

TECHNOLOGY



Automation Testing

POST Request and Response Automation



A Day in the Life of an Automation Test Engineer

Thomas has decided to use REST APIs in his backend server to send data to the server.

He now wants to send various information to the server.

On the other hand, he even wants to validate the HTTP response status using REST Assured.

To achieve all these goals, he must learn how to initiate a POST request and send information to the server and the steps required to validate the HTTP response status using REST Assured.



Learning Objectives

By the end of this lesson, you will be able to:

- 🕒 Learn about the use of the POST request method in HTTP
- 🕒 Remember the process of using the POST request method
- 🕒 Analyze the need to validate response status using REST Assured
- 🕒 Learn about the use of the end-to-end testing methodology in REST Assured



Automate POST Request Using REST Assured

Overview: HTTP Post Request Method

Mostly in the real world, when users submit a registration form on a website (example: Gmail), it always requires the user to provide the necessary data and click on submit.



So, through this action of submitting data, users are actually **posting** or sending the data to the server.



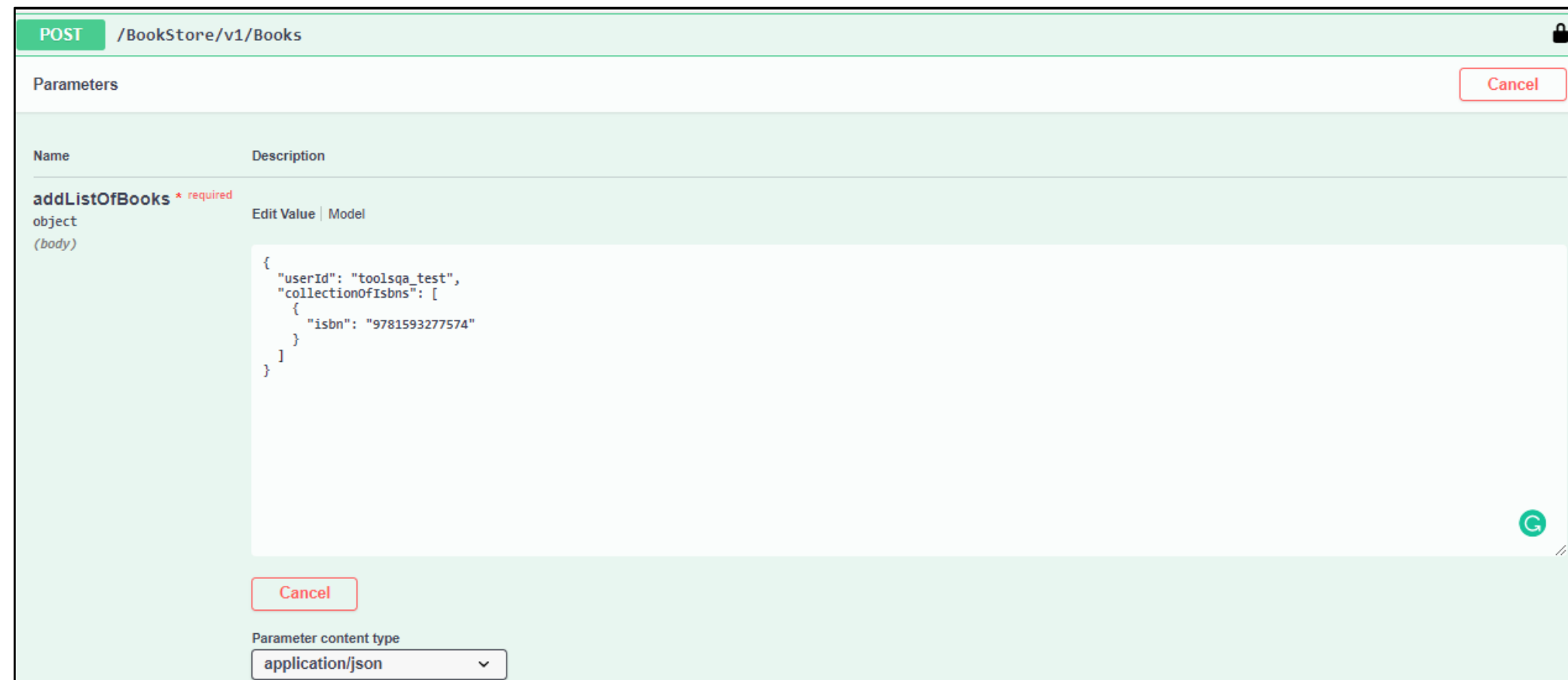
Overview: HTTP Post Request Method

POST request method in HTTP sends data to the server.



Overview: HTTP Post Request Method

For example: When a person creates a new user using a POST request, it should look like this:



The screenshot shows a REST client interface for a POST request to the endpoint `/BookStore/v1/Books`. The request body is a JSON object representing a user and a collection of books. The interface includes a 'Parameters' section, a 'Name' column with the parameter `addListOfBooks` (marked as required), and a 'Description' column showing the JSON body. A 'Parameter content type' dropdown is set to `application/json`. A green 'G' icon is visible in the bottom right corner of the request body area.

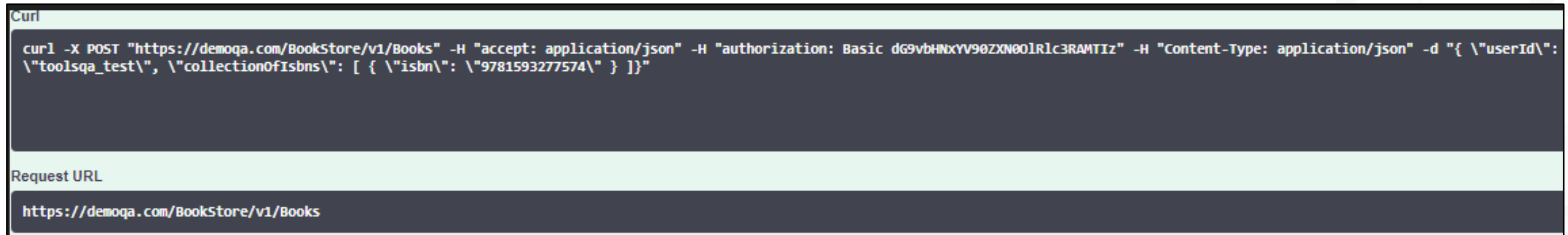
```
{
  "userId": "toolsqa_test",
  "collectionOfIsbns": [
    {
      "isbn": "9781593277574"
    }
  ]
}
```

Image source: www.toolsqa.com

When the user **executes** requests, the user initiates a POST request and sends all the information to the server.

Overview: HTTP Post Request Method

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

A screenshot of a REST client interface. The top section, labeled 'Curl', contains a curl command for a POST request to 'https://demoqa.com/BookStore/v1/Books' with headers for 'accept', 'authorization', and 'Content-Type', and a JSON body. The bottom section, labeled 'Request URL', shows the same URL.

```
Curl
curl -X POST "https://demoqa.com/BookStore/v1/Books" -H "accept: application/json" -H "authorization: Basic dG9vbHNxYV90ZXN0lRlc3RAMTiz" -H "Content-Type: application/json" -d "{ \"userId\": \"toolsqa_test\", \"collectionOfIsbns\": [ { \"isbn\": \"9781593277574\" } ] }"
```

```
Request URL
https://demoqa.com/BookStore/v1/Books
```

Image source: www.toolsqa.com

Overview: HTTP Post Request Method

In the case of the **Curl** command, it has a POST request with all the data. In this case, the operation is successful and the response code is received as 201.

Server response	
Code	Details
201	Success
	<pre>{ "isbn": "9781593277574" }</pre>

Image source: www.toolsqa.com

Overview: HTTP Post Request Method

Here, the content-type parameter is **application/JSON**.



The data sent to the server by POST request can also be XML and is present in the HTTP request body.

Overview: HTTP Post Request Method

The POST method usually creates a new record in the application database.



