

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

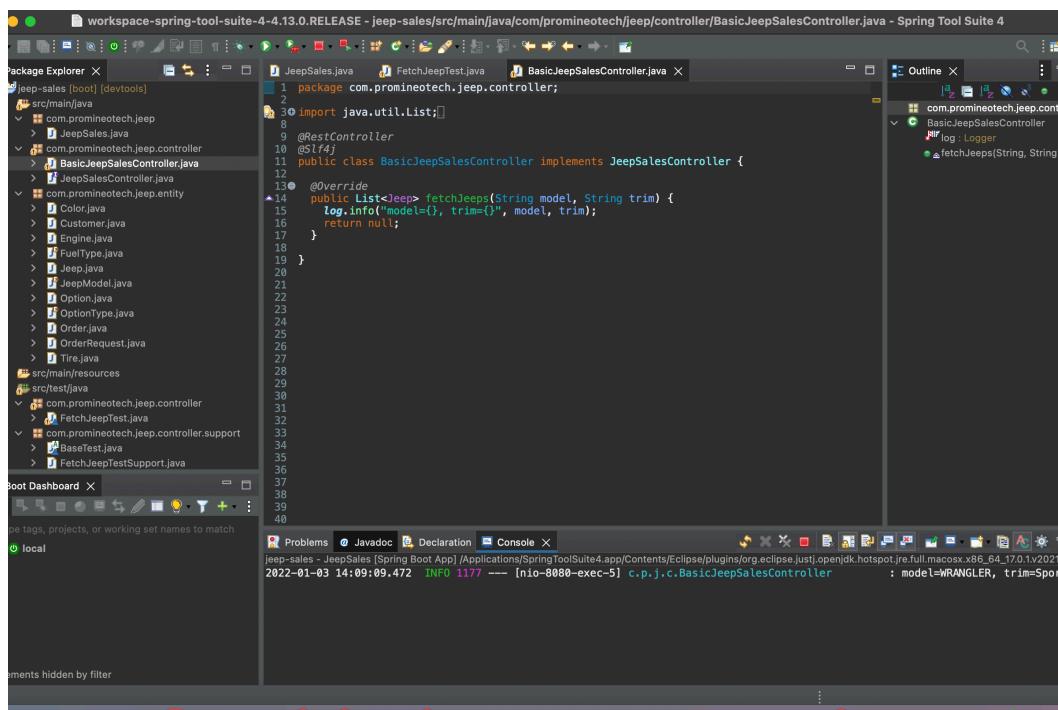
Instructions: In Eclipse, or an IDE of your choice, write the code that accomplishes the objectives listed below. Ensure that the code compiles and runs as directed. Take screenshots of the code and of the running program (make sure to get screenshots of all required functionality) and paste them in 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:

- 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.
- 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). 



Caption

- 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. 

localhost:8080 - Local server. ▾

jeep-sales-controller

`/jeeps` Returns a list of Jeeps

a list of Jeeps given an optional model and/or trim

ters

Description

The model name (i.e., 'WRANGLER')

CHEROKEE

The trim level (i.e., 'Sport')

Open Api 1

X 'GET' \p://localhost:8080/jeeps?model=CHEROKEE&trim=Sport' \ accept: application/json'

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

spone

Details

Response headers

```
connection: keep-alive
content-length: 0
date: Mon, 03 Jan 2022 20:54:57 GMT
keep-alive: timeout=60
```

is

Description

A list of Jeeps is returned

Media type

application/json

Controls Accept header.

Example Value | Schema

{}

Open Api 2

- 4) Run the integration test and show that the test status is green. Produce a screenshot of the test class and the status bar.

```

package com.promitheusjeep.controller;

import org.junit.Test;
import org.springframework.boot.test.autoconfigure.web.servlet.MockMvcMvcBuilder;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.http.MediaType;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.request.MockMvcRequestBuilders;
import org.springframework.test.web.servlet.result.MockMvcResultMatchers;
import static org.junit.Assert.*;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;

@SpringBootTest
public class FetchJeepTest {
    private MockMvc mockMvc;

    @Test
    public void testFetchJeep() throws Exception {
        mockMvc = MockMvcMvcBuilder.create()
                .addFilter(new MockMvcResultHandlerFilter())
                .build();

        mockMvc.perform(MockMvcRequestBuilders.get("/jeeps"))
                .andExpect(status().isOk());
    }
}

```

Test class and green 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...
- The test with the assertion.

- b) The JUnit status bar (should be red).
 - c) The method returning the expected list of Jeeps. 

