For any web application, the API services work as the **building block**. Therefore it becomes very important for you, as a tester, to ensure that the **web services are bug-free** and facilitate the **exchange of information securely**.

**What is Client Server Architecture?**

Client server architecture in a simple sense can be stated as a **consumer-producer model where the client acts as the consumer i.e. the service requestor and the server is the producer, *i.e.* the service provider**. Let us see how they are described in the computing sense.

***What is a Client?***

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

* ***A thick or fat client performs the operation itself without the need for a server****.* ***A personal computer*** *is a fine example of a thick client as the majority of its operations are independent of an external server.*
* ***A thin client uses the host computer's resources, where the thin client presents the data processed by an application server. A web browser is an example of a thin client.***
* ***A hybrid client is a combination of a thick and a thin client****. Just like the thick client it processes data internally but relies on the application server for persistent data.* ***Any online gaming run*** *on a system can be an example of a hybrid client.*

***What is a Server?***

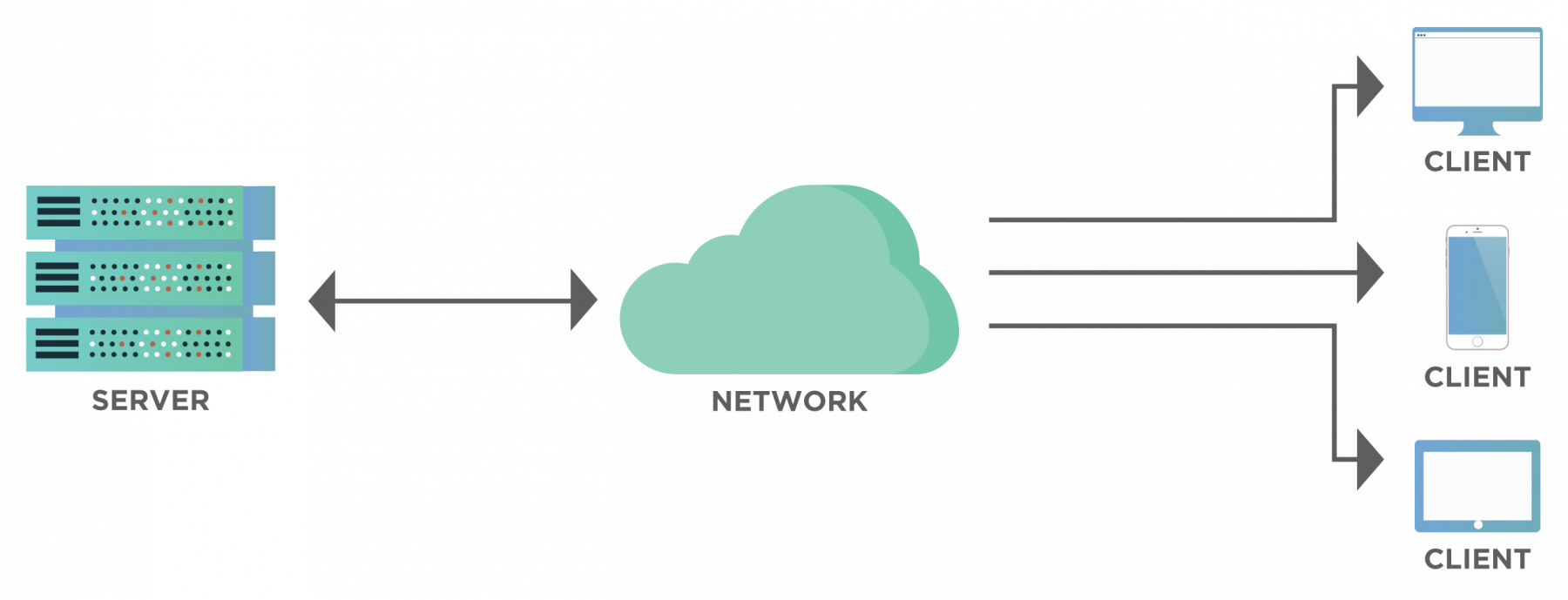
A server is a system or a computer program **that acts as the data provider**. It may provide data through LAN (*Local Area Network*) or WAN (*Wide Area Network*) using the internet. **The functionalities provided by the server are called services**. These services are provided as a response to the request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

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

These clients and servers do not necessarily be at the same location. They could either be located in different locations or may reside as different processes on the same computer. They are connected via the Web and interact via the ***HTTP protocol*** which will be discussed later in this article. There may be multiple clients requesting a server and alternatively, a client can request from multiple servers.

***Visualising the client server architecture***

The below diagram visualizes a typical client-server model-



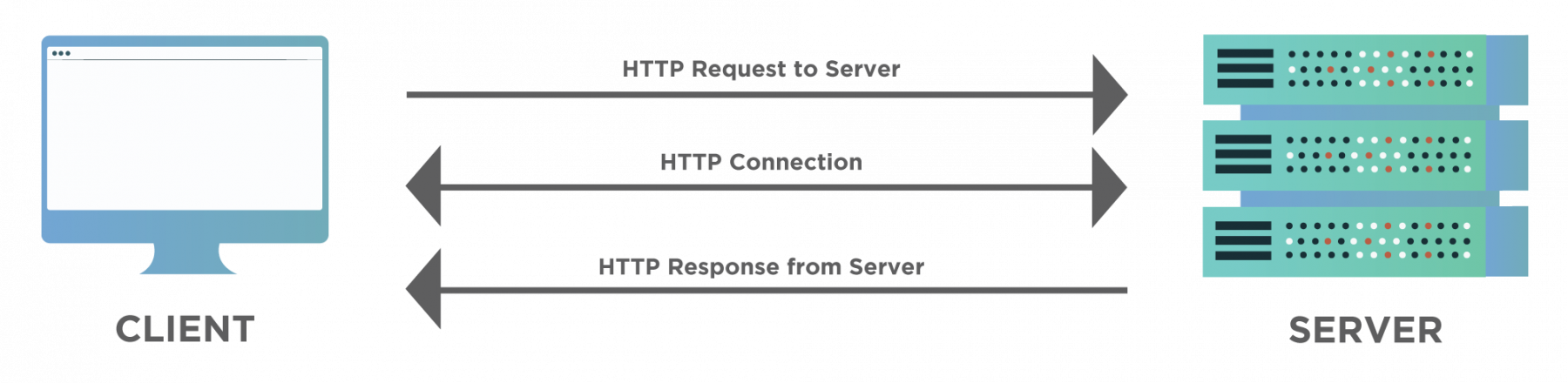
As shown in the diagram, **a single server may serve the requests of multiple clients.** Similarly, **a client may be requesting data from different servers**. For example, consider an example of *Google*. Here, Google acts as the server and the users sitting at different places act as the clients.

Another example that we come across in our everyday life is the Online Banking Portal in which the browser that we use to open a portal acts as the client while the database and the software for banking act as the server. This propagates resource sharing across multiple users and thereby results in cost-efficiency along with time saving.

**How does the Client Server Architecture work?**

The client server model acts like a consumer-server relationship. But how does it work? Let us see some of the flow of information in a client-server architecture through a series of steps as discussed below-

1. ***The HTTP communication protocol helps establish the connection between the client and the server.***
2. ***The client sends a request in the form of an XML or a JSON over the active connection. The client and server both understand the message format.***
3. ***Upon receiving the client request, the server searches the requested data and sends back the relevant details as a response in the same format in which the request was received.***



The above diagram shows the communication process between the client and the server. The client sends an ***HTTP request***, for which the server sends an ***HTTP response***.

***Visualising the flow of client server architecture***

Assume that you need to go shopping from your home to a shop across the road. You may travel on a bicycle and on reaching can order the salesman the goods you need. Now if the salesman finds the goods, he will bring them to you else he would let you know about their unavailability.

Now, you may link the above example with the navigation to a website, the street between your home and the shop is the internet connection. The mode of transport that you chose to travel is the [***TCP/IP***](http://www.tcpipguide.com/), defining the communication protocol for the data to travel through the internet. The address of the shop is the DNS (*Domain Name Server*) of the website. Your communication language with the salesman is the HTTP (*Hypertext Transfer Protocol*) which defines the language for interaction between the client and the server. Your order request is the [***HTTP request***](https://www.toolsqa.com/client-server/http-request/) and the update on item availability is the [***HTTP response***](https://www.toolsqa.com/client-server/http-response/).

When the web browser sends a request to the server with the DNS of the website, and the server approves the client request, it sends a 200 OK success message. This message means that the server has located the website and it sends back the website files in small chunks of data to the browser. The browser then collects and assembles these small chunks to form the complete website and displays it to us.

**What are the different types of Client-Server architectures?**

Client-server architecture has the following four types -

1. *1-Tier Architecture*
2. *2-Tier Architecture*
3. *3-Tier Architecture*
4. *N-Tier Architecture*

***1-Tier Architecture***

In 1-tier architecture, **the business logic, data logic, and the user interface all reside on the same machine**.

The environment is simple and cheap because **the client and the server lie on the same system**, but the data variance leads to the repetition of work. Such systems store data in a local file or a shared driver. Examples of 1-tier applications are the **MP3 player** or the MS Office files.

***2-Tier Architecture***

The 2-tier architecture provides the best environment in terms of performance due to the absence of any intervening server.

**The user interface resides on the client side while the database on the server side. The database and business logic can be stored either at the client or the server end, but they must remain unchanged.**

If both reside at the client end then the architecture is called ***fat client thin server architecture***. On the contrary, if both reside at the server end, the architecture is called ***thin client fat server architecture***. An online ticket reservation system generally uses a 2-tier architecture.

***3-Tier Architecture***

The 3-tier architecture involves a middleware used for interaction between the client and the server. Though it is expensive but is very easy to use. The middleware improves performance and flexibility. It stores the business and the data logic. The three layers in the 3-tier architecture are-

* *Presentation Layer(Client tier)*
* *Application Layer(Business tier)*
* *Database Layer(Data tier)*

**Almost all web applications are examples of 3-tier architecture**.

***N-Tier Architecture***

The n-tier architecture is the scaled form of 3-tier architecture. In such an environment, the processing, data management, and presentation function are isolated in different layers. The isolation makes the system easy to manage and maintain. This is also referred to as multi-tier architecture.

**What are the advantages of using Client-Server Architecture?**

Client-server architecture has the following advantages -

* *It maintains data at one central location.*
* *It provides backup and data recovery options.*
* *Accessing the data from a single server help in cost-efficiency with less maintenance.*
* *The capacity of the client and the server can be individually modified.*

**Issues of Client-Server Architecture?**

Client-server architecture has the following disadvantages -

* *A virus can easily attack a client present on the server.*
* *In case of network failure, the entire architecture of an application can suffer.*
* *Man-in-the-middle attack or data spoofing is possible during data transmission.*
* *Due to vulnerability to different kinds of attacks, it requires a special and secure network operating system.*
* *Possibility of loss of data packets either completely or modification because of some intrusion during the transmission.*

**Key Takeaways**

So, what are the takeaways from this article?

* *The client server model is similar to a salesman-customer relationship.*
* *The communication between the client and the server facilitates through the HTTP protocol.*
* *The client-server architecture is categorized into 1-tier, 2-tier, 3-tier and n-tier.*
* *We discussed the various advantages of the client-server model including its centralized framework.*
* *We then saw the issues with the architecture as to how the client-server model is vulnerable to different attacks.*

The world wide web has become too complex today. With hundreds and thousands of electronic media talking to each other, the process of communication has become too complex and heavy on computation for sure. Such scenarios call for something that can ease out the communication and keep the coding complexity to a minimum as well.  That something is the focus of this post and is termed as REST.

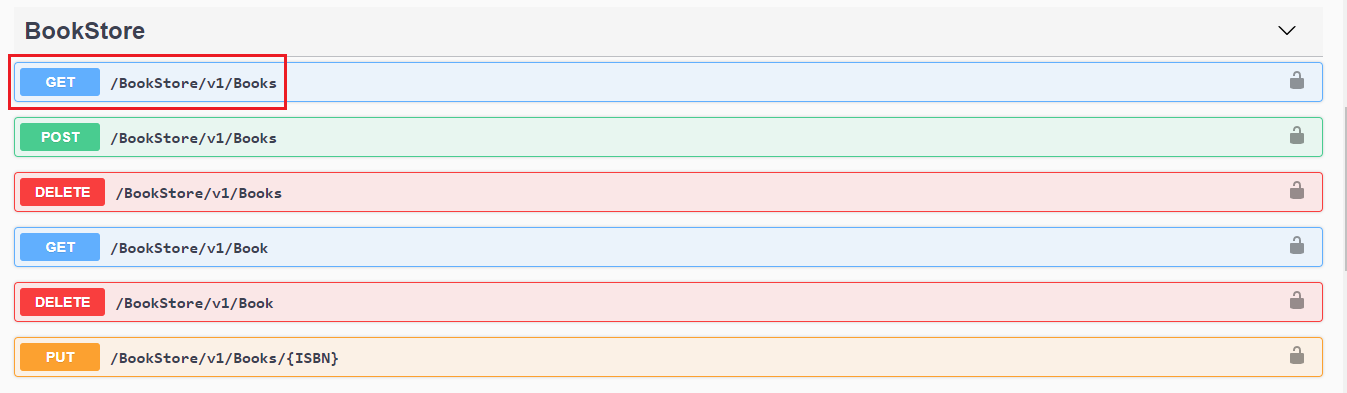
REST, or **RE**presentational **S**tate **T**ransfer, is an architectural style and provides standards between the systems on the web. Popularly, RESTful systems use the REST style to communicate with each other. So in many places, you might see ***"RESTful"*** over the internet and it means the same thing. In this article, we dwell in more details about REST basics and will cover the following topics in this article.

* *What is REST - Representational State Transfer?*
* *What are Clients and Resources?*
* *Guiding principles or constraints of REST*
  + *Uniform interface*
  + *Client-server separation*
  + *Stateless*
  + *Layered system*
  + *Cacheable*
  + *Code-on-demand*

**What is REST - Representational State Transfer?**

A web application developed using REST *( RESTful web application)* exposes the data or information about its resources which can be anything that the developer wants. This allows the client using this application to take action on the resources. For example, using information exposed to users, clients can create a new user.

As another example, suppose we have a pet store and the information related to all pets in the store is stored on the server



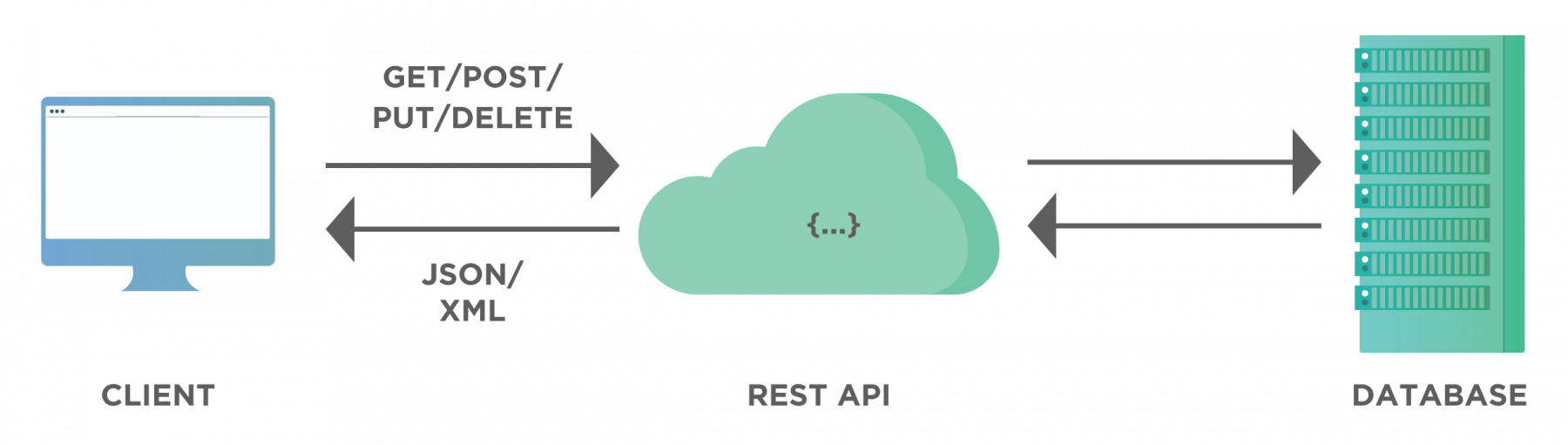
All the methods *(coloured buttons)* shown in the above screenshot correspond to REST APIs that are executed with the click of a button. In general, all this is done in the back-end and the client is not shown any of this information. Coming back, when we click the ***"GET"*** button *(/Books),* we will get the response in JSON format that will show us the details of the particular book. This response is shown below:



*Note: When we click on the****"GET"****button, in the browser we see the link changes to*[***"https://demoqa.com/BookStore/v1/Books".***](https://demoqa.com/swagger/#/BookStore/BookStoreV1BooksGet)

***Representation of REST flow***

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



As the above diagram shows, the API works as a medium of communication between a database *(that is a part of a server)* and a client. When a client sends data through APIs, it goes to the database, do the appropriate operation *(such as add, delete, modify etc.)* and return the response data that contains response code, header files, cookie info etc.

We can summarise the REST characteristics and the working in the above diagram as follows:

* *A client access data from the server passed by REST. This can either be through authorized access or without following any strict guideline.*
* *As we can see, the application developed using REST is an interactive application and mostly it uses web services. In this case, the web service follows RESTful guidelines and fulfils the constraints of REST that are discussed later in this article.*
* *A web service using REST provides web resources in textual format and allows them to be read and edited using a predefined set of operations.*

REST is a way to access a web service and is often viewed as an alternative to [***SOAP***](https://www.toolsqa.com/soapui/what-is-soapui/) *(Simple Object Access Protocol).*

In a RESTFul application, we have entities namely client and resource which are used commonly. Let us discuss them briefly next.

**What are Clients and Resources?**

**A client can be a software or a person or a system using the APIs to access data from another application server**.

For example, a developer might access Facebook APIs to embed a live post in their own website. The developer program will call the Facebook API through a browser. So in this case, the browser acts as a client that is calling the Facebook APIs.

If we visualize this system using the REST diagram above, the client or browser will connect to Facebook-Server over REST API and then get the information required to render it on the screen.

***What Is A Resource in REST?***

**Any smaller unit that can be transformed and addressed through a URL and HTTP method is considered a resource**.

This resource makes changes to the database.

For one application, you may have a lot of resources with all of them assigned a particular task. For example, an online book store may have a resource as a table of the database. A resource in a REST architecture is anything that a client has access to and can modify or update.

**Guiding principles or constraints of REST**

For an API to be RESTful, it has to fulfil or adhere to the following guiding principles or constraints defined by REST.

***Uniform interface***

The uniform interface principle has the following parts that an API has to follow:

1. *The request to the server needs to have a resource identifier.*
2. *The server returns the response and includes information such that the client can modify or edit the resource.*
3. *The request sent through the API contains the information that the server needs to execute the request. Each response returned by the server also contains all the information so that client understands the response.*
4. *Hypermedia as the engine of application state. The application means the web application that is running on the server. Hypermedia is the hyperlinks or simple links included in the response. So the server basically informs the client about the ways to change the application web.*

***Client-server separation***

The interaction between the client and the server is independent and is only in the form of requests. The client initiates a request and the server sends the responses. The response is a reaction to the request. So all server does is wait to receive requests from the client. It never sends out information about the resources to the client on its own. For more details on client-server refer to our tutorial, [***Client-server Architecture.***](https://toolsqa.com/client-server/client-server-architecture-and-model)

***Stateless***

The word ***"stateless"*** means the server does not keep track of the user who uses the API. So when a new request comes in, the server will not know if the same user has sent a GET request in the past for the same resource. It doesn't remember the resources requested by the user earlier. For example, HTTP is a stateless protocol. HTTP server does not keep track of any state of information passed to and fro. Hence at any given time, a client can send a valid command, and the server will not relate or keep track of any previous similar commands.

Hence each request regardless of the other requests made by the user will contain all the data needed by the server to execute the request and send a response.

***Layered system***

A layered system provides a hierarchical structure between a client and a server. There can be a lot of intermediaries between the client and the server working along with REST API without the client's notice. Our clients think there is a direct connection to the server. We then take advantage of it to improve our architecture and bring down our distributed system complexity. These intermediate elements provide a security layer, load-balancing layer, and other functionality to the system. The only guideline is that the presence of these intermediate layers should not affect the request or response.

*Note: The abstraction of layers does not let one layer be aware of the presence of another layer.*

***Cacheable***

The server data received in the response contains information regarding whether the data is cacheable or not. If the data is cacheable, it will contain some kind of version number that makes caching possible. The client will know which version of the data it has got from the previous response. This way client can avoid requesting repeated data. Cacheable data *(and therefore version number)* also helps the client to know the expiration of version data and the requirement of a new request to fetch the latest data.

***Code-on-demand***

This particular constraint ***"Code on demand"*** is optional and without fulfilling it we can have a RESTful API.

The client can send a request to the server asking for the code and then the server will respond with some code in the form of a script or some other entity. For example, servers can extend the client functionality by downloading and executing pre-compiled code like an applet or a client-side script like JavaScript. So when we click on any video on Facebook, Facebook will run a precompiled or any third-party software to run that video.

Once an API fulfils the above constraints we discussed, we can say it is a RESTful API.

**Key Takeaways**

In this article, we discussed the introduction of the REST protocol for web services.

1. *The REST (Representational State Transfer) is a standard that operates between machines mostly executing web services.*
2. *The server receives a request in form of REST API. The server then runs the web application and sends a response to the client.*
3. *For an API to be a valid REST API (RESTful API) it needs to fulfil some constraints that we discussed above.*
4. *A RESTful API adheres to these constraints (also called guiding principles).*

*.*

# What is REST Assured?

Rest-Assured is an open-source Java-based library useful for testing and validating [***REST APIs***](https://www.toolsqa.com/rest-assured/rest-api-end-to-end-test/) or RESTFul Web Services. It is like a headless (no GUI) client that accesses and tests the REST Web Services.

Rest assured simplifies the testing of REST-based services. It brings out the simplicity of dynamic languages like Groovy and Ruby that perform API testing in Java. The library supports any HTTP method and also has explicit support for GET, POST, PUT, DELETE, OPTIONS and HEAD. It also includes specifications and validation like parameters, headers, cookies, etc. We can also use it to verify and validate the response of the HTTP requests.



[***Source***](https://devqa.io/)

Apart from testing JSON-based web services, Rest Assured can also be used to test XML-based web services.

We can also integrate this library with [***JUnit***](https://www.toolsqa.com/java/junit-framework/junit-introduction/) and [***TestNG***](https://www.toolsqa.com/testng/testng-tutorial/) frameworks and write test cases for applications.

Further, it can be integrated well with [***Maven***](https://toolsqa.com/maven/maven-introduction/) and its efficient matching techniques produce straightforward results.

Yet another powerful feature of REST assured is its support for XML Path and JSON Path syntax to check specific elements of the response data similar to using XPath API. For people new to such concepts, please refer to the following syntax examples.

XPath syntax:

<?xml version="1.0" encoding="UTF-8"?>

<bookstore>

<book>

<title>Freedom In Exile</title>

<price>14.29</price>

</book>

</bookstore>

JSON path syntax:

$['store']['book'][3]['title']

Apart from all the above features, this fantastic library also provides various other features like ***DSL-like syntax, Specification Reuse, XPath-Validation, easy file uploads***, and all the features that are conducive to automated testing.

**Why do we need Rest Assured?**

The above discussion about this java library convinces us that it is a reliable library. But why do we need it or what are the reasons to use it in our application testing?

Following are the main reasons we need Rest Assured:

* *It is an* ***open-source library*** *and has an active development community making it a great choice for API automation.*
* *Earlier, we had to use dynamic languages like Ruby, Groovy for API testing and it was quite challenging.*
* ***Validation and testing of REST services are harder in Java. Using REST Assured, it becomes simpler and easier.***
* ***This library uses Java and therefore it becomes simple to send HTTPS requests with customizations using basic Java code****. Once we know the basics of API and*[***integration testing***](https://www.toolsqa.com/software-testing/integration-testing)*, automation using Rest Assured gives good confidence on the backend. Thus we can focus more on front-end testing.*

**Advantages of Rest Assured**

The following table lists some of the advantages of the library.

| **SNo** | **Advantages** |
| --- | --- |
| 1 | It is open-source and hence free to use. |
| 2 | It is very rich in syntax and ready-made assertions. Rest Assured requires less coding as compared to Apache HTTP Client. |
| 3 | The setup of Rest Assured is easy and straightforward. |
| 4 | The response is given in JSON or XML format and is easy to parse and validate. |
| 5 | It uses inbuilt [***Hemcrest Matchers***](https://en.wikipedia.org/wiki/Hamcrest) for easy extraction of values. |
| 6 | Response time is quick as also an assertion of status code. |
| 7 | The library has a powerful logging mechanism. Also, we can verify headers, cookies, content type, etc on the fly. |
| 8 | It can easily be integrated with other Java libraries like TestNG, JUnit, etc. We can also integrate it with Selenium-Java and achieve end-to-end automation. |
| 9 | It has very good support for various API authentication mechanisms. |
| 10 | It supports JsonPath and XmlPath that helps in parsing JSON and XML response. It also has support for the JSON Schema Validation library to verify JSON Schema. |
| 11 | Rest Assured can also be integrated with Maven and CICD. |
| 12 | It supports multi-part form data |
| 13 | Supports Spring Mock MVC, Spring Web Test Client, Scala, and Kotlin. |
| 14 | It follows the [***BDD (Behavioural Data-Driven)***](https://www.toolsqa.com/cucumber/behavior-driven-development/) approach and keywords like given() when(), then() which makes code readable and supports clean coding. This feature is available from version 2.0. |
| 15 | REST Assured 4.1.2 adds support for Java 13. |

**Disadvantages of Rest Assured**

The library has the following disadvantages.

1. *It does not support the testing of*[***SOAP(Simple Object Access Protocol)***](https://www.toolsqa.com/soapui/soapui-tutorial/)*APIs explicitly.*
2. *Using this library requires that the user has good Java programming knowledge*
3. *There is no inbuilt reporting in Rest Assured.*

**Key Takeaways**

We have the following takeaways from this post:

1. *It is an open-source Java library and is free to use.*
2. *It is used for validating and testing Java APIs.*
3. *Apart from other features like Java DSL syntax, also supports all HTTP methods and REST methods like GET, POST, PUT, DELETE etc.*
4. *We can integrate Rest Assured with all major automation frameworks like TestNG, JUnit as well as integrate it with maven and CI/CD.*
5. *The library has a rich syntax and since it is open-source more and more functionality keeps on adding which makes it a very efficient and simple library for API automation.*

# What are the prerequisites to configure Eclipse with Rest Assured?

The first step towards setting up the development environment so that we can test some RESTful APIs is to evaluate the prerequisites. The following list highlights the required software for setting up Eclipse with rest-assured. Before proceeding, please make sure they are installed on your system.

* *Latest Java version.*
* *Working Eclipse IDE.*
* *TestNG setup.*

If any of the above is not present on the system, then we have to follow the links given below :

1. [***Set up Java***](https://toolsqa.com/selenium-webdriver/install-java/)
2. [***Setting up Eclipse***](https://toolsqa.com/selenium-webdriver/download-and-start-eclipse/)
3. [***Set up TestNg***](https://toolsqa.com/testng/install-testng/)

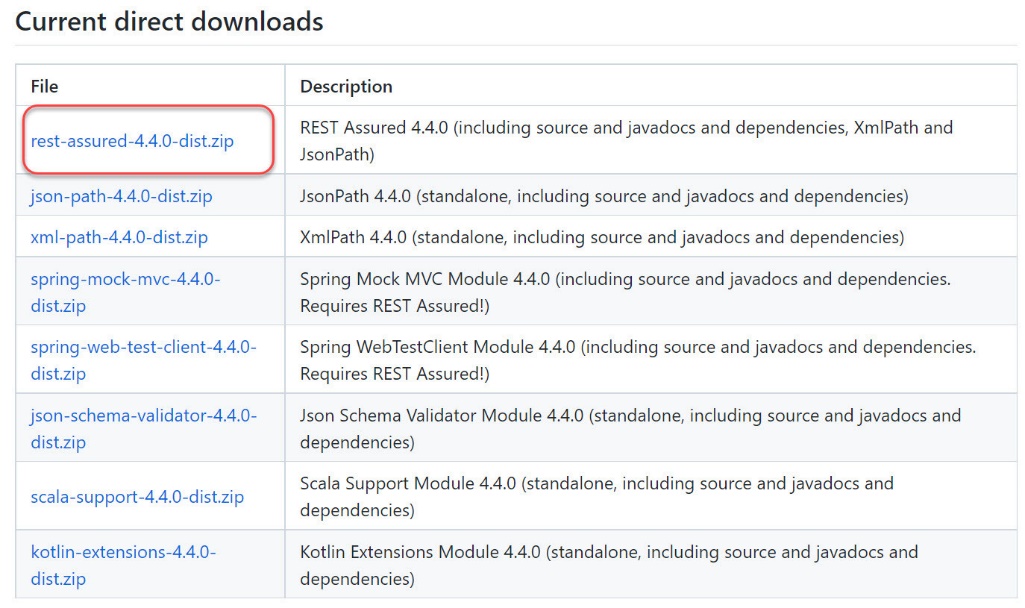
Once the prerequisites are up and running we can proceed with the rest assured set up on Eclipse.

**How to download rest-assured jars in Eclipse?**

The rest assured jar files can be downloaded from the following link.

[***https://github.com/rest-assured/rest-assured/wiki/Downloads.***](https://github.com/rest-assured/rest-assured/wiki/Downloads)

When we navigate to this page, we can see the section “***Current direct downloads***”. Go to this section and click on the link to the ***dist package***.



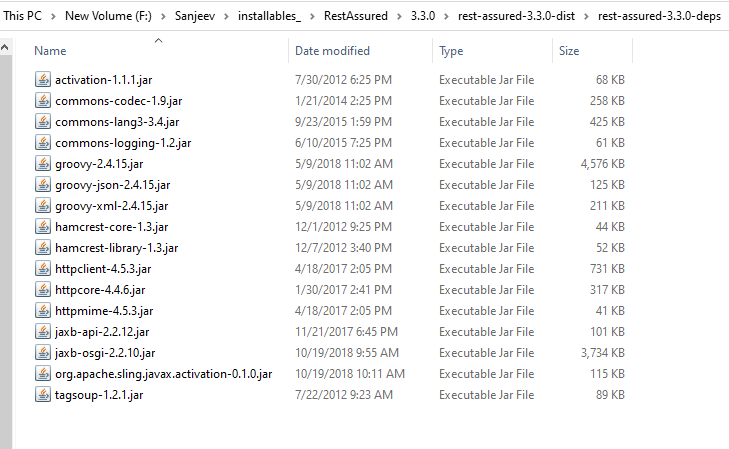
***Note****: At the time of this writing Version 4.4.0 is available for download.*

Once the link is clicked, the browser will download the ***rest-assured-4.4.0-dist.zip*** file on the system (*mostly in the "****downloads****" folder for Windows OS*). Navigate to the folder where the zip file was downloaded and simply unzip the files.

Now go to the folder where the files were just unzipped and open it. The contents should be the following in this folder.

* ***docs****: A folder containing javadocs.*
* ***rest-assured-3.0.3.jar****: jar file that contains rest-sssured classes.*
* ***rest-assured-3.0.3-deps.zip****: Another zip file containing all the dependency jars.*

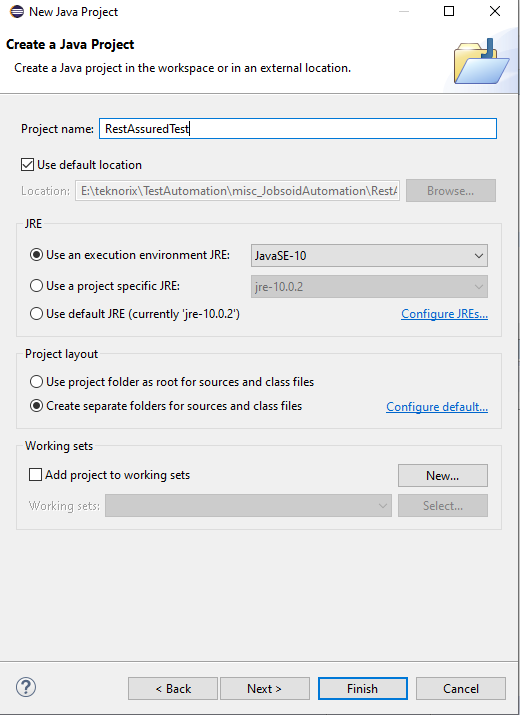
*Note: We have to also unzip the rest-assured-4.4.0-deps.zip dependency jars. In the end, your folder would look like this.*



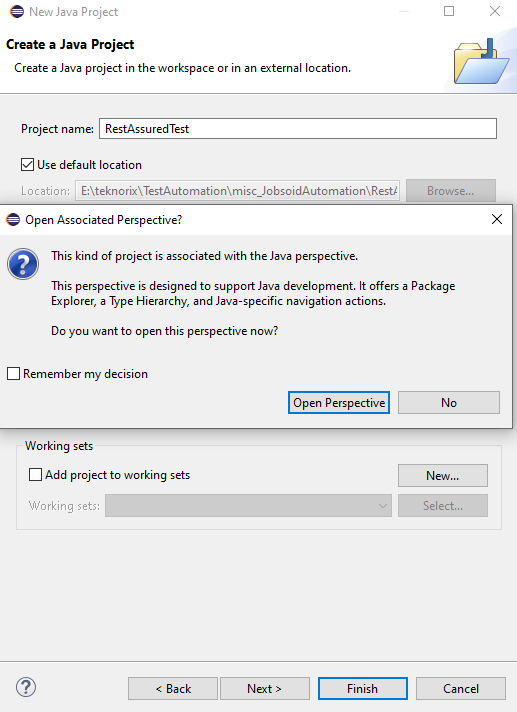
Now that we have rest assured jars package ready,  we can go ahead and create a project in Eclipse that will use rest assured.

**Creating a project in Eclipse for rest assured**

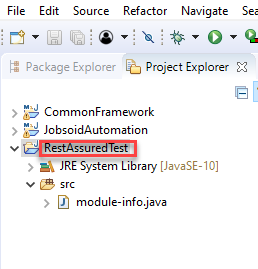
To add and use unzipped rest assured jars in an actual project, we have to first create a new project in Eclipse. So we create a project, let's say, RestAssuredTest as shown below.



Once clicked on the ***Java Project*** link, a new project window opens up. Enter the name of the project as ***RestAussuredTest*** and click on the ***Finish*** button. Once ***Finish*** is clicked, we get the following screen. Click Open Perspective. (*Alternatively, if we have already checked the option "****remember my decision****", then this step will not be seen*)



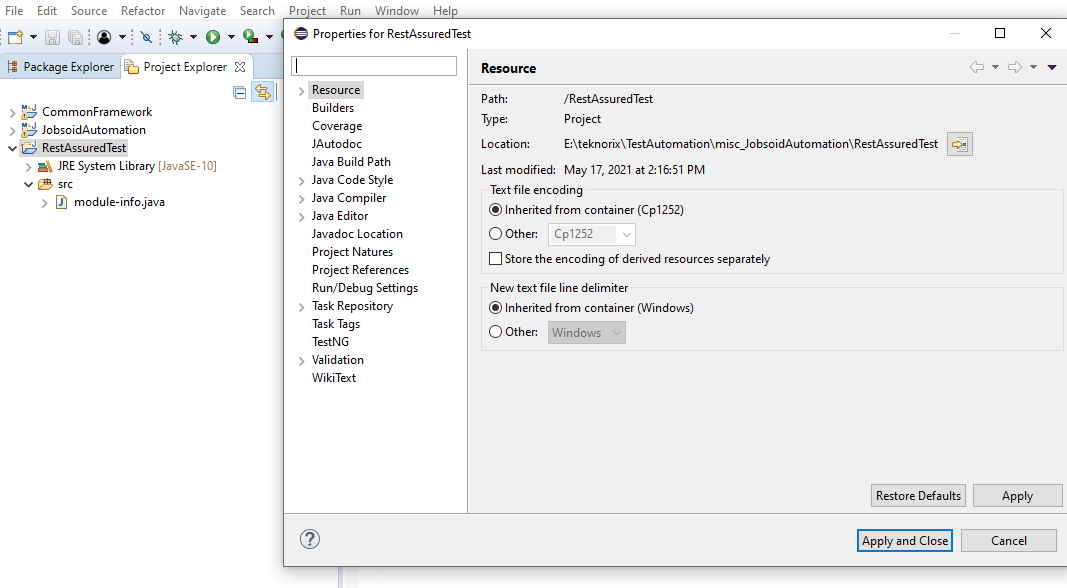
Now the project with the name RestAssuredTest should display in the package explorer.



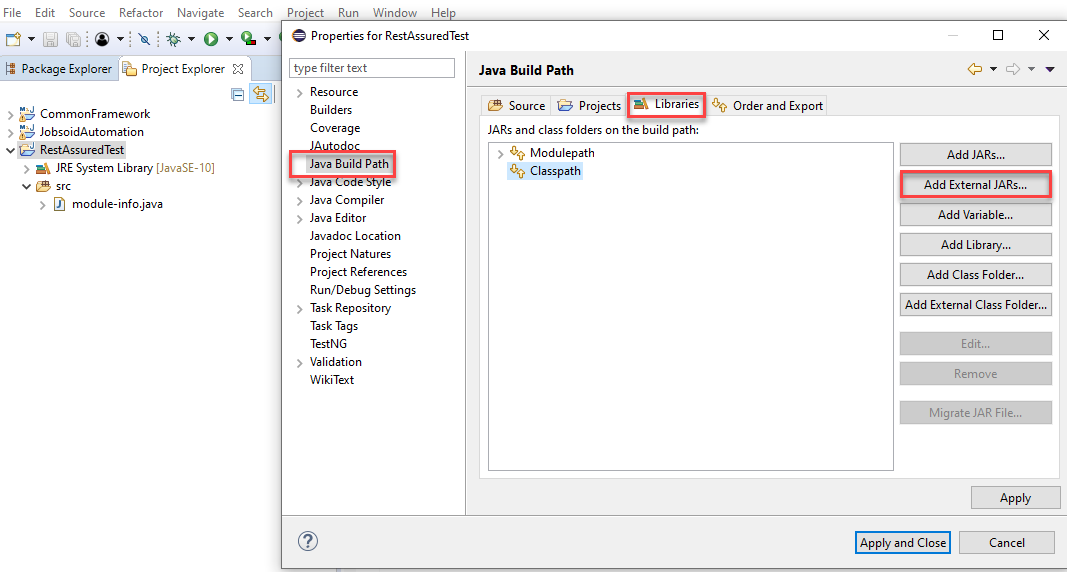
Once the project is created in Eclipse, now it is required to add the unzipped jars in the classpath of the Eclipse project.

**How to setup rest-assured jars in the class path?**

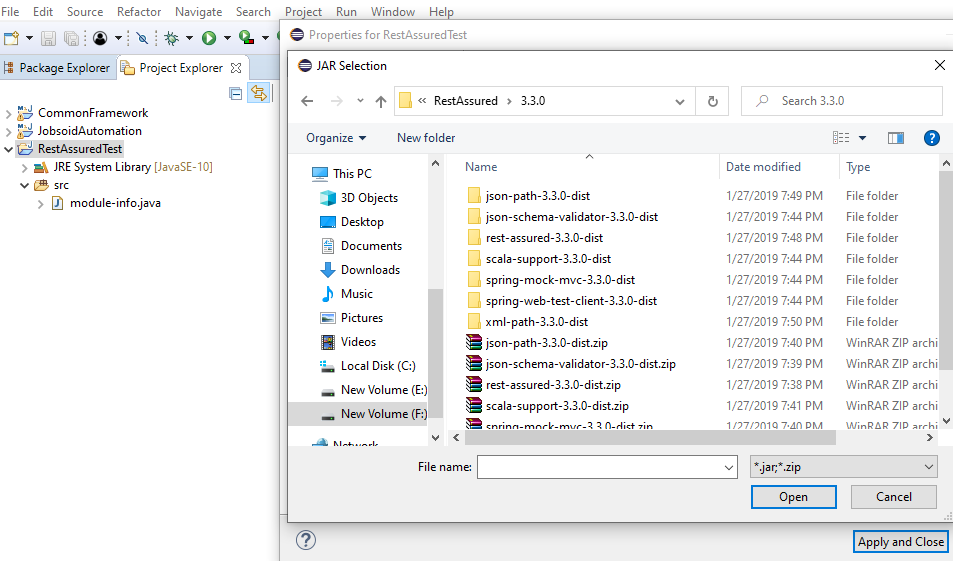
Select the Java project folder we just created and right-click on the project folder in the package explorer pane and choose ***Properties***. This will open up the project properties pop-up window as shown below.



Select  ***Java build path*** option in the left pane of the properties window. In the Java build path pane, on the left-hand side, you will see the ***Libraries*** pane.



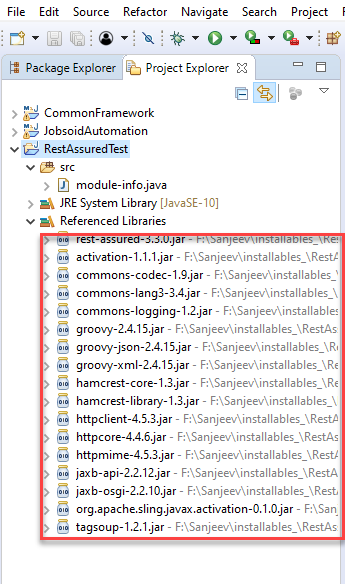
Here we are going to reference the rest assured jar files that we downloaded earlier. To reference the jars,  click on the "***Add external jars***" button and navigate to the folder where we unzipped all our rest-assured jars. Refer to the image shown below.



Now include the following Jars

* *rest-assured-3.3.0.jar*
* *All the jars in the folder rest-assured-3.3.0-deps*

With this, we have successfully set up a Java project in Eclipse with a rest-assured library. If we now check the project explorer, we will get the following view.



Now we are ready to use Rest Assured features in Eclipse.

**Key TakeAways**

In this article, we configured the Eclipse project with Rest Assured Library.

* *Rest Assured library comes in the form of jars which we can download freely from GitHub and unzip on our local machine.*
* *The prerequisites for using Rest Assured jars are we should have Java, Eclipse, and TestNG properly configured and running on the machine.*
* *We can then create a Java project in Eclipse and associate the Rest Assured jars with it by specifying the jar path in the Build path.*

With these steps, we can configure Rest Assured with the Eclipse project. In the next article, we will create a simple REST API with the Rest Assured library.

# Validate Response Status using Rest Assured

Category: [**Rest-Assured**](https://toolsqa.com/categories/rest-assured) , July 29 2022

0

0

9 min read

**Author**



Lakshay Sharma

**Reviewers**



Virender Singh

**[Previous Lesson](https://toolsqa.com/rest-assured/rest-api-test-using-rest-assured/)**

[REST API Testing using Rest Assured](https://toolsqa.com/rest-assured/rest-api-test-using-rest-assured/)

**[Next Lesson](https://toolsqa.com/rest-assured/validate-response-header-using-rest-assured/)**

[Validate Response Header using Rest Assured](https://toolsqa.com/rest-assured/validate-response-header-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. This status code value tells us if HTTP Response was successful or not. This article is a continuation of our previous article in which we performed a sample REST API test call. In this article, we will discuss how to validate the HTTP response status using REST Assured. The content of the post is reflected by the following index:

* Validate HTTP Response Status using Rest Assured
* How to validate HTTP response status code
* Validating the HTTP error status code?
* How to validate HTTP Response Status Line?

## Validate HTTP Response Status using Rest Assured

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

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

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

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

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

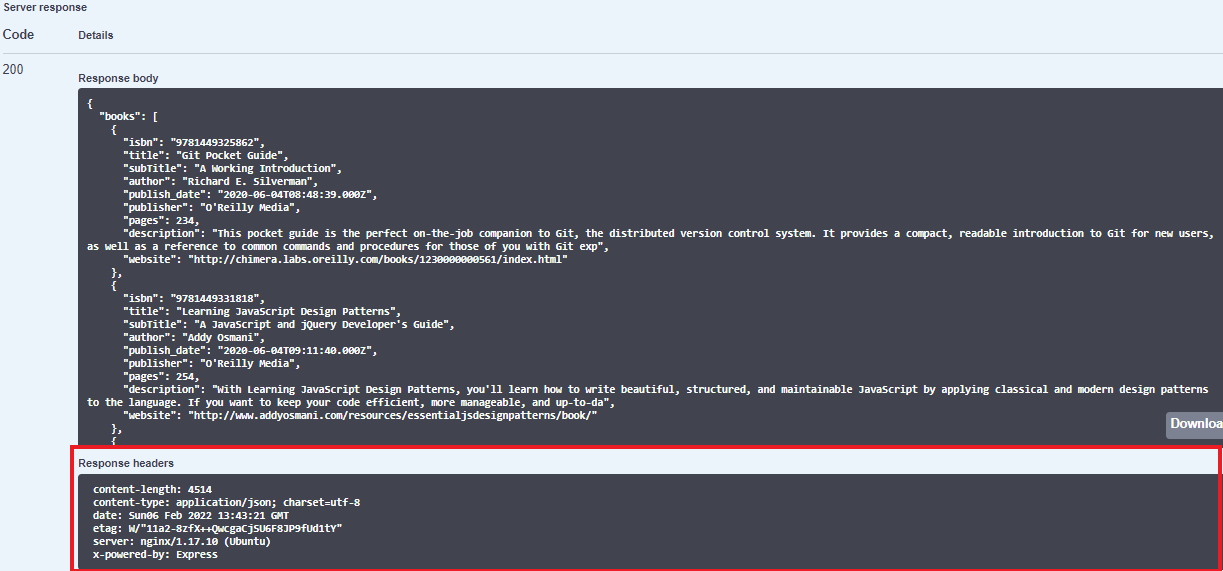
As we already know the same REST API returns a response message in XML or JSON format. This format depends on the ***Media-Type*** attribute in the HTTP request.

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

Consider the Swagger UI example we discussed in earlier articles. Suppose we send a GET request to the Book Store through our browser as follows:

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

When the above command executes we obtain the response shown in below screen:



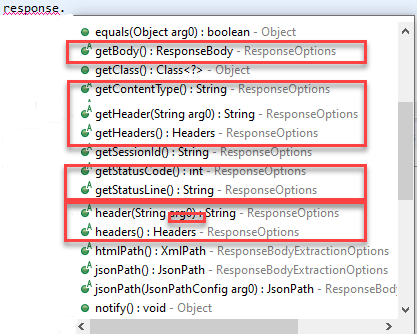
As seen from the above screenshot, the response has a status, headers, and a body. If we check the ***"Response headers"*** section, in the above screen, it has a content-type attribute that has the value along with other attributes. On validating this header, the client knows what type of response (body)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

Let us now proceed with validating the status part of the response.

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

When the client requests a piece of particular information from the server, the server sends a response with a status code back to the client. The status code that the server returns tells us whether the request was successful or not. 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. We can get the list of HTTP status codes along with their description on the [**W3 page.**](https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html)

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. The following screenshot shows some of the important methods of the ***response*** interface.



The method ***getStatusCode()*** is used to get the status code of the response. This method returns an integer and then we can verify its value. [***TestNG Assert***](https://www.original.toolsqa.com/testng-asserts/) is used to verify the Status Code. Now consider the code given 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()

{

// 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");

}

}

The below line of code extracts the status code from the message:

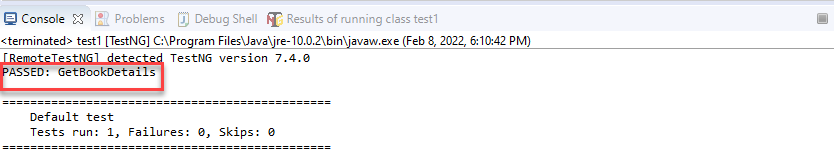
int statusCode = response.getStatusCode();

The return value ***"statusCode"*** is compared with the expected value i.e. 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");

If you run the above test, you will see that the test passes since the web service returns the status code as 200 as shown in the below image.



In this manner, we can verify the status code of the response using the ***"getStatusCode()"*** method of the response interface. Please note that since we already knew that the success code here is 200, we wrote the appropriate code. Your server may respond as success with a code anywhere between 200 and 299. It is best to check that out beforehand. Let us now move on to discuss how to validate the status code that returns values other than 200 i.e. error status code.

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

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

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

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

For the ToolsQA Book Store service, let’s create another test with an erroneous scenario. Here we will validate the HTTP status code returned by Book store Web Service when an invalid parameter is entered.

So here we provide the parameter to get user details. Here we provide nonexistent userId as the parameter. The code looks 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 GetPetDetails()

{

// Specify the base URL to the RESTful web service

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

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

RequestSpecification httpRequest = RestAssured.given();

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

// Get the status code of the request.

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

int statusCode = response.getStatusCode();

// Assert that correct status code is returned.

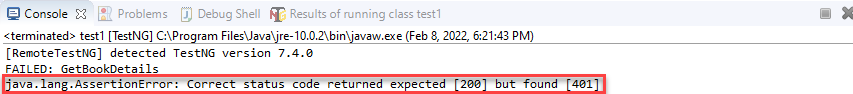
Assert.assertEquals(statusCode /\*actual value\*/, 200 /\*expected value\*/,

"Correct status code returned");

}

}

When we run this test it returns the error code of 401. We can see the result of the test execution below.



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

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

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

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

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

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

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

We can read the entire status line using the method getStatusLine () of the response interface. The following code shows the demonstration.

@Test

public void GetBookDetails()

{

// Specify the base URL to the RESTful web service

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

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

RequestSpecification httpRequest = RestAssured.given();

Response response = httpRequest.get("");

// Get the status line from the Response in a variable called statusLine

String statusLine = response.getStatusLine();

Assert.assertEquals(statusLine /\*actual value\*/, "HTTP/1.1 200 OK"

/\*expected value\*/, "Correct status code returned");

}

Here we perform a similar test as we have done for the status code. We read the status line using the getStatusLine () method into a string value. Then we compare this returned value with ***"HTTP/1.1 200 OK"*** to check if the status is successful.

Note: The postman article on the same topic (Response in Postman) can be found at [***Response in Postman.***](https://www.toolsqa.com/postman/response-in-postman/)

A video tutorial for Verifying the Status code is available at [***Verify Status Code in Rest Assured.***](https://www.youtube.com/watch?v=uD4sKHzWf8Q)

## Key TakeAways

In this article, we have discussed the HTTP status validation in REST Response.

1. The response obtained from the server consists of status, header, and body.
2. The status of the response in turn contains a status code and status string.
3. Rest Assured library provides a "Response" interface that provides numerous methods to extract the response fields.
4. We can read the status code using the getStatusCode() method. Similarly, we can read the status line using the getStatusLine () method of the response interface.
5. Once the status is read, then we can verify if the code is a success (200) or any other code.

# Validate Response Header using Rest Assured

Category: [**Rest-Assured**](https://toolsqa.com/categories/rest-assured) , February 7 2022

0

0

6 min read

**Author**



Shilpa Nadkarni

**Reviewers**



Lakshay Sharma

**[Previous Lesson](https://toolsqa.com/rest-assured/validate-response-status-using-rest-assured/)**

[Validate Response Status using Rest Assured](https://toolsqa.com/rest-assured/validate-response-status-using-rest-assured/)

**[Next Lesson](https://toolsqa.com/rest-assured/read-json-response-body-using-rest-assured/)**

[Read JSON Response Body using Rest Assured](https://toolsqa.com/rest-assured/read-json-response-body-using-rest-assured/)

REST assured has been of great use to us for testing APIs and validating a lot of things. For instance, in the last tutorials, we performed a sample [***Test call***](https://toolsqa.com/rest-assured/rest-api-test-using-rest-assured/) using Rest Assured and also [***validated response status***](https://toolsqa.com/rest-assured/validate-response-status-using-rest-assured/). But that is just one part of the overall response testing when it comes to automation. There is a lot more that we need to validate in order to be an efficient tester and test the quality of our APIs. In this article, let us validate the Response Header using Rest Assured with the following topics in focus:

* What is an HTTP Response Header in REST API?
* REST Assured methods for validating HTTP Response Headers.
* How to access and read HTTP Response headers using REST Assured?
* How to validate Response Header using Rest Assured.

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

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

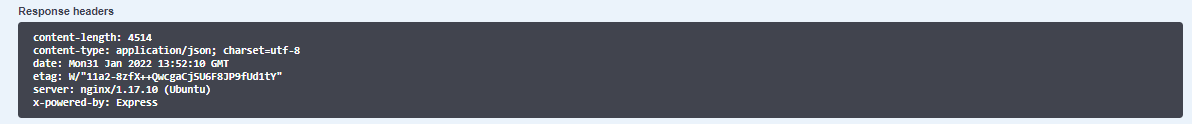
For example, headers contain a "***Content-Type***" attribute that tells us how to interpret the data of the response body. So if the response body contains JSON data, then the corresponding content-type attribute in the header will be "***application/json***". Similarly, if the data in the body is XML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

For example, for the following request URL: [***https://demoqa.com/BookStore/v1/Books***](https://demoqa.com/BookStore/v1/Books)

when we give the following GET request:

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

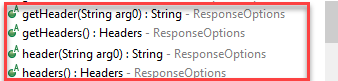
we get the following response.



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

## Rest Assured methods for validating HTTP Response Headers

The Response interface of the Rest Assured library provides methods to access all headers or individual headers. Just typing "***Response.head***" in Eclipse (or any such editor) will show all the methods supported to access headers.



As shown in the above screenshot, the Response interface of REST Assured provides methods related to headers.

