REST ASSURED NOTES – BASICS

Introduction to REST API:

What is REST API? REST, which stands for Representational State Transfer, is an architectural style for designing networked applications. A REST API (Application Programming Interface) is a set of rules and conventions for building and interacting with web services that adhere to the principles of REST. It allows systems to communicate over HTTP by defining a set of constraints on how resources are addressed and transferred.

Key Principles of REST:

1. **Statelessness:** Each request from a client to a server must contain all the information needed to understand and fulfill the request. The server should not store any information about the client's state between requests.
2. **Resource-Based:** Resources (such as data or services) are identified by URLs, and they can be manipulated using standard HTTP methods (GET, POST, PUT, DELETE).
3. **Representation:** Resources can have multiple representations, such as JSON or XML. Clients can request a specific representation when interacting with a resource.
4. **Uniform Interface:** A uniform and consistent interface simplifies and decouples the architecture, making it more scalable and maintainable.

Technical Details:

HTTP Methods:

- **GET:** Retrieve a representation of a resource.
- **POST:** Create a new resource.
- **PUT:** Update an existing resource or create a new resource if it doesn't exist.
- **DELETE:** Remove a resource.
- **PATCH:** Partially update a resource.

Status Codes:

- **2xx:** Success
- **3xx:** Redirection
- **4xx:** Client Error
- **5xx:** Server Error

Content Types:

- Common content types include JSON and XML.

Usage:

REST APIs are widely used in various applications, including:

- Web and mobile applications
- IoT (Internet of Things) devices
- Cloud services
- Microservices architectures

Implementation:
REST APIs are extensively implemented globally, powering a significant portion of the modern web and mobile applications. Major tech companies, social media platforms, e-commerce sites, and cloud service providers expose REST APIs for developers to integrate with their services.

Importance:

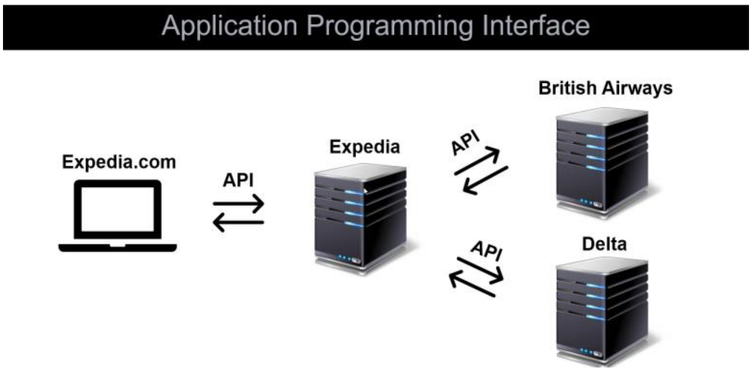
1. **Simplicity:** REST APIs are straightforward and easy to understand, making them accessible to a wide range of developers.
2. **Scalability:** The stateless nature of REST allows for easy scalability, as each request contains all the information needed.
3. **Flexibility:** Clients can request and receive only the data they need, and the server can provide various representations of the same resource.
4. **Interoperability:** REST APIs are platform-independent and can be used with any programming language or framework that supports HTTP.

Practical Examples:

1. **GitHub API:**
    - Allows developers to access and manage GitHub repositories, issues, and user data.
2. **Twitter API:**
    - Provides functionality to interact with Twitter data, including tweets, user profiles, and trends.
3. **Google Maps API:**
    - Enables developers to integrate Google Maps into their applications, accessing features like geolocation, directions, and places.
4. **OpenWeatherMap API:**
    - Allows developers to retrieve weather data for a specific location.
5. **Stripe API:**
    - Facilitates online payment processing, allowing developers to integrate payment functionality into their applications.

Understanding REST API concepts and their practical implementations is crucial for modern web development, enabling seamless integration and communication between different services and platforms.

REST API (Representational State Server) :



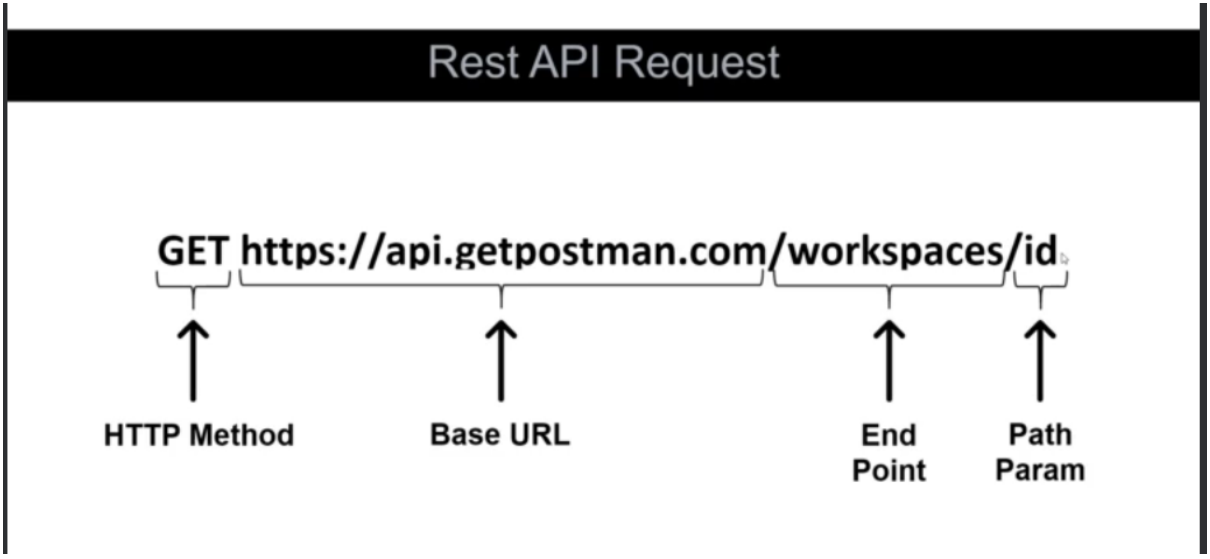
REST 6 Constraints:

- *Client Server,*
- *Stateless,*
- *Cache,*
- *Uniform Interface,*
- *Layered System,*
- *Code on Demand.*

JSON (Javascript Object Notation). It is lightweight, human readable, easy , key-value pairs.

```
{
  "Name": "Test",
  "Mobile": 12345678,
  "Boolean": true,
  "Pets": [
    "Dog",
    "cat"
  ],
  "Address": {
    "Permanent address": "USA",
    "current Address": "AU"
  }
}
```

API Request Format:

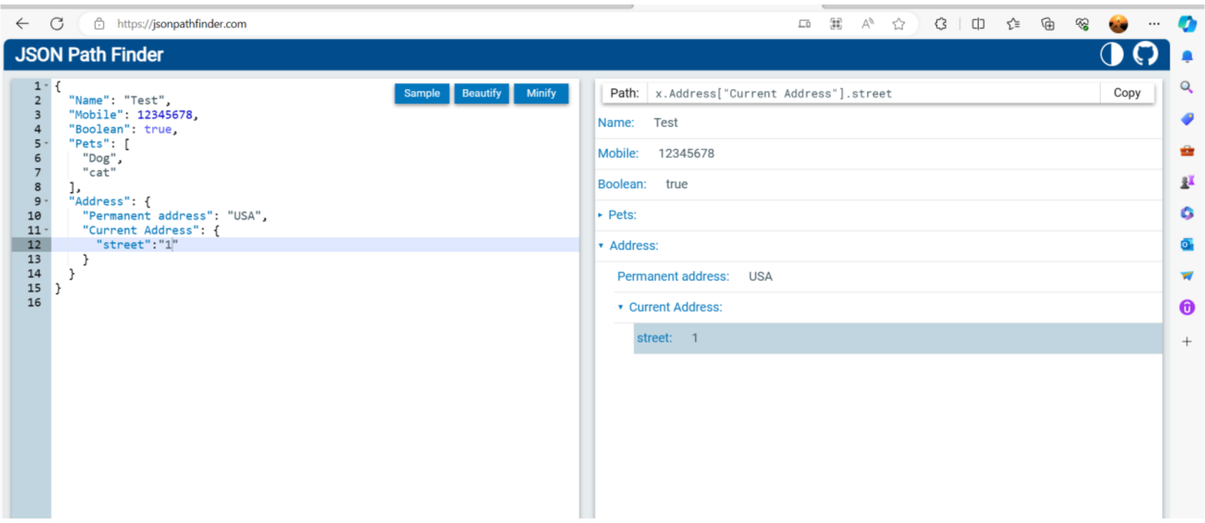


Rest API Request

GET https://api.getpostman.com/workspaces/id

JSON Path:
Groovy GPath , Jayway JsonPath

JSON Path Finder

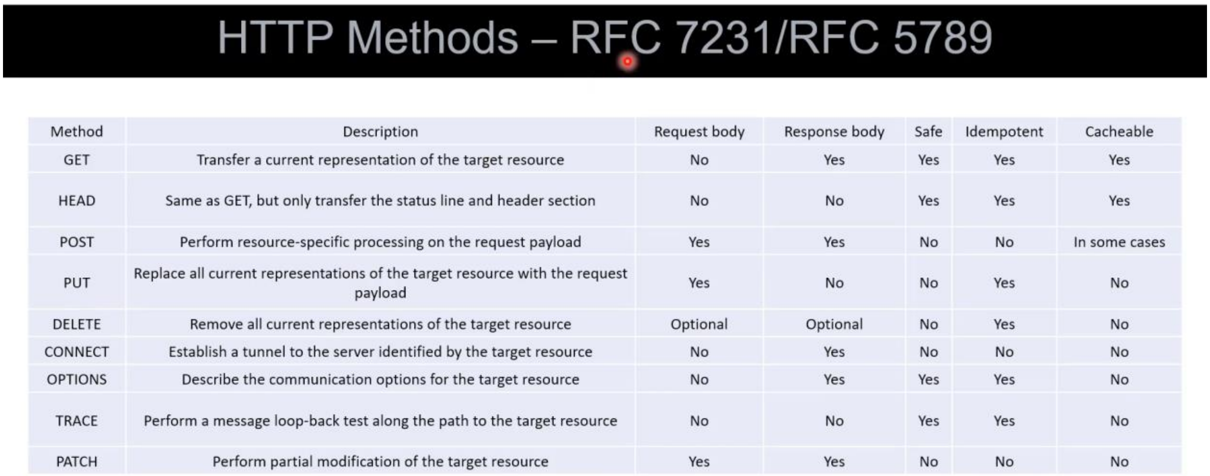


```
{
  "Name": "Test",
  "Mobile": 12345678,
  "Boolean": true,
  "Pets": [
    "Dog",
    "cat"
  ],
  "Address": {
    "Permanent address": "USA",
    "Current Address": {
      "street":"1"
    }
  }
}
```

Path: x.Address["Current Address"].street

Name: Test
Mobile: 12345678
Boolean: true
▸ Pets:
▾ Address:
  Permanent address: USA
  ▾ Current Address:
    street: 1

x.Pets[0]
x.Address["Permanent address"]

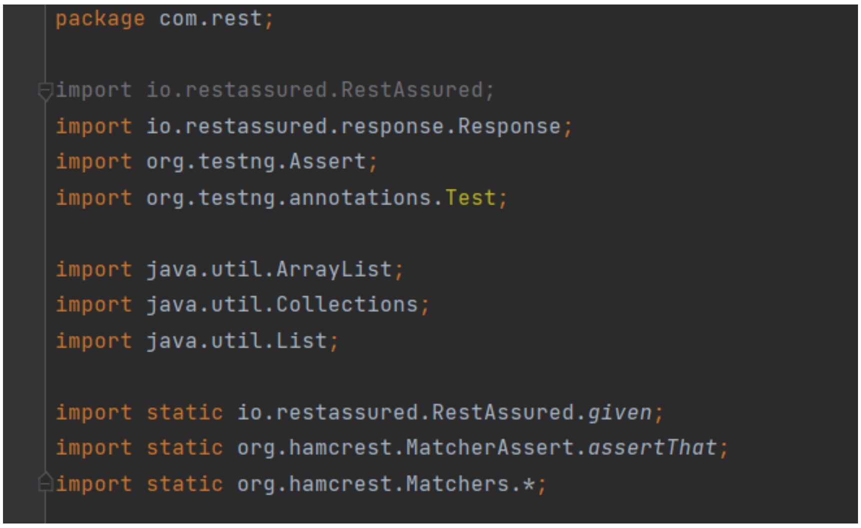


HTTP Methods – RFC 7231/RFC 5789

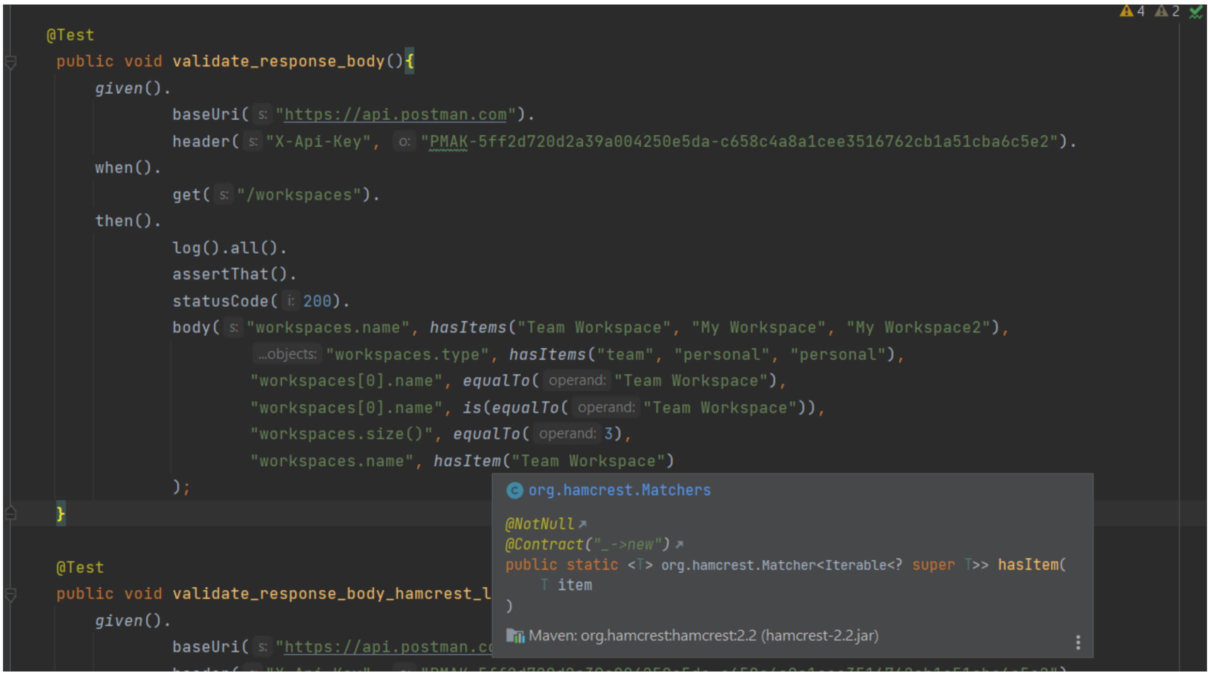
Method	Description	Request body	Response body	Safe	Idempotent	Cacheable
GET	Transfer a current representation of the target resource	No	Yes	Yes	Yes	Yes
HEAD	Same as GET, but only transfer the status line and header section	No	No	Yes	Yes	Yes
POST	Perform resource-specific processing on the request payload	Yes	Yes	No	No	In some cases
PUT	Replace all current representations of the target resource with the request payload	Yes	No	No	Yes	No
DELETE	Remove all current representations of the target resource	Optional	Optional	No	Yes	No
CONNECT	Establish a tunnel to the server identified by the target resource	No	Yes	No	No	No
OPTIONS	Describe the communication options for the target resource	No	Yes	Yes	Yes	No
TRACE	Perform a message loop-back test along the path to the target resource	No	No	Yes	Yes	No
PATCH	Perform partial modification of the target resource	Yes	Yes	No	No	No

REST ASSURED: GETTING STARTED:

Static Imports -> Readability, Reduced Lines:
With the help of Java static import, we can access the static members of a class directly without class name or any object



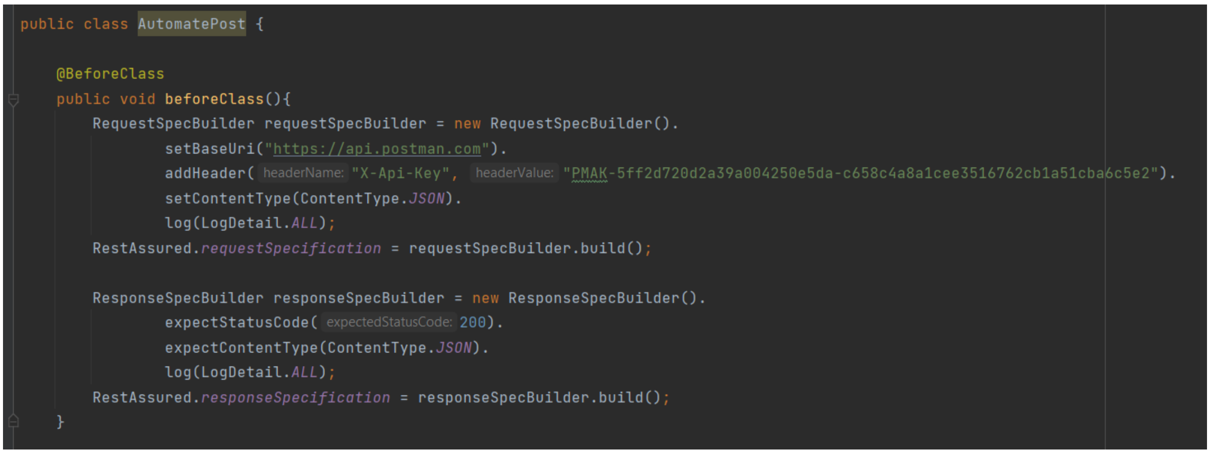
```
package com.rest;

import io.restassured.RestAssured;
import io.restassured.response.Response;
import org.testng.Assert;
import org.testng.annotations.Test;

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

import static io.restassured.RestAssured.given;
import static org.hamcrest.MatcherAssert.assertThat;
import static org.hamcrest.Matchers.*;
```



```
 @Test
 public void validate_response_body(){
     given().
             baseUri( s: "https://api.postman.com").
             header( s: "X-Api-Key",  o: "PMAK-5ff2d720d2a39a004250e5da-c658c4a8a1cee3516762cb1a51cba6c5e2").
     when().
             get( s: "/workspaces").
     then().
             log().all().
             assertThat().
             statusCode( i: 200).
             body( s: "workspaces.name", hasItems("Team Workspace", "My Workspace", "My Workspace2"),
                     ...objects: "workspaces.type", hasItems("team", "personal", "personal"),
                     "workspaces[0].name", equalTo( operand: "Team Workspace"),
                     "workspaces[0].name", is(equalTo( operand: "Team Workspace")),
                     "workspaces.size()", equalTo( operand: 3),
                     "workspaces.name", hasItem("Team Workspace")
             );
 }

 @Test
 public void validate_response_body_hamcrest_l
     given().
             baseUri( s: "https://api.postman.co
```

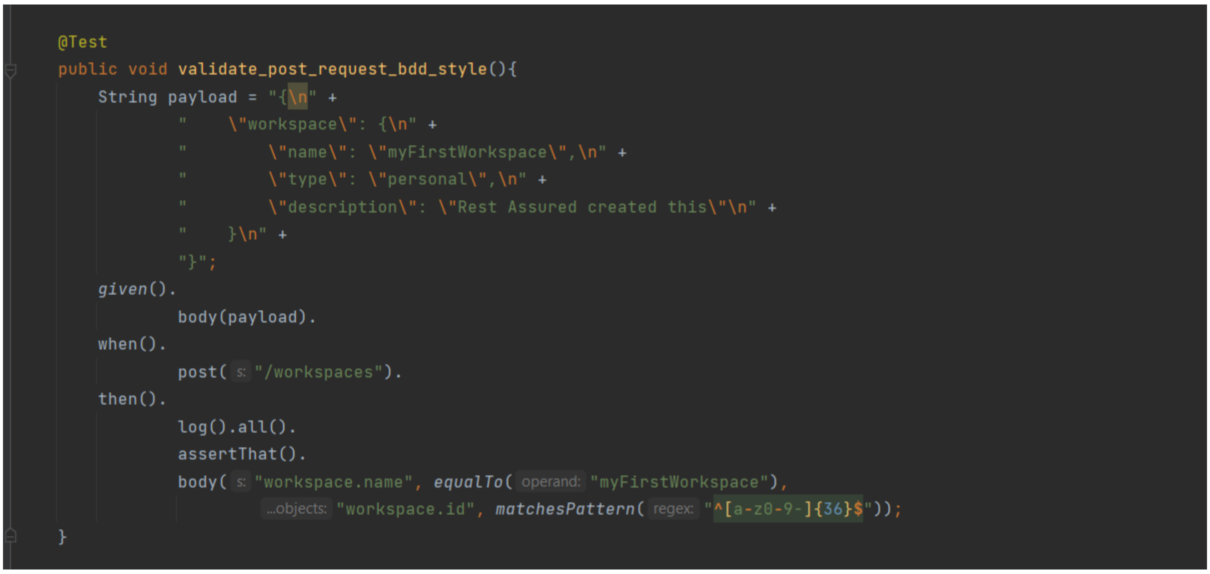
```
org.hamcrest.Matchers

@NotNull
@Contract("_->new")
public static <T> org.hamcrest.Matcher<Iterable<? super T>> hasItem(
    T item
)
Maven: org.hamcrest:hamcrest:2.2 (hamcrest-2.2.jar)
```

Request and Response Specification with Post Request :
Sample Post Spec:



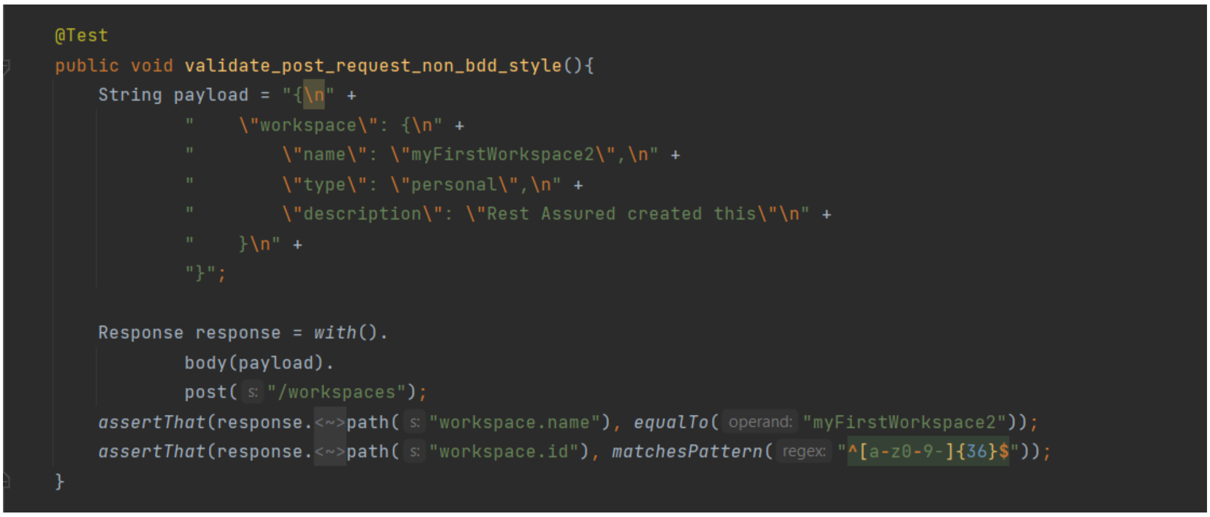
```
public class AutomatePost {

    @BeforeClass
    public void beforeClass(){
        RequestSpecBuilder requestSpecBuilder = new RequestSpecBuilder().
                setBaseUri("https://api.postman.com").
                addHeader( headerName: "X-Api-Key",  headerValue: "PMAK-5ff2d720d2a39a004250e5da-c658c4a8a1cee3516762cb1a51cba6c5e2").
                setContentType(ContentType.JSON).
                log(LogDetail.ALL);
        RestAssured.requestSpecification = requestSpecBuilder.build();

        ResponseSpecBuilder responseSpecBuilder = new ResponseSpecBuilder().
                expectStatusCode( expectedStatusCode: 200).
                expectContentType(ContentType.JSON).
                log(LogDetail.ALL);
        RestAssured.responseSpecification = responseSpecBuilder.build();
    }
```

Sample Test in BDD (Given,When,Then) :



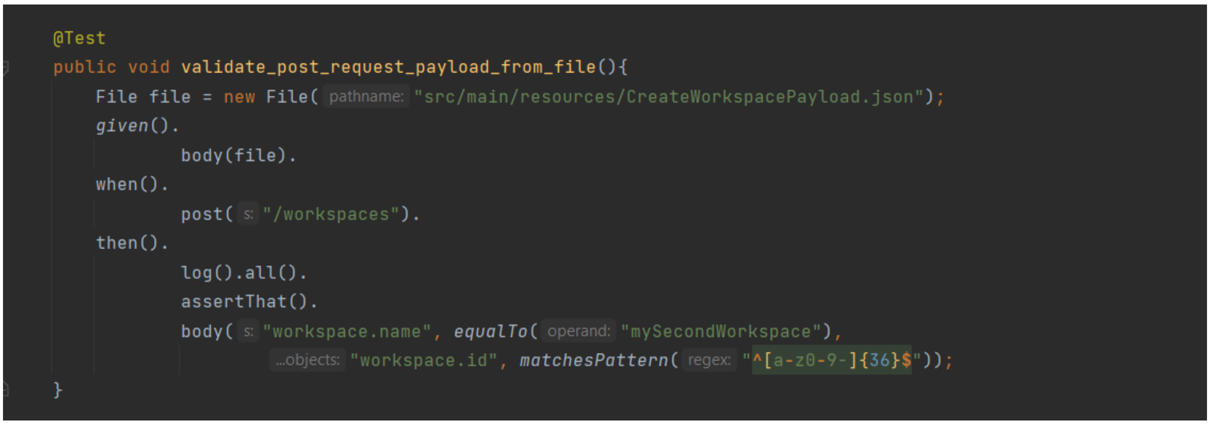
```
    @Test
    public void validate_post_request_bdd_style(){
        String payload = "{\n" +
                "    \"workspace\": {\n" +
                "        \"name\": \"myFirstWorkspace\",\n" +
                "        \"type\": \"personal\",\n" +
                "        \"description\": \"Rest Assured created this\"\n" +
                "    }\n" +
                "}";
        given().
                body(payload).
        when().
                post( s: "/workspaces").
        then().
                log().all().
                assertThat().
                body( s: "workspace.name", equalTo( operand: "myFirstWorkspace"),
                         ...objects: "workspace.id", matchesPattern( regex: "^[a-z0-9-]{36}$"));
    }
```

Sample Test in non BDD : From Request Specification pre-configured.



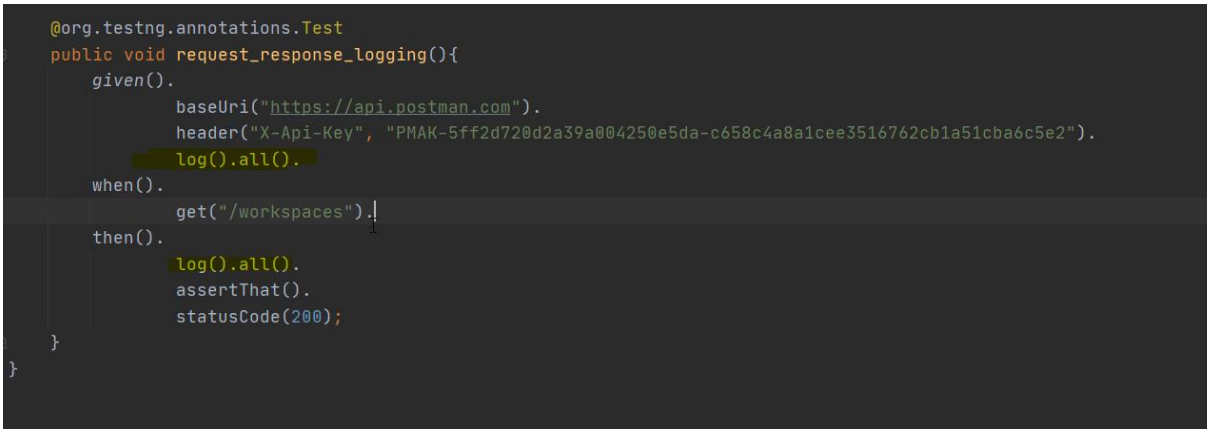
```
@Test
public void validate_post_request_non_bdd_style(){
    String payload = "{\n" +
            "    \"workspace\": {\n" +
            "        \"name\": \"myFirstWorkspace2\",\n" +
            "        \"type\": \"personal\",\n" +
            "        \"description\": \"Rest Assured created this\"\n" +
            "    }\n" +
            "}";

    Response response = with().
            body(payload).
            post( s: "/workspaces");
    assertThat(response.<~>path( s: "workspace.name"), equalTo( operand: "myFirstWorkspace2"));
    assertThat(response.<~>path( s: "workspace.id"), matchesPattern( regex: "^[a-z0-9-]{36}$"));
}
```

Sample Test by passing request body from file. We can also pass values as Collection object such as Map.



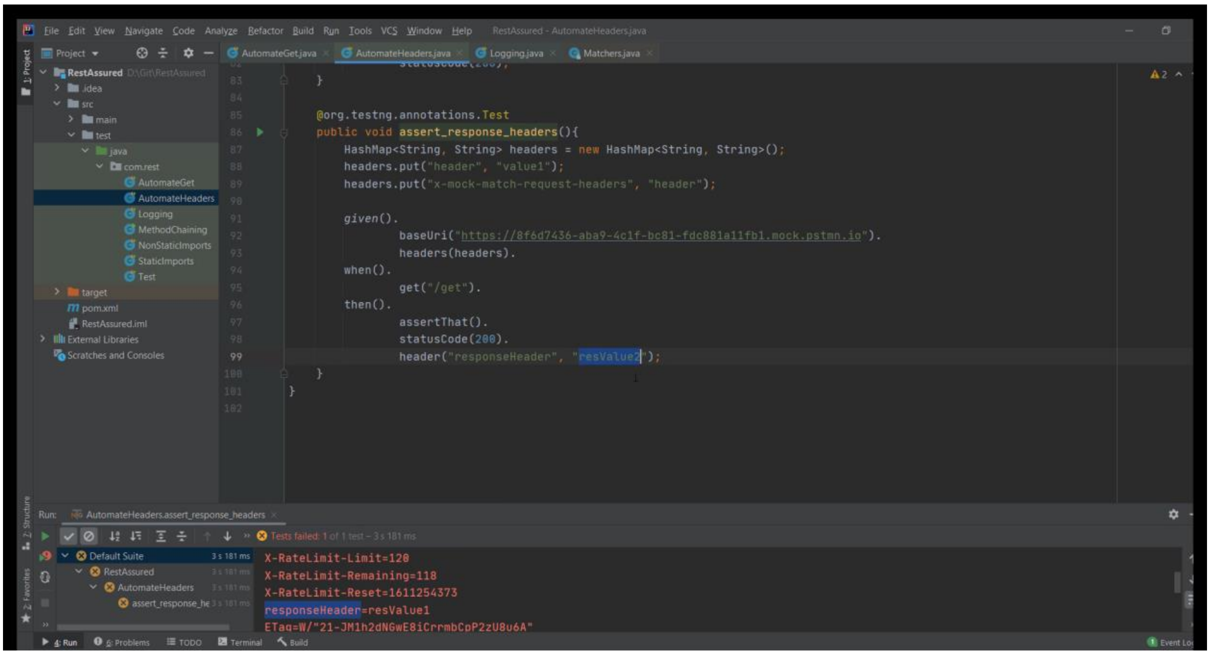
```
@Test
public void validate_post_request_payload_from_file(){
    File file = new File( pathname: "src/main/resources/CreateWorkspacePayload.json");
    given().
            body(file).
    when().
            post( s: "/workspaces").
    then().
            log().all().
            assertThat().
            body( s: "workspace.name", equalTo( operand: "mySecondWorkspace"),
                    ...objects: "workspace.id", matchesPattern( regex: "^[a-z0-9-]{36}$"));
}
```

Rest Assured Enable Logging:



```
@org.testng.annotations.Test
public void request_response_logging(){
    given().
            baseUri("https://api.postman.com").
            header("X-Api-Key", "PMAK-5ff2d720d2a39a004250e5da-c658c4a8a1cee3516762cb1a51cba6c5e2").
            log().all().
    when().
            get("/workspaces").
    then().
            log().all().
            assertThat().
            statusCode(200);
}
}
```

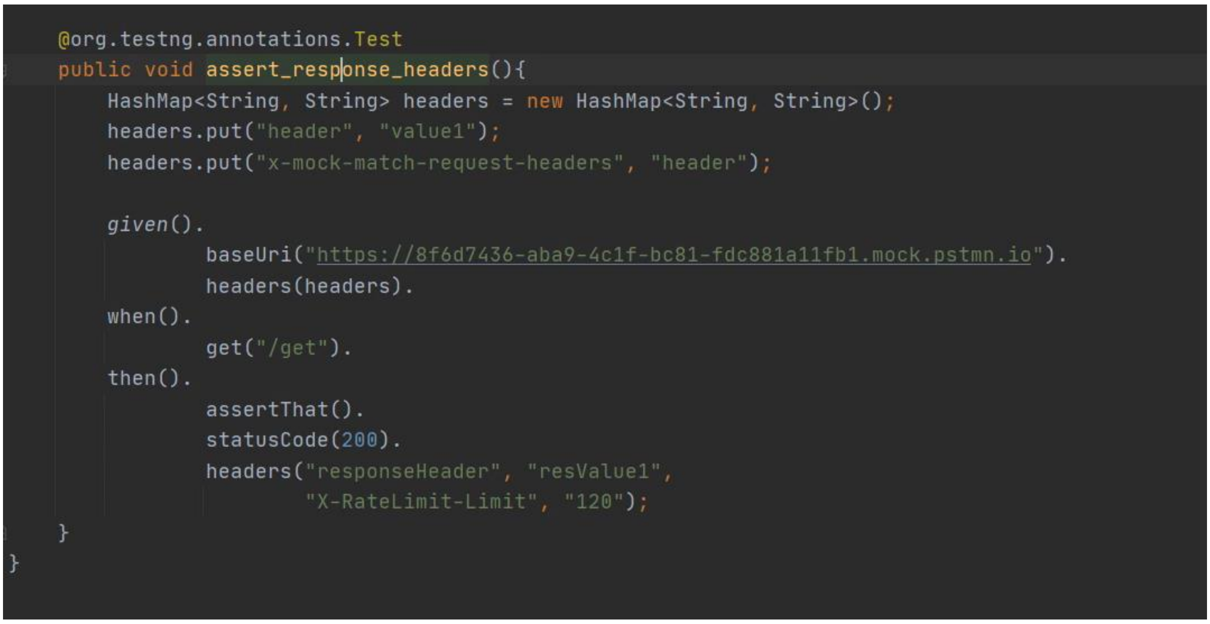
Rest Assured Can Assert Response Headers:



```java
    }

    @org.testng.annotations.Test
    public void assert_response_headers(){
        HashMap<String, String> headers = new HashMap<String, String>();
        headers.put("header", "value1");
        headers.put("x-mock-match-request-headers", "header");

        given().
                baseUri("https://8f6d7436-aba9-4c1f-bc81-fdc881a11fb1.mock.pstmn.io").
                headers(headers).
        when().
                get("/get").
        then().
                assertThat().
                statusCode(200).
                header("responseHeader", "resValue2");
    }
}
```

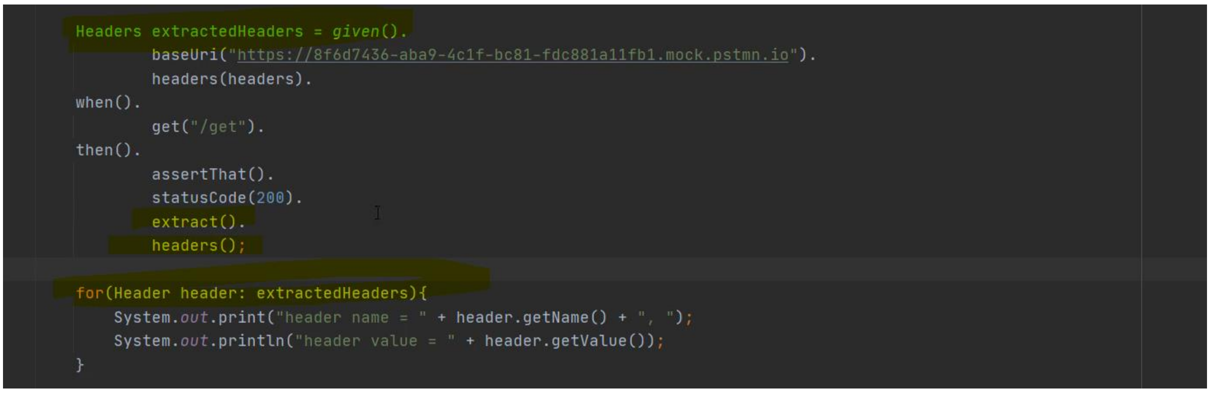
```
X-RateLimit-Limit=120
X-RateLimit-Remaining=118
X-RateLimit-Reset=1611254373
responseHeader=resValue1
ETag=W/"21-JM1h2dNGwE8iCrrmbCpP2zU8u6A"
```

Multiple Response Headers Assertion:



```java
    @org.testng.annotations.Test
    public void assert_response_headers(){
        HashMap<String, String> headers = new HashMap<String, String>();
        headers.put("header", "value1");
        headers.put("x-mock-match-request-headers", "header");

        given().
                baseUri("https://8f6d7436-aba9-4c1f-bc81-fdc881a11fb1.mock.pstmn.io").
                headers(headers).
        when().
                get("/get").
        then().
                assertThat().
                statusCode(200).
                headers("responseHeader", "resValue1",
                        "X-RateLimit-Limit", "120");
    }
}
```

Extract All and Print Response Headers:
Also, We can do the multi value headers too.

