**Simple Example Test**

Before we get started, let’s ensure that our tests have the following static imports:

|  |  |
| --- | --- |
| 1  2  3 | io.restassured.RestAssured.\*  io.restassured.matcher.RestAssuredMatchers.\*  org.hamcrest.Matchers.\* |

We’ll need that to keep tests simple and have easy access to the main APIs.

Now, let’s get started with the simple example – a basic betting system exposing some data for games:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | {      "id": "390",      "data": {          "leagueId": 35,          "homeTeam": "Norway",          "visitingTeam": "England",      },      "odds": [{          "price": "1.30",          "name": "1"      },      {          "price": "5.25",          "name": "X"      }]  } |

Let’s say that this is the JSON response from hitting the locally deployed API – *http://localhost:8080/events?id=390.*:

Let’s now use REST-assured to verify some interesting features of the response JSON:

|  |  |
| --- | --- |
| 1  2  3  4  5 | @Test  public void givenUrl\_whenSuccessOnGetsResponseAndJsonHasRequiredKV\_thenCorrect() {     get("/events?id=390").then().statusCode(200).assertThat()        .body("data.leagueId", equalTo(35));  } |

So, what we did here is – we verified that a call to the endpoint */events?id=390* responds with a body containing a *JSON String* whose *leagueId*of the *data* object is 35.

Let’s have a look at a more interesting example. Let’s say you would like to verify that the *odds* array has records with prices *1.30* and *5.25*:

|  |  |
| --- | --- |
| 1  2  3  4  5 | @Test  public void givenUrl\_whenJsonResponseHasArrayWithGivenValuesUnderKey\_thenCorrect() {     get("/events?id=390").then().assertThat()        .body("odds.price", hasItems("1.30", "5.25"));  } |

**3. REST-assured Setup**

If your favorite dependency tool is Maven, we add the following dependency in the *pom.xml* file:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | <dependency>      <groupId>io.rest-assured</groupId>      <artifactId>rest-assured</artifactId>      <version>3.0.0</version>      <scope>test</scope>  </dependency> |

To get the latest version, follow [this link](https://search.maven.org/classic/#search%7Cga%7C1%7Ca%3A%22rest-assured%22).  
REST-assured takes advantage of the power of Hamcrest matchers to perform its assertions, so we must include that dependency as well:

|  |  |
| --- | --- |
| 1  2  3  4  5 | <dependency>      <groupId>org.hamcrest</groupId>      <artifactId>hamcrest-all</artifactId>      <version>1.3</version>  </dependency> |

The latest version will always be available at [this link](https://search.maven.org/classic/#search%7Cga%7C1%7Ca%3A%22hamcrest-all%22).

**4. Anonymous JSON Root Validation**

Consider an array that comprises of primitives rather than objects:

|  |  |
| --- | --- |
| 1 | [1, 2, 3] |

This is called an anonymous JSON root, meaning that it has no key-value pair nevertheless it is still valid JSON data.

We can run validation in such a scenario by using the *$* symbol or an empty String ( “” ) as path. Assume we expose the above service through *http://localhost:8080/json* then we can validate it like this with REST-assured:

|  |  |
| --- | --- |
| 1 | when().get("/json").then().body("$", hasItems(1, 2, 3)); |

or like this:

|  |  |
| --- | --- |
| 1 | when().get("/json").then().body("", hasItems(1, 2, 3)); |

**5. Floats and Doubles**

When we start using REST-assured to test our REST services, we need to understand that floating point numbers in JSON responses are mapped to primitive type *float.*

The use of *float* type is not interchangeable with *double*as is the case for many scenarios in java.

Case in point is this response:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | {      "odd": {          "price": "1.30",          "ck": 12.2,          "name": "1"      }  } |

assume we are running the following test on the value of *ck*:

|  |  |
| --- | --- |
| 1 | get("/odd").then().assertThat().body("odd.ck", equalTo(12.2)); |

This test will fail even if the value we are testing is equal to the value in the response. This is because we are comparing to a *double* rather than to a *float*.

