

REST ASSURED API AUTOMATION TESTING

BY

MARIMUTHU C

CONTENTS

1: Introduction

1.1: What is API?

1.2: What is API Testing?

1.3: REST (Representational State Transfer)

2: The Architecture of REST Service

2.1: What is Client Server Architecture?

2.2: Client server Architecture

2.3: What is a client?

2.4: What is a server?

2.5: How does the client server architecture work?

2.6: What are the different types of client server architectures?

2.7: 1-tier architecture

2.8: 2-tier architecture

2.9: 3-tier architecture

2.10: N-tier architecture

2.11: Advantages of using client server architecture?

2.12: What are the drawbacks of client server architecture?

3: Parameters

3.1: Header parameter

3.2: Path parameters

3.3: Query parameters

3.4: Request body parameters

4: HTTP Methods for REST API Automation Testing

4.1: RESTful Service

4.2: Methods or Verbs

4.3: How to Make a POST Request with RestAssured?

4.4: Gherkin Style Rest Assured POST Request

4.5: Gherkin Style Rest Assured POST Request

4.6: What are different HTTP Response Status Codes?

5: Validate Response Status, Response Header & Response Body using Rest Assured

5.1: Request

5.2: Response

6. Steps to use POST Request Method using Rest Assured?

7. Serialization and Deserialization in Java

7.1: What is Serialization?

7.2: Serializable Interface

7.3: Serializing an Object in Java

7.4: Deserializing to an Object in Java

8: Deserialize JSON Response using Rest Assured

8.1: What is Serializing a JSON?

8.2: What is Deserializing of a JSON Response?

8.3: How to Deserialize JSON Response to Class with Rest Assured?

8.4: How to Deserialize JSON Response Body based on Response Status?

9. Authentication and Authorization in REST Web Services

9.1: What is Authentication? And how does Authorization work in REST Web Services?

9.2: What is Authorization? And how does Authorization work in REST WebServices?

10. How to make a PUT Request using Rest Assured?

10.1: Appropriate status codes obtained for PUT and POST requests

10.2: PUT Request code step by step

11: DELETE Request using Rest Assured

11.1: What is a delete request method?

11.2: What are the different response codes for Delete Request?

12: Prerequisites for REST API End to End Test

12.1: Java Setup

12.2: IDE Setup

12.3: Maven Setup

12.4: Add Rest Assured Dependencies

12.5: Create an Authorised user for the test

12.6: Run the REST API Test

13: REST API Test in Cucumber BDD Style Test

13.1: Write a test in a Feature File

13.2: Create a Test Runner

13.3: Write test code to Step file

13.4: Run test as JUnit test & Maven Command Line

13.5: Execution Reports

INTRODUCTION

What is API?

API stands for **Application Programming Interface**. It comprises a set of functions that can be accessed and executed by another software system. Thus, it serves as an interface between different software systems and establishes their interaction and data exchange.

What is API Testing?

API testing is a type of software testing that involves testing application programming interfaces (APIs) directly and as part of integration testing to determine if they meet expectations for functionality, reliability, performance, and security. In the modern development world, many web applications are designed based on a three-tier architecture model. These are,

1) Presentation Tier – User Interface (UI)

2) Logic Tier – Business logic is written in this tier. It is also called Business Tier. (API)

3) Data Tier – Here information and data are stored and retrieved from a Database. (DB)

Ideally, these three layers (tiers) should not know anything about the platform, technology, and structure of each other.

We can test UI with GUI testing tools and we can test logic tier (API) with API testing tools. The logic tier comprises all of the business logic and it has more complexity than the other tiers and the test executed on this tier is called API Testing. API testing tests the logic tier directly and checks expected functionality, reliability, performance, and security.

In the agile development world, requirements are changing during short release cycles frequently and GUI tests are more difficult to maintain according to those changes. Thus, API testing becomes critical to test application logic. In GUI testing we send inputs via keyboard texts, button clicks, drop-down boxes, etc., on the other hand in API testing we send requests (method calls) to the API and get output (responses). These APIs are generally REST APIs or SOAP web services with JSON or XML message payloads being sent over HTTP, HTTPS, JMS, and MQ.

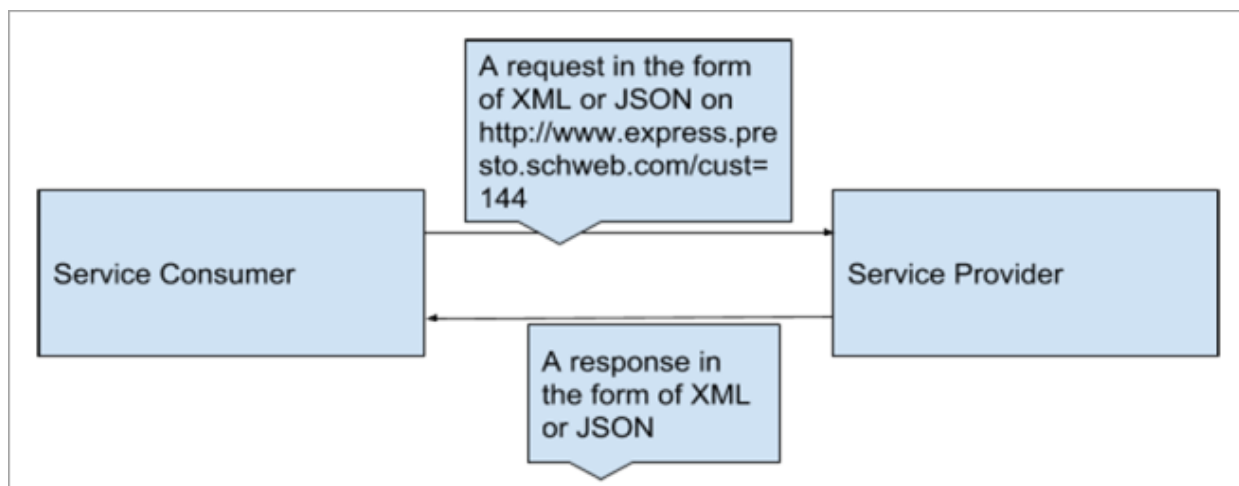
REST (Representational State Transfer):

REST is an architectural style that uses simple HTTP calls for inter-machine communication. REST does not contain an additional messaging layer and focuses on design rules for creating stateless services. A client can access the resource using the unique URI and a representation of the resource is returned. With each new resource representation, the client is said to transfer state. While accessing RESTful resources with HTTP protocol, the URL of the resource serves as the resource identifier, and GET, PUT, DELETE, POST and HEAD are the standard HTTP operations to be performed on that resource.

The Architecture of REST Service:

The architecture of REST service depends on two entities i.e. **Service Consumer or Requestor and Service Provider**. The Service Consumer is the one who avails the Web Service and Service Provider is the collection of software or system which provides the Web Service.

The client application which is typically a Service Consumer uses inbuilt methods of REST, a URL or URI, an HTTP version and a payload (if supported by method).



What is Client Server Architecture?

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

information securely. One of the basic concepts to start with is by understanding the client server architecture. , we will understand below points-

- Client server Architecture
- What is a client?
- What is a server?
- How does the client server architecture work?
- What are the different types of client server architectures?
- 1-tier architecture
- 2-tier architecture
- 3-tier architecture
- N-tier architecture
- Advantages of using client server architecture?
- What are the drawbacks of client server architecture?

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.

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.

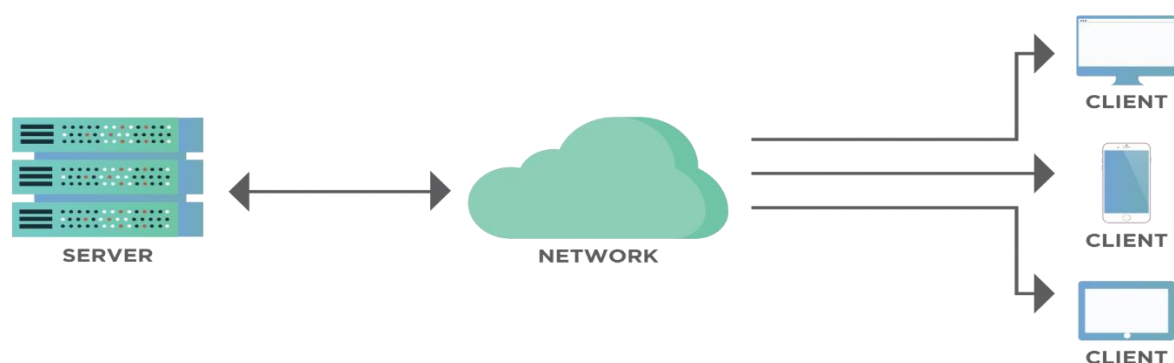
Gaming Server - used for playing multiplayer games.

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. 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.

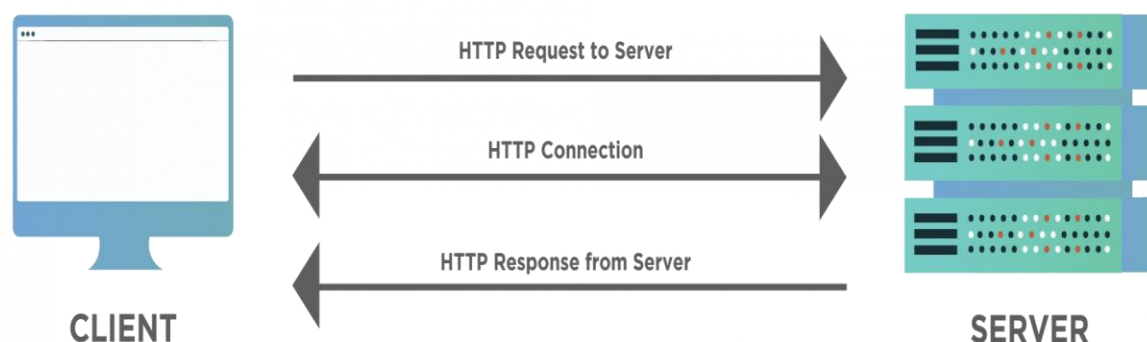
How does the Client Server Architecture work?

