Testing REST services and APIs in Java using REST Assured and JUnit

- Overview
  - REST Assured concepts
- Description
- Tips
- References

Overview

Services are usually acessed using APIs. REST APIs are common both in internal services and in public facing services.

Whenever testing REST APIs, we can either be targetting the client/consumer or the server/producer.

In this tutorial we'll show how a REST API can be tested using Java and the REST Assured library, together with JUnit and WireMock.

REST Assured is a Java library that implements a DSL for easily validating REST API based services by abstracting low-level details of HTTP requests and of parsing responses.

It provides a syntax based on Gherkin (i.e. given, when, then) for better readibility.

From REST Assured documentation, imagine that you have a REST based service accessibile in the `http://localhost:8080/lotto/{id}` endpoint, and returning a JSON content.

```
{
   "lotto":{
      "lottoId":5,
      "winning-numbers":[2,45,34,23,7,5,3],
      "winners":[
         {
            "winnerId":23,
            "numbers":[2,45,34,23,3,5]
         },
         {
            "winnerId":54,
            "numbers":[52,3,12,11,18,22]
         }
      ]
   }
}
```

A JUnit test to check that the HTTP response code is OK (i.e. 200) and that the "lotto" object returned is correct, could be something as follows.

```
@Test public void
lotto_resource_returns_200_with_expected_id_and_winners() {

    when().
            get("/lotto/{id}", 5).
    then().
            statusCode(200).
            body("lotto.lottoId", equalTo(5),
                 "lotto.winners.winnerId", hasItems(23, 54));

}
```

This takes advantage of the JsonPath, a simple JSON parser (there's also an equivalent for XML: XmlPath).

If required, it's also possible to perform JSON Schema validation.

REST Assured concepts

In REST Assured it's all about being able to quickly perform HTTP REST requests and to verify the response easily, no matter if it's JSON or XML based.

Main concepts include:

- **request specification -** the definition of the request (e.g. URL, authentication, etc); the object **RequestSpecification** is usually specified using *given()* and *when()*.
- **response specification -** the definition of the response (e.g. body, status code); the object **ResponseSpecification** is usually specified using *then()*
- **parser -** an entity responsible for processing and abstract the body content of the response, so that data can easily be referred in the matchers /assertions
- **matchers -** used to match either built-in ones (io.restassured.matcher.RestAssuredMatchers.*) or from Hamcrest (org.hamcrest.Matchers.*).

Note: it's possible to reuse request specifications or response specifications between multiple requests/responses for different tests (more info here). This can be extremely useful if you need to perform a set of common assertions (e.g. status code being OK) for a bunch of tests.

Description

In this tutorial, we'll use the code from Bas Dijkstra open-source workshop on REST Assured; you may want to check the workshop presentation for further detail.

We aim to test a service that provides an API (i.e. http://api.zippopotam.us/) to obtain location data based on postal/zip codes on several countries.

> **Please note**
>
> To be clear, in this project tests are performed against the server provided by WireMock that mocks responses from the API; this was done just be convenience of the workshop, so it's not dependent on the public server availability. In a real scenario, you would perform tests against the target service or you use WireMock to validate your client/consumer library implementaiton.

The workshop provides some exercises, stored under src/test/java/exercises; it also provides the answers.

As an example, lets implement a test for checking that the API returns the correct state, in the correct order, for a given zip code of US.

example of a test from RestAssuredAnswers1Test.java

```
        /*******************************************
         * Send a GET request to /us/90210 and check
         * that the state associated with the first place
         * in the list returned is equal to 'California'
         *
         * Use the GPath expression "places[0].state" to
         * extract the required response body element
         *******************************************/

        @Test
        public void requestUsZipCode90210_checkStateForFirstPlace_expectCalifornia() {

                given().
                        spec(requestSpec).
                when().
                        get("/us/90210").
                then().
                        assertThat().
                        body("places[0].state", equalTo("California"));
        }
```

Another variant of it could be performing data-driven testing to check the returned "state" for a set of input data (e.g. country+zip code).

