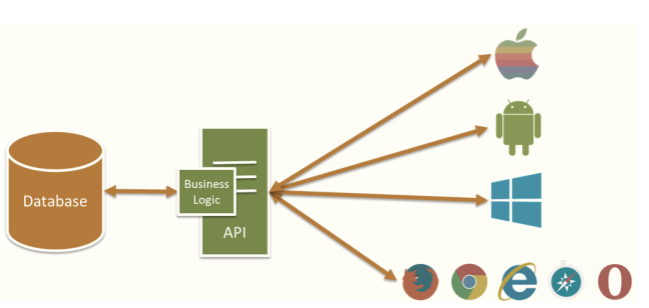
<https://jsoneditoronline.org/> json parsor

<https://www.jsonschema.net/> json schema generator

<https://www.freeformatter.com/json-escape.html> json to String converter

# API testing



In the modern development world, many web applications are designed based on three-tier architecture model. These are:

1) Presentation Tier – User Interface (UI)

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

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

--

API testing is a type of software testing that involves testing application programming interfaces (APIs) directly and as part of integration testing to determine if they meet expectations for functionality, reliability, performance, and security.[1] Since APIs lack a GUI, API testing is performed at the message layer.[2] API testing is now considered critical for automating testing because APIs now serve as the primary interface to application logic and because GUI tests are difficult to maintain with the short release cycles and frequent changes commonly used with Agile software development and DevOps.

API testing involves testing APIs directly (in isolation) and as part of the end-to-end transactions exercised during integration testing. Beyond RESTful APIs, these transactions include multiple types of endpoints such as web services, ESBs, databases, mainframes, web UIs, and ERPs. API testing is performed on APIs that the development team produces as well as APIs that the team consumes within their application (including third-party APIs)

API testing normally includes only the white-box testing approach.

These public APIs allow web communities to create content in one location and share it in multiple locations.User information can be shared from web communities to outside applications, delivering new functionality to the web community.On today's World Wide Web everything is interconnected, and every user is both a consumer and a creator. Almost every website has instant Twitter feeds, targeted Google AdWords, and a multitude of other widgets from third-party providers. Any website can be an API provider, an API consumer, and an end user, all in one.

|  |  |
| --- | --- |
| **Web Service** | **Web API** |
| Defined by W3C, all communication & data exchange is based on XML | Web API communication & data exchange could be XML, JSON or plain data |
| It has defined standards – WSDL | No defined standard |
| You cannot compress the data but you can compress the HTML request | You can compress the data |
| Example: SOAP | Example: REST |

# HTTP Protocol between Client and Server

we would read a newspaper or listen to the radio to get the weather updates. Newspaper and Radio use your local language and you will be able to understand what is written in the paper or spoken on the Radio. However, for the Clients and Servers on the Web we have to come up with two things

A medium for communication, specifically a protocol for two systems to interact. Also called HTTP communication protocol

A protocol to ask for the required details from the server. This could be in any form of formatted data. Most commonly used formats are XML and Json.

Server responds by sending a Response in any form of formatted data, here also it could be XML or JSON.

**To Summarize**: A Client and a Server establishes a connection using HTTP protocol. Once the connection is established, Client sends across the request to the Server in the form of XML or JSON which both entities (Client and Server) understand. After understanding the request Server responds with appropriate data by sending back a Response.

# What is HTTP Request?

HTTP Request is a packet of Information that one computer sends to another computer to communicate something. To its core, HTTP Request is a packet of binary data sent by the Client to server. An HTTP Request contains following parts

* Request Line
* Headers, 0 or more Headers in the request
* An optional Body of the Request



## Endpoint

**The endpoint** (or route) is the url you request for. The **root-endpoint** is the starting point of the API you’re requesting from. The **path** determines the resource you’re requesting for. Example <https://www.smashingmagazine.com> is root endpoint and tag/javascript/ is path

<https://www.smashingmagazine.com/tag/javascript/>

To understand what paths are available to you, you need to look through the API documentation.

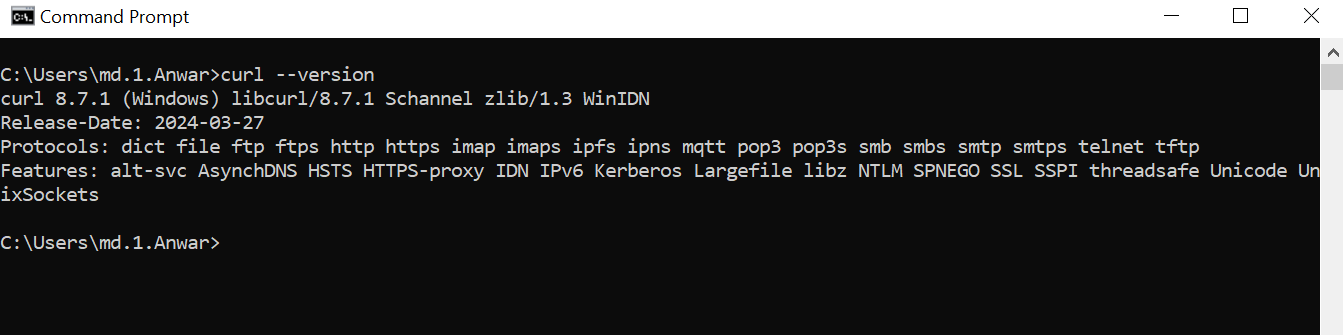
## Query parameters

Technically, query parameters are not part of the REST architecture, but you’ll see lots of APIs use them. So, to help you completely understand how to read and use API’s we’re also going to talk about them. Query parameters give you the option to modify your request with key-value pairs. They always begin with a question mark (?). Each parameter pair is then separated with an ampersand (&), like this:

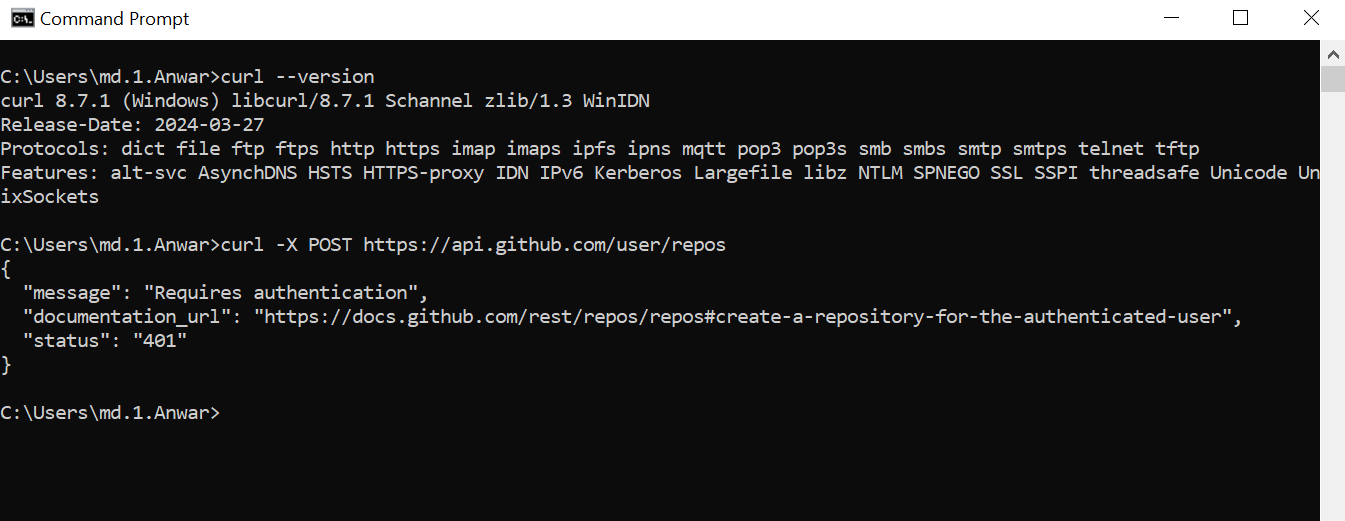
?query1=value1&query2=value2

### Testing Endpoints With curl [#](https://www.smashingmagazine.com/2018/01/understanding-using-rest-api/#testing-endpoints-with-curl)

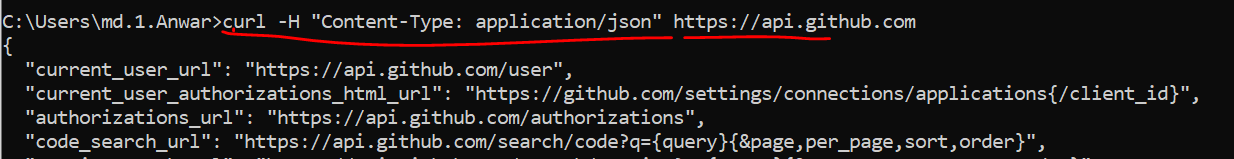
Before you continue, you’ll want to make sure you have cURL installed on your machine. Open up your Terminal and type curl --version. This command checks the version of cURL you have installed on your system.



You can set the request method in cURL by writing -X or --request, followed by the request method. This command below tries to create a repository via cURL



You can send HTTP **headers** with curl through the -H or --header option. To send the above header to Github’s API, you use this command:



To view headers you’ve sent, you can use the -v or --verbose option as you send the request, like this

## HTTP Request Header

which allows the client to pass along information about the request.  the client sends the type of content that it is able to receive from the server. This is called the Accept field, and it ensures that the server does not send data that cannot be understood or processed by the client.

## HTTP Methods

The following HTTP methods are most commonly used in a REST based architecture.

* **GET** – (**READ**) Provides a read only access to a resource. used parameter to send to server;
* **PUT** – (**Modify**) Used to update existing resource. e,g update comment. **body** (mandatory).
* **DELETE** – (**Delete**) Used to remove a resource.
* **POST** – (**Create**) Used to **update an existing resource** or **create a new resource**. Can’t use parameter, data should be sent through **body** (mandatory) as json/XML.

e.g. issue create—new resource

add comment –update existing request

* **OPTIONS** − Used to get the supported operations on a resource.

<http://jsonplaceholder.typicode.com/posts>

In general API's are like below, they have server name, paths.., etc

http://<server name>/v1/export/Publisher/Standard\_Publisher\_Report?format=csv

# What is HTTP Response?

HTTP Response is the packet of information sent by Server to the Client in response to an earlier Request made by Client. HTTP Response contains the information requested by the Client. For example, the request to Weather Web Service made in the HTTP Request tutorial will contain the weather details of the location.

Just like HTTP Request, HTTP Response also has the same structure:

* Status Line
* Headers, 0 or more Headers in the request
* An optional Body of the Request

## HTTP Response Header

### Content Types

In cases where the server is sending a data payload to the client, the server must include a content-type in the header of the response. This content-type header field alerts the client to the type of data it is sending in the response body. These content types are MIME Types, just as they are in the accept field of the request header. The content-type that the server sends back in the response should be one of the options that the client specified in the accept field of the request.