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

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

Note: The collection of headers is represented by a class called ***io.restassured***. HTTP.Headers. This class implements the ***Iterable*** interface. Hence, we can use the "***for each (for( : ))***"  loop to read all the headers.

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

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

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

Shown below is the code for this test:

@Test

public void IteratingHeaders()

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

RequestSpecification httpRequest = RestAssured.given();

Response response = httpRequest.get("");

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

Headers allHeaders = response.headers();

// Iterate over all the Headers

for(Header header : allHeaders) {

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

}

}

In the above code, we access all headers and then extract individual headers by looping through the Headers collection. Shown below is the console output of the above test.



The above output shows all the header key-value pairs that we displayed using for loop in the code.

Let us demonstrate the ***.header***(String arg0) method to get a particular header. Here we pass the exact ***header name*** as an argument.

@Test

public void GetBookHeaders() {

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

RequestSpecification httpRequest = RestAssured.given();

Response response = httpRequest.get("");

// Access header with a given name.

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

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

// Access header with a given name.

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

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

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

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

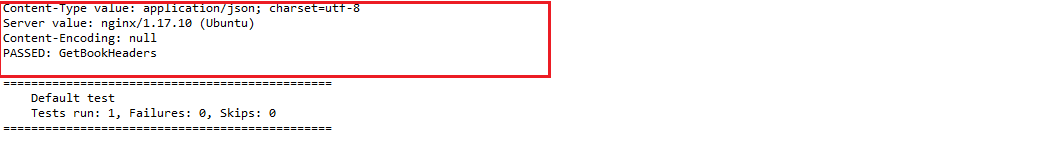
System.out.println("Content-Encoding: " + acceptLanguage);

}

}

}

In the above code, we are accessing ***Content-Type,  Server,*** and ***Content-Encoding*** headers by specifying respective headers names as an argument to the header() method. Shown below is the console output of the above code.



Note: Response.GetHeader(String headerName) method behaves exactly the same way as the Response.Header(String headerName) method. So the above code can be written by replacing the "***.Header()***" method with the "***.GetHeader()***" method.

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

Now that we have had a discussion about methods that access header, let us write a test to validate the values of the header by putting it as Assert. The below code shows the test that is similar to the code in previous sections but only uses Assert to validate the results.

@Test

public void ValidateBookHeaders()

{

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

RequestSpecification httpRequest = RestAssured.given();

Response response = httpRequest.get("");

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

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

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

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

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

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

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

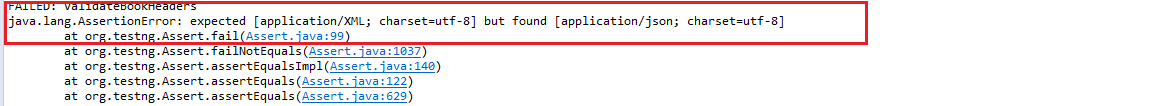


This output indicates that the actual value and expected value of each header matches.

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

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

If we run the same test above after the change, we will get the following output.



As we can see, the difference in actual value "***application/JSON***" and the expected value "***application/XML***" raises an assertion error.

In this way by using the header-specific methods of Response interface, we can validate the Response Header in Rest API.

Note: The video tutorial on the same topic can be found at  Verify [***Response Headers in Rest Assured***](https://www.youtube.com/watch?v=QASNbQFGxVM).

## Key TakeAways

In this article, we discussed the ways to validate the response header in REST using Rest Assured.

1. Every response obtained from the server may contain zero or more headers that provide metadata about the response.
2. Rest Assured provides various methods using which we can read individual or all headers sent by the server.
3. headers() and getHeaders() return a collection of headers that is iterable for individual entries. The method "***getHeader()***" and header() returns individual header whose name is specified as an argument.
4. Using these methods and the TestNG Assert we can then validate the headers by comparing the actual and expected values of each header.

After validating the status and headers of the response, we will move on to reading the actual Response body in the next article.

# Read JSON Response Body using Rest Assured

Category: [**Rest-Assured**](https://toolsqa.com/categories/rest-assured) , November 10 2021

0

0

6 min read

**Author**



Virender Singh

**Reviewers**



Lakshay Sharma

**[Previous Lesson](https://toolsqa.com/rest-assured/validate-response-header-using-rest-assured/)**

[Validate Response Header using Rest Assured](https://toolsqa.com/rest-assured/validate-response-header-using-rest-assured/)

**[Next Lesson](https://toolsqa.com/rest-assured/query-parameters-in-rest-assured/)**

[Working with Query Parameters in Rest Assured | REST API](https://toolsqa.com/rest-assured/query-parameters-in-rest-assured/)

In the last two tutorials we have learnt about Response Status Code, Status line and Headers. We will continue with the same example in those tutorials and verify the body if the Response. If you have not gone through the first two tutorials then I would suggest you go through these links

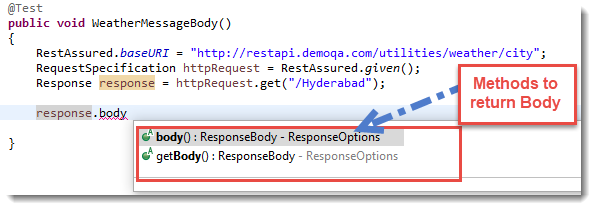
* [***First Test with Rest-Assured***](https://toolsqa.com/rest-assured/rest-api-test-using-rest-assured/)
* [***Validate Response Status code and Status Line using Rest-Assured***](https://toolsqa.com/rest-assured/validate-response-status-using-rest-assured/)
* [***Validate Response Header using Rest-Assured***](https://toolsqa.com/rest-assured/validate-response-header-using-rest-assured/)

In this tutorial, we will learn about ***How to Read JSON Response Body using Rest Assured?*** and ***How to Validate Content of a Response Body?***

## Read JSON Response Body using Rest Assured

Let us continue with the example of Weather web service that we used in the previous tutorials. When we request for the Weather details of a particular city, Server responds by sending the Weather details of the city as the Response Body. Response interface contains  two methods to get the Response Body

* ***Response.body() : returns ResponseBody***
* ***Response.getBody() : returns ResponseBody***



Using these methods we can get an Object of type ***io.restassured.response.ResponseBody***. This class represents the Body of a received Response. Using this class you can get and validate complete or parts of the Response Body. In the below code we will simply read the complete ***Response Body*** by using ***Response.getBody()*** and will print it out on the console window.

@Test

public void WeatherMessageBody()

{

RestAssured.baseURI = "https://restapi.demoqa.com/utilities/weather/city";

RequestSpecification httpRequest = RestAssured.given();

Response response = httpRequest.get("/Hyderabad");

// Retrieve the body of the Response

ResponseBody body = response.getBody();

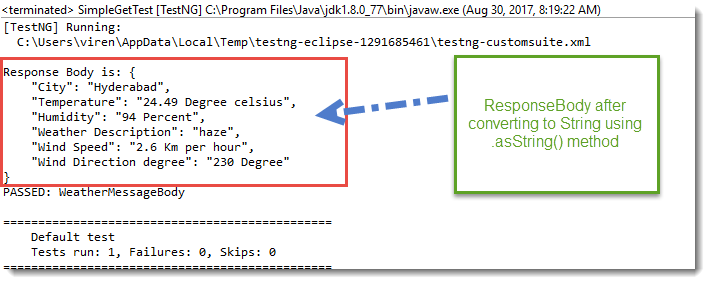
// By using the ResponseBody.asString() method, we can convert the body

// into the string representation.

System.out.println("Response Body is: " + body.asString());

}

***ResponseBody*** interface also has a method called ***.asString()***, as used in the above code, which converts a ***ResponseBody*** into its String representation. If you run this test the output will look something like this:



***Note: Response.body()*** method does exactly the same thing. So you can even use ***.body()*** method in the above code.

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

***ResponseBody*** can return the response body in a String format. We can use simple String methods to verify certain basic level of values in the Response. For e.g. we can use the ***String.contains()*** method to see if the Response contains a "***Hyderabad***" in it. The below code shows how to check for sub string presence.

@Test

public void WeatherMessageBody()

{

RestAssured.baseURI = "https://restapi.demoqa.com/utilities/weather/city";

RequestSpecification httpRequest = RestAssured.given();

Response response = httpRequest.get("/Hyderabad");

// Retrieve the body of the Response

ResponseBody body = response.getBody();

// To check for sub string presence get the Response body as a String.

// Do a String.contains

String bodyAsString = body.asString();

Assert.assertEquals(bodyAsString.contains("Hyderabad") /\*Expected value\*/, true /\*Actual Value\*/, "Response body contains Hyderabad");

}

***Check String presence by ignoring alphabet casing***

We can also ignore the casing using the String internal methods. To do this we will convert the Response in lower case and then compare it with our lower case string value. Below code demonstrates that.

@Test

public void WeatherMessageBody()

{

RestAssured.baseURI = "https://restapi.demoqa.com/utilities/weather/city";

RequestSpecification httpRequest = RestAssured.given();

Response response = httpRequest.get("/Hyderabad");

// Retrieve the body of the Response

ResponseBody body = response.getBody();

// To check for sub string presence get the Response body as a String.

// Do a String.contains

String bodyAsString = body.asString();

// convert the body into lower case and then do a comparison to ignore casing.

Assert.assertEquals(bodyAsString.toLowerCase().contains("hyderabad") /\*Expected value\*/, true /\*Actual Value\*/, "Response body contains Hyderabad");

}

The above two approaches suffer from a classical problem, what if the string "***Hyderabad***" is present in a wrong node or may be multiple instances of the same string are present. This is not a fool proof way of testing a particular node in the ***Response***. There are better ways, ***Response*** interface gives you a mechanism to extract nodes based on a given ***JsonPath***. There is a method called ***Response.JsonPath()***, which returns a ***io.restassured.path.json.JsonPath*** Object. This object can be used to further query specific parts of the ***Response Json***.

If you are not aware of JsonPath, please go through these tutorials

* [***What is JsonPath***](https://toolsqa.com/rest-assured/jsonpath-and-query-json-using-jsonpath/)
* [***JsonPath expressions***](https://toolsqa.com/rest-assured/expressions-in-jsonpath/)

### *****How to Extract a Node text from Response using JsonPath?*****

Let us continue with the above example and retrieve the ***City*** from the ***Response***. To do so, we will simply get the JsonPath object from the Response interface and then query for the particular node. Just to be very clear, let us look at the Weather API response again.

{

"City": "Hyderabad",

"Temperature": "25.51 Degree celsius",

"Humidity": "94 Percent",

"Weather Description": "mist",

"Wind Speed": "1 Km per hour",

"Wind Direction degree": " Degree"

}

In this response, if we want to go to the City node, all we have to do is have the following ***JsonPath: $.City***. Try it out on the [***JsonPath Evaluator***](https://jsonpath.com/) to verify the output.

Now let us look at the code, pay specific attention to the comments in the code.

***Note***: In Java ***JsonPath*** you do not need to have $ as the root node. You can completely skip that.

@Test

public void VerifyCityInJsonResponse()

{

RestAssured.baseURI = "https://restapi.demoqa.com/utilities/weather/city";

RequestSpecification httpRequest = RestAssured.given();

Response response = httpRequest.get("/Hyderabad");

// First get the JsonPath object instance from the Response interface

JsonPath jsonPathEvaluator = response.jsonPath();

// Then simply query the JsonPath object to get a String value of the node

// specified by JsonPath: City (Note: You should not put $. in the Java code)

String city = jsonPathEvaluator.get("City");

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

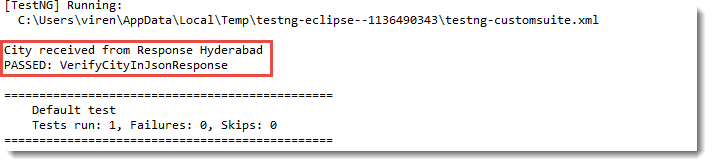
System.out.println("City received from Response " + city);

// Validate the response

Assert.assertEquals(city, "Hyderabad", "Correct city name received in the Response");

}

The output of the code passes the assertion and it also prints the ***City*** name retrieved from the Response. As shown in the image below



On the similar lines, you can extract any part of the **Json** response using the ***JsonPath*** implementation of ***Rest-Assured***. This is very convenient, compact and easy way to write tests.

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

Now that we know how to read a node using JsonPath, here is a small piece of code that reads all the nodes and prints them to the Console.

@Test

public void DisplayAllNodesInWeatherAPI()

{

RestAssured.baseURI = "https://restapi.demoqa.com/utilities/weather/city";

RequestSpecification httpRequest = RestAssured.given();

Response response = httpRequest.get("/Hyderabad");

// First get the JsonPath object instance from the Response interface

JsonPath jsonPathEvaluator = response.jsonPath();

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

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

// Print the temperature node

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

// Print the humidity node

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

// Print weather description

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

// Print Wind Speed

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

// Print Wind Direction Degree

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

}

In the next chapter, we will study about XmlPath usage on XML Responses.

# Working with Query Parameters in Rest Assured | REST API

Category: [**Rest-Assured**](https://toolsqa.com/categories/rest-assured) , January 29 2022

0

0

5 min read

**Author**



Gunjan Kaushik

**Reviewers**



Lakshay Sharma

**[Previous Lesson](https://toolsqa.com/rest-assured/read-json-response-body-using-rest-assured/)**

[Read JSON Response Body using Rest Assured](https://toolsqa.com/rest-assured/read-json-response-body-using-rest-assured/)

**[Next Lesson](https://toolsqa.com/rest-assured/post-request-using-rest-assured/)**

[Understanding HTTP POST Request Method using Rest Assured](https://toolsqa.com/rest-assured/post-request-using-rest-assured/)

So far in this series, we have covered [***reading the response body***](https://www.toolsqa.com/rest-assured/read-json-response-body-using-rest-assured/). But as you proceed ahead, a simple GET or a POST request won't suffice to the complexity your test scenario might have. You may need to test the API for different query parameters. In this article, we will cover how query parameters are passed in the HTTP request using rest assured with the following main sections-

* What is the composition of the URL?
* What are Query Parameters?
* How to send a request using Query Parameters in Rest Assured?

## What is the composition of the URL?

Similar to the examples used in the previous articles we will be using the [***Bookstore API***](https://demoqa.com/swagger/) in this article as well. Consider the below sample URL to fetch details corresponding to a book-

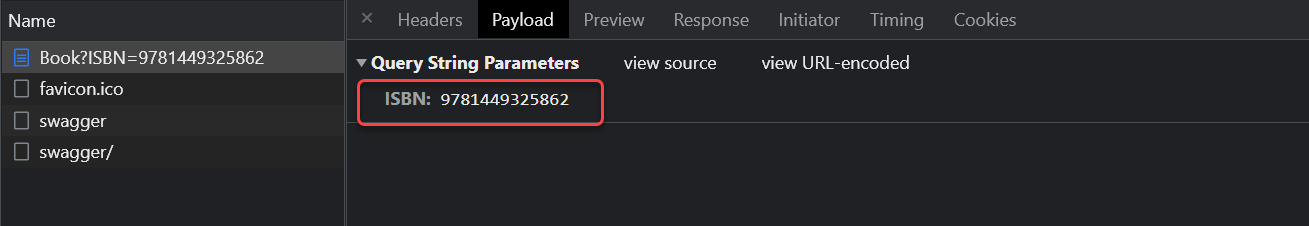
[***https://demoqa.com/BookStore/v1/Book?ISBN=9781449325862***](https://demoqa.com/BookStore/v1/Book?ISBN=9781449325862) Though the above URL looks very basic, it still let us understand its different components.

* ***https://*** - It is a protocol that ensures secure connection between the web server and the web browser.
* ***https://demoqa.com/swagger/#/BookStore*** - It is the domain name that hosts the website. It generally ends with .com, .in, .net, etc.
* ***BookStore/v1/Book*** - This is the path or the URI that identifies the resource applied by the request.
* ***?ISBN=9781449325862*** - This is a string query parameter. The question mark denotes the start of a query string parameter. There can be one or more query parameters in the URL.

Let us now understand more about query parameters in the next section.

## What are Query String Parameters?

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



The query parameter, i.e., ISBN=9781449325862, is displayed under the Query String Parameters field. In a similar way, if there are multiple query parameters passed in the URL, you will see all of them in this field. As an exercise, you may search some keywords on Google search and see the query parameters that get appended to the URL. Observe the search keyword is displayed after 'q=your search keyword'. Additionally, you will observe other query parameters which are separated from each other through an '&' sign.

These signs are termed URL parameters and are beyond the scope of our discussion here.

Now that we have understood what query parameters are and how they are used in requests, we can proceed to see how we can send requests with Query Parameters in Rest Assured.

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

In this section, we will see how we can automate our test case by passing query parameters in Rest assured. We will simply use the URL containing the query parameter as discussed in sections above to show how we can send a request using query parameters in rest assured. Let us see how the code would look like and then we will walkthrough it step by step.

package bookstore;

import org.junit.Test;

import io.restassured.RestAssured;

import io.restassured.path.json.JsonPath;

import io.restassured.response.Response;

import io.restassured.response.ResponseBody;

import io.restassured.specification.RequestSpecification;

public class QueryParam {

@Test

public void queryParameter() {

//Defining the base URI

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

RequestSpecification httpRequest = RestAssured.given();

//Passing the resource details

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

//Retrieving the response body using getBody() method

ResponseBody body = res.body();

//Converting the response body to string object

String rbdy = body.asString();

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

JsonPath jpath = new JsonPath(rbdy);

//Storing publisher name in a string variable

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

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

}

}

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

RequestSpecification httpRequest = RestAssured.given();

We first define the base URI to create a request to the service endpoint.

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

Next we send the resource details like the book ISBN as a query parameter to search in books using the GET request.

***Note***: If you need to send multiple query parameters you simply need to append queryParam() method with the parameter name and value to the RequestSpecification object, i.e. httpRequest in the above case.

ResponseBody body = res.body();

String rbdy = body.asString();

JsonPath jpath = new JsonPath(rbdy);

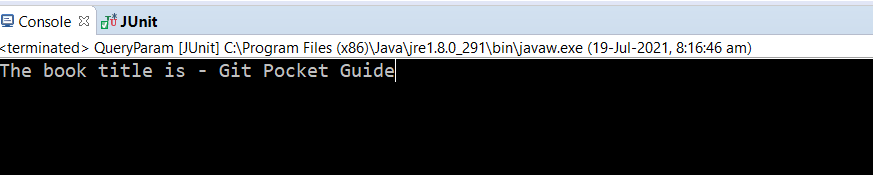
String title = jpath.getString("title");

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

Finally, we are storing the response body as a String object and parsing its different values using the JSONPath object. We then fetch the title from the response body.

***Note***: You may read more about [***JSONPath***](https://www.toolsqa.com/rest-assured/jsonpath-and-query-json-using-jsonpath/) in our elaborate article on the same.

And that's it! See how easily we could send the query parameters to our request by simply using a queryParam() method. The above code execution displays the title of the book with respect to the ISBN.



You can now go ahead with automating your test scripts with request having query parameter(s) and enhance your rest assured code.

## Key Takeaways

* We have seen the different components of a URL and the purpose of each component.
* We also understood what are query parameters and how they work when we access a URL in the browser.
* Query parameters passed to the rest assured tests using the queryParam() method accepts the parameter name and value.
* You may use the queryParam() method not just once, but as many times as the number of query parameters in your GET request.

# Understanding HTTP POST Request Method using Rest Assured

Category: [**Rest-Assured**](https://toolsqa.com/categories/rest-assured) , February 7 2022

0

0

8 min read

**Author**



Shilpa Nadkarni

**Reviewers**



Lakshay Sharma

**[Previous Lesson](https://toolsqa.com/rest-assured/query-parameters-in-rest-assured/)**

[Working with Query Parameters in Rest Assured | REST API](https://toolsqa.com/rest-assured/query-parameters-in-rest-assured/)

**[Next Lesson](https://toolsqa.com/rest-assured/serialization-and-deserialization-in-java/)**

[Serialization and Deserialization in Java](https://toolsqa.com/rest-assured/serialization-and-deserialization-in-java/)

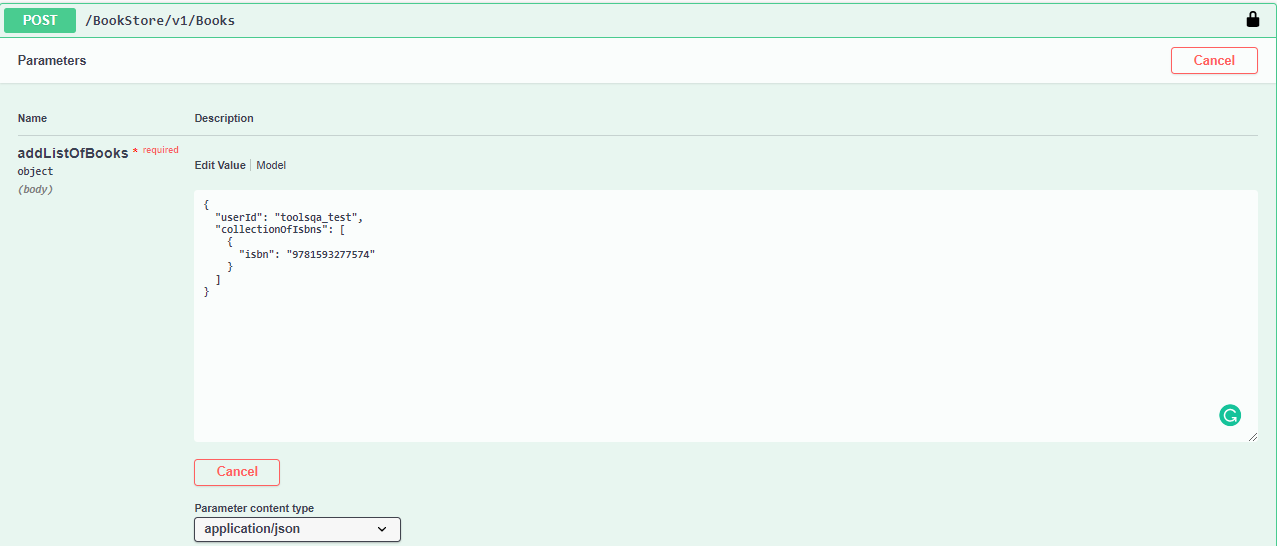
In our discussions of HTTP REST Methods, we went through the [***GET request***](https://toolsqa.com/rest-assured/rest-api-test-using-rest-assured/) in our earlier tutorials. We have also discussed validations like validating [***headers***](https://toolsqa.com/rest-assured/validate-response-header-using-rest-assured/) and the [***status***](https://toolsqa.com/rest-assured/validate-response-status-using-rest-assured/) of the response obtained from the server. In this article, we'll discuss the next method in REST API i.e. the POST request using Rest Assured library. We'll cover the following topics in this article.

* What is an HTTP POST request method?
* How to use POST request method 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 Response.
* Changing the HTTP Method on a POST Request using Rest Assured.

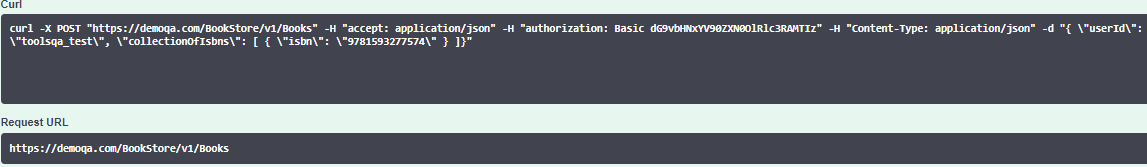
## What is an HTTP Post Request method?

We use the verb "***Post***" everywhere when we are dealing with the web. For example, when we are submitting any registration form on a particular webpage like Gmail. We provide the required data and click submit. So through this action of submitting data, we are actually POSTING or sending the data to the server. The verb "***POST***" in HTTP, also called as POST request method in HTTP sends data to the server. In our example, we are sending our names, email and decided password as data.

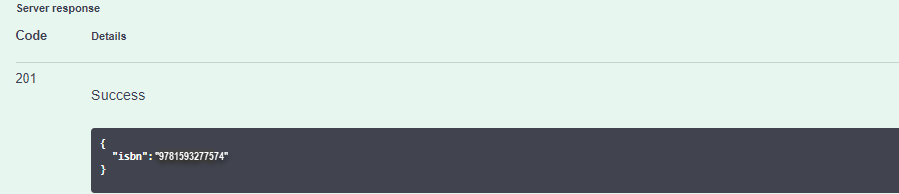
Now let's go to Swagger UI of API [***https://demoqa.com/BookStore/v1/Books***](https://demoqa.com/BookStore/v1/Books) Next, let's create a new user using POST request as shown in the following screenshot.



In the above example, we have provided all the details for the new user including id and ISBN. When we "***execute***" requests we initiate a POST request and send all the information to the server. On the server side, the new user will be created with all the provided information and a response will be sent back to the client. Look at the following screenshot.



Have a look at the "***curl***" command that has a POST request with all the data. The operation is successful and hence we get the response code as 201.



The content-type parameter is "***application/JSON***" in the above example. The data sent to the server by POST request can also be XML(content-type: application/XML). This data is in the body of the HTTP request. As we learned from the above example, the POST method usually creates a new record in the application database. Although, it is not always true. Sometimes in creating web applications we use POST request redirections to ensure that no one hits that URL directly without proper navigation channels. For example, you must have noticed that refreshing or pressing "***back***" on payment pages does not work directly. It is secure as the data being sent is not visible in the URL string. The data can also be encrypted using HTTPS (HTTP Secure) to enhance security.

As far as real-world applications are concerned, the POST request can have a very big size and also a complex body structure. Since it provides an extra layer of security, we often use it for passing sensitive business-related data. It is also noteworthy here 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 you should always go through the documentation before.

Let us now use Rest Assured to simulate an HTTP POST request.

## How to use POST Request Method using Rest Assured?

***In Rest Assured, we make use of the post() method to make an HTTP POST request.***

For making an HTTP Post request using Rest Assured, let's add a simple JSON library in our classpath so that we can create JSON objects in the code. We can use the following URL to download simple JSON from the Maven: [***https://mvnrepository.com/artifact/com.googlecode.json-simple/json-simple***](https://mvnrepository.com/artifact/com.googlecode.json-simple/json-simple). Once the jar is downloaded we can add it to the classpath.

Following are the steps we'll follow to make a POST Request using Rest Assured.

1. ***Create a Request pointing to the service Endpoint.***
2. ***Create a JSON Request which contains all the fields.***
3. ***Add JSON body in the request and send the request.***
4. ***Validate the Request.***
5. ***Changing the HTTP Method on a POST Request.***

Let us now perform each step one by one.

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

We can start coding once the JSON jar downloads.

Consider following code lines.

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

RequestSpecification request = RestAssured.given();

In the above code, we initialize a base URI with a link to the bookstore and the 'createUser' API. Next, we create a 'request' using RequestSpecification.

Note: Above code is self-explanatory. In case of any problems, please refer to previous tutorials.

### *****Create a JSON request which contains all the fields*****

Next, we will create a JSON request object that will contain the data we need to create a new user. Given below is the code for the same:

// 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");

In the above code, we have created a ***JSONObject*** variable (JSONObject belongs to org.json.simple package). Here we provide a JSON String containing the data to post to the server.  Here we have the ***requestParams*** **object** above for our test web service with multiple nodes in the ***JSON***. Each node is added to the JSON string using the ***JSONObject.put***(String, String) method. Once all the nodes are added in this manner, we get the String representation of ***JSONObject*** by calling JSONObject.toJSONString() method.

### *****Add JSON body in the request and send the Request*****

Now that we have created the JSON string with the required data, the next step will be to add this JSON to the request body and send or post the request. Look at the following code:

// 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

So in this step, we simply add the ***JSON*** String to the body of the ***HTTP Request*** and set the ***Content-Type*** header field value to ***application/JSON***. Next, we use the method ***RequestSpecification.body***(JsonString) to put the JSON body into the request. Using this method we can update the content of the ***HTTP Request*** Body.

Next using the post () method on the request object we send this data to the server using the 'BookStoreV1BooksPost' API.

Note: Calling the ***RequestSpecification.body*** method multiple times updates the body each time to the latest JSON String.

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

After posting the request we have to validate the response we received from the server as a result of a POST request. Given below is the code for validating the response:

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

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

Here we read the status obtained using the statusLine() method and print it to the console.

## *****Changing the HTTP Method on a POST Request using Rest Assured*****

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

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

public void UserRegistrationSuccessful()

{

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

RequestSpecification request = RestAssured.given();

JSONObject requestParams = new JSONObject();

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

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

request.body(requestParams.toJSONString());

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

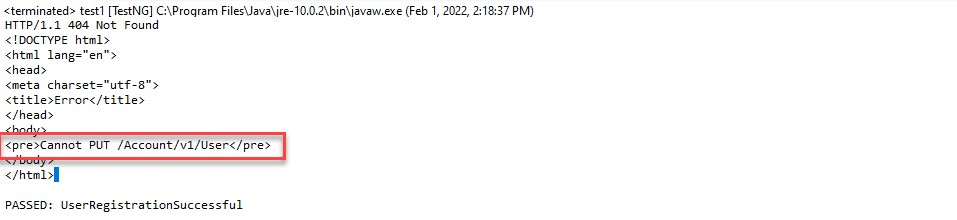
ResponseBody body = response.getBody();

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

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

}

When we execute the above code, we get the following response.



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

* Sending incomplete POST Data.
* Try Sending JSON data with incorrect syntax.
* Sending incorrect HTTP request method in the Request.

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

***Note***: Corresponding Postman tutorial for Post request can be found at [***Response in Postman***](https://www.toolsqa.com/postman/response-in-postman/).

The video tutorial for this topic is available at [***What is Post Request?***](https://www.youtube.com/watch?v=YHWngWsmcWg)

## Key TakeAways

In this article, we discussed the HTTP POST request method using Rest Assured.

* Post request is used to send or post the data to the server.
* Post request mostly results in creating a new record in the database. It can also update the existing record in the database.
* Rest Assured uses a post () method to make HTTP POST requests.
* We usually send the JSON data along with the request object and then POST it to the server.

In the next tutorial, we'll cover the Deserialization technique to convert the JSON Response body into a Class instance.

# Serialization and Deserialization in Java

Category: [**Rest-Assured**](https://toolsqa.com/categories/rest-assured) , July 7 2021

0

0

7 min read

**Author**



Virender Singh

**Reviewers**



Lakshay Sharma

**[Previous Lesson](https://toolsqa.com/rest-assured/post-request-using-rest-assured/)**

[Understanding HTTP POST Request Method using Rest Assured](https://toolsqa.com/rest-assured/post-request-using-rest-assured/)

**[Next Lesson](https://toolsqa.com/rest-assured/deserialize-json-response/)**

[Deserialize JSON Response using Rest Assured](https://toolsqa.com/rest-assured/deserialize-json-response/)

## Serialization and Deserialization in Java

***Serialization*** and ***Deserialization in Java*** is an important programming concept. It is applicable to all major programming languages. In this chapter, we will try to understand this concept in the context of Java language. At the end of this chapter, we will be able to

* ***Conceptually know meaning of Serialization and Deserialization***
* ***Understand the Java code to Serialization and Deserialization***

Let us first define the two terms in Generic form, irrespective of any programming language.

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

Serialization is a process where you convert an Instance of a Class (Object of a class) into a Byte Stream. This Byte Stream can then be stored as a file on the disk or can also be sent to another computer via the network. Serialization can also be used to save the sate of Object when the program shuts down or hibernates. Once the state is saved on disk using Serialization, we can restore the state by DeSerializing the class from disk.

Let us try to visualize this using a small diagram. In this diagram, we will create a small class called Rectangle, which represents a real-life rectangle. Here is the code for the class

public class Rectangle {

private double height;

private double width;

public Rectangle(double height, double width)

{

this.height = height;

this.width = width;

}

public double Area()

{

return height \* width;

}

public double Perimeter()

{

return 2 \* (height + width);

}

}

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

Serialization process on the Rectangle class will look like this.



The encoding scheme that is used to convert from an Object of Class Rectangle to Byte stream is governed by the Serialization encoding standards mentioned [***here***](https://docs.oracle.com/javase/8/docs/technotes/guides/serialization/index.html).

### *****Serializable Interface*****

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

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

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

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

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

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

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

public class Rectangle implements Serializable{

private double height;

private double width;

public Rectangle(double height, double width)

{

this.height = height;

this.width = width;

}

public double Area()

{

return height \* width;

}

public double Perimeter()

{

return 2 \* (height + width);

}

}

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

Let us quickly take a look at the Serialization process in Java. In this process, we will perform these four steps

1. We will create a new ***FileOutputStream*** of a file where we want to serialize the class
2. We will then ***ObjectOutputStream*** on the ***FileOutputStream*** created in step 1
3. We will then write the object into the ***ObjectOutputStream***
4. Finally, we will close all the stream objects to save properly write and terminate all streams.

Below is the code to perform the task. Pay attention to the comments that are mentioned above every line of code.

public static void SerializeToFile(Object classObject, String fileName)

{

try {

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

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

FileOutputStream fileStream = new FileOutputStream(fileName);

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

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

// a byte stream

ObjectOutputStream objectStream = new ObjectOutputStream(fileStream);

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

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

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

objectStream.writeObject(classObject);

// Step 4: Gracefully close the streams

objectStream.close();

fileStream.close();

} catch (FileNotFoundException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} catch (IOException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

public static void main(String[] args)

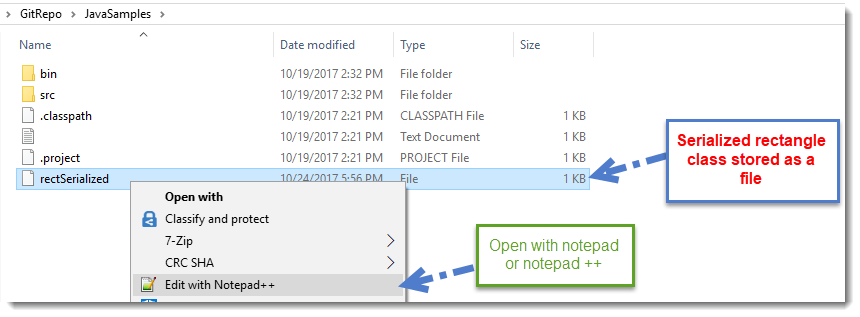
{

Rectangle rect = new Rectangle(18, 78);

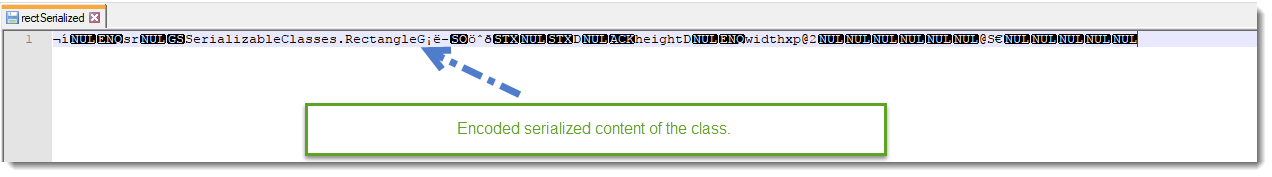
SerializeToFile(rect, "rectSerialized");

}

When we run this program, it creates a file named ***"rectSerialized"***  in the root folder of the project. Just browse to that location and try to open the file using a notepad. Below image shows that



When you open this file you can see that it contains garbled characters. The contents are encoded and are not in human readable format. As shown in the image below.



This shows what exactly serialization means and how we can serialize an object in Java. This should also give you a practical understanding of the different steps involved in serialization process including the input and output results of the different steps.

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

In the previous section, we learnt the Serialization converts a class instance into a byte stream which is then stored in a file on disk. Let us quickly take a look at the Deserialization process in Java, which is opposite of Serialization. In this process, we will read the Serialized byte stream from the file and convert it back into the Class instance representation. Here are the steps that we will follow.

1. We will create a new ***FileInputStream*** to read the file which contains the serialized byte stream of the target class. Rectangle class in our case.
2. We will then create an ***ObjectInputStream*** on the ***FileInputStream*** created in step 1
3. We will then read the object using ***ObjectInputStream*** and store it in a variable of type Rectangle.
4. Finally, we will close all the stream objects to save properly write and terminate all streams.

Below is the code to perform the task. Pay attention to the comments that are mentioned above every line of code.

public static Object DeSerializeFromFileToObject(String fileName)

{

try {

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

// of rectangle class from the file

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

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

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

ObjectInputStream objectStream = new ObjectInputStream(fileStream);

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

Object deserializeObject = objectStream.readObject();

// Step 4: Close all the resources

objectStream.close();

fileStream.close();

// return the deserialized object

return deserializeObject;

} catch (FileNotFoundException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} catch (IOException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} catch (ClassNotFoundException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

return null;

}

public static void main(String[] args)

{

Rectangle rect = new Rectangle(18, 78);

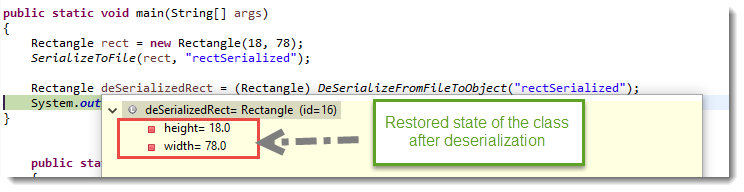
SerializeToFile(rect, "rectSerialized");

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

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

}

Just to verify that the original state of the Rectangle class is restored, we will debug the code and inspect ***deSerializedRect***  variable. The below image shows that the original state of (Height: 18 and Width: 78) Rectangle class is restored.



I hope that this tutorial clears the basic concept of Serialization and Deserialization in Java for you.

# Deserialize JSON Response using Rest Assured

Category: [**Rest-Assured**](https://toolsqa.com/categories/rest-assured) , February 12 2022

0

0

9 min read

**Author**



Shilpa Nadkarni

**Reviewers**



Lakshay Sharma

**[Previous Lesson](https://toolsqa.com/rest-assured/serialization-and-deserialization-in-java/)**

[Serialization and Deserialization in Java](https://toolsqa.com/rest-assured/serialization-and-deserialization-in-java/)

**[Next Lesson](https://toolsqa.com/rest-assured/authentication-and-authorization-in-rest-webservices/)**

[Authentication and Authorization in REST WebServices](https://toolsqa.com/rest-assured/authentication-and-authorization-in-rest-webservices/)

JSON is an extremely popular format when it comes to APIs. Almost all of the APIs either transfer data in the XML format or JSON format of which JSON is a popular one. Not only through APIs, but the companies also use JSON to transfer data between their own server to UI because of its lightweight and easily readable features. As a professional tester or a developer, it will be rare if you do not come across a JSON file and verify its legibility. To work with them, you should know how to read them through code and make use of automation testing to accomplish your tasks easily. In this article, we'll discuss the same concept used in the body of the REST Response. This technique to read the JSON Response is also termed as ***to "Deserialize" the JSON Response.***

***Serialization*** and ***Deserialization*** are programming techniques where we convert ***Objects*** to ***Byte Streams*** and from ***Byte Streams*** back to ***Objects*** respectively. We have already discussed this technique in the article [***Serialization and Deserialization in Java.***](https://toolsqa.com/rest-assured/serialization-and-deserialization-in-java/) We will now cover the following topics in this article:

* What is Serializing a JSON?
* What is Deserializing of a JSON Response?
* How to Deserialize JSON Response to Class with Rest Assured?'
* How to Deserialize JSON Response Body based on Response Status?

## What is Serializing a JSON?

Serialization, as mentioned above, is a process of converting data objects into a stream of data. The reverse process of serialization (stream to object) is deserialization. Both the processes are platform-independent. This means we can serialize an object on a Windows platform and deserialize it on macOS.

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

For example, if we have the following object:

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

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

{

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

"api":"QA"

}

The above data string can be stored or transferred anywhere. The recipient will then deserialize the data back to its original format ie object.

Java POJO classes are one way we can serialize the objects and use them in Rest Assured. The details of implementation are beyond the scope of this tutorial. Serialization is discussed in our previous tutorial [***Serialization and Deserialization***](https://toolsqa.com/rest-assured/serialization-and-deserialization-in-java/) in Java. While working in the testing industry, we will be more concerned about deserialization. The same is discussed in the next section in detail.

## What is Deserializing a JSON Response?

Before going ahead, it is recommended to read the article [***REST Api testing in Rest-Assured***](https://toolsqa.com/rest-assured/rest-api-test-using-rest-assured/) and converse yourself with REST, API testing, REST Assured and other similar concepts.

In REST APIs, the process of deserialization is performed especially for the JSON responses we receive from the API.​ So, when we say we are deserializing the JSON, this means we convert the JSON format into a type we prefer, most frequently [***POJO***](https://en.wikipedia.org/wiki/Plain_old_Java_object) (Plain Old Java Object) classes.

So in this process, we essentially tell our code to use strongly typed strings that are less error-prone instead of JSON strings. Let's go ahead and see practically how to deserialize the JSON responses to class with Rest Assured.

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

We will continue from our example in [***Making a POST request using Rest-Assured***](https://toolsqa.com/rest-assured/post-request-using-rest-assured/) to implement deserialization of JSON responses. In this article, we have seen that a successful post request sends the following response:

{

"SuccessCode": "OPERATION\_SUCCESS",

"Message": "Operation completed successfully"

}

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

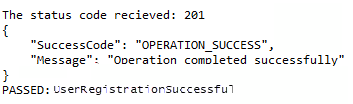
Note: Deserialization is also called ***Object Representation*** of structured data. The structured data here is JSON.

So how do we begin with deserialization?

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

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

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



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

@Test

public class JSONSuccessResponse {

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

// Variable SuccessCode will contain value of SuccessCode node

public String SuccessCode;

// Variable Message will contain the value of Message node

public String Message;

}

Now that we have a  ***JSONSuccessResponse*** class to store all the nodes present in Successful Response, let's understand how to use ***Rest-Assured*** to automatically convert JSON Response Body to the instance of the JSONSuccessResponse class.

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

The class ***io.restassured.response.ResponseBody*** that represents the response in Rest Assured, has a method called ***.as(Class<T>).*** Here, Class*<T>* is a generic form of any class of type T which is also referred to as template class. In this case, Class <T> will be JSONSuccessResponse.

Internally, the ***"as()"*** method performs two actions:

* Create an instance of ***JSONSuccessResponse*** class.
* Copy the JSON node variables to the respective instance variables of the class. For example, the ***value of SuccessCode*** node to the variable ***SucessCode*** and value of Message to variable Message.

Given below is the code demonstrating the ***Response.Body.as(Class<T>)*** method.

@Test

public void UserRegistrationSuccessful() {

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

RequestSpecification request = RestAssured.given();

JSONObject requestParams = new JSONObject();

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

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

request.body(requestParams.toJSONString());

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

ResponseBody body = response.getBody();

// Deserialize the Response body into JSONSuccessResponse

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

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

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

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

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

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

// Response.

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

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

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

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

In the previous section, we discussed how to deserialize the JSON in case of a successful response. But in real-time applications, we can also receive responses that are failures. In such a case, Rest API may return a completely different response body. One such format of failed response may be as shown below:

{

"FaultId": "User already exists",

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

}

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

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

public class JSONFailureResponse {

String FaultId;

String fault;

}

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

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

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

@Test

public void UserRegistrationSuccessful() {

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

RequestSpecification request = RestAssured.given();

JSONObject requestParams = new JSONObject();

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

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

request.body(requestParams.toJSONString());

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

ResponseBody body = response.getBody();

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

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

// Deserialize the Response body into JSONFailureResponse

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

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

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

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

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

// Deserialize the Response body into JSONSuccessResponse

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

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

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

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

}

}

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

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

## Key TakeAways

In this article, we have discussed the deserialization of JSON responses into a POJO class.

* Deserialization is the process of converting JSON responses into classes. By doing this we can convert JSON strings into strongly types strings that are less error-prone.
* For deserializing the responses,  we create a separate class that has the same variables that are present in the JSON response like StatusCode and Message.
* In the code, we can use this class object to read the JSON response as shown in the example above.
* Deserialization of multiple responses like success and failure depends on the status code. For this, we need two POJO classes one each for success and failure. Depending on the value of the Status code we can call instances of the appropriate class and deserialize the response.

In our next article, we will discuss authentication and authorization in Rest

# Authentication and Authorization in REST WebServices

Category: [**Rest-Assured**](https://toolsqa.com/categories/rest-assured) , February 9 2022

0

0

5 min read

**Author**



Virender Singh

**Reviewers**



Lakshay Sharma

**[Previous Lesson](https://toolsqa.com/rest-assured/deserialize-json-response/)**

[Deserialize JSON Response using Rest Assured](https://toolsqa.com/rest-assured/deserialize-json-response/)

**[Next Lesson](https://toolsqa.com/rest-assured/basic-auth/)**

[Basic Authentication](https://toolsqa.com/rest-assured/basic-auth/)

## Authentication and Authorization in REST WebServices

***Authentication and Authorization in REST WebServices*** are two very important concepts in the context of [***REST API***](https://toolsqa.com/rest-assured/what-is-rest/). The majority of the time you will be hitting REST API's which are secured. By secure, we mean that the APIs which require you to provide ***identification***. Identification can be provided in the form of

* Username and a Password
* Authentication tokens
* Secret keys
* Bio-metrics and many other ways

In the context of REST API, we will be more interested in the first three options. The Authentication and Authorization models that we will discuss are spread across multiple tutorials, starting from this tutorial.

You may also go through the recording of the Postman Tutorial where our experts have explained this concepts in depth.

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

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

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

In the context of REST API authentication happens using the HTTP Request.

***Note: Not just REST API, authentication on any application working via HTTP Protocol happens using the HTTP Request.***

### *****Basic Authentication Flow*****

Taking the example of email login, we know that in order to Authenticate our self we have to provide a username and a password. In a very basic Authentication flow using Username and Password, we will do the same thing in REST API call as well. but how do we send the Username and Password in the REST request?

A REST request can have a special header called ***Authorization Header,*** this header can contain the credentials (username and password) in some form. Once a request with Authorization Header is received, the server can validate the credentials and can let you access the private resources.

***Note: I hope from previous tutorials you are able to understand the meaning of a Resource. If not, please go through this tutorial:*** [***Rest architectural elements.***](https://toolsqa.com/rest-assured/rest-architectural-elements/) ***A private resource is one that is not accessible to everyone. You need to Authenticate yourself to access the private resource.  For e.g. the email inbox, you have to log in to see the emails.***

Let us see it with an example, we have created an API that needs a valid Username and Password to access the Resource.

***Endpoint: http://restapi.demoqa.com/authentication/CheckForAuthentication***

In the code below we will try to hit the URL and see what is the Response that we get.

@Test

public void AuthenticationBasics()

{

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

RequestSpecification request = RestAssured.given();

Response response = request.get();

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

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

}

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

Status code: 401

Status message:

{

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

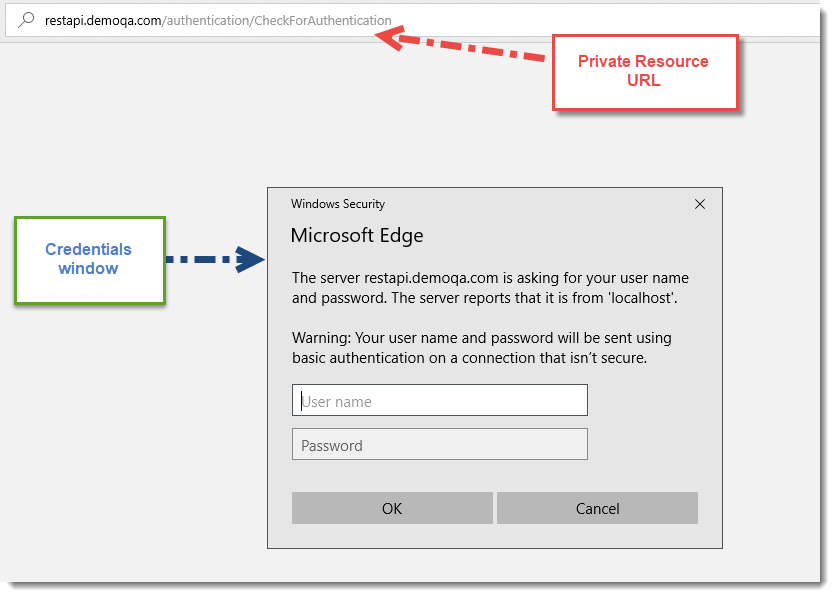
"Status": "Invalid or expired Authentication key provided"

}

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

***Note:*** Pay special attention to the ***Status code*** returned. In case of ***, Authentication*** failures Server should respond with a status code of 401 ***Unauthorized.***

Try to hit that URL using a browser. You should get a Username and Password prompt. The below image shows what you should be getting when you hit this URL from the browser.



In this tutorial, we will not discuss how to pass Authentication information in the Request header. Here we will only focus on the definitions of  ***Authentication*** and ***Authorization***. In the next set of tutorials, we will see different ***Authentication models,*** which will solve the above problem.

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

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

Before ending the tutorial let us see the contents of the private resource in the URL mentioned above. To do that enter the following credentials

* ***Username: ToolsQA***
* ***Password: TestPassword***

The server will be able to **Authenticate** and then **Authorize** you to access the private resource content. The below image shows the content after successful Authentication.



With this basic understanding of Authentication and Authorization, read the coming tutorials where we will discuss the specif types of ***Authentication*** models in ***REST API***.

# Basic Authentication

Category: [**Rest-Assured**](https://toolsqa.com/categories/rest-assured) , April 8 2022

0

0

8 min read

**Author**



Gunjan Kaushik

**[Previous Lesson](https://toolsqa.com/rest-assured/authentication-and-authorization-in-rest-webservices/)**

[Authentication and Authorization in REST WebServices](https://toolsqa.com/rest-assured/authentication-and-authorization-in-rest-webservices/)

**[Next Lesson](https://toolsqa.com/rest-assured/put-request-using-rest-assured/)**

[PUT Request using Rest Assured](https://toolsqa.com/rest-assured/put-request-using-rest-assured/)

In our series, we have so far covered the [***basics of Rest Assured***](https://www.toolsqa.com/rest-assured/rest-api-test-using-rest-assured/), the different types of requests like [***POST***](https://www.toolsqa.com/rest-assured/post-request-using-rest-assured/), [***PUT***](https://www.toolsqa.com/rest-assured/put-request-using-rest-assured/) and [***DELETE.***](https://www.toolsqa.com/rest-assured/delete-request-using-rest-assured/) In addition, we also covered the basics of [***Authentication & Authorization concepts of Rest API.***](https://www.toolsqa.com/rest-assured/authentication-and-authorization-in-rest-webservices/) Practically in the projects, as we proceed with automation, we come across complex APIs. These require basic authentication implementation mixed along with other code. In this article, we will cover the handling of basic authentication in Rest Assured. We will focus on the below points-

* What is Basic Authentication(Auth)?
* What are the different authentication schemes provided by Rest Assured?
  + Basic Authentication in Rest Assured.
    - Preemptive Authentication.
  + Digest Authentication.
  + Form Authentication.
  + OAuth Authentication.
    - OAuth1
    - OAuth2

## What is Basic Authentication (Auth)?

While going through the previous tutorials you must have noticed that we have used the username and the password (authentication credentials) for certain APIs. This credential setting is to enforce access control for the web resources and is generally passed in the header field of an HTTP request. The implementation of basic authentication is to ensure that the APIs are secured and only the users who are authorized have the access to view them. Hence, the authentication information is not encrypted or hashed but encoded as base-64. We will now see the different schemes used in Rest Assured for authentication and you may go through our previous article on [***Authentication and Authorization***](https://www.toolsqa.com/rest-assured/authentication-and-authorization-in-rest-webservices/) for more information.

## What are the different authentication schemes provided by Rest Assured?

To test and validate any secured API, you will have to use some authentication scheme. Rest Assured provides several authentication schemes which we are going to discuss in this part.

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

package org.example;

import org.testng.annotations.Test;

import io.restassured.RestAssured;

import io.restassured.response.Response;

import io.restassured.response.ResponseBody;

import io.restassured.specification.RequestSpecification;

public class BasicAuth {

@Test

public void getData() {

RequestSpecification httpRequest = RestAssured.given();

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

ResponseBody body = res.body();

//Converting the response body to string

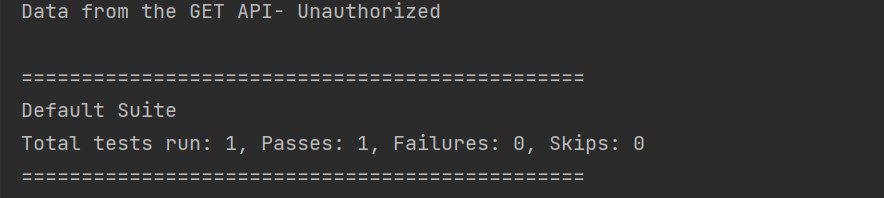
String rbdy = body.asString();

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

}

}

The code is pretty simple and uses the get () method to send requests to the server. Do not worry if you don't understand. It will be explained in the later examples. On executing this code the result would be-



Observe the message in the first line - ***"Data from the GET API - Unauthorized".*** This is the issue that we are going to fix using the basic authentication in our rest assured tests. Let us quickly jump on to understanding the same.

Basic Authentication in Rest Assured

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

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

In the given method you need to append the method of authentication specification followed by the basic HTTP auth where you will pass the credentials as the parameters. Another type of basic authentication is ***preemptive*** which we will discuss next.

package org.example;

import org.testng.annotations.Test;

import io.restassured.RestAssured;

import io.restassured.response.Response;

import io.restassured.response.ResponseBody;

import io.restassured.specification.RequestSpecification;

public class BasicAuth {

@Test

public void getData() {

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

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

ResponseBody body = res.body();

//Converting the response body to string

String rbdy = body.asString();

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

}

}

#### **Preemptive Authentication**

By default, Rest Assured uses the [***challenge-response mechanism.***](https://datatracker.ietf.org/doc/html/rfc2617#section-1.2) This means that it waits for the server to challenge rather than send the credentials directly. By using the preemptive directives we can avoid that additional call that the server makes and hence additional complications. In a way, it is similar to the basic auth we saw above, the only difference is that an additional premptive () directive adds after auth (). Let us see its syntax followed by a working code example.

