10-Spring Web MVC Test

Author: Vincent Lau

Learning Objectives

- Understand how to leverage Spring Test Framework to test Controller for RESTful API - WebMvcTest, MockMvc & MockBean.

Introduction

`@WebMvcTest` is a Spring Boot test annotation that focuses on testing the web layer of your application, specifically the controllers. It allows you to isolate and test individual controllers or slices of the web layer in your Spring Boot application without starting a complete application context.

Key Features and Use Cases

Controller Testing

`@WebMvcTest` is primarily used for testing Spring MVC controllers. It can be used to test the request mappings, request and response handling, and controller logic.

Slicing the Application Context

When you use `@WebMvcTest`, Spring Boot slices the application context to only include the beans relevant to the web layer, such as controllers, advice, and view components. This reduces the overhead of loading unnecessary beans, making the tests faster.

Integration with MockMvc

`@WebMvcTest` integrates with `MockMvc`, which allows you to simulate HTTP requests and verify the behavior of your controllers without making actual HTTP requests.

Limited Component Scanning

Only the components related to the web layer are scanned during testing, which helps maintain isolation from other parts of the application.

Related Annotations

@WebMvcTest

1. @WebMvcTest Configuration: When you use `@WebMvcTest` to annotate your test class, Spring Boot automatically configures a minimal application context that includes the necessary components for testing the web layer. This includes controllers, advice, view resolvers, and, importantly, a configured `MockMvc` instance.
2. MockMvc Auto-Configuration: Part of the `@WebMvcTest` configuration is the automatic configuration of a `MockMvc` bean. This means that Spring Boot creates and configures a `MockMvc` instance specifically for testing the controllers in your application.

@MockBean

The `@MockBean` annotation is a powerful feature of the Spring Boot testing framework that allows you to mock and replace beans in the Spring application context during testing. It's commonly used to isolate components and dependencies, such as services or repositories, when testing a particular part of your application.

Here are the key features and use cases of the `@MockBean` annotation:

Key Features and Use Cases:

1. Bean Mocking: `@MockBean` is used to create a mock instance of a Spring bean. This allows you to replace a real implementation of a bean with a mock implementation in the application context during testing.

2. Isolation: It helps isolate the component you are testing by replacing its dependencies with mock objects. This is particularly useful when you want to focus on testing a specific component in isolation without involving its real dependencies.

The Bean of MockMvc

`MockMvc` is a part of the Spring Test framework and is used for testing Spring MVC applications. It provides a way to perform HTTP requests against your controllers and interact with your Spring MVC application in a controlled and isolated environment during testing. You can think of it as a tool for simulating HTTP requests and verifying the behavior of your controllers without making actual HTTP requests to a running server.

By annotating a field with `@Autowired` (as in `@Autowired private MockMvc mockMvc;`), Spring Boot automatically injects the `MockMvc` instance into your test class. This is possible because Spring Boot knows that you are using `@WebMvcTest` and automatically wires the `MockMvc` bean to the `mockMvc` field.

Simulating HTTP Requests

With `MockMvc`, you can simulate HTTP requests like GET, POST, PUT, DELETE, etc., by specifying the URL, request parameters, headers, and body content.

Testing Controller Behavior

You can use `MockMvc` to test various aspects of your controller's behavior, including request mapping, request handling, model attributes, and response generation.

Assertions and Verifications

After performing a simulated request, you can use assertions to verify the expected behavior of your controller, such as checking the HTTP response status, content, headers, and more.

Isolation

`MockMvc` allows you to isolate the controller under test from the rest of the application, making it suitable for unit testing or focused integration testing of specific controllers.

Code Example

When testing a Spring MVC controller that interacts with services or repositories, it's a common practice to use Mockito to mock those dependencies. This allows you to isolate the controller and focus your testing specifically on its behavior. Here's an example of how to use `@WebMvcTest` in a Spring Boot test.

Suppose you have a `UserService` that the `UserController` relies on:

```
@Service
public class UserService {

    public User getUser() {
        // Simulate fetching a User object from a repository or database
        User user = new User();
        user.setId(1L);
        user.setUsername("john_doe");
        user.setEmail("john@example.com");
        return user;
    }
}
```

Your `UserController` might look like this:

```
@RestController
public class UserController {

    @Autowired
    private final UserService userService;

    @GetMapping("/user")
    public User getUser() {
        return userService.getUser();
    }
}
```

Now, let's write a test for the `UserController` that includes Mockito:

- `@WebMvcTest(UserController.class)` specifies that only the `UserController` and its related components should be loaded into the test context.
- `MockMvc` is auto-injected, allowing you to perform HTTP requests and make assertions about the responses.
- `@MockBean` annotation to inject a mocked `UserService` bean into the test context.
- Using Mockito in this way allows you to isolate the controller and focus on its behavior without making actual service calls, which is a common practice in unit testing.
- In the test method, we use Mockito to mock the behavior of the `userService` by specifying that `userService.getUser()` should return the `expectedUser`.

- The test method uses `MockMvc` to simulate an HTTP GET request to "/users" and then verifies the HTTP status, content type, and JSON data in the response.

```java
import org.junit.jupiter.api.Test;
import org.mockito.Mock;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.ResultActions;
import org.springframework.test.web.servlet.request.MockMvcRequestBuilders;

import static org.mockito.Mockito.when;
import static
org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;

@WebMvcTest(UserController.class)
public class UserControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private UserService userService;

    @Test
    public void testGetUser() throws Exception {
        // Create a User object for expected output
        User expectedUser = new User();
        expectedUser.setId(1L);
        expectedUser.setUsername("john_doe");
        expectedUser.setEmail("john@example.com");

        // Mock the behavior of the userService
        when(userService.getUser()).thenReturn(expectedUser);

        // Perform an HTTP GET request to "/user"
        ResultActions result =
mockMvc.perform(MockMvcRequestBuilders.get("/user"));

        // Verify the HTTP response
        result.andExpect(status().isOk())
              .andExpect(content().contentType("application/json"))
              .andExpect(jsonPath("$.id").value(1))
              .andExpect(jsonPath("$.username").value("john_doe"))
              .andExpect(jsonPath("$.email").value("john@example.com"));
    }
```

```
42 }
```

If the mock result from the service layer is a List or Array data format, the syntax to test jsonPath should be:

```
1 mockMvc.perform(get("/users"))
2             .andExpect(status().isOk())
3             .andExpect(content().contentType(MediaType.APPLICATION_JSON))
4             .andExpect(jsonPath("$[0].name", is("John")))
5             .andExpect(jsonPath("$[1].name", is("Alice")));
```

Limitations

No Full Application Context

`@WebMvcTest` focuses on the web layer and does not load the entire application context. If your test requires access to other parts of the application, you may need to use additional annotations like `@SpringBootTest`.

Controller Test Only

It does not cover integration with other layers of the application, such as service or repository layers. For that, you might need to use `@SpringBootTest` or other testing approaches.

Summary

In summary, `@WebMvcTest` is a valuable tool for testing Spring MVC controllers and the web layer of your Spring Boot application efficiently and in isolation. It promotes a faster and more focused testing approach for your web components.