The screenshot shows the Eclipse IDE interface with several open windows:

- Package Explorer**: Shows the project structure with `FetchJeepesTest` as the active element.
- FetchJeepesTest.java**: The Java code for the test class. It includes imports for `java.util.List`, `java.util.ArrayList`, `java.util.Arrays`, and `org.junit.Assert`. The class contains a single test method, `testThatJeepsAreReturnedWhenValidModelAnd`, which uses `given` to set up a response with status 200 and body "OK", then asserts that the response is OK and its body is "OK".
- Failure Trace**: Shows a stack trace for an `AssertionFailedError` at line 59 of `FetchJeepesTest.java`.
- Boot Dashboard**: A dashboard for Spring Boot applications.
- Problems**: A list of errors and warnings.
- JavaDoc**: Documentation for the current class.
- Declaration**: Declaration information for the current class.
- Console**: The output of the build process.

The console output shows the following log entries:

```
2022-01-03 16:06:31.477 INFO 2877 --- [main] c.j.e.controller.FetchJeepesTest : Starting FetchJeepesTest using
2022-01-03 16:06:31.479 INFO 2877 --- [main] c.j.e.controller.FetchJeepesTest : Running with Spring Boot v2.3.4.RELEASE
2022-01-03 16:06:31.479 INFO 2877 --- [main] c.j.e.controller.FetchJeepesTest : The following profiles are active: "local"
```

Expected list 1

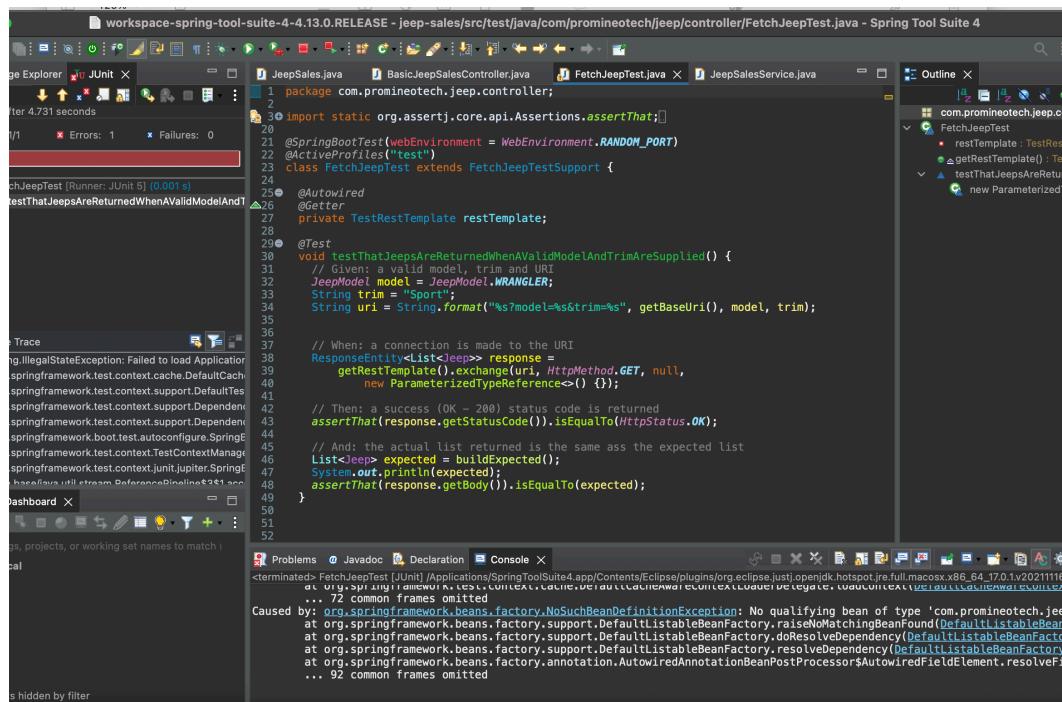
```
e X
ToolSuite4.app/Contents/Eclipse/plugins/org.eclipse.justj.open

1 on Boriss-MacBook-Pro.local with PID 2077 (v5.3.14
(JVM running for 6.79)
trimLevel=Sport, numDoors=4, wheelSize=17, ba
```

Expected list 2

7) Add a service layer in your application as shown in the videos:

- Add a package named `com.promineotech.jeep.service`.
- In the new package, create an interface named `JeepSalesService`.
- In the same package (`service`), create a class named `DefaultJeepSalesService` that implements the `JeepSalesService` interface. Add the class-level annotation, `@Service`.
- Inject the service interface into `DefaultJeepSalesController` using the `@Autowired` annotation. The instance variable should be `private`, and the variable should be named `jeepSalesService`.
- 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);`
- 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.
- Run the test again. Produce a screenshot showing the service class implementation, the log line in the console, and the red status bar. 