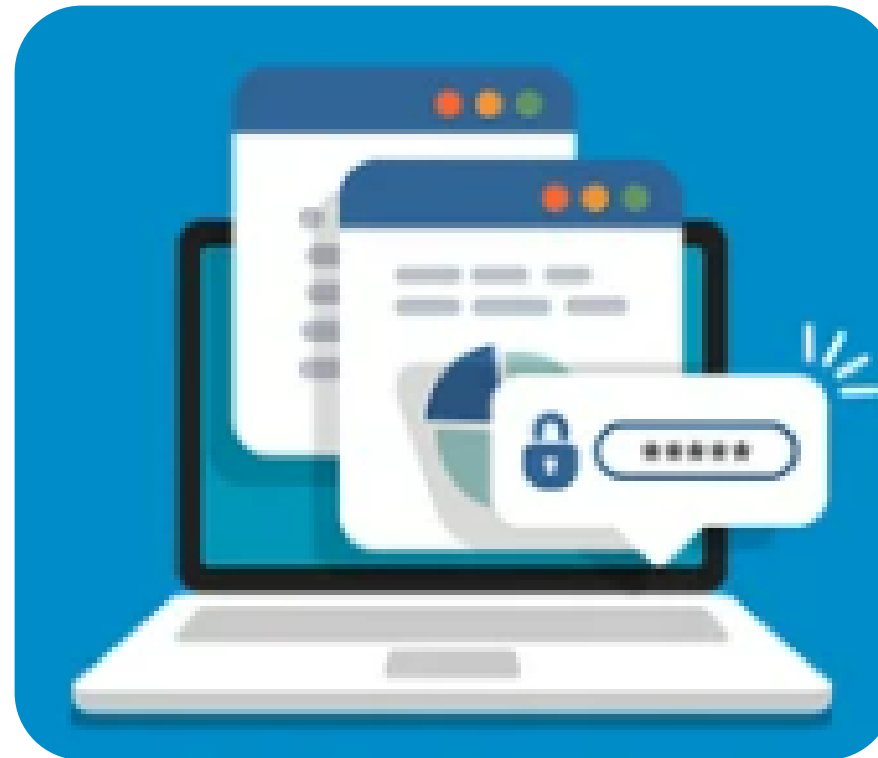
For example:

While refreshing or pressing back on payment pages it does not work directly. It occurs due to security reasons, as the data being sent is not visible in the URL string.

While creating web applications users use POST request redirections to ensure that no one hits that URL directly without proper navigation channels.

Overview: HTTP Post Request Method

The POST request can be very lengthy and a complex body structure in real-world applications.



It provides an extra layer of security; users often use it for passing sensitive business-related data.

Overview: HTTP Post Request Method

It is also important that the POST request does not always require all the data to be filled by the user.



It completely depends on the server implementation, and therefore the user should always go through the documentation before.



Process of Using POST Request Method

Firstly, the user can add a simple JSON library in the classpath so that the user can create JSON objects in the code.



JSON can be downloaded from the Maven. Once the JAR is downloaded the user can add it to the classpath.

Process of Using POST Request Method

Steps required to make a POST Request using REST Assured:



- Create a request pointing to the service Endpoint
- Create a JSON request which contains all the fields
- Add JSON body in the request and send the request
- Validate the request
- Changing the HTTP method on a POST request

Process of Using POST Request Method

Step 1: Creating a request pointing to the service endpoint

In this code, the user initializes a base URI with a link to the bookstore and the **createUser** API. Next, the user creates a **request** using RequestSpecification.

```
RestAssured.baseURI = "https://demoqa.com/BookStore/v1/Books"; RequestSpecification request =  
    RestAssured.given();
```

Process of Using POST Request Method

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

Here, the user will be creating a JSON request object that will contain the data the user needs to create a new user.

```
// JSONObject is a class that represents a Simple JSON.  
// We can add Key - Value pairs using the put method  
JSONObject requestParams = new JSONObject();  
requestParams.put("userId", "TQ123");  
requestParams.put("isbn", "9781449325862");
```

Process of Using POST Request Method

Step 3: Adding JSON body in the request and send the request

The JSON string has already been created with the required data. In this step, the user will be adding the JSON to the request body and post the request.

```
// Add a header stating the Request body is a JSON
request.header("Content-Type", "application/json"); // Add the Json to the body of the request
request.body(requestParams.toJSONString()); // Post the request and check the response
```

Process of Using POST Request Method

Step 4: Validation of Response

After posting the request, the user must validate the response received from the server as a result of a POST request. The code for validating the response is:

```
Response response = request.post("/BookStoreV1BooksPost");  
System.out.println("The status received: " + response.statusLine());
```


Process of Using POST Request Method

Step 5: Changing the HTTP method on a POST request using REST Assured

When the user changes the HTTP request method that is instead of sending a POST request sends a GET request. The code would be:

```
{
RestAssured.baseURI ="https://demoqa.com/Account/v1";
RequestSpecification request = RestAssured.given();
JSONObject requestParams = new JSONObject();
requestParams.put("userName", "test_rest");
requestParams.put("password", "Testrest@123");
request.body(requestParams.toJSONString());
Response response = request.put("/User");
ResponseBody body = response.getBody();
System.out.println(response.getStatusLine());
System.out.println(body.asString());
}
```



Process of Using POST Request Method

When the code is executed, the response would look like this:

```
<terminated> test1 [TestNG] C:\Program Files\Java\jre-10.0.2\bin\javaw.exe (Feb 1, 2022, 2:18:37 PM)
HTTP/1.1 404 Not Found
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Error</title>
</head>
<body>
<pre>Cannot PUT /Account/v1/User</pre>
</body>
</html>

PASSED: UserRegistrationSuccessful
```

Image source: www.toolsqa.com



Process of Using POST Request Method

The output indicates incorrect use of the HTTP request method.



Process of Using POST Request Method

REST Assured uses the `post()` mechanism to send the HTTP request.



Process of Using POST Request Method

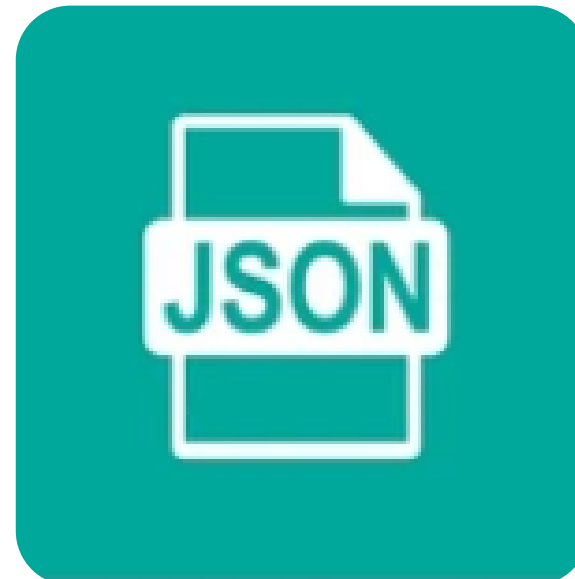
The majority of POST requests result in the creation of new records in the database.



On the other hand, POST requests can also update the existing records in the database.

Process of Using POST Request Method

The users usually send the JSON data along with the request object and then POST it to the server.



Advance Validation on Response

Validate Response Status Using REST Assured

Every HTTP response received as a result of an HTTP request sent by the client to the server has a status code.



REST- Assured

GET, POST, PUT, PATCH, DELETE

This status code value tells the user whether the HTTP response was successful or not.

Validate Response Status Using REST Assured

Web server service typically sends back HTTP packets (response packets) in response to client requests.

