REST testing with Rest-Assured

In this tutorial we will go through the steps of testing a (very) simple application with an internal service which are exposed externally via a REST API.

We will use the code provided here as a starting point for the exercise.

https://github.com/Lars-m/resassuredEx1.git ()

It includes a class Calculator that provides the operations; add, sub, mul and div.

On top of this class is a REST API providing the following four services:

Calculator API	JSON result for OK's	JSON result for Error's

api/calculator/add/{n1}/{n2}
api/calculator/sub/{n1}/{n2}
api/calculator/mul/{n1}/{n2}
api/calculator/div/{n1}/{n2}
	<pre>{
	  "operation": "4 + 2",
	  "result": 6
	}</pre>
		<pre>{
		  "code": 500,
		  "message": "/ by zero"
		}</pre>

This is the API we are going to test.

Unit testing internal classes/methods

Before testing the REST API, let's test the Calculator class so we know that it "works", since our external API will never work, if this class doesn't.

Right click the Calculator class and use NetBeans to create a Unit Test for this test named Calculator**Test**.

*Note: You should name all your unitTest cases using the pattern xxxx**Test**.*

This will automatically add the JUnit-dependencies to our projects pom-file, verify this.

Implement traditional Unit-tests to verify that that the four math-operations works, and also verify the result for a divide by zero operation.

Testing our REST API (integration testing)

Testing a RESTful Service includes the following checks:

- That URL addresses are constituted correctly based on the service deployment end-point and the method annotations. (That you have made semantically sane URL's)
- That the generated server requests call the corresponding methods.
- That the API returns acceptable data. (That the data you want to examine in your PUT/GET/POST methods is valid and correct)
- That the API returns acceptable data also for erroneous scenarios

We have seen that we can test our API manually, through an application like Postman.

This can be fine as a quick start, but the end goal for us is to get test cases that are automated, meaning that they can be executed again and again, whenever we make changes to our code.

When testing the API, what we want to achieve is a combination of testing the individual methods as well as

a series of methods mimicking a real interaction with the service.

Rest Assured

Since a REST API is accessed via HTTP, we need a way to programmatically perform GET, POST, PUT, DELETE etc. up against the API. As mentioned above, we have done this manually using Postman, but we want tests that can execute automatically.

RESTAssured is such a simple Java library for programmatically performing requests up against REST services with focus on testing.

It simplifies testing by eliminating the need for boiler-plate code to test and validate complex responses. It also supports XML and **JSON** Request/Responses.

REST Assured supports the *POST*, *GET*, *PUT*, *DELETE*, *OPTIONS*, *PATCH* and *HEAD* http methods and lets us specify and validate parameters, headers, cookies and body easily.

REST Assured can be used together with JUnit.

Use this tutorial https://semaphoreci.com/community/tutorials/testing-rest-endpoints-using-rest-assured + http://rest-assured.io/ as a reference for REST Assured

Getting Started with Rest Assured

Add the following to your pom.xml file:

```
<dependency>
    <groupId>com.jayway.restassured</groupId>
    <artifactId>rest-assured</artifactId>
    <version>2.9.0</version>
</dependency>
```

Use NetBeans to create a new JUnit test named Service**IntegrationTest**

Note: You should name all your integrations Test cases using the pattern xxxx**IntegrationTest**.

In you test-class, add the following to your imports:

```
import io.restassured.RestAssured;
import static io.restassured.RestAssured.*;
import io.restassured.parsing.Parser;
import static org.hamcrest.Matchers.*;
```

This way the syntax of the framework gets cleaner to work with.

To use these imports add the following to your pom dependencies:

```
<dependency>
      <groupId>io.rest-assured</groupId>
      <artifactId>rest-assured</artifactId>
      <version>3.0.1</version>
      <scope>test</scope>
</dependency>
```

Next up, you want to specify your base URI (what host you are targeting), default Parser (how is

data sent to you, ie. JSON), and base path (what is the root of the REST API).

```
@BeforeClass
  public static void setUpBeforeAll() {
    RestAssured.baseURI = "http://localhost";
    RestAssured.port = 8080;
    RestAssured.basePath = "/Test1";
    RestAssured.defaultParser = Parser.JSON;
  }
```

This obviously puts a tough obligation on our system, in that the server must be running before the tests. We will see later that, using Maven, we can start an in memory version of Tomcat which will remove this obligation from our tests, but for now, just make sure the server is running before executing the unit tests.