To make it work, we have to explicitly specify the operand to the *equalTo* matcher method as a *float*, like so:

|  |  |
| --- | --- |
| 1 | get("/odd").then().assertThat().body("odd.ck", equalTo(12.2f)); |

**6. Specifying the Request Method**

Typically, we would perform a request by calling a method such as *get(),* corresponding to the request method we want to use.

In addition, **we can also specify the HTTP verb using the *request()* method**:

|  |  |
| --- | --- |
| 1  2  3  4 | @Test  public void whenRequestGet\_thenOK(){      when().request("GET", "/users/eugenp").then().statusCode(200);  } |

The example above is equivalent to using *get()* directly.

Similarly, we can send *HEAD*, *CONNECT* and *OPTIONS* requests:

|  |  |
| --- | --- |
| 1  2  3  4 | @Test  public void whenRequestHead\_thenOK() {      when().request("HEAD", "/users/eugenp").then().statusCode(200);  } |

***POST* request also follows a similar syntax and we can specify *the body*by using the *with()*and *body()* methods.**

Therefore, to create a new *Odd*by sending a *POST*request:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | @Test  public void whenRequestedPost\_thenCreated() {      with().body(new Odd(5.25f, 1, 13.1f, "X"))        .when()        .request("POST", "/odds/new")        .then()        .statusCode(201);  } |

The *Odd*object sent as *body*will automatically be converted to JSON. We can also pass any *String*that we want to send as our *POST* *body.*

**7. Default Values Configuration**

We can configure a lot of default values for the tests:

|  |  |
| --- | --- |
| 1  2  3  4  5 | @Before  public void setup() {      RestAssured.baseURI = "[https://api.github.com](https://api.github.com/)";      RestAssured.port = 443;  } |

Here, we’re setting a base URI and port for our requests. Besides these, we can also configure the base path, root pat, and authentication.

Note: we can also reset to the standard REST-assured defaults by using:

|  |  |
| --- | --- |
| 1 | RestAssured.reset(); |

**8. Measure Response Time**

Let’s see how we can **measure the response time using the *time()* and *timeIn()* methods of the *Response* object**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | @Test  public void whenMeasureResponseTime\_thenOK() {      Response response = RestAssured.get("/users/eugenp");      long timeInMS = response.time();      long timeInS = response.timeIn(TimeUnit.SECONDS);        assertEquals(timeInS, timeInMS/1000);  } |

Note that:

* *time()* is used to get response time in milliseconds
* *timeIn()* is used to get response time in the specified time unit

**8.1. Validate Response Time**

We can also validate the response time – in milliseconds – with the help of simple *long* *Matcher:*

|  |  |
| --- | --- |
| 1  2  3  4 | @Test  public void whenValidateResponseTime\_thenSuccess() {      when().get("/users/eugenp").then().time(lessThan(5000L));  } |

If we want to validate the response time in a different time unit, then we’ll use the *time()* matcher with a second *TimeUnit* parameter:

|  |  |
| --- | --- |
| 1  2  3  4 | @Test  public void whenValidateResponseTimeInSeconds\_thenSuccess(){      when().get("/users/eugenp").then().time(lessThan(5L),TimeUnit.SECONDS);  } |

**9. XML Response Verification**

Not only can it validate a JSON response, itcan validate XML as well.

Let’s assume we make a request to *http://localhost:8080/employees* and we get the following response:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | <employees>      <employee category="skilled">          <first-name>Jane</first-name>          <last-name>Daisy</last-name>          <sex>f</sex>      </employee>  </employees> |

We can verify that the *first-name* is *Jane*like so:

|  |  |
| --- | --- |
| 1  2  3  4  5 | @Test  public void givenUrl\_whenXmlResponseValueTestsEqual\_thenCorrect() {      post("/employees").then().assertThat()        .body("employees.employee.first-name", equalTo("Jane"));  } |

