[**https://github.com/pavanoltraining**](https://github.com/pavanoltraining)

**: demo APIs for practice-**

1) [https://reqres.in/](https://www.youtube.com/redirect?event=video_description&redir_token=QUFFLUhqbFBHT1FCMDd5NzE2WjNabmFPN0FtZk5fTHhkQXxBQ3Jtc0ttaHZCMmNQclRNTWVPT2cwVjNaWHlvNjlWY1JSQm1STURQZjRtMW1KSkY3aVNGOGx6RE9jcFllM2VGaUt5aDh4eVJETDIzYms5eEl2UHhTRW5pWXEtRmtuNmFxSHVUV2daM3Fpdk5CQTRkcVhrcmxkUQ&q=https%3A%2F%2Freqres.in%2F&v=rpZcTKJTOi4)

2) [http://dummy.restapiexample.com/](https://www.youtube.com/redirect?event=video_description&redir_token=QUFFLUhqbXBfYkh1SFRGSjZrbnJmT2pKM1dLb1NNSXVkd3xBQ3Jtc0trcFdvaFpmVG5IbnppX1VJd2loVkR1Y3YtLVlfQWREUXNhLTk0N0FtenJWMnB5cnZ4WFdlSzdZbGEyWEczLWtyaGp0b1FqeGo4RVJJdXMxUjE5Y2FLRzRIX290aEozclU3cG1qeXVCTlJRRTFpa0V1OA&q=http%3A%2F%2Fdummy.restapiexample.com%2F&v=rpZcTKJTOi4)

3) [https://jsonplaceholder.typicode.com/](https://www.youtube.com/redirect?event=video_description&redir_token=QUFFLUhqbUpzam10eXFtdlNvMXNYcGpHbnAyUXVPTGhzQXxBQ3Jtc0tsN3FlWkM0am9CN2dhV3REdFhzamctQ0VrcW9wQjVqTzhqQW1RSmZBYTZGZWd4YVp3SGRMVkFXbmhQcjQ3cGZXTUFObFNxT0lrb2IxNVZYQm9Nb19ENllid2VCZWkzY3A1alROaVl2aXdFZVk4UmR6WQ&q=https%3A%2F%2Fjsonplaceholder.typicode.com%2F&v=rpZcTKJTOi4)

4) [https://gorest.co.in/](https://www.youtube.com/redirect?event=video_description&redir_token=QUFFLUhqazBBa1Z5OGNfb3RFc3ZxQnk1VDF0WHlmRFNvd3xBQ3Jtc0trZlNHdk1McjNkeXJoVGotaTdkSUFQVlZFM2dYYnA0bkhLMjFrMF9XZW5mVnhlY01jdVFYWElNQzRDRHdTNXBCRFVkbHFGZ3lkZlQ0LXE3YWdiSkxXOTEwRFlNcWtoTnByNVp2NVRwdm9BV1p1OHZVUQ&q=https%3A%2F%2Fgorest.co.in%2F&v=rpZcTKJTOi4)

5) [https://restful-booker.herokuapp.com/](https://www.youtube.com/redirect?event=video_description&redir_token=QUFFLUhqazg4Ry1tVUZ0aTl2UTl1UHZUbE5odGF5cUp4UXxBQ3Jtc0trbmlvQnVKbXItc2MybXJBc1FQNTBwLXVTZUtRSXpoUDBMbk4xTVdEdy16WEtDbHRBRzdaTDBZRmpWanR2SGtUYXlOTWpvWmU3RlYtOGRhM3V6ZW4zRDZCX0k0S0hFc1BVcWQxYVBpMV9CMGZTYWliTQ&q=https%3A%2F%2Frestful-booker.herokuapp.com%2F&v=rpZcTKJTOi4)

6) [https://httpbin.org/#/](https://www.youtube.com/redirect?event=video_description&redir_token=QUFFLUhqbkZOUkFBQ2w5OXUtX252MUFUQ2VJTE1XMWtIQXxBQ3Jtc0ttNzk3WDF2XzVkYnN6S09JUVpXUzM3QVJFY3dycU9DZXZBMm1lN25IanNydkxDbXM3MmtmMWhDOUxRRjFqajV6SGxJazZzb3ZKd3pmNVBoWm5NODlsTUxuX3NjWXI2dy1CQWIwbUFQanBPLTYydndwSQ&q=https%3A%2F%2Fhttpbin.org%2F%23%2F&v=rpZcTKJTOi4)

7) [https://petstore.swagger.io/](https://www.youtube.com/redirect?event=video_description&redir_token=QUFFLUhqbXNPc2tucG4yM1FfanlFWExKRDZZeE9iQzhqUXxBQ3Jtc0ttVkFuQmIxTlFsQ1RaX3hlSkJLbEVyY3RxbEpPSFp0UDNHMk13NnVLNGhUa21jbkNxdGVZRURyNW54X05URHJnWFZYQVBnYUxpR0Y2ZDBLMzlTZUhnZngxTVdXcWpEbEI1clNGZm1iclhlVnZfZ0xjOA&q=https%3A%2F%2Fpetstore.swagger.io%2F&v=rpZcTKJTOi4)

8) [https://fakerestapi.azurewebsites.net...](https://www.youtube.com/redirect?event=video_description&redir_token=QUFFLUhqbGoxN2xoYlpMaUFLcjZWMTd6a2ZDZEJiLTRMZ3xBQ3Jtc0tsa1g5dWlmTTJxN3RJaVZTYXU0X1hxSEg0dFhzZHRzN25fdG9ROUJvOUZhdlY2amZId0w4WDFkQWhuc1BFdDhLdUpsN0RfR0dVY0NyUlVmTDRFNTNNWjNyUXBiVW9lU1RPZ2U1b1JHV0ZUTjYxRjRKbw&q=https%3A%2F%2Ffakerestapi.azurewebsites.net%2Findex.html&v=rpZcTKJTOi4)

9) [https://documenter.getpostman.com/vie...](https://www.youtube.com/redirect?event=video_description&redir_token=QUFFLUhqbHlvNG5XeWxMdmJPNG9qN04zaGUxQWF2eW1sQXxBQ3Jtc0trMDM1NFpkcEVBdFpLeUFydFhjbkdGelRDWGQ3SWRVb3RXNjl3azV4RnZYTjFHSF9ObE5MUTJtX3Y2Q3dLeEJhZTNlbVJ1aG5QaG4weWFuNzl6bEpzSDdaWElNRjZQWFBPSEZaUFRlNEI1dGFrNzYzSQ&q=https%3A%2F%2Fdocumenter.getpostman.com%2Fview%2F4016432%2FRWToRJCq%23intro&v=rpZcTKJTOi4)

10) Google API's [https://developers.google.com/maps/do...](https://www.youtube.com/redirect?event=video_description&redir_token=QUFFLUhqa0dPLXIzb0RqY3otMlFnQnRmQkZKalNXTlEzd3xBQ3Jtc0trN3RVT0VOaGlhalVMZndNeThJdVV1NjBtUHZ3VWRSYm9LdDRRYjE3cVR5Q0MxZk9uN3dHOURpeEFwTHdJMGtTRTZrZ3lWd1lYcDI0bkJPME5RSXVyWnQyTmtjd3NfTnFfSUwtcjI1QlV6RWozTVM2NA&q=https%3A%2F%2Fdevelopers.google.com%2Fmaps%2Fdocumentation&v=rpZcTKJTOi4)

API test with Rest Assured

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Ziro to Hero by Fun Do Tester \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

RestAssured.baseURI = 'http://xxxxxx';

Response response = RestAssured.given().

When().

header("Key" , “value”).

header("Auth" , “value”).

headers("contentType", "token" “key", "value”). **//multiple headers**

param("key", "value"). **//**

param("key", "value").

or params("key", "value" “key", "value”).

get(**/end point**) **//get takes 2 parts(headers, params)**

**OR**

headers("contentType", "token").

body(Map/JSon/dataProvider/(POJO)). **//body goes with POST request**

post(**/end point**). **//post takes 2 parts(headers , body)**

then(). **//then() is not mandatory.**

assertThat().

statuscode(200); **//verify expected value.**

**We need convert response to Json or asString…**

**Println(response.asString);**

JSonObject jsonobj = new JSonObject(response.asString);

assertThat().

statuscode(200). **//verify expected value.**

responseTime().  **//verify response time.**

//also can verify some response data here

\*\* how to write json path:

jsonpath.com > paste json file in JSON > write $.lastName (in real time no need $.)

\*\* how to write json array object :

jsonpath.com > paste json file in JSON > write $.city[2]. or $.city.street[abc st]

Sample project : https://github.com/Fundootesters/RestAssuredFramework.git

url: “<https://reqres.in/api/users?page=2>”;

baseURI = “https://reqres.in/api/users”;

RestAssured /

RestAssured.baseURI = “https://reqres.in/api”;

.given()

.when()

.header() // for single value- .header(xyz : “value”).

.studentAuth : “ ”,

.contentType : “ ”,

.anyParameter : “ ”

.param() // for single value- .param(xyz : “value”).

.{search : “test”,

PageNo : 1

}

.body() // for single value- .body(xyz : “value”).

{id : “01”,

Name : “john”,

Course : “testing”

For multiple key value pairs I have to use (map object OR Json object OR data-driven) and I have to write .headers(map/jason/data-driven) .params(map/jason/data-driven). Create a map and return with hash map then put the object inside parenthesis.