For example, when a client is accessing a resource with id 23 in an articles resource with this GET Request:

GET /articles/23 HTTP/1.1

Accept: text/html, application/xhtml

The server might send back the content with the response header:

HTTP/1.1 200 (OK)

Content-Type: text/html

## Request Body(Data)

The data (sometimes called “body” or “message”) contains information you want to be sent to the server. This option is only used with POST, PUT, PATCH or DELETE requests.

To send data through cURL, you can use the -d or --data option:

curl -X POST <URL> -d property1=value1

curl -X POST <URL> -d property1=value1 -d property2=value2

If it makes sense, you can break your request into multiple lines \ to make it easier to read:

curl -X POST <URL> \

-d property1=value1 \

-d property2=value2

If you wish to send JSON data, you’ll need to set the Content-Type to application/json, and you’ll need to format your data as a JSON object, like this:

curl -X POST https://requestb.in/1ix963n1 \

-H "Content-Type: application/json" \

-d '{

"property1":"value1",

"property2":"value2"

}'

### Response Codes

Responses from the server contain status codes to alert the client to information about the success of the operation.

| Status code | Meaning |
| --- | --- |
| 200 (OK) | This is the standard response for successful HTTP requests. |
| 201 (CREATED) | This is the standard response for an HTTP request that resulted in an item being successfully created. |
| 204 (NO CONTENT) | This is the standard response for successful HTTP requests, where nothing is being returned in the response body. **for empty response** |
| 400 (BAD REQUEST) | The request cannot be processed because of bad request **syntax**, excessive **size**, or another **client error**. |
| 403 (FORBIDDEN) | The client does not have permission to access this resource. |
| 404 (NOT FOUND) | The resource could not be found at this time. It is possible it was deleted, or does not exist yet. |
| 500 (INTERNAL SERVER ERROR) | The generic answer for an unexpected failure if there is no more specific information available. |
| 503 | Service unavailable |

# REST—REpresentationalState Transfer

In REST Architecture everything is a resource. RESTful web services are light weight, highly scalable and maintainable and are very commonly used to create APIs for web-based applications.

In REST architecture, a REST Server simply provides access to resources and the REST client accesses and presents the resources. Here each resource is identified by URIs/ Global IDs. REST uses various representations to represent a resource like Text, JSON and XML. JSON is now the most popular format being used in Web Services.

A RESTful web service usually defines a URI (Uniform Resource Identifier), which is a service that provides resource representation such as JSON and a set of HTTP Methods.

## REST Constrains

### 1) Client – Server Architecture style

According to this constraint an application should be modeled like a Client – Server. To relate to it, application should have the UI separate from the Data. There should be a distinct components handling Front end (UI) and the Back end (Database).

### 2) Stateless

### A REST implementation should be stateless. It means that the two parties don’t need to store any information about each other and every request and response is independent from all others.

### 3) Cache

According to this constraint Responses from the server should contain relevant information to tell whether the Response can be cached by the client of not. Usually this is done via a Header entry in the Response.

This constraint improves the client efficiency, for cacheable responses Client need not make requests to the server. Client can simply look into its cache for the Response. This save network bandwidth and Client processing power.

### 4) Uniform Interface

The uniform interface simplifies and decouples the architecture, which enables each part to evolve independently. The four guiding principles of this interface are:

* Identification of resources
* Manipulation of resources through these representations
* Self-descriptive messages
* Hypermedia as the engine of application state (A.K.A. HATEOAS)

### 5) Layered System

According to this constraint the system implementation should be layered. Each layer abstracting out certain functionality of the overall system. A layer should not know about the existence of other layers apart from the layers that it directly interacts with.

Layered software design is a common practice outside the realm of REST too. This constraint decreases the overall complexity of the different components within the system. As we have to deal with relatively smaller and logical cohesive pieces of functionality it becomes easier to implement layers.

# What are Rest Architectural Elements

### Resource

Information stored on a Server, which can be requested by a client. It could be Weather info or may be employee details

### Resource Identifier

Now that we have a resource defined, we need to uniquely identify the resource. That is actually the complete URL

### Representation

A resource is the actual data. Now this data can be represented as an XML,Json, HTML or may be simple text. That is what is called a Representation.

### Representation Metadata

In order for the Clients to specify and process a resources given in a particular Representation (XML or HTML etc) some extra data (Metadata) needs to be passed in the request.

### What is Endpoint?

This is address of API to be called. It consist of three categories

1. Base URI --website URL/Host Name like www.google.com
2. Resources
3. parameter

Endpoint=BaseURL/Resource?parameter ---https://google.co.in/maps?hl=en

# Swagger

## Introduction

Most modern day web applications are built with a front end programming language that communicates with an application programming interface(API). Developers will typically create some sort of api documentation which helps other developers understand and use their api. Swagger is a frame work that makes creating these documents easier.

When testing a web application one of the first things I do is figure out if its communicating with an api or not. With the rise of microservices you might notice a lot of websites sending ajax requests to get data from a backend api server. If you notice the target you are interacting with is an api or is communicating with an api you should immediately look for exposed api documentation. Some common paths to find swagger api documentation include:

/api

/swagger/index.html

/swagger/v1/swagger.json/

swagger-ui.html

/swagger-resources

Swagger is the standard way of documenting the Standard APIs. Swagger is helpful when deploying APIs in azure. Swagger is primarily used for documenting API. For the other developers to be able to use our API, the API must be properly documented; otherwise, how would they know that what are the endpoints exposed by the api and what are the operations supported on those endpoints? What parameters should they pass, and what will they get back? What authentication methods to use?. To answer these questions, it is very important to document the APIs; if you want APIs to be consumed and properly used.

### ****API Definition File****

API Definition File is a file that contains all the things that you can do with a file. This file contains the following things:

* Server location
* How security is handled, i.e., authorization.
* All the available resources in that API.
* All the different data that you can send in a request.
* What data is returned
* What HTTP status codes can be returned

# Security

Here, Security means authentication and authorization. Authentication means to validate the user through their username and password. The authorization means allowing the user to access the data.

Since POST, PUT, PATCH and DELETE requests alter the database, developers almost always put them behind an authentication wall. In some cases, a GET request also requires authentication (like when you access your bank account to check your current balance, for example).

* **None:** Here, **None** means that no security is set to access the API.
* **Basic Auth:** It means that the username and password are set for each request.
* **API Key:** The key is set to access the API.(**secret token**)
* **OATH:** It is an authorization scheme.

The secret token method includes [**oAuth**](https://oauth.net/), which lets you to authenticate yourself with social media networks like Github, Google, Twitter, Facebook, etc.

# Authentication and Authorization in REST WebServices

Majority of the time you will be hitting REST API’s which are secured. By secure we mean that the API’s which require you to provide identification. Identification can be provided in the form of

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

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

In a very basic Authentication flow using Username and Password, we will do the same thing in REST API call as well. but how do we send the Username and Password in the REST request ?

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

**Authorization** is the process of giving access to someone. If you are Authorized then you have access to that resource. Now to Authorize you need to present credentials

### Example

@Test

public void test\_APIWithBasicAuthentication\_ShouldBeGivenAccess() {

given().

auth().

preemptive().

basic("username", "password").

when().

get("http://path.to/basic/secured/api").

then().

assertThat().

statusCode(200);

}

# What is JSON

JSON: JavaScript Object Notation.JSON is a syntax for storing and exchanging data.JSON is text, written with JavaScript object notation.

JSON is a lightweight data-interchange formatJSON is "self-describing" and easy to understandJSON is language independent

## Exchanging Data

When exchanging data between a browser and a server, the data can only be text.

JSON is text, and we can convert any JavaScript object into JSON, and send JSON to the server.

We can also convert any JSON received from the server into JavaScript objects.

This way we can work with the data as JavaScript objects, with no complicated parsing and translations.

## Sending Data

If you have data stored in a JavaScript object, you can convert the object into JSON, and send it to a server:

## JSON Syntax Rules

* Data is in name/value pairs
* Data is separated by commas
* Curly braces hold objects
* Square brackets hold arrays
* Keys and values are separated by a colon.

JSON Data - A Name and a ValueJSON data is written as name/value pairs.A name/value pair consists of a field name (in double quotes), followed by a colon, followed by a value:

"name":"John"

JSON ValuesIn JSON, values must be one of the following data types:

**string** --{ "name":"John" }**number** --{ "age":30 }**object** (JSON object)--{ "employee":{ "name":"John", "age":30, "city":"New York" } }

**Nested** **object**—{ "employee":{"name":"John","cars": {"car1":"Ford","car2":"BMW" }}}

**array of element** -- { "employees":[ "John", "Anna", "Peter" ] }**array of object**--{ "employees":[ {"id":1,"name":"John"},{"id":2,"name":"Don"}]

**2D array of element**--{ "emp":[ ["emp1", "emp2"], ["mgr1","mgr2"] ]} 🡪 emp[0][1]

**boolean** ---{ "sale":true }**null** --{ "middlename":null }

In JSON, string values must be written with double quotes---{ "name":"John" }

## Value as Object in JSON

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

## Value as Array in JSON

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

## JSON Files

The file type for JSON files is ".json"

The MIME type for JSON text is "application/json"

## JSON vs XML

Both JSON and XML can be used to receive data from a web server.

The following JSON and XML examples both defines an employees object, with an array of 3 employees:

JSON---

{"employees":[{ "firstName":"John", "lastName":"Doe" },{ "firstName":"Anna", "lastName":"Smith" },{ "firstName":"Peter", "lastName":"Jones" }]}

XML-----

<employees><employee><firstName>John</firstName><lastName>Doe</lastName></employee><employee><firstName>Anna</firstName><lastName>Smith</lastName></employee><employee><firstName>Peter</firstName><lastName>Jones</lastName></employee></employees>

JSON is Like XML BecauseBoth JSON and XML are "self describing" (human readable)Both JSON and XML are hierarchical (values within values)Both JSON and XML can be parsed and used by lots of programming languagesBoth JSON and XML can be fetched with an XMLHttpRequest

JSON is Unlike XML BecauseJSON doesn't use end tagJSON is shorterJSON is quicker to read and writeJSON can use arrays

The biggest difference is:

 XML has to be parsed with an XML parser. JSON can be parsed by a standard JavaScript function.

Why JSON is Better Than XMLXML is much more difficult to parse than JSON.JSON is parsed into a ready-to-use JavaScript object.

# Framework user to develop web services

Our order management system was created using Test Driven Development (TDD), where tests were created first and each design decision and implemented component was focused on passing the created test cases. This not only resulted in a simple set of classes, but a more easily distinguishable set of components. For example, persistence logic and domain logic are not intertwined. Apart from the process used to create the service, there are also numerous tools used to build, test, and deploy the system, including:

[Spring Model-View-Controller (MVC)](https://spring.io/guides/gs/serving-web-content/): the core framework of our web service; this framework provides the necessary annotations and structure required to create our REST endpoints and serve these endpoints over HTTP.

