


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
Model ID	WRANGLER	WRANGLER
Trim Level	Sport	Sport
Num Doors	2	4
Wheel Size	17	17
Base Price	\$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.jeep.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/jeep). 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".

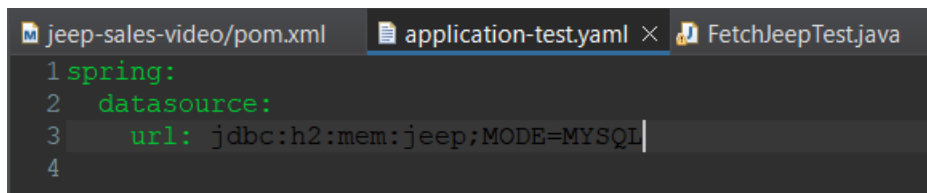
12) Create application-test.yaml in src/test/resources. Add the setting spring.datasource.url that points to the H2 database. It should look like this:

```
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:

A screenshot of an IDE showing the application-test.yaml file. The file content is:

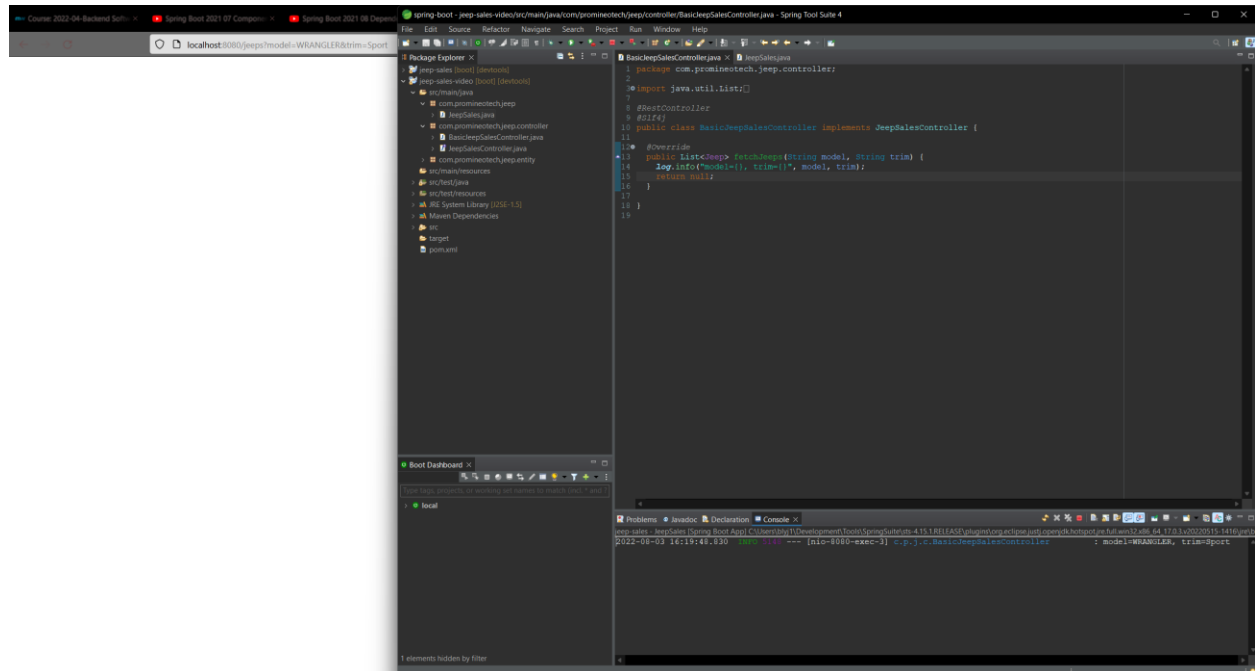
```
1 spring:
2   datasource:
3     url: jdbc:h2:mem:jeep;MODE=MYSQL
4
```

#12.

This is necessary to get this to work now according to Dr. Rob because there was some change that came along with updates in h2 something or other that requires the MODE=MYSQL now.

Screenshots of Running Application:

#2.



#3.

localhost:8080/swagger-ui/index.html#/basic-jeep-sales-controller/fetchJeeps

Parameters

Cancel

Name	Description
model string (query)	The model name
<input type="text" value="GLADIATOR"/>	
trim string (query)	The trim level
<input type="text" value="Sport"/>	

Execute

Clear

Responses

Curl

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

Request URL

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

Server response

Code	Details
200	<div>Response headers</div> <pre>connection: keep-alive content-length: 0 date: Wed, 03 Aug 2022 20:22:16 GMT keep-alive: timeout=60</pre>

#4.

spring-boot - jeep-sales-video/src/test/java/com/promineotech/jeep/controller/FetchJeepTest.java - Spring Tool Suite 4

File Edit Source Refactor Navigate Search Project Run Window Help

Package Explorer JUnit x

BasicJeepSalesController.java JeepSales.java FetchJeepTest.java x

Finished after 2.72 seconds

Runs: 1/1 Errors: 0 Failures: 0

> FetchJeepTest (Runner: JUnit 5) (0.388 s)

Failure Trace