As already discussed above, 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-

The HTTP communication protocol helps establish the connection between the client and the server.

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.

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. Let us consider an example to understand more about it.

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-Tier Architecture
- 2-Tier Architecture
- 3-Tier Architecture
- N-Tier Architecture

1-tier architecture

The business logic, data logic, and the user interface all reside on the same machine in 1 tier. 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.

Parameters:

Parameters are the options that control the resource. There are four kinds of parameters:

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.

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.

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.

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.

HTTP Methods for REST API Automation Testing

We know that REST API uses five HTTP methods to request a command:

| Method | Description |
|--------|--|
| GET | Retrieves the information at a particular URL. |
| PUT | Updates the previous resource if it exists or creates new information at a particular URL. |
| POST | Used to send information to the server like uploading data and also to develop a new entity. |
| DELETE | Deletes all current representations at a specific URL. |
| PATCH | This is used for partial updates of resources. |

Once the request is sent using the above methods, the client receives the numeric codes known as "**Status codes**" or sometimes referred to as "**Response codes**". Then we can interpret these status codes to know what kind of response the server has sent for a particular request.

RESTful Service

REST stands for Representational State Transfer which is a Stateless Service.

It is called Stateless as the Web Server does not store any information about the client session (time duration till the client application is connected and in execution) which means each request type is treated and performed easily with the help of REST inbuilt methods like **GET, POST, PUT, DELETE** and so on.

Methods or Verbs

Each method in REST has its significance. Given Below is the briefing about each of them.

GET: This method is used to retrieve the information that is sent to the server using any of the methods like PUT or POST. This does not have a request body. Successful execution will give 200 response objects.

POST: This method is used to create a document or record using a request body, specified URL, document key, context key, etc. The same can be retrieved using the GET method. Successful execution will give 201 response.

PUT: This method is used to update any document or record that is already present. Successful execution will give 201 or 200 response.

DELETE: This method is used to delete any record. Successful execution will give 204 response (no content).

Note: The HTTP response codes depend on how developers code and can be manipulated at times.

How to Make a POST Request with RestAssured?

The following code uses requestSpecBuilder to make a post request.

Parameter descriptions are listed below.

RestAPIURL – URL of the Rest API

APIBody – Body of the Rest API. Example: {"key1":"value1","key2":"value2"}

SetContentType () – Pass the "application/json", "application/xml" or "text/html" etc. headers to setContentType () method.

Authentication credentials – Pass the username and password to the basic () method or if there is no authentication leave them blank basic ("","")

@Test

```
Public void httpPostMethod () throws JSONException, InterruptedException {  
String restAPIURL = "http://{ URL of API}";  
String apiBody = "{\"key1\":\"value1\", \"key2\":\"value2\", \"key3\":\"value3\"}";  
RequestSpecBuilder builder = new RequestSpecBuilder ();  
builder.setBody (apiBody);  
builder.setContentType("application/json; charset=UTF-8");  
RequestSpecification requestSpec = builder.build();  
Response response = given()  
.auth()  
.preemptive()  
.basic({ username }, { password })  
.spec(requestSpec)  
.when()  
.post(restAPIUrl);  
JSONObject JSONResponseBody = new JSONObject(response.body().asString());  
String result = JSONResponseBody.getString({ key });  
Assert.assertEquals(result, "{expectedValue}");  
}
```

Gherkin Style Rest Assured POST Request

@Test

```
public void postExamplebyGherkin(){  
RestAssured.baseURI = "Our API URL";  
Response res = given()  
.contentType("application/json")  
.body("{\"name\":\"Onur Baskirt\"}")  
.when()  
.post("");
```

```
String body = res.getBody().asString();

System.out.println(body);

}
```

What are different HTTP Response Status Codes?

Status codes separate into 5 different categories. Consequently, let's see those in details:

| Code | Description |
|------|--|
| 1xx | Information, i.e., it denotes that the request has been received and under process. |
| 100 | Continue: The client can continue with the request as long as it doesn't get rejected. |
| 101 | Switching Protocols: The server is switching protocols. |
| 2xx | Success, i.e., it denotes a successful receipt, processing, and acceptance of the request message. |
| 200 | OK: The request is OK. |
| 201 | Created: A successfully created new resource |
| 202 | Accepted: Request accepted for processing, but in progress |
| 203 | Non-Authoritative Information: The information in the entity header is not from an original source but a third-party |
| 204 | No Content: Response with status code and header but no response body |
| 205 | Reset Content: The form for the transaction should clear for additional input |
| 206 | Partial Content: Response with partial data as specified in Range header |
| 3xx | Redirection, i.e., further action has to be taken for the request to complete |
| 300 | Multiple Choices: Response with a list for the user to select and go to a location |
| 301 | Moved Permanently: Requested page moved to a new url |
| 302 | Found: Requested page moved to a temporary new URL |
| 303 | See Other: One can find the Requested page under a different URL |
| 305 | Use Proxy: Requested URL need to access through the proxy mentioned in the Location header |
| 307 | Temporary Redirect: Requested page moved to a temporary new URL |

| Code | Description |
|------|--|
| 4xx | Client Error, i.e., incorrect syntax or error in fulfillment of the request |
| 400 | Bad Request: Server unable to understand the request |
| 401 | Unauthorized: Requested content needs authentication credentials |
| 403 | Forbidden: Access is forbidden |
| 404 | Not Found: Server is unable to find the requested page |
| 405 | Method Not Allowed: Method in the request is not allowed |
| 407 | Proxy Authentication Required: Need to authenticate with a proxy server |
| 408 | Request Timeout: The request took a long time as expected by the server |
| 409 | Conflict: Error in completing request due to a conflict |
| 411 | Length Required: We require the "Content-Length" for the request to process |
| 415 | Unsupported Media Type: Unsupported media-type |
| 5xx | Server Error, i.e., error invalid request fulfilment at server side |
| 500 | Internal Server Error: Request not completed due to server error |
| 501 | Not Implemented: Server doesn't support the functionality |
| 502 | Bad Gateway: Invalid response from an upstream server to the server. Hence, the request not complete |
| 503 | Service Unavailable: The server is temporarily down |
| 504 | Gateway Timeout: The gateway has timed out |
| 505 | HTTP Version Not Supported: Unsupported HTTP protocol version |

Validate Response Status, Response Header & Response Body using Rest Assured:

Request:

Request method: POST

Request URI: <https://rahulshettyacademy.com/maps/api/place/add/json?key=qaclick123>

Proxy: <none>

Request params: <none>

Query params: key=qaclick123

Form params: <none>

Path params: <none>

Headers: Accept= */*
 Content-Type=application/json

Cookies: <none>

Multiparts: <none>

Body:

```
{  
  "accuracy": 50,  
  "name": "Mari",  
  "phone_number": "(+91) 983 893 3937",  
  "address": "Injambakam",  
  "website": "https://rahulshettyacademy.com",  
  "language": "English",  
  "location": {  
    "lat": -38.383494,  
    "lng": 33.427362  
  },  
  "types": [  
    "shoe park",  
    "shop"  
  ]  
}
```

Response:

HTTP/1.1 200 OK -----> **Response Code**

Date: Wed, 16 Feb 2022 13:49:24 GMT

Server: Apache/2.4.18 (Ubuntu)

Access-Control-Allow-Origin: *

Access-Control-Allow-Methods: POST

Access-Control-Max-Age: 3600

Access-Control-Allow-Headers: Content-Type,

Access-Control-Allow-Headers, Authorization, X-Requested-With

Response Header

Content-Length: 194

Keep-Alive: timeout=5, max=100

Connection: Keep-Alive

Content-Type: application/json;charset=UTF-8

```
{
  "status": "OK",
  "place_id": "735dee7651551d3a716a406710a0c4c1",

  "scope": "APP",
  "reference": "7671110478b2786f3867cf780e84f43a767
    1110478b2786f3867cf780e84f43a",
  "id": "7671110478b2786f3867cf780e84f43a"
}
```

Response Body

Steps 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>. 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.

- Create a Request pointing to the service Endpoint.
- Create a JSON Request which contains all the fields.
- Add JSON body in the request and send the request.
- Validate the Request.

Serialization and Deserialization in Java

Serialization and Deserialization in Java is an important programming concept. It is applicable to all major programming languages. We will try to understand this concept in the context of Java language.

