

Web API Design with Spring Boot Week 4 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:

For this week's homework you need to copy source code from the supplied resources.

For this week's homework you need to copy source code from the Source folder in the supplied resources. Wait until the instructions tell you to copy the resources or you will get errors.

- 1) Select some options for a Jeep order:
 - a) Use the `data.sql` file or the jeep database tables to select options for a Jeep order. Select any one of each of the following for the order:
 - i) color
 - ii) customer
 - iii) engine
 - iv) model
 - v) tire(s)
 - b) Select one or more options from the options table as well. Keep in mind that some options may work better than others – but if you want to put 37-inch tires on your Jeep Renegade, so be it!
- 2) Create a new integration test class to test a Jeep order named `CreateOrderTest.java`. Create this class in `src/test/java` in the `com.promineotech.jeep.controller` package.
 - a) Add the Spring Boot Test annotations: `@SpringBootTest`, `@ActiveProfiles`, and `@Sql`. They should have the same parameters as the test created in weeks 1 and 2.
 - b) Create a test method (annotated with `@Test`) named `testCreateOrderReturnsSuccess201`.
 - c) In the test class, create a method named `createOrderBody`. This method returns a type of `String`. In this method, return a JSON object with the IDs that you picked in Step 1a and b. For example:

```
{  
    "customer": "MORISON_LINA",  
    "model": "WRANGLER",  
    "trim": "Sport Altitude",  
    "doors": 4,  
    "color": "EXT_NACHO",  
    "engine": "2_0_TURBO",  
    "tire": "35_TOYO",  
    "options": [
```

```

    "DOOR_QUAD_4",
    "EXT_AEV_LIFT",
    "EXT_WARN_WINCH",
    "EXT_WARN_BUMPER_FRONT",
    "EXT_WARN_BUMPER_REAR",
    "EXT_ARB_COMPRESSOR"
]

}

```

Make sure that the JSON is correct! If necessary, use a JSON formatter/validator like the one here: <https://jsonformatter.curiousconcept.com/>.

Produce a screenshot of the `createOrderBody()` method. 

In the test method, assign the return value of the `createOrderBody()` method to a variable named `body`.

- d) In the test class, add an instance variable named `serverPort` to hold the port that Tomcat is listening on in the test. Annotate the variable with `@LocalServerPort`.
- e) Add another instance variable for an injected `TestRestTemplate` named `restTemplate`.
- f) In the test method, assign a value to a local variable named `uri` as follows:

```
String uri = String.format("http://localhost:%d/orders", serverPort);
```

- g) In the test method, create an `HttpHeaders` object and set the content type to "application/json" like this:

```
HttpHeaders headers = new HttpHeaders();
headers.setContentType(MediaType.APPLICATION_JSON);
```

Make sure to import the package `org.springframework.http.HttpHeaders`.

- h) Create an `HttpEntity` object and set the request body and headers:

```
HttpEntity<String> bodyEntity = new HttpEntity<>(body, headers);
```

- i) Send the request body and headers to the server. The `Order` class should have been copied earlier from the supplied resources. Ensure that you import `com.promineotech.jeep.entity.Order` and not some other `Order` class.

```
ResponseEntity<Order> response = restTemplate.exchange(uri,
    HttpMethod.POST, bodyEntity, Order.class);
```

- j) Add the AssertJ assertions to ensure that the response is correct. Replace the expected values to match the JSON in step 2c.

```
assertThat(response.getStatusCode()).isEqualTo(HttpStatus.CREATED);
assertThat(response.getBody()).isNotNull();

Order order = response.getBody();
assertThat(order.getCustomer().getCustomerId()).isEqualTo("MORISON_LINA");
assertThat(order.getModel().getModelId()).isEqualTo(JeepModel.WRANGLER);
assertThat(order.getModel().getTrimLevel()).isEqualTo("Sport Altitude");
assertThat(order.getModel().getNumDoors()).isEqualTo(4);
assertThat(order.getColor().getColorId()).isEqualTo("EXT_NACHO");
assertThat(order.getEngine().getEngineId()).isEqualTo("2_0_TURBO");
assertThat(order.getTire().getTireId()).isEqualTo("35_TOYO");
assertThat(order.getOptions()).hasSize(6);
```

