Contents

[Rest-Assured 1](#_Toc518294307)

[Introduction 1](#_Toc518294308)

[Setting up Java, Eclipse and TestNG 1](#_Toc518294309)

[Download Rest-Assured Jars 1](#_Toc518294310)

[Creating a Project in Eclipse 2](#_Toc518294311)

[Setting up Rest-Assured Jars in class Path of Eclipse 3](#_Toc518294312)

[First Test with Rest-Assured 6](#_Toc518294313)

[Understanding the code 8](#_Toc518294314)

[Validate Response Status Code using Rest-Assured 9](#_Toc518294315)

[Verifying an error Status code 11](#_Toc518294316)

[Validating Response Status Line using Rest-Assured 12](#_Toc518294317)

[Read JSON Response Body using Rest-Assured 12](#_Toc518294318)

[Check if a String is contained in the Response Body 13](#_Toc518294319)

[Check String presence by ignoring alphabet casing 14](#_Toc518294320)

[Extract a node text using JsonPath 14](#_Toc518294321)

[Sample Code to read all the nodes from Weather API Response 15](#_Toc518294322)

[Making a *POST* request using Rest Assured 16](#_Toc518294323)

# Rest-Assured

## Introduction

Rest-Assured is a Java based library that is used to test RESTful Web Services. This library behaves like a headless Client to access REST web services. We can create highly customize-able HTTP Requests to send to the Restful server. This enables us to test wide variety of Request combinations and in turn test different combinations of core business logic.

Rest-Assured library also provides ability to validate the HTTP Responses received from server. For e.g. we can verify the Status code, Status message, Headers and even the Body of the response. This makes Rest-Assured a very flexible library that can be used for testing.

## Setting up Java, Eclipse and TestNG

1. Install Java JDK and configure JAVA\_HOME settings.
2. Download and install Eclipse IDE.
3. Install TestNG in Eclipse.
   1. Launch the Eclipse IDE and from Help menu, click “Install New Software”.
   2. You will see a dialog window, click “Add” button.
   3. Type name as you wish, lets take “TestNG” and type “http://beust.com/eclipse/” as location. Click OK.
   4. You come back to the previous window but this time you must see TestNG option in the available software list. Just Click TestNG and press “Next” button.
   5. Click “I accept the terms of the license agreement” then click Finish.
   6. You may or may not encounter a Security warning, if in case you do just click OK.
   7. Click Next again on the succeeding dialog box until it prompts you to Restart the Eclipse.
   8. You are all done now, just Click Yes.
   9. After restart, verify if TestNG was indeed successfully installed. Right click on you project and see if TestNG is displayed in the opened menu.

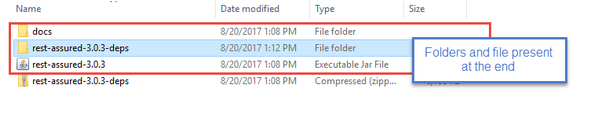
## Download Rest-Assured Jars

First download all the required Rest-Assured jar files. Here is the link to download the Jars: <https://github.com/rest-assured/rest-assured/wiki/Downloads> On this page you will find a section to download “Current direct downloads”. Move to that section and click on the link to ‘dist’ package.

After download, go to that folder where the zip file was downloaded and simply unzip the files. Go to the unzipped folder and open it. There should be following folders and files inside the unzipped folder.

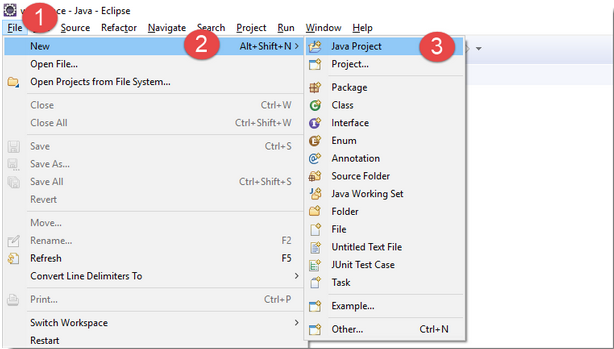
* docs: A folder containing javadocs
* rest-assured-3.0.3.jar: jar file containing Rest-Assured classes
* rest-assured-3.0.3-deps.zip : Another zip file containing all the dependency jars.

make sure to unzip the rest-assured-3.0.3-deps.zip dependency jars as well. At the end your folder would like this.