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

As you may see above, the preemptive authentication view sends the authentication details in the request header irrespective of being asked by the server. In the same line of implementation, we will see a simple API that uses preemptive authentication.

package org.example;

import org.testng.annotations.Test;

import io.restassured.RestAssured;

import io.restassured.response.Response;

import io.restassured.response.ResponseBody;

import io.restassured.specification.RequestSpecification;

public class BasicAuth {

@Test

public void getUserData() {

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

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

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

ResponseBody body = res.body();

//Converting the response body to string

String rbdy = body.asString();

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

}

}

The code example used above is a simple Get API where we are trying to fetch the details corresponding to the user. Note that the server needs the authentication details of the user to get a successful response. Let us glide through the code line-by-line.

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

An object of RequestSpecification is created and using the preemptive directive the credentials of the user are sent in the header. Note that irrespective of being asked for the credentials these would be passed to the server.

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

ResponseBody body = res.body();

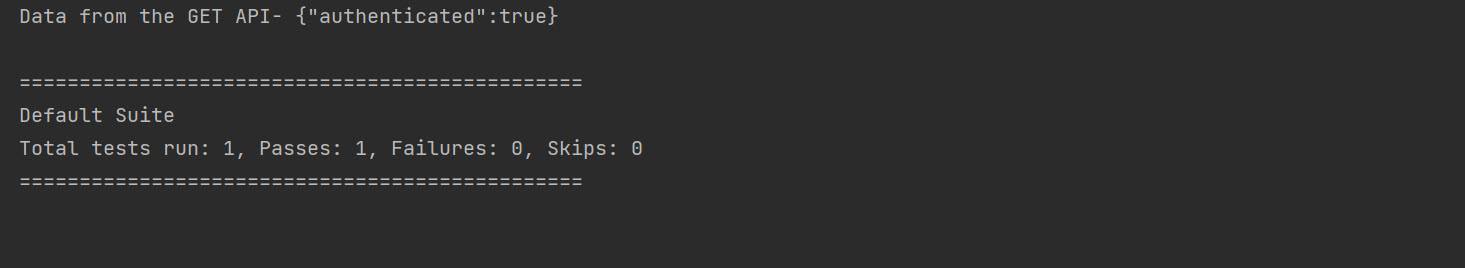
The endpoint URL is accessed using the get method and the response is saved using the ResponseBody object.

String rbdy = body.asString();

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

Finally, we convert the response body to string and print the result. Similarly, you may add additional validations as per your requirements.

Console prints the response of the above code without errors.



And there you go! You have successfully retrieved the user data by simply adding the preemptive authentication in your code and passing the credentials.

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

It is somewhat similar to challenge-based authentication but is more secure as it uses a digestive key in subsequent requests. If at all it is intercepted by an eavesdropper, he will get access only to the transaction performed and not the user password. The transaction might be replayed but a new transaction cannot be made as the password is not exposed. Its syntax is similar to basic authentication-

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

Note that we cannot use the preemptive () similar to basic auth since this scheme uses only challenged authentication.

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

There can be many cases when you need to pass the authentication credentials in an HTML form. This request is generally sent as a post method where the credentials entered in the form are used for authentication. So, if your application uses such a form-based authentication you can easily automate it using the form() scheme. The syntax for it follows-

given ().auth ().digest (***"your username"***, ***"your password"***).get (***"your endpoint URL"***)

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

If you use this approach then Rest Assured will first have to parse through the HTML response to find the fields for input and then send the form parameters. However, there is a high possibility that this approach might fail if the webpage is complex. Additionally, it would also fail if the context path is not included in the action attribute of the service. To optimize it to handle such cases, you may use the below format where you explicitly pass the required fields by providing the [***FormAuthConfig()-***](https://javadoc.io/doc/io.rest-assured/rest-assured/3.0.0/io/restassured/authentication/FormAuthConfig.html)

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

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

Another type of authentication is OAuth authentication. OAuth is an authorization framework that defines an identity protocol. It has wide usage in web applications and there are high chances that you will have to automate those authentication actions. These can be of two types viz, OAuth 1.0 and OAuth 2.0 which we will discuss now.

#### **OAuth 1.0**

Secured resources built using OAuth 1.0 requires passing consumer key, secret, access token, and token secret. The syntax it follows is -

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

OAuth parameters read the required user input dynamically.

#### **OAuth 2.0**

There are cases when we need to generate an access token for a user session. This access token performs various transactions and helps maintain the user session. While using [***OAuth 2.0***](https://www.toolsqa.com/postman/oauth-2-0-authorization/) you need to directly pass the access token generated when the user login using the below syntax-

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

Using the access token you can easily request any of the resources secured using the OAuth scheme.

Now you may identify the types of authentication used in your web application. Similarly, you can use the corresponding authentication scheme to make full use of rest assured capabilities.

Note: Corresponding Postman tutorial for basic auth can be found at [***Basic Authentication in Postman.***](https://www.toolsqa.com/postman/basic-authentication-in-postman/)

Also, note that the video tutorial for this topic is available at [***Basic Authentication in Rest Assured***](https://www.toolsqa.com/postman/basic-authentication-in-postman/)

## Key Takeaways

* Basic authentication helps you access the secured APIs and perform actions on the resources.
* Rest assured has four types of authentication schemes. They are basic, digest, form, and OAuth authentication.
* By default, rest assured uses a challenge-response mechanism. But, a preemptive directive sends the credentials without waiting for the server.
* OAuth has two types - OAuth1.0 or OAuth2.0.

# PUT Request using Rest Assured

Category: [**Rest-Assured**](https://toolsqa.com/categories/rest-assured) , April 4 2022

0

0

7 min read

**Author**



Yashraj Gore

**Reviewers**



Lakshay Sharma

**[Previous Lesson](https://toolsqa.com/rest-assured/basic-auth/)**

[Basic Authentication](https://toolsqa.com/rest-assured/basic-auth/)

**[Next Lesson](https://toolsqa.com/rest-assured/delete-request-using-rest-assured/)**

[DELETE Request using Rest Assured](https://toolsqa.com/rest-assured/delete-request-using-rest-assured/)

In earlier articles, we have seen how to read various components of an \*HTTP Response ([***Headers,***](https://www.original.toolsqa.com/validate-response-header-using-rest-assured/)[***Body,***](https://www.original.toolsqa.com/read-json-response-body-using-rest-assured/) and Status) and to [**Post a request\***](https://www.toolsqa.com/rest-assured/post-request-using-rest-assured/) using Rest Assured. In this article, we will continue our exploration of HTTP request methods and move on to the next method - the PUT request method using REST Assured.

* What is an HTTP PUT Request Method?
* PUT Vs. POST Request.
  + Status codes were obtained for PUT and POST requests.
  + REST API example.
* How to implement a PUT Request using Rest Assured?
  + Create JSON data using Simple JSON library.
  + Send JSON content in the body of the Request.
  + Validate the Response.

## What is an HTTP PUT Request Method?

The \*PUT method \*(HTTP PUT request method) creates a new resource or updates (substitutes) a representation of the target resource with the request payload. This means a Put request updates a resource at a specified URI. It is also used to create a new resource at the given URI or replace the entire product entity.

The official [**HTTP RFC**](https://tools.ietf.org/html/rfc7231#section-4.3.4) specifies:

* A PUT method puts or places a file or resource precisely at a specific URI.
  + In case a file or a resource already exists at that URI, the PUT method replaces that file or resource.
  + If there is no file or resource, PUT creates a new one.
* Responses to the PUT method are non-cacheable.
* PUT requests usually respond back with status code 200.

## PUT Vs. POST Request

Let us now discuss the main differences between a PUT and a POST request.

| **PUT** | **POST** |
| --- | --- |
| This method is idempotent. This means it will produce the same results if executed more than once. | This method is not idempotent. It produces different results every time it is executed. |
| When we need to modify a single resource that is already part of resource collection, we call the PUT method. | POST method is called when a child resource is to be added under resources collection. |
| PUT method syntax : PUT /questions/{question-id} | POST method syntax: POST /questions |
| PUT works as specific. | POST works as abstract. |
| Put method makes use of the "UPDATE" query. | POST method makes use of the "CREATE" query. |
| In the PUT method, the client decides which URI resource should have. | In the POST method, the server decides which URI resource should have. |
| \*PUT /vi/cake/orders/1234 indicates updation of a resource identified by "1234". | POST /vi/cake/orders\* indicate that we are creating a new resource and return an identifier to describe the resource. |

**Status codes obtained for PUT and POST requests**

The methods POST and PUT use the following status codes:

**POST request**

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

**PUT request**

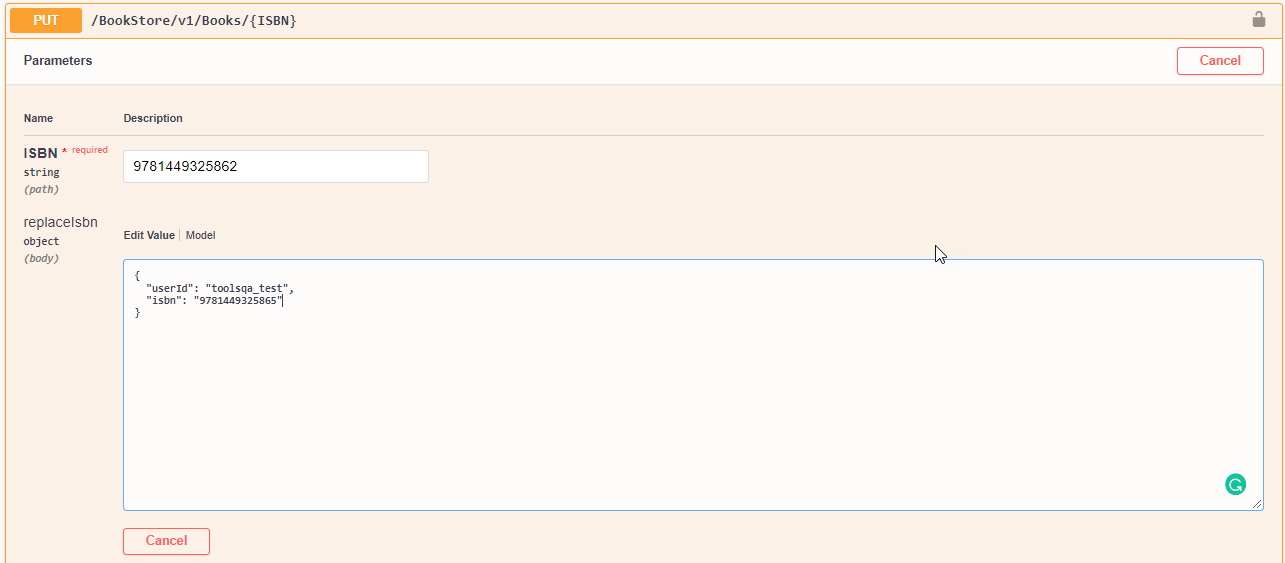
* 204 for OK/SUCCESS (but no content).
* 200 for OK with Content Body (Updated response).
* 400 if the data sent was invalid.

### ****REST API example****

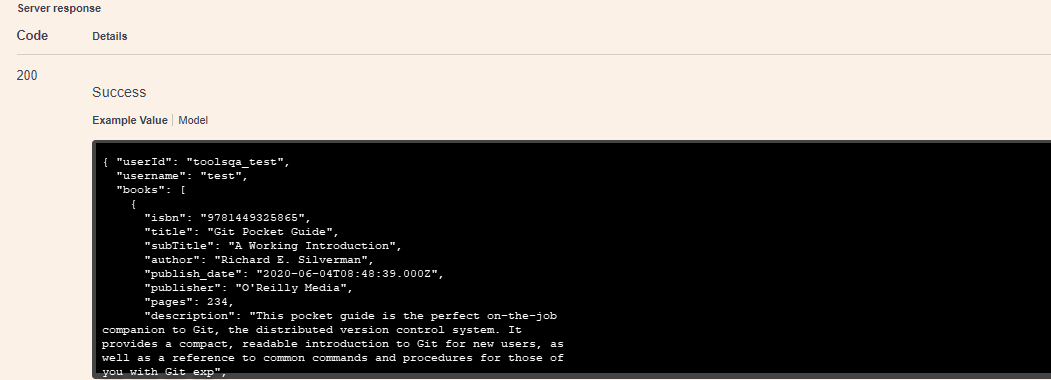
Now we'll demonstrate the PUT request using Swagger UI. The URL for the PUT request is given here [***https://demoqa.com/BookStore/v1/Books/***](https://demoqa.com/BookStore/v1/Books/)

Also, you can take a look at the [**Book Store Swagger**](https://demoqa.com/swagger/)

When we access this URL, we get the following screen.



In the above screen, we a trying to update a book record with ISBN="9781449325862". The two parameters that we will update are userId: "toolsqa\_test" and ISBN: "9781449325865". When we execute this put request we get the following output.



We can see the status code 200 which we received from the server indicating the request was successful.

In the next section, we will implement the PUT request using Rest Assured.

## How to implement a PUT Request using Rest Assured?

As explained in the tutorial on a [**\*POST request,**](https://www.toolsqa.com/rest-assured/post-request-using-rest-assured/) to create JSON objects, we will add a Simple JSON\* library in the classpath in the code. Once this is done, we follow the below-given steps to put a request using REST Assured.

* Create JSON data using a simple JSON library.
* Send JSON content in the body of the request.
* Validate the response.

Let us discuss each of these steps below.

### ****Create JSON data using Simple JSON library****

We will use the same URL for implementing the PUT request using Rest Assured.

Endpoint: [***https://demoqa.com/BookStore/v1/Books/***](https://demoqa.com/BookStore/v1/Books/)

The first step is to create a JSON data request that we need to send using the "put()" method. The following piece of code achieves this.

java import io.restassured.RestAssured; import io.restassured.specification.RequestSpecification; import org.junit.Test;

public class PUTMethod { @Test public void putMethod() {

String userId = "toolsqa\_test";

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

String token = "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyTmFtZSI6InRlc3RpbmcxMjMiLCJwYXNzd29yZCI6IlBhc3N3b3JkQDEiLCJpYXQiOjE2Mjg1NjQyMjF9.lW8JJvJF7jKebbqPiHOBGtCAus8D9Nv1BK6IoIIMJQ4";

String isbn = "9781449325865";

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

}

}

In the above code, we have used an ISBN parameter. Note that we already have a record with this ISBN. We will update this record by sending the PUT request. So for this ISBN, we will update the record with a new ISBN and a userId. We set a new value for the ISBN and also updated userId. So when we send this request, the record having ISBN="9781449325862" will have its userId="toolsqa\_test" and ISBN updated to the new value, "9781449325865".

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

The generated request is sent to the server.

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

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

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

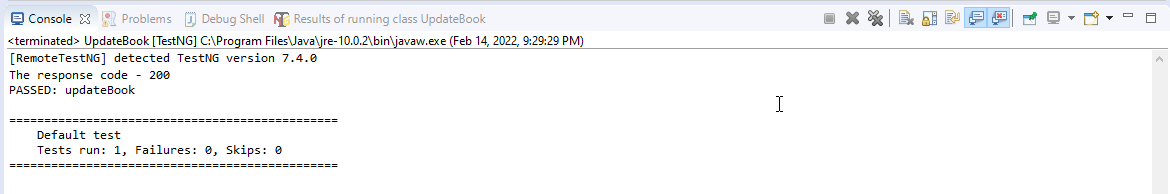
### ****Validate the Response****

The next step is to validate the response to ensure record updation. This is done by reading the status code of the response as follows:

java //validate the response System.out.println("The response code - " +res.getStatusCode());

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

We have used the getStatusCode () method that returns the status code of the response. The value 200 indicates the PUT request was successful. The execution of the above tests gives the following response:



We see that the test case has passed.

The complete code for the PUT request demonstration using Rest Assured is as follows:

java package org.example;

import io.restassured.RestAssured; import io.restassured.specification.RequestSpecification; import org.junit.Assert; import org.junit.Test; import io.restassured.response.Response; import io.restassured.response.ResponseBody; import io.restassured.specification.RequestSpecification; import org.testng.annotations.AfterTest; import org.testng.annotations.BeforeTest;

public class PUTMethod { String userId= "toolsqa\_test"; String baseUrl="https://demoqa.com"; String token = "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyTmFtZSI6InRlc3RpbmcxMjMiLCJwYXNzd29yZCI6IlBhc3N3b3JkQDEiLCJpYXQiOjE2Mjg1NjQyMjF9.lW8JJvJF7jKebbqPiHOBGtCAus8D9Nv1BK6IoIIMJQ4"; String isbn ="9781449325865";

@Test

public void updateBook() {

RestAssured.baseURI = baseUrl;

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

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

//Calling the Delete API with request body

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

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

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

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

}

}

The steps discussed above demonstrate the basic working of a PUT request and how it differs from a POST request. We used the JSON library to create JSON data sent in the content of the body of the request. We then validate the response using the status code after obtaining the response.

Note: Video tutorial for Put request is available at [**Put Request in Rest Assured**](https://www.youtube.com/watch?v=RzWX4z2_QjE)

## Key TakeAways

In this article, we discussed the PUT request.

* Put request is idempotent i.e. it produces the same results irrespective of the number of times the request is executed.
* The HTTP Put request updates or creates a resource.
* Put request usually responds with status code 200 when it is successful. The responses to the PUT request are not cacheable.
* We can use the Rest Assured library to put a request (using the put method). For this, we send a JSON object with the request and obtain a response.
* We validate the response by checking the status code.

In the next article of the Rest Assured series, we will discuss the "DELETE" method.

# DELETE Request using Rest Assured

Category: [**Rest-Assured**](https://toolsqa.com/categories/rest-assured) , February 1 2022

0

0

8 min read

**Author**



Gunjan Kaushik

**Reviewers**



Lakshay Sharma

**[Previous Lesson](https://toolsqa.com/rest-assured/put-request-using-rest-assured/)**

[PUT Request using Rest Assured](https://toolsqa.com/rest-assured/put-request-using-rest-assured/)

**[Next Lesson](https://toolsqa.com/rest-assured/what-is-json/)**

[What is JSON?](https://toolsqa.com/rest-assured/what-is-json/)

[***APIs form the base for any application***](https://toolsqa.com/postman/api-testing-with-postman/) and in every application, you perform a lot of different actions. These actions may range from adding, viewing, editing or deleting data. Since we have so many actions, we have to have a variety of tests to test the correctness of these actions. In our previous post, we learnt how to perform [***PUT requests using Rest Assured***](https://www.toolsqa.com/rest-assured/put-request-using-rest-assured/), and in this post, we will learn about the HTTP DELETE request method using the same method. If you have not gone through our previous articles I would urge you to go through our [***Rest Assured series***](https://www.toolsqa.com/rest-assured-tutorial/) step-by-step to understand the basics.

In this article, we will cover the below point-

* What is a Delete request?
* What are the different response codes for Delete Request?
* How to make a Delete Request using Rest Assured?

## What is a delete request method?

An HTTP delete request is performed using the HTTP delete method (written as delete) which deletes a resource from the server identified by the URI we are sending through it. Also note that a delete method intends to change the state of the server. Although even a 200 code does not guarantee this. A few major points as highlighted in the official [***HTTP RFC***](https://datatracker.ietf.org/doc/html/rfc7231#section-4.3.5) with respect to the delete request method are listed below-

* The Delete method requests the server to delete the resource identified by the request URI.
* The resource deletion depends on the server and is deleted if prescribed for deletion. Additionally, the restoration implementation of the resource is also considerable.
* There is a deletion request for association between the resource and corresponding current functionality.
* It is similar to the rm UNIX command, where the current association of the resource is deleted instead of the previous ones(if any).
* Delete request method is placed under the idempotent category of the W3C documentation. It means that once a resource is requested for deletion, a request on the same resource again would give the same result as the resource has already been deleted.
* Delete method response is non-cacheable. The user cannot cache the server response for later use. Caching a delete request creates inconsistencies.

Let's see the response code of Delete request now.

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

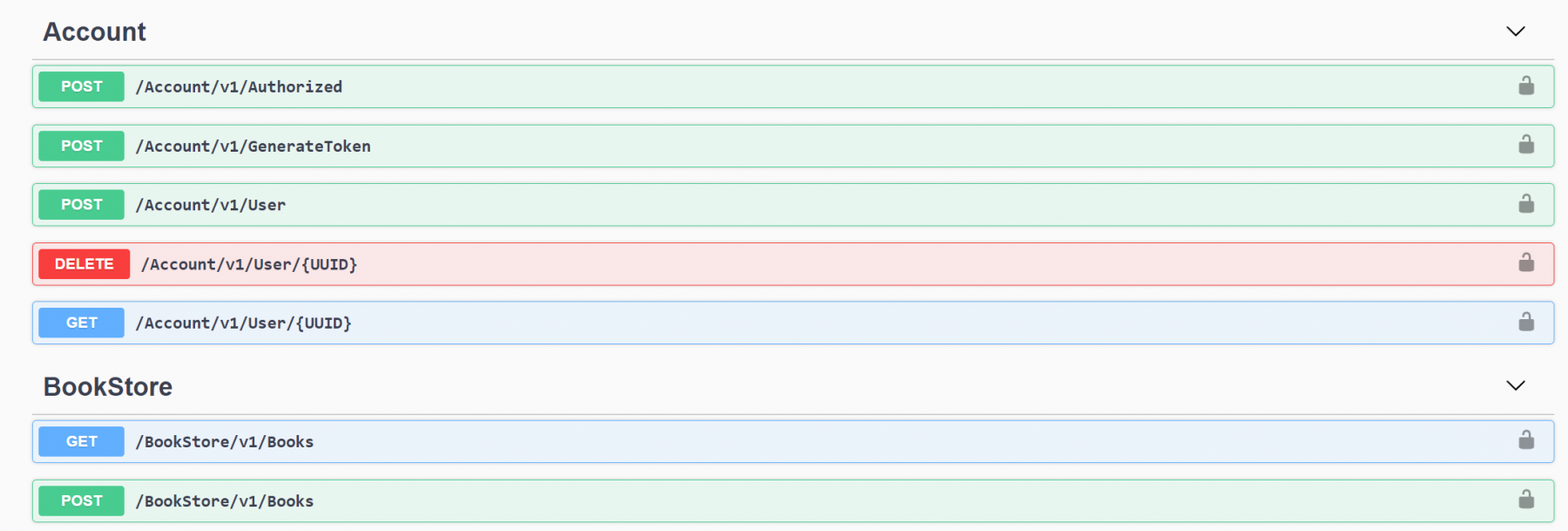
On sending the Delete request, the server responds with some response codes which signify the action performed by the Delete method. Consider the following code for the same-

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

Now that we understand the basics of the Delete method we will see how we can send a Delete request using rest assured but before that let us see a simple DELETE request using Swagger UI. The URL for the request is-

[***https://demoqa.com/swagger/***](https://demoqa.com/swagger/)

The URL will show a screen like below-

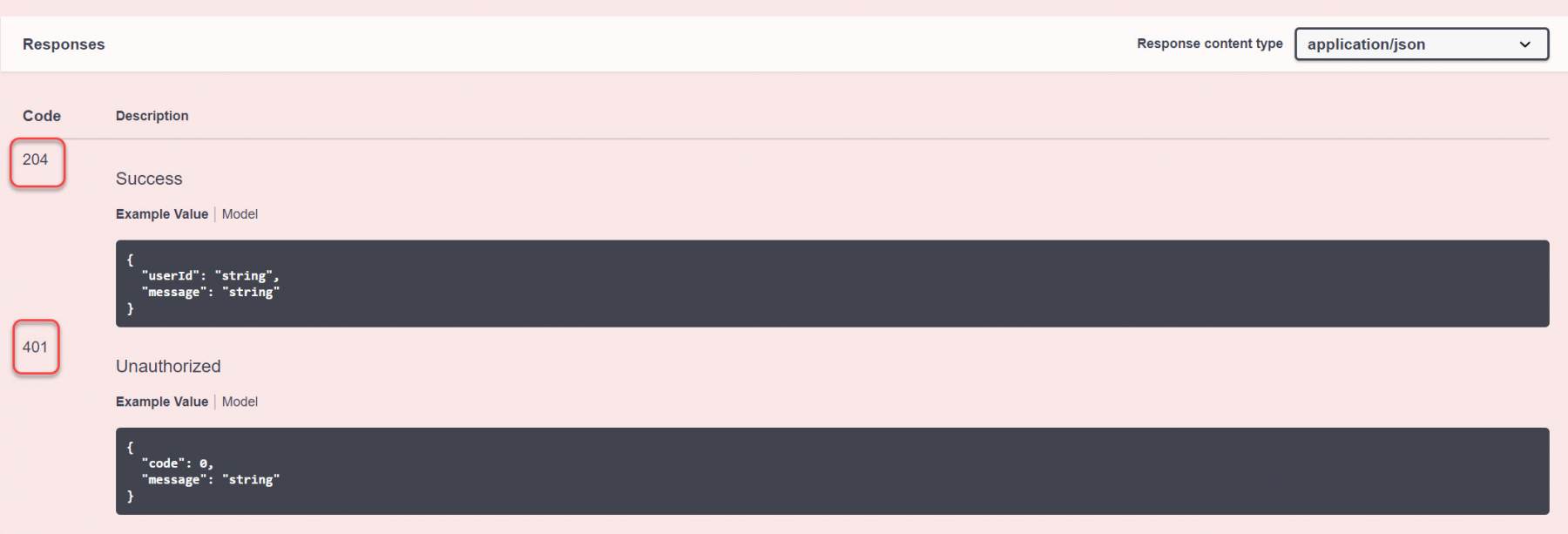


Now you will have to select the first "***Delete***" in the BookStore section from this page.



***Note***: We have disabled the UserID to prevent an empty database. In practical application, you have to pass the required values to the server.

When you will click on the section, you will see a response depicting that the execution was a success. A specific response is received for cases like "***bookid not found***" or "***user not found***". See the snapshot below for more clarity-



In the introduction, we defined the delete method as something that can delete a resource. As a reminder, you can relate the "***resource***" with the pet record here. A generic term "**resource**" denotes varying resources in various requests. Now that we have seen how we may go about manually deleting a resource using swagger next we will try to automate the Delete request using Rest Assured.

## How to make a Delete Request using Rest Assured?

We will now see how we can delete a resource from the server using rest assured. To do so, we will use [***BookStore APIs***](https://demoqa.com/swagger/) which we have been using in our previous examples. We will be automating the following use case-

1. Get a list of available books for to a user using the Get User [***Data API***](https://demoqa.com/swagger/#/Account/AccountV1UserPost).
2. Delete a book corresponding to a specific ISBN and validate the response code using the [***Delete Book API***](https://demoqa.com/swagger/#/BookStore/BookStoreV1BookDelete).
3. Verify that the execution of API from step 1 updates the book list and does not display the deleted book.

Let us now see the code for the above use case using Rest Assured.

package bookstore;

import org.testng.Assert;

import org.testng.annotations.AfterTest;

import org.testng.annotations.BeforeTest;

import org.testng.annotations.Test;

import io.restassured.RestAssured;

import io.restassured.response.Response;

import io.restassured.response.ResponseBody;

import io.restassured.specification.RequestSpecification;

public class DeleteBook {

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

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

String token = "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyTmFtZSI6InRlc3RpbmcxMjMiLCJwYXNzd29yZCI6IlBhc3N3b3JkQDEiLCJpYXQiOjE2Mjg1NjQyMjF9.lW8JJvJF7jKebbqPiHOBGtCAus8D9Nv1BK6IoIIMJQ4";

String isbn ="9781449337711";

@BeforeTest

@AfterTest

public void getUserData() {

RestAssured.baseURI = baseUrl;

RequestSpecification httpRequest =

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

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

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

ResponseBody body = res.body();

//Converting the response body to string

String rbdy = body.asString();

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

}

@Test

public void deleteBook() {

RestAssured.baseURI = baseUrl;

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

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

//Calling the Delete API with request body

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

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

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

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

}

}

#### **Code walkthrough**

Let's walk through the above code and see what we tried to achieve here. Note that the first method, i.e getUserData() is a simple GET method where we are trying to fetch the user data corresponding to a userId. You may refer to our article on [***Rest API using Rest Assured***](https://www.toolsqa.com/rest-assured/rest-api-test-using-rest-assured/) to understand the code if you have not gone through it yet. Additionally, you will notice the authorization field. For this particular tutorial on the delete request method, you need not worry about it. However, if you want to learn about [***authorization***](https://toolsqa.com/postman/basic-authentication-in-postman/) and [***OAuth 2.0***](https://toolsqa.com/postman/oauth-2-0-authorization/), you can refer to our other tutorials. In this tutorial about the delete method, we will go through the code written under deleteBook() method here.

RestAssured.baseURI = baseUrl;

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

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

We first set up the request with the base URI. Next, we add the header(s) which also include the Authorization token using the RequestSpecification interface.

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

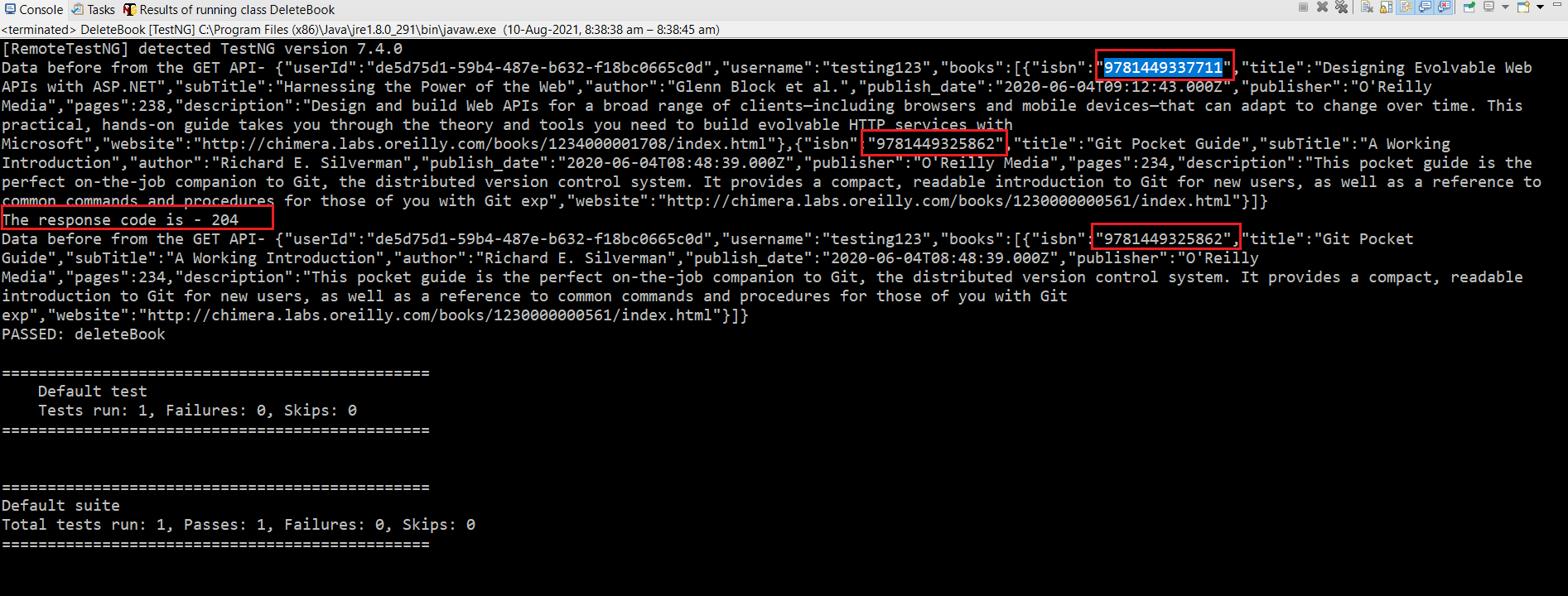
Next, store the Response object using the RequestSpecification object. Here we are passing the JSON body of our request along with the endpoint URL for the Delete method. The response gets store in a variable.

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

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

Lastly, we are simply printing the response status code and using the [***TestNG Assert***](https://www.toolsqa.com/testng/testng-asserts/) to validate the same. Since in our case the expected response code is 204, our test would have it as the expected parameter.

Once you execute the code you will see that the getUserData method executes first, followed by the deleteBook method. And then finally the getUserData method runs again. This is because we used the @BeforeTest and @AfterTest for the getUserData method. The console would show you the result as below-



And there you go! You have successfully deleted a record from the user data using the Delete API. Now you may go ahead and try out the different API methods using Rest Assured and strengthen your concepts. In our next post, we will see how Basic Auth handles Rest Assured.

Note: Video tutorial for Delete request is available at [***Delete Request in Rest Assured***](https://www.youtube.com/watch?v=DZP97jtU3-M)

## Key Takeaways

* The "***delete***" method deletes a resource from the server. It is quite similar to the rm UNIX command.
* The deletion of a resource is based on the server implementation and the response received is non-cacheable.
* The delete request returns any of the three types of response codes, i.e., ***202, 204,*** and ***200***.
* The ***delete()*** method used with "***path***" or "***pathParams***" deletes the request.
* To verify the deletion, you may use assertion or any other relevant code as required.

# What is JSON?

Category: [**Rest-Assured**](https://toolsqa.com/categories/rest-assured) , July 7 2021

0

0

4 min read

**Author**



Virender Singh

**Reviewers**



Lakshay Sharma

**[Previous Lesson](https://toolsqa.com/rest-assured/delete-request-using-rest-assured/)**

[DELETE Request using Rest Assured](https://toolsqa.com/rest-assured/delete-request-using-rest-assured/)

**[Next Lesson](https://toolsqa.com/rest-assured/jsonpath-and-query-json-using-jsonpath/)**

[JSONPath and Query JSON using JSONPath](https://toolsqa.com/rest-assured/jsonpath-and-query-json-using-jsonpath/)

***JSON*** is one of the most used Data-Exchange format. It is a light weight format represented as pure text. Due to the ability of almost all the languages to parse text, Json becomes independent of the programming language.

## What is JSON?

***JSON*** stands for ***JavaScript Object Notation***. JSON is a human and machine-readable format to represent data as ***Structured Data***. JSON is used primarily to transfer data from one computer to another or even between different programs on the same computer.

To visualize JSON, let us say that we want to represent a ***Person*** with following details in JSON

* First Name = Virender
* Last Name = Singh
* Age = 34
* Profession = Engineer

In JSON , this will turn out to be

{

"FirstName" : "Virender",

"LastName" : "Singh",

"Age" : 34,

"Profession": "Engineer"

}

At this point, we do not know the details of the JSON structure but we can still figure out what is being presented in JSON . Let us now understand different elements used in JSON .

***Note***: It is always beneficial to represent Keys without spaces in the name.

### *****Key-Value Pairs*****

***Key-Value*** pairs in JSON are used to represent a property of the Object. In the above example, we tried to represent a ***Person***. This person has some properties like

* First Name
* Last Name
* Age
* Profession

Each of these properties have a value associated with it. For e.g ***First Name*** has a value of ***Virender***. Similarly, ***Age*** has a value of ***34***. To write a Key-Value in JSON we have to follow these rules

* Key-value pairs are separated by a : (colon)
* Key is always present in ***Double Quotes " "***
* Value could be anything depending on the Data Type

From the above example, a Key-Value pair is ***FirstName***. Try to find other Key-Value pairs yourself

"FirstName" : "Virender"

Values can be of following data types

* Boolean: True or False
* Number: Numerical values
* Object: An associative array of Key-Value pairs
* Array: Associative array of just values

### *****Object in JSON*****

In JSON an object is represented by a collection of Key-Value pairs. This collection of Key-Value pairs are grouped using { } (opening and closing curly braces}. Rules to writing an Object are

* Key-Value pairs should be separated by a , (Comma)
* Each Object should ***Start*** with an ***Opening {*** (Opening Curly Brace)
* Each Object should **End** with a ***Closing }*** (Closing Curly Brace)

An example here is the Person Object, discussed above. The Person object follows the rules mentioned for representing an Object

{

"FirstName" : "Virender",

"LastName" : "Singh",

"Age" : 34,

"Profession": "Engineer"

}

### *****Array in JSON*****

Arrays are similar to Arrays that you know from any other programming language. In JSON an Array is collection of Values separated by Comma. Here are the rules to write an Array

* An Array starts with an opening  ***[*** (Bracket)
* An Array ends with a closing ***]*** (Bracket)
* Values in the Array are separated by ***,*** (Comma)

To understand an Array let us add one more property to the ***Person Object***. Let us add hobby also, a Person can have multiple hobbies. This makes it suitable to represent hobbies as an Array. As shown in the JSON below

{

"FirstName" : "Virender",

"LastName" : "Singh",

"Age" : 34,

"Profession": "Engineer",

"Hobbies" : ["Videos games", "Computers", "Music"]

}

See how multiple hobbies are represented using an Array. Array starts and ends with ***[ ]*** and contains values separated by ,

These are the different components of a JSON , using these components we can create complex JSON . As a small exercise for you, try to decode the JSON below and identify different structures involved in it.

{

"Description": "Map containing Country, Capital, Currency, and some States of that Country",

"Region": "Asia",

"Countries": [

{

"Country": "India",

"Data": {

"Capital": "New Delhi",

"minimum temp (Degree Celsius)": 6,

"maximum temp (Degree Celsius)": 45,

"Currency": "Rupee"

}

},

{

"Country": "Nepal",

"Data": {

"Capital": "Katmandu",

"minimum temp (Degree Celsius)": 9,

"maximum temp (Degree Celsius)": 23,

"Currency": "Nepalese rupee"

}

}

]

}

# JSONPath and Query JSON using JSONPath

Category: [**Rest-Assured**](https://toolsqa.com/categories/rest-assured) , November 9 2021

0

0

6 min read

**Author**



Virender Singh

**Reviewers**



Lakshay Sharma

**[Previous Lesson](https://toolsqa.com/rest-assured/what-is-json/)**

[What is JSON?](https://toolsqa.com/rest-assured/what-is-json/)

**[Next Lesson](https://toolsqa.com/rest-assured/expressions-in-jsonpath/)**

[Expressions in JSONPath](https://toolsqa.com/rest-assured/expressions-in-jsonpath/)

## JSONPath and Query JSON using JSONPath

One of the most important advantage of JSON is that it is a light weight format that can be used to interchange data between computers and processes. JSON , like XML, is a format to provide structure to the data. If you are not familiar with JSON, please go through this tutorial first of [***Introduction to JSON***](https://toolsqa.com/rest-assured/what-is-json/).

In this tutorial, we will learn more about ***JSONPath and Query JSON using JSONPath.*** We will be covering the following topic:

* ***What is JSONPath?***
* ***How to Query JSON with JSONPath?***
* ***Different strategies to query JSON object and JSON Array.***

### *****What is JSONPath?*****

Every ***JSON*** object is composed on an inherent hierarchy and structure. Every JSON ends up creating a tree of nodes, where each node is a ***JSON Element***. Let us take an example here, below is a simple JSON expressing a collection of countries

{

"Description": "Map containing Country, Capital, Currency, and some States of that Country",

"Region": "Asia",

"Countries": [

{

"Country": "India",

"Data": {

"Capital": "New Delhi",

"mintemp": 6,

"maxtemp": 45,

"Currency": "Rupee"

}

},

{

"Country": "Nepal",

"Data": {

"Capital": "Katmandu",

"mintemp": 9,

"maxtemp": 23,

"Currency": "Nepalese rupee"

}

}

]

}

At the top most level we have a ***Root*** node, which is basically the node containing all of the current JSON. Inside this root node, we have following nodes

* ***Description***
* ***Region***
* ***Countries***

***Description*** and ***Region*** are simple leaf nodes in the tree. But ***Countries*** is a non-leaf node, which further contains more nodes. Here ***Countries*** node contains an array of two countries. If we were to simply define a hierarchical relation between the ***Root*** node and any node in the JSON we can do like shown below.

***Note:*** Let us represent ***Root*** node by ***$*** and a relationship from parent to child with ***>>***

***Description*** node will be represented by ***$ >> Description***.

***Region*** node will be represented by ***$ >> Region***.

Similarly, we can also define a relationship between the ***Root*** node and the zero'th item in the ***Countries*** array. Relationship will be    ***$ >> Countries[0]*** where **[ ]** is the index operator to represent an item at n index in an Array of ***Countries.***

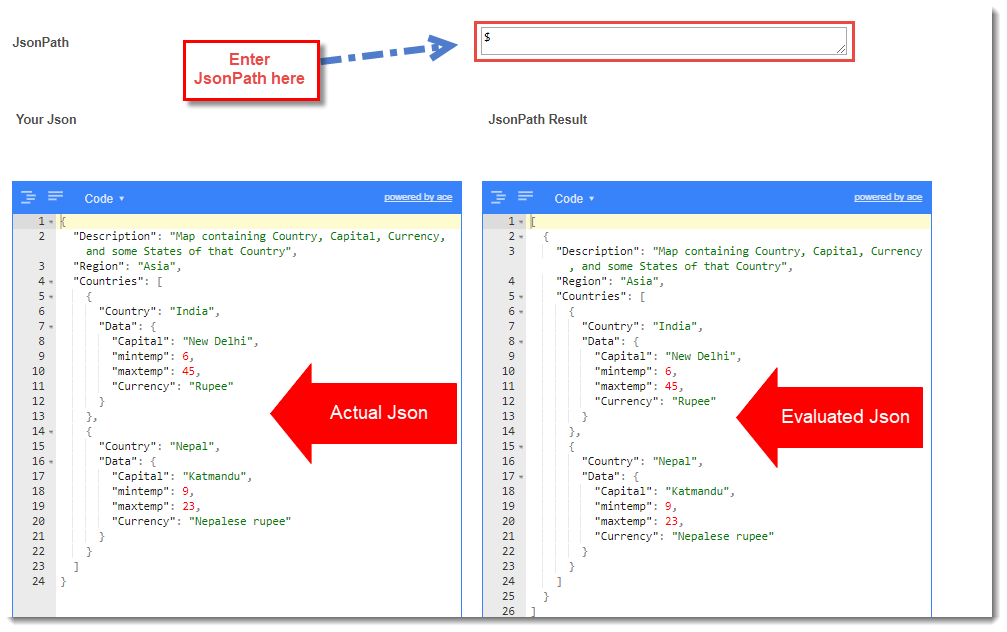
In a way this hierarchy in JSON allows us to create a standard mechanism to traverse through specific parts of the JSON. A standard way to do this is called ***JSONPath***.

### *****How to Query JSON using JSONPath?*****

***JSONPath*** creates a uniform standard and syntax to define different parts of a JSON document. JSONPath defines expressions to traverse through a JSON document to reach to a subset of the JSON. This topic is best understood by seeing it in action. We have created a web page which can help you evaluate a JSONPath. Use this page to practice writing JSONPath. This is the link to [***JSONPath evaluator.***](https://jsonpath.com/)

#### **Get Root node operator in JsonPath**

***Root*** node operator in ***JSON*** is represented by a ***$*** sign. **$** will return all the nodes inside the JSON document. To try this out open the [***JSONPath evaluator***](https://jsonpath.com/) page and type **$** in the JSONPath field. As shown in the image below



#### **Get Children operator in JSONPath**

In order to get children of a given node, we can use the ***Dot (.)*** operator or the ***['childname']*** operator. In order to get all the Countries we can have JSONPath as

* ***$.Countries***
* ***$['Countries']***

Output will be

[

[

{

"Country": "India",

"Data": {

"Capital": "New Delhi",

"mintemp": 6,

"maxtemp": 45,

"Currency": "Rupee"

}

},

{

"Country": "Nepal",

"Data": {

"Capital": "Katmandu",

"mintemp": 9,

"maxtemp": 23,

"Currency": "Nepalese rupee"

}

}

]

]

#### **Wildcard operator in JSONPath**

Wild card operator in ***JSONPath*** is (Star or Asterisk) symbol. This is literally means everything under that node. For example, if you want to display the Data nodes of all the countries you can simple write following JSONPat

* **$.Countries[].Data**\*
* **$['Countries'][].Data**\*

This will display the Data nodes of all the countries

[

{

"Capital": "New Delhi",

"mintemp": 6,

"maxtemp": 45,

"Currency": "Rupee"

},

{

"Capital": "Katmandu",

"mintemp": 9,

"maxtemp": 23,

"Currency": "Nepalese rupee"

}

]

#### **Array Index operator in JSONPath**

Sometimes it is required to access a particular entry at a given index in the JSON array. We can use the Array Index ***[i,j,k... ]***  to identify an entry at a particular index. In the above example, let us find out the last ***Country*** entry in the ***Countries*** array.

* ***$.Countries[-1]***
* ***$['Countries'][-1]***

Here -1 stands for the last item in the Array. You can also refer to the last item by giving a positive value to the index. For e.g.

* ***$.Countries[1]***
* ***$['Countries'][1]***

Output will look like this

[

{

"Country": "Nepal",

"Data": {

"Capital": "Katmandu",

"mintemp": 9,

"maxtemp": 23,

"Currency": "Nepalese rupee"

}

}

]

***Note:*** Array index starts from 0. Hence to refer to the second item in the array we have to use 1 as the index.

Array index is just not limited to displaying only 1 item. We can extract multiple items from the array at different indexes. The syntax to do so is ***[i,j,k..].***  For e.g. to extract the first 2 array items we will write the JSONPath as

* ***$.Countries[0,1]***
* ***$['Countries'][0,1]***

Output will be

[

{

"Country": "India",

"Data": {

"Capital": "New Delhi",

"mintemp": 6,

"maxtemp": 45,

"Currency": "Rupee"

}

},

{

"Country": "Nepal",

"Data": {

"Capital": "Katmandu",

"mintemp": 9,

"maxtemp": 23,

"Currency": "Nepalese rupee"

}

}

]

For the JsonPath shown below, we will change our Json to have more nodes. Let us take a look at a Json representing collections of books. Here is the Json, from now on use this Json in the [***JsonPath Evaluator***](https://jsonpath.com/)

{

"store": {

"book": [

{

"category": "reference",

"author": "Nigel Rees",

"title": "Sayings of the Century",

"price": 8.95

},

{

"category": "fiction",

"author": "Evelyn Waugh",

"title": "Sword of Honour",

"price": 12.99

},

{

"category": "fiction",

"author": "Herman Melville",

"title": "Moby Dick",

"isbn": "0-553-21311-3",

"price": 8.99

},

{

"category": "fiction",

"author": "J. R. R. Tolkien",

"title": "The Lord of the Rings",

"isbn": "0-395-19395-8",

"price": 22.99

}

],

"bicycle": {

"color": "red",

"price": 19.95

}

},

"expensive": 10

}

#### **Array slice operator in JsonPath**

Array slice operator is wonderful operator to extract selected items from Json. Taking the example of books, what if we want to retrieve every alternative book in the Json. To do that we will need the Array, Slice operator. Syntax of Array Slice operator is **[StartIndex : EndIndex : Steps].** Let find out what are books at odd number in the books collection. JsonPath will look like

* ***$..book[1,4,2]***
* ***$..['book'][1,4,2]***

the output of the above JsonPath will be

[

{

"category": "fiction",

"author": "Evelyn Waugh",

"title": "Sword of Honour",

"price": 12.99

},

{

"category": "fiction",

"author": "Herman Melville",

"title": "Moby Dick",

"isbn": "0-553-21311-3",

"price": 8.99

}

]

# Expressions in JSONPath

Category: [**Rest-Assured**](https://toolsqa.com/categories/rest-assured) , November 13 2021

0

0

17 min read

**Author**



Virender Singh

**Reviewers**



Lakshay Sharma

**[Previous Lesson](https://toolsqa.com/rest-assured/jsonpath-and-query-json-using-jsonpath/)**

[JSONPath and Query JSON using JSONPath](https://toolsqa.com/rest-assured/jsonpath-and-query-json-using-jsonpath/)

**[Next Lesson](https://toolsqa.com/rest-assured/deserialize-json-array/)**

[Deserialize JSON Array to List](https://toolsqa.com/rest-assured/deserialize-json-array/)

Usage of ***Expressions in JsonPath*** is a very good feature to have concise and complex JsonPaths. Expressions in JsonPath are basically code snippets that evaluate to the Boolean value. Based on the outcome only the nodes which meet the criteria are selected. Let us see more about it, but before that make sure you have gone through following tutorials on basics of Json and JsonPath

* [***Json***](https://toolsqa.com/rest-assured/what-is-json/)
* [***Jsonpath***](https://toolsqa.com/rest-assured/jsonpath-and-query-json-using-jsonpath/)

In this tutorial, we will use a sample Json which has a few items in the Array. Please copy the below Json in our [***JsonPath Evaluator***](https://jsonpath.com/)

{

"books": [

{

"isbn": "9781593275846",

"title": "Eloquent JavaScript, Second Edition",

"subtitle": "A Modern Introduction to Programming",

"author": "Marijn Haverbeke",

"published": "2014-12-14T00:00:00.000Z",

"publisher": "No Starch Press",

"pages": 472,

"description": "JavaScript lies at the heart of almost every modern web application, from social apps to the newest browser-based games. Though simple for beginners to pick up and play with, JavaScript is a flexible, complex language that you can use to build full-scale applications.",

"website": "https://eloquentjavascript.net/"

},

{

"isbn": "9781449331818",

"title": "Learning JavaScript Design Patterns",

"subtitle": "A JavaScript and jQuery Developer's Guide",

"author": "Addy Osmani",

"published": "2012-07-01T00:00:00.000Z",

"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-date with the latest best practices, this book is for you.",

"website": "https://www.addyosmani.com/resources/essentialjsdesignpatterns/book/"

},

{

"isbn": "9781449365035",

"title": "Speaking JavaScript",

"subtitle": "An In-Depth Guide for Programmers",

"author": "Axel Rauschmayer",

"published": "2014-02-01T00:00:00.000Z",

"publisher": "O'Reilly Media",

"pages": 460,

"description": "Like it or not, JavaScript is everywhere these days-from browser to server to mobile-and now you, too, need to learn the language or dive deeper than you have. This concise book guides you into and through JavaScript, written by a veteran programmer who once found himself in the same position.",

"website": "https://speakingjs.com/"

},

{

"isbn": "9781491950296",

"title": "Programming JavaScript Applications",

"subtitle": "Robust Web Architecture with Node, HTML5, and Modern JS Libraries",

"author": "Eric Elliott",

"published": "2014-07-01T00:00:00.000Z",

"publisher": "O'Reilly Media",

"pages": 254,

"description": "Take advantage of JavaScript's power to build robust web-scale or enterprise applications that are easy to extend and maintain. By applying the design patterns outlined in this practical book, experienced JavaScript developers will learn how to write flexible and resilient code that's easier-yes, easier-to work with as your code base grows.",

"website": "https://chimera.labs.oreilly.com/books/1234000000262/index.html"

},

{

"isbn": "9781593277574",

"title": "Understanding ECMAScript 6",

"subtitle": "The Definitive Guide for JavaScript Developers",

"author": "Nicholas C. Zakas",

"published": "2016-09-03T00:00:00.000Z",

"publisher": "No Starch Press",

"pages": 352,

"description": "ECMAScript 6 represents the biggest update to the core of JavaScript in the history of the language. In Understanding ECMAScript 6, expert developer Nicholas C. Zakas provides a complete guide to the object types, syntax, and other exciting changes that ECMAScript 6 brings to JavaScript.",

"website": "https://leanpub.com/understandinges6/read"

},

{

"isbn": "9781491904244",

"title": "You Don't Know JS",

"subtitle": "ES6 & Beyond",

"author": "Kyle Simpson",

"published": "2015-12-27T00:00:00.000Z",

"publisher": "O'Reilly Media",

"pages": 278,

"description": "No matter how much experience you have with JavaScript, odds are you don’t fully understand the language. As part of the 'You Don’t Know JS' series, this compact guide focuses on new features available in ECMAScript 6 (ES6), the latest version of the standard upon which JavaScript is built.",

"website": "https://github.com/getify/You-Dont-Know-JS/tree/master/es6%20&%20beyond"

},

{

"isbn": "9781449325862",

"title": "Git Pocket Guide",

"subtitle": "A Working Introduction",

"author": "Richard E. Silverman",

"published": "2013-08-02T00:00:00.000Z",

"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 experience.",

"website": "https://chimera.labs.oreilly.com/books/1230000000561/index.html"

},

{

"isbn": "9781449337711",

"title": "Designing Evolvable Web APIs with ASP.NET",

"subtitle": "Harnessing the Power of the Web",

"author": "Glenn Block, et al.",

"published": "2014-04-07T00:00:00.000Z",

"publisher": "O'Reilly Media",

"pages": 538,

"description": "Design and build Web APIs for a broad range of clients—including browsers and mobile devices—that can adapt to change over time. This practical, hands-on guide takes you through the theory and tools you need to build evolvable HTTP services with Microsoft’s ASP.NET Web API framework. In the process, you’ll learn how design and implement a real-world Web API.",

"website": "https://chimera.labs.oreilly.com/books/1234000001708/index.html"

}

]

}

## Expressions in JsonPath

Expressions in JsonPath are one of the most powerful features of ***JsonPath***. Note that expressions are also available in ***Xpath*** and ***CSS Selectors***. Expressions help you create a condition that is evaluated to true or false. There are two important symbols that you have to understand before you can create Expressions in ***JsonPath***.

* ? : Question mark, marks the beginning of an expression. Syntax used ***[? (Expression)]***
* @ : At symbol, signifies the current node being processed. Syntax used ***$.books[?(@.price > 100)]***

Let us now take a simple task from the Json above.

