**REST architecture**

* REST stands for Representational State Transfer.
* REST is a software architecture style that relies on a stateless communications protocol, most commonly, HTTP.
* REST structures message data in XML, YAML or any other format that is machine-readable. JSON is now the most popular format being used in REST Services.
* In REST architecture, a REST Server simply provides access to resources and the REST client accesses and presents the resources. Here each resource is identified by URIs/ Global IDs.

**URI – Uniform Resource Identifier**

* A Uniform Resource Identifier (URI) is a string of characters used to identify a resource. Such identification enables interaction with representations of the resource over a network.
* Each resource is identified by one or more Uniform Resource Identifiers (URIs). To access the resource, an application calls an HTTP operation (Method) on one of the resource's URIs.

**RESTful services**

* Web services based on REST Architecture are known as RESTful Web Services.
* These web services use HTTP methods to implement the concept of REST architecture. A RESTful web service usually defines a URI (Uniform Resource Identifier), which is a service that provides resource representation in JSON format using HTTP Methods.

# REST Assured

* APIs are playing a major role in recent software developments and hence proper automated testing of these APIs is becoming essential. There are many different tools that will help you in writing automated tests at the API level.
* Rest Assured is a Java library that offers a domain-specific language (DSL) to create and maintain tests for RESTful APIs. Integration of Rest Assured with TestNG and JUnit can also be done.
* To simplify the testing and validation of REST APIs, Rest Assured was developed. It is influenced by testing techniques that are used in other languages like Ruby and Groovy.
* One of the most significant features of Rest Assured is that to check the contents present in the response, we can make use of the XML path and JSON path. We can parse the response data and test specific elements of their properties using XML and JSON path.
* REST Assured can also be used to validate and verify the responses of all the requests that are supported by REST Assured,viz., GET, PUT, DELETE, PATCH, and HEAD requests. By starting with the verbs and standard HTTP operations, Rest Assured provides solid support for HTTP.
* The methods/skill required for validation of responses received from the server is given by Rest-Assured Library. 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.

# GET

* The GET method is used for getting data from the server. The data may be anything, e.g. an HTML document, an image, or an XML file.
* For any given HTTP GET API, if the resource is found on the server then it must return HTTP response code 200 (OK) – along with the response body which is usually either XML or JSON content.
* In case the resource is NOT found on the server then it must return HTTP response code 404 (NOT FOUND). Similarly, if it is determined that the GET request itself is not correctly formed then the server will return HTTP response code 400 (BAD REQUEST).

# POST

* The POST method is used to create a new resource or update existing resources in the collection of resources.
* If a resource has been created on the server, the response SHOULD be HTTP response code 201 (Created) and contain an entity which describes the status of the request and refers to the new resource.
* POST is not idempotent and invoking two identical POST requests will result in two different resources containing the same information except resource ids.

# PUT

* PUT methods are primarily used to update an existing resource. If the resource does not exist then PUT will create a new resource.
* Unlike post, if duplicate PUT methods are invoked, PUT replaces the old resource and creates a new one instead of two identical resources.
* If a resource has been created on the server, the response should be HTTP response code 201(Created) and if an existing resource is modified, either the 200(OK) or 204(No Content) response codes SHOULD be sent to indicate successful completion of the request.

# DELETE

* DELETE APIs are used to delete resources identified by the Request- URI.
* The successful response from DELETE requests IS HTTP response code 200 (OK)

# *****Demo 1: Creation of TestNG class and Integration with Rest Assured*****

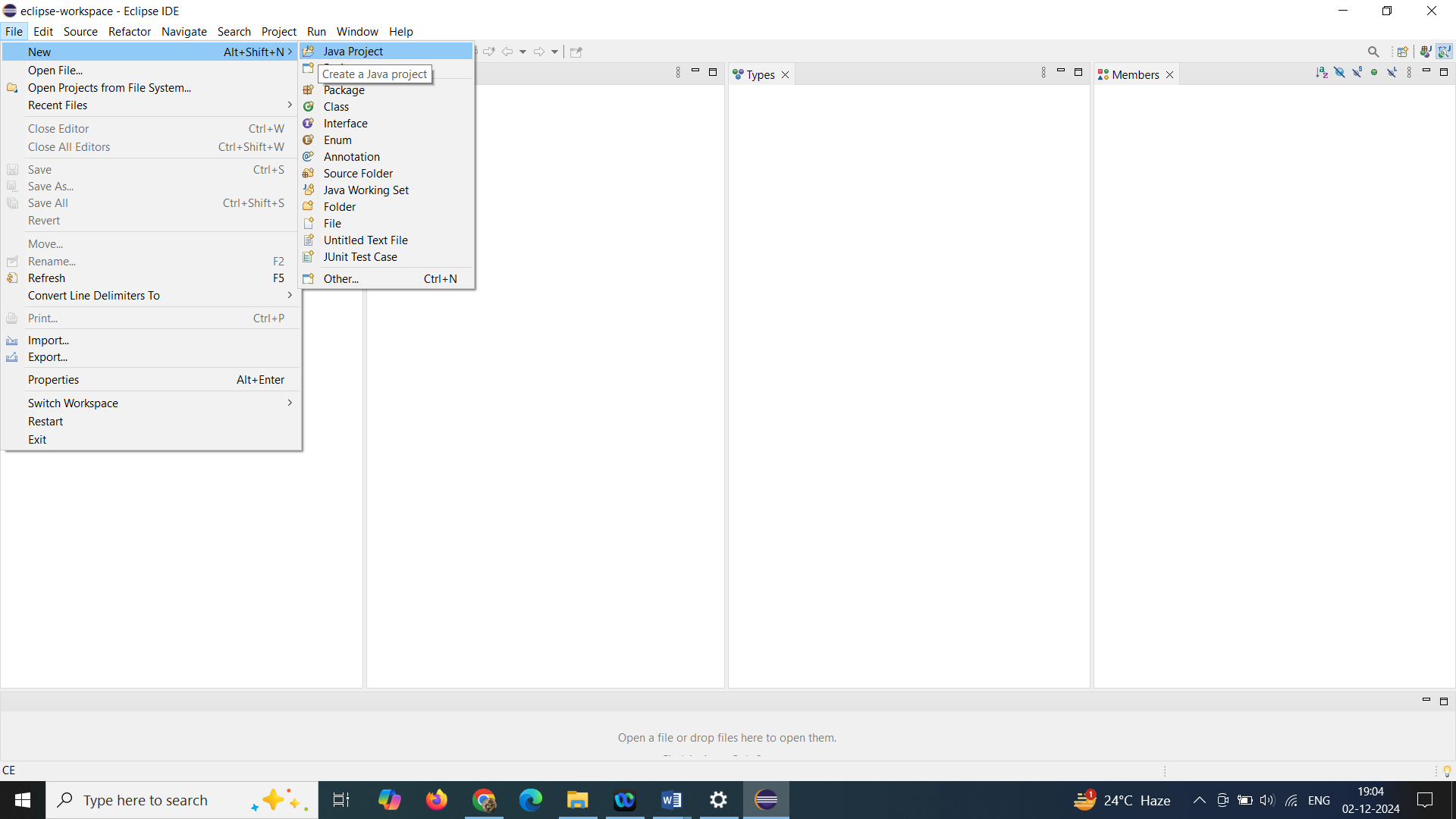
**Highlights:**

* You will learn how to create a new Java Project, package and new TestNG test class file.
* You will learn how to add the TestNG library file and integrate the Java project with Rest Assured.

**Demo Steps:**

**Step 1:** Set the perspective to Java by selecting **Window -> Perspective-> Open Perspective -> Java.**

**Step 2:** Create a new Java Project.**File -> New -> Other-> Java Project.**



***Step 3:*** Name the Project as **RestAssuredLearning** and click on Finish.

**Step 4:**Now the project with name RestAssuredLearning should appear in the package explorer.

**Step 5:**Right Click on the src folder and create a package. **New-> Package**

**Step 6:** Input the name for the package as **Demo**and click on the Finish button.

**Step 7:**Right Click on Demo package, **New -> Other -> TestNG class**. Click on Next.

**Step 8:** Input the name for the TestNG class and click on the Finish button.

***Step 9:*** It will create a TestNG class with a method annotated with @Test annotation. Some error messages will be displayed as the TestNG library file is not added.

**Step 10:** To add the TestNG library file, right click on the project**-> Build Path -> Configure Build Path**.

***Step 11:*** From the newly opened window, click **Libraries -> Add Library** and select the TestNG option listed in Add Library dialog box.

**Step 12:**Click on the Finish button and Click on Apply and Close.

**Step 13:**To integrate the java project with Rest Assured, right click on the project**-> Build Path -> Configure Build Path**.

**Step 14:** From the newly opened window, click **Libraries -> Add External Jars.**

**Step 15:** Navigate to the folder having the required jars. Make sure to include the following jars.

* rest-assured-3.1.1.jar
* All the jars in the folder docs and rest-assured-3.1.1-deps.

**Step 16:** Once all the jars are added, click on Apply and Close in Java Build Path window.

.

# HTTP GET for REST

* As discussed in "Methods in RESTful service" resource, HTTP GET is used to retrieve a resource which may be HTML document, Image, XML file, etc.
* If the resource is found, it returns HTTP response code 200 along with the XML or JSON content. If not found, HTTP response code 404 is returned.
* The response object will contain all the data sent by the server. Different methods can be called on the response object to get different parts of the response.

## getBody() method

* The getBody() method returns an InputStream from which the response can be accessed. The HTTP response which we fetch during the "GET" operation will be in XML or JSON format. To convert it into an understandable format, we use the getBody() method.
* It returns an object of type io.restassured.response.ResponseBody. Using this class, you can validate the complete or parts of the Response Body.

### ResponseBody body = response.getBody ();

## asString () method

ResponseBody interface also has a method called asString() which converts a ResponseBody into its String representation. In short, it returns the response body in a String format.

### String value = body.asString();

## toString () method

toString () method does exactly the same thing that asString() method does. So you can even use toString() method to convert the Response Body into its String format.

### String value = body.toString ();

***Demo 2.1: REST API Test using Rest Assured***

**Highlights:**

* How to communicate with rest services
* Use of getBody() and asString() methods

**Demo Steps:**

***Step 1:***In the method annotated with @Test annotation, create a reference for Response interface.

1. *//Create a reference for Response interface*
2. Response response;

***Step 2:*** Use RestAssured class to set up a request with the specified base URI with HTTP Method Type “get()”.

1. *// Make a request to the server by specifying the method Type and the method URL.*
2. *// This will return the Response from the server. Store the response in a reference variable created above.*
3. response = RestAssured.get("http://localhost:8080/InfyClinicV2/APP\_V2/getDoctor/byId?doctorId=1001");

Here, the base URI is “http://localhost:8080/InfyClinicV2/APP\_V2/getDoctor”. This is called the base URI because this is the root address of the resource. Adding “/byId?doctorId=1001” at the end appends the exact resource name in the URI that we are trying to access.

