Aprendizaje de primavera y arranque de primavera

Ken Kousen
ken.kousen@kousenit.com
13/02/2017

Tabla de contenido

Ejercicios

1. Crear un nuevo proyecto

1. Vaya a http://start.spring.io para acceder a Spring Initializr
2. En el menú desplegable "Generar un", cambie de "Proyecto Maven" a "Proyecto Gradle"
3. Especifique el Grupo como `com.oreilly` y el Artefacto como `demo`
4. Agregue las dependencias "Web" y "Thymeleaf"
5. Haga clic en el botón "Generar proyecto" para descargar un archivo zip que contiene los archivos del proyecto
6. Descomprima el archivo descargado "demo.zip" en cualquier directorio que desee (pero recuerde dónde está)
7. Import the project into your IDE
   a. If you are using IntelliJ IDEA, import the project by selecting the "Import Project" link on the Welcome page and navigating to the `build.gradle` file inside the unzipped archive
   b. If you are using Spring Tool Suite (or any other Eclipse-based tool), generate an Eclipse project using the included `gradlew` script: Navigate to the project root directory in a command window and run the following command:

```
> gradlew cleanEclipse eclipse
```

> **NOTE** On a Unix-based machine (including Macs), use `./gradlew` for the command

   c. Now you should be able to import the project into Eclipse as an existing Eclipse project (File → Import… → General → Existing Projects Into Workspace)
8. As part of the import process, the IDE will download all the required dependencies
9. Open the file `src/main/java/com/oreilly/demo/DemoApplication.java` and note that it contains a standard Java "main" method (with signature: `public static void main(String[] args)`)
10. Start the application by running this method. There won't be any web components available yet, but you can see the start up of the application in the command window.
11. Add a controller by creating a file called `com.oreilly.demo.controllers.HelloController` in the `src/main/java` directory

> **NOTE** The goal is to have the `HelloController` class in the `com.oreilly.demo.controllers` package starting at the root directory `src/main/java`

12. The code for the `HelloController` is:

```java
package com.oreilly.demo.controllers;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;

@Controller
public class HelloController {

    @GetMapping("/hello")
    public String sayHello(
            @RequestParam(value = "name", required = false,
                          defaultValue = "World") String name, Model model) {
        model.addAttribute("user", name);
        return "hello";
    }
}
```

13. Create a file called `hello.html` in the `src/main/resources/templates` folder
14. The code for the `hello.html` file is:

```html
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Hello, World!</title>
</head>
<body>
    <h2 th:text="'Hello, ' + ${user} + '!'"></h2>
</body>
</html>
```

15. Start up the application and navigate to http://localhost:8080/hello. You should see the string "Hello, World!" in the browser
16. Change the URL in the browser to http://localhost:8080/hello?name=Dolly. You should now see the string "Hello, Dolly!" in the browser
17. Shut down the application (there's no graceful way to do that — just hit the stop button in your IDE)
18. Add a home page to the app by creating a file called `index.html` in the `src/main/resources/static` folder

19. The code for the `index.html` file is:

```HTML
<!DOCTYPE HTML>
<html>
<head>
    <title>Hello, World!</title>
</head>
<body>
    <h2>Say hello <a href="/hello">here</a></h2>
</body>
</html>
```

20. From a command prompt in the root of the project, build the application:

```
> gradlew build
```

21. Now you can start the application with a generated executable jar file:

```
> java -jar build/libs/demo-0.0.1-SNAPSHOT.jar
```

22. Navigate to http://localhost:8080 and see the new home page. From there you can navigate to the greeting page, and manually try adding a `name` parameter to the URL there
23. Again stop the application (use Ctrl-C in the command window)
24. Start it one more time using a special gradle task:

```
> gradlew bootRun
```

25. When again you're happy the app is running properly, shut it down
26. Because the controller is a simple POJO, you can unit test it by simply instantiating the controller and calling its `sayHello` method directly. To do so, add a class called `HelloControllerUnitTests` to the `com.oreilly.demo.controllers` package in the *test* folder, `src/test/java`
27. The code for the test class is:

JAVA

```java
package com.oreilly.demo.controllers;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.ui.Model;
import org.springframework.validation.support.BindingAwareModelMap;

import static org.junit.Assert.*;

public class HelloControllerUnitTests {
    @Test
    public void testSayHello() throws Exception {
      HelloController controller = new HelloController();
      Model model = new BindingAwareModelMap();
      String result = controller.sayHello("World", model);
      assertEquals("World", model.asMap().get("user"));
      assertEquals("hello", result);
    }
}
```

28. Run the test by executing this class as a JUnit test. It should pass. It's not terribly useful, however, since it isn't affected by the request mapping or the request parameter.
29. To perform an integration test instead, use the `MockMVC` classes available in Spring. Create a new class called `HelloControllerIntegrationTests` in the `com.oreilly.demo.controllers` package in `src/test/java`
30. The code for the integration test is:

JAVA

```java
package com.oreilly.demo.controllers;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.http.MediaType;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.test.web.servlet.MockMvc;

import static org.hamcrest.Matchers.is;
import static
org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;

@RunWith(SpringRunner.class)
@WebMvcTest(HelloController.class)
public class HelloControllerIntegrationTests {
    @Autowired
    private MockMvc mvc;

    @Test
    public void testHelloWithoutName() throws Exception {
        mvc.perform(get("/hello").accept(MediaType.TEXT_PLAIN))
                .andExpect(status().isOk())
                .andExpect(view().name("hello"))
                .andExpect(model().attribute("user", is("World")));

    }

    @Test
    public void testHelloWithName() throws Exception {
        mvc.perform(get("/hello").param("name",
"Dolly").accept(MediaType.TEXT_PLAIN))
                .andExpect(status().isOk())
                .andExpect(view().name("hello"))
                .andExpect(model().attribute("user", is("Dolly")));
    }
}
```

31. The tests should pass successfully. One of the advantages of the `@WebMvcTest` annotation over the generic `@SpringBootTest` annotation is that it allows you to automatically inject an instance of `MockMvc`, as shown.

