There is no spoon. Innovative software development in action.

Unit and Integration tests in Spring Boot

May 30, 2016 | Tags: testing, spring-boot, spring

Spring Boot is an awesome project that aims to make it easy creating production ready spring powered applications, bringing a convention over configuration based setup. It makes it easier to start a project with a default setup that can be customised as you need.

Today I'd like to share with you how to write different types of tests using Spring Boot. The idea is show how you can write test using Spring Boot easily (specially after the 1.4 release).

Unit Tests

I'm sure that you have a good understanding on unit tests so I'll keep it to the basics. Unit tests are responsible for testing a specific piece of code, just a small functionality (unit) of the code. These are the tests that we want to run as fast as we can as the developer will run these tests a lot of times during the development. If you are using TDD you'll probably run it even more!

Stay out of Spring as much as you can

A good practice when working with Spring (or with JEE) is to isolate your business logic from specific Spring functionalities. Everyone want to keep their code loosely coupled and with high cohesion. When you need to mock lots of dependencies to unit test a specific thing that is a sign of high coupling. If you caught yourself in this situation, maybe it's a good idea to stop and think about separating some concerns into new classes.

Take a look into the `CreateClientService` class. You'll notice that it has a single dependency on the `ClientRepository` class. Now take a look into the `CreateClientServiceTest` and notice how I can mock the dependencies of my `CreateClientService` easily and execute simple unit tests using zero Spring functionality.

```
public class CreateClientServiceTest {

    private CreateClientService createClientService;
    private ClientRepository clientRepositoryMock;

    @Before
    public void setUp() {
```

```
        clientRepositoryMock = Mockito.mock(ClientRepository.class);
        createClientService = new CreateClientService(clientRepositoryMock);
    }

    @Test
    public void createClientSuccessfuly() throws Exception {

when(clientRepositoryMock.findByName(eq("Foo"))).thenReturn(Optional.empty())
;

doAnswer(returnsFirstArg()).when(clientRepositoryMock).save(any(Client.class)
);

        Client client = createClientService.createClient("Foo");
        assertEquals("Foo", client.getName());
        assertNotNull(client.getNumber());
    }

    @Test(expected = InvalidClientNameException.class)
    public void createClientWithEmptyName() throws Exception {
        createClientService.createClient("");
    }

    @Test(expected = ClientNameAlreadyExistsException.class)
    public void createClientWithExistingName() throws Exception {
        doThrow(new
ClientNameAlreadyExistsException()).when(clientRepositoryMock).findByName(eq(
"Foo"));

        createClientService.createClient("Foo");
    }
}
```

Another detail that is important is how we can use constructor injection as a semantic way of declaring the dependencies of an object. When we use constructor injection we are making explicit all required dependencies of that object. Based on the constructor is clear that an instance of `CreateClientService` can't exist without an instance of `ClientRepository`.

When do I need Spring for testing?

Sometimes, you'll need to do some unit tests relying on Spring framework. For example, if you have a repository that has a custom query using the `@Query` annotation, you might need to test your query. Also, if you are serialising/deserialising objects, you'd want to make sure that your object mapping is working. You might want to test your controllers as well, when you have some parameter validation or error handling. How can you be sure that you are using Spring correctly? In these situations you can take advantage of the new Spring Boot's test annotations.

Testing Spring Data repositories

If you look into the `ClientRepositoryTest` you'll see that we are using the annotation `@DataJpaTest`. Using this annotation we can have a `TestEntityManager` injected on the test, and use it to change the database to a state in which it's possible to unit test our

repository methods. In the code example, I'm testing a simple `findByName()` method (that is implemented by Spring and I don't really would test in a real life application). I'm using the injected entity manager to create a `Client`, and then I'm retrieving the created client using the repository.

```
@RunWith(SpringRunner.class)
@DataJpaTest
public class ClientRepositoryTest {

    @Autowired
    private TestEntityManager entityManager;

    @Autowired
    private ClientRepository clientRepository;

    @Test
    public void testFindByName() {
        entityManager.persist(new Client("Foo"));

        Optional<Client> client = clientRepository.findByName("Foo");
        assertEquals("Foo", client.get().getName());
    }
}
```

This kind of test is useful either when you have custom implemented repositories or when you are using `@Query` annotation to specify the queries fired agains the database.

