**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.

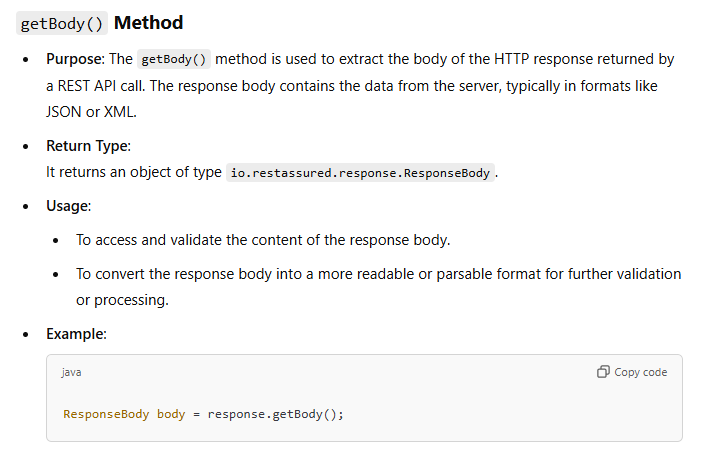
# DELETE

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

# 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.

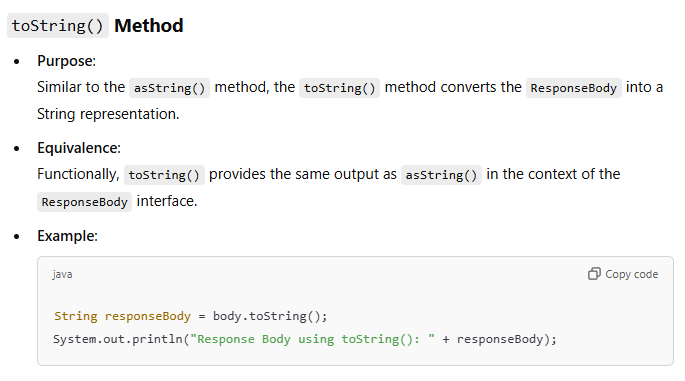
## **getBody( ) method**

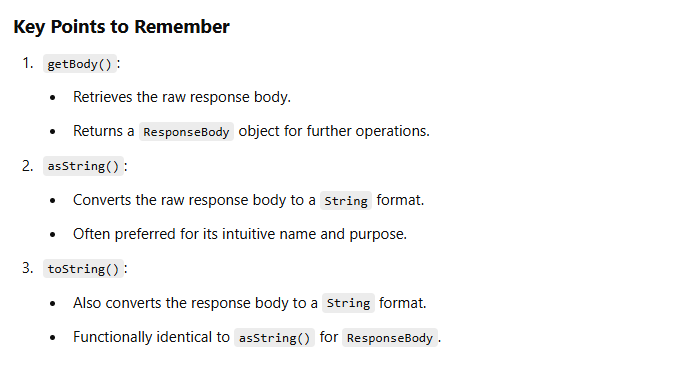


## **asString( ) method**

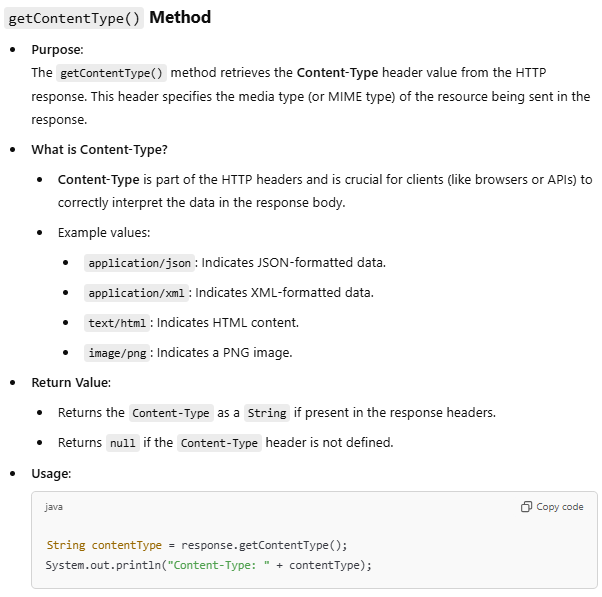