* ***Find out all the books which have the pages greater than 460***

To create a JsonPath which can get us all the books which have pages greater than 460, we have to break the problem into two parts

1. ***Create a JsonPath to retrieve all books***
2. ***Append an expression to filter all the books which have pages greater than 460***

To get all the books we can create a simple ***JsonPath: $.books*** . Now we have to add an expression in the array books. To do so we will simple start an expression with\*\*\*?\*\*\* sign and then add a filter on the current node ***@*** . A simple expression will look like this ***?(@.pages > 460)***.

If we combine JsonPath with the expression we will get this: ***$.books[?(@.pages > 460)]***

In the [***JsonPath Evaluator***](https://jsonpath.com/), simply enter this expression and see the results. As shown in the image below



The result will be all the books with page numbers greater than 460. Here is the result Json

[

{

"isbn": "9781593275846",

"title": "Eloquent JavaScript, Second Edition",

"subtitle": "A Modern Introduction to Programming",

"author": "Marijn Haverbeke",

"published": "2014-12-14T00:00:00.000Z",

"publisher": "No Starch Press",

"pages": 472,

"description": "JavaScript lies at the heart of almost every modern web application, from social apps to the newest browser-based games. Though simple for beginners to pick up and play with, JavaScript is a flexible, complex language that you can use to build full-scale applications.",

"website": "https://eloquentjavascript.net/"

},

{

"isbn": "9781449337711",

"title": "Designing Evolvable Web APIs with ASP.NET",

"subtitle": "Harnessing the Power of the Web",

"author": "Glenn Block, et al.",

"published": "2014-04-07T00:00:00.000Z",

"publisher": "O'Reilly Media",

"pages": 538,

"description": "Design and build Web APIs for a broad range of clients—including browsers and mobile devices—that can adapt to change over time. This practical, hands-on guide takes you through the theory and tools you need to build evolvable HTTP services with Microsoft’s ASP.NET Web API framework. In the process, you’ll learn how design and implement a real-world Web API.",

"website": "https://chimera.labs.oreilly.com/books/1234000001708/index.html"

}

]

### *****Logical operators in JsonPath*****

Just like any programming language, ***JsonPath*** supports all the logical operators. Below is the list of Logical Operators that we can use to create expressions. Every logical operator is discussed in detail below.

| ***Operator*** | ***Description*** |
| --- | --- |
| == | left is equal to right (note that 1 is not equal to '1'). |
| != | left is not equal to right. |
| < | left is less than right. |
| <= | left is less or equal to right. |
| > | left is greater than right. |
| >= | left is greater than or equal to right. |

Try all the above examples and also try to create more expressions based on your needs. This way you will learn more about the ***JsonPath*** expressions.

### *****Equals to (==) operator in JsonPath*****

As the name suggests the operator checks if the left-hand side is equal to the right-hand side. Let us find out all the books that have 352 pages. Here is the JsonPath

***JsonPath:  $.books[?(@.pages == 352)]***

***Result will be:***

[

{

"isbn": "9781593277574",

"title": "Understanding ECMAScript 6",

"subtitle": "The Definitive Guide for JavaScript Developers",

"author": "Nicholas C. Zakas",

"published": "2016-09-03T00:00:00.000Z",

"publisher": "No Starch Press",

"pages": 352,

"description": "ECMAScript 6 represents the biggest update to the core of JavaScript in the history of the language. In Understanding ECMAScript 6, expert developer Nicholas C. Zakas provides a complete guide to the object types, syntax, and other exciting changes that ECMAScript 6 brings to JavaScript.",

"website": "https://leanpub.com/understandinges6/read"

}

]

### *****Not equal to (!=) operator in JsonPath*****

When we want to ***exclude*** a particular set of values based on a condition we use the not equal to operator. Let us just invert the example above and find all the books that have ***page numbers not equal to 352***.

***JsonPath: $.books[?(@.pages != 352)]***

***Result will be:***

[

{

"isbn": "9781593275846",

"title": "Eloquent JavaScript, Second Edition",

"subtitle": "A Modern Introduction to Programming",

"author": "Marijn Haverbeke",

"published": "2014-12-14T00:00:00.000Z",

"publisher": "No Starch Press",

"pages": 472,

"description": "JavaScript lies at the heart of almost every modern web application, from social apps to the newest browser-based games. Though simple for beginners to pick up and play with, JavaScript is a flexible, complex language that you can use to build full-scale applications.",

"website": "https://eloquentjavascript.net/"

},

{

"isbn": "9781449331818",

"title": "Learning JavaScript Design Patterns",

"subtitle": "A JavaScript and jQuery Developer's Guide",

"author": "Addy Osmani",

"published": "2012-07-01T00:00:00.000Z",

"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-date with the latest best practices, this book is for you.",

"website": "https://www.addyosmani.com/resources/essentialjsdesignpatterns/book/"

},

{

"isbn": "9781449365035",

"title": "Speaking JavaScript",

"subtitle": "An In-Depth Guide for Programmers",

"author": "Axel Rauschmayer",

"published": "2014-02-01T00:00:00.000Z",

"publisher": "O'Reilly Media",

"pages": 460,

"description": "Like it or not, JavaScript is everywhere these days-from browser to server to mobile-and now you, too, need to learn the language or dive deeper than you have. This concise book guides you into and through JavaScript, written by a veteran programmer who once found himself in the same position.",

"website": "https://speakingjs.com/"

},

{

"isbn": "9781491950296",

"title": "Programming JavaScript Applications",

"subtitle": "Robust Web Architecture with Node, HTML5, and Modern JS Libraries",

"author": "Eric Elliott",

"published": "2014-07-01T00:00:00.000Z",

"publisher": "O'Reilly Media",

"pages": 254,

"description": "Take advantage of JavaScript's power to build robust web-scale or enterprise applications that are easy to extend and maintain. By applying the design patterns outlined in this practical book, experienced JavaScript developers will learn how to write flexible and resilient code that's easier-yes, easier-to work with as your code base grows.",

"website": "https://chimera.labs.oreilly.com/books/1234000000262/index.html"

},

{

"isbn": "9781491904244",

"title": "You Don't Know JS",

"subtitle": "ES6 & Beyond",

"author": "Kyle Simpson",

"published": "2015-12-27T00:00:00.000Z",

"publisher": "O'Reilly Media",

"pages": 278,

"description": "No matter how much experience you have with JavaScript, odds are you don’t fully understand the language. As part of the 'You Don’t Know JS' series, this compact guide focuses on new features available in ECMAScript 6 (ES6), the latest version of the standard upon which JavaScript is built.",

"website": "https://github.com/getify/You-Dont-Know-JS/tree/master/es6+&+beyond"

},

{

"isbn": "9781449325862",

"title": "Git Pocket Guide",

"subtitle": "A Working Introduction",

"author": "Richard E. Silverman",

"published": "2013-08-02T00:00:00.000Z",

"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 experience.",

"website": "https://chimera.labs.oreilly.com/books/1230000000561/index.html"

},

{

"isbn": "9781449337711",

"title": "Designing Evolvable Web APIs with ASP.NET",

"subtitle": "Harnessing the Power of the Web",

"author": "Glenn Block, et al.",

"published": "2014-04-07T00:00:00.000Z",

"publisher": "O'Reilly Media",

"pages": 538,

"description": "Design and build Web APIs for a broad range of clients—including browsers and mobile devices—that can adapt to change over time. This practical, hands-on guide takes you through the theory and tools you need to build evolvable HTTP services with Microsoft’s ASP.NET Web API framework. In the process, you’ll learn how design and implement a real-world Web API.",

"website": "https://chimera.labs.oreilly.com/books/1234000001708/index.html"

}

]

### *****Less than ( < ) operator in JsonPath*****

Less than operator, as the name suggests will return all the values that are less than the value given on the right. Let us find out all the books with pages less than 352.

***JsonPath: $.books[?(@.pages < 352)]***

***Result will be:***

[

{

"isbn": "9781449331818",

"title": "Learning JavaScript Design Patterns",

"subtitle": "A JavaScript and jQuery Developer's Guide",

"author": "Addy Osmani",

"published": "2012-07-01T00:00:00.000Z",

"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-date with the latest best practices, this book is for you.",

"website": "https://www.addyosmani.com/resources/essentialjsdesignpatterns/book/"

},

{

"isbn": "9781491950296",

"title": "Programming JavaScript Applications",

"subtitle": "Robust Web Architecture with Node, HTML5, and Modern JS Libraries",

"author": "Eric Elliott",

"published": "2014-07-01T00:00:00.000Z",

"publisher": "O'Reilly Media",

"pages": 254,

"description": "Take advantage of JavaScript's power to build robust web-scale or enterprise applications that are easy to extend and maintain. By applying the design patterns outlined in this practical book, experienced JavaScript developers will learn how to write flexible and resilient code that's easier-yes, easier-to work with as your code base grows.",

"website": "https://chimera.labs.oreilly.com/books/1234000000262/index.html"

},

{

"isbn": "9781491904244",

"title": "You Don't Know JS",

"subtitle": "ES6 & Beyond",

"author": "Kyle Simpson",

"published": "2015-12-27T00:00:00.000Z",

"publisher": "O'Reilly Media",

"pages": 278,

"description": "No matter how much experience you have with JavaScript, odds are you don’t fully understand the language. As part of the 'You Don’t Know JS' series, this compact guide focuses on new features available in ECMAScript 6 (ES6), the latest version of the standard upon which JavaScript is built.",

"website": "https://github.com/getify/You-Dont-Know-JS/tree/master/es6+&+beyond"

},

{

"isbn": "9781449325862",

"title": "Git Pocket Guide",

"subtitle": "A Working Introduction",

"author": "Richard E. Silverman",

"published": "2013-08-02T00:00:00.000Z",

"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 experience.",

"website": "https://chimera.labs.oreilly.com/books/1230000000561/index.html"

}

]

### *****Less than equal to ( <= ) operator in JsonPath*****

This operator will let you get all the values that are less than or equal to a given value. Let us find out all the books which have pages less than or equal to 352.

***JsonPath: $.books[?(@.pages < 352)]***

***Result will be:***

[

{

"isbn": "9781449331818",

"title": "Learning JavaScript Design Patterns",

"subtitle": "A JavaScript and jQuery Developer's Guide",

"author": "Addy Osmani",

"published": "2012-07-01T00:00:00.000Z",

"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-date with the latest best practices, this book is for you.",

"website": "https://www.addyosmani.com/resources/essentialjsdesignpatterns/book/"

},

{

"isbn": "9781491950296",

"title": "Programming JavaScript Applications",

"subtitle": "Robust Web Architecture with Node, HTML5, and Modern JS Libraries",

"author": "Eric Elliott",

"published": "2014-07-01T00:00:00.000Z",

"publisher": "O'Reilly Media",

"pages": 254,

"description": "Take advantage of JavaScript's power to build robust web-scale or enterprise applications that are easy to extend and maintain. By applying the design patterns outlined in this practical book, experienced JavaScript developers will learn how to write flexible and resilient code that's easier-yes, easier-to work with as your code base grows.",

"website": "https://chimera.labs.oreilly.com/books/1234000000262/index.html"

},

{

"isbn": "9781491904244",

"title": "You Don't Know JS",

"subtitle": "ES6 & Beyond",

"author": "Kyle Simpson",

"published": "2015-12-27T00:00:00.000Z",

"publisher": "O'Reilly Media",

"pages": 278,

"description": "No matter how much experience you have with JavaScript, odds are you don’t fully understand the language. As part of the 'You Don’t Know JS' series, this compact guide focuses on new features available in ECMAScript 6 (ES6), the latest version of the standard upon which JavaScript is built.",

"website": "https://github.com/getify/You-Dont-Know-JS/tree/master/es6+&+beyond"

},

{

"isbn": "9781449325862",

"title": "Git Pocket Guide",

"subtitle": "A Working Introduction",

"author": "Richard E. Silverman",

"published": "2013-08-02T00:00:00.000Z",

"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 experience.",

"website": "https://chimera.labs.oreilly.com/books/1230000000561/index.html"

}

]

### *****Greater than ( > ) operator in JsonPath*****

Greater than operator will let you get all the values which are greater than the value on the left-hand side. Let us find all the books which have pages more than 460.

***JsonPath: $.books[?(@.pages > 460)]***

***Result will be:***

[

{

"isbn": "9781593275846",

"title": "Eloquent JavaScript, Second Edition",

"subtitle": "A Modern Introduction to Programming",

"author": "Marijn Haverbeke",

"published": "2014-12-14T00:00:00.000Z",

"publisher": "No Starch Press",

"pages": 472,

"description": "JavaScript lies at the heart of almost every modern web application, from social apps to the newest browser-based games. Though simple for beginners to pick up and play with, JavaScript is a flexible, complex language that you can use to build full-scale applications.",

"website": "https://eloquentjavascript.net/"

},

{

"isbn": "9781449337711",

"title": "Designing Evolvable Web APIs with ASP.NET",

"subtitle": "Harnessing the Power of the Web",

"author": "Glenn Block, et al.",

"published": "2014-04-07T00:00:00.000Z",

"publisher": "O'Reilly Media",

"pages": 538,

"description": "Design and build Web APIs for a broad range of clients—including browsers and mobile devices—that can adapt to change over time. This practical, hands-on guide takes you through the theory and tools you need to build evolvable HTTP services with Microsoft’s ASP.NET Web API framework. In the process, you’ll learn how design and implement a real-world Web API.",

"website": "https://chimera.labs.oreilly.com/books/1234000001708/index.html"

}

]

### *****Greater than equal to ( >= ) operator in JsonPath*****

Greater than equal will let you get all the values that are equal to or greater than value on the right-hand side. Let's get all the books with pages either greater than or equal to 460

***JsonPath: $.books[?(@.pages >= 460)]***

***Result will be:***

[

{

"isbn" : "9781593275846",

"title" : "Eloquent JavaScript, Second Edition",

"subtitle" : "A Modern Introduction to Programming",

"author" : "Marijn Haverbeke",

"published" : "2014-12-14T00:00:00.000Z",

"publisher" : "No Starch Press",

"pages" : 472,

"description" : "JavaScript lies at the heart of almost every modern web application, from social apps to the newest browser-based games. Though simple for beginners to pick up and play with, JavaScript is a flexible, complex language that you can use to build full-scale applications.",

"website" : "https://eloquentjavascript.net/"

},

{

"isbn" : "9781449365035",

"title" : "Speaking JavaScript",

"subtitle" : "An In-Depth Guide for Programmers",

"author" : "Axel Rauschmayer",

"published" : "2014-02-01T00:00:00.000Z",

"publisher" : "O'Reilly Media",

"pages" : 460,

"description" : "Like it or not, JavaScript is everywhere these days-from browser to server to mobile-and now you, too, need to learn the language or dive deeper than you have. This concise book guides you into and through JavaScript, written by a veteran programmer who once found himself in the same position.",

"website" : "https://speakingjs.com/"

},

{

"isbn" : "9781449337711",

"title" : "Designing Evolvable Web APIs with ASP.NET",

"subtitle" : "Harnessing the Power of the Web",

"author" : "Glenn Block, et al.",

"published" : "2014-04-07T00:00:00.000Z",

"publisher" : "O'Reilly Media",

"pages" : 538,

"description" : "Design and build Web APIs for a broad range of clients\u2014including browsers and mobile devices\u2014that can adapt to change over time. This practical, hands-on guide takes you through the theory and tools you need to build evolvable HTTP services with Microsoft\u2019s ASP.NET Web API framework. In the process, you\u2019ll learn how design and implement a real-world Web API.",

"website" : "https://chimera.labs.oreilly.com/books/1234000001708/index.html"

}

]

In the next chapters, we will use ***JsonPath*** in ***Rest-Assured*** and see how we can write efficient validations.

# Deserialize JSON Array to List

Category: [**Rest-Assured**](https://toolsqa.com/categories/rest-assured) , November 10 2021

0

0

4 min read

**Author**



Virender Singh

**Reviewers**



Lakshay Sharma

**[Previous Lesson](https://toolsqa.com/rest-assured/expressions-in-jsonpath/)**

[Expressions in JSONPath](https://toolsqa.com/rest-assured/expressions-in-jsonpath/)

**[Next Lesson](https://toolsqa.com/rest-assured/deserialize-json-array-to-an-array/)**

[Deserialize JSON Array to an Array](https://toolsqa.com/rest-assured/deserialize-json-array-to-an-array/)

In this tutorial we will talk about the advance usage of ***JSONPath in Rest Assured*** which is ***How to DeSerialize JSON Array to List.*** If you have reached here directly, I would suggest that you go through these links before starting with this.

* [***REST API Tutorial***](https://toolsqa.com/rest-assured-tutorial/)
* [***Read JSON Response***](https://toolsqa.com/rest-assured/read-json-response-body-using-rest-assured/)
* [***Deserialize JSON Response to Class Object***](https://toolsqa.com/rest-assured/deserialize-json-response/)

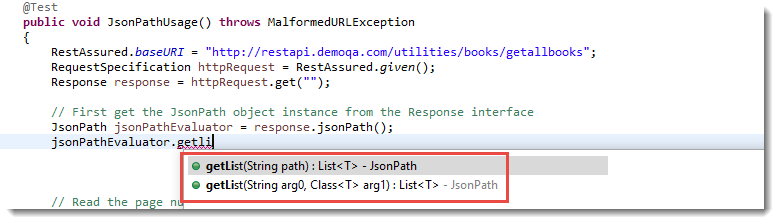
## DeSerialize JSON Array to List

Earlier we saw how ***JSONPath*** can be used to make a deterministic and accurate search in the ***Response JSON***. With this capability, we are able to write better test validations.

In this chapter, we will focus on how we can perform an advance search in the Response JSON by ***Deserialize JSON Array to List***. We will also learn how we can represent the searched section of ***Response*** in various Java data structures. This tutorial is structured based on different methods available in ***JSONPath*** class.

### *****DeSerialize JSON Array to List of String using JSONPath*****

***JsonPath*** class has two overloaded methods (jsonPath.getList) to extract JSON nodes as a Java List. Here are the methods



* ***JsonPath.getList***(String) method lets us get the searched nodes as a List of String
* ***JsonPath.getList***(String, Class T) method let us get the searched nodes as a List of T type

To understand we will make a Get call to the [***https://bookstore.demoqa.com/swagger/#/BookStore/BookStoreV1BooksGet***](https://bookstore.demoqa.com/swagger/#/BookStore/BookStoreV1BooksGet) REST service. This service returns a JSON response consisting of a collection of books(Array of Books). Each book in the collection is a JSON object containing values describing the book. You can directly open the URL in the browser to see the output.

To understand ***JsonPath.getList***(String) method we will try to retrieve all the books as a List of String ***(List<String>).*** All we need to do is give ***"books"*** as the ***JSONPath***.

***Below is the code***

@Test

public void JsonPathUsage() throws MalformedURLException

{

RestAssured.baseURI = "https://restapi.demoqa.com/utilities/books/getallbooks";

RequestSpecification httpRequest = RestAssured.given();

Response response = httpRequest.get("");

// First get the JsonPath object instance from the Response interface

JsonPath jsonPathEvaluator = response.jsonPath();

// Read all the books as a List of String. Each item in the list

// represent a book node in the REST service Response

List<String> allBooks = jsonPathEvaluator.getList("books.title");

// Iterate over the list and print individual book item

for(String book : allBooks)

{

System.out.println("Book: " + book);

}

}

Once you run this, the output will be look like this:

Book: Eloquent JavaScript, Second Edition

Book: Learning JavaScript Design Patterns

Book: Speaking JavaScript

Book: Programming JavaScript Applications

Book: Understanding ECMAScript 6

Book: You Don't Know JS

Book: Git Pocket Guide

Book: Designing Evolvable Web APIs with ASP.NET

### *****DeSerialize JSON Array to List of Class (Type T) Object using JSONPath*****

We all are familiar with ***Serialization*** and ***De-serialization*** support built-in in Rest-Assured. Let us try to convert the searched nodes directly into an object representation. In this example let us retrieve all the books from the JSON response. We will retrieve all the books as a List of Books class. In order to do this, we will first create a ***Class Representation of a Book***. Get all the properties of JSON Book entity and create a class with those member variables. A book can be simply represented by a ***POJO*** class as shown in the code below.

public class Book {

String isbn;

String title;

String subtitle;

String author;

String published;

String publisher;

int pages;

String description;

String website;

}

Now to get the response converted into a List of Books we will simply use the ***JsonPath.getList***(String, Class T) method.

* First parameter is the ***JSONPath ("books")*** in this case
* Second parameter is the Type name to which we want the response to be converted. ***Book*** class in this case.

Below is the complete  code showing the usage

@Test

public void JsonPathUsage() throws MalformedURLException

{

RestAssured.baseURI = "https://restapi.demoqa.com/utilities/books/getallbooks";

RequestSpecification httpRequest = RestAssured.given();

Response response = httpRequest.get("");

// First get the JsonPath object instance from the Response interface

JsonPath jsonPathEvaluator = response.jsonPath();

// Read all the books as a List of String. Each item in the list

// represent a book node in the REST service Response

List<Book> allBooks = jsonPathEvaluator.getList("books", Book.class);

// Iterate over the list and print individual book item

// Note that every book entry in the list will be complete Json object of book

for(Book book : allBooks)

{

System.out.println("Book: " + book.title);

}

}

This is a very good feature of Rest-Assured that enables us to get a targeted part of the response.

# Deserialize JSON Array to an Array

Category: [**Rest-Assured**](https://toolsqa.com/categories/rest-assured) , November 10 2021

0

0

6 min read

**Author**



Virender Singh

**Reviewers**



Lakshay Sharma

**[Previous Lesson](https://toolsqa.com/rest-assured/deserialize-json-array/)**

[Deserialize JSON Array to List](https://toolsqa.com/rest-assured/deserialize-json-array/)

**[Next Lesson](https://toolsqa.com/rest-assured/api-documentation/)**

[What is API Documentation, and Why It Matters?](https://toolsqa.com/rest-assured/api-documentation/)

Let us learn ***How to Deserialize JSON Array to an Array?*** But before we start, it is very important that you go through the previous set of Tutorials. You can take a look at the tutorials here:

* [***REST API  Automation Testing with Rest-Assured***](https://toolsqa.com/rest-assured/rest-api-test-using-rest-assured/)

## Deserialize JSON Array to an Array

This tutorial further builds upon our understanding of ***Deserialization*** of a ***JSON Response*** into an object of a given Type. Please go through the basics of ***Serialization and Deserialization*** in the below tutorial

* [***Deserialize JSON Data to POJO Class Object***](https://toolsqa.com/rest-assured/deserialize-json-response/)
* [***Deserialize JSON Array to List***](https://toolsqa.com/rest-assured/deserialize-json-array/)

We will take an example of demo REST API endpoint: [***https://bookstore.demoqa.com/swagger/#/BookStore/BookStoreV1BooksGet***](https://bookstore.demoqa.com/swagger/#/BookStore/BookStoreV1BooksGet) If you hit this Endpoint you will get a **Collection** of books in the Json response. Below is the response:

{

"books": [

{

"isbn": "9781593275846",

"title": "Eloquent JavaScript, Second Edition",

"subtitle": "A Modern Introduction to Programming",

"author": "Marijn Haverbeke",

"published": "2014-12-14T00:00:00.000Z",

"publisher": "No Starch Press",

"pages": 472,

"description": "JavaScript lies at the heart of almost every modern web application, from social apps to the newest browser-based games. Though simple for beginners to pick up and play with, JavaScript is a flexible, complex language that you can use to build full-scale applications.",

"website": "http:\/\/eloquentjavascript.net\/"

},

{

"isbn": "9781449331818",

"title": "Learning JavaScript Design Patterns",

"subtitle": "A JavaScript and jQuery Developer's Guide",

"author": "Addy Osmani",

"published": "2012-07-01T00:00:00.000Z",

"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-date with the latest best practices, this book is for you.",

"website": "http:\/\/www.addyosmani.com\/resources\/essentialjsdesignpatterns\/book\/"

},

{

"isbn": "9781449365035",

"title": "Speaking JavaScript",

"subtitle": "An In-Depth Guide for Programmers",

"author": "Axel Rauschmayer",

"published": "2014-02-01T00:00:00.000Z",

"publisher": "O'Reilly Media",

"pages": 460,

"description": "Like it or not, JavaScript is everywhere these days-from browser to server to mobile-and now you, too, need to learn the language or dive deeper than you have. This concise book guides you into and through JavaScript, written by a veteran programmer who once found himself in the same position.",

"website": "http:\/\/speakingjs.com\/"

},

{

"isbn": "9781491950296",

"title": "Programming JavaScript Applications",

"subtitle": "Robust Web Architecture with Node, HTML5, and Modern JS Libraries",

"author": "Eric Elliott",

"published": "2014-07-01T00:00:00.000Z",

"publisher": "O'Reilly Media",

"pages": 254,

"description": "Take advantage of JavaScript's power to build robust web-scale or enterprise applications that are easy to extend and maintain. By applying the design patterns outlined in this practical book, experienced JavaScript developers will learn how to write flexible and resilient code that's easier-yes, easier-to work with as your code base grows.",

"website": "http:\/\/chimera.labs.oreilly.com\/books\/1234000000262\/index.html"

},

{

"isbn": "9781593277574",

"title": "Understanding ECMAScript 6",

"subtitle": "The Definitive Guide for JavaScript Developers",

"author": "Nicholas C. Zakas",

"published": "2016-09-03T00:00:00.000Z",

"publisher": "No Starch Press",

"pages": 352,

"description": "ECMAScript 6 represents the biggest update to the core of JavaScript in the history of the language. In Understanding ECMAScript 6, expert developer Nicholas C. Zakas provides a complete guide to the object types, syntax, and other exciting changes that ECMAScript 6 brings to JavaScript.",

"website": "https:\/\/leanpub.com\/understandinges6\/read"

},

{

"isbn": "9781491904244",

"title": "You Don't Know JS",

"subtitle": "ES6 & Beyond",

"author": "Kyle Simpson",

"published": "2015-12-27T00:00:00.000Z",

"publisher": "O'Reilly Media",

"pages": 278,

"description": "No matter how much experience you have with JavaScript, odds are you don\u2019t fully understand the language. As part of the 'You Don\u2019t Know JS' series, this compact guide focuses on new features available in ECMAScript 6 (ES6), the latest version of the standard upon which JavaScript is built.",

"website": "https:\/\/github.com\/getify\/You-Dont-Know-JS\/tree\/master\/es6%20&%20beyond"

},

{

"isbn": "9781449325862",

"title": "Git Pocket Guide",

"subtitle": "A Working Introduction",

"author": "Richard E. Silverman",

"published": "2013-08-02T00:00:00.000Z",

"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 experience.",

"website": "http:\/\/chimera.labs.oreilly.com\/books\/1230000000561\/index.html"

},

{

"isbn": "9781449337711",

"title": "Designing Evolvable Web APIs with ASP.NET",

"subtitle": "Harnessing the Power of the Web",

"author": "Glenn Block, et al.",

"published": "2014-04-07T00:00:00.000Z",

"publisher": "O'Reilly Media",

"pages": 538,

"description": "Design and build Web APIs for a broad range of clients\u2014including browsers and mobile devices\u2014that can adapt to change over time. This practical, hands-on guide takes you through the theory and tools you need to build evolvable HTTP services with Microsoft\u2019s ASP.NET Web API framework. In the process, you\u2019ll learn how design and implement a real-world Web API.",

"website": "http:\/\/chimera.labs.oreilly.com\/books\/1234000001708\/index.html"

}

]

}

This ***JSON Response*** contains an ***Array of Books***. Each item in the JSON Array represents a book and has properties of a book like "***isbn***", "***title"***, "***author***" etc. In the earlier tutorial, we learnt How to ***Deserialize JSON Resposne of Single Node to an Instance of Class***. However, in ***Deserializing a Collection*** of Nodes into Array becomes little tricky. Let us see how Rest Assured helps us achieve this quickly and without writing any boilerplate code.

### *****Deserialize JSON Array to an Array using JSONPath of Rest Assured?*****

Similarly, we can convert a ***Json Array into a Java Array. JsonPath*** class has method called ***getObject***. This method can be used to convert the response directly into a ***Java Array of Book***. The only thing the we need to do is pass ***Book[].class*** as the second argument to the method to signify that we want the Json to be deserialized into an Array of Book. Here is the code that will do this

@Test

public void JsonArrayToArray()

{

RestAssured.baseURI = "https://restapi.demoqa.com/utilities/books/getallbooks";

RequestSpecification request = RestAssured.given();

Response response = request.get();

System.out.println("Response Body -> " + response.body().asString());

// We can convert the Json Response directly into a Java Array by using

// JsonPath.getObject method. Here we have to specify that we want to

// deserialize the Json into an Array of Book. This can be done by specifying

// Book[].class as the second argument to the getObject method.

Book[] books = response.jsonPath().getObject("books",Book[].class );

for(Book book : books)

{

System.out.println("Book title " + book.title);

}

}

The above two techniques are important to write ***Concise*** test code. Apart from writing concise tests by using the above techniques we fully utilize the features given to us by Rest-Assured. This helps us have lesser dependencies outside Rest Assured and also enables us to write reliable tests.

# What is API Documentation, and Why It Matters?

Category: [**Rest-Assured**](https://toolsqa.com/categories/rest-assured) , November 10 2021

0

0

13 min read

**Author**



Yashraj Gore

**Reviewers**



Lakshay Sharma

**[Previous Lesson](https://toolsqa.com/rest-assured/deserialize-json-array-to-an-array/)**

[Deserialize JSON Array to an Array](https://toolsqa.com/rest-assured/deserialize-json-array-to-an-array/)

**[Next Lesson](https://toolsqa.com/rest-assured/rest-api-end-to-end-test/)**

[REST API End to End Test](https://toolsqa.com/rest-assured/rest-api-end-to-end-test/)

Automation has a crucial position in this technology-centric world. Faster and short releases are in vogue. In addition to that, the agile development method has gained a foothold in the software industry. There has been a remarkable change in the way we develop software automation tests in agile.

## Rising Need For API Testing

The old practice of having only the GUI tests for automation does not exist anymore. Moreover, the ***problems in software testing with the GUI tests*** are several. To enumerate a few:

* The tests may fail frequently owing to frequent changes in the UI
* Additional maintenance and refactoring efforts associated with flaky tests
* Moreover, its a time-consuming test process and has slower feedback.
* It requires a large number of initiatives in terms of maintainability and stability.

In a two week aligned [***sprint***](https://www.toolsqa.com/agile/scrum/sprint/), it gets altogether challenging to have a UI ready and test it simultaneously for test coverage results. Unlike the GUI tests, the API testing doesn't rely on the UI. It becomes much more comfortable to conjoin/ integrate them under [***Agile Development***](https://www.toolsqa.com/agile/what-is-agile/). Additionally, it can be brought in much earlier at the development stage. The increasing/ [***rising trend***](https://trends.google.com/trends/explore?date=today%205-y&q=%2Fm%2F0139ygq7) of API Testing underlines this further.

An API abstracts the implementation layer, exposing only the objects needed by the developer. It consists of REST methods. Moreover, these REST methods fetch, manipulate, and delete the data in the application's database. We need to familiarize ourselves with the workings of REST API basics. Please go through the [***REST API  tutorials***](https://www.toolsqa.com/rest-assured-tutorial/) to be acquainted with those.

Once we have familiarised ourselves with the REST API basics, it is time to learn to test the REST APIs. To test a REST API, we must understand the interface of the API. This includes:

* The operations we can perform using the API
* Structure of the requests
* Structure of responses

Therefore, an essential step in that direction would be to look at the ***API Documentations***. In this article, we are going to cover:-

* Why do we need an API Documentation?
* What is an API Documentation?
* How to use API Document?
* What are the various types of tests for API?

Let us first understand the benefits of API Documentation.

## Why do we need an API Documentation?

Documentation of any technology has a direct impact on the adoption and its usage. Without the knowledge of the tool, no one would know how to use it. Moreover, there are other advantages to it as listed below:

* It provides a quick and easy guide for API consumption.
* Reduces user frustrations.
* Demonstrates useful functions.
* Saves support costs and time.

These are some of the benefits of API documentation. Let us learn and understand what all consists of an API Documentation.

## What is an API Documentation?

API documents are similar to a reference manual. It talks about the necessary information of an API, in terms of,

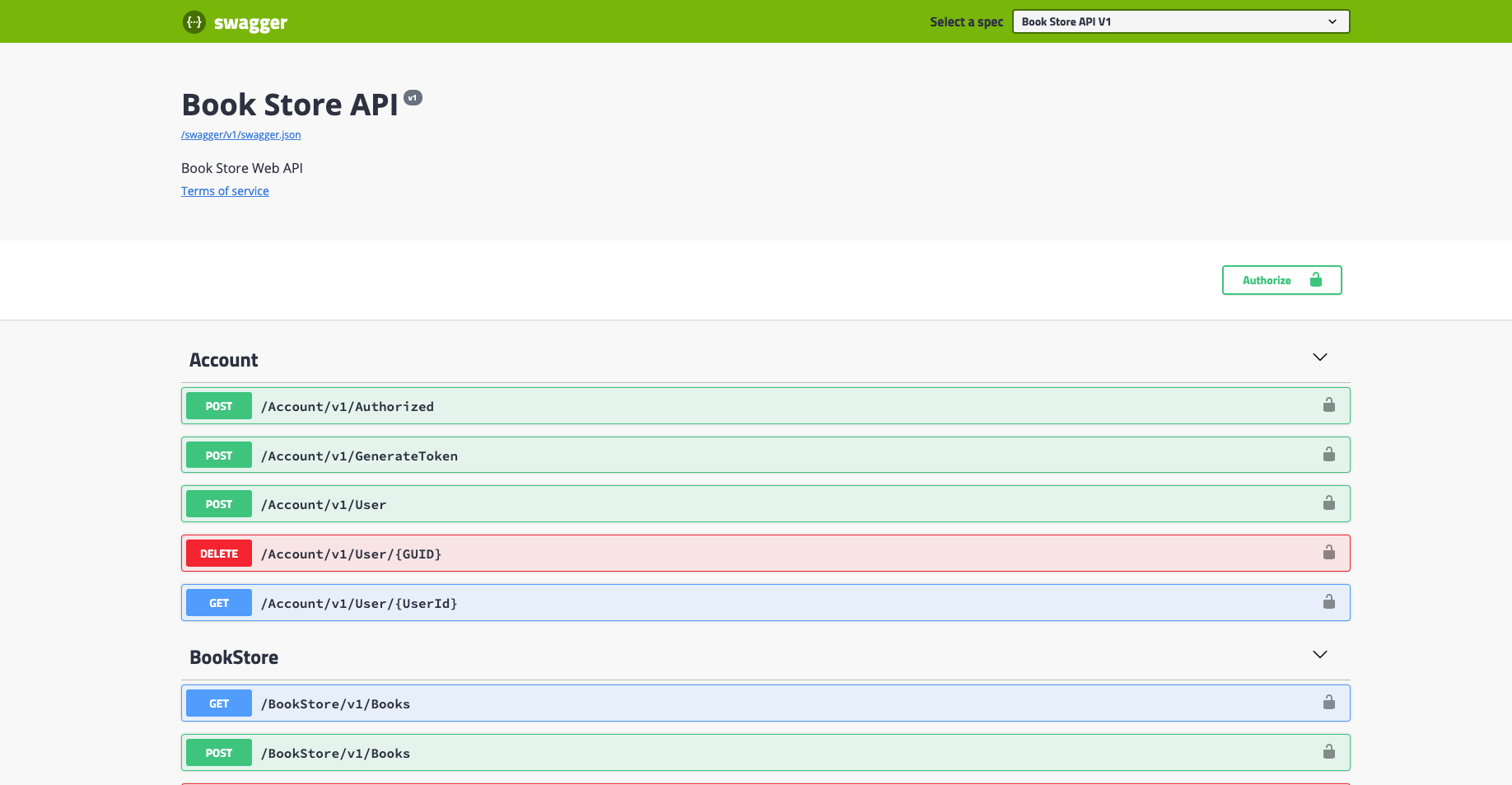
### *****Description of Resources*****

The data returned by the API are resources. Various endpoints can be used to access resources. Under the same resource, an API will have several routes grouped. Resources and endpoint descriptions are generally short and precise. Additionally, the user guide contains a piece of more detailed information.

### *****Methods and their endpoints*****

The endpoint is an essential part of the API documentation. The developers implement it to make the requests. Endpoints indicate ways to access resources. The methods indicate permissible interactions such as GET, PUT, POST, or DELETE with the resource. Endpoints, similar to overall resource descriptions, as well have short brief descriptions. The endpoint shows the end path of a resource. It does not include a common base path to all endpoints.

Consider a sample APIs collection available to us as [***TOOLSQA - Bookstore API***](https://demoqa.com/swagger/) ***provided by TOOLSQA***. Please navigate to the URL. Consequently, once the page opens, we will see this:



In the above image:

Bookstore and Account are **resources** available in Bookstore API

An **HTTP method** acts as a verb. The listed verbs POST, GET, DELETE in the above image are some of the examples. Moreover, there are others as well, like PUT, PATCH, OPTION, etc.

The **routes** in the above image are:

1. /Account/v1/Authorized
2. /Account/v1/GenerateToken
3. /Account/v1/User
4. /Account/v1/User/{GUID}
5. /Account/v1/User/{UserId}
6. /BookStore/v1/Books
7. /BookStore/v1/Books

The routes are following the method to form an **endpoint.** Example, ***GET /Account/v1/Authorized***.

***Note:*** We have a very few endpoints listed here in the sample [***bookstore API URL.***](https://demoqa.com/swagger/) But in actual projects, the list of endpoints can be much large.

For an endpoint, the full resource URL needn't be listed in an API document. So, in our example, ***GET /Account/v1/Authorized*** is listed instead of ***GET  http://bookstore.toolsqa.com/Account/v1/Authorized***

It is done with the thought to make the users focus on the path. Additionally, the user guide provides the full resource URL and the needed authorization in the introductory section.

### *****Parameters used*****

Parameters are the options that control the resource. There are four kinds of parameters:

1. ***Header parameter:*** As the name suggests, these are the parameters passed into the request headers. They are generally related to the authorization and included under a separate section of authorization requirements for standard header parameters. Additionally, in the case of unique parameters passed in the headers, they are documented with the respective endpoint.
2. ***Path parameter:*** Path parameters are part of the endpoint. They are not optional. We usually write them along with curly braces. For example,  ***/Account/v1/User/{UserId}***  Here, ***{UserId}*** is the path parameter.
3. ***Query parameter:*** They are attached to the URL end. Additionally, the Query Parameter is appended to the URL after adding '?' at the end of the URL. You can read more in the [***Query Parameter tutorial***](https://www.toolsqa.com/rest-assured/query-parameters-in-rest-assured/).
4. ***Request body parameters:*** In a POST request, several times, a JSON object is submitted in the request body. Moreover, it is a list of key-value pairs.

***Example:***

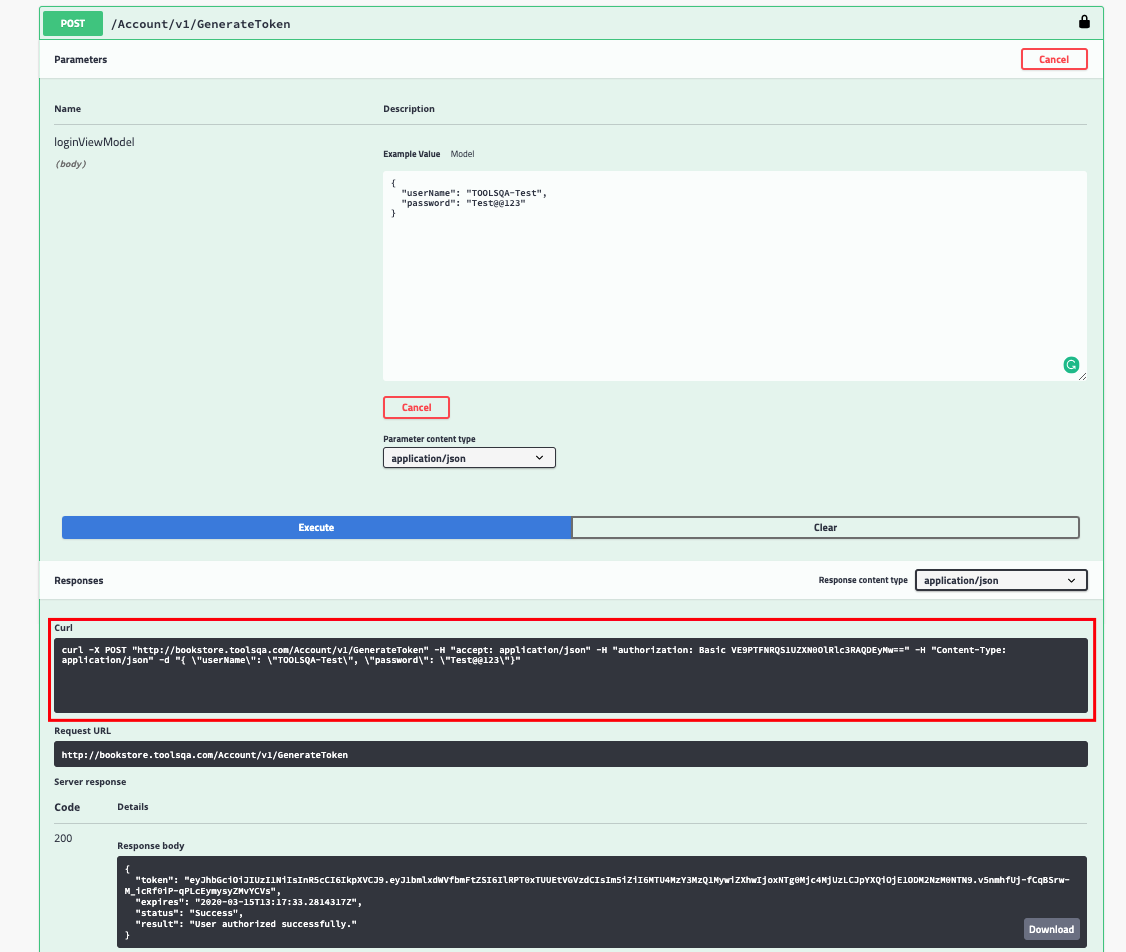
{

"username": "TOOLSQA-Test",

"password": "Test@@123"

}

### *****Examples of Requests and Responses*****



In the above example, we passed the username and password as request body parameters. We got a 200 status code response along with the response body for the POST request of the endpoint ***/Account/v1/GenerateToken***, we sent.

Alongside, if you notice the highlighted section, it is that of a curl command.

curl -X POST "https://bookstore.toolsqa.com/Account/v1/GenerateToken" -H "accept: application/json" -H "authorization: Basic VE9PTFNRQS1UZXN0OlRlc3RAQDEyMw==" -H "Content-Type: application/json" -d "{ \"userName\": \"TOOLSQA-Test\", \"password\": \"Test@@123\"}"

We follow Curl format for several reasons:

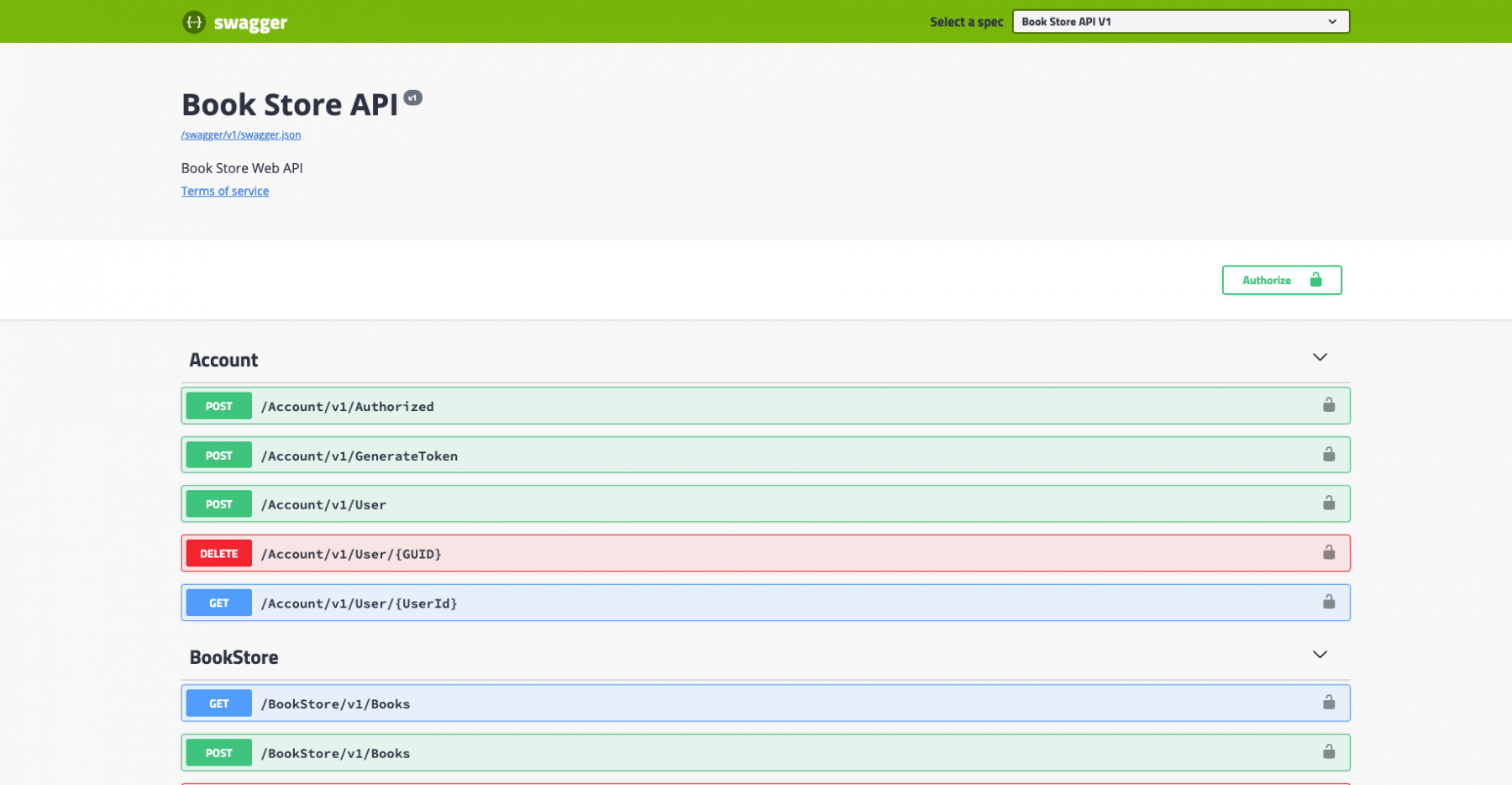
* Firstly, curl is independent of the programming language.
* Secondly, it consists of the request header information.
* Additionally, it displays the Request method.

Thus, we have understood the contents of a typical API document. An API document may contain additional information. But the above part of the tutorial gives us a general idea of everything included under an API document. Subsequently, in our next stage, we will learn the usage of an API document.

## How to use an API Document?

To understand the use of Swagger API documentation, we will use the same [***Bookstore API***](https://demoqa.com/swagger/) used previously. It consists of a sandbox environment that tests the APIs on the documentation.

Navigate to the ***Bookstore API.*** You will see the following:



The grouping of endpoints is as follows:

* Account
* BookStore

Some of those API Endpoints listed below build our End to End Rest API Framework. Please try these examples once you have learned to use the API document at the end of this section.

| ***HTTP VERB*** | ***Route*** | ***PURPOSE*** |
| --- | --- | --- |
| ***POST*** | /Account/v1/GenerateToken | To generate token as the endpoint name suggests |
| ***GET*** | /Account/v1/User | To create a userId with associated user |
| ***GET*** | /BookStore/v1/Books | To fetch a list of books |
| ***POST*** | /BookStore/v1/Books | To add a book associated with the user |
| ***DELETE*** | /BookStore/v1/Books | To remove a book |
| ***GET*** | /Account/v1/User/{UserId} | To get a list of books associated with userId |

We listed some of the APIs below from the ***bookstore API***. They will help us to understand the requests and their response bodies using Swagger.

1. To Create User
2. To Generate Token with the created User

### *****API Documentation - Create User - POST*****

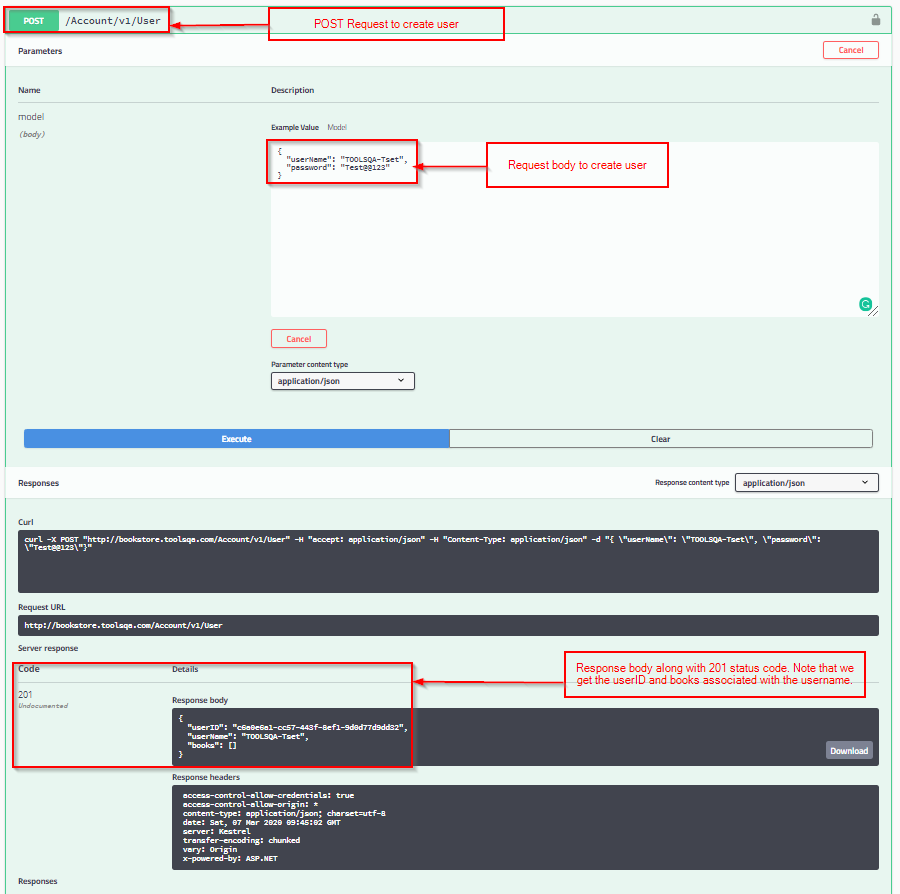
Before we begin exploring the APIs, we need to create a user with a ***username*** and ***password***. Subsequently, add a desired username and password under Book Store Credentials.

***Note***: Please choose your username and password, and once created, remember to use the same username and password for all the tests. For example, throughout we will be using ***TOOLSQA-Test*** as username and ***Test@@123*** as password.

Now, let's try and create a userID using the POST  Request for User.

***Make a request***

1. First, Expand the POST method for the endpoint under Account: ***/Account/v1/User***
2. Second, click the ***Try it out*** button. The Request Body field gets editable.
3. Third, add the username and password. You have added now, as indicated in the below screenshot. After that, select the parameter ***content type as application/json***. Click on the ***Execute*** button.
4. Finally, Swagger submits the request. It shows the curl that was submitted. The Response, for the request, is observed in the response section.



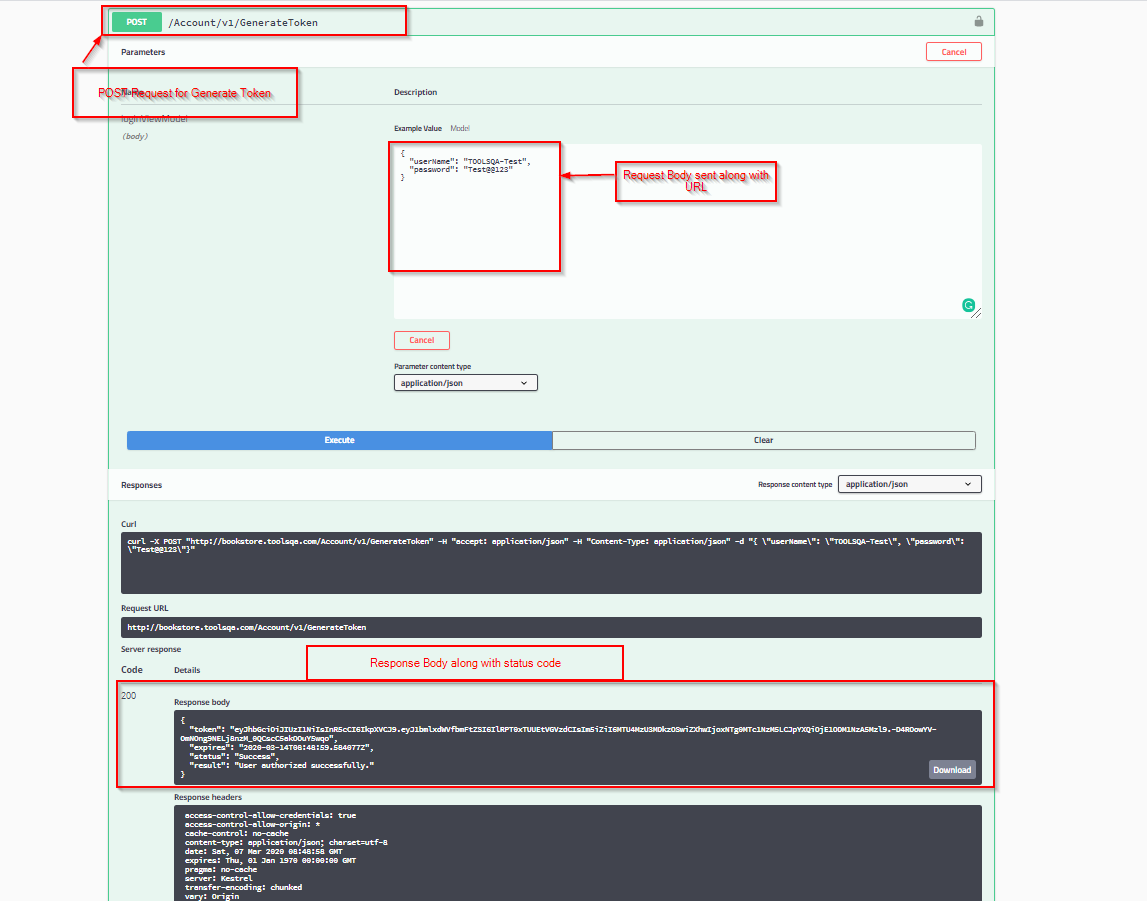
***Note:*** Please note down the userID. You will need it for other APIs in the request body.