[Spring Boot](https://projects.spring.io/spring-boot/): a convention-over-configuration framework that removes a majority of the boilerplate Spring code and configuration; this framework allows us to develop and launch of web service with a fraction of the hassle of a standard Spring web service.

[Apache Maven](https://maven.apache.org/): a build and dependency management tool that is used to build, execute tests, and package our web service into a Java Archive (JAR) file that will be executed to run our RESTful web service.

[JUnit](http://junit.org/junit4/): an automated unit testing framework that will be used for unit and integration tests, as well as to automate the execution of our acceptance tests.

[Cucumber](https://cucumber.io/): an automated acceptance testing framework that allows us to create text-based acceptance criteria and exercise our web service to ensure that all top-level functionality is correct during the development of the service.

[Java 8](http://www.oracle.com/technetwork/java/javase/overview/java8-2100321.html): at the time of writing, the latest version of Java; we will utilize the streams API to reduce the code needed to filter through the domain objects in our system.

[GitHub](https://github.com/): a free hosting service for Git-based projects; our web service will be hosted here and the commit history for the web service can be viewed [here](https://github.com/albanoj2/order-rest-backend/commits/master).

[Travis Continuous Integration (CI)](https://travis-ci.org/): a free continuous integration platform that executes the automated tests in order of web service each time a commit is pushed to our GitHub repository; the build and test history of our web service can be viewed [here](https://travis-ci.org/albanoj2/order-rest-backend/builds).

Although we are using a wide array of frameworks and tools, each has a very important task when building our web service. Before we jump into the implementation, though, we first need to devise a design for our order management system.

# Rest Assured

REST Assured can be used to test XML as well as JSON based web services. REST Assured can be integrated with JUnit and TestNG frameworks for writing test cases for our application.

REST Assured is implemented in Groovy and uses the builder pattern to create requests, set headers, parse the response and then match them with expected data.

One of the powerful features of REST assured is the support of XML Path and JSON Path syntax to check specific elements of the response data. It’s very similar to using XPath API.

## How to download rest Assured framework

<http://rest-assured.io/> 🡪 docs 🡪 downloads

<https://github.com/rest-assured/rest-assured/wiki/Downloads>



## io.restassured.RestAssured

* It creates HTTP Requests against a base URI
* It supports creating Request of different HTTP method types (GET, POST, PUT, PATH, DELETE, UPDATE, HEAD and OPTIONS)
* It makes HTTP communication with the server and passes on the Request that we created in our tests to the server.
* Retrieves the Response from the server.
* Helps validate the Response received from the server.

## RequestSpecification

Every Request in Rest-Assured library is represented by an interface called RequestSpecification. This interface allows to modify the request, like adding headers or adding authentication details. The word specification at the end is used to signify that how the request should look like, when sent to the server.

## Make a request to the server

Issuing request takes two arguments, first argument as HTTP Method Type and second as String (“/Hyderabad”). This step actually sends the request to the remote server and gets a response back. This is why the return type of the request is specified as Response.

## io.restassured.response.Response

In Rest-Assured io.restassured.response.Response interface represents a Response returned from a server. This Response object will contain all the data sent by the server. Different method can be called on the Response object to get different parts of the Response. For e.g. call to get Headers, Status code and the body of the Response. In the next code line we will get the body of the Response.

**getStatusCode()** - return status code

**getStatusLine()** - return status line like "HTTP/1.1 200 OK"

### Example

import static io.restassured.RestAssured.\*;

import static org.hamcrest.Matchers.\*;

// Specify the base URL to the RESTful web service

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

// Get the RequestSpecification of the request that you want to sent

// to the server. The server is specified by the BaseURI that we have

// specified in the above step.

RequestSpecificationhttpRequest = RestAssured.given();

// Make a GET request call directly by using RequestSpecification.get() method.

// Make sure you specify the resource name.

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

// Response.asString method will directly return the content of the body

// as String.

System.out.println("Response Body is => " + response.asString());

**Example**

Nowe consider a scenario where we need to send 100’s of request to the API, in such case we will need to write get() method with the API URL every time we send the request, like below :

get("https://postman-echo.com/GET").then().statusCode(200);

But if you write below code in @BeforeTest, you will need to write only once.

RestAssured.baseURI = "http://jsonplaceholder.typicode.com";

RestAssured.basePath = "/posts/";

If you write above code,youdont need to store the URL in any variable, REST-Assured will automatically provide the URL, whenever you call any HTTP method, in this case we called post() method, without providing any URL parameter in it.

## Validate Response Header using Rest Assured

Every response that is received from a server contains zero or more headers. Headers are the part of Response that is sent by the server. Each header entry is basically a Key-Value pair. Headers are used to send across extra information by the server. This extra information is also referred to as Meta data of the Response.

One of the Headers called Content-Type which tells how to interpret the data present in the Body of the Response. If the Body contains data in the form of JSON, then the value of Content-Type header will be application/json. Similarly if the data in the body is XML the Content-Type header will be application/xml.

In the below code, **Response.header(String arg0)** method is used to get a particular header. In the argument of this method pass the exact header name.

### List of all headers name

**Response.headers()** : returns Headers

**Response.getHeaders()** : returns Headers

This collection is represented by a class called io.restassured.http.Headers. **Headers class implements the Iterable interface. Hence, for each (for( : )) loop can be used to read all the headers**

Headers allHeaders = response.headers();

// Iterate over all the Headers

for(Header header : allHeaders)

{

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

}

## Read JSON Response Body using Rest Assured

Response.body() : returns ResponseBody

Response.getBody() : returns ResponseBody

io.restassured.response.ResponseBody. This class represents the Body of a received Response. Using this class you can get and validate complete or parts of the Response Body.

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

## JsonPath Class

Response interface gives you a mechanism to extract nodes based on a given JsonPath. There is a method called **Response.JsonPath()**, which returns a **io.restassured.path.json.JsonPath** Object.

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

JsonPathjsonPathEvaluator = response.jsonPath();

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

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

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

jsonPathEvaluator.get("Humidity");

# POST Request using Rest Assured

POST request usually result in changes on the Server like addition of new data or may be update to existing data.

The data that is sent to the server in a POST request is sent in the body of HTTP request. The type of body, XML, Json or some other format is defined by the Content-Type header. If a POST request contains Json data then the Content-Type header will have a value of application/json. Similarly, for a POST request containing XML the Content-Type header value will be application/xml.

### Post Request

Endpoint http://restapi.demoqa.com/customer/register

HTTP method type: POST

Body: {

“FirstName” : “value”

“LastName” : “value”,

“UserName” : “value”,

“Password” : “value”,

“Email” : “Value”

}

Success Response: {

“SuccessCode”: “OPERATION\_SUCCESS”,

“Message”: “Operation completed successfully”

}

Failure Response: {

“FaultId”: “User already exists”,

“fault”: “FAULT\_USER\_ALREADY\_EXISTS”

}

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

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

In order to create JSON objects in the code we will add Simple JSON library in the class path. You can download Simple JSON from maven using following URL: https://mvnrepository.com/artifact/com.googlecode.json-simple/**json-simple**. Then add the downloaded Jars to class path.

### Using JSONObject from json-simple jar

// JSONObject is a class that represents a Simple JSON.

// We can add Key - Value pairs using the put method

JSONObject requestParams = new JSONObject();

requestParams.put("FirstName", "Virender");

requestParams.put("LastName", "Singh");

requestParams.put("UserName", "simpleuser001");

requestParams.put("Password", "password1");

requestParams.put("Email", "someuser@gmail.com");

JSONObject is a class that is present in org.json.simple package. This class is a programmatic representation of a JSON string. Take a look at the Request JSON above for our test web service, you will notice that there are multiple nodes in the Json. Each node can be added using the JSONObject.put(String, String) method. Once you have added all the nodes you can get the String representation of JSONObject by calling JSONObject.toJSONString() method.

// Add a header stating the Request body is a JSON

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

// Add the Json to the body of the request

request.body(requestParams.toJSONString());

// Post the request and check the response

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

### Using String

String str=”{\r\n\"name\":\"khalid Anwar\",\r\n\"Age\":30\r\n}”;

Request.body(str).post(“ /register”);

### Using DataProvider

Have multiple data using data provider, replace these values in body string as variable.

### Using Json File

body(generateString())

public String generateString() throws IOException

{

String str="E:\\Mars\_workspace\\RestAssured\\src\\apiResponse\\book.json";

Path path=Paths.get(str);

return new String(Files.readAllBytes(path));

}

### API Test Scenario

* Sending incomplete POST Data
* Sending Json data with incorrect syntax
* Sending incorrect Verb in the Request.

# How to read JSON element

Response.Jsonpath().get() – get(“”) – get(“$”)--primitive type, a List or a Map

String xx=Response.Jsonpath().get("path") --return type may be primitive type, a List or a Map

Response.Jsonpath().getString("Keyname") -- value as array

**if value is JSON Arrays**

Response.jsonPath().getString("username[0]");

List<String>jsonResponse =Response.jsonPath().getList("username");

**if value is Json Object**

Map<String, String> company = response.jsonPath().getMap("company");

System.out.println(company.get("name"));

**if value is object array**

Map<String, String> company = response.jsonPath().getMap("company[0]");

System.out.println(company.get("name"));

--Alternatively

List<Map<String, String>> companies = response.jsonPath().getList("company");

System.out.println(companies.get(0).get("name"));

# What is JsonPath?

|  |
| --- |
| $ – symbol refers to the root object or element. |
| @ – symbol refers to the current object or element. |
| . – operator is the dot-child operator, which you use to denote a child element of the current element. |
| [ ] – is the subscript operator, which you use to denote a child element of the current element (by name or index). |
| \* – operator is a wildcard, returning all objects or elements regardless of their names. |
| , – operator is the union operator, which returns the union of the children or indexes indicated. |
| : – operator is the array slice operator, so you can slice collections using the syntax [start:end:step] to return a subcollection of a collection. |
| ( ) – operator lets you pass a script expression in the underlying implementation’s script language. It’s not supported by every implementation of JSONPath, however. |
| ? ( ) – to query all items that meet a certain criteria. |

Every JSONobject is composed on an inherent hierarchy and structure. Every JSON ends up creating a tree of nodes, where each node is a **JSON Element**.

{

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

"Region": "Asia",

"Countries": [

{

"Country": "India",

"Data": {

"Capital": "New Delhi",

"mintemp": 6,

"maxtemp": 45,

"Currency": "Rupee"

}

},

{

"Country": "Nepal",

"Data": {

"Capital": "Katmandu",

"mintemp": 9,

"maxtemp": 23,

"Currency": "Nepalese rupee"

}

}

]

}

At the top most level we have a **Root** node , which is basically the node containing all of the current JSON.Rootnode operator in **JSON**is represented by a **$** sign. **$**will return all the nodes inside the JSON document.

