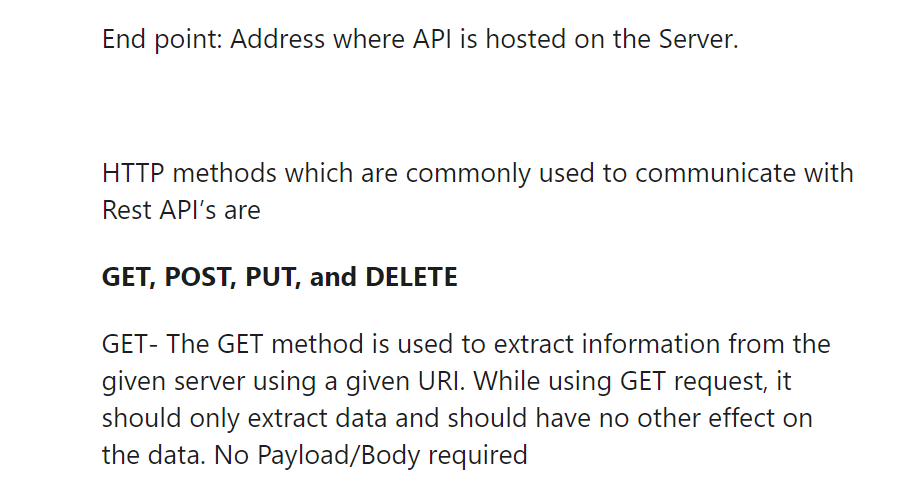
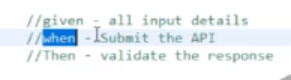
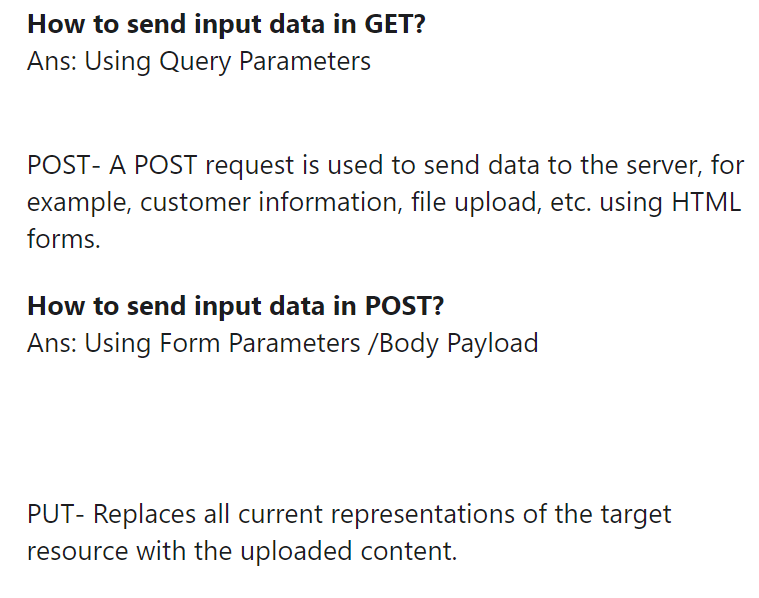
**