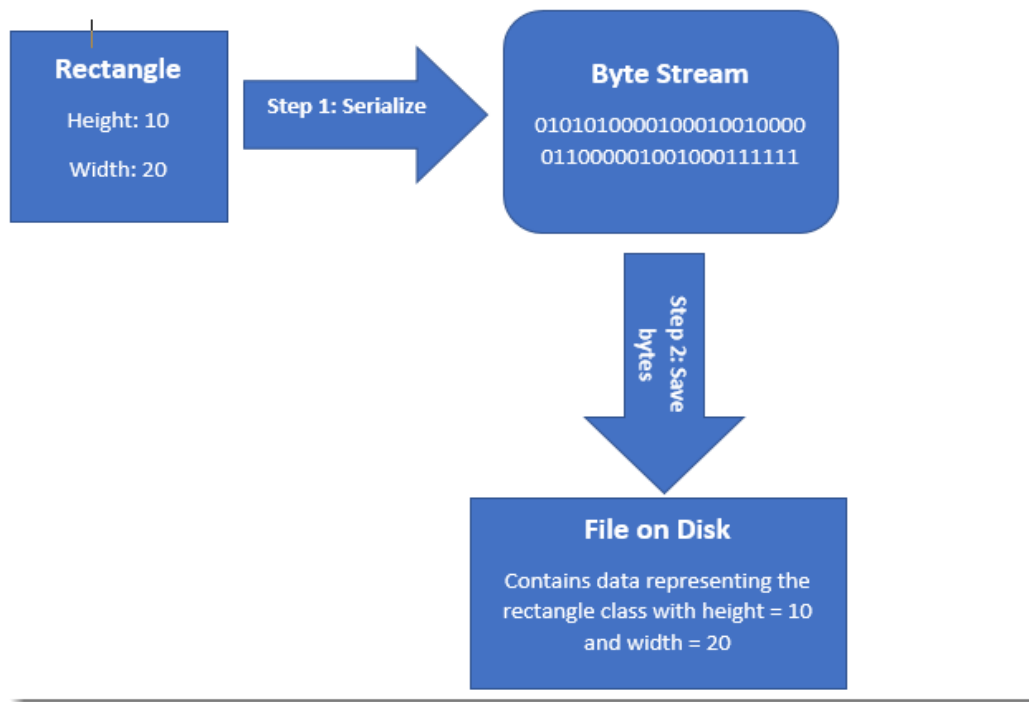
What is Serialization?

Serialization is a process where we 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 state 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.

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);  
    }  
}
```

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

Serializable Interface

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

- `java.io.Serializable`
- `java.io.Externalizable`

Serializable interface is a marker interface. Which means that do not have to implement any methods if our 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 we derive from Externalizable interface we 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 we want to use the default Java's serialization mechanism than we 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

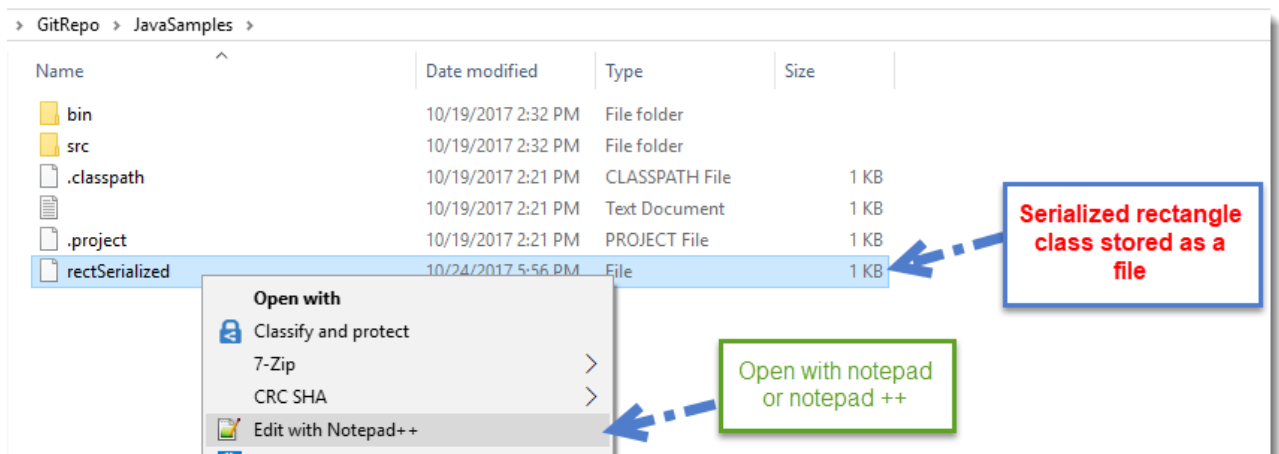
4. Finally, we will close all the stream objects to save properly write and terminate all streams.

Below is the code to perform the task.

```
public static void SerializeToFile(Object classObject, String fileName)
{
    try {
        FileOutputStream fileStream = new FileOutputStream(fileName);
        ObjectOutputStream objectStream = new ObjectOutputStream(fileStream);
        objectStream.writeObject(classObject);
        objectStream.close();
        fileStream.close();
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
}

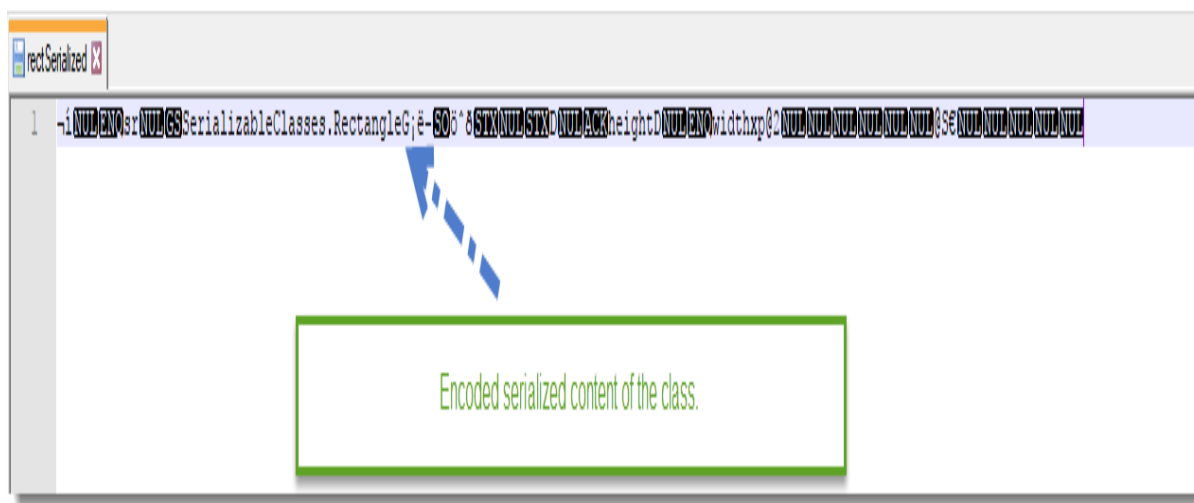
public static void main(String[] args)
{
    Rectangle rect = new Rectangle(18, 78);
    SerializeToFile(rect, "rectSerialized");
}
```

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 we open this file we 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 us a practical understanding of the different steps involved in the serialization process including the input and output results of the different steps.

Deserializing to an Object in Java

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
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.

```
Public static Object DeSerializeFromFileToObject (String fileName)
{
    try {
        FileInputStream fileStream = new FileInputStream(new File(fileName));
        ObjectInputStream objectStream = new ObjectInputStream(fileStream);
        Object deserializeObject = objectStream.readObject();
        objectStream.close();
        fileStream.close();
        return deserializeObject;
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    }
    return null;
}

public static void main(String[] args)
{
    Rectangle rect = new Rectangle(18, 78);
    SerializeToFile(rect, "rectSerialized");
}
```



```

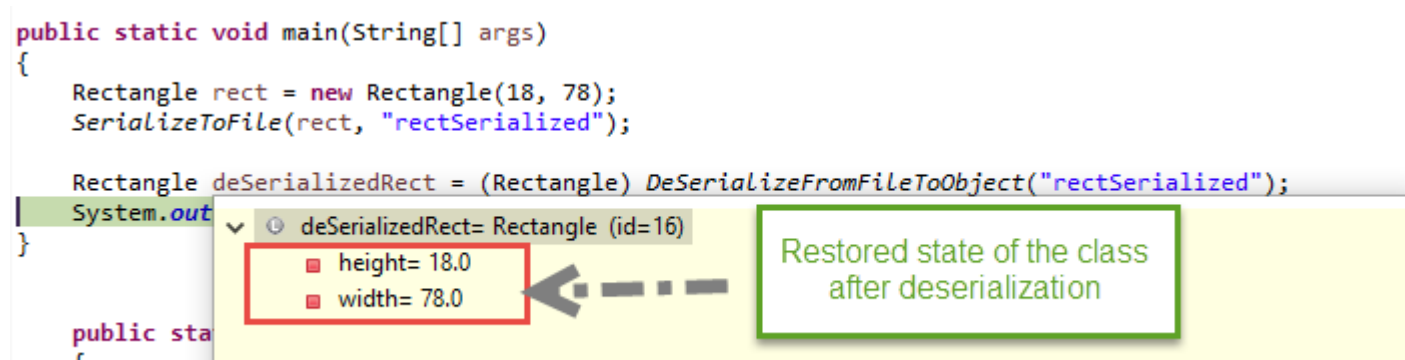
    Rectangle deSerializedRect = (Rectangle)
    DeSerializeFromFileToObject("rectSerialized");

    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.



Deserialize JSON Response using Rest Assured

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 we do not come across a JSON file and verify its legibility. To work with them, we should know how to read them through code and make use of automation testing to accomplish our tasks easily. , 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.

- 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 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 of object.

Java POJO classes are one way we can serialize the objects and use them in Rest Assured.

What is Deserializing a JSON Response?

