Assignment: Controllers/API Testing

In service applications (like your Spring Boot Rest App) it's really important to make sure the application conforms to the expected request and response format or contract. Breaking this contract might impact harmfully all client applications (like your React frontend) depending on it.

In this assignment you will learn a way of testing your controllers on the API level (HTTP requests and responses) using Spring WebMvcTest. It works nicely with Mockito and we will do so in order to mock the business layer.

Testing your controllers on the API level will enable you to evolve your application (including library upgrades) assuring that no breaking change was introduced.

Make sure you have the starter app!

Download the starter app here.

Difficulty: ☆☆☆☆

Learning objectives:

- I can use SpringMvcTest in order to test my controllers
- I can test GET endpoints
- I can test POST endpoints
    - Happy flow
    - Validation errors flow
- I can test PUT and DELETE endpoints

Step 1: Getting familiar with Spring MVC testing

Please scan the following article: Testing MVC Web Controllers with Spring Boot and @WebMvcTest.

Step 2: GET Countries endpoint test example

Let's create tests for the CountriesController class. We will start with the GET endpoint.

This is how the test class looks like. Comments will follow.

```
package fontys.sem3.school.controller;

import fontys.sem3.school.business.CreateCountryUseCase;
import fontys.sem3.school.business.GetCountriesUseCase;
import fontys.sem3.school.domain.Country;
import fontys.sem3.school.domain.GetCountriesResponse;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
```

```java
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.test.context.junit.jupiter.SpringExtension;
import org.springframework.test.web.servlet.MockMvc;

import java.util.List;

import static org.mockito.Mockito.*;
import static org.springframework.http.MediaType.APPLICATION_JSON_VALUE;
import static
org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static
org.springframework.test.web.servlet.result.MockMvcResultHandlers.print;
import static
org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;

@ExtendWith(SpringExtension.class)
@WebMvcTest(CountriesController.class)
class CountriesControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private GetCountriesUseCase getCountriesUseCaseMock;

    @MockBean
    private CreateCountryUseCase createCountryUseCaseMock;


}
```

This is partial integration test. We need a spring extension in order to start-up the application partially (only Spring web components) so our controller can actually receive http requests before the tests start.

Formatted: Underline

The annotations on the class level take care of that:

```java
@ExtendWith(SpringExtension.class)
@WebMvcTest(CountriesController.class)
```

Notice that @WebMvcTest receives the controller class that will be tested, *CountriesController* in this case.

Field *mockMvc* is injected by Spring and will provide us means to send http requests to our web application and also check the responses.

The other fields are dependencies of *CountriesController*. Annotating them with @MockBean tells Spring these fields should be Mockito mocks. Spring automatically takes care of injecting the mocks into *CountriesController*.

Adding the test method for the GET endpoint will result in the following code:

```java
package fontys.sem3.school.controller;

import fontys.sem3.school.business.CreateCountryUseCase;
import fontys.sem3.school.business.GetCountriesUseCase;
import fontys.sem3.school.domain.Country;
```

```java
import fontys.sem3.school.domain.GetCountriesResponse;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.test.context.junit.jupiter.SpringExtension;
import org.springframework.test.web.servlet.MockMvc;

import java.util.List;

import static org.mockito.Mockito.*;
import static org.springframework.http.MediaType.APPLICATION_JSON_VALUE;
import static
org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static
org.springframework.test.web.servlet.result.MockMvcResultHandlers.print;
import static
org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;

@ExtendWith(SpringExtension.class)
@WebMvcTest(CountriesController.class)
class CountriesControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private GetCountriesUseCase getCountriesUseCaseMock;

    @MockBean
    private CreateCountryUseCase createCountryUseCaseMock;

    @Test
    void getCountries_shouldReturn200ResponseWithCountriesArray() throws
Exception {
        GetCountriesResponse response = GetCountriesResponse
                .builder()
                .countries(List.of(
                        Country.builder().id(1L)
                                .name("Netherlands").code("NL").build(),
                        Country.builder().id(2L)
                                .name("Brazil").code("BR").build()
                ))
                .build();
        when(getCountriesUseCaseMock.getCountries())
                .thenReturn(response);

        mockMvc.perform(get("/countries"))
                .andDo(print())
                .andExpect(status().isOk())
                .andExpect(header().string("Content-Type",
APPLICATION_JSON_VALUE))
                .andExpect(content().json("""

{"countries":[{"id":1,"code":"NL","name":"Netherlands"},{"id":2,"code":"BR"
,"name":"Brazil"}]}
                        """));

        verify(getCountriesUseCaseMock).getCountries();
        verifyNoInteractions(createCountryUseCaseMock);
```

```
    }
}
```