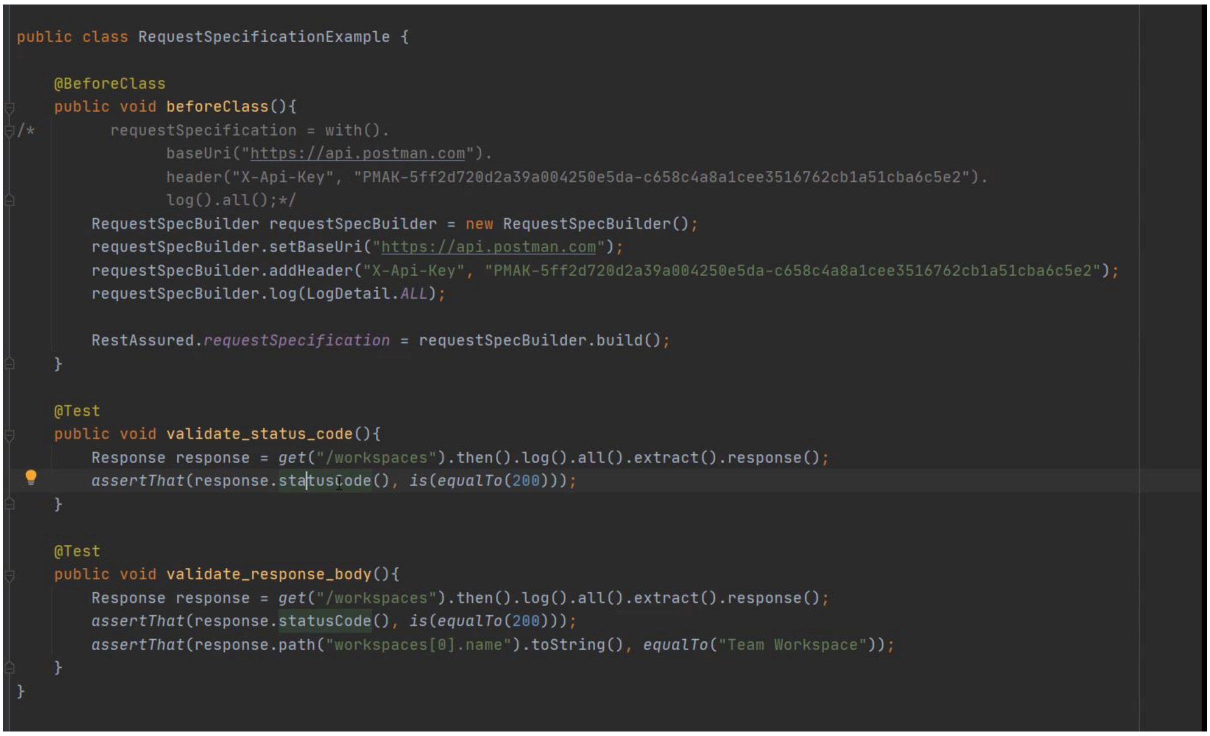


```
Headers extractedHeaders = given().
        baseUri("https://8f6d7436-aba9-4c1f-bc81-fdc881a11fb1.mock.pstmn.io").
        headers(headers).
when().
        get("/get").
then().
        assertThat().
        statusCode(200).
        extract().
        headers();

for(Header header: extractedHeaders){
    System.out.print("header name = " + header.getName() + ", ");
    System.out.println("header value = " + header.getValue());
}
```

Request Specification in Rest Assured:

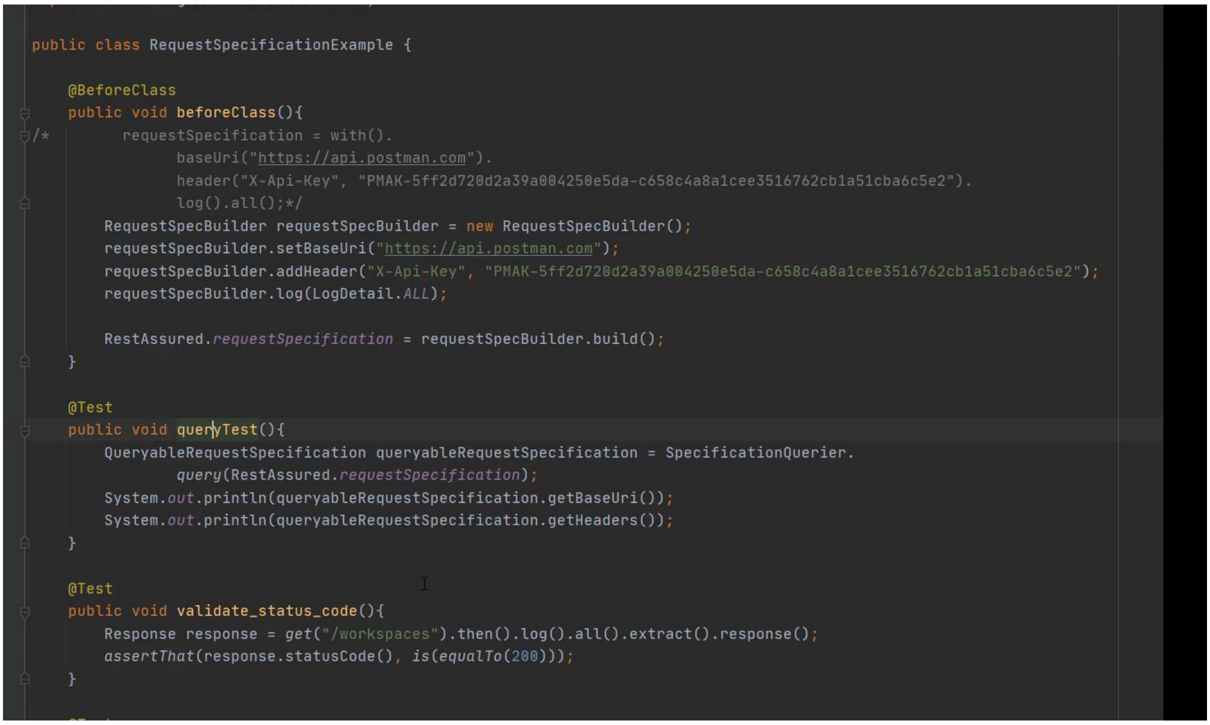
These can be provided in upfront in before class and assign to static variable of RequestSpecification from Rest Assured.

Later, all the requests can be triggered without providing base urls, headers and other repetitive values.



```
public class RequestSpecificationExample {

    @BeforeClass
    public void beforeClass(){
/*          requestSpecification = with().
                baseUri("https://api.postman.com").
                header("X-Api-Key", "PMAK-5ff2d720d2a39a004250e5da-c658c4a8a1cee3516762cb1a51cba6c5e2").
                log().all();*/
        RequestSpecBuilder requestSpecBuilder = new RequestSpecBuilder();
        requestSpecBuilder.setBaseUri("https://api.postman.com");
        requestSpecBuilder.addHeader("X-Api-Key", "PMAK-5ff2d720d2a39a004250e5da-c658c4a8a1cee3516762cb1a51cba6c5e2");
        requestSpecBuilder.log(LogDetail.ALL);

        RestAssured.requestSpecification = requestSpecBuilder.build();
    }

    @Test
    public void validate_status_code(){
        Response response = get("/workspaces").then().log().all().extract().response();
        assertThat(response.statusCode(), is(equalTo(200)));
    }

    @Test
    public void validate_response_body(){
        Response response = get("/workspaces").then().log().all().extract().response();
        assertThat(response.statusCode(), is(equalTo(200)));
        assertThat(response.path("workspaces[0].name").toString(), equalTo("Team Workspace"));
    }
}
```

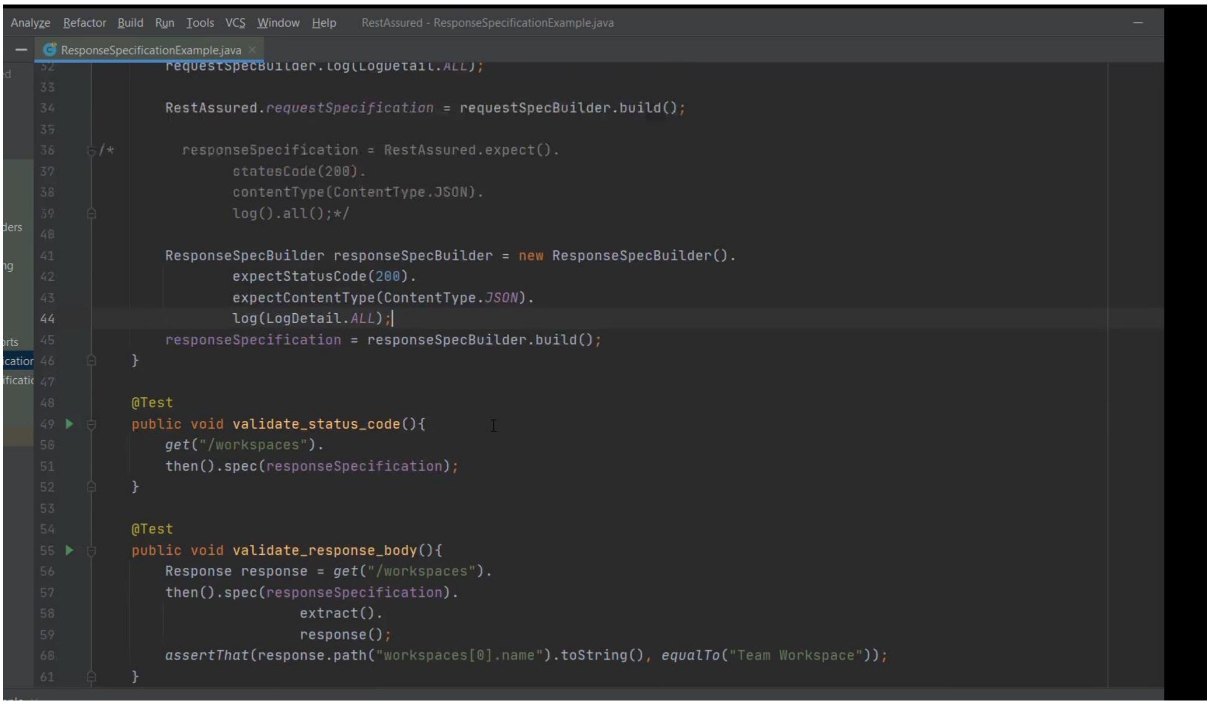
If we want to know what are the request specifications configured in before test, then we have a way to query it in rest assured from class “QueryableRequestSpecification”



```java
public class RequestSpecificationExample {

    @BeforeClass
    public void beforeClass(){
/*        requestSpecification = with().
                baseUri("https://api.postman.com").
                header("X-Api-Key", "PMAK-5ff2d720d2a39a004250e5da-c658c4a8a1cee3516762cb1a51cba6c5e2").
                log().all();*/
        RequestSpecBuilder requestSpecBuilder = new RequestSpecBuilder();
        requestSpecBuilder.setBaseUri("https://api.postman.com");
        requestSpecBuilder.addHeader("X-Api-Key", "PMAK-5ff2d720d2a39a004250e5da-c658c4a8a1cee3516762cb1a51cba6c5e2");
        requestSpecBuilder.log(LogDetail.ALL);

        RestAssured.requestSpecification = requestSpecBuilder.build();
    }

    @Test
    public void queryTest(){
        QueryableRequestSpecification queryableRequestSpecification = SpecificationQuerier.
                query(RestAssured.requestSpecification);
        System.out.println(queryableRequestSpecification.getBaseUri());
        System.out.println(queryableRequestSpecification.getHeaders());
    }

    @Test
    public void validate_status_code(){
        Response response = get("/workspaces").then().log().all().extract().response();
        assertThat(response.statusCode(), is(equalTo(200)));
    }
```

Rest-Assured : Response Specification:

Add common response specification in before class and use them in test as "then().spec(responseSpecObject)"


Analyze Refactor Build Run Tools VCS Window Help
RestAssured - ResponseSpecificationExample.java
ResponseSpecificationExample.java

```java
        requestSpecBuilder.log(LogDetail.ALL);

        RestAssured.requestSpecification = requestSpecBuilder.build();

/*        responseSpecification = RestAssured.expect().
                statusCode(200).
                contentType(ContentType.JSON).
                log().all();*/

        ResponseSpecBuilder responseSpecBuilder = new ResponseSpecBuilder().
                expectStatusCode(200).
                expectContentType(ContentType.JSON).
                log(LogDetail.ALL);
        responseSpecification = responseSpecBuilder.build();
    }

    @Test
    public void validate_status_code(){
        get("/workspaces").
        then().spec(responseSpecification);
    }

    @Test
    public void validate_response_body(){
        Response response = get("/workspaces").
        then().spec(responseSpecification).
                    extract().
                    response();
        assertThat(response.path("workspaces[0].name").toString(), equalTo("Team Workspace"));
    }
```

Encoding and Decoding:

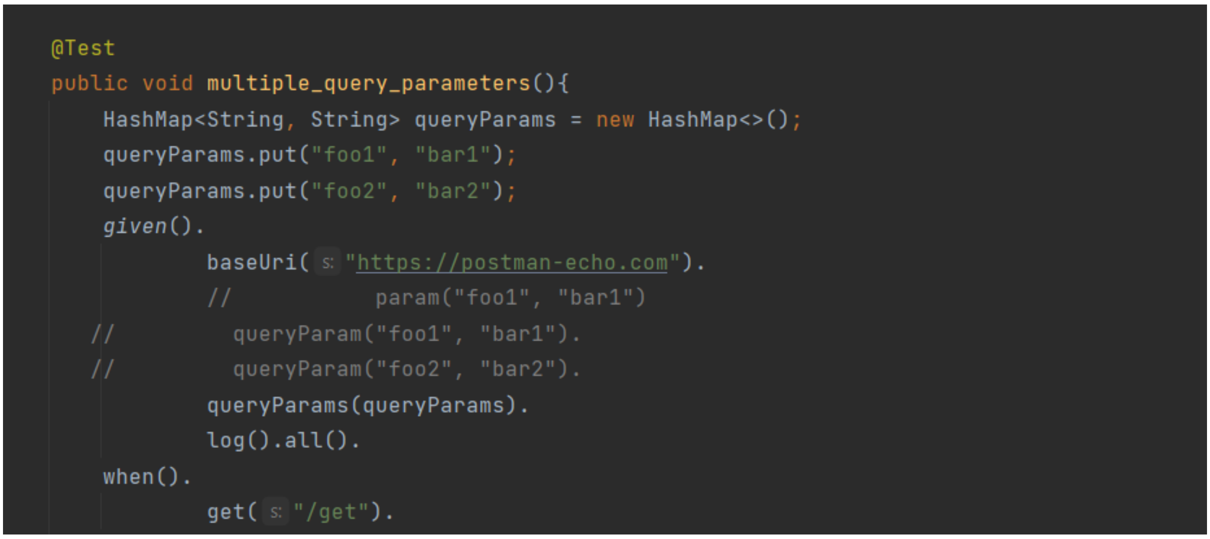
By default encoding added in rest assured is UTF-8. This may cause request failures if they are not configured properly.

https://github.com/rest-assured/rest-assured/wiki/Usage#encoder-config

RestAssured.config = RestAssured.config(config().encoderConfig(encoderConfig().defaultCharsetForContentType(" UTF-16", "application/xml")));

Query Parameter:

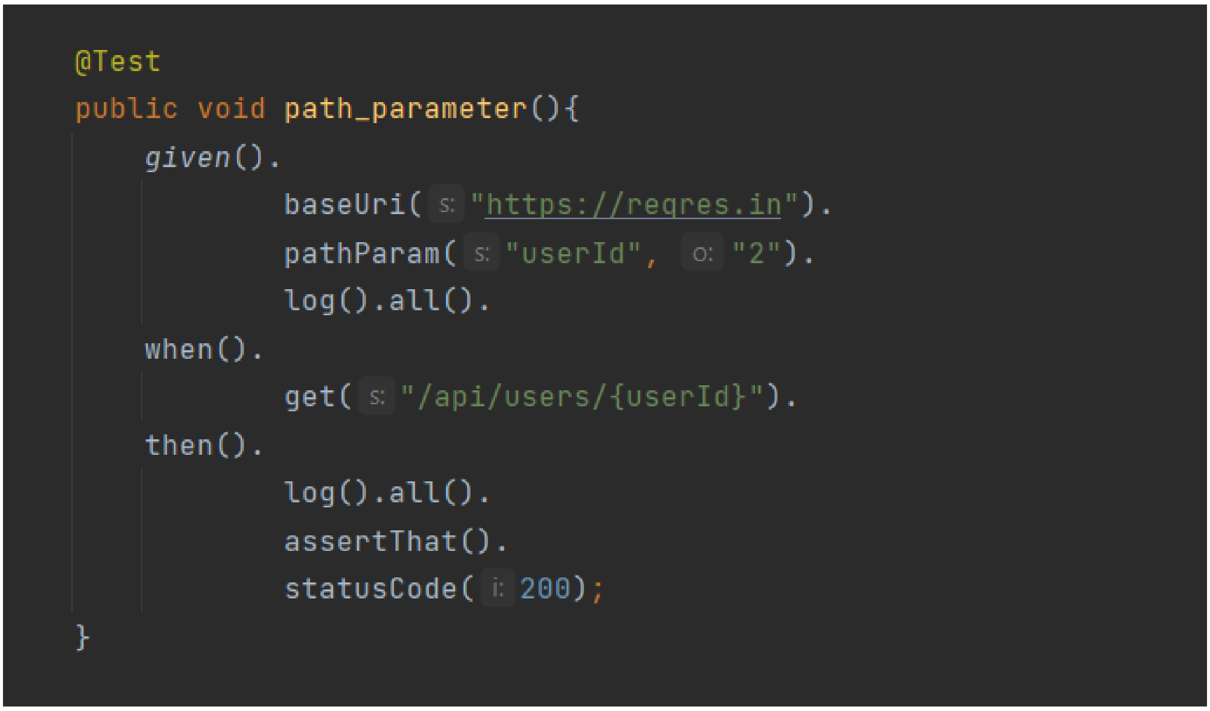
We can use query parameters to control what data is returned in endpoint resources. It appears at the end of the URL after the question mark (?) and helps us to control the set of items and properties in responses, and the order of the items returned.
Consider the following GitHub API URL:
https://api.github.com/user/repos?sort=created&direction=desc
This will list all repositories for an authenticated user but the response properties will be sorted by repository created and in descending order.



```
@Test
public void multiple_query_parameters(){
    HashMap<String, String> queryParams = new HashMap<>();
    queryParams.put("foo1", "bar1");
    queryParams.put("foo2", "bar2");
    given().
            baseUri( s: "https://postman-echo.com").
            //           param("foo1", "bar1")
   //         queryParam("foo1", "bar1").
   //         queryParam("foo2", "bar2").
            queryParams(queryParams).
            log().all().
    when().
            get( s: "/get").
```

Path Parameter:

Path parameters are variables in a URL path. They are used to point to a specific resource within a collection. We can define multiple PATH parameters and each of them is represented by a curly brace {}.
Consider the following GitHub API url:
https://api.github.com/users/:username/repos
This will list all public repositories for the specified user with the value username.
:username is the Path parameter in the above url.

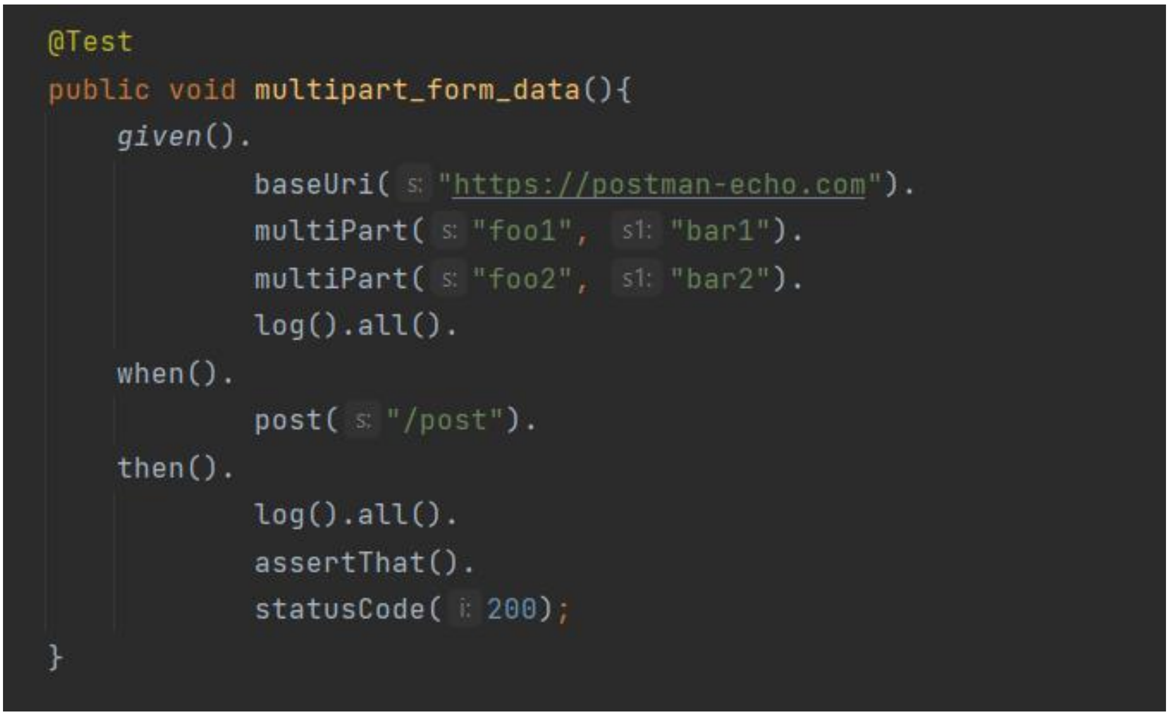


```
@Test
public void path_parameter(){
    given().
            baseUri( s: "https://reqres.in").
            pathParam( s: "userId",  o: "2").
            log().all().
    when().
            get( s: "/api/users/{userId}").
    then().
            log().all().
            assertThat().
            statusCode( i: 200);
}
```

Form Data:

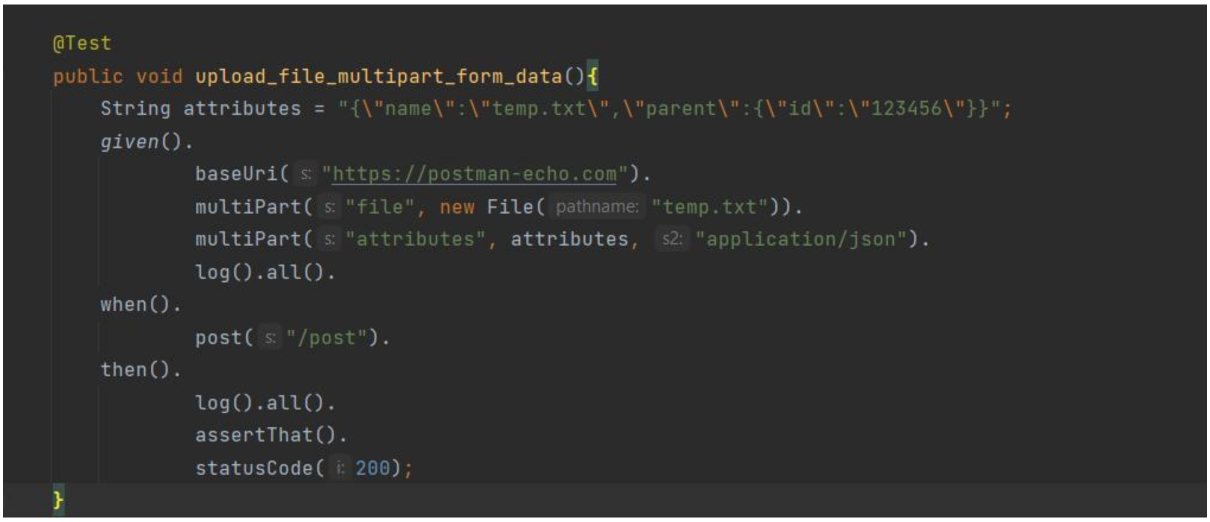
Form-data is one of the formats for data sent from a web form. Specifically, it encodes values entered into a form as name-value pairs and sends them with the Content-Type header set to multipart/form-data. The main features of form-data include:

- *Ability to send not only text but also files.*
- *Ability to split and send the transmitted data into parts.*
- *Ability to specify the content type for each part.*



```
@Test
public void multipart_form_data(){
    given().
            baseUri( s: "https://postman-echo.com").
            multiPart( s: "foo1",  s1: "bar1").
            multiPart( s: "foo2",  s1: "bar2").
            log().all().
    when().
            post( s: "/post").
    then().
            log().all().
            assertThat().
            statusCode( i: 200);
}
```

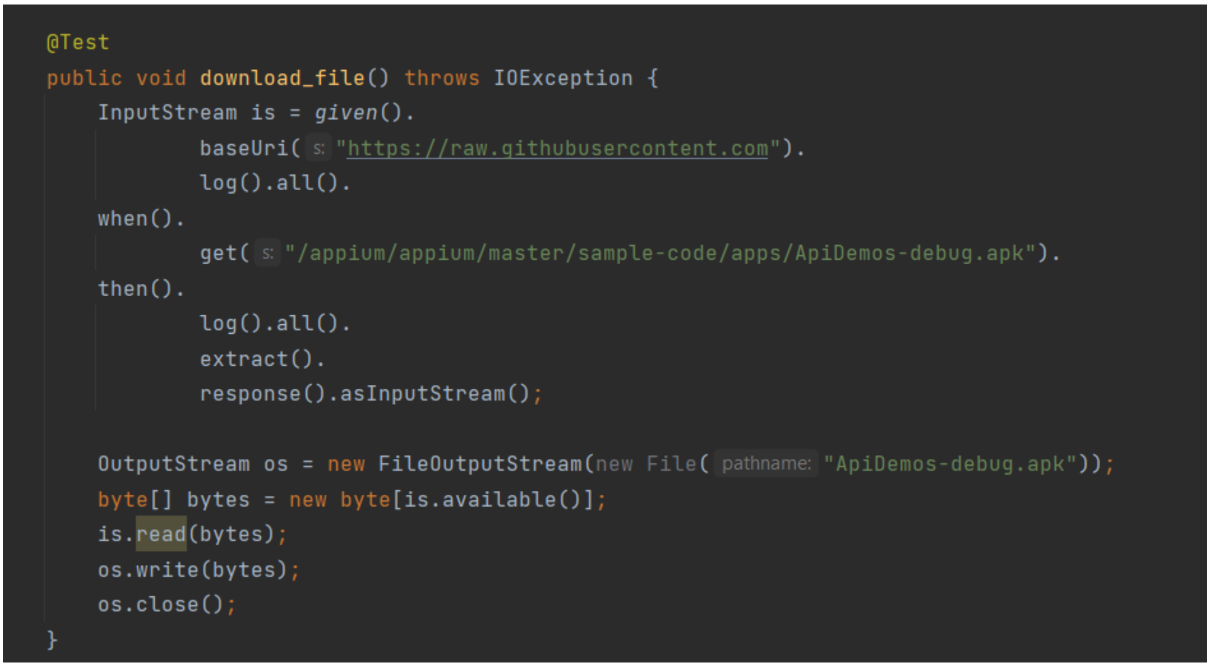
Upload File using Multi-Part Form Data:



```
@Test
public void upload_file_multipart_form_data(){
    String attributes = "{\"name\":\"temp.txt\",\"parent\":{\"id\":\"123456\"}}";
    given().
            baseUri( s: "https://postman-echo.com").
            multiPart( s: "file", new File( pathname: "temp.txt")).
            multiPart( s: "attributes", attributes,  s2: "application/json").
            log().all().
    when().
            post( s: "/post").
    then().
            log().all().
            assertThat().
            statusCode( i: 200);
}
```

Download File using get request :

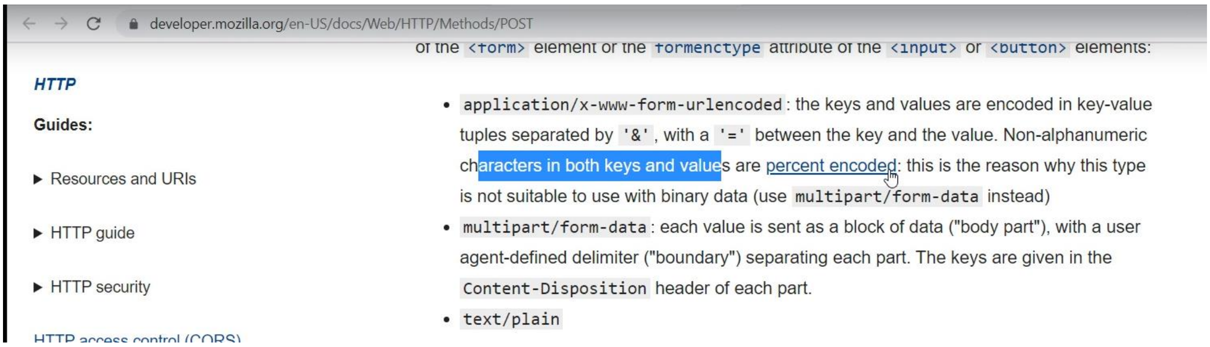
receive response as byte array or Input stream:



```
@Test
public void download_file() throws IOException {
    InputStream is = given().
            baseUri( s: "https://raw.githubusercontent.com").
            log().all().
    when().
            get( s: "/appium/appium/master/sample-code/apps/ApiDemos-debug.apk").
    then().
            log().all().
            extract().
            response().asInputStream();

    OutputStream os = new FileOutputStream(new File( pathname: "ApiDemos-debug.apk"));
    byte[] bytes = new byte[is.available()];
    is.read(bytes);
    os.write(bytes);
    os.close();
}
```

URL Encoded Form:

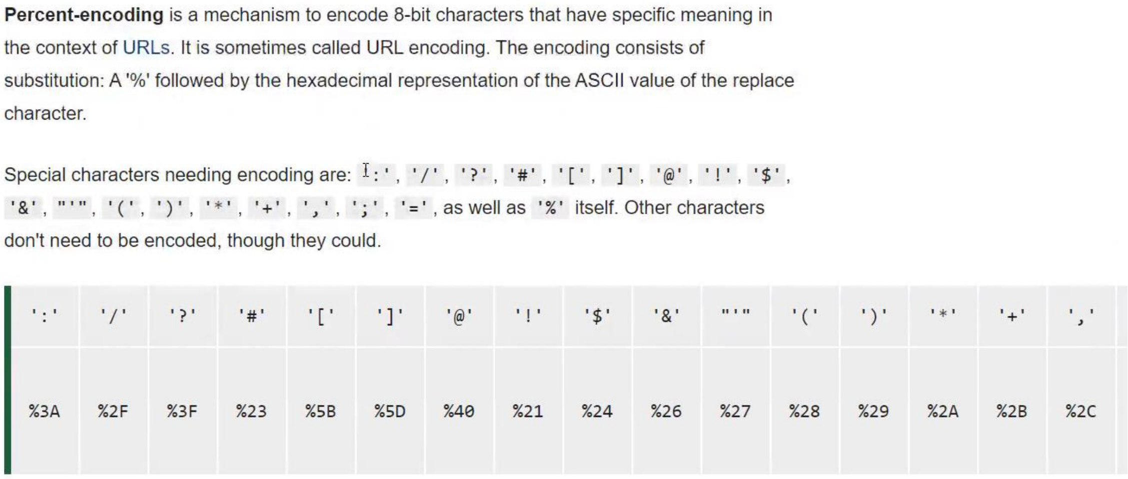
form data key value pairs provided in request with encoding as mentioned below


developer.mozilla.org/en-US/docs/Web/HTTP/Methods/POST
HTTP
Guides:
▶ Resources and URIs
▶ HTTP guide
▶ HTTP security
HTTP access control (CORS)

of the `<form>` element or the `formenctype` attribute of the `<input>` or `<button>` elements:

- `application/x-www-form-urlencoded`: the keys and values are encoded in key-value tuples separated by `'&'`, with a `'='` between the key and the value. Non-alphanumeric characters in both keys and values are percent encoded: this is the reason why this type is not suitable to use with binary data (use `multipart/form-data` instead)
- `multipart/form-data`: each value is sent as a block of data ("body part"), with a user agent-defined delimiter ("boundary") separating each part. The keys are given in the `Content-Disposition` header of each part.
- `text/plain`

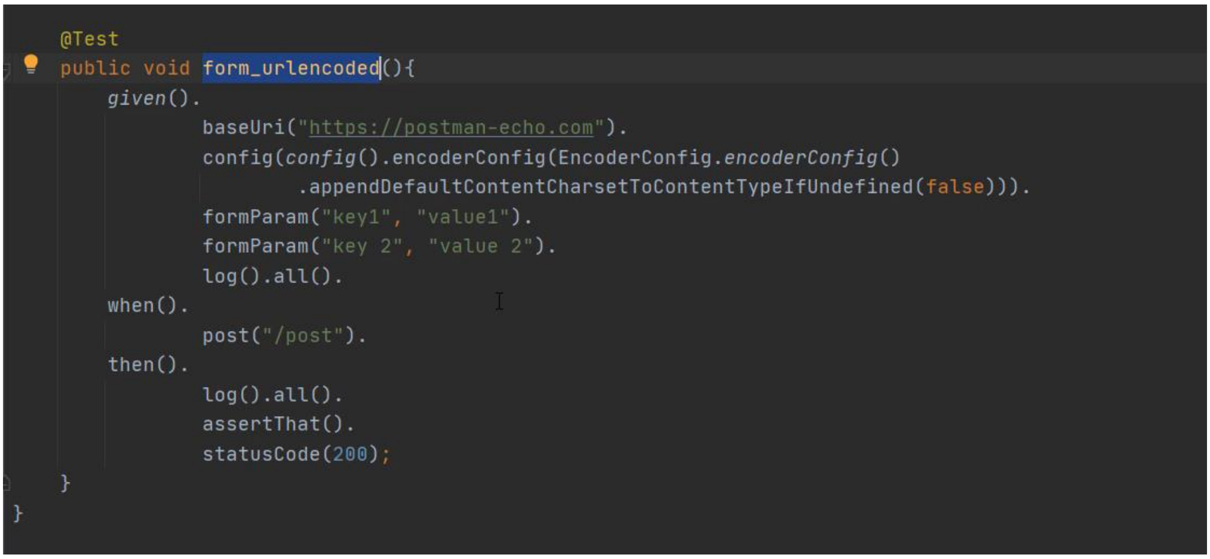
Percent Encoded Non-Alphanumeric :



Percent-encoding is a mechanism to encode 8-bit characters that have specific meaning in the context of URLs. It is sometimes called URL encoding. The encoding consists of substitution: A '%' followed by the hexadecimal representation of the ASCII value of the replace character.

