


# Web API Design with Spring Boot Week 2 Coding Assignment

**Points possible:** 70




Category	Criteria	% of Grade
Functionality	Does the code work?	25
Organization	Is the code clean and organized? Proper use of white space, syntax, and consistency are utilized. Names and comments are concise and clear.	25
Creativity	Student solved the problems presented in the assignment using creativity and out of the box thinking.	25
Completeness	All requirements of the assignment are complete.	25

**Instructions:** In Eclipse, or an IDE of your choice, write the code that accomplishes the objectives listed below. Ensure that the code compiles and runs as directed. Take screenshots of the code and of the running program (make sure to get screenshots of all required functionality) and paste them in this document where instructed below. Create a new repository on GitHub for this week's assignments and push this document, with your Java project code, to the repository. Add the URL for this week's repository to this document where instructed and submit this document to your instructor when complete.

**Here's a friendly tip:** as you watch the videos, code along with the videos. This will help you with the homework. When a screenshot is required, look for the icon:  You will keep adding to this project throughout this part of the course. When it comes time for the final project, use this project as a starter.

**Project Resources:** <https://github.com/promineotech/Spring-Boot-Course-Student-Resources>


**Coding Steps:**


- 1) In the project you started last week, use Lombok to add an info-level logging statement in the controller implementation method that logs the parameters that were input to the method. Remember to add the `@Slf4j` annotation to the class.
- 2) Start the application (not an integration test). Use a browser to navigate to the application passing the parameters required for your selected operation. (A browser, used in this manner, sends an HTTP GET request to the server.) Produce a screenshot showing the browser navigation bar and the log statement that is in the IDE console showing that the controller method was reached (as in the video). 
- 3) With the application still running, use the browser to navigate to the OpenAPI documentation. Use the OpenAPI documentation to send a GET request to the server with a valid model and trim level. (You can get the model and trim from the provided `data.sql` file.) Produce a screenshot showing the `curl` command, the request URL, and the response headers. 
- 4) Run the integration test and show that the test status is green. Produce a screenshot of the test class and the status bar. 
- 5) Add a method to the test to return a list of expected `Jeep` (`model`) objects based on the model and trim level you selected. You can get the expected list of Jeeps from the file `src/test/resources/flyway/migrations/V1.1__Jeep_Data.sql`. So, for example, using the model Wrangler and trim level "Sport", the query should return two rows:

	Row 1	Row 2
<b>Model ID</b>	WRANGLER	WRANGLER
<b>Trim Level</b>	Sport	Sport
<b>Num Doors</b>	2	4
<b>Wheel Size</b>	17	17
<b>Base Price</b>	\$28,475.00	\$31,975.00

The method should be named `buildExpected()`, and it should return a `List` of `Jeep`. The video put this method into a support superclass but you can include it in the main test class if you want.

- 6) Write an AssertJ assertion in the test to assert that the actual list of jeeps returned by the server is the same as the expected list. Run the test. Produce a screenshot showing...
  - a) The test with the assertion.
  - b) The JUnit status bar (should be red).

- c) The method returning the expected list of Jeeps. 
- 7) Add a service layer in your application as shown in the videos:
- a) Add a package named `com.promineotech.jeepproject.service`.
  - b) In the new package, create an interface named `JeepSalesService`.
  - c) In the same package (service), create a class named `DefaultJeepSalesService` that implements the `JeepSalesService` interface. Add the class-level annotation, `@Service`.
  - d) Inject the service interface into `DefaultJeepSalesController` using the `@Autowired` annotation. The instance variable should be `private`, and the variable should be named `jeepSalesService`.
  - e) Define the `fetchJeeps` method in the interface. Implement the method in the service class. Call the method from the controller (make sure the controller returns the list of Jeeps returned by the service method). The method signature looks like this:  

```
List<Jeep> fetchJeeps(JeepModel model, String trim);
```
  - f) Add a Lombok info-level log statement in the service implementation showing that the service was called. Print the parameters passed to the method. Let the method return `null` for now.
  - g) Run the test again. Produce a screenshot showing the service class implementation, the log line in the console, and the red status bar. 
- 8) Add the database dependencies described in the video to the POM file (MySQL driver and Spring Boot Starter JDBC). To find them, navigate to <https://mvnrepository.com/>. Search for `mysql-connector-j` and `spring-boot-starter-jdbc`. In the POM file you don't need version numbers for either dependency because the version is included in the Spring Boot Starter Parent.
- 9) Create `application.yaml` in `src/main/resources`. Add the `spring.datasource.url`, `spring.datasource.username`, and `spring.datasource.password` properties to `application.yaml`. The url should be the same as shown in the video (`jdbc:mysql://localhost:3306/jeepproject`). The password and username should match your setup. If you created the database under your root user, the username is "root", and the password is the root user password. If you created a "jeep" user or other user, use the correct username and password.