***Explanation:***  We passed a ***POST Request*** with username and password in JSON format. We got a ***201 status code*** along with userID, and books associated with the respective username in the ***Response body***. It explains that our request was successfully sent over the server, as you can see in the above image under Description. Additionally, Swagger's implementation is simple and useful. It provides an interactive experience to send a request and receive a response for APIs.

As software professionals, we need to understand the underpinnings on which great software builds. One of the many means is to try and test them over in various ways. We familiarized ourselves with the API Documentation. Additionally, Swagger provided us with the means to test it on [***Swagger UI***](https://swagger.io/tools/swagger-ui/) itself.

### *****API Documentation - Generate Token - POST*****

Now that we have created a user in our previous response let's try and create a token using the POST  Request for User.



***Explanation:***  We sent our request with the body containing a username and password in the ***application/json*** format. When we click the ***Execute*** button, we got a 200 status code. Additionally, we also got a response body with the token, expires, status, and result fields. In the next section, we will understand the context of API tests and study its various types.

## What are the various types of API Tests?

Before we commence actual testing of the APIs, we need to compile associated information of the APIs. The context in which we carry out testing is essential because it drives our testing efforts.

Below is a non-comprehensive list of the information we can seek before we start testing the APIs:

* Expected response code from a successful request
* Expected response code from an unsuccessful request
* Fields to be validated
* Parameters required
* Error handling of unsuccessful requests
* HTTP verbs that we can use with the endpoints

Let us understand, under which all categories the API tests can be divided. It is not to say that we have enlisted a comprehensive list; there could be more types we can add to the list. But more or less, the API tests fall under one of the categories mentioned below.

1. ***Validation Testing:*** It forms one of the last segments in the software development process. It caters to the testing explicitly done to assess the product; its behavior, and its efficiency. The API is examined for correctness in implementation as per the expectations. Additionally, the API is also verified based on pre-established agreed-upon criteria such as the delivery of specific end goals, integration with the specified environment. In addition to that, the testing of API's behavior to access the correct data happens as a part of Validation testing.
2. ***Functional Testing:*** This is one of the broader types of testing. It aims to test specified functions. While testing the APIs for functional testing, the assessment of the responses received happens against expected responses in terms of passing parameters. Additionally, the error handling for unsuccessful requests and invalid responses too are under consideration under these tests. The boundary cases, along with regular tests, fall under the scope of these tests.
3. ***UI Testing:*** UI Tests tend to be more specific. These tests assess the user interface against the APIs and their constituent parts. Additionally, the tests are more generalized with a focus on the API health; it's usability as well as compatibility of the front end and back end.
4. ***Load Testing:*** This test is for the functionality and performance of the APIs under load. The API undergoes theoretical regular traffic. The test results form the baseline to conduct further load testing. Additionally, the testing also includes subjecting the API to maximum possible traffic to test the behavior of the API under full loads. APIs undergo overloading tests. In this test, the verification of the API performance and error handling conditions during failure happens.
5. ***Error Detection:*** The tests for APIs, which include the monitoring, inducing execution errors, sending invalid requests, detecting operational leaks, etc. fall under this category.  The introduction of known failure scenarios in the APIs happens. The testing of these APIs happens to ensure that the errors are detected, handled as well as routed.
6. ***API Security tests:*** Specific tests verify the APIs performance for vulnerabilities from external threats. Security testing, Penetration testing, and Fuzz testing are some of them. For example, ***Security testing*** involves validation of the APIs in terms of security requirements. Additionally, these security requirements could be related to permissions, authorizations, authentications, etc.  An authorized attack launches against the system as a part of ***Penetration testing***. It evaluates the security of the API. A full-scale assessment report of strengths and weaknesses of the APIs is the deliverable issued after such a test. In ***Fuzz testing***, evaluation of API behavior happens. It is subjected to unexpected, invalid, and random data of enormous size. Following this, the crashes, built-in assertions, and memory leaks are known.
7. ***Integration testing:*** The Integration tests focus on API communication with another module APIs.
8. ***Reliability testing:*** The API should display a prompt response for different configurations. It also looks for a response data structure as a part of testing.

To conclude, in this post, we have got an understanding of API documentation. We learned to use the Swagger API document. Additionally, we acquainted ourselves with types of API testing.

In further posts, we will learn to build [***REST API Automation Framework***](https://toolsqa.com/rest-assured/rest-api-test-using-rest-assured/) around the API tests. Try out using the API document as guided above and reach out for your valuable feedback.

# REST API End to End Test

Category: [**Rest-Assured**](https://toolsqa.com/categories/rest-assured) , November 10 2021

0

0

10 min read

**Author**



Yashraj Gore

**Reviewers**



Lakshay Sharma

**[Previous Lesson](https://toolsqa.com/rest-assured/api-documentation/)**

[What is API Documentation, and Why It Matters?](https://toolsqa.com/rest-assured/api-documentation/)

**[Next Lesson](https://toolsqa.com/rest-assured/rest-api-test-in-cucumber/)**

[REST API Test in Cucumber](https://toolsqa.com/rest-assured/rest-api-test-in-cucumber/)

In this tutorial, we will learn to write ***REST API End to End Test.***

We carefully laid the foundations of our API automation framework components. We will be heavily relying on our previous knowledge gained for ***Rest Assured, Cucumber*** as well as ***Maven***. Subsequently, these will form the base over which we will build our API Automation Framework. I will be referring to them every once in a while and adding useful links as we need more of them. It's advisable to clear up the basics by going through the tutorials:

1. [***Rest Assured***](https://www.toolsqa.com/rest-assured-tutorial/)
2. [***Cucumber***](https://www.toolsqa.com/cucumber-tutorial/)
3. [***Maven***](https://www.toolsqa.com/maven/maven-introduction/)

In our automation framework, we will automate an End to End business flow of a use case. Additionally, it will help to demonstrate the right examples of framework components and explain their usages. Besides, our understanding of building the ***Rest Assured API Automation Framework*** from scratch will improve as we go about this tutorial.

Many-a-times we are involved in getting ready-made frameworks that are built by Senior Test Architects or Software Developers. Understanding the underlying principles and design ideas are crucial to develop our skills as software engineers.

## Prerequisites for REST API End to End Test

1. Java Setup
2. IDE Setup
3. Maven Setup
4. Create a maven project
5. Add Rest Assured Dependencies
6. Setup Maven Compiler Plugin
7. Create a user for the test

### *****Step 1 - Java Setup*****

We will use Java as our language, for writing our REST API automation framework based on the Rest Assured library. For this, we will need to install Java on our machines if not previously installed. Likewise, please follow through the tutorial to [***install Java***](https://www.toolsqa.com/selenium-webdriver/install-java/) as our first prerequisite.

### *****Step 2 - IDE Setup*****

As we will be working with Java, we will need an editor to use for Java. ***Eclipse, IntelliJ, Net Beans,*** and several others are popular IDEs you can choose to work. Furthermore, a tutorial on installing [***Eclipse on Windows***](https://www.toolsqa.com/selenium-webdriver/download-and-start-eclipse/) and for [***Mac***](https://toolsqa.com/maven/how-to-install-maven-on-mac/) users exists to ease the process. Please pick an IDE you are comfortable working with and get going.

### *****Step 3 - Maven Setup*****

Build tools enable us to create executable applications from the source code. They help to automate and script everyday activities like downloading dependencies, compiling, running our tests, and deployments. Moreover, we will use the Maven build tool for our End To End Scenarios. We have created a tutorial explaining the [***installation of Maven on Windows***](https://toolsqa.com/maven/how-to-install-maven-on-windows/). Additionally, please install Maven, if not previously installed as we would need it.

### *****Step 4: Create a New Maven Project*****

Eclipse operates with workspaces, folders, and spaces where you add your projects. Software Teams use different approaches for workspaces and project structures but try to follow and stick with the default structure.

In addition to this, to create a new maven project, please follow our article [***Steps to create a New Maven Project***](https://toolsqa.com/maven/create-new-maven-project/). Just to make sure that you specify below values to Maven Archetype Parameters:

* Group ID: ToolsQA
* Artifact ID: API TestingFramework

***Note***:  You can have any names as Group ID and Artifact ID. However, for the ease of this tutorial, it is better to keep the same naming conventions across the project. Also, it helps in resolving issues while copying pasting the code.

### *****Step 5 - Add Rest Assured Dependencies*****

We will add ***Rest Assured Dependencies*** to our project through the pom.xml file. To add the required dependencies, go to [***Rest Assured Maven Repository.***](https://mvnrepository.com/artifact/io.rest-assured/rest-assured) Then, select the latest dependency. Copy-paste it in the project ***pom.xml*** file.

<dependency>

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

<artifactId>rest-assured</artifactId>

<version>4.2.0</version>

<scope>test</scope>

</dependency>

Note: As of  Feb'2020, the latest rest-assured version is 4.2.0.

We should add the dependency tag within the dependencies tag, like below.

<dependencies>

<dependency>Project Dependency</dependency>

</dependencies>

Further, we will be using ***JUnit Dependency***. So please add that dependency in the pom.xml.

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.13</version>

<scope>test</scope>

</dependency>

***Note***: As of Feb'2020, the latest JUnit version is 4.13. Always use the latest Junit version.

### *****Step 6: 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. Additionally, to change the defaults, please set the source and target as described in Setting the –***source*** and –***target*** of the Java Compiler.

Below you can find the ***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>

The complete POM would turn out to be likewise:

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="https://maven.apache.org/POM/4.0.0"

xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="https://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>ToolsQA</groupId>

<artifactId>RestAssured\_APITests</artifactId>

<version>1.0-SNAPSHOT</version>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.13</version>

<scope>test</scope>

</dependency>

<dependency>

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

<artifactId>rest-assured</artifactId>

<version>4.2.0</version>

<scope>test</scope>

</dependency>

</dependencies>

<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>

</project>

### *****Step 7: Create an Authorized user for the test*****

Finally, before we begin automation of our tests, the last piece is remaining. It is of creating an authorized user. We need to create a user that will use our End to End automation tests. We have learned to create a user in our [***previous tutorial***](https://www.toolsqa.com/rest-assured/api-documentation/) under the POST Request. The [***POST Request tutorial***](https://www.toolsqa.com/rest-assured/post-request-using-rest-assured/) explains how a POST Request method sends data to the server.

S

@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");

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");

}

## Write REST API End to End Test:

Let's consider this scenario:-

***Test Scenario: As an existing authorized user, I retrieve a list of books available for me in the library. I will assign a book to myself and later on return it.***

We will divide the test scenario into below steps for simplicity:

***1. Test will start from generating Token for Authorization -*** First, we have the username and password of a registered user. Using these credentials, we will generate a token. Additionally, we will send this token into the Requests instead of the username and password. The reason we follow this practice is to have specific resources allocated in a time-bound manner. This step involves making a POST Request call in Rest Assured by passing username and password. Kindly follow this [***POST Request tutorial***](https://www.toolsqa.com/rest-assured/post-request-using-rest-assured/) to learn about how to send a POST Request.

***Note***: The below-mentioned User won't work, please create your user for practice.

***2. Get List of available books in the library -*** Secondly, it is a GET Request call. It gets us, the list of available books. Visit the [***GET Request tutorial***](https://www.toolsqa.com/rest-assured/rest-api-test-using-rest-assured/) to understand the logic of how we make a GET Request call in Rest-Assured.

***3. Add a book from the list to the user -*** The third is a POST Request call. We will send the user and book details in the Request.

***4. Delete the added book from the list of books -*** Fourth is a DELETE Request call. We will delete the added book from the list. The [***DELETE Request tutorial***](https://www.toolsqa.com/rest-assured/delete-request-using-rest-assured/) will help you with it.

***5. Confirm if the book removal happens successfully -*** Last but not least, as a verification step, we will verify if the book has got removed from the user. Therefore, we will send user details in a GET Request call to get details of the user.

### *****Create a Test Package & a Test Class*****

1. Firstly create a ***New Package*** by right click on the ***src/test/java*** package and select ***New >> Package***. Moreover, give your package a name ***apiTests*** and click ***Finish***.
2. Secondly, create a ***New Class*** file by right click on the ***src/test/java*** package and select ***New >> Class***. Give your test case the correct name in the resulting dialog and click ***Finish*** to create the file. Also, to make sure to check the ***public static void main,*** as we will be running the test from the same primary method. I have named the class as ***E2E\_Tests***, as you can see in the code snippet below.

### *****Write the complete REST API End to End Test*****

I have written an end to end test encompassing the steps as mentioned above. The example is relatively simple to follow through. Moreover, it involves the use of GET, POST, and DELETE Requests. It will benefit you if you go through the [***Rest Assured Tutorial***](https://www.toolsqa.com/rest-assured-tutorial/) at this point if you are not following.

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;

public class E2E\_Tests {

public static void main(String[] args) {

String userID = "9b5f49ab-eea9-45f4-9d66-bcf56a531b85";

String userName = "TOOLSQA-Test";

String password = "Test@@123";

String baseUrl = "https://bookstore.toolsqa.com";

RestAssured.baseURI = baseUrl;

RequestSpecification request = RestAssured.given();

//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");

//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");

//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());

//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());

//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());

}

}

***Note***: We added an import statement for JSONpath, ***import io.restassured.path.json.JsonPath***; It will help us to traverse through the specific parts of the JSON. You can read more in the ***JSONPath article***.

### *****Run the REST API Test*****

Next step, we need to execute the test. Right-click in the test body and select ***Run As >> Java Application***. The test will run, and you'll see the results in the Console. Consequently, the program executes successfully without any error. Just in case you happen to configure the same on IntelliJ IDE, the effect observed will be "***Process finished with exit code 0***". Additionally, it is another sign signifying that there were no errors and our tests executed successfully.

***Note***: It is not a UI based test. Moreover, there will not be any visual output to observe the test execution.

Conclusively, in the next tutorial, we will [***Convert our API tests into Cucumber BDD***](https://www.toolsqa.com/rest-assured-tutorial/) style. It will further our understanding of the way Cucumber Tests are structured.

# REST API Test in Cucumber

Category: [**Rest-Assured**](https://toolsqa.com/categories/rest-assured) , November 10 2021

0

0

9 min read

**Author**



Yashraj Gore

**Reviewers**



Lakshay Sharma

**[Previous Lesson](https://toolsqa.com/rest-assured/rest-api-end-to-end-test/)**

[REST API End to End Test](https://toolsqa.com/rest-assured/rest-api-end-to-end-test/)

**[Next Lesson](https://toolsqa.com/rest-assured/convert-json-to-java-object/)**

[Convert JSON Request Body to JAVA Object](https://toolsqa.com/rest-assured/convert-json-to-java-object/)

In our previous tutorial, we wrote a simple [***End to End Rest API Test***](https://toolsqa.com/rest-assured/rest-api-end-to-end-test/).  The business flow of a use case was converted into a simple API request and response format of ***GET, POST, and DELETE Requests***. Subsequently, our next step is to convert the ***REST API Test in Cucumber***. While we are at it, we need to develop an understanding of the ***Cucumber BDD Framework***. The [***Cucumber tutorials***](https://www.toolsqa.com/cucumber-tutorial/) are relatively simple to understand and would be a great primer before we deep dive.

And now, we start with our Rest API Test in Cucumber.

## REST API Test in Cucumber BDD Style Test

We will use the ***Cucumber BDD Framework*** to execute our tests. Additionally, it would require us to ***Convert our Rest Assured API Tests to the Cucumber BDD Style Test***.

The following steps will help us to achieve this:

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

### *****Step 1: Add Cucumber Dependencies to the Project*****

Firstly, add the following dependencies to our project to execute our tests in Cucumber:

* cucumber-java
* cucumber-jvm-deps
* cucumber-JUnit

As we are developing the framework in Maven, it would be useful to add the dependencies.  Therefore, add the following dependencies into the project POM XML.

***cucumber-java:*** The location of the dependency details is at the [***Maven Repository***]tps://mvnrepository.com/artifact/io.cucumber/cucumber-junit/5.2.0).

<dependency>

<groupId>io.cucumber</groupId>

<artifactId>cucumber-java</artifactId>

<version>5.2.0</version>

</dependency>

***Note***: As of Feb’2020, the latest cucumber-java version is 5.2.0.

***cucumber-jvm-deps:*** The location of the dependency details is at the [***Maven Repository***].(https://mvnrepository.com/artifact/io.cucumber/cucumber-jvm-deps)

<dependency>

<groupId>io.cucumber</groupId>

<artifactId>cucumber-jvm-deps</artifactId>

<version>1.0.6</version>

<scope>provided</scope>

</dependency>

***Note***: As of Feb’2020, the latest cucumber-jvm-deps version is 1.0.6

***cucumber-junit:*** The location of the dependency details is at the [***Maven Repository***](https://mvnrepository.com/artifact/info.cukes/cucumber-junit/1.2.5).

<dependency>

<groupId>io.cucumber<</groupId>

<artifactId>cucumber-junit</artifactId>

<version>5.2.0</version>

<scope>test</scope>

</dependency>

***Note***: As of Feb’2020, the latest cucumber-junit version is 5.2.0

Consequently, the complete pom.xml will look like this:

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="https://maven.apache.org/POM/4.0.0" xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="https://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>ToolsQA</groupId>

<artifactId>RestAssured\_APITests</artifactId>

<version>1.0-SNAPSHOT</version>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.13</version>

<scope>test</scope>

</dependency>

<dependency>

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

<artifactId>rest-assured</artifactId>

<version>4.2.0</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>io.cucumber</groupId>

<artifactId>cucumber-java</artifactId>

<version>5.2.0</version>

</dependency>

<dependency>

<groupId>io.cucumber</groupId>

<artifactId>cucumber-jvm-deps</artifactId>

<version>1.0.6</version>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>io.cucumber</groupId>

<artifactId>cucumber-junit</artifactId>

<version>5.2.0</version>

<scope>test</scope>

</dependency>

</dependencies>

<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>

</project>

***Note***: Update the POM by right click on the project root and select Maven ***>> Update Project***. This action will nullify the dependencies based issues.

***Additional note***: If you feel comfortable adding the jars in the project library instead of using Maven dependencies, that's alright as well.

### *****Step 2: Write a test in a Feature File*****

Secondly, we will highly recommend acquainting yourself with the tutorial on the [***Feature file***](https://mvnrepository.com/artifact/info.cukes/cucumber-junit/1.2.5). It will help in understanding the basics of the Cucumber feature file. Consequently, we will begin to convert our test scenario into the Cucumber Feature file.

In the previous chapter, we broke down the scenario into parts for us to easily convert the scenario into steps. Let's visit the scenario yet again,

1. The test will start by generating a Token for Authorization
2. Get List of available books in the library
3. Add a book from the list to the user
4. Delete the added book from the list of books
5. Confirm if the book removal is successful

***Conversion of above scenario to Cucumber BDD Style Test:***

* ***Background: User generates token for Authorisation***

***Given I am an authorized user***

* ***Scenario: the Authorized user can Add and Remove a book.***

***Given A list of books are available***

***When I add a book to my reading list***

***Then the book is added***

***When I remove a book from my reading list***

***Then the book is removed***

#### **Create Feature File**

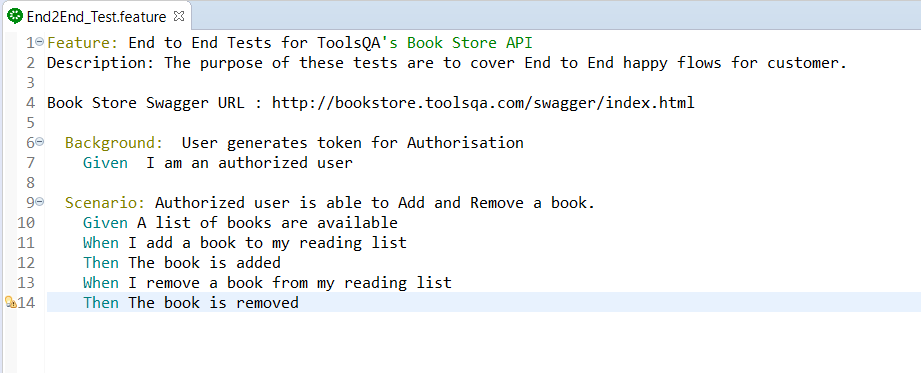
1. First, create a ***New Package*** and name it as ***functionalTests.*** You can do it by right-clicking on the ***src/test/resources*** and select ***New >> Package***.

***Note***: It's always recommendable to put all the feature files in the ***resources*** folder.

1. Secondly, create a **Feature** file and name it as ***End2End\_Test.feature*** by right click on the above created package and select ***New >> File***.

Add ***.feature*** extension to the file.

1. Lastly, add the test steps to the feature file.



***Note***: As you are aware, the keywords used in the test steps are in different colors. Those are the Gherkin keywords. Eclipse does not understand these. However, if we install the cucumber Eclipse plugin-in, this will be recognized. Please follow our tutorial to [***Install Cucumber Eclipse Plugin***](https://toolsqa.com/cucumber/install-cucumber-eclipse-plugin/).

### *****Step 3: Create a JUnit test runner*****

Thirdly, we will create a Cucumber Test Runner to execute our tests. Moreover, we will need a Cucumber Test Runner based on the JUnit Test Runner for our tests. To understand more about Cucumber Test Runner, please refer [***JUnit Test Runner***](https://www.toolsqa.com/cucumber/junit-test-runner-class/) for the tutorial.

1. Firstly, Right-click on the ***src/test/java*** and select a New Package by using ***New >> Package***. Name it as ***runners***.
2. After that, Create a New Java Class file and name it as ***TestRunner*** by right click on the above-created package and select ***New >> Class***.

package runners;

@RunWith(Cucumber.class)

@CucumberOptions(

features = "src/test/resources/functionalTests",

)

public class TestRunner {

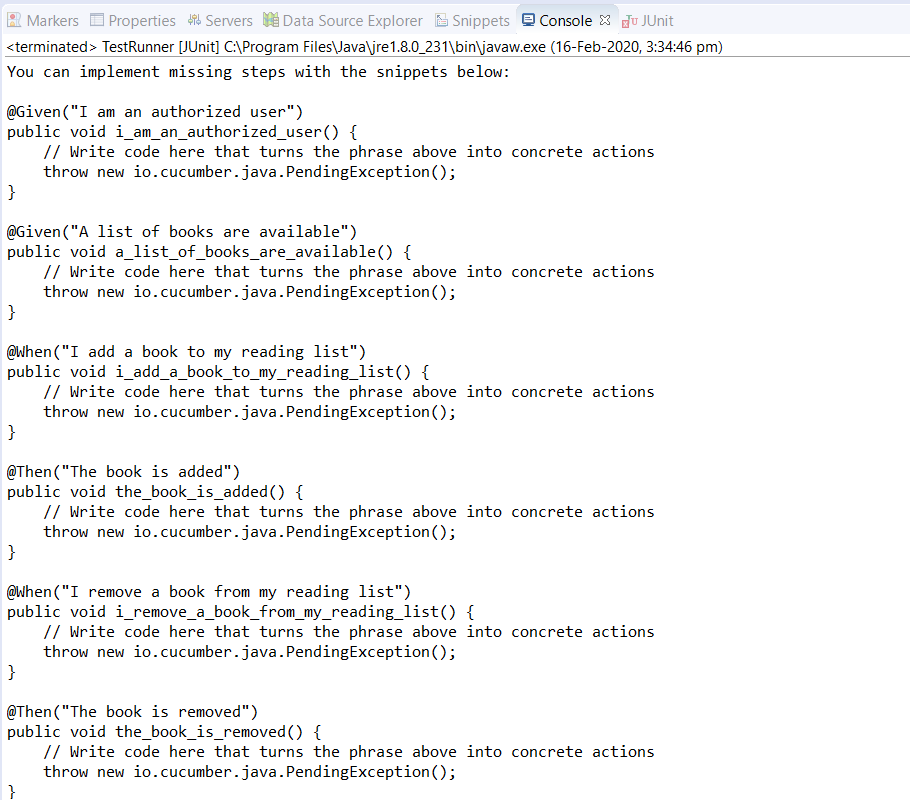
}

Note: Do not select the public static void main while creating the runner class.

### *****Step 4: Write test code to Step file*****

Fourthly, as we move ahead in our next step to convert the test code to Step file, we need to revise our knowledge on [***Cucumber Step Definitions***](https://www.toolsqa.com/cucumber/step-definition/).

1. To reduce our efforts in step creation, we will automatically generate the much required Cucumber Steps for implementation. Consequently, we will execute the ***TestRunner*** class for this. Right-click on the TestRunner file and select ***Run As >> JUnit Test***. Therefore, you would get the below result in the Eclipse Console.



1. Create a ***New Package*** and title it as ***stepDefinitions*** by right click on the src/test/java and select ***New >> Package***.
2. After that, create a ***New Java Class*** and name it is as ***Steps*** by right click on the above created package and select ***New >> Class***.
3. Finally, copy all the steps created by Eclipse to this Steps file and start filling up these steps with Selenium Code.  Select all the code from our Selenium Test file created in the ***End2End\_Test.*** The Steps test file will look like this:

package stepDefinitions;

import java.util.List;

import java.util.Map;

import org.junit.Assert;

import io.cucumber.java.en.Given;

import io.cucumber.java.en.Then;

import io.cucumber.java.en.When;

import io.restassured.RestAssured;

import io.restassured.path.json.JsonPath;

import io.restassured.response.Response;

import io.restassured.specification.RequestSpecification;

public class Steps {

private static final String USER\_ID = "9b5f49ab-eea9-45f4-9d66-bcf56a531b85";

private static final String USERNAME = "TOOLSQA-Test";

private static final String PASSWORD = "Test@@123";

private static final String BASE\_URL = "https://bookstore.toolsqa.com";

private static String token;

private static Response response;

private static String jsonString;

private static String bookId;

@Given("I am an authorized user")

public void iAmAnAuthorizedUser() {

RestAssured.baseURI = BASE\_URL;

RequestSpecification request = RestAssured.given();

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

response = request.body("{ \"userName\":\"" + USERNAME + "\", \"password\":\"" + PASSWORD + "\"}")

.post("/Account/v1/GenerateToken");

String jsonString = response.asString();

token = JsonPath.from(jsonString).get("token");

}

@Given("A list of books are available")

public void listOfBooksAreAvailable() {

RestAssured.baseURI = BASE\_URL;

RequestSpecification request = RestAssured.given();

response = request.get("/BookStore/v1/Books");

jsonString = response.asString();

List<Map<String, String>> books = JsonPath.from(jsonString).get("books");

Assert.assertTrue(books.size() > 0);

bookId = books.get(0).get("isbn");

}

@When("I add a book to my reading list")

public void addBookInList() {

RestAssured.baseURI = BASE\_URL;

RequestSpecification request = RestAssured.given();

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

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

response = request.body("{ \"userId\": \"" + USER\_ID + "\", " +

"\"collectionOfIsbns\": [ { \"isbn\": \"" + bookId + "\" } ]}")

.post("/BookStore/v1/Books");

}

@Then("The book is added")

public void bookIsAdded() {

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

}

@When("I remove a book from my reading list")

public void removeBookFromList() {

RestAssured.baseURI = BASE\_URL;

RequestSpecification request = RestAssured.given();

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

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

response = request.body("{ \"isbn\": \"" + bookId + "\", \"userId\": \"" + USER\_ID + "\"}")

.delete("/BookStore/v1/Book");

}

@Then("The book is removed")

public void bookIsRemoved() {

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

RestAssured.baseURI = BASE\_URL;

RequestSpecification request = RestAssured.given();

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

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

response = request.get("/Account/v1/User/" + USER\_ID);

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

jsonString = response.asString();

List<Map<String, String>> booksOfUser = JsonPath.from(jsonString).get("books");

Assert.assertEquals(0, booksOfUser.size());

}

}

***Note***: As compared to our E2E tests created, we have made the following changes:

* Declared the variables as ***private static final*** to not allow changes outside the class.
* We have updated method names and not kept the same as auto-generated ones.

1. The ***TestRunner*** file must be able to find the steps files. To achieve that, we need to mention the path of the package. This path has all of our step definitions in CucumberOptions. Additionally, to know more about the parameters we can add in the @CucumberOptions, please visit our tutorial [***Cucumber Options***](https://toolsqa.com/cucumber/cucumber-options/).

package runners;

import io.cucumber.junit.Cucumber;

import io.cucumber.junit.CucumberOptions;

import org.junit.runner.RunWith;

@RunWith(Cucumber.class)

@CucumberOptions(

features = "src/test/resources/functionalTests",

glue = {"stepDefinitions"},

monochrome = true,

strict = true

)

public class TestRunner {

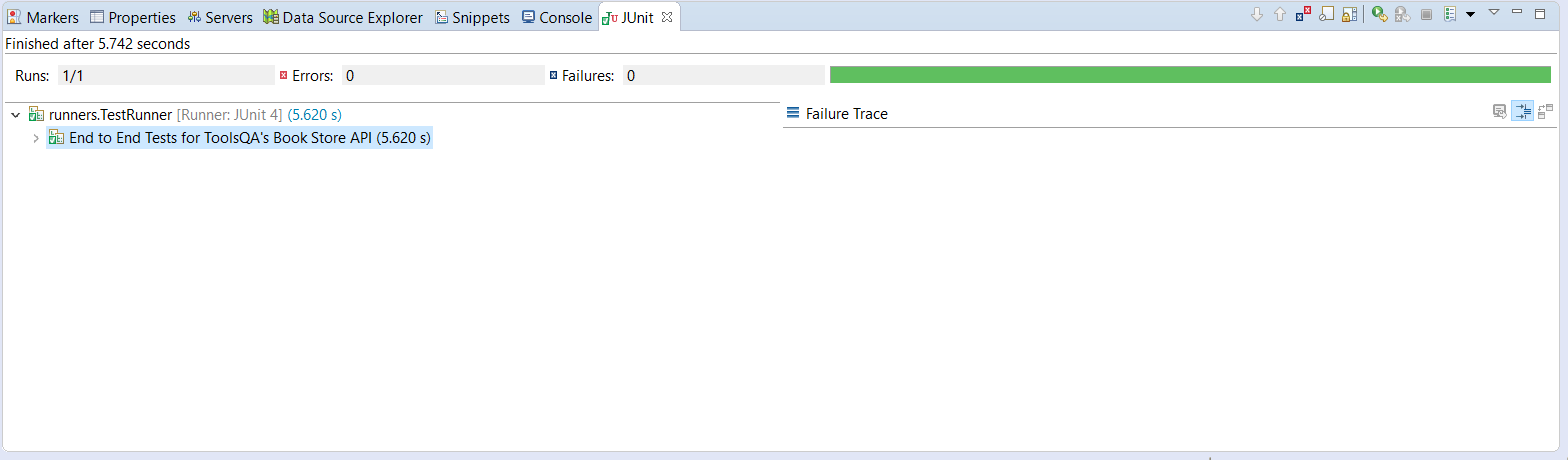
}

***Note***: By default, the Junit/Cucumber finds the test code in the ***src/test/java*** folder. Hence, this is why we just need to specify the package name for the cucumber glue.

### *****Step 5: Run the Cucumber Test*****

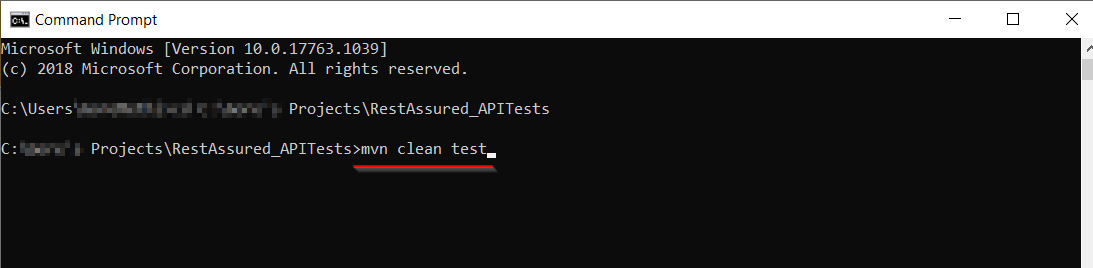
***Run as JUnit***

Finally, we are all set to run the first Cucumber test. Right -Click on ***TestRunner*** class and Click ***Run As  >> JUnit Test.*** Cucumber will execute the script the same way it runs in Selenium WebDriver. Consequently, the result will appear in the left-hand side project explorer window in the JUnit tab.



***Run the Tests from Command Prompt***

We have a Maven Type project, and thus, we can run our tests from the command prompt as well. A simple command to run tests is an mvn clean test. Moreover, to use this command, we have to change our directory to the location of our Cucumber project. In the below screenshot first, I went to my project location, and then I used the Maven as mentioned above command to run the test.



Consequently, we can see the output of the tests below in the command prompt.

To conclude, by now, we have converted our ***Rest Assured API tests into Cucumber based tests***. Subsequently, we will see how to implement the Java serialization concept in our Framework and enhance it. Moreover, it would be useful at this stage to go through the [***Convert JSON to JAVA Object***](https://www.toolsqa.com/rest-assured/convert-json-to-java-object/) tutorial as a refresher before advancing on the next framework tutorial.

# Convert JSON Request Body to JAVA Object

Category: [**Rest-Assured**](https://toolsqa.com/categories/rest-assured) , November 10 2021

0

0

11 min read

**Author**



Yashraj Gore

**Reviewers**



Lakshay Sharma

**[Previous Lesson](https://toolsqa.com/rest-assured/rest-api-test-in-cucumber/)**

[REST API Test in Cucumber](https://toolsqa.com/rest-assured/rest-api-test-in-cucumber/)

**[Next Lesson](https://toolsqa.com/rest-assured/json-response-body-to-java-object/)**

[Convert JSON Response Body to Java Object](https://toolsqa.com/rest-assured/json-response-body-to-java-object/)

So far, we have converted our [***Rest Assured E2E API tests***](https://toolsqa.com/rest-assured/rest-api-end-to-end-test/)  into [***Cucumber BDD Style Tests***](https://www.toolsqa.com/rest-assured/rest-api-test-in-cucumber/). Subsequently, our next step would ***Convert JSON to JAVA Object*** using ***Serialization***. We have covered Serialization and Deserialization tutorial in Java.  It would be highly appreciated if you revisit the [***Serialization and Deserialization***](https://www.toolsqa.com/rest-assured/serialization-and-deserialization-in-java/) chapter to understand well what's going around overall in our next stage of framework development.

## Convert JSON to JAVA Object

The request body we send right now is in this format:

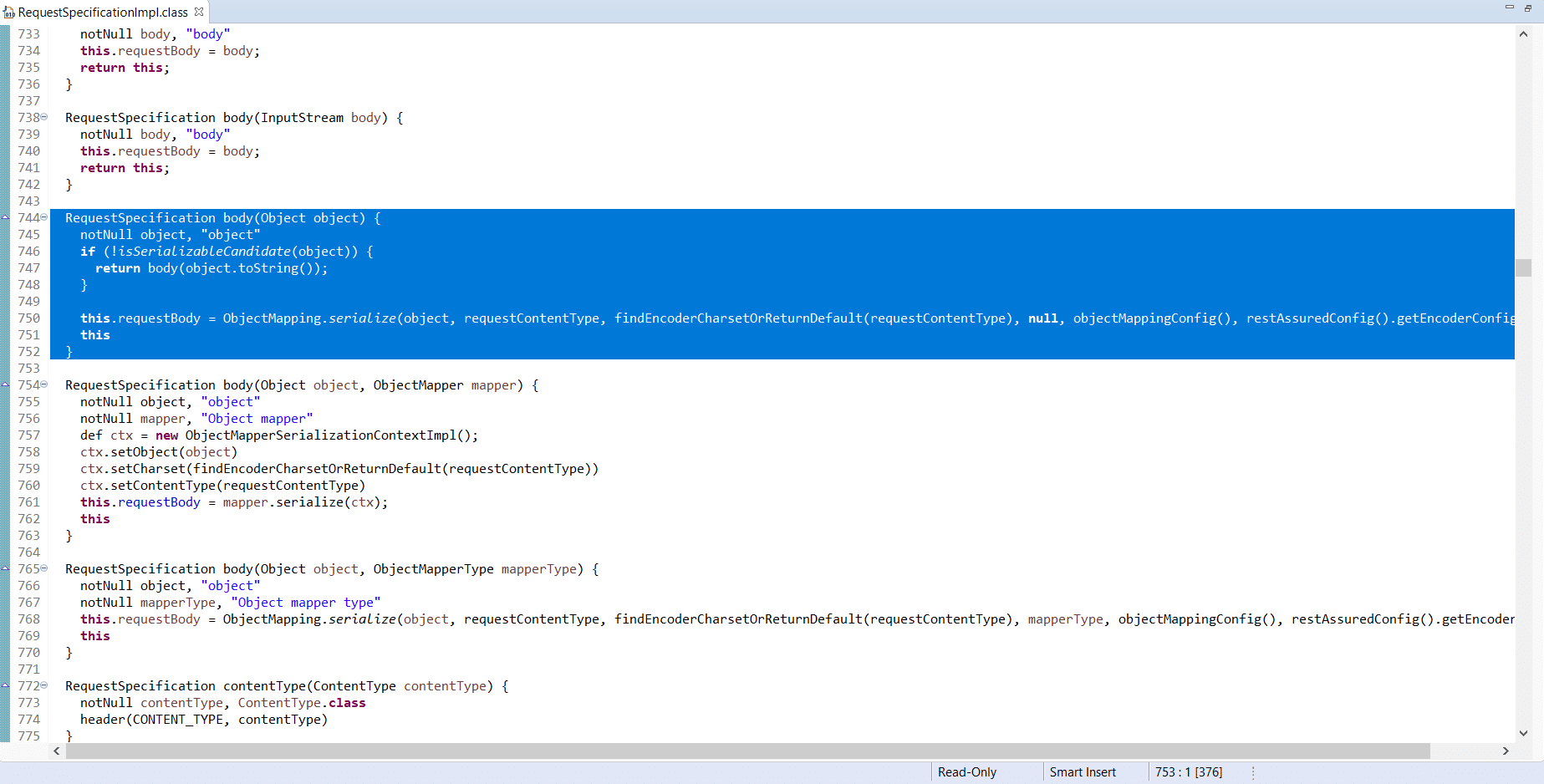
request.body("{ \"userName\":\"" + USERNAME + "\", \"password\":\"" + PASSWORD + "\"}")

.post("/Account/v1/GenerateToken");

We are sending the body request in a raw JSON string format. It is cumbersome to maintain and error-prone to send the request body in this format. Right now, we are dealing with just two parameters. But, there is a possibility that in actual body requests, we could have to deal with more number of parameters.

Additionally, it is advisable to send the username and password in the request body as an object. To achieve this we need to convert JSON to JAVA Object. But, the network does not understand Java objects. So, we would need to serialize the objects into String before sending the request.

We can do this using many serialization-deserialization libraries available. But, Rest Assured has this functionality in-built. It enables us to directly send objects in Rest Assured requests while the Library takes care of Serialization internally. If you dive deeper to understand the implementation of RequestSpecification, you will see the below code snippet which clearly shows how Rest Assured takes care of Serialization.



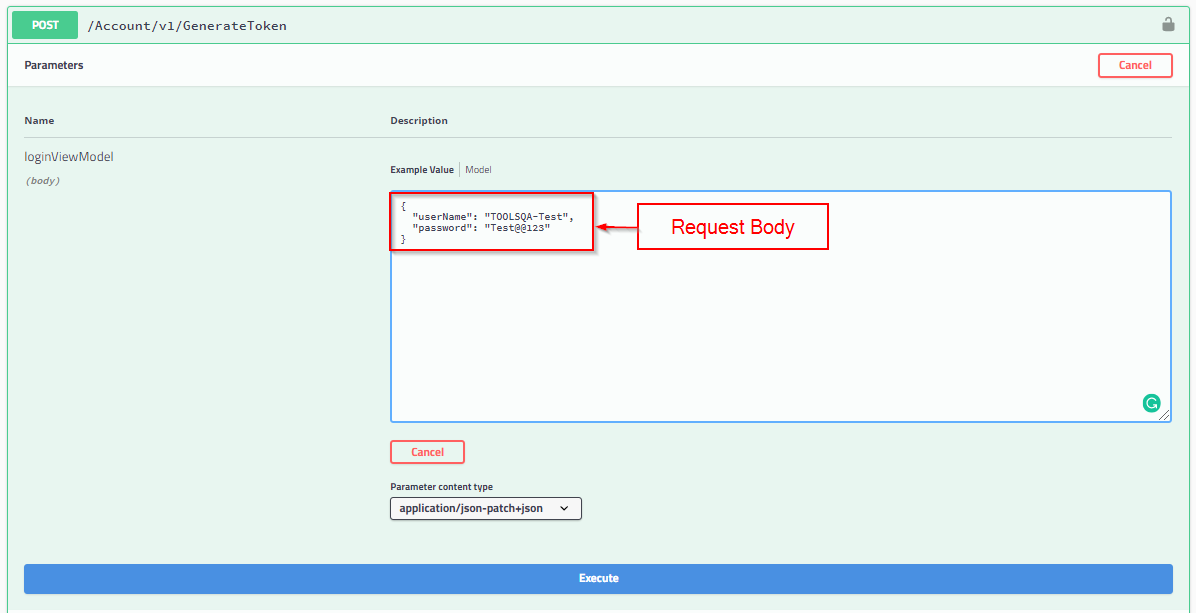
So we will now understand the process of converting a JSON request into a Java object in the next section.

### *****Create a POJO class for a Request Body*****

We are focussing on creating a POJO class for our request object. So, let us learn to create a POJO class out of a JSON. Let's begin with one simple request example from our Request Body:

Consider the API endpoint: ***/Account/v1/GenerateToken***

In the Request body for this API, we are sending the username and password as the request body.



As seen in the above [***Swagger bookstore API***](https://bookstore.toolsqa.com/swagger/index.html) document, the request JSON body parameters we pass in the request body is:

{

"userName": "TOOLSQA-Test",

"password": "Test@@123"

}

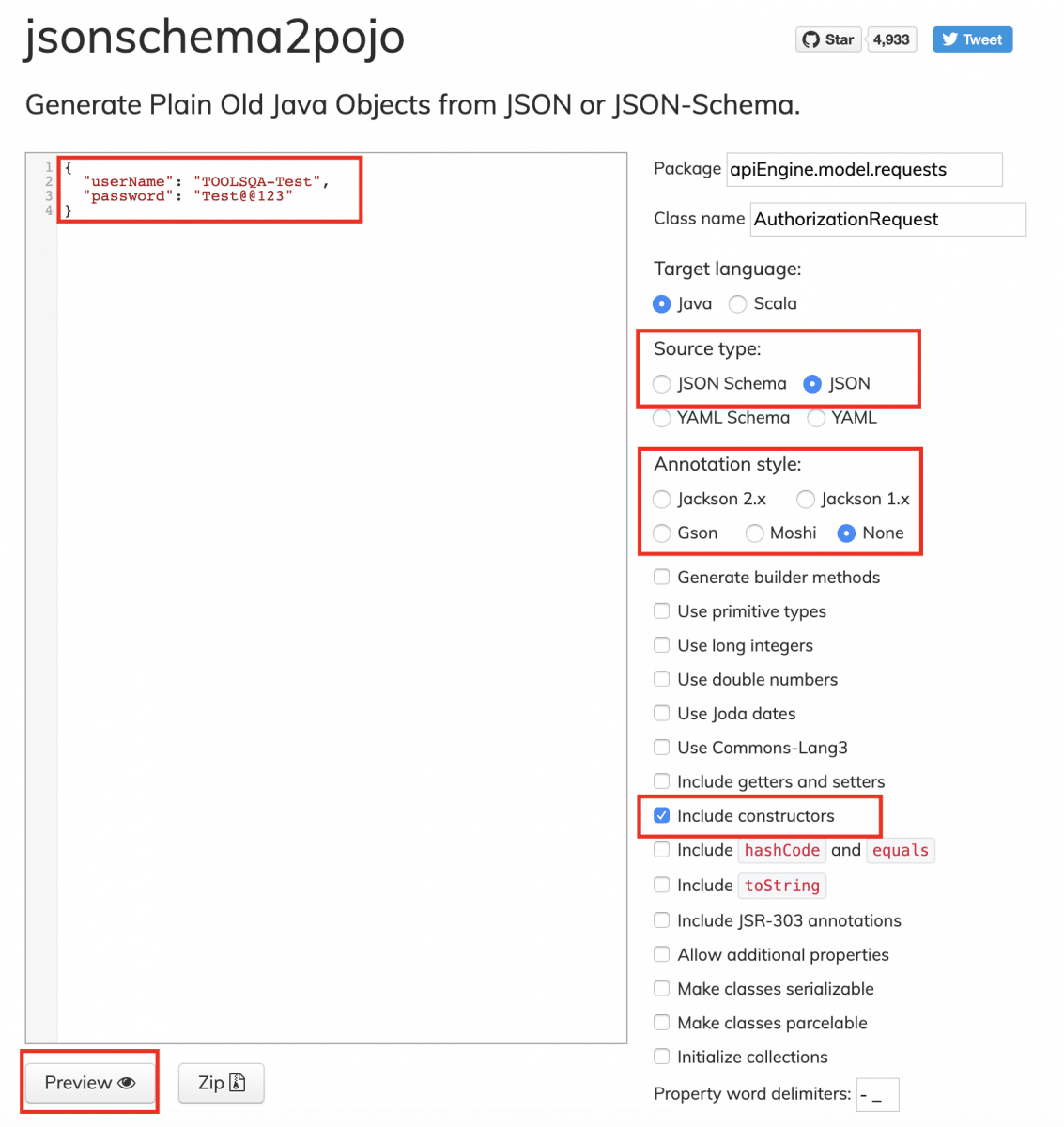
Moreover, you can manually create a POJO class having all listed fields of the JSON body. Also, there are various utilities available online for free using which we can convert any JSON structure to a Java POJO class.

#### **How to create a Java POJO Class for a JSON Request Body using Online Utility?**

Here we will see ***How to Convert a JSON String into Java Object***. So, let's take the help of one of the websites to help us convert the JSON to a Java POJO class.

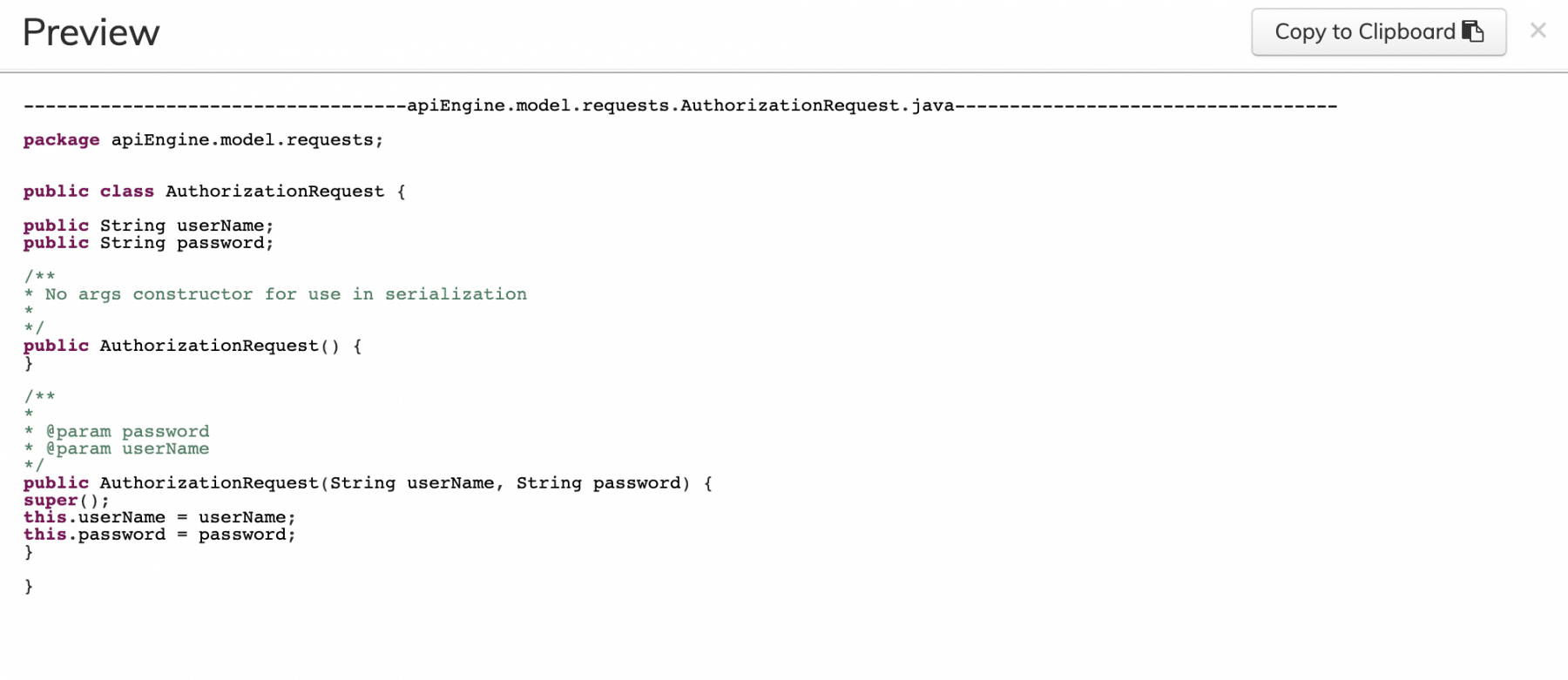
***Please follow these steps***

* Firstly, navigate to the website: [***http://www.jsonschema2pojo.org/***](https://www.jsonschema2pojo.org/)
* Secondly, enter the JSON body in the left text box.
* Thirdly, enter the package name and class name in the right-side panel.
* Finally, enter other required selection details as per the image below.



Click on the preview button highlighted in the above image.

The following image is displayed:



***Explanation***

Thus, with our inputs of a JSON with username and password as fields, we have created a POJO. Additionally, we will use this POJO to send into the request body of our API ***/Account/v1/GenerateToken.***

## Steps to Update the JSON String with POJO classes

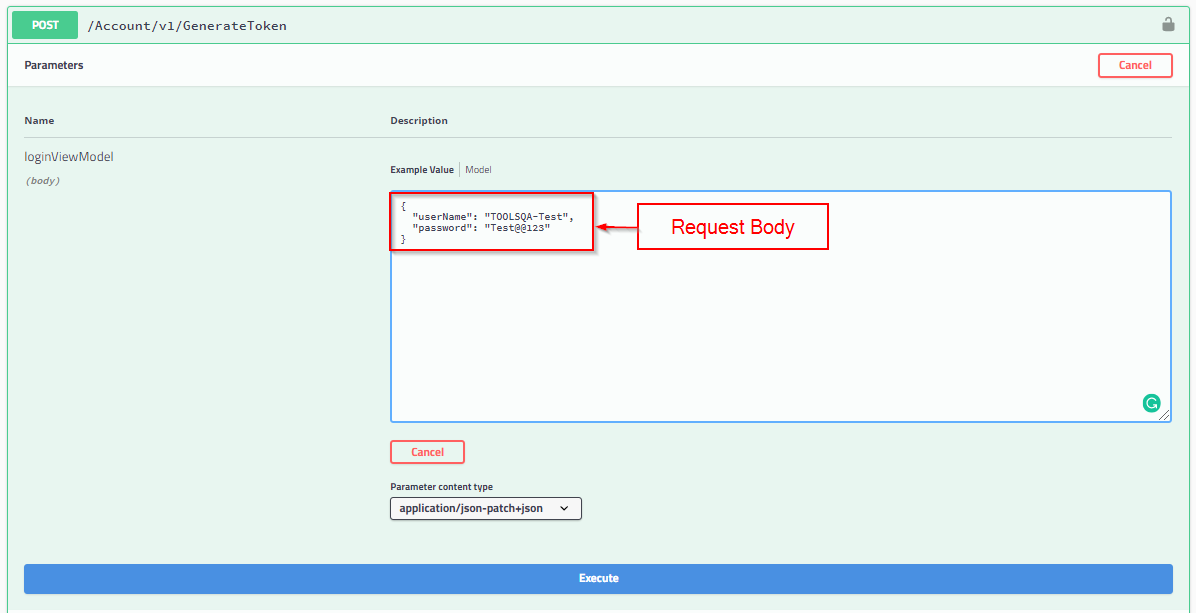
We will follow these steps to implement serialization in our framework:

1. Firstly, create POJO classes for each of our Request Objects:

* Token Request
* Add Books Request
* ISBN
* Remove Books Request

1. Secondly, replace the Request bodies in Step files with POJO class objects.
2. Finally, run the tests.

### *****POJO class for Authorization Token Request:*****



As shown in the above image from our [***Swagger bookstore API documentation***](https://www.jsonschema2pojo.org/) for the Generate Token API, the request body is:

{

"userName": "TOOLSQA-Test",

"password": "Test@@123"

}

To create a POJO class of it, please follow the below steps:

1. Firstly, Right-click on the ***src/test/java*** and select **New >> Package**. After that, create a New Package file and name it as apiEngine. Further inside, the ***apiEngine*** Package creates a new Package with the name as the ***model***. Moreover, inside this ***model*** Package, create a Package with the name ***requests.***  We will capture all the requests classes under this package.
2. Secondly, create a New Class file under it and name it as ***AuthorizationRequest***, by right click on the above-created Package and select ***New >> Class***.

