curly brace off. And then we'll say pairs.add, and we're going to add that object array to our array list of pairs. Now we can call the rideRepository.updateRides, and we have not created this method yet. We're going to, we're going to pass our pairs into it. Now we can save this, click the red X, and create the method updateRides in our RideRepository. Save that, and the same thing, we will get that red X inside of our RideRepository. Same pattern we've been using the last two examples. From here, we can hit Ctrl+spacebar and say updateRides, and this will add the stubbed out functionality of our updateRides method that we're going to use the batch functionality from our JdbcTemplate in. From here, let's say jdbcTemplate.batchUpdate, and we're going to use the SQL of update ride set ride_date = ? where id = ?. Again, that PreparedStatement syntax. We're going to pass in that pair for the array list a pair of objects into here, and this is why that order mattered. The first order it's looking for is the ride date and the second order of objects it's looking for is the ID. So that object array that's inside that array list has the right setup for the order of those elements. From here, we're ready to save that, exit full screen, I'm going to start up our server, everything looks fine on the startup of our server, and I'm going to go to our unit test, right-click on batchUpdate, and say Run As, JUnit Test, and once that executes on our server we can go over to our JUnit test, everything looks good there. I'm going to switch back to our MySQL Workbench and execute that select * from ride again, and we'll see that all of our ride dates are now populated with a date. Now, I get that this example may not be quite as full featured that you may be looking for because we are just taking inside of our RideServiceImpl and putting in a new date, but you see that you could put in whatever value you wanted there. We could calculate a date, or if you're using a different type of column, possibly storing some other value inside of there. For the sake of the example, we just shoved new data in here and you'll see that it populated our database. Pretty simple. All we had to do was go through in our RestControllerTest and create a call to that URL, again passing those values through the JSON instance and to our controllers, really outside the scope of what we're doing inside this course. But it calls the URL inside of our controller. And you can see here we're just going to call rideService.batch, which eventually calls our service tier where we go through and we can see that we're going to get all those rides back, create our array list of temp items. Inside of that we have an array of those temp objects, add that, and then call our batch statement, which eventually goes through and executes our updateRides. Now, it should be noted that the batchUpdate will work with both updating rows, as well as inserting rows. So maybe you had 20 rides you wanted to insert into this database at once, you can use the insert statement. It doesn't care, the batch update will just execute whatever you have based off that array list of objects that you're passing into it and execute it for each row. The order of those elements does matter, though, as it does look for those question marks in a specific order.

Summary

In this module we exposed the getRide method, which selects one record clear through to our UI and test tiers. We had implemented the functionality in a previous module, but hadn't exposed it to the rest of our application. This in turn enabled us to implement a method to update one row in the database. After that we implemented the functionality showing the batchUpdate features available in Spring JDBC to update multiple records. Next we're going to look at deleting records from our application.

Deleting Records from the Database

Introduction

In this module we're going to walk through deleting records from the database. Deleting records is fairly straightforward, we're going to look at a few solutions to incorporate into your application.

Demo Outline

We are focusing on the D of the CRUD functions, the Delete. Many applications are historical and don't allow deletion of data, thus making the delete functions the least used of the API. Although it is quite simple with both, we will look at a JdbcTemplate and a NamedParameterJdbcTemplate to delete records from the database. As with other modules, this demo will be broken into multiple parts. We are going to start off with deleting a record using just the JdbcTemplate, which is very similar to what we've done with other demos using the Update method. After that we're going to explore using a named parameters. Named parameters aren't just for deleting, but we're going to show how you do that while using a NamedParameterJdbcTemplate. As mentioned before, the delete is quite simple for us to complete, especially having done all of the other CRUD functions in our application. It is executed using the Update method of the jdbcTemplate, SQL and the other values being passed in using var args. To complete this demo, we will add this code into our repository, a call in our service tier and then the method in our controller and a test to call it. Let's try this now.