***Step 3:***Return the body of the received response using getBody() method of Response Interface. Response Body is converted into a string value using asString() method and printed on the console using the System.out.println statement.

1. *//print the message body of the response received from the server*
2. String responseBody = response.getBody().asString();
3. System.out.println("Response Body is => " + responseBody);

***Step 4:***The final script looks like the one given below:

1. ​package Demo;
2. import org.testng.annotations.Test;
3. import io.restassured.RestAssured;
4. import io.restassured.response.Response;
5. public class Demo2\_1 {
7. @Test
8. public void f() {
10. *//Create a reference for Response interface*
11. Response response;
12. *// Make a request to the server by specifying the method Type and the method URL.*
13. *// This will return the Response from the server. Store the response in a reference variable created above.*
14. response = RestAssured.get("http://localhost:8080/InfyClinicV2/APP\_V2/getDoctor/byId?doctorId=1001");
15. *//print the message body of the response received from the server*
16. String responseBody = response.getBody().asString();
17. System.out.println("Response Body is => " + responseBody);
19. }
20. }

***Step 5:*** Run the TestNG test.

**Output:**

The details of the account holder with account number given in the get request are displayed.

## getContentType () method

* Content-Type is a two-part identifier for file-formats. The HTTP header Content-Type informs the HTTP client or server about the type of data being sent.
* Content-Type is given as a part of the header name in the request body. In response, a Content-type header tells the client what the content type of the returned content actually is. It indicates the format of the request body.
* If the body contains data in the form of JSON, then the value of the Content-Type header will be application/JSON. Similarly, if the data in the body is XML the Content-Type header will be application/XML.
* getContentType () method returns content type of the response if found, else it returns null.

## getSessionID () method

* Generally, web servers use some algorithms to generate a session ID. It is a unique number that is assigned to the user by the website. It can be stored as a cookie, form field or URL.
* getSessionID () method returns the session id from the response. Session Config class configures the session ID. If the session ID is not defined, then getSessionID () returns null.

# ****Demo 3: Demo on working with getContenttype, getSessionID methods****

**Highlights:**

* Use of getContentType and getSessionID methods

**Demo Steps:**

**Step 1:** In the method annotated with @Test annotation, create a reference for Response interface.

1. *//Create a reference for Response interface*
2. Response response;

**Step 2:** Use RestAssured class to set up a request with the specified base URI using HTTP Method Type “get()”.

1. *// Make a request to the server by specifying the method Type and the method URL.*
2. *// This will return the Response from the server. Store the response in a reference variable created above.*
3. response = RestAssured.get("http://localhost:8080/InfyClinicV2/APP\_V2/getDoctor/byId?doctorId=1001");

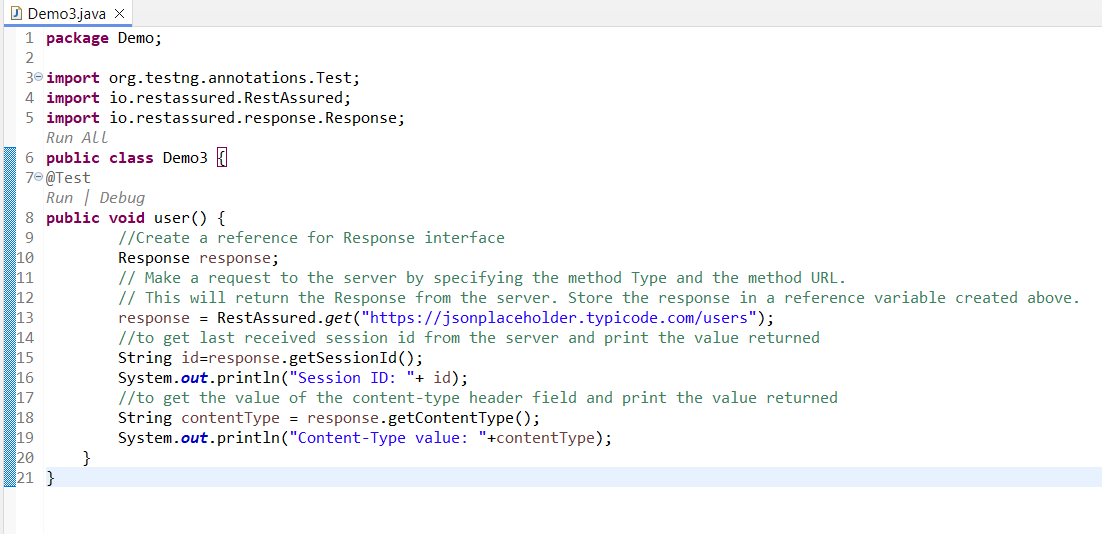
**Step 3:** To return the last received session id from the server, use getSessionId() method of Response Interface.

1. *//to get last received session id from the server and print the value returned*
2. String id=response.getSessionId();
3. System.out.println("Session ID: "+ id);

**Step 4:**  To return the value of the content-type header field, use getContentType() method of Response Interface.

1. *//to get the value of the content-type header field and print the value returned*
2. String contentType = response.getContentType();
3. System.out.println("Content-Type value: "+contentType);

**Step 5:** The final script looks like the one given below:



**Step 6:** Run the TestNG class.

**Output:**

Prints the last Session Id from server and the content type of the resource that the URI references, in console window.

# getStatusCode() method:

* Every response will have a status Code. For GET operations, the status code will be 200 and 404 if the value is not found.
* getStatusCode () method returns the status code from the response. This method returns an integer and an assertion can be used to check the value of the status Code.

 int statuscode = response.getStatusCode();

# getStatusLine() method:

* Status line from the response consists of three sections:

1. HTTP protocol version
2. Status Code
3. Status Code’s string value

* getStatusLine() method returns the status Line of the response in a String format.
* During a success scenario, status line will be “HTTP/1.1 200 OK”. First part is Http protocol (HTTP/1.1). Second is Status Code (200). Third is the Status message (OK).

  String statusLine = response.getStatusLine();

# ****Demo 4: Demo on working with statuscode and statusline methods in rest services****

**Highlights:**

* Use of statuscode and statusline methods

**Demo Steps:**

**Step 1:**In the method annotated with @Test annotation, create a reference for Response interface.

1. *//Create a reference for Response interface*
2. Response response;

**Step 2:** Use RestAssured class to set up a request with the specified base URI with HTTP Method Type “get()”.

1. *// Make a request to the server by specifying the method Type and the method URL.*
2. *// This will return the Response from the server. Store the response in a reference variable created above.*
3. response = RestAssured.get("http://localhost:8080/InfyClinicV2/APP\_V2/getDoctor/byId?doctorId=1001");

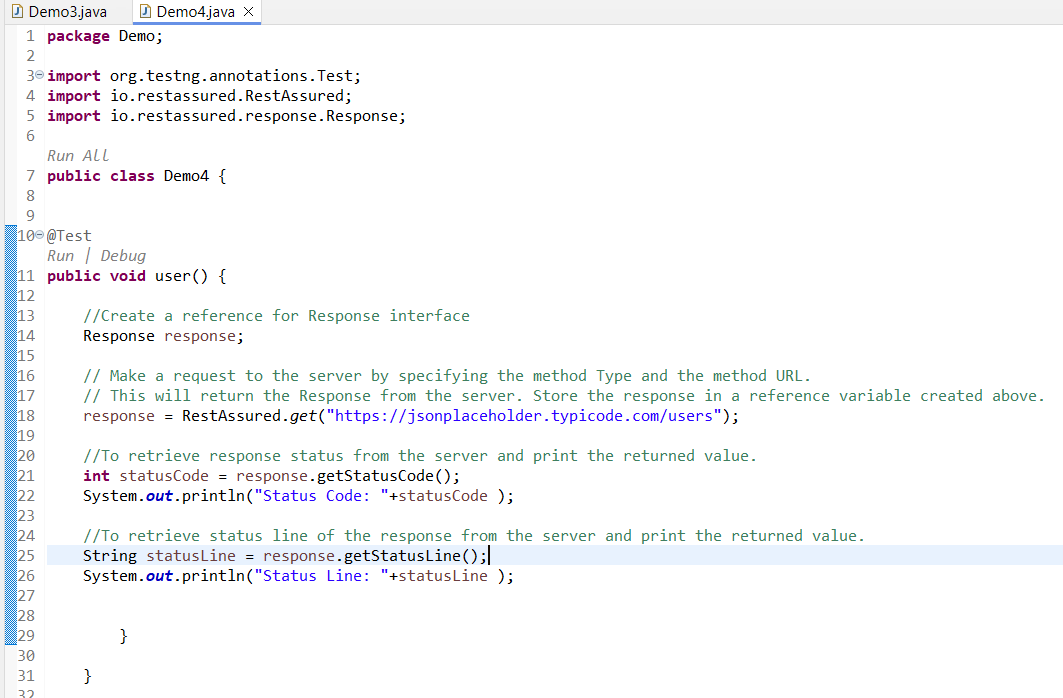
**Step 3:**To return the status of the response from the server for the request from client, use getStatusCode() method of Response interface.

1. *//To retrieve response status from the server and print the returned value.*
2. int statusCode = response.getStatusCode();
3. System.out.println("Status Code: "+statusCode );

**Step 4:** To return the protocol version of response, use getStatusLine() method of Response interface.

1. *//To retrieve status line of the response from the server and print the returned value.*
2. String statusLine = response.getStatusLine();
3. System.out.println("Status Line: "+statusLine );

**Step 5:**The final script looks like the one given below:



**Step 6:** Run the TestNG class.

**Output:**

Prints status and protocol version of response in the console window.

# Headers

* Headers are the part of the response that is sent by the server. Each header will be in the form of a key-value pair. It is used to send the extra information by the server.
* To access individual headers or all the headers, direct methods are provided by the Response interface. Every response received from a server contains zero or more headers.

# getHeaders

**Headers getHeaders():**  
getHeaders() returns Headers from the response. If the response contains multiple headers with the same name, a list of the response header values is returned.

