

# API Testing by Maruf

📌 Status	In Progress
🔗 Course Link	<a href="https://www.udemy.com/course/rest-api-automation-testing-rest-assured/">https://www.udemy.com/course/rest-api-automation-testing-rest-assured/</a>
☰ Course Name	Rest API Testing (Automation) from Scratch-Rest Assured Java
📅 Started On	@February 28, 2023

## ▼ Intro

Lets say Marriot Hotels they have their own website (frontend client) where it shows all the available rooms and through here customers can make reservations. Now there are other platforms like [booking.com](https://www.booking.com), [hotels.com](https://www.hotels.com) who also provide customer the option book rooms in Marriot Hotels. But the problem here is, to let websites like these to able to make reservation Marriot needs to expose their backend code. This is not an ideal approach as there many websites like these and exposing their intellectual property to others poses a security risk. This is where APIs come in. They pass and handle the request sent from the frontend to backend.

Another scenario is, frontend can be written in many languages. However, backend is usually written in a few preferred languages like java, python etc. Now if the frontend is written in Java and the backend is in python. They wont be compatible. This also where APIs come in.



API stands for Application Programing Interface. It is a type of software interface, offering a service to other pieces of software. - Wikipedia

Rest assured is the most common library to automate API's using Java.

In the industry there are 2 types of API/Services that are followed:

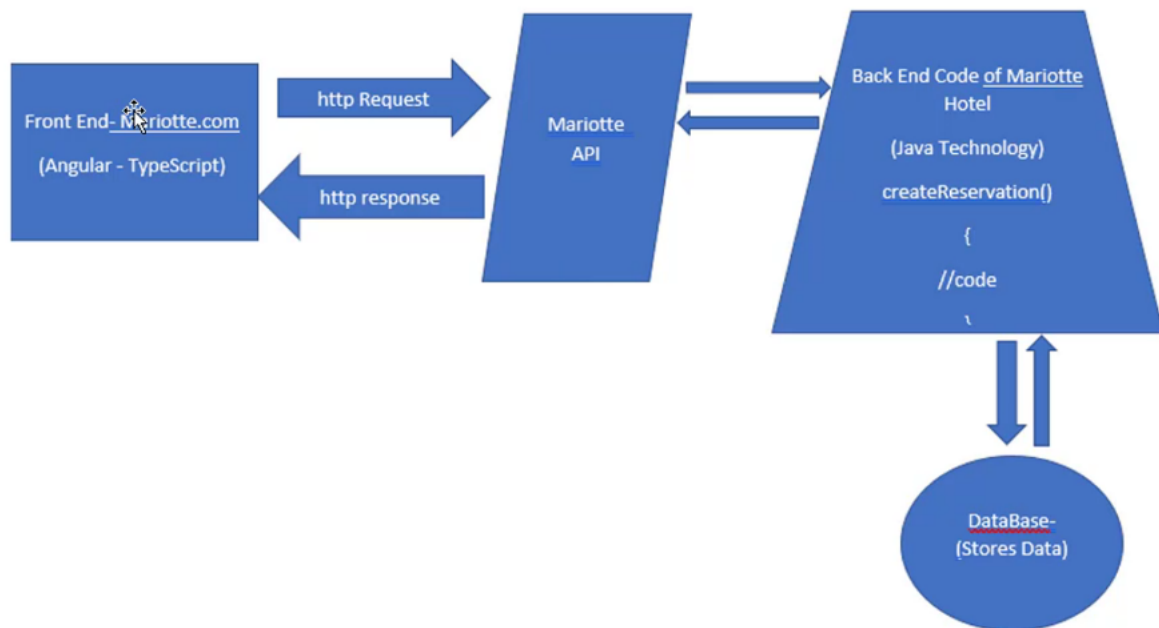
1. REST Assured
2. SOAP

Soap is supported in legacy platforms and services whereas REST is a more newer and modern form. REST is a light weight protocol.

**Representational State Transfer** (REST) is a software architecture that imposes conditions on how an API should work. REST was initially created as a guideline to manage communication on a complex network like the internet.

## ▼ Architecture

Frontend → API → Backend code → Database



- This API handles the request sent from websites like [booking.com](https://www.booking.com) and the Marriot website itself.
- The information sent from the backend is done using the `HTTP` protocol. Which is independent of any language.
- The sent information can be `JSON` or `XML`.

## ▼ CRUD

CRUD stands for **Create Read Update Delete**.

### End point/Base URI

Address where API is hosted on the server.

HTTP methods are commonly used to communicate with Rest API.



URI identifies a resource and differentiates it from others by using a name, location, or both. URL identifies the web address or location of a unique resource. URI contains components like a scheme, authority, path, and query. URL has similar components to a URI, but its authority consists of a domain name and port - [Hostinger](https://www.hostinger.com)

## Methods

1. **GET**: The `GET` method is used to extract information from the given server using a given URI. No payload is required.
2. **POST**: A `POST` request is used to send data to the server. For example: customer information, file upload etc.
3. **PUT**: Replaces all current representations of the target resource with the uploaded content. For example: updating phone number.
4. **DELETE**: Removes all the current representations of the target resource given a URI.

## ▼ Resources and parameters

Resources represent API/Collection which can be accessed from the server.

Google.com/maps

Google.com is the server or the base URL. maps is the resource. So when we are hitting a URL the server knows that we are looking for the maps feature available on their servers. So every API test MUST have a base URL and a resource.

There are few more details that we might need to pass, This is based on the parameters required for the URL to work. There are 2 types of parameters:

1. **Path Parameters:** these are variable parts of URL path. They are used to point to a specific resource within the collection.  
Such as: [www.google.com/images/sdasdawd](http://www.google.com/images/sdasdawd)
2. **Query Parameters:** Query parameters are used to sort/filter the resources. They are identified using `?` character. Each query parameter is separated out using `&`. For example: [www.amazon.com/orders?sort\\_by=2023/03/02](http://www.amazon.com/orders?sort_by=2023/03/02)

Lets look at an example:

<https://www.google.com/search?q=newyork&oq=newyork&aqs=chrome..69i57j46i10i512j0i10i512l4j46i10i512j0i10i512l2j0i271.2160j0j7&sourceid=chrome&ie=UTF8>

- <https://www.google.com> → Base URL
- /search → Resource name
- ?q=newyork → is the query parameter

The final endpoint request URL can be constructed as below:

Base URL/resource(Path Parameters)/?Query Parameters

## ▼ Headers/Cookies

Headers represent the meta data associated with the API request and response. In layman terms, we were sending additional details to API to process our request. An example of this is:

POST

http://216.10.245.166/Library/Addbook.php

Params

Authorization

Headers (8)

Body

Pre-request Script

Tests

Settings

	Key	Value
<input checked="" type="checkbox"/>	Postman-Token	<calculated when request is sent>
<input checked="" type="checkbox"/>	Content-Type	application/json
<input checked="" type="checkbox"/>	Content-Length	<calculated when request is sent>
<input checked="" type="checkbox"/>	Host	<calculated when request is sent>
<input checked="" type="checkbox"/>	User-Agent	PostmanRuntime/7.32.2
<input checked="" type="checkbox"/>	Accept	/*/*
<input checked="" type="checkbox"/>	Accept-Encoding	gzip, deflate, br
<input checked="" type="checkbox"/>	Connection	keep-alive
	Key	Value

## ▼ Using Postman with example APIs

When testing API we need to know all the API contracts. By contract we refer to the following items:

- Base URL
- Resource
- Query Parameters
- HTTP Method
- JSON Body
- Expected Response

Lets look at an example:

## Google Maps Add Location API (POST)

- Base URL: <https://rahulshettyacademy.com>
- Resource: `/maps/api/place/add/json`
- Query Parameters: key =qaclick123
- HTTP Method: `POST`

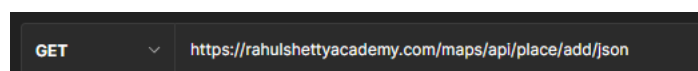
Sample Body :

```
{
  "location": {
    "lat": -38.383494,
    "lng": 33.427362
  },
  "accuracy": 50,
  "name": "Frontline house",
  "phone_number": "(+91) 983 893 3937",
  "address": "29, side layout, cohen 09",
  "types": [
    "shoe park",
    "shop"
  ],
  "website": "http://google.com",
  "language": "French-IN"
}
```

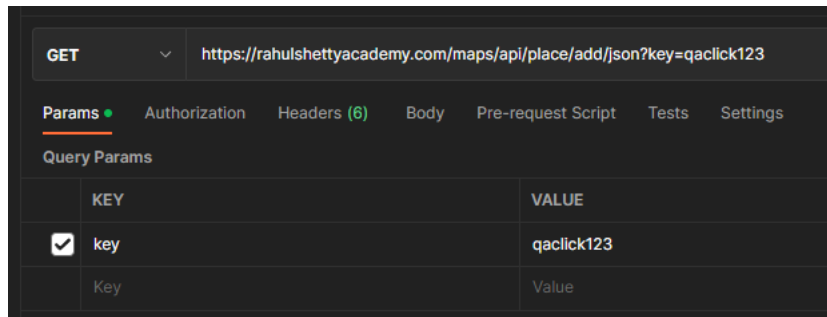
Sample Response

```
{
  "status": "OK",
  "place_id": "928b51f64aed18713b0d164d9be8d67f",
  "scope": "APP",
  "reference": "736f3c9bec384af62a184a1936d42bb0736f3c9bec384af62a184a1936d42bb0",
  "id": "736f3c9bec384af62a184a1936d42bb0"
}
```

We can pass the base URI and resources in the URL section of postman request.

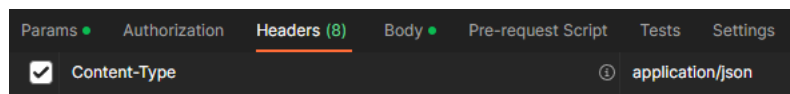


For query parameters we can directly write it here or use the `Params` option where we can add our key value pairs. This would automatically add our parameter values to the URL:



The HTTP method is designed by the developer so we don't have to figure out what method to use! With **POST** method we are sending the body information securely. On the other hand, **GET** would send the information through the URL itself.

With respect to (WRT) our example we need to send a JSON body. We also need to know the format and its sequence. In most cases the body is in JSON format. When adding JSON body postman would automatically add the following indicating that the body is in JSON format.



From running the above sample code we would get a 200 response with the results:

```
{
  "status": "OK",
  "place_id": "27e907cd5baa29271338d8a33d21d5ba",
  "scope": "APP",
  "reference": "e6a34d209958f227c79715230e1d42f2e6a34d209958f227c79715230e1d42f2",
  "id": "e6a34d209958f227c79715230e1d42f2"
}
```

## Google maps get place API (GET):

This API will get existing place details from Server

- Complete URL : [http://rahulshettyacademy.com/maps/api/place/get/json?place\\_id=xxxx&key=qaclick123](http://rahulshettyacademy.com/maps/api/place/get/json?place_id=xxxx&key=qaclick123)
- Base URL: <https://rahulshettyacademy.com>
- Resource: `/maps/api/place/get/json`
- Parameters: key, place\_id (received)
- HTTP request: **GET**
- Note: Key value is hardcoded and it is always qaclick123

Using the place ID from the above results we make the GET request:

```
{
  "location": {
    "latitude": "-38.383494",

```

```

    "longitude": "33.427362"
  },
  "accuracy": "50",
  "name": "Frontline house",
  "phone_number": "(+91) 983 893 3937",
  "address": "29, side layout, cohen 09",
  "types": "shoe park,shop",
  "website": "http://google.com",
  "language": "French-IN"
}

```



**POST** is used for updating anything. **POST** acts like a **superset** so it can be used to **DELETE** items too. It depends on how the API is designed.



**GET** doesn't require any headers because we are not sending any payload.

## Google maps put place API (PUT)

This API Will update existing place in the server with new values

- Complete URL : [http://rahulshettyacademy.com/maps/api/place/update/json?place\\_id=xxxx&key=qaclick123](http://rahulshettyacademy.com/maps/api/place/update/json?place_id=xxxx&key=qaclick123)
- Base URL : <https://rahulshettyacademy.com>
- Resource: `/maps/api/place/update/json`
- Query Parameters: key, place\_id
- HTTP Method: **PUT**
- Note: same key value

Request:

```

{
  "place_id": "27e907cd5baa29271338d8a33d21d5ba",
  "address": "Maruf Monem",
  "key": "qaclick123"
}

```

The response we get is:

```

{
  "msg": "Address successfully updated"
}

```

To confirm that it actually updated it we can use the GET request to verify:

```

{
  "location": {
    "latitude": "-38.383494",
    "longitude": "33.427362"
  },
  "accuracy": "50",
  "name": "Frontline house",
  "phone_number": "(+91) 983 893 3937",
  "address": "Maruf Monem",
  "types": "shoe park,shop",
  "website": "http://google.com",
  "language": "French-IN"
}

```

Again here we can use `POST` but it generally best practice to use `PUT` when updating.

## Rest Assured

Rest assured (RA) is a Java based library used to rest RESTful web services/APIs. its a JAVA jar.

RA works on **3 principals**:

1. **Given**: it takes all the input details needed for an API
2. **When**: Submit the API → resource and http methods
3. **Then**: Validate the response

### ▼ PUT Request code example with validation

Lets look at an example code of this:

Below we are using the previously discussed PUT request.

```
import io.restassured.RestAssured;
import static io.restassured.RestAssured.*;

public class BasicApiTest_PUT {
    public static void main(String[] args) {

        // 1. **Given**: it takes all the input details needed for an API
        // 2. **When**: Submit the API
        // 3. **Then**: Validate the response

        RestAssured.baseURI="https://rahulshettyacademy.com";
        // to use given we need to add the static package
        given()
            .log().all()
            .queryParams("key", "qaclick123")
            .header("Content-Type", "application/json")
            .body("{\n" +
                "    \"location\": {\n" +
                "        \"lat\": -38.383494,\n" +
                "        \"lng\": 33.427362\n" +
                "    },\n" +
                "    \"accuracy\": 50,\n" +
                "    \"name\": \"Frontline house\",\n" +
                "    \"phone_number\": \"(+91) 983 893 3937\",\n" +
                "    \"address\": \"29, side layout, cohen 09\",\n" +
                "    \"types\": [\n" +
                "        \"shoe park\",\n" +
                "        \"shop\"\n" +
                "    ],\n" +
                "    \"website\": \"http://google.com\",\n" +
                "    \"language\": \"French-IN\"\n" +
                "}")
            .when()
            .post("/maps/api/place/add/json")
            .then().log().all().assertThat().statusCode(200);

    }
}
```

Lets do a breakdown of this code:

`given()`, as discussed takes all the items needed for the API request which in our case is the query parameters, the header information and the payload/body. The values for query parameter and header are passed as a key-value pair. Each of these methods can be chained together using ( `.` ). The `JSON` content when pasting would automatically be formatted with backward slashes to avoid any issues.



`given()` can be only used when the static package is imported: `import static io.restassured.RestAssured.*;`

After adding all the necessary info now we need to tell what type of request this is and what is the resource. Which is done by the method(): `.post("/maps/api/place/add/json")`

After that we need specify if we want anything to verify such as for our case we verify the response code using assert.  
`.then().log().all().assertThat().statusCode(200);`

Any validation related stuff starts after `assertThat()`.

We are using `log().all()` to print out all the logs for both the request and response. The final response looks like this:

```
"C:\Program Files\Java\jdk-19\bin\java.exe" "-javaagent:G:\Program Files (x86)\IntelliJ IDEA Community Edition 2022.2\lib\idea_rt.jar=62
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:
{
  "location": {
    "lat": -38.383494,
    "lng": 33.427362
  },
  "accuracy": 50,
  "name": "Frontline house",
  "phone_number": "(+91) 983 893 3937",
  "address": "29, side layout, cohen 09",
  "types": [
    "shoe park",
    "shop"
  ],
  "website": "http://google.com",
  "language": "French-IN"
}
HTTP/1.1 200 OK
Date: Sun, 05 Mar 2023 02:50:17 GMT
Server: Apache/2.4.41 (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": "6cd1a546a0b15725f11fc0fab176fab6",
  "scope": "APP",
  "reference": "59414a1d2dfb6c30cbe51d2cc288e82159414a1d2dfb6c30cbe51d2cc288e821",
  "id": "59414a1d2dfb6c30cbe51d2cc288e821"
}

Process finished with exit code 0
```

We can validate other items too such as:



- The body items like the scope: `body("scope", equalTo("APP"))`  
This (`equalTo()`) uses the hamcrest static package: `import static org.hamcrest.Matchers.*;`
- The header items (a very common item to test): `header("server", "Apache/2.4.41 (Ubuntu)");`



When using header method after `given()` it acts as an input and the same method after `then()` acts as an output.