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

In order to get children of a given node, we can use the Dot (.) operator

**$.Countries**

[

{

"Country": "India",

"Data": {

"Capital": "New Delhi",

"mintemp": 6,

"maxtemp": 45,

"Currency": "Rupee"

}

},

{

"Country": "Nepal",

"Data": {

"Capital": "Katmandu",

"mintemp": 9,

"maxtemp": 23,

"Currency": "Nepalese rupee"

}

}

]

]

### Wildcard operator in JSONPath

Wild card operator in JSONPath is \*(Star or Asterisk) symbol. This is literally means everything under that node.

if you want to display the Data nodes of all the countries--$.Countries[\*].Data

[

{

"Capital": "New Delhi",

"mintemp": 6,

"maxtemp": 45,

"Currency": "Rupee"

},

{

"Capital": "Katmandu",

"mintemp": 9,

"maxtemp": 23,

"Currency": "Nepalese rupee"

}

]

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

Here -1 stands for the last item in the Array.Array index starts from 0. Hence to refer to the second item in the array we have to use 1 as the index.

$.Countries[-1]

[

{

"Country": "Nepal",

"Data": {

"Capital": "Katmandu",

"mintemp": 9,

"maxtemp": 23,

"Currency": "Nepalese rupee"

}

}

]

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

$.Countries[0,1]

[

{

"Country": "India",

"Data": {

"Capital": "New Delhi",

"mintemp": 6,

"maxtemp": 45,

"Currency": "Rupee"

}

},

{

"Country": "Nepal",

"Data": {

"Capital": "Katmandu",

"mintemp": 9,

"maxtemp": 23,

"Currency": "Nepalese rupee"

}

}

]

### Array slice operator in JsonPath

Array slice operator is wonderful operator to extract selected items from Json. Taking the example of books, what if we want to retrieve every alternative book in the Json. To do that we will need the Array Slice operator. Syntax of Array Slice operator is [StartIndex :EndIndex : Steps].

## Schema Validation

We are not dealing with validation of data here, validation of schema means, checking whether our JSON response is in its standard structure or not, all the required fields are there or not, with the required data types.

<http://www.projectdebug.com/extracting-values-from-the-response/>

<https://github.com/rest-assured/rest-assured/wiki/Usage#example-1---json> -latest

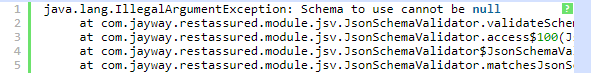
<https://techbeacon.com/how-perform-api-testing-rest-assured> --good content

### How to find classpath in eclipse

<http://b4bug.blogspot.com/2014/11/json-schema-validation-using-rest.html>

System.out.println(this.getClass().getResource("/").getPath());

### TroubleShoot



Solution: Generally this exception noticed when we have the JSON schema file which is absent on the root classpath.

# API Test Approach

## How to do API testing?

The best way to approach API testing is to build a solid testing practice from the bottom up. To this end, a great way to design a test strategy is to follow Martin Fowler’s testing pyramid. The pyramid approach suggests that you build a wide array of API tests.

1. Contract Test
2. Component Test –Individual API
3. Scenario Test –combining different APIs together
4. Performance Testing-
5. Security Testing-

### Contract Tests

An API represents a contract between 2 or more applications. The contract describes how to interact with the interface.The first and most basic type of API tests are contract tests, which test the service contract itself (Swagger, PACT, WSDL or RAML). This type of test validates that the contract is written correctly and can be consumed by a client.

* the service contract is written according to specifications
* a message request and response are semantically correct (schema validation)
* the endpoint is valid (HTTP, MQ/JMS Topic/Queue, etc)
* the service contract hasn’t changed

### Component Tests

Component tests are like unit tests for the API – you want to take the individual methods available in the API and test each one of them in isolation.

You create these tests by making a test step for each method or resource that is available in the service contract.

* The request payload is well-formed (schema validation)
* The response payload is well-formed (schema validation)
* The response status is as expected (200 OK, SQL result set returned, or even an error if that’s what you’re going for)
* The response error payloads contain the correct error messages
* The response matches the expected baseline. This can take two forms:
* Regression/diff - the response payload looks exactly the same from call to call (a top-down approach where you essentially take a snapshot of the response and verify it every time). This can also be a great catalyst to identify API change (more about that later).
* Assertion - the individual elements in the response match your expectations (this is a more surgical, bottom-up approach targeted at a specific value in the response).
* The service responds within an expected timeframe—Performance

### Scenario Tests

There are two great techniques for obtaining the sequence

1. Review the user story to identify the individual API calls that are being made.

2. Exercise the UI and capture the traffic being made to the underlying APIs.

Scenario tests allow you to understand if defects might be introduced by combining different data points together.

**Example**

I ran into a very interesting example of this while working with a customer. They had employed a series of services to call a customer’s financial profile, available accounts, credit cards, and recent transactions. Each of these API calls worked individually, but when you put them together in a sequence they started failing. The reason for this turned out to be a simple timestamp, which, when returned from one API call was in a different format than the one expected in a subsequent request. They didn’t catch this when they were doing unit testing or smoke testing because they had asserted that a timestamp was returned without specifying the format. It wasn’t until testing the overall scenario that it became clear that transferring the timestamp from one call to another caused the breakdown.

To safeguard against this, you want to create many scenario tests with different combinations of APIs to bulletproof your application against a critical breakdown.

### Performance Tests

You can simply take your scenario tests, load them up into your performance testing tool, and run them with a higher number of users.If these tests fail, you can trace the failure back to the individual user story and have a better level of understanding for what will be affected. Managers can then use this understanding to make a go or no go decision about releasing the application.

### Security Tests

Security testing is important to all stakeholders in your organization. If a security vulnerability is exposed and exploited, it can lead to significant reputation loss and financial penalties. Much like a user can accidentally use your APIs in ways you wouldn’t expect, a user can also intentionally try to exploit your APIs. A hacker can get a hold of your API, discover vulnerabilities, and take advantage of them.

A good example of this is combining different types of parameter fuzzing or SQL injection attacks with your scenario tests. That way, any changes that propagate through the application will be picked up by your security tests.

### Omni-Channel Tests

This means taking a test that is exercising one of the interfaces and coordinating it with another – executing your UI tests such as Web (Selenium) or Mobile (Appium) and interlacing them with any of your API or database tests, exchanging data points from the system through the test execution. With effective omni-channel testing, you can create stable, reusable test cases that can be easily automated.

<https://techbeacon.com/how-perform-api-testing-rest-assured>.

### Schedule It Every Day

A very good practice of API testing is to schedule the API tests every day while the testing process is live. For this, we need to have the API tests prepared well in advance. Then, we can schedule their execution every day. Needless to say, this is only possible with automated API testing tools

# How to do API testing with rest assured

#### Import at least below to start

import static io.restassured.RestAssured.\*;

import static org.hamcrest.Matchers.\*;

@Test

public void test\_NumberOfCircuitsFor2017Season\_ShouldBe20() {

given().

when().

get("http://ergast.com/api/f1/2017/circuits.json").

then().

assertThat().

body("MRData.CircuitTable.Circuits.circuitId",hasSize(20));

}

Note that the fluent API used by REST Assured supports the familiar **Given/When/Then** syntax from behavior-driven development (**BDD**), resulting in a test that is easy to read and takes care of everything (setup, execution, and verification) with just a single line of code.

#### Hamcrest matcher

The hasSize() Hamcrest matcher counts the number of circuits—that's why you needed to add Hamcrest as a static import.TheHamcrest library contains a collection of matchers that allow you to create verifications of all kinds while keeping them readable.

The verification part of the test does the following:

Captures the (JSON) response of the API call

Queries for all elements called circuitId using the Groovy GPath expression "MRData.CircuitTable.Circuits.circuitId"

Verifies (using the aforementioned Hamcrest matcher) that the resulting collection of circuitId elements has size 20

There are Hamcrest matchers for a large number of different checks, including equalTo() for equality, lessThan() and greaterThan() for comparison, hasItem() to check whether a collection contains a given element, and many more. Reference the Hamcrest library

# Parameterizing tests

Often, you'll want to repeat the same test with various sets of (input and output) parameters—a concept known as data-driven testing.

RESTful APIs support two different types of parameters:

**Query parameters**

These are appended at the end of a RESTful API endpoint and can be identified by the question mark in front of them. For example, in the endpoint http://md5.jsontest.com/?text=test, "text" is a query parameter (with value "test").

@Test

public void test\_Md5CheckSumForTest\_ShouldBe098f6bcd4621d373cade4e832627b4f6() {

String originalText = "test";

String expectedMd5CheckSum = "098f6bcd4621d373cade4e832627b4f6";

given().

param("text",originalText).

when().

get("http://md5.jsontest.com").

then().

assertThat().

body("md5",equalTo(expectedMd5CheckSum));

}

**Path parameters**

These are part of the RESTful API endpoint. For example, in the endpoint we used earlier: http://ergast.com/api/f1/2017/circuits.json, "2017" is a path parameter value. Try and replace it with "2016" and see what happens (hint: the previous test should fail, since there were 21 Formula 1 races in 2016, not 20).

@Test

public void test\_NumberOfCircuits\_ShouldBe20\_Parameterized() {

String season = "2017";

Int numberOfRaces = 20;

given().

pathParam("raceSeason",season).

when().

get("http://ergast.com/api/f1/{raceSeason}/circuits.json").

then().

assertThat().

body("MRData.CircuitTable.Circuits.circuitId",hasSize(numberOfRaces));

}

Instead of param(), path parameters are defined using the pathParam() method. In addition, you'll need to define which part of the endpoint path represents the path variable, which is done using the curly bracket notation seen in the example above.You can easily create more than a single path parameter in the same way, and even combine both path and query parameters in a single call if the API supports or requires this.

## Accessing secured APIs

Often, APIs are secured using some sort of authentication mechanism. REST Assured supports basic, digest, form, and OAuth authentication. Here's an example of how to call a RESTful API that has been secured using basic authentication (i.e., the consumer of this API needs to provide a valid username and password combination every time they call the API):

### Basic Authentication

@Test

public void test\_APIWithBasicAuthentication\_ShouldBeGivenAccess() {

given().

auth().

preemptive().

basic("username", "password").

when().

get("http://path.to/basic/secured/api").

then().

assertThat().

statusCode(200);

}

### Using authentication token

@Test

public void test\_APIWithOAuth2Authentication\_ShouldBeGivenAccess() {

given().

auth().

oauth2(YOUR\_AUTHENTICATION\_TOKEN\_GOES\_HERE).

when().

get("http://path.to/oath2/secured/api").

then().

assertThat().

statusCode(200);

}

### Token Authentication

Given().

Header(“Authorization”,”Bearer “+ tokenAsString).

