

## 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

### getBody() Method

- **Purpose:** The `getBody()` method is used to extract the body of the HTTP response returned by a REST API call. The response body contains the data from the server, typically in formats like JSON or XML.
- **Return Type:**  
It returns an object of type `io.restassured.response.ResponseBody`.
- **Usage:**
  - To access and validate the content of the response body.
  - To convert the response body into a more readable or parsable format for further validation or processing.
- **Example:**

```
java

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

 Copy code


## asString( ) method

### asString() Method

- **Purpose:**  
Converts the `ResponseBody` object into a `String` representation for easier handling and validation.
- **Use Case:**  
Often used when we need to log, assert, or process the raw response body as plain text.
- **How It Differs:**  
The `asString()` method is specifically tailored for converting the `ResponseBody` into its String representation.
- **Example:**

```
java

String responseBody = body.asString();
System.out.println("Response Body as String: " + responseBody);
```


 Copy code

## toString () method

### toString() Method

- **Purpose:**  
Similar to the `asString()` method, the `toString()` method converts the `ResponseBody` into a String representation.
- **Equivalence:**  
Functionally, `toString()` provides the same output as `asString()` in the context of the `ResponseBody` interface.
- **Example:**

java

 Copy code

```
String responseBody = body.toString();  
System.out.println("Response Body using toString(): " + responseBody);
```

### Key Points to Remember

1. `getBody()` :
  - Retrieves the raw response body.
  - Returns a `ResponseBody` object for further operations.
2. `asString()` :
  - Converts the raw response body to a `String` format.
  - Often preferred for its intuitive name and purpose.
3. `toString()` :
  - Also converts the response body to a `String` format.
  - Functionally identical to `asString()` for `ResponseBody`.

## getContentType () method

### getContentType() Method

- **Purpose:**

The `getContentType()` method retrieves the **Content-Type** header value from the HTTP response. This header specifies the media type (or MIME type) of the resource being sent in the response.

- **What is Content-Type?**


- **Content-Type** is part of the HTTP headers and is crucial for clients (like browsers or APIs) to correctly interpret the data in the response body.
- Example values:
  - `application/json` : Indicates JSON-formatted data.
  - `application/xml` : Indicates XML-formatted data.
  - `text/html` : Indicates HTML content.
  - `image/png` : Indicates a PNG image.

- **Return Value:**

- Returns the `Content-Type` as a `String` if present in the response headers.
- Returns `null` if the `Content-Type` header is not defined.

- **Usage:**

java

 Copy code

```
String contentType = response.getContentType();  
System.out.println("Content-Type: " + contentType);
```

## getSessionID () method

### getSessionId() Method

- **Purpose:**

The `getSessionId()` method retrieves the session ID from the HTTP response. Session IDs are typically used by web servers to maintain state across multiple HTTP requests from the same client.

- **What is a Session ID?**


- A session ID is a unique identifier assigned to a client by a server to identify their session.
- It can be stored in:
  - Cookies (most common)
  - Form fields
  - URL parameters

- **Return Value:**

- Returns the session ID as a `String` if it is defined in the response.
- Returns `null` if no session ID is present.

- **Usage:**

java

 Copy code

```
String sessionId = response.getSessionId();  
System.out.println("Session ID: " + sessionId);
```

- **How Session IDs Are Typically Used:**

- **State Management:** Servers use session IDs to identify a user's session and maintain context between requests (e.g., logged-in user data).
- **Validation:** During automated testing, you can verify if the session ID is correctly issued and used.

## getStatusCode() method:

### getStatusCode() Method

- **Purpose:**

The `getStatusCode()` method retrieves the **HTTP status code** from the server's response. The status code indicates the result of the HTTP request and provides information about the response's success or failure.

- **What is a Status Code?**

A status code is a 3-digit number returned in the HTTP response to indicate the outcome of the request. Examples include:


- `200` : Success (OK)
- `404` : Not Found
- `500` : Internal Server Error

- **Return Type:**

Returns the status code as an `int`.

- **Usage:**

java


 Copy code

```
int statusCode = response.getStatusCode();  
System.out.println("Status Code: " + statusCode);
```

- **Example Use Case:**

- You can assert the status code to verify if the API call was successful:

java

 Copy code

