**API TESTING(POSTMAN) NOTES(By USMAN JILANI)**

## Postman Introduction

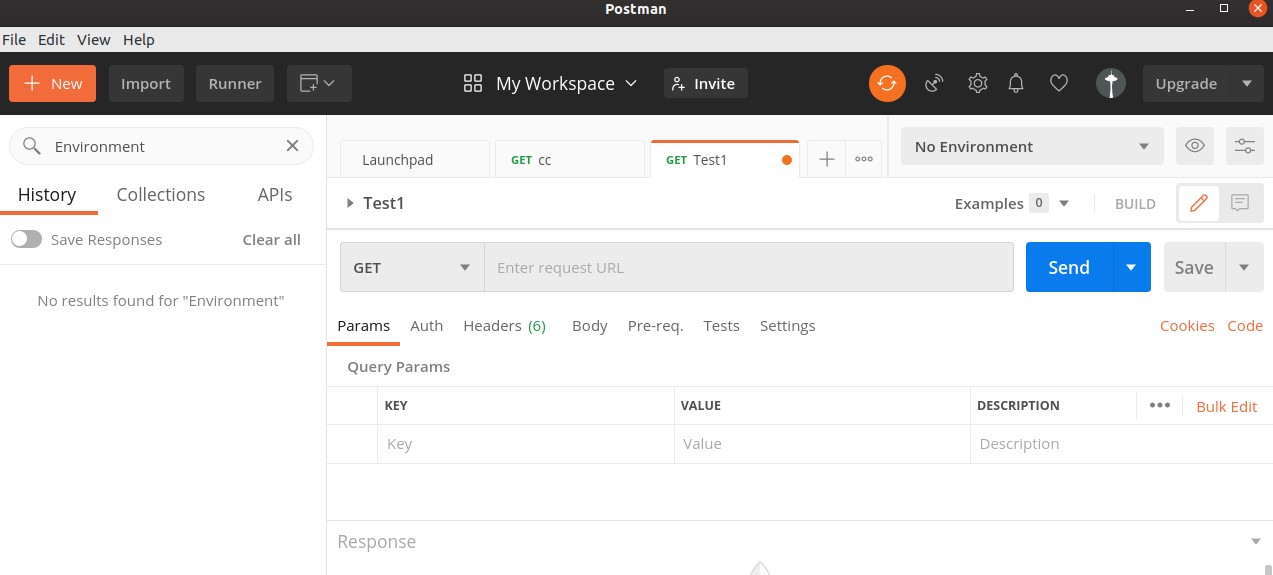
Postman is an Application Programming Interface (API) testing tool. API acts like an interface between a couple of applications and establishes a connection between them.

## Working with Postman

To start working with Postman, we have the navigations as shown below. It primarily consists of four sections:

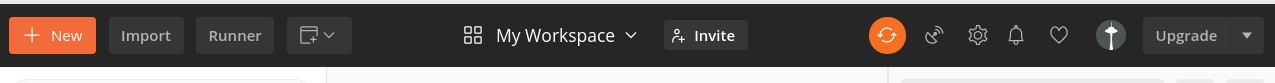
* Header
* Response
* Sidebar
* Builder

Given below is the screenshot of the navigations available in Postman:

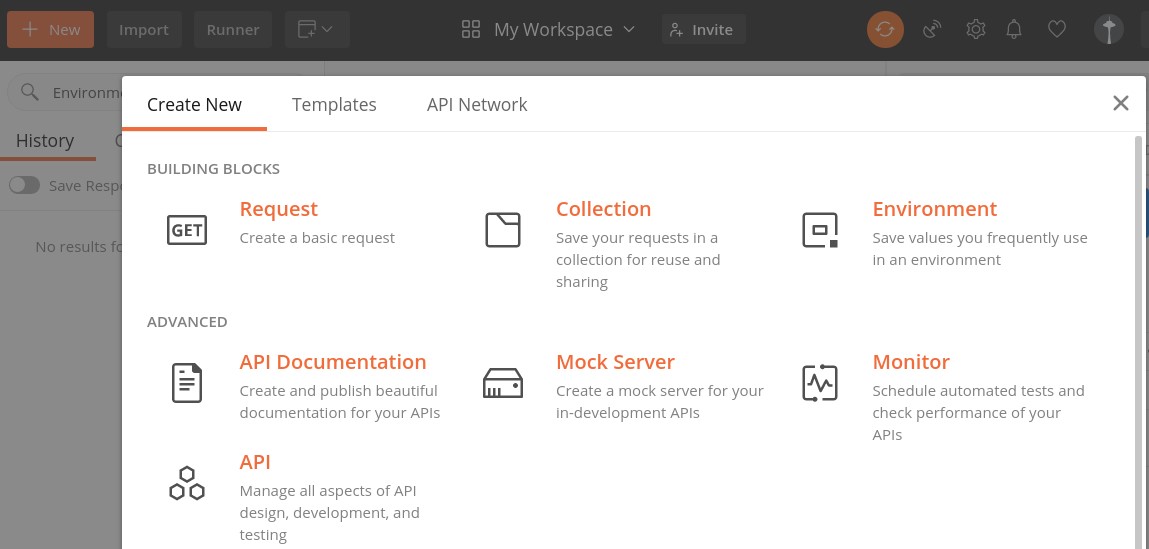


**HEADER**

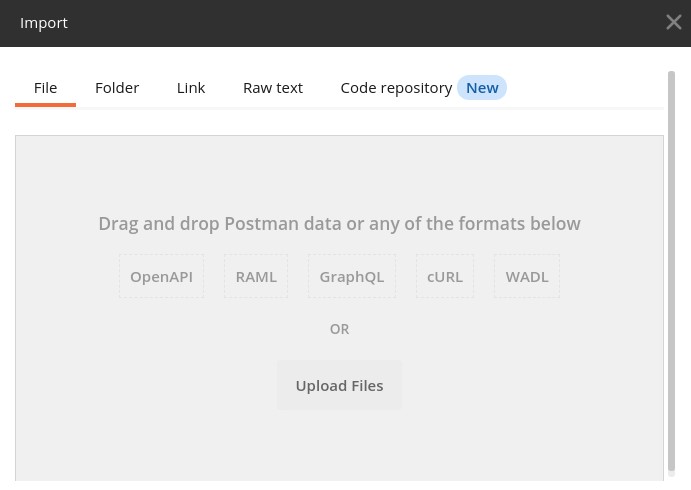
Postman consists of New, Import, Runner (used to execute tests with Collection Runner), Open New, Interceptor, Sync menus, and so on. It shows the workspace name – My Workspace along with the option for Invite for sharing it among teams.



New menu is used to create a new Environment, Collection or request. The Import menu helps to import an Environment/Collection.

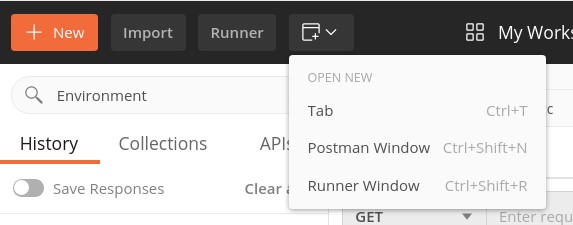


We can import from a File, Folder, Link, Raw text or from Code repository options which are also available under Import.



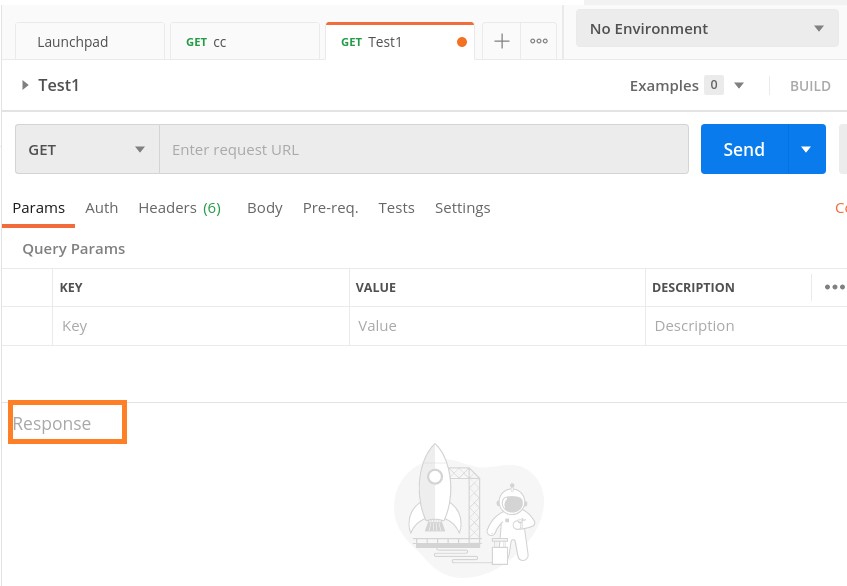
Here, Open New is used to open a new tab, Postman or a Runner Window.