When().Post().then().assertThat().statuscode(200);

## Passing parameters between tests

Often, when testing RESTful APIs, you might need to create more complex test scenarios where you'll need to capture a value from the response of one API call and reuse it in a subsequent call. This is supported by REST Assured using the extract() method. As an example, here's a test scenario that extracts the ID for the first circuit of the 2017 Formula 1 season and uses it to retrieve and verify additional information on that circuit (in this case, the circuit is located in Australia):

@Test

public void test\_ScenarioRetrieveFirstCircuitFor2017SeasonAndGetCountry\_ShouldBeAustralia() {

// First, retrieve the circuit ID for the first circuit of the 2017 season

String circuitId = given().

when().

get("http://ergast.com/api/f1/2017/circuits.json").

then().

extract().

path("MRData.CircuitTable.Circuits.circuitId[0]");

// Then, retrieve the information known for that circuit and verify it is located in Australia

given().

pathParam("circuitId",circuitId).

when().

get("http://ergast.com/api/f1/circuits/{circuitId}.json").

then().

assertThat().

body("MRData.CircuitTable.Circuits.Location[0].country",equalTo("Australia"));

}

## Reusing checks with ResponseSpecBuilder

Another way to improve the reusability and maintainability of your RESTful API tests is by reusing specific checks. For example, if you want to check that all your API responses have a status code equal to 200 and a content type equal to "application/json," specifying this for each and every test can get tiring quickly.

write below code just below of test method class and use same in @Test method.

ResponseSpecificationcheckStatusCodeAndContentType =

newResponseSpecBuilder().

expectStatusCode(200).

expectContentType(ContentType.JSON).

build();

--

@Test

public void test\_NumberOfCircuits\_ShouldBe20\_UsingResponseSpec() {

given().

when().

get("http://ergast.com/api/f1/2017/circuits.json").

then().

assertThat().

spec(checkStatusCodeAndContentType).

and().

body("MRData.CircuitTable.Circuits.circuitId",hasSize(20));

}

## Request logging

**Before When().**

given().log().all(). .. // Log all request specification details including parameters, headers and body

given().log().params(). .. // Log only the parameters of the request

given().log().body(). .. // Log only the request body

given().log().headers(). .. // Log only the request headers

given().log().cookies(). .. // Log only the request cookies

given().log().method(). .. // Log only the request method

given().log().path(). .. // Log only the request path

## Response Logging

**After then().**

get("/x").then().log().body()--response body regardless of the status code

get("/x").then().log().ifError().--response body regardless if an error occurred

get("/x").then().log().all()--log all details in the response including status line, headers and cookies

get("/x").then().log().statusLine(). .. // Only log the status line

get("/x").then().log().headers(). .. // Only log the response headers

get("/x").then().log().cookies(). .. // Only log the response cookies

**Example**

Below is output of **log().all()**

HTTP/1.1 200 OK

Content-Encoding: gzip

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

Date: Sat, 26 Jan 2019 03:49:28 GMT

ETag: W/"f3-tPjX3fYfOSeaeEnr+m0eRum0zPY"

Server: nginx

set-cookie: sails.sid=s%3A6IWcg5TmuSw2\_F9SuzCatleNMmcqD6ZI.PyOlKs%2FBXwweVIAC7bRLIBfx5B9%2FgfaF%2FgenWFuIjdU; Path=/; HttpOnly

Vary: Accept-Encoding

Content-Length: 190

Connection: keep-alive

{

"args": {

},

"headers": {

"x-forwarded-proto": "https",

"host": "postman-echo.com",

"accept": "\*/\*",

"accept-encoding": "gzip,deflate",

"user-agent": "Apache-HttpClient/4.5.3 (Java/1.8.0\_191)",

"x-forwarded-port": "443"

},

"url": "https://postman-echo.com/GET"

}

# BDD framework architecture

given(). //parameter , request header, request cookies

param("key1","value1").

param("key2","value2").

pathParam("pathvar","pathValue").

auth().

preemptive().

basic("username", "password").

auth().

oauth2(YOUR\_AUTHENTICATION\_TOKEN\_GOES\_HERE).

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

body(JSONObj.toJSONString()).

when(). // get(resource),post(resource),put(resource)

get("http://172.11.10.10/abc/{pathvar}/cde")

then(). // asertion, response header

assertthat().

body("abc.c.a[0]",hasSize(10))

and().

body("abc.c.a[0]",equalTo("bb"))

and().

statuscode(200).

contentType(ContentType.JSON)

extract(). // extraction

path("abc.c.a[0]") // partial Response

response() // full response

spec(

# PostMan

<https://reqres.in/> --- Free APIs

## Why use Postman?

**Accessibility-** One can use it anywhere after installing Postman into the device by simply logging in to the account.

**Use Collections**-Postman allows users to build collections for their API-calls. Every set can create multiple requests and subfolders. It will help to organize the test suites.

**Test development-** To test checkpoints, verification of successful HTTP response status shall be added to every API- calls.

**Automation Testing-**Tests can be performed in several repetitions or iterations by using the Collection Runner or Newman, which saves time for repeated tests.

**Creating Environments-** The design of multiple environments results in less replication of tests as one can use the same collection but for a different setting.

**Debugging-** To effectively debug the tests, the postman console helps to track what data is being retrieved.

**Collaboration-** You can import or export collections and environments to enhance the sharing of files. You may also use a direct connection to share the collections.

**Continuous integration-**It can support continuous integration.

## Authorizing Requests

Authorizing requests include authenticating the identity of the client who sends the request and verifies whether the client is allowed to access and conduct the endpoint operations. APIs use authorization details to make sure that the client requests access data safely.

No Auth

If you select this type, Postman will not send any auth data with the request.

**API key**  
This is to send the Key and Value along with the API request.

**Bearer Token**  
This auth type allows the Authorization of requests by using an access key.

**Basic Auth**  
This allows users to send username and password along with the request for API login.

OAuth  
This auth type is to access third-party API data.

**AWS Signature**  
For Amazon Web Services requests, this auth type is used.

## Postman Response

**Status Code**

A status code defines the status of the request. On entering URL, a mistake can be typed in the URL, or there may be a server-side problem. Status code is used to know about what went wrong and where you made a mistake.  E.g 200, 404, 500

### Time

Time is the duration between the sent request time and the received response time. Means, this is the duration which the response took after we sent the request and received the response.

### Size

Size is the size of the response when it is processed within memory. This response size includes the size of response, cookies, headers, and everything that has been transmitted along with the response.

### Response Body

A Response body is the body of the response, which actually contains the response content that has been sent from the server.

There are three ways to see the response in the response body:

**Pretty**:  prettier way to see the content. In this option, code will colorfully show with different keywords and have indentations in the code, which is useful for reading.

**Raw:** This is almost similar to pretty but shows the code without colors and in single lines. It is just a raw version of the code without any colorful keywords.

**Preview**

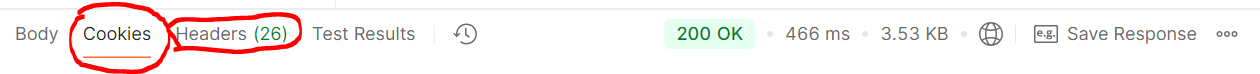
### Format Type

Each request has a defined response to it as defined by the Content-Type header. That response can be in any format. You will see that we have multiple format types:

* JSON
* XML
* HTML
* Text
* Auto

### Cookies

Cookies are the small size of files which contain the information related to server files or website pages. Postman offers you to see the cookies that have been sent from the server as a response. We cannot make any changes to the cookies since we got from the server.



### Response Header

Headers are the extra information that is transferred to the server or the client. In the postman, headers will show like key-value pairs under the headers tab. Shown in above screen.

## Request Parameters in Postman

### ****Query Parameter****

These are appended to the end of the request URL, Query parameters are appended to the end of the request URL, following '?' and listed in key-value pairs, separated by '&' Syntax:

endpoint? ?qry1=1& qry1=2

### ****Path Parameters****

These are part of the request URL, which are accessed using the placeholders preceded by ':'

/customer/:id

## POST Request

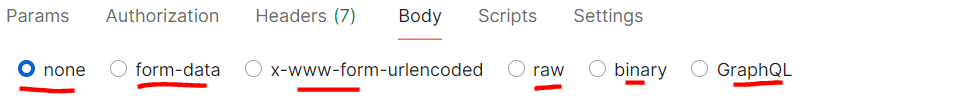
The POST request is a fundamental method, and this method is mostly used when a user wants to send some sensitive data to the server like to send a form or some confidential data.

Important points of the POST request:

* POST requests will not take the cookies.
* POST requests are not left in the history of browsers.
* We cannot bookmark the POST requests.
* There is no restriction of data length in POST requests.

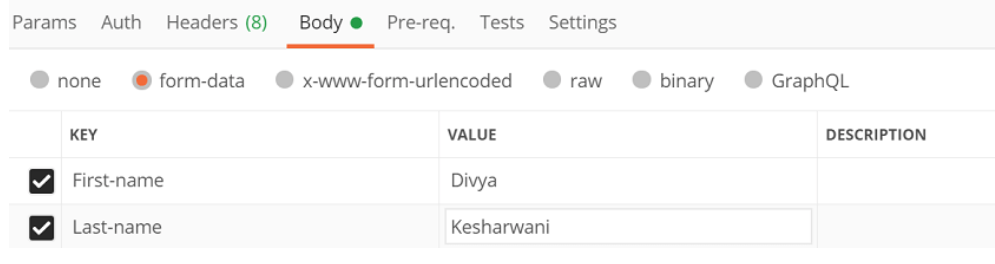
you know that we need to send the body data with requests whenever you need to add or update structured data. Select the POST request method, and go to Body option where we have different options for sending data.

### Sending Body Data



#### form-data

form-data sends the form's data. Such as the information you enter while filling out a form. To send these details, write them as key-value pairs.



#### x-www-form-urlencoded

It works similar to form-data. The only difference between both of them is that, when you sent the data via x-www-form-urlencoded, the url is encoded. Encoded indicates that the transmitted data is converted to various characters so that unauthorized persons cannot recognize the data.

#### Raw

This is a very useful option while sending the body to the POST method. Here the body data will be presented in the form of a stream of bits.

#### Binary

Binary is used to send the data in a different format. Such as a **file**, **image**, etc. To use this option, select binary and then click on Select File to browse any file from your system

## Postman Collections

### Create Collections

* Collection in Postman means a group of API requests that are already saved in the Postman and can be arranged into folders. **Any number of folders** can be created inside a collection.
* Putting similar requests into folders and collections helps the client in better organization and documentation of their requests.

### Sharing Collections

Users are not required to sign-in to their Postman account if they are sending collection as a file. Simply download the collection in the form of [JSON](https://www.javatpoint.com/json-tutorial) file and share it.