2. Add a Rest Controller

1. Add another class to the `com.oreilly.demo.controllers` package called `HelloRestController`. This controller will be used to model a RESTful web service, though at this stage it will be limited to HTTP GET requests (for reasons explained below).
2. Add the `@RestController` annotation to the class.
3. By default, REST controllers will serialize and deserialize Java classes into JSON data using the Jackson 2 JSON library, which is currently on the classpath by default. To have an object (other than a trivial `String`) to serialize, add a class called `Greeting` to the `com.oreilly.demo.entities` package. In a larger application, this would represent a domain class that you can store in a database or other persistent storage mechanism.
4. In the `Greeting` class, add a private attribute of type `String` called `greeting`.
5. Add a `getGreeting` method for the `greeting` attribute that returns the current greeting.
6. Add a constructor to `Greeting` that takes a `String` argument and saves it to the attribute.
7. Add a default constructor that does nothing. This constructor will be used by the JSON parser to convert a JSON response into an instance of `Greeting`.
8. Add an `equals` method, a `hashCode` method, and a `toString` method in the usual manner. A reasonable version would be:

JAVA

```java
package com.oreilly.demo.entities;

import java.util.Objects;

public class Greeting {
    private String greeting;

    public Greeting() {}  // used for de-serialization

    public Greeting(String greeting) {
        this.greeting = greeting;
    }

    public String getGreeting() {
        return greeting;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Greeting)) return false;
        Greeting gr = (Greeting) o;
        return Objects.equals(greeting, gr.greeting);
    }

    @Override
    public int hashCode() {
        return Objects.hash(greeting);
    }

    @Override
    public String toString() {
        return greeting;
    }
}
```

9. Back in the `HelloRestController`, add a method called `greet` that takes a `String` called `name` as an argument and returns a `Greeting`.
10. Annotate the `greet` method with a `@GetMapping` whose argument is `"/rest"`, which means that the URL to access the method will be http://localhost:8080/rest.
11. Add the `@RequestParam` annotation to the argument, with the properties `required` set to `false` and `defaultValue` set to `World`.
12. In the body of the method, return a new instance of `Greeting` whose constructor argument should be `"Hello, " + name + "!"`.
13. The full class looks like (note that the string concatenation has been replaced with a `String.format` method)

JAVA

```java
package com.oreilly.hello.controllers;

import com.oreilly.hello.entities.Greeting;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class HelloRestController {

    @GetMapping("/rest")
    public Greeting greet(@RequestParam(required = false,
            defaultValue = "World") String name) {
        return new Greeting(String.format("Hello, %s!", name));
    }
}
```

14. You can now run the application and check the behavior using either `curl` or a similar command-line tool, or simply accessing the URL in a browser, either with or without a name.
15. To create a test for the REST controller, use the `TestRestTemplate` class, which is a testing version of the `RestTemplate` that will be used in upcoming exercises. Add a class called `HelloRestControllerTests` in the `src/test/java` tree in the same package as the REST controller class.
16. Add the `@RunWith(SpringRunner.class)` annotation to the class.
17. This time, when adding the `@SpringBootTest` annotation, add the argument `webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT`. This will autoconfigure several properties of the test, including making a `TestRestTemplate` available to inject.
18. Autowire in a private attribute called `template` of type `TestRestTemplate`.
19. Add two tests, one for greetings without a name and one for greetings with a name.
20. The tests should look like:

JAVA

```
@Test
public void greetWithoutName() {
    ResponseEntity<Greeting> entity = template.getForEntity("/rest",
Greeting.class);
    assertEquals(HttpStatus.OK, entity.getStatusCode());
    assertEquals(MediaType.APPLICATION_JSON_UTF8,
entity.getHeaders().getContentType());
    Greeting response = entity.getBody();
    assertEquals("Hello, World!", response.getGreeting());
}

@Test
public void greetWithName() {
    Greeting response = template.getForObject("/rest?name=Dolly", Greeting.class);
    assertEquals("Hello, Dolly!", response.getGreeting());
}
```

21. The first test uses the `getForEntity` method of the template, which returns a `ResponseEntity<Greeting>`. The response entity gives access to the headers, so the two provided asserts check the status code and the media type of the response. The actual response is inside the body. By calling `getBody`, the response is returned as a de-serialized `Greeting` instance, which allows you to check its message.
22. The second test uses the `getForObject` method, which returns the de-serialized response directly. This is simpler, but does not allow access to the headers. You can use either approach in your code.
23. The tests should now pass. This application only checks HTTP GET requests, because the application doesn't have any way to save `Greeting` instances. Once that is added, you could include analogous POST, PUT, and DELETE operations.

3. Building a REST client

This exercise uses the Spring `RestTemplate` class to access a RESTful web service. The template is used to convert the response into an object for the rest of the system.

1. Create a new Spring Boot project (either by using the Initializr at http://start.spring.io or using your IDE) called `restclient`. Add the Web dependency, but no others are necessary
2. Create a service class called `JokeService` in a `com.oreilly.restclient.services` package under `src/main/java`
3. Add the annotation `@Service` to the class (from the `org.springframework.stereotype` package, so you'll need an `import` statement)
4. Add a private attribute to `JokeService` of type `RestTemplate` called `restTemplate`
5. Add a constructor to `JokeService` that takes a single argument of type `RestTemplateBuilder`

> **NOTE** Because there are so many possible configuration options, Spring does not automatically provide a `RestTemplate`. It does, however, provide a `RestTemplateBuilder`, which can be used to configure and create the `RestTemplate`

6. Inside the constructor, invoke the `build()` method on the `RestTemplateBuilder` and assign the result to the `restTemplate` attribute

> **NOTE** If you provide only a single constructor in a class, you do not need to add the `@Autowired` annotation to it. Spring will inject the arguments anyway

7. The site providing the joke API is http://icndb.com, the Internet Chuck Norris Database. The site exposes the jokes through the URL http://api.icndb.com. The API supports a few properties that will be useful here: the client can specify the hero's first and last names and the joke category.
8. For our service, add a `private`, `final`, `static String` constant attribute called `BASE` and assign it to the URL "http://api.icndb.com/jokes/random?limitTo=[nerdy]"
9. Add a `public` method to the service called `getJoke` that takes two `String` arguments, `first` and `last` and returns a `String`
10. Inside the method, create the full URL for the API:

```java
String url = String.format("%s&firstName=%s&lastName=%s", BASE, first, last);
```

11. The `RestTemplate` class has a `getForObject` method that takes two arguments: the URL and the class to instantiate with the resulting JSON response. Note on the web page that the resulting JSON takes the form:

```javascript
{
  "type": "success",
  "value": {
    "id": 268,
    "joke": "Time waits for no man. Unless that man is Chuck Norris."
  }
}
```

12. Since there are only two nested JSON objects, you can create a class that models them with an inner class. Create a new class called `JokeResponse` in the `com.oreilly.restclient.json` package

13. The code for the `JokeResponse` class is shown below. Note how the properties match the keys in the JSON response exactly. You can use annotations from the included Jackson 2 JSON parser to map them if you like, but in this case it's easy enough to make them the same.

