Testing Spring Boot Applications

Overview

In this lab you'll write comprehensive tests for your "online retailer" application.

IntelliJ starter project

If you're happy to continue where you left off in the previous lab, use the following project:

- **`student\student-online-retailer`**

If you'd prefer a fresh start, use the solution project from the previous lab instead:

- **`solutions\solution-full-rest-services`**

IntelliJ solution project

The solution project for this lab is located here:

- **`solutions\solution-testing`**

Roadmap

There are 4 exercises in this lab. Each exercise is independent of the others, so you can tackle the exercises in any order. The *solution* project contains full solutions for everything. Here is a brief summary of the tasks you will perform in each exercise; more detailed instructions follow later:

1. Simple unit testing (**`CartRepositoryImpl`**)
2. Mocking beans (**`CartServiceImpl`**)
3. Testing a CrudRepository (**`ProductSuggestionCrudRepository`**)
4. Testing a REST controller (**`ProductSuggestionController`**)

Exercise 1: Simple unit testing (`CartRepositoryImpl`)

Take a look at **`CartRepositoryImpl`** in the source code folder. This class holds a **`HashMap`** of items in the user's shopping cart, in memory.

In the test folder, add a class named **`CartRepositoryImplTests`** to test the class. Here are some hints and suggestions:

- You might be tempted to autowire a **`CartRepositoryImpl`** bean into your test class, but this isn't a good idea because **`CartRepositoryImpl`** is stateful (i.e., it stores data in memory). If you use autowiring, Spring Boot will autowire the same bean for all your test methods, which means the test methods will cumulatively change the state of the bean. Thus, test method #2 would need to know what happened in test method #1, and so on. Tests are meant to be independent of each other, so this approach isn't appropriate.
- A better approach when you have a stateful bean is to manually create your object yourself, i.e., using **`new`**. This way, a fresh object will be created for each test, so the test methods are independent of each other.
- Write test methods in **`CartRepositoryImpl`** as follows:

    1. **`cart_emptyInitially()`**
       Test that the shopping cart is empty initially.

    2. **`addItems_itemsAdded()`**
       Test that if you add different item IDs to the cart, each item is held as a separate entry in the map.

    3. **`addSameItem_countIncremented()`**
       Test that if you add the same item ID to the cart several times, the item quantity is incremented (rather than inserting a separate entry in the map).

    4. **`removeItem_itemRemoved()`**
       Test that if you remove an item from the cart, the item entry is completely removed from the cart (rather than just decrementing the quantity).

Exercise 2: Mocking beans (`CartServiceImpl`)

Take a look at **`CartServiceImpl`** in the source code folder. This class provides business methods for managing the shopping cart, e.g., calculating the total cost of the shopping cart etc. **`CartServiceImpl`** uses an autowired **`CartRepository`** to handle all persistence.

In the test folder, add a class named **`CartServiceImplTests`** to test the class. Here are some hints and suggestions:

- Autowire a **`CartServiceImpl`** bean into your test class. This is an appropriate approach now (rather than manually creating a **`CartServiceImpl`** object yourself) because it's a stateless bean. Spring Boot will create a **`CartServiceImpl`** bean and use the same bean for all your tests. This is fine now because the bean is stateless, so it's OK to reuse the same bean in all the test methods.
- When Spring Boot creates the **`CartServiceImpl`** bean, it needs to autowire in a **`CartRepository`** bean. It's important it doesn't autowire a "real" repository bean because that might entail database access etc., which is inappropriate in a unit test. Therefore, in your test class, use the **`@MockBean`** annotation to create a mock **`CartRepository`** bean. Spring Test will automatically inject this mock bean into your **`CartServiceImpl`** bean.
- Write test methods in **`CartServiceImpl`** as follows:

    1. **`addItemToCart_itemAdded()`**
       Call **`addItemToCart()`** on your cart service bean. Specify a valid item ID between 0 and 4 inclusive (these are the valid item IDs in the catalog, as defined in the **`Catalog`** component).

       If the cart service bean method is implemented correctly, it should call **`add()`** on the cart repository bean. To verify it did this, call the Mockito **`verify()`** function and verify that **`add()`** was invoked upon the mock cart repository bean, with the ID and quantity you specified. For example:

       ```
       verify(mockRepo).add(eq(1),eq(100));
       ```

    2. **`addUnknownItemToCart_noAction()`**
       Call **`addItemToCart()`** on your cart service bean with an unknown item ID. In this case it's important the cart service bean method *doesn't* call **`add()`** on the cart repository bean. You can verify this as follows:

       ```
       verify(mockRepo,times(0)).add(anyInt(),anyInt());
       ```

3. **`removeItemFromCart_itemRemoved()`**
   Call **`removeItemFromCart()`** on your cart service bean with a valid item ID. The cart service bean should call **`remove()`** on the cart repository bean in this case. Verify this is what happens.

4. **`removeUnknownItemFromCart_noAction()`**
   Call **`removeItemFromCart()`** on your cart service bean with an unknown item ID. In this case, the cart service bean should not call **`remove()`** on the cart repository bean. Verify this is the case.