```
1 package com.promineotech.jeep.controller;
2
3 import static org.assertj.core.api.Assertions.assertThat;
13
14 @SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
15 class FetchJeepTest extends FetchJeepTestSupport{
16
17     @Test
18     void testThatJeepsAreReturnedWhenAValidModelAndTrimAreSupplied() {
19         // Given a valid model, trim, and URI
20         JeepModel model = JeepModel.WRANGLER;
21         String trim = "Sport";
22         String uri = String.format("%s?model=%s&trim=%s", getBaseUri(), model, trim);
23
24
25         // When a connection is made to the URI
26         ResponseEntity<Jeep> response =
27             getRestTemplate().getForEntity(uri, Jeep.class);
28
29         // Then a success (OK - 200) is returned
30         assertThat(response.getStatusCode()).isEqualTo(HttpStatus.OK);
31     }
32 }
33
34
```

#6.

The screenshot shows an IDE with a test failure on the left and a Java code snippet on the right. The test failure is an `AssertionError` from `org.opentest4j.AssertionFailedError`. The expected value is a `Jeep` object with `modelId=WRANGLER`, `trimLevel=Sport`, `numDoors=2`, `wheelSize=17`, and `basePrice=28475.00`. The actual value is a `Jeep` object with `modelId=WRANGLER`, `trimLevel=Sport`, `numDoors=4`, `wheelSize=17`, and `basePrice=31975.00`. The code snippet on the right is a `protected List<Jeep> buildExpected()` method that returns a list of two `Jeep` objects. The first object has `modelId=WRANGLER`, `trimLevel=Sport`, `numDoors=2`, `wheelSize=17`, and `basePrice=28475.00`. The second object has `modelId=WRANGLER`, `trimLevel=Sport`, `numDoors=4`, `wheelSize=17`, and `basePrice=31975.00`.

```
private int serverPort;

protected List<Jeep> buildExpected() {
    List<Jeep> list = new LinkedList<>();
    list.add(Jeep.builder()
        .modelId(JeepModel.WRANGLER)
        .trimLevel("Sport")
        .numDoors(2)
        .wheelSize(17)
        .basePrice(new BigDecimal("28475.00"))
        .build());
    list.add(Jeep.builder()
        .modelId(JeepModel.WRANGLER)
        .trimLevel("Sport")
        .numDoors(4)
        .wheelSize(17)
        .basePrice(new BigDecimal("31975.00"))
        .build());
    return list;
}

@Test
void testThatJeepsAreReturnedWhenAValidModelAndTrimAreSupplied() {
    // Given a valid model, trim, and URI
    JeepModel model = JeepModel.WRANGLER;
    String trim = "Sport";
    String uri = String.format("http://localhost:%d/jeeps?model=%s&trim=%s", serverPort, model, trim);