# ****Demo 5:  Demo to print all the Headers from HTTP Response****

**Highlights:**

* Use of getHeaders() method to print all the Headers from HTTP Response

**Demo Steps:**

**Step 1:** In the method annotated with @Test annotation, create a reference for the Response interface.

1. *//Create a reference for Response interface*
2. Response response;

**Step 2:** Use RestAssured class to set up a request with the specified base URI using HTTP Method Type “get()”.

1. *// Make a request to the server by specifying the method Type and the method URL.*
2. *// This will return the Response from the server. Store the response in a reference variable created above.*
3. response = RestAssured.get("http://localhost:8080/InfyClinicV2/APP\_V2/getDoctor/byId?doctorId=1001");

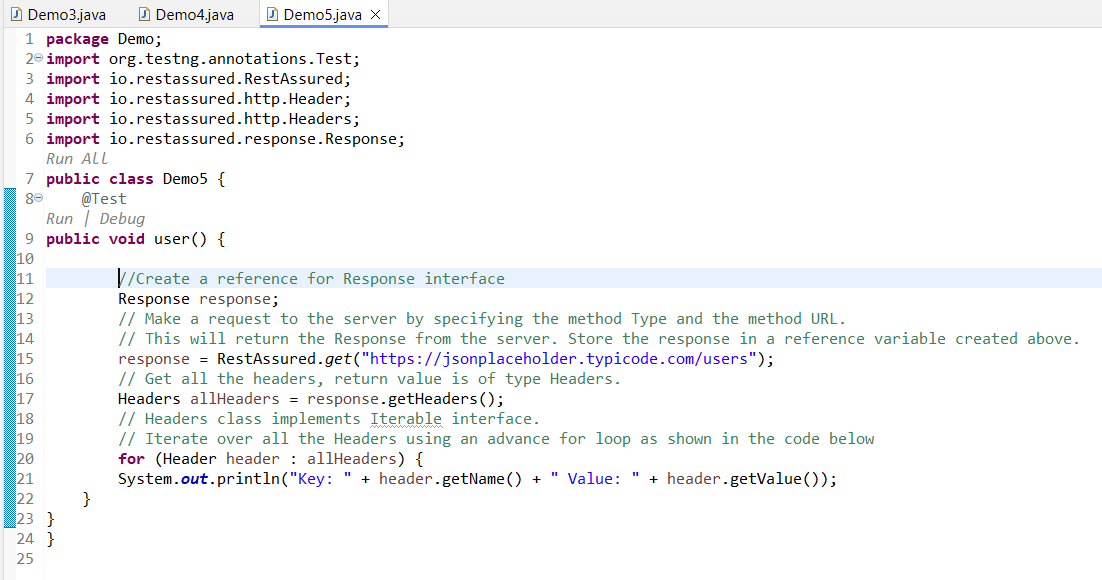
**Step 3:** Retrieve all the headers from the response using getHeaders() method of Response interface.

1. *// Get all the headers, return value is of type Headers.*
2. Headers allHeaders = response.getHeaders();

**Step 4:** Iterate over all the headers retrieved print the header name and their respective values.

1. *// Headers class implements Iterable interface.*
2. *// Iterate over all the Headers using an advance for loop as shown in the code below*
3. for(Header header : allHeaders) {
4. System.out.println("Key: " + header.getName() + " Value: " + header.getValue());
5. }

**Step 5:** The final script looks like the one given below:



**Step 6:** Run the TestNG class.

**Output:**

Prints list of all header names present in the response with their respective values.

As we have seen in getHeaders() method, Each header entry is present in a key-value pair. The Header carries extra information sent by the server. This extra information is considered as Metadata of the Response.

One of the headers called **Content-Type** which tells how to interpret the data present in the body of the response. The Content-Type header indicates the media type of the resource.

Example: If the body contains data in the form of JSON, then the value of the Content-Type header will be application/JSON. Similarly if the data in the body is XML the Content-Type header will be application/xml.

# getHeader(String name):

* getHeader() method returns a String containing the value of the requested header, or null if the request does not have a header of that name.
* If multiple headers of the same name are present in the response, getHeader() method returns the first header value in the request.

String headername = response.getHeader(String name)

name – a string specifying the header name

* If you want to fetch multiple header values for the same name, you have to use Headers.getList(String) to get all the values.The header name is case insensitive.

# Demo 6: Demo to read different header types from HTTP Response

**Highlights:**

To read different header types from HTTP Response.

**Demo Steps:**

**Step 1:** In the method annotated with @Test annotation, create a reference for Response interface.

1. *//Create a reference for Response interface*
2. Response response;

**Step 2:** Use RestAssured class to set up a request with the specified base URI with HTTP Method Type “get()”.

1. *// Make a request to the server by specifying the method Type and the method URL.*
2. *// This will return the Response from the server. Store the response in a reference variable created above.*
3. response = RestAssured.get("http://localhost:8080/InfyClinicV2/APP\_V2/getDoctor/byId?doctorId=1001");

**Step 3:** In the below code, .header(String arg0) method is used to get a particular header. In the argument of this method, the exact header name should be passed.

Here, Content-Type, Server and Date headers are read.

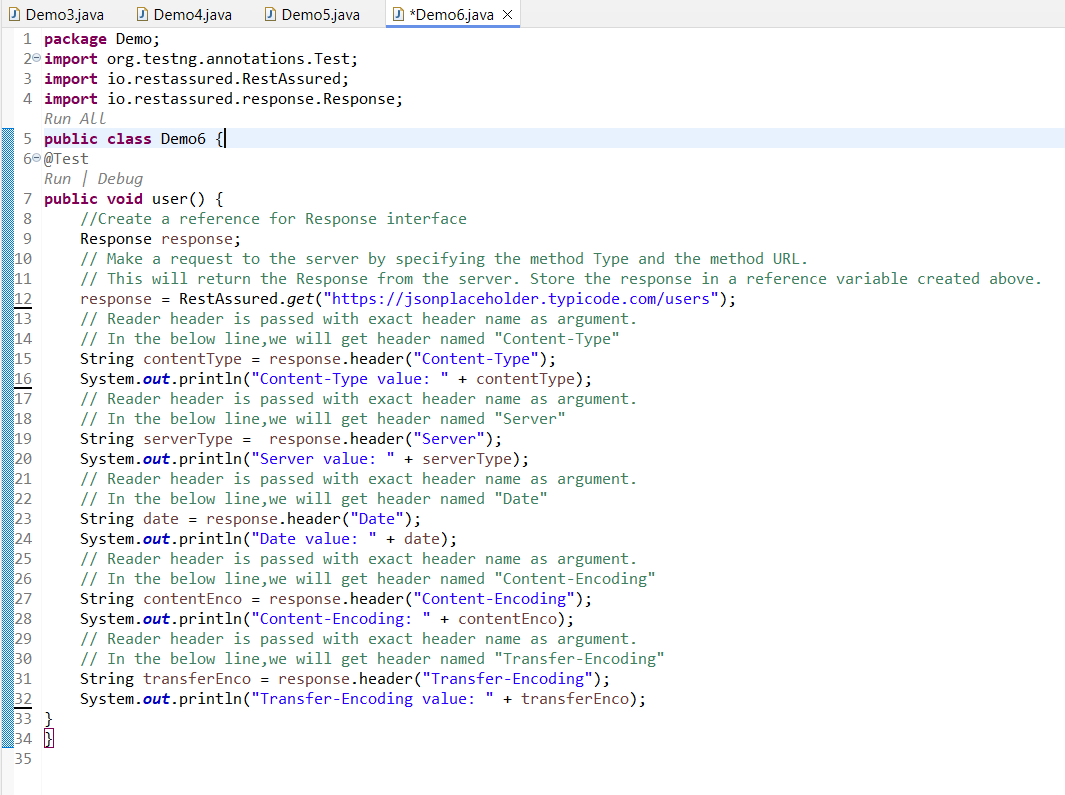
1. *// Reader header is passed with exact header name as argument.*
2. *// In the below line,we will get header named "Content-Type"*
3. String contentType = response.header("Content-Type");
4. System.out.println("Content-Type value: " + contentType);
5. *// Reader header is passed with exact header name as argument.*
6. *// In the below line,we will get header named "Server"*
7. String serverType = response.header("Server");
8. System.out.println("Server value: " + serverType);
9. *// Reader header is passed with exact header name as argument.*
10. *// In the below line,we will get header named "Date"*
11. String date = response.header("Date");
12. System.out.println("Date value: " + date);

**Step 4:** In the below code, Content-Encoding and Transfer-Encoding headers are read.

1. *// Reader header is passed with exact header name as argument.*
2. *// In the below line,we will get header named "Content-Encoding"*
3. String contentEnco = response.header("Content-Encoding");
4. System.out.println("Content-Encoding: " + contentEnco);
6. *// Reader header is passed with exact header name as argument.*
7. *// In the below line,we will get header named "Transfer-Encoding"*
8. String transferEnco = response.header("Transfer-Encoding");
9. System.out.println("Transfer-Encoding value: " + transferEnco);

**Step 5:**In the same way, the headers X-AspNet-Version, X-Powered-By, Content-Length can be read.

**Step 6:**The final script looks like the one given below:



# JSON object:

In Restful services, different types of responses like XML, JSON, and HTML can be achieved by making configuration to the service. If the response returned by the service is of the JSON object, we need a way to traverse through parts of JSON. Every JSON object has a tree of nodes, where each node represents a JSON Element. In an XML document, XPath lets us extract the data we need. For JSON, a query language with features similar to that of XPath is emerged - **JSONPath**.

# JSONPath:

It creates a syntax to access various parts of a JSON document and it is uniform. Using the expressions created, we can reach the subset of JSON by traversing through the document. JsonPath can be used as an alternative to get the values from an Object document. It follows the Groovy dot notation syntax while fetching an object from the document.

# ****Demo 7: Restful services with JSON****

**Highlights:**

* To extract a node text from Response using JsonPath.
* Validating the extracted text from node using assertions

**Demo Steps:**

**Step 1:**In the method annotated with @Test annotation, create a reference for Response interface.

1. *//Create a reference for Response interface*
2. Response response;

**Step 2:** Use RestAssured class to set up a request with the specified base URI using HTTP Method Type “get()”.

1. *// Make a request to the server by specifying the method Type and the method URL.*
2. *// This will return the Response from the server. Store the response in a reference variable created above.*
3. response = RestAssured.get("http://localhost:8080/InfyClinicV2/APP\_V2/getDoctor/byId?doctorId=1001");

**Step 3:** Retrieve the JsonPath object from the response.

1. *// First get the JsonPath object instance from the Response interface*
2. JsonPath jsonPathEvaluator = response.jsonPath();

**Step 4:**To get the String value of a particular node of the Json response, simply perform get operation on the required node using JsonPath object.