## **toString () method**

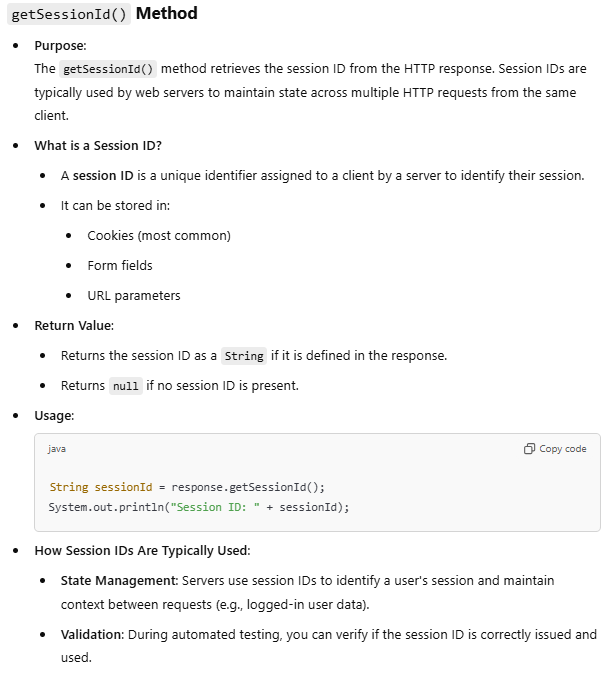




## **getContentType () method**



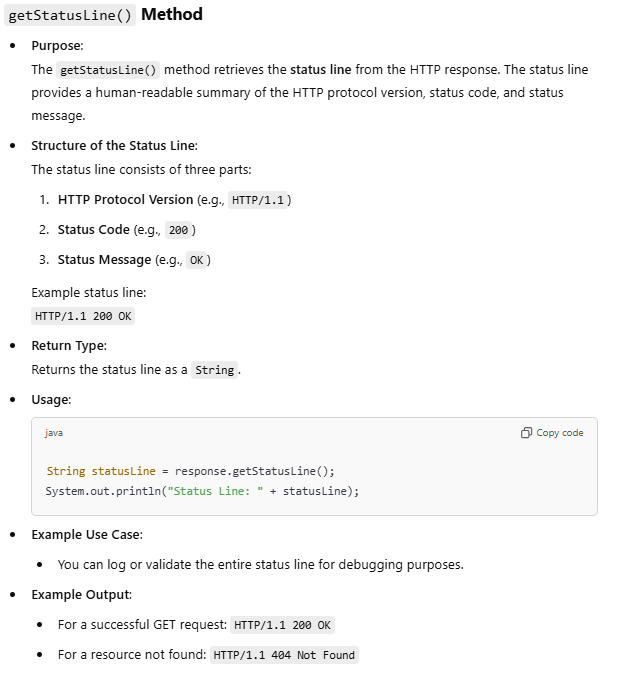
## **getSessionID () method**

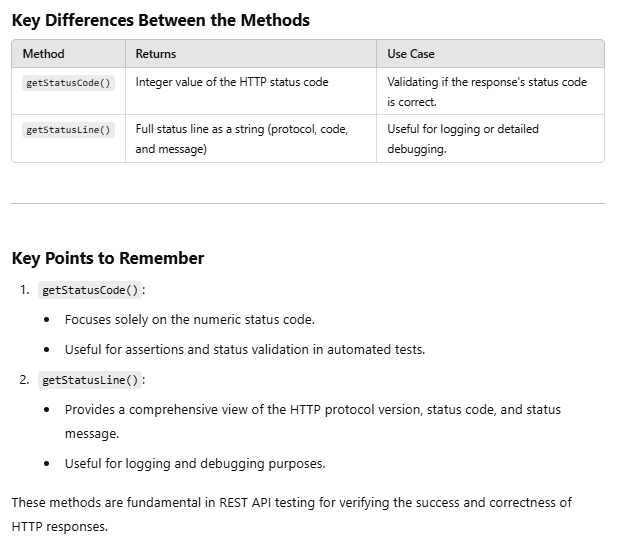


# getStatusCode( ) method:

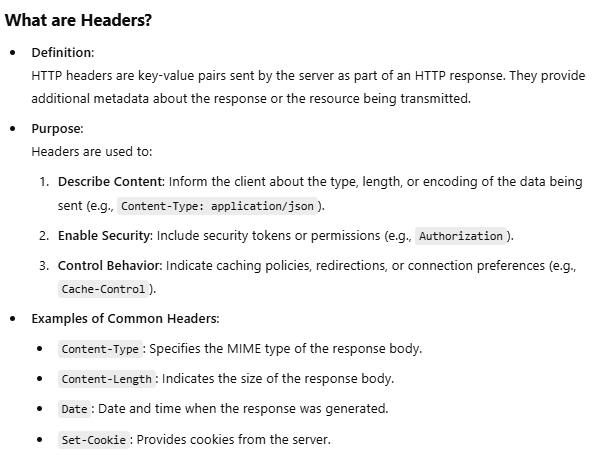


# getStatusLine( ) method:

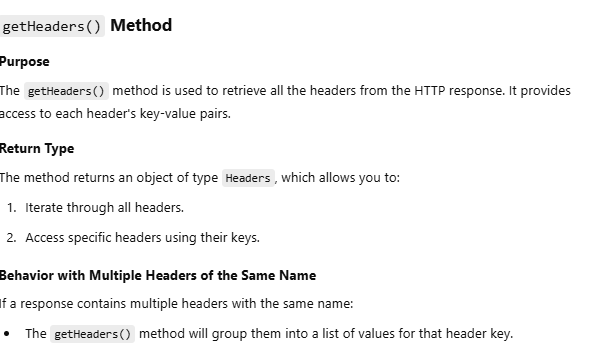