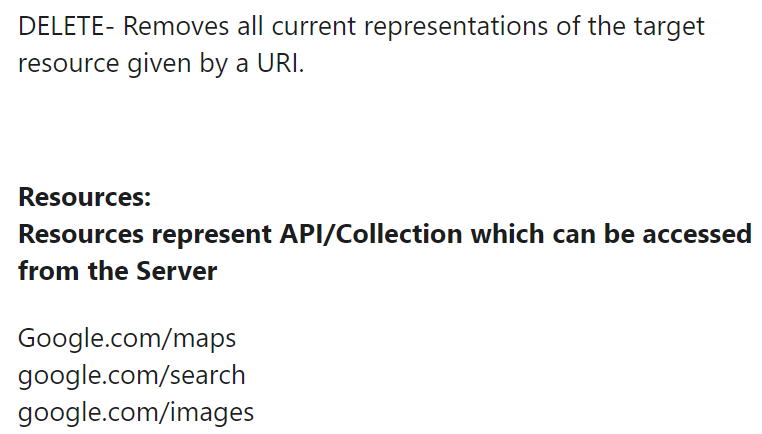
**Patch** : bulk update

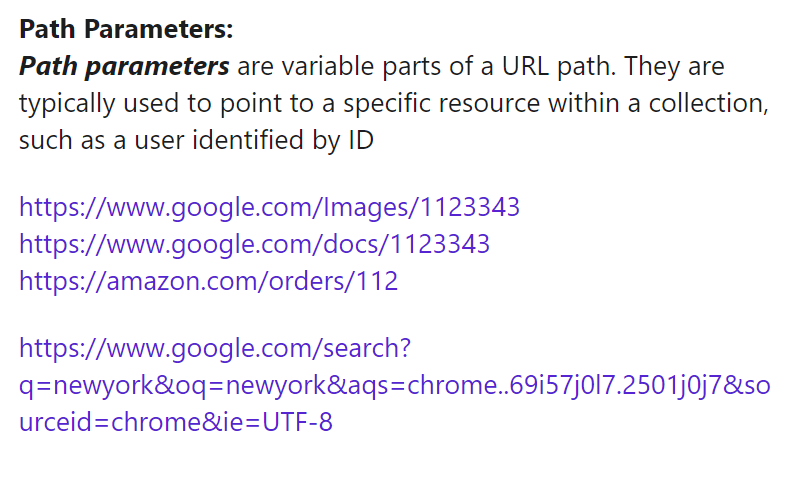
**put** : at time update 1 resource

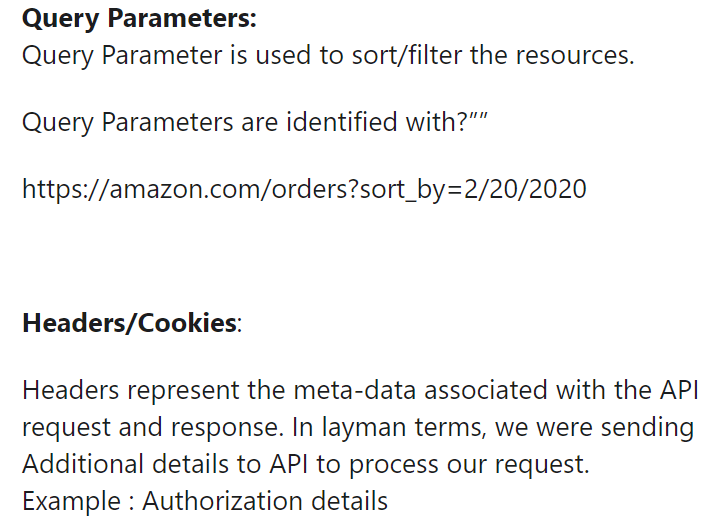
In RestAssured, we use given, when, then for below reasones :



**

**

**

**

**

**Cookies :** A computer cookie consists of information. When you visit a website, the website sends the cookie to your computer. Your computer stores it in a file located inside your web browser. The purpose of the computer cookie is to help the website keep track of your visits and activity.

**Header** : Headers are nothing but additional information we are sending to server. Header may contain Authorization (uname, pwd), cookies, body-type, proxies, additional data for API to work. Request and response both can have associated headers. It possess key-value pairs. We use header() and headers() methods to pass header parameters.

* ***headers()****: returns****Headers***
* ***getHeaders()****: returns****Headers***

Response response=

RestAssured

.*given*()

.get("https://restful-booker.herokuapp.com/booking/1")

.then()

.extract()

.response();

System.***out***.println("All Headers of response are :- ");

Headers allHeaders = response.getHeaders();

**for**(Header header : allHeaders) // PRINT HEADERS

{

System.***out***.print(header.getName() +" : ");

System.***out***.println(header.getValue());

}

System.***out***.println("Value of Header Content-Type : "+response.getHeader("Content-Type"));

// ASSERT HEADERS

String resp = response.getHeaders().getValue("Server");//Capture actual header value

String expectedResp = "Apache/2.4.18 (Ubuntu)";//expected header response

Assert.*assertEquals*(resp,expectedResp);// assert expected equal to actual

few frequently used headers :

1. Authorization : we send credentials in different way
2. Accept : Tells server what is accepted
3. Content type : tells what it contains (generally we use "Content-Type", "application/json")
4. Accept encoding : what kind of encoding we are supporting can mention here
5. Cookies : Sets cookies
6. Server (in response we check this)

Example :

header("Server", "Apache/2.4.18 (Ubuntu)") // checks from which server the response is coming from \*IMP check

RequestSpecBuilder abc = new RequestSpecBuilder();

abc.addHeader("Accept-Language", "en-US");

abc.addHeader("Accept-Encoding", "gzip, deflate, br");

abc.addHeader("amz-sdk-request", "attempt=1; max=3r");

abc.addHeader("Content-Type", "application/x-amz-json-1.1");

RequestSpecification def = abc.build();

RestAssured.given().log().all().header("Content-Type","application/x-amz-json-1.1")

.config(RestAssured.config().encoderConfig(

encoderConfig().encodeContentTypeAs("application/x-amz-json-1.1", ContentType.)))