code snippet from RestAssuredAnswers2Test.java

```
..
        /*****************************************************
         * Create a DataProvider with three test data rows:
         * -----------------------------------
         * country code | zip code | state
         * -----------------------------------
         * us           | 90210    | California
         * us           | 12345    | New York
         * ca           | Y1A      | Yukon
         *****************************************************/

        @DataProvider
        public static Object[][] zipCodeData() {
                return new Object[][] {
                                { "us", "90210", "California" },
                                { "us", "12345", "New York" },
                                { "ca", "Y1A", "Yukon" }
                };
        }

        /*****************************************************
         * Request zip code data for the given country / zip
         * combinations by sending a GET to /<countryCode>/<zipCode>.
         *
         * Use the test data collection created
         * above. Check that the state returned by the API
         * matches the expected value.
         *
         * Use the GPath expression "places[0].state" to
         * extract the required response body element
         *****************************************************/

        @Test
        @UseDataProvider("zipCodeData")
        public void checkStateForCountryCodeAndZipCode(String countryCode, String zipCode, String
expectedState) {

                given().
                        spec(requestSpec).
                and().
                        pathParam("countryCode", countryCode).
                        pathParam("zipCode", zipCode).
                when().
                        get("/{countryCode}/{zipCode}").
                then().
                        assertThat().
                        body("places[0].state",equalTo(expectedState));
        }

...
```

We'll see ahead how this specific data-driven test will be mapped to Jira.

Tests can be run using Maven; in this case, we'll be only executing the ones inside the "answers" package.

example of a Bash script to run the tests

```
mvn clean compile test -Dtest=answers.*
```

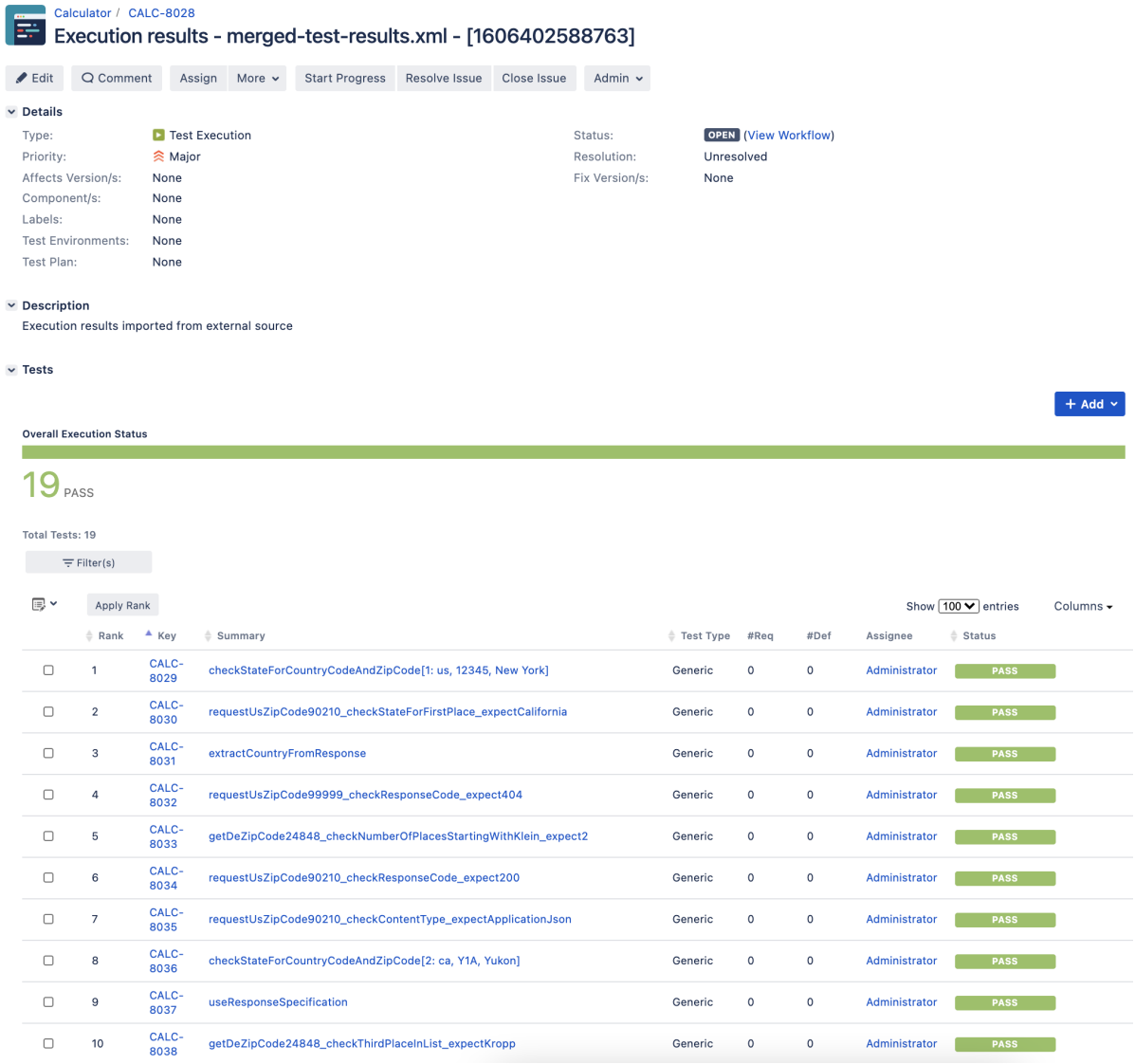
Since the previous command generates multiple JUnit XML files, we may need to merge them into a single XML file so it can be submitted into a Test Execution more easily. That can be achieved by using the junit-merge utility.

```
junit-merge -d target/surefire-reports/ -o merged-test-results.xml
```