## Postman Variables

<https://www.javatpoint.com/postman>

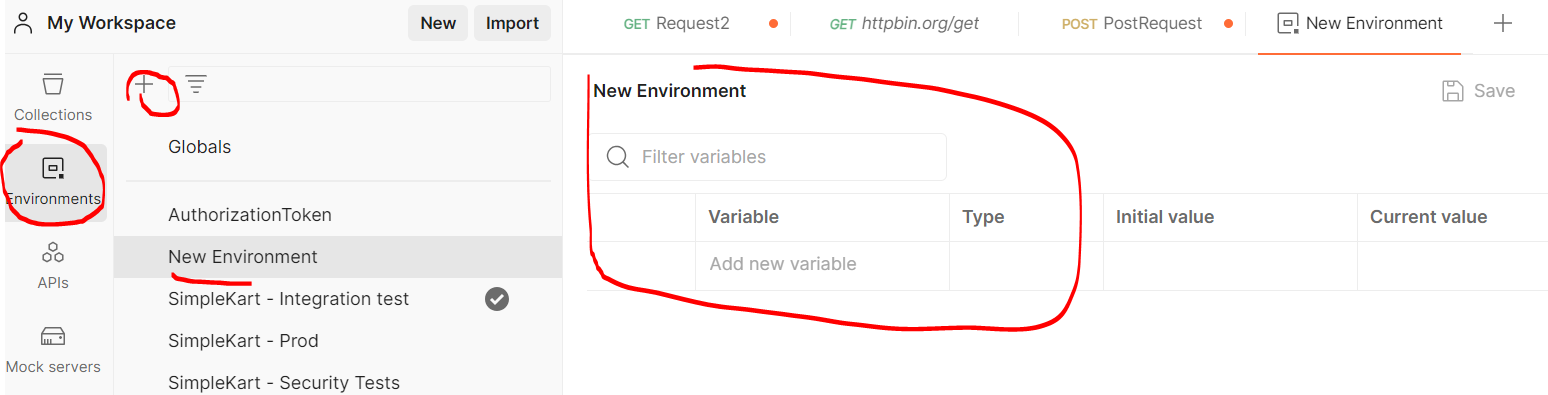
Postman variables work in the same way as that of the programming variables. You can store the values in variables and can use it throughout in requests, environments, collections, and scripts.

### Postman Environment

A collection of key-value pairs is called an environment. Each name of the variable represents its keys. And referencing the name of the variable allows you to access its value. It is a **set of variables** that differentiate among the requests.

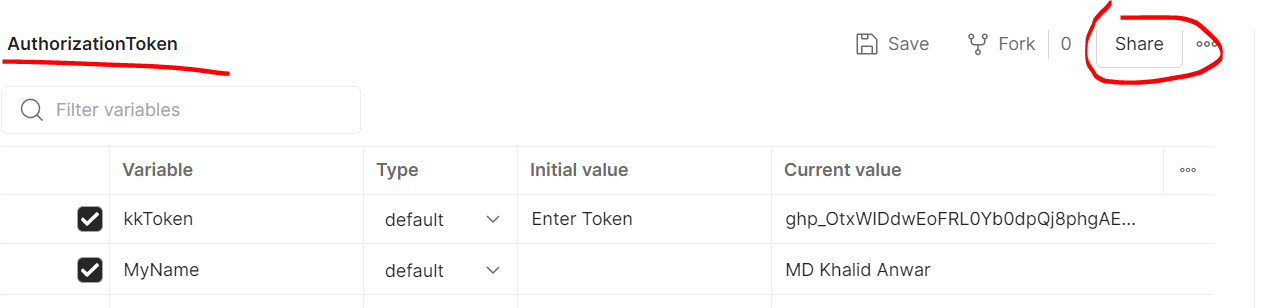
#### Creating an Environment

Click on Environment🡪+ sign 🡪Enter environment name🡪 enter variable name and default value



#### Share Environment

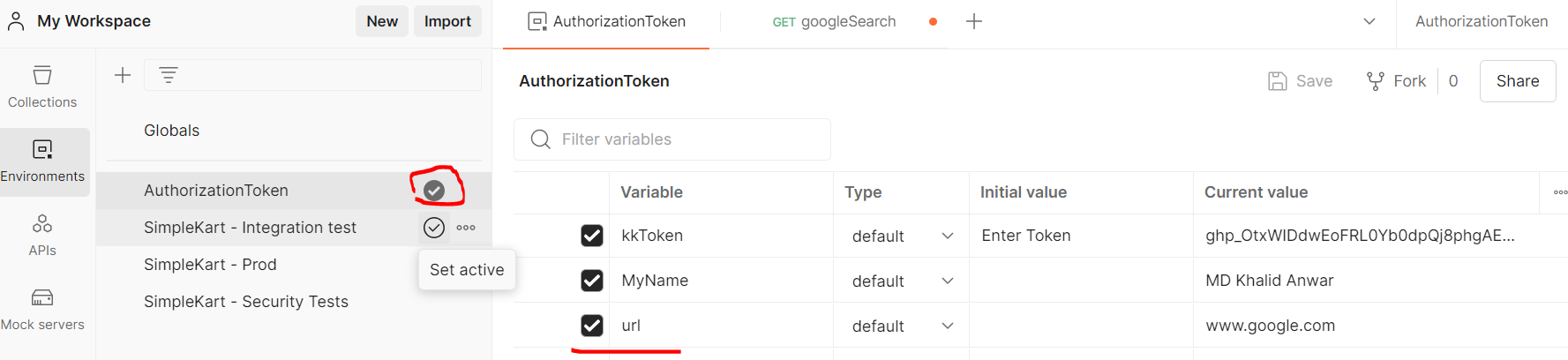
We can also download, share, delete, duplicate, or import the environment.

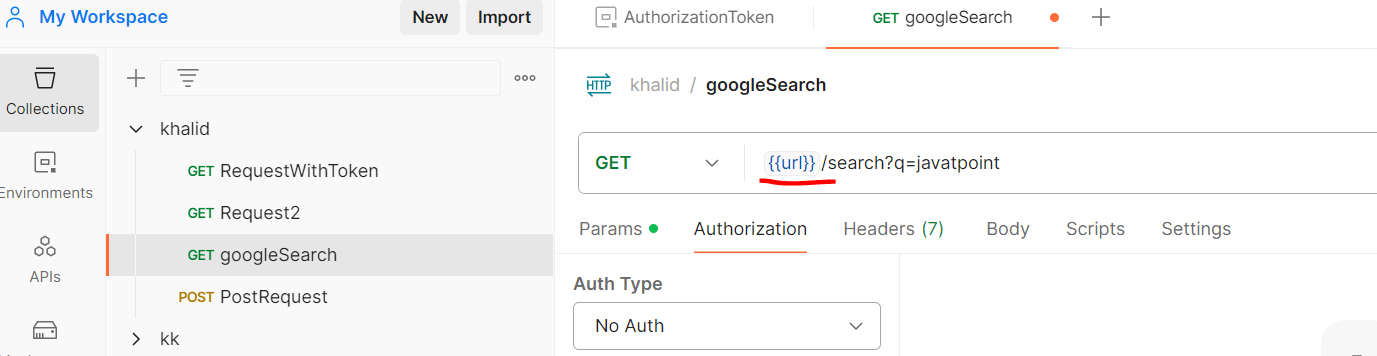


#### Using Environment Variable

One environment should be active at a time. Listed variable can be used anywhere in the http request by

{{url}}



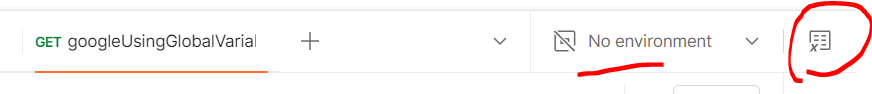


## Global Variables in Postman

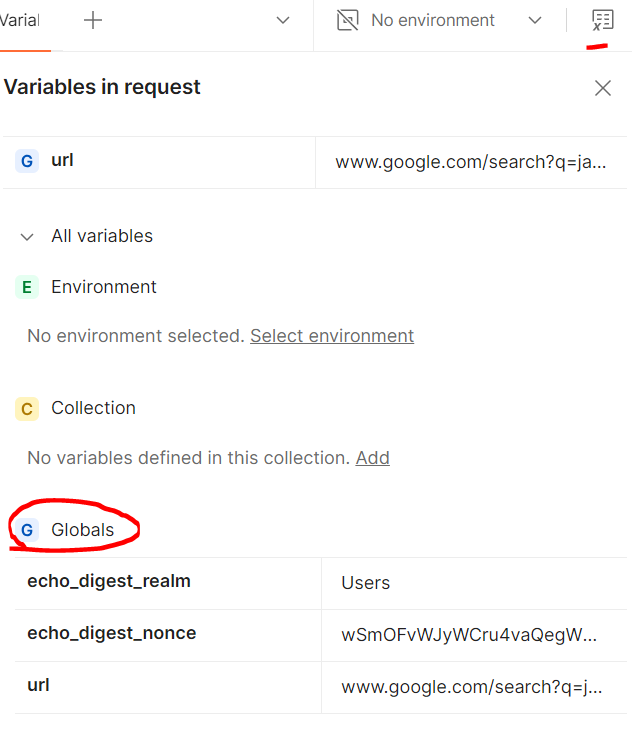
Unlike the environment variable, in the case of global variables, we don't need to create an environment first.

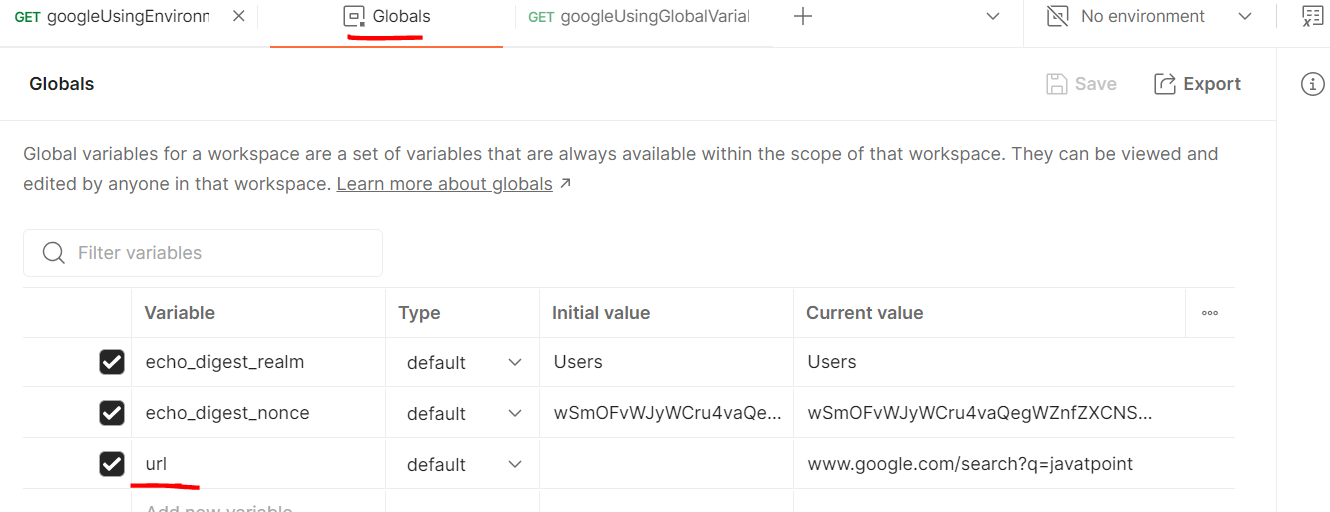
**Step1**: Open request

**Step2**: Click on **round** Click



Step3: Click on **Global**





## Variable Scopes

The following variable scopes are supported by Postman, which is used for different tasks. The narrowest variable scope is Local, and then next is Data, Environment, Collection, and the broadest one is Global.