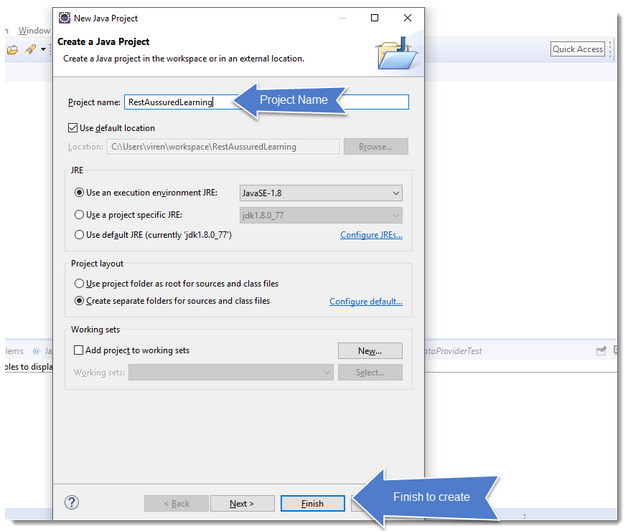


## Creating a Project in Eclipse

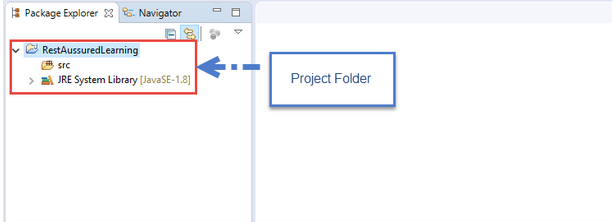
Now it is required to add the unzipped jars in the class path of Eclipse project. But before that create Eclipse project with name RestAussuredLearning.



Once clicked on the Java Project link, a new project window opens up. Enter the name of the project as RestAussuredLearning and click on Finish button.

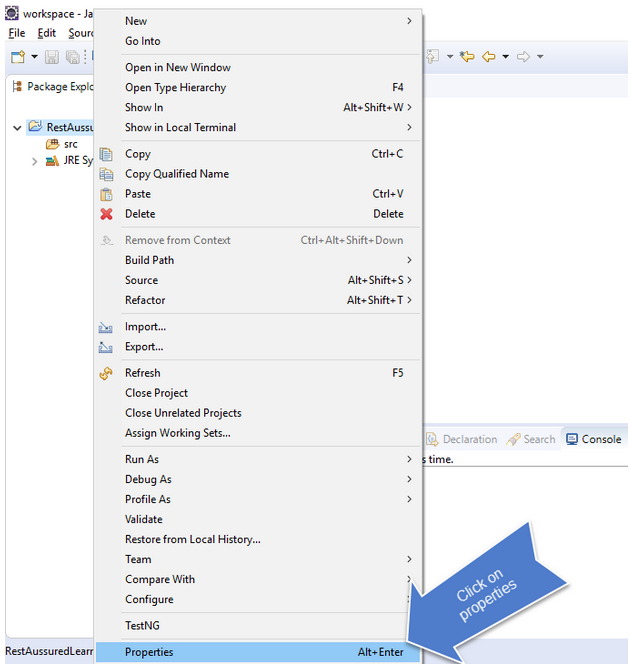


Now the project with name RestAssuredLearning should display in the package explorer.