After successfully running the tests and generating the aggregated JUnit XML report (e.g., merged-test-results.xml), it can be imported to Xray (either by the REST API or through the **Import Execution Results** action within the Test Execution, or even by using a CI tool of your choice).

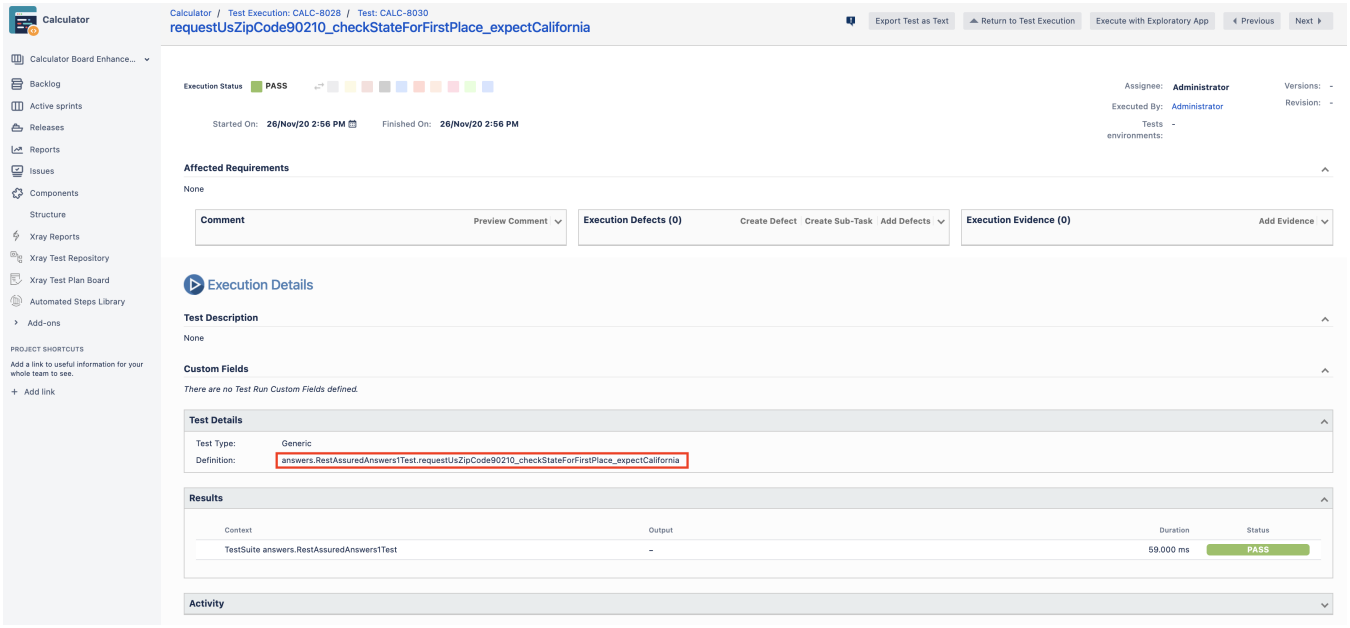
example of a Bash script to import results

```
curl -H "Content-Type: multipart/form-data" -u admin:admin -F "file=@./merged-test-results.xml"
http://jiraserver.example.com/rest/raven/1.0/import/execution/junit?projectKey=CALC
```



Each JUnit's test is mapped to a Generic Test in Jira, and the **Generic Test Definition** field contains the name of the package, the class and the method name that implements the Test Case. The summary of each Test issue is filled out with the name of the method corresponding to the JUnit Test.