## ▼ Modularizing Payload

In the above example we have added the payload in the body itself which is not an ideal approach. To solve this we can have a static class that would handle this payload information.

```
package files;

public class payload {
    public static String addPlace(){
        return "{\n" +
            "    \"location\": {\n" +
            "        \"lat\": -38.383494,\n" +
            "        \"lng\": 33.427362\n" +
            "    },\n" +
            "    \"accuracy\": 50,\n" +
            "    \"name\": \"Frontline house\",\n" +
            "    \"phone_number\": \"(+91) 983 893 3937\",\n" +
            "    \"address\": \"29, side layout, cohen 09\",\n" +
            "    \"types\": [\n" +
            "        \"shoe park\",\n" +
            "        \"shop\"\n" +
            "    ],\n" +
            "    \"website\": \"http://google.com\",\n" +
            "    \"language\": \"French-IN\"\n" +
            "    }";
    }
}
```

Then we just call the method: `.body(payload.addPlace())`

## ▼ Capturing the response

We might need to extract the information from the response for further testing. To do that we can use the following:

```
String response = given()
    .log().all()
    .queryParams("key", "qaclick123")
    .header("Content-Type", "application/json")
    .body(payload.addPlace())
    .when()
    .post("/maps/api/place/add/json")
    .then().assertThat().statusCode(200).body("scope", equalTo("APP")).header("server", "Apache/2.4.41 (Ubuntu)").extract()
```

Printing the result:

```
{"status":"OK","place_id":"9ec461168af7fc70af1a1d16eeb92456","scope":"APP","reference":"a08306ab0de7bbcd1e6d8b7e1e33f628a08306ab0de7bber"}
```

This string now contains all our variable values but we need to extract that information out of that string, To do that we use the following. It would help to parse the JSON file.

```
JsonPath js = new JsonPath(response);
```

the `JsonPath` object can now get all the attribute values.

```
String place_id = js.getString("place_id");
```

the `getString` method above takes a path. WTO example, to get the `place_id` we check what its path is.

```
{
  "status": "OK",
  "place_id": "6cd1a546a0b15725f11fc0fab176fab6",
  "scope": "APP",
  "reference": "59414a1d2dfb6c30cbe51d2cc288e82159414a1d2dfb6c30cbe51d2cc288e821",
  "id": "59414a1d2dfb6c30cbe51d2cc288e821"
}
```

here `place_id` is in the parent location so directly writing the attribute/key name should suffice. However if it was something like this:

```
{
  "location": {
    "lat": -38.383494,
    "lng": 33.427362
  }
}
```

it would be `"location.lat"`

## ▼ Validating the response

We have a GET method that returns the details of a place based on the `place_id`. We can use that combined with the `assert` method to verify whether the previously updated address was the correct one. To do that we write the following:

```
given()
    .log().all()
    .queryParams("key", "qaclick123")
    .queryParams("place_id", place_id)
.when()
    .get("/maps/api/place/get/json")
.then()
    .log().all().assertThat().statusCode(200).body("address", equalTo("Maruf Monem"));
```

We can also use TestNG assert methods (more on this in the later sections).

```
JsonPath jsGET = ResuableMethods.rawToJson(getResponse);
String addressGET = jsGET.get("address");
Assert.assertEquals(addressGET, addressValue);
```

## ▼ Parsing JSON

### Objective

Given the following JSON

```
{
  "dashboard": {
    "purchaseAmount": 910,
    "website": "rahulshettyacademy.com"
  },
  "courses": [
    {
      "title": "Selenium Python",
      "price": 50,
      "copies": 6
    }
  ]
}
```

```

    },
    {
      "title": "Cypress",
      "price": 40,
      "copies": 4
    },
    {
      "title": "RPA",
      "price": 45,
      "copies": 10
    }
  ]
}

```

We want to:

1. Print No of courses returned by the API
2. Print Purchase Amount
3. Print Title of the first course
4. Print All course titles and their respective Prices
5. Print no of copies sold by RPA Course
6. Verify if Sum of all Course prices matches with Purchase Amount



The `[]` in the JSON represents an array

During initial phases of development we might not have the actual API ready for testing. For those cases we need to have a mock API ready (which we got during analysis and product design phases). This would make sure we can continue with our API testing development even before the actual feature has been implemented.

## Print No of courses returned by the API

As seen in JSON above the courses property is an array. So the typical array method apply such as size;

```
int courseArraySize = js.getInt("courses.size()");
```

## Print Purchase Amount

```
int purchaseAmount = js.getInt("dashboard.purchaseAmount");
```

## Print Title of the first course

```
String firstCourse = js.getString("courses[0].title");
```



The `get()` method by default returns Object

## Print All course titles and their respective Prices

```

for (int i =0; i<courseArraySize; i++){
    System.out.println("Course " + (i+1) + ": " + js.getString("courses["+i+"].title") + " - price: " + js.getString("courses["+i+"].price") + " - copies: " + js.getString("courses["+i+"].copies"));
}

```

## Print no of copies sold by RPA Course

```

for (int i =0; i<courseArraySize; i++){
    String courseTitle = js.getString("courses["+i+"].title");
    if (courseTitle.equalsIgnoreCase("RPA")){
        System.out.println("RPA course sold: " + js.getString("courses["+i+"].copies")+ " copies");
    }
}

```

## Verify if Sum of all Course prices matches with Purchase Amount

```

int sum=0;
for (int i =0; i<courseArraySize; i++){
    int priceOfEach = js.getInt("courses["+i+"].price");
    int amount = js.getInt("courses["+i+"].copies");
    sum = sum + (priceOfEach*amount);
}
System.out.println("Total calculated earnings: " + sum);
if (sum==purchaseAmount){
    System.out.println("Matched");
}else{
    System.out.println("Did not match");
}

```

## ▼ Dynamic JSON Payloads

To understand the concepts discussed below we need to understand the goal. Below is a test to create a new book entry for the LibraryApi.

```

@Test
public void addBook(){
    RestAssured.baseURI= "http://216.10.245.166";

    String response = given()
        .log().all()
        .header("Content-Type", "application/json")
        .body(DynamicJsonPayload.newBook("isbn1", "aisle1"))
    .when()
        .post("Library/Addbook.php")
    .then()
        .assertThat().statusCode(200).extract().response().asString();

    JsonPath js = ResuableMethods.rawToJson(response);
    String bookID = js.getString("ID");
    System.out.println(bookID);
}

```

There is a payload method:

```

public static String newBook(String isbn, String aisle){

    String bookDetails = "{\n" +
        "  \"name\": \"Learn Appium Automation with Java\",\n" +
        "  \"isbn\": \""+isbn+"\",\n" +
        "  \"aisle\": \""+aisle+"\", \n" +

```

```

        " \author\": \"John foe\\n\" +
        "}\";
    return bookDetails;
}

```

Now we want to run this tests automatically from a given dataset that contains unique ISBN and Aisle numbers.

## Data Provider

Data provider is a TestNG annotation. Which works as a source of data for any test desired. We need to configure the tests to make sure of this `@DataProvider`.

The `dataProvider` method looks like the following WRTO our example:

```

@DataProvider(name="booksData")
public Object[][] getData(){
    return new Object[][]{
        {"isbn1", "aisle1"},
        {"isbn2", "aisle2"},
        {"isbn3", "aisle3"}
    };
}

```

Here we are returning an multidimensional array. Each entry in that array contains another array holding the ISBN and Aisle value.

To connect it to our test we can write the following:

```

@Test(dataProvider = "booksData")

```

Now the problem is that we did make the connection however the test itself needs to have the ability to handle the returned data in our case the multidimensional array. This is quite simple, we just need to add arguments in the method:

```

@Test(dataProvider = "booksData")
public void addBook(String isbn, String aisle){
    ...
    .body(DynamicJsonPayload.newBook(isbn, aisle))
}

```

<https://s3-us-west-2.amazonaws.com/secure.notion-static.com/a5cd4814-18e7-4d06-a970-f0c4c8fd9866/DynamicJsonPayload.java>

<https://s3-us-west-2.amazonaws.com/secure.notion-static.com/6f2017df-e3a4-43c8-8c33-934b74509a93/DynamicJsonPayload.java>

## ▼ Reading a static JSON file

There are cases where the JSON file would not update so we can directly use that JSON file as our payload. To do that we do the following:

```

RestAssured.baseURI="https://rahulshettyacademy.com";
String path = "X:\\Career\\Learning\\API\\Rest-Assured\\src\\main\\java\\files\\newBook.Json";
//      to use given we need to add the static package
given()
    .log().all()
    .queryParams("key", "qaclick123")
    .header("Content-Type", "application/json")
    .body(new String(Files.readAllBytes(Paths.get(path))))
    .when()
    .post("/maps/api/place/add/json")
    .then().log().all().assertThat().statusCode(200).body("scope", equalTo("APP")).header("server", "Apache/2.4.41 (Ubuntu)");

}

```

The key item here is the path of the JSON file and how it would be read. `new String(Files.readAllBytes(Paths.get(path)))` : here Java is reading the file in bytes then its converting the bytes to String for the `body()` method to able to read it.

API token:

ATATT3xFfGF0KMKSBT63rW0sN9bm2oZnH4zX8jZ2cR6sNNnRQLCQpi3QGQZTofmhFy8D-

l1ob\_xC\_sJB7XsbkM4b55BiGde49aBMSM2Zv3laUYJI4ROgNumMsYSov0L6JcGtj7TCG5pjOLgBTI

## ▼ Capturing session information

There are APIs that help us to get the session information. In modern web applications we use basic auth or OAuth however, there are application that use cookies based authentication (Jira [example](#)). These related APIs help us to get the session ID. Now we need that session ID to run our later tests.

Below is sample response from Jira cookie based auth (deprecated):

```

{
  "loginInfo" : {
    "lastFailedLoginTime" : "2013-11-27T09:43:28.839+0000",
    "previousLoginTime" : "2013-12-04T07:54:59.824+0000",
    "loginCount" : 2,
    "failedLoginCount" : 1
  },
  "session" : {
    "name" : "JSESSIONID",
    "value" : "6E3487971234567896704A9EB4AE501F"
  }
}

```

As previously seen we can capture the response and from there extract whatever property value we wanted. However, for sessions there is another approach. We need to use the following class instance:

```

SessionFilter session=new SessionFilter();

```

```

String response=given()
    .header("Content-Type", "application/json")
    .body("...")
    .log().all()
    .filter(session)
    .when()
    .post("/rest/auth/1/session")
    .then().log().all().extract().response().asString();

```

We are passing the session object through the `filter()` method. It will store the session information. This session object then can be used throughout other tests.

The [documentation](#) of `SessionFilter` says:

A session filter can be used record the session id returned from the server as well as automatically apply this session id in subsequent requests



Using `SessionFilter` is ideal when it comes to session handling, over `JsonPath`

## ▼ Passing path parameters

Previously we have seen only using query parameters and resources. Now we would see how to pass path parameters. When reading documentations path parameters are usually indicated by curly braces. Here is an example from [JIRA Add comment API](#):

POST `/rest/api/3/issue/{issueIdOrKey}/comment`

In code it would be passed using `.pathParam()` method:

```
given().pathParam ("key", "10001").body("...")
...
.when()
.post("/rest/api/3/issue/{key}/comment")
```

Here we are inputting the path parameter as a key-value pair. Now when we are passing our resources information in the `post()` method we add the key name in curly braces in the exact location mentioned in the document. So WRT example it is after `/issue/ ...`

## ▼ Sending attachments

We can send attachments through the POST API call. Below is an example of uploading files to Jira.

### Using postman


- Request: `POST`
- URI: [https://domain.atlassian.net/rest/api/3/issue/ISSUE\\_NUMBER/attachments](https://domain.atlassian.net/rest/api/3/issue/ISSUE_NUMBER/attachments)
- Headers:

<input checked="" type="checkbox"/>	Content-Type	application/json
<input checked="" type="checkbox"/>	Authorization	<div></div>
<input checked="" type="checkbox"/>	Accept	application/json
<input checked="" type="checkbox"/>	X-Atlassian-Token	no-check
	Key	Value

- In the body select form data instead of raw-json (what we have been using so far). In the form column the key WRT0 example is "file" and the value is the file itself.
  - When hovering over the key cell there would be an option to select file or text. Select file which would enable the browse option in the value column. Browse and select your file.
- Send the request and wait for the 200 OK response.

Below is the documentation from Jira

The Jira Cloud platform REST API

 <https://developer.atlassian.com/cloud/jira/platform/rest/v3/api-group-issue-attachments/#api-rest-api-3-issue-issueid-or-key-attachments-post>

## Code

To send files we need to use the `multiPart()` method. This method takes the key which in our case is "file" and the path of the file: `new File (path)`. This has to in the `given` section. For all this to work we need to send another header info: `.header("Content-Type", "multipart/form-data")`.

```
given()
  .header("X-Atlassian-Token", "no-check").filter(session).pathParam("key", "10101")
  .header("Content-Type", "multipart/form-data")
  .multiPart("file", new File("jira.txt"))
  .when()
  .post("rest/api/2/issue/{key}/attachments")
  .then()
  .log().all().assertThat().statusCode(200);
```

Whenever we are using multipart that info has to be passed in the header info.

## ▼ Limiting JSON response

WTR0 example of JIRA when receiving a `GET` issue details response Jira sends over 67 different field values. Which might not be needed in most cases, so we can limit the response to have only the items we want to see using the query parameters.

- Request: `GET`
- URI: <https://DOMAIN.atlassian.net/rest/api/2/issue/issueNumber>
- Query parameters:

	KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/>	fields	summary	
<input checked="" type="checkbox"/>	fields	comment	
	Key	Value	Description

The response:

```
{
  "expand": "renderedFields,names,schema,operations,editmeta,changelog,versionedRepresentations",
  "id": "10000",
  "self": "https://DOMAIN.atlassian.net/rest/api/2/issue/10000",
  "key": "issueName",
  "fields": {
    "summary": "Sample task",
    "comment": {
      "comments": [],
      "self": "https://DOMAIN.atlassian.net/rest/api/2/issue/10000/comment",
      "maxResults": 0,
    }
  }
}
```



```

        "total": 0,
        "startAt": 0
    }
}
}

```

## ▼ Skipping HTTPS validation

`relaxedHTTPSValidation()` is used to bypass the HTTPS validation in case Rest Assured is unable to validate the certificate it generates. This can be useful when testing APIs that use self-signed certificates or other types of certificates that Rest Assured is unable to validate by default.

```
given().relaxedHTTPSValidation()
```

## ▼ OAuth

OAuth 2.0 is an **open authorization framework** that allows applications to access user data without requiring the user to disclose their login credentials. It is commonly used as a way for users to grant access to third-party applications such as mobile apps or websites, without giving them direct access to their account credentials.

Some of the most famous applications that use OAuth 2.0 include Google, Facebook, Twitter, and LinkedIn.

OAuth 2.0 defines several grant types, including:

- Authorization Code Grant
- Implicit Grant
- Resource Owner Password Credentials Grant
- Client Credentials Grant

Each grant type is used for a specific purpose and has its own set of rules for how access tokens are obtained and used.



Benefits of third party authentication like google is that the server doesn't have to store sensitive user information. It would be handled by google. That doesn't mean the password is shared. It's the basic information that is passed.

## ▼ Backend

- **Client** → who we are serving: BookMyShow  
The service has to apply to google to use their services. If everything is properly processed then google would create a client ID.
- **ClientID** → Uniquely identifies our target server (BookMyShow)
- **Client Secret ID** → Similar to a password
- **Resource Owner** → In OAuth, the resource owner is the user who owns the protected resource (such as an account on a website) and who grants permission to a third-party application to access that resource on their behalf. The third-party application can then access the resource without the resource owner having to reveal their login credentials to the application.
- **Resource/Authorization server** → Who is providing the service of authentication? It's google so it has all the information about the user.

Below is the URL that is shown when log in with google is clicked:

[https://accounts.google.com/o/oauth2/v2/auth/identifier?gsiwebsdk=3&client\\_id=700206021005-as1l679sch207mp70msgihma1krf3k9g.apps.googleusercontent.com&scope=email%20profile&redirect\\_uri=storagerelay%3A%2F%2Fhttps%2Fwww.udemy.com%3Fid%3Dauth26285&prompt=consent&access\\_type=offline](https://accounts.google.com/o/oauth2/v2/auth/identifier?gsiwebsdk=3&client_id=700206021005-as1l679sch207mp70msgihma1krf3k9g.apps.googleusercontent.com&scope=email%20profile&redirect_uri=storagerelay%3A%2F%2Fhttps%2Fwww.udemy.com%3Fid%3Dauth26285&prompt=consent&access_type=offline)

## Steps performed

1. Users clicks on sign in with google option.
2. Enters his/her details.
3. Google would send a authorization code.
4. Application (BookMyShow) will use this code to hit google resources server (the resource server has info about the user whereas the authorization server sends the code/OTP type information after successful login).
5. Google would verify the code and the corresponding users existence.
6. Then the information such as access token, first name, last name, email id etc is given. (this depends on how the scope is designed).
  - a. The access token is given to the application and it is stored in the browser. This token is used every time the user performs an action and is used by the application to identify the user. if the token expires then it would ask you to login again.

## ▼ Authorization Code Grant Type

The best example of OAuth2.0 this is seen in our day to day lives. We use login using google, facebook in different websites such as udemy, IMDB, Uber etc. And these techniques use the Authorization grant type.