In case, the variable name is the same, in two scopes with a different value, then the value of the narrowest variable scope will be used while running the request. Let's say the name of both local and global variables is **My Variable.** Then Postman will use a local variable value to run the request, as a **local variable** is the narrowest variable scope.

Global Variables  
These are independent of environments and function outside the environment. Users are not required to create an environment for global variables. Through global variables, we can access data between **collections**, **test scripts**, **requests**, and **environments**.

Collection Variables  
these are independent of environments and are available for all the requests in a collection.

Environment Variables  
These are the most used variable scope in Postman. At a time, a single environment can be active. They are bound to a specified environment used to execute the request.

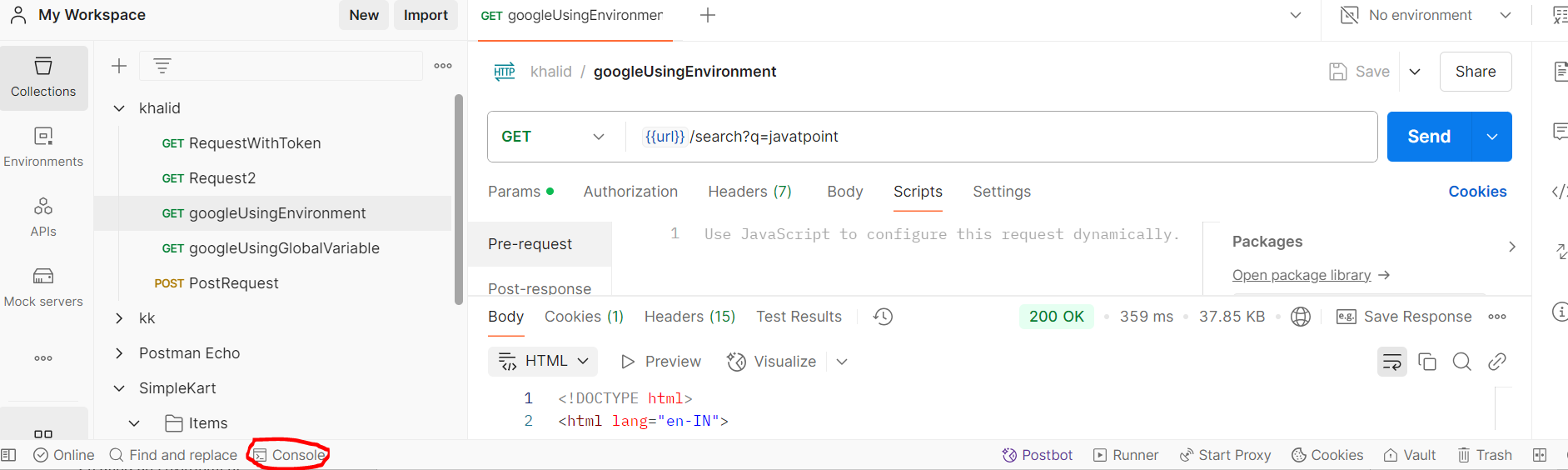
Data Variables  
Data variables come from external CSV and JSON files to define data sets that you can use while collections are running through Newman or the Collection Runner.

Local Variables  
These are temporary variables that will only function inside the environment they were created in. When you change the environment, the variable will stop its access, and an error will occur.

## ****Postman console****

It stores all the things that happen in the request. Through the postman console, we can see the logs and errors. It is like browsers development console in which all the things are visible, including whole requests that users have sent to that website or to the page code.

To open the postman console, select the "Postman Console" icon from the bottom of the window or press ctrl+alt+c.



## Postman Automation Testing

Postman is not only used for manual testing, but we can also do automation testing of API. Postman requests and collections can be applied to dynamic behavior.

In Postman Scripts are the lines of code that allow you to automate an API test. It offers you to write pre-request and test scripts.

* Before sending a request, a **pre-request script** will run and,
* After receiving a response, **test scripts** will run.

### Postman Sandbox

To write the script in Postman, we use Postman Sandbox.

[Postman](https://www.javatpoint.com/postman) sandbox has been written in [Javascript](https://www.javatpoint.com/javascript-tutorial). It is an execution environment. Since the sandbox has written in Javascript, therefore, it will only receive a script written in Javascript.

### Execution Order



The scripts should always run according to the following hierarchy for every request in a collection:

1. Collection level script
2. Folder level script
3. Request level script

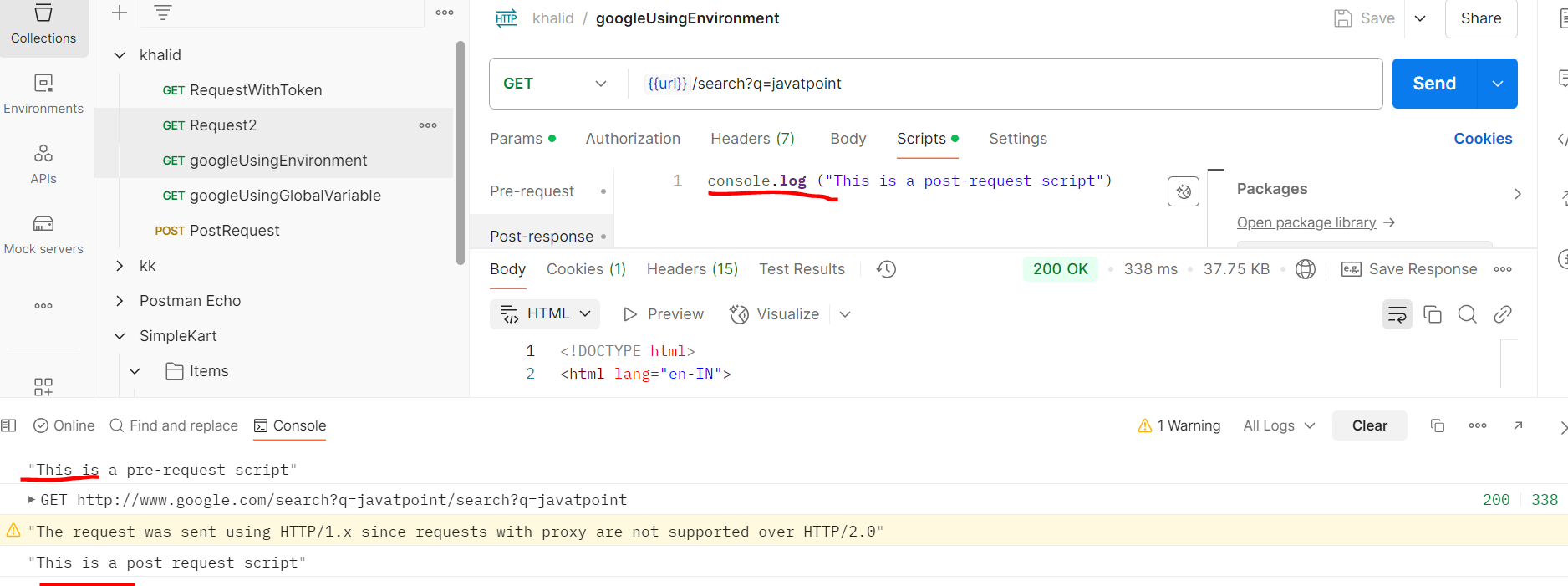
### Pre-request Script

<https://www.javatpoint.com/postman-test-scripts>

Pre-request scripts are a piece of code that will run before the execution of a request. We can apply the pre-request scripts at a **request level**, a **collection level**, or a **folder level**, which means that a pre-request script will apply to all the requests that are part of that collection or folder.

Click on below highlighted link to write Pre and Post Script

console.log ("This is a pre-request script")



### Post-request Script

Test scripts execute after the server receives the response. We can run the test scripts to a single request, folders, or collections.

console.**log**("This is a post-request script ")

var abc**=**pm.response.json() //Reading complete JSON response and assigned to abc variable

console.**log**(abc.data.id) //data🡪id key hierarchy

#### pm.test()

Syntax

pm.test(“Test Steps Name”,function()

{

pm.response.to.have.status(200);  //pm=Object name pm.response will return entire response body

})

This is writing test steps and status will be shown after test execution

#### pm.except()

Assertion method. The pm.except() function is used to print the test result messages in a different format

pm.**test**("Status", function () {

    pm.**expect(**pm.response.code**)**.to.equal(404);

});

**Variable in Script**

pm.variables.**set**("KeyName’",”Value”); #to **create** local variable

pm.variables.**get**(‘KeyName’);  #to **read** local variable

pm.variables.**unset**(‘KeyName’);  #to **Clear** an local Variable

**Environment variable in Script**

pm.environment.**set**('username','khalidUser');  #to **create** environment variable

pm.environment.**get**(‘KeyName’);  #to **read** environment variable

pm.environment.**unset**(‘KeyName’);  #to **Clear** an Environment Variable

**Collection variable in Script**

pm.collectionVariables.**set**('username','khalidUser');  #to **create** collection variable

pm.collectionVariables.**get**(‘KeyName’);  #to **read** collection variable

pm.collectionVariables.**unset**(‘KeyName’);  #to **Clear** an collection Variable

**Global variable in Script**

pm.globals.**set**('username','khalidUser');  #to **create** collection variable

pm.globals.**get**(‘KeyName’);  #to **read** collection variable

pm.globals.**unset**(‘KeyName’);  #to **Clear** an collection Variable

### Response Handling

<https://www.javatpoint.com/postman-test-examples>

#### ****Search for a String in Response Body****

pm.test("Body Contains", function () {

    pm.expect(pm.response.text()).to.**include**("search\_string");

});

#### ****Check if Response Body is equal to a String****

pm.test("Body is correct", function () {

    pm.response.to.have.body("response\_body\_string");

});

#### ****Check for a JSON Value****

pm.test("Your test name", function () {

    var jsonData = pm.response.json();

    pm.expect(jsonData.value).to.**eql**(100);

