**How to perform API testing with REST Assured**

Now that APIs are playing an ever more important role in software trends (such as mobile applications, the Internet of Things, etc.), proper automated testing of these APIs is becoming indispensable. There are many different tools out there that can assist you in writing these automated tests at the API level. I'm going to show you how to use one of the most popular open-source tools for this task: [REST Assured](http://rest-assured.io/).

REST Assured is a Java library that provides a domain-specific language (DSL) for writing powerful, maintainable tests for RESTful APIs. In the following sections, I'll show you how to set up and configure REST Assured, write and run REST Assured tests, and apply some of its most powerful features. I'll be using real-world code examples you can copy, run, and reuse directly in your own test automation efforts.

**Getting started: Configuration**

To get started with REST Assured, simply add it as a dependency to your project. If you're using [Maven](https://maven.apache.org/what-is-maven.html), add the following entry to your pom.xml (change the version number to reflect the version you want to use):

<dependency>  
    <groupId>io.rest-assured</groupId>  
    <artifactId>rest-assured</artifactId>  
    <version>3.0.2</version>  
    <scope>test</scope>  
</dependency>

And for [Gradle](https://gradle.org/):

testCompile 'io.rest-assured:rest-assured:3.0.2'

REST Assured can be used easily in combination with existing unit testing frameworks, such as JUnit and TestNG. For the examples presented in this tutorial, I used REST Assured with TestNG.

Once you have the import of REST Assured set up, add the following static imports to your test class:

import static io.restassured.RestAssured.\*;  
import static org.hamcrest.Matchers.\*;

and you're ready to create your first REST Assured test.

**First test: Understanding the syntax**

For this tutorial, we'll test the Ergast Motor Racing Database API, which can be found [here](http://ergast.com/mrd/). This API provides historical data related to Formula 1 races, drivers, circuits, and much more.

To illustrate the way tests are written using REST Assured, here is a test that retrieves the list of circuits for the 2017 Formula 1 season in JSON format and checks that there are 20 circuits in the list:

@Test  
public void test\_NumberOfCircuitsFor2017Season\_ShouldBe20() {  
          
    given().  
    when().  
        get("http://ergast.com/api/f1/2017/circuits.json").  
    then().  
        assertThat().  
        body("MRData.CircuitTable.Circuits.circuitId",hasSize(20));  
}

Note that the fluent API used by REST Assured supports the familiar Given/When/Then syntax from [behavior-driven development](https://en.wikipedia.org/wiki/Behavior-driven_development) (BDD), resulting in a test that is easy to read and takes care of everything (setup, execution, and verification) with just a single line of code.

The *hasSize()* Hamcrest matcher counts the number of circuits—that's why you needed to add Hamcrest as a static import. The Hamcrest library contains a collection of matchers that allow you to create verifications of all kinds while keeping them readable.

The verification part of the test does the following:

1. Captures the (JSON) response of the API call
2. Queries for all elements called circuitId using the Groovy GPath expression *"MRData.CircuitTable.Circuits.circuitId"*
3. Verifies (using the aforementioned Hamcrest matcher) that the resulting collection of circuitId elements has size 20

There are Hamcrest matchers for a large number of different checks, including *equalTo()*for equality, *lessThan()* and *greaterThan()* for comparison, *hasItem()* to check whether a collection contains a given element, and many more. Reference the [Hamcrest library documentation](http://hamcrest.org/JavaHamcrest/)for a full list of matchers.

To run the tests, just use the test runner associated with the unit testing framework of your choice (i.e., JUnit or TestNG).

**Validating technical response data**

With REST Assured, you can not only verify response body contents, but also check the correctness of technical response data, such as the HTTP response status code, the response content type, and other response headers. The example below checks that:

* The response status code is equal to 200.
* The response content type (telling the receiver of the response how to interpret the response body) equals "application/json."
* The value of the response header "Content-Length" equals "4567."

@Test  
public void test\_ResponseHeaderData\_ShouldBeCorrect() {  
          
    given().  
    when().  
        get("http://ergast.com/api/f1/2017/circuits.json").  
    then().  
        assertThat().  
        statusCode(200).  
    and().  
        contentType(ContentType.JSON).  
    and().  
        header("Content-Length",equalTo("4567"));  
}This example also shows how you can easily concatenate checks in a readable manner using the *and()*method, which is only syntactic sugar, meaning that making the code more readable is all that it does.