    // When a connection is made to the URI
    ResponseEntity<List<Jeep>> response = restTemplate.exchange(uri,
        HttpMethod.GET, null, new ParameterizedTypeReference<>() {});

    // Then a success (200) is returned
    assertThat(response.getStatusCode(), isEqualTo(HttpStatus.OK));

    // And the actual list returned is the same as the expected list
    List<Jeep> expected = buildExpected();
    assertThat(response.getBody(), isEqualTo(expected));
}
```

#7.

The screenshot shows an IDE with a test failure on the left and a Java code snippet on the right. The test failure is an `AssertionError` from `org.opentest4j.AssertionFailedError`. The expected value is a `Jeep` object with `modelId=WRANGLER`, `trimLevel=Sport`, `numDoors=2`, `wheelSize=17`, and `basePrice=28475.00`. The actual value is a `Jeep` object with `modelId=WRANGLER`, `trimLevel=Sport`, `numDoors=4`, `wheelSize=17`, and `basePrice=31975.00`. The code snippet on the right is a `JeepSalesService` interface and its implementation `DefaultJeepSalesService`. The `DefaultJeepSalesService` class implements the `JeepSalesService` interface and has a `fetchJeeps` method that returns a list of `Jeep` objects. The `fetchJeeps` method is annotated with `@Override` and `Log.info`. The `fetchJeeps` method returns a list of two `Jeep` objects. The first object has `modelId=WRANGLER`, `trimLevel=Sport`, `numDoors=2`, `wheelSize=17`, and `basePrice=28475.00`. The second object has `modelId=WRANGLER`, `trimLevel=Sport`, `numDoors=4`, `wheelSize=17`, and `basePrice=31975.00`.

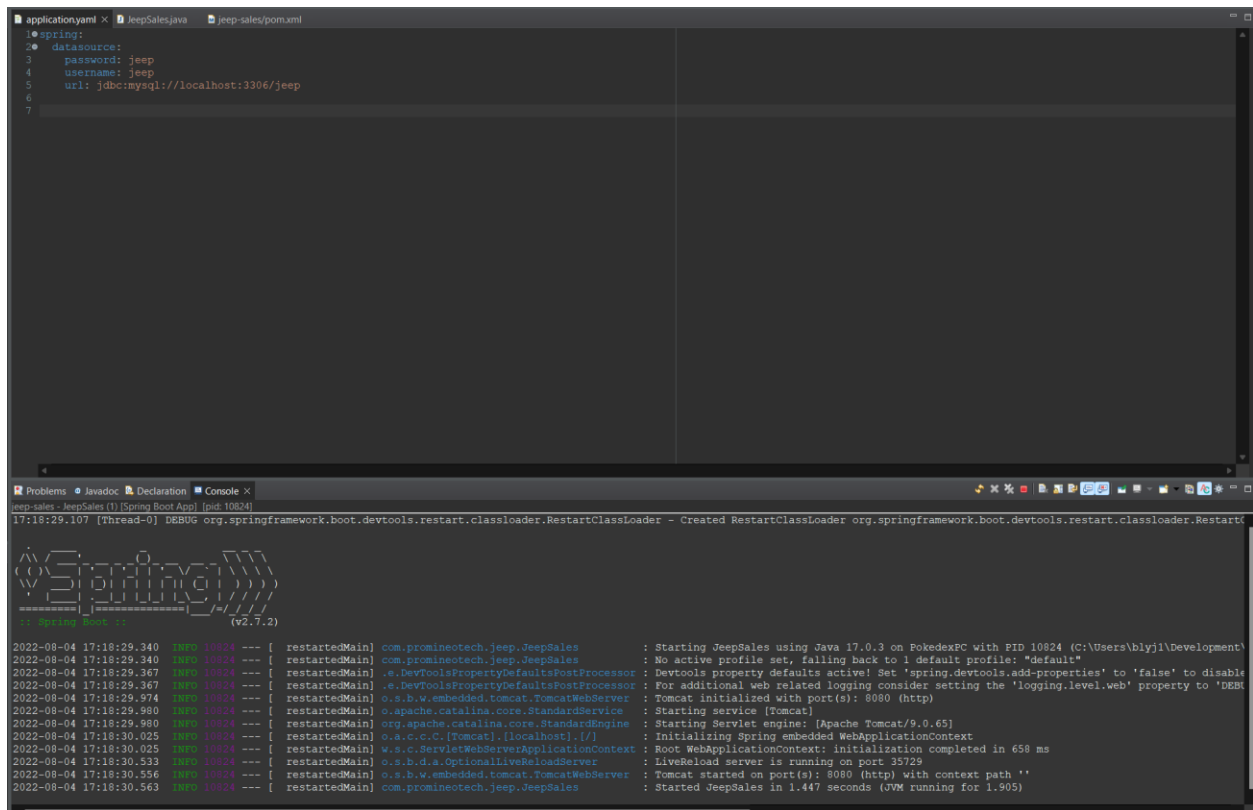
```
package com.promineotech.jeep.service;