.spec(def).when().body(request).post(URL).then().log().all();

Few other Realtime headers of our own response:

Expected header "Content-Type" was not "application/json; charset=UTF-8", was "application/json;

charset=UTF-8".

Date=Tue, 14 Sep 2021 08:49:02 GMT

Server=Apache/2.4.18 (Ubuntu)

Access-Control-Allow-Origin=\*

Access-Control-Allow-Methods=POST

Access-Control-Max-Age=3600

Access-Control-Allow-Headers=Content-Type, Access-Control-Allow-Headers, Authorization, X-Requested-With

Content-Length=194

Keep-Alive=timeout=5, max=100

Connection=Keep-Alive

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

**Authentication** :

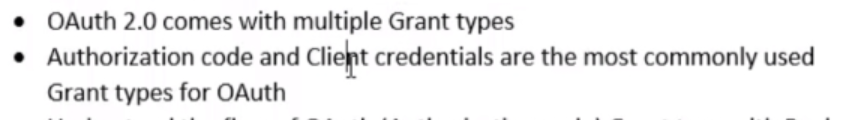
few popular types are :

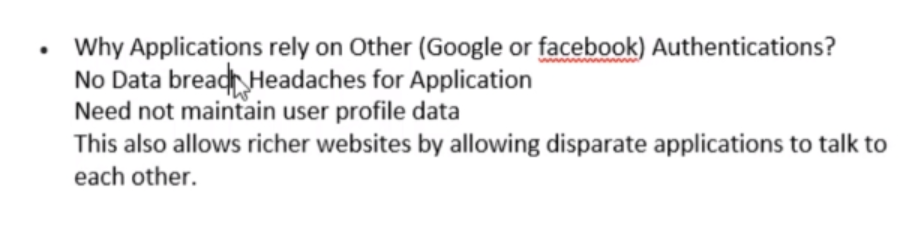
* 1. Basic authentication (use Uname and Pwd)
  2. Digest authentication
  3. Form authentication
  4. Cookie based
  5. OAuth 1 and OAuth 2
* **Basic authentication** (use Uname and Pwd) : Here we it uses base64 to encrypt credentials and then send it over.
  1. Given().auth().preemptive().basic(“username”,”password”).when().get(“URL/end point”).then () .. .. .
  2. Given().auth().basic(“username”,”password”).when().get(“URL/end point”).then () .. .. .
* **Digest** : It is same as basic but it uses other than base64 for encryption
  1. Given().auth().digest(“username”,”password”).when().get(“URL/end point”).then () .. .. .

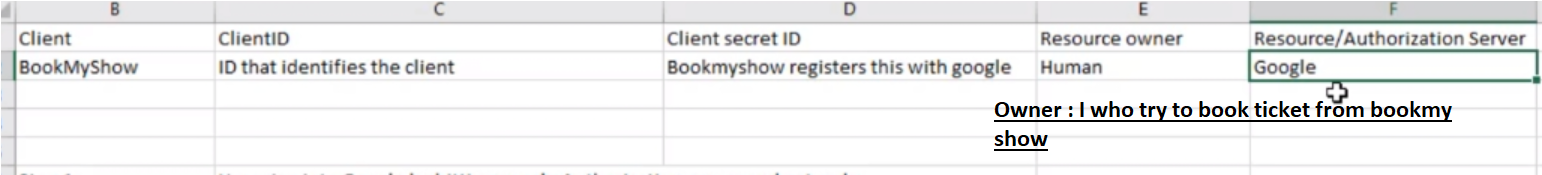
**OAuth 2.0 :** OAuth 2.0 is the industry-standard protocol for authorization. Ex. : If you just hit twitter’s endpoint, you directly can not see every tweet. First we have to authenticate ourselves. If we are a valid and authorized user to see that tweet, then only we are allowed to see the tweet. OAuth comes with multiple grant types. We will cover two types under OAuth 2.0 :