**Parameterizing tests**

Often, you'll want to repeat the same test with various sets of (input and output) parameters—a concept known as data-driven testing. Instead of writing a new test for each test data record, you'll want to create a parameterized test and feed it with as many test data records as your desired test coverage requires.

RESTful APIs support two different types of parameters:

* **Query parameters:** These are appended at the end of a RESTful API endpoint and can be identified by the question mark in front of them. For example, in the endpoint http://md5.jsontest.com/?text=test, "text" is a query parameter (with value "test").
* **Path parameters:** These are part of the RESTful API endpoint. For example, in the endpoint we used earlier: http://ergast.com/api/f1/2017/circuits.json, "2017" is a path parameter value. Try and replace it with "2016" and see what happens (hint: the previous test should fail, since there were 21 Formula 1 races in 2016, not 20).

REST Assured can work with both types of parameters. First, let's see how you can specify and use the query parameter from the example above:

@Test  
public void test\_Md5CheckSumForTest\_ShouldBe098f6bcd4621d373cade4e832627b4f6() {  
      
    String originalText = "test";  
    String expectedMd5CheckSum = "098f6bcd4621d373cade4e832627b4f6";  
          
    given().  
        param("text",originalText).  
    when().  
        get("http://md5.jsontest.com").  
    then().  
        assertThat().  
        body("md5",equalTo(expectedMd5CheckSum));  
}

As you can see, using query parameters in REST Assured is as easy as specifying their name and value using the *param()*method. Path parameters are specified in a similar fashion:

@Test  
public void test\_NumberOfCircuits\_ShouldBe20\_Parameterized() {  
          
    String season = "2017";  
    int numberOfRaces = 20;  
          
    given().  
        pathParam("raceSeason",season).  
    when().  
        get("http://ergast.com/api/f1/{raceSeason}/circuits.json").  
    then().  
        assertThat().  
        body("MRData.CircuitTable.Circuits.circuitId",hasSize(numberOfRaces));  
}

Instead of *param()*, path parameters are defined using the *pathParam()* method. In addition, you'll need to define which part of the endpoint path represents the path variable, which is done using the curly bracket notation seen in the example above. You can easily create more than a single path parameter in the same way, and even combine both path and query parameters in a single call if the API supports or requires this.

Now that we have created a parameterized test, making it data-driven by using an external test data set is a straightforward task. As I said in the beginning of this tutorial, this is easiest when done with TestNG, but JUnit supports parameterization as well. With TestNG, all you need to do is create a *DataProvider* object containing the required test data—in this case, a set of records containing years and the number of Formula 1 races in each year:

@DataProvider(name="seasonsAndNumberOfRaces")  
public Object[][] createTestDataRecords() {  
    return new Object[][] {  
        {"2017",20},  
        {"2016",21},  
        {"1966",9}  
    };  
}

Then you just pass the test data object to the parameterized test through test method parameters:

@Test(dataProvider="seasonsAndNumberOfRaces")  
public void test\_NumberOfCircuits\_ShouldBe\_DataDriven(String season, int numberOfRaces) {  
                  
    given().  
        pathParam("raceSeason",season).  
    when().  
        get("http://ergast.com/api/f1/{raceSeason}/circuits.json").  
    then().  
        assertThat().  
        body("MRData.CircuitTable.Circuits.circuitId",hasSize(numberOfRaces));  
}

When you have a test data object, you can add, remove, or update individual test cases simply by creating, deleting, or modifying the appropriate test data record. If the test itself needs to be updated—for example when the endpoint or the response body structure changes—all you need to do is to update the test and all test cases will follow the updated procedure.