Here is an example we would be using:

[https://accounts.google.com/o/oauth2/v2/auth?scope=https://www.googleapis.com/auth/userinfo.email&auth\\_url=https://accounts.google.com/o/oauth2/v2/auth&client\\_id=69p0m7ent2hk7suguv4vq22hjcfc43pj.apps.googleusercontent.com&response\\_type=code&redirect\\_uri=https://rahulshettyacademy.com](https://accounts.google.com/o/oauth2/v2/auth?scope=https://www.googleapis.com/auth/userinfo.email&auth_url=https://accounts.google.com/o/oauth2/v2/auth&client_id=69p0m7ent2hk7suguv4vq22hjcfc43pj.apps.googleusercontent.com&response_type=code&redirect_uri=https://rahulshettyacademy.com)

To run an OAuth request we require some mandatory parameters:

- **scope:** Asking google to give me these these information of the user. WRTTO example we are asking for the users email:  
<https://www.googleapis.com/auth/userinfo.email>

The scope parameter is used to specify the specific permissions that an application is requesting from the user.

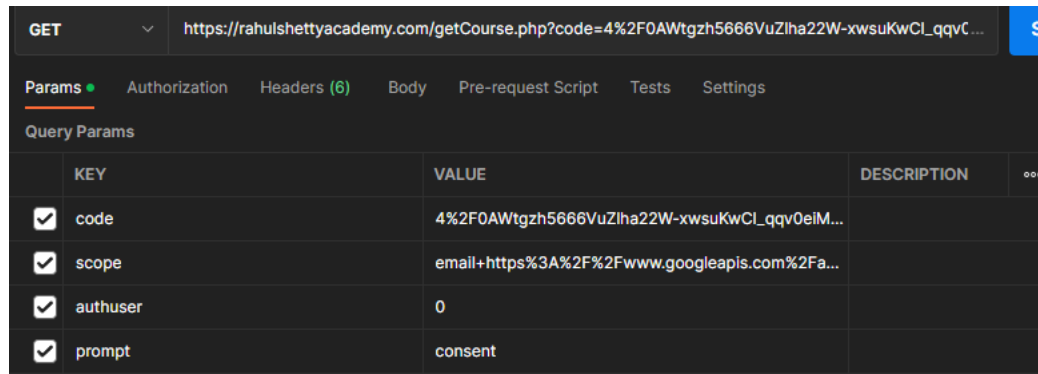
- **auth\_url:** googles authorization url → <https://accounts.google.com/o/oauth2/v2/auth>
- **client\_id:** Applications ID → [client\\_id=69p0m7ent2hk7suguv4vq22hjcfc43pj.apps.googleusercontent.com](https://accounts.google.com/o/oauth2/v2/auth?scope=https://www.googleapis.com/auth/userinfo.email&auth_url=https://accounts.google.com/o/oauth2/v2/auth&client_id=69p0m7ent2hk7suguv4vq22hjcfc43pj.apps.googleusercontent.com&response_type=code&redirect_uri=https://rahulshettyacademy.com)
- **response\_type:** What response we are expecting back from google. WRTTO example its the authorization code →  
`response_type=code`
- **redirect\_url:** Where google has to send you back after authorization →  
`redirect_uri=https://rahulshettyacademy.com/getCourse.php`
- **state [optional]:** Mainly used for security purpose.

In the OAuth process, the `state` parameter is used to protect against CSRF (Cross-Site Request Forgery) attacks. It is a randomly generated value that is passed to the authorization server along with the other parameters. The authorization server returns the same value in the

response, and the client application can then verify that the value matches the original value to ensure that the response is not the result of a malicious attack.

After sending the request I get the following response:

[https://rahulshettyacademy.com/getCourse.php?code=4%2F0AWtgzh5666VuZlha22W-xwsuKwCl\\_qqv0eiMjHiNTbtFMlanfK9fR8zLDbpygyILEuFh3g&scope=email+https%3A%2F%2Fwww.googleapis.com%2Fauth](https://rahulshettyacademy.com/getCourse.php?code=4%2F0AWtgzh5666VuZlha22W-xwsuKwCl_qqv0eiMjHiNTbtFMlanfK9fR8zLDbpygyILEuFh3g&scope=email+https%3A%2F%2Fwww.googleapis.com%2Fauth)  
pasting the link in postman would automatically parse it



	KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/>	code	4%2F0AWtgzh5666VuZlha22W-xwsuKwCl_qqv0eiM...	
<input checked="" type="checkbox"/>	scope	email+https%3A%2F%2Fwww.googleapis.com%2Fa...	
<input checked="" type="checkbox"/>	authuser	0	
<input checked="" type="checkbox"/>	prompt	consent	

Here we need the access code:

[code=4%2F0AWtgzh5666VuZlha22W-xwsuKwCl\\_qqv0eiMjHiNTbtFMlanfK9fR8zLDbpygyILEuFh3g](#)

Now using the above code we need to hit the the application authorization server to validate our code:

[https://www.googleapis.com/oauth2/v4/token?code=THE\\_CODE\\_HERE&client\\_id=692183103107-p0m7ent2hk7suguv4vq22hjcfr43pj.apps.googleusercontent.com&client\\_secret=erZOWM9g3UtwNRj340YYaK\\_W&redirect\\_](https://www.googleapis.com/oauth2/v4/token?code=THE_CODE_HERE&client_id=692183103107-p0m7ent2hk7suguv4vq22hjcfr43pj.apps.googleusercontent.com&client_secret=erZOWM9g3UtwNRj340YYaK_W&redirect_)

This time around the following information is required:

- The code
- Client ID
- Client secret key
- Redirect URI
- Grant Type (for our case its authorization code grant type)

The above details would be provided by the developer.

Sending a **POST** request: [https://www.googleapis.com/oauth2/v4/token?](https://www.googleapis.com/oauth2/v4/token?code=4%2F0AWtgzh5G8hLIPQKkxXelvBB8NF4mxJPC1bnC-8B5gPBjJpA6wkdf9ZQZGJTSWj4-Aox7LQ&client_id=692183103107-p0m7ent2hk7suguv4vq22hjcfr43pj.apps.googleusercontent.com&client_secret=erZOWM9g3UtwNRj340YYaK_W&redirect_)

[code=4%2F0AWtgzh5G8hLIPQKkxXelvBB8NF4mxJPC1bnC-8B5gPBjJpA6wkdf9ZQZGJTSWj4-](#)

[Aox7LQ&client\\_id=692183103107-](#)

[p0m7ent2hk7suguv4vq22hjcfr43pj.apps.googleusercontent.com&client\\_secret=erZOWM9g3UtwNRj340YYaK\\_W&redirect\\_](#)  
→ to get the access token

This would give us the following:

```
{
  "access_token": "XXXX-YE-JePxUc3PpP5-XXXX-VQM-XXXXX",
  "expires_in": 3599,
  "scope": "https://www.googleapis.com/auth/userinfo.email openid",
  "token_type": "Bearer",
```

```
}
  "id_token": "XXXXX.XXXX.XXX-XXXX-2HVoi0QC-XXXX-kg"
}
```

Now lets use this access token to get the final response containing our requested info (from the resource server)

**GET** request: [https://rahulshettyacademy.com/getCourse.php?access\\_token=TOKEN\\_VALUE](https://rahulshettyacademy.com/getCourse.php?access_token=TOKEN_VALUE)

```
{
  "instructor": "RahulShetty",
  "url": "rahulshettyacademy.com",
  "services": "projectSupport",
  "expertise": "Automation",
  "courses": {
    "webAutomation": [
      {
        "courseTitle": "Selenium Webdriver Java",
        "price": "50"
      },
      {
        "courseTitle": "Cypress",
        "price": "40"
      },
      {
        "courseTitle": "Protractor",
        "price": "40"
      }
    ],
    "api": [
      {
        "courseTitle": "Rest Assured Automation using Java",
        "price": "50"
      },
      {
        "courseTitle": "SoapUI Webservices testing",
        "price": "40"
      }
    ],
    "mobile": [
      {
        "courseTitle": "Appium-Mobile Automation using Java",
        "price": "50"
      }
    ]
  },
  "linkedIn": "https://www.linkedin.com/in/rahul-shetty-trainer/"
}
```

## Recap

## OAuth 2.0 Contract Details:

GrantType	Authorization code
redirect URL/Callback URL	https://rahulshettyacademy.com/getCourse.php
Authorization server url	https://accounts.google.com/o/oauth2/v2/auth
Access token url	https://www.googleapis.com/oauth2/v4/token
Client ID	692183103107-p0m7ent2hk7suguv4vq22hjcfr43pj.apps.googleusercontent.com
Client Secret	erZOWM9g3UtwNRj340YYaK W
Scope	https://www.googleapis.com/auth/userinfo.email
State	Any random string
How to pass oauth in request	Headers

### Mandatory fields for GetAuthorization Code Request ;

**End Point:** Authorization server URL

**Query Params:** Scope, Auth\_url, client\_id, response\_type, redirect\_uri

**output:** Code

### Mandatory fields for GetAccessToken Request:

**End point:** Access token URL

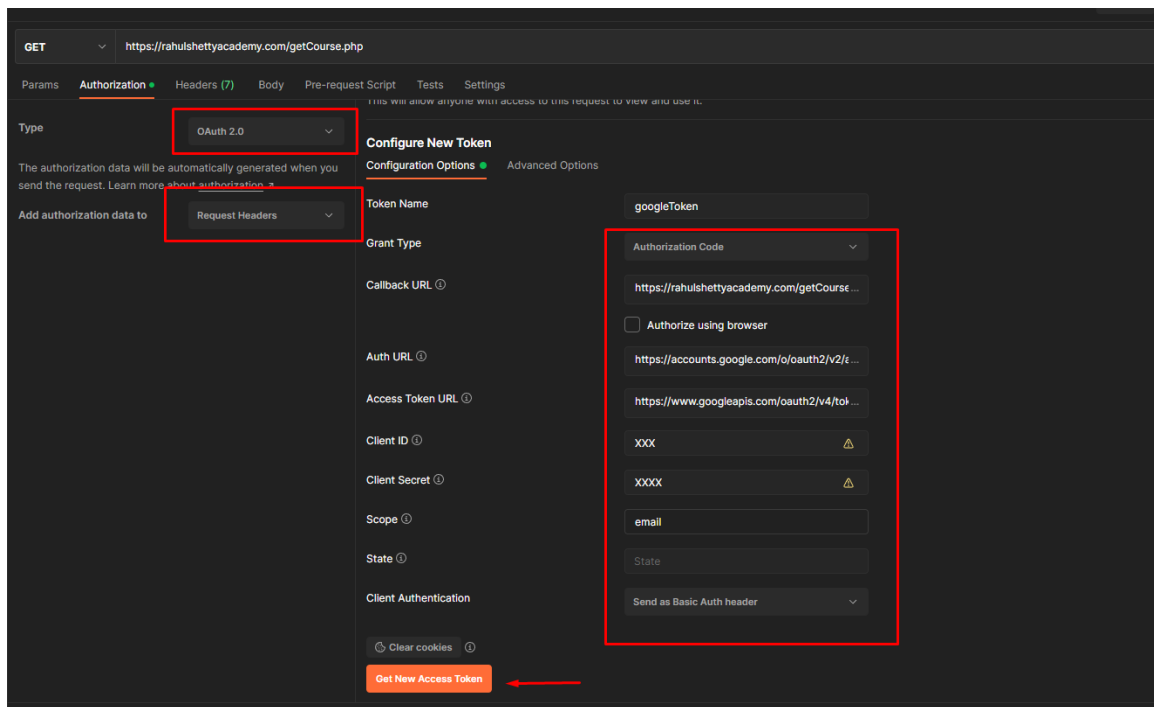
**Query Params:** Code, client\_id, client secret, redirect\_uri, grant\_type

**Output:** Access token

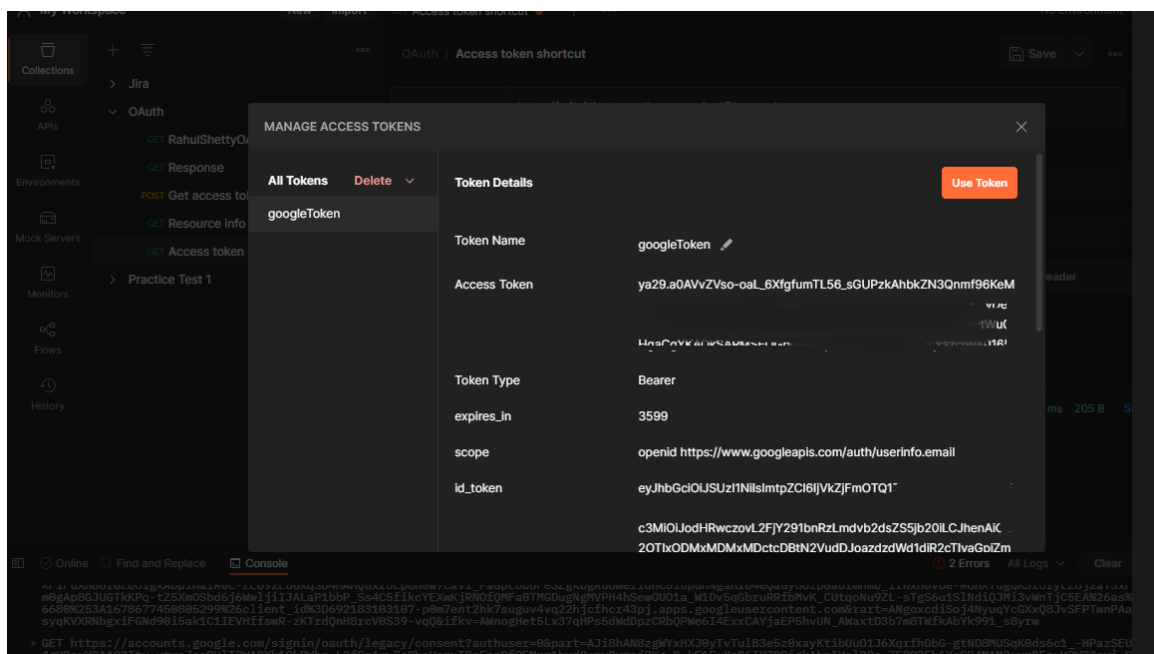
## ▼ Postman shortcut to get Access Token

There is an option in Postman to run the whole process that we did in one go. However, the problem is that that can only be used in Postman and when we are automating such features there isn't any shortcut process.

Add the related information in Postman



Clicking on Get New Access Token would take you to the google login page and after successful login a token would be generated



## ▼ Automating the process

Getting the code we need to login with our google account. That can not be done with Rest Assured. For that we need manually login with google credentials and get the URL.

```
String urlForCode = "https://raahulshettyacademy.com/getCourse.php?code=4%XXX&scope=email+https%3A%2F%2Fwww.googleapis.com%2Fauth";

String array1[] = urlForCode.split("code=");
String array2[] = array1[1].split("&scope");
String code = array2[0];
```

The URL gives us the code and using java methods we extract the code information. Then we need to pass the code to the next phase which gives us the access token

```
String accessTokenResponse =
given().urlEncodingEnabled(false)
    .log().all()
    .queryParam("code", code)
    .queryParam("client_id", "692183183187-p0m7ent2hk7suguv4vq22hjcfhcr43pj.apps.googleusercontent.com")
    .queryParam("client_secret", "erZOWM9g3UtwNRj340YYak_w")
    .queryParam("redirect_uri", "https://raahulshettyacademy.com/getCourse.php")
    .queryParam("grant_type", "authorization_code")
.when()
    .log().all()
    .post("https://www.googleapis.com/oauth2/v4/token").asString();

JsonPath js = new JsonPath(accessTokenResponse);
String accessTokenValue = js.getString("access_token");
```

Then we use the token value to get our results.

We can skip the `then()` and its related extract and string methods to get the response and directly write the following:

```
String response =
given()
    .log().all()
    .queryParams("access_token", accessTokenValue)
.when()
    .get("https://rahulshettyacademy.com/getCourse.php").asString();

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

This would give the following response:

```
{ "instructor": "RahulShetty", "url": "rahulshettycademy.com", "services": "projectSupport", "expertise": "Automation", "courses": [{"price": "40"}, {"courseTitle": "Protractor", "price": "40"}], "api": [{"courseTitle": "Rest Assured Automation using Java", "price": "50"}], {"courseTitle": "SoapUI Webservices testing", "price": "40"}], "mobile": [{"courseTitle": "Appium-Mobile Automation using Java", "price": "50"}], "linkedIn": "https://www.linkedin.com/in/rahul-shetty-trainer/"}
```

Process finished with exit code 0

This process is definitely annoying to login before running tests so it's better to ask the development team to increase the timeout of the token.

## ▼ Client Credentials

The Client Credentials grant type is used when the client application wants to access its own resources, rather than a user's resources. This is typically used for machine-to-machine authentication, where the client application needs to communicate with a server or API without user interaction. To use this grant type, the client application must know its own client ID and client secret, which are used to authenticate with the authorization server and obtain an access token. The access token can then be used to make requests to the protected resources.



In client credentials there is no Human (Resource Owner). There is no 3rd party here.

Here as there is no human resource needed there is no authorization code needed. As mentioned above it is accessing its own resources so only the client ID and secret is needed. In Postman the following information is needed:

GET NEW ACCESS TOKEN

Token Name: Token Name

Grant Type: Client Credentials

Access Token URL: <https://www.googleapis.com/oauth2/v4/token>

Client ID: 692183103107-p0m7ent2hk7suguv4vq22hjcfr43pj.apps.googleusercontent.com

Client Secret: erZOWM9g3UtwNRj340YYaK\_W

Scope: <https://www.googleapis.com/auth/userinfo.email>

Client Authentication: Send as Basic Auth header

Request Token

So the automation steps are also similar except we don't need the code portion.

## ▼ Serialization and deserialization

### ▼ POJO

In Java, a POJO (Plain Old Java Object) class is a simple object class that does not extend or implement any specialized classes or interfaces. It is used to encapsulate data and provide getter and setter methods for accessing the data. POJO classes are commonly used for serialization and deserialization of JSON and XML data in RESTful web services. Below is an example of a POJO class:



```

public class Message {
    private String message;

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }
}

```

The best thing about using POJO classes is that when we are sending the payload we defining the key and the value but when using POJO classes we can just send the object and Rest Assured would automatically extract the key value pairs.

