Points possible: 70

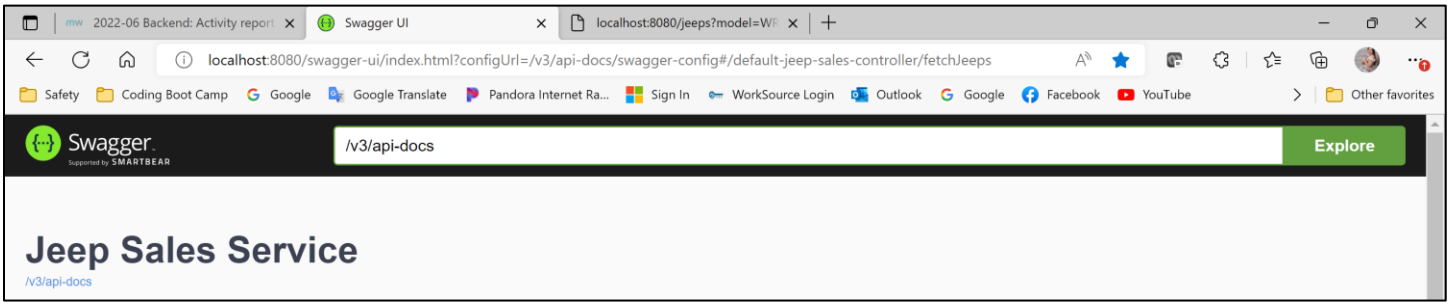
Category	Criteria	% of Grade
Functionality	Does the code work?	25
Organization	Is the code clean and organized? Proper use of white space, syntax, and consistency are utilized. Names and comments are concise and clear.	25
Creativity	Student solved the problems presented in the assignment using creativity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

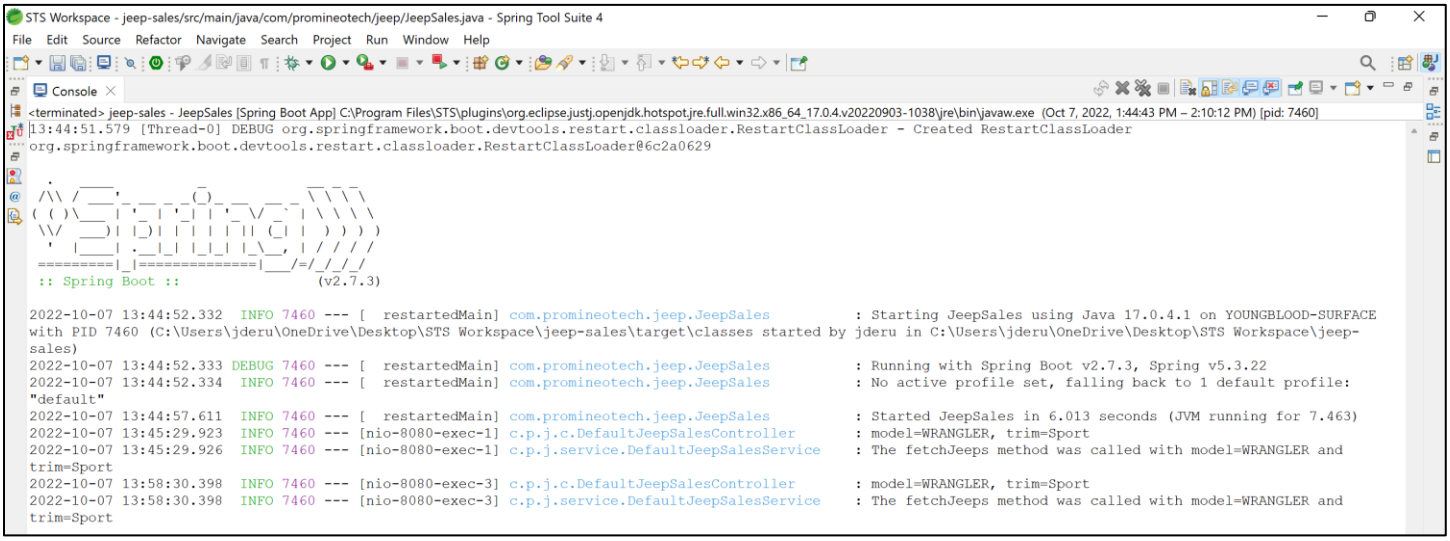
Instructions: *In Eclipse, or an IDE of your choice, write the code that accomplishes the objectives listed below. Ensure that the code compiles and runs as directed. Take screenshots of the code and of the running program (make sure to get screenshots of all required functionality) and paste them in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