***AuthorizationRequest.class***

package apiEngine.model.requests;

public class AuthorizationRequest {

public String username;

public String password;

public AuthorizationRequest(String username, String password) {

this.username = username;

this.password = password;

}

}

***Code Explanation***

The Bookstore URL requires the client to send the username and password as we studied in the [***API Documentation tutorial***](https://www.toolsqa.com/rest-assured/api-documentation/) for Authorization Token. We have defined the class AuthorizationRequest for this purpose. We will create an object of AuthorizationRequest with a constructor with the parameters username and password.

Similarly, we will create classes for Add Books Request, Remove Books Request, and ISBN.

### *****POJO class for Add Books Request:*****

As we saw in the Swagger bookstore API documentation for the Add Books API, the request body is:

{

"userId": "string",

"collectionOfIsbns": [

{

"isbn": "string"

}

]

}

To create a POJO class of the JSON request body, Right-click on the above-created ***request*** Package and select ***New >> Class***. Additionally, name it as ***AddBooksRequest***.

***AddBooksRequest.class***

package apiEngine.model.requests;

import java.util.ArrayList;

import java.util.List;

public class AddBooksRequest {

public String userId;

public List<ISBN> collectionOfIsbns;

//As of now this is for adding a single book, later we will add another constructor.

//That will take a collection of ISBN to add multiple books

public AddBooksRequest(String userId, ISBN isbn){

this.userId = userId;

collectionOfIsbns = new ArrayList<ISBN>();

collectionOfIsbns.add(isbn);

}

}

***Code Explanation***

The AddBooksRequest class will be responsible for providing us an object that adds a book into the user account. While we are at it, we would need the userId and the unique identifier of the book, ISBN. Thus, the userID and isbn pass as a parameter into the AddBooksRequest class object.

### *****POJO class for ISBN:*****

Right-click on the above-created ***request*** Package. After that, select ***New >> Class***. Name it as ***ISBN***.

package apiEngine.model.requests;

public class ISBN {

public String isbn;

public ISBN(String isbn) {

this.isbn = isbn;

}

}

***Code Explanation***

We created the ISBN class to use in the AddBooksRequest class for storing a collection of the type ISBN.

### *****POJO class for Remove Book Request:*****

As we saw in the Bookstore API for Remove Books Endpoint, the request body is:

{

"isbn": "string",

"userId": "string"

}

To create a POJO class of the JSON request body, Right-click on the above-created ***request*** Package and select ***New >> Class***. Additionally, name it as ***RemoveBookRequest***.

package apiEngine.model.requests;

public class RemoveBookRequest {

public String isbn;

public String userId;

public RemoveBookRequest(String userId, String isbn) {

this.userId = userId;

this.isbn = isbn;

}

}

***Code Explanation***

With the help of RemoveBooksRequest class, we will create an object by passing the parameters userId and isbn. Moreover, the request body uses this object.

### *****Replace the Request Bodies in Step files with POJO class objects*****

Lets just first start with the first step of the test that is ***"Given("^I am an authorized user$")"***

We made the following change in its Step Definition:

@Given("^I am an authorized user$")

public void iAmAnAuthorizedUser() {

AuthorizationRequest authRequest = new AuthorizationRequest("TOOLSQA-Test","Test@@123");

RestAssured.baseURI = BASE\_URL;

RequestSpecification request = RestAssured.given();

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

response = request.body(authRequest).post("/Account/v1/GenerateToken");

String jsonString = response.asString();

token = JsonPath.from(jsonString).get("token");

}

***Code Explanation***

We have created an object, authRequest of the class AuthorizationRequest. In this object, we are passing username and password. Additionally, this authRequest object will pass in the body of the request.

In a similar vein, we will make changes for the following Step Definitions to use Java objects that we created using the POJOs we defined. Internally, the rest assured library will take care of serializing this object into JSON string before sending the request over the network. Thus, we do not have to worry about serializing the request objects.

@When("^I add a book to my reading list$")

@When("I remove a book from my reading list")

Also, note that for the remaining three Step Definitions, we would not be making any changes for now.

@Given("A list of books are available")

@Then("^The book is added$")

@Then("^The book is removed$")

We put together all these code changes for Steps file

package stepDefinitions;

import java.util.List;

import java.util.Map;

import apiEngine.model.requests.AddBooksRequest;

import apiEngine.model.requests.AuthorizationRequest;

import apiEngine.model.requests.ISBN;

import apiEngine.model.requests.RemoveBookRequest;

import org.junit.Assert;

import cucumber.api.java.en.Given;

import cucumber.api.java.en.Then;

import cucumber.api.java.en.When;

import io.restassured.RestAssured;

import io.restassured.path.json.JsonPath;

import io.restassured.response.Response;

import io.restassured.specification.RequestSpecification;

public class Steps {

private static final String USER\_ID = "9b5f49ab-eea9-45f4-9d66-bcf56a531b85";

private static final String BASE\_URL = "https://bookstore.toolsqa.com";

private static String token;

private static Response response;

private static String jsonString;

private static String bookId;

@Given("^I am an authorized user$")

public void iAmAnAuthorizedUser() {

AuthorizationRequest credentials = new AuthorizationRequest("TOOLSQA-Test","Test@@123");

RestAssured.baseURI = BASE\_URL;

RequestSpecification request = RestAssured.given();

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

response = request.body(credentials).post("/Account/v1/GenerateToken");

String jsonString = response.asString();

token = JsonPath.from(jsonString).get("token");

}

@Given("^A list of books are available$")

public void listOfBooksAreAvailable() {

RestAssured.baseURI = BASE\_URL;

RequestSpecification request = RestAssured.given();

response = request.get("/BookStore/v1/Books");

jsonString = response.asString();

List<Map<String, String>> books = JsonPath.from(jsonString).get("books");

bookId = books.get(0).get("isbn");

}

@When("^I add a book to my reading list$")

public void addBookInList() {

AddBooksRequest addBooksRequest = new AddBooksRequest(USER\_ID, new ISBN(bookId));

RestAssured.baseURI = BASE\_URL;

RequestSpecification request = RestAssured.given();

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

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

response = request.body(addBooksRequest).post("/BookStore/v1/Books");

}

@Then("^The book is added$")

public void bookIsAdded() {

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

}

@When("^I remove a book from my reading list$")

public void removeBookFromList() {

RestAssured.baseURI = BASE\_URL;

RequestSpecification request = RestAssured.given();

RemoveBookRequest removeBookRequest = new RemoveBookRequest(USER\_ID, bookId);

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

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

response = request.body(removeBookRequest).delete("/BookStore/v1/Book");

}

@Then("^The book is removed$")

public void bookIsRemoved(){

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

RestAssured.baseURI = BASE\_URL;

RequestSpecification request = RestAssured.given();

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

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

response = request.get("/Account/v1/User/" + USER\_ID);

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

jsonString = response.asString();

List<Map<String, String>> booksOfUser = JsonPath.from(jsonString).get("books");

Assert.assertEquals(0, booksOfUser.size());

}

}

***Note***: We added an import statement for JSONpath, ***import io.restassured.path.json.JsonPath***; It will help us to traverse through the specific parts of the JSON. Moreover, you can read more in the [***JSONPath article***](https://www.toolsqa.com/rest-assured/jsonpath-and-query-json-using-jsonpath/).

Now, if we try to run our tests, using ***TestRunner*** after making all the changes in the Steps file, our tests will ***fail***.

The ***Runtime-Exception***, we will come across is:

java.lang.IllegalStateException: Cannot serialize object because no JSON serializer found in classpath. Please put Jackson (Databind), Gson, Johnzon, or Yasson in the classpath.

at sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)

at sun.reflect.NativeConstructorAccessorImpl.newInstance(Unknown Source)

at sun.reflect.DelegatingConstructorAccessorImpl.newInstance(Unknown Source)

at java.lang.reflect.Constructor.newInstance(Unknown Source)

at org.codehaus.groovy.reflection.CachedConstructor.invoke(CachedConstructor.java:80)

at org.codehaus.groovy.runtime.callsite.ConstructorSite$ConstructorSiteNoUnwrapNoCoerce.callConstructor(ConstructorSite.java:105)

at org.codehaus.groovy.runtime.callsite.CallSiteArray.defaultCallConstructor(CallSiteArray.java:59)

at org.codehaus.groovy.runtime.callsite.AbstractCallSite.callConstructor(AbstractCallSite.java:237)

at org.codehaus.groovy.runtime.callsite.AbstractCallSite.callConstructor(AbstractCallSite.java:249)

at io.restassured.internal.mapping.ObjectMapping.serialize(ObjectMapping.groovy:160)

at io.restassured.internal.mapping.ObjectMapping$serialize.call(Unknown Source)

at org.codehaus.groovy.runtime.callsite.CallSiteArray.defaultCall(CallSiteArray.java:47)

at org.codehaus.groovy.runtime.callsite.AbstractCallSite.call(AbstractCallSite.java:115)

at io.restassured.internal.RequestSpecificationImpl.body(RequestSpecificationImpl.groovy:751)

at stepDefinitions.Steps.iAmAnAuthorizedUser(Steps.java:38

We are missing a key element in the whole show. As explained above in the first section, Rest Assured takes care of Serialization internally. We also confirmed it in the code snippet. The Rest Assured library depends on the [***Jackson (Databind)***](https://mvnrepository.com/artifact/com.fasterxml.jackson.core/jackson-databind/2.10.2) library, to do this work of Serialization. Since we haven’t yet added this library, we faced the error.

Let us quickly add the dependency in our framework project file pom.xml:

<!-- https://mvnrepository.com/artifact/com.fasterxml.jackson.core/jackson-databind -->

<dependency>

<groupId>com.fasterxml.jackson.core</groupId>

<artifactId>jackson-databind</artifactId>

<version>2.10.2</version>

</dependency>

***Note:*** As of Feb 2019, the latest available dependency for jackson-databind was of version 2.10.2. Please use the latest dependencies when you build your framework.

Our new POM file will look like this:

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="https://maven.apache.org/POM/4.0.0" xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="https://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>ToolsQA</groupId>

<artifactId>APITestingFramework</artifactId>

<version>1.0-SNAPSHOT</version>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.13</version>

<scope>test</scope>

</dependency>

<dependency>

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

<artifactId>rest-assured</artifactId>

<version>4.2.0</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>io.cucumber</groupId>

<artifactId>cucumber-java</artifactId>

<version>5.2.0</version>

</dependency>

<dependency>

<groupId>io.cucumber</groupId>

<artifactId>cucumber-jvm-deps</artifactId>

<version>1.0.6</version>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>io.cucumber</groupId>

<artifactId>cucumber-junit</artifactId>

<version>5.2.0</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>com.fasterxml.jackson.core</groupId>

<artifactId>jackson-databind</artifactId>

<version>2.10.2</version>

</dependency>

</dependencies>

<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>

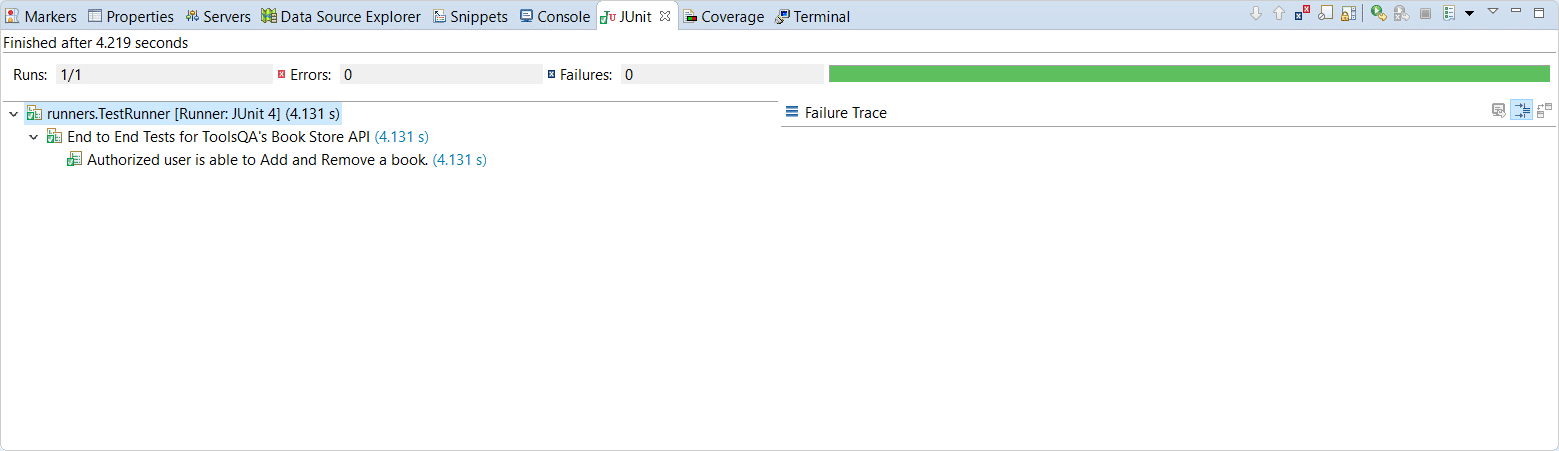
</build>

</project>

#### **Run the Cucumber Test**

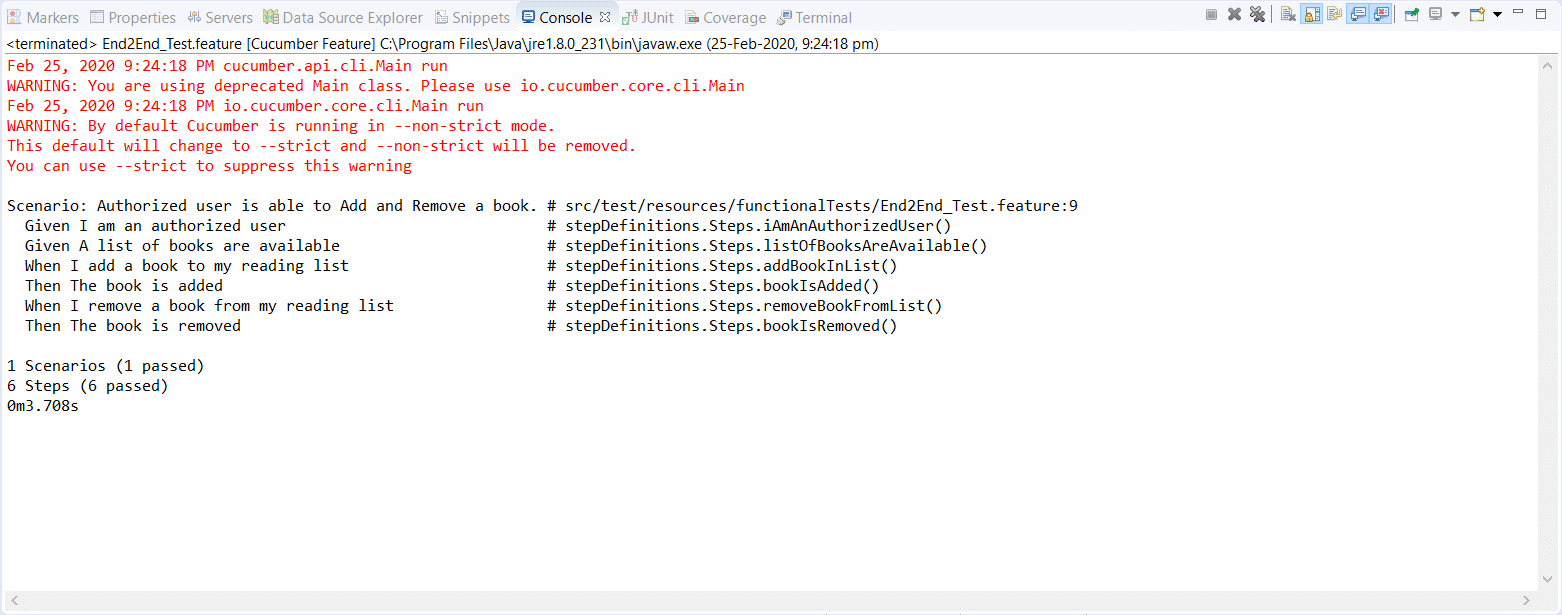
***Run the Tests as JUnit***

Now we are all set to run the updated Cucumber test. Right -Click on ***TestRunner*** class, and after that, click ***Run As  >> JUnit Test.***  Consequently, the result will display in the JUnit tab of the console.

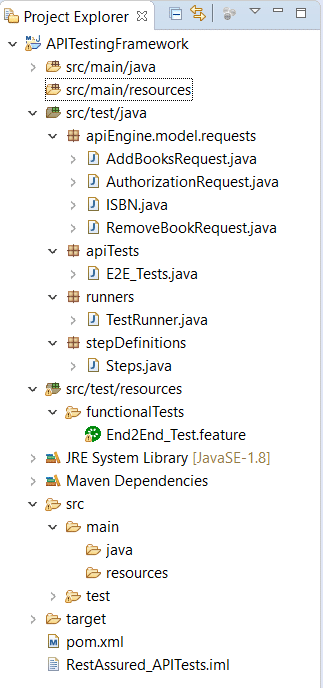


***Run the Tests from Cucumber Feature***

To run the tests as a Cucumber Feature, right-click on the End2End\_Test.feature file. After that, select the Run As>>Cucumber Feature.



As you can see in the screenshot attached above, our tests have passed with the changes. We have Convert JSON to JAVA Object using Serialization in this chapter. Please try to implement it in your framework, as explained above. Below is the screenshot of the current project explorer.



Subsequently, we will visit the [***Convert JSON Response Body to POJO***](https://www.toolsqa.com/rest-assured/convert-json-to-java-object/) in our next chapter.

# Convert JSON Response Body to Java Object

Category: [**Rest-Assured**](https://toolsqa.com/categories/rest-assured) , July 7 2021

0

0

8 min read

**Author**



Yashraj Gore

**Reviewers**



Lakshay Sharma

**[Previous Lesson](https://toolsqa.com/rest-assured/convert-json-to-java-object/)**

[Convert JSON Request Body to JAVA Object](https://toolsqa.com/rest-assured/convert-json-to-java-object/)

**[Next Lesson](https://toolsqa.com/rest-assured/separation-of-test-layer/)**

[Separation of Test Layer with API Services](https://toolsqa.com/rest-assured/separation-of-test-layer/)

In this chapter, we will implement the ***Deserialization of JSON Responses*** we receive. It helps us to read the body of the response. Subsequently, in this process of deserialization, we will ***Convert JSON Response Body to Java Objects***. It is also known as Object Representation of JSON responses.

Additionally, to learn more about the deserialization of JSON response, we will recommend you to go through the [***Deserialize JSON Response***](https://www.toolsqa.com/rest-assured/deserialize-json-response/).

We will follow the below steps to achieve the deserialization of our responses in the framework. We followed them in the previous chapter of [***Convert JSON Request Body to Java Object***](https://www.toolsqa.com/rest-assured/convert-json-to-java-object/) as well.

1. Create POJO classes for each of our Response Objects:

* Book
* Books
* Token
* User Account

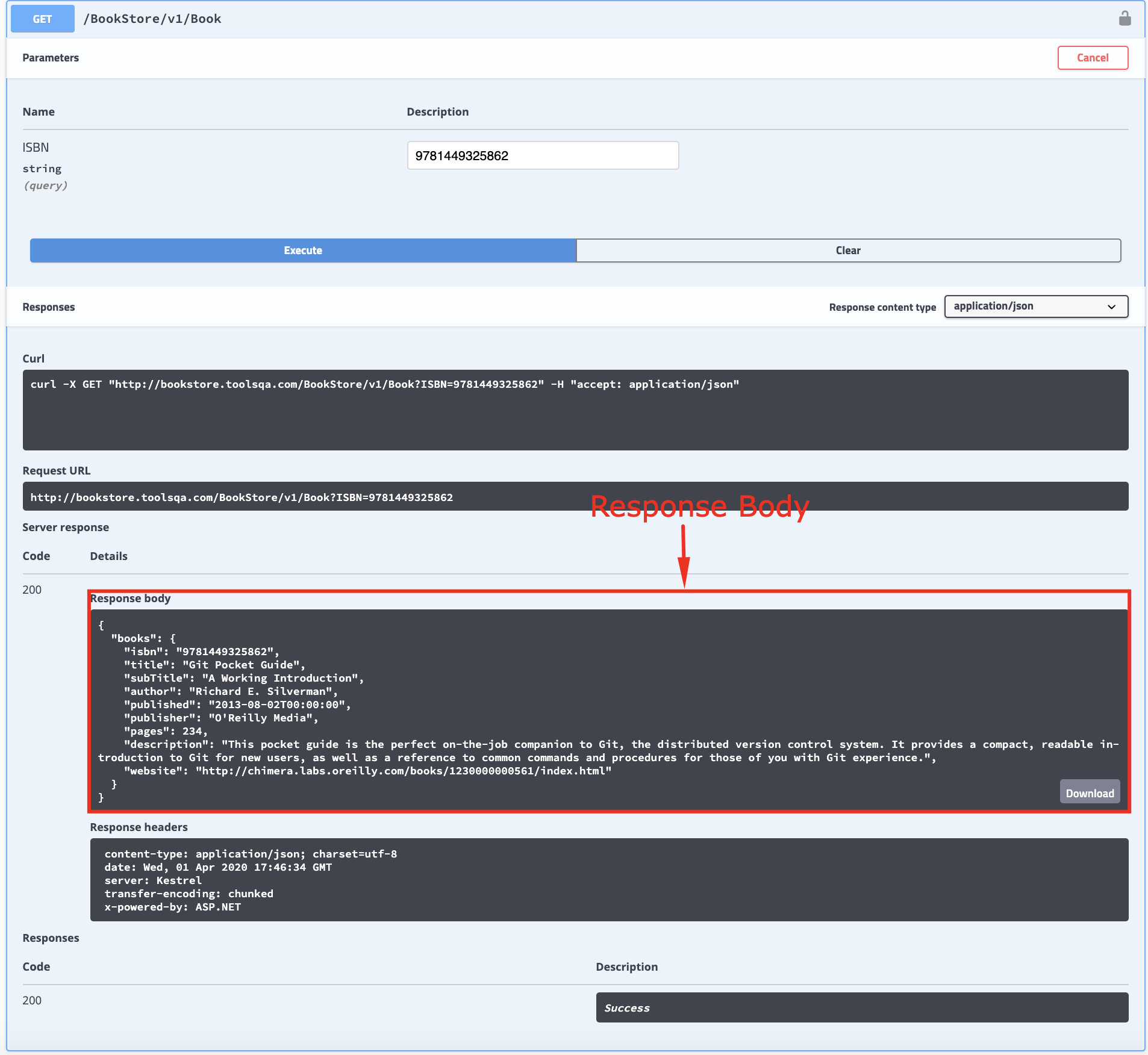
1. Replace the Response bodies in Step files with POJO class objects
2. Run the tests

## Convert JSON Response Body to Java Object

Firstly, we need to convert the JSON Response into a POJO class for our response object. So, let us learn to create a POJO class out of a JSON Response. Under deserialization, we will study our JSON body parameters and create a POJO class of it.  Let's begin with a straightforward request example for Get A Book request:

### *****Create POJO class Book*****

As a part of this request, we are sending the ISBN parameter to get details of a specific book.



Consequently, the response body from the [***Swagger Bookstore API***](https://bookstore.toolsqa.com/swagger/index.html), as highlighted in the above image is:

{

"books": [

{

"isbn": "9781449325862",

"title": "Git Pocket Guide",

"subTitle": "A Working Introduction",

"author": "Richard E. Silverman",

"published": "2013-08-02T00:00:00",

"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 experience.",

"website": "https://chimera.labs.oreilly.com/books/1230000000561/index.html"

}

}

As a part of the next step, we will create a POJO class for our response object. Additionally, you may use the same [***online utility***](https://www.jsonschema2pojo.org/), as we discussed in the previous chapter, to convert the response to a POJO class.

To create a POJO class of it, follow the below steps:

1. Firstly, Right-click on the ***model*** Package under the ***apiEngine*** package. After that, select ***New>>Class***. Name it as Book. Moreover, we will capture the book response under this class.

***Book.class***

package apiEngine.model;

public class Book {

public String isbn;

public String title;

public String subTitle;

public String author;

public String published;

public String publisher;

public int pages;

public String description;

public String website;

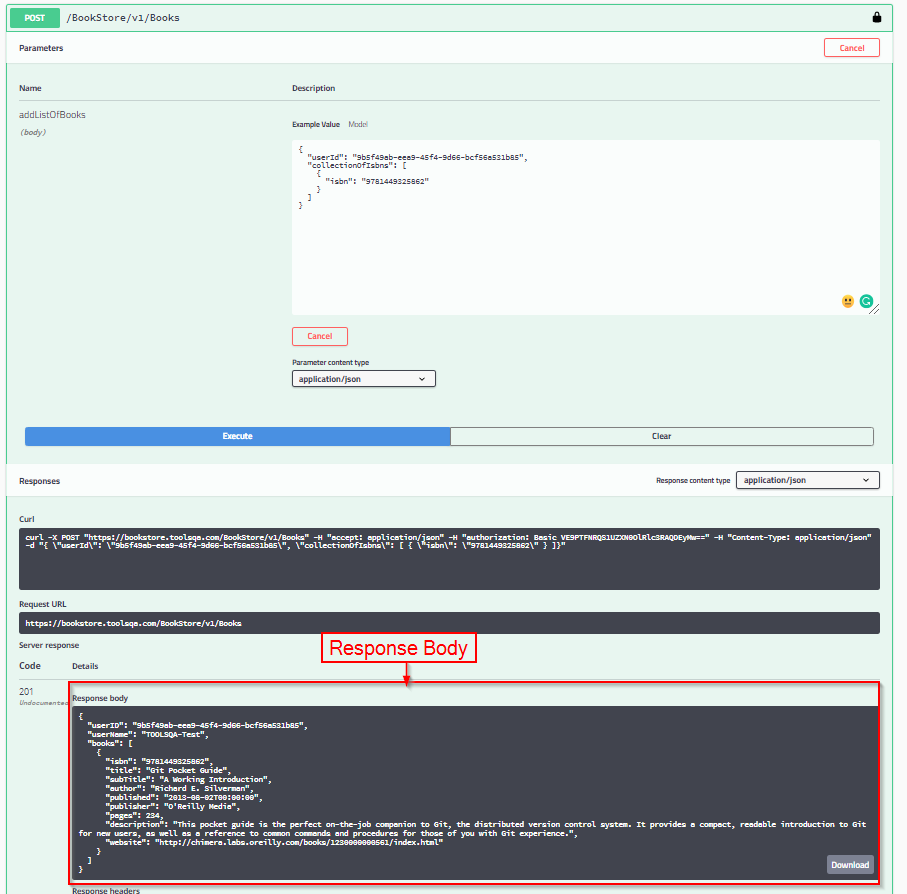
}

***Code Explanation***

This Book class will contain all the fields we got in the response, such as title, isbn, author, no.of pages, etc.

### *****Create a POJO class for Books Response Object:*****

The response body from the Bookstore API, as highlighted in the below image is:



{

"userID": "9b5f49ab-eea9-45f4-9d66-bcf56a531b85",

"userName": "TOOLSQA-Test",

"books": [

{

"isbn": "9781449325862",

"title": "Git Pocket Guide",

"subTitle": "A Working Introduction",

"author": "Richard E. Silverman",

"published": "2013-08-02T00:00:00",

"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 experience.",

"website": "https://chimera.labs.oreilly.com/books/1230000000561/index.html"

}

]

}

***Note***: As a part of the response body, we got the book details of the book we added for the user as well as other user details such as ***userName*** and ***userID.***

To create a POJO class of it, follow the below steps:

1. Firstly, in this ***model*** Package, Right-click on the model and select ***New >> Package.*** Name it as ***responses.***  Additionally, we will capture all the response classes under this package.
2. Secondly, Right-click on the above-created ***responses*** Package and select ***New >> Class***. Name it as ***Books.***

***Books.class***

package apiEngine.model.responses;

import java.util.List;

import apiEngine.model.Book;

public class Books {

public List<Book> books;

}

***Code Explanation*** We created the class Books to hold a list of the type Book which we receive in the JSON response in GET Books Request.

### *****Create a POJO class for Token Object:*****

We have understood the process of creating a POJO class by now, as shown in the above examples. So, in the next APIs, I will directly add the POJO of respective API's. Consequently, the response body for Generate Token API from our ***Bookstore API*** is:

{

"token": "string",

"expires": "2020-03-14T13:42:15.716Z",

"status": "string",

"result": "string"

}

To create a POJO class of it, Right-click on the above-created ***responses*** Package and select ***New >> Class***. After that, name it as ***Token***.

package apiEngine.model.responses;

public class Token {

public String token;

public String expires;

public String status;

public String result;

}

***Code Explanation***

We created the class Token that will contain all the fields we got in the response field, namely, token, expiry, status, and a result which we receive in the JSON response of GenerateToken.

### *****Create a POJO class for User Account Object:*****

Similarly, for User Account API, the response body is:

{

"userID": "9b5f49ab-eea9-45f4-9d66-bcf56a531b85",

"userName": "TOOLSQA-Test",

"books": []

}

To create a POJO class of the response body, Right-click on the above-created ***responses*** Package. After that, select ***New >> Class***. Name it as ***UserAccount***.

package apiEngine.model.responses;

import java.util.List;

import apiEngine.model.Book;

public class UserAccount {

public String userID;

public String userName;

public List<Book> books;

}

***Code Explanation***

We created the class UserAccount to hold the information associated with the user account, such as books associated with the user, the userID, and userName.

As our next step of project implementation, we shall modify our Steps class.

## Replace the Response bodies in Step files with POJO class objects

Now it is the time to make use of the POJO classes created above in the test code. Let's have a look at the one of the old cucumber step below in the Step Definition:

@Given("^I am an authorized user$")

public void iAmAnAuthorizedUser() {

AuthorizationRequest credentials = new AuthorizationRequest("TOOLSQA-Test","Test@@123");

RestAssured.baseURI = BASE\_URL;

RequestSpecification request = RestAssured.given();

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

response = request.body(credentials).post("/Account/v1/GenerateToken");

String jsonString = response.asString();

//We will deserialize the Response body into Token Response

token = JsonPath.from(jsonString).get("token");

}

we update it likewise:

@Given("^I am an authorized user$")

public void iAmAnAuthorizedUser() {

RestAssured.baseURI = BASE\_URL;

RequestSpecification request = RestAssured.given();

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

AuthorizationRequest authRequest = new AuthorizationRequest("TOOLSQA-Test", "Test@@123");

response = request.body(authRequest).post("/Account/v1/GenerateToken");

// Deserializing the Response body into tokenResponse

tokenResponse = response.getBody().as(Token.class);

}

Note The import statement: ***import io.restassured.path.json.JsonPath***; is no longer needed as we have deserialized our response to Token class.

***Code Explanation:*** We deserialized the response body into the ***Token*** class in this step above. This response body gets saved in the "***tokenResponse***" variable. The saved variable will be used further in a request along with the token.

We put together these code snippets for our Steps file:

package stepDefinitions;

import java.util.List;

import java.util.Map;

import org.junit.Assert;

import apiEngine.model.Book;

import apiEngine.model.requests.AddBooksRequest;

import apiEngine.model.requests.AuthorizationRequest;

import apiEngine.model.requests.ISBN;

import apiEngine.model.requests.RemoveBookRequest;

import apiEngine.model.response.Books;

import apiEngine.model.response.Token;

import apiEngine.model.response.UserAccount;

import io.cucumber.java.en.Given;

import io.cucumber.java.en.Then;

import io.cucumber.java.en.When;

import io.restassured.RestAssured;

import io.restassured.path.json.JsonPath;

import io.restassured.response.Response;

import io.restassured.specification.RequestSpecification;

public class Steps {

private static final String USER\_ID = "9b5f49ab-eea9-45f4-9d66-bcf56a531b85";

private static final String BASE\_URL = "https://bookstore.toolsqa.com";

private static Response response;

private static Token tokenResponse;

private static Book book;

@Given("I am an authorized user")

public void iAmAnAuthorizedUser() {

AuthorizationRequest credentials = new AuthorizationRequest("TOOLSQA-Test","Test@@123");

RestAssured.baseURI = BASE\_URL;

RequestSpecification request = RestAssured.given();

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

response = request.body(credentials).post("/Account/v1/GenerateToken");

// Deserializing the Response body into tokenResponse

tokenResponse = response.getBody().as(Token.class);

}

@Given("A list of books are available")

public void listOfBooksAreAvailable() {

RestAssured.baseURI = BASE\_URL;

RequestSpecification request = RestAssured.given();

response = request.get("/BookStore/v1/Books");

// Deserializing the Response body into Books class

Books books = response.getBody().as(Books.class);

book = books.books.get(0);

}

@When("I add a book to my reading list")

public void addBookInList() {

ISBN isbn = new ISBN(book.isbn);

AddBooksRequest addBooksRequest = new AddBooksRequest(USER\_ID, isbn);

RestAssured.baseURI = BASE\_URL;

RequestSpecification request = RestAssured.given();

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

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

response = request.body(addBooksRequest).post("/BookStore/v1/Books");

}

@Then("The book is added")

public void bookIsAdded() {

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

UserAccount userAccount = response.getBody().as(UserAccount.class);

Assert.assertEquals(USER\_ID, userAccount.userID);

Assert.assertEquals(book.isbn, userAccount.books.get(0).isbn);

}

@When("I remove a book from my reading list")

public void removeBookFromList() {

RestAssured.baseURI = BASE\_URL;

RequestSpecification request = RestAssured.given();

RemoveBookRequest removeBookRequest = new RemoveBookRequest(USER\_ID, book.isbn);

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

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

response = request.body(removeBookRequest).delete("/BookStore/v1/Book");

}

@Then("The book is removed")

public void bookIsRemoved() {

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

RestAssured.baseURI = BASE\_URL;

RequestSpecification request = RestAssured.given();

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

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

response = request.get("/Account/v1/User/" + USER\_ID);

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

UserAccount userAccount = response.getBody().as(UserAccount.class);

Assert.assertEquals(0, userAccount.books.size());

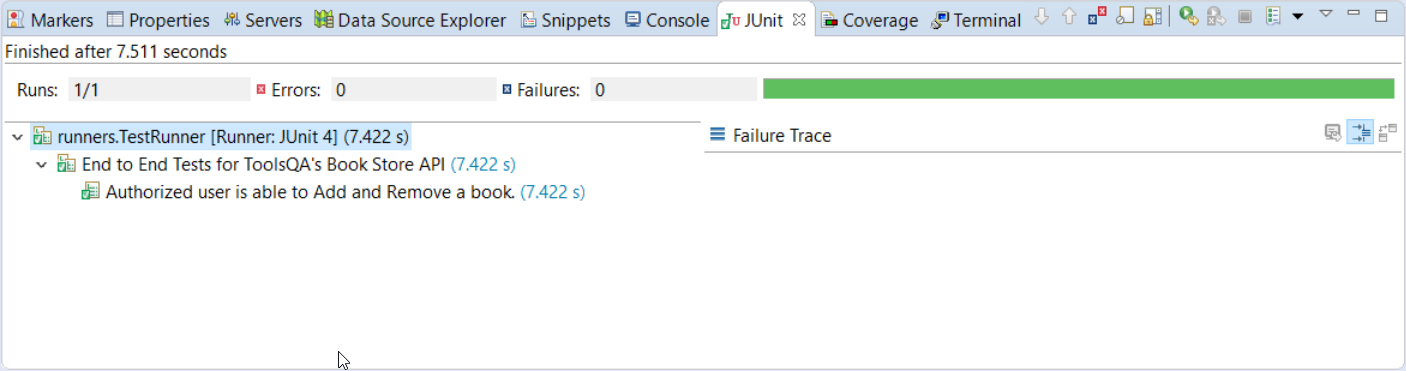
}

}

## Run the Cucumber Test

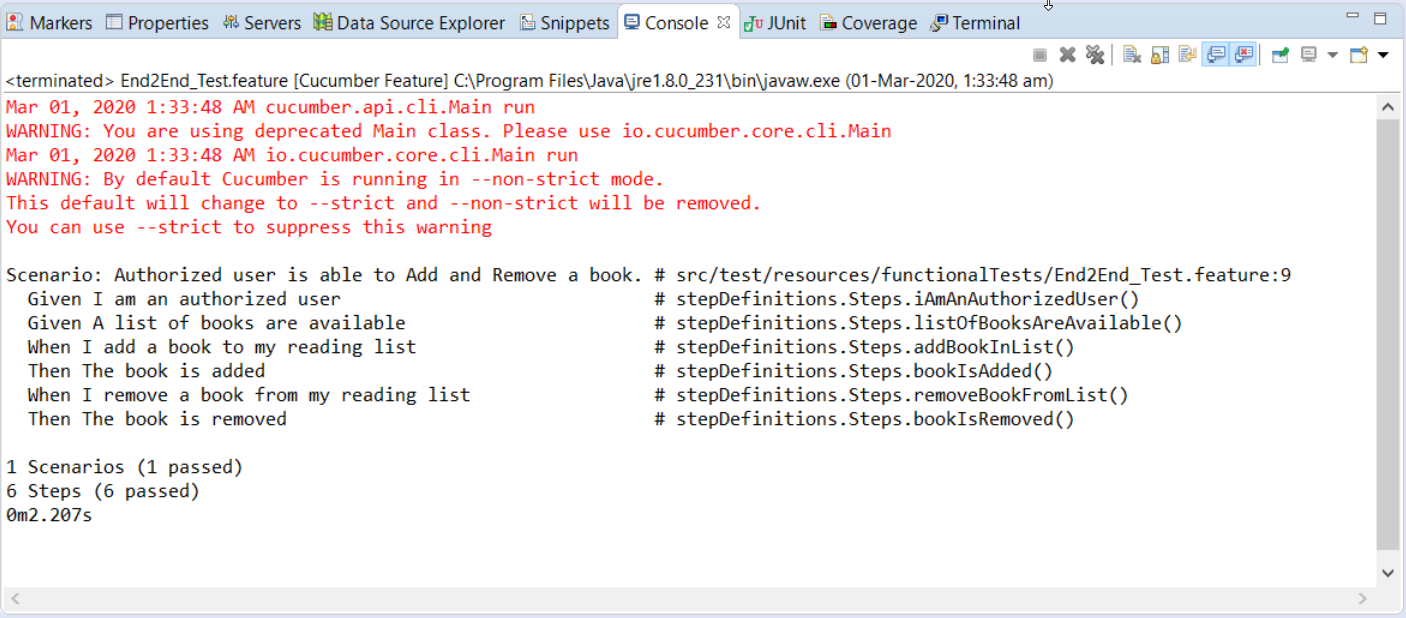
***Run the Tests as JUnit***

We are all set now to run the updated Cucumber test. First, Right -Click on ***TestRunner*** class and Click ***Run As  >> JUnit Test.*** Cucumber runs the script in the same way as Selenium WebDriver.  Consequently, the result will display in the JUnit tab of the console.



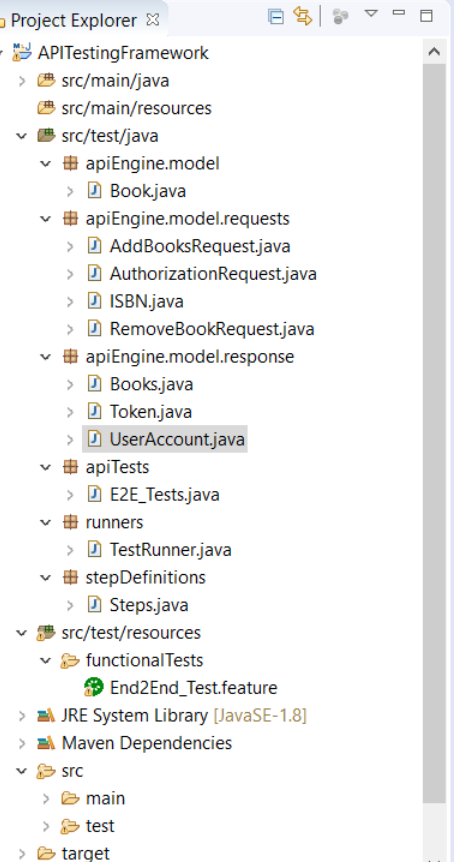
***Run the Tests from Cucumber Feature***

To run the tests as a Cucumber Feature, right-click on the End2End\_Test.feature file. After that, select the Run As>>Cucumber Feature.



Our tests passed with the changes we made for the conversion of the JSON Response Body to Java Object. Please try to implement it in your framework, as explained above, and share your valuable feedback.

After creating the above POJO classes, the project folder structure will look likewise:



Subsequently, we will attempt the [***separation of Test Layer with API Services***](https://www.toolsqa.com/rest-assured/separation-of-test-layer/) in our next chapter.

# Separation of Test Layer with API Services

Category: [**Rest-Assured**](https://toolsqa.com/categories/rest-assured) , July 7 2021

0

0

9 min read

**Author**



Yashraj Gore

**Reviewers**



Lakshay Sharma

**[Previous Lesson](https://toolsqa.com/rest-assured/json-response-body-to-java-object/)**

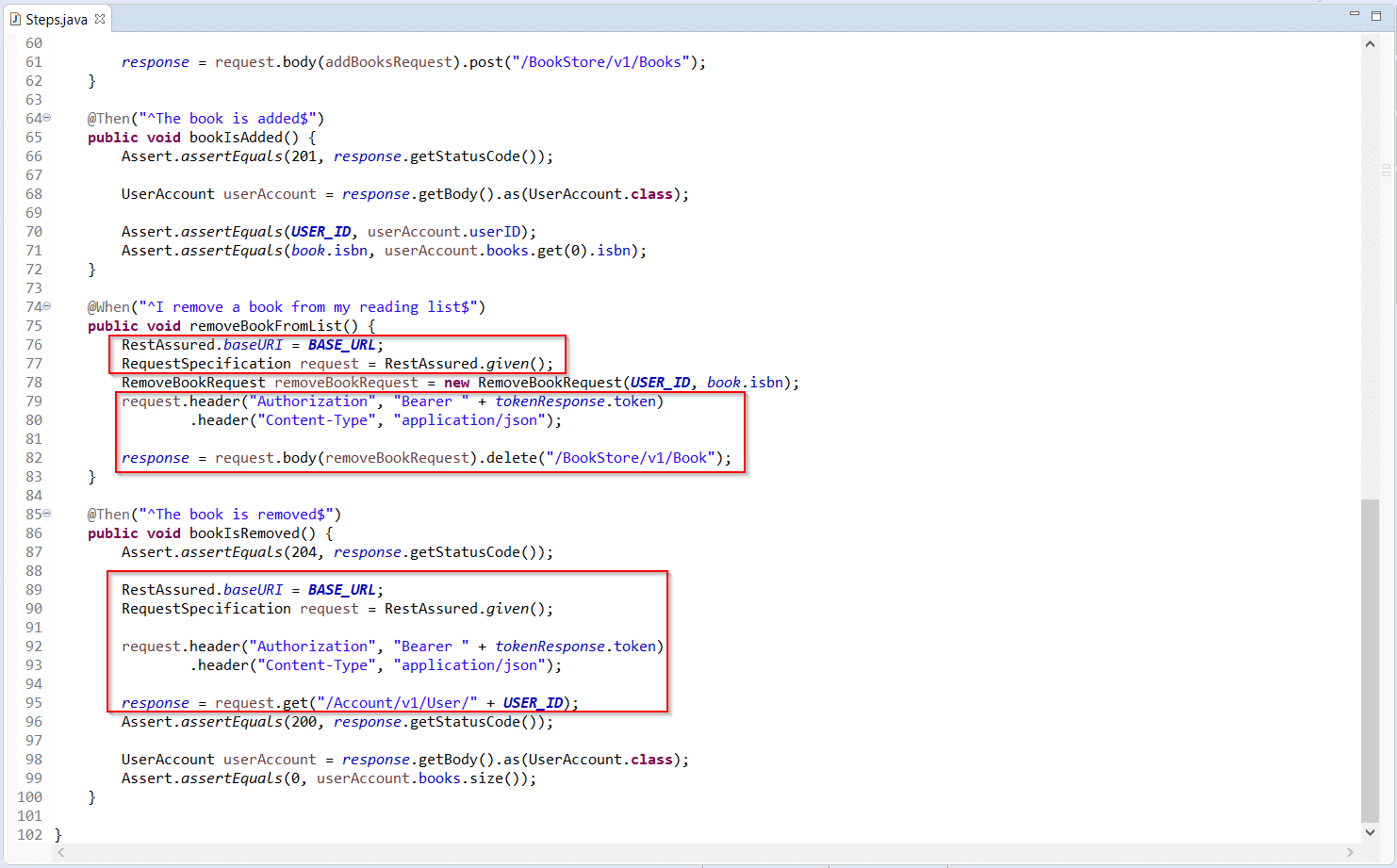
[Convert JSON Response Body to Java Object](https://toolsqa.com/rest-assured/json-response-body-to-java-object/)

**[Next Lesson](https://toolsqa.com/rest-assured/rest-routes/)**

[Implementation of REST Routes](https://toolsqa.com/rest-assured/rest-routes/)

We learned to convert [***JSON Request Body***](https://www.toolsqa.com/rest-assured/convert-json-to-java-object/) as well as [***JSON Response Body***](https://www.toolsqa.com/rest-assured/json-response-body-to-java-object/)  to Java Object in the previous chapters. We used Serialization and Deserialization concepts to achieve this. It led to our tests becoming more maintainable and less error-prone. Now in this chapter, we will work on the ***Separation of Test Layer with API Services*** and take a step further to make our tests cleaner and maintainable.

If you look at the Steps class, which is our test layer, you would see that we have a lot of logic that isn't actually related to testing. It concerns communicating with the server with a given test request and getting a response. Refer to the screenshot below:



In actual projects, we have a large number of APIs and tests implemented. They all contain the same logic of communicating with the server in every test, like:

* Buiding RequestSpecification object
* Adding headers
* Making calls to the server

This leads to a lot of code duplication in every test file. Additionally, if tomorrow there are some changes required in a particular endpoint, we will need to make changes in several places. It is not advisable because it makes our framework less maintainable.

Additionally, the test layer needs to focus only on the test data (parameters) sent in the request and receive responses from the APIs. It should not be focused on the heavy logic of the internals of API implemented. So, as software testers, we are only interested in the request and response obtained for these requests. Moreover, the test data sent in the request generally pass from the feature file in the Cucumber framework. This test data can be parameters to form a request body, request header or request resources.

As we have seen in our first chapter [***Understanding API Documentation***](https://www.toolsqa.com/rest-assured/api-documentation/), endpoints indicate ways to access resources from the server. Thus, we will essentially combine all the logic of ***endpoints*** from our Steps file and move it to a common class. The class will contain methods that take the required request parameters and send back a response received from the server. The endpoints should always get the request body from the test layer. In this article, we are going to cover:-

* Separation of Test Layer with API Services
* Create API Services for the Test
  + Create API Services
  + Add methods of API Services
  + Call API Services in the Step Definitions(test layer)
  + Run the tests

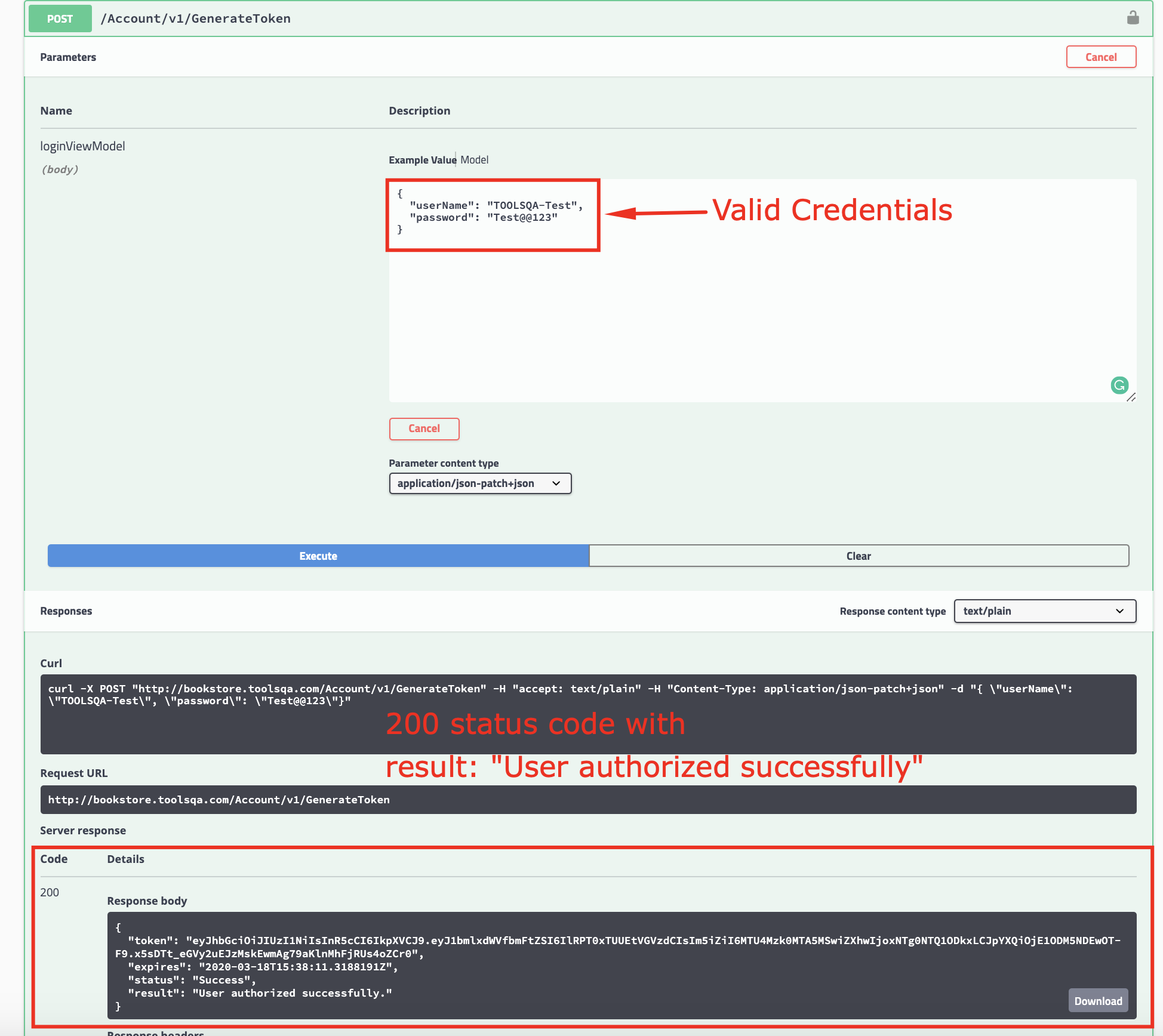
## Separation of Test Layer with API Services

We abstract the logic of communication with the server into a separate class. It will make our Steps class cleaner. Subsequently, as we progress to write more steps into Step-Definition files in the framework, for different scenarios, we can reuse this logic of communicating with the server.

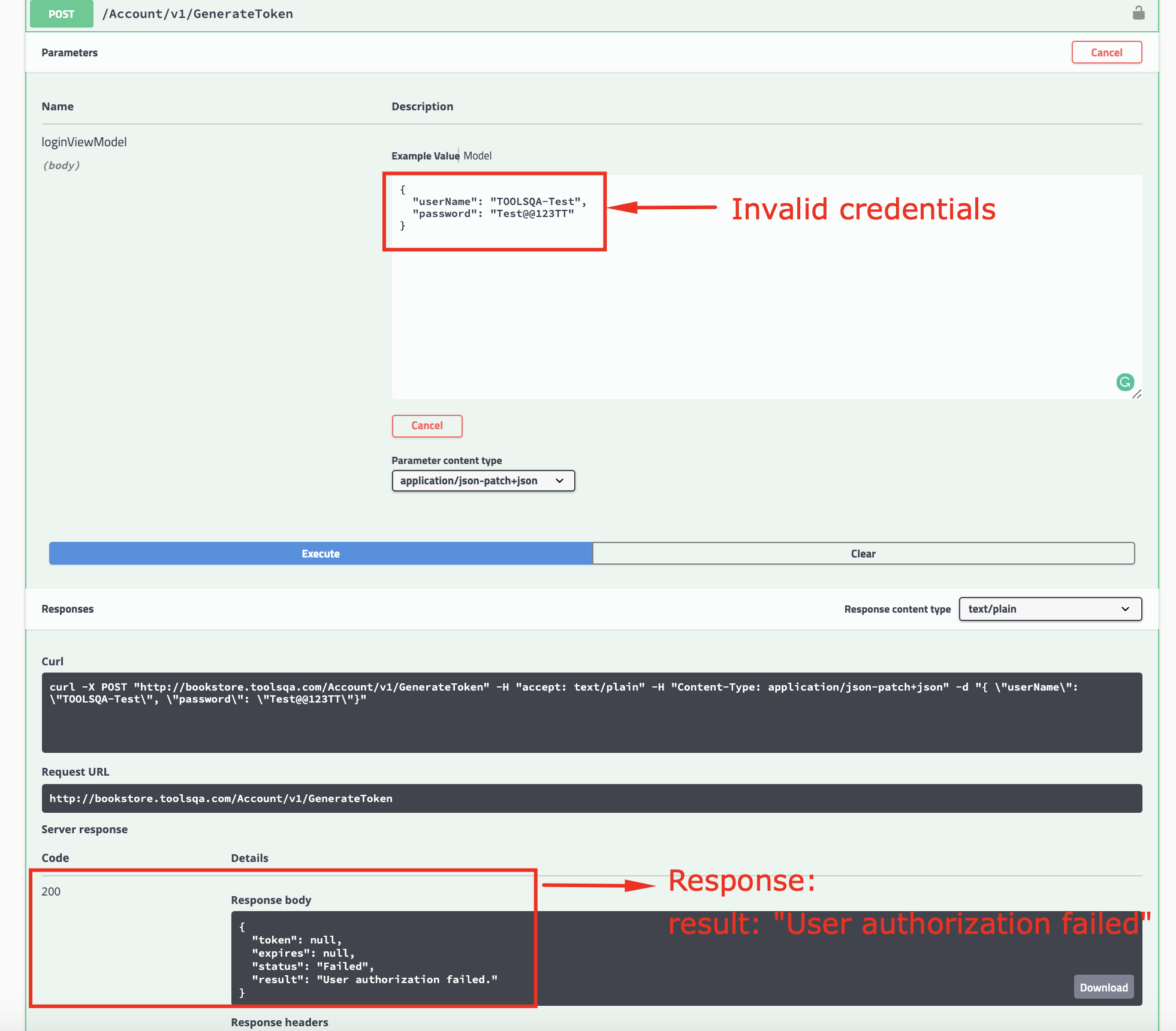
Consider the example to generate a token request of our [***Book Store API***](https://bookstore.toolsqa.com/swagger/index.html): ***/Account/v1/GenerateToken***

For writing tests to test the generate token, our tests should focus the response obtained for the request body passed. In addition to this, they should not focus on internal implementation for the API. In a project with a large number of APIs, it is better to separate the Endpoints from Steps file.

