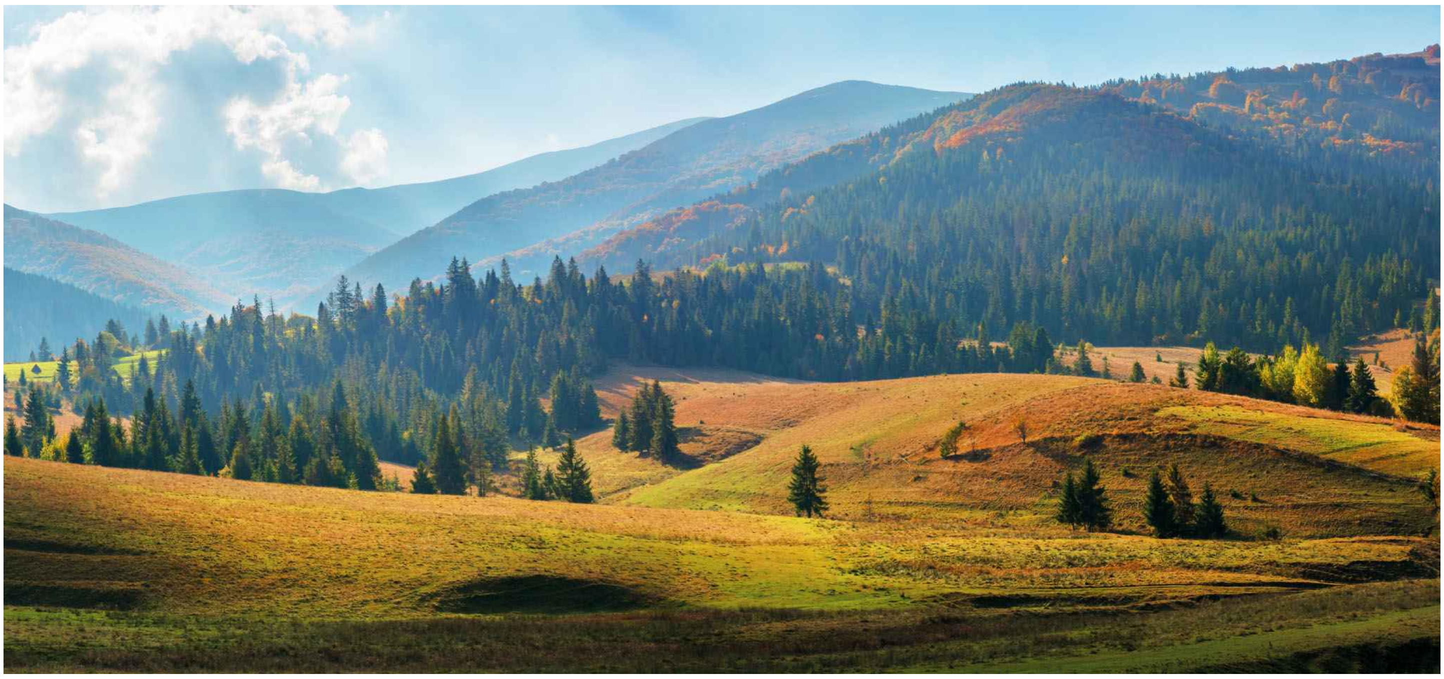

11-MVC Test Annotations

Author: Vincent Lau

Learning Objectives

- Understand the difference between @WebMockMvc and @SpringBootTest
- Understand the difference between @Mock and @MockBean
- Understand the relationship between @InjectMock, @Mock, @MockBean

Introduction

In this chapter, we will try to compare those similar annotations, so that you will be getting more familiar with using them in different scenarios. For example, in Spring, why do we need @MockBean? When do we use @Mock? What are the differences between @SpringBootTest & @WebMockMvc? Let's start.

@Mock vs @MockBean

`@Mock` and `@MockBean` are annotations commonly used in testing frameworks, but they serve different purposes and are used in different contexts.

@Mock (from Mockito)

- `@Mock` is used to create a mock object of a class.
- It is often used in unit testing to isolate the behavior of a specific class or method by mocking its dependencies.
- `@Mock` is typically used when you want to mock objects that are not managed by the Spring application context.

@MockBean(Spring-specific)

- `@MockBean` is used to create a mock object of a bean managed by the Spring application context.
- It is commonly used in integration testing to replace real beans with mock implementations when testing components or services.
- `@MockBean` is specifically designed for Spring Boot applications, and it's used to mock Spring beans, allowing you to control their behavior during testing.

@Mock Example

Suppose you have a class `Calculator` that depends on a `MathService`. You want to test `Calculator` in isolation by mocking the `MathService` dependency.

In this example, `@Mock` is used to create a mock of `MathService` so that you can control its behavior during the test.

```java
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.mockito.Mock;
import org.mockito.MockitoAnnotations;

import static org.mockito.Mockito.when;
import static org.junit.jupiter.api.Assertions.assertEquals;

@ExtendWith(MockitoExtension.class)
class CalculatorTest {

    // Mock the object
    // so that we can isolate the test of mathService.add() from calculator.add()
    @Mock
    private MathService mathService;

    private Calculator calculator;

    @BeforeEach
    void init() {
        // Inject the mock into the class under test
        calculator = new Calculator(mathService);
    }

    @Test
    void testAdd() {
        // Mockito to mock the service behavior
        when(mathService.add(2, 3)).thenReturn(5);
        // Perform the test, invoking the call to calculator.add()
        int result = calculator.add(2, 3);
        // Check if the expected result = actual result
        assertEquals(5, result);
    }
}
```

@MockBean Example (with Spring Boot)