Coding Steps:

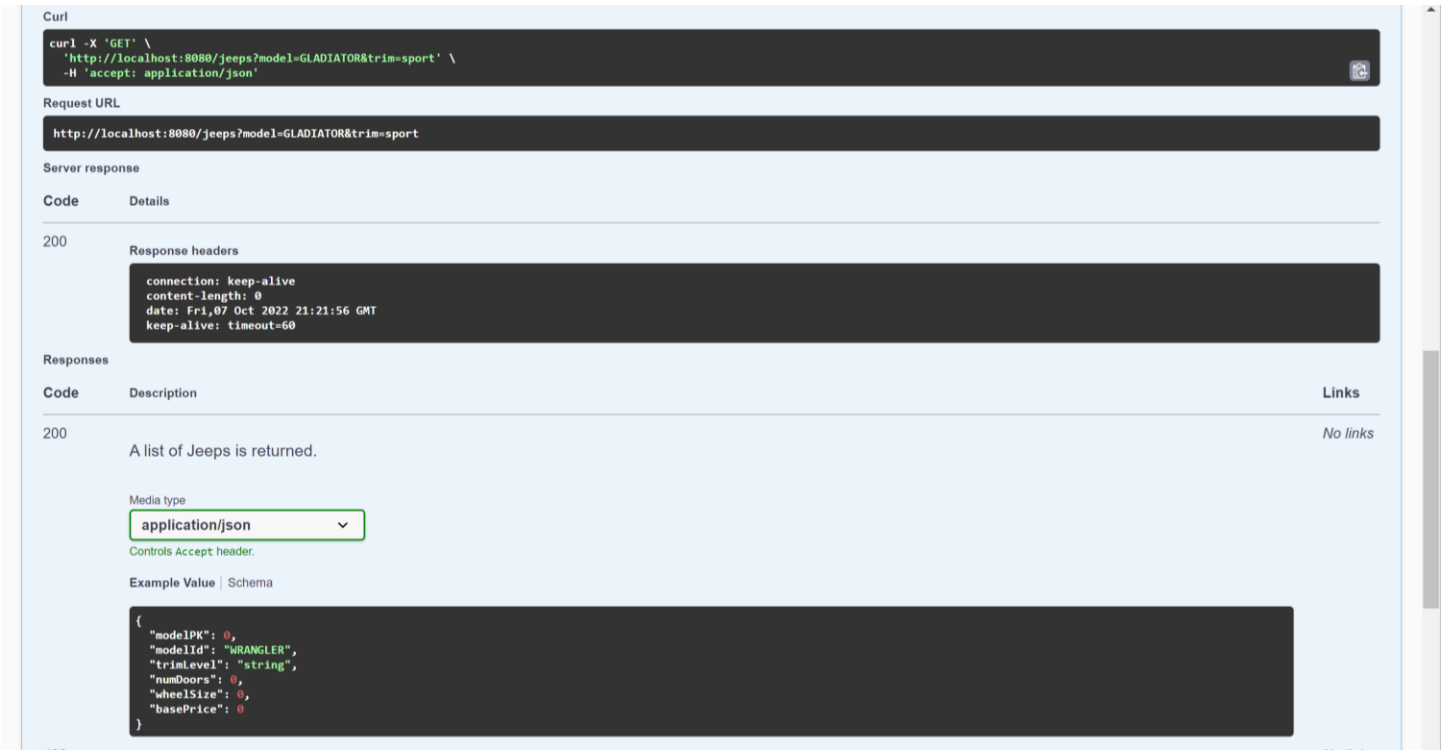
✓1) In the project you started last week, use Lombok to add an info-level logging statement in the controller implementation method that logs the parameters that were input to the method. Remember to add `@Slf4j` annotation to the class.

✓2) Start the application (not an integration test). Use a browser to navigate to the application passing the parameters required for your selected operation. 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. Produce a screenshot showing the `curl` command, the request URL, and the response headers.



✓4) Run the integration test and show test status is green. Post a screenshot of the test class and status bar.