**Accessing secured APIs**

Often, APIs are secured using some sort of authentication mechanism. REST Assured supports basic, digest, form, and OAuth authentication. Here's an example of how to call a RESTful API that has been secured using basic authentication (i.e., the consumer of this API needs to provide a valid username and password combination every time they call the API):

@Test  
public void test\_APIWithBasicAuthentication\_ShouldBeGivenAccess() {  
          
    given().  
        auth().  
        preemptive().  
        basic("username", "password").  
    when().  
        get("http://path.to/basic/secured/api").  
    then().  
        assertThat().  
        statusCode(200);  
}

Accessing an OAuth2-secured API is just as straightforward, assuming you have a valid authentication token:

@Test  
public void test\_APIWithOAuth2Authentication\_ShouldBeGivenAccess() {  
          
    given().  
        auth().  
        oauth2(YOUR\_AUTHENTICATION\_TOKEN\_GOES\_HERE).  
    when().  
        get("http://path.to/oath2/secured/api").  
    then().  
        assertThat().  
        statusCode(200);  
}

**Passing parameters between tests**

Often, when testing RESTful APIs, you might need to create more complex test scenarios where you'll need to capture a value from the response of one API call and reuse it in a subsequent call. This is supported by REST Assured using the *extract()*method. As an example, here's a test scenario that extracts the ID for the first circuit of the 2017 Formula 1 season and uses it to retrieve and verify additional information on that circuit (in this case, the circuit is located in Australia):

@Test  
public void test\_ScenarioRetrieveFirstCircuitFor2017SeasonAndGetCountry\_ShouldBeAustralia() {  
          
    // First, retrieve the circuit ID for the first circuit of the 2017 season  
    String circuitId = given().  
    when().  
        get("http://ergast.com/api/f1/2017/circuits.json").  
    then().  
        extract().  
        path("MRData.CircuitTable.Circuits.circuitId[0]");  
          
    // Then, retrieve the information known for that circuit and verify it is located in Australia  
    given().  
        pathParam("circuitId",circuitId).  
    when().  
        get("http://ergast.com/api/f1/circuits/{circuitId}.json").  
    then().  
        assertThat().  
        body("MRData.CircuitTable.Circuits.Location[0].country",equalTo("Australia"));  
}

**Reusing checks with ResponseSpecBuilder**

Another way to improve the reusability and maintainability of your RESTful API tests is by reusing specific checks. For example, if you want to check that all your API responses have a status code equal to 200 and a content type equal to "application/json," specifying this for each and every test can get tiring quickly.

Additionally, if for some reason the default status code and content type returned changes, wouldn't it a be great to only have to update this in one place, instead of throughout your test suite? REST Assured supports the reuse of specific verifications using the ResponseSpecBuilder mechanism.

Here's an example of how to create a reusable ResponseSpecification that checks the aforementioned status code and content type, as well how to use it in a test:

ResponseSpecification checkStatusCodeAndContentType =   
    new ResponseSpecBuilder().  
        expectStatusCode(200).  
        expectContentType(ContentType.JSON).  
        build();

@Test  
public void test\_NumberOfCircuits\_ShouldBe20\_UsingResponseSpec() {  
          
    given().  
    when().  
        get("http://ergast.com/api/f1/2017/circuits.json").  
    then().  
        assertThat().  
        spec(checkStatusCodeAndContentType).  
    and().  
        body("MRData.CircuitTable.Circuits.circuitId",hasSize(20));  
}

You can create checks that are specific to that test in addition to using the ResponseSpecification.

**Other interesting REST Assured features**

Apart from those introduced above, REST Assured offers a number of other useful features that let you create even more powerful tests, such as:

* The ability to (de-)serialize Plain Old Java Objects (POJOs). This allows you to serialize the properties and values associated with a Java object instance directly into a JSON or an XML document, which can then be sent to a RESTful API using the POST method. This also works the other way around—a JSON or an XML response returned by an API can be deserialized into a POJO instance by REST Assured as well.
* Logging requests and responses. This can be especially useful when you need to inspect API responses to create the appropriate checks, or when you want to make sure that the request you're sending to an API is correct. You can choose to log everything (parameters, headers, and body), only headers, only parameters, and much more.
* REST Assured comes with a Spring Mock MVC module, allowing you to write tests for Spring controllers using the REST Assured syntax.

**Resources for further learning**

If you want to learn more about REST Assured, take a look at the [REST Assured usage guide on GitHub](https://github.com/rest-assured/rest-assured/wiki/Usage).

If you want to try out REST Assured for yourself, I have created and published a [REST Assured workshop](https://github.com/basdijkstra/workshops/tree/master/rest-assured), including slides, examples, exercises and their answers.