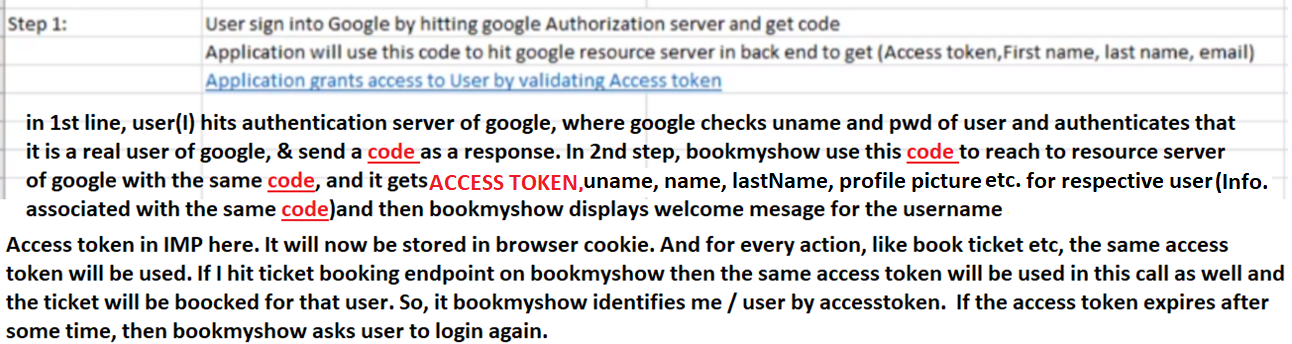
1. **Grant Type : Authorization** (where to use this : on book my show, I(human) login to google and then get redirected to bookmyshow as a loggedin user. Get code and use code to get access token)
2. **Grant type : Client credentials** (where to use this : No human / user involved here. Only two applications communicate with each other. NO code, only works with access token)

**\*\*\*\*\*\*\*\*\*\* Grant Type : Authorization 🡪**

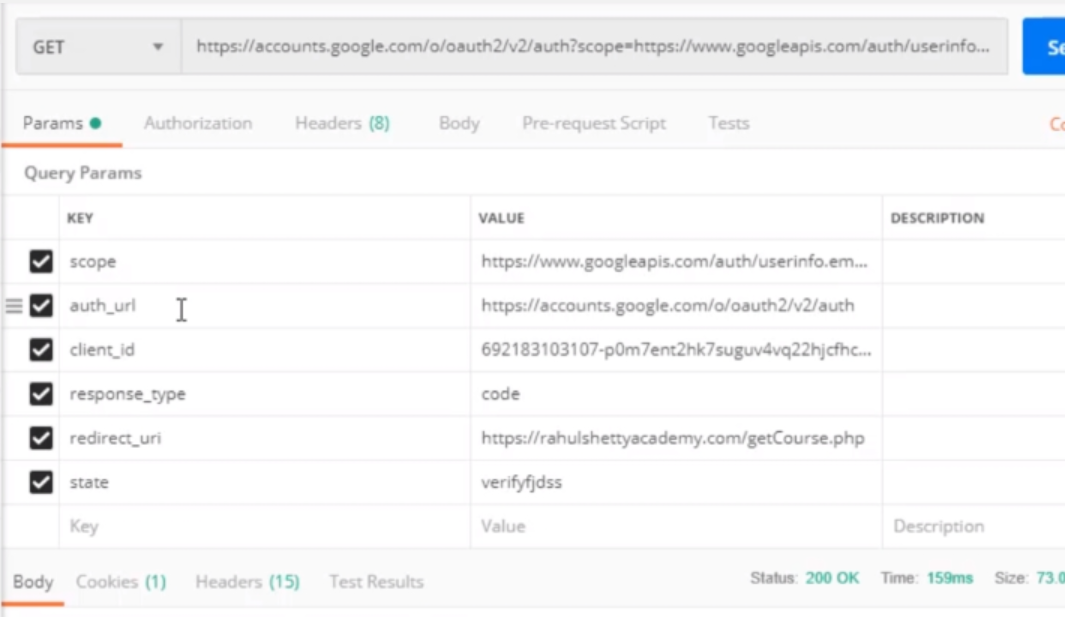








1. **TO GET CODE :** Below are some standard parameters used : 99% developer give us the URL directly (URL to get code). But for understanding purpose, we will see the actual meaning of each section / part / parameter used in the URL.



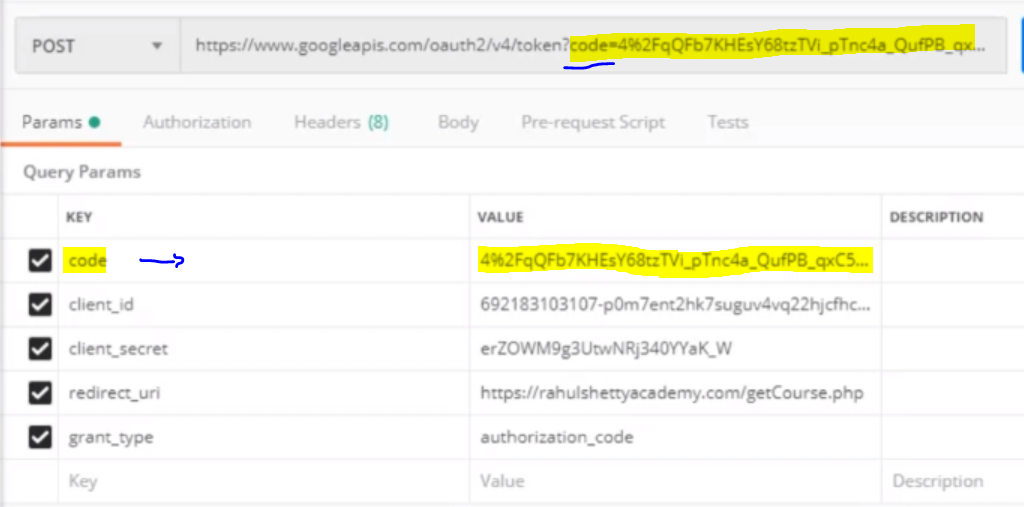
Below info used to form URL. We get URL (URL to get code) from **developer** and these all info also we can get from developer only.

