

To build a Java application, the first step is to create a Java project. Most Java projects rely on third-party Java archive dependencies, and these third-party archives usually have dependencies of their own. On top of that, each version of the dependencies relies on other versions. Managing all these dependencies is a nightmare that Java developers have nicknamed JAR hell.

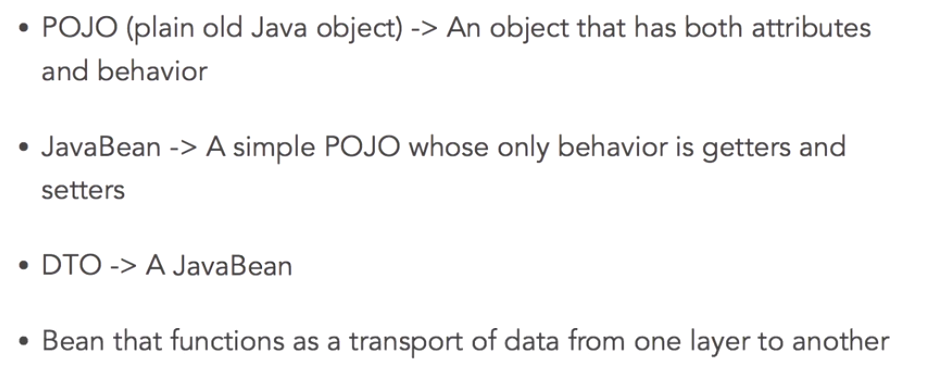
To avoid JAR hell, we use build dependency management systems like Maven or Gradle. But even with Maven and Gradle, versioning between individual .jar files can be a nuisance.

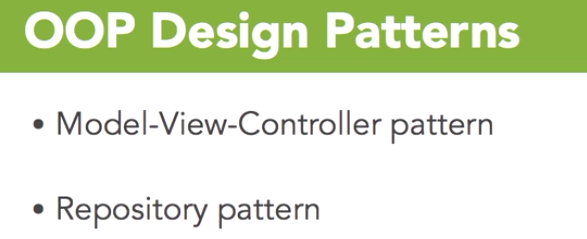
Spring Boot recognizes this, and created the notion of a Spring Boot Starter, which bundles several dependencies into a grouping that is easier to manage. There are a lot, and I mean a lot of Spring Boot Starter dependencies so even cobbling together a project on your own can be difficult.

**What is Spring Framework:**

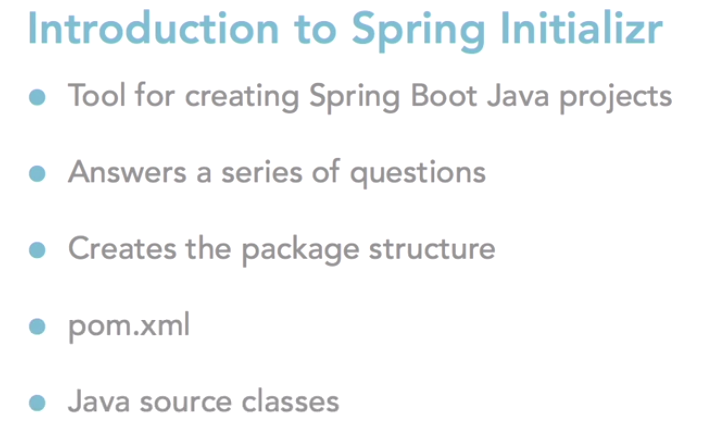
The Spring framework's purpose is to provide a comprehensive infrastructural support for developing enterprise Java applications. Now, what does that really mean? In layman's terms, Spring provides the plumbing. You focus on the business logic. It removes a significant amount of boilerplate code and as any developer knows, boilerplate code means copy and paste, which in turn means defects.

It also makes application development much easier because you focus on what you know instead of having to go look up those boring boilerplate tasks that nobody every commit to memory. Spring also promotes good object-oriented practices, as well as do-not-repeat-yourself principles because they're really baked into the framework. Well-defined interface definitions, leveraging dependency injection, and the concepts of true POJOs, are really part of that object-oriented programming practice.