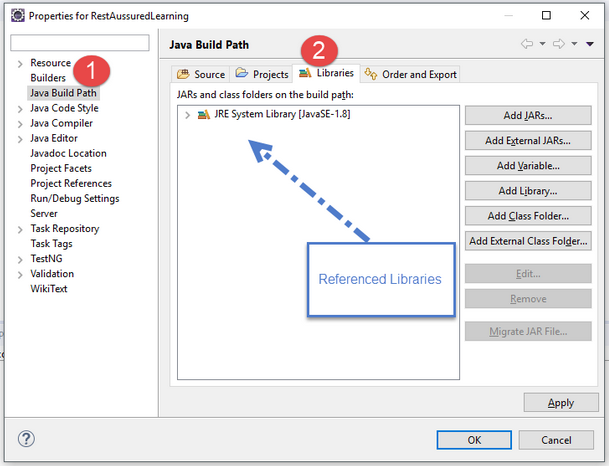


## Setting up Rest-Assured Jars in class Path of Eclipse

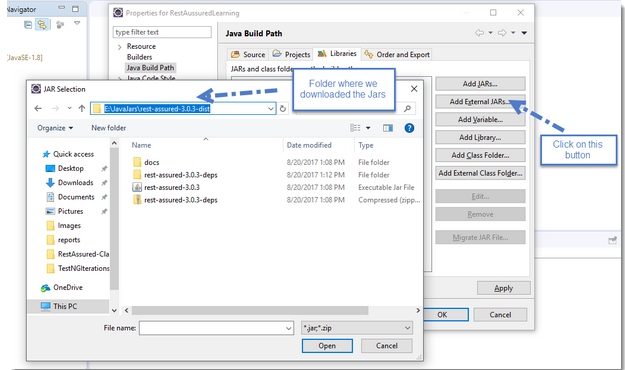
Right click on the Java project folder in the package explorer pane and choose Properties. This will open up the Project properties pop up window.



In the properties window, go to Java build path option in the left pane. In the Java build path pane, on the left hand side you will see the Libraries pane.



Now reference the jar files that we downloaded earlier. To do that click on the Add external jars button and browse to the folder where we unzipped all our Rest-Assured jars. Just like shown in the image below.



Make sure to include the following Jars

* rest-assured-3.0.3.jar
* All the jars in the folder rest-assured-3.0.3-deps

That is all have to do, we have successfully set up a Java project with Rest-Assured library.

## First Test with Rest-Assured

This test will hit a simple Restful web service. Details of the Restful Web service are mentioned in the below table:

Endpoint: [http://restapi.demoqa.com/utilities/weather/city/<City](http://restapi.demoqa.com/utilities/weather/city/%3cCity)>

HTTP method: GET

Comments: Here <City> means the city for which we are trying to retrieve the weather data. For e.g. if you want to know the weather conditions of Hyderabad, you would simple replace the <City> text with Hyderabad.

The Restful resource URL for Hyderabad becomes:

<http://restapi.demoqa.com/utilities/weather/city/Hyderabad>

Response:

{

“City”: “Hyderabad”,

“Temperature”: “31.49 Degree celsius”,

“Humidity”: “62 Percent”,

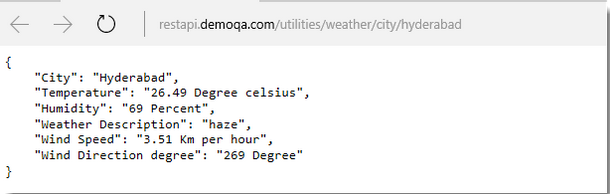
“Weather Description”: “scattered clouds”,

“Wind Speed”: “3.6 Km per hour”,

“Wind Direction degree”: “270 Degree”

}

Try to open <http://restapi.demoqa.com/utilities/weather/city/Hyderabad> in browser. Output will look like below:



In order to do the same thing using Rest-Assured, need to follow the steps below:

1. Use the RestAssured class to generate a RequestSpecification for the URL: http://restapi.demoqa.com/utilities/weather/city/Hyderabad
2. Specify the HTTP Method type
3. Send the Request to the Server
4. Get the Response back from the server
5. Print the returned Response’s Body

Below is the code to hit the above end point. Let’s have a look at the code first and then at the explanation of each line of code in the bottom.

import org.testng.annotations.Test;

import io.restassured.RestAssured;

import io.restassured.http.Method;

import io.restassured.response.Response;

import io.restassured.specification.RequestSpecification;

public class SimpleGetTest {

@Test

public void GetWeatherDetails()

{

// Specify the base URL to the RESTful web service

RestAssured.baseURI = "http://restapi.demoqa.com/utilities/weather/city";

// Get the RequestSpecification of the request that you want to sent

// to the server. The server is specified by the BaseURI that we have

// specified in the above step.

RequestSpecification httpRequest = RestAssured.given();

// Make a request to the server by specifying the method Type and the method URL.

// This will return the Response from the server. Store the response in a variable.

Response response = httpRequest.request(Method.GET, "/Hyderabad");

// Now let us print the body of the message to see what response

// we have recieved from the server

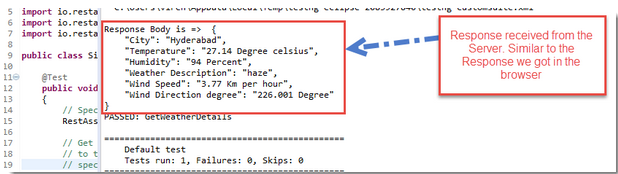
String responseBody = response.getBody().asString();

System.out.println("Response Body is => " + responseBody);

}

}

The above code will produce the same response which was received when opened the same URL on a browser. Here is how the response will be printed in Eclipse console window.