Delete JdbcTemplate Demo

To implement our delete functionality, we're going to follow the same approach we did with our update functionality. I'm going to start off by making sure that my server is stopped, and then I'm going to open up our test and scroll to the bottom of it and copy one of these existing tests that we have and just paste it down below. I'm going to change the name to testDelete. And inside of here, the delete is actually quite simple. We're just going to call the delete method with the URL of the ID that we want to delete. So we're going to say /delete and the actual ID of one of the ones that we want to delete. Now to figure out what that ID is, I'm actually going to switch over to my MySQL Workbench really quick and run a select statement and see what the last created record I have in there is. Your numbers may not be as high as mine because I have implemented the test a few extra times just to see some test data that's out there, but my last created record was 16. Again, it doesn't matter if yours is 9, 10, 11, whatever. I'm going to grab that number 16 there and we're going to switch over and say that we want to delete that URL. So localhost:8080/ride_tracker/delete/16, and 16 is going to be the ID that we catch inside of our controller. So let's save this, open up our controller, and inside of here we're going to go ahead and grab this getRide method that we've already created because we want to grab a path variable just like this one. Same thing, we'll go down to the bottom, paste that in, and instead of it being a value of /ride, we're going to do /delete and grab that ID and change the request method to delete as well. Now, for our return type we're going to just change this to Object, and I'll explain why here in a little bit. We'll change the method name to delete, and then we can change the code that calls our service tier and our return down below. Now as I mentioned a minute ago that we were going to return object type from our body, and this actually has to do with the REST service and returning a valid response versus an invalid response. And void will return a 404 error. Don't worry about it now, we're going to go into this deeper when we talk about exception handling in the next section, but we'll just say rideService.deleteRide, and this doesn't currently exist. We'll pass in the ID, and then we're going to below that say return null, just like we did with the batch up above. Now we can save this and click on the red X over in the left-hand column, and just like we've done in previous modules, we're going to create that method deleteRide in our service tier. And this is going to say void deleteRide and take an Integer id, we'll save that, and now it will ask us to provide an implementation in our RideServiceImpl. I'm just going to scroll down to the bottom of here, hit Ctrl+spacebar, and add the deleteRide functionality. And this will just call our rideRepository.deleteRide, and it will take the ID as well. We'll save this. And same as before, we'll click that red X and say Create method deleteRide in our RideRepository. Save that, and now we can add our implementation to our RideRepositoryImpl. Now inside of here I'm going to scroll down to the bottom and make sure I'm inside that closing curly brace, hit Ctrl+spacebar, and add the deleteRide functionality in here. And now we can call our jdbcTemplate update. So we'll say jdbcTemplate.update. And we're going to just pass inside of here some basic SQL of delete from ride where id = ?, close that quote off, and then we're going to pass in the id and close that statement off. That's all we have to do to actually implement our delete functionality inside of our repository. I'm going to exit full screen, start our server backup, make sure that there's no errors on startup, everything looks good there. And then I'm going to switch back to our test and make sure you've updated your ID with the correct one from your database, and then I'm going to right-click on my testDelete and say Run As, JUnit Test, and when this is done I will see a green bar here and I can switch over to my MySQL Workbench, execute that select statement again, and record 16 has been deleted from our database. It's pretty straightforward. Let's walk through that again. We went ahead and created that RestTemplate delete and we did URL rewriting and just tacked the ID onto the end of there. Inside of our controller, we have a delete method that we created that does URL rewriting and grabs an ID off of that and then it uses the method type of delete and grabs that ID, passing it to our service tier. Our service tier passes that through to our repository where we now implement our update to issue the delete statement, passing that ID in. In the next section, we're going to go through and show you how to do the same thing using named parameters to get rid of that question mark.

NamedParameterJdbcTemplate

Since the addition of var args to the JDK, a lot of people don't know or think about using named parameters. Named parameters allow us to lose the numbering of parameters in our SQL statements and instead pass in a map of names with values. This is especially useful if you already have a map for some other reason in your application and just want to modify the database with that information. I should note that named parameters can be used for almost every function that we have done so far. So if you like this syntax better, you can use it instead of numbers in any of the SQL we've done in our repository tier. Let's change our delete to implement this now.