1. Scope : Application / client is asking what all details needed for me (resource owner / I who is logging in)
2. auth\_url : which server we are trying to authorized with (google who authorizes the user)
3. client\_id : When bookmyshow 1st time get registered with google, it gets this id.
4. Response\_type : what response we expect from google (we expect authorization code)
5. Redirect\_url : After login to google, where to go back? (here we come back to bookmyshow)
6. State : it is optional parameter used for security.

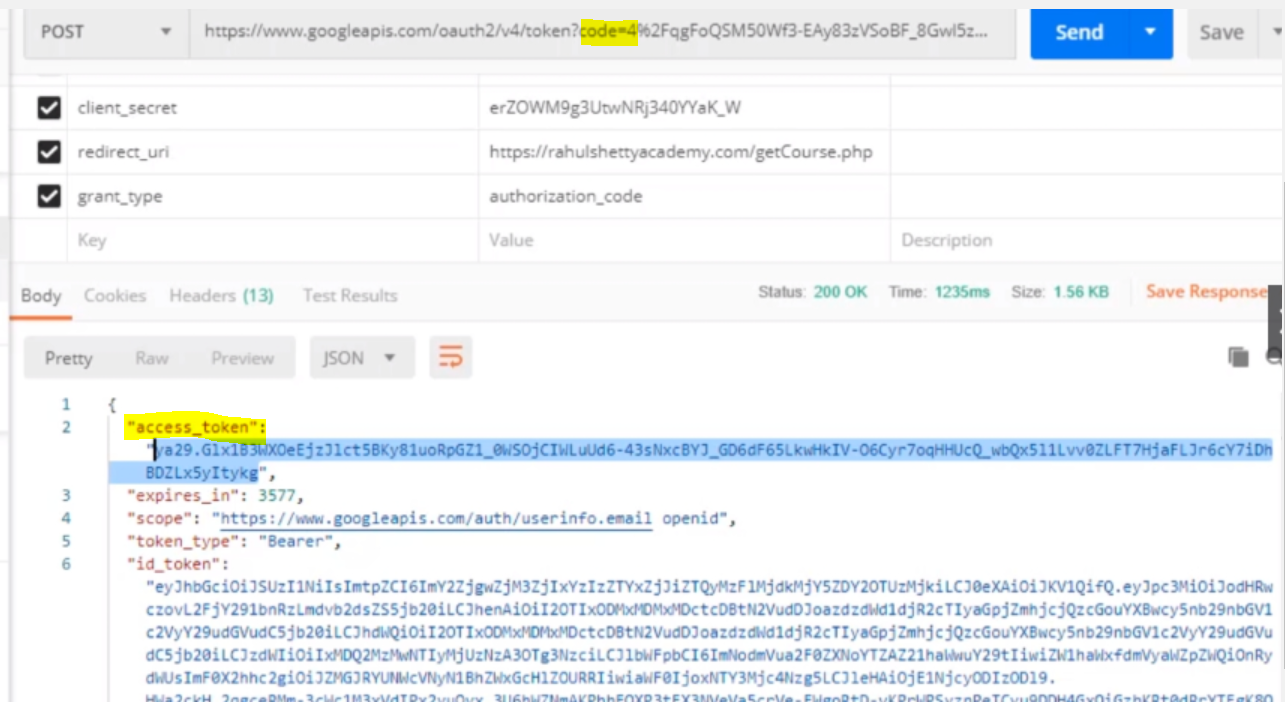
Now get the URL (URL to get code), if we hit the URL, will get the **code.** If we paste it in postman, there also we can see the **code**. It is a **GET** request.