## Understanding the code

Code line 1

// Specify the base URL to the RESTful web service

RestAssured.baseURI = "http://restapi.demoqa.com/utilities/weather/city";

This line uses a class called RestAssured to set up a request with the specified base URI. In our case the base URI is “http://restapi.demoqa.com/utilities/weather/city”. This is called the base URI because the is root address of the resource. Adding “/Hyderabad” at the end appends the exact resource name in the URI that we are trying to access.

Coming back to the class io.restassured.RestAssured , this class forms the basis of any kind of HTTP request that is required in the tests. Some key features of this class are

* It creates HTTP Requests against a base URI
* It supports creating Request of different HTTP method types (GET, POST, PUT, PATH, DELETE, UPDATE, HEAD and OPTIONS)
* It makes HTTP communication with the server and passes on the Request that we created in our tests to the server.
* Retrieves the Response from the server.
* Helps validate the Response received from the server.

Internally this class uses HTTP builder library, Http builder is a Groovy language based HTTP client.

Code line 2

Once the request is set up, store the Request in a variable so that it can be modified. In this particular test, it is not required to modify the test. Still following the same approach to understand the basics.

// Get the RequestSpecification of the request that you want to sent

// to the server. The server is specified by the BaseURI that we have

// specified in the above step.

RequestSpecification httpRequest = RestAssured.given();

Here RestAssured class is returning the Request against the base URI, as specified in line 1. Every Request in Rest-Assured library is represented by an interface called RequestSpecification. This interface allows to modify the request, like adding headers or adding authentication details. The word specification at the end is used to signify that how the request should look like, when sent to the server.

Code line 3

Now that RequestSpecification object is there, call the server to get the resource. This piece of code tells RequestSpecification to issue a request to the server.

// Make a request to the server by specifying the method Type and the method URL.

// This will return the Response from the server. Store the response in a variable.

Response response = httpRequest.request(Method.GET, "/Hyderabad");

Issuing request takes two arguments, first argument as HTTP Method Type and second as String (“/Hyderabad”). This step actually sends the request to the remote server and gets a response back. This is why the return type of the request is specified as Response.

In Rest-Assured io.restassured.response.Response interface represents a Response returned from a server. This Response object will contain all the data sent by the server. Different method can be called on the Response object to get different parts of the Response. For e.g. call to get Headers, Status code and the body of the Response. In the next code line we will get the body of the Response.

Code line 4 and 5

This piece of code just tries to read the response and print the response.

// Now let us print the body of the message to see what response

// we have recieved from the server

String responseBody = response.getBody().asString();

System.out.println("Response Body is =>  " + responseBody);

Response interface has a method called getBody() , this method will return the body of the received response. Response Body is converted into a string value and printed on the console using the System.out.println statement.

Rest-Assured provides a lot of short hand methods which can help you write short code. The above test method can be written in a little different way, here is the code snippet. Please go through the code comments to understand the usage.

@Test

public void GetWeatherDetailsCondensed()

{

// Specify the base URL to the RESTful web service

RestAssured.baseURI = "http://restapi.demoqa.com/utilities/weather/city";

// Get the RequestSpecification of the request that you want to sent

// to the server. The server is specified by the BaseURI that we have

// specified in the above step.

RequestSpecification httpRequest = RestAssured.given();

// Make a GET request call directly by using RequestSpecification.get() method.

// Make sure you specify the resource name.

Response response = httpRequest.get("/Hyderabad");

// Response.asString method will directly return the content of the body

// as String.

System.out.println("Response Body is =>  " + response.asString());

}

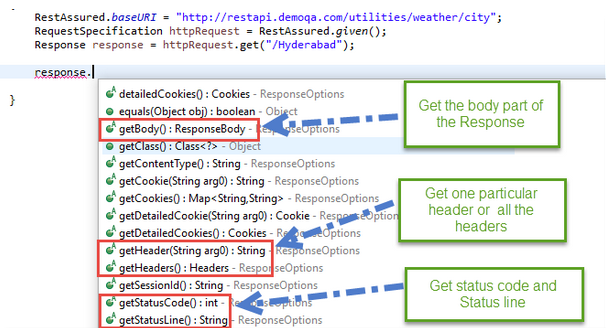
## Validate Response Status Code using Rest-Assured

Functional testing of Web Services involves verifying the Responses returned from various End Points. Some of the important Test Verification are

* HTTP Status Code returned from the server
* Contents of the header
* Content of the body

Response object which represents the HTTP Response packet received from the Web service Server. HTTP Response contains Status, collection of Headers and a Body. Hence, it becomes obvious that the Response object should provide mechanisms to read Status, Headers and Body.