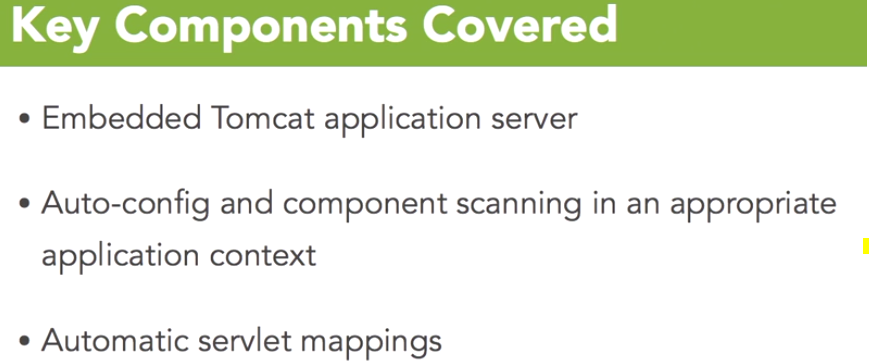
We need to know the above 2 design pattern while learning the Spring Boot

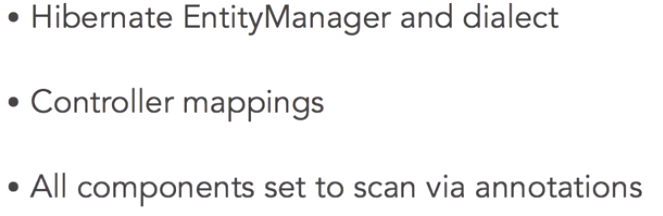


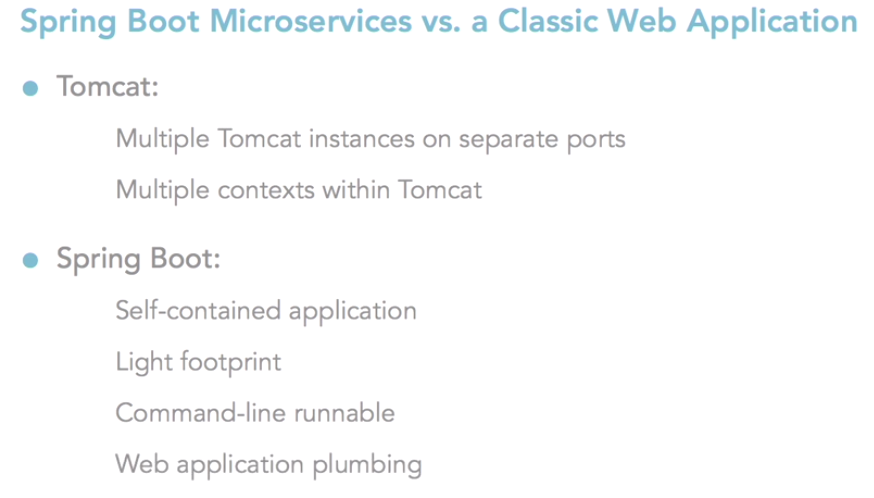
<https://start.spring.io/>

Adding the **@SpringBootApplication** annotation to this class with the main method tells Java here is where our Spring Boot Microservice starts. Command line parameters or special startup logic resides here.

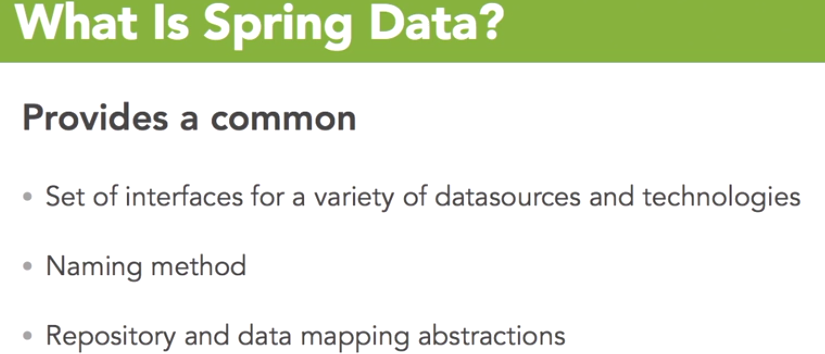
With this one pom file, and one Java source file, we can now build and run the project as a Spring Boot application.