Testing Spring MVC controllers

How many times have you made a mistake and mapped a wrong parameter in the controller? When was the last time that you tested all your bean validation to make sure that you weren't missing anything? With Spring Boot 1.4, now it's even simpler testing these controller specific responsibilities, you just need to use the `@WebMvcTest` annotation.

Take a look into the `ClientControllerTest` test class. I'm using `@WebMvcTest(ClientController.class)` to declare that this is a unit test for the `ClientController` class. Also, notice that I can use the `@MockBean` annotation to easily create a Mockito mock of my dependency `CreateClientService` class, and that mock will be injected automatically into my controller. Using these annotations you can mock your controller dependencies easily, and create isolated test that will be concerned only about your controller responsibilities.

Another good technique for unit testing controllers is to use a `MockMvc` instance. It's possible to dispatch requests to your controllers that will be processed as they will be in the web server. This makes test bean validations and url/parameters mappings easier.

```
@RunWith(SpringRunner.class)
public class ClientControllerTest {

    @Autowired
    MockMvc mockMvc;
```

```
    @MockBean
    CreateClientService createClientServiceMock;

    @Autowired
    ObjectMapper objectMapper;

    @Test
    public void testCreateClientSuccessfully() throws Exception {
        given(createClientServiceMock.createClient("Foo")).willReturn(new
Client("Foo"));

        mockMvc.perform(post("/clients")
            .contentType(MediaType.APPLICATION_JSON)
            .content(objectMapper.writeValueAsBytes(new
CreateClientRequest("Foo"))))
            .andExpect(status().isCreated())
            .andExpect(jsonPath("$.name", is("Foo")))
            .andExpect(jsonPath("$.number", notNullValue()));
    }
    ...
}
```

Integration Tests

Spring Boot offers a lot of support for writing integration tests for your application. First of all, you can use the new `@SpringBootTest` annotation. This annotation configures a complete test environment with all your beans and everything set up. You can choose to start a mock servlet environment, or even start a real one with the `webEnvironment` attribute of the `@SpringBootTest` annotation.

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class CreateClientIntegrationTest {

    @Autowired
    private TestRestTemplate restTemplate;

    @Test
    public void createClient() {
        ResponseEntity<Client> responseEntity =
            restTemplate.postForEntity("/clients", new
CreateClientRequest("Foo"), Client.class);
        Client client = responseEntity.getBody();

        assertEquals(HttpStatus.CREATED, responseEntity.getStatusCode());
        assertEquals("Foo", client.getName());
    }
}
```

In the example code above, we are starting a full Spring container with default configuration. The `TestRestTemplate` is configured automatically when you use the `@SpringBootTest` annotation, and is configured to resolve relative paths to

`http://localhost:${local.server.port}`.
When we execute the `createClient()` test method, we'll send a POST to the web server that will be routed to the `ClientController`. After that, the controller will call the `CreateClientService` to create the client. In the end, the service will use the `ClientRepository` to create the client in the database. We are testing our full application stack with a few configurations.

Conclusion

I hope that after reading this post you'll have a starting point for writing tests with Spring Boot. When writing a microservice, you want to be fast, and nothing is better than having a lot of test configurations and setup done by the framework, and not by you. This is just one of the things that makes me believe that currently, Spring Boot is the best choice for writing microservices in the Java ecosystem. Don't forget to read its documentation, it has a lot about testing with Spring Boot, what I've showed in this post is just the tip of the iceberg!

References

- Software Testing Levels - Tutorials Point
- Spring Boot Docs - Testing
- Testing improvements in Spring Boot 1.4

Lucas Saldanha

Passionate software developer. Constantly evolving. Always open to new ideas.