Response is an interface which lives in the package: io.restassured.response. This interface has lots of method, mostly methods which can help to get parts of the received response. Take a look at some of the important methods. A simple Response followed by a dot (Response.) in eclipse would be shown the available methods on the interface. As shown in the image below



Noticed, getStatusCode() method can be used to get the status code of the Response. This method returns an integer and test will verify the value of this integer. TestNG Assert is used to verify the Status Code.

@Test

public void GetWeatherDetails()

{

RestAssured.baseURI = "http://restapi.demoqa.com/utilities/weather/city";

RequestSpecification httpRequest = RestAssured.given();

Response response = httpRequest.get("/Hyderabad");

// Get the status code from the Response. In case of

// a successfull interaction with the web service, we

// should get a status code of 200.

int statusCode = response.getStatusCode();

// Assert that correct status code is returned.

Assert.assertEquals(statusCode /\*actual value\*/, 200 /\*expected value\*/, "Correct status code returned");

}

Below line of code extracts the status code from the message:

int statusCode = response.getStatusCode();

Once the status code is received, it is compared with the expected value of 200.

// Assert that correct status code is returned.

Assert.assertEquals(statusCode /\*actual value\*/, 200 /\*expected value\*/, "Correct status code returned");

Try running the test and notice that the test will pass. This is because the web service indeed returns a status code of 200. Try to print the value of **statusCode.**

## Verifying an error Status code

Status codes returned by the Server depends on whether the Request was successful or not. If the Request is successful, Status Code 200 is returned. status code. If the Request is not successful, Status Code other than 200 will be returned. Get a list of HTTP Status codes and their meanings on the W3 page.

For ToolsQA Weather web service, let’s create another test which tests a negative scenario. The scenario is

* Verify the Status Code returned by Weather web service on providing invalid City name.

City name that this test will use here is a long number, like 78789798798. Just copy the code from previous test and simple replace the city name with 78789798798. At this moment do not change the status code. The code looks like this

@Test

public void GetWeatherDetailsInvalidCity()

{

RestAssured.baseURI = "http://restapi.demoqa.com/utilities/weather/city";

RequestSpecification httpRequest = RestAssured.given();

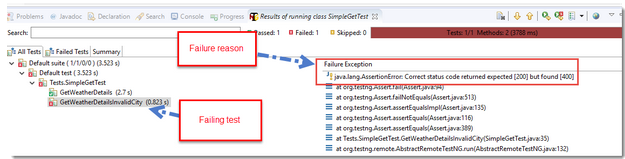
Response response = httpRequest.get("/78789798798");

int statusCode = response.getStatusCode();

Assert.assertEquals(statusCode /\*actual value\*/, 200 /\*expected value\*/, "Correct status code returned");

}

Now run your test class and check the result. Our negative test GetWeatherDetailsInvalidCity will fail. The reason is that the web service returns an error code of 400 when invalid city name is sent to it. However, the test is validating for expected value of 200. Look at the TesNG results, as shown in the image below.



A quick change in the expected result in the code will make sure that this test passes. Update respective line as in the code below

Assert.assertEquals(statusCode /\*actual value\*/, 400 /\*expected value\*/, "Correct status code returned");

## Validating Response Status Line using Rest-Assured

The first line returned in the Response from Server is called Status Line. Status line is composed of three sub strings

* Http protocol version
* Status Code
* Status Code’s string value

During a success scenario a status line will look something like this “HTTP/1.1 200 OK”. First part is Http protocol (HTTP/1.1). Second is Status Code (200). Third is the Status message (OK).

Just as we go the Status Code in the line above, we can get the Status Line as well. To get the Status line we will simply call the Response.getStatusLine() method. Here is the code to do that.