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, most frequently POJO (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 to implement deserialization of JSON responses. , 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.

```
The status code recieved: 201  
{  
  "SuccessCode": "OPERATION_SUCCESS",  
  "Message": "Operation completed successfully"  
}  
PASSED: UserRegistrationSuccessful
```

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 SuccessCode 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.

```

    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?

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 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.

Authentication and Authorization in REST Web Services

Authentication and Authorization in REST Web Services are two very important concepts in the context of REST API. The majority of the time we will be hitting REST API's which are secured. By secure, we mean that the APIs which require we 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

What is Authentication? And how does Authorization work in REST Web Services?

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

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

In the context of REST API authentication 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 we access the private resources.

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 the Response that we get is.

@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 behaviour is that we will get Authorization error. If we run this test, we will get the following output.

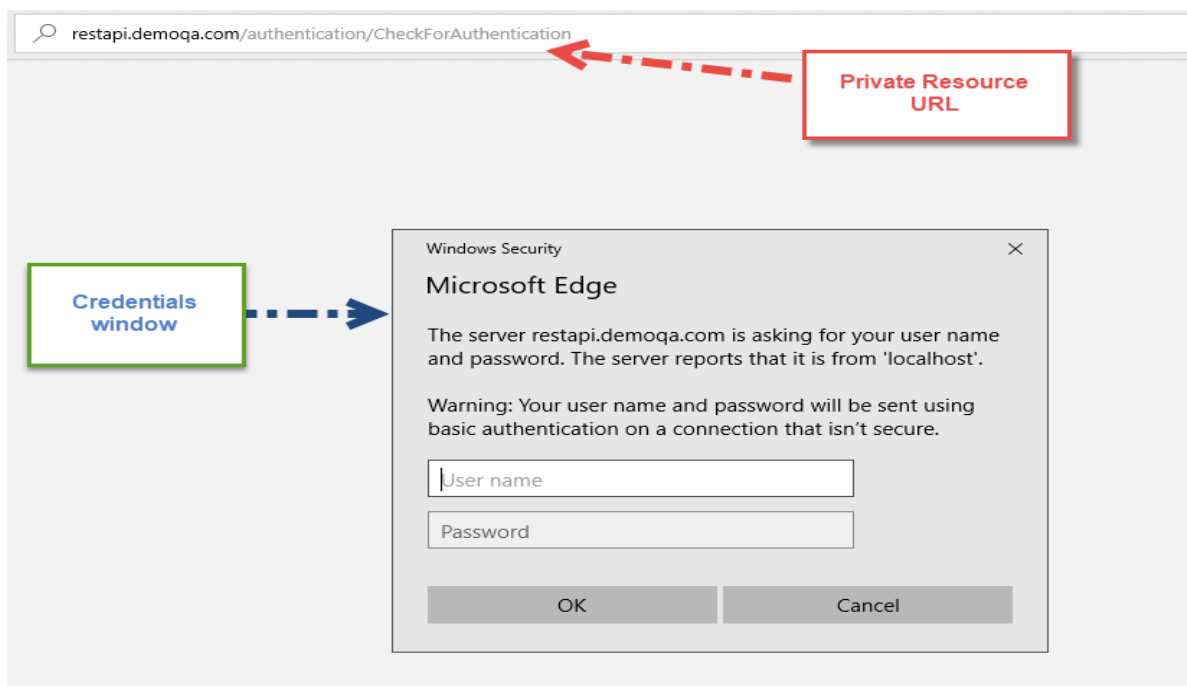
Status code: 401

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.

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



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?

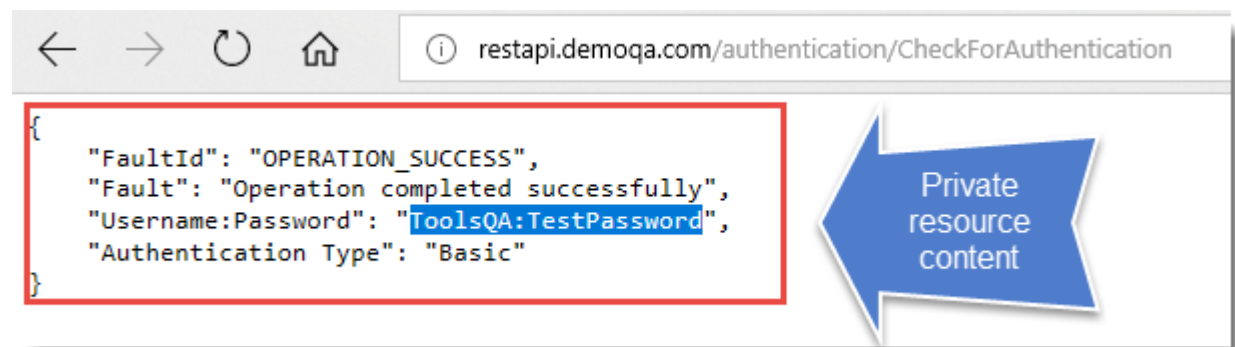
Authorization is the process of giving access to someone. If we are authorized then we have access to that resource. Now to Authorize we need to present credentials and as we discussed earlier that process is called Authentication. Hence Authorization and Authentication are closely related terms and 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 we to access the private resource content. The below image shows the content after successful Authentication.



How to make a PUT Request using Rest Assured?

The **HTTP PUT request method** creates a new resource or substitutes a representation of the target resource with the request payload.

PUT is **idempotent** while POST is not.

Therefore, the PUT method is idempotent because no matter how many times we send the same request, the results will always be the same.

POST is not an idempotent method. In other words, creating a POST multiple times may result in various resources getting created on the server.

Moreover, there is no distinction between PUT and POST if the resource already exists.

The following example will help in conceptually clarifying the difference between PUT and POST requests.

POST /vehicle-management/vehicle: Create a new device

PUT /vehicle-management/vehicle/ {id}: Updates the vehicle information identified by “id”

Appropriate status codes obtained for PUT and POST requests:

POST

201 and a location header pointed to the new resource

400 if we are not able to create an item

PUT

204 for OK (but no content)

200 for OK with Body (Updated response)

400 if the supplied data was invalid

PUT Request code step by step:

Step 1: Create a variable empid which we intend to update with our PUT request.

```
int empid = 15410;
```

Note: The employee ID 15410 used in the example, has been previously created as a resource on the server, and we will update the associated employee information in the PUT request.

Step 2: Create a Request pointing to the Service Endpoint

```
RestAssured.baseURI ="https://dummy.restapiexample.com/api/v1";
```

```
RequestSpecification request = RestAssured.given ();
```

Step 3: Create a JSON request which contains all the fields which we wish to update.

```
// 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("name", "Zion");
```

```
requestParams.put("age", 23);
```

```
requestParams.put("salary", 12000);
```

JSONObject class is present in org.json.simple package. There are multiple nodes of JSON. However, in this example, we are adding using the JSONObject.put(String, String) method. Here, we are updating the name, age, and salary of the employee by passing the values in Request Body.

Step 4. Send JSON content in the body of Request and pass PUT Request

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

```
request.body(requestParams.toJSONString());
```

```
Response response = request.put("/update/"+ empid);
```

Step 5: Validate the PUT Request response received

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

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

```
Assert.assertEquals(statusCode, 200);
```

Once we send the request, we assert the status code obtained to be equivalent to 200 OK.

Putting it all together:

```
public void UpdateRecords(){  
    int empid = 15410;  
    RestAssured.baseURI = "https://dummy.restapiexample.com/api/v1";  
    RequestSpecification request = RestAssured.given();  
    JSONObject requestParams = new JSONObject();  
    requestParams.put("name", "Zion"); // Cast  
    requestParams.put("age", 23);  
    requestParams.put("salary", 12000);  
    request.body(requestParams.toJSONString());  
    Response response = request.put("/update/"+ empid);  
    int statusCode = response.getStatusCode();  
}
```

```
        System.out.println(response.asString());  
        Assert.assertEquals(statusCode, 200);  
    }  
}
```

So far, we have learned about the basics of PUT Request and how it is different from a POST request. JSON data was created using a simple JSON Library. Additionally, we sent it in the content of the body of Request. Moreover, we validated the response by status code as well as the response string obtained once we send the request successfully.

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 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.

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.

Prerequisites for REST API End to End Test

1. Java Setup
2. IDE Setup
3. Maven Setup
4. Add Rest Assured Dependencies
5. Create an Authorised user for the test
6. Run the REST API 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](#) 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](#) and for [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](#). Additionally, please install Maven, if not previously installed as we would need it.

Step 4 - 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.

```

<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>bdd</groupId>
  <artifactId>APIFramework</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>
  <name>APIFramework</name>
  <url>http://maven.apache.org</url>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
  </properties>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-plugin</artifactId>
        <configuration>
          <testFailureIgnore>>true</testFailureIgnore>
        </configuration>
      </plugin>
      <plugin>
        <groupId>net.masterthought</groupId>
        <artifactId>maven-cucumber-reporting</artifactId>
        <version>5.6.2</version>
        <executions>
          <execution>
            <id>execution</id>
            <phase>verify</phase>
            <goals>
              <goal>generate</goal>
            </goals>
            <configuration>
              <projectName>cucumber-jvm-example</projectName>
              <!-- optional, per documentation set this to "true" to bypass generation of
Cucumber Reports entirely, defaults to false if not specified -->
              <skip>>false</skip>
              <!-- output directory for the generated report -->
              <outputDirectory>${project.build.directory}</outputDirectory>
            </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>

```

```

        <!-- optional, defaults to outputDirectory if not specified -->
        <inputDirectory>${project.build.directory}/jsonReports</inputDirectory>
        <jsonFiles>
            <!-- supports wildcard or name pattern -->
            <param>**/*.json</param>
        </jsonFiles>
        <!-- optional, defaults to outputDirectory if not specified -->
        <parallelTesting>>false</parallelTesting>
        <!-- optional, set true to group features by its Ids -->
        <mergeFeaturesById>>false</mergeFeaturesById>
        <!-- optional, set true to get a final report with latest results of the same test
from different test runs -->
        <mergeFeaturesWithRetest>>false</mergeFeaturesWithRetest>
        <!-- optional, set true to fail build on test failures -->
        <checkBuildResult>>false</checkBuildResult>
    </configuration>
</execution>
</executions>
</plugin>
</plugins>
</build>

```