```

Message m = new Message();
m.setMessage("Hi");

```

In our code:

```

given().body(m)

```

This whole process is serialization.

## ▼ De-serialization

In this process we need get the values. Rest Assured is smart enough to map the values to the created object and the user then can use the get methods to access those values.



To achieve de-serialization we need some external libraries: Jackson, Jackson2m GSON or Johnzon in the classpath and for XML we need JAXB.

### Dependencies

```

<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-databind</artifactId>
  <version>2.14.2</version>
</dependency>
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-core</artifactId>
  <version>2.14.2</version>
</dependency>
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-annotations</artifactId>
  <version>2.14.2</version>
</dependency>
<dependency>
  <groupId>com.google.code.gson</groupId>
  <artifactId>gson</artifactId>
  <version>2.10.1</version>
</dependency>

```

We are going to use the previous `getCourse` payload from our OAuth example.

```

{
  "instructor": "RahulShetty",
  "url": "rahulshettyacademy.com",
  "services": "projectSupport",
  "expertise": "Automation",
  "courses": {
    "webAutomation": [
      {
        "courseTitle": "Selenium Webdriver Java",
        "price": "50"
      },
      {
        "courseTitle": "Cypress",
        "price": "40"
      },
      {
        "courseTitle": "Protractor",
        "price": "40"
      }
    ],
    "api": [
      {
        "courseTitle": "Rest Assured Automation using Java",
        "price": "50"
      },
      {
        "courseTitle": "SoapUI Webservices testing",
        "price": "40"
      }
    ],
    "mobile": [
      {
        "courseTitle": "Appium-Mobile Automation using Java",
        "price": "50"
      }
    ]
  },
  "linkedIn": "https://www.linkedin.com/in/rahul-shetty-trainer/"
}

```

Here there are 6 objects and among them `courses` in a nested JSON which has 3 objects/properties of its own. So we create getters and setter for 6 of them. However, to handle the nested json (courses) we need another set of getters and setters. To handle this we would write another class called `courses` that handles these properties.

The class to get all 6 properties

```

package Serialization_deserialization;

public class GetCourse {

    private String url;
    private String services;
    private String expertise;

    private Courses courses;
    private String instructor;
    private String linkedIn;

    public String getUrl() {
        return url;
    }

    public void setUrl(String url) {
        this.url = url;
    }

    public String getServices() {
        return services;
    }
}

```

```

    public void setServices(String services) {
        this.services = services;
    }

    public String getExpertise() {
        return expertise;
    }

    public void setExpertise(String expertise) {
        this.expertise = expertise;
    }

    public Courses getCourses() {
        return courses;
    }

    public void setCourses(Courses courses) {
        this.courses = courses;
    }

    public String getInstructor() {
        return instructor;
    }

    public void setInstructor(String instructor) {
        this.instructor = instructor;
    }

    public String getLinkedIn() {
        return linkedIn;
    }

    public void setLinkedIn(String linkedIn) {
        this.linkedIn = linkedIn;
    }
}

```

The class to handle the courses

```

package Serialization_deserialization;

public class Courses {
    private String webAutomation;
    private String api;
    private String mobile;

    public String getWebAutomation() {
        return webAutomation;
    }

    public void setWebAutomation(String webAutomation) {
        this.webAutomation = webAutomation;
    }

    public String getApi() {
        return api;
    }

    public void setApi(String api) {
        this.api = api;
    }

    public String getMobile() {
        return mobile;
    }

    public void setMobile(String mobile) {
        this.mobile = mobile;
    }
}

```

See how we have connected them by changing the type of courses variable to Courses object.

Again we see that the `webAutomation`, `api` and `mobile` properties also has a mini `json` inside. So the same process has to followed.

In the courses class:

```
private WebAutomation webAutomation;  
private Api api;  
private Mobile mobile;
```

If look at what the `webAutomation` sub json return we would see that its an array. An array containing multiple smaller jsons. So to handle this we need to make the object a `List`. A List of `WebAutomation object`.

```
private List<WebAutomation> webAutomation;  
  
public List<WebAutomation> getWebAutomation() {  
    return webAutomation;  
}  
  
public void setWebAutomation(List<WebAutomation> webAutomation) {  
    this.webAutomation = webAutomation;  
}
```

Same for API and mobile class.

```
package Deserialization;  
  
import java.util.List;  
  
public class Courses {  
    private List<WebAutomation> webAutomation;  
    private List<Api> api;  
    private List<Mobile> mobile;  
  
    public List<WebAutomation> getWebAutomation() {  
        return webAutomation;  
    }  
  
    public void setWebAutomation(List<WebAutomation> webAutomation) {  
        this.webAutomation = webAutomation;  
    }  
  
    public List<Api> getApi() {  
        return api;  
    }  
  
    public void setApi(List<Api> api) {  
        this.api = api;  
    }  
  
    public List<Mobile> getMobile() {  
        return mobile;  
    }  
  
    public void setMobile(List<Mobile> mobile) {  
        this.mobile = mobile;  
    }  
}
```

As all the structure has been created we can now use these classes: `as(GetCourse.class);`

```
GetCourse gc =  
given()
```

```

    .queryParams("access_token", accessTokenValue)
    .expect().defaultParser(Parser.JSON)
    .when()
    .get("https://rahulshettyacademy.com/getCourse.php").as(GetCourse.class);

```

Here we are using `.defaultParser(Parser.JSON)` this would ensure that we are reading the response as a JSON.



if the content type in the response header mentions Application/Json then we dont need the default parser.



If the POJO classes aren't created correctly the process would not work.

Accessing nested properties like a course name in the API sub JSON:

```
String apiCourseTitleOne = gc.getCourses().getApi().get(1).getCourseTitle();
```

Iterating through all the courses:

```

List<Api> apiObject = gc.getCourses().getApi();
for (int i=0; i <apiObject.size(); i++){
    String courseTitle = apiObject.get(i).getCourseTitle();
    if(courseTitle.contains("Soap")){

        System.out.println(courseTitle + " - " + apiObject.get(i).getPrice());
    }
}

List<WebAutomation> webAutomationObject = gc.getCourses().getWebAutomation();
for (int i=0; i <apiObject.size(); i++){
    String courseTitle = apiObject.get(i).getCourseTitle();
    System.out.println(courseTitle + " - " + apiObject.get(i).getPrice());
}

```

The results:

```

LinkedIn: https://www.linkedin.com/in/rahul-shetty-trainer/
Instructor name: RahulShetty
Expertise: Automation
SoapUI Webservices testing - 40
Rest Assured Automation using Java - 50
SoapUI Webservices testing - 40

```

<https://s3-us-west-2.amazonaws.com/secure.notion-static.com/e99a5d52-743e-49a3-9545-74df58b469f1/Api.java>

<https://s3-us-west-2.amazonaws.com/secure.notion-static.com/6b470e94-2e8a-43cf-8d4c-89ab39b62d2b/Courses.java>

<https://s3-us-west-2.amazonaws.com/secure.notion-static.com/0586245e-e216-4dd0-a0f8-732650b97519/GetCourse.java>

<https://s3-us-west-2.amazonaws.com/secure.notion-static.com/b41ab1f6-0749-4cad-8a11-24c409078e4b/Message.java>

<https://s3-us-west-2.amazonaws.com/secure.notion-static.com/95d28284-501a-4d66-83c1-a556489938a3/Mobile.java>

[https://s3-us-west-2.amazonaws.com/secure.notion-static.com/ccbeeca6-3335-4e94-876e-cd97763738b9/Response\\_used.json](https://s3-us-west-2.amazonaws.com/secure.notion-static.com/ccbeeca6-3335-4e94-876e-cd97763738b9/Response_used.json)

<https://s3-us-west-2.amazonaws.com/secure.notion-static.com/2b2c5c4d-168d-4a01-ae83-284438291475/Test.java>

<https://s3-us-west-2.amazonaws.com/secure.notion-static.com/fc0c2186-7e28-4239-8779-99df0999ecd6/WebAutomation.java>

## ▼ Serialization

Serialization is a Rest Assured is a process of converting a java object into request body/payload.

In de-serialization we have seen extracting information from the JSON body. In serialization we creating this JSON body.

The JSON body we are creating:

```
{
  "location": {
    "lat": -38.383494,
    "lng": 33.427362
  },
  "accuracy": 50,
  "name": "Frontline house",
  "phone_number": "(+91) 983 893 3937",
  "address": "29, side layout, cohen 09",
  "types": [
    "shoe park",
    "shop"
  ],
  "website": "http://google.com",
  "language": "French-IN"
}
```

Here there are 8 attributes and location is a nested JSON. So we would have a class that has getter and setter for all these attributes and location would have its own class with its own getter and setters.

Add place class

```

package Serialization;

import java.util.List;

public class AddPlace {
    private Location location;
    private int accuracy;
    private String name;
    private String phone_number;
    private String address;
    private List<String> types;
    private String website;
    private String language;

    public Location getLocation() {
        return location;
    }

    public void setLocation(Location location) {
        this.location = location;
    }

    public int getAccuracy() {
        return accuracy;
    }

    public void setAccuracy(int accuracy) {
        this.accuracy = accuracy;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getPhone_number() {
        return phone_number;
    }

    public void setPhone_number(String phone_number) {
        this.phone_number = phone_number;
    }

    public String getAddress() {
        return address;
    }

    public void setAddress(String address) {
        this.address = address;
    }

    public List<String> getTypes() {
        return types;
    }

    public void setTypes(List<String> types) {
        this.types = types;
    }

    public String getWebsite() {
        return website;
    }

    public void setWebsite(String website) {
        this.website = website;
    }

    public String getLanguage() {
        return language;
    }

    public void setLanguage(String language) {
        this.language = language;
    }
}

```

## Location class

```
package Serialization;

public class Location {
    private double lat;
    private double lng;

    public double getLat() {
        return lat;
    }

    public void setLat(double lat) {
        this.lat = lat;
    }

    public double getLng() {
        return lng;
    }

    public void setLng(double lng) {
        this.lng = lng;
    }
}
```

## Main test class

```
package Serialization;
import com.beust.ah.A;
import io.restassured.RestAssured;
import io.restassured.response.Response;

import java.util.ArrayList;
import java.util.List;

import static io.restassured.RestAssured.*;
import static org.hamcrest.Matchers.*;

public class test {
    public static void main(String[] args) {
        // Creating the body
        AddPlace ap = new AddPlace();
        ap.setAccuracy(50);
        ap.setAddress("Dhaka");
        ap.setLanguage("Bangla");
        ap.setName("Maruf");
        ap.setPhone_number("12345");
        ap.setWebsite("www.google.com");
        List<String> type = new ArrayList<String>();
        type.add("Phone");
        type.add("laptop");
        ap.setTypes(type);

        Location l = new Location();
        l.setLat(-38.3834);
        l.setLng(33.4273);
        ap.setLocation(l);

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

        String res = given()
            .queryParams("key", "qaclick123")
            .body(ap)
            .when()
            .post("/maps/api/place/add/json")
            .then()
            .assertThat().statusCode(200).extract().response().asString();

        System.out.println(res);
    }
}
```

## Result



```
{"status": "OK", "place_id": "13f0579b657858990f258383c104edd5", "scope": "APP", "reference": "97bc5b042987340a172c8bfee4be08c497bc5b042987"}
```

## ▼ Spec Builders

With respect to our previous location API examples we can see that there are few things that are always common such as the content type, base URI, key values, etc. Now its not ideal to write these for every class (add, update, delete, etc.). To handle with we use spec builder.

Below is the piece of code that needs to be written that specifies the base URI, query parameter and the content type.

```
RequestSpecification requestSpec = new RequestSpecBuilder().setContentType(ContentType.JSON)
    .setBaseUri("https://rahulshettyacademy.com")
    .addQueryParam("key", "qaclick123")
    .setContentType(ContentType.JSON)
    .build();
```

Similarly we have some common code for handling response such as the status code and content type. The build method would create a request spec which can be used across all cases.

```
ResponseSpecification responseSpec = new ResponseSpecBuilder()
    .expectStatusCode(200)
    .expectContentType(ContentType.JSON).build();
```

Then we create a variable specifically to hold for the request:

```
RequestSpecification res = given().spec(requestSpec).body(ap);
```

Then pass these variable through the full rest assured request:

```
String responseJson = res.when()
    .post("/maps/api/place/add/json")
    .then().spec(responseSpec).extract().response().asString();
```

As we can we are separating out all the reusable elements.

## ▼ End to End Example of a Ecommerce website

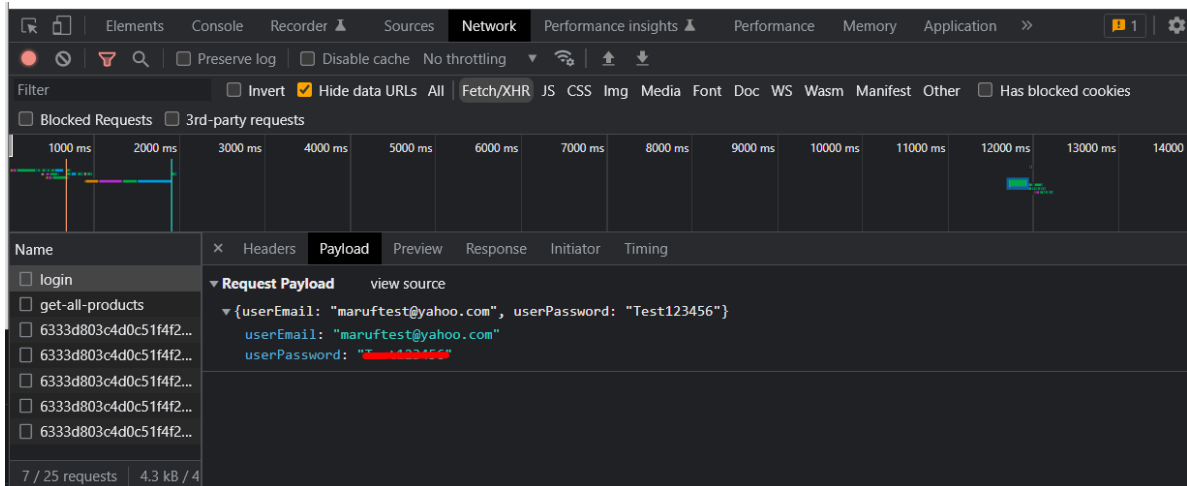
To see what APIs our website is hitting we can open the network tab (F12) → filter → Fetch/XHR. This would show all the APIs that are being used for that page. Using that we can create our own documentation and automation.

The example website we are going to use for this section: <https://www.rahulshettyacademy.com/>

## Manual

### ▼ Login

<https://rahulshettyacademy.com/client>



From the network tab we were able to identify the URI and payload:

**POST :** <https://rahulshettyacademy.com/api/ecom/auth/login>

**Body :**

```
{
  "userEmail": "maruftest@yahoo.com",
  "userPassword": "Test"
}
```

**Response :**

```
{
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJfawQiOiI2MzMzZDgwM2M0ZDBjNTFmNGYyZWU0OGQlLCJ1c2VyRW1hawwiOiJtYXJ1ZnRlc3RAeWVob",
  "userId": "6333d803c4d0c51f4f2ee48d",
  "message": "Login Successfully"
}
```

## ▼ Create a product

**POST :** <https://rahulshettyacademy.com/api/ecom/product/add-product>

To add products now we need to add our body containing all the product information. For this API we would require 8 different key values.

In this example we are going to use `form-data` instead of the `raw` format.

Body → form-data → Bulk edit → paste the below content → key-value edit