(I am too far past this point where my Junit test no longer runs green, it runs red as it is supposed to by the end of this week's video instructions. See below for red test)

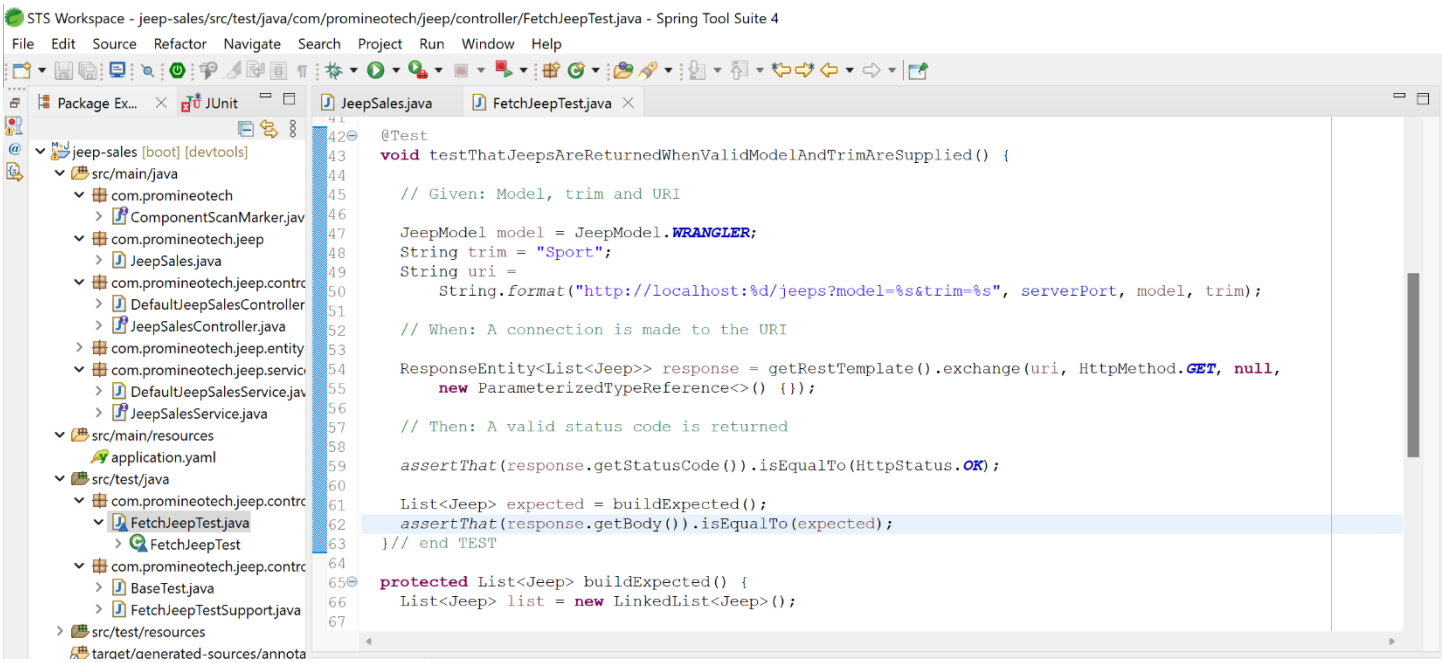
✓5) Add a method to the test to return a list of expected `Jeep` (`model`) objects based on the model and trim level you selected. So, for example, using the model Wrangler and trim level "Sport", the query should return two rows:

	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…

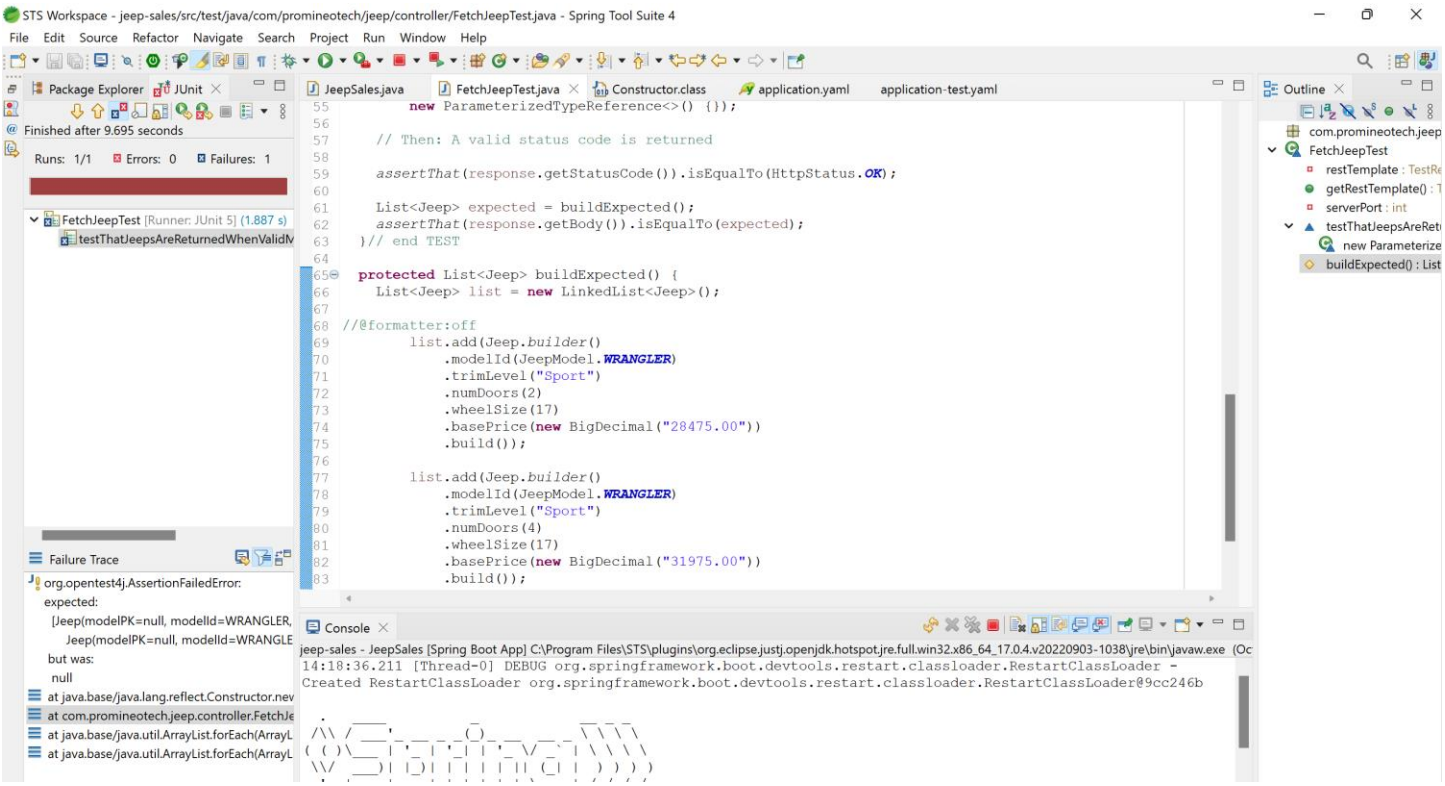
a) The test with the assertion.
b) The JUnit status bar (should be red).
c) The method returning the expected list of Jeeps.



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).
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.

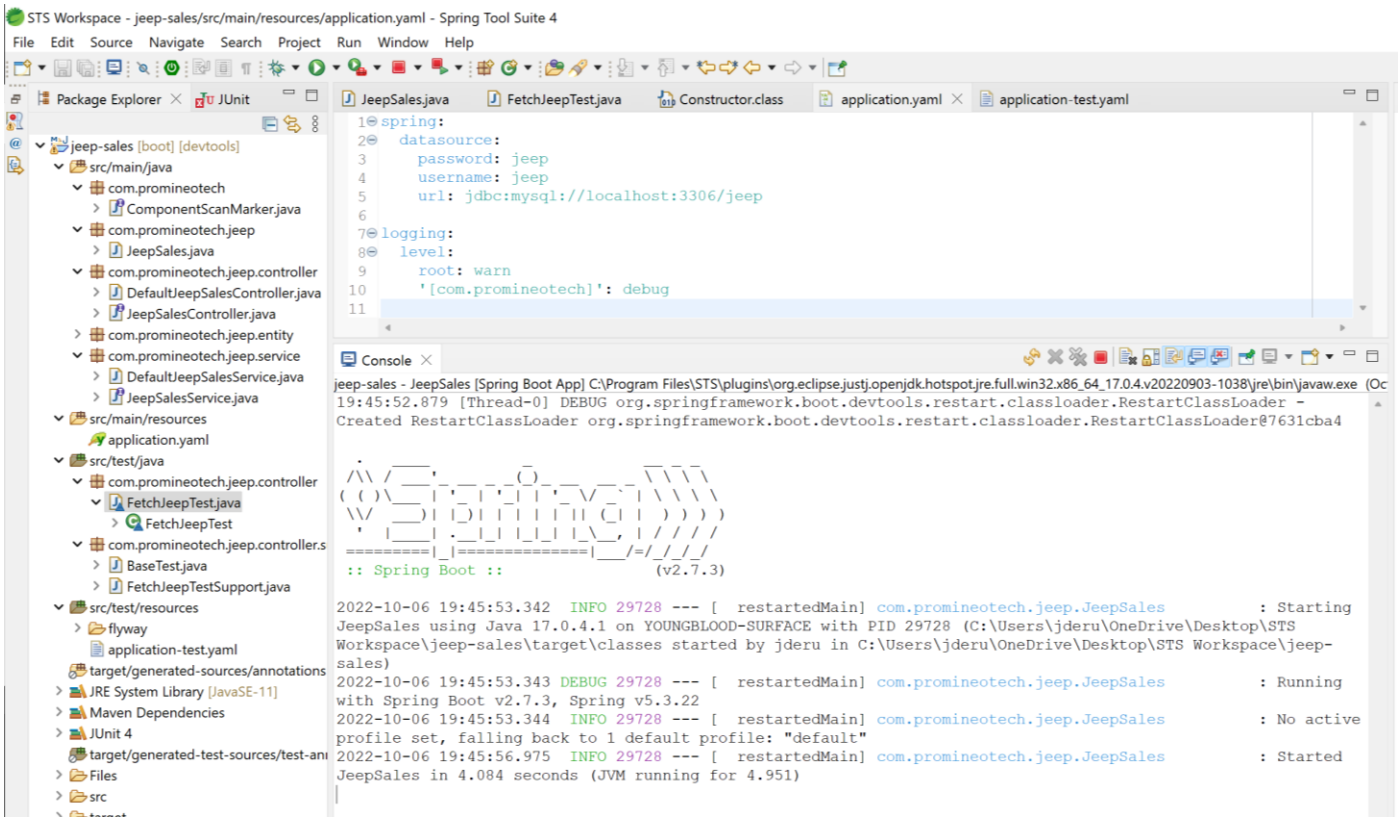
✓ g) Run the test again. Produce a screenshot showing the service class implementation, the log line in the console, and the red status bar. 💻