.post(“/users”)/) //post API contatains 2 things(header and body)

.get(“/users”) //get API contains 2 things (param and header)

.then()

.assertThat()

.statusCode(200)

Response response = *get*("https://reqres.in/api/users?page=2");

**int** statusCode = response.getStatusCode();

**long** time = response.getTime();

**String** firtstName = response.getBody().jsonPath().get("data[3].first\_name"));

**int** totlaEmployee = response.getBody().jsonPath().get("Employeedata.size()");

Assert.*assertEquals*(statusCode, 200);

Assert.*assertTrue*(time < 6000);

Assert.*assertEquals*(firstName, “Jawad”);

System.***out***.println("status Code: "+statusCode);

System.***out***.println(response.getStatusLine());

System.***out***.println("response time: "+time);

System.***out***.println(response.body().asString());

System.***out***.println(response.getHeader("content-type"));

System.***out***.println("firstName : "+ firstName);

**GET Request**

The HTTP GET request is used to fetch a resource from a server.

The following example uses the get() method from REST-assured library.

Example:

import io.restassured.RestAssured;

import io.restassured.http.ContentType;

import io.restassured.response.Response;

import org.junit.jupiter.api.Assertions;

import org.junit.jupiter.api.BeforeAll;

import org.junit.jupiter.api.Test;

import static io.restassured.RestAssured.given;

public class RestAssuredRequests {

@BeforeAll

public static void setup() {

RestAssured.baseURI = "https://jsonplaceholder.typicode.com";

}

@Test

public void getRequest() {

Response response = given()

.contentType(ContentType.JSON)

.when()

.get("/posts")

.then()

.extract().response();

Assertions.assertEquals(200, response.statusCode());

Assertions.assertEquals("qui est esse", response.jsonPath().getString("title[1]"));

}

}

### GET Request With Query Params

To send query parameters along with the GET request, we use the queryParam method:

import io.restassured.RestAssured;

import io.restassured.http.ContentType;

import io.restassured.response.Response;

import org.junit.jupiter.api.Assertions;

import org.junit.jupiter.api.BeforeAll;

import org.junit.jupiter.api.Test;

import static io.restassured.RestAssured.given;

public class RestAssuredRequests {

@BeforeAll

public static void setup() {

RestAssured.baseURI = "https://jsonplaceholder.typicode.com";

}

@Test

public void getRequestWithQueryParam() {

Response response = given()

.contentType(ContentType.JSON)

.param("postId", "2")

.when()

.get("/comments")

.then()

.extract().response();

Assertions.assertEquals(200, response.statusCode());

Assertions.assertEquals("Meghan\_Littel@rene.us", response.jsonPath().getString("email[3]"));

}

}

### POST Request

HTTP POST request is used to post data or create a resource on a server.

To send a POST request in REST-assured, we use the post() method:

import io.restassured.RestAssured;

import io.restassured.response.Response;

import org.junit.jupiter.api.Assertions;

import org.junit.jupiter.api.BeforeAll;

import org.junit.jupiter.api.Test;

import static io.restassured.RestAssured.given;

public class RestAssuredRequests {

private static String requestBody = "{\n" +

" \"title\": \"foo\",\n" +

" \"body\": \"bar\",\n" +

" \"userId\": \"1\" \n}";

@BeforeAll

public static void setup() {

RestAssured.baseURI = "https://jsonplaceholder.typicode.com";

}

@Test

public void postRequest() {

Response response = given()

.header("Content-type", "application/json")

.and()

.body(requestBody)

.when()

.post("/posts")

.then()

.extract().response();

Assertions.assertEquals(201, response.statusCode());

Assertions.assertEquals("foo", response.jsonPath().getString("title"));

Assertions.assertEquals("bar", response.jsonPath().getString("body"));

Assertions.assertEquals("1", response.jsonPath().getString("userId"));

Assertions.assertEquals("101", response.jsonPath().getString("id"));

}

}

**Related:**

* [REST-assured POST Request](https://devqa.io/rest-assured-post-request/)

### PUT Request

The PUT request updates a resource but requires the full JSON payload.

To send a PUT request in REST-assured, we use the put() method:

import io.restassured.RestAssured;

import io.restassured.response.Response;

import org.junit.jupiter.api.Assertions;

import org.junit.jupiter.api.BeforeAll;

import org.junit.jupiter.api.Test;

import static io.restassured.RestAssured.given;

public class RestAssuredRequests {

private static String requestBody = "{\n" +

" \"title\": \"foo\",\n" +

" \"body\": \"baz\",\n" +

" \"userId\": \"1\",\n" +

" \"id\": \"1\" \n}";

@BeforeAll

public static void setup() {

RestAssured.baseURI = "https://jsonplaceholder.typicode.com";

}

@Test

public void putRequest() {

Response response = given()

.header("Content-type", "application/json")

.and()

.body(requestBody)

.when()

.put("/posts/1")

.then()

.extract().response();

Assertions.assertEquals(200, response.statusCode());

Assertions.assertEquals("foo", response.jsonPath().getString("title"));

Assertions.assertEquals("baz", response.jsonPath().getString("body"));

Assertions.assertEquals("1", response.jsonPath().getString("userId"));

Assertions.assertEquals("1", response.jsonPath().getString("id"));

}

}

### PATCH Request

The PATCH request updates a resource but requires only the field(s) which is being updated in the payload:

import io.restassured.RestAssured;

import io.restassured.response.Response;

import org.junit.jupiter.api.Assertions;

import org.junit.jupiter.api.BeforeAll;

import org.junit.jupiter.api.Test;

import static io.restassured.RestAssured.given;

public class RestAssuredRequests {

private static String requestBody = "{\n" +

" \"title\": \"bax\" \n}";

@BeforeAll

public static void setup() {

RestAssured.baseURI = "https://jsonplaceholder.typicode.com";

}

@Test

public void patchRequest() {

Response response = given()

.header("Content-type", "application/json")

.and()

.body(requestBody)

.when()

.patch("/posts/1")

.then()

.extract().response();

Assertions.assertEquals(200, response.statusCode());

Assertions.assertEquals("bax", response.jsonPath().getString("title"));

Assertions.assertEquals("1", response.jsonPath().getString("userId"));

Assertions.assertEquals("1", response.jsonPath().getString("id"));

}

}

**Related:**

* [Difference between PUT and PATCH requests](https://devqa.io/difference-put-patch-requests/)

### DELETE Request

The DELETE request is used to delete a resource from a server.

To send a DELETE request in REST-assured, we use the delete() method:

import io.restassured.RestAssured;

import io.restassured.response.Response;

import org.junit.jupiter.api.Assertions;

import org.junit.jupiter.api.BeforeAll;

import org.junit.jupiter.api.Test;

import static io.restassured.RestAssured.given;

public class RestAssuredRequests {

@BeforeAll

public static void setup() {

RestAssured.baseURI = "https://jsonplaceholder.typicode.com";

}

@Test

public void deleteRequest() {

Response response = given()

.header("Content-type", "application/json")

.when()

.delete("/posts/1")

.then()

.extract().response();

Assertions.assertEquals(200, response.statusCode());

}

}

RestAssured with Raghav

For prctice : reqres.in (<https://reqres.in/api/users?page=2>)

//add Json pathFinder Chrome extension to the browser.or go to jsonpathfinder.com

Paste json response file to pathfinder> select item > copy path > then use in code.

Dependencies :

<dependency>

<groupId>io.rest-assured</groupId>

<artifactId>rest-assured</artifactId>

<version>5.2.0</version>

<scope>test</scope>

</dependency>

<dependency> [for json process]

<groupId>com.googlecode.json-simple</groupId>

<artifactId>json-simple</artifactId>

<version>1.1.1</version>

</dependency>

Step 1 : Create a class

Step 2 : Create a function and annotate with @Test (TestNG)

Step 3 : Run a GET request

Step 4 : Store response and print response data

Step 5 : Add assertions

Step 6 : Run and verify

**package** restAssredTests;

**import** org.testng.Assert;

**import** org.testng.annotations.Test;

**import** **static** org.hamcrest.Matchers.\*;

//import io.restassured.RestAssured; NB. imports are copy/paste from RestAssured Usage.

**import** **static** io.restassured.RestAssured.\*;

**import** io.restassured.response.Response;

**public** **class** TestExample {

@Test

**public** **void** test1() {

//Response response = RestAssured.get("https://reqres.in/api/users?page=2");

Response response = *get*("https://reqres.in/api/users?page=2");

**int** statusCode = response.getStatusCode();

**long** time = response.getTime();

System.***out***.println("status Code: "+statusCode);

System.***out***.println("status line: "+response.getStatusLine());

System.***out***.println("response time: "+time);

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

System.***out***.println("content type: "+response.getHeader("content-type"));

System.***out***.println("content type: "+response.getContentType());

Assert.*assertEquals*(statusCode, 200);

}

//add Json pathFinder Chrome extension to the browser to get Json file.

@Test

**public** **void** test2() {

//given().get("https://reqres.in/api/users?page=2")

*baseURI* = "https://reqres.in/api";

*given*()

.get("/users?page=2")

.then()

.statusCode(200)

.body("data[1].id", *equalTo*(8))

.log().all(); //.log().all() gets the whole response.