```
productName:qwerty
productAddedBy:{{userId}}
productCategory:fashion
productSubCategory:shirts
productPrice:11500
productDescription:Addias Originals
productFor:women
```

POST ▼ https://rahulshettyacademy.com/api/ecom/product/add-product Send ▼

Params Authorization Headers (8) **Body** ● Pre-request Script Tests Settings Cookies

☐ none ☒ form-data ☐ x-www-form-urlencoded ☐ raw ☐ binary ☐ GraphQL

	Key	Value	Description	...	Bulk Edit
<input checked="" type="checkbox"/>	productName	qwerty			
<input checked="" type="checkbox"/>	productAddedBy	{{userId}}			
<input checked="" type="checkbox"/>	productCategory	fashion			
<input checked="" type="checkbox"/>	productSubCategory	shirts			
<input checked="" type="checkbox"/>	productPrice	11500			
<input checked="" type="checkbox"/>	productDescription	Addias Originals			
<input checked="" type="checkbox"/>	productFor	women			
	Key	Value	Description		

As adding product can only be done by users we need to add the token value in the header information.

POST ▼ https://rahulshettyacademy.com/api/ecom/product/add-product

Params Authorization **Headers (9)** ● Body ● Pre-request Script Tests Settings

<input checked="" type="checkbox"/>	Postman-Token	<calculated when request is sent>
<input checked="" type="checkbox"/>	Content-Type	multipart/form-data; boundary=<calculated ...>
<input checked="" type="checkbox"/>	Content-Length	<calculated when request is sent>
<input checked="" type="checkbox"/>	Host	<calculated when request is sent>
<input checked="" type="checkbox"/>	User-Agent	PostmanRuntime/7.31.3
<input checked="" type="checkbox"/>	Accept	*/*
<input checked="" type="checkbox"/>	Accept-Encoding	gzip, deflate, br
<input checked="" type="checkbox"/>	Connection	keep-alive
<input checked="" type="checkbox"/>	Authorization	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJf...

To send attachments (for the product):

Go to form data → add a new key → in the key cell hover to change the option from text to file → browse for the file and select.

⚠ postman might show a warning here. We can ignore that or we can go to setting and allow files from other directories.

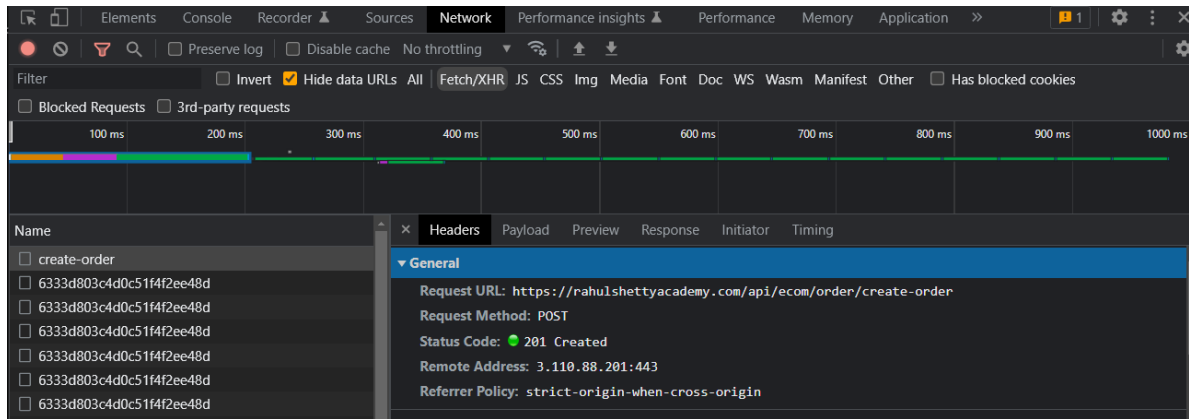
📢 Sending file can only be done using form data!

Response:

```
{
  "productId": "6420d6a9568c3e9fb13ee51b",
}
```

```
{
  "message": "Product Added Successfully"
}
```

## ▼ Order



POST: <https://rahulshettyacademy.com/api/ecom/order/create-order>

Body:

Product ID is the ID we got from add product.

```
{
  "orders": [
    {
      "country": "India",
      "productOrderedId": "6420d6a9568c3e9fb13ee51b"
    }
  ]
}
```

Again as this requires a user to be logged in we again need to put the authorization token.

Response:

```
{
  "orders": [
    "6420d975568c3e9fb13ee5d8"
  ],
  "productOrderId": [
    "6262e990e26b7e1a10e89bfa"
  ],
  "message": "Order Placed Successfully"
}
```

## ▼ Delete

Now if we do a bunch of tests like these it might clutter up our page. So we need to do some clean up by running a delete request.

DELETE: <https://rahulshettyacademy.com/api/ecom/product/delete-product/> PRODUCT\_ID

Again we need to add the authorization code.

Response:

```
{
  "message": "Product Deleted Successfully"
}
```

## ▼ Order details

To get the order we placed before:

GET: [https://rahulshettyacademy.com/api/ecom/order/get-orders-details?id=ORDER\\_ID\\_QUERY\\_PARAM](https://rahulshettyacademy.com/api/ecom/order/get-orders-details?id=ORDER_ID_QUERY_PARAM)

We need authorization token here too.

Response:

```
{
  "data": {
    "_id": "6420d9b9568c3e9fb13ee5e8",
    "orderId": "6333d803c4d0c51f4f2ee48d",
    "orderBy": "maruftest@yahoo.com",
    "productOrderedId": "6420d6a9568c3e9fb13ee51b",
    "productName": "qwerty",
    "country": "India",
    "productDescription": "Camera",
    "productImage": "https://rahulshettyacademy.com/api/ecom/uploads/productImage_1679873705767.jpg",
    "orderPrice": "1000",
    "__v": 0
  },
  "message": "Orders fetched for customer Successfully"
}
```

## Automated

With the concepts we have learn from before we are going to automate this. The only thing we haven't learned previously is how to deal with form data.

The form data needs to sent as a `param` value.

```
RequestSpecification addProductReq = given().spec(addProductBaseReq)
    .param("productName", "Laptop")
```

To send attachments:

```
.multiPart("productImage", new File("C:\\Users\\Anik\\Desktop\\camera.jpg"));
```

To send `path parameters` (for DELETE call)

```
RequestSpecification deleteReq = given().spec(deleteBaseReq).pathParam("productId",newlyCreatedProduct.getProductId() );
```

### Bypassing SSL certification:

There are some cases where the API request might fail for SSL certification. To handle such cases we use `relaxedHTTPSValidation` which is a method in Rest Assured that disables SSL validation, which allows the user to send requests to servers with self-signed SSL certificates or those that are not trusted by the JVM. It is not recommended for production use, but can be helpful during development and testing.

## ▼ Full code

```
package Ecommerce_test;

import io.restassured.builder.RequestSpecBuilder;
import io.restassured.http.ContentType;
import io.restassured.path.json.JsonPath;
import io.restassured.specification.RequestSpecification;
import org.testng.Assert;

import java.io.File;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

import static io.restassured.RestAssured.given;

public class ECommerceAPITest {
    public static void main(String[] args) {

        RequestSpecification baseLoginReq = new RequestSpecBuilder()
            .setBaseUri("https://rahulshettyacademy.com")
            .setContentType(ContentType.JSON).build();

        // Setting username and password
        loginReq lr = new loginReq();
        lr.setUserEmail("maruftest@yahoo.com");
        lr.setUserPassword("Test");

        // Making the request
        RequestSpecification reqlogin = given().relaxedHTTPSValidation().log().all().spec(baseLoginReq).body(lr);
        loginResponse loginResponse = reqlogin.when().log().all().post("/api/ecom/auth/login").then().extract().response().as(LoginR

        System.out.println("The token is: " + loginResponse.getToken());

        // Create product
        // ----- FORM PARAMETERS -----
        RequestSpecification addProductBaseReq = new RequestSpecBuilder()
            .setBaseUri("https://rahulshettyacademy.com")
            .addHeader("Authorization", loginResponse.getToken()).build();

        RequestSpecification addProductReq = given().spec(addProductBaseReq)
            .param("productName", "Laptop")
            .param("productAddedBy", loginResponse.getUserId())
            .param("productCategory", "Tech")
            .param("productSubCategory", "Portable")
            .param("productPrice", "6969")
            .param("productDescription", "HP")
            .param("productFor", "Students")
            .multiPart("productImage", new File("C:\\Users\\Anik\\Desktop\\camera.jpg"));

        Product newlyCreatedProduct = addProductReq.when().post("api/ecom/product/add-product").then().log().all().extract().as(Prod

        Assert.assertEquals("Product Added Successfully", newlyCreatedProduct.getMessage());

        System.out.println("The product ID is: " + newlyCreatedProduct.getProductId());

        // Creating order
        RequestSpecification orderBaseReq = new RequestSpecBuilder()
            .setBaseUri("https://rahulshettyacademy.com")
            .setContentType(ContentType.JSON)
            .addHeader("Authorization", loginResponse.getToken()).build();

        // Order details creation
        OrderDetails od = new OrderDetails();
        od.setCountry("India");
        od.setProductOrderedId(newlyCreatedProduct.getProductId());

        List<OrderDetails> orderDetailsList = new ArrayList<OrderDetails>();
        orderDetailsList.add(od);

        Order fullOrder = new Order();
        fullOrder.setOrders(orderDetailsList);

        RequestSpecification orderReq = given().spec(orderBaseReq).body(fullOrder);
        OrderResponse or = orderReq.when().post("api/ecom/order/create-order").then().extract().as(OrderResponse.class);
```

```

        System.out.println("Order ID: " + or.getProductOrderId()[0]);
        System.out.println(or.getMessage());
        Assert.assertEquals("Order Placed Successfully", or.getMessage());

    //      Deleting the product created
    RequestSpecification deleteBaseReq = new RequestSpecBuilder()
        .setBaseUrl("https://rahulshettyacademy.com")
        .addHeader("Authorization", loginResponse.getToken()).build();

    RequestSpecification deleteReq = given().spec(deleteBaseReq).pathParam("productId", newlyCreatedProduct.getProductID() );

    String deleteResponse = deleteReq.when().delete("/api/ecom/product/delete-product/{productId}").then().log().all().extract().asString();

    JsonPath js = new JsonPath(deleteResponse);
    String deleteMessage = js.get("message");
    System.out.println(deleteMessage);
    Assert.assertEquals("Product Deleted Successfully", deleteMessage);

    System.out.println("-----DONE-----");

    }
}

```

## Framework Design

Based on what we learned so far we are going to design a full API testing framework from scratch. We are going to use Cucumber and Maven as our base.

### ▼ Maven Basics

Maven is a build management tool used primarily for Java projects. It is used to manage project dependencies, compile code, run tests, and create executable jar files or war files. It is often used in conjunction with other tools such as Jenkins for continuous integration and deployment. Maven uses a project object model (POM) file to define the project structure, dependencies, and build process.

#### ▼ Group ID

In Maven, a Group ID is a unique identifier for a group of related projects. It helps to organize and manage dependencies of a project by grouping them into logical categories.

A Group ID is typically written in reverse domain name notation, such as com.example or org.apache.maven. This ensures that the Group ID is globally unique and helps to avoid naming conflicts between different organizations and projects.

For example, the Spring Framework project has a Group ID of org.springframework. All the artifacts related to the Spring Framework, such as spring-core, spring-web, spring-boot, etc., belong to this group.

When you declare dependencies for your project in the Maven POM (Project Object Model) file, you specify the Group ID along with the artifact ID and version number of the dependency. Maven uses this information to resolve the dependencies and download the required JAR files from the central repository or any other specified repository.

#### ▼ Artifact ID

In Maven, an Artifact ID is the unique identifier for a specific project or module within a group. It identifies a single artifact, such as a JAR or a WAR file, that is produced by a project or module.

The Artifact ID is typically the name of the project or module, but it can also include additional information to differentiate it from other artifacts with the same name within the same group.

For example, the Spring Framework project has multiple modules, each with a unique Artifact ID, such as spring-core, spring-web, spring-boot, and so on. The Artifact ID, along with the Group ID and version number, is used to uniquely identify and manage dependencies for each module.

When you define a dependency on an external artifact in your Maven project, you specify its Group ID, Artifact ID, and version number in the POM (Project Object Model) file. Maven uses this information to download the required JAR files from the central repository or any other specified repository.

## ▼ Cucumber Basics

Cucumber is a testing framework that enables Behavior Driven Development (BDD) by allowing tests to be written in a natural language format that can be easily understood by non-technical stakeholders. In the context of API testing, Cucumber can be used to write feature files that describe the desired behavior of an API, and step definitions that map the steps in the feature file to actual API calls. This allows for tests to be written in a way that is both easily understandable and maintainable.



Cucumber can be used in any automated tests. But we can not use cucumber to automate.

## ▼ Gherkin (GUR-kin)

Cucumber is used with Gherkin which describes software behavior.

Gherkin is a domain-specific language used for writing executable specifications in software development. It is designed to be human-readable and easy to understand by stakeholders, including business analysts, developers, and testers.

Gherkin uses a syntax that is based on keywords and structured in a way that resembles natural language. The language is used to describe the behavior of a software system in terms of features, scenarios, and steps. Gherkin is often used in conjunction with the Behavior Driven Development (BDD) methodology, which emphasizes collaboration and communication between stakeholders in the software development process.

Example of Gherkin:

```
Feature: Login
  As a user
  I want to be able to log in
  So that I can access my account

  Scenario: Successful login
    Given I am on the login page
    When I enter valid credentials
    And click the login button
    Then I should see the dashboard page
```

This scenario describes the behavior of the login feature from the perspective of a user. The

**Given**

, **When**, and **Then** steps are written in a structured format that can be easily understood by both



technical and non-technical stakeholders. The scenario describes the steps that should be taken to log in successfully and the expected outcome. The scenario can be automated and used as an acceptance test to ensure that the login feature is working correctly.

- **Given**: The pre-condition
- **When**: The action
- **Then**: What is the result of these steps
- **But**: To showcase negative test cases

More examples:

```
Scenario: Search for a product
  Given I am on the home page
  When I enter "laptop" in the search field
  And I click the search button
  Then I should see a list of search results
  But I should not see any out of stock items
```

In this scenario, the **But** keyword is used to introduce a step that contrasts with the previous step. The previous step ( **Then I should see a list of search results** ) indicates that the user should see a list of search results. However, the **But** step ( **But I should not see any out of stock items** ) introduces a contrast to that by indicating that the user should not see any out of stock items in the search results. This helps to clarify the expected behavior of the system and makes the scenario more precise

To write a feature that contains all our gherkin test cases we need to create a file with **.feature** extension. Feature file acts as a test suite which consists of all scenarios.



We can think of feature file as a test suite

*We need to add some plugins in our IDE to enable it to understand these syntaxes. Search for Gherkin and cucumber plugins and add them to your preferred IDE.*

## ▼ Setting up Cucumber

For cucumber we need the following jars:

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.13.2</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>io.cucumber</groupId>
    <artifactId>cucumber-java</artifactId>
    <version>7.11.1</version>
  </dependency>
  <dependency>
    <groupId>io.cucumber</groupId>
    <artifactId>cucumber-junit</artifactId>
```

```
<version>7.11.1</version>
</dependency>
</dependencies>
```

To use cucumber we first need to create a feature file. if we have the proper plugins then it would automatically apply formatting and try creating connection between the statements and the actual code.



The feature file does not have code rather its how a BA reads requirements.

The feature file: `Login.feature`

```
Feature: Application Login
  Scenario: Home page default login
    Given User is on NetBanking landing page
    When User logs into the application with username and password
    Then Home page is populated
    And Cards are displayed
```

Now we can use a chrome extension called tidy gherkin or intellij has a built in functionality to convert the Gherkin steps to java methods.

The step definition would look like the following based on the feature file we wrote:

```
package stepDefinitions;

import io.cucumber.java.en.And;
import io.cucumber.java.en.Given;
import io.cucumber.java.en.Then;
import io.cucumber.java.en.When;
import io.cucumber.junit.Cucumber;
import org.junit.runner.RunWith;

@RunWith(Cucumber.class)
public class stepdefinitions {

    @Given("^User is on NetBanking landing page$")
    public void user_is_on_netbanking_landing_page() throws Throwable {
        // throw new PendingException();
    }

    @When("^User logs into the application with username and password$")
    public void user_logs_into_the_application_with_username_and_password() throws Throwable {
        // throw new PendingException();
    }

    @Then("^Home page is populated$")
    public void home_page_is_populated() throws Throwable {
        // throw new PendingException();
    }

    @And("^Cards are displayed$")
    public void cards_are_displayed() throws Throwable {
        // throw new PendingException();
    }

}
```

Now to run our tests we need a java file called `TestRunner.java`



This file and the step definition file has to under the same parent folder.

In the `TestRunner.java` we need mention which is our feature file and what is the step definition file.

`TestRunner.java`

```
@RunWith(Cucumber.class)
@cucumberOptions(
    features = "src/test/java/Features/Login.feature",
    glue = "stepdefinitions"
)
public class TestRunner {
}
```

Here `cucumberOptions` has to 2 items (@RunWith is needed to use the options):

1. `feature`: the location of the feature package **or** the individual feature file.
2. `glue`: the step definition **package**. Here we dont need to mention the file, just the package name is fine. The feature file would take care of finding the correct step definition.



There should be one mapping implementation for each Gherkin line.