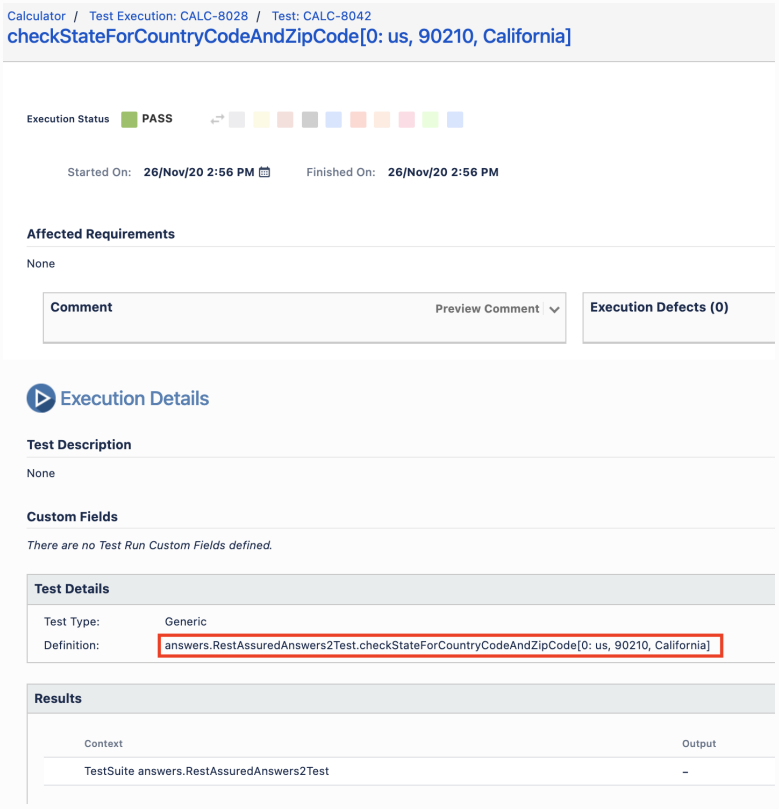
The Execution Details page also shows information about the Test Suite, which in this case corresponds to the Test class, including its namespace.


Calculator
Calculator / Test Execution: CALC-8028 / Test: CALC-8030
requestUsZipCode90210_checkStateForFirstPlace_expectCalifornia
Export Test as Text
Return to Test Execution
Execute with Exploratory App
Previous
Next
Calculator Board Enhance...
Backlog
Active sprints
Releases
Reports
Issues
Components
Structure
Xray Reports
Xray Test Repository
Xray Test Plan Board
Automated Steps Library
Add-ons
PROJECT SHORTCUTS
Add a link to useful information for your whole team to see.
Add link
Execution Status PASS
Started On: 26/Nov/20 2:56 PM
Finished On: 26/Nov/20 2:56 PM
Assignee: Administrator
Executed By: Administrator
Tests environments: -
Versions: -
Revision: -
Affected Requirements
None
Comment
Preview Comment
Execution Defects (0)
Create Defect
Create Sub-Task
Add Defects
Execution Evidence (0)
Add Evidence
Execution Details
Test Description
None
Custom Fields
There are no Test Run Custom Fields defined.
Test Details
Test Type: Generic
Definition: answers.RestAssuredAnswers1Test.requestUsZipCode90210_checkStateForFirstPlace_expectCalifornia
Results
Context
Output
Duration
Status
TestSuite answers.RestAssuredAnswers1Test
-
59.000 ms
PASS
Activity

Please note

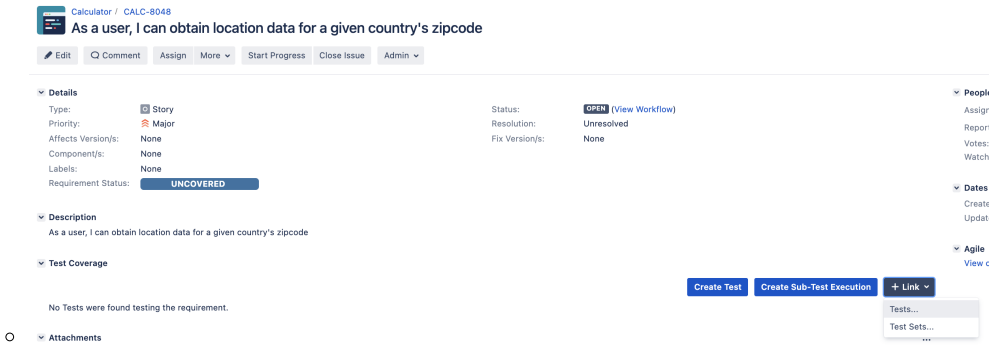
In the case of data-driven tests, as the JUnit XML report treats each row of the data provider as a different "test case", then you'll end up with several Tests in Xray.