1. *// Then simply query the JsonPath object to get a String value of the node*
2. *// specified by JsonPath: doctorName(Note: You should not put $. in the Java code)*
3. String name = jsonPathEvaluator.get("doctorName");

**Step 5:** Print the obtained result in the console window.

1. *// Print the name variable to see what we got*
2. System.out.println("Name of the doctor is: " + name);
3. *//directly fetch the value as string by using getString() & print it in the console*
4. System.out.println("Name of the doctor is: " + jsonPathEvaluator.getString("doctorName"));

**Step 6:** Validating the obtained account holder name with appropriate error message for the failed assertion.

1. *//Validating the account holder name*
2. Assert.assertEquals(name */\*actual value\*/*, "John" */\*Expected Value\*/*, "Incorrect account holder name");

**Step 7:** The final script looks like the one given below:

1. package Demo;
2. import org.testng.Assert;
3. import org.testng.annotations.Test;
4. import io.restassured.RestAssured;
5. import io.restassured.path.json.JsonPath;
6. import io.restassured.response.Response;
7. public class Demo9 {
8. @Test
9. public void user() {
10. *// Create a reference for Response interface*
11. Response response;
12. *// Make a request to the server by specifying the method Type and the method URL.*
13. *// This will return the Response from the server. Store the response in a*
14. *// reference variable created above.*
15. response = RestAssured.get("http://localhost:8080/InfyClinicV2/APP\_V2/getDoctor/byId?doctorId=1001");
16. *// First get the JsonPath object instance from the Response interface*
17. JsonPath jsonPathEvaluator = response.jsonPath();
18. *// Then simply query the JsonPath object to get a String value of the node*
19. *// specified by JsonPath: doctorName (Note: You should not put $. in the Java code)*
20. String name = jsonPathEvaluator.get("doctorName");
21. *// Print the name variable to see what we got*
22. System.out.println("Name of the doctor is: " + name);
24. *//directly fetch the value as string by using getString() & print it in the console*
25. System.out.println("Name of the doctor is: " + jsonPathEvaluator.getString("doctorName"));
26. *// Validating the account holder name*
27. Assert.assertEquals(name */\* actual value \*/*, "John"*/\* Expected Value \*/*,"Incorrect account holder name");
28. }
29. }

**Step 8:** Run the TestNG Class.

**Output:**

Prints the doctor name retrieved from the response and validates it.

If response is a JSON Array, the getList() method will help to fetch the response in the form of List.

Size of the list will help to know the number of JSON Object the response contains.

Perform GET Operation on the URI:

1. http:*//10.82.180.36:8080/rest-session-demo/api/students/sorted*

Code:

1. import java.util.List;
2. import org.testng.annotations.Test;
3. import io.restassured.RestAssured;
4. import io.restassured.path.json.JsonPath;
5. import io.restassured.response.Response;
6. public class NewTest1 {
7. @Test
8. public void f() {
9. Response response = RestAssured.get("http://10.82.180.36:8080/rest-session-demo/api/students/sorted");
10. JsonPath path = response.jsonPath();
12. *//use getList() to fetch the response in the form of List*
13. List<Object> names = path.getList("name");
15. *//or alternate way is to specify the datatype in List<>*
16. *//List<String> names = path.getList("name");*
18. *//get the size of the list to know the number of JSON Object the response contains*
19. System.out.println("Total number of students: "+names.size());
21. *//print the obtained list in the console*
22. System.out.println("Names of the students are: "+names);
24. *//use for each loop if required to traverse through each element in the list*
25. for(Object name:names) {
26. System.out.println(name);
27. }
28. }
29. }

Run the above code as TestNG to get the desired output which is as shown below.

***Demo 10: RESTful services with JSON request and XML response***

**Highlights:**

* To communicate with RESTful services using JSON request and XML response

**Demo Steps:**

**Step 1:** In the method annotated with @Test annotation, create a reference for Response interface.

1. *//Create a reference for Response interface*
2. Response response;

**Step 2:** Use RestAssured class to set up a request with the specified base URI and HTTP Method Type "get()".

1. *// Make a request to the server by specifying the method Type and the method URL.*
2. *// This will return the Response from the server. Store the response in a reference variable created above.*
3. response = RestAssured.get("http://10.82.181.42/WebService/HMSWebServices/BloodBankManagement.asmx/FetchBloodBankDetailsByName?bloodBankName=Ayush");

**Step 3:** Return the body of the received response using getBody() method of Response interface. Response body is converted into a string value using asString() method and printed on the console using the System.out.println statement.

1. String responseBody = response.getBody().asString();
2. System.out.println("Response Body is => " + responseBody);

**Step 4:** Final script looks like the one given below:

1. package Demo;
2. import org.testng.annotations.Test;
3. import io.restassured.RestAssured;
4. import io.restassured.response.Response;
5. public class Demo10 {
6. @Test
7. public void f() {
9. *//Create a reference for Response interface*
10. Response response;
12. *// Make a request to the server by specifying the method type and the method URL.*
13. *// This will return the response from the server. Store the response in the reference variable created above.*
14. response = RestAssured.get("http://10.82.181.42/WebService/HMSWebServices/BloodBankManagement.asmx/FetchBloodBankDetailsByName?bloodBankName=Ayush");
16. String responseBody = response.getBody().asString();
17. System.out.println("Response Body is => " + responseBody);
18. }
19. }

**Step 5:** Run the TestNG test.

**Output:**

The details of the blood bank with blood bank name given in GET request are displayed.

The details of the blood bank with blood bank name given in GET request are displayed.

# Assertions:

* All the data which we are fetching from the response needs to be validated to make sure whether they are the same as that of the expected one.
* To validate these data, we can make use of assertions.
* In assertions, they are many methods to validate a response but to check whether the fetched data is the same as expected data, we will use assertEquals.
* Assert.assertEquals() method which is imported from the package org.testng checks if the two objects are equal. If they are not equal, an AssertionError without a message is thrown.

        Assert.assertEquals(String expected,String actual)

# Demo 8: Assertions

**Highlights:**

* Perform validations on various headers from the response using assertions

**Demo Steps:**

**Step 1:**In the method annotated with @Test annotation, create a reference for Response interface.

1. *//Create a reference for Response interface*
2. Response response;

**Step 2:**Use RestAssured class to set up a request with the specified base URI using HTTP Method Type “get()”.

1. *// Make a request to the server by specifying the method Type and the method URL.*
2. *// This will return the Response from the server. Store the response in a reference variable created above.*
3. response = RestAssured.get("http://localhost:8080/InfyClinicV2/APP\_V2/getDoctor/byId?doctorId=1001");

**Step 3:**Return the status of the response from the server for the request from client, using getStatusCode() method of Response interface and validate it.

1. *//Validating STATUS CODE from response*
2. int statusCode = response.getStatusCode();
3. Assert.assertEquals(statusCode */\*actual value\*/*, 200 */\*expected value\*/*, "INCorrect status code returned");

**Step 4:**Return the protocol version of the response using getStatusLine () method of Response interface and validate it.

1. *//Validating STATUS LINE from response*
2. String statusLine = response.getStatusLine();
3. Assert.assertEquals(statusLine */\*actual value\*/*, "HTTP/1.1 200 " */\*expected value\*/*, "INCorrect status Line returned");

**Step 5:** Return the value of the content-type header field using getContentType() method of Response Interface and validate it.

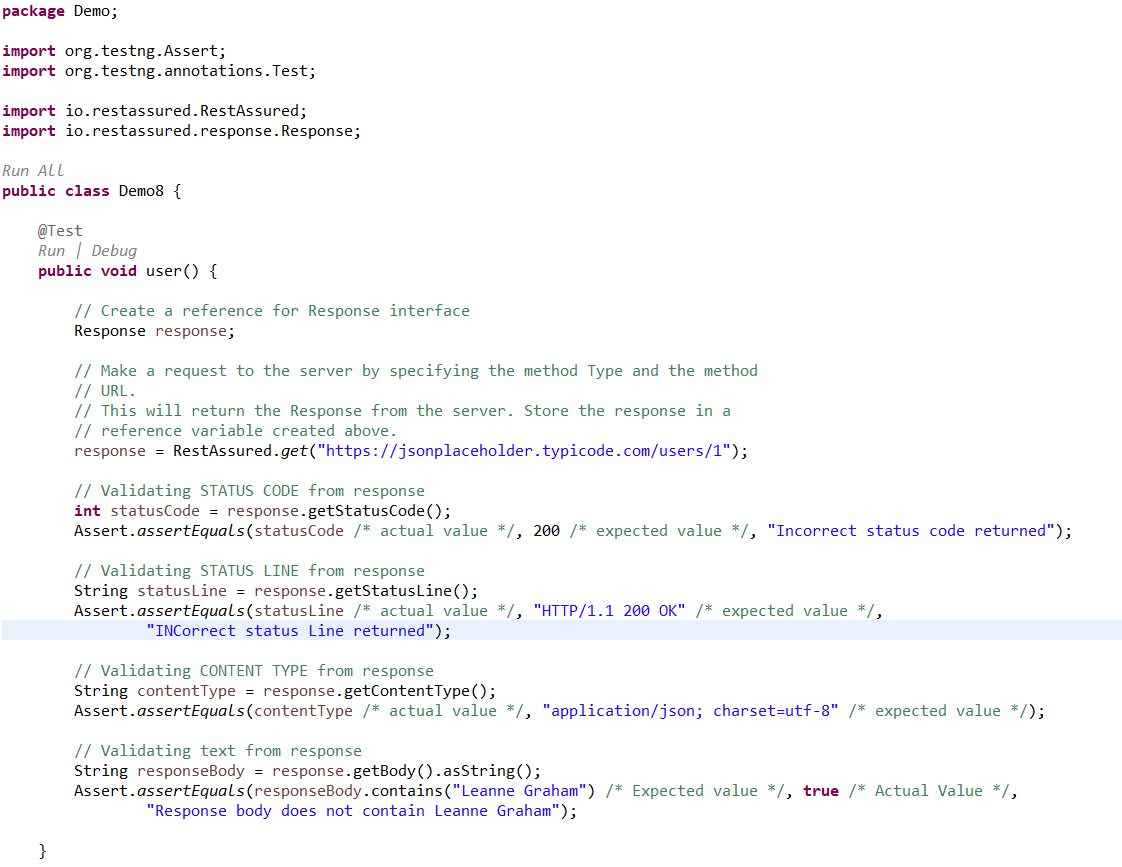
1. *//Validating CONTENT TYPE from response*
2. String contentType = response.getContentType();
3. Assert.assertEquals(contentType */\* actual value \*/*, "application/json;charset=UTF-8" */\* expected value \*/*);

**Step 6:**Convert the body of the response to a string and validate a particular text from the response using contains() method.

1. *//Validating text from response*
2. String responseBody = response.getBody().asString();
3. Assert.assertEquals(responseBody.contains("John") */\*Expected value\*/*, true */\*Actual Value\*/*, "Response body does not contain John");

**Step 7:** In the same way, different headers of the response can be validated.

**Step 8:** The final script looks like the one given below:



**Step 9:** Run the TestNG class.

**Output:**

Assertion validates the actual value with expected value. If actual value is same as expected value, assertion passes. Else, assertion fails and prints the given error message in the console window.