## ▼ Passing values

During automation some steps like login and navigating to the homepage are always the same. So when we write a method for that particular line, it would be used by all the test cases that match. We dont need to write the same method multiple times.

For scenarios like this:

```
When User logs into the application with "Maruf" and "Monem"
```

each line for every scenario might be different (positive and negative test). Do we write methods for every one of them? NO! For this case, we just need to pass the string value;

```
@When("User logs into the application with {string} and {string}")
public void userLogsIntoTheApplicationWithAnd(String arg0, String arg1) {
    System.out.println(arg0 + " " + arg1);
}
```

Result

```
Maruf Monem
```

We can also use regex instead of `{string}`. such as

```
@When("User logs into the application with \"([^\"])\") and {string}")
```

## ▼ Logs

To see the logs for the request we would be using the following code:

```

PrintStream log = new PrintStream(new FileOutputStream("logging.txt"));
req = new RequestSpecBuilder().setContentType(ContentType.JSON)
    .addFilter(RequestLoggingFilter.logRequestTo(log))
    .setBaseUrl("https://rahulshettyacademy.com")
    .addQueryParam("key", "qaclick123")
    .setContentType(ContentType.JSON)
    .build();

```

Here `PrintStream` is an object that handles creating the log files. We pass this object to the `.addFilter(RequestLoggingFilter.logRequestTo(log))` method so when a request is sent it would store all the logs in the file name mentioned in `PrintStream` object `logging.txt`.

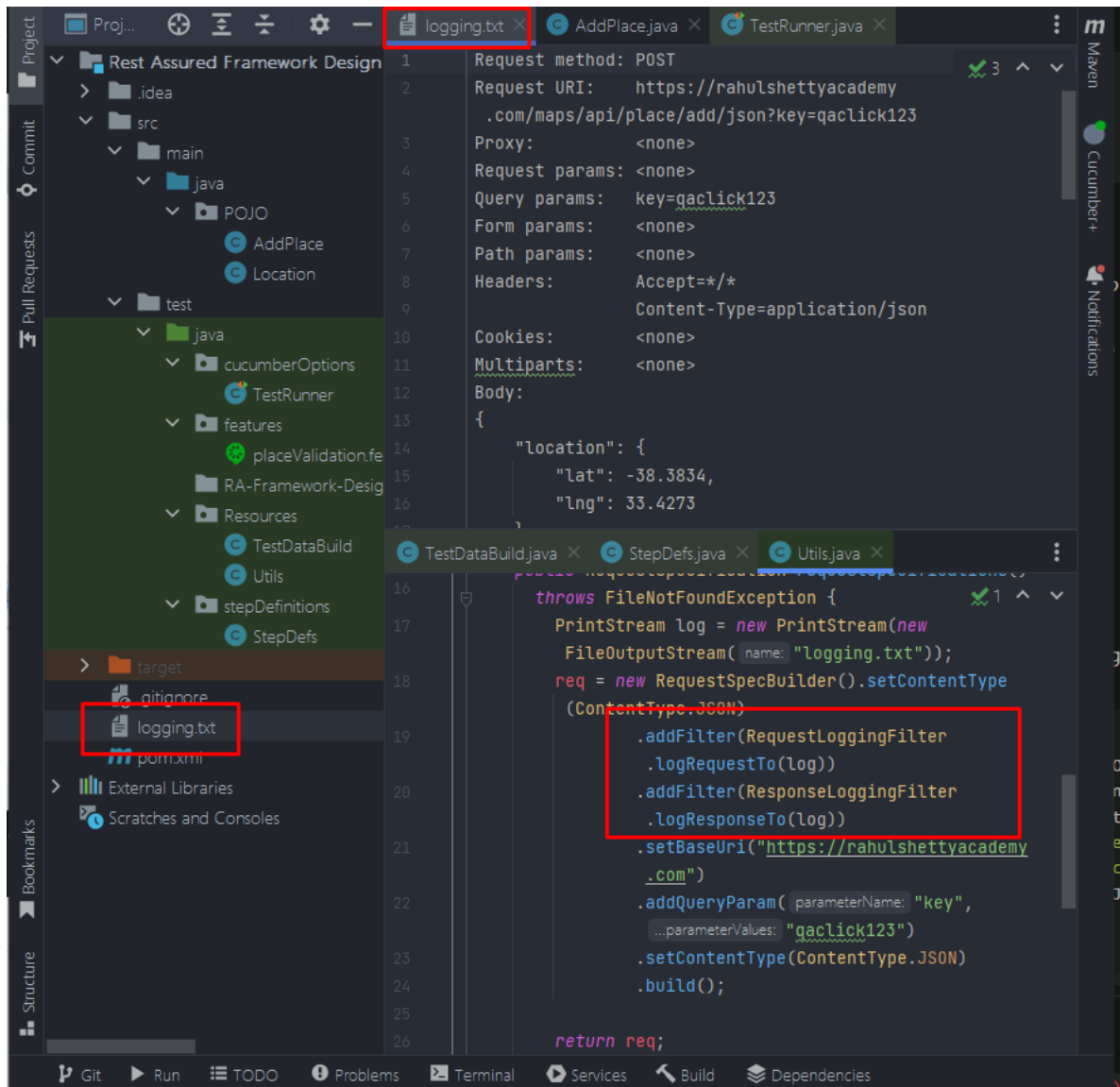
Similarly we want to record response. To do that just add another filter:

```

.addFilter(ResponseLoggingFilter.logResponseTo(log))

```

Project Structure:



## ▼ Global Properties

As there are items such as the `baseUrl` which is used in multiple sections we want to separate it out. To do that we would need to create a new file called `global.properties` which would contain the following:

```
baseUrl=https://rahulshettyacademy.com
```

Now to access this we need to write the following code:

```
public String getGlobalValue(String key) throws IOException {
    Properties prop = new Properties();
    FileInputStream fis = new FileInputStream("src/test/java/Resources/global.properties");
    prop.load(fis);
    return prop.getProperty(key);
}
```

Here we are creating a method that takes the property key as an argument making it easier to get any key value just by calling the method. Then we are using it in the request:

```
.setBaseUrl(getGlobalValue("baseUrl"))
```

## ▼ Data driven testing

So far we hard coded our data for all of our tests.

```
TestDataBuild.java
```

```
ap.setPhone_number("12345");
ap.setWebsite("www.google.com");
```

However, to create a framework that is not the ideal approach. We have to make the data dynamic. To do so, we would be updating our feature file: `placeValidation.feature`

The following section needs to be added to pass in values/data:

```
Examples:
  | name      | language | address |
  | A house   | English  | World cross center |
```

Think of it as a table where each column is an attribute value we are setting. And to use this data we need replace the previous `Scenario:` keyword with `Scenario Outline:` and make sure the column values are passed in the Given tag (as that is creating the payload). Below is the full feature file:

```
Feature: Validating place APIs

Scenario Outline: Verify the add place functionality
  Given Add place payload with "<name>" "<language>" "<address>"
  When User calls "AddPlaceAPI" with Post http request
  Then The API call will get success with status code 200
  And The "status" in response body is "OK"
  And The "scope" in response body is "APP"

Examples:
  | name      | language | address |
  | A house   | English  | World cross center |
```

As the Given tag has changed we need to update the methods. As there are 3 different values being passed, the method would look like this:

```
@Given("Add place payload with {string} {string} {string}")
public void addPlacePayloadWith(String name, String language, String address) throws IOException {
    TestDataBuild td = new TestDataBuild();
    ap=td.addPlacePayload(name, language, address);
    // Request specification
    req = requestSpecifications();
    res = given().spec(req).body(ap);
}
```



Each set of data (row) represents a test!

The problem now is the info in the log file gets replaced. To handle this

```
public class Utils {
    public static RequestSpecification req;

    public RequestSpecification requestSpecifications() throws IOException {
        if(req == null) {
            PrintStream log = new PrintStream(new FileOutputStream("logging.txt"));
            req = new RequestSpecBuilder().setContentType(ContentType.JSON)
                .addFilter(RequestLoggingFilter.logRequestTo(log))
                .addFilter(ResponseLoggingFilter.logResponseTo(log))
                .setBaseUrl(getGlobalValue("baseUrl"))
                .addQueryParam("key", "qaclick123")
                .setContentType(ContentType.JSON)
                .build();
        }
        return req;
    }
}
```

Here we are changing the `req` to static so its shared between all our test cases and ensuring that is the `req` has been triggered once no new initialization is required. This ensures that the log files content dont get replaced for each dataset.

## ▼ Enum

From ChatGPT:

In Java, an `enum` class is a special type of class that represents a fixed set of constants. An `enum` class can be used to define a collection of values that are used as symbolic names (often referred to as "enumeration constants" or simply "enums") for a group of related items or options. Enums are commonly used to represent sets of values that are known at compile time and are not expected to change during runtime.

```
enum EnumName {
    CONSTANT1,
    CONSTANT2,
    CONSTANT3,
    // ...
}
```

Here, `EnumName` is the name of the `enum` class, and `CONSTANT1`, `CONSTANT2`, `CONSTANT3`, and so on, are the enumeration constants defined within the `enum` class. Enumeration constants are implicitly declared as `public static final` members of the `enum` class, and their names are usually in uppercase letters by convention.

`enum` classes can also have constructors, methods, and fields, similar to regular Java classes. Enums can be used in switch statements, as well as in collections and arrays. Enum constants can be compared using the `==` operator, and you can also iterate over the values of an `enum` using the `values()` method.

`enum` classes are a powerful feature in Java that provide a convenient way to represent a fixed set of values and can help improve code readability and maintainability by providing type-safe and self-documenting constants.

Now that all the theory is done let's see how we incorporate ENUM to our project:

We need to create a separate class that would store all the API values:

```
package Resources;

public enum APIResources {
    AddPlaceAPI("/maps/api/place/add/json"),
    GetPlaceAPI("/maps/api/place/get/json"),
    DeletePlaceAPI("/maps/api/place/delete/json");
    private String resource;

    APIResources(String resource) {
        this.resource=resource;
    }

    public String getResource(){
        return resource;
    }
}
```

Here `AddPlaceAPI`, `GetPlaceAPI`, `DeletePlaceAPI` are the constant values. They are treated more like methods. The constructor takes a string as the argument as the value for the enums. The get method helps us to get the string value. The `resource` variable stores the value associated with the constant.

In our step definition we write the following:

```
APIResources resourceObject = APIResources.valueOf(arg0);
String apiValue = resourceObject.getResource();
```

here `valueOf` is an `enum` method which takes the constant value name ex: `AddPlaceAPI`. This stores the constant in a `APIResources` object. When using: `APIResources.valueOf(arg0)` it takes the constants associated API value and stores it in the `resource` variable. Then we can call the `getResource` method to extract the string.

ChatGPT explains this well:

The code you provided defines an `enum` class called `APIResources` in Java, which represents a set of constants for API resources related to a mapping service. Let's go through the code step by step:

1. `public enum APIResources`: This declares an `enum` class named `APIResources` with a `public` access modifier, meaning it can be accessed from anywhere in the code.
2. `AddPlaceAPI("/maps/api/place/add/json"), GetPlaceAPI("/maps/api/place/get/json"), DeletePlaceAPI("/maps/api/place/delete/json");`: These are the enumeration constants defined within the `APIResources` enum. They represent the API resources for adding, getting, and deleting places, respectively. Each constant is followed by its corresponding resource path as a string.
3. `private String resource;`: This is a private field defined within the `APIResources` enum, which will be used to store the resource path associated with each enumeration constant.
4. `APIResources(String resource)`: This is a constructor for the `APIResources` enum, which takes a `String` parameter named `resource`. This constructor is called when each enumeration constant is defined, and it sets the value of the `resource` field for that constant.
5. `this.resource = resource;`: This statement inside the constructor assigns the value of the `resource` parameter to the `resource` field of the enum.

In summary, the `APIResources` enum defines a set of constants representing API resources for a mapping service, and each constant has an associated resource path that is stored in the `resource` field. The constructor is used to initialize the `resource` field with the corresponding value for each constant.

## ▼ Running specific tests

Lets say we have multiple scenarios and now we want to only run one tests from our feature file. To run specific test we can introduce tagging.

```
Feature: Validating place APIs

@AddPlace
Scenario Outline: Verify the add place functionality
  Given Add place payload with "<name>" "<language>" "<address>"
  When User calls "AddPlaceAPI" with "Post" http request
  Then The API call will get success with status code 200
  And The "status" in response body is "OK"
  And The "scope" in response body is "APP"
  And Verify place_ID created maps to "<name>" using "getPlaceAPI"

Examples:
  | name   | language | address          |
  | A house | English  | World cross center |
  | Maruf  | Bangla   | Dhaka              |

@DeletePlace
Scenario: Verify if the the Delete Place API is working
  Given DeletePlace payload
  When User calls "DeletePlaceAPI" with "Post" http request
  And The "status" in response body is "OK"
```

In the test runner we add the tests (tags) we want to run:

```
@RunWith(Cucumber.class)
@cucumberOptions(
    features = "src/test/java/features/placeValidation.feature",
    glue = "stepDefinitions",
    tags = "@AddPlace"
)
public class TestRunner {
}
```

## ▼ Hooks

In case we have tests that are dependent on others we need to specify that X method needs to run before a particular test.

In our framework design, we want to run the Delete Place API. However, that API relies on the Add Place API to get the place ID. To resolve this we create a new class called hooks that would give us the place ID by running before the Delete Place tag.

```
package stepDefinitions;

import io.cucumber.java.Before;
import java.io.IOException;

public class Hooks {
```



```

@Before("@DeletePlace")
public void beforeScenario() throws IOException {
    // Execute this if place id is null
    StepDefs sd = new StepDefs();
    if (StepDefs.placeID == null){
        sd.addPlacePayloadWith("Anik", "Bangla", "Dhaka");
        sd.userCallsWithPostHttpRequest("AddPlaceAPI", "Post");
        sd.verifyPlace_IDCreatedMapsToUsing("Anik", "getPlaceAPI");
    }
}
}

```

Here, the `if` statement is present to ensure that this process is only run when the place ID is null because if we do run the Add place API tests before then the place ID value would be present and we don't need to run duplicate code again.

## ▼ Executing tests using maven

So far we have used IntelliJ or Eclipse to run tests but in production we won't have IDEs so there we use maven commands.

1. Go to our project path

2. Run `mvn test`

It would check the `cucumber.Options` package and run the `TestRunner.java`

**To pass in tags:**

We can write the following:

```

mvn test -Dcucumber.filter.tags="@AddPlace" - single tag
mvn test -Dcucumber.filter.tags="@AddPlace or @DeletePlace" - multiple tags

```

-D refers to parameters.

Each test can have multiple tags like

`@AddPlace @Regression`

## ▼ Reports

Before creating reports we need `TestRunner` to create a JSON results file which would be used to generate reports. This isn't done by default so we need to specify:

```

@RunWith(Cucumber.class)
@CucumberOptions(
    features = "src/test/java/features/placeValidation.feature",
    plugin="json:target/jsonReports/cucumber-report.json",
    glue = "stepDefinitions",
    tags = "@AddPlace"
)

```

`target/jsonReports/cucumber-report.json` → this refers to the location where we want the JSON file to be created/stored (this path is the standard).

To create reports there is a plugin available here: <https://github.com/damianszczepanik/maven-cucumber-reporting>

Copy the below code:

```

<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>5.7.5</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 -->
        <skip>false</skip>
        <!-- output directory for the generated report -->
        <outputDirectory>${project.build.directory}</outputDirectory>
        <!-- optional, defaults to outputDirectory if not specified -->
        <inputDirectory>${project.build.directory}/jsonReports</inputDirectory>
        <jsonFiles>
          <!-- supports wildcard or name pattern -->
          <param>/**/*.json</param>
        </jsonFiles>
        <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>

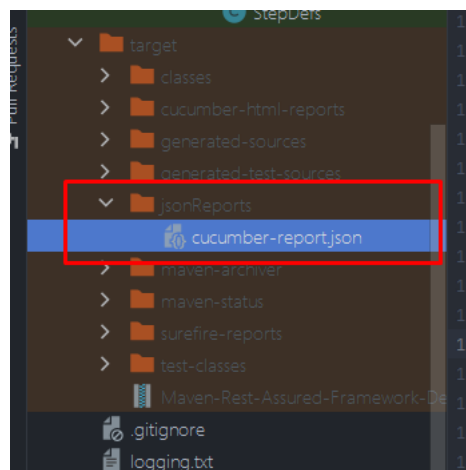
```

And add it before dependencies. After adding this run `mvn test verify`



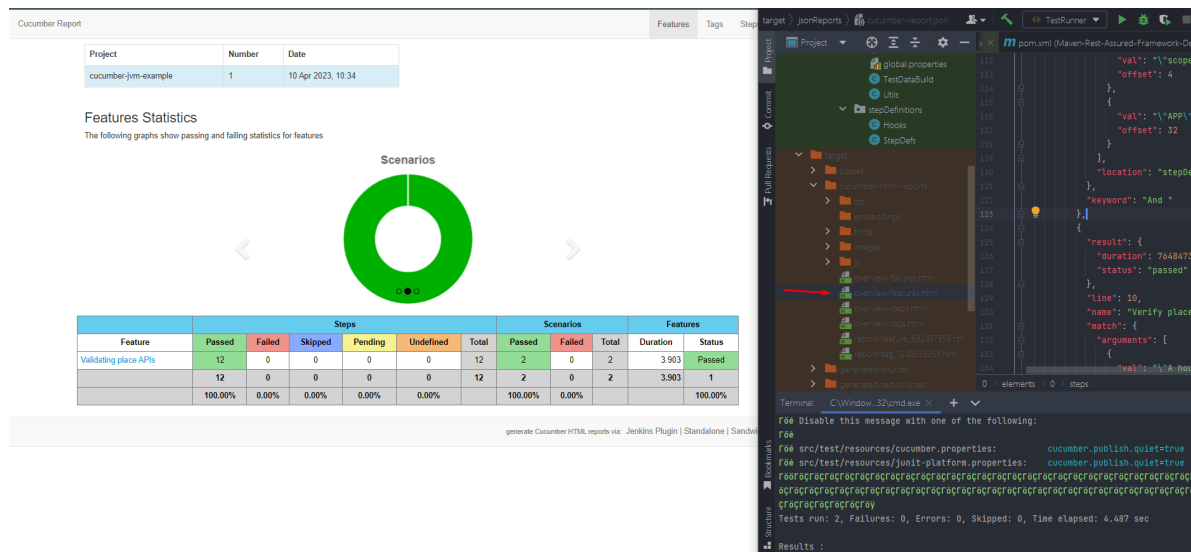
`mvn test` basically runs the tests. `mvn verify` on the other hand generates a report.