For more information on what can be set globally, look at the javadocs

Writing REST Assured test cases

Test1 – verify that the server is running

Add the following test case to your test-file:

```
@Test
public void serverIsRunning() {
   given().
   when().get().
   then().
   statusCode(200);
}
```

This works because of the `setupBeforAll()` method given above. We could also have written it like this:

```
@Test
public void serverIsRunningV2() {
  given().when().get("http://localhost:8080/Test2/").then().statusCode(200);
}
```

This JUnit test connects to the root of our project, performs a *GET* call and makes sure that HTTP code200/success is returned. Notice the complete absence of the usual JUnit assert statements. REST Assured takes care of this for us and will automatically pass/fail this test according to the error code.

Refer to the documentation and slides to understand the purpose of the structure of the test.

Start your server, and execute the test to verify that the server is running

Test2 – verify that the add method works for OK scenarios

Add the following test case to your test-file:

```
@Test
public void addOperation() {
  given().pathParam("n1", 2).pathParam("n2", 2).
   when().get("/api/calculator/add/{n1}/{n2}").
   then().
   statusCode(200).
   body("result", equalTo(4), "operation", equalTo("2 + 2"));
```

```
}
```

This test connects to the *add* service, performs a *GET* call, verifies that the HTTP code 200 is returned and then verifies whether the correct JSON is returned, i.e.: `{"operation": "2+ 2","result": 4}`

Observe how the object returned by `given()`, add `pathParams()` to the object and uses them in the `get(..)` method.
Also observe how we can access the returned JSON elements (*result* and *operation*) in the `body(..)` method. This works because we have set the `defaultParser` to JSON in the initial setup method.
The example above used the pathParam(..) methods to show how given() is used to pass parameter with the request. The test could also have been written like:

```
@Test
public void addOperationv2() {
  given().
   when().get("/api/calculator/add/2/2").
   then().
   statusCode(200).
   body("result", equalTo(4), "operation", equalTo("2 + 2"));
}
```

When we start testing POST, PUT methods however, we need to use the `pathParam()` method to pass in arguments to the server. Also, if we need to set http-headers (Content-type, security tokens etc.) then we do it via the `given()` method.

<u>The remaining OK tests</u>

Implement test cases similar to the one above to test the remain operations (sub, mul and div)

<u>The remaining NOT-OK tests</u>

Implement the test cases to verify the correct behaviour (status code and JSON) for:

- A request for a non-existing route
- Use of illegal parameters (remember, this simple calculator only handles integers)
- Division by zero

Use this example as inspiration for how to implement the test requested above.

```
@Test
public void addOperationWrongArguments() {
 given().pathParam("n1", 2).pathParam("n2", 2.2).
  when().get("/api/calculator/add/{n1}/{n2}").
  then().
  statusCode(400).
  body("code", equalTo(400));
}
```

Maven and Testing

The maven Build Lifecycle https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html

Maven is based around the concept of a build life cycle. There are three build in lifecycles but the only one we will focus on here is the *default* lifecycle. The default lifecycle handles project deployment and is defined by a list of build phases wherein a build phase represents a stage in the lifecycle.

The default lifecycle is comprised of the following phases:

- **validate** - validate the project is correct and all necessary information is available
- **compile** - compile the source code of the project
- **test** - test the compiled source code using a suitable unit testing framework. *These tests should not require the code be packaged or deployed*
- **package** - take the compiled code and package it in its distributable format, such as a JAR.
- **verify** - run any checks on results of integration tests to ensure quality criteria are met
- **install** - install the package into the local repository, for use as a dependency in other projects locally
- **deploy** - done in the build environment, copies the final package to the remote repository for sharing with other developers and projects.