@Test

public void GetWeatherStatusLine()

{

RestAssured.baseURI = "http://restapi.demoqa.com/utilities/weather/city";

RequestSpecification httpRequest = RestAssured.given();

Response response = httpRequest.get("/Hyderabad");

// Get the status line from the Response and store it in a variable called statusLine

String statusLine = response.getStatusLine();

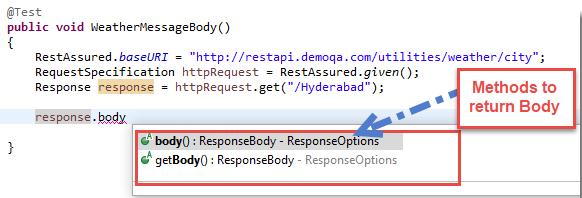
Assert.assertEquals(statusLine /\*actual value\*/, "HTTP/1.1 200 OK" /\*expected value\*/, "Correct status code returned");

}

## Read JSON Response Body using Rest-Assured

When we request for the Weather details of a particular city, Server responds by sending the Weather details of the city as the Response Body. Response interface contains two methods to get the Response Body.

* Response.body() : returns ResponseBody
* Response.getBody() : returns ResponseBody



Using these methods we can get an Object of type io.restassured.response.ResponseBody. This class represents the Body of a received Response. Using this class you can get and validate complete or parts of the Response Body. In the below code we will simply read the complete Response Body by using Response.getBody() and will print it out on the console window.

@Test

public void WeatherMessageBody()

{

RestAssured.baseURI = "http://restapi.demoqa.com/utilities/weather/city";

RequestSpecification httpRequest = RestAssured.given();

Response response = httpRequest.get("/Hyderabad");

// Retrieve the body of the Response

ResponseBody body = response.getBody();

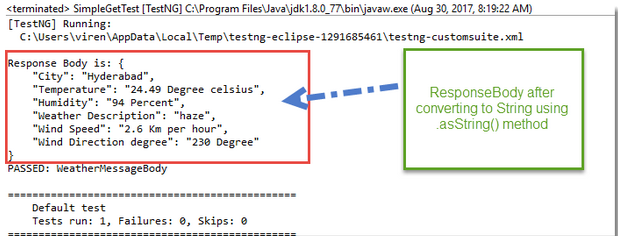
// By using the ResponseBody.asString() method, we can convert the  body

// into the string representation.

System.out.println("Response Body is: " + body.asString());

}

ResponseBody interface also has a method called .asString(), as used in the above code, which converts a ResponseBody into its String representation. If you run this test the output will look something like this:



***Note:*** ***Response.body()*** method does exactly the same thing. So you can even use ***.body()*** method in the above code.

### Check if a String is contained in the Response Body

ResponseBody can return the response body in a String format. We can use simple String methods to verify certain basic level of values in the Response. For e.g. we can use the String.contains() method to see if the Response contains a “Hyderabad” in it. The below code shows how to check for sub string presence.

@Test

public void WeatherMessageBody()

{

RestAssured.baseURI = "http://restapi.demoqa.com/utilities/weather/city";

RequestSpecification httpRequest = RestAssured.given();

Response response = httpRequest.get("/Hyderabad");

// Retrieve the body of the Response

ResponseBody body = response.getBody();

// To check for sub string presence get the Response body as a String.

// Do a String.contains

String bodyAsString = body.asString();

Assert.assertEquals(bodyAsString.contains("Hyderabad") /\*Expected value\*/, true /\*Actual Value\*/, "Response body contains Hyderabad");

}

### Check String presence by ignoring alphabet casing

We can also ignore the casing using the String internal methods. To do this we will convert the Response in lower case and then compare it with our lower case string value. Below code demonstrates that.

@Test

public void WeatherMessageBody()

{

RestAssured.baseURI = "http://restapi.demoqa.com/utilities/weather/city";

RequestSpecification httpRequest = RestAssured.given();

Response response = httpRequest.get("/Hyderabad");

// Retrieve the body of the Response

ResponseBody body = response.getBody();

// To check for sub string presence get the Response body as a String.

// Do a String.contains

String bodyAsString = body.asString();

// convert the body into lower case and then do a comparison to ignore casing.

Assert.assertEquals(bodyAsString.toLowerCase().contains("hyderabad") /\*Expected value\*/, true /\*Actual Value\*/, "Response body contains Hyderabad");

}

The above two approaches suffer from a classical problem, what if the string “Hyderabad” is present in a wrong node or may be multiple instances of the same string are present. This is not a fool proof way of testing a particular node in the Response. There are better ways, Response interface gives you a mechanism to extract nodes based on a given JsonPath. There is a method called Response.JsonPath(), which returns a io.restassured.path.json.JsonPath Object. This object can be used to further query specific parts of the Response Json.