**Parameterization** is the separation of data components from the flow control component of the script.

# Need for Parameterization:

Consider the following scenario.

If you have to test 200 test cases to get 200 types of products in an inventory management system of an e-commerce website, the flow through the application screens and the fields used would be the same. However, the requests to be populated and the responses returned by the service would be different. If we parameterize this test, we will end up with a single flow control script and the data (test data and expected message texts) populated in 200 rows of an excel file – one for each test case. During execution, the same flow control script would be running iteratively, once for each row of data.

This approach will:

* Reduce the amount of code to be maintained. Instead of 200 test methods, you will have only one test method to maintain.
* Allow non-technical testers (acceptance testers and business users) to add, remove or modify test cases easily by just working with the file which stores test data.

In order to implement parameterization, it is necessary to know how to integrate excel workbooks with your REST Assured script. The POI interface will help you do that.

Apache POI is a popular API that allows Java programmers to create, read and edit Microsoft Office files.

Its JAR files, which contain the class library, can be downloaded and used for free from Apache's download portal.

The classes that you need to know about for working with excel files are:

Next, let's look at how we can automate reading from an excel file and parameterize a REST Assured script.

# *****Demo 9 Parameterization to read the data from the file and push it into web service*****

**Prerequisite:**

* Configure the project with the Apache POI jar file.

**Highlights:**

* Perform HTTP Get on restful web services using parameterization to read the data from the file and push it into web service

**Demo Steps:**

**Step 1:**Download the excel file having the test data to parameterize.

**Step 2:** In the method annotated with @Test annotation, create a reference for the Response interface.

1. *//Create a reference for Response interface*
2. Response response;

**Step 3:** Create a reference for the FileInputStream class and pass the excel file location downloaded in Step 1.

1. *//Path from where the excel file has to be read*
2. String path = "D:\\get.xlsx";
4. *//File input stream which needs the input as the file location*
5. FileInputStream fis = new FileInputStream(path);

**Step 4:** Create a reference for the XSSFWorkbook class available in the POI jar file and pass the fileStream reference variable as an argument to access the excel workbook.

1. *//Workbook reference of the excel file*
2. XSSFWorkbook wb = new XSSFWorkbook(fis);

**Step 5:**Create a reference for XSSFSheet class available in the POI jar file and pass the workbook reference variable as an argument to access the sheet present in the workbook.

1. *//Sheet which needs to be accessed from within the workbook*
2. XSSFSheet sheet = wb.getSheetAt(0);

**Step 6:**Count the number of rows present in the excel file.

1. *//Count the number of rows*
2. int rowCount = sheet.getLastRowNum() - sheet.getFirstRowNum();

**Step 7:** Iterate over all the rows present in the excel sheet to fetch the values from the respective cell and parameterize the test data script.

1. *//Iterate the AccountNumber*
2. for(int i=1;i<=rowCount;i++) {
3. *//code here*
5. }

**Step 8:** For every iteration, pass the data retrieved in a variable.

1. *//Iterate the AccountNumber*
2. for(int i=1;i<=rowCount;i++) {
4. *//Pass the row number and the cell number from where the value has to be fetched*
5. String accno = sheet.getRow(i).getCell(0).getStringCellValue();
7. }

**Step 9:** Make a HTTP Method Type “get()” with the input as data retrieved from the excel file using get method from RestAssured class with a specified base URI and store the response.

1. *//Iterate the AccountNumber*
2. for(int i=1;i<=rowCount;i++) {
4. *//Pass the row number and the cell number from where the value has to be fetched*
5. String id = sheet.getRow(i).getCell(0).getStringCellValue();
6. *// Make a request to the server by specifying the method Type and the method URL.*
7. *// This will return the Response from the server. Store the response in a reference variable created above.*
8. response = RestAssured.get("http://localhost:8080/InfyClinicV2/APP\_V2/getDoctor/byId?doctorId=" + id + "");
10. }

**Step 10:** Return the body of the received response using getBody() method of Response Interface. Response Body is converted into a string value using the asString() method and printed on the console using the System.out.println statement.

1. *//Iterate the AccountNumber*
2. for(int i=1;i<=rowCount;i++) {
4. *//Pass the row number and the cell number from where the value has to be fetched*
5. String id = sheet.getRow(i).getCell(0).getStringCellValue();
6. *// Make a request to the server by specifying the method Type and the method URL.*
7. *// This will return the Response from the server. Store the response in a reference variable created above.*
8. response = RestAssured.get("http://localhost:8080/InfyClinicV2/APP\_V2/getDoctor/byId?doctorId= + "");
9. //print the message body of the response received from the server
10. String responseBody = response.getBody().asString();
11. System.out.println("Response Body is => " + responseBody);
12. }

**Step 11:** The final script looks like the one given below:

**package** Demo;

/\*import java.io.FileInputStream;

import java.io.IOException;

import org.apache.poi.xssf.usermodel.XSSFSheet;

import org.apache.poi.xssf.usermodel.XSSFWorkbook;

import org.testng.annotations.Test;

import io.restassured.RestAssured;

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

**public** **class** Demo9 {

@Test

**public** **void** user() **throws** IOException {

// Create a reference for Response interface

/\*Response response;

// Path from where the excel file has to be read

String path = "D:\\get.xlsx";

// File input stream which needs the input as the file location

FileInputStream fis = new FileInputStream(path);

// Workbook reference of the excel file

XSSFWorkbook wb = new XSSFWorkbook(fis);

// Sheet which needs to be accessed from within the workbook

XSSFSheet sheet = wb.getSheetAt(0);

// Count the number of rows

int rowCount = sheet.getLastRowNum() - sheet.getFirstRowNum();

// Iterate the AccountNumber

for (int i = 1; i <= rowCount; i++) {

// Pass the row number and the cell number from where the value has to be

// fetched

String id = sheet.getRow(i).getCell(0).getStringCellValue();

// 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

// reference variable created above.

response = RestAssured.get("http://localhost:8080/InfyClinicV2/APP\_V2/getDoctor/byId?doctorId=" + id + "");

// print the message body of the response received from the server

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

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

}\*/

}

}

**Step 12:** Run the TestNG class file.

**Output:**

The details of the doctor with doctor Id given in the get request are displayed.

# HTTP POST:

* The POST method is used to add a new resource to the collection of resources. It accepts the data enclosed in the body of request message and stores it in the database.
* On adding the resource into the collection, it returns a response code 201 which contains Location header with a link to the newly added resource.
* Since it makes changes to the existing resources, it is considered an unsafe method. Calling the POST method multiple times will result in the addition of resources containing the same information.

**To perform a POST operation in REST Assured**:

* POST method is used to send request data to the server. The data that is sent to the server in a POST request is sent in the body of HTTP request. The type of body, XML, JSON or some other format is defined by the Content-Type header.
* If a POST request contains JSON data then the Content-Type header will have a value of application/JSON. Similarly, if a POST request contains XML the Content-Type header value will be application/XML.

1. request.header(request data format)

* JSONObject is a class imported from org.json.simple package. It is a programmatic representation of a JSON string. To add each node of the JSON string, we use JSONObject.put(String, String) method. After the addition of nodes in JSON string,we can convert the JSON object to its String representation by using toJSONString() method.
* You can put the JSON string in the body using the method called RequestSpecification.body(JsonString). It allows you to update the content of HTTP request. However, if you call the method multiple times the body will be updated to the latest JSON String.
* The response can be fetched as JSON object or XML file.
* The demos will help you understand how to post a request and fetch a response in JSON as well as in XML format.

# ***Demo 10: HTTP POST with JSON request and JSON response***

**Highlights:**

* Perform HTTP Post on RESTful web services with JSON request and JSON response
* Validating the response obtained upon POST request.

**Demo Steps:**

**Step 1:**In the method annotated with @Test annotation, create a request pointing to the required end point.

1. *// Specify the base URL to the RESTful web service*
2. RestAssured.baseURI = "http://localhost:8081/InfyClinic/APP\_V1/FixAppointment";
3. *// Get the RequestSpecification of the request that you want to sent to the server*
4. *// The server is specified by the BaseURI that we have specified in the above step*
5. RequestSpecification request = RestAssured.given();

**Step 2:**The content type of the request can be specified by adding the below header.

1. *// Add a header stating the Request body is a JSON*
2. request.header("Content-Type", "application/json");

**Step 3:**Create an object of JSONObject class to add each node in the request.

1. *// Create an object for JSONObject class*
2. JSONObject requestParams = new JSONObject();

**Step 4:** Each node can be added using JSONObject.put(String, String) method.

1. *// We can add key-Value pairs using put method*
2. requestParams.put("patientid", "1006");
3. requestParams.put("patientName", "nikhil");
4. requestParams.put("age", "24");
5. requestParams.put("temperature","095.20");
6. requestParams.put("diagnosis","High fever");

**Step 5:** Once all the nodes are added, the String representation of JSONObject can be obtained by calling JSONObject.toJSONString() method.

**Step 6:** Add the obtained String form of JSON in the request body and send the Request.

1. *// Add string form of JSON to the body of the request*
2. request.body(requestParams.toString());

**Step 7:**Obtain the response as with Post method and store it to the reference of Response interface.

1. *// Post the request and check the response*
2. Response response = request.post("/AdmitPatient");

**Step 8:**The status code upon successfull patient addition can be printed in console using getStatusCode() method. The obtained result can be validated using assertions.

1. *//Obtain and print the response status code*
2. int statusCode = response.getStatusCode();
3. System.out.println(statusCode);
4. *//Asserting the response*
5. Assert.assertEquals(statusCode */\*actual value\*/*, 200 */\*expected value\*/*, "INCorrect status code returned");

**Step 9:**The final script looks like the one given below:

1. package Demo;
2. import org.testng.Assert;
3. import org.testng.annotations.Test;
4. import org.json.\*;
5. import io.restassured.RestAssured;
6. import io.restassured.response.Response;
7. import io.restassured.specification.RequestSpecification;
8. public class Demo11 {
9. @Test
10. public void user() throws JSONException {

13. *// Specify the base URL to the RESTful web service*
14. RestAssured.baseURI = "http://localhost:8081/InfyClinic/APP\_V1/FixAppointment";
15. *// Get the RequestSpecification of the request that you want to sent to the server*
16. *// The server is specified by the BaseURI that we have specified in the above step*
17. RequestSpecification request = RestAssured.given();
19. *// Add a header stating the Request body is a JSON*
20. request.header("Content-Type", "application/json");
21. *// Create an object for JSONObject class*
22. JSONObject requestParams = new JSONObject();
24. *// We can add key-Value pairs using put method*
25. requestParams.put("patientid", "1006");
26. requestParams.put("patientName", "nikhil");
27. requestParams.put("age", "24");
28. requestParams.put("temperature","095.20");
29. requestParams.put("diagnosis","High fever");
30. *// Add string form of JSON to the body of the request*
31. request.body(requestParams.toString());
32. *// Post the request and check the response*
33. Response response = request.post("/AdmitPatient");
34. *//Obtain and print the response status code*
35. int statusCode = response.getStatusCode();
36. System.out.println(statusCode);
37. *//Asserting the response*
38. Assert.assertEquals(statusCode */\*actual value\*/*, 200 */\*expected value\*/*, "INCorrect status code returned");
39. }
40. }

**Step 10:**Run the TestNG class file.

**Output:**

Prints the status code with respect to details that were added on post request in the console window and validates the status code obtained upon post request.