Special characters needing encoding are: `':'`, `'/'`, `'?'`, `'#'`, `'['`, `']'`, `'@'`, `'!'`, `'$'`, `'&'`, `"'"`, `'('`, `')'`, `'*'`, `'+'`, `','`, `';'`, `'='`, as well as `'%'` itself. Other characters don't need to be encoded, though they could.

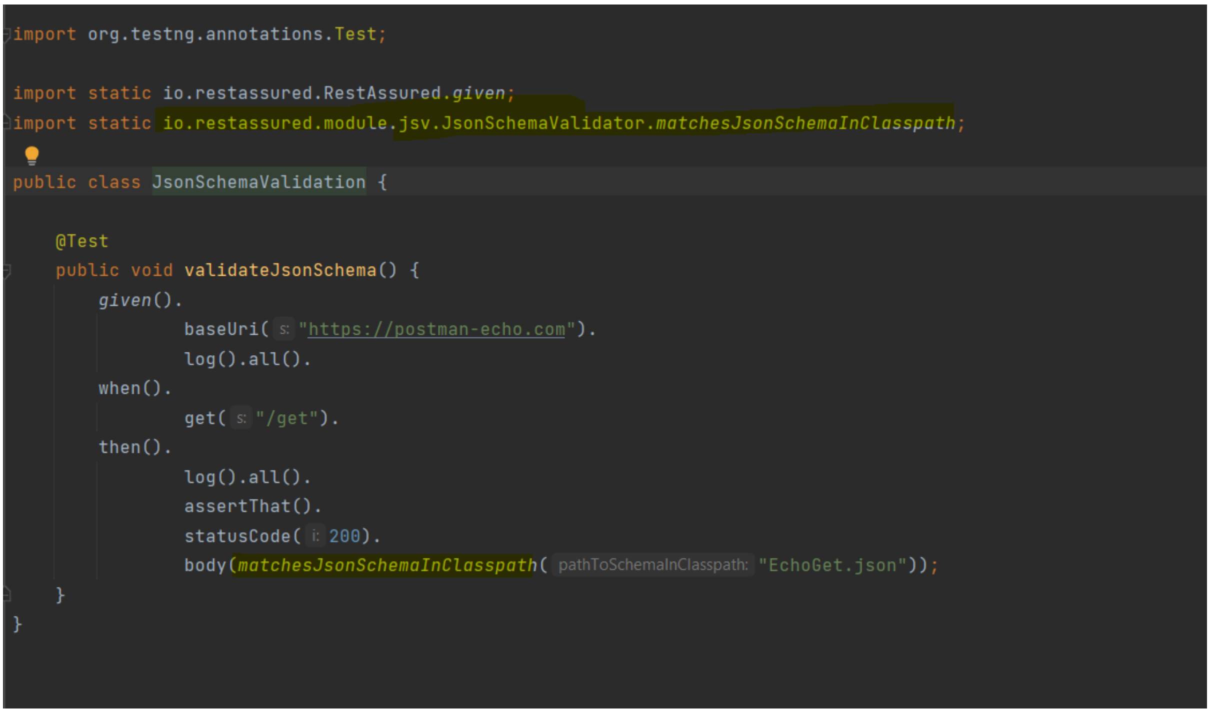
':'	'/'	'?'	'#'	'['	']'	'@'	'!'	'$'	'&'	"'"	'('	')'	'*'	'+'	','
%3A	%2F	%3F	%23	%5B	%5D	%40	%21	%24	%26	%27	%28	%29	%2A	%2B	%2C



```java
    @Test
    public void form_urlencoded(){
        given().
                baseUri("https://postman-echo.com").
                config(config().encoderConfig(EncoderConfig.encoderConfig()
                        .appendDefaultContentCharsetToContentTypeIfUndefined(false))).
                formParam("key1", "value1").
                formParam("key 2", "value 2").
                log().all().
        when().
                post("/post").
        then().
                log().all().
                assertThat().
                statusCode(200);
    }
}
```

Validating JSON Schema from Rest Assured: External Library

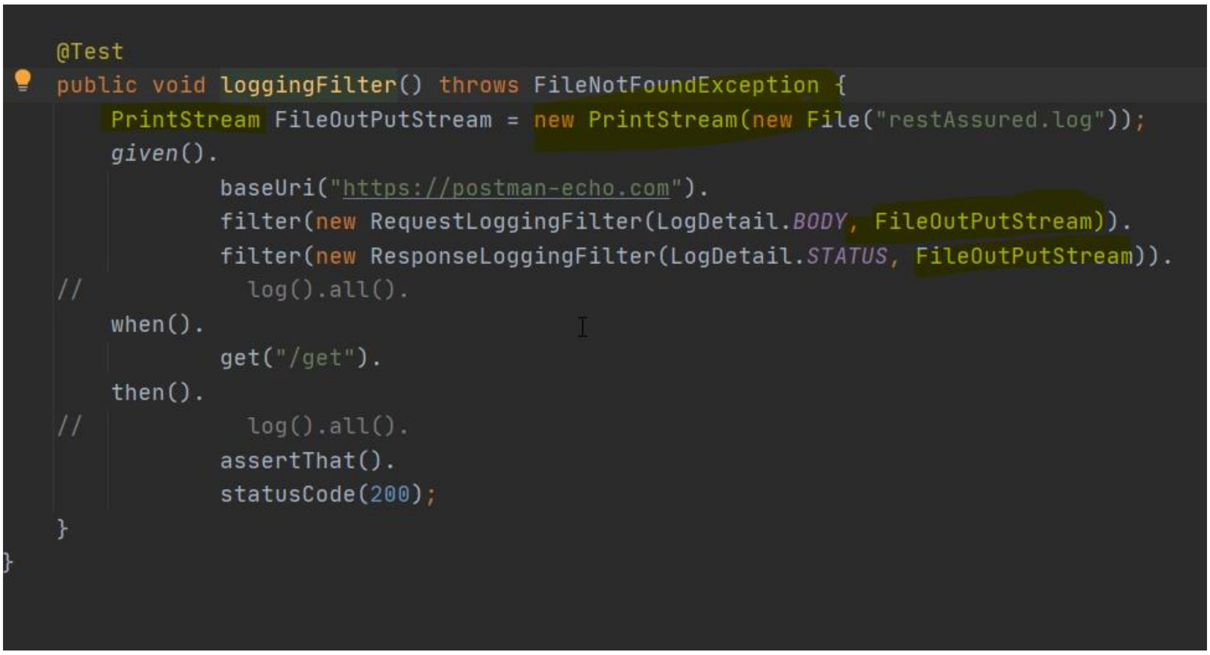
Generally used to validate the json response data format and the mandatory fields, field type etc.



```
import org.testng.annotations.Test;

import static io.restassured.RestAssured.given;
import static io.restassured.module.jsv.JsonSchemaValidator.matchesJsonSchemaInClasspath;

public class JsonSchemaValidation {

    @Test
    public void validateJsonSchema() {
        given().
                baseUri( s: "https://postman-echo.com").
                log().all().
        when().
                get( s: "/get").
        then().
                log().all().
                assertThat().
                statusCode( i: 200).
                body(matchesJsonSchemaInClasspath( pathToSchemaInClasspath: "EchoGet.json"));
    }
}
```

Log Rest Assured Req/Res details to the Log File:

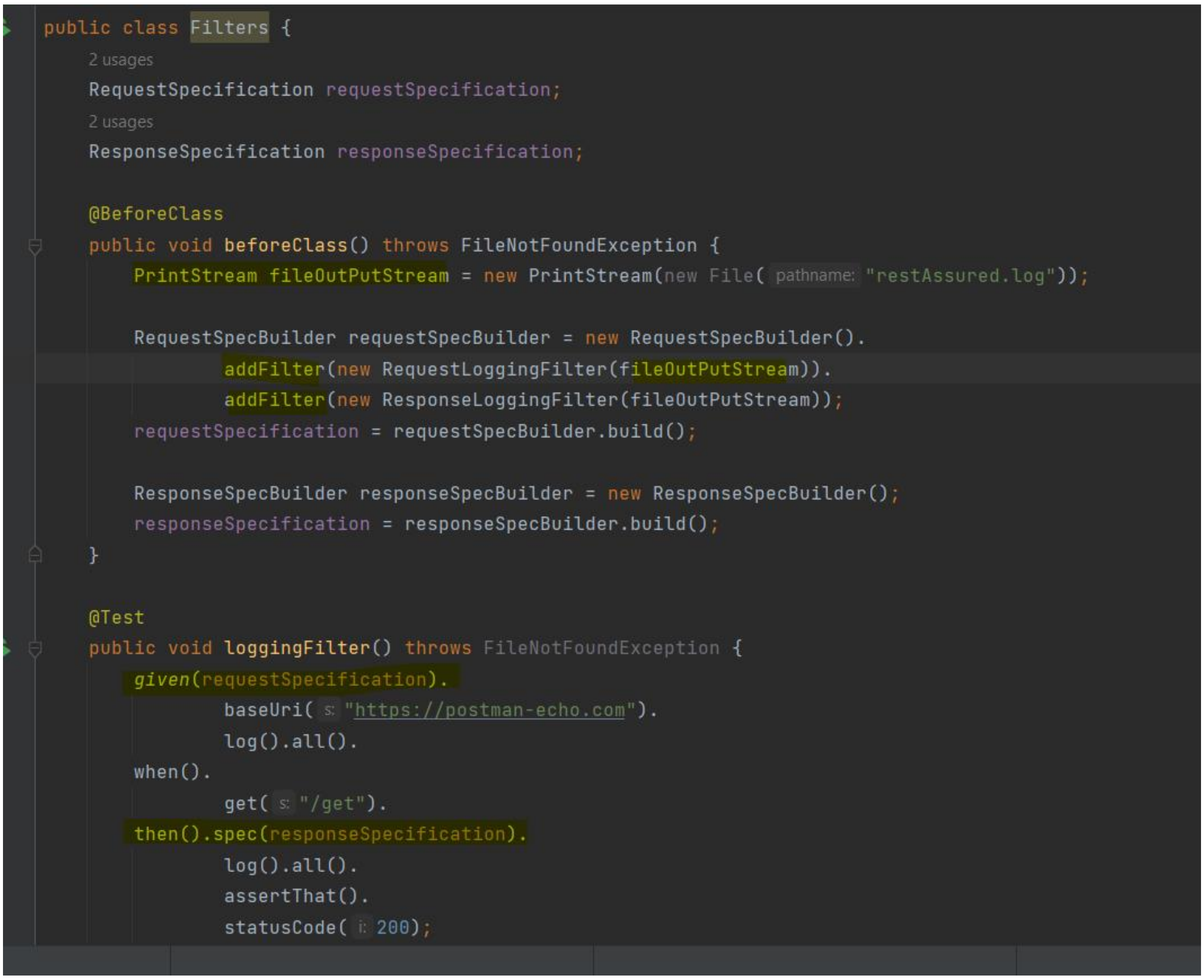
Useful in CI CD via PrintStream class. Also it has option to print what is necessary with Pretty print and other options.



```
    @Test
    public void loggingFilter() throws FileNotFoundException {
        PrintStream FileOutPutStream = new PrintStream(new File("restAssured.log"));
        given().
                baseUri("https://postman-echo.com").
                filter(new RequestLoggingFilter(LogDetail.BODY, FileOutPutStream)).
                filter(new ResponseLoggingFilter(LogDetail.STATUS, FileOutPutStream)).
//              log().all().
        when().
                get("/get").
        then().
//              log().all().
                assertThat().
                statusCode(200);
    }
}
```

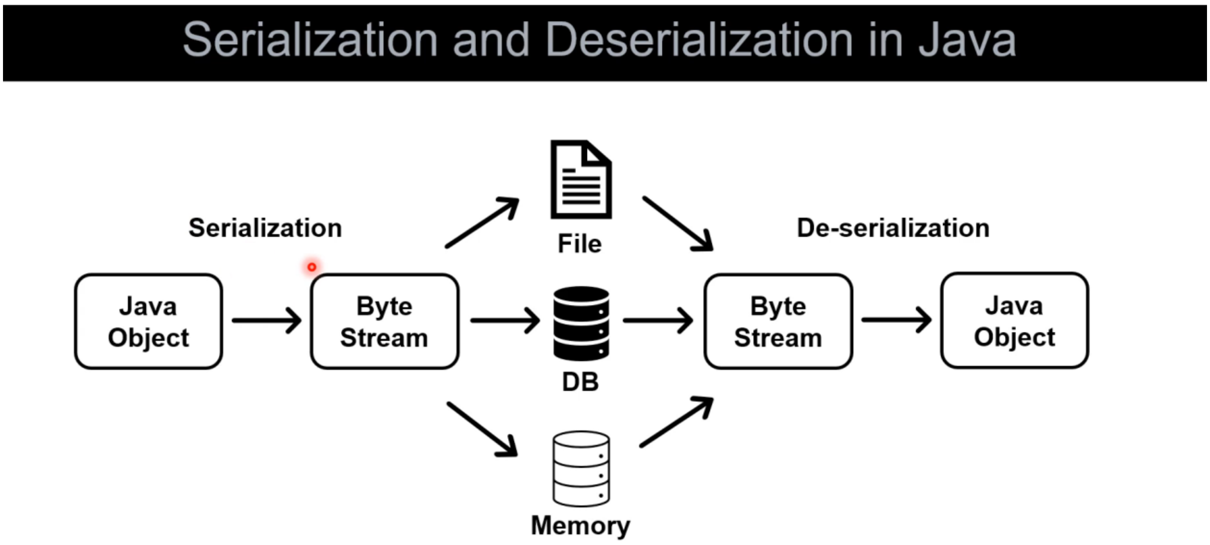
Provide Logging in Req/Res Specification builder class so it can be used again in another tests.

given should contain request Specification object and then spec should contain response specification

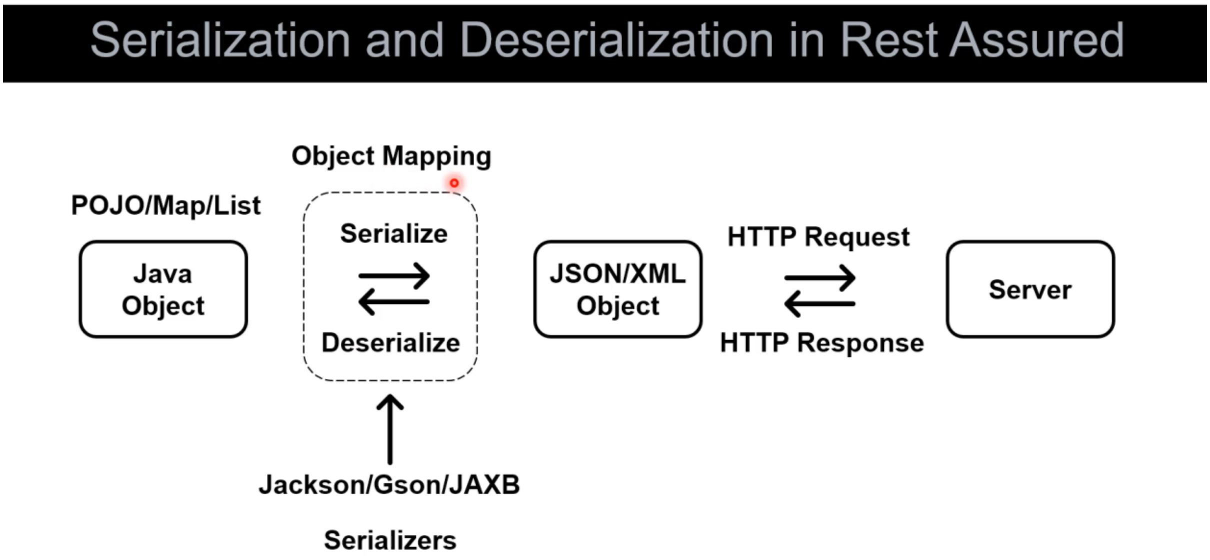


```
public class Filters {
    2 usages
    RequestSpecification requestSpecification;
    2 usages
    ResponseSpecification responseSpecification;

    @BeforeClass
    public void beforeClass() throws FileNotFoundException {
        PrintStream fileOutPutStream = new PrintStream(new File( pathname: "restAssured.log"));

        RequestSpecBuilder requestSpecBuilder = new RequestSpecBuilder().
                addFilter(new RequestLoggingFilter(fileOutPutStream)).
                addFilter(new ResponseLoggingFilter(fileOutPutStream));
        requestSpecification = requestSpecBuilder.build();

        ResponseSpecBuilder responseSpecBuilder = new ResponseSpecBuilder();
        responseSpecification = responseSpecBuilder.build();
    }

    @Test
    public void loggingFilter() throws FileNotFoundException {
        given(requestSpecification).
                baseUri( s: "https://postman-echo.com").
                log().all().
        when().
                get( s: "/get").
        then().spec(responseSpecification).
                log().all().
                assertThat().
                statusCode( i: 200);
```

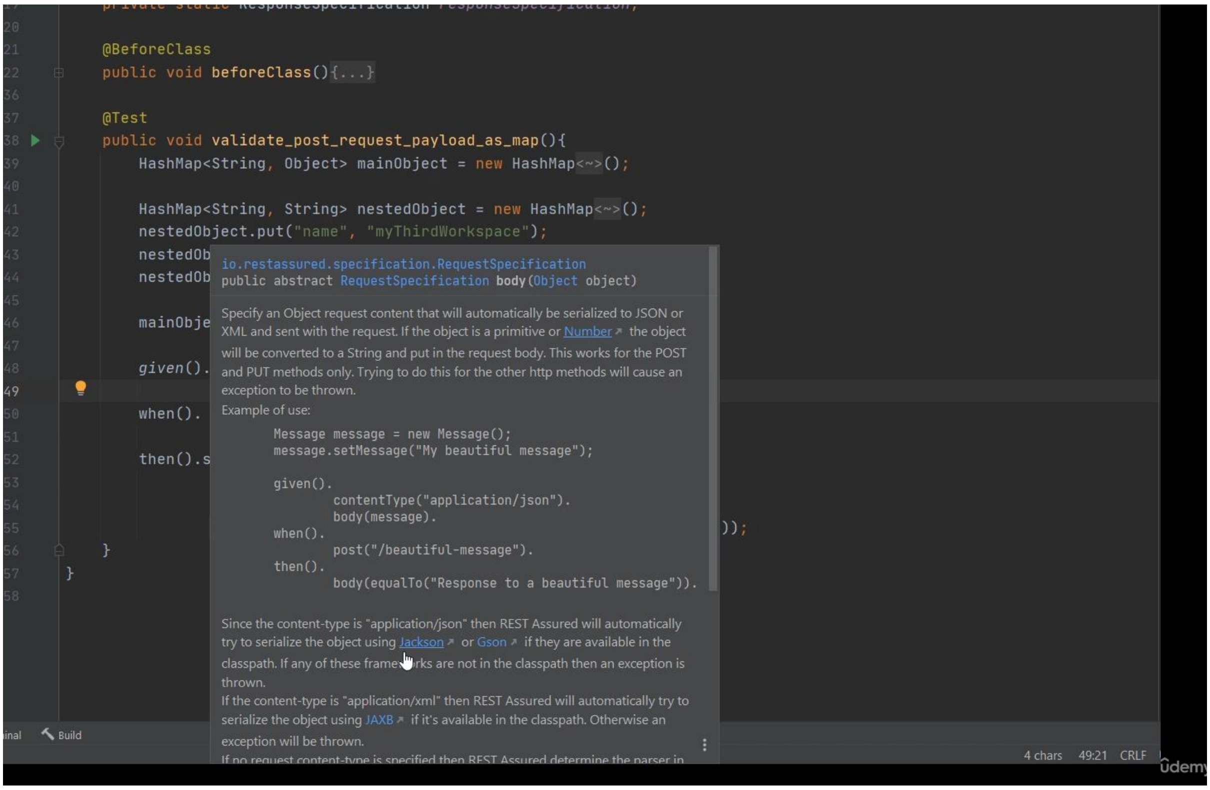
Serialization and De-Serialization in Java:



Rest Assured: *Object mapping of JSON/XML files into Java object either POJO/relevant class/collections and vice versa.*



Serialize Map to JSON Object using Jackson-databind Maven Dependency:
Method used to serialize java object to JSON here is writeValueAsString(javaObjects)



```
    private static ResponseSpecification responseSpecification;

    @BeforeClass
    public void beforeClass(){...}

    @Test
    public void validate_post_request_payload_as_map(){
        HashMap<String, Object> mainObject = new HashMap<~>();

        HashMap<String, String> nestedObject = new HashMap<~>();
        nestedObject.put("name", "myThirdWorkspace");
        nestedOb
        nestedOb

        mainObje

        given().

        when().

        then().s
                                                        ));
    }
}
```

io.restassured.specification.RequestSpecification
public abstract RequestSpecification body(Object object)

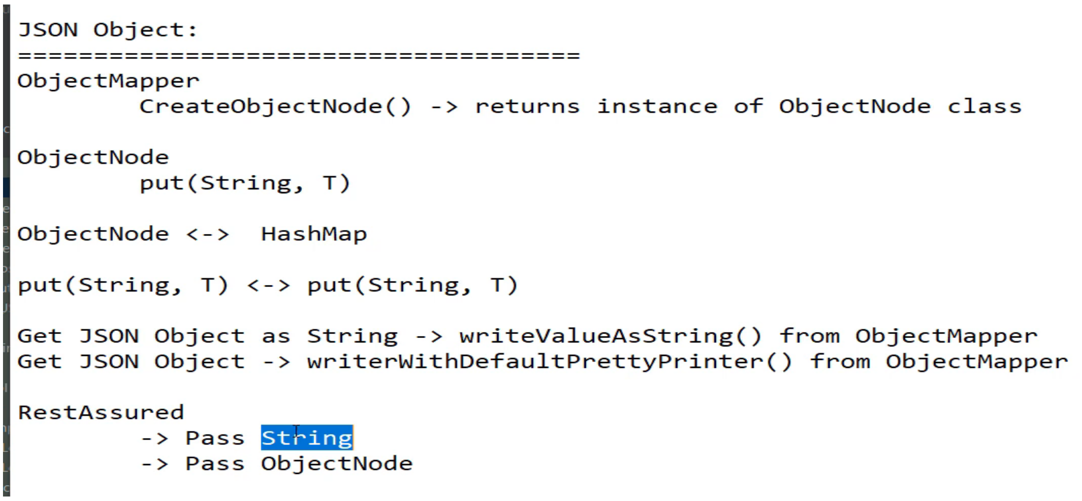
Specify an Object request content that will automatically be serialized to JSON or XML and sent with the request. If the object is a primitive or Number ↗ the object will be converted to a String and put in the request body. This works for the POST and PUT methods only. Trying to do this for the other http methods will cause an exception to be thrown.

Example of use:

```
        Message message = new Message();
        message.setMessage("My beautiful message");

        given().
                contentType("application/json").
                body(message).
        when().
                post("/beautiful-message").
        then().
                body(equalTo("Response to a beautiful message")).
```

Since the content-type is "application/json" then REST Assured will automatically try to serialize the object using Jackson ↗ or Gson ↗ if they are available in the classpath. If any of these frameworks are not in the classpath then an exception is thrown.
If the content-type is "application/xml" then REST Assured will automatically try to serialize the object using JAXB ↗ if it's available in the classpath. Otherwise an exception will be thrown.
If no request content-type is specified then REST Assured determine the parser in

We can also create JSON file via Object-Mapper.



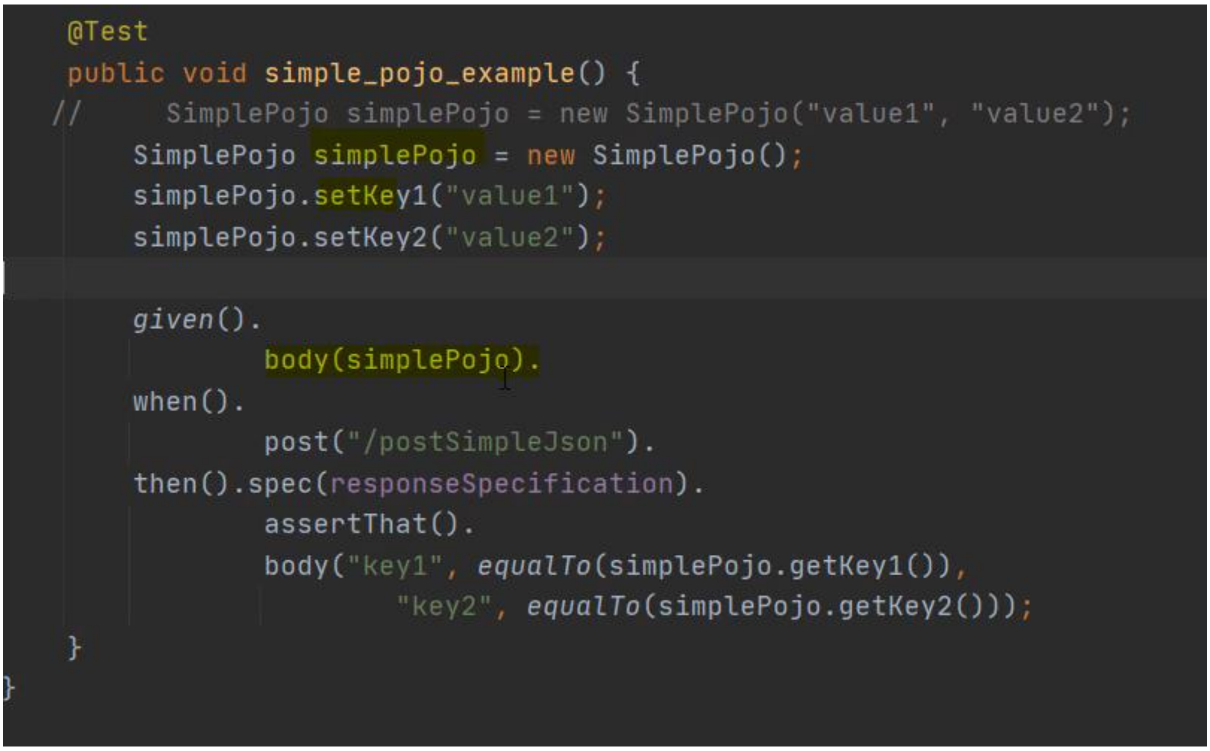
```
JSON Object:
=====================================
ObjectMapper
        CreateObjectNode() -> returns instance of ObjectNode class

ObjectNode
        put(String, T)

ObjectNode <->  HashMap

put(String, T) <-> put(String, T)

Get JSON Object as String -> writeValueAsString() from ObjectMapper
Get JSON Object -> writerWithDefaultPrettyPrinter() from ObjectMapper

RestAssured
        -> Pass String
        -> Pass ObjectNode
```

Simple POJO Object to JSON in Rest Assured (Serialization) :

Sample simple JSON here used is:

{
"key1":"value1",
"key2":"value2"
}

Passing POJO object directly in Rest Assured body method which will do serialization internally using Jackson or Gson libraries and feed as in application/json format for the request. Also, make note to support this it is needed to include default public constructor in the POJO though we are going to use only parameterized constructor to initialize the values or going to use the setter methods.