}

}

**How to use Get and Post method:**

**package** restAssredTests;

**import** **static** io.restassured.RestAssured.\*;

**import** **static** io.restassured.matcher.RestAssuredMatchers.\*;

**import** **static** org.hamcrest.Matchers.\*;

**import** java.util.HashMap;

**import** java.util.Map;

**import** org.json.simple.JSONObject;

**import** org.testng.annotations.Test;

**import** io.restassured.http.ContentType;

**public** **class** GetAndPostExample {

//@Test

**public** **void** GetExample() {

*baseURI* = "https://reqres.in/api";

*given*()

.get("/users?page=2")

.then()

.statusCode(200)

.body("data[4].first\_name", *equalTo*("George"))

.body("data.first\_name", *hasItems*("George", "Rachel"));

}

//@Test

**public** **void** PostExample() {

Map <String, Object > map = **new** HashMap <String, Object>();

map.put("name", "jawad");

map.put("job", "tester");

System.***out***.println(map);

JSONObject jsObject = **new** JSONObject(map);

System.***out***.println(jsObject);

System.***out***.println(jsObject.*toJSONString*(map));

}

@Test

**public** **void** jsonObjectExample() {

*baseURI* = "https://reqres.in/api";

JSONObject jsObject = **new** JSONObject();

jsObject.put("name", "jawad");

jsObject.put("job", "tester");

System.***out***.println(jsObject.toJSONString());

*given*()

.headers("Content-Type", "application/json")

.contentType(ContentType.***JSON***)

.accept(ContentType.***JSON***)

.body(jsObject.toJSONString())

.when()

.post("/users")

.then()

.statusCode(201)

.log().all();

//check HTTP statusCode https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

}

}

**Put, Patch, Delete method**

Sample project : GitHub - [https://github.com/Raghav-Pal/](https://www.youtube.com/redirect?event=video_description&redir_token=QUFFLUhqbUxHdlZTdDF0R0xMQnd5ME5oeFhBV2s0dlNpQXxBQ3Jtc0tsZjJwdW1aZ0VwYU42UVJTaXNKMFNFNlFLUExxTVpKZTBTYW1oS3RRVXdtTWx1VnI5Q2VXbXNneHdlRUtUN19PZGZURjhEaTE4eUNmcllOSzhzeW50QWs2Z2ZRRno2eXNfZi1vYkVZTExRTlBHVjdOWQ&q=https%3A%2F%2Fgithub.com%2FRaghav-Pal%2F&v=QDp7EQga3ME)

**package** restAssredTests;

**import** **static** io.restassured.RestAssured.*baseURI*;

**import** **static** io.restassured.RestAssured.\*;

**import** org.json.simple.JSONObject;

**import** org.testng.annotations.Test;

**import** io.restassured.http.ContentType;

**public** **class** PutPatchDeleteExample {

@Test

**public** **void** PUTexample() {

*baseURI* = "https://reqres.in/api"; //https://reqres.in/api/users/2

JSONObject jsObject = **new** JSONObject();

jsObject.put("name", "jawad");

jsObject.put("job", "tester");

System.***out***.println(jsObject.toJSONString());

*given*()

.headers("Content-Type", "application/json")

.contentType(ContentType.***JSON***)

.accept(ContentType.***JSON***)

.body(jsObject.toJSONString())

.when()

.put("/users/2")

.then()

.statusCode(200)

.log().all(); //check HTTP statusCode https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

}

@Test

**public** **void** PATCHexample() {

*baseURI* = "https://reqres.in/api"; //https://reqres.in/api/users/2

JSONObject jsObject = **new** JSONObject();

jsObject.put("name", "jawad");

jsObject.put("job", "tester");

System.***out***.println(jsObject.toJSONString());

*given*()

.headers("Content-Type", "application/json")

.contentType(ContentType.***JSON***)

.accept(ContentType.***JSON***)

.body(jsObject.toJSONString())

.when()

.patch("/users/2")

.then()

.statusCode(200)

.log().all(); //check HTTP statusCode https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

}

@Test

**public** **void** DELETEexample() {

*baseURI* = "https://reqres.in/api"; //https://reqres.in/api/users/2

*when*()

.delete("/users/2")

.then()

.statusCode(204)

.log().all();

//check HTTP statusCode https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

}

}

**Rest Assured by Pavan:**

API testing with Rest Assured using BDD approach…

For usage guide always check Rest Assured website…

url: <https://reqres.in/api/users?page=2>

= go to json pathFinder To find the JSON path..

> copy the json file and paste it to json pathfinder and get the path.

1. keywords used in BDD.

**validations:**

statusCode

statusLine

response body

header.

given ()

set coockies, add auth, add param, set headers info etc…

when ()

get, post, put, delete…..

then ()

validate status code, extract response, extract headers, coockies & response body…

**package** restAssuredTests;

**import** org.testng.annotations.Test;

**import** **static** io.restassured.RestAssured.\*;

**import** **static** io.restassured.matcher.RestAssuredMatchers.\*;

**import** **static** org.hamcrest.Matchers.\*;

/\*given ()

set coockies, add auth, add param, set headers info etc…

when ()

get, post, put, delete ...

then ()

validate status code, extract response, extract headers coockies & response body…

\*/

**public** **class** Get\_Request {

@Test

**public** **void** getWeatherDeatails() {

*given*()

.when()

.get("https://reqres.in/api/users?page=2")

.then()

.statusCode(200)

.statusLine("HTTP/1.1 200 OK")

.assertThat()

.body("first\_name", *equalTo*("Michael"))

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

}

}

**RestAssured by Mukesh**

**: What is API ?**

: REST stands for Representational State Transfer.

: Rest API is an architechtural syle (not protocol)

: Rest Assured is open source Java domain specific language(DSL)

: using Rest Assured we can eleminates large amount of code to test complex

API response and output.

: it supports both xml and Json format.

: Rest Assured is so easy even your grand mother can learn it.

**: Why API ?**

: testing the CORE functionality

: time effective/quick response time.

: language independent

: easy integration with GUI.

**: Rest Assured supports the requests\_**

.GET() will get the response from the server.

.POST() will create resources.

.PUT() will update resources.

.DELETE() will delete resources.

.HEAD() .PATCH() .OPTIONS() will make changes in the server.

Step 1. Create a maven project.

Step 2. Add rest-assured dependency, Json-simple

Step 3. Use the API URL that given to you.

**.GET()** will get the response from the server.

**package** getPackage;

**import** io.restassured.RestAssured;

**import** io.restassured.response.Response;

**public** **class** GetData {

@Test

**public** **void** testResponseCode1() {

Response resp = RestAssured.*get*("https://reqres.in/api/users?page=2");

**int** respCode = resp.getStatusCode();

System.***out***.println("response code : "+ respCode);

Assert.*assertEquals*(respCode, 200);

}

@Test

**public** **void** testBody1() {

Response resp = RestAssured.*get*("https://reqres.in/api/users?page=2");

String respData = resp.asString();

System.***out***.println("response data : "+ respData);

**long** respTime = resp.getTime();

System.***out***.println("response time : "+ respTime);

}

**public** **class** GetData2 {

**import** **static** io.restassured.RestAssured.\*;

**import** io.restassured.response.Response;

// if I use static rest-assured \* then no need to write RestAssured.get()

@Test

**public** **void** testResponseCode() {

Response resp = *get*("https://reqres.in/api/users?page=2");

**int** respCode = resp.getStatusCode();

System.***out***.println("response code : "+ respCode);

Assert.*assertEquals*(respCode, 200);

}

@Test

**public** **void** testBody() {

Response resp = *get*("https://reqres.in/api/users?page=2");

String respData = resp.asString();

System.***out***.println("response data : "+ respData);

**long** respTime = resp.getTime();

System.***out***.println("response time : "+ respTime);

}

@Test

**public** **void** testBody2() {

String respData = *get*("https://reqres.in/api/users?page=2").asString();

System.***out***.println("response data : "+ respData);

**long** respTime = *get*("https://reqres.in/api/users?page=2").getTime();

System.***out***.println("response time : "+ respTime);

}

}

**.GET() .PUT() .DELETE()** will not create additional load on the server while making multiple calls.

**.POST()** will create additional load on server if made multiple calls.

**.GET()** method simply retrieves data from server. Doesn’t make any changes to server/resources.

Example: check any news/anything on the web and so on.

**.POST()** method perform the changes to the server. It always creates resources in server.

Example: uploading a picture, send a tweet or submitting a form and so on.

**Note: POST()** is NOT idempotent. So if you retry N times, you will end up having N resources with N different URIs created on server.

**.DELETE()** method simply delete the data/resources from the server.

Example: deleting any file, phtotos from fb/any account and so on.

**.PUT()** method can update Existing file/resources on server. If you call multiple times, it will overrite the file. Will not create additional file.

**What are API parameters?**

API parameters are the variable parts of a resource. They determine the type of action you want to take on the resource. Each parameter has a name, value type and optional description.

**Types of REST API Parameters :**

There are four different parts of parameters which are often documented in separate groups on the same page. They include:

* Header parameters – These parameters are featured in the request header and are usually related to authorization.
* [Query parameters](https://rapidapi.com/blog/api-glossary/parameters/query/) – These are separated from the hierarchy parameters by a question mark. It is used to fetch a specific record from the resource.
* Request body parameters – they are included in the request body and are used to send and receive data via the REST API.
* [Template/Path parameters](https://rapidapi.com/blog/api-glossary/parameters/path/) – Set off within curly braces, path parameters are placed within the path of an endpoint just before the query string, and they provide developers with an effective way to parameterize resources.
* Matrix parameters – They come in between the resource path and Query parameters and are separated from the hierarchy parameters by a semicolon.
* Plain Parameters – These are parameters which are defined in a request and are easily accessible in ReadyAPI but are omitted when the request is submitted.