# Assertions:

* Once we get the response from the server, all we have to do is validate the parts of the response.
* So in POST operation, we can validate the success code which we can fetch from the response in JSON format.
* To access the data in JSON format, we will use JsonPath(). JsonPath is an alternative to XPath for getting values easily from a JSON document.
* Assert.assertEquals() method is used to compare the values fetched from the response and the expected value. If the values do not match, an AssertionError is thrown.

***Demo 12: Parameterization to read the data from file and push it into web service***

**Prerequisite:**

* Configure the project with Apache POI jar file.

**Highlights:**

* Perform HTTP POST on RESTful web services using parameterization to read the data from file and push it into web service

**Demo Steps:**

**Step 1:**  Download the excel file having the test data to parameterize.

**Step 2:**In the method annotated with @Test annotation, create a reference for Response interface.

1. *// Create a reference for Response interface*
2. Response response;

**Step 3:**Create a request pointing to the required end point.

1. *// Specify the base URI to the RESTful web service*
2. RestAssured.baseURI = "http://localhost:8081/InfyClinic/APP\_V1";
4. *// Get the RequestSpecification of the request that you want to sent to the server*
5. RequestSpecification request = RestAssured.given();

**Step 4:** The content type of the request can be specified by adding the below header.

1. *// Add a header stating the Request body is a JSON*
2. request.header("Content-Type", "application/json");

**Step 5:**Create an object of JSONObject class to add each node in the request.

1. *// Create an object for JSONObject class*
2. JSONObject requestParams = new JSONObject();

**Step 6:**Create a reference for FileInputStream class and pass the excel file location downloaded in Step 1.

1. *// Path from where the excel file has to be read*
2. String path = "D:\\post.xlsx";
3. *// File input stream which needs the input as the file location*
4. FileInputStream fis = new FileInputStream(path);

**Step 7:** Create a reference for XSSFWorkbook class available in POI jar file and pass the FileInputStream reference variable as an argument to access the excel workbook.

1. *// Workbook reference of the excel file*
2. XSSFWorkbook wb = new XSSFWorkbook(fis);

**Step 8:** Create a reference for XSSFSheet class available in POI jar file and pass the workbook reference variable as an argument to access the sheet present in the workbook.

1. *// Sheet which needs to be accessed from within the workbook*
2. XSSFSheet sheet = wb.getSheetAt(0);

**Step 9:** Count the number of rows present in the excel file.

1. *// Count the number of rows*
2. int rowNo = sheet.getLastRowNum() - sheet.getFirstRowNum();

**Step 10:**Iterate over all the rows present in the excel sheet to fetch the values from the respective cell and parameterize the test data script.

1. *//Iterate the rows of excel sheet*
2. for (int i = 1; i <= rowNo; i++) {
3. *//code here*
4. }

**Step 11:**Each node can be added using JSONObject.put(String, String) method.

Pass the first parameter as the node name and second parameter as the input value that is iterated and retrieved from the excel.

1. *//Iterate the the rows of excel sheet*
2. for (int i = 1; i <= rowNo; i++) {
3. *//We can add key-Value pairs using put method*
4. requestParams.put("patientName", sheet.getRow(i).getCell(0).getStringCellValue());
5. requestParams.put("age", sheet.getRow(i).getCell(1).getNumericCellValue());
6. requestParams.put("temperature", sheet.getRow(i).getCell(2).getNumericCellValue());
7. }

**Step 12:** Once all the nodes are added, the String representation of JSONObject can be obtained by calling JSONObject.toJSONString() method.

1. *//Iterate the rows of excel sheet*
2. for (int i = 1; i <= rowNo; i++) {
3. *//We can add key-Value pairs using put method*
4. requestParams.put("patientName", sheet.getRow(i).getCell(0).getStringCellValue());
5. requestParams.put("age", sheet.getRow(i).getCell(1).getNumericCellValue());
6. requestParams.put("temperature", sheet.getRow(i).getCell(2).getNumericCellValue());
7. *// Add string form of JSON to the body of the request*
8. request.body(requestParams.toString());
9. }

**Step 13:**Add the obtained String form of JSON in the request body and send the request.

1. *//Iterate the rows of excel sheet*
2. for (int i = 1; i <= rowNo; i++) {
3. *//We can add key-Value pairs using put method*
4. requestParams.put("patientName", sheet.getRow(i).getCell(0).getStringCellValue());
5. requestParams.put("age", sheet.getRow(i).getCell(1).getNumericCellValue());
6. requestParams.put("temperature", sheet.getRow(i).getCell(2).getNumericCellValue());
7. *// Add string form of JSON to the body of the request*
8. request.body(requestParams.toString());
9. }

**Step 14:**Obtain the response as with POST method.

1. *//Iterate the rows of excel sheet*
2. for (int i = 1; i <= rowNo; i++) {
3. *//We can add key-Value pairs using put method*
4. requestParams.put("patientName", sheet.getRow(i).getCell(0).getStringCellValue());
5. requestParams.put("age", sheet.getRow(i).getCell(1).getNumericCellValue());
6. requestParams.put("temperature", sheet.getRow(i).getCell(2).getNumericCellValue());
7. *// Add string form of JSON to the body of the request*
8. request.body(requestParams.toString());
10. *//Post the request and check the response*
11. response = request.post("/AdmitPatient");
12. int statusCode = response.getStatusCode();
13. Assert.assertEquals(statusCode */\*actual value\*/*, 200 */\*expected value\*/*, "INCorrect status code returned");
14. }

**Step 15:** The final script looks like the one given below:

1. package Demo;
2. import java.io.FileInputStream;
3. import java.io.IOException;
4. import org.apache.poi.xssf.usermodel.XSSFSheet;
5. import org.apache.poi.xssf.usermodel.XSSFWorkbook;
6. import org.json.JSONException;
7. import org.json.JSONObject;
8. import org.testng.Assert;
9. import org.testng.annotations.Test;
10. import io.restassured.RestAssured;
11. import io.restassured.response.Response;
12. import io.restassured.specification.RequestSpecification;
13. public class Demo12 {
14. @Test
15. public void user() throws IOException, JSONException {
17. *// Create a reference for Response interface*
18. Response response;
19. *// Specify the base URI to the RESTful web service*
20. RestAssured.baseURI = "http://localhost:8081/InfyClinic/APP\_V1";
21. *// Get the RequestSpecification of the request that you want to sent to the server*
22. RequestSpecification request = RestAssured.given();
23. *// Add a header stating the Request body is a JSON*
24. request.header("Content-Type", "application/json");
25. *// Create an object for JSONObject class*
26. JSONObject requestParams = new JSONObject();
27. *// Path from where the excel file has to be read*
28. String path = "C:\\Users\\doddi.sowmya\\Desktop\\accno.xlsx";
29. *// File input stream which needs the input as the file location*
30. FileInputStream fis = new FileInputStream(path);
31. *// Workbook reference of the excel file*
32. XSSFWorkbook wb = new XSSFWorkbook(fis);
33. *// Sheet which needs to be accessed from within the workbook*
34. XSSFSheet sheet = wb.getSheetAt(0);
35. *// Count the number of rows*
36. int rowNo = sheet.getLastRowNum() - sheet.getFirstRowNum();
37. *// Iterate the rows of excel sheet*
38. for (int i = 1; i <= rowNo; i++) {
39. *// We can add key-Value pairs using put method*
40. requestParams.put("patientName", sheet.getRow(i).getCell(0).getStringCellValue());
41. requestParams.put("age", sheet.getRow(i).getCell(1).getNumericCellValue());
42. requestParams.put("temperature", sheet.getRow(i).getCell(2).getNumericCellValue());
43. *// Add string form of JSON to the body of the request*
44. request.body(requestParams.toString());
45. *// Post the request and check the response*
46. response = request.post("/AdmitPatient");
47. int statusCode = response.getStatusCode();
48. Assert.assertEquals(statusCode */\*actual value\*/*, 200 */\*expected value\*/*, "INCorrect status code returned");
49. }
50. }
51. }

**Step 16:**Run the TestNG class file.

**Output:**

Prints the details that were added on POST request in the console window.

Framing Nested JSON object.

If JSON data to be sent to server is of below format.

1. {
2. "Designation": "Test Engineer",
3. "name": "John",
4. "id": 100,
5. "Contact": {
6. "Official Number": 1234567890,
7. "Personal Number": 1234567891,
8. "Land line": 223344,
9. "Email id": "abc@gmail.com"
10. }
11. }

In the above example provided, the data to be sent is not simple Key: Value pair instead it is a Nested Object.

The Value for the Key “Contact” is one more JSON Object.

Will see how to structure Nested JSON object.

1. import org.json.JSONException;
2. import org.json.JSONObject;
3. import org.testng.annotations.Test;
4. public class Demo1 {
5. @Test
6. public void f() throws JSONException {
8. JSONObject parameters = new JSONObject();
9. parameters.put("name", "John");
10. parameters.put("Designation", "Test Engineer");
11. parameters.put("id", 100);
13. *//Create new JSON Object to store contact details*
14. JSONObject details = new JSONObject();
15. details.put("Official Number", 1234567890);
16. details.put("Personal Number",1234567891);
17. details.put("Land line", 223344);
18. details.put("Email id", "abc@gmail.com");
20. *//make details as value to "Contact" Key*
21. parameters.put("Contact", details);
23. *//print the JSON object as String in the console*
24. System.out.println(parameters.toString());
25. }
26. }

Once the required JSON data is ready it can be placed in the request body & POST to the required URI.

If JSON data to be sent to server is of below format.

1. {
2. "Designation": "Test Engineer",
3. "name": "John",
4. "id": 100,
5. "Contact": [
6. {
7. "Official Number": 1234567890,
8. "Personal Number": 1234567891,
9. "Land line": 223344,
10. "Email id": "abc@gmail.com"
11. }
12. ]
13. }

Here the Value to the Key “Contact" is a JSON Array.