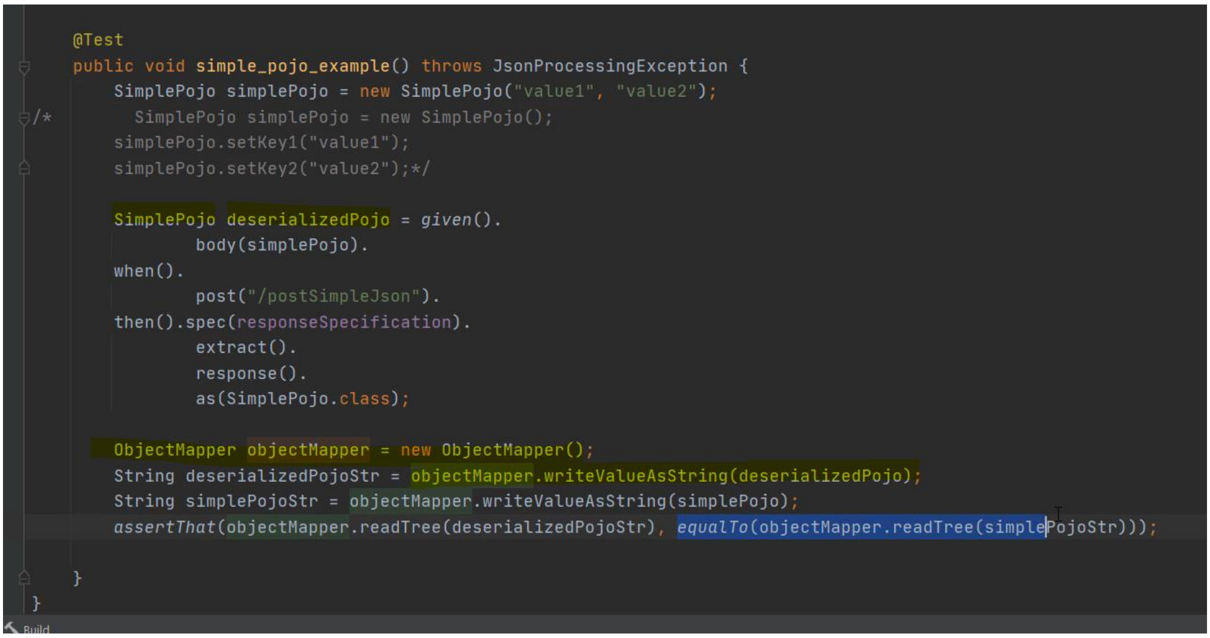


```
    @Test
    public void simple_pojo_example() {
//      SimplePojo simplePojo = new SimplePojo("value1", "value2");
        SimplePojo simplePojo = new SimplePojo();
        simplePojo.setKey1("value1");
        simplePojo.setKey2("value2");

        given().
                body(simplePojo).
        when().
                post("/postSimpleJson").
        then().spec(responseSpecification).
                assertThat().
                body("key1", equalTo(simplePojo.getKey1()),
                        "key2", equalTo(simplePojo.getKey2()));
    }
}
```

Deserialization of simple POJO:

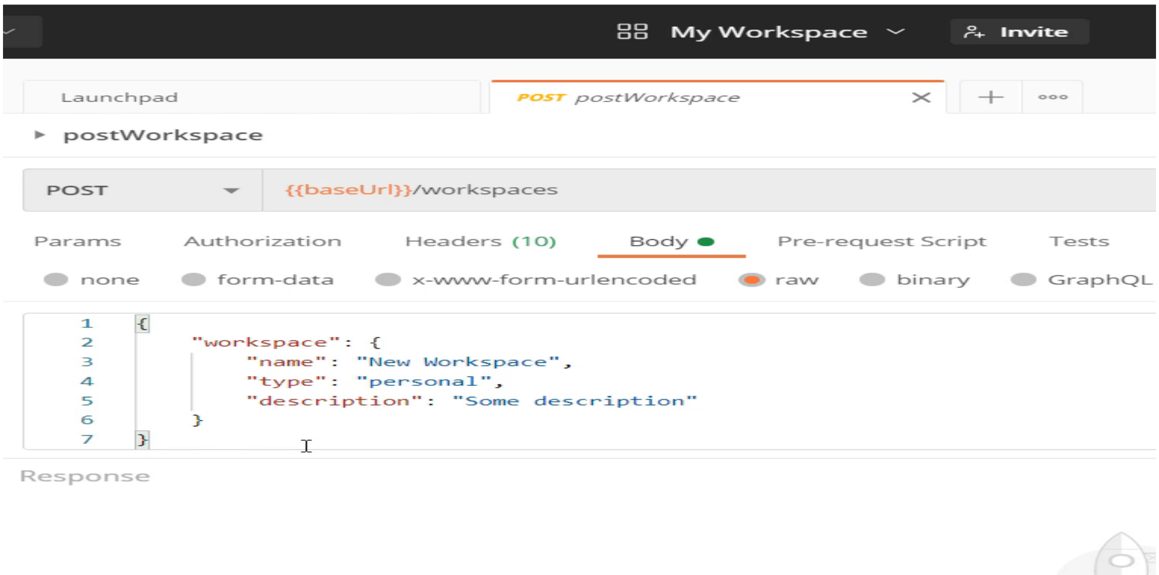
*Below example will receive the response in POJO format and it has to be provided in **then()** condition with the method as where pojo class structure is passed.*

ObjectMapper is an important class in Jackson which here used to convert the POJO objects into JSON values in String formation for comparison.



```
    @Test
    public void simple_pojo_example() throws JsonProcessingException {
        SimplePojo simplePojo = new SimplePojo("value1", "value2");
/*          SimplePojo simplePojo = new SimplePojo();
        simplePojo.setKey1("value1");
        simplePojo.setKey2("value2");*/

        SimplePojo deserializedPojo = given().
                body(simplePojo).
        when().
                post("/postSimpleJson").
        then().spec(responseSpecification).
                extract().
                response().
                as(SimplePojo.class);

        ObjectMapper objectMapper = new ObjectMapper();
        String deserializedPojoStr = objectMapper.writeValueAsString(deserializedPojo);
        String simplePojoStr = objectMapper.writeValueAsString(simplePojo);
        assertThat(objectMapper.readTree(deserializedPojoStr), equalTo(objectMapper.readTree(simplePojoStr)));

    }
}
```

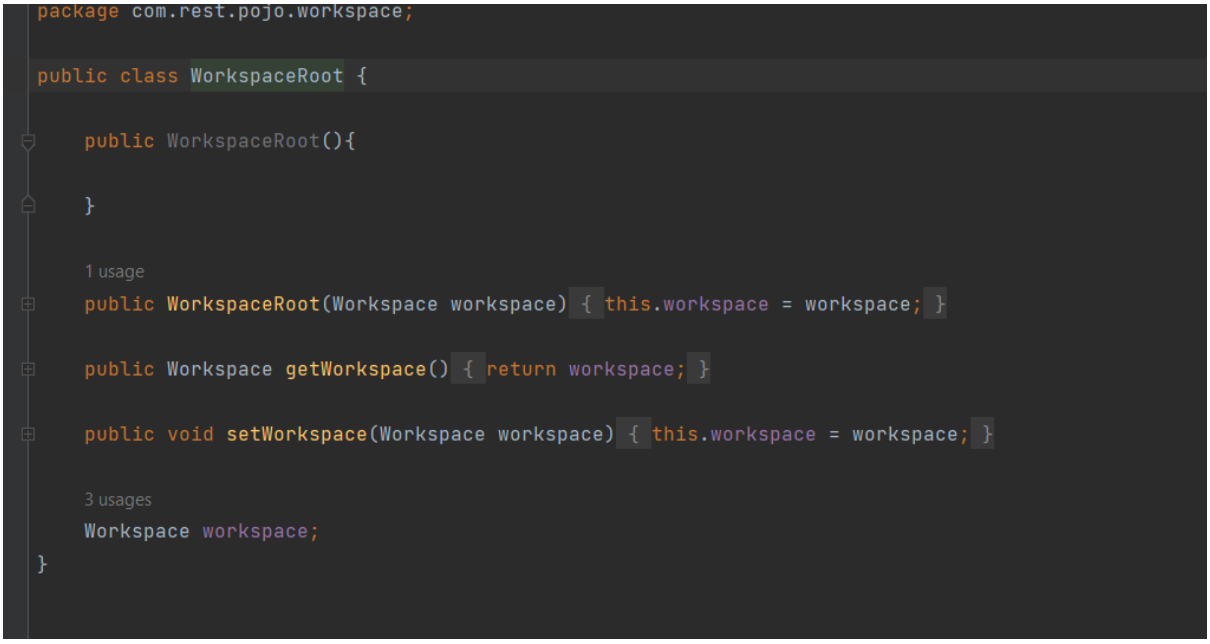
POJO Object for below Payload:
workpsace property root and inside / nested name, type, description



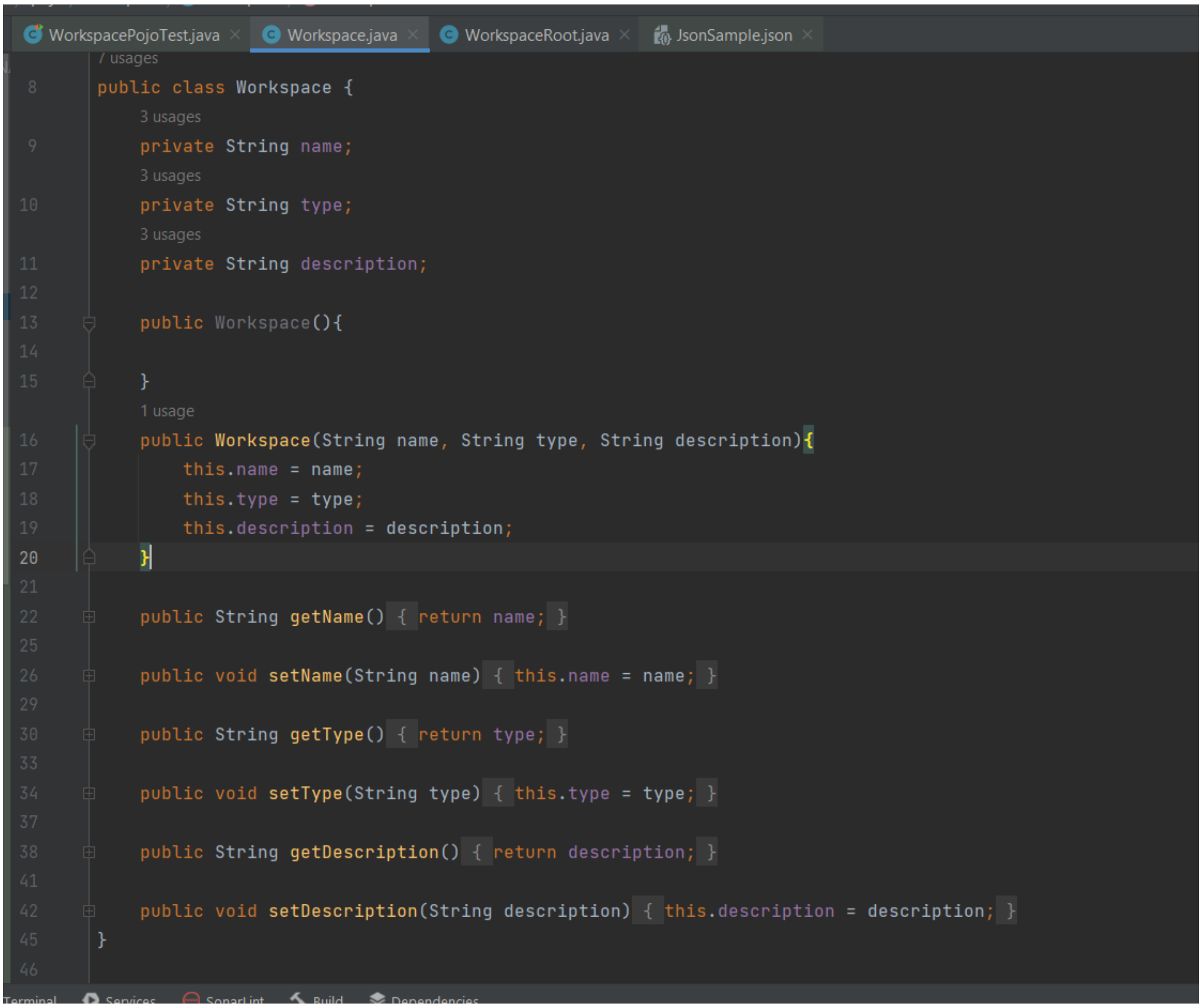
POST {{baseUrl}}/workspaces

```
{
    "workspace": {
        "name": "New Workspace",
        "type": "personal",
        "description": "Some description"
    }
}
```

Create POJO with getter , setter for these two nodes:

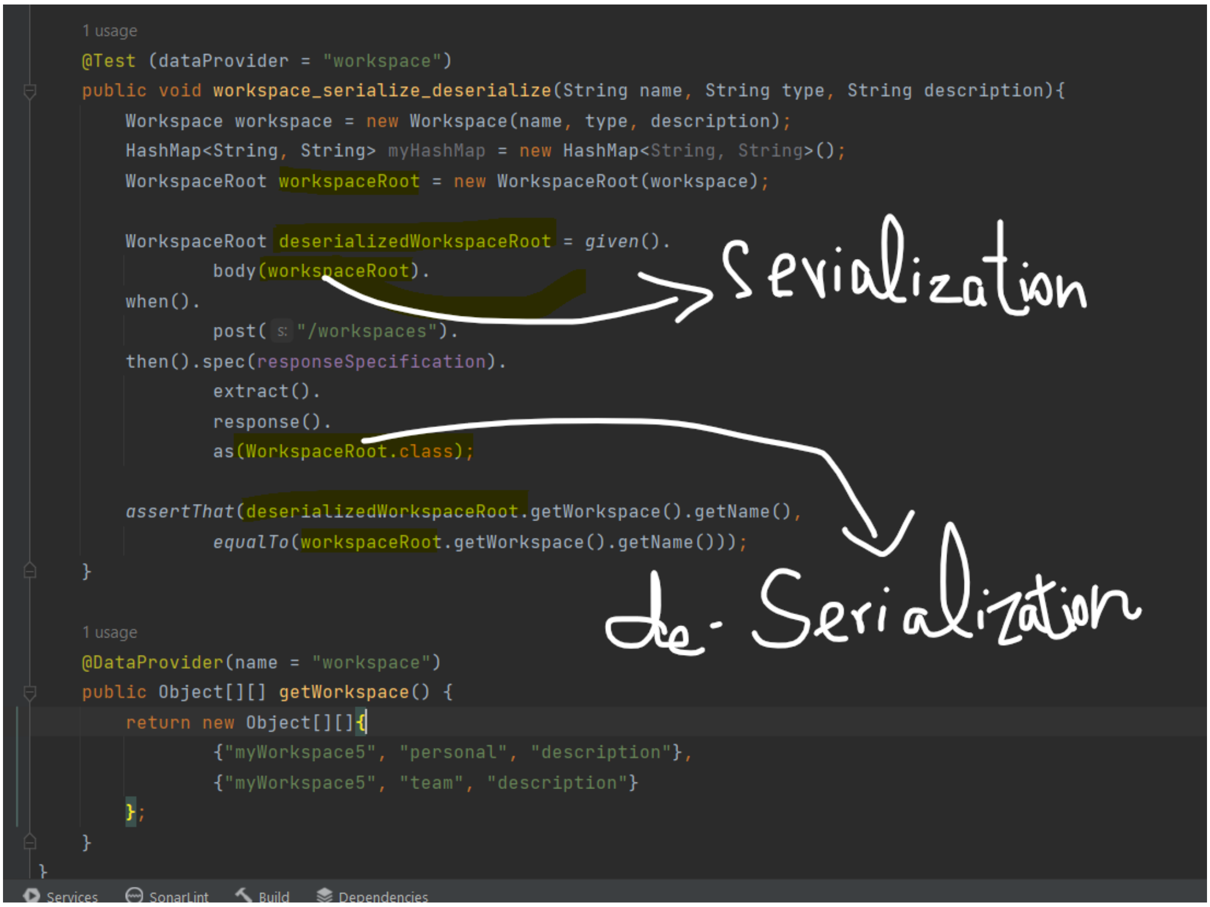


```java
package com.rest.pojo.workspace;

public class WorkspaceRoot {

    public WorkspaceRoot(){

    }

    1 usage
    public WorkspaceRoot(Workspace workspace) { this.workspace = workspace; }

    public Workspace getWorkspace() { return workspace; }

    public void setWorkspace(Workspace workspace) { this.workspace = workspace; }

    3 usages
    Workspace workspace;
}
```



```java
7 usages
public class Workspace {
    3 usages
    private String name;
    3 usages
    private String type;
    3 usages
    private String description;

    public Workspace(){

    }
    1 usage
    public Workspace(String name, String type, String description){
        this.name = name;
        this.type = type;
        this.description = description;
    }

    public String getName() { return name; }

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

    public String getType() { return type; }

    public void setType(String type) { this.type = type; }

    public String getDescription() { return description; }

    public void setDescription(String description) { this.description = description; }
}
```

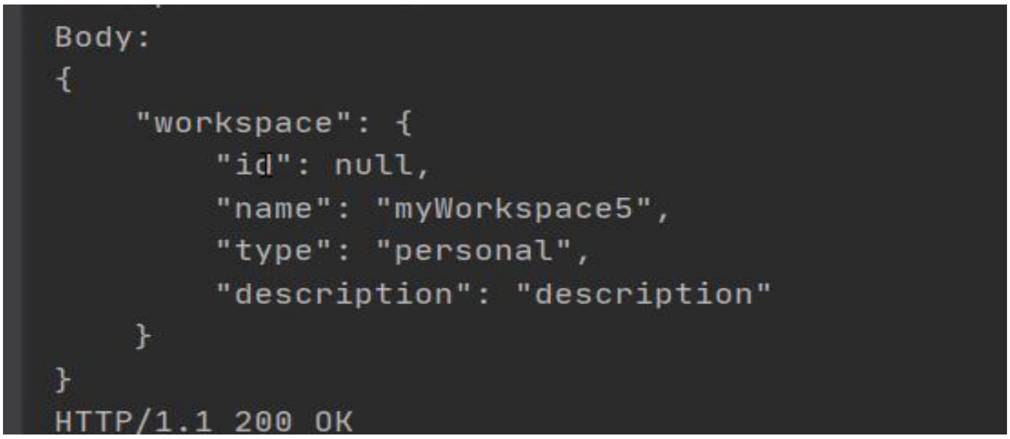
POJO Serialization and Deserialization in Rest Assured:



```
1 usage
@Test (dataProvider = "workspace")
public void workspace_serialize_deserialize(String name, String type, String description){
    Workspace workspace = new Workspace(name, type, description);
    HashMap<String, String> myHashMap = new HashMap<String, String>();
    WorkspaceRoot workspaceRoot = new WorkspaceRoot(workspace);

    WorkspaceRoot deserializedWorkspaceRoot = given().
            body(workspaceRoot).
    when().
            post( s: "/workspaces").
    then().spec(responseSpecification).
            extract().
            response().
            as(WorkspaceRoot.class);

    assertThat(deserializedWorkspaceRoot.getWorkspace().getName(),
            equalTo(workspaceRoot.getWorkspace().getName()));
}

1 usage
@DataProvider(name = "workspace")
public Object[][] getWorkspace() {
    return new Object[][]{
            {"myWorkspace5", "personal", "description"},
            {"myWorkspace5", "team", "description"}
    };
}
}
```

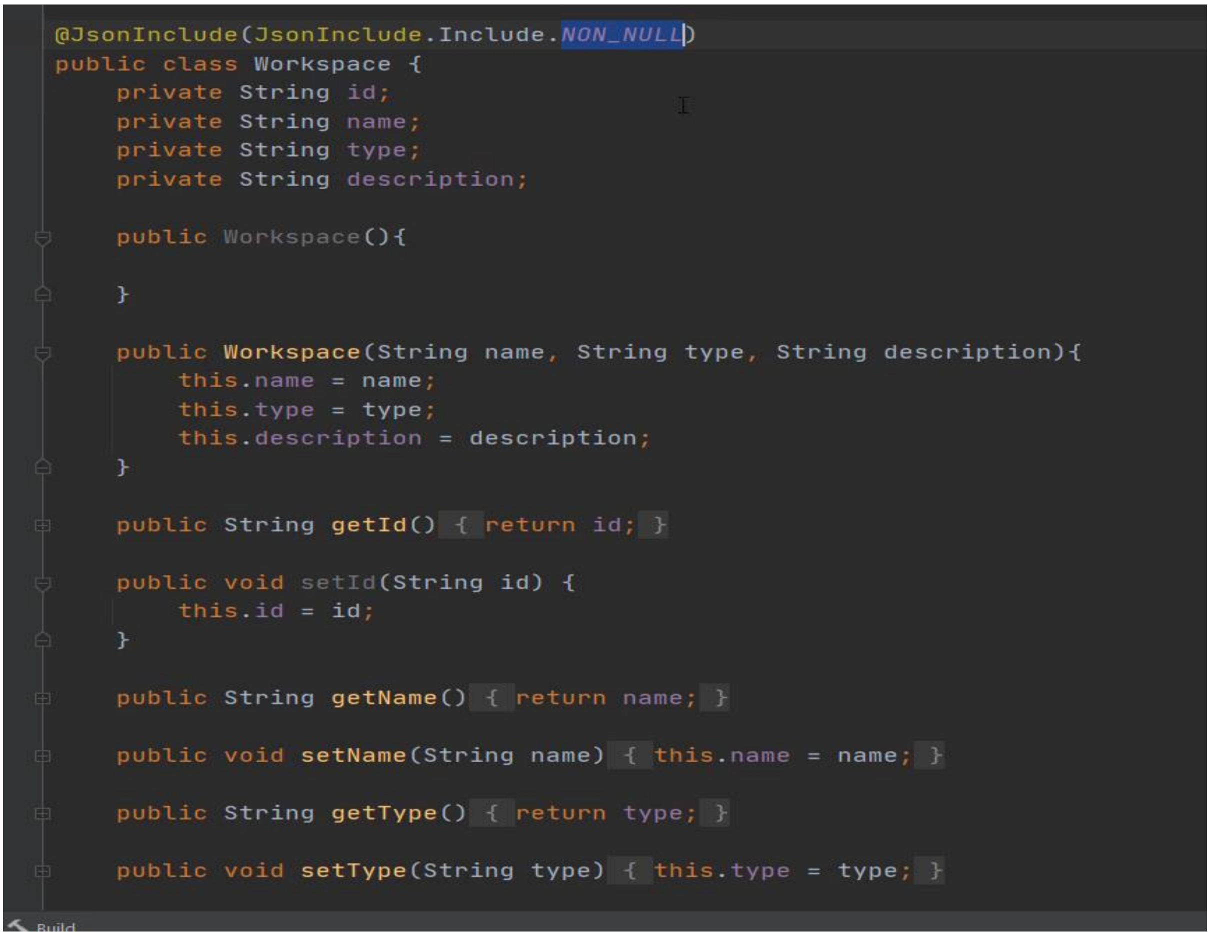
Deserialization :

If a String property is present extra in the POJO closs, then it will go in the request like below.

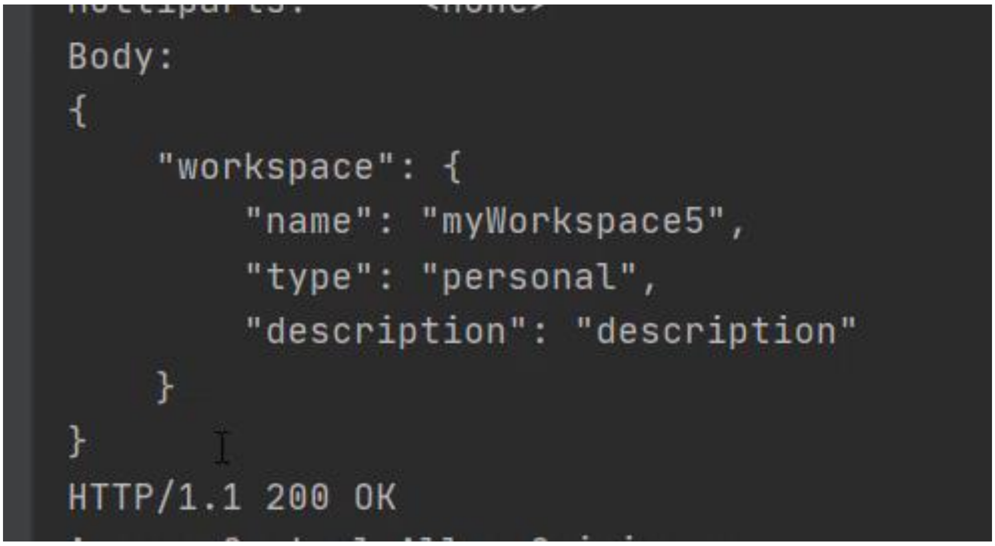


```
Body:
{
    "workspace": {
        "id": null,
        "name": "myWorkspace5",
        "type": "personal",
        "description": "description"
    }
}
HTTP/1.1 200 OK
```

*Add Non null annotation from Jackson either at Class level or on separate fields/string members**, then the de-serialized JSON in request will not contain id as it is null.***
Similar to this we have other properties like Non-null (strings), non-default (integer) , non-empty (objects) can be explored in Jackson.

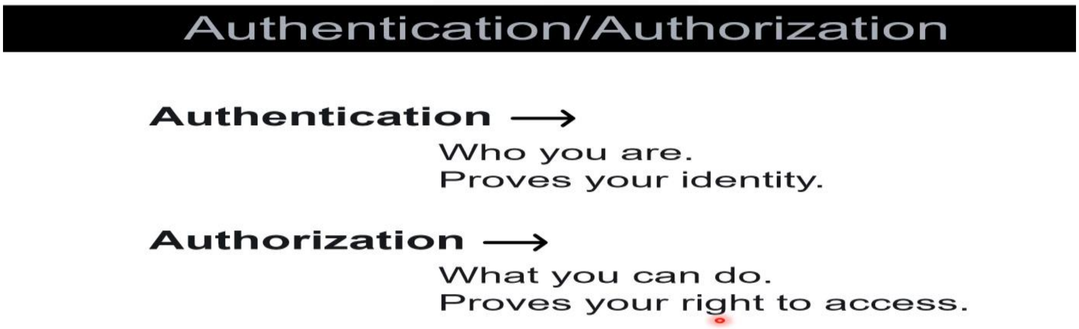


```java
@JsonInclude(JsonInclude.Include.NON_NULL)
public class Workspace {
    private String id;
    private String name;
    private String type;
    private String description;

    public Workspace(){

    }

    public Workspace(String name, String type, String description){
        this.name = name;
        this.type = type;
        this.description = description;
    }

    public String getId() { return id; }

    public void setId(String id) {
        this.id = id;
    }

    public String getName() { return name; }

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

    public String getType() { return type; }

    public void setType(String type) { this.type = type; }
```



```
Body:
{
    "workspace": {
        "name": "myWorkspace5",
        "type": "personal",
        "description": "description"
    }
}
HTTP/1.1 200 OK
```

Authentication and Authorization:



Authentication/Authorization

Authentication →

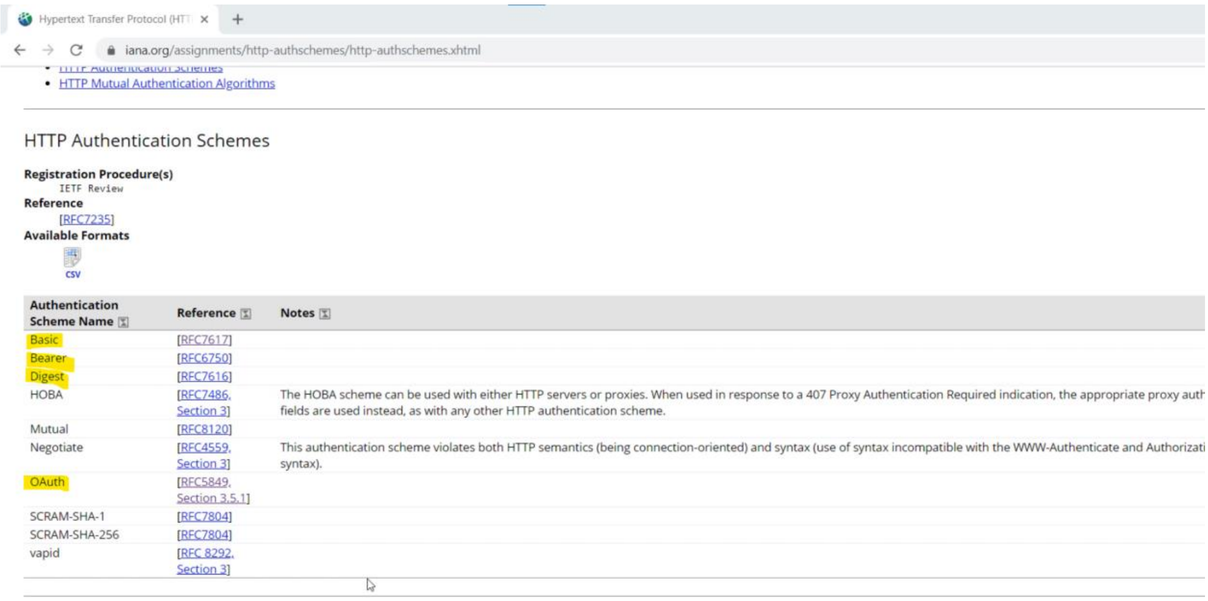
Who you are.
Proves your identity.

Authorization →

What you can do.
Proves your right to access.

List of Authentication Schemes in HTTP protocol:

Most common ones are highlighted below.



Hypertext Transfer Protocol (HTT ×
+
iana.org/assignments/http-authschemes/http-authschemes.xhtml

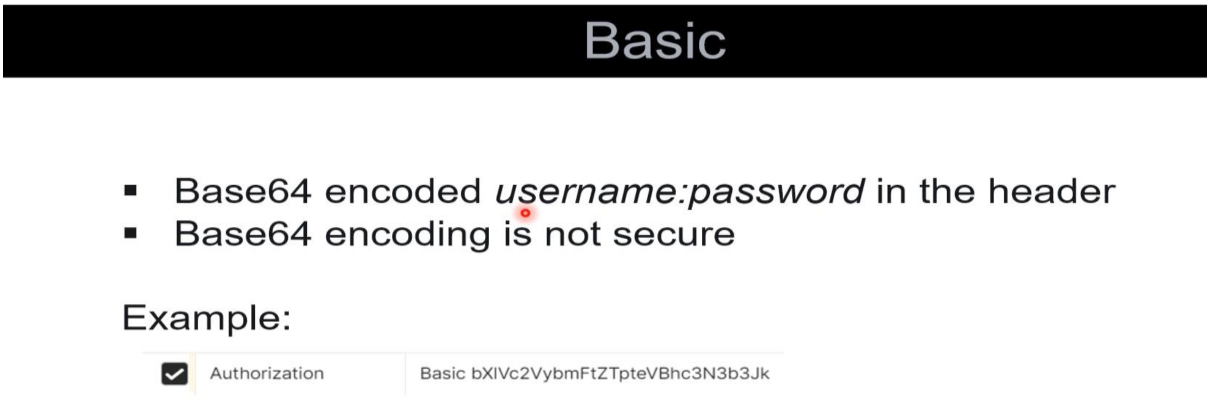
- HTTP Authentication Schemes
- HTTP Mutual Authentication Algorithms

HTTP Authentication Schemes

Registration Procedure(s)
IETF Review
Reference
[RFC7235]
Available Formats
CSV

Authentication Scheme Name	Reference	Notes
Basic	[RFC7617]	
Bearer	[RFC6750]	
Digest	[RFC7616]	
HOBA	[RFC7486, Section 3]	The HOBA scheme can be used with either HTTP servers or proxies. When used in response to a 407 Proxy Authentication Required indication, the appropriate proxy auth fields are used instead, as with any other HTTP authentication scheme.
Mutual	[RFC8120]	
Negotiate	[RFC4559, Section 3]	This authentication scheme violates both HTTP semantics (being connection-oriented) and syntax (use of syntax incompatible with the WWW-Authenticate and Authorizati syntax).
OAuth	[RFC5849, Section 3.5.1]	
SCRAM-SHA-1	[RFC7804]	
SCRAM-SHA-256	[RFC7804]	
vapid	[RFC 8292, Section 3]	

Basic Authentication Type: *Not that much safe and it is used internal services in general.*



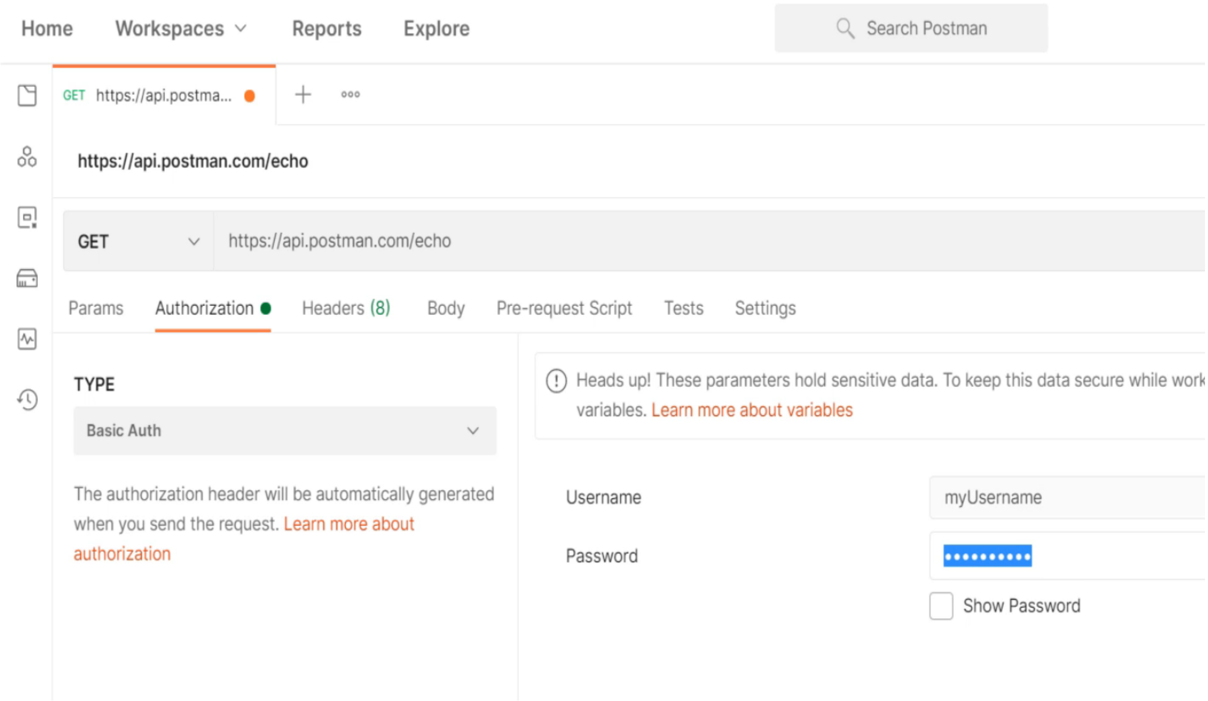
Basic

- Base64 encoded *username:password* in the header
- Base64 encoding is not secure

Example:

☑	Authorization	Basic bXlVc2VybmFtZTpteVBhc3N3b3Jk

Way to provide this input in Postman:



https://api.postman.com/echo

GET https://api.postman.com/echo

TYPE

Basic Auth

The authorization header will be automatically generated when you send the request. Learn more about authorization