**use Json-simple to format the data....**

**How to perform post, put, delete method....**

**package** restAPI;

**import** org.json.simple.JSONObject;

**import** io.restassured.RestAssured;

**import** io.restassured.response.Response;

**import** io.restassured.specification.RequestSpecification;

**public** **class** Post\_Delete\_Put {

@Test

**public** **void** testPOST() {

//create a specification for .given() request

RequestSpecification request = RestAssured.*given*();

//set the header() request

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

//send .put() request in json format to update "name", "job" in json

format.

JSONObject jsonObject = **new** JSONObject();

jsonObject.put("name", "jawad");

jsonObject.put("job", "tester");

//send .body() request to get the body in jsonString format

request.body(jsonObject.toJSONString());

// send post request and store the response in Response object.

Response response = request.post("https://reqres.in/api/users");

**int** statusCode = response.getStatusCode();

System.***out***.println("statusCode : "+statusCode); //201

Assert.*assertEquals*(statusCode, 201);

}

@Test

**public** **void** testDELETE() {

//create a specification for .given() request

RequestSpecification request = RestAssured.*given*();

// send delete request and store the response in Response object.

Response response = request.delete("https://reqres.in/api/users/2");

**int** statusCode = response.getStatusCode();

System.***out***.println("statusCode : "+statusCode); //204

Assert.*assertEquals*(statusCode, 204);

}

@Test

**public** **void** testPUT() {

//create a specification for .given() request

RequestSpecification request = RestAssured.*given*();

//set the header() request

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

//send .put() request in json format to update "name", "job" in json format.

JSONObject jsonObject = **new** JSONObject();

jsonObject.put("name", "jawad");

jsonObject.put("job", "tester");

//send .body() request to get the body in jsonString format

request.body(jsonObject.toJSONString());

// send post request and store the response in Response object.

Response response = request.put("https://reqres.in/api/users/2");

**int** statusCode = response.getStatusCode();

System.***out***.println("statusCode : "+statusCode); //200

Assert.*assertEquals*(statusCode, 200);

}

}

**How handle Authentication.....**

: Authentication is process to prove that you are the Authentic person or not.

: RestAssured supports multiple authentication schemes.

Example: Auth, digest, certificate, form based and preemptive basic authentication.

**preemptive basic authentication** will send the basic authentication credential even before the server gives unauthorised response in certain situations. It sends the authentication along with request/calls.

thus reducing the overhead of making an additional connection.

: Challenge with basic authentication- will not send the authentication with request/calls. Will need an additional request/call for authentication, while **preemptive basic authentication** will send authentication along with one request/authentication.

**package** restAPI;

**import** org.testng.annotations.Test;

**import** io.restassured.RestAssured;

**public** **class** Authentication {

@Test

**public** **void** test1() {

**int** statusCode = RestAssured.*given*()

.auth().preemptive()

.basic("ToolsQA", "TestPassword")

.when()

.get("https://restapi.demoqa.com/authentication/CheckForAuthentication/")

.statusCode();

System.***out***.println("statusCode : "+ statusCode); //200

}

}

**Set authentication in base class instead using in every class..**

**package** restAPI;

**import** org.testng.annotations.BeforeClass;

**import** io.restassured.RestAssured;

**public** **class** BaseClass4authentication {

@BeforeClass

**public** **void** setupAuthentication() {

RestAssured.*authentication* = RestAssured.*preemptive*().basic("ToolsQA", "TestPassword");

RestAssured.*baseURI* = "https://restapi.demoqa.com/authentication/CheckForAuthentication/";

}

}

**package** restAPI;

**import** org.testng.annotations.Test;

**import** io.restassured.RestAssured;

**public** **class** AuthenticationWithBaseClass **extends** BaseClass4authentication {

@Test

**public** **void** test1() {

**int** statusCode = RestAssured.*given*()

.get()

.statusCode();

System.***out***.println("statusCode : "+ statusCode); //200

}

}

---------------------------------------------------------------

**What is OAuth ?**

: OAuth is an open standard for authorization protocol.

: it is widely used for web application, web services to get user information authorised by the user.

: it allow you to access particular resources based on your roles and responsibility. you don’t need to provide those services your user credentials like id, passwords.

**package** restAPI;

**import** io.restassured.RestAssured;

**import** io.restassured.response.Response;

**: OAuth 1 example**

**public** **class** OAth1 {

**public** **void** testOAth1() {

Response resp = RestAssured.*given*()

.auth()

.oauth("hlaioeiukld984rklji945lijo9jp",

"kdjfioajpiijio7ij9jklji7y6ijy80790jl",

"ksjjihgldri7i4;l-0tkjha9ejkfh8e9j9p9uuhkl",

"hlkfguopirtjjfd8034o58756kljkjgkshg0kg94uj")

.post("https://reqres.in/register/update.json?status = this post is via api");

System.***out***.println("status code : "+ resp.getStatusCode());

System.***out***.println(resp.getBody().jsonPath().prettify());

}

**: OAuth 2 example**

**public** **void** testOAth2() {

Response resp = RestAssured.*given*()

.auth()

.oauth2("")

.post("");

System.***out***.println("code : "+ resp.getStatusCode());

System.***out***.println("code : "+ resp.getBody().asString());

}

}

Rest Assured ToolsQA

### ****What is a Client ?****

A client in computing is a system or a program that connects with a remote system or software to fetch information. The client makes a request to the server and is responded with information. There may be three types of clients - thick, thin or hybrid client.

**what is a Server ?**

A server is a system or a computer program that acts as the data provider. It may provide data through LAN(*Local Area Network*) or WAN(*Wide Area Network*) using the internet. The functionalities provided by the server are called services. These services are provided as a response to the request made by the clients. Some of the common servers are-

* ***Database Server****-  used to maintain and share the database over a network.*
* ***Application Server****-  used to host applications inside a web browser allowing to use them without installation locally.*
* ***Mail Server****-  used for email communication.*
* ***Web Server****-  used to host web pages because of which worldwide web is possible.*
* ***Gaming Server****-  used for playing multiplayer games.*
* ***File Server****- used for sharing files and folders over a network.*

**What is an HTTP Request ?**

client's request is an HTTP Request, which communicates between the client and the server, or you may say, two different computer systems.

### W****hat are the different HTTP Request methods?****

HTTP request methods specify the action to perform through the request. These are also known as verbs and generally used for ***CRUD operations***, i.e., Create, Read, Update & Delete.

**1.** ***GET -*** As the name suggests, the Get method fetches the information from the server. Moreover, it is the most commonly used method which does not have a request body. Every time you open a website, the Get request fires to retrieve the website contents. Additionally, it is equivalent to the read operation.

**2.** ***HEAD:*** The Head method is similar to the Get method, but it retrieves only the header data and not the entire response body. Moreover, we use it when you need to check the document's file size without downloading the document.

**3.** ***POST:*** The Post method works to send data to the server. You may add the data using the Post request. We send the information that needs to update in the request body. In the real world, the form data on website updates using the Post request.

**4.** ***PUT:*** The Put method is similar to the Post method since it updates the data. The only difference is that we use it when we have to replace an existing entity completely.

**5.** ***PATCH:*** This method is again similar to Post and Put methods, but we use it when we have to update some data partially.

**6.** ***DELETE:*** Like its name, the Delete method deletes the server's representations of resources through the specific URL. Additionally, just like the Get method, they do not have a request body.

**7.** ***OPTIONS:*** This is not a widely used method when compared to other ones. It returns data specifying the different methods and the operations supported by the server at the given URL. Moreover, it responds with an Allow header giving a list of the HTTP methods allowed for the resource.

# What is HTTP Response?

Every HTTP Response received as a result of an HTTP request sent by the client to the server has a status code. This status code value tells us if HTTP Response was successful or not.

An HTTP response object typically represents the HTTP packet (response packet) sent back by Web Service Server in response to a client request. An HTTP Response contains:

1. A status.
2. Collection of Headers.
3. A Body.

As we already know the same REST API returns a response message in XML or JSON format.

But then how will the client know what type of response it will get from the API? Well, this is managed by the response headers. Response Header contains a **Content-Type** attribute that informs about the type of response body format.

the response has a status, headers, and a body. If we check the "Response headers" section, in the above screen, it has a content-type attribute that has the value along with other attributes. On validating this header, the client knows what type of response (body) we can expect.

### ****How to validate HTTP response status code****

When the client requests a piece of particular information from the server, the server sends a response with a status code back to the client. The status code that the server returns tells us whether the request was successful or not.

**package** pages;

**public** **class** RestAssuredTestResponse {

@Test **public** **void** GetBookDetails() {

// Specify the base URL to the RESTful web service

RestAssured.baseURI = "https://demoqa.com/BookStore/v1/Books";

// Get the RequestSpecification of the request to be sent to the server

RequestSpecification httpRequest = RestAssured.given();

Response response = httpRequest.get("");

// Get the status code of the request.

//If request is successful, status code will be 200

**int** statusCode = response.getStatusCode();

// Assert that correct status code is returned.

Assert.assertEquals(statusCode /\*actual value\*/, 200 /\*expected value\*/,

"Correct status code returned");

}

}

### ****How to validate the HTTP error status code?****

So far the request-response situations are all good and we have only received 200 status codes indicating successful requests. But this may not be always true in the real world. There can be reasons like the server is down or REST API not functioning properly or the requests themselves may be problematic. In conclusion, we may face the following scenarios:

1. The server hosting REST API is down.
2. Incorrect client request.
3. The resource requested by the client does not exist.
4. An error occurs on the server side during the processing of the request.