- k) Produce a screenshot of the test method. 
- 3) In the controller sub-package in `src/main/java`, create an interface named `JeepOrderController`. Add `@RequestMapping("/orders")` as a class-level annotation.
- Create a method in the interface to create an order (`createOrder`). It should return an object of type `Order` (see below). It should accept a single parameter of type `OrderRequest` as described in the video. Make sure it accepts an HTTP POST request and returns a status code of 201 (created).
 - Add the `@RequestBody` annotation to the `orderRequest` parameter. Make sure to add the `RequestBody` annotation from the `org.springframework.web.bind.annotation` package.
 - Produce a screenshot of the finished `JeepOrderController` interface showing no compile errors. 
- 4) Create a class that implements `JeepOrderController` named `DefaultJeepOrderController`.
- Add `@RestController` as a class-level annotation.

- b) Add a log line to the implementing controller method showing the input request body (orderRequest)
 - c) Run the test to show a red status bar. Produce a screenshot that shows the test method, the log line, and the red JUnit status bar. 
- 5) Find the Maven dependency `spring-boot-starter-validation` by looking it up at <https://mvnrepository.com/>. Add this repository to the project POM file (pom.xml).
- 6) Add the class-level annotation `@Validated` to the `JeepOrderController` interface.
- 7) Add Bean Validation annotations to the `OrderRequest` class as shown in the video.
- a) Use these annotations for String types:
 - i) `@NotNull`
 - ii) `@Length(max = 30)`
 - iii) `@Pattern(regexp = "[\\w\\s]*")`
 - b) Use these annotations for integer types:
 - i) `@Positive`
 - ii) `@Min(2)`
 - iii) `@Max(4)`
 - c) Add `@NotNull` to the enum type.
 - d) Add validation to the list element (type String) by adding the validation annotations *inside* the generic definition. So, to add the String validation to the options, you would do this:
- ```
private List<@NotNull @Length(max = 30) @Pattern(regexp = "[\\w\\s]*") String> options;
```
- Do not apply a `@NotNull` annotation to the `List` because if you have no options the `List` may be null. 
- e) Produce a screenshot of this class with the annotations. 
- 8) In the `jeep.service` sub-package, create the empty (no methods yet) order service interface (named `JeepOrderService`) and implementation (named `DefaultJeepOrderService`).
- a) Inject the interface into the order controller implementation class.
  - b) Add the `@Service` annotation to the service implementation class.
  - c) Create the `createOrder` method in the interface and implementing service. The method signature should look like this:

```
Order createOrder(OrderRequest orderRequest);
```

- d) Call the `createOrder` method from the controller and return the value returned by the service.
  - e) Add a log line in the `createOrder` method and log the `orderRequest` parameter.
  - f) Run the test `CreateOrderTest` again. Produce a screenshot showing that the `createOrder` method in the service was called in the service class. 
- 9) In the `jeep.dao` sub-package, create the empty (no methods yet) DAO interface (named `JeepOrderDao`) and implementation (named `DefaultJeepOrderDao`).
- a) Inject the DAO interface into the order service implementation class.
  - b) Add the `@Component` annotation to the DAO implementation class.
- 10) Replace the entire content of `JeepOrderDao.java` with the source found in `JeepOrderDao.source`. The source file is found in the Source folder of the supplied project resources.
- 11) \*\*\* The next steps require you to copy source code from the Source directory in the supplied resources. Please follow the instructions EXACTLY. Some steps require you to replace ALL the source in a file. Some steps require you to ADD source to a file.**
- 12) Replace the entire contents of `DefaultJeepOrderDao.java` with the source found in `DefaultJeepOrderDao.source`. The source file is found in the Source folder of the supplied project resources. After this step you will see errors in `DefaultJeepOrderDao`. This will be fixed shortly.
- 13) Copy the *contents* of the file `DefaultJeepOrderDao.source` *into* `DefaultJeepOrderDao.java`. The source file is found in the Source folder of the supplied project resources.