```

<dependencies>

```

```

    <!-- https://mvnrepository.com/artifact/org.hamcrest/hamcrest-library -->

```

```

    <dependency>
        <groupId>org.hamcrest</groupId>
        <artifactId>hamcrest-library</artifactId>
        <version>1.3</version>
        <scope>test</scope>
    </dependency>

```

```

        <!-- https://mvnrepository.com/artifact/io.rest-assured/rest-assured -->

```

```

    <dependency>
        <groupId>io.rest-assured</groupId>
        <artifactId>rest-assured</artifactId>
        <version>4.4.0</version>
    </dependency>

```

```

    <!-- https://mvnrepository.com/artifact/com.fasterxml.jackson.core/jackson-databind -->

```

```

    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-databind</artifactId>
        <version>2.13.0</version>
    </dependency>

```

```

    <!-- https://mvnrepository.com/artifact/io.cucumber/cucumber-java -->

```

```

    <dependency>

```



```

    <groupId>io.cucumber</groupId>
    <artifactId>cucumber-java</artifactId>
    <version>6.10.3</version>
</dependency>

<!-- https://mvnrepository.com/artifact/io.cucumber/cucumber-junit -->
<dependency>
    <groupId>io.cucumber</groupId>
    <artifactId>cucumber-junit</artifactId>
    <version>6.10.3</version>
    <scope>test</scope>
</dependency>

</dependencies>
</project>

```

Step 5: 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.

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. 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.
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.
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

Firstly create a New Package by right click on the src/test/java package and select New >> Package. Moreover, give package a name apiTests and click Finish.

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.

Step 6 - Run the REST API Test

Right-click in the test body and select Run As >> Java Application. The test will run, and see the results in the Console. Consequently, the program executes successfully without any error.

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:

- Write a test in a Feature File
- Create a Test Runner
- Write test code to Step file
- Run test as JUnit test & Maven Command Line
- Execution Reports

Step 1: Write a test in a Feature File

We will highly recommend acquainting with the tutorial on the Feature file. 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.

Feature: Validating place apis

Scenario Outline: Verify if the place is being successfully added

Given Add Place Payload with "<name>" "<language>" "<address>"

When User call "AddPlaceAPI" with "post" method

Then API get success with code 200

And "status" in response body is "OK"

And "scope" is "APP"

Examples:

|name|language|address|

|Mari|English|Injambakam|

|Mathan|Telugu|sithalabaakam|

Create Feature File

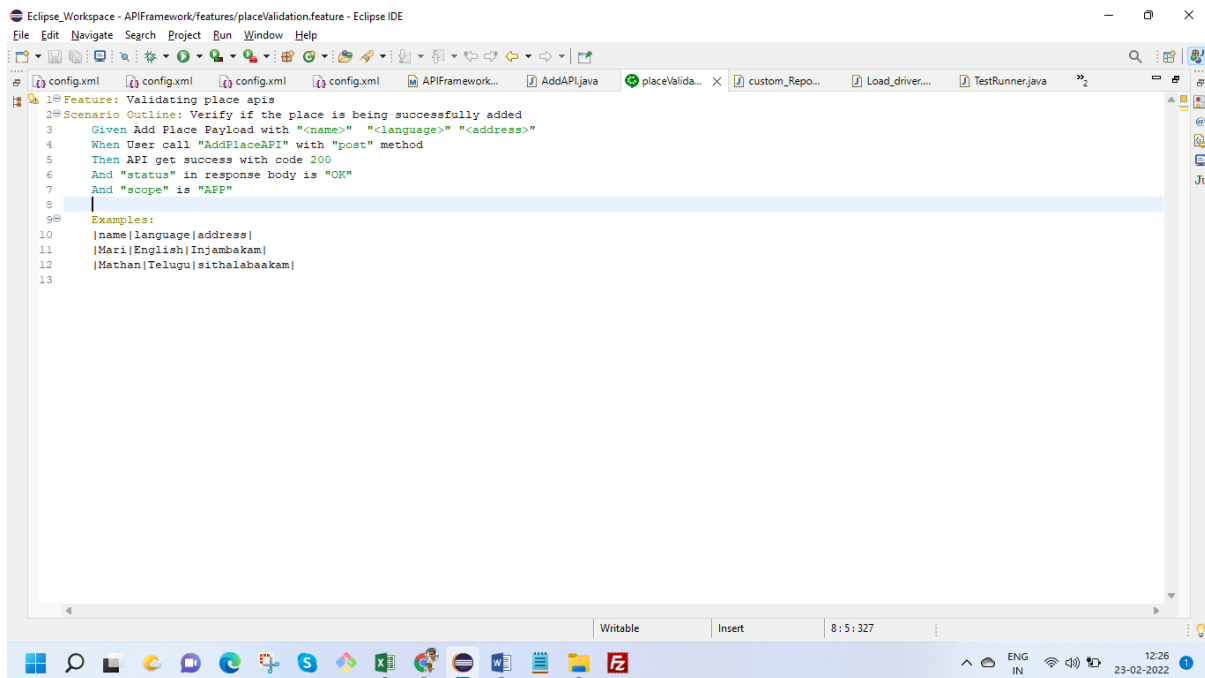
First, create a New Package and name it as functionalTests. We can do it by right-clicking on the src/test/resources and select New >> Package.

Note: It's always recommendable to put all the feature files in the resources folder.

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.

Lastly, add the test steps to the feature file.



Note: As we 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.

Step 2: 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.

Firstly, Right-click on the src/test/java and select a New Package by using New >> Package. Name it as runners.

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",`

)

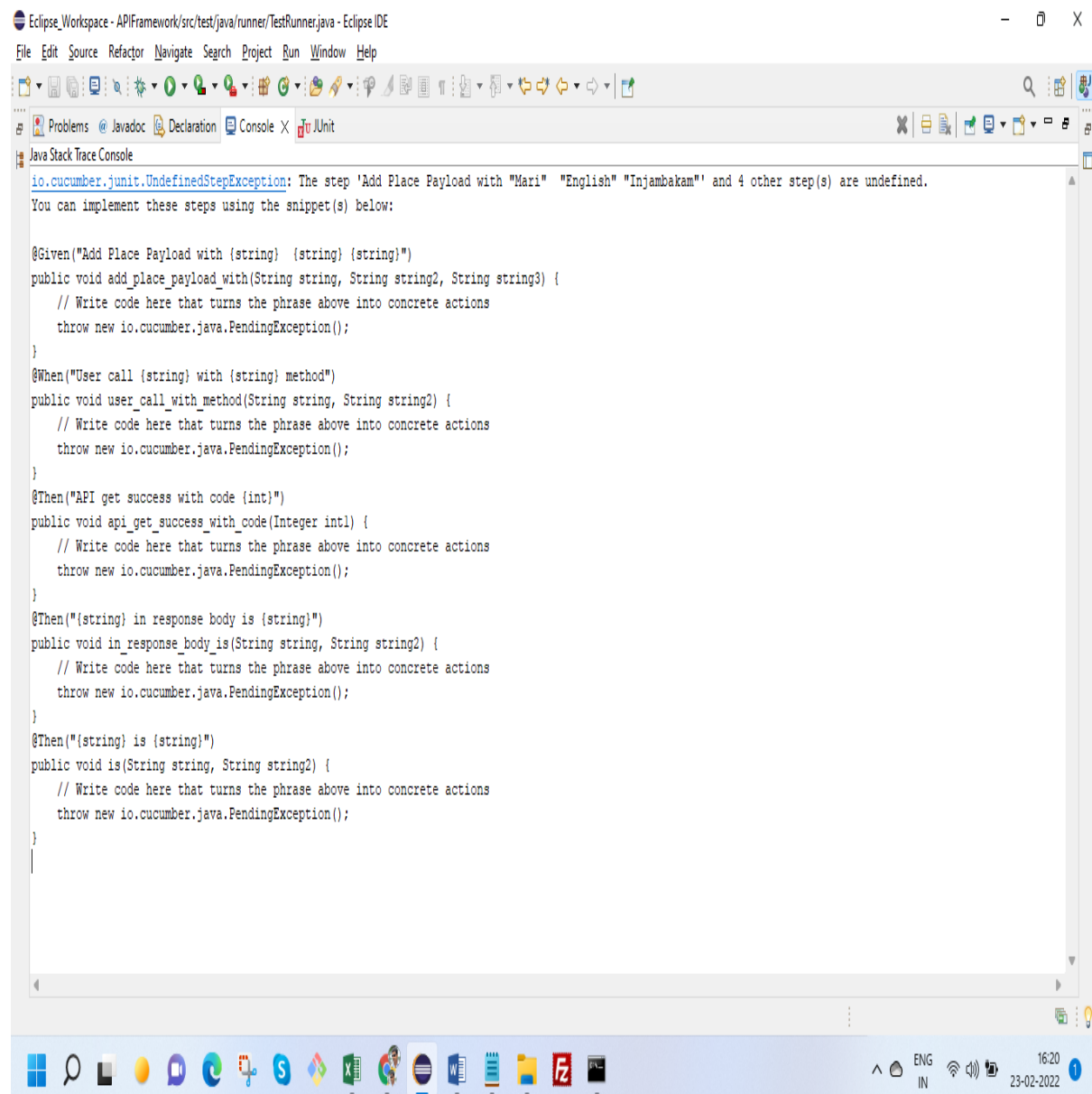
```
public class TestRunner {  
  
}
```

Note: Do not select the public static void main while creating the runner class.

Step 3: Write test code to Step file

Fourthly, as we move ahead in our next step to convert the test code to Step file.

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, we would get the below result in the Eclipse Console.