So when any of the above scenarios occur, the REST API will return an appropriate status code other than 200. The client in turn has to validate this status code and process it accordingly.

**package** pages;

**public** **class** RestAssuredTestResponse {

@Test

**public** **void** GetPetDetails()

{

// Specify the base URL to the RESTful web service

RestAssured.baseURI = "https://demoqa.com/Account/v1/User/";

// Get the RequestSpecification of the request to be sent to the server

RequestSpecification httpRequest = RestAssured.given();

Response response = httpRequest.get("test");

// Get the status code of the request.

//If request is successful, status code will be 200

**int** statusCode = response.getStatusCode();

// Assert that correct status code is returned.

Assert.assertEquals(statusCode /\*actual value\*/, 200 /\*expected value\*/,

"Correct status code returned");

}

}

When we run this test it returns the error code of 401.

Note: We can make a quick change to the code above to make sure the test passes. This change is shown below:

java Assert.assertEquals(statusCode /*actual value*/, 401 /*expected value*/, "Correct status code returned");

So here we expect the value returned to be 401 instead of 200, hence the test is passed. Next, we will validate the *"Status line"*.

**How to validate the response status line?**

A *"Status-Line"* is the first line returned in the HTTP response. The status line consists of three substrings:

* HTTP protocol version.
* Status Code.
* Status Code’s string value.

For example, when the request is successful the status line will have the value *"HTTP/1.1 200 OK".* Here, the first part is the HTTP protocol *(HTTP/1.1).* Next is the HTTP status code *(200).* The third is the status message *(OK).*

**package** pages;

**public** **class** RestAssuredTestResponse {

@Test **public** **void** GetBookDetails() {

// Specify the base URL to the RESTful web service

RestAssured.baseURI = "https://demoqa.com/BookStore/v1/Books";

// Get the RequestSpecification of the request to be sent to the server

RequestSpecification httpRequest = RestAssured.given();

Response response = httpRequest.get("");

// Get the status line from the Response 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");

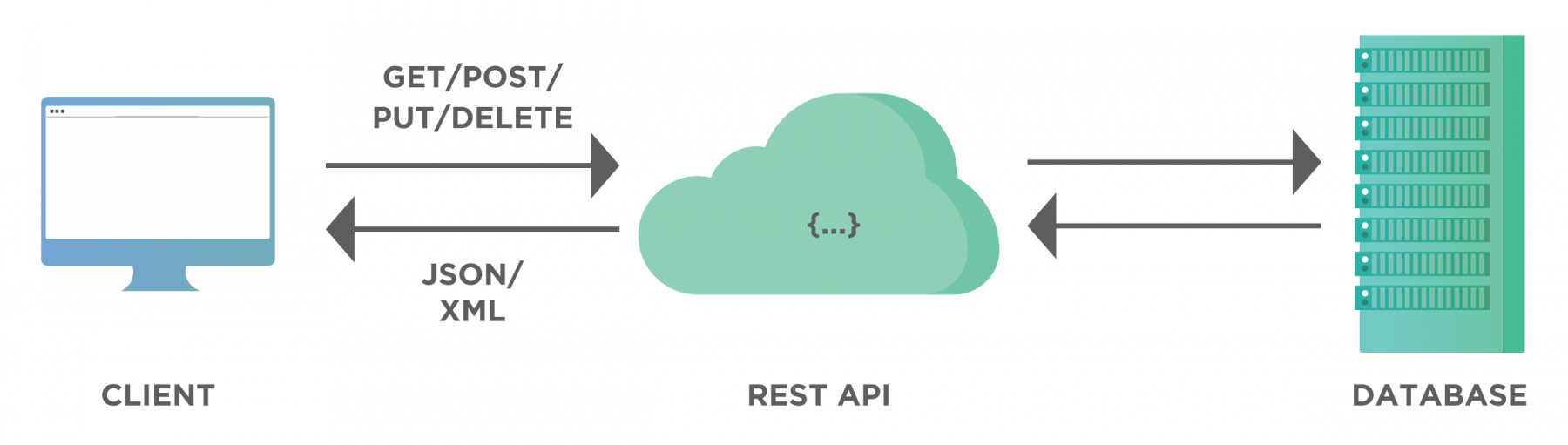
}

## What is REST - Representational State Transfer?

REST (***RE***presentational ***S***tate ***T***ransfer) was first presented in the year 2000 by [***Roy Fielding***](https://en.wikipedia.org/wiki/Roy_Fielding) as an architectural style for ***distributed hypermedia systems.*** REST-compliant or RESTful systems, are ***"stateless"*** (discussed later in this article) and separate a client and a server.

### *****Representation of REST flow*****

Now let us depict the actual REST data transfer in the above example in the diagram below.



**what is REST API testing?**

REST API testing is a technique to test RESTful APIs and validate their correctness. We send the request (*preferably using automation*) and record the response for further assertions. This way we can check if the REST API is working fine or not. REST API testing is mainly done using four REST methods, viz, GET, POST, PUT, DELETE.

## HTTP Methods and status codes for REST API Automation Testing

We know that REST API uses five HTTP methods to request a command:

| ***Method*** | ***Description*** |
| --- | --- |
| **GET** | Retrieves the information at a particular URL. |
| ***PUT*** | Updates the previous resource if it exists or creates new information at a particular URL. |
| **POST** | Used to send information to the server like uploading data and also to develop a new entity. |
| **DELETE** | Deletes all current representations at a specific URL. |
| **PATCH** | This is used for partial updates of resources. |

Once the request is sent using the above methods, the client receives the numeric codes known as ***"Status codes"*** or sometimes referred to as ***"Response codes"***. Then we can interpret these status codes to know what kind of response the server has sent for a particular request.  Status codes are mainly classified into five categories as shown in the table below.

| ***No*** | ***Status Code*** | ***Description*** |
| --- | --- | --- |
| 1 | 1xx (100 – 199) | The response is informational. |
| 2 | 2xx (200 – 299) | Assures successful response. |
| 3 | 3xx (300 – 399) | You are required to take further action to fulfil the request. |
| 4 | 4xx (400 – 499) | There’s a bad syntax and the request cannot be completed. |
| 5 | 5xx (500 – 599) | The server entirely fails to complete the request. |

## REST API test using Rest Assured

[***REST Assured***](https://toolsqa.com/rest-assured/rest-assured-library/) is a Java library for testing RESTful APIs. It is widely used to test JSON and XML-based web applications. In addition, it fully supports all REST methods like the GET, PUT, POST, PATCH, and DELETE. Next, we will see a detailed walkthrough of testing one REST API using the Rest Assured library.

### *****How to write REST API test using Rest Assured?*****

To write a sample REST API test we will make use of the following REST API link.

|  |  |
| --- | --- |
| Request URL | [**https://demoqa.com/BookStore/v1/Books**](https://demoqa.com/BookStore/v1/Books) |
| HTTP Method | ***GET*** |
| Comments | This URL will return the inventory details of a Book store. There are no input parameters for the request. |
| Response | {"books": [{"isbn": "string","title": "string","subTitle": "string","author":"string","publish\_date": "2022-01-25T13:44:50.276Z","publisher": "string","pages": 0,"description": "string","website": "string"}]} |

public class RestAssuredAPITest {

@Test

public void GetBooksDetails() {

// Specify the base URL to the RESTful web service

RestAssured.baseURI = "https://demoqa.com/BookStore/v1/Books";

// Get the RequestSpecification of the request to be sent to the server.

RequestSpecification httpRequest = RestAssured.given();

// specify the method type (GET) and the parameters if any.

//In this case the request does not take any parameters

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

// Print the status and message body of the response received from the server

System.out.println("Status received => " + response.getStatusLine());

System.out.println("Response=>" + response.prettyPrint());

}

}

: Get the resonse in String format.

@Test

public void GetWeatherDetailsCondensed()

{

// Specify the base URL to the RESTful web service

RestAssured.baseURI = "https://demoqa.com/BookStore/v1/Books";

// Get the RequestSpecification of the request that is to be sent

// to the server.

RequestSpecification httpRequest = RestAssured.given();

// Call RequestSpecification.get() method to get the response.

// Make sure you specify the resource name.

Response response = httpRequest.get("");

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

// as String.

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

}

## Validate HTTP Response Status using Rest Assured

An HTTP response object typically represents the HTTP packet (response packet) sent back by Web Service Server in response to a client request. An HTTP Response contains:

1. A status.
2. Collection of Headers.
3. A Body.

We have a detailed article on HTTP Response [***here.***](https://www.toolsqa.com/client-server/http-response/)

So when we say we need to validate HTTP response status, we are looking forward to having a mechanism to read and validate the entire response object including the status, headers, and the body. Hence, we will validate each of the HTTP response components separately. So in this article, the validation of an HTTP response status will be dealt with in three parts as follows:

* Validating HTTP Response Status Code.
* How to validate the Error Status Code
* Validating Response Status Line.

As we already know the same REST API returns a response message in XML or JSON format. This format depends on the ***Media-Type*** attribute in the HTTP request.

But then how will the client know what type of response it will get from the API? Well, this is managed by the response headers. Response Header contains a ***Content-Type*** attribute that informs about the type of response body format.

public class RestAssuredTestResponse {

@Test

public void GetBookDetails()

{

// Specify the base URL to the RESTful web service

RestAssured.baseURI = "https://demoqa.com/BookStore/v1/Books";

// Get the RequestSpecification of the request to be sent to the server

RequestSpecification httpRequest = RestAssured.given();

Response response = httpRequest.get("");

// Get the status code of the request.