JAVA

```java
package com.oreilly.restclient.json;

public class JokeResponse {
    private String type;
    private Value value;

    public String getType() {
        return type;
    }

    public void setType(String type) {
        this.type = type;
    }

    public Value getValue() {
        return value;
    }

    public void setValue(Value value) {
        this.value = value;
    }

    public class Value {
        private int id;
        private String joke;

        public String getJoke() {
            return joke;
        }

        public void setJoke(String joke) {
            this.joke = joke;
        }
    }
}
```

14. Now the JSON response from the web service can be converted into an instance of the `JokeResponse` class. Add a line to do that inside the `getJoke` method:

JAVA

```java
JokeResponse jokeResponse = restTemplate.getForObject(url, JokeResponse.class);
```

15. Return the value of the `joke` field inside the nested `Value` object:

JAVA

```java
return jokeResponse.getValue().getJoke();
```

16. It will be convenient to log the jokes to the console. Spring Boot provides loggers from a variety of sources. In this case, use the one from the SLF4J library by adding an attribute to the `JokeService` class:

JAVA

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

// ...

private Logger logger = LoggerFactory.getLogger(JokeService.class);
```

17. Use the logger inside the `getJoke` method to log the joke:

JAVA

```
logger.info(jokeResponse.getValue().getJoke());
```

18. To demonstrate how to use the service, create a test for it. Create a class called `JokeServiceTest` in the `com.oreilly.services` package under the test hierarchy, `src/test/java`.
19. The source for the test is:

JAVA

```
package com.oreilly.restclient.services;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;

import static org.hamcrest.CoreMatchers.containsString;
import static org.junit.Assert.assertThat;

@RunWith(SpringRunner.class)
@SpringBootTest
public class JokeServiceTest {
    @Autowired
    private JokeService service;

    @Test
    public void getJoke() throws Exception {
        String joke = service.getJoke("Craig", "Walls");
        assertTrue(joke.contains("Craig") ||
                   joke.contains("Walls"));
    }

}
```

20. Feel free to change the name of the hero to anyone you prefer. (Craig Walls is the author of both *Spring in Action* and *Spring Boot in Action*.)
21. Execute the test and make any needed corrections until it passes.

4. Accessing the Google Geocoder

Google provides a free geocoding web service that converts addresses into geographical coordinates.

This exercise uses the `RestTemplate` to access the Google geocoder and converts the responses into Java objects.

1. The documentation for the Google geocoder is at https://developers.google.com/maps/documentation/geocoding/intro. Take a look at the page there to see how the geocoder is intended to be used. The base URL for the service is (assuming you want JSON responses) https://maps.googleapis.com/maps/api/geocode/json?address=street,city,state. The `address` parameter needs to be URL encoded and the parts of the address are joined using commas.

> **NOTE** The address components can be anything appropriate to the host country. The URL includes a string which separates the values by commas. The components don't have to be street, city, and state.

2. Rather than creating a new project, we'll add a `GeocoderService` to the existing `restclient` project. In that project, add the new class to the `services` package
3. Add the `@Service` annotation to the class so that Spring will automatically load and manage the bean during its component scan at start up.
4. Give the class an attribute of type `RestTemplate` called `restTemplate`
5. Add a constructor to the class that takes an argument of type `RestTemplateBuilder` called `builder`
6. Inside the constructor, set the value of the `restTemplate` field by invoking the `build` method on the builder
7. Map the JSON response to classes in a `json` package. The JSON response for the URL https://maps.googleapis.com/maps/api/geocode/json?address=1600+Amphitheatre+Parkway,Mountain+View,CA is:

JAVASCRIPT


```
{
   "results" : [
      {
         "address_components" : [
            {
               "long_name" : "1600",
               "short_name" : "1600",
               "types" : [ "street_number" ]
            },
            {
               "long_name" : "Amphitheatre Pkwy",
               "short_name" : "Amphitheatre Pkwy",
               "types" : [ "route" ]
            },
            {
               "long_name" : "Mountain View",
               "short_name" : "Mountain View",
               "types" : [ "locality", "political" ]
            },
            {
               "long_name" : "Santa Clara County",
               "short_name" : "Santa Clara County",
               "types" : [ "administrative_area_level_2", "political" ]
            },
            {
               "long_name" : "California",
               "short_name" : "CA",
               "types" : [ "administrative_area_level_1", "political" ]
            },
            {
               "long_name" : "United States",
               "short_name" : "US",
               "types" : [ "country", "political" ]
            },
            {
               "long_name" : "94043",
               "short_name" : "94043",
               "types" : [ "postal_code" ]
            }
         ],
         "formatted_address" : "1600 Amphitheatre Parkway, Mountain View, CA 94043,
   USA",
         "geometry" : {
            "location" : {
               "lat" : 37.4224764,
               "lng" : -122.0842499
            },
            "location_type" : "ROOFTOP",
            "viewport" : {
               "northeast" : {
                  "lat" : 37.4238253802915,
                  "lng" : -122.0829009197085
               },
               "southwest" : {
                  "lat" : 37.4211274197085,
                  "lng" : -122.0855988802915
```

```
                }
             }
          },
          "place_id" : "ChIJ2eUgeAK6j4ARbn5u_wAGqWA",
          "types" : [ "street_address" ]
       }
    ],
    "status" : "OK"
}
```

No nos interesan los componentes de la dirección, aunque la dirección formateada parece útil. En un `json` subpaquete, cree las siguientes clases:

JAVA

```java
package com.oreilly.restclient.json;

import java.util.List;

public class Response {
    private List<Result> results;
    private String status;

    public String getStatus() {
        return status;
    }

    public void setStatus(String status) {
        this.status = status;
    }

    public List<Result> getResults() {
        return results;
    }

    public void setResults(List<Result> results) {
        this.results = results;
    }

    public Location getLocation() {
        return results.get(0).getGeometry().getLocation();
    }

    public String getFormattedAddress() {
        return results.get(0).getFormattedAddress();
    }
}

package com.oreilly.restclient.json;

public class Result {
    private String formattedAddress;
    private Geometry geometry;

    public String getFormattedAddress() {
        return formattedAddress;
    }

    public void setFormattedAddress(String formattedAddress) {
        this.formattedAddress = formattedAddress;
    }

    public Geometry getGeometry() {
        return geometry;
    }

    public void setGeometry(Geometry geometry) {
        this.geometry = geometry;
    }
}
```

```
package com.oreilly.restclient.json;

public class Geometry {
    private Location location;

    public Location getLocation() {
        return location;
    }

    public void setLocation(Location location) {
        this.location = location;
    }
}
```