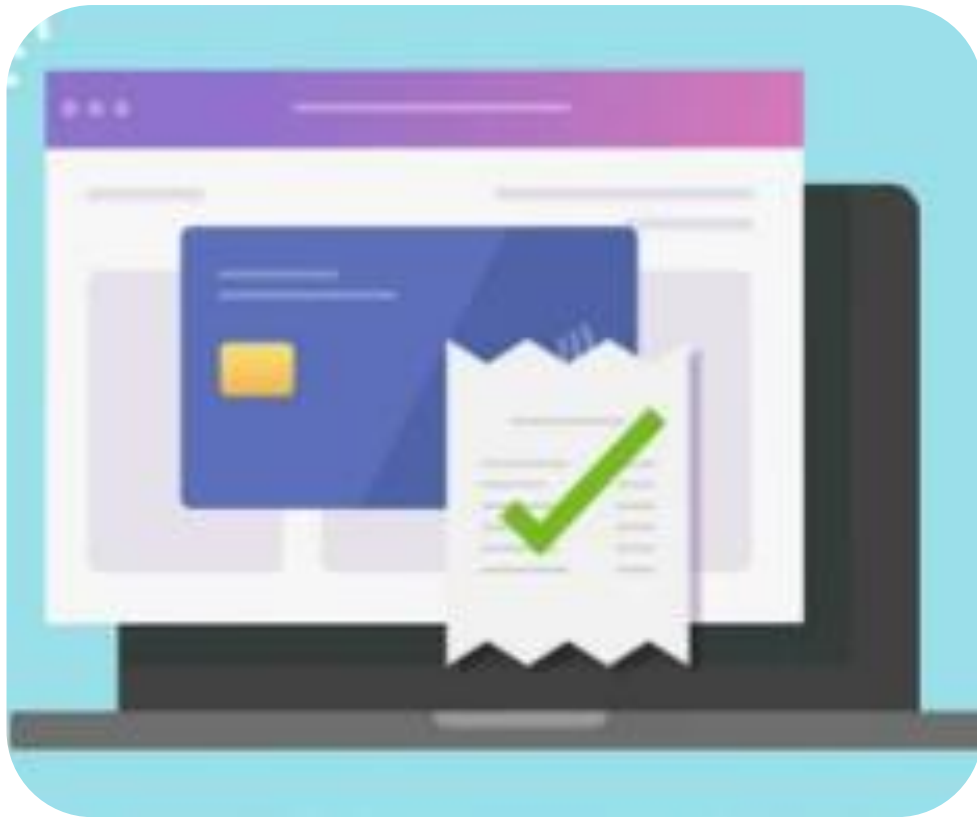
An HTTP response consists of:



- A status
- Collection of headers
- A body

Validate Response Status Using REST Assured

The validation of an HTTP response status is done in three parts as follows:



- Validate HTTP response status code
- Validate the error status code
- Validate response status line

Validate Response Status Using REST Assured

The REST API returns a response message in XML or JSON format.



This format depends on the **Media-Type** attribute in the HTTP request.



Validate Response Status Using REST Assured

Response header contains a **content-type** attribute that explains the type of response.

Suppose a user sends a GET request to the BookStore through a browser it would look like this:

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

Validate Response Status Using REST Assured

When the above command is executed, the response received looks like this:

Server response

Code	Details
200	<div><div>Response body</div><pre>{ "books": [{ "isbn": "9781449325862", "title": "Git Pocket Guide", "subTitle": "A Working Introduction", "author": "Richard E. Silverman", "publish_date": "2020-06-04T08:48:39.000Z", "publisher": "O'Reilly Media", "pages": 234, "description": "This pocket guide is the perfect on-the-job companion to Git, the distributed version control system. It provides a compact, readable introduction to Git for new users, as well as a reference to common commands and procedures for those of you with Git exp", "website": "http://chimera.labs.oreilly.com/books/1230000000561/index.html" }, { "isbn": "9781449331818", "title": "Learning JavaScript Design Patterns", "subTitle": "A JavaScript and jQuery Developer's Guide", "author": "Addy Osmani", "publish_date": "2020-06-04T09:11:40.000Z", "publisher": "O'Reilly Media", "pages": 254, "description": "With Learning JavaScript Design Patterns, you'll learn how to write beautiful, structured, and maintainable JavaScript by applying classical and modern design patterns to the language. If you want to keep your code efficient, more manageable, and up-to-da", "website": "http://www.addyosmani.com/resources/essentialjsdesignpatterns/book/" }] }</pre><div>Download</div></div> <div><div>Response headers</div><pre>content-length: 4514 content-type: application/json; charset=utf-8 date: Sun06 Feb 2022 13:43:21 GMT etag: W/"11a2-8zfX++QmcgaCjSU6F8JJP9fud1tY" server: nginx/1.17.10 (Ubuntu) x-powered-by: Express</pre></div>

Image source: www.toolsqa.com

Validate Response Status Using REST Assured

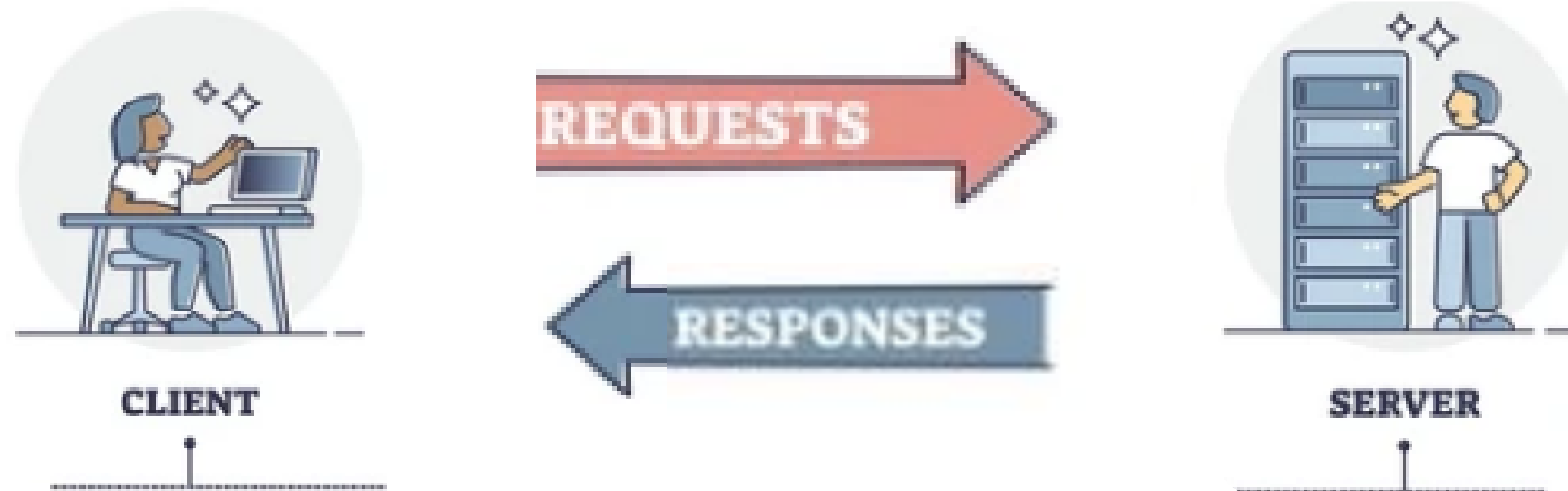
According to the screenshot, the response has a status, headers, and a body.



If the user checks the **Response headers** section, it has a content-type attribute that has the value along with other attributes. On validating the header, the user will get to know the type of response that can be expected.

Validating HTTP Response Status Code

When the user requests particular information from the server, the server sends a response with a status code back to the user.



The status code that the server returns tells whether the request was successful or not.

Validating HTTP Response Status Code

If the request was successful, the server sends the status code in the range of 200-299.



If the request was not successful, then the status code other than the range is returned.

Validating HTTP Response Status Code

REST Assured library provides a package named **io.restassured.response**, which has a response interface.



The response interface provides methods that can help to get parts of the received response.