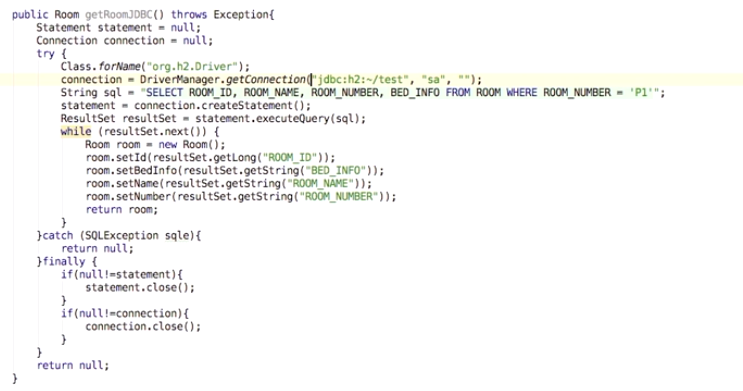


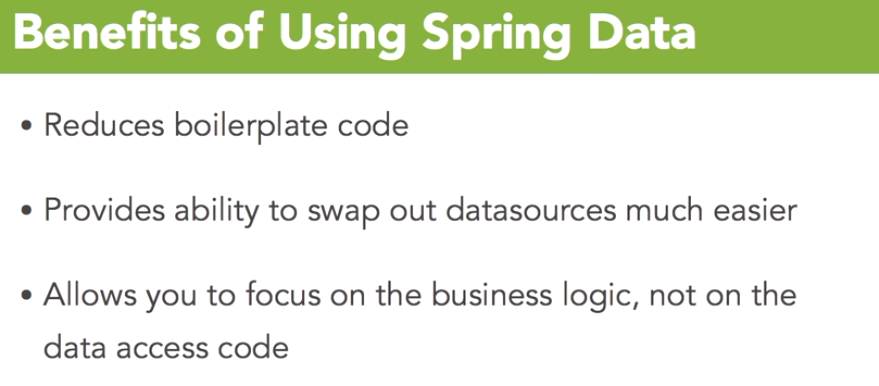
Beauty of a microservice, it's completely self-contained with a light footprint and it's runnable with a single command. This makes it a favorite with the dev ops and cloud computing crowds because the web application and server start up can be scripted as one command.



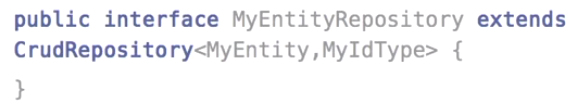
Spring Data provides a common set of interfaces for a variety of data sources and technologies. Now, what that really means is not only do I have the traditional RDBMS data sources and all its various flavors, but I also have all the NoSQL database options, and Spring Data's interfaces connect to both in almost seamless integration.

Spring Data also provides a common method naming convention that allows me to swap those datasources behind the scenes almost effortlessly.