THIS response URL also we get from **DEVELOPER.** So, request and response URLS we get from developers only

1. **Now use the code from above step in 2nd URL to get access token :**



**Now, use the code and add it in 2nd URL given by developer. And hit the 2nd URL after adding the code in it. As a response, we get ACCESS TOKEN. 🡪 it is a POST request**



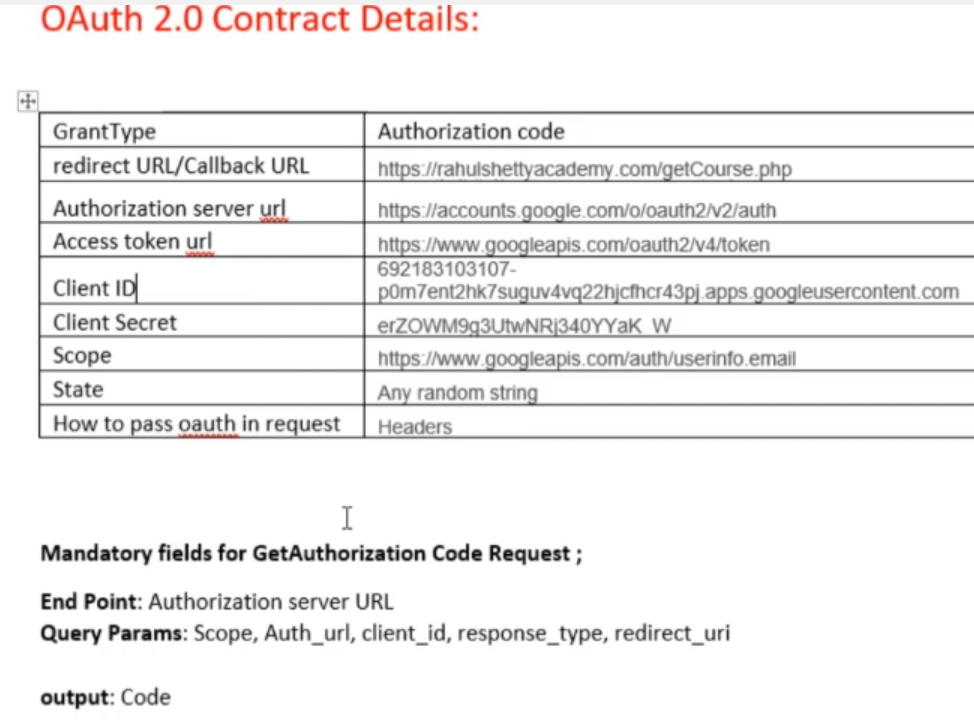
Now, the access token can be used to perform further operations.

**VIPM NOTE** : We get 2 URLs from developers

1. 1st URL after hiting we get a code(it’s a GET request). Need selenium to login and then response URL posses the code.
2. 2nd URL + add above code in it and then hit it > we get access token (it’s a POST request)

Now, use access token for all further operations

Contract document :



Redirect url / call back URL : bookmyshow

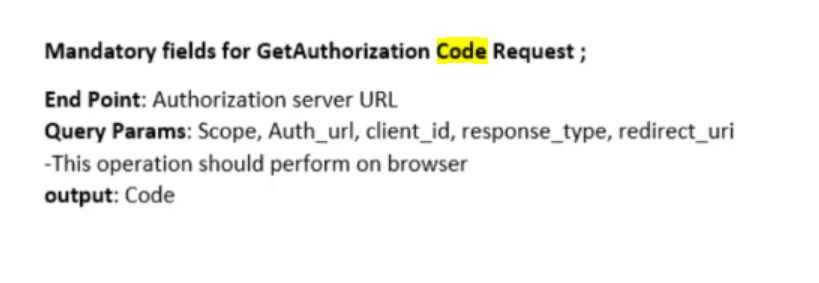
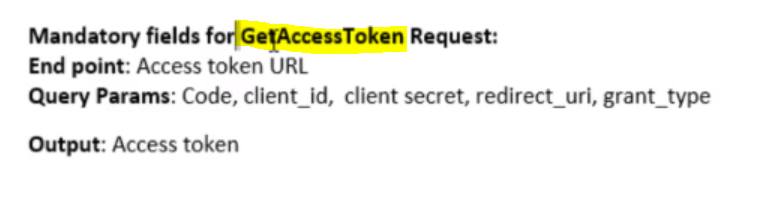
Authorization server url : google [once authorized, we get code and we use the code in below step]

Access token URL : google [use the code from above step to get access toke]

Client id and client secret : unique for client

Scope : what information google shares with us

State : any random string used for security

**VIPM :** 

**Code - Authorization:**

**public** **static** **void** main(String[] args) **throws** InterruptedException {

ChromeOptions options = **new** ChromeOptions();