Delete NamedParameterJdbcTemplate Demo

This is where we left off after the previous demo of implementing our deleteRide functionality. I'm going to go ahead and give myself some whitespace down below here to create a new instance of a NamedParameterJdbcTemplate. Now if you like this approach, you don't have to create this NamedParameterJdbcTemplate per method, you can create one up above and pass that in. So, all we want to do is create that instance and use our NamedParameterJdbcTemplate. I'll go ahead and bring this down to the next line so that you can see the whole thing. And from here, we now need to create a map with our name value pairs in it. So we'll say Map and import a java.util map, and it's a String Object pair. We'll call this paramMap. And this is equal to a new HashMap. And inside of here, we're going to say paramMap.put, and we'll pass in the string of id and the value of id. And now all we have to do is that similar SQL that we implemented up above, but we no longer have the parameters of question marks in here, rather named parameters. So we'll say namedTemplate.update, and I am going to grab the SQL from up above and just modify it to show you the change here. So we'll paste that in, and this is now changed to id and we pass in our paramMap. This may seem like a little bit of overkill for just this one parameter, and I won't argue that it's not, but think in terms of if you had a search screen where you were maybe passing in 5, 10, 15 different parameters that you're trying to gather and then have that dynamically map to the SQL in the right positions. This is a lot better approach than trying to use question marks and get those built the right way in there because I can just pass that map of parameterized values in. Now that we've saved that, I'm going to exit full screen and switch over to my workbench to see what the last created record I have in there, and the ID is 15 for me, yours may be something different depending on how many times you've run this test. Now we can go back to our test and change the ID of the record that we want to delete. I'm going to change mine to 15 because that was the last record I have in there. Make sure your server is started up, and then right-click on that test and say Run As, JUnit Test, verify that it came back green, which it did. Switch back over to our workbench, and 15 has now been deleted. And again, what we had to change there was inside of our RideRepositoryImpl to use that NamedParameterJdbcTemplate. So the NamedParameterJdbcTemplate just gives us another option instead of having those question marks inside of our application.

Summary

In this module we deleted one record using the update method of the JdbcTemplate to issue our delete SQL. This is quite similar to what we have done with other functions in our code now, so it should look pretty similar. We also showed the functionality of using a NamedParameterJdbcTemplate. This is a great alterative to using the numbering scheme that we have seen so far in our application. Next we're going to look at exception handling and transactions inside of our application.

Exceptions and Transactions

Introduction

In this module we're going to talk about handling exceptions and how to work with transactions. Up until this point we have executed code, being careful not to instigate any exceptions nor requiring the need to roll back from a transaction. Let's dive into how to handle exceptions and gracefully roll back a transaction.

Overview

Every interaction with the database inside of Spring has the possibility to throw an exception. Spring handles exceptions differently than you're possibly used to. They do not use checked exceptions. Spring rather handles everything through runtime exceptions. Runtime exceptions are not a checked exception and thus don't require you to catch them, but they can still be thrown by your code. Although not specific to Spring JDBC, but rather Spring and Spring Web Development, we're going to use an exception handler inside of our controller to catch any exceptions thrown by our repository tier. An ExceptionHandler is what we use to capture exceptions that are thrown from other tiers in our application and still return an acceptable message to our front end. In this code snippet we catch an exception, but still return an HttpStatus of 200 as indicated by the OK in our response entity.

Demo Outline

Let's add exception handling to our application to catch potential database exceptions from being thrown to our client. We're going to create an ExceptionHandler in our controller, create a custom ServiceError class and then modify one of our methods to throw an error to test this functionality.