Will see how to pass JSON Array as value to the Key

1. import org.json.JSONArray;
2. import org.json.JSONException;
3. import org.json.JSONObject;
4. import org.testng.annotations.Test;
5. public class Demo2 {
6. @Test
7. public void f() throws JSONException {
9. JSONObject parameters = new JSONObject();
10. parameters.put("name", "John");
11. parameters.put("Designation", "Test Engineer");
12. parameters.put("id", 100);
13. *//Create instance of JSONObject to store contact details*
14. JSONObject details = new JSONObject();
15. details.put("Official Number", 1234567890);
16. details.put("Personal Number",1234567891);
17. details.put("Land line", 223344);
18. details.put("Email id", "abc@gmail.com");
20. *//Create instance of JSONArray*
21. JSONArray contactDetails = new JSONArray();
23. *//Move the created JSONObject to JSONArray*
24. contactDetails.put(details);
26. *//Make contactDetails as value to "Contact" Key*
27. parameters.put("Contact", contactDetails);
29. *//print the JSON object as String in the console*
30. System.out.println(parameters.toString());
31. }
32. }

If JSON data to be sent to server is of below format.

1. {
2. "Designation": "Test Engineer",
3. "name": "John",
4. "id": 100,
5. "Contact": [
6. {
7. "Official Number": 1234567890,
8. "Personal Number": 1234567891,
9. "Land line": 223344
10. },
11. {
12. "Email id2": "def@gmail.com",
13. "Email id1": "abc@gmail.com"
14. }
15. ]
16. }

In the above example provided the Key “Contact” has JSON Array as values.

The code to frame such JSON Object is as follows

1. import org.json.JSONArray;
2. import org.json.JSONException;
3. import org.json.JSONObject;
4. import org.testng.annotations.Test;
5. public class Demo3 {
6. @Test
7. public void f() throws JSONException {
9. JSONObject parameters = new JSONObject();
10. parameters.put("name", "John");
11. parameters.put("Designation", "Test Engineer");
12. parameters.put("id", 100);
14. *//Create instance of JSONObject to store contact numbers*
15. JSONObject detailsNumber = new JSONObject();
16. detailsNumber.put("Official Number", 1234567890);
17. detailsNumber.put("Personal Number",1234567891);
18. detailsNumber.put("Land line", 223344);
20. *//Create instance of JSONObject to store Email id's*
21. JSONObject detailsEmail = new JSONObject();
22. detailsEmail.put("Email id1", "abc@gmail.com");
23. detailsEmail.put("Email id2", "def@gmail.com");
25. *//Create instance of JSONArray*
26. JSONArray contactDetails = new JSONArray();
27. *//Move the created JSONObjects to JSONArray*
28. contactDetails.put(detailsNumber);
29. contactDetails.put(detailsEmail);
31. *//Make contactDetails as value to "Contact" Key*
32. parameters.put("Contact", contactDetails);
34. *//print the JSON object as String in the console*
35. System.out.println(parameters.toString());
36. }
37. }

# DELETE:

* As the name suggests, we use the Delete method to delete an existing resource identified by URI.
* After deleting a resource, it returns HTTP status 200 with a response body that contains either representation of the deleted item or wrapped response. Another HTTP status 204 will be returned if there is no content.
* DELETE operations are idempotent. If you remove a resource from the list of resources, it stays removed even if you passed the same request “n” number of times.
* Calling the Delete method on a deleted resource for the second time will return an HTTP status 404.DELETE is an unsafe method because it modifies the resources.

**To delete an entry and assert the response**:

* The syntax for Delete request is no different, we just have to call the delete() method instead of get().
* delete() method will delete an entry and in response, we can validate whether the entry is deleted or not with the help of status code using Assert.assertEquals()

Maven is a build automation tool used primarily for Java projects. Maven uses **convention** over **configuration** which means developers are not required to create build process themselves. Maven provides sensible default behaviour for projects. When a Maven project is created, it creates default project structure and developer is only required to place files accordingly.

There can be various problems faced during the project development such as :

1. Adding set of Jars in each project: In case of selenium projects, multiple jar files in each project must be added.
2. Dependencies and Versions:  Ensuring that the jar files and the required dependencies are added to the project for developing, compiling and executing the same.

# Maven structure

POM is fundamental Unit of Work in Maven. It is an XML file which resides in the base directory of the project as **pom.xm**l. **POM** contains information about the project and various configuration detail used by Maven to build the projects.

POM also contains the goals and plugins. While executing a task or goal, Maven looks for the POM in the current directory. It reads the POM, gets the needed configuration information, then executes the goal.

A maven repository is a directory of packaged JAR file with pom.xml file. Maven searches for dependencies in the repositories. There are three types of maven repository:

1. Local Repository
2. Central Repository
3. Remote Repository

Maven searches for the dependencies in the following order:

**Note:** If dependency is not found in these repositories, maven stops processing and throws an error.

**Maven Local Repository**

Maven local repository is located in your local system. It is created by the Maven when you run any Maven command.

By default, maven local repository is **%USER\_HOME%\.m2** directory. For example: C:\Users\<<Username>>\.m2 .

**Maven Central Repository**

Maven central repository is located on the web. It has been created by the apache Maven community itself.

Then, access the central repository. The central repository contains a lot of common libraries that can be viewed by using the search page.

**Maven Remote Repository**

Maven remote repository is located on the web. Some libraries can be missing from the central repository and may be required to be downloaded from the individual repositories by the firm. For Example:  JBoss library file needs to be downloaded from the JBoss nexus repository.

**Note:** a Nexus repository is used for working to download dependencies instead of maven central repository. Nexus is a repository manager which proxies and cache’s external repositories. Let us proceed on to understand, how to configure the system to make use of Nexus repository while working with Maven and the installation of Maven plugin in eclipse IDE.

* Java Setup

We will need to install Java on our machines or writing our REST API automation framework based on the Rest Assured library.

* IDE Setup

***Eclipse, IntelliJ, Net Beans,*** and several others are popular IDEs you can choose to work As we will be working with Java,

* Maven Setup

We will use the Maven build tool for our End To End Scenarios. Please install Maven, if not previously installed .

We will need to install Java on our machines or writing our REST API automation framework based on the Rest Assured library.

* Create Maven Project

After opening Eclipse, choose the workspace you want to use. The Eclipse window opens on the screen. Since there aren’t any projects yet, complete the following steps:

* Go to the File option
* In the drop-down menu, select New
* Select the Project option Maven
* Add Rest Assured Dependencies

We will add ***Rest Assured Dependencies*** to our project through the pom.xml file. For All  required dependencies, go to <https://mvnrepository.com/> Then, select the latest dependency. Copy-paste it in the project ***pom.xml*** file.

* Setup Maven Compiler Plugin

The ***Compiler Plugin*** compiles the sources of the project. Regardless of the JDK you run Maven with, the default source setting is 1.5, and the default target setting is 1.5.

Demo11:

* Creation of maven project for Rest Assure Project
* Configuration of pom.xml file

Demo steps:

Step 1**:** Create a maven project in Eclipse IDE. Go to File -> New -> Other... and search for Maven .Choose Maven Project and click on Next.

Step 2**:** Select the checkbox corresponding to **Create a simple project(skip archetype selection)**and click on**Next.**

Step 3**:** Enter values for the field **GroupId** and **ArtifactId**in the **New Maven Project** window, and click on **Finish.**

Group Id field is similar to the package name and Artifact Id is your project name. For an organization with different projects, the Group Id for all their projects will be the same.

The version value can be changed to the current version, if you are working on a previously created project . Currently the Version field will have the value Snapshot as the project is under development.

Packaging signifies that the end result after the project has been compiled.

Step 4: Go to the Package Explorer tab to access the newly created maven project. Open the pom.xml file, which resides in the project folder and go to the pom.xml tab to see the auto generated code.

Step 5: Modify the pom.xml file, by adding the following lines as by default pom will be taking JRE 1.5  and we need 1.7 or above to  execute the project. We are setting the JRE to be used as 1.8 by using the below code.

1. <build>
2. <plugins>
3. <plugin>
4. <groupId>org.apache.maven.plugins</groupId>
5. <artifactId>maven-compiler-plugin</artifactId>
6. <configuration>
7. <source>1.8</source>
8. <target>1.8</target>
9. </configuration>
10. </plugin>
11. </plugins>
12. </build>

**Step 6**: To add the dependencies for the project , you will have to provide three parameters, i.e. groupId,  artifactId, version corresponding to the jar file you need for running the project. Navigate to [Maven Repository](http://www.mvnrepository.com/) and search for the artifact and version you want for your project.? Let’s search for Rest Assure  in the search tab.

Step 7: Click on [REST Assured](https://mvnrepository.com/artifact/io.rest-assured/rest-assured), choose the version as 5.1.1

Step 8: Copy the artifact details from the pane displayed on the page.

1. *<!-- https://mvnrepository.com/artifact/io.rest-assured/rest-assured -->*
2. <dependency>
3. <groupId>io.rest-assured</groupId>
4. <artifactId>rest-assured</artifactId>
5. <version>5.1.1</version>
6. <scope>test</scope>
7. </dependency>

Step 9: Add a tag called as dependencies in the pom.xml within the project tag. Inside the dependencies tag, provide the dependencies Rest Assure, Junit, TestNg for your project.  Please ensure that the dependencies for are added in the pom.xml. Please note that the version number can be different for different artifacts depending upon the time of execution.

1. <dependencies>
2. <!-- https:*//mvnrepository.com/artifact/io.rest-assured/rest-assured -->*
3. <dependency>
4. <groupId>io.rest-assured</groupId>
5. <artifactId>rest-assured</artifactId>
6. <version>5.1.1</version>
7. <scope>test</scope>
8. </dependency>
9. <dependency>
10. <groupId>junit</groupId>
11. <artifactId>junit</artifactId>
12. <version>4.13</version>
13. <scope>test</scope>
14. </dependency>
15. <!-- https:*//mvnrepository.com/artifact/org.testng/testng -->*
16. <dependency>
17. <groupId>org.testng</groupId>
18. <artifactId>testng</artifactId>
19. <version>6.0</version>
20. <scope>test</scope>
21. </dependency>
22. </dependencies>

Step 10:Right click on the project--> Maven-->Update Project .

Step 11: Choose the option Force update of Snapshots/releases and click on ok. This would trigger the download of dependencies from the repository.   
   