The TestRunner created the below



The plugin is basically reading it and generating a report.

The report;



## ▼ GraphQL

According to chatGPT:

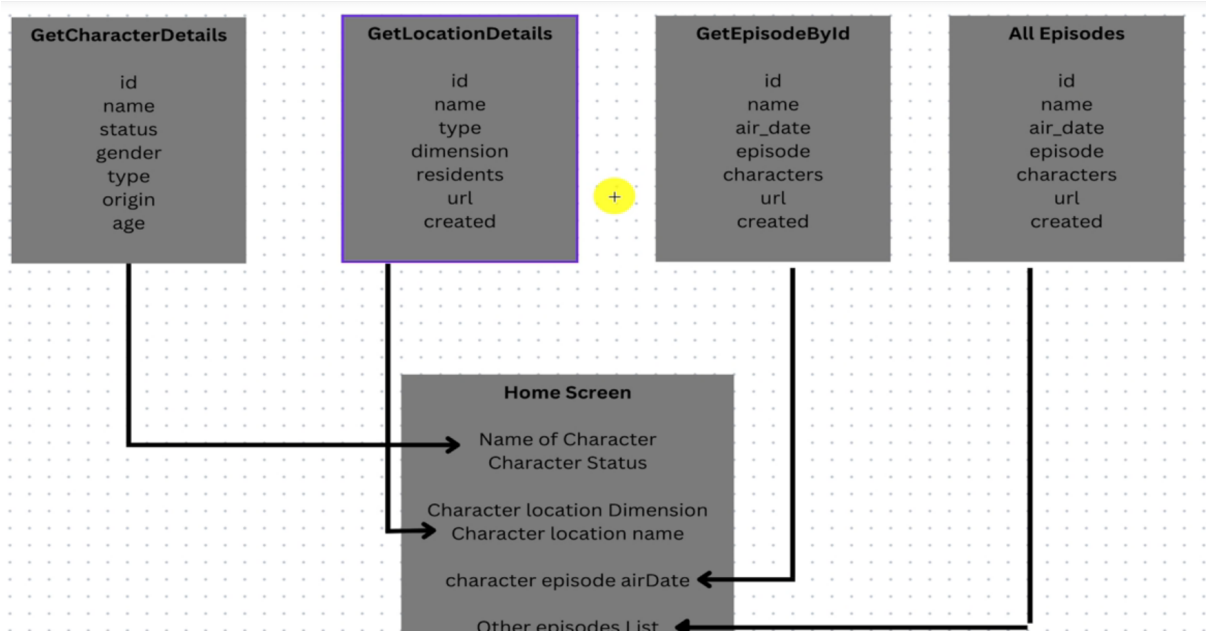
GraphQL is a query language for your API and a runtime for executing those queries against your data. It was developed by Facebook in 2012 and has gained popularity as a powerful and flexible alternative to traditional REST APIs.

At its core, GraphQL allows clients to request exactly the data they need, and nothing more. Clients can specify the shape and structure of the response, making it a highly efficient and customizable way to fetch data from APIs. GraphQL enables declarative data fetching, where clients specify what they want and servers respond with only that data, eliminating over-fetching or under-fetching of data, which can be common in REST APIs.

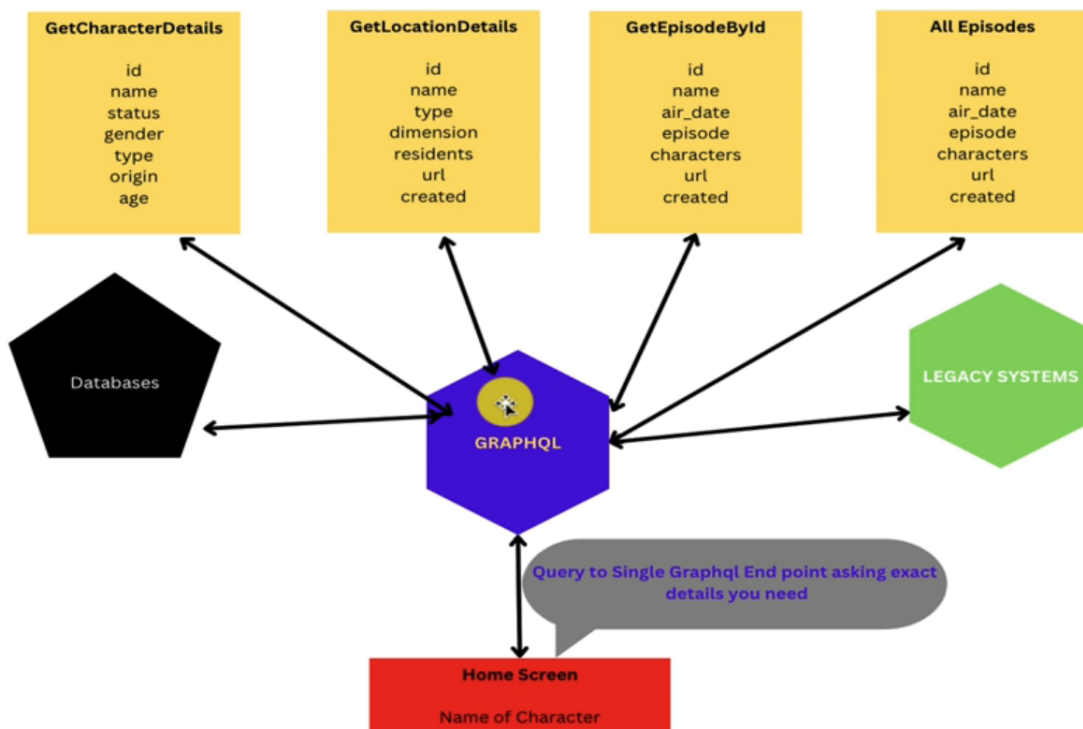
GraphQL is also strongly typed, which means that APIs built with GraphQL have a well-defined schema that describes the data available and the operations that can be performed on it. This makes it easy to understand and work with APIs, as well as enabling powerful tools for code generation, validation, and documentation.

GraphQL is not tied to any specific programming language or framework, and it can be used in various environments, including server-side applications, mobile apps, and web applications. It has a growing ecosystem of libraries, tools, and community support, making it a popular choice for modern API development.

Below is websites structure:



Here if we want to get character details an API call is made which gets all the information from the **GetCharacterDetails** API. Here if we want to just get the name we would still have to call the API and have POJO classes in place to get the info. Furthermore, if we want to load the homepage we are querying multiple APIs. Which might cause performance issues.



With GraphQL (GQL) we would query to a single GraphQL end point which would get all the details.

GraphQL provides a more flexible approach compared to traditional APIs like REST. In REST APIs, endpoints are pre-defined and often return fixed data structures, while in GraphQL, clients can request exactly the data they need and get back a response tailored to their request. This allows for more efficient and precise data fetching, reducing over-fetching or under-fetching of data



GraphQL is not a storage system and its not a replacement for API's.

To query GQL we first need to know about the GQL schema.

In GraphQL, a schema is a fundamental concept that defines the structure, types, and operations available in a GraphQL API. It serves as a contract between the server and the client, specifying how the data is organized and what operations can be performed on it.

A GraphQL schema is typically defined using the GraphQL Schema Definition Language (SDL), which is a human-readable syntax for describing the types and operations in a GraphQL API. The schema defines the available types, such as objects, scalars, and enums, their fields, relationships, and the operations that can be performed on them, such as queries, mutations, and subscriptions.

Here are some key components of a GraphQL schema:

1. **Types:** Types define the shape and structure of the data in a GraphQL API. They can represent objects, scalars (e.g., strings, numbers, booleans), enums (a fixed set of values), and custom types. Types can have fields that represent the data associated with them.
2. **Fields:** Fields are the individual pieces of data that can be requested on a type. They can have arguments that allow clients to pass parameters to customize the result, and they can return other types or scalars.
3. **Relationships:** Types can be related to each other through fields, representing relationships between different types of data. For example, a "User" type may have a field "posts" that returns a list of "Post" type objects associated with that user.
4. **Operations:** GraphQL supports three main types of operations: queries, mutations, and subscriptions. Queries are used to fetch data, mutations are used to modify data, and subscriptions are used to receive real-time updates when data changes.
5. **Directives:** Directives are used to provide additional instructions to the GraphQL server on how to handle certain aspects of the query, such as filtering, pagination, or authentication. Directives are prefixed with "@" in the SDL.

Once the schema is defined, it serves as a contract that describes the capabilities of the GraphQL API, and clients can use this schema to introspect and understand the available types, fields, and operations. The schema is typically implemented on the server-side and used to validate and execute GraphQL queries and mutations to retrieve or modify data according to the defined structure and operations.

An example:

```
type User {
  id: ID!
  name: String!
  age: Int
  posts: [Post]
}

type Post {
  id: ID!
  title: String!
  body: String!
  author: User!
}

type Query {
  getUser(id: ID!): User
  getPost(id: ID!): Post
}

type Mutation {
  createUser(name: String!, age: Int): User
  createPost(title: String!, body: String!, authorId: ID!): Post
}
```

In this example, we define three types: `User`, `Post`, and `Query`, and one mutation `Mutation`.

- `User` type has fields: `id` of type `ID`, `name` of type `String`, `age` of type `Int`, and `posts` which is an array of `Post` objects.
- `Post` type has fields: `id` of type `ID`, `title` of type `String`, `body` of type `String`, and `author` of type `User` representing the author of the post.
- `Query` type has two queries: `getUser` which takes an `id` argument of type `ID` and returns a `User` object, and `getPost` which takes an `id` argument of type `ID` and returns a `Post` object.
- `Mutation` type has two mutations: `createUser` which takes `name` and `age` arguments of respective types `String` and `Int`, and returns a `User` object, and `createPost` which takes `title`, `body`, and `authorId` arguments of respective types `String`, `String`, and `ID`, and returns a `Post` object.

This schema describes the types, fields, and operations available in a GraphQL API. Clients can use this schema to query for users and posts, create new users and posts, and retrieve data in a structured and customizable way. The actual implementation of resolvers for these types and operations would be done on the server-side using a GraphQL server.

Website to practice: <https://rahulshettyacademy.com/gg/graphql> → graphql explorer (comes by default). This would have all the documentation for the example we are going to use for the upcoming sections.

## ▼ Queries

Based on the screen shot we have seen so far we would write queries with that example in the above mentioned practice section.

In the documentation if we see an argument with `!` after the type that means its a mandatory argument. For example:

```
character(characterId: Int!): CharacterDetails

Get one character by its id
```

In the example we have a `character` type with a bunch of fields associated with it. Now when we are querying a character we want just the characters name and gender. So to do that we write:

```

query{
  character(characterId: 8){
    name
    gender
  }
}

```

In the same query we can make GQ talk to multiple types:

```

query{
  character(characterId: 8){
    name
  }
  location(locationId:8!){
    dimension
    name
  }
}

```

if we used APIs this would have been done by 2 separate requests.

We can also have arguments that take different types;

```

characters(filters: CharacterFilters, pagination:
Pagination): CharactersResult!

Get all characters

```

The `CharacterFilters` has the below fields:

#### FIELDS

`name: String`

Character's name

`status: String`

Character's status

`species: String`

Character's species

`type: String`

Character's type

`gender: String`

Character's gender

`CharacterResult` is the response we get. However, that itself is a type so we need to get the field name.

info: Info!

Information of the result

result: [CharacterDetails]!

Query result

Below is the query we wrote to get the Rahul characters → info → count (how many characters have the name Rahul):

```
characters(filters:{name: "Rahul"})
{
  info{
    count
  }
}
```

Another example:

```
characters(filters:{name: "Rahul"})
{
  info{
    count
  }
  result{
    name
    type
  }
}
```

Passing multiple arguments in the filter:

We can use query parameters that would be used like variables throughout the query.

Query variable:

```
{
  "characterID":8
}
```

used in the query:

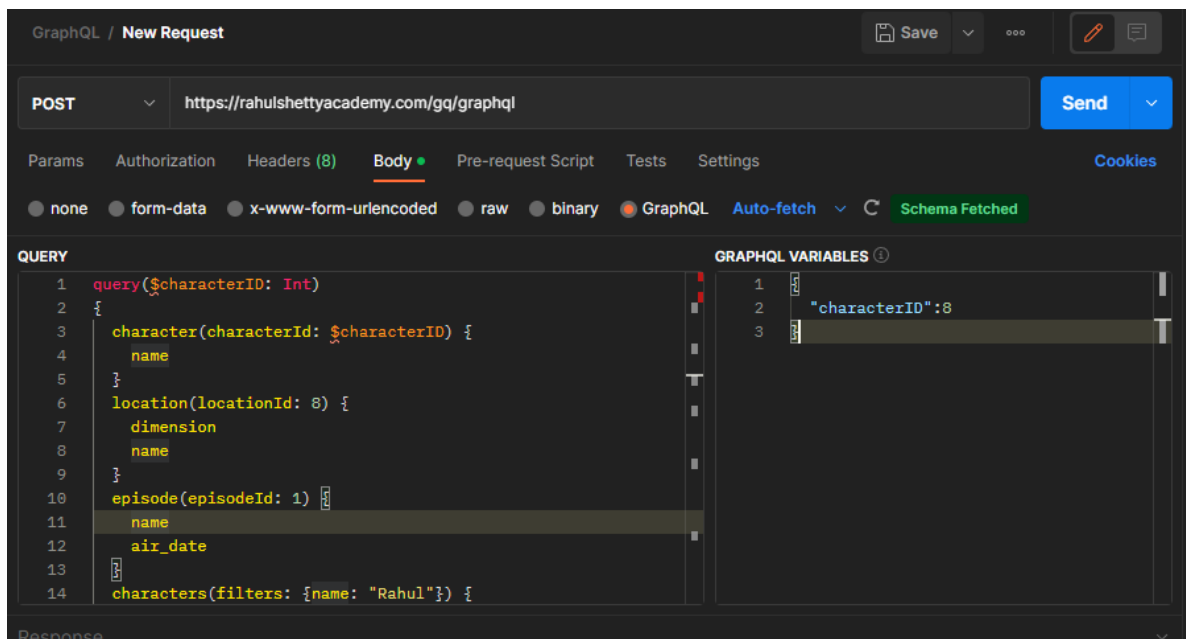
```
query($characterID: Int!)
{
  character(characterId: $characterID) {
    name
  }
}
```



By default all GQL requests are POST calls because we are sending the body



We can pass what we generated so far in postman by selecting the GQL option.