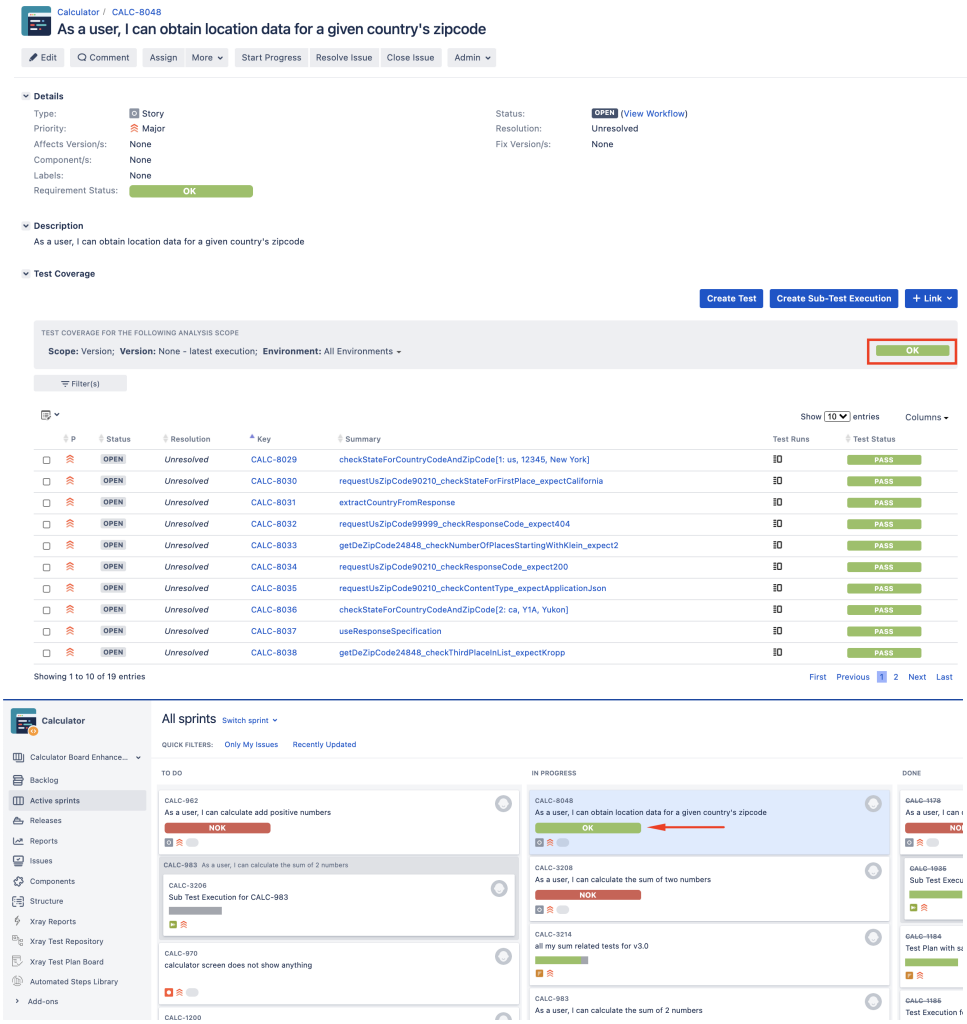
	Rank	Key	Summary	Test Type	#Req	#Def	Assignee	Status
☐	14	CALC-8042	checkStateForCountryCodeAndZipCode[0: us, 90210, California]	Generic	0	0	Administrator	PASS
☐	1	CALC-8029	checkStateForCountryCodeAndZipCode[1: us, 12345, New York]	Generic	0	0	Administrator	PASS
☐	8	CALC-8036	checkStateForCountryCodeAndZipCode[2: ca, Y1A, Yukon]	Generic	0	0	Administrator	PASS



Tips

- Multiple runs of your tests can be grouped and consolidate in a Test Plan, so you can have an updated overview of their current state
- After importing the results, you can link the corresponding Test issues with an existing requirement or user story and thus truck coverage directly on the respective issue, or even on a Agile board





References

- https://rest-assured.io/
- https://github.com/rest-assured/rest-assured
- REST Assured Usage Guide
- Bas Dijkstra open-source workshop on REST Assured
- Bas Dijkstra blog entry about WireMock
- WireMock