5. **`calculateCartCost_correctCostReturned()`**
   In this test you'll confirm that the cart service bean calculates the total cart cost successfully. To make this calculation, the cart service bean must first call **`getAll()`** on the cart repository bean. In this unit test, you're using a mock cart repository bean, so you must tell Mockito what to return when the **`getAll()`** function is called. You can do this as follows:

   ```
   Map<Integer, Integer> cart = new HashMap<>();
   cart.put(2, 1);
   cart.put(3, 2);
   cart.put(4, 5);
   when(mockRepo.getAll()).thenReturn(cart);
   ```

   Once you've done this, you can proceed to call **`calculateCartCost()`** on the cart service bean. Assert that it returns the correct value.

Exercise 3: Testing a CrudRepository (`ProductSuggestionCrudRepository`)

Take a look at **`ProductSuggestionCrudRepository`** in the source code folder. It's a simple CrudRepository interface with some custom queries to modify product suggestions in the database.

In the test folder, add a class named **`ProductSuggestionCrudRepositoryTests`** to test the interface. Here are some hints and suggestions:

- Annotate the test class with **`@DataJpaTest`**. As you will recall from the chapter, this annotation sets up a minimal set of beans to help with testing an in-memory database.
- Autowire a **`ProductSuggestionCrudRepository`** bean into the test class. You'll write methods to test this bean shortly.
- Also, autowire a **`TestEntityManager`** bean into the test class. You'll use this bean to populate the database with sample data, for use in the tests.
- Write test methods in **`ProductSuggestionCrudRepository`** as follows (note that Spring Boot will start/rollback a new transaction for each test method, to ensure the data in the database is unaffected after each test):

    1. **`testModifyPrice()`**
       First, insert a product suggestion into the database (call **`persist()`** on the **`TestEntityManager`** bean to do this). Then call **`modifyPrice()`** on your repository bean, to modify its *price* value in the database. To confirm the update worked successfully, call **`findById()`** on your repository bean to fetch it from the database, then assert it has the new price.

    2. **`testModifySales()`**
       First, insert a product suggestion into the database (call **`persist()`** on the **`TestEntityManager`** bean to do this). Then call **`modifySales()`** on your repository bean, to modify its *sales* value in the database. To confirm the update worked successfully, call **`findById()`** on your repository bean to fetch it from the database, then assert it has the new sales value.

Exercise 4: Testing a REST controller (`ProductSuggestionController`)

Take a look at **`ProductSuggestionController`** in the source code folder. It's a REST controller to query and modify product suggestions.

In the test folder, add a class named **`ProductSuggestionControllerTests`** to perform *integration testing* on this REST controller. Here are some hints and suggestions:

- Annotate the test class with **`@SpringBootTest`**. On this annotation, set the property **`webEnvironment=SpringBootTest.WebEnvironment.RANDOM_PORT`**. This causes Spring Boot to start a real web server (Tomcat) on a random port, which is a good test of reality in an integration test. (Note that the default value of **`webEnvironment`** is **`SpringBootTest.WebEnvironment.MOCK`**, which loads a mock web server rather than a real one; this is OK for unit testing, but not good enough for integration testing).
- Also annotate the test class with **`@TestMethodOrder`**. On this annotation, pass in **`MethodOrderer.OrderAnnotation.class`** as a parameter. This is a JUnit mechanism that indicates tests should be invoked in the order specified by **`@Order`** annotations. You'll see why this is important shortly…
- Autowire a **`TestRestTemplate`** bean into the test class. You'll use this bean to call REST endpoints in your tests.
- Write test methods in **`ProductSuggestionControllerTests`** as follows. Annotate each test method with **`@Order(x)`**, where x is sub-bullet number:

    1. **`testInsertProductSuggestion()`**
       This test will execute first, and it will verify that a product suggestion can be inserted successfully. In this test:
        - Create a sample **`ProductSuggestion`** object.
        - Send a POST request to the **`/productSuggestions`** REST endpoint to insert the product suggestion object.
        - Verify that the HTTP response is correct, i.e., the status code should be 201 and the HTTP response body should contain the product suggestion with a valid ID.
        - You'll need the ID for subsequent test methods, so stick it into a static variable (for example) so you can use it later.

    2. **`testGetAllProductSuggestions()`**
       Send a GET request to the **`/productSuggestions`** REST endpoint. Verify it returns a status code of 200, and the HTTP response body contains collection of all the product suggestions (the collection should contain the item you inserted in test 1 above, plus any items you might have inserted into the database in the **`SeedDb.java`** class in the app).

3. **`testGetProductSuggestion()`**
   Send a GET request to **`/productSuggestions/<id>`** (where <id> is the ID of the item you inserted in test 1 above). Verify it returns a status code of 200, and the HTTP response body contains the correct product suggestion details.

4. **`testModifyPrice()`**
   Send a PUT request to a URL such as the following:

   `/productSuggestions/modifyPrice/<id>?newPrice=<price>`

   where <id> is the ID of the item you inserted in test 1 above, and <price> is a suitable new price value.

   To test that it worked, issue a GET request to get the product suggestion back again, and verify its price value has been updated successfully.

5. **`testModifySales()`**
   Send a PUT request to a URL such as the following:

   `/productSuggestions/modifySales/<id>?newSales=<sales>`

   where <id> is the ID of the item you inserted in test 1 above, and <sales> is a suitable new sales value.

   To test that it worked, issue a GET request to get the product suggestion back again, and verify its sales value has been updated successfully.

6. **`testDeleteAll()`**
   Send a DELETE request to **`/productSuggestions`**. To test that it worked, issue a GET request to get all product suggestions, and verify the collection is empty.