We can also verify that all values match our expected values by chaining body matchers together like so:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | @Test  public void givenUrl\_whenMultipleXmlValuesTestEqual\_thenCorrect() {      post("/employees").then().assertThat()        .body("employees.employee.first-name", equalTo("Jane"))          .body("employees.employee.last-name", equalTo("Daisy"))            .body("employees.employee.sex", equalTo("f"));  } |

Or using the shorthand version with variable arguments:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | @Test  public void givenUrl\_whenMultipleXmlValuesTestEqualInShortHand\_thenCorrect() {      post("/employees")        .then().assertThat().body("employees.employee.first-name",          equalTo("Jane"),"employees.employee.last-name",            equalTo("Daisy"), "employees.employee.sex",              equalTo("f"));  } |

**10. XPath for XML**

**We can also verify our responses using XPath.** Consider the example below that executes a matcher on the *first-name*:

|  |  |
| --- | --- |
| 1  2  3  4  5 | @Test  public void givenUrl\_whenValidatesXmlUsingXpath\_thenCorrect() {      post("/employees").then().assertThat().        body(hasXPath("/employees/employee/first-name", containsString("Ja")));  } |

XPath also accepts an alternate way of running the *equalTo* matcher:

|  |  |
| --- | --- |
| 1  2  3  4  5 | @Test  public void givenUrl\_whenValidatesXmlUsingXpath2\_thenCorrect() {      post("/employees").then().assertThat()        .body(hasXPath("/employees/employee/first-name[text()='Jane']"));  } |

**11. Logging Test Details**

**11.1. Log Request Details**

First, let’s see how to **log entire request details using *log().all()****:*

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | @Test  public void whenLogRequest\_thenOK() {      given().log().all()        .when().get("/users/eugenp")        .then().statusCode(200);  } |

This will log something like this:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | Request method: GET  Request URI:    https://api.github.com:443/users/eugenp  Proxy:          <none>  Request params: <none>  Query params:   <none>  Form params:    <none>  Path params:    <none>  Multiparts:     <none>  Headers:        Accept=\*/\*  Cookies:        <none>  Body:           <none> |

To log only specific parts of the request, we have the *log()* method in combination with *params(), body(), headers(), cookies(), method(), path()* eg *log.().params().*

**Note that other libraries or filters used may alter what’s actually sent to the server, so this should only be used to log the initial request specification.**

**11.2. Log Response Details**

Similarly, we can log the response details.

In the following example we’re logging the response body only:

|  |  |
| --- | --- |
| 1  2  3  4  5 | @Test  public void whenLogResponse\_thenOK() {      when().get("/repos/eugenp/tutorials")        .then().log().body().statusCode(200);  } |

Sample output:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | {      "id": 9754983,      "name": "tutorials",      "full\_name": "eugenp/tutorials",      "private": false,      "html\_url": "<https://github.com/eugenp/tutorials>",      "description": "The \"REST With Spring\" Course: ",      "fork": false,      "size": 72371,      "license": {          "key": "mit",          "name": "MIT License",          "spdx\_id": "MIT",          "url": "<https://api.github.com/licenses/mit>"      },  ...  } |

**11.3. Log Response if Condition Occurred**

We also have the option of logging the response only if an error occurred or the status code matches a given value:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | @Test  public void whenLogResponseIfErrorOccurred\_thenSuccess() {        when().get("/users/eugenp")        .then().log().ifError();      when().get("/users/eugenp")        .then().log().ifStatusCodeIsEqualTo(500);      when().get("/users/eugenp")        .then().log().ifStatusCodeMatches(greaterThan(200));  } |

**11.4. Log if Validation Failed**

We can also log both request and response only if our validation failed:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | @Test  public void whenLogOnlyIfValidationFailed\_thenSuccess() {      when().get("/users/eugenp")        .then().log().ifValidationFails().statusCode(200);        given().log().ifValidationFails()        .when().get("/users/eugenp")        .then().statusCode(200);  } |

In this example, we want to validate that the status code is 200. Only if this fails, the request and response will be logged.