```
package com.oreilly.restclient.json;

public class Location {
    private double lat;
    private double lng;

    public double getLat() {
        return lat;
    }

    public void setLat(double lat) {
        this.lat = lat;
    }

    public double getLng() {
        return lng;
    }

    public void setLng(double lng) {
        this.lng = lng;
    }

    public String toString() {
        return String.format("(%s,%s)", lat, lng);
    }
}
```

8. En la `GeocoderService` clase, agregue constantes para la URL base y una clave.

JAVA

```
private static final String BASE =
"https://maps.googleapis.com/maps/api/geocode/json";
private static final String KEY = 'AIzaSyDw_d6dfxDEI7MAvqfGXEIsEMwjC1PWRno';
```

9. Agregue un `public` método que formule la URL completa con una dirección codificada y la convierta en un `Response` objeto. El codigo es:

JAVA

```
private String encodeString(String s) {
    try {
        return URLEncoder.encode(s,"UTF-8");
    } catch (UnsupportedEncodingException e) {
        e.printStackTrace();
    }
    return s;
}

public Site getLatLng(String... address) {
    String encodedAddress = Stream.of(address)
            .map(this::encodeString)
            .collect(Collectors.joining(","));
    String url = String.format("%s?address=%s&key=%s", BASE, encodedAddress, KEY);
    Response response = restTemplate.getForObject(url, Response.class);
    return new Site(response.getFormattedAddress(),
            response.getLocation().getLat(),
            response.getLocation().getLng());
}
```

El uso del `private` método es para evitar el bloque try / catch dentro del `map` método directamente, solo para mejorar la legibilidad.

10. Para que esto funcione, necesitamos una entidad llamada `Site` . Agregue un POJO al `com.oreilly.restclient.entities` paquete llamado `Site` que envuelve una cadena de dirección formateada y duplica la latitud y la longitud. El codigo es:

JAVA

```java
package com.oreilly.restclient.entities;

public class Site {

    private Integer id;
    private String name;
    private double latitude;
    private double longitude;

    public Site() {}

    public Site(String name, double latitude, double longitude) {
        this.name = name;
        this.latitude = latitude;
        this.longitude = longitude;
    }

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public double getLatitude() {
        return latitude;
    }

    public void setLatitude(double latitude) {
        this.latitude = latitude;
    }

    public double getLongitude() {
        return longitude;
    }

    public void setLongitude(double longitude) {
        this.longitude = longitude;
    }

    @Override
    public String toString() {
        return "Site{" +
                "name='" + name + '\'' +
                ", latitude=" + latitude +
                ", longitude=" + longitude +
                '}';
```

```
    }
}
```

11. Ahora necesitamos una prueba para asegurarnos de que funciona correctamente. Agregue una clase de prueba llamada `GeocoderServiceTests` al `com.oreilly.restclient.services` paquete en el directorio de prueba `src/test/java`.
12. Agregue las anotaciones de prueba a la prueba:

JAVA

```
@RunWith(SpringRunner.class)
@SpringBootTest
```

13. Cablear automáticamente en el `GeocoderService` campo llamado `service`
14. Agregue dos pruebas: una con una ciudad y estado de Boston, MA, y otra con una dirección de 1600 Ampitheatre Parkway, Mountain View, CA. Las pruebas son:

JAVA

```
@Test
public void getLatLngWithoutStreet() throws Exception {
    Site site = service.getLatLng("Boston", "MA");
    assertEquals(42.36, site.getLatitude(), 0.01);
    assertEquals(-71.06, site.getLongitude(), 0.01);
}

@Test
public void getLatLngWithStreet() throws Exception {
    Site site = service.getLatLng("1600 Ampitheatre Parkway",
            "Mountain View", "CA");
    assertEquals(37.42, site.getLatitude(), 0.01);
    assertEquals(-122.08, site.getLongitude(), 0.01);
}
```

15. Ejecute las pruebas y asegúrese de que pasen.
16. En realidad todavía tenemos un problema. Para verlo, registre el `Site` objeto devuelto en la consola. Primero agregue un registrador SLF4J al `GeocoderService`

JAVA

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

// ...

private Logger logger = LoggerFactory.getLogger(GeocoderService.class);
```

17. Luego, en el `getLatLng` método, después de crear una `Site` instancia, regístrela antes de devolverla.

JAVA

```
public Site getLatLng(String... address) {
    // ... as before, but instead of returning Site directly ...
    Site site = new Site(response.getFormattedAddress(),
            response.getLocation().getLat(),
            response.getLocation().getLng());
    logger.info(site.toString());
    return site;
}
```

18. Ejecute una o ambas pruebas y mire los sitios registrados.
19. The name of the site is null! That's because our `Result` class has a `String` field called `formattedAddress`, but the JSON response uses underscores instead of camel case (i.e., `formatted_address`).

    There are a couple of different ways to solve this. As a one-time fix, you can add an annotation to the `formatted_address` field in the `Result` class

    JAVA

    ```
    import com.fasterxml.jackson.annotation.JsonProperty;

    public class Result {
        @JsonProperty("formatted_address")
        private String formattedAddress;

        // ... rest as before ...
    ```

    The `@JsonProperty` annotation is a general purpose mechanism you can use whenever the property in the bean does not match the JSON field. Run your test again and see that the `name` value in the `Site` is now correct.
20. The other way to fix the issue is to set a global property that converts all camel case properties to underscores during the JSON parsing process. To use this, first remove the `@JsonProperty` annotation from `Result`.
21. We will then add the required property to a YAML properties file. By default, Spring Boot generates a file called `application.properties` in the `src/main/resources` folder. Rename that file to `application.yml`
22. Inside `application.yml`, add the following setting:

    YML

    ```
    spring:
      jackson:
        property-naming-strategy: CAMEL_CASE_TO_LOWER_CASE_WITH_UNDERSCORES
    ```

23. Once again run the tests and see that the `name` field in `Site` is set correctly. The advantage of the YAML file is that you can nest multiple properties without too much code duplication.

We'll return to this project in later exercises to (1) save the `Site` instances in a database, (2) expose the instances via REST calls, and (3) plot them on a Google map.

5. Using the JDBC template

Spring provides a class called `JdbcTemplate` in the `org.springframework.jdbc.core` package. All it needs in order to work is a data source. It removes almost all the boilerplate code normally associated with JDBC. In this exercise, you'll use the `JdbcTemplate` to implement the standard CRUD (create, read, update, delete) methods on an entity.