Handling Exceptions Demo

To simulate exception handling inside of our application, I'm going to first open up our test and scroll to the bottom and copy the testDelete method and paste it below that. I'm going to change the name of it to testException and we'll call a getForObject on it with a URL of test. And it's going to want a return type of Ride.class, which we don't really care about because it's always going to just throw an exception. So this is what our method looks like. To call this, let's go ahead and define that test URL inside of our application. Now if I open up my src/main/java RightController, I'm going to do something quite similar to what we just did there. I'm going to grab this batch method that we had created and copy and paste that down below. And I did the batch method because it's the closest to what we're trying to do here. So I'm going to change the value of that to test, and for the ResponseBody object we can leave that, the name, I'm just going to leave it test as well, and then inside of here I'm going to throw a new DataAccessException, and it takes a string for the constructor. So we'll say Testing exception thrown. We'll save this. Now let's go ahead and run this and see what happens. So if I start my server up, make sure there's no errors upon startup, and everything looks fine there, I'm going to go back to the test method that we created and right-click on it and say Run As, JUnit Test, it should come back and give me a stack trace saying that it failed. And this is the exact response we were expecting to get, an ugly error with a stack trace that was propagated clear back to our client. Let's see how we can make that a little bit better of an experience for them. Now I will say there are lots of ways you can deliver messages back to the end user, and we're not going to cover all of them, we're just going to show you one good way using the exception handling mechanism to give them a better error message. So let's right-click on src/main/java and say New, Package, and we're going to put in here com.pluralsight.util and click Finish. We're going to click on that newly created package and create a class. I'm going to call this class ServiceError. I'll click Finish. And this is a very simple class. We're going to have two field member variables inside of here called private int code and private String message. We're going to do two things inside of here. First, let's right-click on these and say Source, Generate Getters and Setters, and make sure you have both of them selected, code and message, and click OK. And then the second thing we're going to do is create some constructors inside of here. So we want to go ahead and say public ServiceError, and we'll just create a default no arguments constructor, and then the second one we'll do is public ServiceError that takes an int code and a String message and assigns those to our two member variables. So we'll say this.code = code and this.message = message. We'll save those. And I'm going to right-click and say Source, Format, and it should clean up that for us, as well as a Source, Sort members, and click OK. Now this looks like what we're expecting to see inside of our application. We can go ahead and go back to our controller, and at the bottom of our application inside of our controller, let's add an exception handler here. So we'll say @ExceptionHandler, and this is an exception handler for any RuntimeException that is thrown. Now we can generate the body of the method that we want to to handle these responses. So we'll say public ResponseEntity, and this class uses generics, so we're going to pass in a ServiceError, the service error that we just created, and we want to say handle and pass in the RuntimeException that we are handling, and give that a variable name of ex. Now that we've got the body, let's put the meat of this exception handler in here. We'll say ServiceError. And this is just the instance that we're creating to pass back to that end user. So we'll say ServiceError = new ServiceError. And, you may have different types of errors that you pass back to your user. For our sample, we're just going to pass back the OK value. You could have some specific ones coded to your application, but that's really outside the scope of what we're doing for this example. So we're going to say OK.value, and then we're going to say ex.getMessage. Now we want to return that new ResponseEntity and we're going to pass back our error instance that we just created and an HttpStatus of ok. Now, our exception that gets thrown up above of our DataAccessException will get handled down below by our exception handler. Let's try this out. I'm going to exit full screen, go back to our test, and I'm going to right-click and say Run As, JUnit Test. Once this runs, we'll see we have a green bar now, and in our console we have the error message from before in there, but now when we ran there is no big stack trace that's dumped out to our application. So it gracefully handled that message, and you can go ahead and look at the response body that comes back, but that's just one way we can grab those exceptions from our database tand pass back a more user friendly to our client.

Transactions

Individual methods in our repository tier are ran in a single transaction for the lifecycle of that JdbcTemplate call. But if we want to implement transactions across multiple calls, we need to create an instance of a transactionManager like the one listed here. This transactionManager wraps our dataSource and utilizes that transaction capability of our JDBC driver that we have chosen for the platform we are using. Let's add transactions to our application. We first need to add a transaction manager to our JDBC config file. We also need to create a sample call that spans two JDBC transactions. From there we can annotate that method with @Transactional, signifying that we want to roll back if something happens. And finally, we can verify that it did complete with a rollback.