**Earlier Code:**

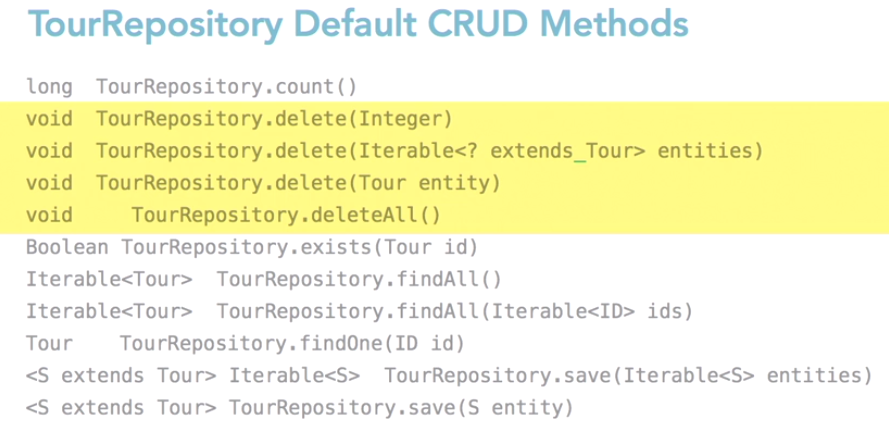




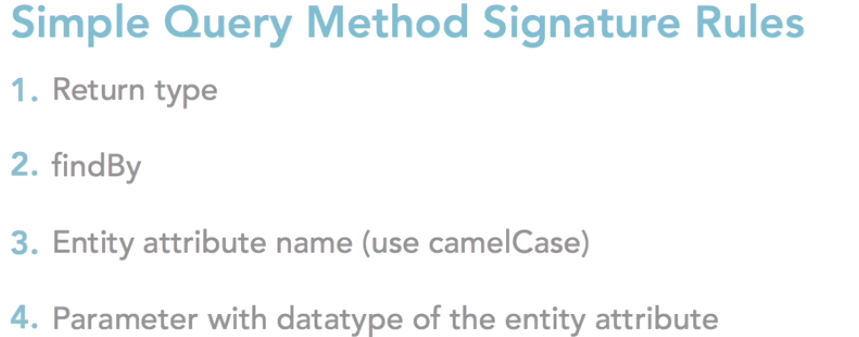


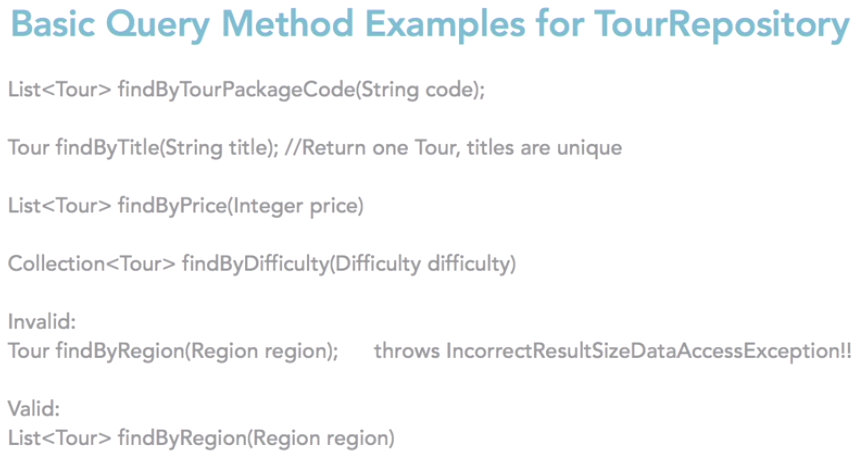
To use Spring Data JPA for a JPA entity we declare a Java interface class that extends from CrudRepository, where T is the domain type the repository manages and ID is the type of the ID of the entity the repository manages.

Ex:

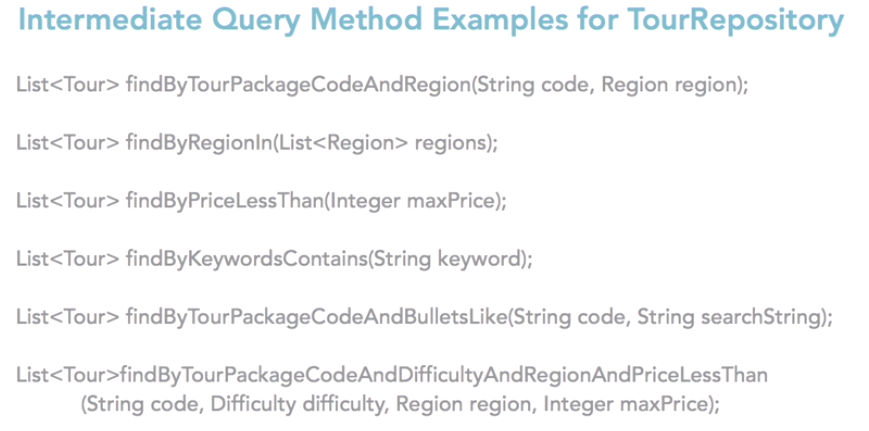


Spring Data JPA eliminates ugly boilerplate querying code. To create a simple query method, you only need to declare the return type. Begin the signature with findBy, followed by an attribute name in camelCase and query parameters whose type matches the attribute type.

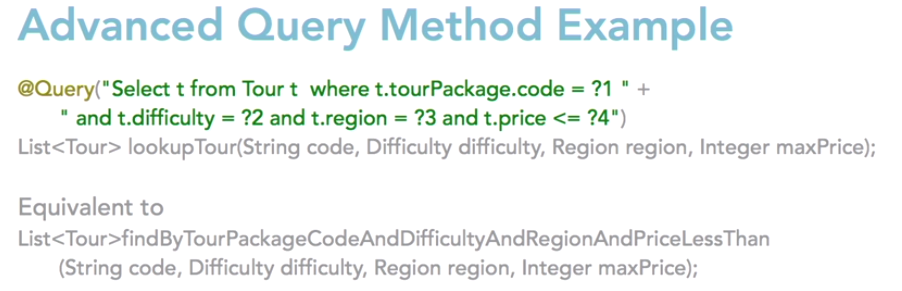




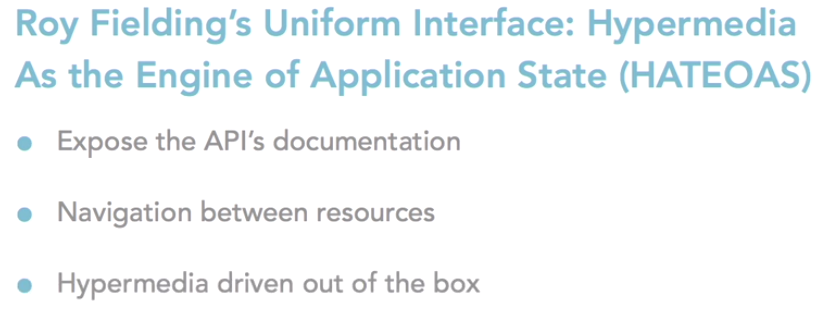




As you can see, the signatures on the methods become longer as the query method becomes more complex. You could leverage complex JQL queries inside a repository interface by using the @Query annotation.



