6 Lab: Additional tools for integration tests

v2025-03-20

6.0 Introduction

Learning objectives

- Apply the Test Containers framework to use managed instances of containerized dependencies in integration tests.
- Use REST Assured as an alternative approach to test REST API.

Key points

- To be fully confident with our application, we should write integration tests along with unit tests to ensure that our application is fully functional.
- The Test Containers framework makes it easy to "fire up" a container for the specific objective of running a test and then dispose it. The lifecycle of the container is managed by the test runner. A common use case is setting a *dockerized* database to run the test on a real server.
- Testcontainers is a testing library that provides easy and lightweight APIs for bootstrapping integration tests with real services wrapped in Docker containers. Using Testcontainers, you can write tests talking to the same type of services you use in production without mocks or in-memory services.

6.1 Test Containers in Spring Boot

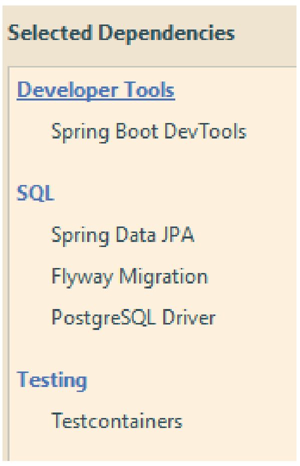
Test Containers (TC) can be integrated in test cases to prepare ephemeral docker containers required to run integration tests. A common scenario is to prepare a database server with a well-known state.

To use TC you need to have Docker support in your system.

TC are not specific to Spring Boot but there is good support available, which we will use in this example.

a) Create a simple **Spring Boot project** to manage a simple entity (e.g: Customer, or Employee, or Student,...) to be persisted into a PostgreSQL database.

   Take note of the selected dependencies for Spring Initialzr →



Selected Dependencies

Developer Tools
- Spring Boot DevTools

SQL
- Spring Data JPA
- Flyway Migration
- PostgreSQL Driver

Testing
- Testcontainers

b) Create the entity and the repository.

c) Create an integration test that prepares a PostgreSQL database, by inserting some content into the repository and them search to retrieve it.

   Notes:

   - you may learn from this code example[1]
   - you should ask Spring Boot to create the database schema, e.g.:

```
registry.add("spring.jpa.hibernate.ddl-auto", () -> "create-drop");
```

d) Develop your test scenario to include different test methods (e.g.: insert, retrieve, update and retrieve,...). Executing the test methods in a specific order is required. Consider using JUnit support for ordered tests.

6.2 Database migrations

Having code to insert data records for tests is not a practical or scalable approach.

A more convenient way would be using scripts that load data from a sample dataset.

e) Continue the previous project and create a Flyway initialization script in which you create the schema and add a few sample entries (example). Adapt the example; include insert statements to append few records into the database.

   Flyway uses "convention over configuration", meaning it expects to find your migrations under "**resources" folder**, with a specific path and file name convention, e.g.:

   [proj root]/test/**resources/db/migration/V001**__INIT.**sql**

f) Create a new test class, based on the example from the previous section, and ensure that:

   - Use @SpringBootTest
   - Remove the `spring.jpa.hibernate.ddl-auto` dynamic property setting.

   Run the test.

6.3 REST Assured

REST Assured can be used as a REST-client test library for Java integration testing, e.g.:

```
given().
        param("x", "y").
when().
        get("/lotto").
then().
        statusCode(400).body("lotto.lottoId", equalTo(6));
```

[1] There is a companion video available.

Consider that we want to "test" the behavior of the (fake) JSONPlaceholder API.

a) Create Java project with a test classes (no SpringBoot involved).

   Be sure to include the REST Assured dependencies and that you are using the static imports as presented in the documentation.

b) Create tests to verify that:

   - the endpoint to list all ToDos is available (status code 200)
   - when querying for ToDo #4, the API returns an object with title "et porro tempora"
   - When listing all "todos", you get **id** #198 and #199 in the JSON results.
   - When listing all "todos", you the results in less than 2secs.

Note: you may explore related examples from documentation or Baeldung.

6.4 Integration tests for Cars (with REST Assured)

Consider the integration tests developed in Lab 3 for the Cars (SpringBoot) project.

Create a new version of the RestController test, using REST Assured.

A related example is discussed here.

a) REST Assured has special support to work with Spring MockMvc (thus reducing the resources used in the test). Be sure to add the dependency for spring-mock-mvc:

```
<dependency>
      <groupId>io.rest-assured</groupId>
      <artifactId>spring-mock-mvc</artifactId>
      <scope>test</scope>
</dependency>
```

b) Create a new test class annotated with *@WebMvcTest(CarController.class)* to minimize contexts loading.

c) Since you are only loading the context for the controller, you should mock the required service implementation (e.g. *@MockBean CarManagerService service;*)

c) Use REST Assured with MockMvc to exercise the tests.

```
@Autowired private MockMvc mockMvc;
//…
RestAssuredMockMvc
        .given()
            .mockMvc( mockMvc)
        .when().get("/api/cars")
```

Note that, in this example, we are using `RestAssuredMockMvc` instead of `RestAssured.`

You do not need to provide a full server URL for the request since the MockMvc sliced environment is being used, not the full web environment.

6.5 Cars project with integration test

Let us create a different version of the full integration test (as in the last task in Lab 3), by using:

- RestAssured + Test Containers (for PostgresSQL provision) + Flyway migrations

Be sure to:

- launch the test in full web environment this time
- launch the database in a [test] container (e.g.: Postgres)
- use RestAssured to invoke the API.

Note:

You may use a database initialization library or not:

- if you use Flyway, for example, adapting to this new scenario, an initialization script must be included in the project.
- or you can rely on Spring Boot context loading tasks to initialize the database. Take note that additional configuration will be required as the database will be provised by Test Containers, e.g.: using annotation @TestPropertySource (to set spring.jpa.hibernate.ddl-auto) or adding a dynamic property (to set spring.jpa.hibernate.ddl-auto).