Transactions Demo

To demonstrate how a transaction can help us roll back changes if we're partially through a transaction and want to undo anything because an error occurred, the first thing I'm going to do is start up my server. Then we're going to go through a series of changes inside of our test just to prove out this functionality so that we don't have to write a lot of extra code just to show you this demo. First inside of our test, I'm going to execute our createRide method. So I'm going to right-click and say Run As, JUnit Test, and once this runs, you'll see that it's ran successfully, I'm going to switch over to our database and run this select * from Ride. You see inside of here I have two rides at the bottom of my database, you may only have one because you just executed it, but notice how they have no ride data in there, and that's because when we created our createRideTest, we didn't have it adding the date, we updated those dates in the batch method. Let's go ahead and go over to our code now and open up our service tier, and inside of our service tier let's find that batchUpdate method. You can see here, we have all of our code in place to get all the rides, pull them back, get a name value pair of the date, ride id, we have this batch method where we grab all the rides and create a list of object array pairs of the date and the ID to update. Let's go to the end of this method and throw a new DataAccessException. And what this would simulate is that as we got partially through our code, something happened and we would want to roll back those changes. So we'll just put in here a string of Testing Exception Handling and save that. And now when we go to execute that batch service, it should go through and roll them back rather than update all of them. So let's scroll down in our application and find that testBatchUpdate, and there's mine. I'm going to right-click on it and say Run As, JUnit Test. Now if things were set up with transactions, it would roll back all those changes. Let's execute that and see what happens to our database. It's successfully completed. I'm going to switch over to our MySQL Workbench and run that select statement again, and you'll notice that all of my rides have now been updated even though that exception was thrown. Let's go ahead and fix that so that it doesn't do that. The first thing that we want to do is open up our src/main/resources jdbc-config file, and we're going to do two things in here. First, I'm going to go down to the Namespaces tab and check the tx namespace so that it adds that to the declaration in our XML file. Then I'm going to switch back to the Source tab, and now I can come in here and add two beans inside of here. The first one we're going to use the tx prefix of annotation-driven, and it has an attribute of transaction-manager, and we're just going to pass in transactionManager. The second bean I'm going to add is using the bean prefix, and I'm going to say id=transactionManager. And I'm going to type this one out because there's a few things I want to show you in here. And there is a class associated with this that is org.springframework.jdbc.datasource.DataSourceTransactionManager. And then inside of the bean tags, make sure you're in between the angle brackets, there is a property that we're going to add to it, and the property name is dataSource, and we're going to do a reference to our data source, so that is dataSource as well. Now, our tx annotation-driven and our transactionManager are defined. Yours should automatically try to deploy like mine has here. Now everything is set up inside of our XML file to handle transactions. The only thing we need to go back and add to our code is at the top of our batch we want to say that it is @Transactional. Now you may be wondering why I didn't have to include another JAR or anything like that to get this functionality, and that is because this is automatically included as a transitive dependency in Spring JDBC. So these classes were already available because we are using Maven. Now to test this, I'm going to go through and do the exact same thing again. I'm going to go back in our RestControllerTest and I'm going to scroll up to our testCreateRide because, remember, we don't have a ride now that doesn't have a date associated with it. So I'm going to right-click on testCreateRide and say Run As, JUnit Test, switch over to my database, run that select statement again, and I should have one at the bottom of here that does not have a ride date, and I do. Then I'm going to switch back and run my batch, but this time that transaction should be rolled back. So I'm going to right-click on testBatchUpdate and say Run As, JUnit Test, it successfully completed. That is because we have an exception handler in there that caught that exception instead of dumping out a stack trace. But if I go check my database again, we will see that it did not save those changes. So our transaction did in fact roll back. If you question that, you can test that by going back into your RideServiceImpl, going down to your batch, and either comment that transactional up above or comment out that throw new DataAccessException and save that. It will redeploy, and we'll go back to our testBatchUpdate and run that JUnit test again. Now it should have saved those to our database, our test was successful. Because we didn't throw that exception again, it should commit them, and it did. So quite simple. All we had to do was inside of our application we added the name space for tx so that we could put our tx annotation-driven transaction-manager in there, and then we added a bean for the org.springframework.jdbc.datasource.DataSourceTransactionManager that we wrap our data source with, and then inside of our code all we had to do was throw a DataAccessException to test it, but add that @Transactional annotation to our code to say anything that takes place in this unit of work, I want to be inside of a transaction. Very easy to do, and now you're starting to see the full power of why you have a service tier that calls multiple functions from inside of it. Think of a more complex application. If I was updating three, four or five database tables, all I have to do is still put that one annotation at the top and it will roll back all of our changes if an exception is thrown somewhere along the way.

Summary

In this module we learned exceptions and how to implement an exception handler since Spring handles all of its exceptions through unchecked runtime exceptions. Then we added transactions to our application using a transaction manager and annotations to mark which methods we want to be part of that transaction. And with exception handling and transactions, that actually concludes our course on using Spring JDBC to develop applications. Thank you for sticking with us this far and if you enjoy this course I recommend watching some of the other courses I have on Pluralsight.