**Response**

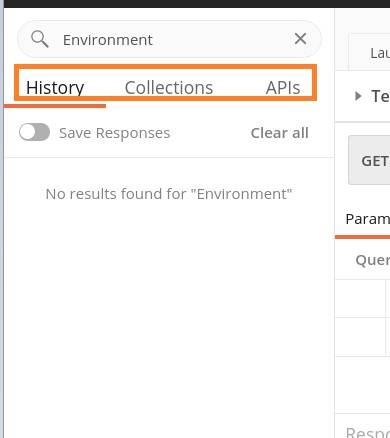


Response section shall have values populated only when a request is made. It generally contains the Response details.

**Sidebar**

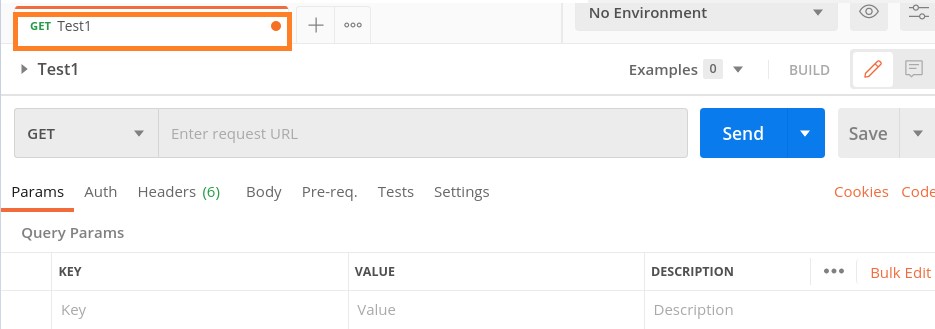


Sidebar consists of Collections (used to maintain tests, containing folders, sub-folders, requests), History (records all API requests made in the past), and APIs.

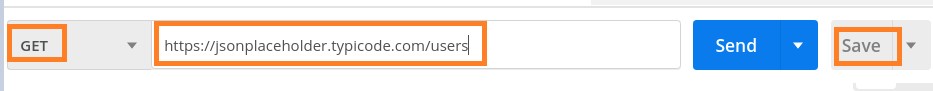


### Builder

Builder is the most important section of the Postman application. It has the request tab and displays the current request name. By default, Untitled Request is mentioned if no title is provided to a request.



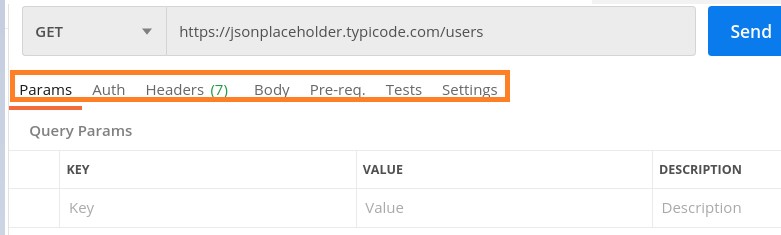
The Builder section also contains the request type (GET, POST, PUT, and so on) and URL. A request is executed with the Send button. If there are any modifications done to a request, we can save it with the Save button.



The Builder section has the tabs like Param, Authorization, Headers, Body, Pre-req., Tests and Settings. The parameters of a request in a key-value pair are mentioned within the Params tab. The Authorization for an API with username, password, tokens, and so on are within the Authorization tab.

The request headers, body are defined within the Headers and Body tab respectively. Sometimes, there are pre-condition scripts to be executed prior to a request. These are mentioned within the Pre-req. tab.

The Tests tab contains scripts that are run when a request is triggered. This helps to validate if the API is working properly and the obtained data and Response code is correct.