Be careful with the indentation! YAML allows hierarchical configuration but it reads the hierarchy based on the indentation level. The keyword "spring" MUST start in the first column. It should look similar to this when done:

```
spring:

  datasource:

    username: username

    password: password

    url: jdbc:mysql://localhost:3306/jeep
```

- 10) Start the application (the real application, not the test). Produce a screenshot that shows application.yaml and the console showing that the application has started with no errors.



- 11) Add the H2 database as dependency. Search for the dependency in the Maven repository like you did above. Search for "h2" and pick the latest version. Again, you don't need the version number, but the scope should be set to "test".


- 12) Create application-test.yaml in src/test/resources. Add the setting spring.datasource.url that points to the H2 database. It should look like this:

```
spring:

  datasource:

    url: jdbc:h2:mem:jeep
```

You do not need to set the username and password because the in-memory H2 database does not require them.

Produce a screenshot showing application-test.yaml. 

## Screenshots of Code & Application Running:

Preview File Edit View Go Tools Window Help

localhost:8080/jeeps?model=WRANGLER&trim=Sport

Back-end Resources Learn to code | Cod... MT Code School Lo... Google Clean eating! Java | Hello World | ... Software Engineerin... Getting Started Stretches to do daily! Green Warrior Prote... iCloud >>

Problems Javadoc Declaration Console

jeep-sales - JeepSales (1) [Spring Boot App] /Library/Java/JavaVirtualMachines/dk-11.0.12.jdk/Contents/Home/bin/java (Sep 25, 2021, 4:10:37 PM)

```
2021-09-25 16:12:04.567 INFO 8837 --- [nio-8080-exec-2] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring DispatcherServlet 'dispatcherServlet'
2021-09-25 16:12:04.567 INFO 8837 --- [nio-8080-exec-2] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'
2021-09-25 16:12:04.568 INFO 8837 --- [nio-8080-exec-2] o.s.web.servlet.DispatcherServlet : Completed initialization in 1 ms
2021-09-25 16:12:11.116 INFO 8837 --- [nio-8080-exec-4] c.p.j.c.DefaultJeepSalesController : model=WRANGLER, trim=Sport
2021-09-25 16:13:46.311 INFO 8837 --- [nio-8080-exec-6] c.p.j.c.DefaultJeepSalesController : model=WRANGLER, trim=Sport
2021-09-25 16:14:33.858 INFO 8837 --- [nio-8080-exec-8] c.p.j.c.DefaultJeepSalesController : model=WRANGLER, trim=Sport
```

Responses

Curl

```
curl -X 'GET' \
'http://localhost:8080/jeeps?model=CHEROKEE&trim=Sport' \
-H 'accept: application/json'
```

Request URL

```
http://localhost:8080/jeeps?model=CHEROKEE&trim=Sport
```

Server response		
Code	Details	
200	<div>Response headers</div> <pre> connection: keep-alive content-length: 0 date: Sat, 25 Sep 2021 22:22:57 GMT keep-alive: timeout=60 </pre>	
Responses		
Code	Description	Links
200	<p>A list of Jeeps is returned.</p> <p>Media type</p> <div>application/json</div> <p>Controls Accept header:</p> <p>Example Value   Schema</p> <pre> {   "modelPK": 0,   "modelId": "WRANGLER",   "trimLevel": "string",   "numDoors": 0,   "wheelSize": 0,   "basePrice": 0 } </pre>	No links

Finished after 8.84 seconds

Runs: 1/1   Errors: 0   Failures: 0

FetchJeepTest [Runner: JUnit 5] (0.562 s)

testThatJeepsAreReturnedWhenValidModelAndTrimAreSupplied() (0.562 s)

Finished after 9.135 seconds

Runs: 1/1

✖ Errors: 0

✖ Failures: 1

FetchJeepTest [Runner: JUnit 5] (0.754 s)

