Spring Boot
Integration tests

Let's see how to integrate all the branches where we developed independently the features of our Spring Boot application, *demo-spring-testing*.

We'll do the integration in a separate branch, *integration*. In such a branch we merge another branch at a time. For each merge we'll probably have to fix some merge conflicts (due to changes made in both branches to the same classes and to temporary implementations of dependencies). After the merge, we'll write an integration test for the two components. After the integration tests succeed, we create a PR merging the *integration* branch into the *master* branch. (Remember that we have setup GitHub Actions, SonarCloud and Coveralls; we must make sure that code quality metrics are still satisfied before merging the PR) After the PR has been merged (of course, after reviewing the PR), we go back to the *integration* branch and we merge with another branch, and so on.

When we merge two branches, we expect the code coverage to reach 100%, since we get rid of fake implementations of dependencies (that were not covered by our unit tests).

Remember instead that integration tests are not meant to increase the code coverage. Indeed, the POM has been configured in order to record code coverage only during unit tests.

1 Service and Repository

We create a new branch "**integration**" starting from the branch "**service**" and we merge with "**persistence**".

We have an expected conflict with *EmployeeRepository*: in *service* it's a class with a fake temporary implementation, while in *persistence* it's the real repository declared as interface. Of course, we take the version from the branch *persistence*. (in Eclipse use the context menu "Replace with **Theirs**") There's no conflict on the *Employee* class since we modified that only in the *persistence* branch.

Before finalizing the branch, make sure that all tests still succeed (remember that the unit test for service is using Mockito, thus it should still succeed).

Let's write an integration test (this is just a demonstration; it might not make sense to write such assertions or, depending on the project, it might be worthwhile to verify specific scenarios of interactions):

```
package com.examples.spring.demo;

import org.springframework.context.annotation.Import;
...
/**
 * A possible integration test verifying that the service and repository
 * interact correctly.
 */
@RunWith(SpringRunner.class)
@DataJpaTest
@Import(EmployeeService.class)
public class EmployeeServiceRepositoryIT {

	@Autowired
	private EmployeeService employeeService;

	@Autowired
```

```
	private EmployeeRepository employeeRepository;

	@Test
	public void testServiceCanInsertIntoRepository() {
		Employee saved = employeeService
			.insertNewEmployee(new Employee(null, "an employee", 1000));
		employeeRepository.findById(saved.getId()).isPresent();
	}

	@Test
	public void testServiceCanUpdateRepository() {
		Employee saved = employeeRepository
			.save(new Employee(null, "an employee", 1000));
		Employee modified = employeeService.
			updateEmployeeById(saved.getId(),
				new Employee(saved.getId(), "modified", 2000));
		assertThat(employeeRepository.findById(saved.getId()))
			.contains(modified);
	}
}
```

This is *@DataJpaTest*, so that the testing database is autoconfigured; however, we need also the *EmployeeService*, which would not be injected (since the *@DataJpaTest* slice does not scan other beans, like services). Infact, we manually *@Import* it in the test case definition.

Create a PR and make sure that *integration* branch can be merged into master. When everything is fine, merge the PR. After merging, we don't remove the branch *integration*, since we'll keep on working on that branch for integrating other existing branches.

2 Integrate the REST controller

Continue working on the branch *integration*: we merge with the branch "**rest**".

We have two conflicts:

1. In the *rest* branch the *EmployeeService* was once again implemented with a fake temporary implementation; of course, we fix the conflict by keeping the version of *EmployeeService* of the current branch (*integration*) since it's the real one; (in Eclipse use the context menu "Replace with **Ours**")
2. The *Employee* has modifications in both branches; in the *rest* branch we had added a no-arg constructor; in the *integration* branch we have the modifications of the *persistence* branch for storing such objects in the database. Thus, once again, we keep the version of the current branch (*integration*).

Make sure everything still builds and tests are all green (both unit and integration tests).

Write an integration test with the rest controller running on a real web environment and use an injected repository to manually modify and query the database.