**HTTP REQUESTS:**

Following are the well known http request we handle in postman for API testing

* GET
* POST
* PUT
* PATCH
* DELETE

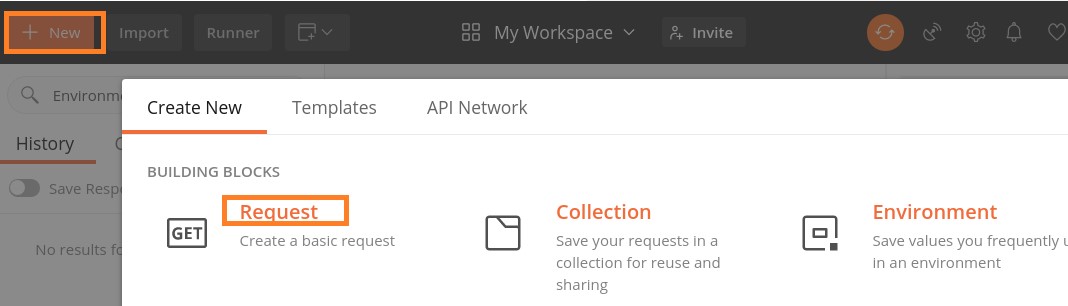
**GET REQUEST:**

A GET request is used to obtain details from the server and does not have any impact on the server. The GET request does not update any server data while it is triggered. The server only sends its Response to the request.

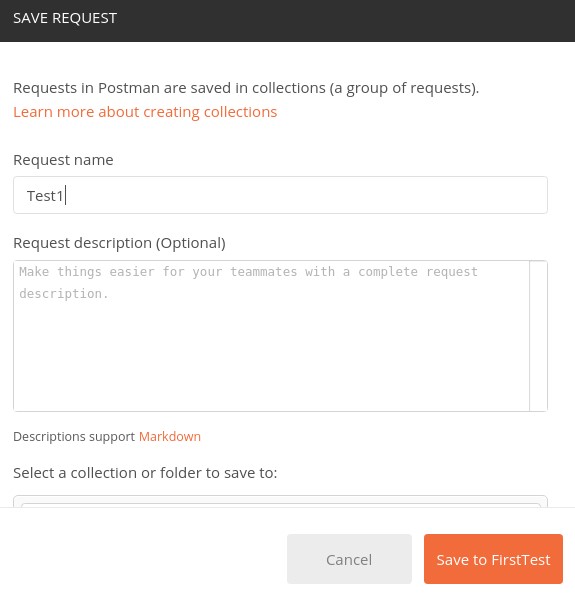
## Create a GET Request

Follow the steps given below to create a GET request successfully in Postman:

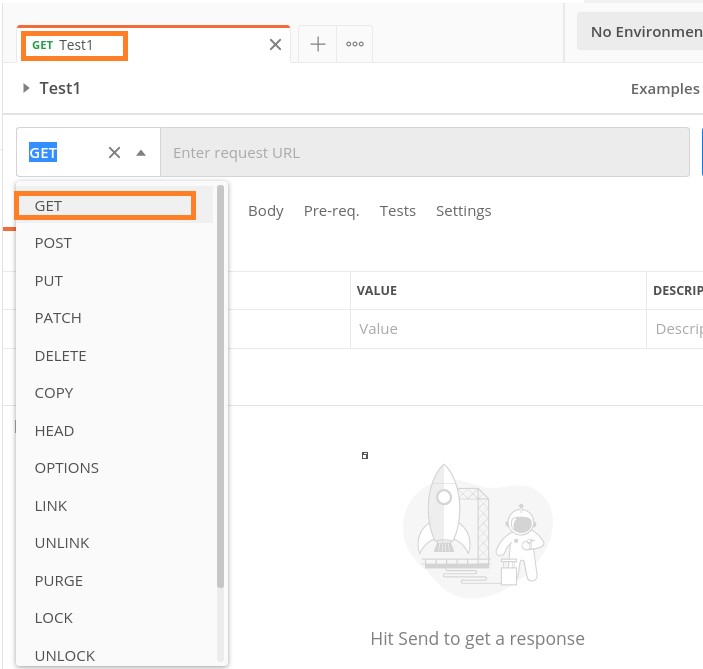
**Step 1**: Click on the New menu from the Postman application. The Create New pop-up comes up. Then click on the Request link.