**HATEOS:**



Spring Data REST which is a service for creating Hypermedia driven RESTful APIs. what is a Hypermedia driven RESTful API?

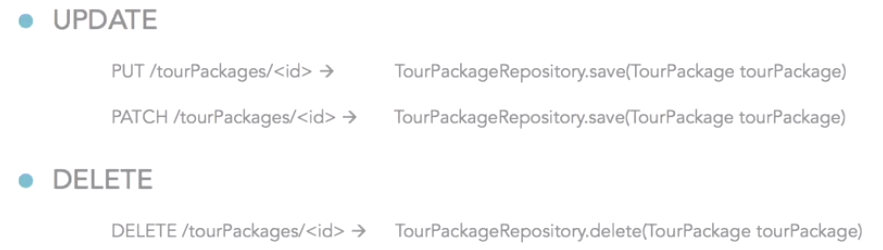
According to Roy Fielding, who first published the REST API Specification, an API is not truly Restful unless it follows a uniform interface. One of his constraints for a uniform interface is followed when Hypermedia as an Engine of Application State, or HATEOAS, is employed.

A RESTful API should do more than expose resource endpoints over HTTP. It should also expose the API's documentation and automatically provide navigation between resources. Hypermedia driven APIs accomplish just that. The implementers of Spring Data REST agreed, and that's why Spring Data REST APIs are Hypermedia driven out of the box. No extra configuration is needed. Spring Data REST employs the Hypermedia Application Language, or HAL standard, to associate resource objects to one another.

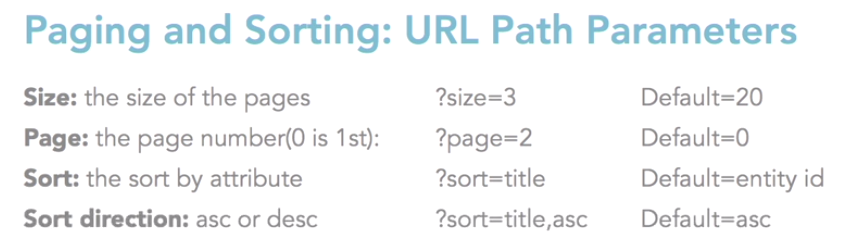
At application start up, Spring Data REST scans the classes and finds all the Spring Data repositories, creates an endpoint that matches the entity name, appends an s and exposes the operations as APIs. So here are the mappings for CREATE, the mapping for READ, the mapping for UPDATE and DELETE.At application start up, Spring Data REST scans the classes and finds all the Spring Data repositories, creates an endpoint that matches the entity name, appends an s and exposes the operations as APIs. So here are the mappings for CREATE, the mapping for READ, the mapping for UPDATE and DELETE.At application start up, Spring Data REST scans the classes and finds all the Spring Data repositories, creates an endpoint that matches the entity name, appends an s and exposes the operations as APIs. So here are the mappings for CREATE, the mapping for READ, the mapping for UPDATE and DELETE.At application start up, Spring Data REST scans the classes and finds all the Spring Data repositories, creates an endpoint that matches the entity name, appends an s and exposes the operations as APIs. So here are the mappings for CREATE, the mapping for READ, the mapping for UPDATE and DELETE.Bottom of Form

At application start up, Spring Data REST scans the classes and finds all the Spring Data repositories, creates an endpoint that matches the entity name, appends an s and exposes the operations as APIs. So here are the mappings for CREATE, the mapping for READ, the mapping for UPDATE and DELETE.

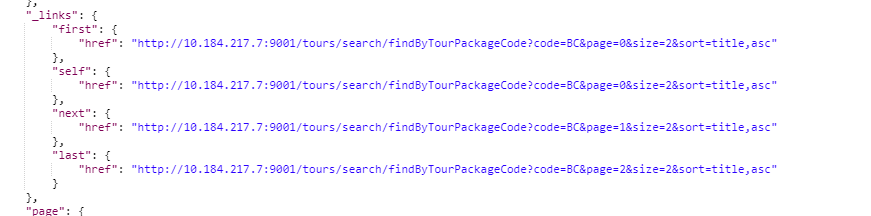




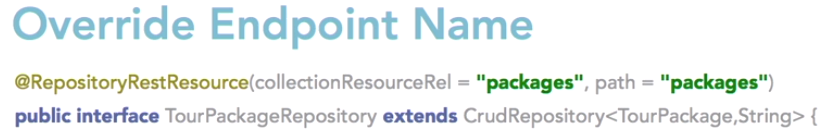
Every endpoint that Spring Data REST exposes also creates a /search resource