Validating HTTP Response Status Code

Some of the important methods of the **response** interface are as below:

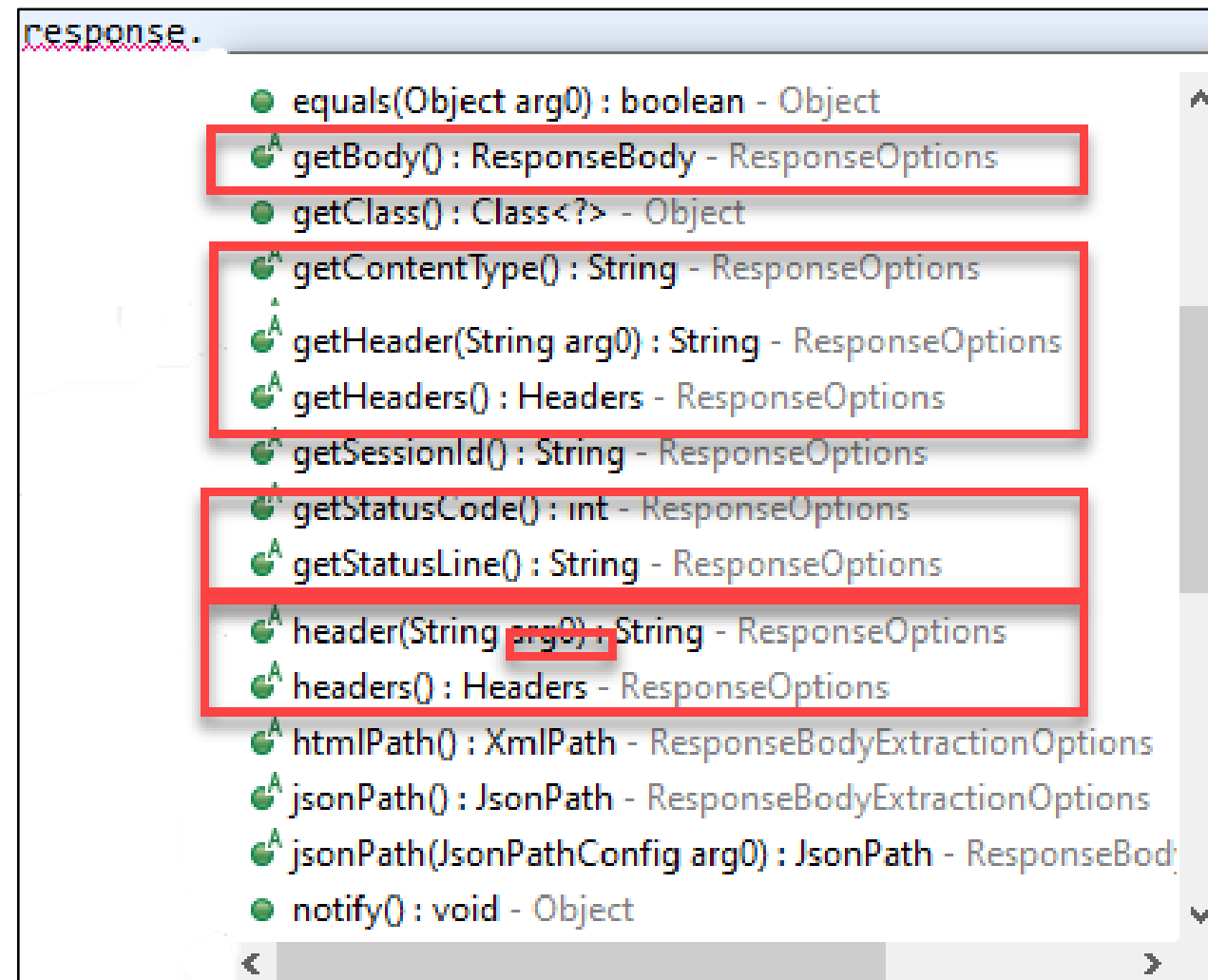


Image source: www.toolsqa.com



Validating HTTP Response Status Code

The method **getStatusCode()** in the image is used to get the status code of the response.



This method returns an integer and then the user can verify its value.



Validating HTTP Response Status Code

TestNG Assert is used to verify the status code.

The code is as below:

```
import static org.junit.Assert.*;
import org.testng.Assert;    //used to validate response status
import org.testng.annotations.Test;
import io.restassured.RestAssured;
import io.restassured.response.Response;
import io.restassured.specification.RequestSpecification;

public class RestAssuredTestResponse {
    @Test
    public void GetBookDetails()
```

Validating HTTP Response Status Code

```
{  
    // Specify the base URL to the RESTful  
web service  
    RestAssured.baseURI =  
"https://demoqa.com/BookStore/v1/Books";  
    // Get the RequestSpecification of the  
request to be sent to the server  
    RequestSpecification httpRequest =  
RestAssured.given();  
  
    Response response =  
httpRequest.get("");  
  
    // Get the status code of the request.  
    //If request is successful, status code  
will be 200
```

```
int statusCode = response.getStatusCode();  
  
    // Assert that correct status code is  
returned.  
    Assert.assertEquals(statusCode  
/*actual value*/, 200 /*expected value*/,  
        "Correct status code returned");  
}  
}
```


Validating HTTP Response Status Code

The below code extracts the status code from the message:

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



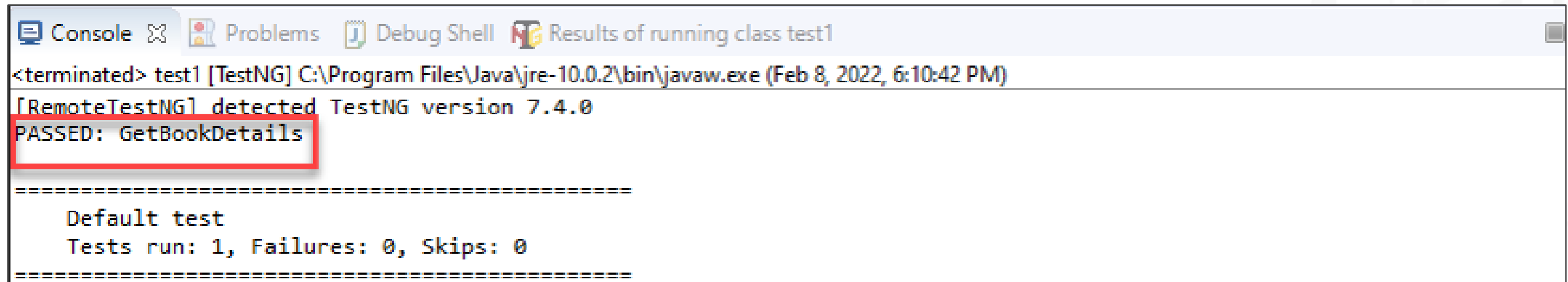
Validating HTTP Response Status Code

The return value **statusCode** is compared with the expected value which is 200. If both values are equal, then an appropriate message is returned.

```
// Assert that correct status code is returned.  
Assert.assertEquals(statusCode /*actual value*/, 200 /*expected value*/, "Correct status code  
returned");
```

Validating HTTP Response Status Code

If the user runs the above test, the user will see that the test passes since the web service returns the status code as 200, as shown in the below image:

A screenshot of an IDE's console window. The title bar shows tabs for 'Console', 'Problems', 'Debug Shell', and 'Results of running class test1'. The console output shows the test execution details, including the path to the Java executable and the test result 'PASSED: GetBookDetails', which is highlighted with a red rectangle. Below this, a summary line indicates 'Tests run: 1, Failures: 0, Skips: 0'.

```
<terminated> test1 [TestNG] C:\Program Files\Java\jre-10.0.2\bin\javaw.exe (Feb 8, 2022, 6:10:42 PM)
[RemoteTestNG] detected TestNG version 7.4.0
PASSED: GetBookDetails

=====
Default test
Tests run: 1, Failures: 0, Skips: 0
=====
```

Image source: www.toolsqa.com

Validating HTTP Response Status Code

In this process, the user can verify the status code of the response using the **getStatusCode()** method of the response interface.



In the example, the user already knew that the success code is 200 and wrote the appropriate code, but in the real case server may respond as success with a code anywhere between 200 and 299.

Validating the HTTP Error Status Code

Sometimes situations can be different, the user may receive status codes other than 200.

There can be several reasons for this, few reasons are listed below:

- Server hosting REST API is down
- Incorrect client requests
- Resource requested does not exist
- Error occurs on the server side during the processing of the request



Validating the HTTP Error Status Code

So, when any of such scenarios occur, the REST API will return a status code other than 200.