The screenshot shows the Spring Tool Suite 4 IDE interface. The central area displays Java code for a test class, `FetchJeepTest.java`. The code is annotated with `@SpringBootTest`, `@ActiveProfiles("test")`, and `@RunWith(SpringRunner.class)`. It contains a test method `void testThatJeepsAreReturnedWhenValidModelAndTrimAreSupplied()` that uses `RestTemplate` to make a GET request to a specific URL and assert the response against an expected list of Jeeps. The code uses Lombok annotations like `@Data` and `@Builder` on the `JeepModel` class. The right side of the interface shows the `Outline` view with the package structure `com.promineotech.jeep.controller` and the file `FetchJeepTest.java`. The bottom right corner of the IDE shows a red status bar with the text "terminated: FetchJeepTest [JUnit]". The bottom of the screen shows the Eclipse-based interface with tabs for Javadoc, Declaration, and Console, and a terminal window showing command-line output related to the test execution.

- 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.



```
Spring:
  datasource:
    password: jeep
    username: jeep
    url: jdbc:mysql://localhost:3306/jeep
  logging:
    level:
      root: warn
      '[com.promineotech]': debug
```

```
-01-04 13:13:14.486 INFO 6348 --- [ restartedMain] com.promineotech.jeep.JeepSales      : Starting JeepSales using Java 17.0.
-01-04 13:13:14.486 DEBUG 6348 --- [ restartedMain] com.promineotech.jeep.JeepSales      : Running with Spring Boot v2.6.2, Sp
-01-04 13:13:14.486 DEBUG 6348 --- [ restartedMain] o.s.boot.SpringApplication : No active profile set, falling back to default [SPRING_PROFILE]
-01-04 13:13:17.982 WARN 6348 --- [ restartedMain] o.s.data.convert.CustomConversions  : Registering converter from class ja
-01-04 13:13:18.120 INFO 6348 --- [ restartedMain] com.promineotech.jeep.JeepSales      : Started JeepSales in 4.117 seconds
-01-04 13:14:35.868 DEBUG 6348 --- [nio-8080-exec-3] c.p.j.c.BasicJeepSalesController     : model=WRANGLER, trim=Sport
```

- 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:

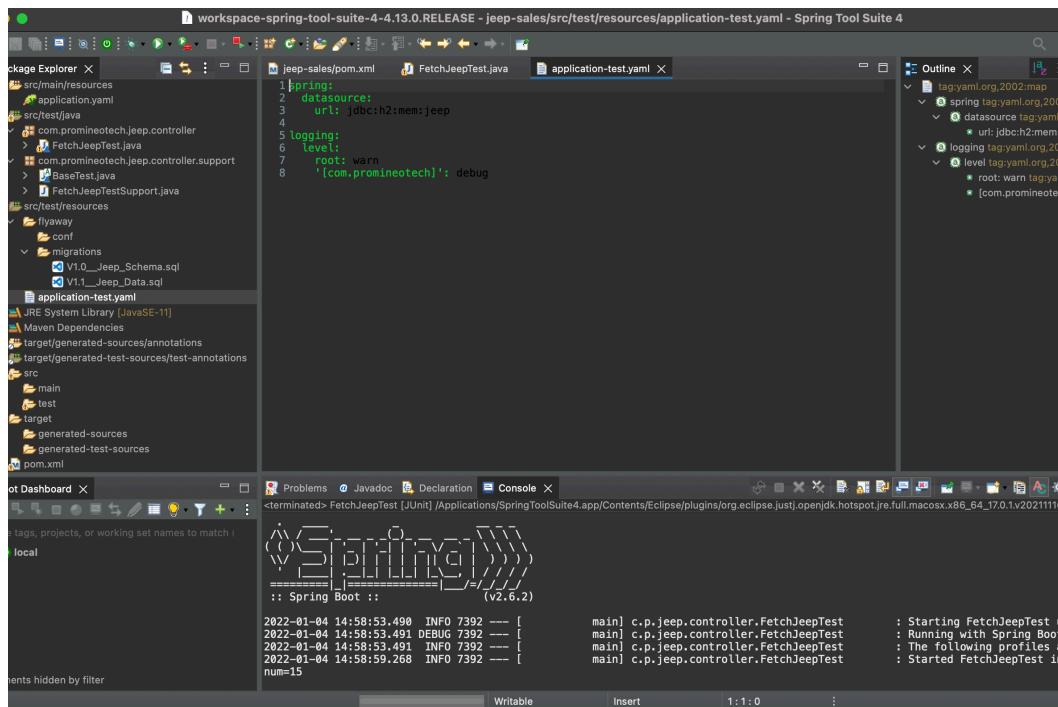
```
spring:
```

```
  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`.



Caption

URL to GitHub Repository:

<git@github.com:bgoetz22/SpringBootWeek2.git>