```
Eclipse_Workspace - APIFramework/src/test/java/runner/TestRunner.java - Eclipse IDE  
File Edit Source Refactor Navigate Search Project Run Window Help  
Problems Javadoc Declaration Console X JUnit  
Java Stack Trace Console  
io.cucumber.junit.UndefinedStepException: The step 'Add Place Payload with "Mari" "English" "Injambakam" and 4 other step(s) are undefined.  
You can implement these steps using the snippet(s) below:  
  
@Given("Add Place Payload with {string} {string} {string}")  
public void add_place_payload_with(String string, String string2, String string3) {  
    // Write code here that turns the phrase above into concrete actions  
    throw new io.cucumber.java.PendingException();  
}  
  
@When("User call {string} with {string} method")  
public void user_call_with_method(String string, String string2) {  
    // Write code here that turns the phrase above into concrete actions  
    throw new io.cucumber.java.PendingException();  
}  
  
@Then("API get success with code {int}")  
public void api_get_success_with_code(Integer int1) {  
    // Write code here that turns the phrase above into concrete actions  
    throw new io.cucumber.java.PendingException();  
}  
  
@Then("{string} in response body is {string}")  
public void in_response_body_is(String string, String string2) {  
    // Write code here that turns the phrase above into concrete actions  
    throw new io.cucumber.java.PendingException();  
}  
  
@Then("{string} is {string}")  
public void is(String string, String string2) {  
    // Write code here that turns the phrase above into concrete actions  
    throw new io.cucumber.java.PendingException();  
}  
}
```

Create a New Package and title it as stepDefinitions by right click on the src/test/java and select New >> Package.

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.

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 static org.junit.Assert.*;
import java.io.IOException;
import io.cucumber.java.en.*;
import io.restassured.builder.ResponseSpecBuilder;
import io.restassured.http.ContentType;
import io.restassured.path.json.JsonPath;
import io.restassured.response.Response;
import io.restassured.specification.RequestSpecification;
import io.restassured.specification.ResponseSpecification;
import resources.APIResources;
import resources.TestDataBuild;
import resources.Utills;
import static io.restassured.RestAssured.given;
```

```
public class AddAPI extends Utills {
```

```
    RequestSpecification res;
    ResponseSpecification resspec;
    Response response;
    TestDataBuild data = new TestDataBuild();
```

```
    @Given("Add Place Payload with {string} {string} {string}")
```

```
    public void add_place_payload_with(String name, String language, String address)
    throws IOException {
```

```
        resspec = new
        ResponseSpecBuilder().expectStatusCode(200).expectContentType(ContentType.JSON).build();
        res = given().spec(reqspec()).body(data.addPayload(name, language,
        address));
```

```
    }
```

```

@When("User call {string} with {string} method")
public void user_call_with_method(String resource, String method) {

    APIResources api = APIResources.valueOf(resource) ;
    System.out.println(api.getResource());

    resspec = new
ResponseSpecBuilder().expectStatusCode(200).expectContentType(ContentType.JSON).build();

    if(method.equalsIgnoreCase("POST")) {
        response = res.when().post(api.getResource()).

then().assertThat().statusCode(200).spec(resspec).extract().response();
    }
}

@Then("API get success with code {int}")
public void api_get_success_with_code(Integer int1) {
    // Write code here that turns the phrase above into concrete actions

    String responseString=response.asString();
    System.out.println(responseString);
    assertEquals(response.getStatusCode(),200);
}

@Then("{string} in response body is {string}")
public void in_response_body_is(String key, String value) {
    // Write code here that turns the phrase above into concrete actions
    String responseString=response.asString();
    JsonPath js = new JsonPath(responseString);
    assertEquals(js.get(key).toString(),value);
}

@Then("{string} is {string}")
public void is(String string, String string2) {
    // Write code here that turns the phrase above into concrete actions

    String responseString=response.asString();
    JsonPath js = new JsonPath(responseString);
    assertEquals(js.get(string).toString(),string2);

}

```

```
}
```

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.

```
package runner;
```

```
import org.junit.runner.RunWith;  
import io.cucumber.junit.CucumberOptions;  
import io.cucumber.junit.Cucumber;
```

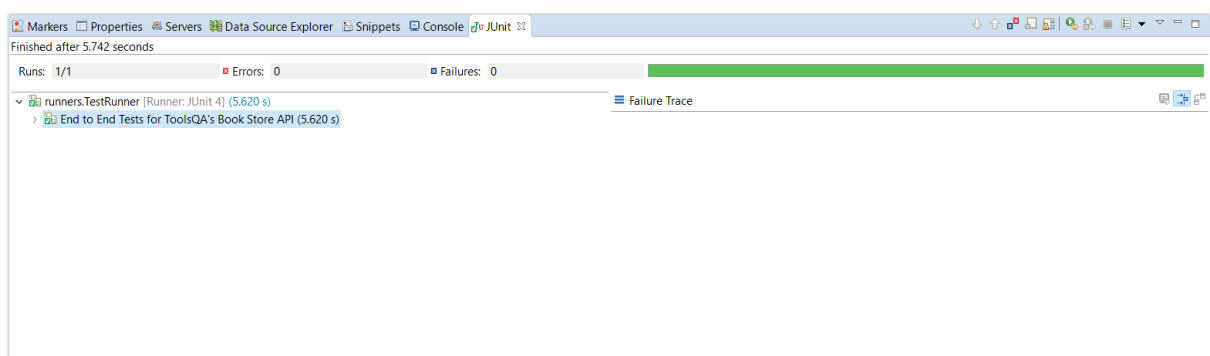
```
@RunWith(Cucumber.class)  
@CucumberOptions(features="features/placeValidation.feature", glue = {"stepDefinitions"},  
monochrome = true)  
public class TestRunner {  
  
}
```

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 4: Run the Cucumber Test

Run as JUnit

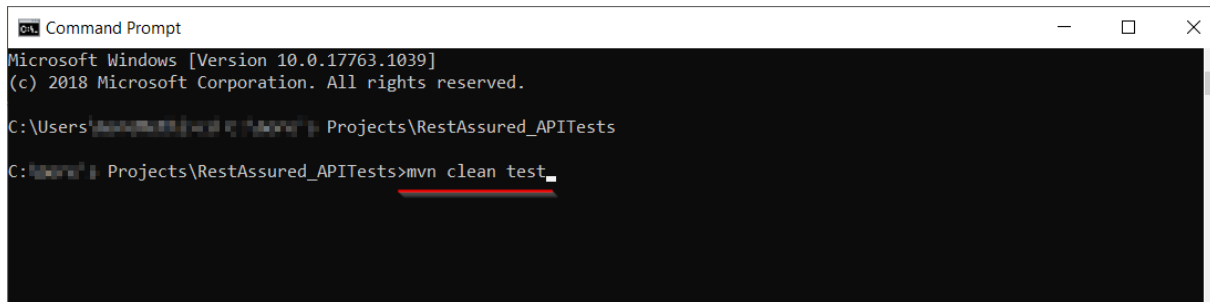
Finally, we are all set to run the first Cucumber test. Right-Click on TestRunner class and Click Run As >> JUnit Test. Cucumber will execute the script the same way it runs in Selenium WebDriver. Consequently, the result will appear in the left-hand side project explorer window in the JUnit tab.



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.



```
Command Prompt
Microsoft Windows [Version 10.0.17763.1039]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\... Projects\RestAssured_APITests
C:\Users\... Projects\RestAssured_APITests>mvn clean test
```

Consequently, we can see the output of the tests below in the command prompt.

Execution Reports:

We can see our request and Response in the text file as below

Request:

```
Request method:    POST
Request URI:  https://rahulshettyacademy.com/maps/api/place/add/json?key=qaclick123
Proxy:         <none>
Request params: <none>
Query params:  key=qaclick123
Form params:   <none>
Path params:   <none>
Headers:       Accept=/*/*
               Content-Type=application/json
Cookies:       <none>
Multiparts:    <none>
Body:
{
  "accuracy": 50,
  "name": "Mari",
  "phone_number": "(+91) 983 893 3937",
  "address": "Injambakam",
  "website": "https://rahulshettyacademy.com",
  "language": "English",
  "location": {
    "lat": -38.383494,
    "lng": 33.427362
  },
  "types": [
    "shoe park",
    "shop"
  ]
}
```


Response:

HTTP/1.1 200 OK
Date: Wed, 16 Feb 2022 13:49:24 GMT
Server: Apache/2.4.18 (Ubuntu)
Access-Control-Allow-Origin: *
Access-Control-Allow-Methods: POST
Access-Control-Max-Age: 3600
Access-Control-Allow-Headers: Content-Type, Access-Control-Allow-Headers, Authorization, X-Requested-With
Content-Length: 194
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: application/json;charset=UTF-8

```
{
  "status": "OK",
  "place_id": "735dee7651551d3a716a406710a0c4c1",
  "scope": "APP",
  "reference":
"7671110478b2786f3867cf780e84f43a7671110478b2786f3867cf780e84f43a",
  "id": "7671110478b2786f3867cf780e84f43a"
}
```

Request:

Request method: POST
Request URI: <https://rahulshettyacademy.com/maps/api/place/add/json?key=qaclick123>
Proxy: <none>
Request params: <none>
Query params: key=qaclick123
Form params: <none>
Path params: <none>
Headers: Accept=/*/*
Content-Type=application/json
Cookies: <none>
Multiparts: <none>
Body:

```
{
  "accuracy": 50,
  "name": "Mathan",
  "phone_number": "(+91) 983 893 3937",
  "address": "sithalabaakam",
  "website": "https://rahulshettyacademy.com",
  "language": "Telugu",
  "location": {
    "lat": -38.383494,
```

```
    "lng": 33.427362
  },
  "types": [
    "shoe park",
    "shop"
  ]
}
```

Response:

```
HTTP/1.1 200 OK
Date: Wed, 16 Feb 2022 13:49:25 GMT
Server: Apache/2.4.18 (Ubuntu)
Access-Control-Allow-Origin: *
Access-Control-Allow-Methods: POST
Access-Control-Max-Age: 3600
Access-Control-Allow-Headers: Content-Type, Access-Control-Allow-Headers, Authorization, X-Requested-With
Content-Length: 194
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: application/json;charset=UTF-8
```

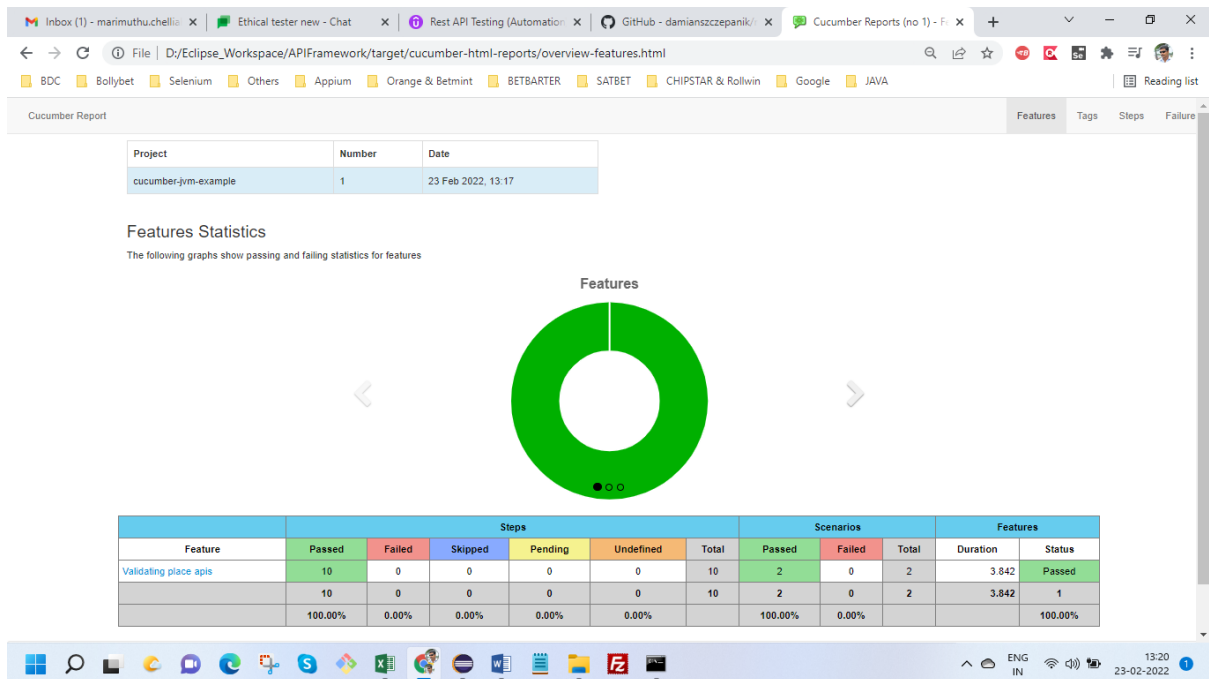
```
{
  "status": "OK",
  "place_id": "44970474f3cea0090a741f4ff50b79c0",
  "scope": "APP",
  "reference":
"78324f40d5ef54b25ca37f3c9e9b7af678324f40d5ef54b25ca37f3c9e9b7af6",
  "id": "78324f40d5ef54b25ca37f3c9e9b7af6"
}
```

```
Eclipse_Workspace - APIFramework/logging.txt - Eclipse IDE
File Edit Navigate Search Project Run Window Help

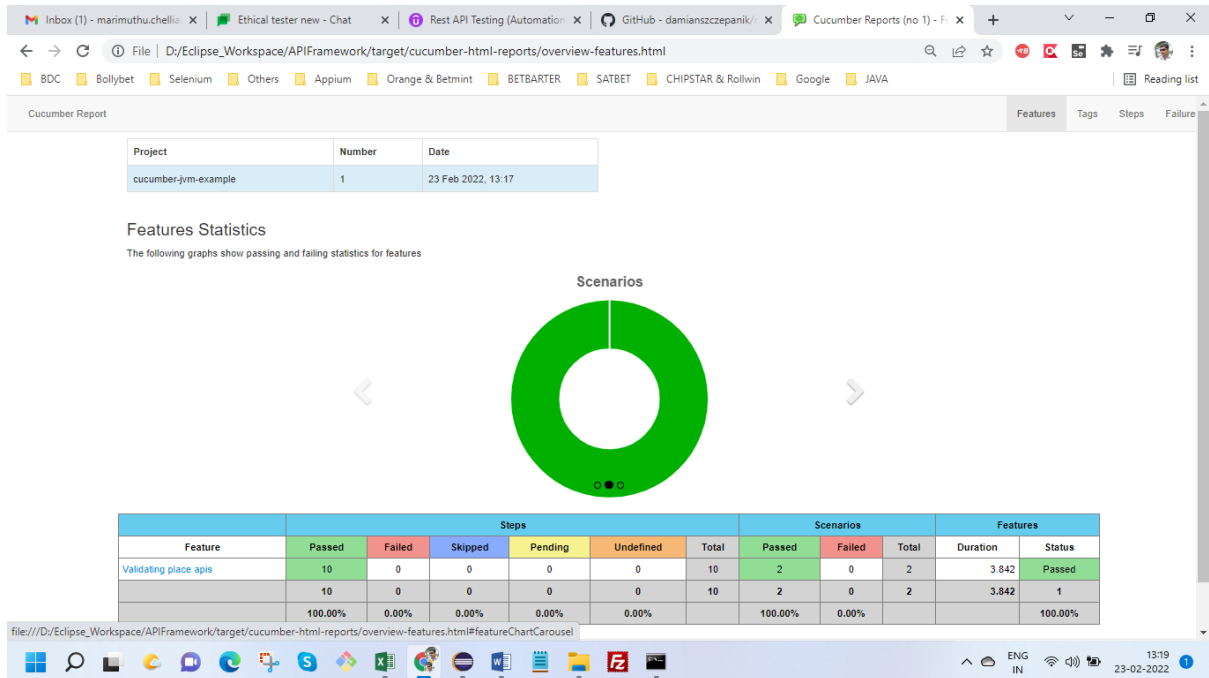
1 Request method: POST
2 Request URI: https://rahulshettyacademy.com/maps/api/place/add/json?key=qaclick123
3 Proxy: <none>
4 Request params: <none>
5 Query params: key=qaclick123
6 Form params: <none>
7 Path params: <none>
8 Headers: Accept= */*
          Content-Type=application/json
9
10 Cookies: <none>
11 Multiparts: <none>
12 Body:
13 {
14   "accuracy": 50,
15   "name": "Mari",
16   "phone_number": "(+91) 983 893 3937",
17   "address": "Injambakam",
18   "website": "https://rahulshettyacademy.com",
19   "language": "English",
20   "location": {
21     "lat": -38.383494,
22     "lng": 33.427362
23   },
24   "types": [
25     "shoe park",
26     "shop"
27   ]
28 }
29 HTTP/1.1 200 OK
30 Date: Wed, 16 Feb 2022 13:49:24 GMT
31 Server: Apache/2.4.18 (Ubuntu)
32 Access-Control-Allow-Origin: *
33 Access-Control-Allow-Methods: POST
34 Access-Control-Max-Age: 3600
```

HTML REPORT:

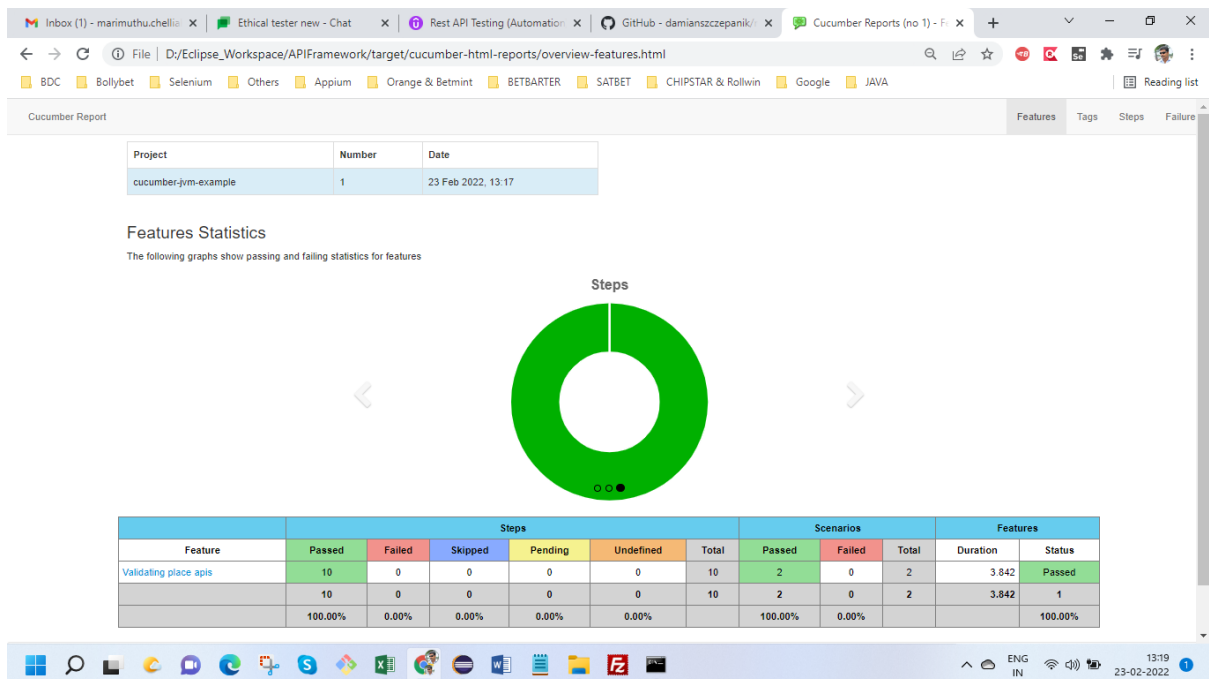
Features:



Scenarios:



Steps:



Feature Report:

Inbox (1) - marimuthu.chelli... x Ethical tester new - Chat x Rest API Testing (Automation... x GitHub - damianszczepanik/... x Cucumber Reports (no 1) - F... x

File D:\Eclipse_Workspace\APIFramework\target\cucumber-html-reports\report-feature_90703443.html

BDC Bollybet Selenium Others Appium Orange & Betmint BETBARTER SATBET CHIPSTAR & Rollwin Google JAVA

Cucumber Report Features Tags Steps Failures

| Project | Number | Date |
|----------------------|--------|--------------------|
| cucumber-jvm-example | 1 | 23 Feb 2022, 13:17 |

Feature Report

| Feature | Steps | | | | | | Scenarios | | | Features | |
|-----------------------|--------|--------|---------|---------|-----------|-------|-----------|--------|-------|----------|--------|
| | Passed | Failed | Skipped | Pending | Undefined | Total | Passed | Failed | Total | Duration | Status |
| Validating place apis | 10 | 0 | 0 | 0 | 0 | 10 | 2 | 0 | 2 | 3.842 | Passed |

Feature Validating place apis

Scenario Outline Verify if the place is being successfully added > 3.482

Scenario Outline Verify if the place is being successfully added > 0.360

generate Cucumber HTML reports via: Jenkins Plugin | Standalone | Sandwich | Maven



Inbox (1) - marimuthu.chelli... x Ethical tester new - Chat x Rest API Testing (Automation... x GitHub - damianszczepanik/... x Cucumber Reports (no 1) - F... x

File D:\Eclipse_Workspace\APIFramework\target\cucumber-html-reports\report-feature_90703443.html

BDC Bollybet Selenium Others Appium Orange & Betmint BETBARTER SATBET CHIPSTAR & Rollwin Google JAVA

Cucumber Report Features Tags Steps Failures

Feature Report

| Feature | Steps | | | | | | Scenarios | | | Features | |
|-----------------------|--------|--------|---------|---------|-----------|-------|-----------|--------|-------|----------|--------|
| | Passed | Failed | Skipped | Pending | Undefined | Total | Passed | Failed | Total | Duration | Status |
| Validating place apis | 10 | 0 | 0 | 0 | 0 | 10 | 2 | 0 | 2 | 3.842 | Passed |

Feature Validating place apis

Scenario Outline Verify if the place is being successfully added > 3.482

Steps >

Given Add Place Payload with "Mari" "English" "Injambakam" 1.469

When User call "AddPlaceAPI" with "post" method 1.400

Then API get success with code 200 0.000

And "status" in response body is "OK" 0.599

And "scope" is "APP" 0.013

Scenario Outline Verify if the place is being successfully added > 0.360

Steps >

Given Add Place Payload with "Mathan" "Telugu" "sithalabaakam" 0.010

When User call "AddPlaceAPI" with "post" method 0.299

Then API get success with code 200 0.000

And "status" in response body is "OK" 0.020

And "scope" is "APP" 0.029



Steps Statistics:

Inbox (1) - marimuthu.chellia x

Ethical tester new - Chat x

Rest API Testing (Automation) x

Github - damianszczepanik/ x

Cucumber Reports (no 1) - S x

File | D:/Eclipse_Workspace/APIFramework/target/cucumber-html-reports/overview-steps.html

BDC Bollybet Selenium Others Appium Orange & Betmint BETBARTER SATBET CHIPSTAR & Rollwin Google JAVA

Reading list

Cucumber Report

FeaturesTagsStepsFailures

| Project | Number | Date |
|----------------------|--------|--------------------|
| cucumber-jvm-example | 1 | 23 Feb 2022, 13:17 |

Steps Statistics

The following graph shows step statistics for this build. Below list is based on results. step does not provide information about result then is not listed below. Additionally @Before and @After are not counted because they are part of the scenarios, not steps.

| Implementation | Occurrences | Average duration | Max duration | Total durations | Ratio |
|---|-------------|------------------|--------------|-----------------|---------|
| stepDefinitions.AddAPI.add_place_payload_with(java.lang.String,java.lang.String,java.lang.String) | 2 | 0.739 | 1.469 | 1.479 | 100.00% |
| stepDefinitions.AddAPI.api_get_success_with_code(java.lang.Integer) | 2 | 0.000 | 0.000 | 0.000 | 100.00% |
| stepDefinitions.AddAPI.in_response_body_is(java.lang.String,java.lang.String) | 2 | 0.309 | 0.599 | 0.619 | 100.00% |
| stepDefinitions.AddAPI.is(java.lang.String,java.lang.String) | 2 | 0.021 | 0.029 | 0.042 | 100.00% |
| stepDefinitions.AddAPI.user_call_with_method(java.lang.String,java.lang.String) | 2 | 0.849 | 1.400 | 1.699 | 100.00% |
| 5 | 10 | 0.384 | 1.699 | 3.842 | Totals |

generate Cucumber HTML reports via: Jenkins Plugin | Standalone | Sandwich | Maven

ENG IN 13:23 23-02-2022

Failures:

Inbox (1) - marimuthu.chelli x Ethical tester new - Chat x Rest API Testing (Automation) x GitHub - damianszczepanik/ x Cucumber Reports (no 1) - F x

D:\Eclipse_Workspace\APIFramework\target\cucumber-html-reports\overview-failures.html

BDC Bollybet Selenium Others Appium Orange & Betmint BETBARTER SATBET CHIPSTAR & Rollwin Google JAVA

Cucumber Report Features Tags Steps Failures

| Project | Number | Date |
|----------------------|--------|--------------------|
| cucumber-jvm-example | 1 | 23 Feb 2022, 13:17 |

Failures Overview

The following summary displays scenarios that failed.

You have no failed scenarios in your Cucumber report

generate Cucumber HTML reports via: Jenkins Plugin | Standalone | Sandwich | Maven

Json Report:

Eclipse_Workspace - APIFramework\target\jsonReports\cucumber-report.json - Eclipse IDE

File Edit Navigate Search Project Run Window Help

Package Explorer

- APIFramework
 - src/main/java
 - src/test/java
 - Maven Dependencies
 - src/test/resources
 - JRE System Library [J2SE-1.5]
 - features
 - resources
 - src
 - target
 - cucumber-html-reports
 - generated-sources
 - generated-test-sources
 - jsonReports
 - cucumber-report.json
 - maven-archiver
 - maven-status
 - surefire-reports
 - APIFramework-0.0.1-SNAPSHOT
 - logging.txt
- pom.xml
- ATTesting
- BDC_PROD
- BDC_PROD_MOBILE
- BETBARTER_PROD
- BETBARTER_PROD_MOBILE
- BETBARTER_PROD_SPORTS
- BETBARTER_PROD_SPORTS_MOBILE
- BOLLYBET_PROD
- BOLLYBET_PROD_MOBILE
- BOLLYBET_PROD_SPORTS

```
1 [
2 {
3   "line": 1,
4   "elements": [
5     {
6       "start_timestamp": "2022-02-23T07:46:33.637Z",
7       "line": 11,
8       "name": "Verify if the place is being successfully added",
9       "description": "",
10      "id": "validating-place-apis;verify-if-the-place-is-being-successfully-added;2",
11      "type": "scenario",
12      "keyword": "Scenario Outline",
13      "steps": [
14        {
15          "result": {
16            "duration": 1469760700,
17            "status": "passed"
18          },
19          "line": 3,
20          "name": "Add Place Payload with \"Mari\" \"English\" \"Injambakam\"",
21          "match": {
22            "arguments": f
```

Problems Javadoc Declaration Console JUnit

<terminated> TestRunner [JUnit] C:\Program Files\Java\jdk-11.0.13\bin\javaw.exe (23-Feb-2022, 1:11:10 pm - 1:11:16 pm)

```
/maps/api/place/add/json
{"status":"OK","place_id":"34789866e1013695f46bf5577d1a14d","scope":"APP","reference":"96562e49f27391b63cd8bd2bbf80909096562e49f27391b6
/maps/api/place/add/json
{"status":"OK","place_id":"6fc7478cc0701821b5d33521f1aba2b7","scope":"APP","reference":"a6eb266131d068eb268de2a6b41cdbeca6eb266131d068eb
```

Writable Insert 16:38:465

ENG IN 23-02-2022