**Step 2**: SAVE REQUEST pop-up comes up. Enter the Request name then click on Save.



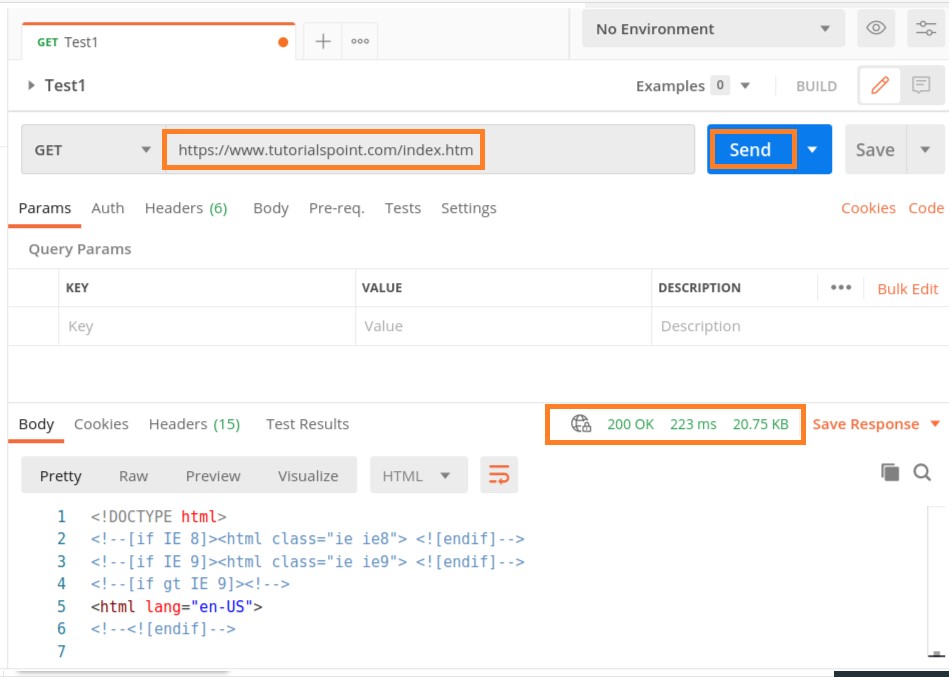
**Step 3**: The **Request name (Test1)** gets reflected on the Request tab. We shall then select the option **GET** from the HTTP request dropdown.



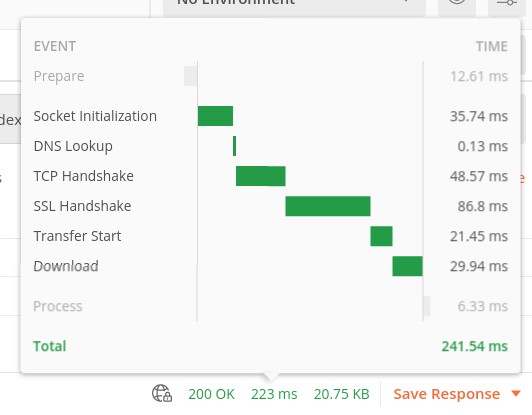
**Step 4**: Enter an URL -<https://www.tutorialspoint.com/index.htm>in the address bar and click on Send.

### Response

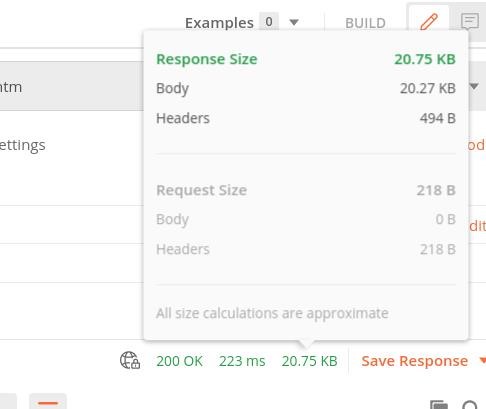
Once a request has been sent, we can see the response code **200 OK** populated in the Response. This signifies a successful request and a correct endpoint. Also, information on the time consumed to complete the request (223 ms) and payload size (20.75 KB) are populated.



On hovering over the response time, we can see the time taken by different events like DNS Lookup, SSL Handshake and so on.