The response contains the hrefs called First, Last, Pre, and Next. Front end developers love this. By using these hrefs, they don't have to calculate the last page number, nor keep state of the current page. We also see at the bottom, there is some meta data. If the client chooses not to use those links, the API also provides more page meta data at the bottom of the response packets, so they can calculate it themselves. In the past, we would have had to code all these features ourselves.







Earlier we saw that we cannot only look up entities but also create update and delete them with HTTP post, put, patch and delete. But I do not think we want to allow the public to modify our database in this way.

There are two annotations to accomplish this.@RepositoryRestResource is used to control access at the class level and @RestResource is used to control access at the method level.

We can also use @RestRepositoryResource to override the default end point name. In this example, we set the TourPackages endpoint name to packages. So, in TourRepository, we'll use our IDE to help us override methods. So, the methods we're overriding here are save and delete.

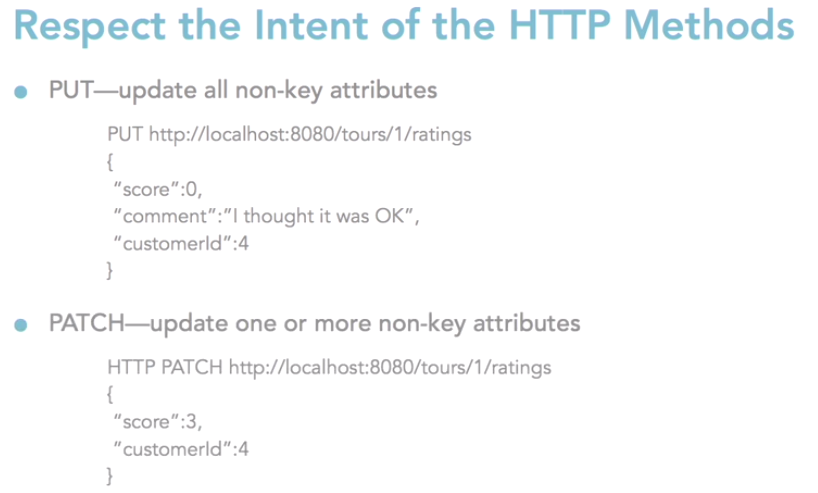


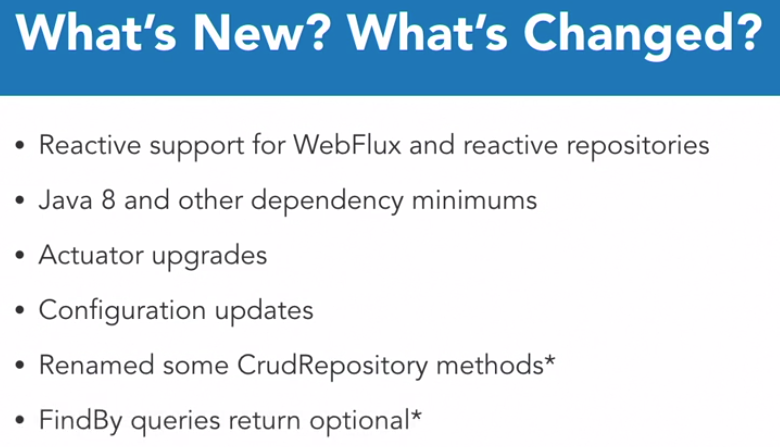
Annotating a class with @RestController tells Spring Web MVC that this class follows RESTful web service stereotypical behavior.