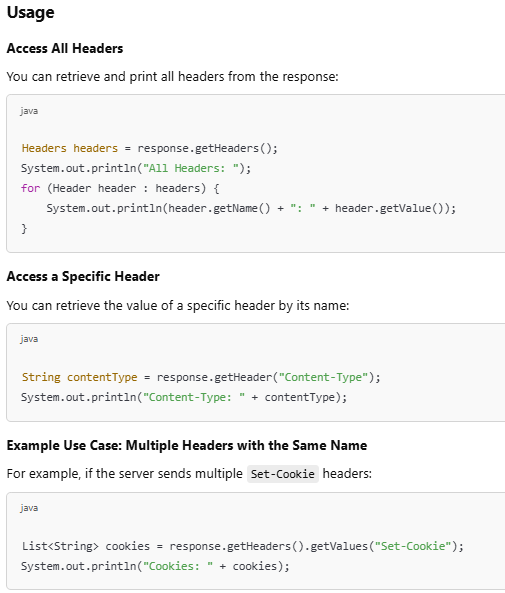


# Headers



# getHeaders



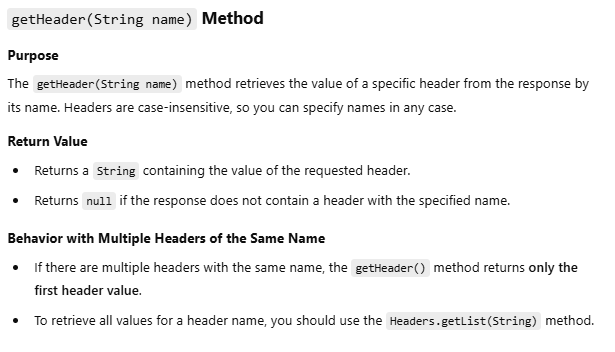


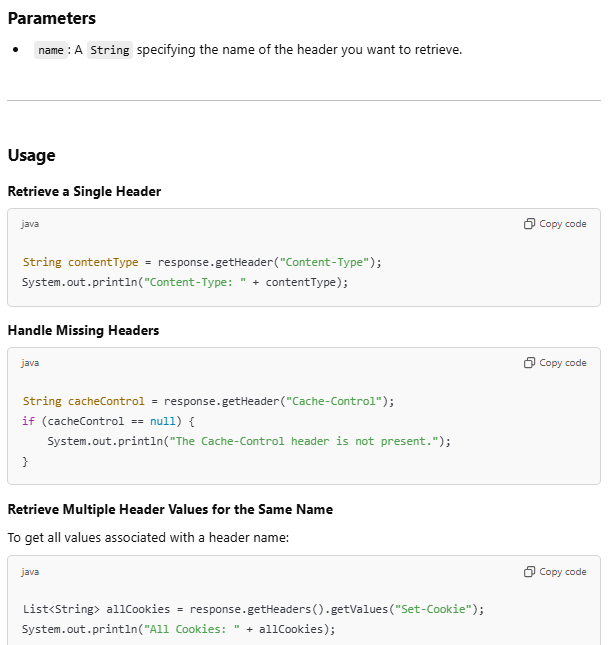
As we have seen in getHeaders() method, Each header entry is present in a key-value pair. The Header carries extra information sent by the server. This extra information is considered as Metadata of the Response.