System.setProperty("webdriver.chrome.driver", "C:\\Users\\Owner\\Pictures\\chromedriver.exe");

WebDriver driver= **new** ChromeDriver(options);

// To login to application by selenium and after login, get the URL and fetch CODE from URL

driver.get("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=692183103107-p0m7ent2hk7suguv4vq22hjcfhcr43pj.apps.googleusercontent.com&response\_type=code&redirect\_uri=https://rahulshettyacademy.com/getCourse.php&state=verifyfjdss");

driver.findElement(By.cssSelector("input[type='email']")).sendKeys("srinath19830@gmail.com");

driver.findElement(By.cssSelector("input[type='email']")).sendKeys(Keys.ENTER);

String url = driver.getCurrentUrl();

String url= "https://rahulshettyacademy.com/getCourse.php?state=verifyfjdss&code=4%2FuwHwSwjuJWi4OqB4h1aPxFt25B92HCYT\_cMwllKg5VmXASkJ7Wczma4OkW53xxhwNSE3QyM4wgvOkYS1DNnAHos&scope=email+https%3A%2F%2Fwww.googleapis.com%2Fauth%2Fuserinfo.email+openid&authuser=1&session\_state=dc2d5c1602dcfa5f73882f98510761055ee744e7..8dd9&prompt=none#";

String partialcode=url.split("code=")[1]; // split URL to get CODE

String code=partialcode.split("&scope")[0];// Final CODE

String accessTokenResponse=

given().urlEncodingEnabled(**false**)// CODE possess special characters, Rest assured performs Encoding operation on special characters. If we use urlEncodingEnabled(false) then it will not decode values from our CODE

.queryParams("code",code)

.queryParams("client\_id","692183103107-p0m7ent2hk7suguv4vq22hjcfhcr43pj.apps.googleusercontent.com")

.queryParams("client\_secret","erZOWM9g3UtwNRj340YYaK\_W")

.queryParams("redirect\_uri","https://rahulshettyacademy.com/getCourse.php")

.queryParams("grant\_type","authorization\_code")

.when().log().all()

.post("https://www.googleapis.com/oauth2/v4/token").asString();

JsonPath js=**new** JsonPath(accessTokenResponse);

String accessToken=js.getString("access\_token");// fetch ACCESS-TOKEN

// Use above access token in next all requests :

String r2= given().contentType("application/json")

.queryParams("access\_token", accessToken).expect().defaultParser(Parser.JSON)//use fetched ACCESS-TOKEN

.when()

.get("https://rahulshettyacademy.com/getCourse.php")

.asString();

// In case of NON-POJO, use below code to verify response body fields

JsonPath jp = **new** JsonPath(getResp);

String capturedAdd = jp.getString("address");

System.out.println(capturedAdd);// we captured the JSON response of GET and we got here the address

Assert.assertEquals(capturedAdd, ExpectedAddress);//we are aserting whether actual and expected address are same

// once the authentication automation is successful, we can use the code like below

// to further verify the response body fields.

// In case of ## pojo, verify like this

System.out.println(gc.getLinkedIn());

System.out.println(gc.getInstructor());

System.out.println(gc.getCourses().getApi().get(1).getCourseTitle());

List<Api> apiCourses=gc.getCourses().getApi();

**for**(**int** i=0;i<apiCourses.size();i++)

{

**if**(apiCourses.get(i).getCourseTitle().equalsIgnoreCase("SoapUI Webservices testing"))

{

System.out.println(apiCourses.get(i).getPrice());

}

}

//Get the course names of WebAutomation

ArrayList<String> a= **new** ArrayList<String>();

List<pojo.WebAutomation> w=gc.getCourses().getWebAutomation();

**for**(**int** j=0;j<w.size();j++)

{

a.add(w.get(j).getCourseTitle());

}

List<String> expectedList= Arrays.asList(courseTitles);

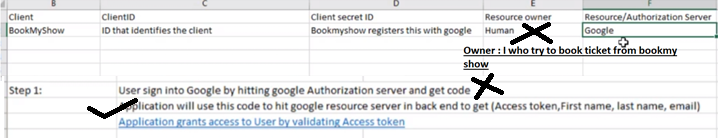
Assert.assertTrue(a.equals(expectedList))