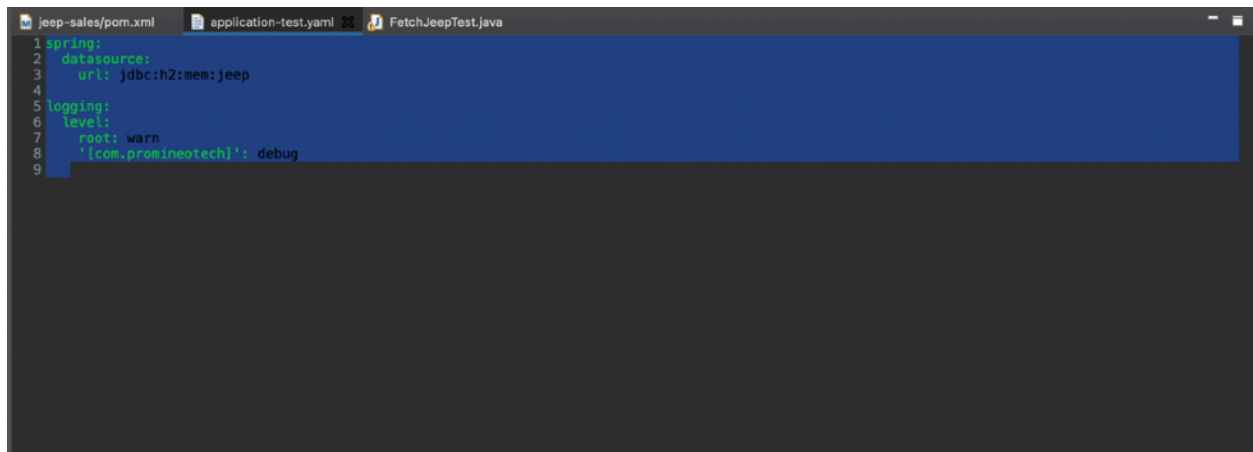
testThatJeepsAreReturnedWhenValidModelAndTrimAreSupplied() (0.754 s)

### Failure Trace

org.opentest4j.AssertionFailedError:  
expected: [Jeep(modelPk=null, modelId=WRANGLER, trimLevel=Sport, numDoors=2, wheelSize=16, ...)  
Jeep(modelPk=null, modelId=WRANGLER, trimLevel=Sport, numDoors=4, wheelSize=16, ...)]  
but was : null  
at com.promineotech.jeep.controller.FetchJeepTest.testThatJeepsAreReturnedWhenValidModelAndTrimAreSupplied()  
at java.base/java.util.ArrayList.forEach(ArrayList.java:1541)  
at java.base/java.util.ArrayList.forEach(ArrayList.java:1541)

```
45 ● @Test
46 void testThatJeepsAreReturnedWhenValidModelAndTrimAreSupplied() {
47     // Given: A valid model, trim and URI
48
49     JeepModel model = JeepModel.WRANGLER;
50     String trim = "Sport";
51     String uri =
52         String.format("http://localhost:%d/jeeps?model=%s&trim=%s", serverPort, model, trim);
53
54     // When: A connection is made to the URI
55
56     ResponseEntity<List<Jeep>> response =
57         getRestTemplate().exchange(uri, HttpMethod.GET, null, new ParameterizedTypeReference<>() {
58             });
59
60     // Then: A valid status code is returned
61
62     assertThat(response.getStatusCode()).isEqualTo(HttpStatus.OK);
63
64     List<Jeep> expected = buildExpected();
65     assertThat(response.getBody()).isEqualTo(expected);
66 }
67
68 // And: the actual list returned is the same as the expected list
69
70 protected List<Jeep> buildExpected() {
71     List<Jeep> list = new LinkedList<Jeep>();
72
73 }
```



A screenshot of an IDE window with three tabs: 'jeep-sales/pom.xml', 'application-test.yml', and 'FetchJeepTest.java'. The 'application-test.yml' tab is active, showing a YAML configuration for Spring. The configuration includes a 'spring' section with a 'datasource' subsection containing 'url: jdbc:h2:mem:jeep'. Below this is a 'logging' section with a 'level:' subsection containing 'root: warn' and a specific logger for 'com.promineotech' set to 'debug'. Line numbers 1 through 9 are visible on the left side of the editor.

```
1 spring:
2   datasource:
3     url: jdbc:h2:mem:jeep
4
5 logging:
6   level:
7     root: warn
8     'com.promineotech': debug
9
```

**URL to GitHub Repository:**