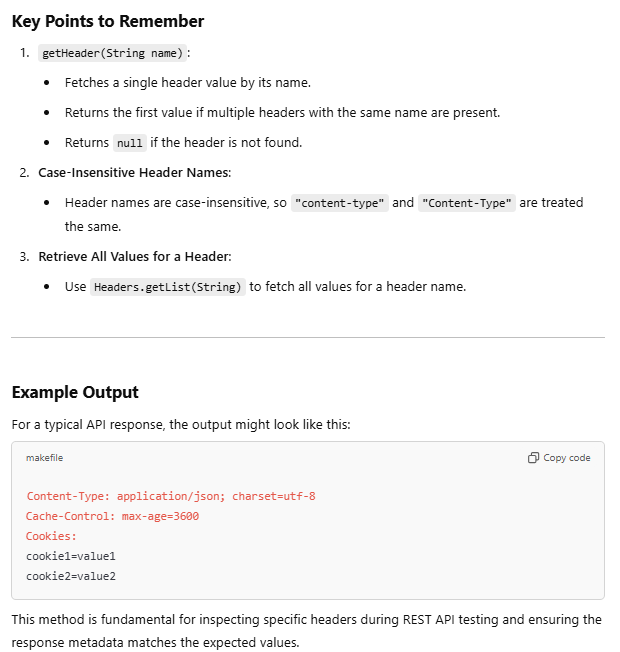
One of the headers called **Content-Type** which tells how to interpret the data present in the body of the response. The Content-Type header indicates the media type of the resource.

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.

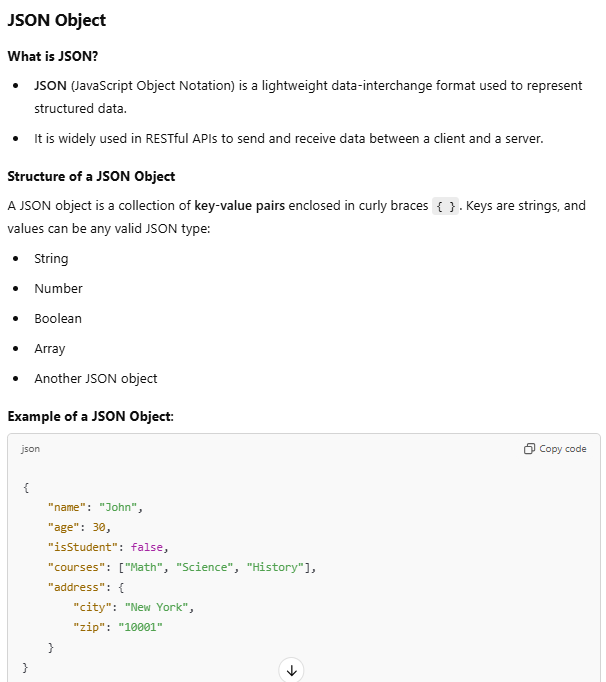
# getHeader(String name):