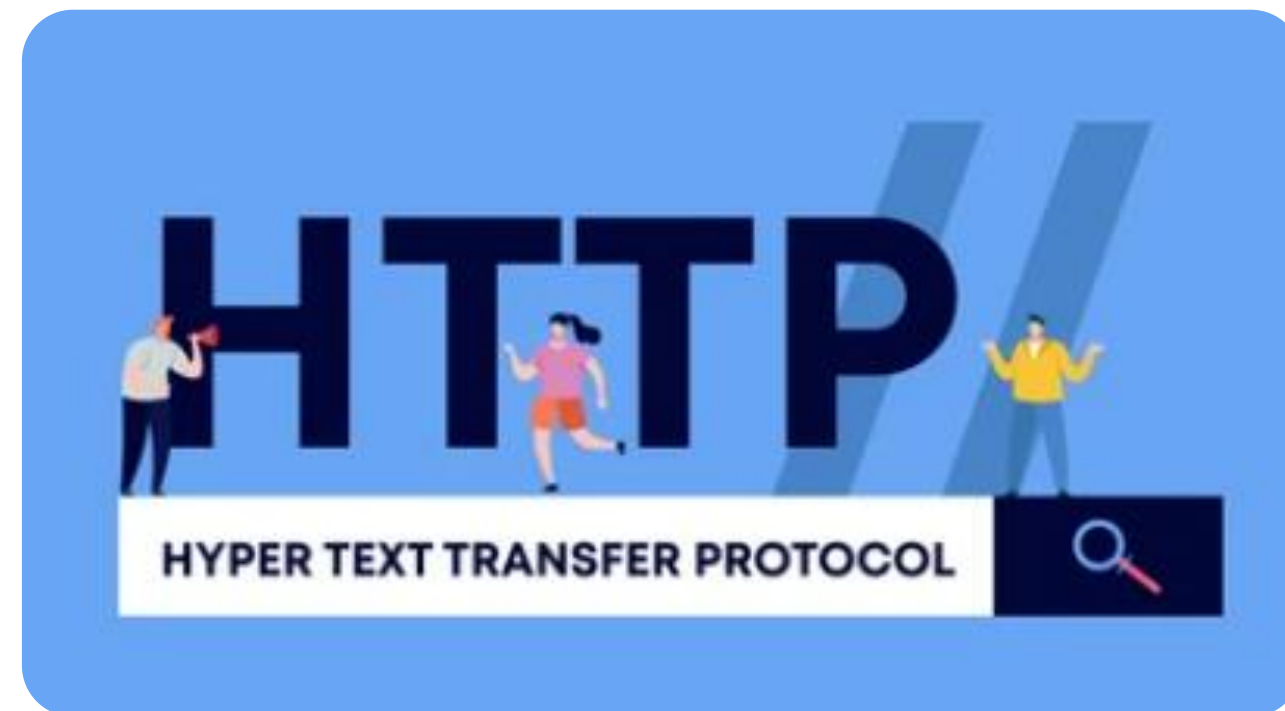


The user in turn must validate this status code and process it accordingly.



Validating the HTTP Error Status Code

Scenario: User validates the HTTP status code returned by BookStore web service when an invalid parameter is entered.



Validating the HTTP Error Status Code

Here the user has provided a nonexistent user ID as the parameter to get user details.

The code is as below:

```
import static org.junit.Assert.*;
import org.testng.Assert;    //used to validate response status
import org.testng.annotations.Test;
import io.restassured.RestAssured;
import io.restassured.response.Response;
import io.restassured.specification.RequestSpecification;
```

Validating the HTTP Error Status Code

```
public class RestAssuredTestResponse {

@Test
    public void GetPetDetails()
    {

        // Specify the base URL to the RESTful web service
        RestAssured.baseURI =
        "https://demoqa.com/Account/v1/User/";

        // Get the RequestSpecification of the request to
        be sent to the server
        RequestSpecification httpRequest =
        RestAssured.given();
```

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

// Get the status code of the request.

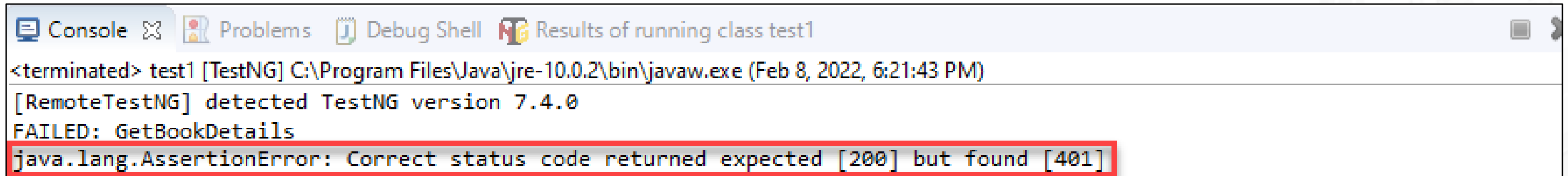
//If request is successful, status code will be
200
int statusCode = response.getStatusCode();

// Assert that correct status code is returned.
Assert.assertEquals(statusCode /*actual value*/,
200 /*expected value*/,
"Correct status code returned");
}
```

Validating the HTTP Error Status Code

When the user runs the test, it returns the error code 401.

The result of the test execution is given below:

A screenshot of an IDE's console window. The title bar shows tabs for 'Console', 'Problems', 'Debug Shell', and 'Results of running class test1'. The console output shows the following text: '<terminated> test1 [TestNG] C:\Program Files\Java\jre-10.0.2\bin\javaw.exe (Feb 8, 2022, 6:21:43 PM)', '[RemoteTestNG] detected TestNG version 7.4.0', 'FAILED: GetBookDetails', and 'java.lang.AssertionError: Correct status code returned expected [200] but found [401]'. The last line is highlighted with a red background.

```
<terminated> test1 [TestNG] C:\Program Files\Java\jre-10.0.2\bin\javaw.exe (Feb 8, 2022, 6:21:43 PM)
[RemoteTestNG] detected TestNG version 7.4.0
FAILED: GetBookDetails
java.lang.AssertionError: Correct status code returned expected [200] but found [401]
```

Image source: www.toolsqa.com

Validating the HTTP Error Status Code

Now, the user changes the code to make sure the test passes.

The change is as below:

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

Here the expected value returned is 401 instead of 200, hence the test is passed.

Validating the Response Status Line

A **status line** is the first line returned in the HTTP response.



Validating the Response Status Line

The status line consists of three substrings:



- HTTP protocol version
- Status code
- Status code's string value

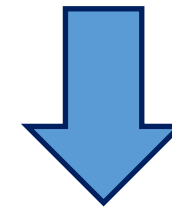
Validating the Response Status Line

For example, when the request is successful the status line will have the value **HTTP/1.1 200 OK**.



Validating the Response Status Line

HTTP/1.1 200 OK



- **HTTP/1.1:** The HTTP protocol
- **200:** The HTTP status code
- **OK:** The status message



Validating the Response Status Line

The user can read the entire status line using the method `getStatusLine()` of the response interface.

The code is given below:

```
@Test
public void GetBookDetails()
{

    // Specify the base URL to the RESTful web service
    RestAssured.baseURI = "https://demoqa.com/BookStore/v1/Books";
```

Validating the Response Status Line

```
// Get the RequestSpecification of the request to be sent to the server
RequestSpecification httpRequest = RestAssured.given();
Response response = httpRequest.get("");

// Get the status line from the Response in a variable called statusLine
String statusLine = response.getStatusLine();
Assert.assertEquals(statusLine /*actual value*/, "HTTP/1.1 200 OK"
/*expected value*/, "Correct status code returned");