### Extract a node text using JsonPath

Let us continue with the above example and retrieve the City from the Response. To do so, we will simply get the JsonPath object from the Response interface and then query for the particular node. Just to be very clear, let us look at the Weather API response again.

{

    "City": "Hyderabad",

    "Temperature": "25.51 Degree celsius",

    "Humidity": "94 Percent",

    "Weather Description": "mist",

    "Wind Speed": "1 Km per hour",

    "Wind Direction degree": " Degree"

}

In this response, if we want to go to the City node, all we have to do is have the following JsonPath: $.City. Try it out on the JsonPath Evaluator to verify the output.

Now let us look at the code, pay specific attention to the comments in the code.

Note: In Java JsonPath you do not need to have $ as the root node. You can completely skip that.

@Test

public void VerifyCityInJsonResponse()

{

RestAssured.baseURI = "http://restapi.demoqa.com/utilities/weather/city";

RequestSpecification httpRequest = RestAssured.given();

Response response = httpRequest.get("/Hyderabad");

// First get the JsonPath object instance from the Response interface

JsonPath jsonPathEvaluator = response.jsonPath();

// Then simply query the JsonPath object to get a String value of the node

// specified by JsonPath: City (Note: You should not put $. in the Java code)

String city = jsonPathEvaluator.get("City");

// Let us print the city variable to see what we got

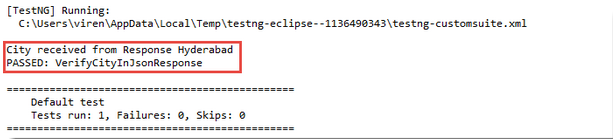
System.out.println("City received from Response " + city);

// Validate the response

Assert.assertEquals(city, "Hyderabad", "Correct city name received in the Response");

}

The output of the code passes the assertion and it also prints the City name retrieved from the Response. As shown in the image below



On the similar lines you can extract any part of the Json response using the JsonPath implementation of Rest-Assured. This is very convenient, compact and easy way to write tests.

### Sample Code to read all the nodes from Weather API Response

Now that we know how to read a node using JsonPath, here is a small piece of code that reads all the nodes and prints them to the Console.

@Test

public void DisplayAllNodesInWeatherAPI()

{

RestAssured.baseURI = "http://restapi.demoqa.com/utilities/weather/city";

RequestSpecification httpRequest = RestAssured.given();

Response response = httpRequest.get("/Hyderabad");

// First get the JsonPath object instance from the Response interface

JsonPath jsonPathEvaluator = response.jsonPath();

// Let us print the city variable to see what we got

System.out.println("City received from Response " + jsonPathEvaluator.get("City"));

// Print the temperature node

System.out.println("Temperature received from Response " + jsonPathEvaluator.get("Temperature"));

// Print the humidity node

System.out.println("Humidity received from Response " + jsonPathEvaluator.get("Humidity"));

// Print weather description

System.out.println("Weather description received from Response " + jsonPathEvaluator.get("Weather"));

// Print Wind Speed

System.out.println("City received from Response " + jsonPathEvaluator.get("WindSpeed"));

// Print Wind Direction Degree

System.out.println("City received from Response " + jsonPathEvaluator.get("WindDirectionDegree"));

}

## Making a POST request using Rest Assured

Let us now try to test POST web service. We will learn about

* POST request using Rest Assured
* Creating Json data using Simple Json library
* Sending JSON content in the body of Request
* Validating the Response.

In order to create JSON objects in the code we will add Simple Json library in the class path. You can download Simple json from maven using following URL: <https://mvnrepository.com/artifact/com.googlecode.json-simple/json-simple> Then add the downloaded Jars to class path.

Let us begin step by step with the code

Step 1: Create a Request pointing to the Service Endpoint

RestAssured.baseURI ="http://restapi.demoqa.com/customer";

RequestSpecification request = RestAssured.given();

Step 2: Create a JSON request which contains all the fields

// JSONObject is a class that represents a simple

// JSON. We can add Key - Value pairs using the put

// method

JSONObject requestParams = new JSONObject();

requestParams.put("FirstName", "Virender"); // Cast

requestParams.put("LastName", "Singh");

requestParams.put("UserName", "simpleuser001");

requestParams.put("Password", "password1");

requestParams.put("Email",  "someuser@gmail.com");

JSONObject is a class that is present in org.json.simple package. This class is a programmatic representation of a JSON string. Take a look at the Request JSON above for our test web service, you will notice that there are multiple nodes in the Json. Each node can be added using the JSONObject.put(String, String) method. Once you have added all the nodes you can get the String representation of JSONObject by calling JSONObject.toJSONString() method.