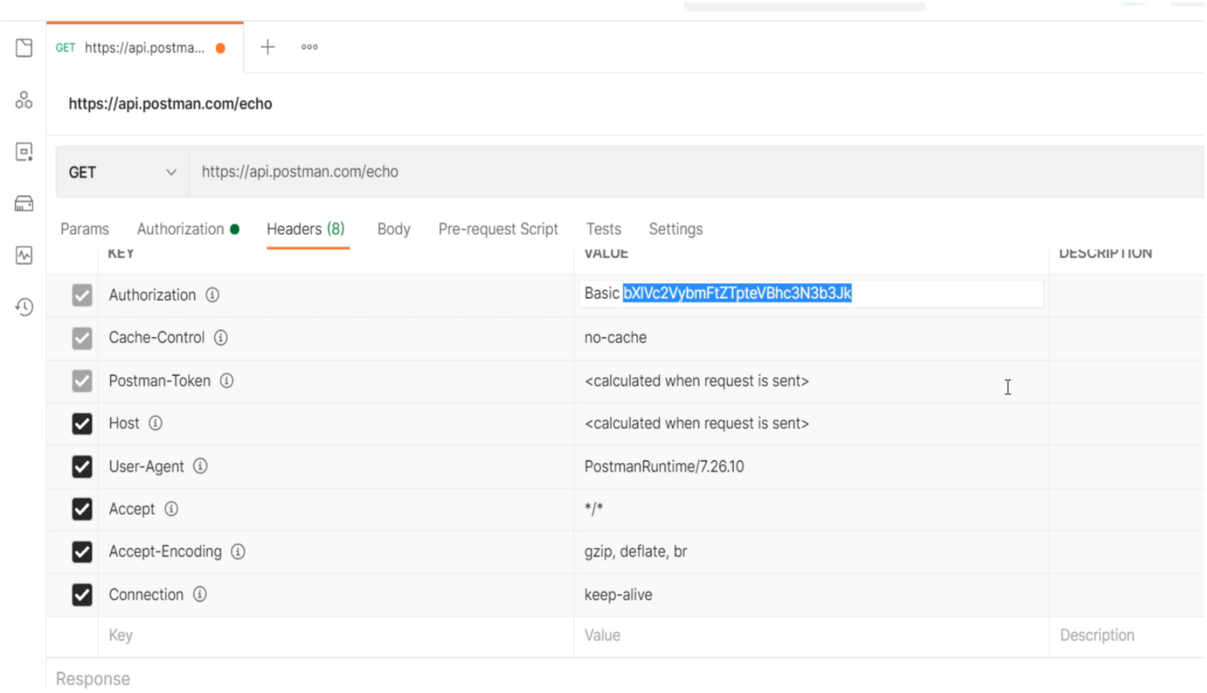
Heads up! These parameters hold sensitive data. To keep this data secure while work variables. Learn more about variables

Username myUsername

Password ••••••••••

□ Show Password

Below is the way it is getting passed in the request:

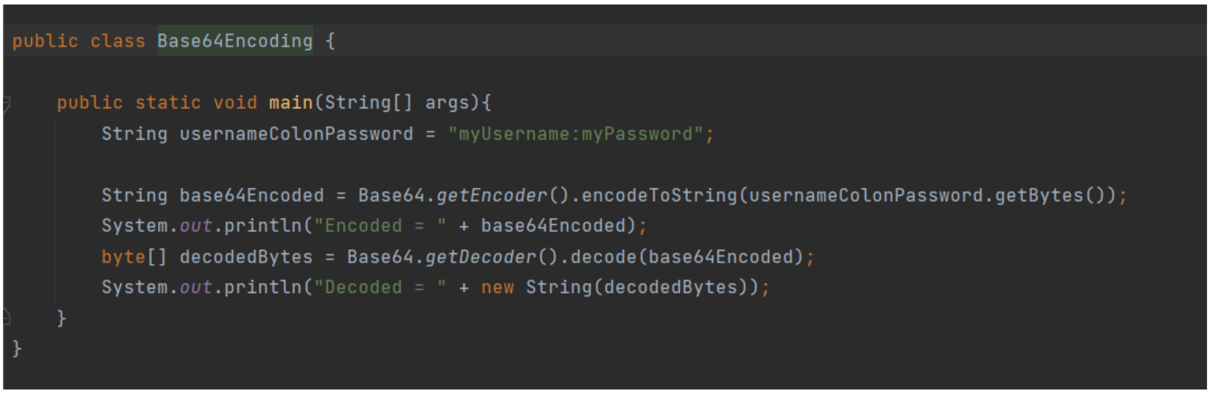


https://api.postman.com/echo

GET https://api.postman.com/echo

	KEY	VALUE	DESCRIPTION
☑	Authorization ⓘ	Basic bXlVc2VybmFtZTpteVBhc3N3b3Jk	
☑	Cache-Control ⓘ	no-cache	
☑	Postman-Token ⓘ	<calculated when request is sent>	
☑	Host ⓘ	<calculated when request is sent>	
☑	User-Agent ⓘ	PostmanRuntime/7.26.10	
☑	Accept ⓘ	*/*	
☑	Accept-Encoding ⓘ	gzip, deflate, br	
☑	Connection ⓘ	keep-alive	
	Key	Value	Description

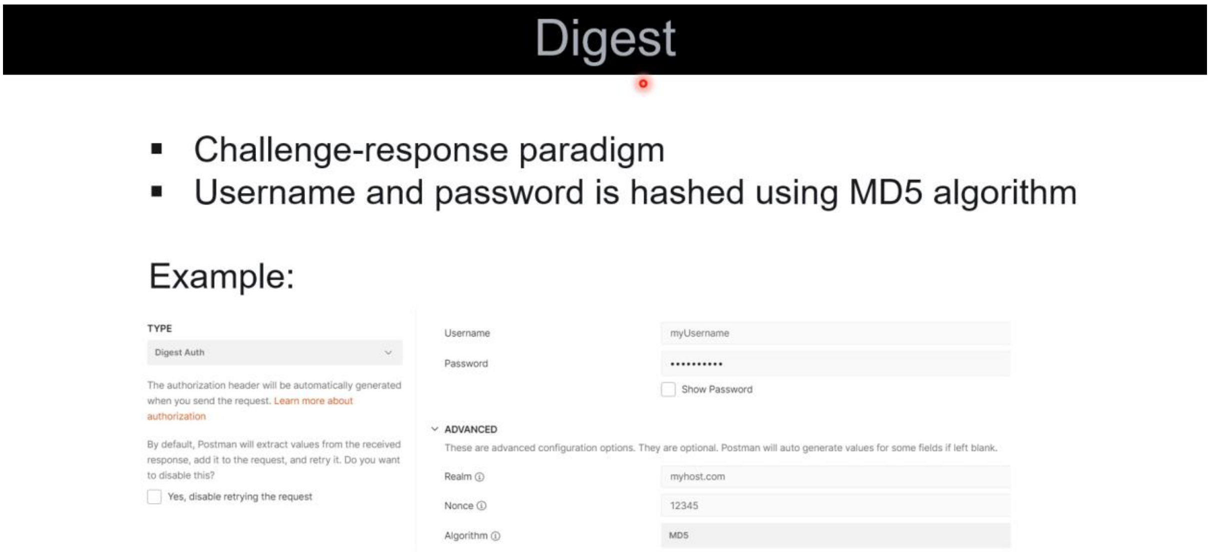
Base 64 Encoding and Decoding in Java Util package :



```java
public class Base64Encoding {

    public static void main(String[] args){
        String usernameColonPassword = "myUsername:myPassword";

        String base64Encoded = Base64.getEncoder().encodeToString(usernameColonPassword.getBytes());
        System.out.println("Encoded = " + base64Encoded);
        byte[] decodedBytes = Base64.getDecoder().decode(base64Encoded);
        System.out.println("Decoded = " + new String(decodedBytes));
    }
}
```

Digest Authentication Scheme:

Client first sends the request without username and password. Server sends other parameters such as realm, nonce, algorithm to the client which client uses to encrypt the username and password. Later encrypted details are put in request to the server. Relatively safer than basic authentication. MD5 algorithm is a sample and it can be any other from the allowed list.



Digest

- Challenge-response paradigm
- Username and password is hashed using MD5 algorithm

Example:

TYPE

Digest Auth

The authorization header will be automatically generated when you send the request. Learn more about authorization

By default, Postman will extract values from the received response, add it to the request, and retry it. Do you want to disable this?

□ Yes, disable retrying the request

Username: myUsername

Password: ••••••••••

□ Show Password

ADVANCED

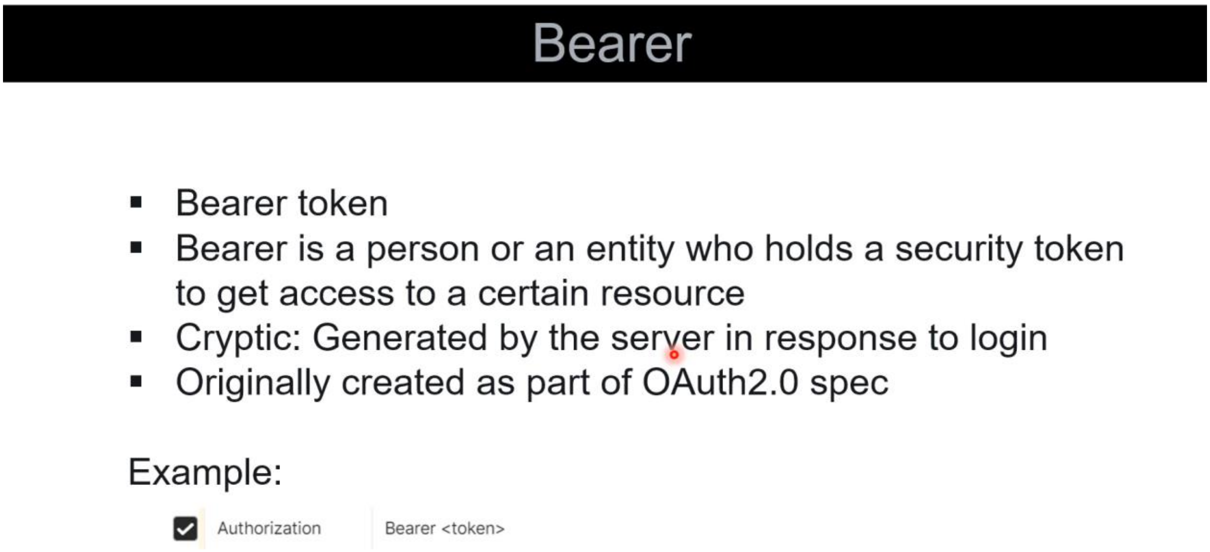
These are advanced configuration options. They are optional. Postman will auto generate values for some fields if left blank.

Realm: myhost.com

Nonce: 12345

Algorithm: MD5

Bearer Token Authentication mechanism :



Bearer

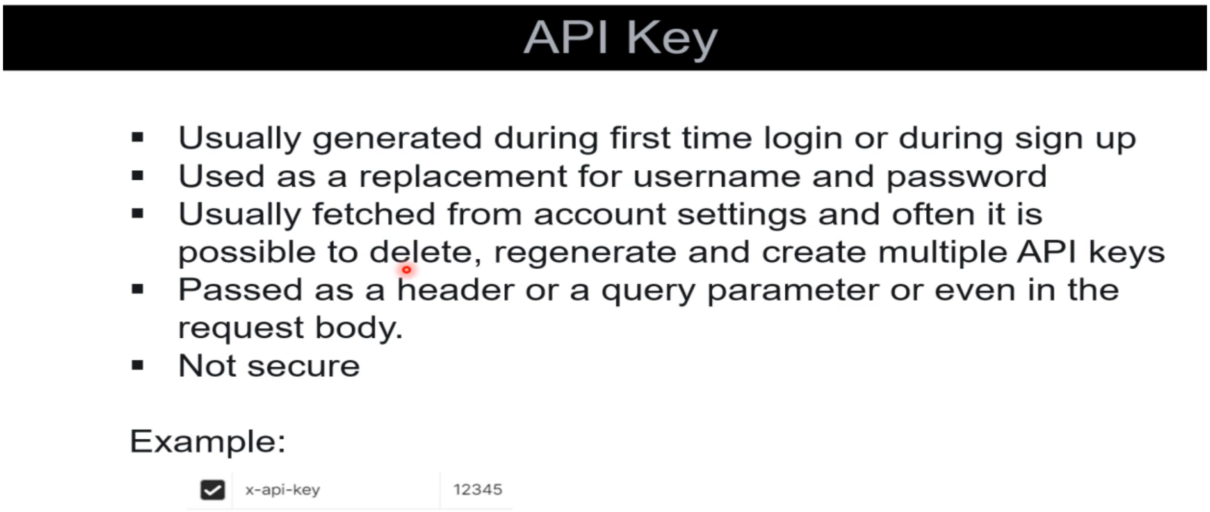
- Bearer token
- Bearer is a person or an entity who holds a security token to get access to a certain resource
- Cryptic: Generated by the server in response to login
- Originally created as part of OAuth2.0 spec

Example:

☑	Authorization	Bearer <token>

Another Type – API Key :

Used for Internal and good example is sauce lab keys, cloud git hub tokens etc.

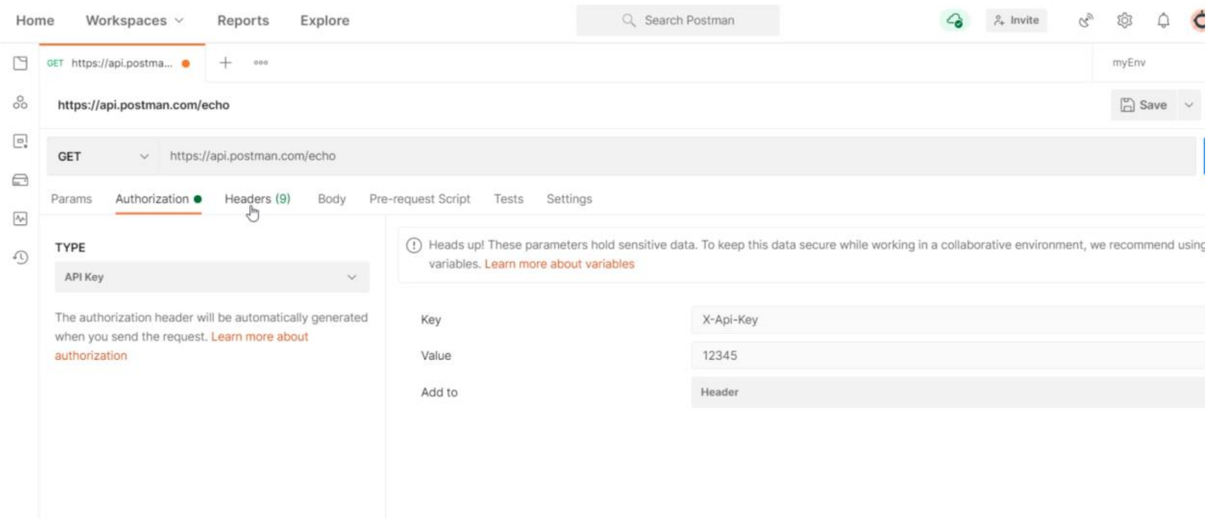


API Key

- Usually generated during first time login or during sign up
- Used as a replacement for username and password
- Usually fetched from account settings and often it is possible to delete, regenerate and create multiple API keys
- Passed as a header or a query parameter or even in the request body.
- Not secure

Example:

☑	x-api-key	12345



GET https://api.postma...

https://api.postman.com/echo

GET https://api.postman.com/echo

TYPE

API Key

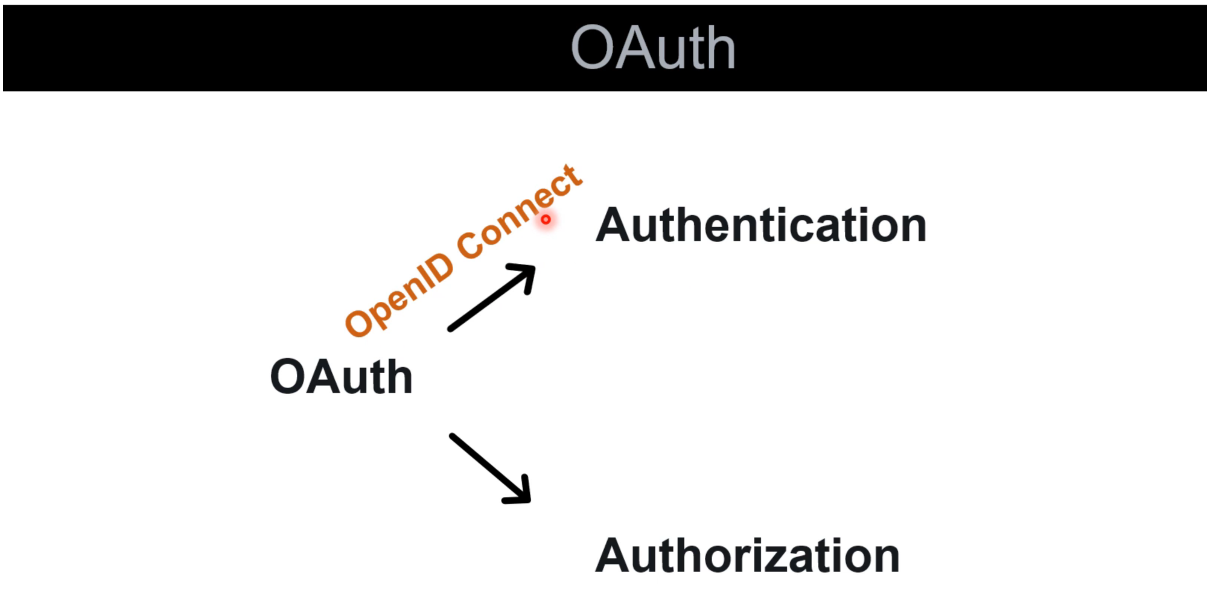
The authorization header will be automatically generated when you send the request. Learn more about authorization

Heads up! These parameters hold sensitive data. To keep this data secure while working in a collaborative environment, we recommend using variables. Learn more about variables

Key	X-Api-Key
Value	12345
Add to	Header

OAuth:

Primarily used for authorization but it also does the authentication via OpenID connect and so it is considered as HTTP authentication mechanism.

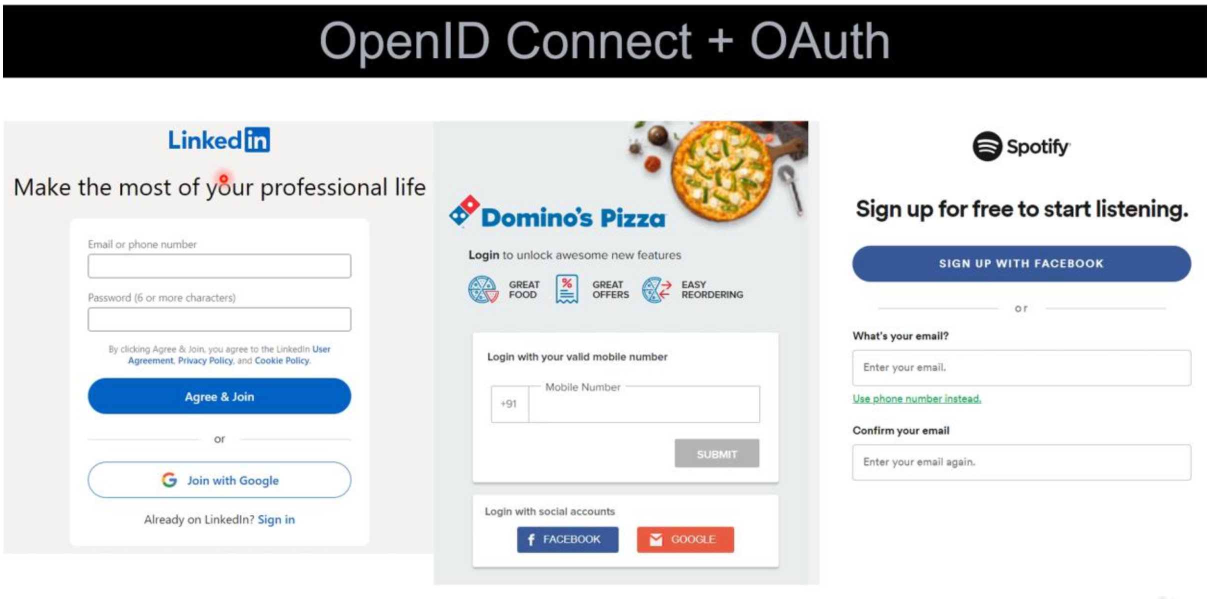


Open ID Connect + OAuth:

***OpenID** is an identity layer on top of the OAuth 2.0 protocol. It allows a user to prove their identity to a third-party application by using an account they already have with an OpenID provider. For example, you can use your Google or Facebook account to log in to other websites that support OpenID.*

OpenID extends OAuth 2.0 with additional features, such as:

> ***ID tokens:*** *These are JSON web tokens (JWTs) that contain information about the user's identity and authentication status. They are issued by the OpenID provider and validated by the third-party application.*
> ***Standard claims:*** *These are predefined sets of attributes that the user can share with the third-party application, such as name, email, picture, etc. They are included in the ID token or returned by the user info endpoint.*
> ***Discovery and dynamic registration:*** *These are mechanisms that enable the third-party application to discover the OpenID provider's configuration and capabilities, and to register itself dynamically with the provider.*


OpenID Connect + OAuth

LinkedIn
Make the most of your professional life
Email or phone number
Password (6 or more characters)
By clicking Agree & Join, you agree to the LinkedIn User Agreement, Privacy Policy, and Cookie Policy.
Agree & Join
or
Join with Google
Already on LinkedIn? Sign in
Domino's Pizza
Login to unlock awesome new features
GREAT FOOD
GREAT OFFERS
EASY REORDERING
Login with your valid mobile number
Mobile Number
+91
SUBMIT
Login with social accounts
FACEBOOK
GOOGLE
Spotify
Sign up for free to start listening.
SIGN UP WITH FACEBOOK
or
What's your email?
Enter your email.
Use phone number instead.
Confirm your email
Enter your email again.

OAuth and Delegated Authorization:

OAuth is a standard that enables access delegation. It allows a user to grant a third-party application access to their protected resources on a resource server, without sharing their credentials. This way, the user can control the level and duration of access that the third-party application has, and revoke it at any time.

Delegated authorization is a type of OAuth flow that involves the user, the third-party application, the resource server, and the authorization server. The user authorizes the third-party application to access their resources on the resource server by obtaining an access token from the authorization server. The access token is a credential that represents the user's consent and specifies the scope and duration of the access. The third-party application can then use the access token to act on the user's behalf when accessing the protected resources.

Example:

Suppose you want to use a web application that allows you to create and share playlists of your favourite songs. The web application uses Spotify as the resource server, where you have an account with your music preferences and playlists. The web application is the third-party application that wants to access your Spotify data.

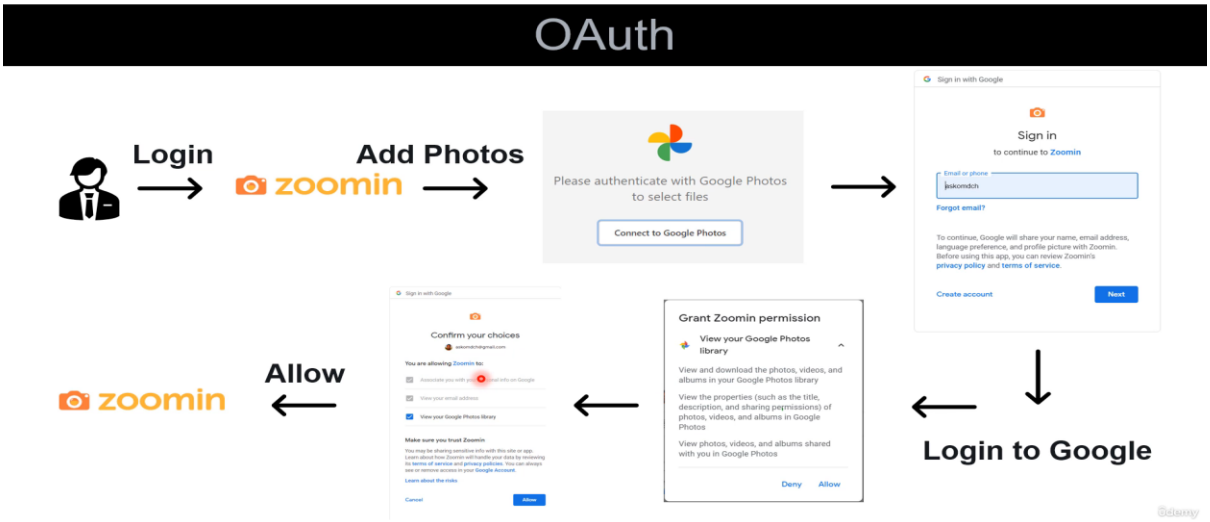
To use the web application, you need to authorize it to access your Spotify account. This is where OAuth and delegated authorization come in. The web application will redirect you to Spotify's authorization server, where you will be asked to log in with your Spotify credentials and grant permission to the web application. You can choose the level of access that you want to give to the web application, such as read-only or read-write, and the duration of the access, such as one hour or one month.

Once you grant permission, Spotify's authorization server will issue an access token to the web application. The access token is a string of characters that represents your consent and the scope and duration of the access. The web application can then use the access token to request your Spotify data from the resource server. The resource server will verify the access token and return the requested data to the web application. The web application can then display your Spotify data and allow you to create and share playlists.

This is how OAuth and delegated authorization work. They enable you to securely share your data with third-party applications without giving away your credentials. You can also revoke the access token at any time if you change your mind or suspect any misuse. OAuth is a widely used standard that supports many web services and applications.

Another Example with OAuth Flow Diagram :

zoomin is an app (3rd Party App) that creates an album or frame with the photos we provide. And we can provide photos from Google photos (Resource app that we need to access) without sharing the credentials. (OAuth)



Please authenticate with Google Photos to select files

Connect to Google Photos

Sign in with Google

Sign in

to continue to Zoomin

Email or phone: askomdch

Forgot email?

To continue, Google will share your name, email address, language preference, and profile picture with Zoomin. Before using this app, you can review Zoomin's privacy policy and terms of service.

Create account | Next

Grant Zoomin permission

View your Google Photos library

View and download the photos, videos, and albums in your Google Photos library

View the properties (such as the title, description, and sharing permissions) of photos, videos, and albums in Google Photos

View photos, videos, and albums shared with you in Google Photos

Deny | Allow

Sign in with Google

Confirm your choices

askomdchv@gmail.com

You are allowing Zoomin to:

☑ Associate you with your personal info on Google

☑ View your email address

☑ View your Google Photos library

Make sure you trust Zoomin

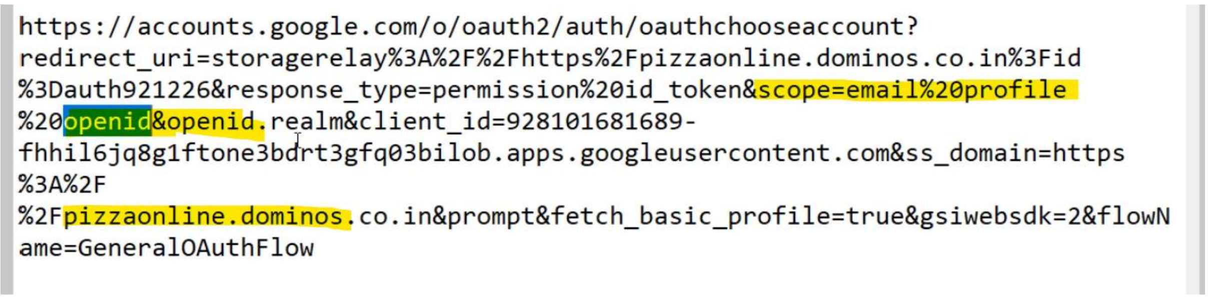
You may be sharing sensitive info with this site or app. Learn about how Zoomin will handle your data by reviewing its terms of service and privacy policies. You can always see or remove access in your Google Account.

Learn about the risks

Cancel | Allow

Sample OAuth URL for Authentication: OpenID connect

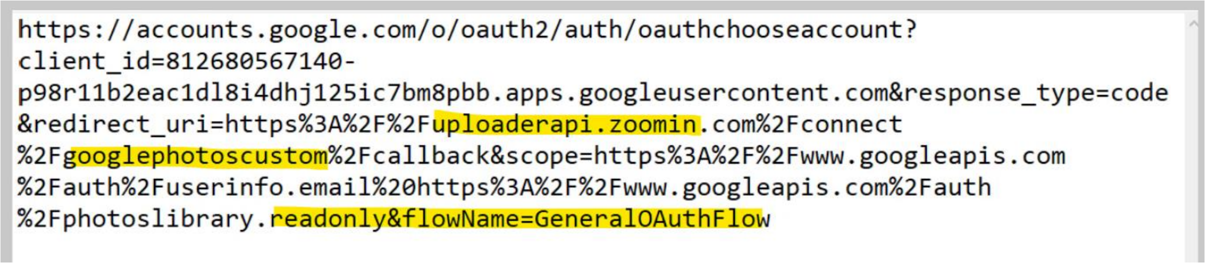
dominos 3rd party application accessing the Google Account resource



```
https://accounts.google.com/o/oauth2/auth/oauthchooseaccount?
redirect_uri=storagerelay%3A%2F%2Fhttps%2Fpizzaonline.dominos.co.in%3Fid
%3Dauth921226&response_type=permission%20id_token&scope=email%20profile
%20openid&openid.realm&client_id=928101681689-
fhhil6jq8g1ftone3bdrt3gfq03bilob.apps.googleusercontent.com&ss_domain=https
%3A%2F
%2Fpizzaonline.dominos.co.in&prompt&fetch_basic_profile=true&gsiwebsdk=2&flowN
ame=GeneralOAuthFlow
```

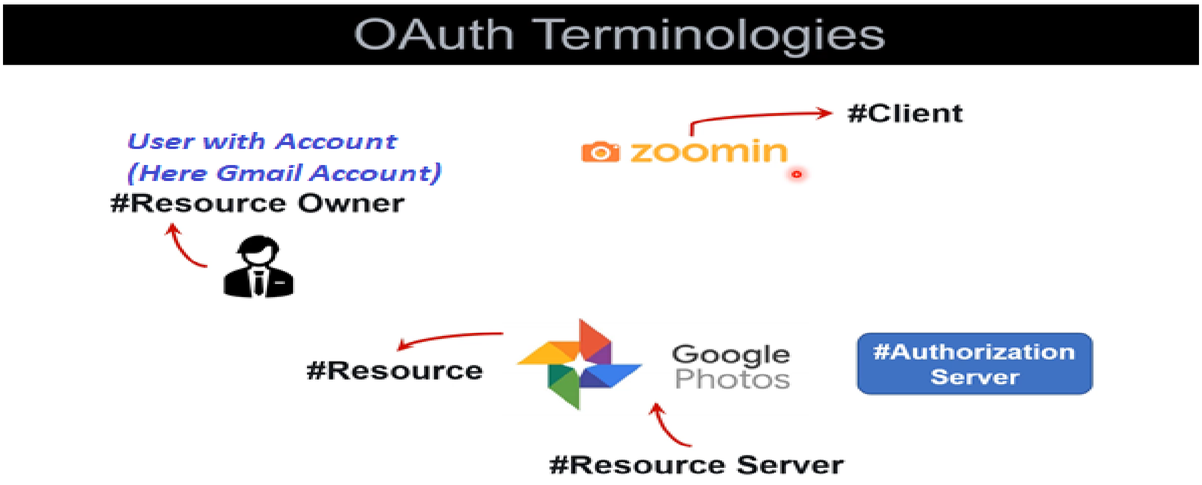
Sample OAuth URL for Authorization:

zoomin 3rd party application accessing the Google photos resource

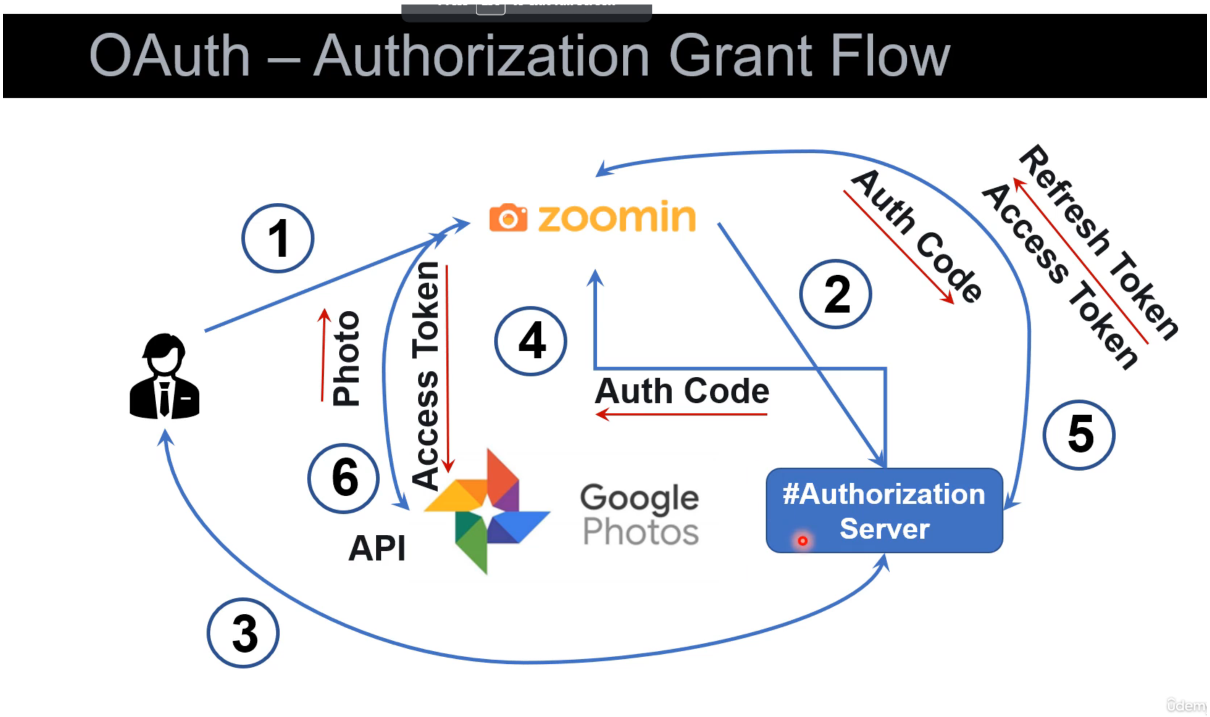


```
https://accounts.google.com/o/oauth2/auth/oauthchooseaccount?
client_id=812680567140-
p98r11b2eac1dl8i4dhj125ic7bm8pbb.apps.googleusercontent.com&response_type=code
&redirect_uri=https%3A%2F%2Fuploaderapi.zoomin.com%2Fconnect
%2Fgooglephotoscustom%2Fcallback&scope=https%3A%2F%2Fwww.googleapis.com
%2Fauth%2Fuserinfo.email%20https%3A%2F%2Fwww.googleapis.com%2Fauth
%2Fphotoslibrary.readonly&flowName=GeneralOAuthFlow
```

1. ***OAuth Terminologies:***



OAuth – Authorization Grant Flow : *A widely used flow for OAuth mechanisms*



OAuth – Authorization Grant Flow

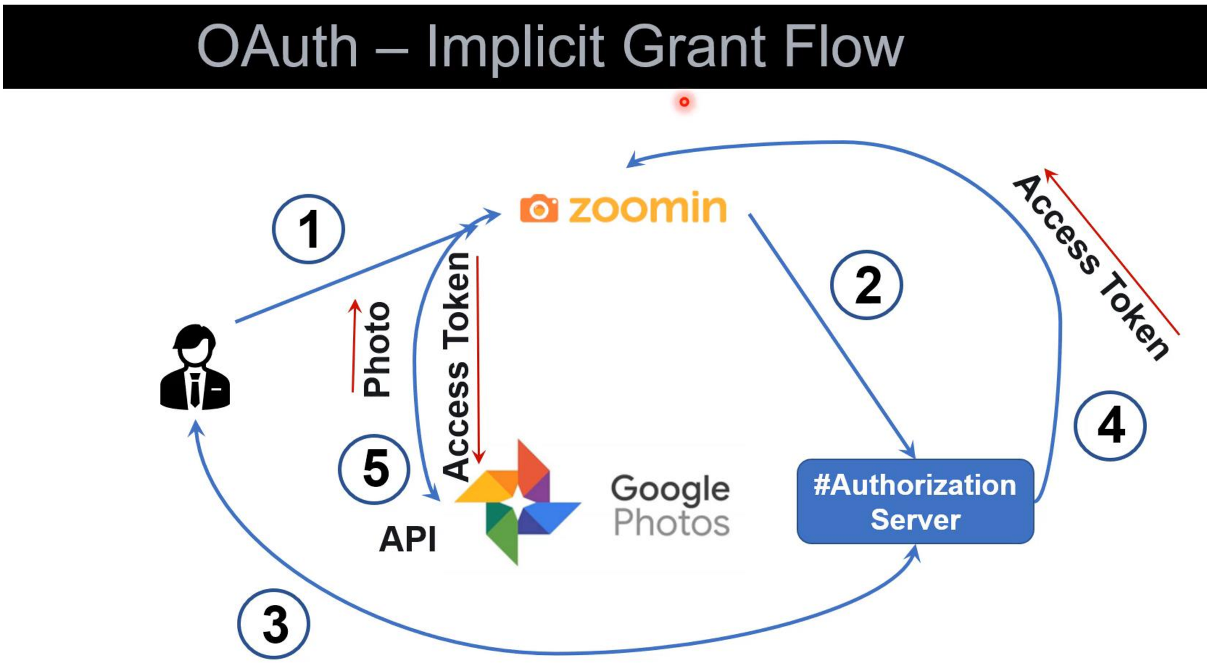
2. **User Authorization:** The user initiates the process by requesting access to their Google Photos via the Zoomin application. This is typically done through an interface in the Zoomin application that asks the user to log in to their Google account and authorize Zoomin to access their Google Photos.
3. **Authorization Request:** Zoomin redirects the user to Google's authorization server, where they can log in and grant permission for Zoomin to access their photos. This is done by presenting the user with a consent screen where they can see what kind of access Zoomin is requesting (in this case, access to Google Photos) and decide whether to grant it.
4. **Authorization Grant:** Upon approval, an authorization code is generated and sent back to Zoomin. This authorization code is a temporary credential that represents the user's consent for Zoomin to access their Google Photos. It is important to note that the authorization code is short-lived and can only be used once.
5. **Exchange Auth Code for Tokens:** Zoomin exchanges this authorization code for an access token and a refresh token from Google's authorization server. The access token is a string representing the authorization granted to the client (Zoomin). This token provides the client with secure access to the user's Google Photos resources via the Google Photos API. The access token is used in every API request and it has a limited lifetime, typically around one hour. Once the access

token expires, it can no longer be used to access the user's resources. The refresh token is also a string, but its purpose is to obtain a new access token when the current access token expires. Unlike access tokens, refresh tokens are usually long-lived. When the access token expires, the client can send a request to the authorization server, including the refresh token, to get a new access token. This process is done without any interaction from the user, providing a seamless experience.

