Writing unit test cases

Writing unit test cases for an e-commerce application in Spring Boot involves testing individual code units, such as classes and methods, in isolation to ensure they work correctly. In this documentation, we will cover the theory and provide examples of how to write unit test cases using the JUnit and Mockito frameworks.

Table of Contents

Overview of Unit Testing in Spring Boot:

Unit testing is a vital part of the software development process as it helps identify bugs early, increases code quality and provides confidence in the application's behaviour. Spring Boot offers excellent support for unit testing, allowing you to test individual components, services, and controllers.

Writing Unit Test Cases:

Let's explore how to write unit test cases for different components in a Spring Boot application.

Testing Spring Components:

In Spring Boot, you can use JUnit to test Spring components like services and repositories. Here's an example:

```
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import static org.junit.jupiter.api.Assertions.assertEquals;

@SpringBootTest
public class ProductServiceTest {

    @Autowired
    private ProductService productService;

    @Test
    public void testGetProductById() {
        Product product = productService.getProductById(1L);
        assertEquals("Sample Product", product.getName());
    }
}
```

In the above example, we use JUnit's @Test annotation to mark the method as a test case. The @Autowired annotation injects the ProductService instance for testing. We then assert that the product's name retrieved using the getProductById method matches our expectations.

Mocking Dependencies with Mockito:

When testing components with dependencies, it's often necessary to mock them. Mockito is a popular mocking framework that works seamlessly with Spring Boot. Here's an example:

```
import org.junit.jupiter.api.Test;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.springframework.boot.test.context.SpringBootTest;
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.mockito.Mockito.when;


@SpringBootTest
public class OrderServiceTest {
    @Mock
    private ProductService productService;

    @InjectMocks
    private OrderService orderService;

    @Test
    public void testPlaceOrder() {
        // Mocking the behaviour of the productService
        when(productService.getProductById(1L)).thenReturn(new Product("Sample
Product"));

        Order order = new Order();
        order.setProductId(1L);
        order.setQuantity(5);
        orderService.placeOrder(order);

        assertEquals(1L, order.getProductId());
        assertEquals(5, order.getQuantity());
    }
}
```

In this example, we're using the @Mock annotation to create a mock instance of the ProductService, and @InjectMocks to inject the mocked dependency into the OrderService, by using when(...).thenReturn(...) with Mockito, we define the behaviour of the mocked productService. We then create an Order object, call the placeOrder method, and assert the expected values.

Testing Spring MVC Controllers:

You can use the MockMvc class provided by Spring Boot to test Spring MVC controllers. It allows you to perform requests and verify the responses. Here's an example:

```
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockMvc;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.http.MediaType;
```

```java
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.springframework.test.web.servlet.result.MockMvcResultHandlers.print;

@SpringBootTest
@AutoConfigureMockMvc
public class ProductControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @Test
    public void testGetProductById() throws Exception {
        mockMvc.perform(MockMvcRequestBuilders.get("/products/{id}", 1L)
                .contentType(MediaType.APPLICATION_JSON))
                .andExpect(MockMvcResultMatchers.status().isOk())

.andExpect(MockMvcResultMatchers.jsonPath("$.name").value("Sample Product"))
                .andDo(print());
    }
}
```

In this example, we're using MockMvc to perform a GET request to the "/products/{id}" endpoint and verify the response using MockMvcResultMatchers. The andDo(print()) method prints the response details for debugging purposes.

Best Practices for Unit Testing:

- Write tests early in the development process.
- Test edge cases, invalid inputs, and exception handling.
- Follow the Arrange-Act-Assert (AAA) pattern for the test method structure.
- Aim for test independence and avoid test order dependencies.
- Use meaningful test methods and variable names.
- Keep tests focused and avoid testing multiple functionalities in a single test.
- Regularly maintain and update tests as the codebase evolves.

Conclusion:

Unit testing is crucial for ensuring the reliability and correctness of your e-commerce application. With the help of Spring Boot, JUnit, and Mockito, you can write effective unit test cases for different components of your application. By following best practices and maintaining a robust test suite, you can improve the quality and stability of your e-commerce application.