```
assert statusCode == 200 : "Expected status code 200, but got " + statusCode;
```

- **Example Output:**

- For a successful GET request: `200`
- If the resource is not found: `404`

## getStatusLine() method:

### getStatusLine() Method

- **Purpose:**

The `getStatusLine()` method retrieves the **status line** from the HTTP response. The status line provides a human-readable summary of the HTTP protocol version, status code, and status message.

- **Structure of the Status Line:**

The status line consists of three parts:

1. **HTTP Protocol Version** (e.g., `HTTP/1.1`)
2. **Status Code** (e.g., `200`)
3. **Status Message** (e.g., `OK`)

Example status line:

```
HTTP/1.1 200 OK
```

- **Return Type:**

Returns the status line as a `String`.

- **Usage:**

```
java
```

[Copy code](#)

```
String statusLine = response.getStatusLine();  
System.out.println("Status Line: " + statusLine);
```

- **Example Use Case:**

- You can log or validate the entire status line for debugging purposes.

- **Example Output:**

- For a successful GET request: `HTTP/1.1 200 OK`
- For a resource not found: `HTTP/1.1 404 Not Found`



## Key Differences Between the Methods

Method	Returns	Use Case
<code>getStatusCode()</code>	Integer value of the HTTP status code	Validating if the response's status code is correct.
<code>getStatusLine()</code>	Full status line as a string (protocol, code, and message)	Useful for logging or detailed debugging.

## Key Points to Remember

- `getStatusCode()` :
  - Focuses solely on the numeric status code.
  - Useful for assertions and status validation in automated tests.
- `getStatusLine()` :
  - Provides a comprehensive view of the HTTP protocol version, status code, and status message.
  - Useful for logging and debugging purposes.

These methods are fundamental in REST API testing for verifying the success and correctness of HTTP responses.

## Headers

### What are Headers?

- **Definition:**  
HTTP headers are key-value pairs sent by the server as part of an HTTP response. They provide additional metadata about the response or the resource being transmitted.
- **Purpose:**  
Headers are used to:
  1. **Describe Content:** Inform the client about the type, length, or encoding of the data being sent (e.g., `Content-Type: application/json`).
  2. **Enable Security:** Include security tokens or permissions (e.g., `Authorization`).
  3. **Control Behavior:** Indicate caching policies, redirections, or connection preferences (e.g., `Cache-Control`).
- **Examples of Common Headers:**
  - `Content-Type` : Specifies the MIME type of the response body.
  - `Content-Length` : Indicates the size of the response body.
  - `Date` : Date and time when the response was generated.
  - `Set-Cookie` : Provides cookies from the server.

## getHeaders

### `getHeaders()` Method

#### Purpose

The `getHeaders()` method is used to retrieve all the headers from the HTTP response. It provides access to each header's key-value pairs.

#### Return Type

The method returns an object of type `Headers`, which allows you to:

1. Iterate through all headers.
2. Access specific headers using their keys.

#### Behavior with Multiple Headers of the Same Name

If a response contains multiple headers with the same name:

- The `getHeaders()` method will group them into a list of values for that header key.

## Usage

### Access All Headers

You can retrieve and print all headers from the response:

```
java

Headers headers = response.getHeaders();
System.out.println("All Headers: ");
for (Header header : headers) {
    System.out.println(header.getName() + ": " + header.getValue());
}
```

### Access a Specific Header

You can retrieve the value of a specific header by its name:

```
java

String contentType = response.getHeader("Content-Type");
System.out.println("Content-Type: " + contentType);
```

### Example Use Case: Multiple Headers with the Same Name

For example, if the server sends multiple `Set-Cookie` headers:

```
java

List<String> cookies = response.getHeaders().getValues("Set-Cookie");
System.out.println("Cookies: " + cookies);
```

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(String name) Method

#### Purpose

The `getHeader(String name)` method retrieves the value of a specific header from the response by its name. Headers are case-insensitive, so you can specify names in any case.

#### Return Value

- Returns a `String` containing the value of the requested header.
- Returns `null` if the response does not contain a header with the specified name.

#### Behavior with Multiple Headers of the Same Name

- If there are multiple headers with the same name, the `getHeader()` method returns only the first header value.
- To retrieve all values for a header name, you should use the `Headers.getList(String)` method.

#### Parameters

- `name`: A `String` specifying the name of the header you want to retrieve.
- 

## Usage

### Retrieve a Single Header

```
java Copy code

String contentType = response.getHeader("Content-Type");
System.out.println("Content-Type: " + contentType);
```

### Handle Missing Headers

```
java Copy code

String cacheControl = response.getHeader("Cache-Control");
if (cacheControl == null) {
    System.out.println("The Cache-Control header is not present.");
}
```

### Retrieve Multiple Header Values for the Same Name

To get all values associated with a header name:

```
java Copy code

List<String> allCookies = response.getHeaders().getValues("Set-Cookie");
System.out.println("All Cookies: " + allCookies);
```


## Key Points to Remember

1. `getHeader(String name)`:
    - Fetches a single header value by its name.
    - Returns the first value if multiple headers with the same name are present.
    - Returns `null` if the header is not found.
  2. Case-Insensitive Header Names:
    - Header names are case-insensitive, so `"content-type"` and `"Content-Type"` are treated the same.
  3. Retrieve All Values for a Header:
    - Use `Headers.getList(String)` to fetch all values for a header name.
- 

## Example Output

For a typical API response, the output might look like this:

makefile

 Copy code

```
Content-Type: application/json; charset=utf-8
Cache-Control: max-age=3600
Cookies:
cookie1=value1
cookie2=value2
```

This method is fundamental for inspecting specific headers during REST API testing and ensuring the response metadata matches the expected values.

## JSON object:

### JSON Object

#### What is JSON?

- JSON (JavaScript Object Notation) is a lightweight data-interchange format used to represent structured data.
- It is widely used in RESTful APIs to send and receive data between a client and a server.

#### Structure of a JSON Object

A JSON object is a collection of **key-value pairs** enclosed in curly braces `{ }`. Keys are strings, and values can be any valid JSON type:

- String
- Number
- Boolean
- Array
- Another JSON object

#### Example of a JSON Object:

```
json Copy code

{
  "name": "John",
  "age": 30,
  "isStudent": false,
  "courses": ["Math", "Science", "History"],
  "address": {
    "city": "New York",
    "zip": "10001"
  }
}
```

#### Tree-Like Structure

- JSON objects have a hierarchical structure resembling a **tree**.
- Each **node** represents a JSON element (key-value pair, array element, or nested object).

---

### Accessing JSON Elements

To retrieve data from a JSON object, you need a way to traverse the tree structure:

1. **XPath** (used for XML) is not applicable to JSON.
2. **JSONPath** provides a similar mechanism to navigate and query JSON documents.

JSONPath:

What is JSONPath?

- JSONPath is a query language used to extract specific elements from JSON documents.
- It is similar to XPath for XML but tailored for JSON structures.

Key Features of JSONPath:

1. Uniform Syntax: Provides a consistent way to traverse and extract data.
2. Dot Notation: Uses a syntax similar to Groovy's dot notation for accessing nested elements.

Syntax

Expression	Description
\$	Refers to the root element of the JSON document.
.	Accesses a child element (e.g., \$.name ).
[]	Accesses array elements by index (e.g., \$.courses[0] ).
..	Recursive search for elements matching a key.
*	Wildcard to match all elements at a level (e.g., \$.* ).
[?(@.key == value)]	Filters elements based on a condition.

Key Points to Remember

1. JSON Object:
  - Hierarchical tree structure with key-value pairs.
  - Widely used in RESTful APIs as a response format.
2. JSONPath:
  - A query language for JSON, similar to XPath for XML.
  - Simplifies data retrieval from JSON documents.
  - Supported in tools like RestAssured, Postman, and Python's jsonpath-ng .
3. Practical Usage:
  - Extract specific values for validation in API testing.
  - Simplify navigation through complex JSON responses.


## Usage in RESTful Services

- **Why JSONPath?**

JSONPath simplifies the process of querying and extracting specific data from JSON responses in REST APIs.

- **Practical Example:** Consider a REST API returning the following JSON response:

json

 Copy code

```
{
  "employees": [
    {"name": "John", "age": 30},
    {"name": "Jane", "age": 25},
    {"name": "Mike", "age": 35}
  ]
}
```

- Retrieve all employee names:

JSONPath: `$.employees[*].name`

Result: `["John", "Jane", "Mike"]`

---

## Advantages of JSONPath

1. **Simplified Traversal:** Easily navigate through complex JSON structures.
2. **Dynamic Queries:** Filter or select elements based on conditions.
3. **Compatibility:** Supported by many tools and libraries for API testing (e.g., RestAssured, Postman).



## Assertions:

### What are Assertions?

- Assertions are used to validate the data fetched from a response to ensure that it matches the expected result.
  - Assertions help verify the correctness of the application under test (AUT) by comparing actual and expected outcomes.
- 

### Why Use Assertions?

- Ensure data integrity.
  - Detect deviations from expected behavior.
  - Automate validation during API testing, reducing manual effort.
- 

### Key Assertion Method: `Assert.assertEquals()`

#### Purpose


- Validates that two values (expected and actual) are equal.
- Commonly used for comparing fetched response data with predefined expected values.

#### Package

- The `Assert.assertEquals()` method is imported from the package:  
`org.testng.Assert` (for TestNG framework).

## Syntax

java

 Copy code