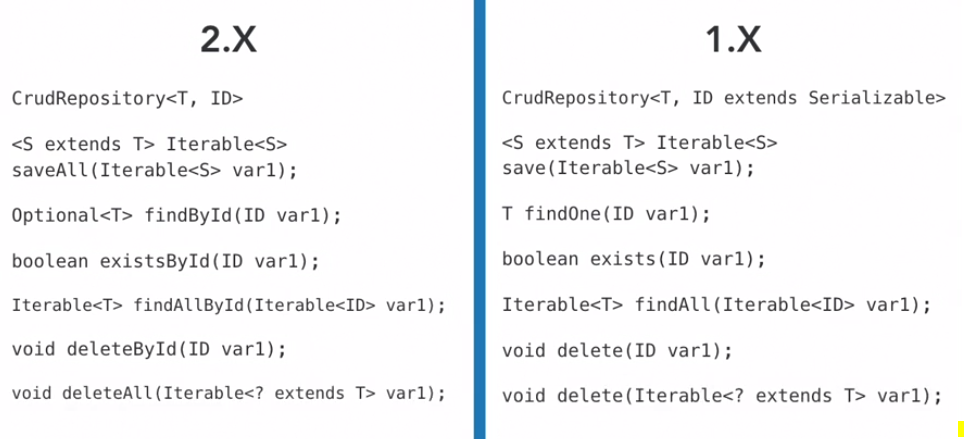


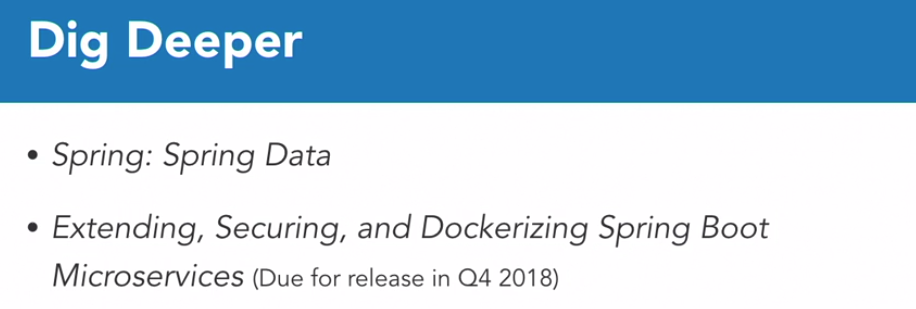
A RestController class mediates between web requests and the internals of the application. It is responsible for directing requests to the appropriate services, then sending a response back to the client. In Spring Data REST, the framework itself mediates between web requestsand the persistent domain model.

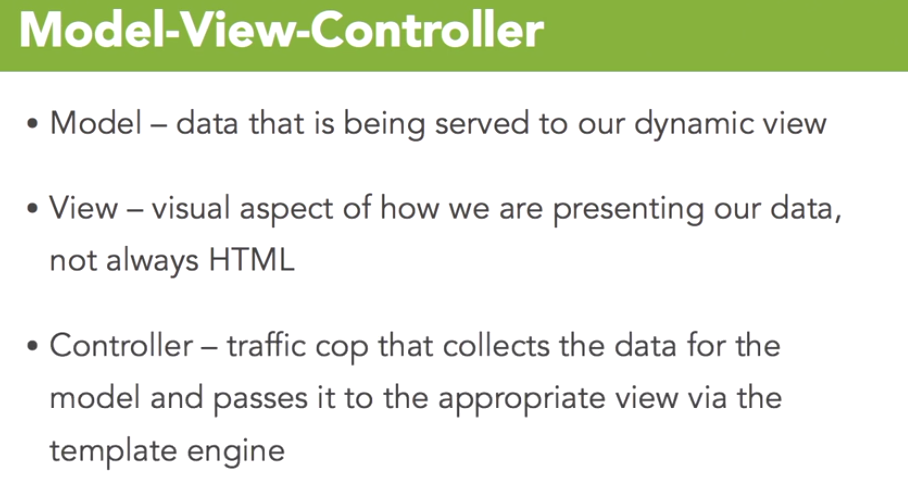


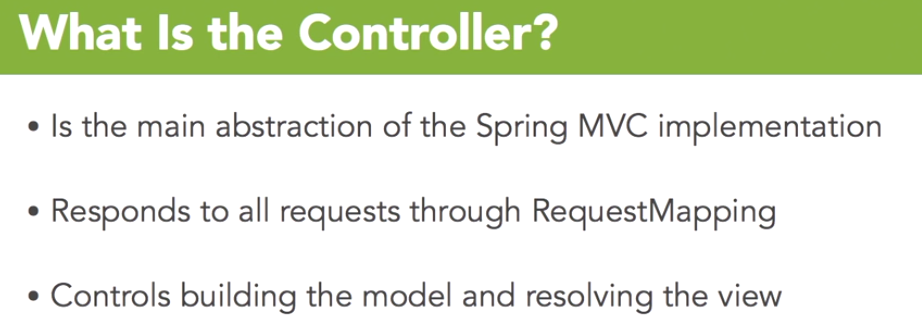


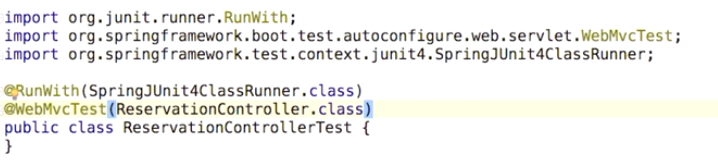












So, this will allow Spring to be involved in our test. We're also going to annotate this class with an indication of what you're going to be using, the mock MVC framework. So, we are going to run a web MVC test, and we are going to put the reservation controller under test. Now what this going to do is it's going to build an application context specifically for our controller, and we're going to mock all the dispatcher servlet and what not through the mock MVC itself.