1. Make a new Spring Boot project with group `com.oreilly` and artifact called `persistence` using the Spring Initializr. Generate a Gradle build file and select the JPA dependency, which will include JDBC. Also select the H2 dependency, which will provide a JDBC driver for the H2 database as well as a connection pool.
2. Import the project into your IDE in the usual manner.
3. For this exercise, as well as the related exercises using JPA and Spring Data, we'll use a domain class called `Officer`. An `Officer` will have a generated `id` of type `Integer`, strings for `firstName` and `lastName`, and a `Rank`. The `Rank` will be a Java enum.
4. First define the `Rank` enum in the `com.oreilly.persistence.entities` package and give it a few constants:

JAVA

```
public enum Rank {
    ENSIGN, LIEUTENANT, COMMANDER, CAPTAIN, COMMODORE, ADMIRAL
}
```

5. Now add the `Officer` class with the attributes as specified.

JAVA

```java
public class Officer {
    private Integer id;
    private Rank rank;
    private String first;
    private String last;

    public Officer() {}

    public Officer(Rank rank, String first, String last) {
        this.rank = rank;
        this.first = first;
        this.last = last;
    }

    public Officer(Integer id, Rank rank, String first, String last) {
        this.id = id;
        this.rank = rank;
        this.first = first;
        this.last = last;
    }

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public Rank getRank() {
        return rank;
    }

    public void setRank(Rank rank) {
        this.rank = rank;
    }

    public String getFirst() {
        return first;
    }

    public void setFirst(String first) {
        this.first = first;
    }

    public String getLast() {
        return last;
    }

    public void setLast(String last) {
        this.last = last;
    }

    @Override
    public String toString() {
        return "Officer{" +
```

```
                "id=" + id +
                ", rank=" + rank +
                ", first='" + first + '\'' +
                ", last='" + last + '\'' +
                '}';
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Officer)) return false;

        Officer officer = (Officer) o;

        if (!id.equals(officer.id)) return false;
        if (rank != officer.rank) return false;
        if (first != null ? !first.equals(officer.first) : officer.first != null)
return false;
        return last.equals(officer.last);
    }

    @Override
    public int hashCode() {
        int result = id.hashCode();
        result = 31 * result + rank.hashCode();
        result = 31 * result + (first != null ? first.hashCode() : 0);
        result = 31 * result + last.hashCode();
        return result;
    }
}
```

6. One of the features of Spring Boot is that you can create and populate database tables by define scripts with the names `schema.sql` and `data.sql` in the `src/main/resources` folder. First define the database table in `schema.sql`

```SQL
DROP TABLE IF EXISTS officers;
CREATE TABLE officers (
  id         INT         NOT NULL AUTO_INCREMENT,
  rank       VARCHAR(20) NOT NULL,
  first_name VARCHAR(50) NOT NULL,
  last_name  VARCHAR(20) NOT NULL,
  PRIMARY KEY (id)
);
```

7. Next populate the table by adding the following `INSERT` statements in `data.sql`

```sql
INSERT INTO officers(rank, first_name, last_name) VALUES('CAPTAIN', 'James',
'Kirk');
INSERT INTO officers(rank, first_name, last_name) VALUES('CAPTAIN', 'Jean-Luc',
'Picard');
INSERT INTO officers(rank, first_name, last_name) VALUES('CAPTAIN', 'Benjamin',
'Sisko');
INSERT INTO officers(rank, first_name, last_name) VALUES('CAPTAIN', 'Kathryn',
'Janeway');
INSERT INTO officers(rank, first_name, last_name) VALUES('CAPTAIN', 'Jonathan',
'Archer');
```

8. When Spring starts up, the framework will automatically create a DB connection pool based on the H2 driver and then create and populate the database tables for you. Now we need a DAO (data access object) interface holding the CRUD methods that will be implemented in the different technologies. Define a Java interface called `OfficerDAO` in the `com.oreilly.persistence.dao` package.

```java
package com.oreilly.persistence.dao;

import com.oreilly.persistence.entities.Officer;

import java.util.Collection;
import java.util.Optional;

public interface OfficerDAO {
    Officer save(Officer officer);
    Optional<Officer> findById(Integer id);
    List<Officer> findAll();
    long count();
    void delete(Officer officer);
    boolean existsById(Integer id);
}
```

   As an aside, the names and signatures of these methods were chosen for a reason, which will become obvious when you do the Spring Data implementation later

9. In this exercise, implement the interface using the `JdbcTemplate` class. Start by creating a class in the `com.oreilly.persistence.dao` package called `JdbcOfficerDAO`. Inject a `DataSource` using the constructor and from it instantiate the `JdbcTemplate`

JAVA

```
public class JdbcOfficerDAO implements OfficerDAO {
    private JdbcTemplate jdbcTemplate;

    @Autowired
    public JdbcOfficerDAO(DataSource dataSource) {
        jdbcTemplate = new JdbcTemplate(dataSource);
    }

// ... more to come ...

}
```

10. Para hacer que Spring detecte esto como un bean que debería administrar, agregue la `@Repository` anotación a la clase

JAVA

```
@Repository
public class JdbcOfficerDAO implements OfficerDAO {
    // ... as before ...
}
```

11. Algunos de los métodos DAO son trivialmente fáciles de implementar. Implemente el `count` método ejecutando un `queryForObject` que usa una `SELECT count(*)` instrucción SQL y asigna el resultado a un largo.

JAVA

```
@Override
public long count() {
    return jdbcTemplate.queryForObject(
        "select count(*) from officers", Long.class);
}
```

12. Del mismo modo, el `delete` método es fácil de implementar utilizando el `update` método de la clase de plantilla. La parte interesante es que al poner un `?` comodín en la instrucción SQL, la plantilla usará automáticamente a `PreparedStatement` para ejecutar el SQL

JAVA

```
@Override
public void delete(Officer officer) {
    jdbcTemplate.update("DELETE FROM officers WHERE id=?", officer.getId());
}
```

13. El `exists` método también usa a `PreparedStatement` con an `id` , pero esta vez el resultado debe asignarse a un booleano.

JAVA

```
@Override
public boolean existsById(Integer id) {
    return jdbcTemplate.queryForObject(
        "SELECT EXISTS(SELECT 1 FROM officers where id=?)", Boolean.class, id);
}
```

14. Ahora para los métodos de búsqueda. Cuando una consulta SQL produce un `ResultSet` , la plantilla solicita una implementación de la `RowMapper` interfaz como otro argumento para el `queryForObject` método. Esta interfaz tiene un único método abstracto llamado `mapRow` , que toma el `ResultSet` y un número de fila como argumentos. La implementación luego usa los argumentos para convertir una fila del conjunto de resultados en una instancia de la clase de dominio. Para hacer esto, implemente aquí el `findById` método en términos de una consulta usando una clase interna anónima estándar que funciona en Java 7 y siguientes para `RowMapper`

JAVA

```
@Override
public Optional<Officer> findById(Integer id) {
    if (!existsById(id)) return Optional.empty();
    return Optional.of(jdbcTemplate.queryForObject(
        "SELECT * FROM officers WHERE id=?",
            new RowMapper<Officer>() {  // Java 7 anonymous inner class
                @Override
                public Officer mapRow(ResultSet rs, int rowNum) throws SQLException
{
                    return new Officer(rs.getInt("id"),
                            Rank.valueOf(rs.getString("rank")),
                            rs.getString("first_name"),
                            rs.getString("last_name"));
                }
            },
            id));
}
```

15. Se puede usar el mismo mapeador de filas para encontrar todas las instancias de `Officer` . El `JdbcTemplate` utiliza el `query` método para repetir automáticamente el conjunto de resultados, llamando al asignador de fila para cada fila para convertirla en una `Officer` , y finalmente devuelve una colección de la Mesa. Esta vez, sin embargo, aproveche Java 8 utilizando una expresión lambda para implementar el mapeador de filas.