In Eclipse, click the "Source" menu and select "Organize Imports". Pick packages from your project where applicable. Make sure you pick the import `java.util.Optional`, `java.util.List`, and `org.springframework.jdbc.core.RowMapper`.

- 14) Copy the *contents* of the file `DefaultJeepOrderService.source` *into* `DefaultJeepOrderService.java`. Add the source after the `createOrder()` method, but *inside* the class body. The source file is found in the Source folder of the supplied project resources.

In Eclipse, click the "Source" menu and select "Organize Imports". Pick packages from your project where applicable.

15) In `DefaultJeepOrderService.java`, work with the method `createOrder`.

- a) Add the `@Transactional` annotation to the `createOrder` method.
- b) In the `createOrder` method call the copied methods: `getCustomer`, `getModel`, `getColor`, `getEngine`, `getTire` and `getOption`, assigning the return values of these methods to variables of the appropriate types.
- c) Calculate the price, including all options.

16) In `JeepOrderDao.java` and `DefaultJeepOrderDao.java`, add the method:

```
Order saveOrder(Customer customer, Jeep jeep, Color color, Engine
engine, Tire tire, BigDecimal price, List<Option> options);
```

- a) Call the method from the order service. Produce a screenshot of the service method. 
- b) Write the implementation of the `saveorder` method in the DAO.
  - i) Call the supplied `generateInsertSql` method, passing in the customer, jeep, color, engine, tire and price. Assign the return value of the method to a `sqlParams` object.
  - ii) Call the `update` method on the `NamedParameterJdbcTemplate` object, passing in a `KeyHolder` object as shown in the video. Create the `KeyHolder` like this:

```
KeyHolder keyHolder = new GeneratedKeyHolder();
```

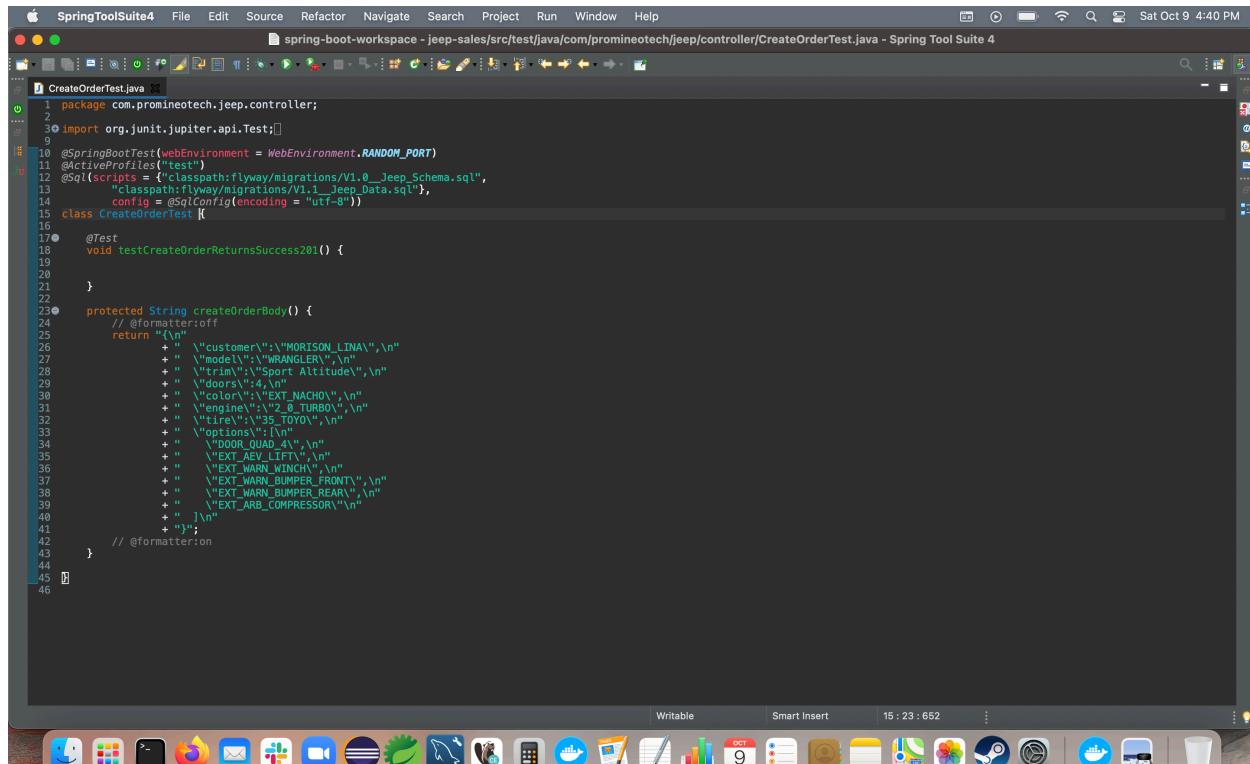
Be sure to extract the order primary key from the `KeyHolder` object into a variable of type `Long` named `orderPK`.
- iii) Write a method named `saveOptions` as shown in the video. This method should have the following method signature:

```
private void saveOptions(List<Option> options, Long orderPK)
```