6. **Access Granted:** With the access token, Zoomin can now retrieve resources (photos) from Google Photos API on behalf of the user. The access token is included in the header of each API request, allowing Zoomin to access the user's photos without needing to know their Google credentials.
7. **Resource Delivery:** The requested photos are delivered to the user through the Zoomin application. This is done by Zoomin making a request to the Google Photos API with the access token, retrieving the photos, and then displaying them in the Zoomin application.

This flow ensures that the user's credentials are not shared with the third-party application (Zoomin), enhancing the security of the user's data. The access token allows Zoomin to access the user's Google Photos resources without needing the user's Google credentials. The refresh token allows Zoomin to obtain a new access token once the current one expires, ensuring a seamless user experience. This is a common practice in many web and mobile applications to securely access user data from other platforms.

OAuth Implicit Grant Flow:



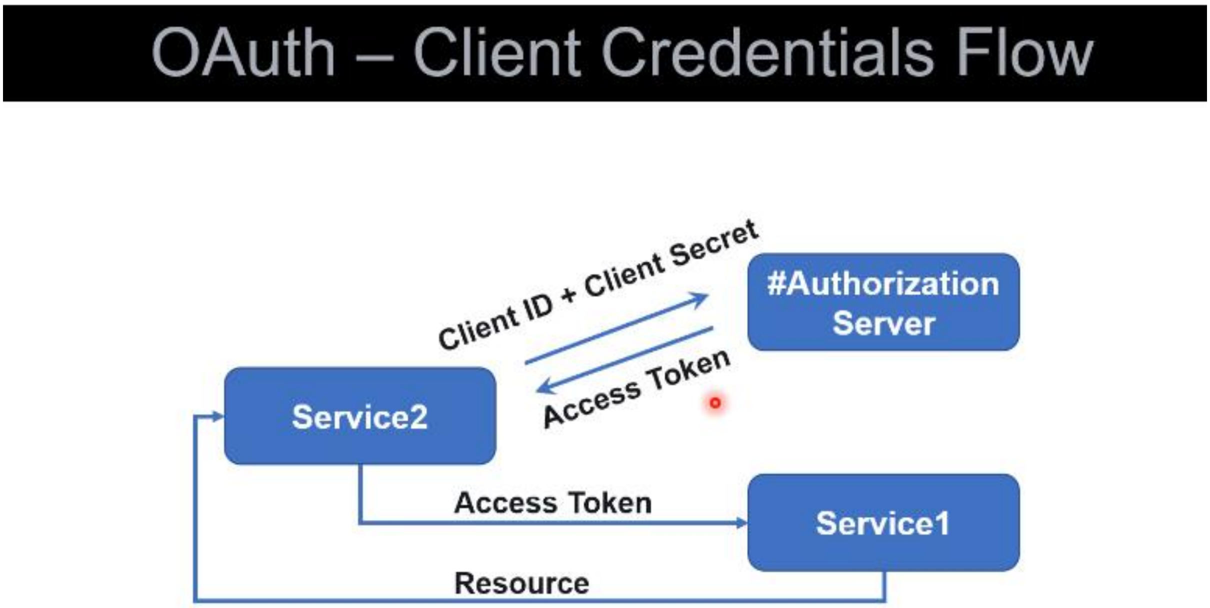
OAuth – Implicit Grant Flow

1. **User Authentication:** The user initiates the process by requesting access to their Google Photos via the Zoomin application. This is typically done through an interface in the Zoomin application that asks the user to log in to their Google account and authorize Zoomin to access their Google Photos.
2. **Authorization Request:** Zoomin redirects the user to Google's authorization server, where they are prompted to grant or deny permission for Zoomin to access their photos. This is done by presenting the user with a consent screen where they can see what kind of access Zoomin is requesting (in this case, access to Google Photos) and decide whether to grant it.
3. **Access Token Retrieval:** If permission is granted, an access token is immediately issued by Google's authorization server. The user is then redirected back to Zoomin with this token included in the URL fragment. Unlike the Authorization Code Grant flow, there is no need to exchange an authorization code for an access token in the Implicit Grant flow. This makes the process faster, but also less secure as the access token can potentially be exposed in the browser history or logs.
4. **Access Token Extraction:** Zoomin extracts the access token from the URL fragment. This is done using JavaScript running in the user's browser. Once the token is extracted, it is typically stored in the application's session for use in subsequent API requests.

5. **Resource Access:** With this token, Zoomin can now directly request and retrieve specific resources (like photos) from Google Photos API on behalf of the user. The access token is included in the header of each API request, allowing Zoomin to access the user’s photos without needing to know their Google credentials.

   This flow is typically used for JavaScript-based applications running in the browser where the access token is immediately needed and the refresh token is not used. It’s worth noting that due to its inherent security risks, the Implicit Grant flow is no longer recommended for most applications

OAuth Client Credentials Flow:



OAuth – Client Credentials Flow

1. **Client Authentication:** The client application (Service2) sends its Client ID and Client Secret to the Authorization Server. These credentials are used to authenticate the client application. The Client ID is a public identifier for the application, while the Client Secret is a confidential key used to secure the application’s communication with the Authorization Server.

2. **Access Token Issuance:** The Authorization Server validates the Client ID and Client Secret. If these credentials are valid, the Authorization Server issues an Access Token to the client application (Service2). This Access Token is a string representing the authorization granted to the client.

3. **Access Token Usage:** The client application (Service2) uses this Access Token to request the protected resource from the Resource Server (Service1). The Access Token is included in the header of each API request.

4. **Resource Access:** The Resource Server (Service1) validates the Access Token. If the token is valid, the Resource Server allows access to the requested resource and returns it to the client application (Service2).

   This flow is typically used when the client application needs to access resources from a Resource Server on its own behalf, rather than on behalf of a user. It's a simpler flow compared to others in OAuth, as it doesn't involve any user interaction or redirections. However, it should only be used when the client application can securely store the Client Secret. If the Client Secret is compromised, anyone can impersonate the client application and gain access to the protected resources.

Important Links :

Using OAuth 2.0 to Access Google APIs | Authorization | Google for Developers

API Playground : OAuth 2.0 Playground (google.com)

JWT tokens. Start from here: JSON Web Tokens (auth0.com)

OAuth 2.0 and OpenID Connect Authentication Flow The usual OAuth 2.0 authorization code flow looks like this:

1. The client requests authorization from the resource owner (usually the user).
2. If the owner gives authorization, the client passes the authorization grant to the authorization server.
3. If the grant is valid, the authorization server returns an access token, possibly alongside a refresh and/or ID token.
4. The client now uses that access token to access the resource server.

OpenID Connect (OIDC) OpenID Connect is an identity layer on top of the OAuth 2.0 protocol. It extends OAuth 2.0 with user authentication and Single Sign-On (SSO) functionality. It enables you to retrieve and store authentication information about your end users.

ID Token The ID token is an artifact that proves that the user has been authenticated3. It contains claims that carry information about the user. They can be sent alongside or instead of an access token. Information in ID tokens enables the client to verify that a user is who they claim to be.

JWT (JSON Web Token) – ID Token Way: JWT is an open standard that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret (with the HMAC algorithm) or a public/private key pair using RSA or ECDSA.

JWT Structure A typical JWT consists of three parts separated by dots (.):

Header: The header typically consists of two parts: the type of the token, which is JWT, and the signing algorithm being used, such as HMAC SHA256 or RSA. For example:

JSON

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

Payload: The payload contains the claims. Claims are statements about an entity (typically, the user) and additional data. There are three types of claims: registered, public, and private claims. For example:

```
JSON

{
  "userId":"b07f85be-45da",
  "iss": "https://provider.domain.com/",
  "sub": "auth/some-hash-here",
  "exp": 153452683
}
```

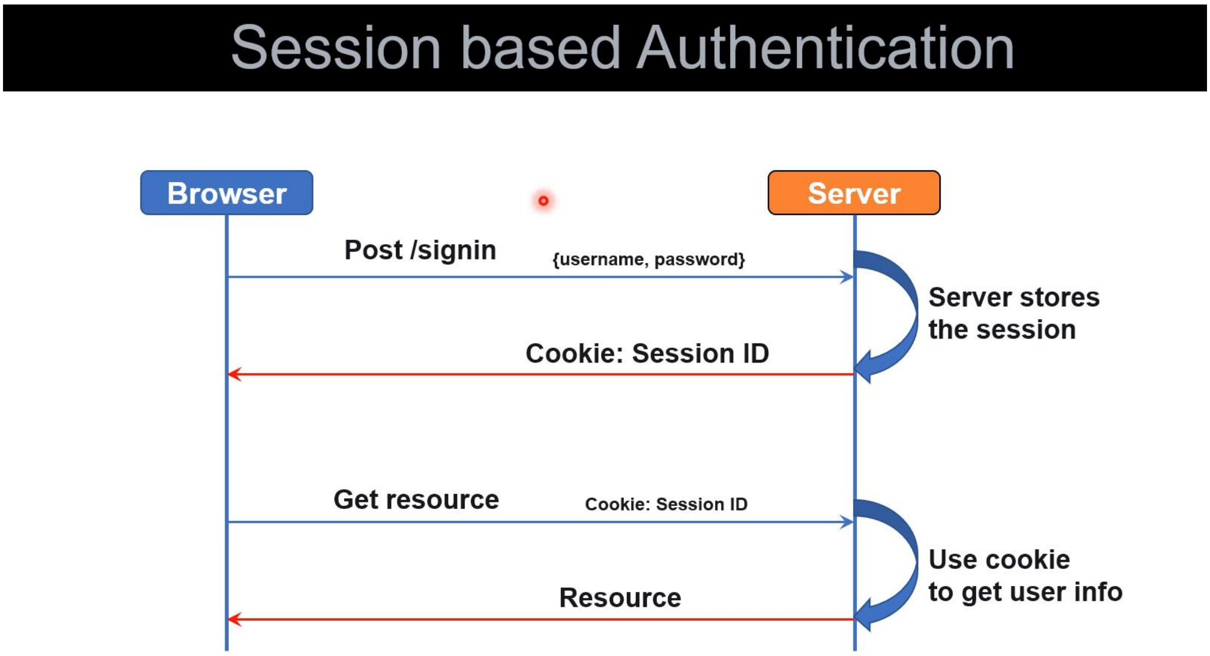
Signature (CERT): The signature is used to verify that the sender of the JWT is who it says it is and to ensure that the message wasn't changed along the way.

Session-Based Authentication:

Session-based authentication is a mechanism where the user state is maintained on the server side. When a user logs in, using their username and password, via a POST request to the "/signin" endpoint, the server validates these credentials. Upon successful validation, it generates a unique session ID for that particular user session.

This session ID is then sent back to the client's browser and stored as a cookie. Every subsequent request from the client to access restricted resources includes this cookie (session ID). The server uses this ID to retrieve the user's session information stored on its end.

In contrast with token-based authentication where tokens are used for validating and maintaining user states across multiple services or domains, session IDs in session-based authentication are specific to one domain and one server. This ensures an additional layer of security as these cannot be used interchangeably across different domains or services.



Session based Authentication

Here’s how it works step by step as depicted in the image:

1. The browser sends a POST request containing username and password to “/signin” endpoint.
2. Server validates credentials; if valid, creates a unique Session ID.
3. Server sends back Session ID which gets stored as a cookie in browser.
4. For subsequent requests for resources, browser sends Cookie containing Session ID.
5. Server retrieves Session info using Session ID; if valid, sends requested resource back.

In summary, session-based authentication involves the server maintaining user sessions and storing session information. This is different from token-based authentication, such as JWT, which relies on the validity of tokens for user authentication. The choice between these approaches often depends on the web application’s requirements and whether it aims to maintain user state. Some websites may even opt for a combination of both authentication methods based on their specific needs.

Form-Based Authentication and CSRF Token :

Definition:

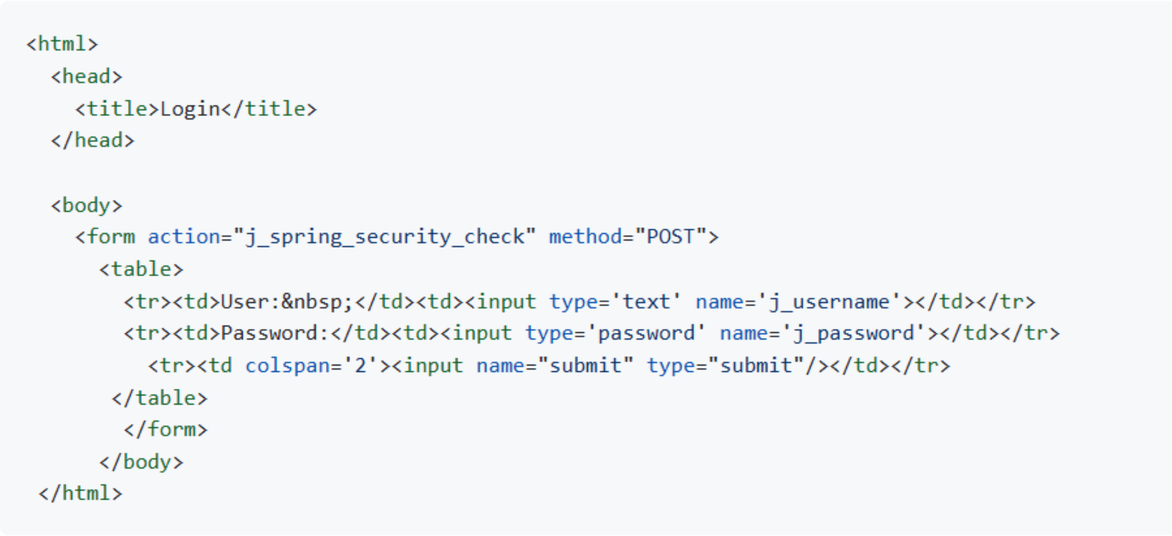
User authentication through an editable HTML form.
Users fill in and submit the form with a username and password.
Commonly used for web applications that maintain user state or sessions.

Rest Assured Support for Form-Based Authentication :

Rest Assured supports form-based authentication using the auth.form method. A simple HTML form is presented to the user for login, typically with username and password fields.

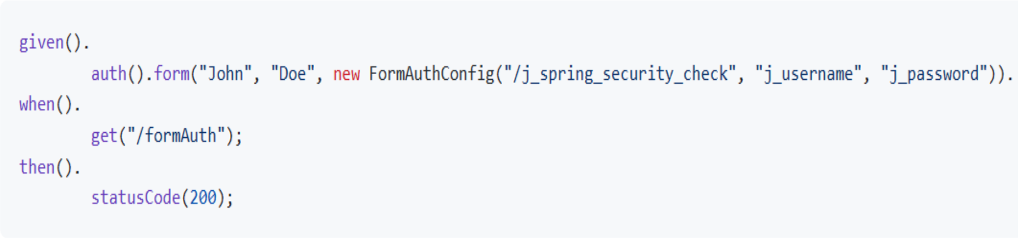
URL : [Usage · rest-assured/rest-assured Wiki · GitHub]()

Sample HTML :



```
<html>
  <head>
    <title>Login</title>
  </head>

  <body>
    <form action="j_spring_security_check" method="POST">
      <table>
        <tr><td>User:&nbsp;</td><td><input type='text' name='j_username'></td></tr>
        <tr><td>Password:</td><td><input type='password' name='j_password'></td></tr>
          <tr><td colspan='2'><input name="submit" type="submit"/></td></tr>
       </table>
        </form>
      </body>
 </html>
```

Rest Assured :



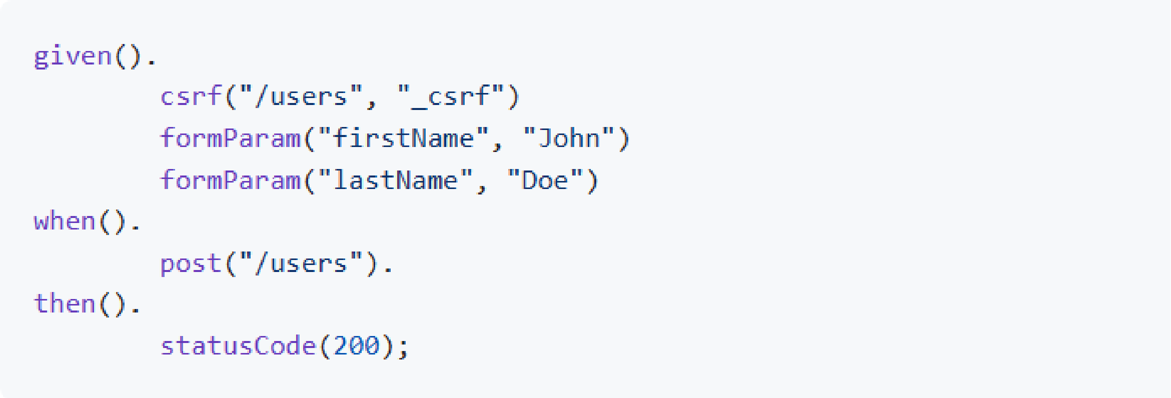
```
given().
        auth().form("John", "Doe", new FormAuthConfig("/j_spring_security_check", "j_username", "j_password")).
when().
        get("/formAuth");
then().
        statusCode(200);
```

CSRF Token and Its Importance

CSRF (Cross-Site Request Forgery) Token:
A security measure to prevent unauthorized actions initiated by an attacker on behalf of an authenticated user. The server may send a CSRF token along with the session ID to enhance security. CSRF token is a randomly generated string that is unpredictable and challenging to hack.

Rest Assured Support for CSRF Token

Rest Assured has built-in support for CSRF token handling. It has multiple ways to achieve it according to needs.

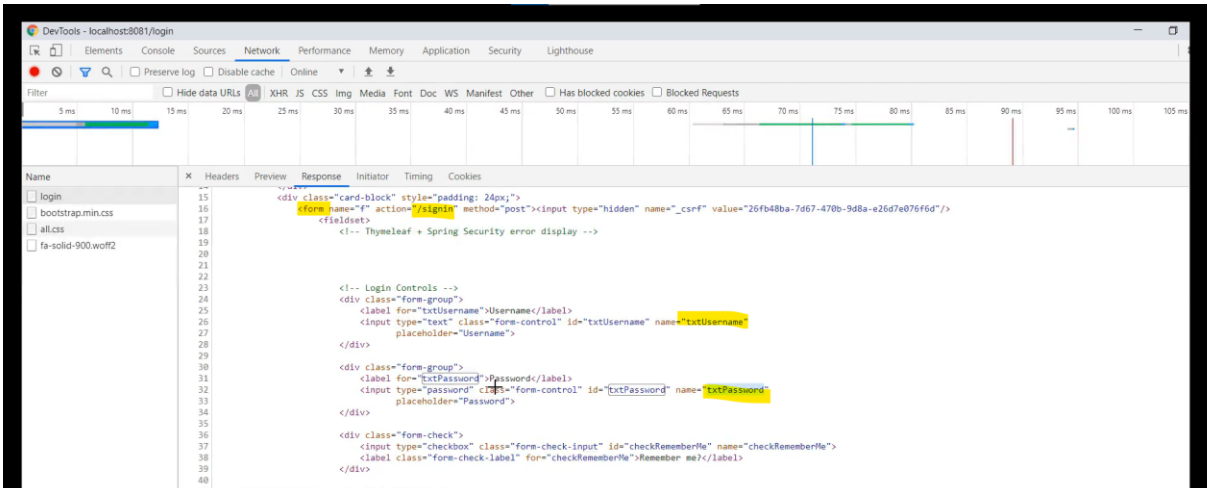


```
given().
        csrf("/users", "_csrf")
        formParam("firstName", "John")
        formParam("lastName", "Doe")
when().
        post("/users").
then().
        statusCode(200);
```

Considerations for Automated Testing

Understand the structure of the HTML form and identify the fields needed for authentication. Ensure proper configuration for CSRF token handling based on the application's requirements.

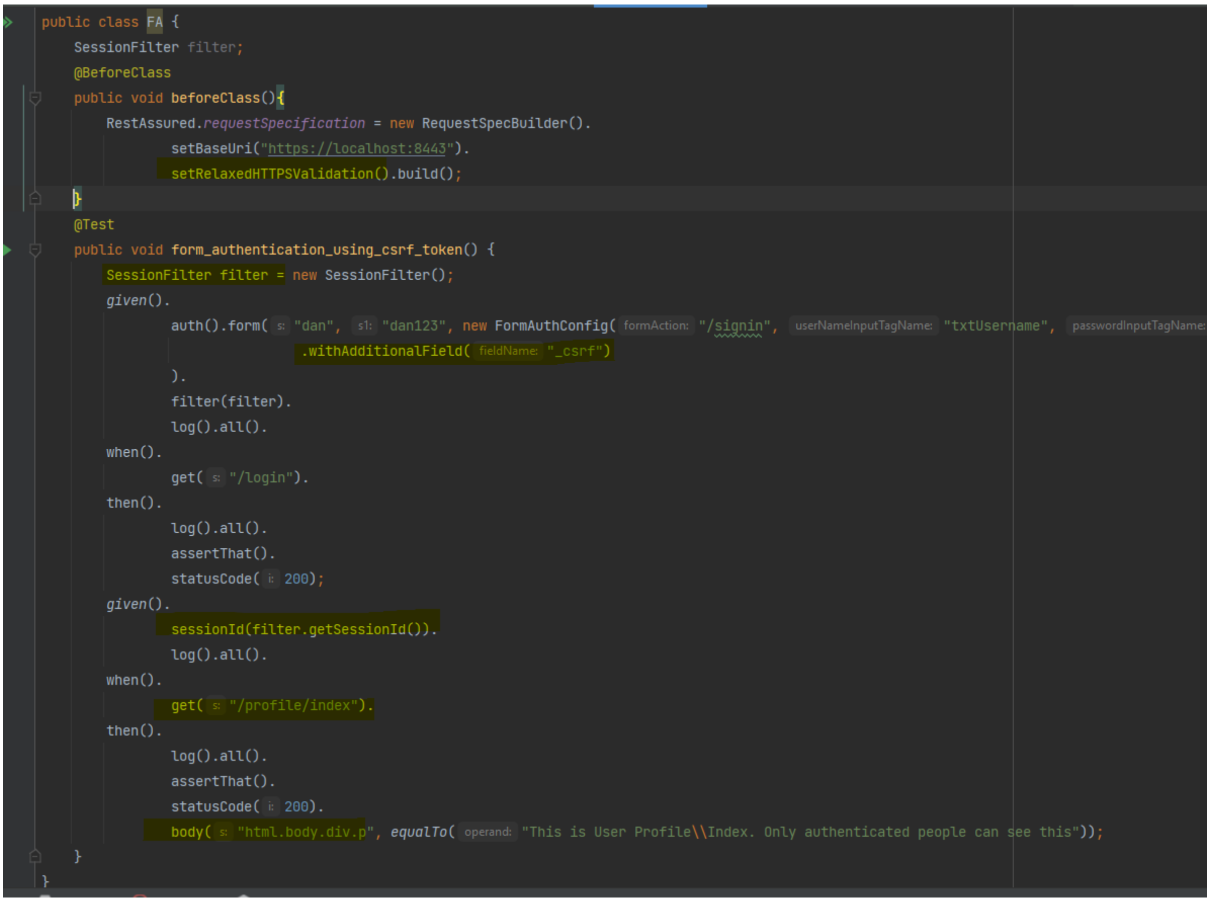
Sample HTML Form from the Demo App: [RomanianCoderExamples/SpringBootSecurity at master · dangeabunea/RomanianCoderExamples · GitHub]()



```
        <div class="card-block" style="padding: 24px;">
            <form name="f" action="/signin" method="post"><input type="hidden" name="_csrf" value="26fb48ba-7d67-470b-9d8a-e26d7e076f6d"/>
                <fieldset>
                    <!-- Thymeleaf + Spring Security error display -->




                    <!-- Login Controls -->
                    <div class="form-group">
                        <label for="txtUsername">Username</label>
                        <input type="text" class="form-control" id="txtUsername" name="txtUsername"
                               placeholder="Username">
                    </div>

                    <div class="form-group">
                        <label for="txtPassword">Password</label>
                        <input type="password" class="form-control" id="txtPassword" name="txtPassword"
                               placeholder="Password">
                    </div>

                    <div class="form-check">
                        <input type="checkbox" class="form-check-input" id="checkRememberMe" name="checkRememberMe">
                        <label class="form-check-label" for="checkRememberMe">Remember me?</label>
                    </div>
```

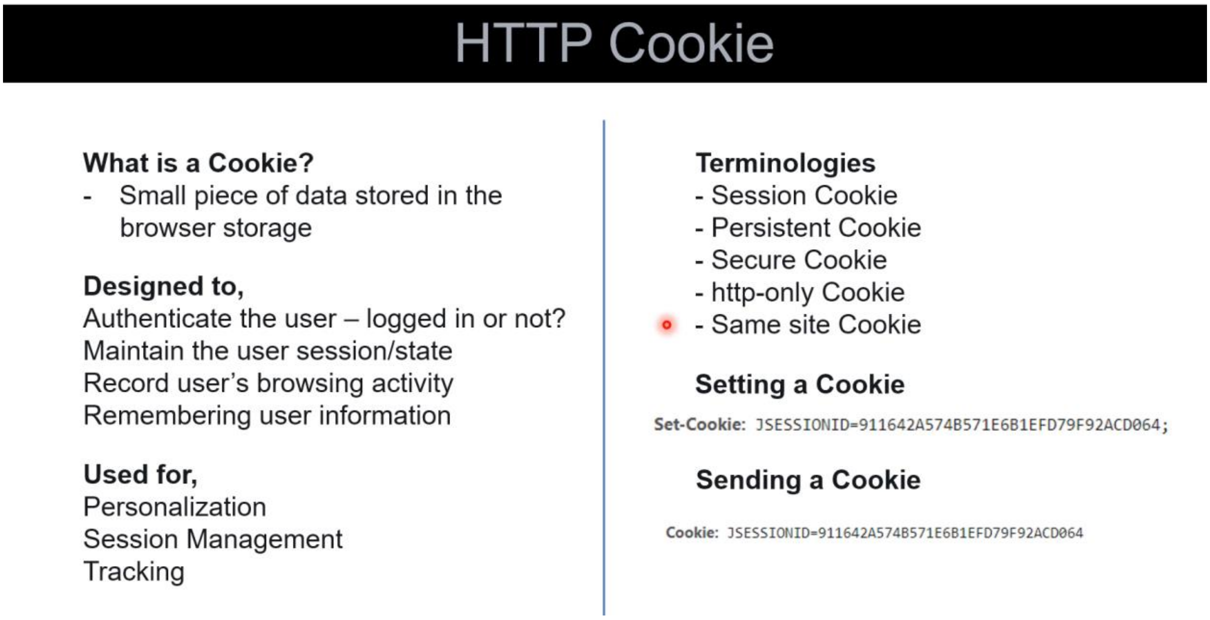
Rest Assured Form authentication for the above HTML form:

HTTP validation as rest assured by default looking for HTTPS. Field csrf token is passed to have the token and extracting JSESSIONID and giving for next request. Finally HTML content is show which can also be validated.



```
public class FA {
    SessionFilter filter;
    @BeforeClass
    public void beforeClass(){
        RestAssured.requestSpecification = new RequestSpecBuilder().
                setBaseUri("https://localhost:8443").
                setRelaxedHTTPSValidation().build();
    }
    @Test
    public void form_authentication_using_csrf_token() {
        SessionFilter filter = new SessionFilter();
        given().
                auth().form( s: "dan",  s1: "dan123", new FormAuthConfig( formAction: "/signin",  userNameInputTagName: "txtUsername",  passwordInputTagName:
                                .withAdditionalField( fieldName: "_csrf")
                ).
                filter(filter).
                log().all().
        when().
                get( s: "/login").
        then().
                log().all().
                assertThat().
                statusCode( i: 200);
        given().
                sessionId(filter.getSessionId()).
                log().all().
        when().
                get( s: "/profile/index").
        then().
                log().all().
                assertThat().
                statusCode( i: 200).
                body( s: "html.body.div.p", equalTo( operand: "This is User Profile\\Index. Only authenticated people can see this"));
    }
}
```

HTTP Cookies in Rest Assured*:*



HTTP Cookie

What is a Cookie?

- Small piece of data stored in the browser storage

Designed to,
Authenticate the user – logged in or not?
Maintain the user session/state
Record user's browsing activity
Remembering user information

Used for,
Personalization
Session Management
Tracking

Terminologies
- Session Cookie
- Persistent Cookie
- Secure Cookie
- http-only Cookie
- Same site Cookie

Setting a Cookie

```
Set-Cookie: JSESSIONID=911642A574B571E6B1EFD79F92ACD064;
```

Sending a Cookie

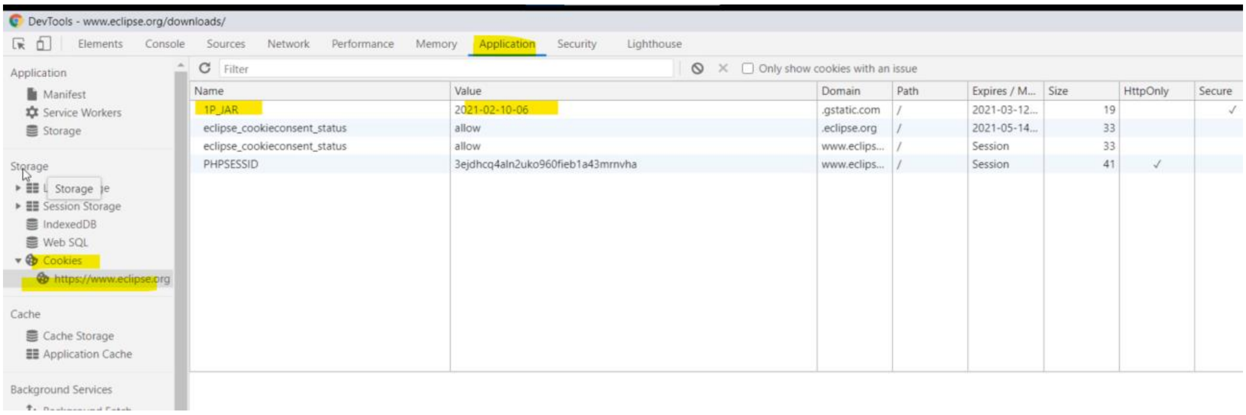
```
Cookie: JSESSIONID=911642A574B571E6B1EFD79F92ACD064
```

Understanding HTTP Cookies

Definition: Small pieces of data stored in browser storage. Key – Value pairs in general. Also known as browser cookies or internet cookies. Designed to authenticate users and maintain user sessions.