Since we need the persistence, service and controller layers altogether, we use *@SpringBootTest*. In particular, we use the argument *webEnvironment* and require Spring to start a real web server listening on a random port (to avoid conflicts with other running servers); the random port is then injected in the test with *@LocalServerPort*, and used to setup *RestAssured* (the real one, not the one for MockMvc):

```
package com.examples.spring.demo;

import static io.restassured.RestAssured.given;
import org.springframework.boot.test.context.SpringBootTest.WebEnvironment;
import org.springframework.boot.web.server.LocalServerPort;
...
/**
 * Some examples of tests for the rest controller when running in a real web
```

```
 * container, manually using the {@link EmployeeRepository}.
 *
 * The web server is started on a random port, which can be retrieved by
 * injecting in the test a {@link LocalServerPort}.
 *
 * In tests you can't rely on fixed identifiers: use the ones returned by the
 * repository after saving (automatically generated)
 */
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
public class EmployeeRestControllerIT {

	@Autowired
	private EmployeeRepository employeeRepository;

	@LocalServerPort
	private int port;

	@Before
	public void setup() {
		RestAssured.port = port;
		// always start with an empty database
		employeeRepository.deleteAll();
		employeeRepository.flush();
	}

	@Test
	public void testNewEmployee() throws Exception {
		// create an employee with POST
		Response response = given().
			contentType(MediaType.APPLICATION_JSON_VALUE).
			body(new Employee(null, "new employee", 1000)).
		when().
			post("/api/employees/new");

		Employee saved = response.getBody().as(Employee.class);

		// read it back from the repository
		assertThat(employeeRepository.findById(saved.getId()))
			.contains(saved);
	}

	@Test
	public void testUpdateEmployee() throws Exception {
		// create an employee with the repository
		Employee saved = employeeRepository
			.save(new Employee(null, "original name", 1000));

		// modify it with PUT
		given().
			contentType(MediaType.APPLICATION_JSON_VALUE).
			body(new Employee(null, "modified name", 2000)).
		when().
			put("/api/employees/update/" + saved.getId()).
		then().
			statusCode(200).
			body(
				// in the JSON response the id is an integer
				"id", equalTo(saved.getId().intValue()),
				"name", equalTo("modified name"),
				"salary", equalTo(2000)
			);
	}
}
```

Again, depending on your application, you should choose integration tests that make sense. Here we verify that if we create a new employee with POST, we can find it in the repository. Vice-versa, if we create an employee in the repository, we can modify it with PUT. We make sure that we always run each single test with an empty database. Note how we can reconstruct a Java object from a

JSON response obtained through REST Assured.

Remember that the ids are now automatically generated by the persistence layer; we can't assume that the previous ids are reused (actually they're always incremented), thus we do not hardcode in the tests the ids (like we used to do in unit tests).

Make sure it succeeds.

NOTE: SonarQube will complain about a critical vulnerability issue (rule java:S4684, https://rules.sonarsource.com/java/tag/spring/RSPEC-4684). This happens only at this time, since our REST controller handles *Employee* objects that are used as entities stored in a database (In the *rest* branch, *Employee* class was not using JPA annotations). As an exercise, try to understand that rule and why our code violates that. Also try to implement the suggested compliant solution. In this project, we simply disable that rule, but in your projects you must take it into consideration.

As usual, create PR, and after verification, merge it.

3 Integrate the Web controller

Continue working on the branch *integration*: we merge with the branch "**web**".

Fix merge conflicts in the *pom.xml*:

- use the dependency io.rest-assured:spring-mock-mvc from the current branch AND
- use net.sourceforge.htmlunit:htmlunit from the *web* branch

Fix merge conflict in *Employee* keeping the version of the current branch (*integration*).

Fix merge conflict in *EmployeeService* keeping the version of the current branch (*integration*), getting rid of the fake temporary implementation of the other branch.

Make sure everything still builds and tests are all green (both unit and integration tests).

Let's write an integration test for the Web controller.

Alternatively to HTMLUnit, we could leverage additional abstractions within *WebDriver*, https://docs.seleniumhq.org/projects/webdriver/, to make things even easier. WebDriver provides a very elegant API and allows us to easily organize our code. To better understand, let's explore an example.