Moreover, if we send correct credentials, we get a success response as a result. Refer below screenshot:



But, if we send invalid credentials, we will get an error response.



Thus, as testers, the focus is on the parameters sent in the body of the request and response we get. It is as same as what we are doing at the Book Store API in Swagger Tool above. We do not have a clue how swagger is making use of HTTP and how it is communicating with the server. The same way the test layer should be agnostic, the fact of how the service layer is handling the request. Additionally, a test layer should only focus on parameters passed to the request, whether it is query parameter, header parameters, or a body parameter, etc.

Let's see how to separate the Service layer from the Test layer.

## Create API Services for the Test

Now we are going to create API Services, which will be consumed by the test layer, and this way, the test layer will be clean and efficiently focus only on test parameters.

1. Firstly, create API Services
2. Secondly, add methods of API Services
3. Thirdly, call API Services in the Step Definitions(test layer)
4. Fourthly, run the tests

### *****Create API Services*****

1. Firstly, Right-click on the apiEngine package and select Class. Additionally, name it as ***Endpoints***. It would be our API Services class.

package apiEngine;

public class Endpoints {

}

We are moving all our endpoints to the Endpoints class. It is rational to move the ***BASE URL*** as well to the Endpoints class from Steps file. Moreover, it will help us to refer to the ***BASE URL*** when sending requests to our endpoints.

Thus, the updated Endpoints class will look likewise:

package apiEngine;

public class Endpoints {

private static final String BASE\_URL = "https://bookstore.toolsqa.com";

}

### *****Add methods of API Services*****

Firstly, we will extract the way to authenticate the user from Steps into Endpoints class. Additionally, we shall pass the object of \*AuthorizationRequest \*to the method.

public static Response authenticateUser(AuthorizationRequest authRequest) {

RestAssured.baseURI = BASE\_URL;

RequestSpecification request = RestAssured.given();

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

Response response = request.body(authRequest).post("/Account/v1/GenerateToken");

return response;

}

***Code Explanation:***

As you can see, we have created a method authenticateUser. In this method, we pass user credentials through AuthorizationRequest object, authRequest in the Request. Moreover, we are passing the BASE\_URL and the headers as part of the request as well.

Similar to the above authenticateUser() method, we will create methods for :

* getBooks()
* addBook()
* removeBook()
* getUserAccount()

Putting all the methods together for the Endpoints class:

***Endpoints.java class***

package apiEngine;

import apiEngine.model.requests.AddBooksRequest;

import apiEngine.model.requests.AuthorizationRequest;

import apiEngine.model.requests.RemoveBookRequest;

import io.restassured.RestAssured;

import io.restassured.response.Response;

import io.restassured.specification.RequestSpecification;

public class EndPoints {

private static final String BASE\_URL = "https://bookstore.toolsqa.com";

public static Response authenticateUser(AuthorizationRequest authRequest) {

RestAssured.baseURI = BASE\_URL;

RequestSpecification request = RestAssured.given();

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

Response response = request.body(authRequest).post("/Account/v1/GenerateToken");

return response;

}

public static Response getBooks() {

RestAssured.baseURI = BASE\_URL;

RequestSpecification request = RestAssured.given();

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

Response response = request.get("/BookStore/v1/Books");

return response;

}

public static Response addBook(AddBooksRequest addBooksRequest, String token) {

RestAssured.baseURI = BASE\_URL;

RequestSpecification request = RestAssured.given();

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

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

Response response = request.body(addBooksRequest).post("/BookStore/v1/Books");

return response;

}

public static Response removeBook(RemoveBookRequest removeBookRequest, String token) {

RestAssured.baseURI = BASE\_URL;

RequestSpecification request = RestAssured.given();

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

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

Response response = request.body(removeBookRequest).delete("/BookStore/v1/Book");

return response;

}

public static Response getUserAccount(String userId, String token) {

RestAssured.baseURI = BASE\_URL;

RequestSpecification request = RestAssured.given();

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

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

Response response = request.get("/Account/v1/User/" + userId);

return response;

}

}

We now move to our next step. In the next step, we will replace the endpoints in the Steps file with our newly created methods.

### *****Talk with API Services in the Step Definitions*****

We have abstracted the logic of communication with the server into an Endpoint class. It will now help us make our Steps class cleaner as we progress into replacing the endpoints with the associated methods. In addition to this, we will be reusing this logic of communicating with the server.

Thus, the first part of the Steps class, with changes in the declared variables will be:

public class Steps {

private static final String USER\_ID = "9b5f49ab-eea9-45f4-9d66-bcf56a531b85";

private static Response response;

private static Token tokenResponse;

private static Book book;

For the step definition:

@Given("I am an authorized user")

is updated as:

@Given("I am an authorized user")

public void iAmAnAuthorizedUser() {

AuthorizationRequest authRequest = new AuthorizationRequest("TOOLSQA-Test", "Test@@123");

response = EndPoints.authenticateUser(authRequest);

tokenResponse = response.getBody().as(Token.class);

}

We will likewise modify the rest of our step definitions. Moreover, we put all these changes for the Step Definitions together into the Steps file.

The updated Steps file will look likewise:

package stepDefinitions;

import java.util.List;

import java.util.Map;

import org.junit.Assert;

import apiEngine.Endpoints;

import apiEngine.model.Book;

import apiEngine.model.requests.AddBooksRequest;

import apiEngine.model.requests.AuthorizationRequest;

import apiEngine.model.requests.ISBN;

import apiEngine.model.requests.RemoveBookRequest;

import apiEngine.model.response.Books;

import apiEngine.model.response.Token;

import apiEngine.model.response.UserAccount;

import io.cucumber.java.en.Given;

import io.cucumber.java.en.Then;

import io.cucumber.java.en.When;

import io.restassured.response.Response;

public class Steps {

private static final String USER\_ID = "9b5f49ab-eea9-45f4-9d66-bcf56a531b85";

private static Response response;

private static Token tokenResponse;

private static Book book;

@Given("I am an authorized user")

public void iAmAnAuthorizedUser() {

AuthorizationRequest authRequest = new AuthorizationRequest("TOOLSQA-Test", "Test@@123");

response = Endpoints.authenticateUser(authRequest);

tokenResponse = response.getBody().as(Token.class);

}

@Given("A list of books are available")

public void listOfBooksAreAvailable() {

response = Endpoints.getBooks();

Books books = response.getBody().as(Books.class);

book = books.books.get(0);

}

@When("I add a book to my reading list")

public void addBookInList() {

ISBN isbn = new ISBN(book.isbn);

AddBooksRequest addBooksRequest = new AddBooksRequest(USER\_ID, isbn);

response = Endpoints.addBook(addBooksRequest, tokenResponse.token);

}

@Then("The book is added")

public void bookIsAdded() {

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

UserAccount userAccount = response.getBody().as(UserAccount.class);

Assert.assertEquals(USER\_ID, userAccount.userID);

Assert.assertEquals(book.isbn, userAccount.books.get(0).isbn);

}

@When("I remove a book from my reading list")

public void removeBookFromList() {

RemoveBookRequest removeBookRequest = new RemoveBookRequest(USER\_ID, book.isbn);

response = Endpoints.removeBook(removeBookRequest, tokenResponse.token);

}

@Then("The book is removed")

public void bookIsRemoved() {

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

response = Endpoints.getUserAccount(USER\_ID, tokenResponse.token);

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

UserAccount userAccount = response.getBody().as(UserAccount.class);

Assert.assertEquals(0, userAccount.books.size());

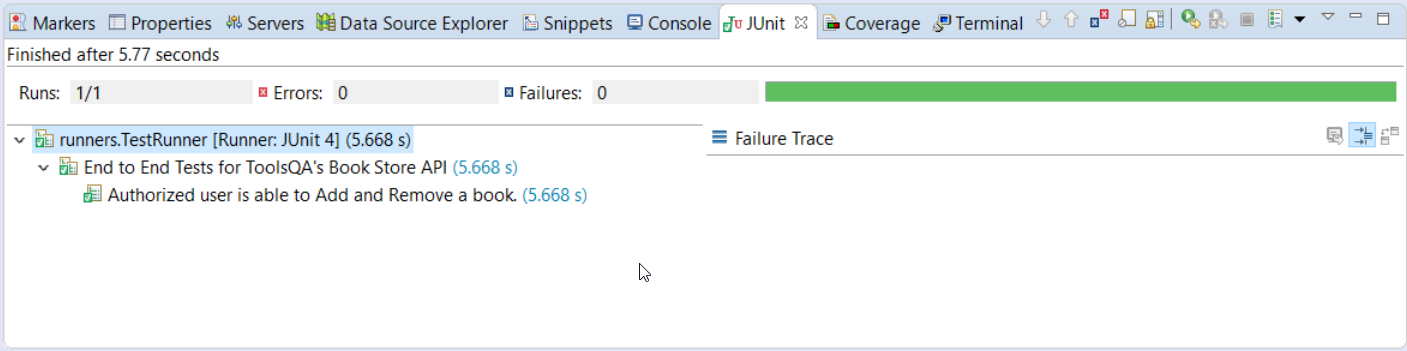
}

}

### *****Run the Test*****

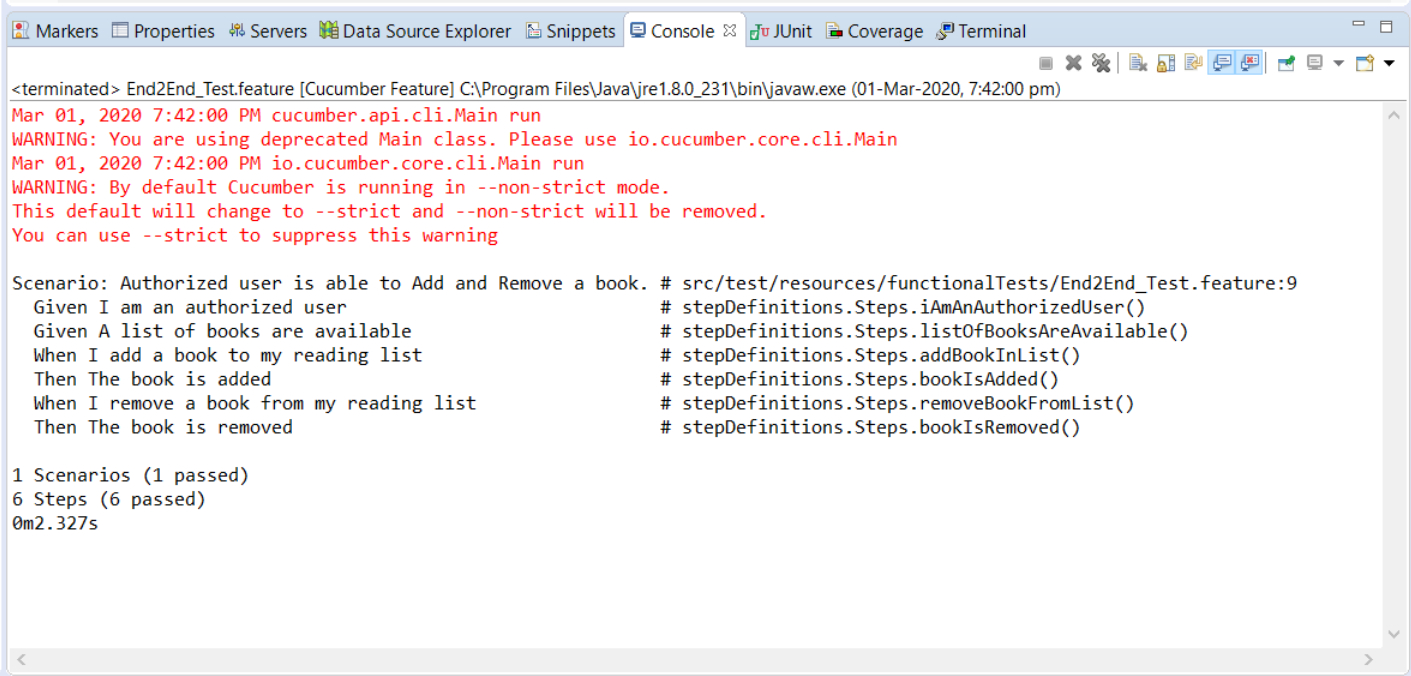
***Run the Tests as JUnit***

We are all set now to run the updated Cucumber test. Firstly, Right -Click on ***TestRunner*** class and Click ***Run As  >> JUnit Test***. Cucumber will run the script in the same fashion as it runs in Selenium WebDriver. Finally, the result will display in the left-hand side project explorer window in the JUnit tab.

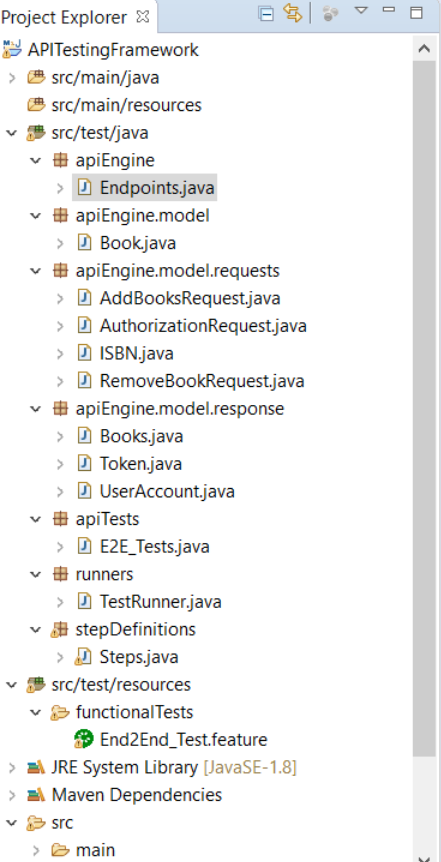


***Run the Tests from Cucumber Feature***

To run the tests as a Cucumber Feature, right-click on the End2End\_Test.feature file. After that, select the Run As>>Cucumber Feature.



Moreover, our updated project folder structure of the framework will look likewise:



In the next chapter, we will introduce the [***Implementation of Routes in Endpoints***](https://www.toolsqa.com/rest-assured/rest-routes/) in our API automation framework. That would help us to make the framework maintainable and we won't have to make changes everywhere when the route changes except for Routes class. Additionally, please implement the Endpoints class as discussed above, and share your valuable feedback with us.

# Implementation of REST Routes

Category: [**Rest-Assured**](https://toolsqa.com/categories/rest-assured) , July 7 2021

0

0

5 min read

**Author**



Yashraj Gore

**Reviewers**



Lakshay Sharma

**[Previous Lesson](https://toolsqa.com/rest-assured/separation-of-test-layer/)**

[Separation of Test Layer with API Services](https://toolsqa.com/rest-assured/separation-of-test-layer/)

**[Next Lesson](https://toolsqa.com/rest-assured/generics-in-api-framework/)**

[Implementation of Generics in API Framework](https://toolsqa.com/rest-assured/generics-in-api-framework/)

In our previous chapter, we introduced the concept of using an ***Endpoints class*** in our framework. As a means to abstract the logic of communication with the server into a separate class and to make our steps cleaner, we introduced this Endpoints class. Subsequently, in this chapter, we are going to take it a step further by introducing the ***REST Routes*** concept in our framework. As you all know, we use an API endpoint to carry out a specific task. Moreover, Routes are the paths through which we can access the endpoints. In simpler terms, we call the ***route as URI***, while an endpoint is an action we perform on the URI.

## What are REST Routes in API?

As we know, client-server communication is one of the REST (representational state transfer) principles. Additionally, it provides a mapping between the HTTP verbs like GET, PUT, POST, DELETE, etc. with CRUD(create, read, update, and delete) operations. We click on links and navigate through various sites. Moreover, in this process, we are merely making a state transition when we navigate to one site from another. The REST-compliant systems are often referred to as RESTful systems.

In addition to the above, the RESTful routes can be akin to a traditional pattern followed when structuring different routes. It interacts with the server each time an HTTP request happens.

For example in the URL,

***http://bookstore.toolsqa.com/BookStore/v1/Books:***

***BasePath***: http://bookstore.toolsqa.com/

***Route:  /BookStore/v1/Books***

***Endpoint:*** There can be multiple tasks we can perform on this route. For example:

|  |  |
| --- | --- |
| GET- http://bookstore.toolsqa.com/BookStore/v1/Books | To fetch a list of books |
| POST- http://bookstore.toolsqa.com/BookStore/v1/Books | To add a book associated with the user |
| DELETE - http://bookstore.toolsqa.com/BookStore/v1/Books | To remove books associated with the user |

It is how the endpoints looked in our Swagger [***Bookstore API***](https://bookstore.toolsqa.com/swagger/index.html)

## Implementation of REST Routes in Framework

We will follow the steps below to implement Routes in our framework:

1. Firstly, Create a Route class
2. Secondly, Modify the Endpoints class
3. Thirdly, Run the tests

### *****Create a Route class*****

Let us add the Routes we discussed above in the first section to our Route class. We will keep all the routes at a single place, and wherever the routes are required, we will use it from this class.

Moreover, the advantage it gives is that suppose any of the route changes. We won't have to make changes everywhere. In other words, it's just at a single place in the Routes class.

1. Firstly, to create a ***Route*** class, Right-click on the ***apiEngine*** Package and select ***New >>Class***. Name it as ***Route***.
2. After that, add the following code snippet to it:

package apiEngine;

public class Route {

private static final String BOOKSTORE = "/BookStore";

private static final String ACCOUNT = "/Account";

private static final String VERSION = "/v1";

public static String generateToken(){

return ACCOUNT + VERSION + "/GenerateToken";

}

public static String books(){

return BOOKSTORE + VERSION + "/Books";

}

public static String book(){

return BOOKSTORE + VERSION + "/Book";

}

public static String userAccount(String userId){

return ACCOUNT + VERSION + "/User" + "/" + userId;

}

}

### *****Modify the Endpoints class for REST Routes*****

We will modify the methods in the endpoints class. Additionally, we will pass the routes from the Routes class instead of writing them to a specific route in the method itself.

For Example:

Our method authenticateUser() is updated from:

public static Response authenticateUser(AuthorizationRequest authRequest) {

RestAssured.baseURI = BASE\_URL;

RequestSpecification request = RestAssured.given();

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

//The hardcoded route "/Account/vi/GenerateToken" will be modified to take the route from the Route class

Response response = request.body(authRequest).post("/Account/v1/GenerateToken");

return response;

}

to

public static Response authenticateUser(AuthorizationRequest authRequest) {

RestAssured.baseURI = BASE\_URL;

RequestSpecification request = RestAssured.given();

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

Response response = request.body(authRequest).post(Route.generateToken());

return response;

}

***Explanation:***

In this method, the route for generating token is now taken from the Route class instead of passing the route ***"/Account/v1/GenerateToken"***.

Similarly, we update the below methods for the Endpoints class:

1. getBooks()
2. addBook()
3. removeBook()
4. getUserAccount()

Consequently, our updated Endpoints class will look likewise:

package apiEngine;

import apiEngine.model.requests.AddBooksRequest;

import apiEngine.model.requests.AuthorizationRequest;

import apiEngine.model.requests.RemoveBookRequest;

import io.restassured.RestAssured;

import io.restassured.response.Response;

import io.restassured.specification.RequestSpecification;

public class EndPoints {

private static final String BASE\_URL = "https://bookstore.toolsqa.com";

public static Response authenticateUser(AuthorizationRequest authRequest) {

RestAssured.baseURI = BASE\_URL;

RequestSpecification request = RestAssured.given();

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

Response response = request.body(authRequest).post(Route.generateToken());

return response;

}

public static Response getBooks() {

RestAssured.baseURI = BASE\_URL;

RequestSpecification request = RestAssured.given();

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

Response response = request.get(Route.books());

return response;

}

public static Response addBook(AddBooksRequest addBooksRequest, String token) {

RestAssured.baseURI = BASE\_URL;

RequestSpecification request = RestAssured.given();

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

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

Response response = request.body(addBooksRequest).post(Route.books());

return response;

}

public static Response removeBook(RemoveBookRequest removeBookRequest, String token) {

RestAssured.baseURI = BASE\_URL;

RequestSpecification request = RestAssured.given();

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

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

Response response = request.body(removeBookRequest).delete(Route.book());

return response;

}

public static Response getUserAccount(String userId, String token) {

RestAssured.baseURI = BASE\_URL;

RequestSpecification request = RestAssured.given();

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

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

Response response = request.get(Route.userAccount(userId));

return response;

}

}

### *****Run the Cucumber Test*****

***Run the Tests as JUnit***

We are all set now to run the updated Cucumber test. First, Right -Click on the ***TestRunner*** class. Secondly, Click ***Run As  >> JUnit Test***. Finally, the result will display in the JUnit tab of the console.

***Run the Tests from Cucumber Feature***

To run the tests as a [***Cucumber Feature***](https://www.toolsqa.com/cucumber/cucumber-jvm-feature-file/), right-click on the End2End\_Test.feature file. After that, select the Run As>>Cucumber Feature.

Our tests passed with the changes we made for the Rest Routes implementation in our framework. Our updated project folder structure of the framework will look likewise:

Subsequently, in the next chapter, we will implement [***generics in our framework***](https://www.toolsqa.com/rest-assured/generics-in-api-framework/) to handle response objects of various data types. Meanwhile, please try and apply the above changes in your framework, as explained.

# Implementation of Generics in API Framework

Category: [**Rest-Assured**](https://toolsqa.com/categories/rest-assured) , November 10 2021

0

0

8 min read

**Author**



Yashraj Gore

**Reviewers**



Lakshay Sharma

**[Previous Lesson](https://toolsqa.com/rest-assured/rest-routes/)**

[Implementation of REST Routes](https://toolsqa.com/rest-assured/rest-routes/)

**[Next Lesson](https://toolsqa.com/rest-assured/refactoring-for-request-headers/)**

[Refactoring for Request Headers](https://toolsqa.com/rest-assured/refactoring-for-request-headers/)

In the previous chapter, we introduced the concept of using [***REST Routes***](https://www.toolsqa.com/rest-assured/rest-routes/) in our Endpoints class for the framework. In case if the route changes, we won't have to make changes everywhere, it's just at a single place in the Routes class.

Subsequently, in this chapter, we are going to implement the Generics concept in our framework. We are dealing with the response objects of various data types such as Token, Books, and User Account. Moreover, one can build these response objects fail-safe by using generics. Generics adds a layer of abstraction. In addition to that, they add a way to specify types for classes and methods. You can read more about the Generics concept in this  [***Generics tutorial***](https://toolsqa.com/java/generics/). In this article, we are going to cover:-

* What is the need for a Generics class implementation?
* Implementation of Generics

## What is the need for a Generics class implementation

Consider an example of returning a [***Book object***](https://bookstore.toolsqa.com/swagger/index.html) from Endpoint method like below:

public static Books getBooks() {

RequestSpecification request = RestAssured.given();

Response response = request.get(Route.books());

return response.getBody().as(Books.class);

}

If we would get a failed response, our code will fail in the above method implementation.

So supposedly if we were to handle the failure like below:

public static Books getBooks() {

RequestSpecification request = RestAssured.given();

Response response = request.get(Route.books());

int code = response.getStatusCode();

if( code == 200 || code == 201 || code == 202 || code == 203 || code == 204 || code == 205) {

return response.getBody().as(Book.Class());

}

}

we would not be able to test negative tests by sending a wrong body and expecting a 204. We would likely be stuck in such a case.

Thus we need a class, which will return the Response Body as well as status and in case of failure return exception or error message.

Now, the response received from the server can be of several data types. Hence, we need an ***interface*** capable of handling different response objects. To provide this parameterized value to a parameterized type, we implement this as a ***Generic Interface***. This interface will contain all the methods we need when we operate on a REST Response.

## Implementation of Generics

We will follow the steps below to implement Generics in our framework:

1. Create a Generic Interface
2. Create a class to implement generic interface methods
3. Modification of the Endpoints class
4. Modification of Steps class
5. Run the tests

### *****Create a Generic Interface*****

Firstly, right-click on the ***apiEngine*** package and select ***New >> Interface***. Name it as ***IRestResponse***.

***IRestResponse.java***

package apiEngine;

import io.restassured.response.Response;

public interface IRestResponse<T>{

public T getBody();

public String getContent();

public int getStatusCode();

public boolean isSuccessful();

public String getStatusDescription();

public Response getResponse();

public Exception getException();

}

***Explanation:***

We created a generic interface of type <T>. Thus, we can use this interface to hold responses of different types. For example, IRestResponse*<Books>* will hold the response of type Books. Likewise, it will be for IRestResponse*<UserAccount>* and IRestResponse*<Token>*.

***Note***: As explained above, we need this interface to return the Body as well as status, and in case of failure return exception or error message. Therefore, that's why we have those attributes in the interface.

#### **Create a class to implement generic interface methods**

We defined the methods we need to operate on the REST response. We need to apply these methods next in our Rest Response class.

Right-click on the a***piEngine*** package and select ***New >> Class***. Name it as ***RestResponse.***

package apiEngine;

import io.restassured.response.Response;

public class RestResponse<T> implements IRestResponse<T> {

private T data;

private Response response;

private Exception e;

public RestResponse(Class<T> t, Response response) {

this.response = response;

try{

this.data = t.newInstance();

}catch (Exception e){

throw new RuntimeException("There should be a default constructor in the Response POJO");

}

}

public String getContent() {

return response.getBody().asString();

}

public int getStatusCode() {

return response.getStatusCode();

}

public boolean isSuccessful() {

int code = response.getStatusCode();

if( code == 200 || code == 201 || code == 202 || code == 203 || code == 204 || code == 205) return true;

return false;

}

public String getStatusDescription() {

return response.getStatusLine();

}

public Response getResponse() {

return response;

}

public T getBody() {

try {

data = (T) response.getBody().as(data.getClass());

}catch (Exception e) {

this.e=e;

}

return data;

}

public Exception getException() {

return e;

}

}

***Explanation:***

We implemented the methods to return us relevant details of REST responses as needed for testing.

* ***isSuccessful()***: Will validate if the sending request was successful. It validates the response status code received against the numerous HTTP status codes denoting that request was successfully processed
* ***getResponse().getBody().asString()***: We will need the response body content in String format at times. This method implementation takes care of it.
* ***getException()***: In case our response body is not deserialized successfully, we will get an exception. e will contain this exception, which we get using this method.

### *****Modification of Endpoints class*****

Our methods for Endpoints class will change. They will return responses of the type RestResponse*<Books>*, RestResponse*<Token>* and RestResponse*<UserAccount>* for the respective methods.

For Example: Our method authenticateUser() is updated from:

public static Response authenticateUser(AuthorizationRequest authRequest) {

RestAssured.baseURI = BASE\_URL;

RequestSpecification request = RestAssured.given();

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

Response response = request.body(authRequest).post(Route.generateToken());

return response;

}

to

public static IRestResponse<Token> authenticateUser(AuthorizationRequest authRequest) {

RestAssured.baseURI = BASE\_URL;

        RequestSpecification request = RestAssured.given();

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

       Response response = request.body(authRequest).post(Route.generateToken());

       return new RestResponse(Token.class, response);

}

***Explanation:***

In this method, we have wrapped the Rest Assured Response into our RestResponse class of the type Token, where we have deserialized the responses.

Similarly, we updated the below methods for the Endpoints class:

1. getBooks()
2. addBook()
3. removeBook()
4. getUserAccount()

Our updated Endpoints class will look likewise:

package apiEngine;

import apiEngine.model.requests.AddBooksRequest;

import apiEngine.model.requests.AuthorizationRequest;

import apiEngine.model.requests.RemoveBookRequest;

import apiEngine.model.responses.Books;

import apiEngine.model.responses.Token;

import apiEngine.model.responses.UserAccount;

import io.restassured.RestAssured;

import io.restassured.response.Response;

import io.restassured.specification.RequestSpecification;

public class EndPoints {

private static final String BASE\_URL = "https://bookstore.toolsqa.com";

public static IRestResponse<Token> authenticateUser(AuthorizationRequest authRequest) {

RestAssured.baseURI = BASE\_URL;

RequestSpecification request = RestAssured.given();

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

Response response = request.body(authRequest).post(Route.generateToken());

return new RestResponse(Token.class, response);

}

public static IRestResponse<Books> getBooks() {

RestAssured.baseURI = BASE\_URL;

RequestSpecification request = RestAssured.given();

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

Response response = request.get(Route.books());

return new RestResponse(Books.class, response);

}

public static IRestResponse<UserAccount> addBook(AddBooksRequest addBooksRequest, String token) {

RestAssured.baseURI = BASE\_URL;

RequestSpecification request = RestAssured.given();

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

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

Response response = request.body(addBooksRequest).post(Route.books());

return new RestResponse(UserAccount.class, response);

}

public static Response removeBook(RemoveBookRequest removeBookRequest, String token) {

RestAssured.baseURI = BASE\_URL;

RequestSpecification request = RestAssured.given();

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

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

return request.body(removeBookRequest).delete(Route.book());

}

public static IRestResponse<UserAccount> getUserAccount(String userId, String token) {

RestAssured.baseURI = BASE\_URL;

RequestSpecification request = RestAssured.given();

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

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

Response response = request.get(Route.userAccount(userId));

return new RestResponse(UserAccount.class, response);

}

}

### *****Modification of Steps class*****

We will modify the step definitions to call the methods listed in the endpoints class.

Moreover, you will directly obtain the response in the step definition class. As already explained, the logic of communication with the server and converting it into the response class moves out. Thus, our step definition consists of only the testing layer which we are interested in and not the internal workings of the API.

Our updated step definition file would look like:

package stepDefinitions;

import apiEngine.EndPoints;

import apiEngine.IRestResponse;

import apiEngine.model.\*;

import apiEngine.model.requests.\*;

import apiEngine.model.responses.\*;

import org.junit.Assert;

import cucumber.api.java.en.Given;

import cucumber.api.java.en.Then;

import cucumber.api.java.en.When;

import io.restassured.response.Response;

public class Steps {

private static final String USER\_ID = "9b5f49ab-eea9-45f4-9d66-bcf56a531b85";

private static Response response;

private static Token tokenResponse;

private static IRestResponse<UserAccount> userAccountResponse;

private static Book book;

@Given("^I am an authorized user$")

public void iAmAnAuthorizedUser() {

AuthorizationRequest authRequest = new AuthorizationRequest("TOOLSQA-Test", "Test@@123");

tokenResponse = EndPoints.authenticateUser(authRequest).getBody();

}

@Given("^A list of books are available$")

public void listOfBooksAreAvailable() {

IRestResponse<Books> booksResponse = EndPoints.getBooks();

book = booksResponse.getBody().books.get(0);

}

@When("^I add a book to my reading list$")

public void addBookInList() {

ISBN isbn = new ISBN(book.isbn);

AddBooksRequest addBooksRequest = new AddBooksRequest(USER\_ID, isbn);

userAccountResponse = EndPoints.addBook(addBooksRequest, tokenResponse.token);

}

@Then("^The book is added$")

public void bookIsAdded() {

Assert.assertTrue(userAccountResponse.isSuccessful());

Assert.assertEquals(201, userAccountResponse.getStatusCode());

Assert.assertEquals(USER\_ID, userAccountResponse.getBody().userID);

Assert.assertEquals(book.isbn, userAccountResponse.getBody().books.get(0).isbn);

}

@When("^I remove a book from my reading list$")

public void removeBookFromList() {

RemoveBookRequest removeBookRequest = new RemoveBookRequest(USER\_ID, book.isbn);

response = EndPoints.removeBook(removeBookRequest, tokenResponse.token);

}

@Then("^The book is removed$")

public void bookIsRemoved() {

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

userAccountResponse = EndPoints.getUserAccount(USER\_ID, tokenResponse.token);

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

Assert.assertEquals(0, userAccountResponse.getBody().books.size());

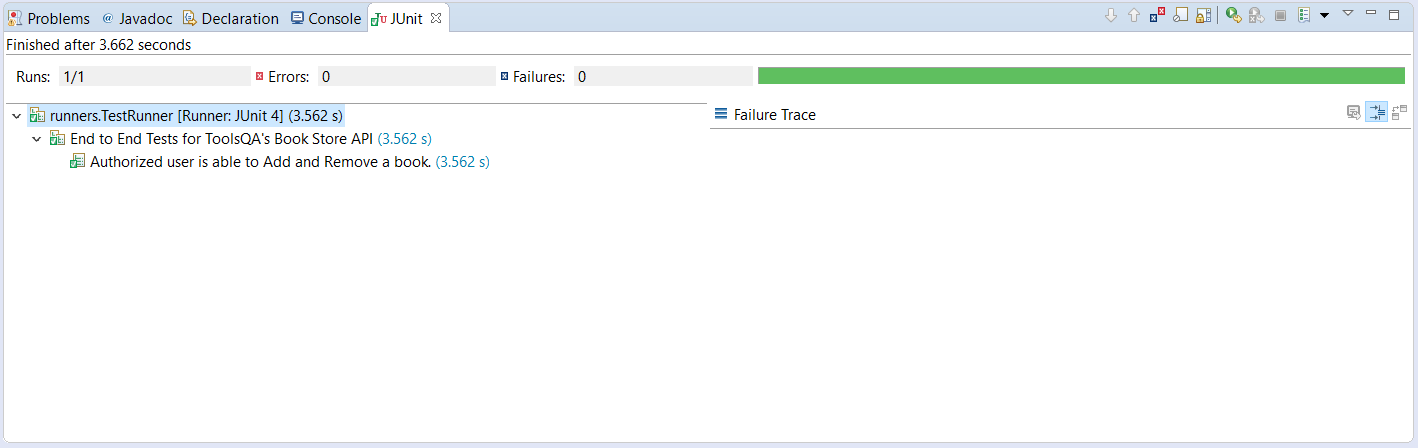
}

}

#### **Run the Cucumber Test**

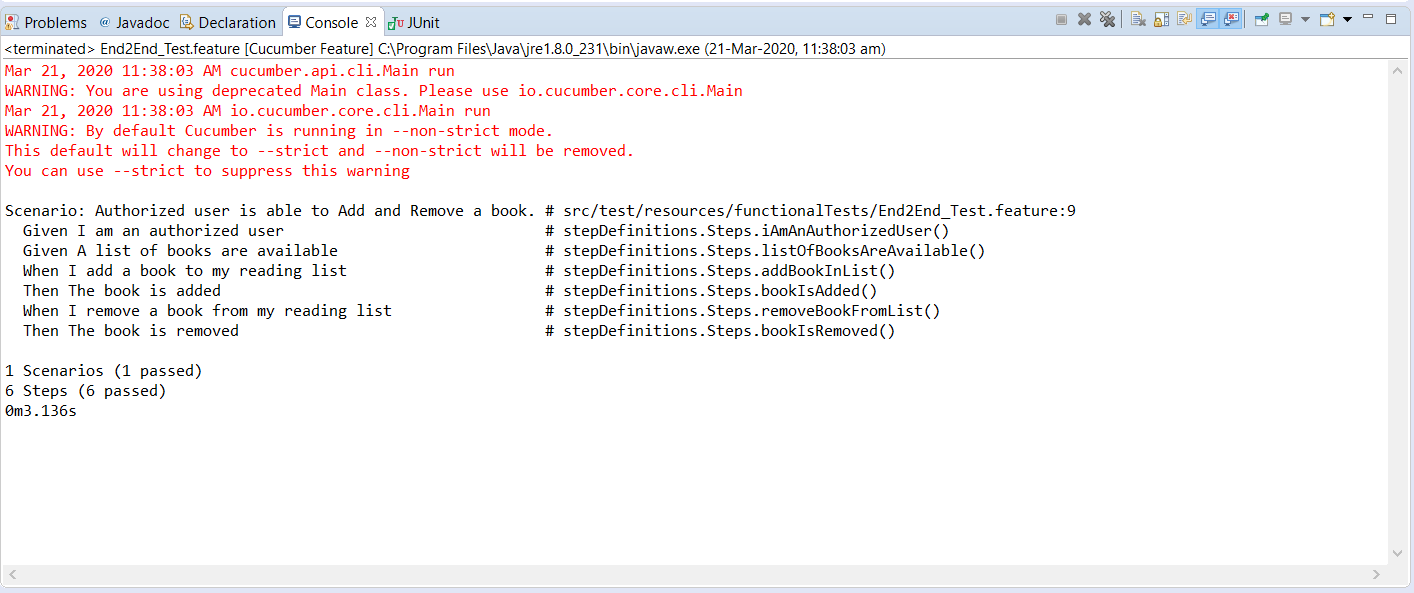
***Run the Tests as JUnit***

We are all set now to run the updated Cucumber test. Right -Click on ***TestRunner*** class and Click ***Run As  >> JUnit Test***. Consequently, you will see the result in the left-hand side project explorer window in the JUnit tab.

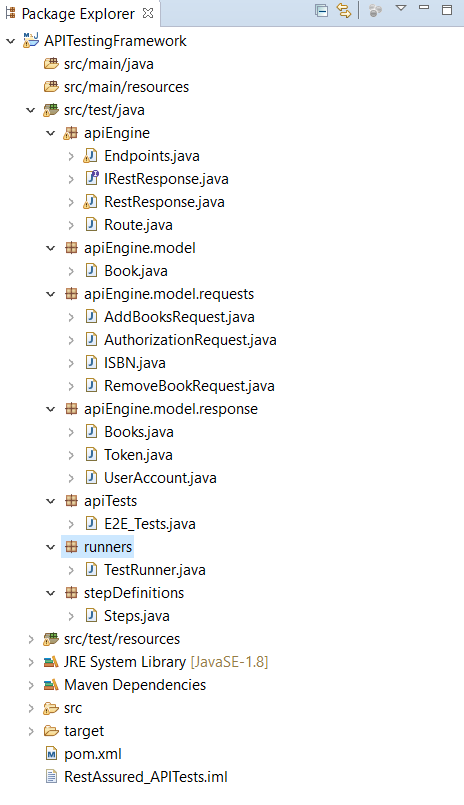


***Run the Tests from Cucumber Feature***

To run the tests as a Cucumber Feature, right-click on the End2End\_Test.feature file. After that, select the ***Run As>>Cucumber Feature***.



Our updated project folder structure of the framework will look likewise:



Our tests passed with the changes we made for the Generics implementation in our framework. We will be [***Refactoring for Request Headers***](https://www.toolsqa.com/rest-assured/refactoring-for-request-headers/) so that we can make use of a single request object in our next chapter. Moreover, it will avoid the complexity of adding the auth header repeatedly for each request.

Meanwhile, please try to implement the above changes in your framework, as explained above.

# Refactoring for Request Headers

Category: [**Rest-Assured**](https://toolsqa.com/categories/rest-assured) , July 7 2021

0

0

6 min read

**Author**



Yashraj Gore

**Reviewers**



Lakshay Sharma

**[Previous Lesson](https://toolsqa.com/rest-assured/generics-in-api-framework/)**

[Implementation of Generics in API Framework](https://toolsqa.com/rest-assured/generics-in-api-framework/)

**[Next Lesson](https://toolsqa.com/rest-assured/share-test-context/)**

[Share Test Context in Cucumber](https://toolsqa.com/rest-assured/share-test-context/)

In the previous chapter, we introduced the usage of [***Generics for the Endpoints class***](https://www.toolsqa.com/rest-assured/generics-in-api-framework/) of our application framework. It helped us to handle different response objects we get. In this chapter, we will change the static methods of ***Endpoints.java*** class into instance methods and work on ***Refactoring Request Headers***. It is also important to understand the difference between the [***Static and Instance Method***](https://www.geeksforgeeks.org/static-methods-vs-instance-methods-java/) in programming.

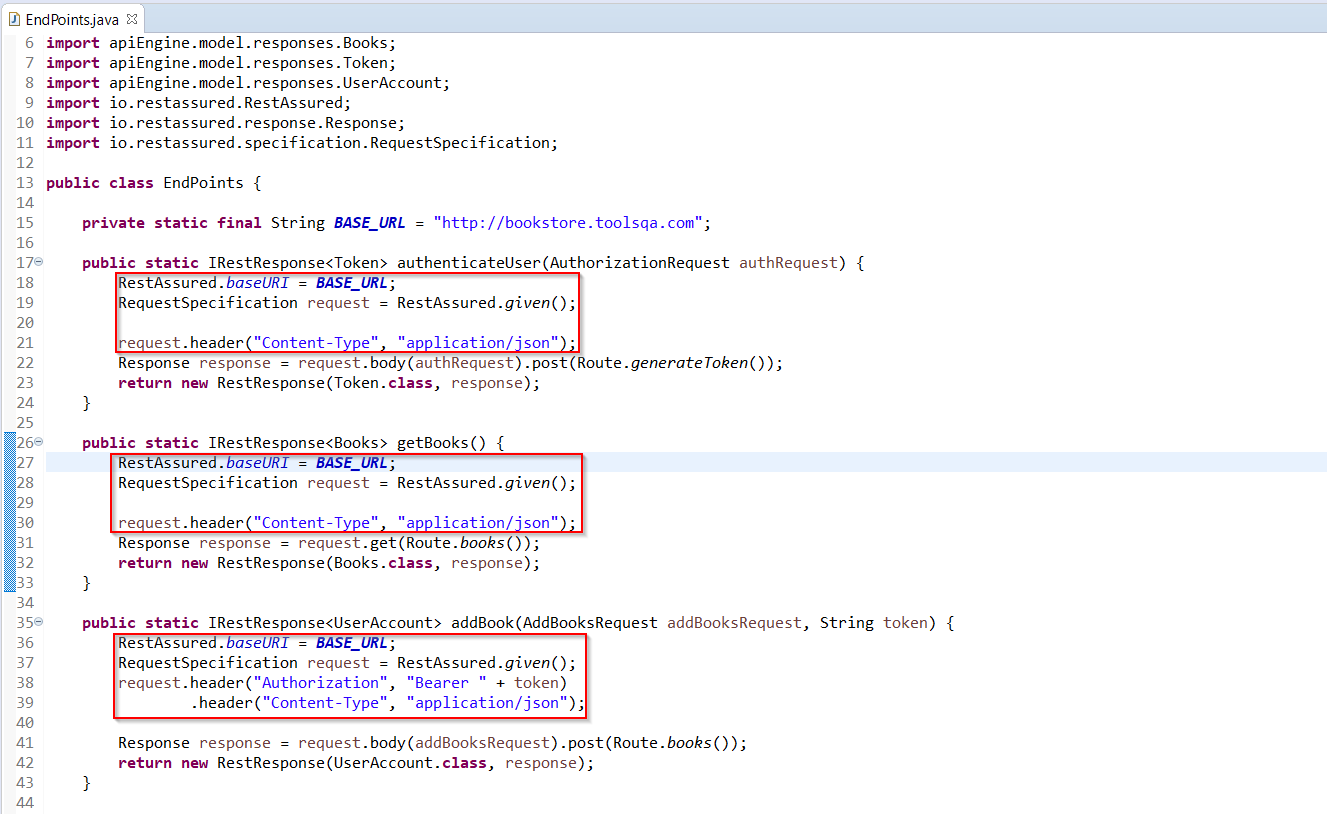
By converting the EndPoint class into an Instance class, it will help us to make use of a ***Single Request Object***. It will save us the trouble of adding the request header each time we pass the request. Here are the topics which we are going to cover in this article:-

* Need for Refactoring Request Headers
* Steps for Refactoring Request Headers
  + Convert static methods into instance methods
  + Update the Steps file
  + Run the tests

So let's get started.

## Need for Refactoring Request Headers

Consider the methods from the below image of ***Endpoints.java*** class:



We are sending this highlighted piece of code - the ***BASE\_URL***, the ***headers*** in every method, as we call our methods in the Step Definitions. It leads to us creating the ***RequestSpecification*** object again and again when it is the same for every step. It would be simpler if we created this once for all the steps.

If one uses the static methods, it causes random failures in the tests as they turn flaky if the tests run in parallel in the future course. Every thread tries to access the available static resource. During this, the thread could come in a situation where another thread is in a position to access the same static resources. Thus, to avoid the test failures during parallel execution.

To achieve this, we will need to initialize the RequestSpecification object in the constructor. Therefore now we will need to make all these methods in ***Endpoints.java*** class as non-static.

***But what we would achieve from this for the framework?***  Once the authentication is set in the RequestSpecification object, it is good for all the subsequent requests. It is not required to set it again during the next request.

Let's see how to do this.

## Steps for Refactoring Request Headers

To refactor the existing code to create just a single request object, we follow these steps:

1. Convert static methods into instance methods
2. Update the Steps file
3. Run the tests

So let's begin with our first step.

### *****Convert Static Methods into Instance Methods*****

The changes in the Endpoints.java class would be:

package apiEngine;

import org.apache.http.HttpStatus;

import apiEngine.model.requests.AddBooksRequest;

import apiEngine.model.requests.AuthorizationRequest;

import apiEngine.model.requests.RemoveBookRequest;

import apiEngine.model.responses.Books;

import apiEngine.model.responses.Token;

import apiEngine.model.responses.UserAccount;

import io.restassured.RestAssured;

import io.restassured.response.Response;

import io.restassured.specification.RequestSpecification;

public class EndPoints {

private final RequestSpecification request;

public EndPoints(String baseUrl) {

RestAssured.baseURI = baseUrl;

request = RestAssured.given();

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

}

public void authenticateUser(AuthorizationRequest authRequest) {

Response response = request.body(authRequest).post(Route.generateToken());

if (response.statusCode() != HttpStatus.SC\_OK)

throw new RuntimeException("Authentication Failed. Content of failed Response: " + response.toString() + " , Status Code : " + response.statusCode());

Token tokenResponse = response.body().jsonPath().getObject("$", Token.class);

request.header("Authorization", "Bearer " + tokenResponse.token);

}

public IRestResponse<Books> getBooks() {

Response response = request.get(Route.books());

return new RestResponse(Books.class, response);

}

public IRestResponse<UserAccount> addBook(AddBooksRequest addBooksRequest) {

Response response = request.body(addBooksRequest).post(Route.books());

return new RestResponse(UserAccount.class, response);

}

public Response removeBook(RemoveBookRequest removeBookRequest) {

return request.body(removeBookRequest).delete(Route.book());

}

public IRestResponse<UserAccount> getUserAccount(String userId) {

Response response = request.get(Route.userAccount(userId));

return new RestResponse(UserAccount.class, response);

}

}

***Code Explanation:***

As you can see in the above code, we have created a constructor which takes care of initializing the RequestSpecification object with ***baseURL*** and ***Request Headers*** in the constructor when the Endpoints class initializes in the Steps file.

We also need token to be passed as well in request Header. For that in the method authenticateUser() when we get tokenResponse, we set it in the same instance of ***request.*** This header will then be available when we make subsequent calls to the server. Since authenticating the user will always be the first step, we will not need to add the token in the header for every request we make after that.

### *****Update the Steps file for the above change*****

Our Steps file too would change as per the ***Endpoints.java*** class

package stepDefinitions;

import apiEngine.EndPoints;

import apiEngine.IRestResponse;

import apiEngine.model.\*;

import apiEngine.model.requests.\*;

import apiEngine.model.responses.\*;

import org.junit.Assert;

import cucumber.api.java.en.Given;

import cucumber.api.java.en.Then;

import cucumber.api.java.en.When;

import io.restassured.response.Response;

public class Steps {

private final String USER\_ID = "9b5f49ab-eea9-45f4-9d66-bcf56a531b85";

private Response response;

private IRestResponse<UserAccount> userAccountResponse;

private Book book;

private final String BaseUrl = "https://bookstore.toolsqa.com";

private EndPoints endPoints;

@Given("^I am an authorized user$")

public void iAmAnAuthorizedUser() {

endPoints = new EndPoints(BaseUrl);

AuthorizationRequest authRequest = new AuthorizationRequest("TOOLSQA-Test", "Test@@123");

endPoints.authenticateUser(authRequest);

}

@Given("^A list of books are available$")

public void listOfBooksAreAvailable() {

IRestResponse<Books> booksResponse = endPoints.getBooks();

book = booksResponse.getBody().books.get(0);

}

@When("^I add a book to my reading list$")

public void addBookInList() {

ISBN isbn = new ISBN(book.isbn);

AddBooksRequest addBooksRequest = new AddBooksRequest(USER\_ID, isbn);

userAccountResponse = endPoints.addBook(addBooksRequest);

}

@Then("^The book is added$")

public void bookIsAdded() {

Assert.assertTrue(userAccountResponse.isSuccessful());

Assert.assertEquals(201, userAccountResponse.getStatusCode());

Assert.assertEquals(USER\_ID, userAccountResponse.getBody().userID);

Assert.assertEquals(book.isbn, userAccountResponse.getBody().books.get(0).isbn);

}

@When("^I remove a book from my reading list$")

public void removeBookFromList() {

RemoveBookRequest removeBookRequest = new RemoveBookRequest(USER\_ID, book.isbn);

response = endPoints.removeBook(removeBookRequest);

}

@Then("^The book is removed$")

public void bookIsRemoved() {

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

userAccountResponse = endPoints.getUserAccount(USER\_ID);

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

Assert.assertEquals(0, userAccountResponse.getBody().books.size());

}

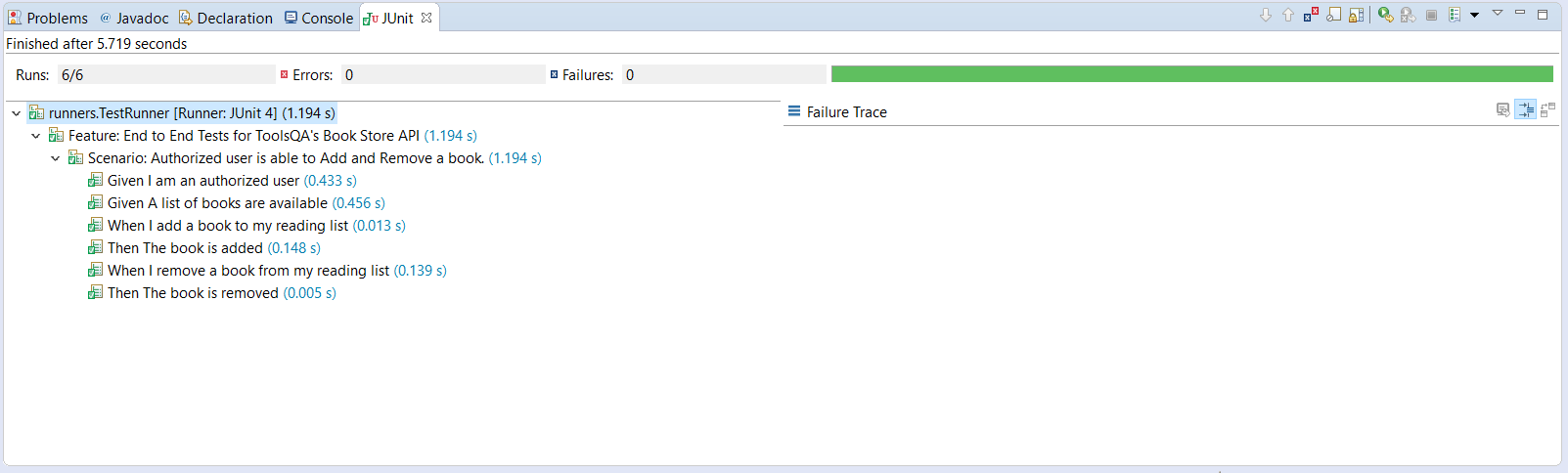
}

***Code Explanation:***

We have initialized an Endpoints class object-***endpoints*** and passed the ***baseURL*** in the step ***iAmAnAuthorizedUser()*** by invoking a constructor of Endpoints class. Also, this same endpoint object is being used by all the step definitions instead of calling static Endpoint methods.