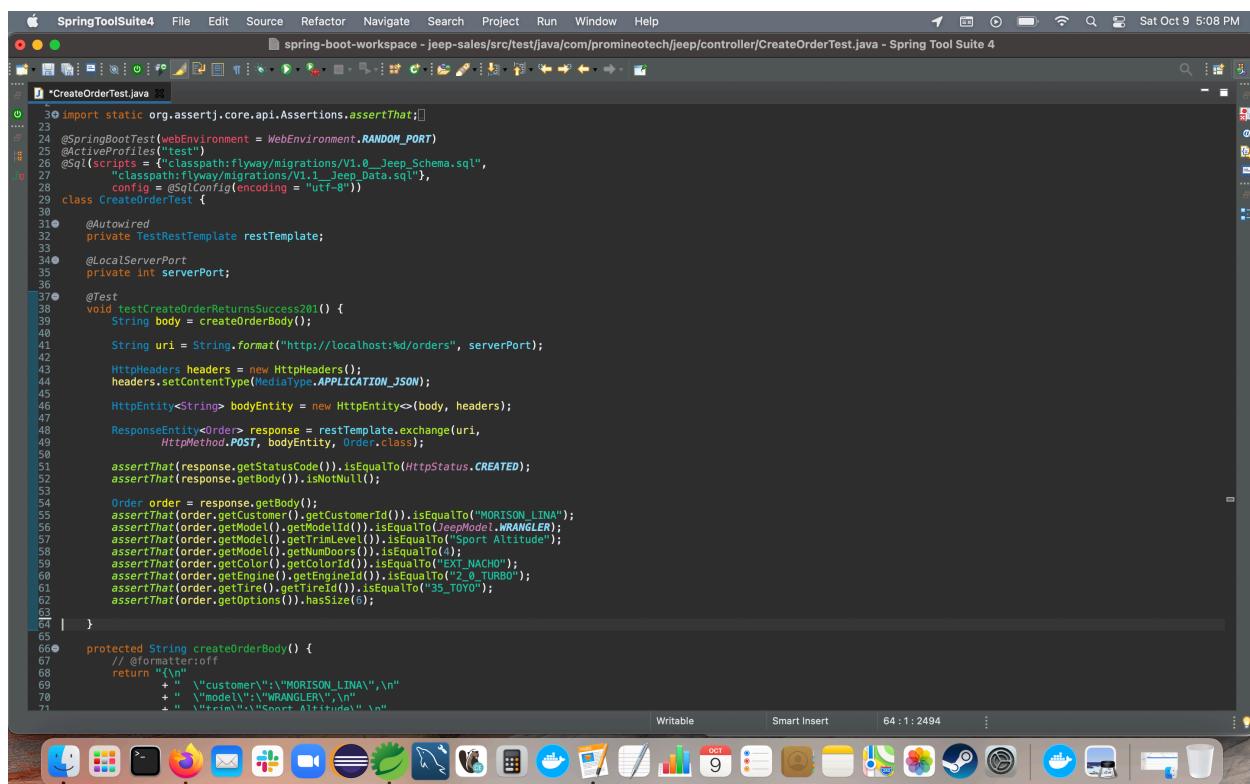
For each option in the `options` list, call the supplied `generateInsertSql` method passing the parameters `option` and order primary key (`orderPK`). Call the `update` method on the `NamedParameterJdbcTemplate` object.
- iv) In the `saveOrder` method in the DAO implementation, return an `Order` object using the `Order.builder`. The `Order` should include `orderPK`, `customer`, `jeep` (model), `color`, `engine`, `tire`, `options` and `price`.