```
Assert.assertEquals(String expected, String actual);
```

### Parameters:

1. `expected`: The value you expect the response to contain.
2. `actual`: The value retrieved from the API response.


### Behavior:

- If the `expected` and `actual` values are equal, the assertion passes, and the test continues.
- If they are not equal, an `AssertionError` is thrown, causing the test to fail.

### Optional Message Parameter:

You can include a custom failure message:

java

 Copy code

```
Assert.assertEquals(String expected, String actual, String message);
```

**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.xml**. **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, VJUnit, 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>
41.

```

**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.
3.   Scenario: Get Student details
4.     Given A list of students are available
5.     When Get the student details
6.     Then Validate status code and status line
7.

```

**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 samplerestassuredbdd;
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.
10.    public class Student {
11.        Response response;
12.        JsonPath json_res;
13.        @Given("A list of students are available")
14.        public void a_list_of_students_are_available() {
15.            // Write code here that turns the phrase above into concrete actions
16.            //throw new io.cucumber.java.PendingException();
17.            response=RestAssured.get("http://10.82.180.36:8080/rest-session-
demo/api/student?rollNo=101");
18.            json_res=response.jsonPath();
19.            System.out.println(json_res);
20.
21.        }
22.
23.        @When("Get the student details")
24.        public void get_the_student_details() {
25.            // Write code here that turns the phrase above into concrete actions
26.            //throw new io.cucumber.java.PendingException();
27.            System.out.println("Name of the student: " + json_res.get("name"));
28.            System.out.println("Standard of the student: " + json_res.get("std"));
29.            Assert.assertEquals(json_res.get("name"),"Harvey","Incorrect student
name");
30.        }
31.
32.        @Then("Validate status code and status line")
33.        public void validate_status_code_and_status_line() {
34.            // Write code here that turns the phrase above into concrete actions
35.            //throw new io.cucumber.java.PendingException();
36.            System.out.println(response.getStatusCode());
37.            System.out.println(response.getStatusLine());
38.        }
39.    }
```

**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.
10. <!-- https://mvnrepository.com/artifact/org.testng/testng -->
11. <dependency>
12.   <groupId>org.testng</groupId>
13.   <artifactId>testng</artifactId>
14.   <version>6.0</version>
15.   <scope>test</scope>
16. </dependency>
17.
18. <dependency>
19.   <groupId>com.google.code.gson</groupId>
20.   <artifactId>gson</artifactId>
21.   <version>2.10.1</version>
22. </dependency>
23. </dependencies>
```

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

```
1.
2. public class Student {
3.
4.   String name;
5.   String rollNo;
6.   String std;
7.   public Student(String name,String rollNo,String std)
```

```

8.    {
9.        this.name=name;
10.       this.rollNo=rollNo;
11.       this.std=std;
12.
13.    }
14.    public String getName() {
15.        return name;
16.    }
17.    public void setName(String name) {
18.        this.name = name;
19.    }
20.    public String getRollNo() {
21.        return rollNo;
22.    }
23.    public void setRollNo(String rollNo) {
24.        this.rollNo = rollNo;
25.    }
26.    public String getStd() {
27.        return std;
28.    }
29.    public void setStd(String std) {
30.        this.std = std;
31.    }
32. }

```

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.
6.  public class StudentPojo {
7.      @Test
8.      public void f() {
9.          RestAssured.baseURI = "http://10.82.180.36:8080/rest-session-demo/api";
10.
11.         RequestSpecification request =RestAssured.given();
12.
13.         Student student=new Student("Isha","100","X");
14.         request.contentType("application/json");
15.         request.body(student);

```

```
16.     request.post("/student");
17. }
18. }

19.
```

**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.
4. public class Deserialize {
5.     @Test
6.     public void f() {
7.         Student student =RestAssured.get("http://10.82.180.36:8080/rest-session-
demo/api/student?rollNo=106").as(Student.class);
8.         System.out.println(student.name);
9.         System.out.println(student.rollNo);
10.        System.out.println(student.std);
11.    }
12. }
```

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