Read carefully line-by-line the test method. Can you figure it out? If yes, go ahead and run the test.

Step 3: Individual assignment GET endpoint. Your turn!

Choose a similar GET endpoint of your own individual assignment and write a SpringWebMvc test for it.

Step 4: POST Country endpoint test example

Now let's test the POST endpoint of *CountriesController*. It should create a country if all validations are ok.

For checking that we will need at least two tests: one for the happy flow and one for validation errors.

Happy flow test:

```
@Test
void createCountry_shouldCreateAndReturn201_WhenRequestValid() throws
Exception {
    CreateCountryRequest expectedCountry = CreateCountryRequest
            .builder()
            .code("NL")
            .name("Netherlands")
            .build();
    when(createCountryUseCaseMock.createCountry(expectedCountry))
            .thenReturn(CreateCountryResponse
                            .builder()
                            .countryId(100L)
                            .build());

    mockMvc.perform(post("/countries")
                    .contentType(APPLICATION_JSON_VALUE)
                    .content("""
                            {
                                "code": "NL",
                                "name": "Netherlands"
                            }
                            """))
            .andDo(print())
            .andExpect(status().isCreated())
            .andExpect(header().string("Content-Type",
APPLICATION_JSON_VALUE))
            .andExpect(content().json("""
                        {"countryId": 100}
                    """));

    verify(createCountryUseCaseMock).createCountry(expectedCountry);
}
```

Read it carefully line-by-line and if you understand try it out. You will need to import extra static methods, like *post*.

Input validations test:

```
@Test
void createCountry_shouldNotCreateAndReturn400_WhenMissingFields() throws
Exception {
    mockMvc.perform(post("/countries")
                    .contentType(APPLICATION_JSON_VALUE)
                    .content("""
                            {
                                "code": "",
                                "name": ""
                            }
                            """))
            .andDo(print())
            .andExpect(status().isBadRequest())
            .andExpect(header().string("Content-Type",
APPLICATION_JSON_VALUE))
            .andExpect(content().json("""
                    [
                    {"field":"name","error":"must not be blank"},
                    {"field":"code","error":"must not be blank"},
                    {"field":"name","error":"length must be between 2
and 50"}
                    ]
                    """));

    verifyNoInteractions(createCountryUseCaseMock);
}
```

Again: read it carefully line-by-line and if you understand try it out.

Step 5: Individual assignment POST endpoint. Your turn!

Choose a similar POST endpoint of your own individual assignment and write a SpringWebMvc test for it.

Step 6: Other HTTP methods

Testing other http methods like PUT, DELETE, PATCH and others is pretty similar to the previous examples.

Choose a PUT and DELETE endpoint from your individual assignment and write SpringWebMvc tests for them.

Final remark

Nice that you made it so far. Congrats!

A important final remark is that after implementing security in your project, you will probably need to switch from annotation

```
@WebMvcTest(<Controller Name>.class)
```

to

```
@SpringBootTest
@AutoConfigureMockMvc
```

That is needed to make sure all your security configuration is properly loaded. But no need to worry about it now, it will be mentioned later in the authentication and authorization assignment. Just be aware that once you plug security in the project probably the controller tests will fail and this is why.