✓ 8) Add the database dependencies described in the video to the POM file (MySQL driver and Spring Boot Starter JDBC). To find them, nagivate 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.

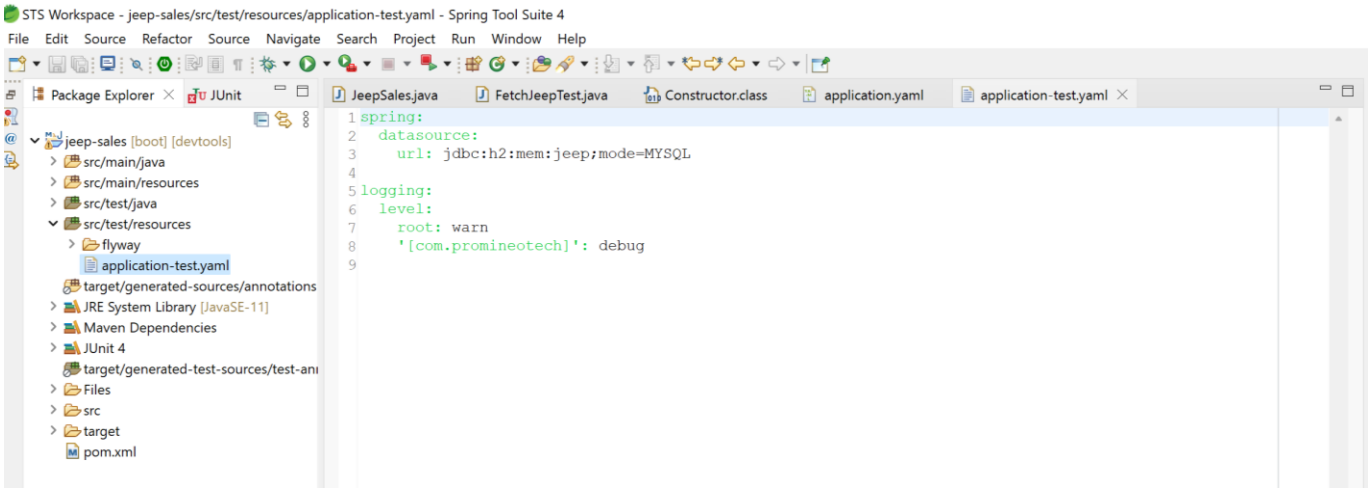
✓ 9) Create `application.yaml` in `src/main/resources`. Add the `spring.datasource.url`, `spring.datasource.username`, and `spring.datasource.password` properties to `application.yaml`. The URL should be the same as shown in the video (`jdbc:mysql://localhost:3306/jeep`). The password and username should match your setup.

✓ 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. Produce a screenshot showing `application-test.yaml`.



URL to GitHub Repository:

https://github.com/JaxYoungblood/Week14-SpringBootCodingAssignment-.git