- v) Produce a screenshot of the `saveOrder` method. 
- c) Run the integration test in `CreateOrderTest`. Produce a screenshot of the test method that shows the green JUnit status bar, the console output, and the test class. 

## Screenshots of Code and Screenshots of Running Application:



A screenshot of the Spring Tool Suite 4 interface. The main window displays the Java code for `CreateOrderTest.java`. The code is annotated with JUnit and Spring Boot annotations, including `@Test`, `@ActiveProfiles`, and `@Sql`. It defines a test method `testCreateOrderReturnsSuccess201()` and a protected helper method `createOrderBody()` that returns a JSON string representing a car order. The JSON string includes fields like customer ID, model, trim level, doors, color, engine, tire, options, and various sensor configurations. The code editor shows syntax highlighting and code completion. The bottom status bar indicates the date as Sat Oct 9 4:40 PM.



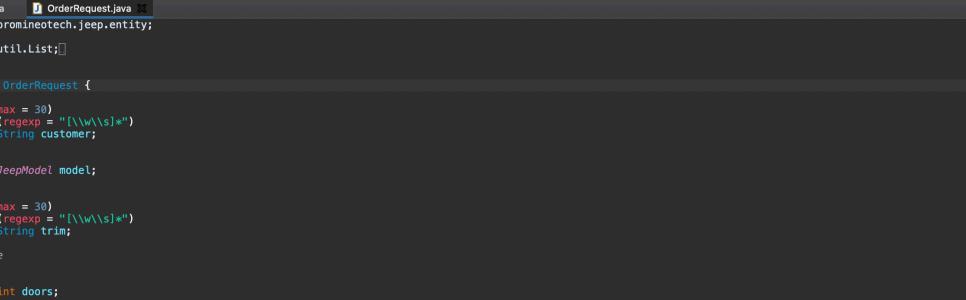
A screenshot of the Spring Tool Suite 4 interface, showing the same `CreateOrderTest.java` file as the previous screenshot, but with more annotations and assertions added. The code now includes `@Autowired` for the `restTemplate` field, a `@LocalServerPort` annotation for the `serverPort` field, and a `@Test` annotation for the `testCreateOrderReturnsSuccess201()` method. The `createOrderBody()` method is annotated with `@formatter:off` and contains a large JSON string. The code editor shows syntax highlighting and code completion. The bottom status bar indicates the date as Sat Oct 9 5:08 PM.