You can execute each of these phases via a terminal and: navigate to the root of your project and type:

```
mvn <phase name>
```

Each phase will always execute all the phases that comes before it, so if you type `mvn package` it will run the; *validate*, *compile*, *test* and *package* phases.

Maven and UnitTests

Maven (and NetBeans with a maven project) always runs your test cases as part of a build cycle. This means you will see your tests being executed each time you do a clean-and-build, and when you try to run the project.

It also STOPS the execution if your unit tests fails. This makes sense for normal unit tests, since a red test should be considered just as serious as a compile error. It does however create a big problem if we write test cases that breaks the rule marked in red above.

And this is exactly what you did when you wrote your tests in the section "Writing REST Assured test cases".

These tests require that our project is packaged, deployed and even started in order to execute. Big problem, your test requires the project to be build, deployed and started, but you cannot deploy and start because your test won't run.

The problems is that our REST assured tests are integration tests and not unit test and actually should be executed in the *verify* phase and not the *test* phase.

We will solve this problem in the next section, but for now, in order to complete your REST assured test you should do the following:

Add this entry to the plugin section of your pom-file:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>2.12.1</version>
    <configuration>
       <excludes>
          <exclude>**/*IntegrationTest*</exclude>
       </excludes>
    </configuration>
</plugin>
```

The surfire-plugin is actually the part that executes maven tests, and if you have followed the naming conventions given in the exercise, the exclude section will leave out all your integration tests so you can build, package, deploy and run your project.

Once deployed you can comment out the exclude section `<!--<exclude>**/*IntegrationTest*</exclude>-->` and run your tests again. This time it should also run your integration tests.

Following this strategy you can comment this section in and out while you develop, which is cumbersome but will work for this exercise, or until you have read the next section.

Maven and Integration Tests

As explained above integration test (tests that requires code to be packaged, deployed and perhaps even run) does not belong in mavens test phase.

Maven supply a specific plugin for integration tests (for the verify phase) which is the Failsafe plugin.

http://maven.apache.org/surefire/maven-failsafe-plugin/

Add this plugin to the plugins section of your pom-file.

```
<plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-failsafe-plugin</artifactId>
   <version>2.12.4</version>
   <configuration>
     <includes>
       <include>**/*IntegrationTest*</include>
     </includes>
   </configuration>
   <executions>
     <execution>
       <goals>
         <goal>integration-test</goal>
         <goal>verify</goal>
       </goals>
     </execution>
   </executions>
</plugin>
```

It will include all your integration tests in this phase (which are being excluded from your unit-tests) and is executed by typing `mvn verify`.

Your tests still require (obviously) that the server is running, but if they fail, tests in this phase will NOT stop the lifecycle tasks. This is because this phase actually has a two additional phases, pre-integration-test and post-integration-test, which can be used to start and stop a server. We will not use this feature in this exercise, but since these tests does not stop execution, we will always reach the post-integration-test phase so a test server can be stopped, evenwhen a one or more tests are failing.

I have not succeeded in running this part from inside NetBeans, so execute your tests from a terminal: navigate to the root of your project and type: `mvn verify`

Starting a test server with Maven (If you want more)

If you want to try an example where maven will start and embedded tomcat for your tests, add this to your pom-file:

```
<plugin>
   <groupId>org.apache.tomcat.maven</groupId>
   <artifactId>tomcat7-maven-plugin</artifactId>
   <version>2.2</version>
   <configuration>
     <path>/</path>
   </configuration>
   <executions>
     <execution>
       <id>start-tomcat</id>
       <phase>pre-integration-test</phase>
       <goals>
         <goal>run</goal>
       </goals>
       <configuration>
         <fork>true</fork>
       </configuration>
     </execution>
     <execution>
       <id>stop-tomcat</id>
       <phase>post-integration-test</phase>
       <goals>
         <goal>shutdown</goal>
       </goals>
     </execution>
   </executions>
</plugin>
```

It will start an embedded tomcat server (running default on port 8080) before your integration test and stop it again after. This will require you to change the port number in your tests to 8080 in order to execute the tests. This is an extremely cool feature, since it removes the obligation on us, to always manually ensure the server is running before the tests.