COURSEWARE

Testing

Contents

Overview

Spring provides several utilities to help with testing enterprise applications.

Integration testing with JUnit 5 allows us to see if the HTTP endpoints in our application work with various CRUD-based HTTP requests. This makes use of the **Controller** layer, which can be 'mocked' so that we can test the outcomes of sending data to each HTTP endpoint.

@SpringBootTest

The only annotation which is required at the class level for any testing done via Spring is the `@SpringBootTest` annotation. This automatically finds a `SpringBootConfiguration`, which will load an `ApplicationContext` just as if you were running the application normally.

This is most typically used for integration tests. The internal `ApplicationContext` is useful for running tests which mock certain objects, but is usually overkill for standard unit testing which does not require mocking.

Object mocking

When integration testing a Spring application, typically at the Controller layer, you will usually need to spin up a mocked Controller class to ensure that the program is working as expected.

To do this, you'll need two things: a `MockMVC` object, and its corresponding `@AutoConfigureMockMvc` annotation (at the class level) to spin it up for your test class.

The `MockMVC` object, or `Mock Model View Controller`, performs mocked *HTTP Requests* as a true user might. This allows you to test each of the endpoints as defined in your `RestController` classes.

*(note: this uses **Spring MVC**, which is a different method of mocking objects to [unit testing Spring applications, where we would use Mockito instead]().)*

Tutorial

The following tutorial uses a sample `DuckController` class, as defined below:

▶ DuckController Example

We'll be building a batch of integration tests for this `DuckController` by using a `MockMVC` to mock our Controller.

Start off by creating the test class and applying any necessary annotations.

Here, we'll:

- use the `webEnvironment` parameter to give the tests a random web port to work on, to ensure that we use one which is free;

- use the `@ActiveProfiles` annotation to point at a testing profile which utilises an embedded H2 database;
- use the `@Sql` annotation to prepopulate our database with dummy data before running any of our tests:

```
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
@AutoConfigureMockMvc
@Sql(scripts = { "classpath:wood-schema.sql", "classpath:wood-data.sql" },
    executionPhase = ExecutionPhase.BEFORE_TEST_METHOD)
@ActiveProfiles(profiles = "test")
public class DuckIntegrationTest {
    //TODO
}
```

First, we'll inject our required objects:

- we need a `MockMvc` to mock our controller, as well as any relevant mappers.
- we need a `ModelMapper` to convert our POJOs (`Duck`s) to DTOs (`DuckDTO`s).
- we need an `ObjectMapper` to convert our requests to JSON format.

```
    @Autowired
    private MockMvc mock;

    @Autowired
    private ModelMapper mapper;

    private DuckDTO mapToDTO(Duck duck) {
        return this.modelMapper.map(duck, DuckDTO.class);
    }

    @Autowired
    private ObjectMapper jsonifier;
```

Now we'll declare any required objects for our tests:

```
    private final Long TEST_ID = 1L;
    private final Duck TEST_DUCK = new Duck("Barry", "blue", "puddle");
```

Let's write a test for our **create** functionality:

```
    @Test
    public void testCreateDuck() throws Exception {
        //TODO
    }
```

First, we can make use of `MockMvcRequestBuilders` to create the mock HTTP request, as well as to set the `HTTPMethod` to use and the **URL** that your endpoint points to:

```
    MockHttpServletRequestBuilder mockRequest =
MockMvcRequestBuilders.request(HttpMethod.POST, "/duck/createDuck");
```

next, we'll set the content type of the request, in this case to `application/json`:

```
    mockRequest.contentType(MediaType.APPLICATION_JSON);
```

Next, we'll set the content of the `MockRequest`. Here we'll be using the `ObjectMapper` to convert a test object into a JSON string:

```
    mockRequest.content(this.jsonifier.writeValueAsString(testDuck));
```

Finally, we'll set the request to expect a JSON response (`application/json`):

```
    mockRequest.accept(MediaType.APPLICATION_JSON);
```

Once the mock request is set up, we can create `ResultMatcher`s to test the response.

First, we'll check that the response has the correct status code (201 in this case).

```
    ResultMatcher matchStatus = MockMvcResultMatchers.status().isCreated();
```

Second, we'll check that the content of the response matches our test object (remember we need to check it's in JSON format):

```
    ResultMatcher matchContent =
MockMvcResultMatchers.content().json(this.jsonifier.writeValueAsString(duckDTO))
;
```

Once we've done that then we'll use the `MockMvc` object to perform the mock request, checking that the result matches both of the conditions.

```

this.mock.perform(mockRequest).andExpect(matchStatus).andExpect(matchContent);
```

A note on `.perform()` and `.andExpect()`

You can also write the entire request-response logic as a single command, by using the `MockMvc.perform()` method and lots of method-chaining!

To make things more readable, you can combine this with *static imports* for specific methods inside the `MockMvcRequestBuilders` and `MockMvcResultMatchers` libraries.

For instance, here's an example `testCreateDuck()` method using method-chaining and static imports:

```
import static
org.springframework.test.web.servlet.request.MockMvcRequestBuilders.post;
import static
org.springframework.test.web.servlet.request.MockMvcRequestBuilders.put;
import static
org.springframework.test.web.servlet.result.MockMvcResultMatchers.content;
import static
org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

// etc.

    @Test
    void testCreateDuck() throws Exception {
        this.mock
            .perform(post("/duck/create")
                .accept(MediaType.APPLICATION_JSON)
                .contentType(MediaType.APPLICATION_JSON)
                .content(this.jsonifier.writeValueAsString(TEST_DUCK)))
            .andExpect(status().isCreated())

.andExpect(content().json(this.jsonifier.writeValueAsString(this.mapToDTO(TEST_D
UCK))));
    }
```

It's entirely up to you to decide which syntax is most helpful for your learning style.

Exercises

Implement integration testing for your account project.