}

}

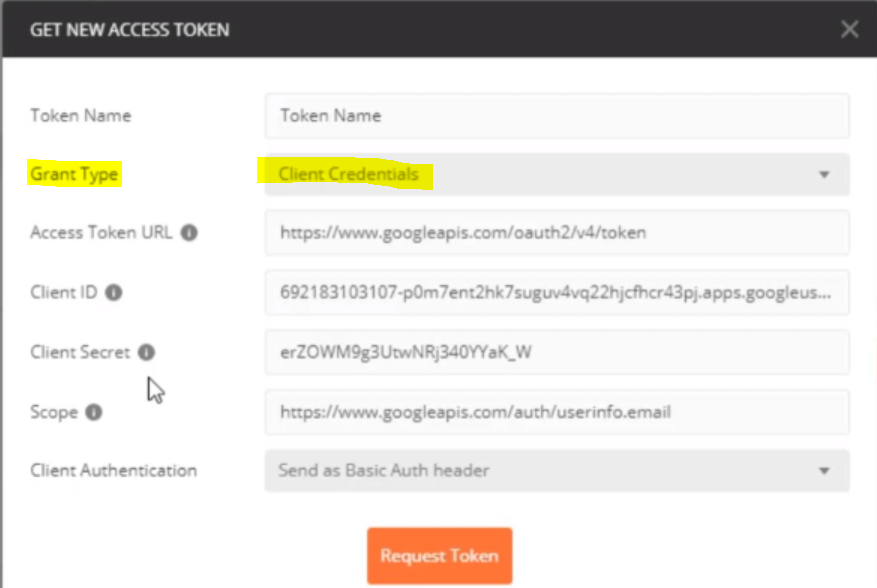
**\*\*\*\*\*\*\*\*\*\*\* Grant type : Client credentialn 🡪**

will be used when two applications communicate, one request data and another one sends the data back.

****

Here, we do not need Resource owner (human), and no CODE is required. Directly we get URL, client id, client secret, from developer / contract doc. We get access token as a response, we use the same access token for all next requests as a authentication mechanism. Token expires after some time(depends on the way developer developed it. Need new token once it is expired.)

Things we need for client credentials grant type:



**CODE – client credentials :**

**public** **static** **void** main(String[] args) {

String accessTokenResponse=

given()

.queryParams("client\_id","692183103107-p0m7ent2hk7suguv4vq22hjcfhcr43pj.apps.googleusercontent.com")

.queryParams("client\_secret","erZOWM9g3UtwNRj340YYaK\_W")

.queryParams("grant\_type","client\_credentials")

.when().log().all() .post("https://www.googleapis.com/oauth2/v4/token").asString();

JsonPath js=**new** JsonPath(accessTokenResponse);

String accessToken=js.getString("access\_token");// fetch ACCESS-TOKEN

// NOW use the access token for next all requests to authenticate and get the response in these 3 ways :

1. String r2= given().contentType("application/json")

.queryParams("access\_token", accessToken).expect().defaultParser(Parser.JSON)//use fetched ACCESS-TOKEN

.when()

.get("https://rahulshettyacademy.com/getCourse.php")

.asString();

// NOW use the access token for next all requests to authenticate and get the response

1. String resp =

given().contentType(ContentType.JSON)

.auth().oauth2(accessToken)//use the ACCESSTOKEN we got in above step

.queryParam("", "")// pass any query param as required

.body("PASS JSON BODY")

.log().all()

.when()

.post("URL")

.then()

.assertThat()

.statusCode(200)

.body("field name", equalTo("Expected value"))

.extract().response().asPrettyString();

// once the authentication automation is successful, we can use the code like below

// to further verify the response body fields.

// In case of NON-POJO, use below code to verify response body fields

1. JsonPath jp = **new** JsonPath(resp);

String capturedAdd = jp.getString("address");

System.out.println(capturedAdd);// we captured the JSON response of GET and we got here the address

Assert.assertEquals(capturedAdd, ExpectedAddress);//we are aserting whether actual and expected address are same

// In case of ## pojo, verify like this

System.out.println(gc.getLinkedIn());

System.out.println(gc.getInstructor());

System.out.println(gc.getCourses().getApi().get(1).getCourseTitle());

List<Api> apiCourses=gc.getCourses().getApi();

**for**(**int** i=0;i<apiCourses.size();i++)

{

**if**(apiCourses.get(i).getCourseTitle().equalsIgnoreCase("SoapUI Webservices testing"))

{

System.out.println(apiCourses.get(i).getPrice());

}

}

//Get the course names of WebAutomation

ArrayList<String> a= **new** ArrayList<String>();

List<pojo.WebAutomation> w=gc.getCourses().getWebAutomation();

**for**(**int** j=0;j<w.size();j++)

{

a.add(w.get(j).getCourseTitle());

}

List<String> expectedList= Arrays.asList(courseTitles);

Assert.assertTrue(a.equals(expectedList))

}

}

**headers** : (mentioned in contract) USED BY **HIRESH**

content type

caller id

corelation id

date and time

**Authorization** :

header : jwt token (Auth 2.0)