Step12: Go to the   src/test/java folder in the project created  Right Click-→New -→Package -→give package Name

Step13: Right click on package created and create a new TestNg class

Step 14: Write the below code in the TestNg class created to get all questions from the

"http://10.82.180.36:8080/AmigoWallet/RegistrationAPI"

1. package samplerestassure;
2. import org.testng.annotations.Test;
3. import io.restassured.RestAssured;
4. import io.restassured.response.Response;
5. public class Demo1 {
6. Response response;
7. @Test
8. public void f() {
9. response=RestAssured.get("http://10.82.180.36:8080/AmigoWallet/RegistrationAPI/getAllQuestions");
10. String getbody=response.getBody().asString();
11. System.out.println(getbody);
12. }
13. }

Step15: Run the TestNg class to get the response from the service

Step16:Observe the output in console which displays all questions from the "http://10.82.180.36:8080/AmigoWallet/RegistrationAPI"

# Behavior Driven Development (BDD) methodology is a refinement of TDD and ATDD by implementing the following 4 practices:

## **1. Specification by Example (SbE):**

Writing acceptance test cases for features in free-form English has a possibility of developers misinterpreting it as it would happen from a requirement specification document too.

For business analysts to easily create acceptance tests using plain English and also for developers to understand the acceptance tests in an unambiguous way, they are written as **behaviors**.

A behavior is a way of describing an acceptance with high specificity using **Given-When-Then** notation.

**Given:** Describes the specific context/initial conditions of the acceptance test.

**When:** Describes the specific input values supplied and/or specific actions carried out

**Then:** Describes the specific observable consequences on the application under the corresponding ‘Given’ and ‘When’ conditions.

**Example:**

**Given:** The screen is displaying the login page

**When:** The user enters user id as 'admin', password as 'admin123' and clicks on 'Submit' button

**Then:** The application navigates to the welcome page

This technique of documentation is called **Specifications by Example** (SbE) because:

Each behavior is documented with the help of concrete examples on how the application feature must behave

Acceptance tests are used as specifications in ATDD methodology

## **2. The Three Amigos model of working:**

In order to succeed in ATDD, every addition or modification at any level (acceptance tests, unit tests and the actual code) needs to be analyzed from three perspectives.

**Business:** What is the problem we are trying to solve?

**Developer:** How should the solution be implemented?

**Tester:** How to validate whether the implemented solution solves the problem?

The team must be made up of people who

Possess the skillset to address the three perspectives.

Develop a shared understanding of the relationship between acceptance tests and unit tests used.

Co-located to work together and reduce the feedback times.

In Agile world, this is called 'The Three Amigos' model of working.

## **3. A shared tool to implement SbE based ATDD tests:**

Though SbE based acceptance tests can be constructed even without the help of a tool, using it helps in:

Automating the acceptance tests i.e. **executable specifications**, by invoking automated tests in other testing tools such as Selenium, UFT etc.

Serving as a centralized repository of specifications/ acceptance tests.

Eliminating tedious manual tasks like version control, handover and allocations, by becoming a part of ecosystem of tools (like TFS, Jenkins, Maven etc.) used in the project.

There are a lot of BDD tools in the market which implement SbE like RSpec, Cucumber, JBehave etc.

## **4. An automation testing framework to implement TDD and ATDD tests:**

Given the number and complexities involved in automating and maintaining a large test suite, TDD and ATDD is virtually impossible to implement without a test automation framework like JUnit, VBUnit, TestNG etc.

In this course, for demo purposes, we will be using JUnit framework to construct Selenium tests.

Now that you are aware of the principles behind BDD, we will look at how projects implement BDD using Cucumber.

### We can use  **Cucumber BDD Framework** to execute Rest API tests. It would require us to **Convert our Rest Assured API Tests to the Cucumber BDD Style Test**.

We can achieve the same by the below steps

1. Add Cucumber Dependencies to the Project
2. Write a test in a Feature File
3. Write test code to the Step file
4. Create a Test Runner
5. Run the test as a JUnit test

**Step1**: Add below cucumber dependencies to the Maven project

1. <dependencies>
2. <dependency>
3. <groupId>io.cucumber</groupId>
4. <artifactId>cucumber-java</artifactId>
5. <version>5.2.0</version>
6. </dependency>
7. <dependency>
8. <groupId>io.cucumber</groupId>
9. <artifactId>cucumber-jvm-deps</artifactId>
10. <version>1.0.6</version>
11. <scope>provided</scope>
12. </dependency>
13. *<!-- https://mvnrepository.com/artifact/io.cucumber/cucumber-junit -->*
14. <dependency>
15. <groupId>io.cucumber</groupId>
16. <artifactId>cucumber-junit</artifactId>
17. <version>5.2.0</version>
18. <scope>test</scope>
19. </dependency>
20. <dependency>
21. <groupId>io.rest-assured</groupId>
22. <artifactId>rest-assured</artifactId>
23. <version>5.1.1</version>
24. <scope>test</scope>
25. </dependency>
26. *<!-- https://mvnrepository.com/artifact/io.rest-assured/json-schema-validator -->*
27. <dependency>
28. <groupId>junit</groupId>
29. <artifactId>junit</artifactId>
30. <version>4.13</version>
31. <scope>test</scope>
32. </dependency>
33. *<!-- https://mvnrepository.com/artifact/org.testng/testng -->*
34. <dependency>
35. <groupId>org.testng</groupId>
36. <artifactId>testng</artifactId>
37. <version>6.0</version>
38. <scope>test</scope>
39. </dependency>
40. </dependencies>

**Step2**:Create a package samplerestassurebdd in the src/test/java folder

**Step3:**Create feature file student. feature under the package

**Step4:**Write the below code in the feature file

1. Feature: Check for Student Details
2. Scenario: Get Student details
3. Given A list of students are available
4. When Get the student details
5. Then Validate status code and status line

**Step5:**Run the feature file using Right click-→Run As-→cucumber feature

**Step6:**You will get the Snippet for feature file .

**Step7:**Implement the step definition file to fetch the details requested from the service

1. package samplerestassurebdd;
2. import org.testng.Assert;
3. import io.cucumber.java.en.Given;
4. import io.cucumber.java.en.Then;
5. import io.cucumber.java.en.When;
6. import io.restassured.RestAssured;
7. import io.restassured.path.json.JsonPath;
8. import io.restassured.response.Response;
9. public class Student {
10. Response response;
11. JsonPath json\_res;
12. @Given("A list of students are available")
13. public void a\_list\_of\_students\_are\_available() {
14. *// Write code here that turns the phrase above into concrete actions*
15. *//throw new io.cucumber.java.PendingException();*
16. response=RestAssured.get("http://10.82.180.36:8080/rest-session-demo/api/student?rollNo=101");
17. json\_res=response.jsonPath();
18. System.out.println(json\_res);
20. }
21. @When("Get the student details")
22. public void get\_the\_student\_details() {
23. *// Write code here that turns the phrase above into concrete actions*
24. *//throw new io.cucumber.java.PendingException();*
25. System.out.println("Name of the student: " + json\_res.get("name"));
26. System.out.println("Standard of the student: " + json\_res.get("std"));
27. Assert.assertEquals(json\_res.get("name"),"Harvey","Incorrect student name");
28. }
29. @Then("Validate status code and status line")
30. public void validate\_status\_code\_and\_status\_line() {
31. *// Write code here that turns the phrase above into concrete actions*
32. *//throw new io.cucumber.java.PendingException();*
33. System.out.println(response.getStatusCode());
34. System.out.println(response.getStatusLine());
35. }
36. }

### **Step8:**Run the feature file using Right click-→Run As-→cucumber feature

**Step9:**You will get the below output in console window

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

To achieve Serialization, a class needs to implement Serializable Interface and such class are actually Java Beans or say POJO (Plain Old Java Object). So, basically Serialization is the process of Converting a POJO to a JSON object

## Serialization of POJO into a JSON Request Body Object

Let’s take an example, Here we are using a ‘Student’ class as a POJO which is holding some basic attributes like name, rollNo, and std number.

**Step1:** Create a maven project, add Rest Assured & GSON dependencies to POM.xml file.  GSON is required to perform Serialization & Deserialization.

1. <dependencies>
2. *<!-- https://mvnrepository.com/artifact/io.rest-assured/rest-assured -->*
3. <dependency>
4. <groupId>io.rest-assured</groupId>
5. <artifactId>rest-assured</artifactId>
6. <version>5.1.1</version>
7. <scope>test</scope>
8. </dependency>
9. *<!-- https://mvnrepository.com/artifact/org.testng/testng -->*
10. <dependency>
11. <groupId>org.testng</groupId>
12. <artifactId>testng</artifactId>
13. <version>6.0</version>
14. <scope>test</scope>
15. </dependency>
17. <dependency>
18. <groupId>com.google.code.gson</groupId>
19. <artifactId>gson</artifactId>
20. <version>2.10.1</version>
21. </dependency>
22. </dependencies>

**Step2:**Create a normal Java class which is a POJO class with below details.

1. public class Student {
3. String name;
4. String rollNo;
5. String std;
6. public Student(String name,String rollNo,String std)
7. {
8. this.name=name;
9. this.rollNo=rollNo;
10. this.std=std;
12. }
13. public String getName() {
14. return name;
15. }
16. public void setName(String name) {
17. this.name = name;
18. }
19. public String getRollNo() {
20. return rollNo;
21. }
22. public void setRollNo(String rollNo) {
23. this.rollNo = rollNo;
24. }
25. public String getStd() {
26. return std;
27. }
28. public void setStd(String std) {
29. this.std = std;
30. }
31. }

 Create a Rest Assured Test to perform the serialization bypassing this POJO to the API. Let us use the API which is having a POST Endpoint on which we are making a call request with the above-mentioned POJO (in the form of the object instance) in the Request body object.

**Step 3:**Create the TestNg Class StudentPojo with below

1. import org.testng.annotations.Test;
2. import io.restassured.RestAssured;
3. import io.restassured.response.Response;
4. import io.restassured.specification.RequestSpecification;
5. public class StudentPojo {
6. @Test
7. public void f() {
8. RestAssured.baseURI = "http://10.82.180.36:8080/rest-session-demo/api";
10. RequestSpecification request =RestAssured.given();
12. Student student=new Student("Isha","100","X");
13. request.contentType("application/json");
14. request.body(student);
15. request.post("/student");
16. }
17. }

**Step 4:**Run the TestNg Test and observe the output.

Rest-Assured will render the “Student” object instance into the JSON formatted request body object.

#### **De-Serialization of the API Response into a POJO**

In De Serialization we are doing the reverse by transforming the API response to a POJO Java instance.

Let us take the same example used in serialization and create a new TestNg Test for deserialization. We will be using the same “Student” Pojo class.

**Step 1:**Create the TestNG class to perform GET Operation

1. import org.testng.annotations.Test;
2. import io.restassured.RestAssured;
3. public class Deserialize {
4. @Test
5. public void f() {
6. Student student =RestAssured.get("http://10.82.180.36:8080/rest-session-demo/api/student?rollNo=106").as(Student.class);
7. System.out.println(student.name);
8. System.out.println(student.rollNo);
9. System.out.println(student.std);
10. }
11. }

Step2: Run the TestNG Class created, below student details will be displayed in console.