Purpose of Cookies in Web Applications

- Authenticate users: Verify if a user is logged in.
- Session management: Maintain user sessions and states.
- Personalization: Store user preferences, shopping cart items, etc.
- Tracking: Record browsing activity, pages visited, buttons clicked, etc.



Name	Value	Domain	Path	Expires / M...	Size	HttpOnly	Secure
1P_JAR	2021-02-10-06	.gstatic.com	/	2021-03-12...	19		✓
eclipse_cookieconsent_status	allow	.eclipse.org	/	2021-05-14...	33		
eclipse_cookieconsent_status	allow	www.eclips...	/	Session	33		
PHPSESSID	3ejdhcq4aln2uko960fieb1a43mrnvha	www.eclips...	/	Session	41	✓	

Types of Cookies

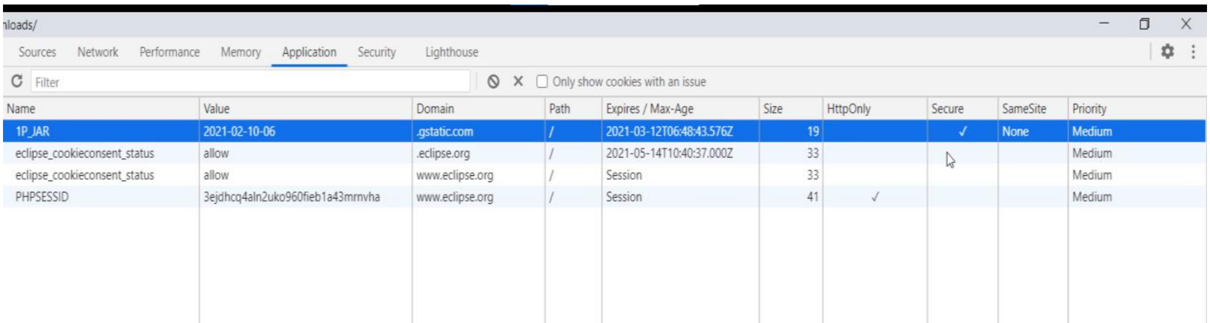
- **Session Cookie:**
  - Active as long as the user is on the browser.
  - Expires when the user closes the browser or signs out.
- **Persistent Cookie:**
  - Persists even if the browser is closed or the user signs out.
  - Typically has an expiry time.
- **Secure Cookie:**
  - Works only on HTTPS protocol.
  - Cannot be transmitted on HTTP for enhanced security.
- **HttpOnly Cookie:**
  - Cannot be read by JavaScript running on the client side.
  - Adds security by preventing JavaScript access.
- **SameSite Cookie:**
  - Restricts cookies to a single domain.
  - A cookie set for one domain cannot be used for another.

Cookie Setup Process

1. Client makes the first request to the server.
2. Server creates a cookie and sends it as part of the response header (Set-Cookie).
3. Client stores the cookie in the browser storage.
4. Subsequent API calls include the cookie as part of the request header (Cookie).

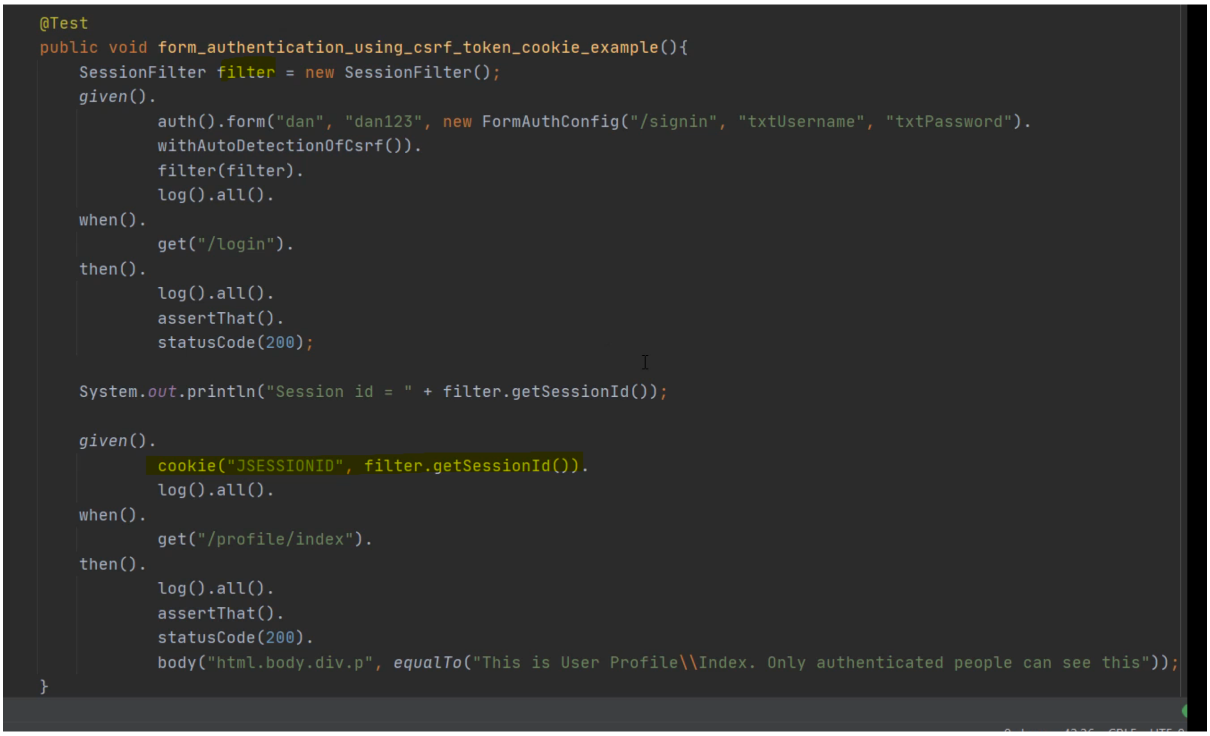
Example:

- Cookies can be inspected using browser developer tools.
- Cookies are visible under the "Application" tab in Chrome DevTools.
- Different cookies have varying attributes such as session, expiration, secure, and HttpOnly.



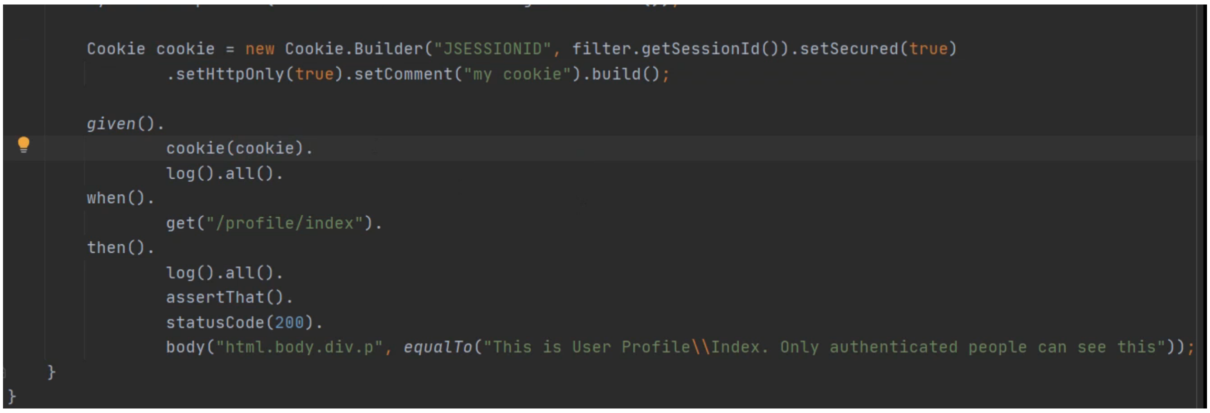
Name	Value	Domain	Path	Expires / Max-Age	Size	HttpOnly	Secure	SameSite	Priority
1P_JAR	2021-02-10-06	.gstatic.com	/	2021-03-12T06:48:43.576Z	19		✓	None	Medium
eclipse_cookieconsent_status	allow	.eclipse.org	/	2021-05-14T10:40:37.000Z	33				Medium
eclipse_cookieconsent_status	allow	www.eclipse.org	/	Session	33				Medium
PHPSESSID	3ejdhcq4aln2uko960fieb1a43mrnvha	www.eclipse.org	/	Session	41	✓			Medium

Rest Assured Code:



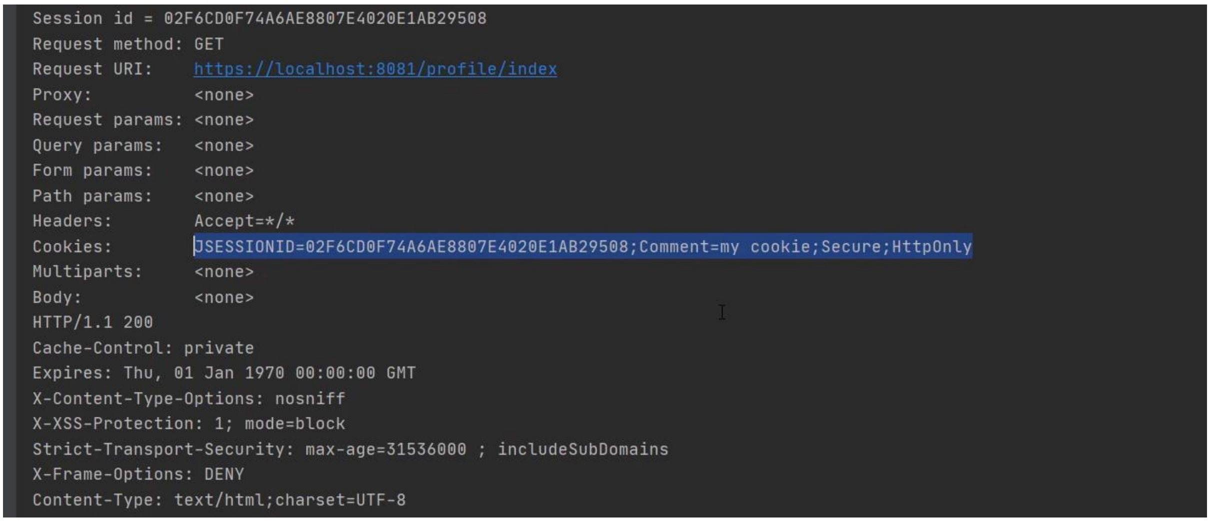
```java
@Test
public void form_authentication_using_csrf_token_cookie_example(){
    SessionFilter filter = new SessionFilter();
    given().
            auth().form("dan", "dan123", new FormAuthConfig("/signin", "txtUsername", "txtPassword").
            withAutoDetectionOfCsrf()).
            filter(filter).
            log().all().
    when().
            get("/login").
    then().
            log().all().
            assertThat().
            statusCode(200);

    System.out.println("Session id = " + filter.getSessionId());

    given().
            cookie("JSESSIONID", filter.getSessionId()).
            log().all().
    when().
            get("/profile/index").
    then().
            log().all().
            assertThat().
            statusCode(200).
            body("html.body.div.p", equalTo("This is User Profile\\Index. Only authenticated people can see this"));
}
```

Sending Cookies Using Cookie Builder from Rest Assured:



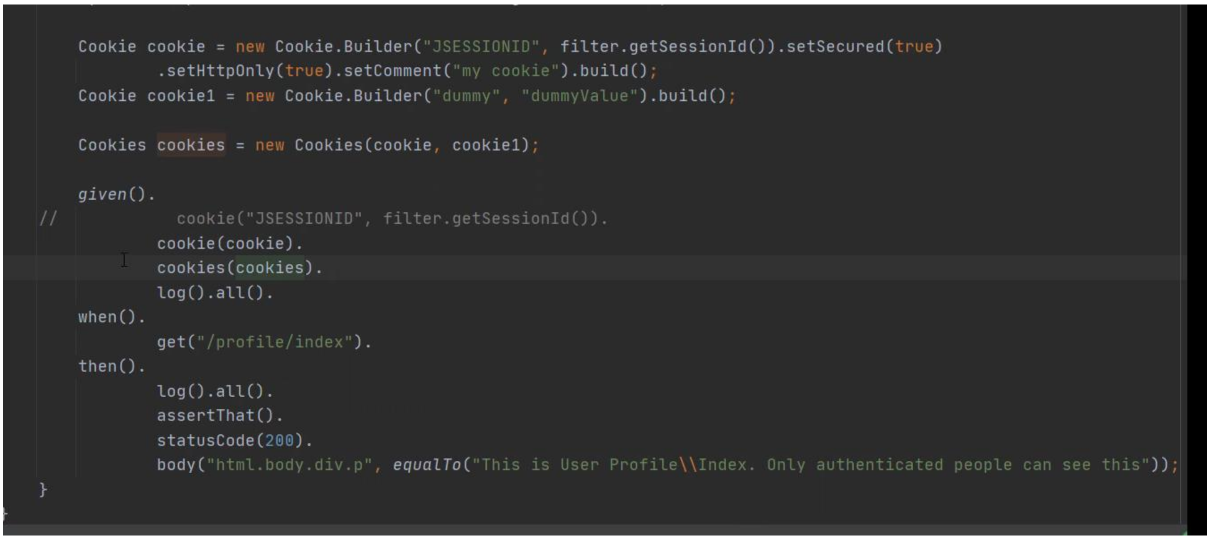
```java
        Cookie cookie = new Cookie.Builder("JSESSIONID", filter.getSessionId()).setSecured(true)
                .setHttpOnly(true).setComment("my cookie").build();

        given().
                cookie(cookie).
                log().all().
        when().
                get("/profile/index").
        then().
                log().all().
                assertThat().
                statusCode(200).
                body("html.body.div.p", equalTo("This is User Profile\\Index. Only authenticated people can see this"));
    }
}
```

Output:



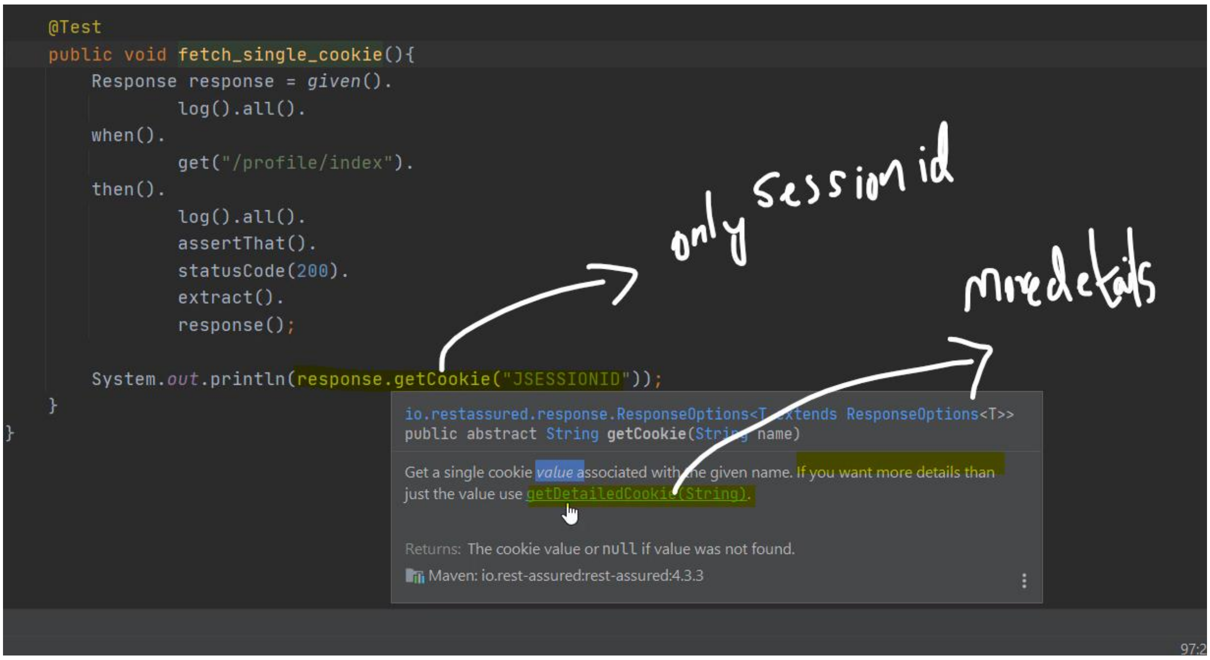
```
Session id = 02F6CD0F74A6AE8807E4020E1AB29508
Request method: GET
Request URI:    https://localhost:8081/profile/index
Proxy:          <none>
Request params: <none>
Query params:   <none>
Form params:    <none>
Path params:    <none>
Headers:        Accept=*/*
Cookies:        JSESSIONID=02F6CD0F74A6AE8807E4020E1AB29508;Comment=my cookie;Secure;HttpOnly
Multiparts:     <none>
Body:           <none>
HTTP/1.1 200
Cache-Control: private
Expires: Thu, 01 Jan 1970 00:00:00 GMT
X-Content-Type-Options: nosniff
X-XSS-Protection: 1; mode=block
Strict-Transport-Security: max-age=31536000 ; includeSubDomains
X-Frame-Options: DENY
Content-Type: text/html;charset=UTF-8
```

Multiple Cookies Passing:



```
    Cookie cookie = new Cookie.Builder("JSESSIONID", filter.getSessionId()).setSecured(true)
            .setHttpOnly(true).setComment("my cookie").build();
    Cookie cookie1 = new Cookie.Builder("dummy", "dummyValue").build();

    Cookies cookies = new Cookies(cookie, cookie1);

    given().
//            cookie("JSESSIONID", filter.getSessionId()).
            cookie(cookie).
            cookies(cookies).
            log().all().
    when().
            get("/profile/index").
    then().
            log().all().
            assertThat().
            statusCode(200).
            body("html.body.div.p", equalTo("This is User Profile\\Index. Only authenticated people can see this"));
}
```

Fetch Single Cookie:



```java
@Test
public void fetch_single_cookie(){
    Response response = given().
            log().all().
    when().
            get("/profile/index").
    then().
            log().all().
            assertThat().
            statusCode(200).
            extract().
            response();

    System.out.println(response.getCookie("JSESSIONID"));
}
}
```

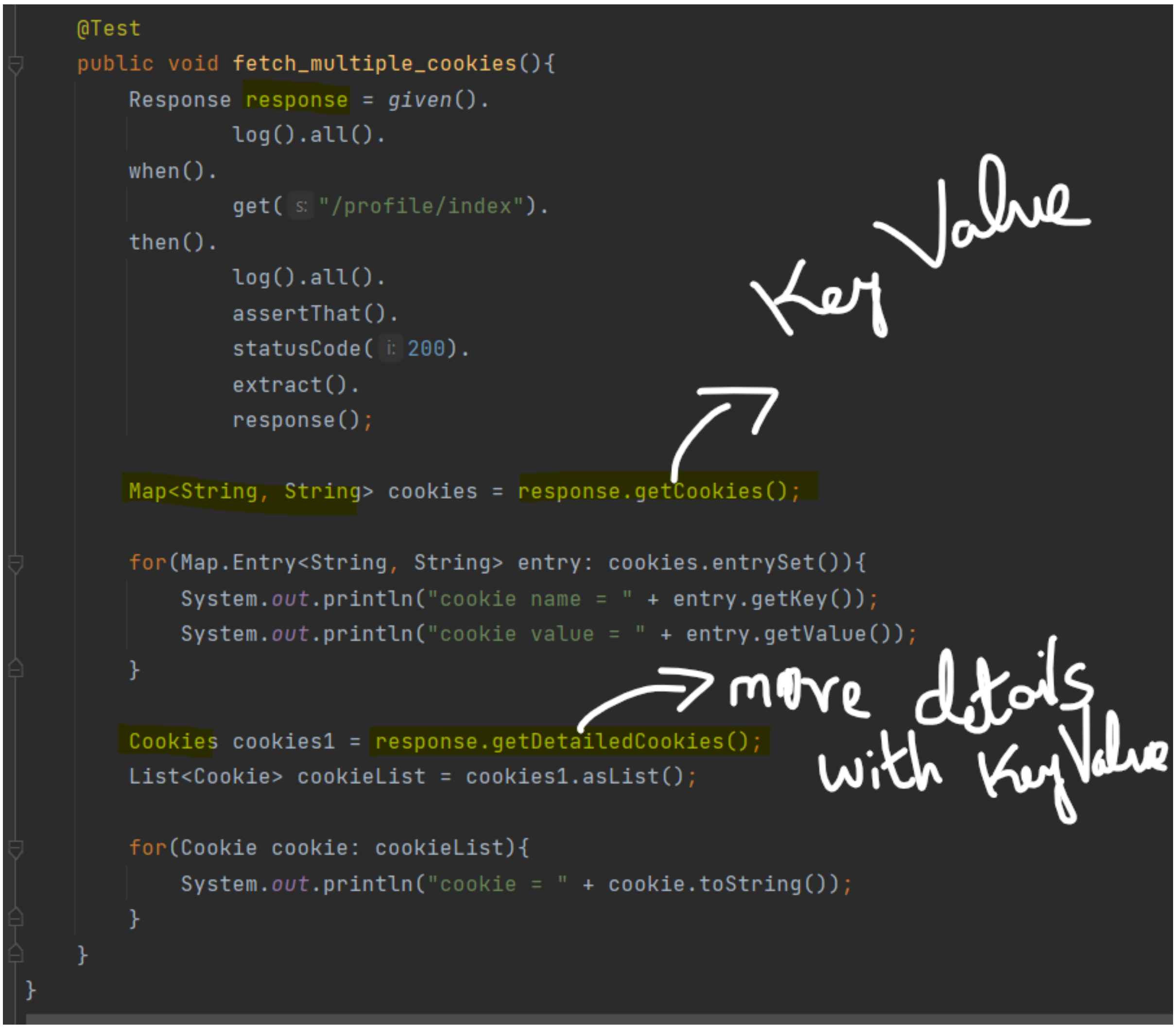
```
io.restassured.response.ResponseOptions<T extends ResponseOptions<T>>
public abstract String getCookie(String name)
```

Get a single cookie *value* associated with the given name. If you want more details than just the value use getDetailedCookie(String).

Returns: The cookie value or `null` if value was not found.

Maven: io.rest-assured:rest-assured:4.3.3

Fetch Multiple Cookies:



```
    @Test
    public void fetch_multiple_cookies(){
        Response response = given().
                log().all().
        when().
                get( s: "/profile/index").
        then().
                log().all().
                assertThat().
                statusCode( i: 200).
                extract().
                response();

        Map<String, String> cookies = response.getCookies();

        for(Map.Entry<String, String> entry: cookies.entrySet()){
            System.out.println("cookie name = " + entry.getKey());
            System.out.println("cookie value = " + entry.getValue());
        }

        Cookies cookies1 = response.getDetailedCookies();
        List<Cookie> cookieList = cookies1.asList();

        for(Cookie cookie: cookieList){
            System.out.println("cookie = " + cookie.toString());
        }
    }
}
```

MISC:

Certs and Rest Assured :
https://www.youtube.com/playlist?list=PLIWM60RqoEduEKvBntAArbI43mDPQWhWd

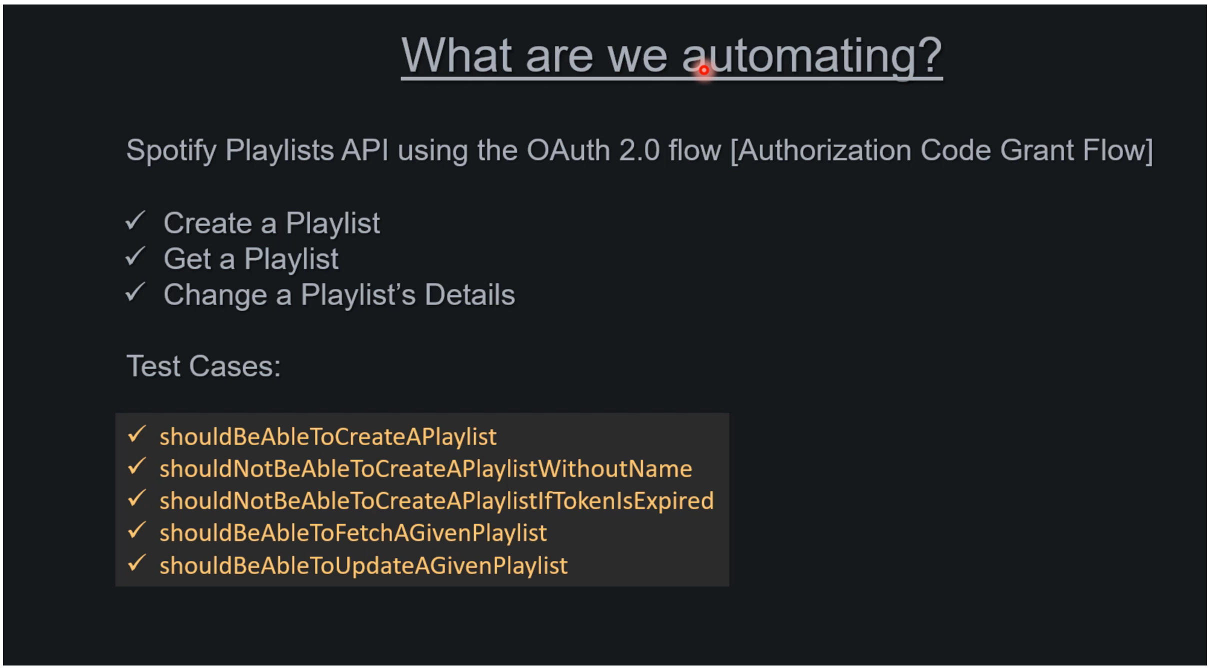
Framework Design

- Scalable and extensible for adding new APIs.
- Reusable methods for request and response specifications.
- Reusable methods for API requests to reduce code.
- Separation of API layer from the test layer for clarity.
- Use of POJOs for serialization and deserialization.
- Implementation of Singleton design pattern for configuration.
- Integration of Lombok for boilerplate code reduction.
- Implementation of the builder pattern for POJOs.
- Integration of Extent Reports for reporting and logging.
- Automation of positive and negative scenarios.
- Support for parallel execution.
- Implementation of data-driven test cases using test data providers.
- Token management: check expiry, renew if needed.
- Integration with Maven Surefire plugin for command-line execution.
- Integration with GitHub and Jenkins for end-to-end execution.

Tools and Technologies:

- Rest Assured for automation library testing.
- Java as the programming language.
- Reports for reporting.
- Hamcrest for assertions.
- Jackson API for Serialization and Deserialization.
- Lombok for reducing boilerplate code.

Goal for Automation:



What are we automating?

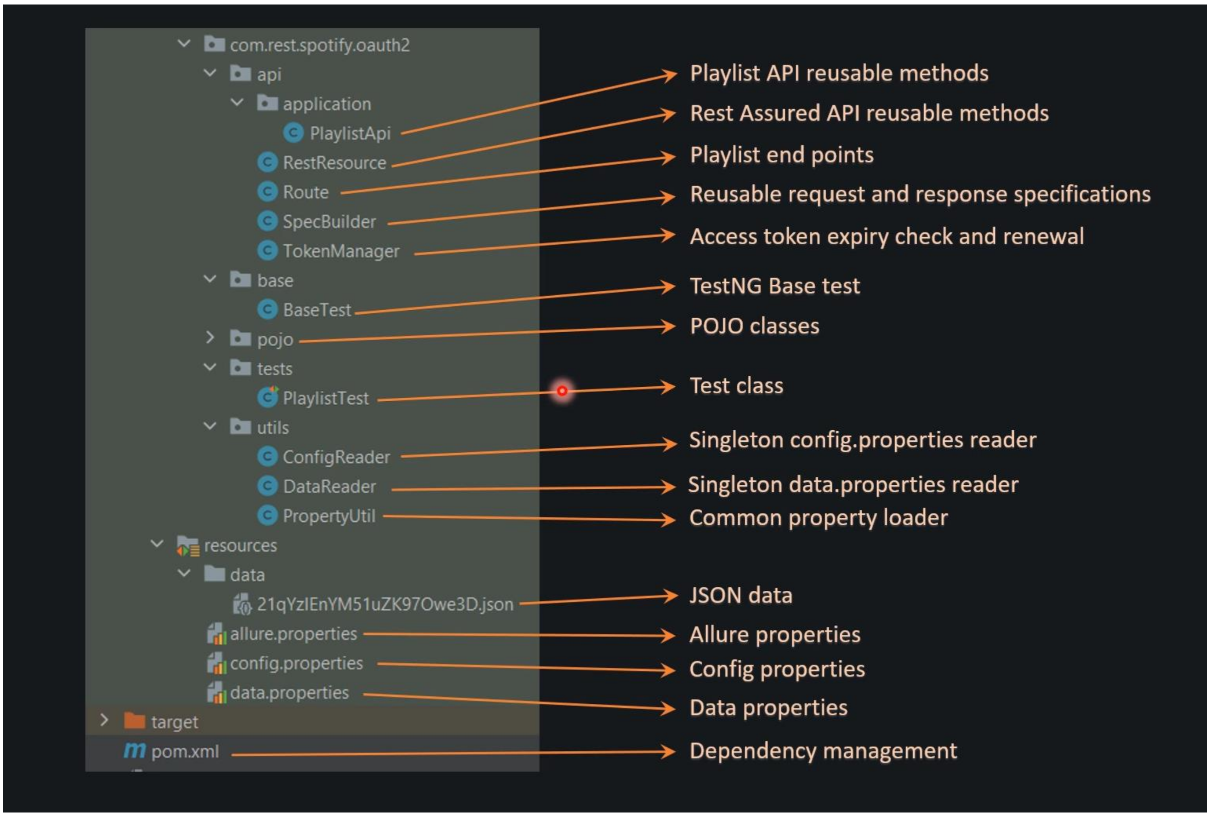
Spotify Playlists API using the OAuth 2.0 flow [Authorization Code Grant Flow]

- ✓ Create a Playlist
- ✓ Get a Playlist
- ✓ Change a Playlist's Details

Test Cases:

- ✓ shouldBeAbleToCreateAPlaylist
- ✓ shouldNotBeAbleToCreateAPlaylistWithoutName
- ✓ shouldNotBeAbleToCreateAPlaylistIfTokenIsExpired
- ✓ shouldBeAbleToFetchAGivenPlaylist
- ✓ shouldBeAbleToUpdateAGivenPlaylist

Structure:



```
com.rest.spotify.oauth2
  api
    application
      PlaylistApi
    RestResource
    Route
    SpecBuilder
    TokenManager
  base
    BaseTest
  pojo
  tests
    PlaylistTest
  utils
    ConfigReader
    DataReader
    PropertyUtil
resources
  data
    21qYzIEnYM51uZK97Owe3D.json
  allure.properties
  config.properties
  data.properties
target
pom.xml
```

Playlist API reusable methods

Rest Assured API reusable methods

Playlist end points

Reusable request and response specifications

Access token expiry check and renewal

TestNG Base test

POJO classes

Test class

Singleton config.properties reader

Singleton data.properties reader

Common property loader

JSON data

Allure properties

Config properties

Data properties

Dependency management

Sample API URLs: API calls | Spotify for Developers
Authorization : Authorization Code Flow | Spotify for Developers
List of API's : Web API Reference | Spotify for Developers

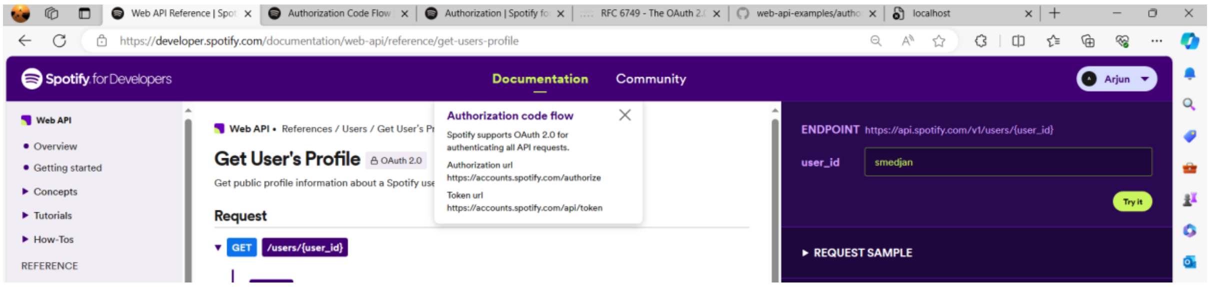
Steps to *Spotify* Register:

Step 1 : Sign Up https://www.spotify.com/
Step 2: Log in Home | Spotify for Developers and go to spotify-for-developers for creating an APP. Also, verify your registered email here.
Step 3: Make a note of Client ID and Client Secret from (Dashboard -> ApiTestSpotify -> Settings -> Basic -> Information)

Base URI for API : https://api.spotify.com/v1

Authorization Endpoint: https://accounts.spotify.com/authorize

Scope choosen from the guide (Scopes | Spotify for Developers) is:
playlist-read-private playlist-read-collaborative playlist-modify-private playlist-modify-public



Get User's Profile OAuth 2.0

Get public profile information about a Spotify use

Request

GET /users/{user_id}

Authorization code flow

Spotify supports OAuth 2.0 for authenticating all API requests.

