**WEEK 4&5**

**EXERCISE 13**

**Online Bookstore - Unit Testing REST Controllers**

**Business Scenario**

The task is to write unit tests for the bookstore's REST controllers to ensure that the endpoints work correctly and handle various scenarios as expected. This involves setting up JUnit and Mockito, using MockMvc for testing, and ensuring comprehensive test coverage.

**Instructions**

**1. JUnit Setup**

**Task:** Set up JUnit and Mockito in your project.

**Implementation:**

* **Add Dependencies:**

**For Maven:**

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-core</artifactId>

<version>4.5.1</version>

<scope>test</scope>

</dependency>

**For Gradle:**

testImplementation 'org.springframework.boot:spring-boot-starter-test'

testImplementation 'org.mockito:mockito-core:4.5.1'

**Explanation:** The spring-boot-starter-test dependency includes JUnit, Mockito, and other testing libraries.

**2. MockMvc**

**Task:** Use MockMvc to write unit tests for your REST controllers.

**Implementation:**

1. **Create Test Class for BookController:**

**BookControllerTest.java:**

import com.fasterxml.jackson.databind.ObjectMapper;

import org.junit.jupiter.api.BeforeEach;

import org.junit.jupiter.api.Test;

import org.mockito.InjectMocks;

import org.mockito.Mock;

import org.mockito.MockitoAnnotations;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;

import org.springframework.test.web.servlet.MockMvc;

import org.springframework.test.web.servlet.setup.MockMvcBuilders;

import org.springframework.web.context.WebApplicationContext;

import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.\*;

import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.\*;

@WebMvcTest(BookController.class)

public class BookControllerTest {

@Autowired

private MockMvc mockMvc;

@Mock

private BookService bookService;

@InjectMocks

private BookController bookController;

@BeforeEach

public void setup() {

MockitoAnnotations.openMocks(this);

mockMvc = MockMvcBuilders.standaloneSetup(bookController).build();

}

@Test

public void testGetBookById() throws Exception {

Book book = new Book(1L, "Book Title", "Book Author", 29.99, "1234567890");

Mockito.when(bookService.getBookById(1L)).thenReturn(book);

mockMvc.perform(get("/books/1"))

.andExpect(status().isOk())

.andExpect(jsonPath("$.title").value("Book Title"))

.andExpect(jsonPath("$.author").value("Book Author"))

.andExpect(jsonPath("$.price").value(29.99))

.andExpect(jsonPath("$.isbn").value("1234567890"));

}

@Test

public void testCreateBook() throws Exception {

Book book = new Book(1L, "New Book", "New Author", 19.99, "0987654321");

Mockito.when(bookService.createBook(Mockito.any(Book.class))).thenReturn(book);

mockMvc.perform(post("/books")

.contentType("application/json")

.content(new ObjectMapper().writeValueAsString(book)))

.andExpect(status().isCreated())

.andExpect(jsonPath("$.title").value("New Book"))

.andExpect(jsonPath("$.author").value("New Author"))

.andExpect(jsonPath("$.price").value(19.99))

.andExpect(jsonPath("$.isbn").value("0987654321"));

}

@Test

public void testUpdateBook() throws Exception {

Book book = new Book(1L, "Updated Book", "Updated Author", 25.99, "1234567890");

Mockito.when(bookService.updateBook(Mockito.anyLong(), Mockito.any(Book.class))).thenReturn(book);

mockMvc.perform(put("/books/1")

.contentType("application/json")

.content(new ObjectMapper().writeValueAsString(book)))

.andExpect(status().isOk())

.andExpect(jsonPath("$.title").value("Updated Book"))

.andExpect(jsonPath("$.author").value("Updated Author"))

.andExpect(jsonPath("$.price").value(25.99))

.andExpect(jsonPath("$.isbn").value("1234567890"));

}

@Test

public void testDeleteBook() throws Exception {

Mockito.doNothing().when(bookService).deleteBook(1L);

mockMvc.perform(delete("/books/1"))

.andExpect(status().isNoContent());

}

}

**Explanation:**

* + @WebMvcTest is used for testing the controller layer.
  + MockMvc is used to perform HTTP requests and assert responses.
  + Mockito.when() is used to mock service layer responses.

**3. Test Coverage**

**Task:** Ensure comprehensive test coverage and follow best practices for testing.

**Implementation:**

* + **Test All Scenarios:** Cover positive and negative test cases.
  + **Use Mocking:** Mock dependencies to isolate the controller logic.
  + **Validate Responses:** Assert that responses contain the expected data.
  + **Test Edge Cases:** Handle invalid inputs and unexpected scenarios.

**Example:** Ensure tests cover:

* + Successful creation, update, and deletion of books.
  + Handling of not found and bad request scenarios.

**Conclusion:**

By writing unit tests for your REST controllers, you ensure that your endpoints function correctly and handle various scenarios. Using JUnit and Mockito with MockMvc helps in isolating the controller logic and validating the behaviour of your REST API.