JAVA

```
@Override
public List<Officer> findAll() {
    return jdbcTemplate.query("SELECT * FROM officers",
            (rs, rowNum) -> new Officer(rs.getInt("id"), // Java 8 lambda
expression
                    Rank.valueOf(rs.getString("rank")),
                    rs.getString("first_name"),
                    rs.getString("last_name")));
}
```

La implementación del mapeador de filas es exactamente la misma, pero usa una expresión lambda de Java 8 en lugar de la clase interna anónima. El tipo de retorno es un `Collection<Officer>`

16. Finalmente, para el inserto, tomaremos un enfoque diferente. Si bien puede escribir la instrucción de inserción de SQL y usar el `update` método en el `JdbcTemplate` , no hay una manera fácil de devolver la clave primaria generada. Entonces, en su lugar, usemos una clase relacionada llamada a `SimpleJdbcInsert` . Agregue esa clase como un atributo e instancia y configúrela en el constructor

JAVA

```
public class JdbcOfficerDAO implements OfficerDAO {
    // ... jdbcTemplate from earlier ...
    private SimpleJdbcInsert insertOfficer;

    @Autowired
    public JdbcOfficerDAO(DataSource dataSource) {
        // ... jdbcTemplate from earlier ...
        insertOfficer = new SimpleJdbcInsert(jdbcTemplate)
                .withTableName("officers")
                .usingGeneratedKeyColumns("id");
    }
```

Observe cómo puede especificar la tabla que utilizará la inserción, así como las columnas de clave generadas.

17. Implemente el `save` método usando la `SimpleJdbcInsert` instancia

JAVA

```
@Override
public Officer save(Officer officer) {
    Map<String,Object> parameters = new HashMap<>();
    parameters.put("rank", officer.getRank());
    parameters.put("first_name", officer.getFirst());
    parameters.put("last_name", officer.getLast());
    Integer newId = (Integer) insertOfficer.executeAndReturnKey(parameters);
    officer.setId(newId);
    return officer;
}
```

Observe el enfoque típico de Spring: hay una interfaz en la biblioteca llamada `SqlParameterSource` junto con varias clases de implementación, una de las cuales es `MapSqlParameterSource` . Cualquiera de ellos puede usarse como argumento del `executeAndReturnKey` método.

18. Necesitamos un caso de prueba para asegurarnos de que todo funcione correctamente. Cree una clase de prueba llamada `JdbcOfficerDAOTest` autowires en la clase DAO

JAVA

```java
@SpringBootTest
@RunWith(SpringRunner.class)
public class JdbcOfficerDAOTest {
    @Autowired
    private OfficerDAO dao;

// ... more to come ...
}
```

19. Ahora viene la parte divertida: agregue la `@Transactional` anotación a la clase. En una clase de prueba como esta, Spring interpretará que eso significa que cada prueba debe ejecutarse en una transacción que se *revierte al final de la prueba* . Eso evitará que la base de datos de prueba se vea afectada por las pruebas y mantendrá las pruebas en sí mismas, todas independientes

20. Agregar una prueba para el `save` método

JAVA

```java
@Test
public void save() throws Exception {
    Officer officer = new Officer(Rank.LIEUTENANT, "Nyota", "Uhuru");
    officer = dao.save(officer);
    assertNotNull(officer.getId());
}
```

La presencia de la `@Transactional` anotación significa que se agregará el nuevo oficial, y podemos verificar que el `id` valor se genera correctamente, pero al final de la prueba, el inserto se revertirá

21. Pruebe `findById` pero utilizando uno de los identificadores conocidos (que se conocen porque la base de datos se restablece cada vez)

JAVA

```
@Test
public void findByIdThatExists() throws Exception {
    Optional<Officer> officer = dao.findById(1);
    assertTrue(officer.isPresent());
    assertEquals(1, officer.get().getId().intValue());
}

@Test
public void findByIdThatDoesNotExist() throws Exception {
    Optional<Officer> officer = dao.findById(999);
    assertFalse(officer.isPresent());
}
```

22. La prueba para el método de recuento también se basa en conocer el número de filas en la base de datos de prueba

JAVA

```
@Test
public void count() throws Exception {
    assertEquals(5, dao.count());
}
```

23. El resto de las pruebas son bastante sencillas, aparte del hecho de que usaremos construcciones Java 8 para implementarlas.

JAVA

```
@Test
public void findAll() throws Exception {
    List<String> dbNames = dao.findAll().stream()
            .map(Officer::getLast)
            .collect(Collectors.toList());
    assertThat(dbNames, containsInAnyOrder("Kirk", "Picard", "Sisko", "Janeway",
"Archer"));
}

@Test
public void delete() throws Exception {
    IntStream.rangeClosed(1, 5)
            .forEach(id -> {
                Optional<Officer> officer = dao.findById(id);
                assertTrue(officer.isPresent());
                dao.delete(officer.get());
            });
    assertEquals(0, dao.count());
}

@Test
public void existsById() throws Exception {
    IntStream.rangeClosed(1, 5)
            .forEach(id -> assertTrue(String.format("%d should exist", id),
dao.existsById(id)));
}
```

Hablaremos sobre los detalles de estas pruebas en clase. Tenga en cuenta, sin embargo, que la prueba `delete` elimina a todos los oficiales de la mesa y verifica que se hayan ido. Eso sería un problema, excepto, una vez más, la reversión automática en la que confiamos al final de cada prueba.

24. Asegúrese de que todas las pruebas funcionen correctamente, luego haya terminado.

6. Implementando la capa CRUD usando JPA

La Java Persistence API (JPA) es una capa sobre los llamados proveedores de persistencia, el más común de los cuales es Hibernate. Con Spring regular, la configuración de JPA requiere varios beans, incluida una fábrica de administrador de entidades y un adaptador de proveedor de JPA. Afortunadamente, en Spring Boot, la presencia de la dependencia JPA hace que el marco implemente todo eso por usted.