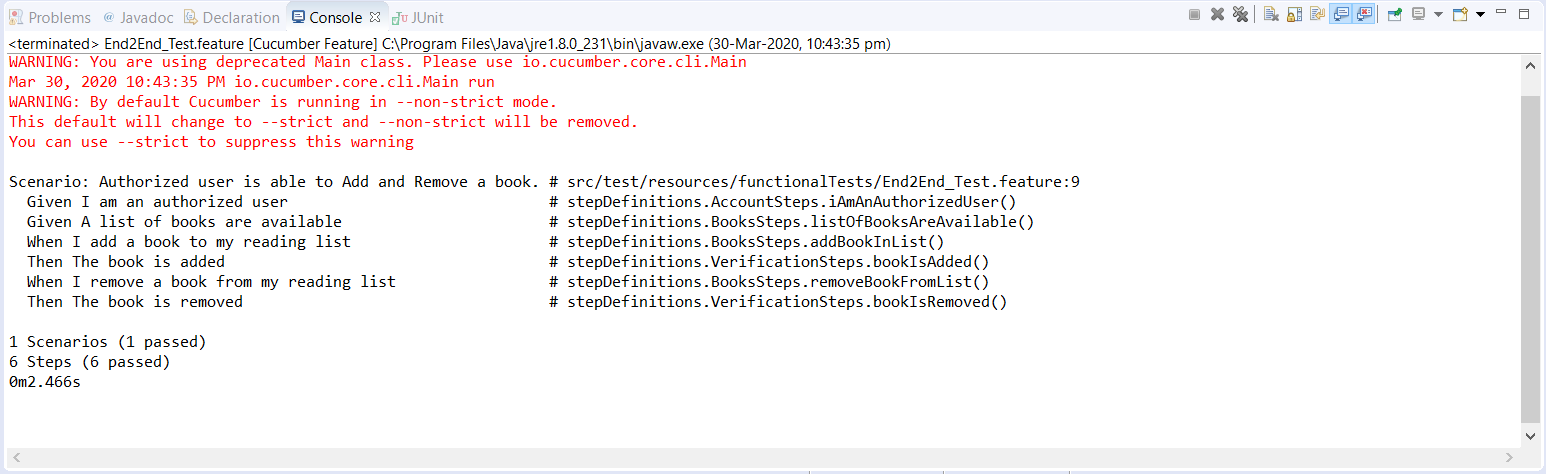
### *****Run the tests*****

We are all set now to run the updated Cucumber test. Subsequently, we will Right-Click on the ***TestRunner*** class. After that, we will Click ***Run As  >> JUnit Test.*** Consequently, the result will appear in the left-hand side project explorer window in the JUnit tab.

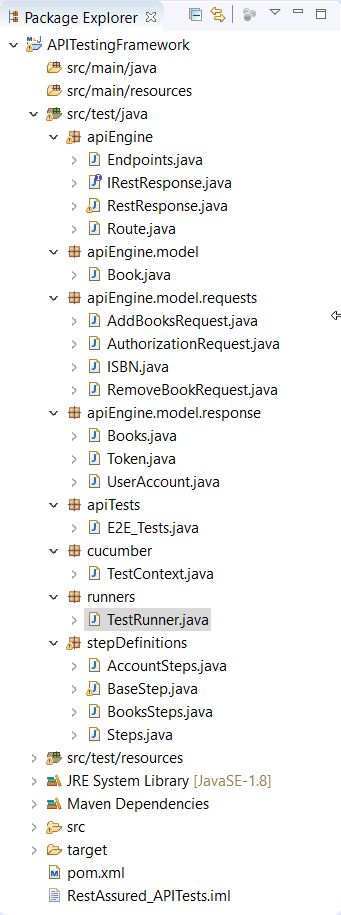


***Run the Tests from Cucumber Feature***

To run the tests as a Cucumber Feature, right-click on the End2End\_Test.feature file. Select the ***Run As>>Cucumber Feature***.



Our updated project folder structure of the framework will look likewise:



Our tests passed with the changes we made for the Refactoring of the Request Headers in our framework. We will learn to [***share Test Context between our Step Definitions***](https://www.toolsqa.com/rest-assured/share-test-context/) so that we can have the step definition file uncluttered and share the ***Test Context*** with all the ***Step Definition*** files. It will avoid the complexity of adding the auth header repeatedly for each request.

* [**Home**](https://toolsqa.com/)
* [**Selenium Training**](https://toolsqa.com/selenium-training?q=headers)
* [**Demo Site**](https://demoqa.com/)
* [**About**](https://toolsqa.com/about)

Top of Form



Bottom of Form

**Tutorials**

* [Home](https://toolsqa.com/)

* [Rest-Assured](https://toolsqa.com/categories/rest-assured)

* Share Test Context in Cucumber
* **Rest Assured Tutorial**
* **Client Server Basics**
* **RESTful Basics**
* **REST API Testing - Basics**
* **REST API Testing - Advance**
* **JSON Manipulations**
* **REST API Framework**
  + [What is API Documentation?](https://toolsqa.com/rest-assured/api-documentation/)
  + [REST API End to End Test](https://toolsqa.com/rest-assured/rest-api-end-to-end-test/)
  + [REST API Test in Cucumber](https://toolsqa.com/rest-assured/rest-api-test-in-cucumber/)
  + [Convert JSON Request Body to POJO](https://toolsqa.com/rest-assured/convert-json-to-java-object/)
  + [Convert JSON Response Body to POJO](https://toolsqa.com/rest-assured/json-response-body-to-java-object/)

## [Se Need for Sharing the Test Context between Cucumber Step Definitions](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[In real-life projects, the expectation is we build a scalable automation framework. Moreover, automation coverage increases as we add multiple scenarios. If a single step definition file contains all the step definitions, it clutters the step definition file. Additionally, it becomes non-maintainable. It is thus, imperative to refactor the step definitions separated under different classes.](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[In any scenario of a feature file in Cucumber, a series of steps get executed one after one. Every step of the scenario may possess a state which could prove useful to another step in the same scenario. Thus, the steps depend on the previously executed steps. Moreover, for this reason, sharing the state among the steps is needed.](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[In our framework, we have just one scenario for illustration purposes. But, it will grow over a while as we add numerous scenarios. Additionally, to keep the step definition file uncluttered and share the](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)***[Test Context](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)***[with all the](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)***[Step Definition](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)***[files, we will resort to the dependency injection](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)***[Container PicoContainer](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)***[. We have implemented this before in our former](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)***[Cucumber tutorial with Selenium](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)***[on](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)[***[sharing Test Context with Cucumber Step Definitions](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)***](https://www.toolsqa.com/selenium-cucumber-framework/sharing-test-context-between-cucumber-step-definitions/)[.](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[As a next step, we will learn the way to implement the Test Context sharing between our Step Definitions in our API Rest Assured Automation Framework.](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

**[How to Share the Test Context between Cucumber Steps using PicoContainer](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)**

[We will follow the same steps as](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)[***[sharing Test Context with Cucumber Step Definitions](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)***](https://www.toolsqa.com/selenium-cucumber-framework/sharing-test-context-between-cucumber-step-definitions/)[to share the data state across steps:](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

1. *[Firstly, add PicoContainer to the Project.](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)*
2. *[Secondly, create a Test Context class to hold all the objects state.](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)*
3. *[Thirdly, divide the Steps class into multiple steps classes.](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)*
4. *[Fourth, write Constructor to share Test Context.](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)*
5. *[Finally, run the Tests.](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)*

***[Add PicoContainer Library to the Maven Project](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)***

[Add the maven dependency of dependency injection-PicoContainer into our project pom.xml. Additionally, you can find more details about the version etc. at [](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)***[Maven Repository - Cucumber PicoContainer](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)***[]](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[(https://mvnrepository.com/artifact/io.cucumber/cucumber-picocontainer).](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[<!-- https://mvnrepository.com/artifact/io.cucumber/cucumber-picocontainer -->](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[<dependency>](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[<groupId>io.cucumber</groupId>](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[<artifactId>cucumber-picocontainer</artifactId>](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[<version>5.5.0</version>](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[<scope>test</scope>](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[</dependency>](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

***[Create a Test Context class which will hold all the objects state](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)***

[The Text Context class shall encompass all information your Steps files are using.  In our framework, we are likewise using the information from the Endpoints class in the Steps file. So simply, we need to add an object of Endpoints class into this Test Context class.](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[To create a Test Context class,](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

1. [Firstly,](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)*[Right-click](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)*[on the](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)***[src/test/java](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)***[and select](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)***[New >> Package.](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)***[Moreover, name it as a](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)***[cucumber](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)***[. All the Cucumber Helper classes will be in the same package hereon.](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)
2. [Secondly,](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)*[Right-click](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)*[on the above-created package and select](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)***[New >> Class](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)***[. Name it as](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)***[TestContext.](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)***

***[TestContext.java](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)***

[package cucumber;](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[import apiEngine.EndPoints;](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[public class TestContext {](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[private String BASE\_URL = "https://bookstore.toolsqa.com";](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[private EndPoints endPoints;](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[public TestContext() {](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[endPoints = new EndPoints(BASE\_URL);](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[}](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[public EndPoints getEndPoints() {](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[return endPoints;](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[}](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[}](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[Consequently, we kept the initialization of Endpoints in the constructor and created](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)***[getEndPoints()](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)***[for the object.](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

***[Divide the Steps class into multiple steps classes](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)***

[We can logically divide our endpoints into Account Steps and Book Steps. Some of the necessary steps would include in the BaseStep.](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[Create three](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)***[New Classes](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)***[in the](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)***[stepDefinitions](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)***[package with the following names:](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

* *[AccountSteps](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)*
* *[BooksSteps](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)*
* *[BaseStep](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)*

***[Note](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)****[: You may replace the Steps file with BaseSteps during clean up by moving the required step definitions to associated classes. Alternatively, you can create a BaseStep class, and once we completed this tutorial, then delete the Steps file as we no longer would be using it.](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)*

***[Write Constructor to share Test Context](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)***

[We divided the Step Definitions to different Step classes. These classes commonly share the Endpoints class amongst themselves. Additionally, in real-life projects, we may have such multiple classes that have an extensive requirement for every Step class. Moreover, to create objects for all standard classes using a new operator, again and again, is not advisable.](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[If we add a](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)**[constructor](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)**[to our Steps class file and pass](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)***[TestContext as a Parameter to the constructor,](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)***[it would solve all the problems. In the](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)*[TestContext](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)*[object, we got everything required for the test.](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[Thus the](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)***[AccountsSteps.java](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)***[class will look as:](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[package stepDefinitions;](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[import apiEngine.model.requests.\*;](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[import cucumber.TestContext;](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[import cucumber.api.java.en.Given;](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[public class AccountSteps extends BaseStep {](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[public AccountSteps(TestContext testContext){](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[super(testContext);](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[}](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[@Given("^I am an authorized user$")](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[public void iAmAnAuthorizedUser() {](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[AuthorizationRequest authRequest = new AuthorizationRequest("TOOLSQA-Test", "Test@@123");](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[getEndPoints().authenticateUser(authRequest);](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[}](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[}](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[Likewise, the changes for](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)***[BooksSteps.java](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)***[class is:](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[package stepDefinitions;](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[import apiEngine.IRestResponse;](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[import apiEngine.model.Book;](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[import apiEngine.model.requests.AddBooksRequest;](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[import apiEngine.model.requests.ISBN;](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[import apiEngine.model.requests.RemoveBookRequest;](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[import apiEngine.model.responses.Books;](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[import apiEngine.model.responses.UserAccount;](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[import cucumber.TestContext;](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[import cucumber.api.java.en.Given;](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[import cucumber.api.java.en.Then;](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[import cucumber.api.java.en.When;](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[import io.restassured.response.Response;](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[import org.junit.Assert;](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[public class BooksSteps extends BaseStep {](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[public BooksSteps(TestContext testContext){](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[super(testContext);](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[}](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[private final String USER\_ID = "9b5f49ab-eea9-45f4-9d66-bcf56a531b85";](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[private Response response;](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[private IRestResponse<UserAccount> userAccountResponse;](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[private Book book;](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[@Given("^A list of books are available$")](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[public void listOfBooksAreAvailable() {](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[IRestResponse<Books> booksResponse = getEndPoints().getBooks();](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[book = booksResponse.getBody().books.get(0);](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[}](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[@When("^I add a book to my reading list$")](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[public void addBookInList() {](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[ISBN isbn = new ISBN(book.isbn);](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[AddBooksRequest addBooksRequest = new AddBooksRequest(USER\_ID, isbn);](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[userAccountResponse = getEndPoints().addBook(addBooksRequest);](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[}](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[@Then("^The book is added$")](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[public void bookIsAdded() {](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[Assert.assertTrue(userAccountResponse.isSuccessful());](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[Assert.assertEquals(201, userAccountResponse.getStatusCode());](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[Assert.assertEquals(USER\_ID, userAccountResponse.getBody().userID);](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[Assert.assertEquals(book.isbn, userAccountResponse.getBody().books.get(0).isbn);](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[}](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[@When("^I remove a book from my reading list$")](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[public void removeBookFromList() {](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[RemoveBookRequest removeBookRequest = new RemoveBookRequest(USER\_ID, book.isbn);](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[response = getEndPoints().removeBook(removeBookRequest);](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[}](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[@Then("^The book is removed$")](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[public void bookIsRemoved() {](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[Assert.assertEquals(204, response.getStatusCode());](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[userAccountResponse = getEndPoints().getUserAccount(USER\_ID);](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[Assert.assertEquals(200, userAccountResponse.getStatusCode());](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[Assert.assertEquals(0, userAccountResponse.getBody().books.size());](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[}](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[}](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[Lastly, the](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)**[BaseSteps.java](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)**[class updates as:](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[package stepDefinitions;](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[import apiEngine.EndPoints;](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[import cucumber.TestContext;](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[public class BaseStep {](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[private static final String BASE\_URL = "https://bookstore.toolsqa.com";](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[private EndPoints endPoints;](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[public BaseStep(TestContext testContext) {](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[endPoints = testContext.getEndPoints();](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[}](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[public EndPoints getEndPoints() {](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[return endPoints;](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

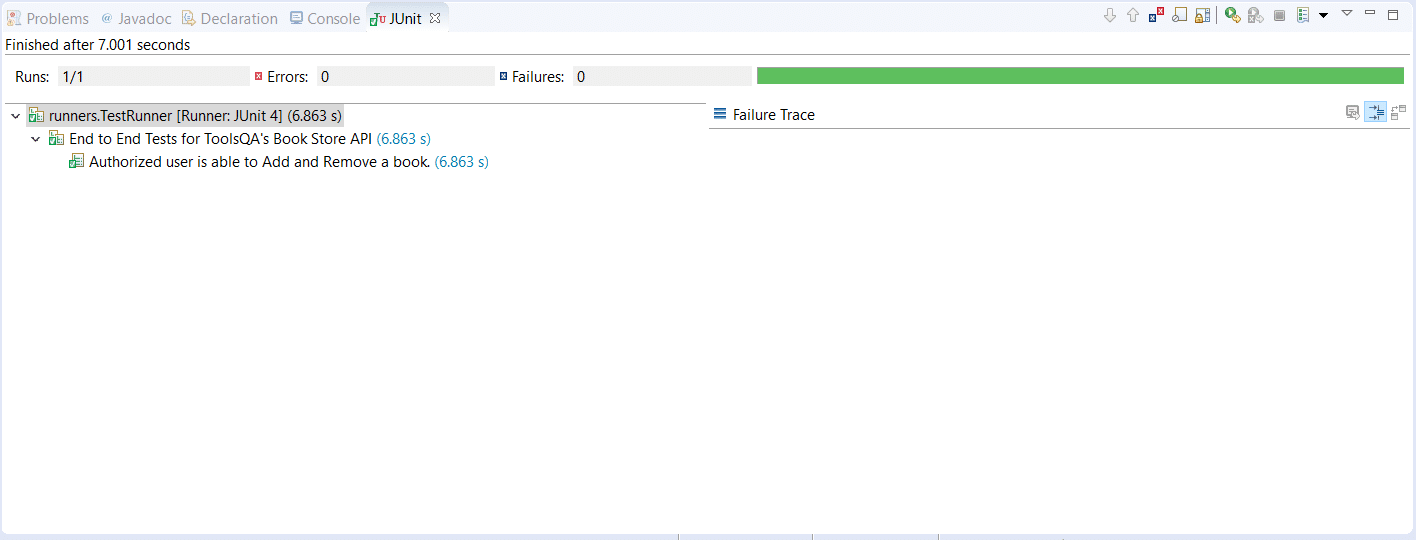
[}](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[}](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

***[Run the Cucumber Test](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)***

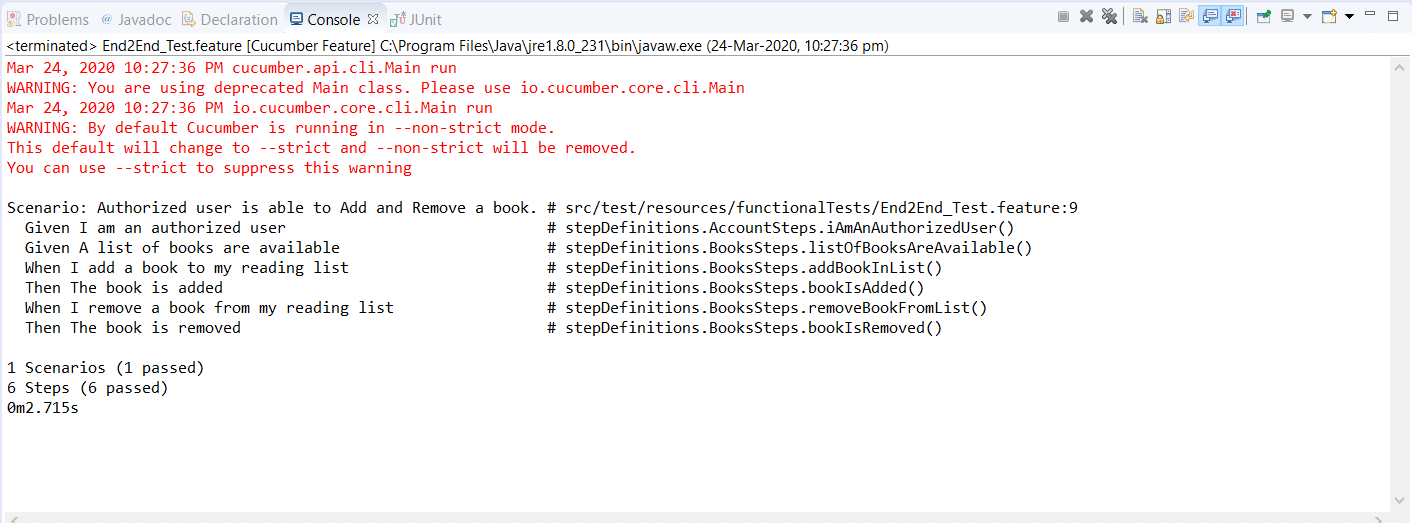
***[Run the Tests as JUnit](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)***

[We are all set now to run the updated Cucumber test. First,](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)*[Right -Click](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)*[on](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)***[TestRunner](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)***[class and Click](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)***[Run As  >> JUnit Test.](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)***[Cucumber runs the script in the same way as](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)*[Selenium WebDriver.](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)*[Consequently, the result will display in the](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)*[JUnit](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)*[tab of the console.](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

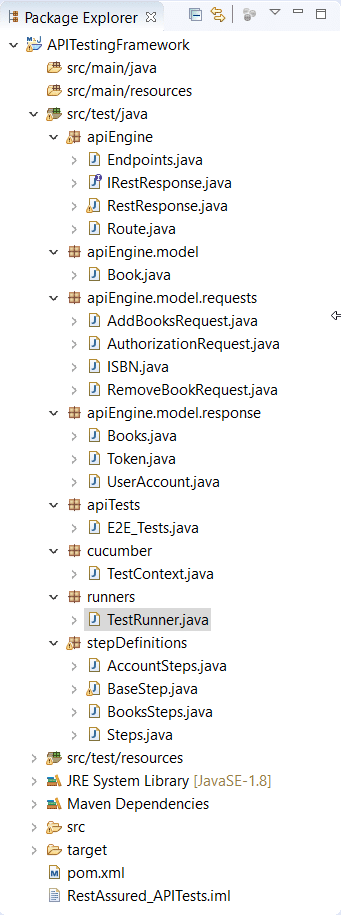
***[Run the Tests from Cucumber Feature](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)***

[To run the tests as a Cucumber Feature, right-click on the](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)*[End2End\_Test.feature](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)*[file. After that, select the](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)*[Run As>>Cucumber Feature.](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)*

[](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[The tests passed successfully with the changes we made for](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)***[sharing the Test Context with Cucumber Step Definitions](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)***[in our framework. Please try to implement the above changes in your framework, as explained above.](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[Our updated project folder structure of the framework will look likewise:](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)

[In the next chapter, we will introduce the concept of usage of](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)[***[Scenario Context in Framework](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)***](https://toolsqa.com/selenium-cucumber-framework/sharing-test-context-between-cucumber-step-definitions/)[to share the test data information specifically. Additionally, we will further segregate our steps into the](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)***[Verification Steps from the Step classes](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)***

* + [paration of Test Layer with API Services](https://toolsqa.com/rest-assured/separation-of-test-layer/" \o "Separation of Test Layer with API Services)
  + [Implementation of REST Routes](https://toolsqa.com/rest-assured/rest-routes/)
  + [Implementation of Generics in API Framework](https://toolsqa.com/rest-assured/generics-in-api-framework/)
  + [Refactoring for Request Headers](https://toolsqa.com/rest-assured/refactoring-for-request-headers/)
  + [**Sharing Test Context**](https://toolsqa.com/rest-assured/share-test-context/)
  + [Sharing Scenario Context](https://toolsqa.com/rest-assured/share-scenario-context/)
  + [Implement Configuration Reader](https://toolsqa.com/rest-assured/configuration-reader/)
* **Blogs**

# Share Scenario Context in Cucumber

Category: [**Rest-Assured**](https://toolsqa.com/categories/rest-assured) , September 16 2021

0

0

7 min read

**Author**



Yashraj Gore

**Reviewers**



Lakshay Sharma

**[Previous Lesson](https://toolsqa.com/rest-assured/share-test-context/)**

[Share Test Context in Cucumber](https://toolsqa.com/rest-assured/share-test-context/)

**[Next Lesson](https://toolsqa.com/rest-assured/configuration-reader/)**

[Implement Configuration Reader](https://toolsqa.com/rest-assured/configuration-reader/)

In the previous chapter, we implemented a [***Sharing Test Context between Cucumber Steps***](https://www.toolsqa.com/rest-assured/share-test-context/). Additionally, we made the changes to make the step definition file uncluttered and share the ***Test Context*** with all the ***Step Definition*** files. In continuation of the earlier chapter, we will now proceed to Share Data by using Scenario Context in Cucumber.

***Scenario Context*** class holds the test data information explicitly. It helps you store values in a key-value pair between the steps. Moreover, it helps in organizing step definitions better rather than using private variables in step definition classes. We have implemented this before in our previous Cucumber tutorial on [***sharing data between steps in Cucumber using Scenario Context***](https://www.toolsqa.com/selenium-cucumber-framework/share-data-between-steps-in-cucumber-using-scenario-context/).

As a next step, we are going to implement the Scenario Context in cucumber for sharing data between our Step Definitions in our API Rest Assured Automation Framework.

## Share Scenario Context in Cucumber

In the end-to-end tests, you're interested in verifying the response for the requests sent. Let’s say you wish to check the subsequent actions within the tests:

* Addition of Book
* Removal of Book

In that case, we want to store the book details in Scenario Context object while adding the book to the user account. Likewise, in our next verification step, we remove the book from the user account. Moreover, we retrieve from the ScenarioContext,  book details of the added book, and validate its removal from the user account.

We will follow the identical steps as [***sharing Scenario Context with Cucumber Step Definitions***](https://www.toolsqa.com/selenium-cucumber-framework/share-data-between-steps-in-cucumber-using-scenario-context/) to share the test data information state across steps:

1. Firstly, create a Scenario Context class.
2. Secondly, save the test information state within the Scenario Context.
3. Thirdly, add a Verification Step file to Test for Responses.
4. Fourthly, run the Cucumber Test.

### *****Step 1: Create a Scenario Context class*****

It is a clean code practice to create enums for all the fixed set of values in the project.

1. Firstly, Right-click on the ***src/test/java*** and select ***New >> Package***. Moreover, name it as ***enums***. We will keep all the project enums in this package.
2. Secondly, Right-click on the ***enums package*** and select ***New >> Enum***. Name it as ***Context***. Moreover, we will add our enum in it.

***Context.java***

package enums;

public enum Context {

BOOK,

USER\_ID,

USER\_ACCOUNT\_RESPONSE,

BOOK\_REMOVE\_RESPONSE;

}

1. After that, Right-click on the ***cucumber package*** and select ***New >> Class***. Name it as ***ScenarioContext***.

package cucumber;

import enums.Context;

import java.util.HashMap;

import java.util.Map;

public class ScenarioContext {

private Map<String, Object> scenarioContext;

public ScenarioContext(){

scenarioContext = new HashMap<String, Object>();

}

public void setContext(Context key, Object value) {

scenarioContext.put(key.toString(), value);

}

public Object getContext(Context key){

return scenarioContext.get(key.toString());

}

public Boolean isContains(Context key){

return scenarioContext.containsKey(key.toString());

}

}

***Code Explanation:***

* ***scenarioContext***: It's a ***HashMap*** object storing the information in the ***Key-Value*** pair. The key type is ***String***. Moreover, theValue can be of any ***Object*** Type.
* ***setContext()***: This method takes two parameters,  key as Context and value as an object. Key is nothing but a ***Context*** enum.
* ***getContext()***: This method takes a key as a parameter to returns the object matching the key.
* ***isContains()***: This method checks in the complete Map if the respective key exists or not.

1. Fourthly, include ***ScenarioContext*** in ***TextContext***.  It will ensure that ***ScenarioContext*** is shared across all the Cucumber Steps using the Pico-Container library. We added in our previous chapter.  Moreover, add a ***getter*** method as ***getScenarioContext()*** to get the scenarioContext object.

***TestContext.java***

package cucumber;

import apiEngine.EndPoints;

import enums.Context;

public class TestContext {

private final String BASE\_URL = "https://bookstore.toolsqa.com";

private final String USER\_ID = "9b5f49ab-eea9-45f4-9d66-bcf56a531b85";

private EndPoints endPoints;

private ScenarioContext scenarioContext;

public TestContext() {

endPoints = new EndPoints(BASE\_URL);

scenarioContext = new ScenarioContext();

scenarioContext.setContext(Context.USER\_ID, USER\_ID);

}

public EndPoints getEndPoints() {

return endPoints;

}

public ScenarioContext getScenarioContext() {

return scenarioContext;

}

}

Also, the ***BaseSteps.java*** class updates as:

package stepDefinitions;

import apiEngine.EndPoints;

import cucumber.ScenarioContext;

import cucumber.TestContext;

public class BaseStep {

private EndPoints endPoints;

private ScenarioContext scenarioContext;

public BaseStep(TestContext testContext) {

endPoints = testContext.getEndPoints();

scenarioContext = testContext.getScenarioContext();

}

public EndPoints getEndPoints() {

return endPoints;

}

public ScenarioContext getScenarioContext() {

return scenarioContext;

}

}

### *****Step 2: Save the test information state in the Scenario Context*****

In this step, we save the responses of our requests to ScenarioContext object. To begin with, we will add the book to the reading list of the user and save our response. Subsequently, the responses will then validate in our assertions as a part of the verification steps.

In the below steps of the ***BooksSteps.java*** class, we are saving responses to scenario context objects.

***BooksSteps.java***

package stepDefinitions;

import apiEngine.IRestResponse;

import apiEngine.model.Book;

import apiEngine.model.requests.AddBooksRequest;

import apiEngine.model.requests.ISBN;

import apiEngine.model.requests.RemoveBookRequest;

import apiEngine.model.responses.Books;

import apiEngine.model.responses.UserAccount;

import cucumber.TestContext;

import cucumber.api.java.en.Given;

import cucumber.api.java.en.When;

import enums.Context;

import io.restassured.response.Response;

public class BooksSteps extends BaseStep {

public BooksSteps(TestContext testContext) {

super(testContext);

}

@Given("^A list of books are available$")

public void listOfBooksAreAvailable() {

IRestResponse<Books> booksResponse = getEndpoints().getBooks();

Book book = booksResponse.getBody().books.get(0);

getScenarioContext().setContext(Context.BOOK, book);

}

@When("^I add a book to my reading list$")

public void addBookInList() {

Book book = (Book) getScenarioContext().getContext(Context.BOOK);

String userId = (String) getScenarioContext().getContext(Context.USER\_ID);

ISBN isbn = new ISBN(book.isbn);

AddBooksRequest addBooksRequest = new AddBooksRequest(userId, isbn);

IRestResponse<UserAccount> userAccountResponse = getEndpoints().addBook(addBooksRequest);

getScenarioContext().setContext(Context.USER\_ACCOUNT\_RESPONSE, userAccountResponse);

}

@When("^I remove a book from my reading list$")

public void removeBookFromList() {

Book book = (Book) getScenarioContext().getContext(Context.BOOK);

String userId = (String) getScenarioContext().getContext(Context.USER\_ID);

RemoveBookRequest removeBookRequest = new RemoveBookRequest(userId, book.isbn);

Response response = getEndpoints().removeBook(removeBookRequest);

getScenarioContext().setContext(Context.BOOK\_REMOVE\_RESPONSE, response);

}

}

### *****Step 3: Add a Verification Step file to Test for Responses*****

We will verify the responses saved in our ScenarioContext Objects as a part of this step. We will move all our verification steps to a new class created as ***VerificationSteps.java***.

Create a ***New Class*** and name it as ***VerificationSteps***, by right-clicking on the ***stepDefinitions***. After that, select ***New >> Class***.

***VerificationSteps.java***

package stepDefinitions;

import apiEngine.IRestResponse;

import apiEngine.model.Book;

import apiEngine.model.responses.UserAccount;

import cucumber.TestContext;

import cucumber.api.java.en.Then;

import enums.Context;

import io.restassured.response.Response;

import org.junit.Assert;

public class VerificationSteps extends BaseStep {

public VerificationSteps(TestContext testContext) {

super(testContext);

}

@Then("^The book is added$")

public void bookIsAdded() {

Book book = (Book) getScenarioContext().getContext(Context.BOOK);

String userId = (String) getScenarioContext().getContext(Context.USER\_ID);

IRestResponse<UserAccount> userAccountResponse = (IRestResponse<UserAccount>) getScenarioContext().getContext(Context.USER\_ACCOUNT\_RESPONSE);

Assert.assertTrue(userAccountResponse.isSuccessful());

Assert.assertEquals(201, userAccountResponse.getStatusCode());

Assert.assertEquals(userId, userAccountResponse.getBody().userID);

Assert.assertEquals(book.isbn, userAccountResponse.getBody().books.get(0).isbn);

}

@Then("^The book is removed$")

public void bookIsRemoved() {

String userId = (String) getScenarioContext().getContext(Context.USER\_ID);

Response response = (Response) getScenarioContext().getContext(Context.BOOK\_REMOVE\_RESPONSE);

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

IRestResponse<UserAccount> userAccountResponse = getEndPoints().getUserAccount(userId);

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

Assert.assertEquals(0, userAccountResponse.getBody().books.size());

}

}

***Explanation***

* We just need to pass the ***Key*** to the ***getContext()*** method, which we can access like ***scenarioContext.getContext(Key Name)***. Additionally, it will retrieve the value returned by the response for Book.
* Since the method returns an object, we need to cast it to the right type. If it's cast it to a wrong type, the test will fail here. So, we must be sure of the object type we stored for the respective key.

### *****Step 4: Run the Cucumber Test*****

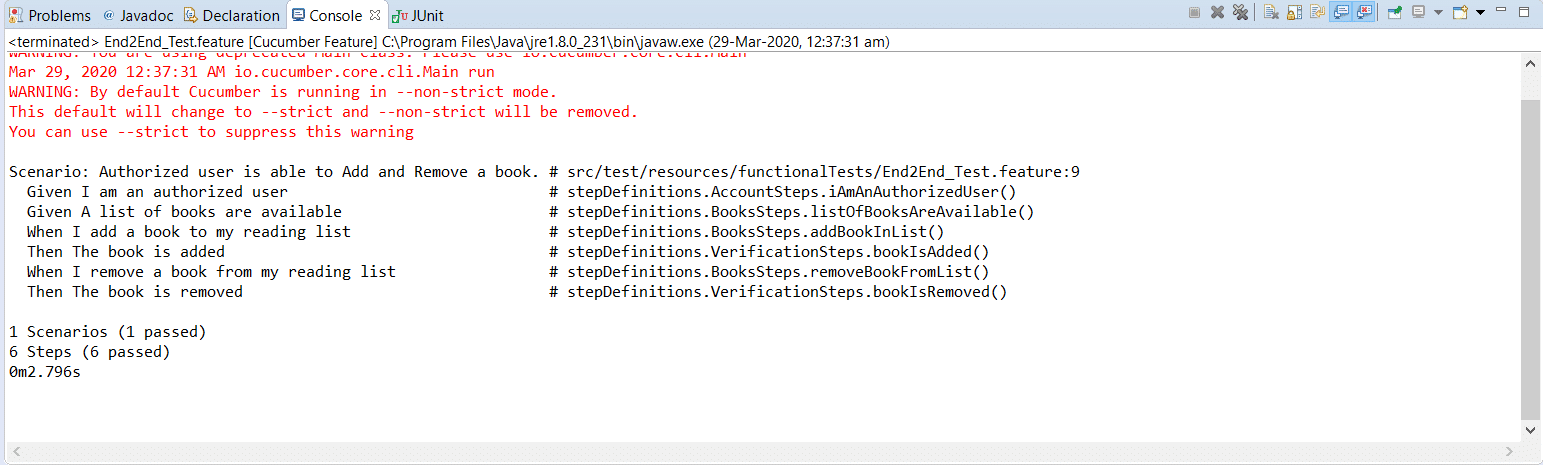
***Run the Tests as JUnit***

We are all set now to run the updated Cucumber test. Firstly, Right -Click on ***TestRunner*** class and Click ***Run As  >> JUnit Test***. Cucumber runs the script in the same way as Selenium WebDriver.  Consequently, the result will display in the JUnit tab of the console.



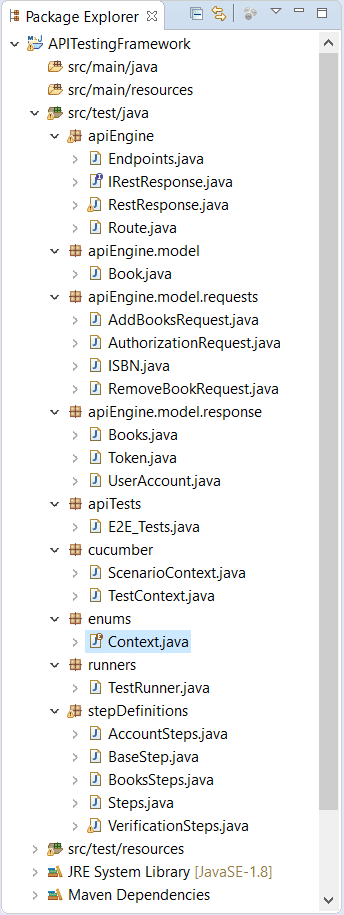
***Run the Tests from Cucumber Feature***

To run the tests as a Cucumber Feature, right-click on the End2End\_Test.feature file. Select the Run As>>Cucumber Feature.



The tests passed successfully with the changes we made for ***sharing test data in Cucumber Steps using Scenario Context*** in our framework.

Our updated project folder structure of the framework will look likewise:



In the next chapter, we will add [***Config Reader class***](https://www.toolsqa.com/rest-assured/configuration-reader/) for BaseUrl and UserId. Please try to implement the above changes in your framework, as explained above.

**Share this post:**

**Facebook**

# Implement Configuration Reader

Category: [**Rest-Assured**](https://toolsqa.com/categories/rest-assured) , November 10 2021

0

0

7 min read

**Author**



Yashraj Gore

**Reviewers**



Lakshay Sharma

**[Previous Lesson](https://toolsqa.com/rest-assured/share-scenario-context/)**

[Share Scenario Context in Cucumber](https://toolsqa.com/rest-assured/share-scenario-context/)

**[Next Lesson](https://toolsqa.com/rest-assured/rest-assured-examples/)**

[Rest Assured Examples](https://toolsqa.com/rest-assured/rest-assured-examples/)

In our previous chapter, we accomplished the [***sharing of test data in Cucumber Steps using Scenario Context***](https://toolsqa.com/selenium-cucumber-framework/share-data-between-steps-in-cucumber-using-scenario-context/). Subsequently, in this cImplement Configuration Readerhapter, we shall read the configuration values from a property file and use them in our code with the help of ***Configuration Reader***.

If you may have noticed, so far, we are passing the ***user\_Id***and the ***base\_Url*** values as hardcoded ones. However, the problem with hardcoded values is that they are non-maintainable. In other words, the changes in configuration value amount to making changes in several places in the code. Therefore, it is not a clean code practice.

As a solution to this, we will go for ***property file*** implementation in Java. If you wish to understand this on a conceptual level, you could refer to the [***Read Configurations from Property File***](https://www.toolsqa.com/selenium-cucumber-framework/read-configurations-from-property-file/). Let us learn how to implement the configurations from the property file.

## Implement Configuration Reader to Read Project Configurations

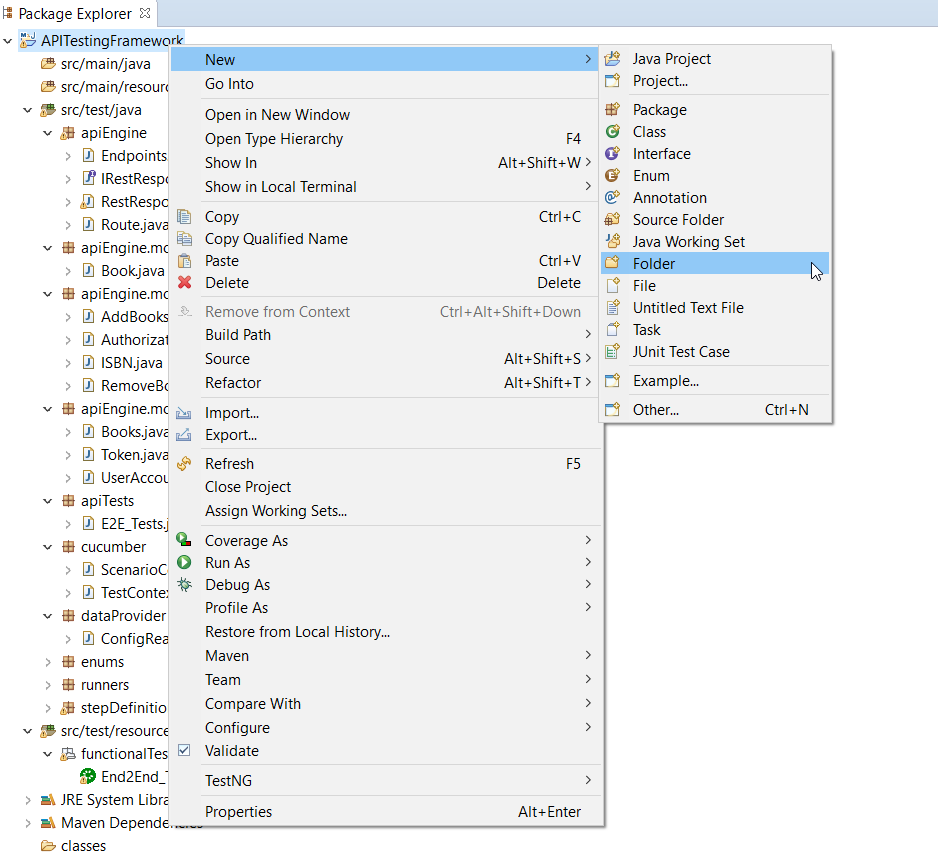
We will follow the below steps and implement Read Configurations:

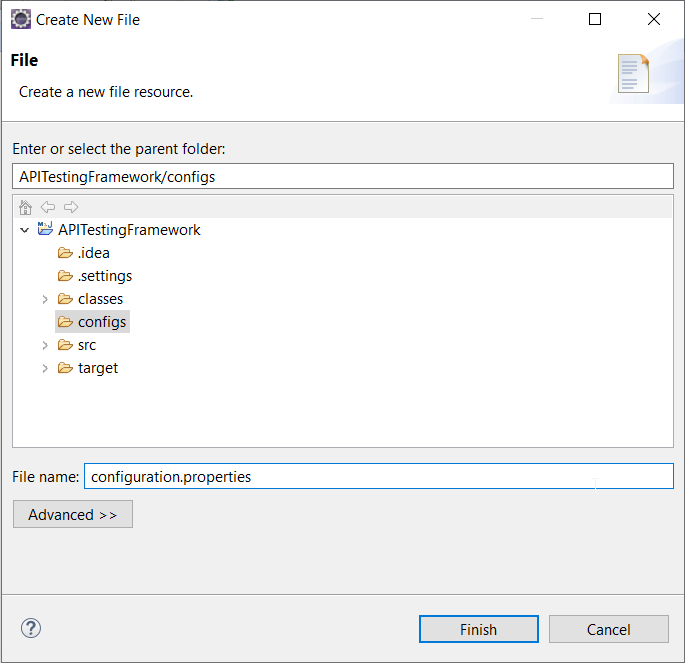
1. How to Read Configurations from Property File
2. Write Hard-Coded Values in the Property File
3. Create a Configuration Reader File
4. Use ConfigFileReader object in the TestContext file
5. Run the Cucumber Test

### *****How to Read Configurations from Property File*****

Firstly, right-click on the root Project and select ***New >> Folder***.  Additionally, name it as ***configs***. Moreover, we will keep config files with in the same folder.

Secondly, right-click the above-created folder and select ***New >> File***. Additionally, name it as ***configuration.properties***. Here, the ***.properties*** is the file extension.





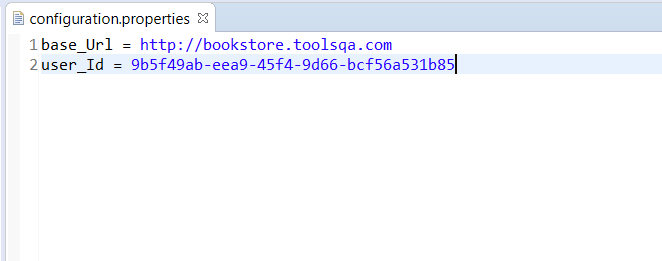
### *****Step 2: Write Hard-Coded Values to Property File*****

If we look at our ***TestContext.java*** class,  we have been using two hardcoded values:

base\_Url = http://bookstore.toolsqa.com

user\_Id = 9b5f49ab-eea9-45f4-9d66-bcf56a531b85

In the ***configuration.properties*** file, we will move these values in key-pair format.



### *****Step 3: Create a Config File Reader*****

Firstly, right-click on the ***src/test/java*** and select ***New >> Package***.  In addition to that, name it as ***dataProvider***.  Moreover, all the data-readers files will be kept here in this package.

Secondly, right-click on the above created package and select ***New >> Class***. After that, name it as ***ConfigFileReader***.

***ConfigReader.java***

package configs;

import java.io.BufferedReader;

import java.io.FileNotFoundException;

import java.io.FileReader;

import java.io.IOException;

import java.util.Properties;

public class ConfigReader {

private Properties properties;

private static ConfigReader configReader;

private ConfigReader() {

BufferedReader reader;

String propertyFilePath = "configs//Configuration.properties";

try {

reader = new BufferedReader(new FileReader(propertyFilePath));

properties = new Properties();

try {

properties.load(reader);

reader.close();

} catch (IOException e) {

e.printStackTrace();

}

} catch (FileNotFoundException e) {

e.printStackTrace();

throw new RuntimeException("Configuration.properties not found at " + propertyFilePath);

}

}

public static ConfigReader getInstance( ) {

if(configReader == null) {

configReader = new ConfigReader();

}

return configReader;

}

public String getBaseUrl() {

String baseUrl = properties.getProperty("base\_Url");

if(baseUrl != null) return baseUrl;

else throw new RuntimeException("base\_Url not specified in the Configuration.properties file.");

}

public String getUserID() {

String userId = properties.getProperty("user\_Id");

if(userId != null) return userId;

else throw new RuntimeException("user\_Id not specified in the Configuration.properties file.");

}

}

***Code Explanation:***

***How to Load Property File***

BufferedReader reader = new BufferedReader(new FileReader(propertyFilePath));

Properties properties = new Properties();

properties.load(reader);

* ***propertyFilePath***: It would be a String variable containing the information of the config file path.
* ***new FileReader***(propertyFilePath): Creates a new FileReader containing file-name, to be read from.
* ***new BufferedReader***(new FileReader(propertyFilePath)): Reads the text from a character-input stream. Moreover, it ensures efficient reading of the characters, arrays, and lines by buffering the characters.
* ***new Properties():*** The Properties class represents a persistent set of properties. which can be saved to a stream or loaded from it. Additionally, the key and the corresponding value in the property list is a string.
* ***properties.load***(reader): It reads a property list of key and element pairs from the input character stream in a simple line-oriented format.

If you have noticed the class ***ConfigReader.java***, we have implemented it as a singleton class. The Singleton class’s purpose is to control object creation, limiting the number of objects to only one. Additionally, there is only one Singleton instance. Therefore, any instance fields of a Singleton would occur only once per class, similar to static fields.

#### **Why do we need Singleton class implementation?**

Singleton pattern ensures the creation of only one instance of a class in the JVM. Moreover, it enables a global point of access to the object. In our case, we have ***ConfigReader.java***, which should be accessed globally. So it is better to make the ***ConfigReader.java*** class as a ***singleton***.

***How to implement a Singleton Pattern?***

The common concepts among several approaches to implement a Singleton pattern are:

* Private constructor
* Static field containing only it's an instance
* A public static method returning an instance of the class

public class ConfigReader {

// The Static member holds a single instance of the

// ConfigReader class

private static ConfigReader configReader;

// ConfigReader prevents any other class from instantiating

private ConfigReader() {

}

// Provides Global point of access

public static ConfigReader getInstance() {

if (configReader == null) {

configReader = new ConfigReader();

}

return configReader;

}

}

The ***ConfigReader.java*** class maintains a static reference to its instance. Additionally, it returns that reference from the ***static getInstance()*** method.

***ConfigReader.java*** implements a ***private constructor*** so clients cannot instantiate ***ConfigReader*** instances.

***ConfigFileReader Method***

public String getBaseUrl() {

String baseUrl = properties.getProperty("base\_Url");

if(baseUrl != null) return baseUrl;

else throw new RuntimeException("base\_Url not specified in the configuration.properties file.");

}

***properties.getBaseUrl()***: Properties object gives us a .getProperty method. Additionally, the input is Key of the property sent as a parameter while the Value of the matched key is the output from the .properties file.

Moreover, if the properties file doesn't have the specified key, it returns the null. Hence, we have to put the null check and, in case of null, throw an exception to stop the test with stack trace information.

For more details, you can visit the article on [***Singleton Pattern in Java***](https://www.journaldev.com/1377/java-singleton-design-pattern-best-practices-examples).

### *****Step 4: Use ConfigFileReader object in the TestContext file*****

After making the necessary changes for the ***BASE\_URL*** and ***USER\_ID***  in***TextContext.java class***:

private final String BASE\_URL = "https://bookstore.toolsqa.com";

private EndPoints endPoints;

public TestContext() {

endPoints = new EndPoints(BASE\_URL);

scenarioContext = new ScenarioContext();

scenarioContext.setContext(Context.USER\_ID, USER\_ID);

}

Consequently, the ***TextContext.java*** class transforms to:

package cucumber;

import apiEngine.EndPoints;

import configs.ConfigReader;

import enums.Context;

public class TestContext {

private EndPoints endPoints = new EndPoints(ConfigReader.getInstance().getBaseUrl());

private ScenarioContext scenarioContext;

public TestContext() {

scenarioContext = new ScenarioContext();

scenarioContext.setContext(Context.USER\_ID, ConfigReader.getInstance().getUserID());

}

public EndPoints getEndPoints() {

return endPoints;

}

public ScenarioContext getScenarioContext() {

return scenarioContext;

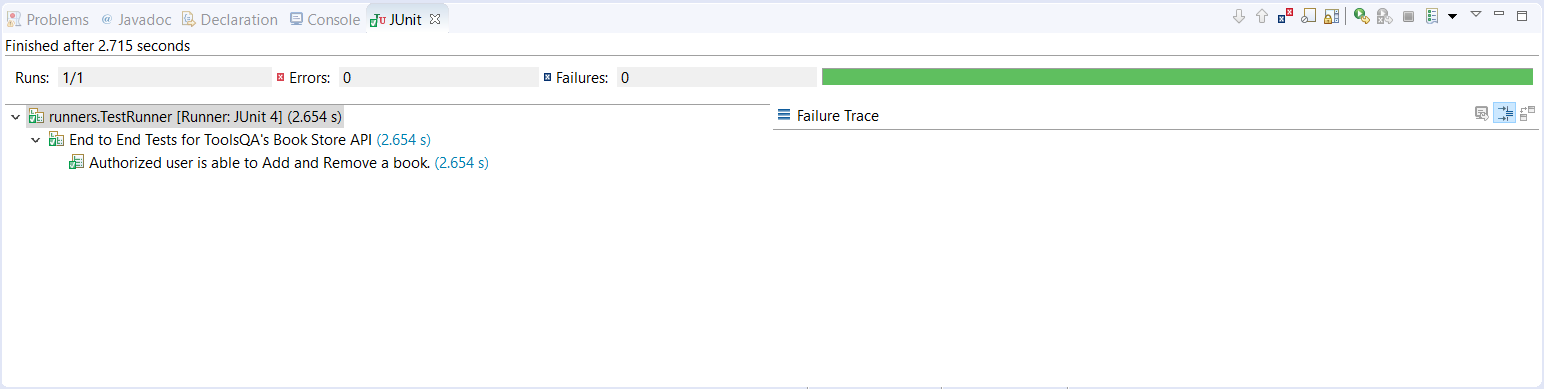
}

}

### *****Step 5: Run the Cucumber Test*****

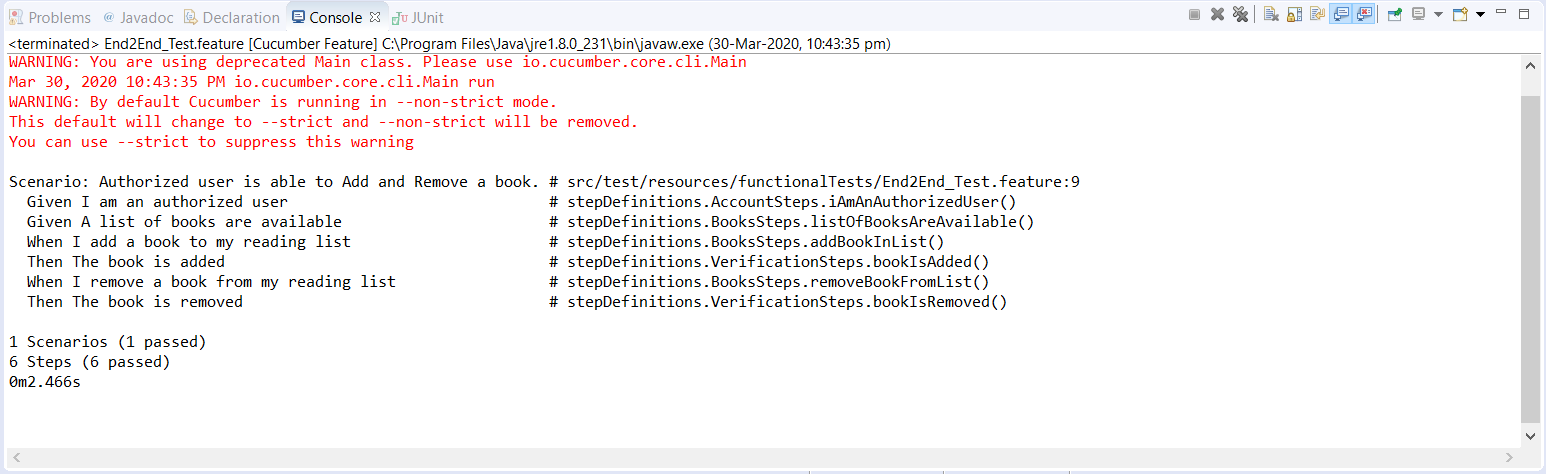
***Run the Tests as JUnit***

We are all set now to run the updated Cucumber test. Firstly, Right -Click on \*\*\*TestRunner \*class and Click ***Run As  >> JUnit Test***. Cucumber runs the script in the same way as Selenium WebDriver.  Finally, the result will display in the JUnit tab of the console.

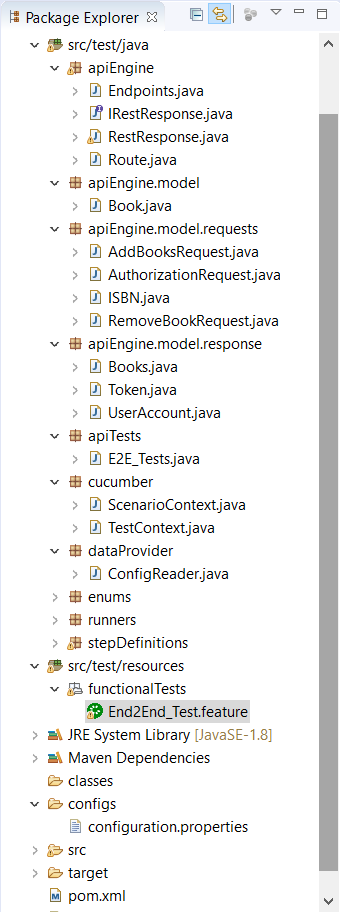


***Run the Tests from Cucumber Feature***

To run the tests as a Cucumber Feature, right-click on the End2End\_Test.feature file. Select the Run As>>Cucumber Feature.



Moreover, please try to implement the above changes in your framework, as explained above. Subsequently, our updated project folder structure of the framework will look likewise:



The tests passed successfully with the changes we made to the ***Configuration Reader*** in our framework.