Step 3: Add JSON body in the request and send the Request

// Add a header stating the Request body is a JSON

request.header("Content-Type", "application/json");

// Add the Json to the body of the request

request.body(requestParams.toJSONString());

// Post the request and check the response

Response response = request.post("/register");

This web service accepts a JSON body. By this step we have created our JSON body that needs to be sent. In this step we will simply add the JSON String to the body of the HTTP Request and make sure that the Content-Type header field has a value of application/json.

You can put the JSON string in the body using the method called RequestSpecification.body(JsonString). This method lets you updated the content of HTTP Request Body. However, if you call this method multiple times the body will be updated to the latest Json String.

Step 4: Validate the Response

Once we get the response back, all we have to do is validate parts of the response. Let us validate Success Code text in the Response, we will be using JsonPath.

int statusCode = response.getStatusCode();

Assert.assertEquals(statusCode, "201");

String successCode = response.jsonPath().get("SuccessCode");

Assert.assertEquals( "Correct Success code was returned", successCode, "OPERATION\_SUCCESS");

Now that we have sent the Request and received a Response, let us validate Status Code and Response Body content. This is similar to validation techniques that we discussed in previous tutorials. I would strongly suggest that you read those tutorials. The above code does the validation, it is self-explanatory.

Complete code

Here is the complete code for the above example

@Test

public void RegistrationSuccessful()

{

RestAssured.baseURI ="http://restapi.demoqa.com/customer";

RequestSpecification request = RestAssured.given();

JSONObject requestParams = new JSONObject();

requestParams.put("FirstName", "Virender"); // Cast

requestParams.put("LastName", "Singh");

requestParams.put("UserName", "sdimpleuser2dd2011");

requestParams.put("Password", "password1");

requestParams.put("Email",  "sample2ee26d9@gmail.com");

request.body(requestParams.toJSONString());

Response response = request.post("/register");

int statusCode = response.getStatusCode();

Assert.assertEquals(statusCode, "201");

String successCode = response.jsonPath().get("SuccessCode");

Assert.assertEquals( "Correct Success code was returned", successCode, "OPERATION\_SUCCESS");

}

Changing the HTTP Verb on a POST request

One of the key aspect of Web service testing is to verify negative scenarios on the Endpoint. There could be many negative scenarios, some of them are

* Sending incomplete POST Data
* Sending Json data with incorrect syntax
* Sending incorrect Verb in the Request.

Let us see what the impact will be if we send a GET request to an Endpoint that expects POST. Below code tries to do that, we will just print out the Response status code and the Response body to see if we get any error.

public void RegistrationSuccessful()

{

RestAssured.baseURI ="http://restapi.demoqa.com/customer";

RequestSpecification request = RestAssured.given();

JSONObject requestParams = new JSONObject();

requestParams.put("FirstName", "Virender"); // Cast

requestParams.put("LastName", "Singh");

requestParams.put("UserName", "sdimpleuser2dd2011");

requestParams.put("Password", "password1");

requestParams.put("Email",  "sample2ee26d9@gmail.com");

request.body(requestParams.toJSONString());

Response response = request.get("/register");

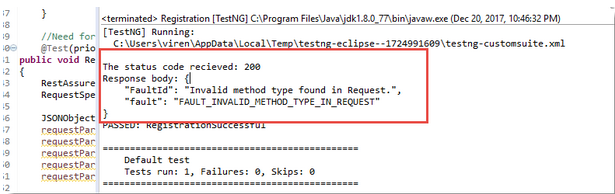
int statusCode = response.getStatusCode();

System.out.println("The status code recieved: " + statusCode);

System.out.println("Response body: " + response.body().asString());

}

If you run this test following output is generated. In the output we can clearly see that the Response body tells us about the incorrect usage of HTTP Verb. HTTP verb are also known as the Method types, given we actually make a remote method call and specify the type of the method call.



try out further scenarios yourself on the registration API that is described.