Authorization url
https://accounts.spotify.com/authorize

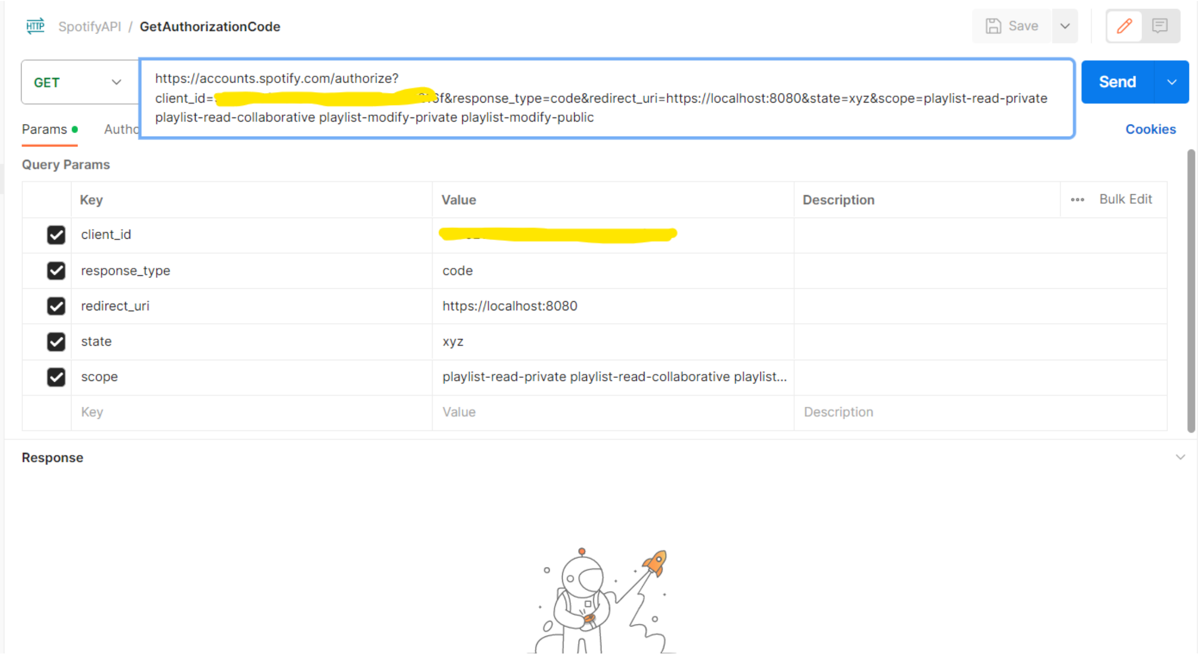
Token url
https://accounts.spotify.com/api/token

ENDPOINT https://api.spotify.com/v1/users/{user_id}

user_id: smedjan

Generating the Authorization Code:

Frame the request in Postman and hit it in browser for getting the code. Response can be seen in the URL.



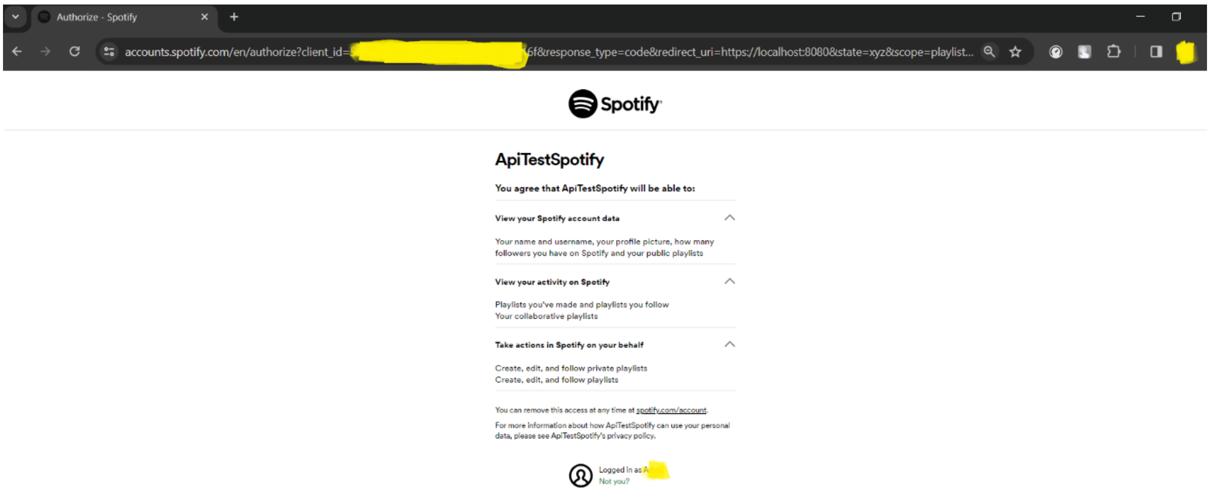
```
https://accounts.spotify.com/authorize?client_id=[redacted]&response_type=code&redirect_uri=https://localhost:8080&state=xyz&scope=playlist-read-private playlist-read-collaborative playlist-modify-private playlist-modify-public
```

Query Params

	Key	Value	Description
☑	client_id	[redacted]	
☑	response_type	code	
☑	redirect_uri	https://localhost:8080	
☑	state	xyz	
☑	scope	playlist-read-private playlist-read-collaborative playlist...	
	Key	Value	Description

Response

Agree the concern



accounts.spotify.com/en/authorize?client_id=[redacted]&response_type=code&redirect_uri=https://localhost:8080&state=xyz&scope=playlist...

ApiTestSpotify

You agree that ApiTestSpotify will be able to:

View your Spotify account data

Your name and username, your profile picture, how many followers you have on Spotify and your public playlists

View your activity on Spotify

Playlists you've made and playlists you follow
Your collaborative playlists

Take actions in Spotify on your behalf

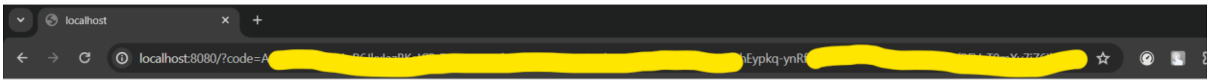
Create, edit, and follow private playlists
Create, edit, and follow playlists

You can remove this access at any time at spotify.com/account.

For more information about how ApiTestSpotify can use your personal data, please see ApiTestSpotify's privacy policy.

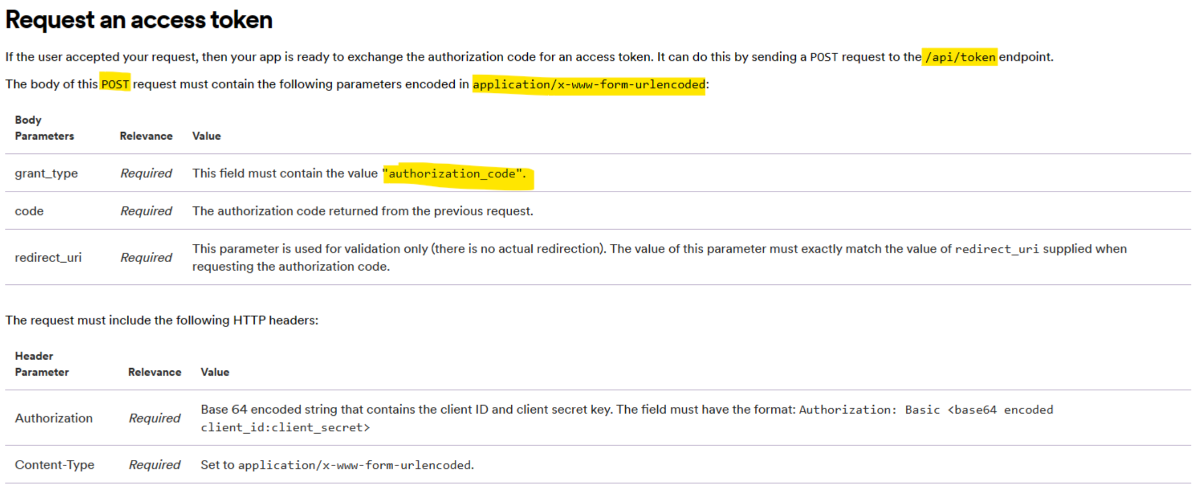
Logged in as [redacted]
Not you?

URL will have the code which is nothing authorization code and make a note of it. Authorization code is short lived and will expire quickly in general.



localhost:8080/?code=A[redacted]hEypkq-ynR[redacted]

Create Access & Refresh Token via POST request :



Request an access token

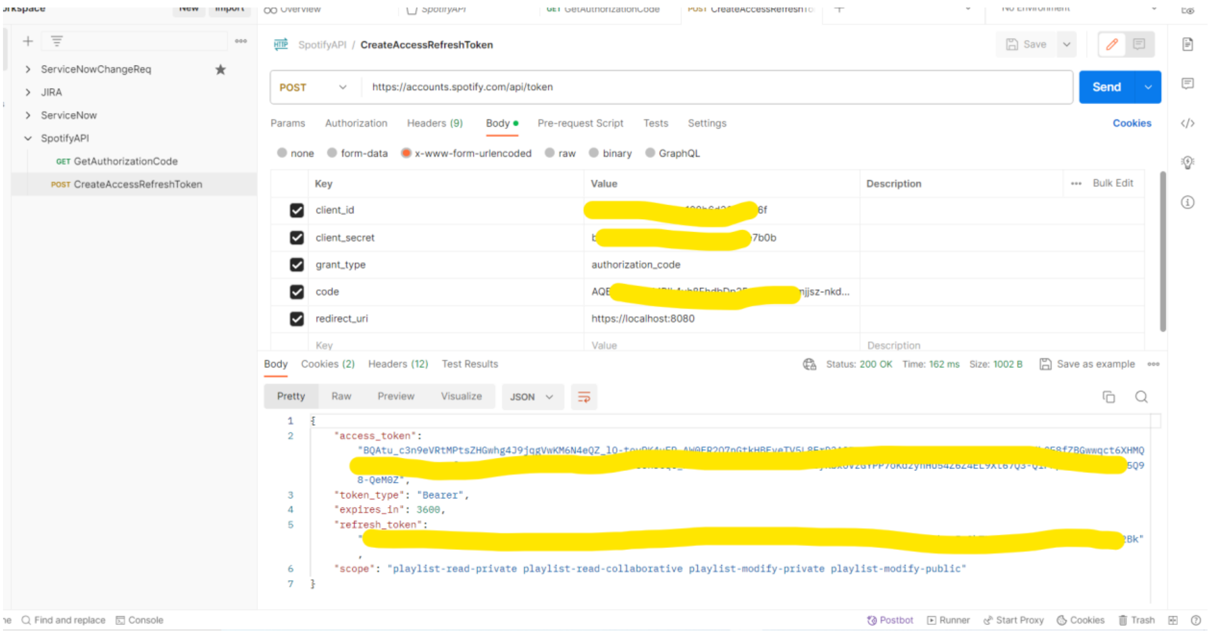
If the user accepted your request, then your app is ready to exchange the authorization code for an access token. It can do this by sending a POST request to the `/api/token` endpoint.

The body of this POST request must contain the following parameters encoded in `application/x-www-form-urlencoded`:

Body Parameters	Relevance	Value
grant_type	*Required*	This field must contain the value `"authorization_code"`.
code	*Required*	The authorization code returned from the previous request.
redirect_uri	*Required*	This parameter is used for validation only (there is no actual redirection). The value of this parameter must exactly match the value of `redirect_uri` supplied when requesting the authorization code.

The request must include the following HTTP headers:

Header Parameter	Relevance	Value
Authorization	*Required*	Base 64 encoded string that contains the client ID and client secret key. The field must have the format: `Authorization: Basic <base64 encoded client_id:client_secret>`
Content-Type	*Required*	Set to `application/x-www-form-urlencoded`.



SpotifyAPI / CreateAccessRefreshToken

POST https://accounts.spotify.com/api/token

○ none ○ form-data ● x-www-form-urlencoded ○ raw ○ binary ○ GraphQL

	Key	Value	Description
☑	client_id	[redacted]6f	
☑	client_secret	b[redacted]7b0b	
☑	grant_type	authorization_code	
☑	code	AQE[redacted]njjsz-nkd...	
☑	redirect_uri	https://localhost:8080	
	Key	Value	Description

Status: 200 OK Time: 162 ms Size: 1002 B

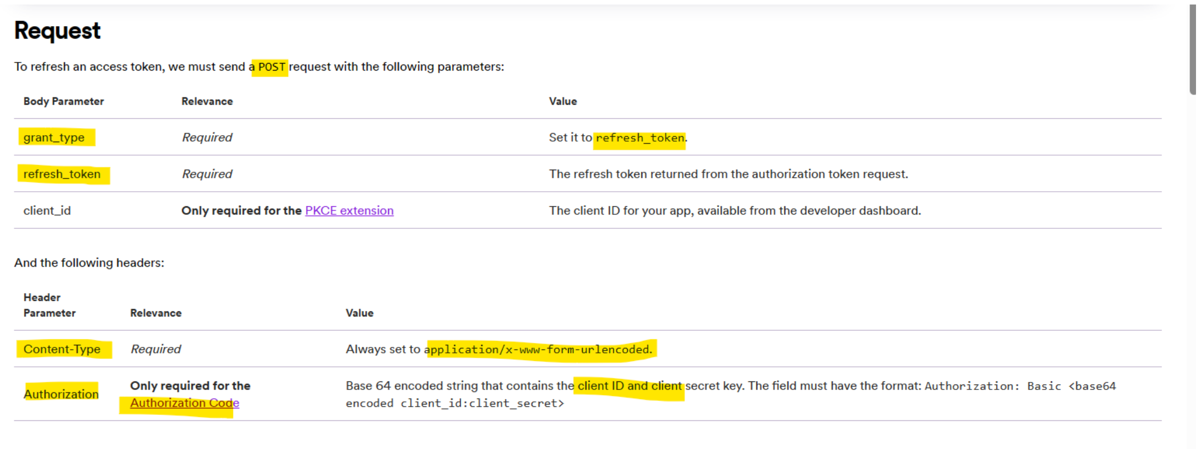
```
{
    "access_token":
      "BQAtu_c3n9eVRtMPtsZHGwhg4J9jqgVwKM6N4eQZ_lO-[redacted]5Q9
      8-QeM0Z",
    "token_type": "Bearer",
    "expires_in": 3600,
    "refresh_token":
      "[redacted]Bk"
      ,
    "scope": "playlist-read-private playlist-read-collaborative playlist-modify-private playlist-modify-public"
}
```

Access Token Regeneration:

We can regenerate the access token if expired with the help of refresh token available with us. Even if refresh token has expired, then we have follow from the start of authorization code.

A refresh token is a security credential that allows client applications to obtain new access tokens without requiring users to reauthorize the application.

Access tokens are intentionally configured to have a limited lifespan (1 hour), at the end of which, new tokens can be obtained by providing the original refresh token acquired during the authorization token request response:



Request

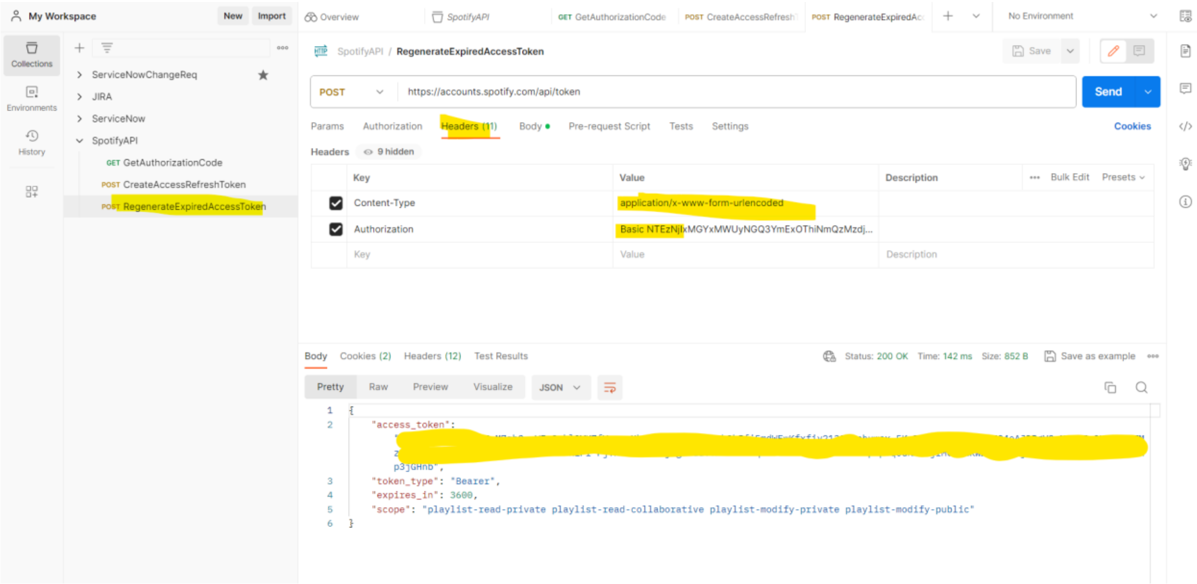
To refresh an access token, we must send a POST request with the following parameters:

Body Parameter	Relevance	Value
grant_type	*Required*	Set it to `refresh_token`.
refresh_token	*Required*	The refresh token returned from the authorization token request.
client_id	**Only required for the** PKCE extension	The client ID for your app, available from the developer dashboard.

And the following headers:

Header Parameter	Relevance	Value
Content-Type	*Required*	Always set to `application/x-www-form-urlencoded`.
Authorization	**Only required for the** Authorization Code	Base 64 encoded string that contains the client ID and client secret key. The field must have the format: `Authorization: Basic <base64 encoded client_id:client_secret>`

Request Header:



SpotifyAPI / RegenerateExpiredAccessToken

POST https://accounts.spotify.com/api/token

Headers

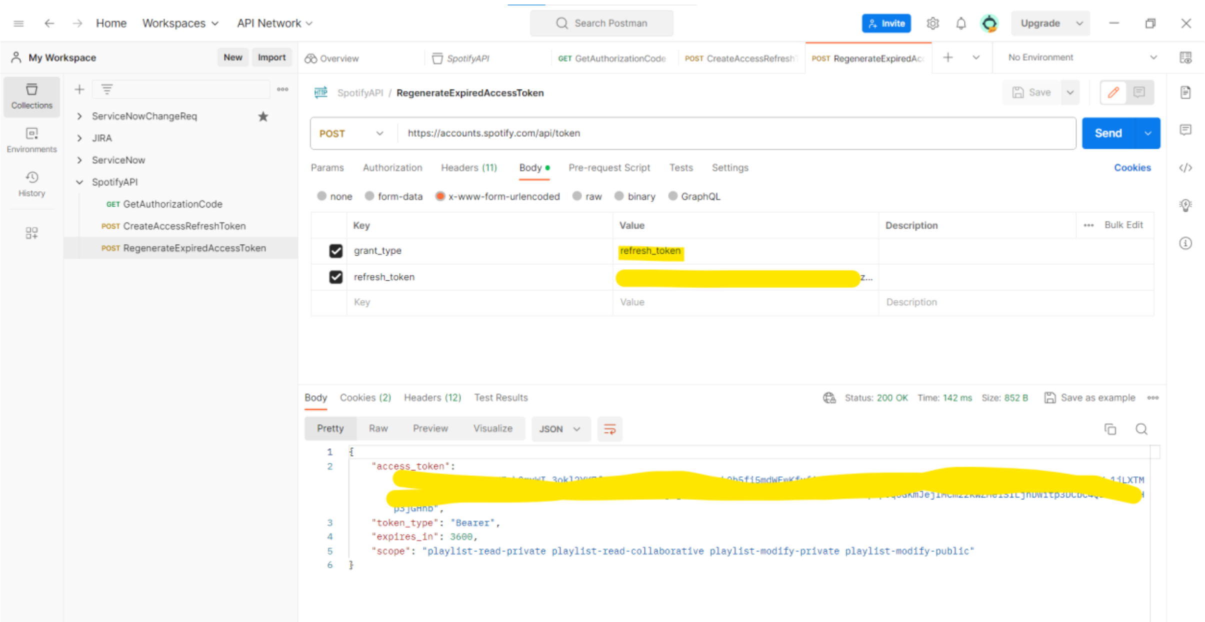
	Key	Value	Description
☑	Content-Type	application/x-www-form-urlencoded	
☑	Authorization	Basic NTEzNjIxMGYxMWUyNGQ3YmExOThiNmQzMzdj...	
	Key	Value	Description

Status: 200 OK Time: 142 ms Size: 852 B

```
{
    "access_token":
        [redacted]
        p3jGHnb",
    "token_type": "Bearer",
    "expires_in": 3600,
    "scope": "playlist-read-private playlist-read-collaborative playlist-modify-private playlist-modify-public"
}
```

Request Body:

Response is received with the newly generated access token.



SpotifyAPI / RegenerateExpiredAccessToken

POST https://accounts.spotify.com/api/token Send

○ none ○ form-data ● x-www-form-urlencoded ○ raw ○ binary ○ GraphQL

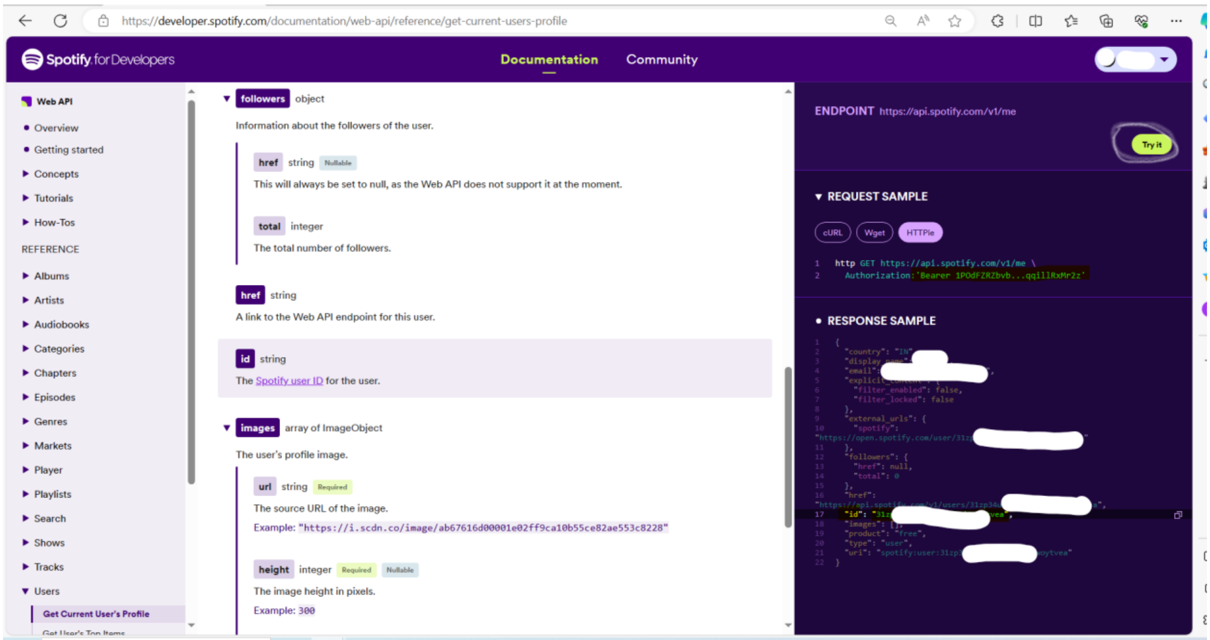
	Key	Value	Description
☑	grant_type	refresh_token	
☑	refresh_token	[redacted]z...	
	Key	Value	Description

Body Status: 200 OK Time: 142 ms Size: 852 B

```
{
    "access_token":
        [redacted]
        p3jGHnb",
    "token_type": "Bearer",
    "expires_in": 3600,
    "scope": "playlist-read-private playlist-read-collaborative playlist-modify-private playlist-modify-public"
}
```

Get User ID for access Playlist API as user id is in Path Parameter variable and differs.

Spotify provides web console here to try APIs and get the responses.



followers object

Information about the followers of the user.

href string Nullable

This will always be set to null, as the Web API does not support it at the moment.

total integer

The total number of followers.

href string

A link to the Web API endpoint for this user.

id string

The Spotify user ID for the user.

images array of ImageObject

The user's profile image.

url string Required

The source URL of the image.

Example: "https://i.scdn.co/image/ab67616d00001e02ff9ca10b55ce82ae553c8228"

height integer Required Nullable

The image height in pixels.

Example: 300

ENDPOINT https://api.spotify.com/v1/me

REQUEST SAMPLE

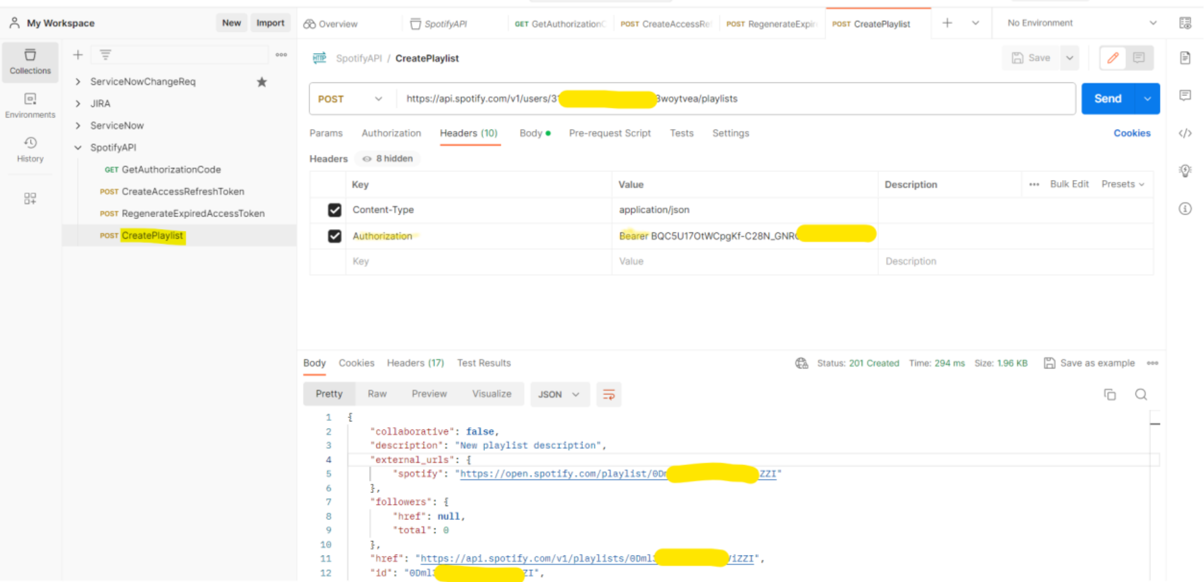
```
http GET https://api.spotify.com/v1/me \
  Authorization:'Bearer 1POdFZRZbvb...qqillRxMr2z'
```

RESPONSE SAMPLE

```
{
  "country": "IN",
  "display [redacted]
  "email": [redacted],
  "explicit[redacted] {
    "filter_enabled": false,
    "filter_locked": false
  },
  "external_urls": {
    "spotify":
"https://open.spotify.com/user/31zp[redacted]"
  },
  "followers": {
    "href": null,
    "total": 0
  },
  "href":
"https://api.spotify.com/v1/users/31zp34u[redacted]",
  "id": "31z[redacted]vea",
  "images": [],
  "product": "free",
  "type": "user",
  "uri": "spotify:user:31zp[redacted]oytvea"
}
```

Create a PlayList:

Request Header:



POST https://api.spotify.com/v1/users/3[redacted]3woytvea/playlists

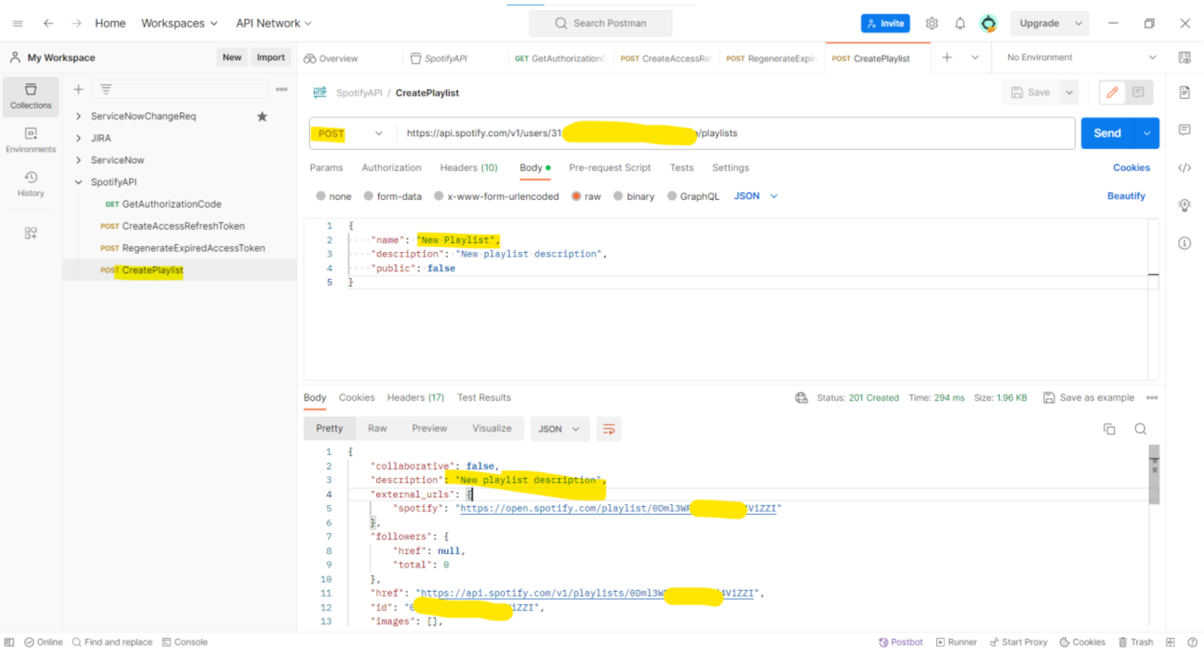
Headers (10)

Key	Value	Description
☑ Content-Type	application/json	
☑ Authorization	Bearer BQC5U17OtWCpgKf-C28N_GNR[redacted]	

Status: 201 Created Time: 294 ms Size: 1.96 KB

```
{
    "collaborative": false,
    "description": "New playlist description",
    "external_urls": {
        "spotify": "https://open.spotify.com/playlist/0Dr[redacted]ZZI"
    },
    "followers": {
        "href": null,
        "total": 0
    },
    "href": "https://api.spotify.com/v1/playlists/0Dml[redacted]/iZZI",
    "id": "0Dml[redacted]ZI",
```

Request Body:



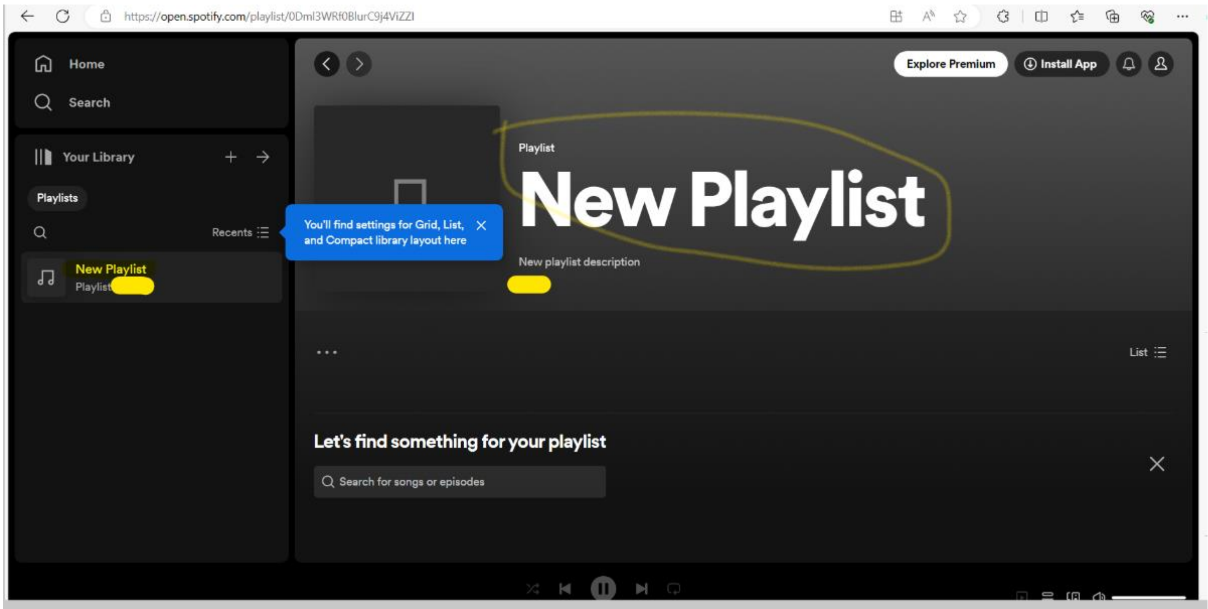
POST https://api.spotify.com/v1/users/31[redacted]/playlists

```
{
    "name": "New Playlist",
    "description": "New playlist description",
    "public": false
}
```

Status: 201 Created Time: 294 ms Size: 1.96 KB

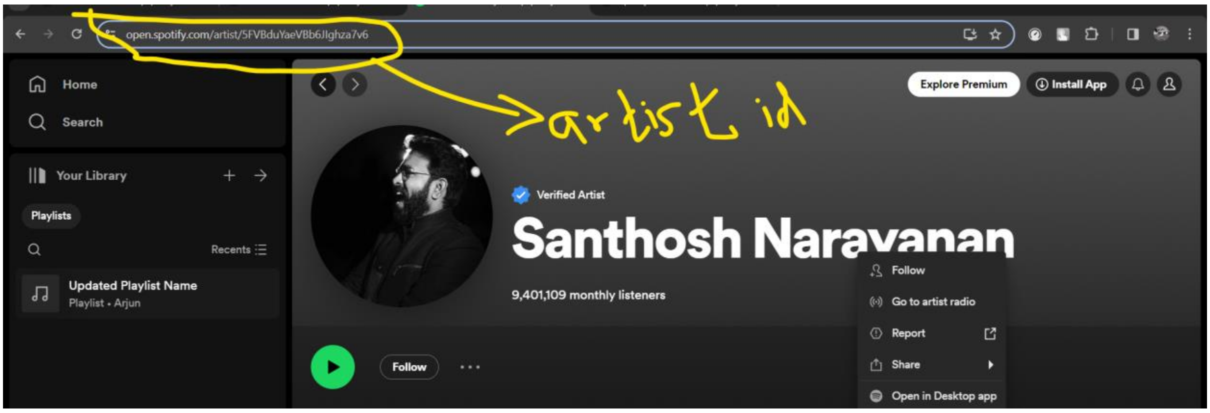
```
{
    "collaborative": false,
    "description": "New playlist description",
    "external_urls": {
        "spotify": "https://open.spotify.com/playlist/0Dml3WR[redacted]VlZZI"
    },
    "followers": {
        "href": null,
        "total": 0
    },
    "href": "https://api.spotify.com/v1/playlists/0Dml3W[redacted]4VlZZI",
    "id": "0[redacted]iZZI",
    "images": [],
```

New Playlist in the Spotify APP Created :



https://open.spotify.com/playlist/0DmI3WRf0BlurC9j4ViZZI

You'll find settings for Grid, List, and Compact library layout here



open.spotify.com/artist/5FVBduYaeVBb6JIghza7v6