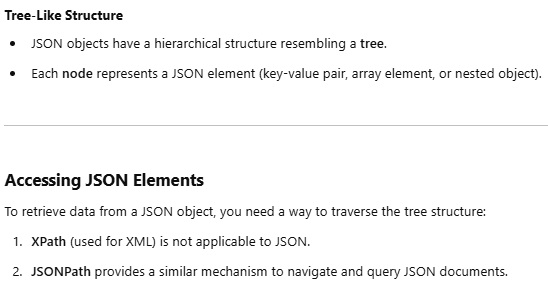




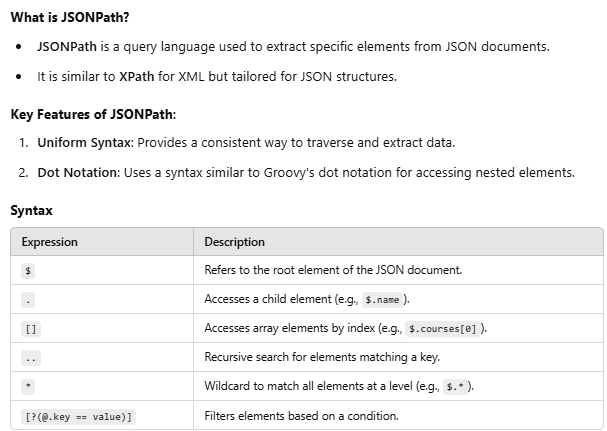


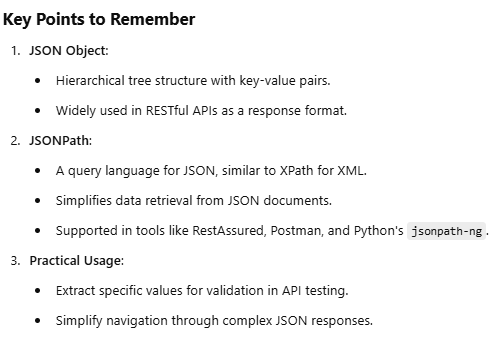
# JSON object:





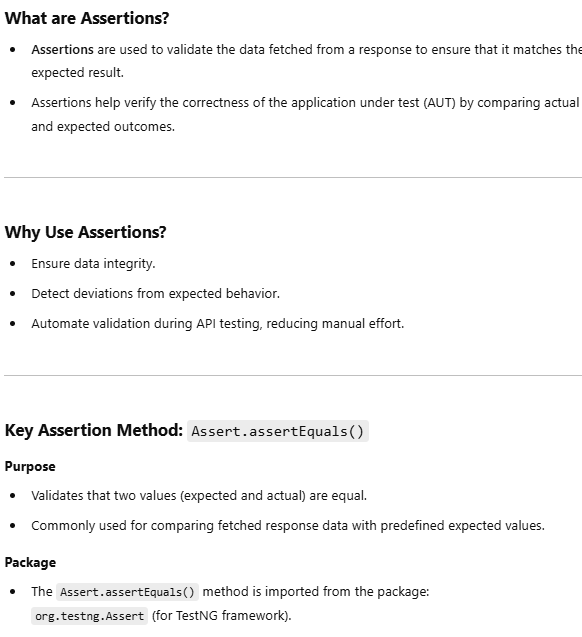
# JSONPath:

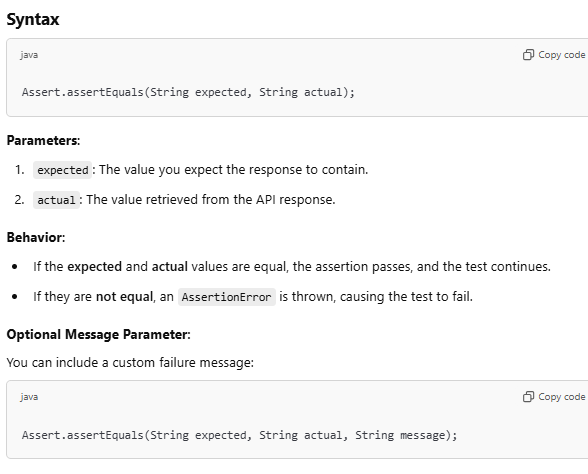






# Assertions:





**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.

# 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.

# 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.

# 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.

# 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 analysed 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, and 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**

**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.