The screenshot shows the Spring Tool Suite 4 interface with the code editor open. The file is `JeepOrderController.java`. The code defines a REST controller for managing orders. It includes annotations for `@RequestMapping`, `@OpenAPIDefinition`, and `@ApiResponse` to define the API endpoints and their responses. The controller has a method `createOrder` that returns a `Created` status with a `Content-Type` of `application/json`.

```
1 package com.promineotech.jeep.controller;
2
3 import org.springframework.http.HttpStatus;
4
5 import org.springframework.web.bind.annotation.*;
6
7 import com.promineotech.jeep.model.Order;
8 import com.promineotech.jeep.repository.OrderRepository;
9
10 import java.util.List;
11
12 @RequestMapping("/orders")
13 @OpenAPIDefinition(info = @Info(title = "Jeep Order Service"), servers = {
14 @Server(url = "http://localhost:8080", description = "Local server.")})
15 public interface JeepOrderController {
16
17 // @formatter:off
18 @PostMapping
19 @Operation(summary = "Create an order for a Jeep",
20 description = "Returns the created Jeep",
21 responses = {
22 @ApiResponse(
23 responseCode = "201",
24 description = "The created Jeep is returned",
25 content = @Content(
26 mediaType = "application/json",
27 schema = @Schema(implementation = Order.class))),
28 @ApiResponse(
29 responseCode = "400",
30 description = "The request parameters are invalid",
31 content = @Content(
32 mediaType = "application/json")),
33 @ApiResponse(
34 responseCode = "404",
35 description = "A Jeep component was not found with the input criteria",
36 content = @Content(
37 mediaType = "application/json")),
38 @ApiResponse(
39 responseCode = "500",
40 description = "An unhandled error occurred",
41 content = @Content(
42 mediaType = "application/json"))
43 },
44 parameters = {
45 @Parameter(name = "orderRequest",
46 required = true,
47 description = "The order as JSON")
48 }
49)
50 @ResponseStatus(code = HttpStatus.CREATED)
51 Order createOrder(@RequestBody OrderRequest orderRequest);
52 // @formatter:on
53 }
```

The screenshot shows the Spring Tool Suite 4 interface with the test editor open. The file is `CreateOrderTest.java`. The test case `testCreateOrderReturnsSuccess201` uses the `RestTemplate` to send a `POST` request to the `/orders` endpoint. It then asserts that the response status is `CREATED` and that the returned `Order` object has the expected properties.

```
1 package com.promineotech.jeep;
2
3 import com.fasterxml.jackson.databind.ObjectMapper;
4 import com.promineotech.jeep.model.Order;
5 import com.promineotech.jeep.repository.OrderRepository;
6 import org.junit.jupiter.api.Test;
7 import org.springframework.beans.factory.annotation.Autowired;
8 import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
9 import org.springframework.boot.test.mock.mockito.MockBean;
10 import org.springframework.http.MediaType;
11 import org.springframework.test.web.servlet.MockMvc;
12
13 import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.post;
14 import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;
15
16 @WebMvcTest
17 @MockBean(OrderRepository.class)
18 @Autowired(ObjectMapper.class)
19 public class CreateOrderTest {
20
21 @Test
22 void testCreateOrderReturnsSuccess201() {
23 String body = createOrderBody();
24
25 HttpHeaders headers = new HttpHeaders();
26 headers.setContentType(MediaType.APPLICATION_JSON);
27
28 HttpEntity<String> bodyEntity = new HttpEntity<String>(body, headers);
29
30 ResponseEntity<Order> response = restTemplate.exchange(uri,
31 HttpMethod.POST, bodyEntity, Order.class);
32
33 assertEquals(HttpStatus.CREATED, response.getStatusCode());
34 assertEquals("201", response.getHeaders().get("Content-Type"));
35
36 Order order = response.getBody();
37 assertEquals("MORISON_LINA", order.getCustomer().getCustomerId());
38 assertEquals("WRANGLER", order.getModel().getModelName());
39 assertEquals("EXT_NACHO", order.getColor().getColorId());
40 assertEquals(4, order.getNumDoors());
41 assertEquals("Sport Altitude", order.getTrimLevel());
42 assertEquals("EXT_NACHO", order.getEngine().getEngineId());
43 assertEquals("2_0_TURBO", order.getTire().getTireId());
44 assertEquals("35_TOYO", order.getOptions());
45
46 assertEquals(6, order.getOptions().size());
47 }
48
49 protected String createOrderBody() {
50 // @formatter:off
51 return """
52 {
53 "customer": {
54 "customerId": "MORISON_LINA"
55 },
56 "model": {
57 "modelId": "WRANGLER"
58 },
59 "color": {
60 "colorId": "EXT_NACHO"
61 },
62 "options": [
63 "Sport Altitude"
64],
65 "tire": {
66 "tireId": "35_TOYO"
67 }
68 }
69 """;
70 }
71 }
```