//If request is successful, status code will be 200

int statusCode = response.getStatusCode();

// Assert that correct status code is returned.

Assert.assertEquals(statusCode /\*actual value\*/, 200 /\*expected value\*/,

"Correct status code returned");

}

}

### *****How to validate the HTTP error status code?*****

here we provide the parameter to get user details. Here we provide nonexistent userId as the parameter. The code looks as below:

public class RestAssuredTestResponse {

@Test

public void GetPetDetails()

{

// Specify the base URL to the RESTful web service

RestAssured.baseURI = "https://demoqa.com/Account/v1/User/";

// Get the RequestSpecification of the request to be sent to the server

RequestSpecification httpRequest = RestAssured.given();

Response response = httpRequest.get("test");

// Get the status code of the request.

//If request is successful, status code will be 200

int statusCode = response.getStatusCode();

// Assert that correct status code is returned.

Assert.assertEquals(statusCode /\*actual value\*/, 200 /\*expected value\*/,

"Correct status code returned");

}

}

When we run this test it returns the error code of 401.

Note: We can make a quick change to the code above to make sure the test passes. This change is shown below:

Assert.assertEquals(statusCode /\*actual value\*/, 401 /\*expected value\*/, "Correct status code returned");

So here we expect the value returned to be 401 instead of 200, hence the test is passed. Next, we will validate the ***"Status line"***.

***How to validate the response status line?***

A ***"Status-Line"*** is the first line returned in the HTTP response. The status line consists of three substrings:

* *HTTP protocol version.*
* *Status Code.*
* *Status Code’s string value.*

For example, when the request is successful the status line will have the value ***"HTTP/1.1 200 OK".*** Here, the first part is the HTTP protocol ***(HTTP/1.1).*** Next is the HTTP status code *(200).* The third is the status message ***(OK).***

@Test

public void GetBookDetails()

{

// Specify the base URL to the RESTful web service

RestAssured.baseURI = "https://demoqa.com/BookStore/v1/Books";

// Get the RequestSpecification of the request to be sent to the server

RequestSpecification httpRequest = RestAssured.given();

Response response = httpRequest.get("");

// Get the status line from the Response 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");

}

## What is an HTTP Response Header in REST API?

The response received from the server consists of zero or more headers along with response status and response body. Each header is a key-value pair. The header part of the response is used by the server to send extra information.

For example, headers contain a "***Content-Type***" attribute that tells us how to interpret the data of the response body. So if the response body contains JSON data, then the corresponding content-type attribute in the header will be "***application/json***". Similarly, if the data in the body is XML the ***Content-Type*** header will be "***application/xml***".

GET "https://demoqa.com/BookStore/v1/Books" -H "accept: application/json"

Note the response header that is obtained (red rectangle). Since the body is JSON, the ***Content-Type*** is set to "***application/JSON***".

the Response interface of REST Assured provides methods related to headers.

* ***headers()*** : returns ***Headers***
* ***getHeader()*** : returns a ***Header***
* ***getHeaders()*** : returns ***Headers***

When all the headers in a Response are returned, we can print each header by simply iterating over each of them.

## How to access and read HTTP Response headers using REST Assured?

Now let us see how we can read a Header using Rest-Assured. Let’s write a test to record the following Header Types from the Response:

* Content-Type.
* Server.
* Content-Encoding.

Shown below is the code for this test:

@Test

public void IteratingHeaders()

{ RestAssured.baseURI = "https://demoqa.com/BookStore/v1/Books";

RequestSpecification httpRequest = RestAssured.given();

Response response = httpRequest.get("");

// Get all the headers and then iterate over allHeaders to print each header

Headers allHeaders = response.headers();

// Iterate over all the Headers

for(Header header : allHeaders) {

System.out.println("Key: " + header.getName() + " Value: " + header.getValue());

}

}

@Test

public void GetBookHeaders() {

RestAssured.baseURI = "https://demoqa.com/BookStore/v1/Books";

RequestSpecification httpRequest = RestAssured.given();

Response response = httpRequest.get("");

// Access header with a given name.

String contentType = response.header("Content-Type");

System.out.println("Content-Type value: " + contentType);

// Access header with a given name.

String serverType = response.header("Server");

System.out.println("Server value: " + serverType);

// Access header with a given name. Header = Content-Encoding

String acceptLanguage = response.header("Content-Encoding");

System.out.println("Content-Encoding: " + acceptLanguage);

}

## How to validate HTTP Response Header using Rest Assured?

@Test

public void ValidateBookHeaders()

{

RestAssured.baseURI = "https://demoqa.com/BookStore/v1/Books";

RequestSpecification httpRequest = RestAssured.given();

Response response = httpRequest.get("");

// Access header with a given name. Header = Content-Type

String contentType = response.header("Content-Type");

Assert.assertEquals(contentType /\* actual value \*/, "application/json; charset=utf-8" /\* expected value \*/);

// Access header with a given name. Header = Server

String serverType = response.header("Server");

Assert.assertEquals(serverType /\* actual value \*/, "nginx/1.17.10 (Ubuntu)" /\* expected value \*/);

In the above code, we verify the actual value of each header viz., Content-Type, Server, and Content-Encoding with the expected value. Shown below is the screenshot of the test result.

Now suppose we provide the expected value of Content-Type to "***application/XML***" in the above code as below.

Assert.assertEquals(contentType /\* actual value /, "application/xml" / expected value \*/);

# Read JSON Response Body using Rest Assured

Let us continue with the example of Weather web service that we used in the previous tutorials. 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***

@Test

public void WeatherMessageBody()

{

RestAssured.baseURI = "https://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.

### *****How to Validate Response Body contains some String?*****

***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 = "https://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***

@Test

public void WeatherMessageBody()

{

RestAssured.baseURI = "https://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");

}

### *****How to Extract a Node text from Response 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"

}

@Test

public void VerifyCityInJsonResponse()

{

RestAssured.baseURI = "https://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 jpath = 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 = jpath.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");

}

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

@Test

public void DisplayAllNodesInWeatherAPI()

{

RestAssured.baseURI = "https://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 jsnPath = response.jsonPath();

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

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

// Print the temperature node

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

// Print the humidity node

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

// Print weather description

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

// Print Wind Speed

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

// Print Wind Direction Degree

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

}

## What are Query String Parameters?

You might not always want to fetch all the results corresponding to a request. There may be scenarios where you need to fetch only a few or a single record. In such cases, query string parameters play an important role. These are appended at the end of the URL after using a '?'. Try entering the [***example URL***](https://bookstore.toolsqa.com/BookStore/v1/Book?ISBN=9781449325862) in the browser address bar and observe the results available under Network -> Payload-

## How to send a request using Query Parameters in Rest Assured?

public class QueryParam {

@Test

public void queryParameter() {

//Defining the base URI

RestAssured.baseURI= "https://bookstore.toolsqa.com/BookStore/v1";

RequestSpecification httpRequest = RestAssured.given();

//Passing the resource details

Response res = httpRequest.queryParam("ISBN","9781449325862").get("/Book");

//Retrieving the response body using getBody() method

ResponseBody body = res.body();

//Converting the response body to string object

String rbdy = body.asString();

//Creating object of JsonPath and passing the string response body as parameter

JsonPath jpath = new JsonPath(rbdy);

//Storing publisher name in a string variable

String title = jpath.getString("title");

System.out.println("The book title is - "+title);

}

}

What is an HTTP Post Request method ?

when we are submitting any registration form on a particular webpage like Gmail. We provide the required data and click submit. So through this action of submitting data, we are actually POSTING or sending the data to the server.

 On the server side, the new user will be created with all the provided information and a response will be sent back to the client.

### *****Create a Request pointing to the Service Endpoint*****

RestAssured.baseURI = "https://demoqa.com/BookStore/v1/Books";

RequestSpecification request = RestAssured.given();

### *****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("userId", "TQ123");

requestParams.put("isbn", "9781449325862");

### *****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

### *****Validate the Response*****

Response response = request.post("/BookStoreV1BooksPost");

System.out.println("The status received: " + response.statusLine());

what happens when we change the HTTP Request method on a POST request? For example what happens when instead of the expected POST we send the GET? Let's discuss this scenario.

Following is the code wherein we have sent a GET request to an Endpoint when it actually expects POST.

public void UserRegistrationSuccessful()

{

RestAssured.baseURI ="https://demoqa.com/Account/v1";

RequestSpecification request = RestAssured.given();

JSONObject requestParams = new JSONObject();

requestParams.put("userName", "test\_rest");

requestParams.put("password", "Testrest@123");

request.body(requestParams.toJSONString());

Response response = request.put("/User");

ResponseBody body = response.getBody();

System.out.println(response.getStatusLine());

System.out.println(body.asString());

}

We can clearly see the output says the incorrect usage of the HTTP Request Method. Similarly, we have other negative scenarios listed below which we will leave to users to try themselves.

You can try the above scenarios on the same URL used above to demonstrate the POST request.