1. Para usar JPA, necesitamos una entidad. Vamos a utilizar la misma `Officer` clase del ejercicio anterior, pero esta vez vamos a añadir las correspondientes anotaciones JPA `@Entity` , `@Id` , `@GeneratedValue` , `@Table` , y `@Column`

   JAVA

   ```
   @Entity
   @Table(name = "officers")
   public class Officer {
       @Id
       @GeneratedValue(strategy = GenerationType.IDENTITY)
       private Integer id;

       @Column(nullable = false)
       private Rank rank;

       @Column(nullable = false, name = "first_name")
       private String first;

       @Column(nullable = false, name = "last_name")
       private String last;

   // ... rest as before ...
   }
   ```

   (Tenga en cuenta que hemos dejado intencionalmente un problema que se solucionará en un paso posterior).

2. Cree una clase llamada `JpaOfficerDAO` que implemente la `OfficerDAO` interfaz y agregue un `EntityManagerFactory` como atributo

   JAVA

   ```
   @Repository
   public class JpaOfficerDAO implements OfficerDAO {
       @PersistenceContext
       private EntityManager entityManager;

   // ... more to come ...
   }
   ```

La `@PersistenceContext` anotación se utiliza para inyectar un administrador de entidades en el DAO. Normalmente, también deberíamos hacer que la clase sea transaccional, pero de acuerdo con la práctica común que se puede manejar en una capa de servicio. En este caso particular, sin embargo, haremos las transacciones en las pruebas.

3. La implementación de los métodos individuales es muy simple. Dado que este es un curso en primavera y no en JPA, se dan aquí sin comentarios. Añádelos a la `JpaOfficerDAO` clase

JAVA

```
@Override
public Officer save(Officer officer) {
    entityManager.persist(officer);
    return officer;
}

@Override
public Optional<Officer> findById(Integer id) {
    return Optional.ofNullable(entityManager.find(Officer.class, id));
}

@Override
public List<Officer> findAll() {
    return entityManager.createQuery("select o from Officer o", Officer.class)
                        .getResultList();
}

@Override
public long count() {
    return entityManager.createQuery("select count(o.id) from Officer o",
Long.class)
                        .getSingleResult();
}

@Override
public void delete(Officer officer) {
    entityManager.remove(officer);
}

@Override
    public boolean existsById(Integer id) {
        Long count = entityManager.createQuery(
                "select count(o.id) from Officer o where o.id=:id", Long.class)
                                  .setParameter("id", id)
                                  .getSingleResult();
        return count > 0;
    }
```

4. Las mismas pruebas utilizadas para verificar `JdbcOfficerDAO` se pueden hacer nuevamente, solo usando un DAO diferente como la clase bajo prueba, con una excepción:

JAVA

```java
@SpringBootTest
@RunWith(SpringRunner.class)
@Transactional
public class JpaOfficerDAOTest {
    @Autowired @Qualifier("jpaOfficerDAO")  // Note use of @Qualifier!
    private OfficerDAO dao;

    @Autowired
    private JdbcTemplate template;

    @Test
    public void testSave() throws Exception {
        Officer officer = new Officer(Rank.LIEUTENANT, "Nyota", "Uhuru");
        officer = dao.save(officer);
        assertNotNull(officer.getId());
    }

    @Test
    public void findOneThatExists() throws Exception {
        template.query("select id from officers", (rs, num) -> rs.getInt("id"))
                .forEach(id -> {
                    Optional<Officer> officer = dao.findById(id);
                    assertTrue(officer.isPresent());
                    assertEquals(id, officer.get().getId());
                });
    }

    @Test
    public void findOneThatDoesNotExist() throws Exception {
        Optional<Officer> officer = dao.findById(999);
        assertFalse(officer.isPresent());
    }

    @Test
    public void findAll() throws Exception {
        List<String> dbNames = dao.findAll().stream()
                                  .map(Officer::getLast)
                                  .collect(Collectors.toList());
        assertThat(dbNames, containsInAnyOrder("Kirk", "Picard", "Sisko",
"Janeway", "Archer"));
    }

    @Test
    public void count() throws Exception {
        assertEquals(5, dao.count());
    }

    @Test
    public void delete() throws Exception {
        template.query("select id from officers", (rs, num) -> rs.getInt("id"))
                .forEach(id -> {
                    Optional<Officer> officer = dao.findById(id);
                    assertTrue(officer.isPresent());
                    dao.delete(officer.get());
                });
        assertEquals(0, dao.count());
```

```
    }

    @Test
    public void existsById() throws Exception {
        template.query("select id from officers", (rs, num) -> rs.getInt("id"))
                .forEach(id -> assertTrue(String.format("%d should exist", id),
                        dao.existsById(id)));
    }

    @Test
    public void doesNotExist() {
        List<Integer> ids = template.query("select id from officers",
                                           (rs, num) -> rs.getInt("id"));
        assertThat(ids, not(contains(999)));
        assertFalse(dao.existsById(999));
    }
}
```

Debido a que ahora hay dos beans separados disponibles para Spring que implementan la misma `OfficerDAO` interfaz, la `@Autowired` anotación fallaría, alegando que esperaba un solo bean de ese tipo pero encontró dos. La `@Qualifier` anotación se usa para indicarle a Spring el nombre del bean que se debe inyectar. Sin embargo, *varias de las pruebas fallarán* porque tenemos otra configuración que tenemos que modificar

5. Si ejecuta las pruebas, verá que rápidamente nos encontramos con un problema, ¡que los datos de muestra no están allí! Esto se debe a que, por defecto, Hibernate está en lo que se llama modo "crear-soltar", lo que significa que descarta la base de datos después de cada ejecución y la vuelve a crear en el inicio. Sin embargo, podemos evitar eso agregando una configuración al `application.yml` archivo:

YAML

```
spring:
    jpa:
        show-sql: true
        hibernate:
            ddl-auto: update
```

Cambiamos la `spring.jpa.hibernate.ddl-auto` propiedad a `update` (otras opciones son `create`, `create-drop` y `validate`), que agregará columnas según sea necesario pero no descarte ninguna tabla o dato

6. Ahora las pruebas funcionarían normalmente, pero encontramos el problema mencionado brevemente cuando se analiza la entidad anterior. De manera predeterminada, Hibernate asigna un valor enumerado almacenando un índice entero del elemento, que no coincide con los datos de muestra almacenados, que utiliza una cadena. Cambie eso agregando otra anotación a la `Officer` clase

JAVA

```java
@Column(nullable = false)
@Enumerated(EnumType.STRING)
private Rank rank;
```

La `@Enumerated` anotación le dice a Hibernate que almacene el valor de la enumeración como una cadena en lugar de un índice. Ahora las pruebas deberían pasar.