}
```



Validating the Response Status Line

Here, the user has performed a similar test as it was done for the status code.



- The user must read the status line using the `getStatusLine()` method into a string value.
- Then the user must compare the returned value with **HTTP/1.1 200 OK** to check if the status is successful or not.

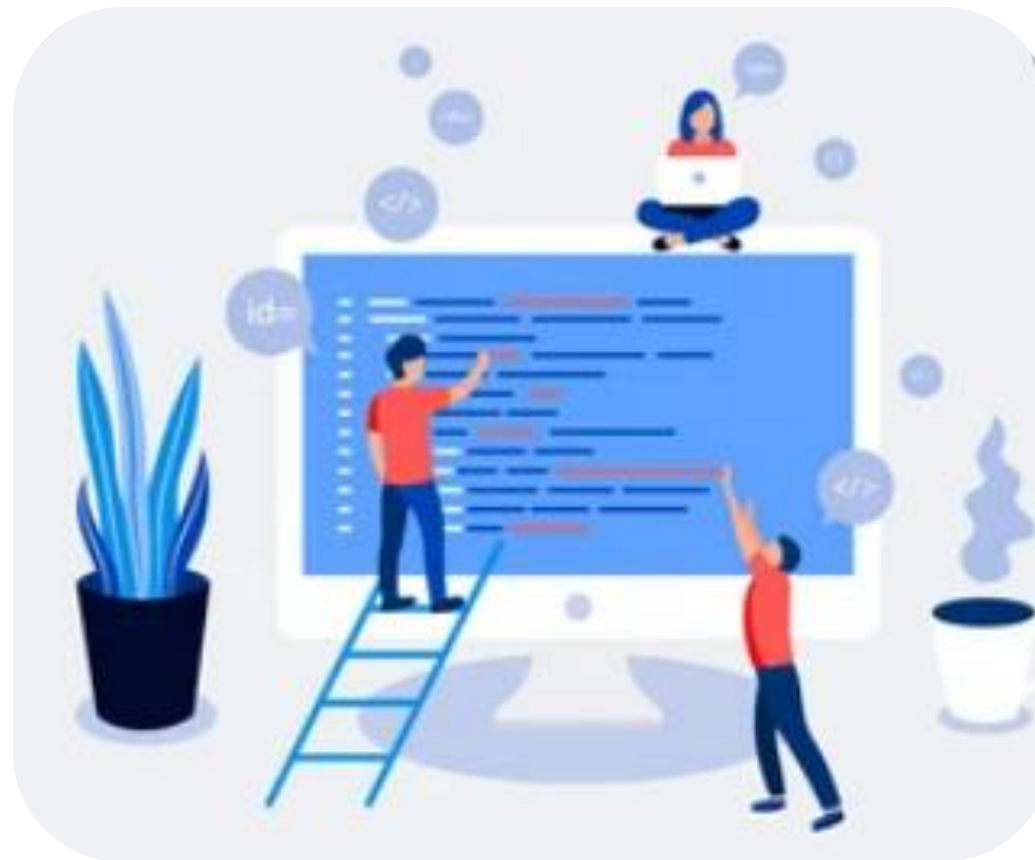
Validating the Response Status Line

Thus, the user can read the status code using the `getStatusCode()` method and the status line using the `getStatusLine()` method of the response interface.



Validating the Response Status Line

Once the status is read, then the user can easily verify if the code is a success (200) or any other code.



TECHNOLOGY

End-to-End Case Automation

Overview: End-to-End Testing

End-to-end testing is a methodology used in the Software Development Lifecycle (SDLC) to evaluate an application's performance and functionality using data that closely resembles real-world conditions.



Overview: End-to-End Testing

The objective is to completely recreate a real user scenario in the simulation.



The goal of this testing is to validate the system being tested as well as to make sure that all its supporting systems function properly and behave as intended.

Overview: End-to-End Testing

The complexity of software has increased a lot in the present world.



Complete networks of sub-systems and layers, such as UI and API layers, external databases, networks, and even third-party integrations, are the foundations on which applications are created.

Overview: End-to-End Testing

The stability of each element is crucial for the success of an application because when one fails, the whole thing fails.



Thus, end-to-end tests should be conducted utilizing both automated testing and manual testing strategies, enabling the users to maximize coverage and discover new bugs in an exploratory manner.

Requirements for REST API End-to-End Test

Listed below are the tools which are required for end-to-end testing in REST API:



- Java setup
- IDE setup
- Maven setup
- Create a Maven project
- Add REST Assured dependencies
- Setup Maven compiler plugin
- Create a user for the test

Setting up Tools for End-to-End Test

Step 1: Java installation



Java language can be used for writing the REST API automation framework based on the REST Assured library. For this, the user needs to install Java in the system.

Setting up Tools for End-to-End Test

Step 2: IDE setup



If the user wants to work with Java, an editor would be required to use Java.
Eclipse, IntelliJ, Net Beans, and several other popular IDEs can be chosen to work with.

Setting up Tools for End-to-End Test

Step 3: Maven setup



The build tools support the automation and scripting of routine tasks like compiling, downloading dependencies, running our tests, and deployments. The Maven build tool is available for usage with end-to-end scenarios.

Setting up Tools for End-to-End Test

Step 4: Creating a new Maven project



Eclipse operates with workspaces, folders, and spaces where users can add projects. Thus, a user can create a new Maven project in the Eclipse workspace.

Setting up Tools for End-to-End Test

While creating a new Maven project, values must be specified to Maven archetype parameters, as shown below:

New Maven Project

New Maven project

Specify Archetype parameters

Group Id: ToolsQA

Artifact Id: APITestingFramework

Version: 0.0.1-SNAPSHOT

Package: ToolsQA.APITestingFramework

Properties available from archetype:

Name	Value

Add...

Remove

Advanced

?

< Back

Next >

Finish

Cancel

Image source: www.toolsqa.com

Setting up Tools for End-to-End Test

Step 5: Adding the REST Assured dependencies

Through the pom.xml file, the user can add REST Assured dependencies to the project.



- Users can visit the REST Assured Maven repository to add the necessary dependencies.
- Users can copy-paste the most recent dependency into the project's pom.xml file after choosing it.

Setting up Tools for End-to-End Test

The required dependency according to the Eclipse IDE should be copy pasted in the pom.xml file:

```
<!-- https://mvnrepository.com/artifact/io.rest-assured/rest-assured -->
<dependency>
  <groupId>io.rest-assured</groupId>
  <artifactId>rest-assured</artifactId>
  <version>5.1.0</version>
  <scope>test</scope>
</dependency>
```

Setting up Tools for End-to-End Test

The user should add the dependency tag as shown below:

```
<dependencies>
    <dependency>Project Dependency</dependency>
</dependencies>
```

Setting up Tools for End-to-End Test

The user should also add the **JUnit dependency** in the pom.xml.

```
<!--  
https://mvnrepository.com/artifact/org.junit.jupiter/junit-  
jupiter-api -->  
<dependency>  
  <groupId>org.junit.jupiter</groupId>  
  <artifactId>junit-jupiter-api</artifactId>  
  <version>5.9.0</version>  
  <scope>test</scope>  
</dependency>
```


Setting up Tools for End-to-End Test

Step 6: Set up the Maven compiler plugin

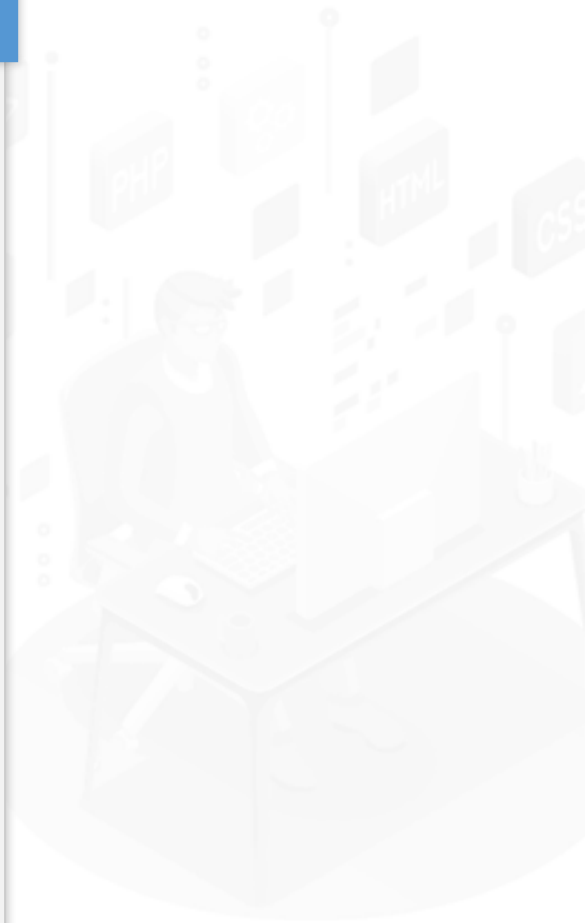


The project's sources are compiled via the compiler plugin. Users may use any JDK to execute Maven, but 1.5 should be the default feature for both the source and target settings.

Setting up Tools for End-to-End Test

Maven compiler plugin settings:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.7.0</version>
      <configuration>
        <source>1.8</source>
        <target>1.8</target>
        <encoding>UTF-8</encoding>
      </configuration>
    </plugin>
  </plugins>
</build>
```



Setting up Tools for End-to-End Test

Step 7: Create an authorized user for the test



The last step is creating an authorized user before the user starts automation testing. A user must be authorized before automating tests.