## Serialization and Deserialization in Java

### *****What is Serialization?*****

Serialization is a process where you convert an Instance of a Class (Object of a class) into a Byte Stream. This Byte Stream can then be stored as a file on the disk or can also be sent to another computer via the network.

public class Rectangle {

private double height;

private double width;

public Rectangle(double height, double width)

{

this.height = height;

this.width = width;

}

public double Area()

{

return height \* width;

}

public double Perimeter()

{

return 2 \* (height + width);

}

}

***Note:****This class is not yet Serializable as per Java standards, let us ignore it for the time being.*

***Serializable Interface***

In Java, a **Serializable** object is an object which inherits from either of the two interfaces

* [***java.io.Serializable***](https://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html)
* [***java.io.Externalizable***](https://docs.oracle.com/javase/6/docs/api/java/io/Externalizable.html)

Serializable interface is a marker interface. Which means that you do not have to implement any methods if your class derives from this interface. This is just a marker and the Java runtime, when trying to Serialize the class, will just check for the presence of this interface in the class. If Serializable interface is present in the class inheritance hierarchy, Java run time will take care of Serialization of the class.

On the other hand, the Externalizable interface is not a marker interface. If you derive from Externalizable interface you have to implement these two methods

* ***readExternal(ObjectInput input)***
* ***writeExternal(ObjectOutput output)***

We should inherit from Externalizable interface only when we want to overtake the Java's default serialization mechanism. If you want to use the default Java's serialization mechanism than you should inherit from Serializable interface only.

With this understanding, our Rectangle class will now inherit from Serializable interface.

public class Rectangle implements Serializable{

private double height;

private double width;

public Rectangle(double height, double width)

{

this.height = height;

this.width = width;

}

public double Area()

{

return height \* width;

}

public double Perimeter()

{

return 2 \* (height + width);

}

}

### *****Serializing an Object in Java*****

public static void SerializeToFile(Object classObject, String fileName)

{

try {

// Step 1: Open a file output stream to create a file object on disk.

// This file object will be used to write the serialized bytes of an object

FileOutputStream fileStream = new FileOutputStream(fileName);

// Step 2: Create a ObjectOutputStream, this class takes a files stream.

// This class is responsible for converting the Object of any type into

// a byte stream

ObjectOutputStream objectStream = new ObjectOutputStream(fileStream);

// Step 3: ObjectOutputStream.writeObject method takes an Object and

// converts it into a ByteStream. Then it writes the Byte stream into

// the file using the File stream that we created in step 1.

objectStream.writeObject(classObject);

// Step 4: Gracefully close the streams

objectStream.close();

fileStream.close();

} catch (FileNotFoundException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} catch (IOException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

public static void main(String[] args)

{

Rectangle rect = new Rectangle(18, 78);

SerializeToFile(rect, "rectSerialized");

}

### *****Deserializing to an Object in Java*****

public static Object DeSerializeFromFileToObject(String fileName)

{

try {

// Step 1: Create a file input stream to read the serialized content

// of rectangle class from the file

FileInputStream fileStream = new FileInputStream(new File(fileName));

// Step 2: Create an object stream from the file stream. So that the content

// of the file is converted to the Rectangle Object instance

ObjectInputStream objectStream = new ObjectInputStream(fileStream);

// Step 3: Read the content of the stream and convert it into object

Object deserializeObject = objectStream.readObject();

// Step 4: Close all the resources

objectStream.close();

fileStream.close();

// return the deserialized object

return deserializeObject;

} catch (FileNotFoundException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} catch (IOException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} catch (ClassNotFoundException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

return null;

}

public static void main(String[] args)

{

Rectangle rect = new Rectangle(18, 78);

SerializeToFile(rect, "rectSerialized");

Rectangle deSerializedRect = (Rectangle) DeSerializeFromFileToObject("rectSerialized");

System.out.println("Rect area is " + deSerializedRect.Area());

}

# Deserialize JSON Response using Rest Assured

JSON is an extremely popular format when it comes to APIs. Almost all of the APIs either transfer data in the XML format or JSON format of which JSON is a popular one.

***Serialization*** and ***Deserialization*** are programming techniques where we convert ***Objects*** to ***Byte Streams*** and from ***Byte Streams*** back to ***Objects*** respectively.

## What is Serializing a JSON?

As far as [***rest assured***](https://www.toolsqa.com/rest-assured-tutorial/) is concerned, we are aware that the data exchange between client and server takes place in JSON format by REST web service. The stream of data here is JSON data.

For example, if we have the following object:

{tools: [1, 3, 7, 9], api: "QA"}

when the above object is serialized into JSON, the output will look like the one shown below:

{

"tools":[1, 3, 7, 9],

"api":"QA"

}

## What is Deserializing a JSON Response?

when we say we are deserializing the JSON, this means we convert the JSON format into a type we prefer, most frequently [***POJO***](https://en.wikipedia.org/wiki/Plain_old_Java_object) (Plain Old Java Object) classes.

## How to Deserialize JSON Response to Class with Rest Assured?

: serialized json

{

"SuccessCode": "OPERATION\_SUCCESS",

"Message": "Operation completed successfully"

}

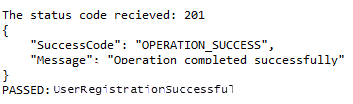
In the POST request example, we have used ***JSONPath*** to validate the response body parts. Now we have to convert this response body to a Java class (POJO). In other words, we'll convert the JSON which is a string form to a class form i.e. deserialize JSON.

**So how do we begin with deserialization?**

First and foremost we need to create a class that has all the nodes *(or key)* of the JSON response. As we are already aware, the Success response has two nodes:

* ***SuccessCode***
* ***Message***

Both these nodes contain String values. The screenshot below shows the part of the response we receive.



So we have to create a class with two string variables that will represent nodes in the JSON. Given below is the class code for the same.

@Test

public class JSONSuccessResponse {

// Note: The name should be exactly as the JSON node name

// Variable SuccessCode will contain value of SuccessCode node

public String SuccessCode;

// Variable Message will contain the value of Message node

public String Message;

}

### *****Converting JSON Response Body to JSONSuccessResponse class*****

@Test

public void UserRegistrationSuccessful() {

RestAssured.baseURI ="https://demoqa.com";

RequestSpecification request = RestAssured.given();

JSONObject requestParams = new JSONObject();

requestParams.put("UserName", "test\_rest");

requestParams.put("Password", "rest@123");

request.body(requestParams.toJSONString());

Response response = request.post("/Account/v1/User");

ResponseBody body = response.getBody();

// Deserialize the Response body into JSONSuccessResponse

JSONSuccessResponse responseBody = body.as(JSONSuccessResponse.class);

// Use the JSONSuccessResponseclass instance to Assert the values of Response.

Assert.assertEquals("OPERATION\_SUCCESS", responseBody.SuccessCode);

Assert.assertEquals("Operation completed successfully", responseBody.Message); }

Once we get the value in responseBody variable, we can validate the response as shown in the code below.

// Use the RegistrationSuccessResponse class instance to Assert the values of

// Response.

Assert.assertEquals("OPERATION\_SUCCESS", responseBody.SuccessCode);

Assert.assertEquals("Operation completed successfully", responseBody.Message);

In this way, we can apply assertions to validate the response or even pass this response as input to other tests.

## How to Deserialize JSON Response Body based on Response Status?

One such format of failed response may be as shown below:

{

"FaultId": "User already exists",

"fault": "FAULT\_USER\_ALREADY\_EXISTS"

}

If we use the class ***JSONSuccessResponse*** to deserialize the above response, it will not work. This is because Rest Assured will not find the nodes ***SuccessCode*** and ***Message*** in the response body like in the above section. These two variables in the class will have values as null.

The solution to this is to maintain another class that will be used for deserializing the failure response. This class will have the following structure.

public class JSONFailureResponse {

String FaultId;

String fault;

}

But now that we have two classes, one for a successful response and another for failure, how can we handle both these in an application?

We can do this using the [***HTTP Status Code***](https://www.toolsqa.com/client-server/http-response/) returned by the server. Rest API in this series returns ***Status Code = 201*** in case of ***success*** and ***200*** in case of ***failure.***

So by making use of the status code we can deserialize the response into appropriate POJO classes depending on success or failure. Below given code is an updated version of the above code and it takes care of success as well as failure response.

@Test

public void UserRegistrationSuccessful() {

RestAssured.baseURI ="https://demoqa.com";

RequestSpecification request = RestAssured.given();

JSONObject requestParams = new JSONObject();

requestParams.put("UserName", "test\_rest");

requestParams.put("Password", "rest@123");

request.body(requestParams.toJSONString());

Response response = request.post("/Account/v1/User");

ResponseBody body = response.getBody();

System.out.println(response.body().asString());

if(response.statusCode() == 200) {

// Deserialize the Response body into JSONFailureResponse

JSONFailureResponse responseBody = body.as(JSONFailureResponse.class);

// Use the JSONFailureResponse class instance to Assert the values of Response.

Assert.assertEquals("User already exists", responseBody.FaultId);

Assert.assertEquals("FAULT\_USER\_ALREADY\_EXISTS", responseBody.fault);

} else if (response.statusCode() == 201) {

// Deserialize the Response body into JSONSuccessResponse

JSONSuccessResponse responseBody = body.as(JSONSuccessResponse.class);

// Use the JSONSuccessResponse class instance to Assert the values of response.

Assert.assertEquals("OPERATION\_SUCCESS", responseBody.SuccessCode);

Assert.assertEquals("Operation completed successfully", responseBody.Message);

}

}

In the above code, we deserialize the response to JSONSuccessResponse or JSONFailureResponse class depending on whether the Status code is Success or Failure.

Note: Another way to deserialize multiple responses is by using the inheritance chain which readers can implement as an exercise.

### *****What is Authentication? and How does Authorization work in REST WebServices?*****

Authentication is a process to prove that you are the person you intend to be.