Suppose you have a Spring Boot service `UserService` that depends on a repository `UserRepository`. You want to write an integration test for `UserService` while mocking `UserRepository`.

```java
import org.junit.jupiter.api.Test;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.mock.mockito.MockBean;

import static org.mockito.Mockito.when;
import static org.junit.jupiter.api.Assertions.assertEquals;

@WebMvcTest
class MathControllerTest {

    @Autowird
    private MathController mathController;

    @Autowird
    private MockMvc mockMvc;

    @MockBean
    private MathService mathService;

    @Test
    void testController() {
        // Mock the behavior of userRepository
        when(MathService.add(2, 3)).thenReturn(100);

        mockMvc.perform(get("/api/v1/math")
                .param("x", 2)
                .param("y", 3)) //
            .andExpect(status().isOk()) // HTTP 200
            .andExpect(content().contentType(MediaType.APPLICATION_JSON))
            .andExpect(jsonPath("$.result").value(100))
            .andDo(print());
    }
}
```

In this example, `@MockBean` is used to create a mock of `MathService`, which is a Spring bean. This allows you to control the behavior of the object mathService when testing `MathController` within the Spring application context.

In summary, `@Mock` is used for creating mock objects in unit tests, while `@MockBean` is used for creating mock beans in integration tests within a Spring context, typically in Spring Boot applications. The choice between them depends on the context and the type of testing you are performing.

@SpringBootTest vs @WebMvcTest

`@WebMvcTest` and `@SpringBootTest` are annotations used in Spring Boot applications to facilitate testing at different levels of your application.

@SpringBootTest

- `@SpringBootTest` is used to perform integration testing by loading the entire Spring application context.
- It allows you to test the entire application, including all the layers (web, service, repository, etc.).
- It provides a broader scope for testing and is suitable for testing interactions between various components.

@WebMvcTest

- `@WebMvcTest` is used to test the web layer of a Spring Boot application, specifically the controllers.

- It focuses on the MVC (Model-View-Controller) components of your application.
- It loads only the web-related components, such as controllers, and doesn't load the entire Spring application context.

@WebMvcTest Example

In this example, `@WebMvcTest` is used to test the `MyController` without loading the entire Spring context. It focuses on the web layer and provides a `MockMvc` instance for simulating HTTP requests and responses.

```java
import org.junit.jupiter.api.Test;
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.request.MockMvcRequestBuilders;
import org.springframework.test.web.servlet.result.MockMvcResultMatchers;

@WebMvcTest(MyController.class) // Specify the controller(s) to test
public class MyControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @Test
    public void testController() throws Exception {
        mockMvc.perform(MockMvcRequestBuilders.get("/your-endpoint"))
               .andExpect(MockMvcResultMatchers.status().isOk())
               .andExpect(/* Your expectations for the controller */);
    }
}
```

@SpringBootTest Example

In this example, `@SpringBootTest` is used to perform an integration test by **loading the entire Spring context, including all layers of the application**. It uses `TestRestTemplate` to make HTTP requests to a running instance of the application.

```java
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.web.client.TestRestTemplate;
import org.springframework.boot.web.server.LocalServerPort;
import org.springframework.http.ResponseEntity;
```

```java

@SpringBootTest // Loads the entire Spring context
public class MyIntegrationTest {

    @Autowird
    private MathController mathController;

    @Autowird
    private MockMvc mockMvc;

    @Test
    void testController() {

        mockMvc.perform(get("/api/v1/math")
                .param("x", 2)
                .param("y", 3)) //
            .andExpect(status().isOk()) // HTTP 200
            .andExpect(content().contentType(MediaType.APPLICATION_JSON))
            .andExpect(jsonPath("$.result").value(5))
            .andDo(print());
    }
}
```

In summary, `@WebMvcTest` is used for testing the web layer (controllers) in isolation, while `@SpringBootTest` is used for broader integration testing of the entire application. The choice between them depends on the specific testing needs of your application and the scope of the tests you want to perform.

Relationship between @InjectMock, @Mock & @MockBean

In the context of Mockito and Spring testing, `@InjectMocks` is used to inject mocks into an instance of the class being tested, while `@Mock` and `@MockBean` are used to create mock objects.

Here's how they work together:

@Mock

- @Mock is used to create mock objects. You annotate fields with @Mock to create mock instances of classes or interfaces.
- It is typically used in unit testing with Mockito to **isolate the class being tested** (let's call it Class A) by mocking its dependencies (Class B, for example).

Example:

```java
@Mock
private ClassB classB;
```

@InjectMocks

- @InjectMocks is used to inject mocks (created with @Mock) into an instance of the class being tested (Class A).
- **It tells Mockito to inject the mocked dependencies into the class instance being tested.**

Example:

```java
@Mock
private ClassB classB;

// Injects the mock dependencies (classB) into this instance
@InjectMocks
private ClassA classA;
```

@MockBean (Spring-specific)

- `@MockBean` is used in Spring testing, particularly in Spring Boot applications.
- It creates a mock bean for a Spring-managed component (e.g., a service or repository) and registers it in the Spring application context.
- **It doesn't require `@InjectMocks` because the Spring context takes care of injecting the mock bean into the class being tested.**

Example:

```java
@MockBean
private UserService userService; // Mocking a Spring bean, which will be
loaded into Spring Application Context
```

Summary

When you're using `@Mock` and `@InjectMocks`, you're typically working with Mockito in a non-Spring context to manually inject mocks into a class.

In contrast, when you're using `@MockBean`, you're working within a Spring context (e.g., in a Spring Boot integration test) where **Spring manages bean injection, so you don't need `@InjectMocks`**. Spring Boot will automatically inject the `@MockBean` into the class under test as long as it's properly configured and initialized by Spring.