import java.util.List;
import org.springframework.stereotype.Service;
import com.promineotech.jeep.entity.Jeep;
import com.promineotech.jeep.entity.JeepModel;
import lombok.extern.slf4j.Slf4j;

@Service
@Slf4j
public class DefaultJeepSalesService implements JeepSalesService {

    @Override
    public List<Jeep> fetchJeeps(JeepModel model, String trim) {
        log.info("The fetchJeeps method was called with model={} and trim={}", model, trim);
        return null;
    }
}
```

#10.



The screenshot shows an IDE with two tabs: 'application.yaml' and 'jeep-sales/pom.xml'. The 'application.yaml' tab is active, displaying the following configuration:

```
spring:
  datasource:
    password: jeep
    username: jeep
    url: jdbc:mysql://localhost:3306/jeep
```

Below the code editor, the 'Console' tab is open, showing a log of application startup. The log includes the Spring Boot logo and the following messages:

```
2022-08-04 17:18:29.340 INFO 10824 --- [ restartedMain] com.promineotech.jeepp.JeeppSales : Starting JeeppSales using Java 17.0.3 on PokedexPC with PID 10824 (C:\Users\blyj1\Development\
2022-08-04 17:18:29.340 INFO 10824 --- [ restartedMain] com.promineotech.jeepp.JeeppSales : No active profile set, falling back to 1 default profile: "default"
2022-08-04 17:18:29.367 INFO 10824 --- [ restartedMain] .e.DevToolsPropertyDefaultsPostProcessor : DevTools property defaults active! Set 'spring.devtools.add-properties' to 'false' to disable
2022-08-04 17:18:29.367 INFO 10824 --- [ restartedMain] .e.DevToolsPropertyDefaultsPostProcessor : For additional web related logging consider setting the 'logging.level.web' property to 'DEBUG'
2022-08-04 17:18:29.974 INFO 10824 --- [ restartedMain] o.s.b.v.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 (http)
2022-08-04 17:18:29.980 INFO 10824 --- [ restartedMain] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2022-08-04 17:18:29.980 INFO 10824 --- [ restartedMain] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.65]
2022-08-04 17:18:30.025 INFO 10824 --- [ restartedMain] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2022-08-04 17:18:30.025 INFO 10824 --- [ restartedMain] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 658 ms
2022-08-04 17:18:30.533 INFO 10824 --- [ restartedMain] o.s.b.d.a.OptionalLiveReloadServer : LiveReload server is running on port 35729
2022-08-04 17:18:30.556 INFO 10824 --- [ restartedMain] o.s.b.v.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path ''
2022-08-04 17:18:30.563 INFO 10824 --- [ restartedMain] com.promineotech.jeepp.JeeppSales : Started JeeppSales in 1.447 seconds (JVM running for 1.905)
```

URL to GitHub Repository:

<https://github.com/CoconutMacaron/Week-13-Spring-Assignment.git>

I realize I named this repository unfortunately so I'm just going to keep adding to this one for now to avoid anything weird happening to me.