NOTE: Despite being a part of Selenium, WebDriver does not require a Selenium Server to run your tests. In our tests, we won't even use a browser to test the HTML: we use the Selenium support for HTMLUnit.

First we need to add the corresponding dependencies (again, the versions are inherited by the Spring Boot test-starter):

```
<dependency>
  <groupId>org.seleniumhq.selenium</groupId>
  <artifactId>htmlunit-driver</artifactId>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.seleniumhq.selenium</groupId>
  <artifactId>selenium-support</artifactId>
  <scope>test</scope>
</dependency>
```

The main class is *WebDriver*. Spring Boot auto-configuration provides a WebDriver bean, that we can simply inject it in the test, without further configuration, when using MockMvc. In this test, however, we want to use a real server, like we did in the previous section. Thus, we'll have to create

the *HtmlUnitDriver* (an implementation of Selenium *WebDriver* using *HtmlUnit*) manually in a *@Before* method. We have already seen the *WebDriver* in the initial lesson on Java servlets.

In the following integration test we also inject the repository to add employees (used by the web interface) and to read employees (that should be inserted through the web interface). Thus, the setup for the tests and the assertions are done using the repository, while the *WebDriver* is used to interact with the web interface:

```
package com.examples.spring.demo;

import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.htmlunit.HtmlUnitDriver;
...
/**
 * Some examples of tests for the web controller when running in a real web
 * container, manually using the {@link EmployeeRepository}.
 *
 * The web server is started on a random port, which can be retrieved by
 * injecting in the test a {@link LocalServerPort}.
 *
 * In tests you can't rely on fixed identifiers: use the ones returned by the
 * repository after saving (automatically generated)
 */
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
public class EmployeeWebControllerIT {

	@Autowired
	private EmployeeRepository employeeRepository;

	@LocalServerPort
	private int port;

	private WebDriver driver;

	private String baseUrl;

	@Before
	public void setup() {
		baseUrl = "http://localhost:" + port;
		driver = new HtmlUnitDriver();
		// always start with an empty database
		employeeRepository.deleteAll();
		employeeRepository.flush();
	}

	@After
	public void teardown() {
		driver.quit();
	}

	@Test
	public void testHomePage() {
		Employee testEmployee =
			employeeRepository.save(new Employee(null, "test employee", 1000));

		driver.get(baseUrl);

		// the table shows the test employee
		assertThat(driver.findElement(By.id("employee_table")).getText()).
			contains("test employee", "1000", "Edit");

		// the "Edit" link is present with href containing /edit/{id}
		driver.findElement
			(By.cssSelector
				("a[href*='/edit/" + testEmployee.getId() + "']"));
	}

	@Test
```

```
	public void testEditPageNewEmployee() throws Exception {
		driver.get(baseUrl + "/new");

		driver.findElement(By.name("name")).sendKeys("new employee");
		driver.findElement(By.name("salary")).sendKeys("2000");
		driver.findElement(By.name("btn_submit")).click();

		assertThat(employeeRepository.findByName("new employee").getSalary())
			.isEqualTo(2000L);
	}

	@Test
	public void testEditPageUpdateEmployee() throws Exception {
		Employee testEmployee =
			employeeRepository.save(new Employee(null, "test employee", 1000));

		driver.get(baseUrl + "/edit/" + testEmployee.getId());

		final WebElement nameField = driver.findElement(By.name("name"));
		nameField.clear();
		nameField.sendKeys("modified employee");
		final WebElement salaryField = driver.findElement(By.name("salary"));
		salaryField.clear();
		salaryField.sendKeys("2000");
		driver.findElement(By.name("btn_submit")).click();

		assertThat(employeeRepository.findByName("modified employee").getSalary())
			.isEqualTo(2000L);
	}
}
```

The API of *WebDriver* should be straightforward to understand.

In the above tests we show a few scenarios that we verify. For example, we verify that the home page has a few elements, and that we can create and update employees through the *edit* page (using the “/new” and the “/edit/{id}” end-points, respectively).

4 Improvements

Explore and use the Selenium (WebDriver) *Page Object Pattern*.

Remember that SonarQube is scanning and analyzing also HTML files, according to a few HTML rules.