The screenshot shows the Spring Tool Suite 4 interface with the following details:

- Title Bar:** SpringToolSuite4 - File Edit Source Refactor Navigate Search Project Run Window Help
- Toolbar:** Includes icons for New, Open, Save, Cut, Copy, Paste, Find, Replace, Undo, Redo, and various project and file management tools.
- Code Editor:** The main window displays the `OrderRequest.java` file under the package `com.promineotech.jeep.entity`. The code defines a POJO with annotations for data validation (e.g., @Data, @NotNull, @Length, @Pattern).
- Status Bar:** Shows "Writable", "Smart Insert", "16 : 28 : 388", and a battery icon.
- Mac OS Dock:** At the bottom, it shows the Dock with icons for Finder, Home, Mail, Safari, and other applications.

The screenshot shows a Java IDE interface with several windows:

- Code Editor:** Displays the `OrderControllerTest.java` file, which contains a test method `testCreateOrderThatIsSuccessful()`. The code uses the `MockMvc` framework to send a POST request to the `/order` endpoint and verify the response status and content.
- Terminal:** Shows the command `mvn spring-boot:run` being run in a terminal window, starting a Spring Boot application named `OrderControllerApplication`.
- Output:** Displays the logs from the running Spring Boot application. It includes initialization messages like "Starting OrderControllerApplication using Java 16.0.1 in Detached Mode", configuration details for port 8080, and various log entries such as "2023-03-20T14:45:45.772Z [main] INFO o.s.b.a.e.web.embedded.netty.NettyWebServer - Netty Web Server initialized" and "2023-03-20T14:45:45.772Z [main] INFO o.s.b.a.e.web.embedded.netty.NettyWebServer - Started OrderControllerApplication in 0.001 seconds (JVM: 0.001s)".

This screenshot shows a Java code editor with a dark theme. The main window displays a class named `CustomerService` which implements the `CustomerService` interface. The class contains numerous methods, many of which are annotated with `@Override`. The code includes several imports at the top, such as `java.util.List`, `java.util.Optional`, and various domain model classes like `Customer`, `Order`, and `OrderLine`. The implementation logic involves creating new objects, updating existing ones, and performing database queries using `Optional` to handle null values.

This screenshot shows a continuation of the `CustomerService` implementation. It includes methods for saving and deleting customers, as well as for finding specific customers by ID or name. The code uses `Optional` for nullable fields and `Map` for mapping customer IDs to names. There are also sections for handling `Order` and `OrderLine` entities, including their creation, update, and deletion.

This screenshot shows a test class named `CreateCustomerTest` located in the `com.johnathanschoenbeck.springbootbestworkshop` package. The class extends `AbstractIntegrationTest` and contains several test methods. One method, `testCreateCustomer()`, demonstrates how to create a new customer using a `RestTemplate` to send a POST request to the `/customers` endpoint. The response is then checked for its status and content. Other methods in the class likely cover the other CRUD operations and validation logic.

**URL to GitHub Repository:**

<https://github.com/johnbarts/springbootweek4>