Below is an example:

Query:

```
query($characterID: Int!)
{
  character(characterId: $characterID) {
    name
  }
  location(locationId: 2577) {
    dimension
    name
  }
  episode(episodeId: 1463) {
    name
    air_date
  }
  characters(filters: {name: "Maruf"}) {
    info {
      count
    }
    result {
      name
    }
  }
  episodes(filters: {episode: "01"}) {
    result {
      id
      name
      air_date
      episode
    }
  }
}
```

Result:

```
{
  "data": {
    "character": {
```

```

        "name": "Maruf"
      },
      "location": {
        "dimension": "147570",
        "name": "Bangladesh"
      },
      "episode": {
        "name": "Scooby Doo",
        "air_date": "1990"
      },
      "characters": {
        "info": {
          "count": 1
        },
        "result": [
          {
            "name": "Maruf"
          }
        ]
      },
      "episodes": {
        "result": [
          {
            "id": 1373,
            "name": "Prison Break",
            "air_date": "01-06-2000",
            "episode": "01"
          },
          {
            "id": 1374,
            "name": "Aquaman",
            "air_date": "01-06-2000",
            "episode": "01"
          },
          {
            "id": 1463,
            "name": "Scooby Doo",
            "air_date": "1990",
            "episode": "01"
          }
        ]
      }
    }
  }
}

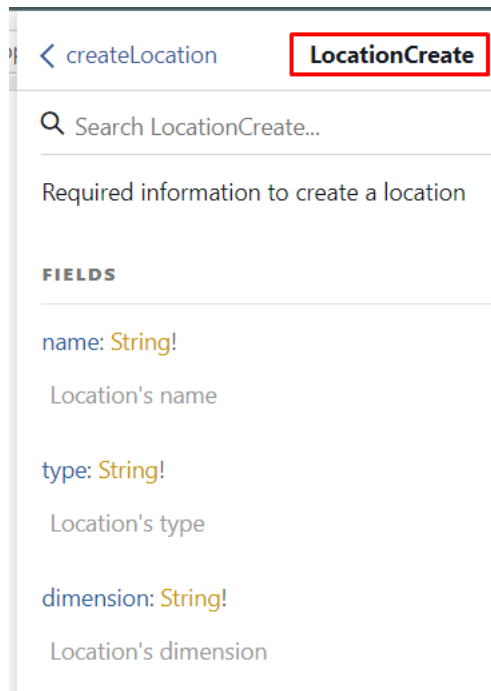
```

## ▼ Mutations

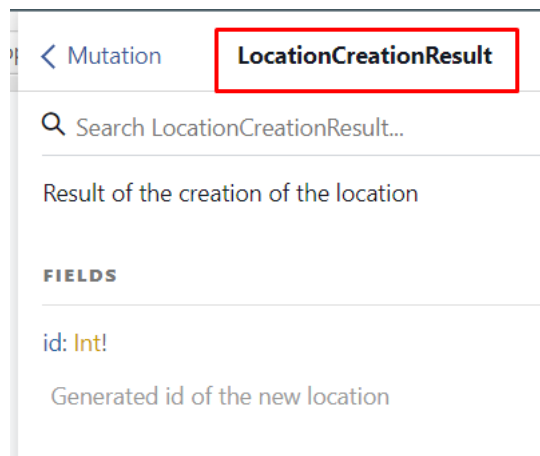
Creating data

`createLocation(location: LocationCreate):`  
`LocationCreationResult`

Create a new location



Returns:



In mutation we can create and retrieve information (hybrid).

```
mutation {
  createLocation(location: {name: "Bangladesh2", type: "South East Asia", dimension: "147570"}) {
    id
  }
  createCharacter(character: {name: "Maruf", type: "learner", status: "alive", species: "Human", gender: "male", image: "png", origin: "Bangladesh"}) {
    id
  }
}
```

Result;

```
{
  "data": {
    "createLocation": {
      "id": 2578
    }
  }
}
```

```

    },
    "createCharacter": {
      "id": 2034
    }
  }
}

```

### Deleting multiple locations

```

deleteLocations(locationIds: [2578, 2579]){
  locationsDeleted
}

```

When we see an array symbol that means we can pass multiple values.

### Mutation using query variables:

```

mutation($LocationName:String!, $characterName: String!, $episodeName:String!) {
  createLocation(location: {name: $LocationName, type: "South East Asia", dimension: "147570"}) {
    id
  }
  createCharacter(character: {name: $characterName, type: "learner", status: "alive", species: "Human", gender: "male", image:"png",
    id
  })
  createEpisode(episode: {name: $episodeName, air_date:"1990", episode:"01"}){
    id
  }
}

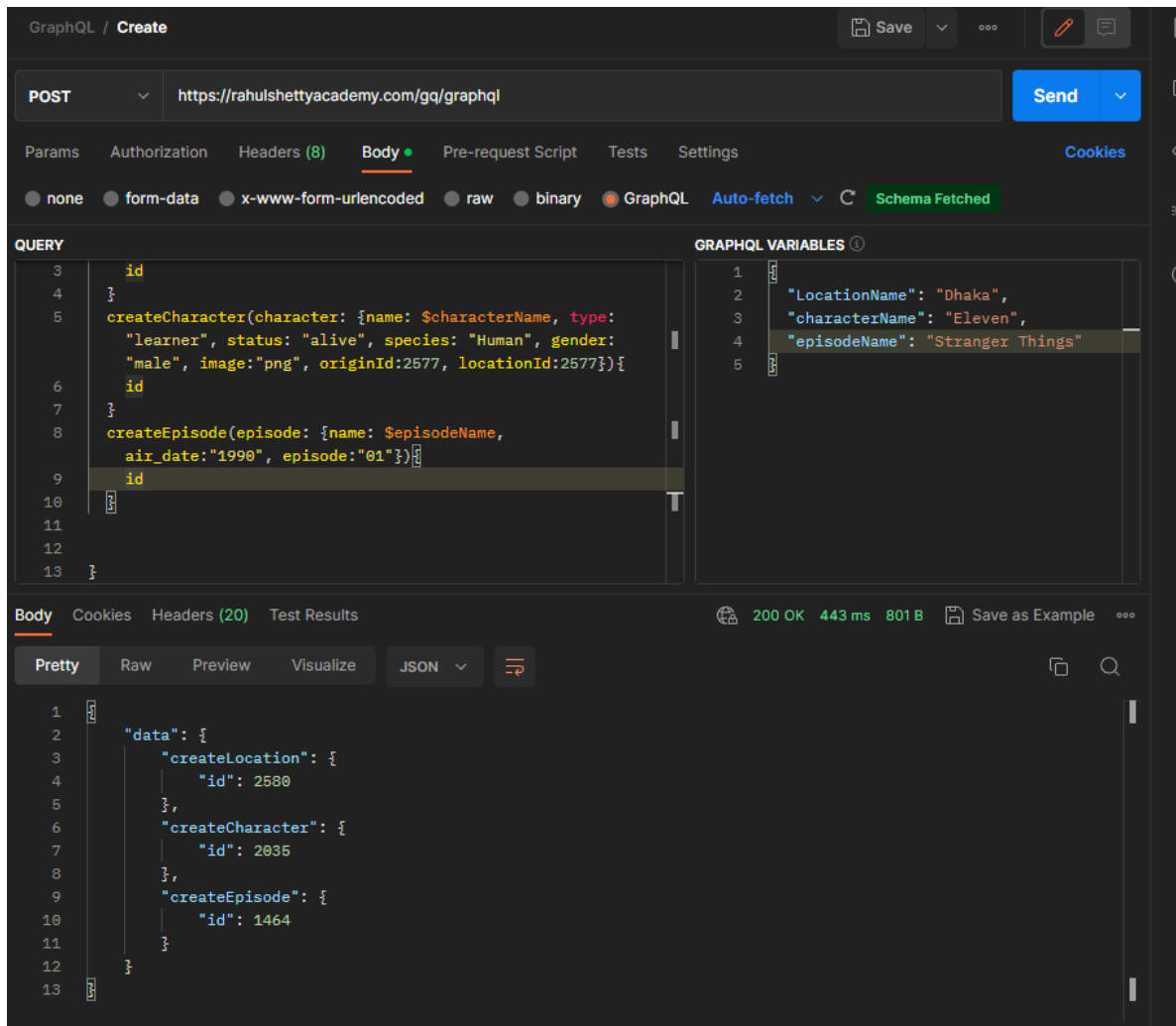
```

### Query variables

```

{
  "LocationName": "Dhaka",
  "characterName": "Eleven",
  "episodeName": "Stranger Things"
}

```



## ▼ GQL with RestAssured

So far we know that RestAssured only takes JSON in its body. However, the GQL body isn't in one OR is it? 😞

Checking the payload from the GQL request `network tab` → `payload` we get the following:

```
{"query": "mutation($LocationName:String!, $characterName: String!, $episodeName:String!) {\n  createLocation(location: {name: $LocationName, status: 'alive', species: 'Human', gender: 'male', image: 'png', originId: 2577, locationId: 2577}) {\n    id\n  }\n  createCharacter(character: {name: $characterName, type: 'learner', status: 'alive', species: 'Human', gender: 'male', image: 'png', originId: 2577, locationId: 2577}) {\n    id\n  }\n  createEpisode(episode: {name: $episodeName, air_date: '1990', episode: '01'}) {\n    id\n  }\n}
```

If we carefully look `query` and `variables` are the key and the other info are values.

The full code:

```
import io.restassured.path.json.JsonPath;
import netscape.javascript.JSObject;

import static io.restassured.RestAssured.*;

public class GraphQLScript {
    public static void main(String[] args) {
        String response = given().log().all().header("Content-Type", "application/json")
            .body("{\"query\": \"mutation($LocationName:String!, $characterName: String!, $episodeName:String!) {\n  createLocation(location: {name: $LocationName, status: 'alive', species: 'Human', gender: 'male', image: 'png', originId: 2577, locationId: 2577}) {\n    id\n  }\n  createCharacter(character: {name: $characterName, type: 'learner', status: 'alive', species: 'Human', gender: 'male', image: 'png', originId: 2577, locationId: 2577}) {\n    id\n  }\n  createEpisode(episode: {name: $episodeName, air_date: '1990', episode: '01'}) {\n    id\n  }\n}\n\"}")
            .when().post("https://rahulshettyacademy.com/gq/graphql")
            .then().statusCode(200)
            .extract().response().asString();

        JSObject jsObject = new JSObject(response);
        JsonPath jsonPath = new JsonPath(jsObject);
        String id = jsonPath.getString("data.createLocation.id");
        String id2 = jsonPath.getString("data.createCharacter.id");
        String id3 = jsonPath.getString("data.createEpisode.id");
        System.out.println(id + " " + id2 + " " + id3);
    }
}
```

```

        .when().post("https://rahulshettyacademy.com/gq/graphql")
        .then().extract().response().asString();
        System.out.println(response);

        System.out.println("-----");

        JsonPath js = new JsonPath(response);
        System.out.println("location ID: " + js.getString("data.createLocation.id"));

    }
}

```

Response:

```

{"data":{"createLocation":{"id":2582},"createCharacter":{"id":2037},"createEpisode":{"id":1466}}}
-----
location ID: 2582

```

To make location name dynamic:

```

String locationName = "Ontario";
String response = given().log().all().header("Content-Type", "application/json")
    .body("{\"query\":\"mutation($LocationName:String!, $characterName: String!, $episodeName:String!) {\n  createLocat

```

## ▼ Excel integration with Rest Assured

To read data from excel we need Apache poi dependencies.

## ▼ Apache POI

Dependencies

```

<dependency>
  <groupId>org.apache.poi</groupId>
  <artifactId>poi</artifactId>
  <version>5.2.2</version>
</dependency>
<dependency>
  <groupId>org.apache.poi</groupId>
  <artifactId>poi-ooxml</artifactId>
  <version>5.2.2</version>
</dependency>

```

## ▼ Reading cells

Adding ability to read the `xlsx` file:

```

FileInputStream fis = new FileInputStream("X:\\Career\\Learning\\API\\Excel_Data_Driven\\excelDataDrivenData.xlsx");
XSSFWorkbook workbook = new XSSFWorkbook(fis);

```

Getting the desired sheet from excel;

```

int sheetCount = workbook.getNumberOfSheets();
for(int i=0; i<sheetCount; i++){

```

```

    if (workbook.getSheetName(i).equalsIgnoreCase("main")){
        XSSFSheet targetSheet= workbook.getSheetAt(i);
    }
}

```

## Getting access to rows

```

Iterator<Row> rows = targetSheet.iterator();
Row firstRow = rows.next(); // comes to the first row

```

## Identify the desired column (Testcases)

```

Iterator<Cell> cells = firstRow.cellIterator();
int count =0;
int columnNumber = 0;
while (cells.hasNext()){
    Cell cellValue = cells.next();
    if(cellValue.getStringCellValue().equalsIgnoreCase("Testcases")){
        System.out.println("target cell number" + count );
        columnNumber=count;
    }
    count++;
}

```

## Identifying the purchase testcase row:

```

while (rows.hasNext()){
    Row r = rows.next();
    if(r.getCell(columnNumber).getStringCellValue().equalsIgnoreCase("Purchase")){
    }
}

```

## Getting all the data

```

Iterator<Cell> c = r.cellIterator();
while (c.hasNext()){
    String cellValue = c.next().getStringCellValue();
    System.out.println("Purchase cell value: " + cellValue);
    dataStorage.add(cellValue);
}

```

The `c.next().getStringCellValue()` doesnt work if the cell value is Int. Its hard to know what type of value a cell might contain. To handle this:

```

while (c.hasNext()){
    Cell cellItem = c.next();
    if(cellItem.getCellType() == CellType.STRING){
        cellValue = cellItem.getStringCellValue();
    }else {
        cellValue = NumberToTextConverter.toText(cellItem.getNumericCellValue());
    }
    // System.out.println("Purchase cell value: " + cellValue);
    dataStorage.add(cellValue);
}

```

Full code:

```

package resources;

import org.apache.poi.ss.usermodel.Cell;
import org.apache.poi.ss.usermodel.CellType;
import org.apache.poi.ss.usermodel.Row;
import org.apache.poi.ss.util.NumberToTextConverter;
import org.apache.poi.xssf.usermodel.XSSFSheet;
import org.apache.poi.xssf.usermodel.XSSFWorkbook;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Iterator;

public class dataDriven {

    public ArrayList<String> getData(String testcaseName) throws IOException {
        ArrayList<String> dataStorage = new ArrayList<>();
        FileInputStream fis = new FileInputStream("X:\\Career\\Learning\\API\\Excel_Data_Driven\\excelDataDrivenData.xlsx");
        XSSFWorkbook workbook = new XSSFWorkbook(fis);

        int sheetCount = workbook.getNumberOfSheets();
        for(int i=0; i<sheetCount; i++){
            if (workbook.getSheetName(i).equalsIgnoreCase("main")){
                XSSFSheet targetSheet= workbook.getSheetAt(i);

                Iterator<Row> rows = targetSheet.iterator();
                Row firstRow = rows.next(); // comes to the first row

                Iterator<Cell> cells = firstRow.cellIterator();
                int count =0;
                int columnNumber = 0;
                while (cells.hasNext()){
                    Cell cellValue = cells.next();
                    if(cellValue.getStringCellValue().equalsIgnoreCase("Testcases")){
                        columnNumber=count;
                        break;
                    }
                    count++;
                }
                System.out.println("target cell number: " + columnNumber );

                // Finding the purchase testcase
                while (rows.hasNext()){
                    Row r = rows.next();
                    if(r.getCell(columnNumber).getStringCellValue().equalsIgnoreCase(testcaseName)){
                        Iterator<Cell> c = r.cellIterator();
                        String cellValue;
                        while (c.hasNext()){
                            Cell cellItem = c.next();
                            if(cellItem.getCellType() == CellType.STRING){
                                cellValue = cellItem.getStringCellValue();
                            }else {
                                cellValue = NumberToTextConverter.toText(cellItem.getNumericCellValue());
                            }
                        }
                        // System.out.println("Purchase cell value: " + cellValue);
                        dataStorage.add(cellValue);
                    }
                }
            }
        }
        return dataStorage;
    }
}

```

## ▼ Rest Assured Hash Maps

Documentation is available here: <https://github.com/rest-assured/rest-assured/wiki/Usage>

Creating a JSON from a HashMap:



```

Map<String, Object> jsonAsMap = new HashMap<>();
jsonAsMap.put("firstName", "John");
jsonAsMap.put("lastName", "Doe");

given().
    contentType(JSON).
    body(jsonAsMap).
when().
    post("/somewhere").
then().
    statusCode(200);

```

This will provide a JSON payload as:

```
{ "firstName" : "John", "lastName" : "Doe" }
```

Below is an example WRTO the Library API we have seen previously:

```

package LibraryAPI;
import ResuableItems.ResuableMethods;
import io.restassured.RestAssured;
import io.restassured.path.json.JsonPath;
import io.restassured.response.Response;

import java.util.HashMap;
import java.util.Map;

import static io.restassured.RestAssured.given;

public class AddBookWithHashMap {
    public static void main(String[] args) {

        Map<String, Object> jsonAsMap = new HashMap<>();
        jsonAsMap.put("name", "Marufs Book 1");
        jsonAsMap.put("isbn", "isbnn");
        jsonAsMap.put("aisle", "9876");
        jsonAsMap.put("author", "Maruf Monem");

        RestAssured.baseURI= "http://216.10.245.166";
        Response response = given().header("Content-Type", "application/json")
            .body(jsonAsMap)
            .when().post("Library/Addbook.php")
            .then().assertThat().statusCode(200).extract().response();
        JsonPath js = ResuableMethods.rawToJson(response.asString());
        String id = js.get("ID");
        System.out.println(id);

    }
}

```

When dealing with nested JSON similar to POJO classes we need to created another hashmap and pass that information:

```

Map<String, Object> jsonAsMap1 = new HashMap<>();
Map<String, Object> jsonAsMap2 = new HashMap<>();

jsonAsMap2.put("mobile", "123");
jsonAsMap2.put("homeophone", "456");

jsonAsMap1.put("contact", jsonMap2);

```