7. Sin embargo, hay otro paso de limpieza requerido. Esta prueba debería pasar, pero `JdbcOfficerDAOTest` no será así porque también tenemos que agregarla `@Qualifier` .

JAVA

```java
public class JdbcOfficerDAOTest {
    @Autowired @Qualifier("jdbcOfficerDAO")
    private OfficerDAO dao;
```

Ahora ambas pruebas deberían funcionar correctamente.

7. Usando Spring Data

El proyecto Spring Data JPA hace que sea increíblemente fácil implementar una capa DAO. Extiende la interfaz adecuada y la infraestructura subyacente genera todas las implementaciones para usted.

Spring Data es una API grande y poderosa. En este ejercicio, solo mostraremos los conceptos básicos.

1. Dado que creamos este proyecto basado en la dependencia de Spring Data JPA, no necesitamos modificar el archivo de compilación de Gradle para agregarlo. Tenga en cuenta que el archivo de compilación ya incluye las dependencias requeridas:

   GROOVY
   ```
   dependencies {
           compile('org.springframework.boot:spring-boot-starter-data-jpa')
           runtime('com.h2database:h2')
           testCompile('org.springframework.boot:spring-boot-starter-test')
   }
   ```

2. Spring Data funciona definiendo una interfaz que extiende una de las pocas interfaces proporcionadas, donde especifica la clase de dominio y su tipo de clave principal. Por lo tanto, cree una interfaz llamada `OfficerRepository` en el `com.oreilly.persistence.dao` paquete

   JAVA
   ```
   public interface OfficerRepository extends JpaRepository<Officer, Integer> {
   }
   ```

   La interfaz puede extenderse `CrudRepository` , `PagingAndSortingRepository` o, como aquí `JpaRepository` ,. Solo tiene que especificar los dos parámetros genéricos que representan la clase de dominio y el tipo de clave primaria. Aquí usamos `Officer` y `Integer` .

   El marco ahora generará las implementaciones de aproximadamente una docena de métodos diferentes, incluidos todos los métodos enumerados en la `OfficerDAO` interfaz (razón por la cual esos métodos fueron elegidos en primer lugar)

3. Spring Data también generará las implementaciones de métodos adicionales que declaramos, siempre y cuando sigamos un patrón específico. Agregue métodos a la interfaz para encontrar oficiales por sus apellidos y por su rango

   JAVA
   ```
   List<Officer> findByRank(@Param("rank") Rank rank);
   List<Officer> findByLast(@Param("last") String last);
   ```

Porque `findByLast` estamos usando un tipo de devolución `Optional<Officer>`, en caso de que no haya oficiales disponibles en absoluto. Las `@Param` anotaciones se agregan para cuando exponemos los datos usando Spring Data Rest.

4. La clase de prueba es similar a las otras, excepto que está escrita en términos del `OfficerRepository` bean. Añadir una clase de prueba en la `src/test/java` llamada `OfficerRepositoryTest` en el `com.oreilly.persistence.dao` paquete

JAVA

```
@DataJpaTest
@RunWith(SpringRunner.class)
@Transactional
public class OfficerRepositoryTest {
    @Autowired
    private OfficerRepository repository;

// ... more to come ...
}
```

5. Esta vez estamos usando la anotación especial `@DataJpaTest` que está específicamente diseñada para manejar las pruebas de Spring Data. En lugar de valores de código duro para los identificadores, esta vez inyectamos uno `JdbcTemplate` que podemos usar para leer los identificadores de la base de datos

JAVA

```
@Autowired
private JdbcTemplate template
```

6. El resto de las pruebas se muestran a continuación. Observe cómo `JdbcTemplate` se usa para recuperar identificadores de la tabla. Además, tenemos una prueba adicional para el `findByLast` método que se agregó a la interfaz

JAVA

```java
@Test
public void testSave() throws Exception {
        Officer officer = new Officer(Rank.LIEUTENANT, "Nyota", "Uhuru");
        officer = repository.save(officer);
        assertNotNull(officer.getId());
}

@Test
public void findById() throws Exception {
        template.query("select id from officers", (rs, num) -> rs.getInt("id"))
                        .forEach(id -> {
                                Optional<Officer> officer =
repository.findById(id);
                                assertTrue(officer.isPresent());
                                assertEquals(id, officer.get().getId());
                        });
}

@Test
public void findAll() throws Exception {
        List<String> dbNames = repository.findAll().stream()

.map(Officer::getLast)

.collect(Collectors.toList());
        assertThat(dbNames, containsInAnyOrder("Kirk", "Picard", "Sisko",
"Janeway", "Archer"));
}

@Test
public void count() throws Exception {
        assertEquals(5, repository.count());
}

@Test
public void deleteById() throws Exception {
        template.query("select id from officers", (rs, num) -> rs.getInt("id"))
                        .forEach(id -> repository.deleteById(id));
        assertEquals(0, repository.count());
}

@Test
public void existsById() throws Exception {
        template.query("select id from officers", (rs, num) -> rs.getInt("id"))
                        .forEach(id -> assertTrue(String.format("%d should exist",
id),
                                        repository.existsById(id)));
}

@Test
public void doesNotExist() {
    List<Integer> ids = template.query("select id from officers",
                                    (rs, num) -> rs.getInt("id"));
    assertThat(ids, not(contains(999)));
    assertFalse(repository.existsById(999));
}
```

```
@Test
public void findByRank() throws Exception {
        repository.findByRank(Rank.CAPTAIN).forEach(captain ->
                        assertEquals(Rank.CAPTAIN, captain.getRank()));

}

@Test
public void findByLast() throws Exception {
        List<Officer> kirks = repository.findByLast("Kirk");
        assertEquals(1, kirks.size());
        assertEquals("Kirk", kirks.get(0).getLast());
}
```

7. Una vez que se ejecutan las pruebas, agregue dos dependencias al archivo de compilación de Gradle: una para el proyecto Spring Data Rest (que expondrá los datos a través de una interfaz REST) y para el navegador HAL, que nos dará un cliente conveniente para usar

GROOVY

```
compile('org.springframework.boot:spring-boot-starter-data-rest')
compile 'org.springframework.data:spring-data-rest-hal-browser'
```

8. Después de reconstruir el proyecto, inicie la aplicación (usando la clase con el método principal) y navegue a http: // localhost: 8080 . Spring insertará el navegador HAL en ese punto para permitirle agregar, actualizar y eliminar elementos individuales, lo que haremos en clase.

Última actualización 2018-08-14 12:41:10 EDT