This is a powerful tool. I use this a lot when writing web services specifically because I can build data sets, and then use a tool to scan the json that is returned for specific elements to make sure, not only am I returning the data, but I'm returning it in a specific fashion. We get a lot of the same value proposition on the web side, through again, this mock MVC framework. So, let's kind of get into this here now.

We're going to take a reservation service because we must have that in order for our controller to work. But we are going to do is we're going to annotate this with a mock bean annotation. Now much like the way that Mockito works, if you're familiar with that, it's going to take an interface from this class and build a mock around it. So, we are going to specify the data to return from this service prior to making calls to it.

We're going to also add an element, the mock MVC object, and we will go ahead and auto-wire that. Now when it comes to writing tests, I'm much less concerned about good object-oriented practices because these are just tests, and if you write tests, you've already won a large part of the battle because so many people don't do this. So, we also need to bring in an instance of the date formatter once again, so that we can create a date from a formatted string.

And again, we will follow the same pattern of four lowercase y's, hyphen, two uppercase n's, hyphen, two lowercase d's. Now we're going to build our tests. With JUnit4, the @test annotation allows you to indicate where your test is. We will create a void method. Name doesn't matter a lot. Now in this method, we need to create a date element, and we will do it through the date formatter.

Obviously, we could do this other way to create a simply date, but this way is just as easy as any other. Now we're going to have to build the data that we want to come back from the service through a mock reservation list because that is what our business service will return. So, we will create a new array list, and I like to name my mocks as mocks, so that I always can kind of keep track of what is real data and what is data that I am passing into it.

So, we are going to create a mock room reservation to add to that list, and we will go ahead and populate that with some data. Now, everybody's going to have their own sort of pattern that they do if they write tests. This is kind of how I do names and values. The actual data that we're passing doesn't matter, if when we search for it in the result, it matches what we've passed in.

So, we need to set all the elements in our mock room reservation here, and this is part of the reason, I think, that a lot of people don't like writing tests, is that there is so much writing just to get the test up and running. But when you start using test-driven development and some of the more quality-focused practices of writing software, these types of things will save you.

I can't tell you how many times I've caught bugs in my code because I've done test-driven development and I've gone through this practice. So now that we've build our mock room reservation, let's go ahead and add that to the mock reservation list, and we are going to add the mock reservation. Alright, so our mock is ready to be served, and we're going to use the syntax specifically for the mock bean annotation, so on the reservation service, when we call the get room reservation for date, passing it a date, and that date again is what we will build from the date format.

We want to return a mock reservation list. So what this line 46 is actually doing is it is saying, when the system calls into the reservation service, which in this case we are using a mock bean, and it calls the method get room reservation for date, passing a date object that is equivalent to January 1st, 2017, we want that mock bean to return the mock reservation list that we just built.

Now, what we're going to do here then, is we're going to use the mock MVC framework, and we're going to do a perform action on it, and we are going to call a get operation on slash reservations with a query parameter of date equals to 2017, 01, 01, and we will expect a status of OK, and we will expect the content to contain a string of test comma JUnit.

Now obviously, case sensitivity matters. Okay, so now let's bring in these imports. And we're going to want to bring them in through the static lines, so import static org.hamcrest.CoreMatchers.containsString. We're going to bring in the org.springframework.test.web.servlet.request.

MockMVC Builders.get, and we're going to do the same for the other two that are missing here. Again, these are in the same mock MVC area, but we are going to go to test.web.servlet.result and then MockMVCResultMatchers.content, and then the results Matchers once again, and status.

So, at this point, our test is ready to run. So, if I go up here and I run my test, my JUnit will pop up, you will see the application context executing. If you look through our console here, you'll see that banner that we've seen so many times. The actual application context run and then the test executes. Just to show you that it's testing something, let's change our date to two, and I would expect at this point my test to fail, because that data will not be there, and indeed you will see my test has failed.

So that's how we're going to go through and test a simple controller method. Now obviously we can get as sophisticated as we want with this test framework, and I encourage you to take some time and learn its power and learn what you can do with the test frameworks themselves.