Setting up Tools for End-to-End Test

The code is as shown below:

```
@Test
    public void RegistrationSuccessful() {
        RestAssured.baseURI =
"https://bookstore.toolsqa.com";
        RequestSpecification request = RestAssured.given();

        JSONObject requestParams = new JSONObject();
        /*I have put a unique username and password as below,
        you can enter any as per your liking. */
        requestParams.put("UserName", "TOOLSQA-Test");
        requestParams.put("Password", "Test@@123");
```



Setting up Tools for End-to-End Test

The code is as shown below:

```
request.body(requestParams.toJSONString());
Response response = request.post("/Account/v1/User");

Assert.assertEquals(response.getStatusCode(), 201);
// We will need the userID in the response body for
our tests, please save it in a local variable
String userID =
response.getBody().jsonPath().getString("userID");
}
```



Method of Writing an End-to-End Test

The method of writing an end-to-end test can be justified using a scenario.

The scenario is as follows:



It is considered that the user is an existing authorized user, so:

- The user can retrieve a list of books available in the library.
- The user can assign a book to himself and later return it.

Method of Writing an End-to-End Test

Step 1: Generation of token for authorization



- The user should hold a registered username and password.
- The user should create a token with the help of these credentials.
- The user will then substitute the token for the username and password in the request.

Method of Writing an End-to-End Test

This procedure is followed to allocate particular resources in a time-bound manner.



The initial requirement in this stage is to make a POST request call in REST Assured and provide a username and password.

Method of Writing an End-to-End Test

Step 2: Get the list of available books in the library



This is a GET request call. It gets the user the list of available books.



Method of Writing an End-to-End Test

Step 3: Addition of a book from the list to the user



The next procedure is a POST request call where user and book details are sent in the request.

Method of Writing an End-to-End Test

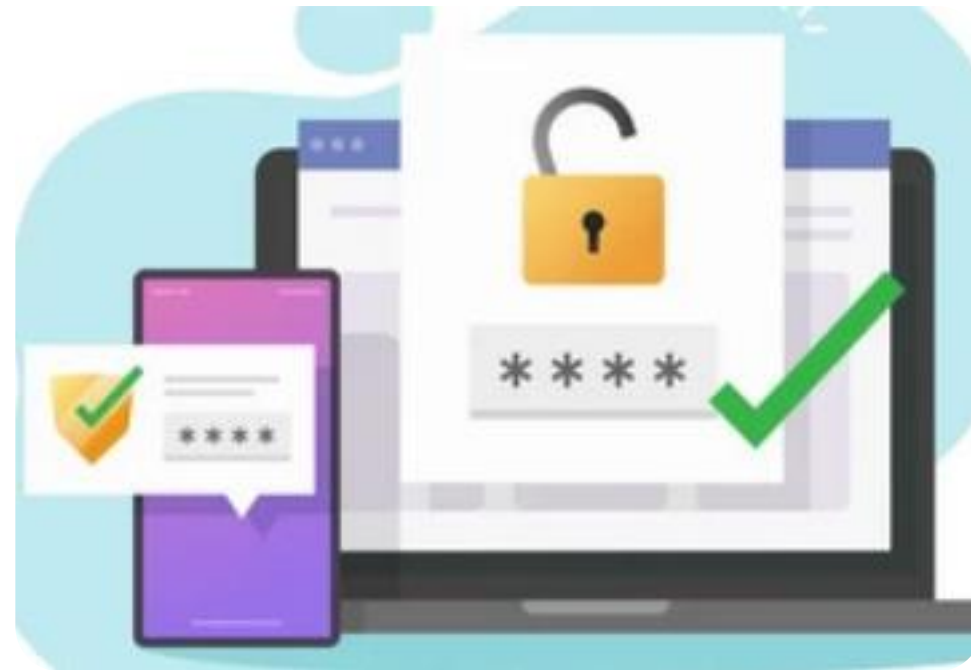
Step 4: Deletion of the added book from the list



This step is performed for the DELETE request call. Here the user can delete the added book from the list.

Method of Writing an End-to-End Test

Step 5: Confirmation of whether the book has been removed successfully or not



Lastly, the user will verify if the book has got removed from the list or not. Therefore, the user will send the user details in a GET request call to know the details.

Method of Writing an End-to-End Test

Make a test class and test package

Firstly, the user must create a **new package** by right-clicking on the **src/test/java** package and selecting **new >> package**.



The user can name the package as **apiTests** and click **finish**.

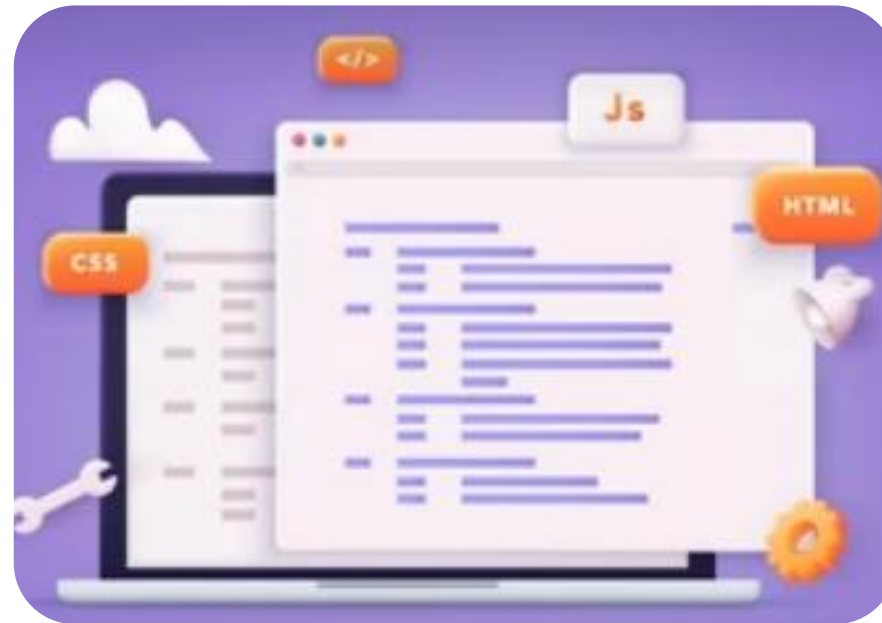
Method of Writing an End-to-End Test

Secondly, the user must create a **new class** file by right-clicking on the **src/test/java** package and selecting **new >> class**.



Method of Writing an End-to-End Test

The user should provide a name to the test case in the dialog box and click **finish** to create the file.



The user should check the **public static void main**, as the test will be performed from the same primary method.