});

#### ****Check if Content-Type header is present****

pm.test("Content-Type header is present", function () {

    pm.response.to.have.**header**("Content-Type");

});

#### ****Response Time is less than 200 ms****

pm.test("Response time is less than 200ms", function () {

    pm.expect(pm.response.responseTime).to.be.below(200);

});

#### ****Status code is 200****

pm.test("Status code is 200", function () {

    pm.response.to.have.status(200);

});

#### ****Successful POST request status code****

pm.test("POST request Successful ", function () {

    pm.expect(pm.response.code).to.be.**oneOf**([201,202]);

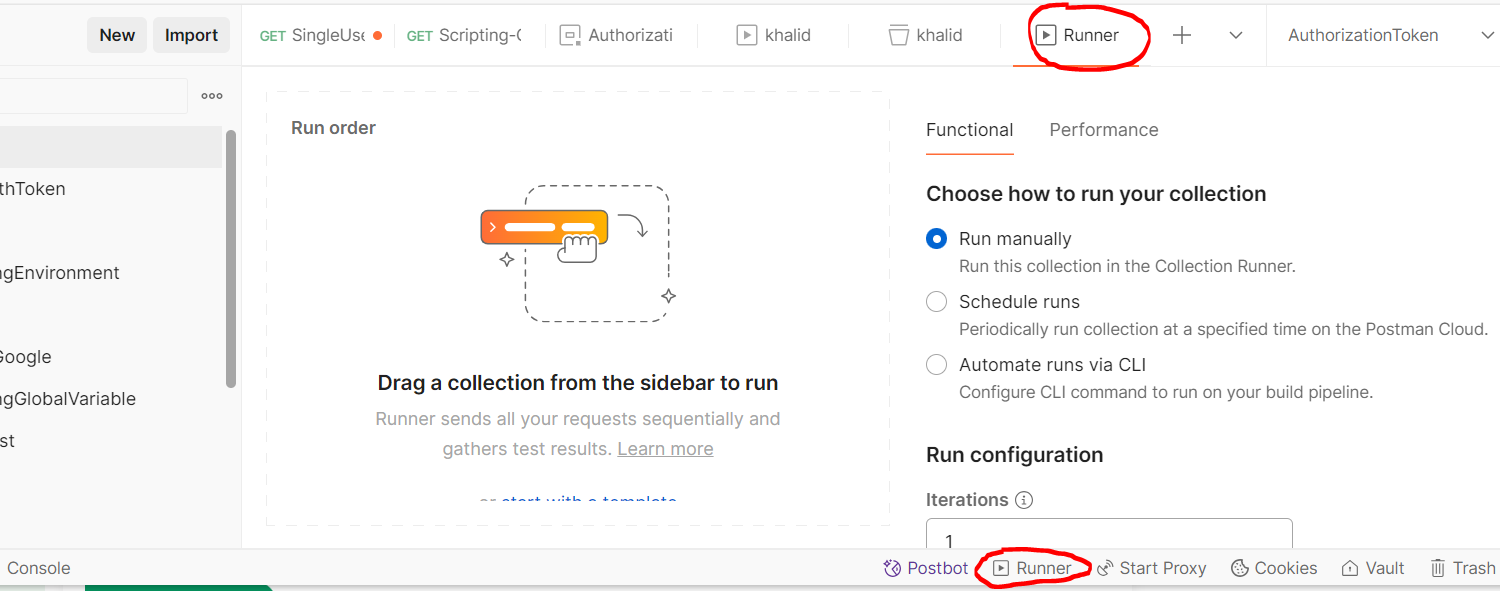
});

### Postman Workflow

POSTMAN workflow is the order of request execution within a collection. When you run the collection directly, then the requests are either executed randomly or executed serially. But we can also set the priority of the request, and we can also choose the order of execution of request.

Postman provides a feature to control the workflow of requests in the collection, and the feature is called **Collection Runner** (**or Runner**).

Once a request is executed by the collection runner, all requests within the collection are executed, but there is a pattern in which all requests are executed. [Postman](https://www.javatpoint.com/postman) provides many inbuilt algorithms to decide in which pattern the request has to be run.



### Postman Assertion

**The assertion** checks whether the given predict is true or false in any programming language. In [Postman](https://www.javatpoint.com/postman), we can write the assertion in many ways. One of the simplest ways is the snippets, which are nothing but a block of codes that have some unique functions inside it and are available in the postman app.

We can access the name and value of the JSON array with the help of the dot (.) operator.

console.log(jsonData.data[2].id);

### [Mock Servers](https://www.javatpoint.com/postman-mock-servers)

<https://www.javatpoint.com/postman-mock-servers>

* The mock server is a fake server that simulates as a real server to help the users for testing and checking their APIs and APIs responses.
* During the various API testing, a Mock server is used when specific responses are required to be received that are not readily available on the web servers

#### Types of Mock Servers

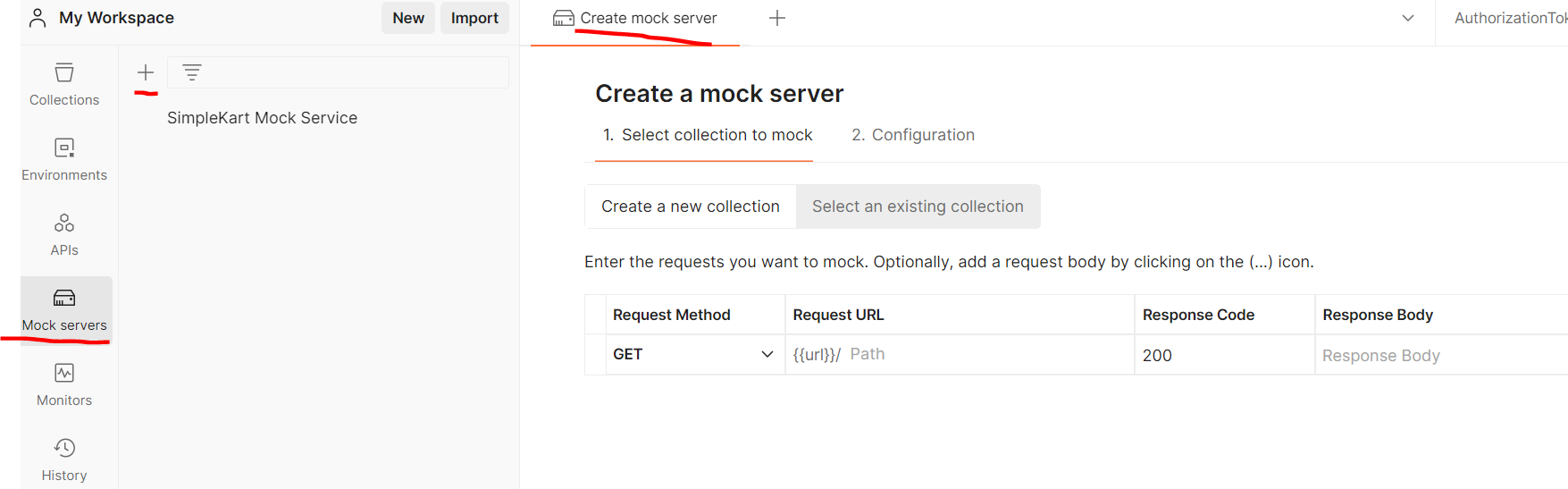
**Private mock server:** These cannot be accessed by everyone.

**Public mock server:** Public mock servers are open to all. When you share this mock server, users are not required to add a Postman API key to consume the mock.

#### Setting Up a Mock Server

To set up a mock server in Postman, we will need a collection because mocks in Postman are tied with a collection. Postman compares the requests and creates mock responses from the examples in the requests of a collection.

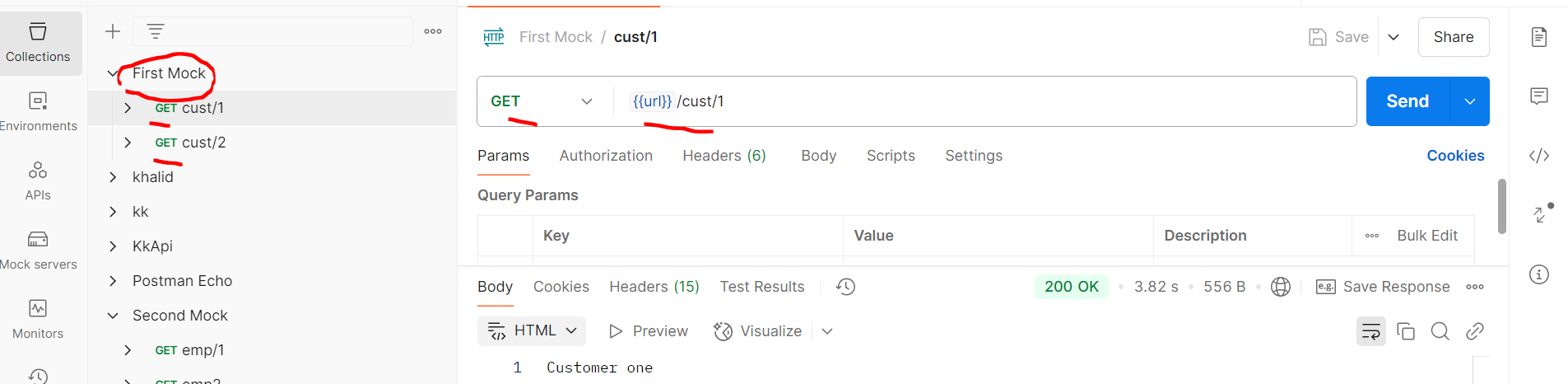
Click on “Mock Servers”🡪+plus sign🡪



Here,

* **The method** is request type methods such as GET, POST, etc.
* **Request Path** is the URL of the request for your API.
* **Response Code** is the code in which we want to get in response.
* **Response Body** is the body of the response that you want to show in response.

After creating mock server, same name as collections will be created and all APIs will be shown under that collection



## Postman Continuous Integration

<https://www.javatpoint.com/postman-continuous-integration>

## Branching and Looping

In Postman, we can also work on branching and looping. We can branch and loop across API requests with the help of **setNextRequest**() function. This function is used to set the request to be executed next. This function takes one argument, i.e., the name of the request, which will execute next.

### features of setNextRequest():

postman.setNextRequest("request\_name");

* Provide the request name or request ID, and the collection runner will handle the rest.
* We can use the setNextRequest() in the test script as well as in the pre-request script. The last set value takes precedence when you have more than one assignment.
* The collection runner will follow the linear execution settings from default settings and moves to the next request if Postman.setNextRequest() isn't given in a request.

### Loop Over the Current Request

If you give the currently running request name in the argument of setNextRequest() function, then Postman will run the same request continuously.

### Stop Workflow Execution

To stop the execution of workflow pass the null argument in setNextRequest() function.

postman.setNextRequest(**null**);