For e.g. while logging into your email account, you prove that ***you are you*** by providing a ***Username*** and a ***Password***. If you have the ***Username*** and the ***Password*** you are who you profess to be. This is what Authentication means.

@Test

public void AuthenticationBasics()

{

RestAssured.baseURI = "https://restapi.demoqa.com/authentication/CheckForAuthentication";

RequestSpecification request = RestAssured.given();

Response response = request.get();

System.out.println("Status code: " + response.getStatusCode());

System.out.println("Status message " + response.body().asString());

}

In the code above we are simply making an ***HTTP GET*** request to the endpoint. In this code, we have not added any ***Authorization*** header. So the expected behavior is that we will get Authorization error. If you run this test, you will get the following output.

Status code: 401

Status message:

{

"StatusID": "FAULT\_USER\_INVALID\_USER\_PASSWORD",

"Status": "Invalid or expired Authentication key provided"

}

The output clearly says that we have ***"Invalid or expired Authentication key provided"*** error. This means that either there was no Authentication information or the information supplied was invalid. Eventually, the server denies our request and returns an error response.

### *****What is Authorization? and How does Authorization work in REST WebServices?*****

Authorization is the process of giving access to someone. If you are Authorized then you have access to that resource. Now to Authorize you to need to present credentials and as we discussed earlier that process is called Authentication. Hence Authorization and Authentication are closely related terms and often used interchangeably.

## What is Basic Authentication (Auth)?

The implementation of basic authentication is to ensure that the APIs are secured and only the users who are authorized have the access to view them.

Before proceeding to understand the use of authentication in Rest Assured, let us execute our Rest Assured test without using any sort of authentication. Below is the code for your reference-

public class BasicAuth {

@Test

public void getData() {

RequestSpecification httpRequest = RestAssured.given();

Response res = httpRequest.get("https://postman-echo.com/basic-auth");

ResponseBody body = res.body();

//Converting the response body to string

String rbdy = body.asString();

System.out.println("Data from the GET API- "+rbdy);

}

}

Basic Authentication in Rest Assured

As discussed above, the basic authentication scheme uses the username and password in base64 encoded format. The request header needs to contain the credentials of the user for access to the resource. It is very easy to send the credentials using the basic auth and you may use the below syntax-

given().auth().basic("your username", "your password").get("your end point URL");

public class BasicAuth {

@Test

public void getData() {

RequestSpecification httpRequest = RestAssured.given().auth().basic("postman", "password");

Response res = httpRequest.get("https://postman-echo.com/basic-auth");

ResponseBody body = res.body();

//Converting the response body to string

String rbdy = body.asString();

System.out.println("Data from the GET API- "+rbdy);

}

}

#### **Preemptive Authentication**

By default, Rest Assured uses the [***challenge-response mechanism.***](https://datatracker.ietf.org/doc/html/rfc2617#section-1.2) This means that it waits for the server to challenge rather than send the credentials directly.

given().auth().preemptive().basic("your username", "your password").get("your end point URL");

public class BasicAuth {

@Test

public void getUserData() {

//Using the preemptive directive of basic auth to send credentials to the server

RequestSpecification httpRequest = RestAssured.given().auth().preemptive().basic("postman", "password");

Response res = httpRequest.get("https://postman-echo.com/basic-auth");

ResponseBody body = res.body();

//Converting the response body to string

String rbdy = body.asString();

System.out.println("Data from the GET API- "+rbdy);

}

}

RequestSpecification httpRequest = RestAssured.given().auth().preemptive().basic("postman", "password");

Response res = httpRequest.get("https://postman-echo.com/basic-auth");

ResponseBody body = res.body();

String rbdy = body.asString();

System.out.println("Data from the GET API- "+rbdy);

### *****Digest Authentication*****

given().auth().digest("your username", "your password").get("your end point URL")

### *****Form Authentication*****

given() .auth().form("your username", "your password").post("your end point URL")

given().auth().form("your username", "your password", new FormAuthConfig("/perform\_signIn","user","password"));

### *****OAuth Authentication*****

#### **OAuth 1.0**

given().auth().oauth(consumerKey, consumerSecret, accessToken, tokenSecret).get("your end point URL")

#### **OAuth 2.0**

given().auth().oauth2("Access token").get("your end point URL")

## What is an HTTP PUT Request Method?

Put request updates a resource at a specified URI. It is also used to create a new resource at the given URI or replace the entire product entity.

The methods POST and PUT use the following status codes:

**POST request**

* 201 with a location header pointing to the new resource.
* 400 if the new item is not created.

**PUT request**

* 204 for OK/SUCCESS (but no content).
* 200 for OK with Content Body (Updated response).
* 400 if the data sent was invalid.

**public** **class** PUTMethod {

@Test

**public** **void** putMethod() {

String userId = "toolsqa\_test";

String baseUrl = "https://demoqa.com";

String token = "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyTmFtZSI6InRlc3RpbmcxMjMiLCJwYXNzd29yZCI6IlBhc3N3b3JkQDEiLCJpYXQiOjE2Mjg1NjQyMjF9.lW8JJvJF7jKebbqPiHOBGtCAus8D9Nv1BK6IoIIMJQ4";

String isbn = "9781449325865";

RequestSpecification httpRequest = RestAssured.given().header("Authorization", "Bearer " + token).header("Content-Type", "application/json");

}

### ****Send JSON content in the body of the Request****

The generated request is sent to the server.

java Response res = httpRequest.body("{ "isbn": "" + isbn + "", "userId": "" + userId + ""}")

.put("/BookStore/v1/Book/9781449325862");

So in the above code, we have created a request body as a JSON string and then we call the "put()" method with this request by sending the ISBN as an argument. This ensures the record with the given ISBN is updated.

### ****Validate the Response****

**public** **class** PUTMethod {

String userId= "toolsqa\_test";

String baseUrl="https://demoqa.com";

String token = "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyTmFtZSI6InRlc3RpbmcxMjMiLCJwYXNzd29yZCI6IlBhc3N3b3JkQDEiLCJpYXQiOjE2Mjg1NjQyMjF9.lW8JJvJF7jKebbqPiHOBGtCAus8D9Nv1BK6IoIIMJQ4";

String isbn ="9781449325865";

@Test

**public** **void** updateBook() {

RestAssured.baseURI = baseUrl;

RequestSpecification httpRequest = RestAssured.given()

.header("Authorization", "Bearer " + token)

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

//Calling the Delete API with request body

Response res = httpRequest.body("{ \"isbn\": \"" + isbn + "\", \"userId\": \"" + userId + "\"}").put("/BookStore/v1/Book/9781449325862");

//Fetching the response code from the request and validating the same

System.out.println("The response code - " +res.getStatusCode());

Assert.assertEquals(res.getStatusCode(),200);

}

## What is a delete request method?

HTTP delete method (written as delete) which deletes a resource from the server identified by the URI we are sending through it.

## What are the different response codes for  Delete Request?

* ***202***(Accepted): The server accepts the request but does not enact.
* ***204***(No Content)- A status code of 204 on the HTTP delete request method denotes successful enactment of the delete request without any content in the response.
* ***200***(OK)- The action was successful and the response message includes representation with the status.
* ***404***(Not Found) - When the server can't find the resource. The reason could either does not exist or previously deleted.

public class DeleteBook {

String userId= "de5d75d1-59b4-487e-b632-f18bc0665c0d";

String baseUrl="https://demoqa.com/swagger/";

String token = "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyTmFtZSI6InRlc3RpbmcxMjMiLCJwYXNzd29yZCI6IlBhc3N3b3JkQDEiLCJpYXQiOjE2Mjg1NjQyMjF9.lW8JJvJF7jKebPiHOBGtCAu";

String isbn ="9781449337711";

@BeforeTest

@AfterTest

public void getUserData() {

RestAssured.baseURI = baseUrl;

RequestSpecification httpRequest =

RestAssured.given().header("Authorization", "Bearer " + token)

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

Response res = httpRequest.get("/Account/v1/User/"+userId);

ResponseBody body = res.body();

//Converting the response body to string

String rbdy = body.asString();

System.out.println("Data from the GET API- "+rbdy);

}

@Test

public void deleteBook() {

RestAssured.baseURI = baseUrl;

RequestSpecification httpRequest = RestAssured.given().header("Authorization", "Bearer " + token)

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

//Calling the Delete API with request body

Response res = httpRequest.body("{ \"isbn\": \"" + isbn + "\", \"userId\": \"" + userId + "\"}").delete("/BookStore/v1/Book");

//Fetching the response code from the request and validating

the same

System.out.println("The response code is - " +res.getStatusCode());

Assert.assertEquals(res.getStatusCode(),204);

}

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* REST API Http Response Codes \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

200: OK. Everything worked as expected.

201: A resource was successfully created in response to a POST request. The Location header contains the URL pointing to the newly created resource.

204: The request was handled successfully and the response contains no body content (like a DELETE request).

304: The resource was not modified. You can use the cached version.

400: Bad request. This could be caused by various actions by the user, such as providing invalid JSON data in the request body etc.

401: Authentication failed.

403: The authenticated user is not allowed to access the specified API endpoint.

404: The requested resource does not exist.

405: Method not allowed. Please check the Allow header for the allowed HTTP methods.

415: Unsupported media type. The requested content type or version number is invalid.

422: Data validation failed (in response to a POST request, for example). Please check the response body for detailed error messages.

429: Too many requests. The request was rejected due to rate limiting.

500: Internal server error. This could be caused by internal program errors.