Method of Writing an End-to-End Test

In this case, the class name given is **E2E_Tests**

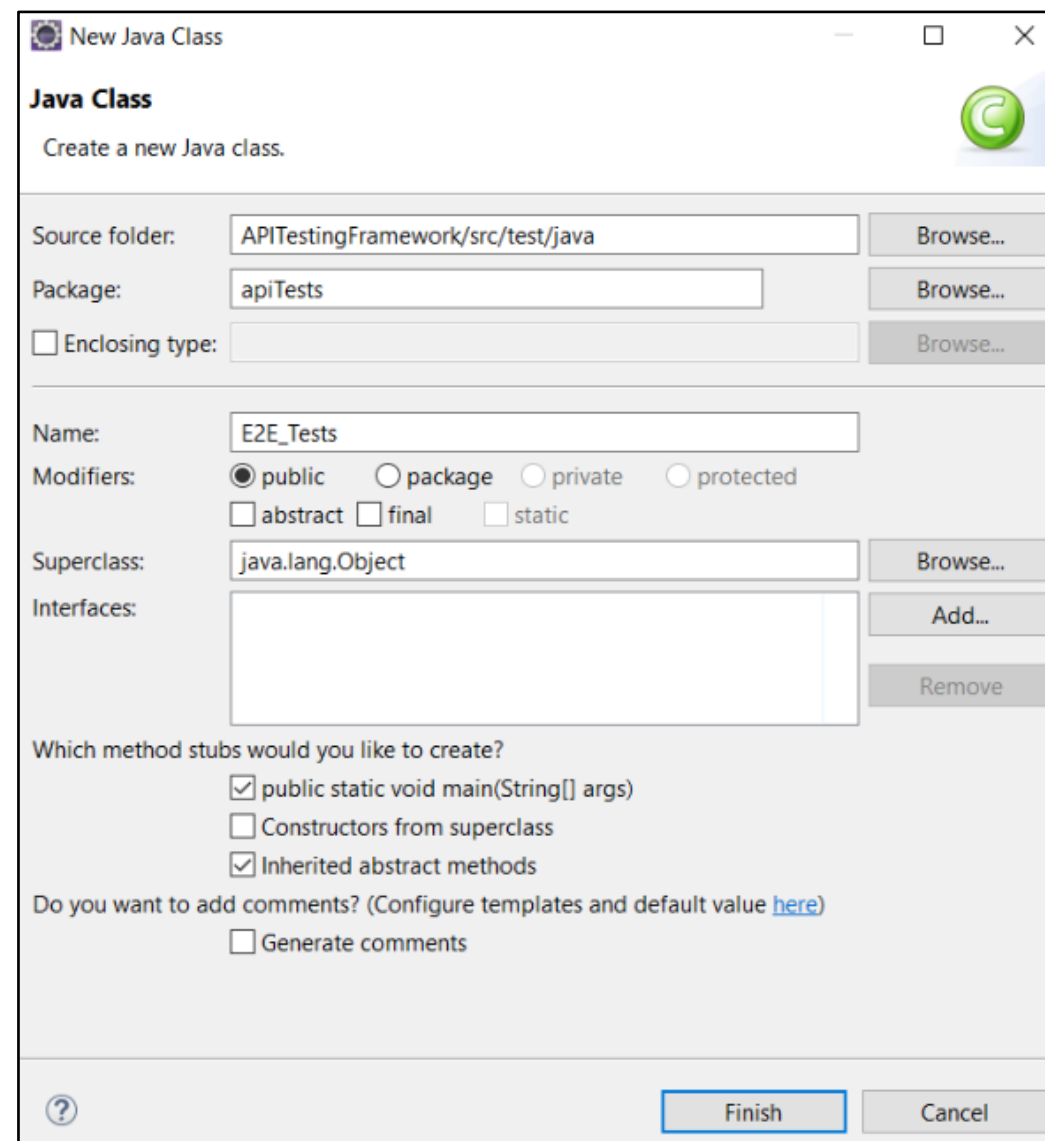
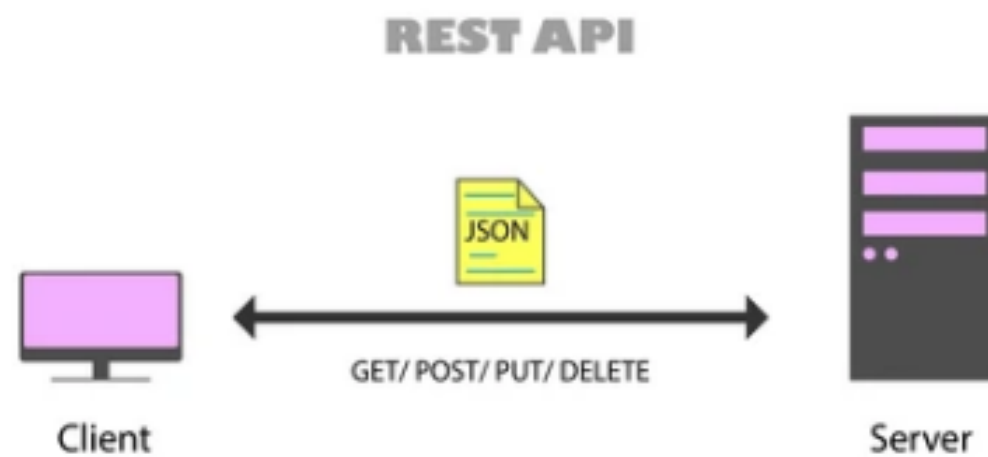


Image source: www.toolsqa.com

Method of Writing an End-to-End Test

Now, the user should write an end-to-end test including all the steps.



The test involves the use of GET, POST, and DELETE requests.

Method of Writing an End-to-End Test

```
package apiTests;

import java.util.List;
import java.util.Map;

import org.junit.Assert;

import io.restassured.RestAssured;
import io.restassured.path.json.JsonPath;
import io.restassured.response.Response;
import io.restassured.specification.RequestSpecification;
```



Method of Writing an End-to-End Test

```
public class E2E_Tests {  
  
    public static void main(String[] args) {  
        String userID = "9b5f49ab-eea9-45f4-9d66-  
bcb56a531b85";  
        String userName = "TOOLSQA-Test";  
        String password = "Test@@123";  
        String baseUrl = "https://bookstore.toolsqa.com";  
  
        RestAssured.baseURI = baseUrl;  
        RequestSpecification request = RestAssured.given();
```



Method of Writing an End-to-End Test

```
//Step - 1
//Test will start from generating Token for Authorization
request.header("Content-Type", "application/json");

Response response = request.body("{ \"userName\": \"" +
userName + "\", \"password\": \"" + password + "\"}")
    .post("/Account/v1/GenerateToken");

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

String jsonString = response.asString();
Assert.assertTrue(jsonString.contains("token"));

//This token will be used in later requests
String token = JsonPath.from(jsonString).get("token");
```

Method of Writing an End-to-End Test

```
//Step - 2
// Get Books - No Auth is required for this.
response = request.get("/BookStore/v1/Books");

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

jsonString = response.asString();
List<Map<String, String>> books =
JsonPath.from(jsonString).get("books");
Assert.assertTrue(books.size() > 0);

//This bookId will be used in later requests, to add the book with
respective isbn
String bookId = books.get(0).get("isbn");
```


Method of Writing an End-to-End Test

```
//Step - 3
// Add a book - with Auth
//The token we had saved in the variable before from response in Step
1,
//we will be passing in the headers for each of the succeeding request
request.header("Authorization", "Bearer " + token)
    .header("Content-Type", "application/json");

response = request.body("{ \"userId\": \"\" + userID + "\", \" +
    \"collectionOfIsbns\": [ { \"isbn\": \"\" + bookId + "\" }
] }")
    .post("/BookStore/v1/Books");

Assert.assertEquals( 201, response.getStatusCode());
```

Method of Writing an End-to-End Test

```
//Step - 4
// Delete a book - with Auth
request.header("Authorization", "Bearer " + token)
    .header("Content-Type", "application/json");

response = request.body("{ \"isbn\": \"\" + bookId + \"\", \"userId\": \"\" + userID + \"\"}")
    .delete("/BookStore/v1/Book");

Assert.assertEquals(204, response.getStatusCode());
```

Method of Writing an End-to-End Test

```
//Step - 5
// Get User
request.header("Authorization", "Bearer " + token)
    .header("Content-Type", "application/json");

response = request.get("/Account/v1/User/" + userID);
Assert.assertEquals(200, response.getStatusCode());

jsonString = response.asString();
List<Map<String, String>> booksOfUser =
JsonPath.from(jsonString).get("books");
Assert.assertEquals(0, booksOfUser.size());
    }
}
```

Run the API Test

The final step is to execute the test.



Here, the user should right-click in the test body to select **run as** and then should click on the **java application** option.

Run the API Test

Once the test runs, the user will be able to view the results in the Console.



Consequently, the program gets executed successfully without any errors.

Run the API Test

In some cases, if the user configures or runs the test on IntelliJ IDE, the result is: **process finished with exit code 0.**



Run the API Test

This kind of message indicates that the user's test got executed successfully.



Key Takeaways

- REST Assured uses a `post ()` method to make HTTP POST requests.
- REST Assured library provides a response interface that provides numerous methods to extract the response fields.
- The response obtained from the server consists of status, header, and body.
- A user can read the status code using the `getStatusCode()` method and can read the status line using the `getStatusLine()` method of the response interface.