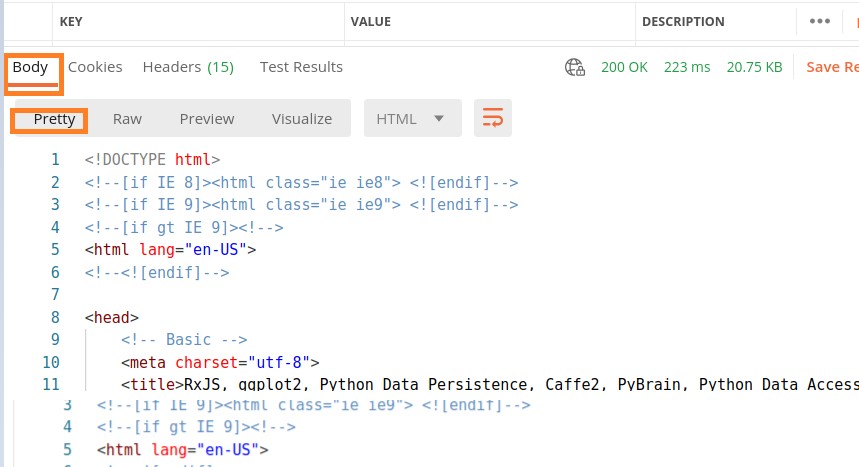
On hovering over the payload size, the details on the size of response, headers, Body, and so on are displayed.



The Response Body contains the sub-tabs – Pretty, Raw and Preview. The Pretty format shows color formatting for keywords and indentation for easy reading. The Raw format displays the same data displayed in the Pretty tab but without any color or indentation.

The Preview tab shows the preview of the page.

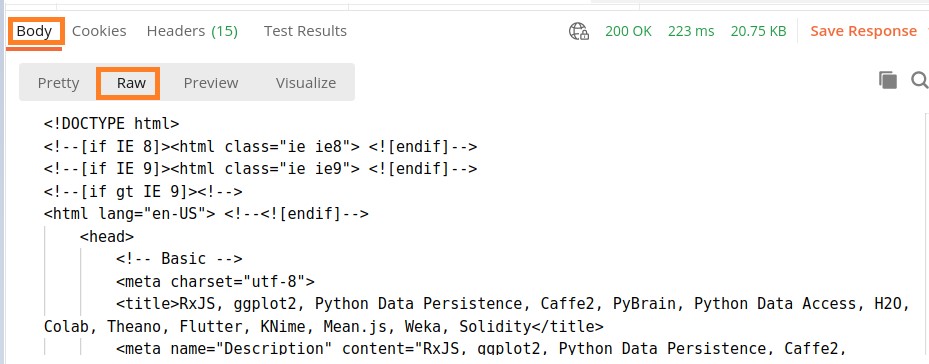
**Raw tab**



The following screen will appear:

**Preview ta**

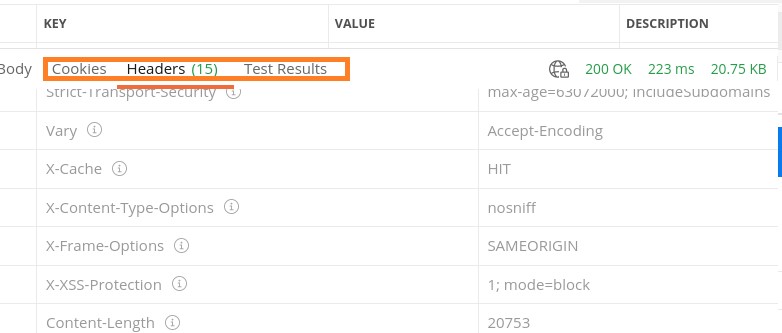
**b**



The following screen will appear:



The Response also contains the Cookies, Headers and Test Results.



**POST REQUEST**

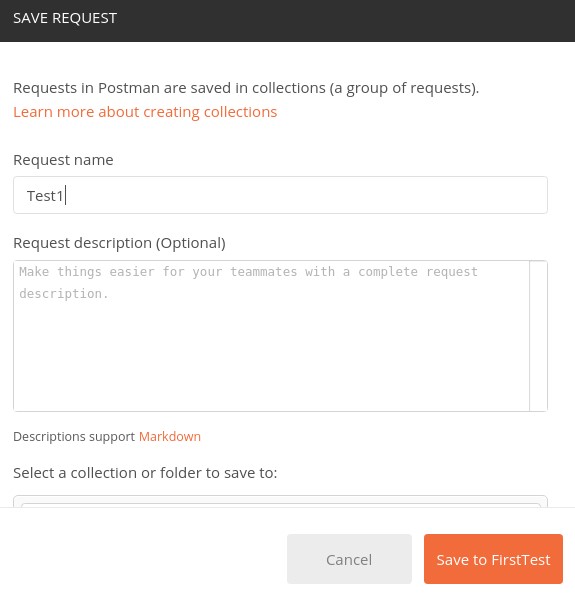
Postman POST request allows appending data to the endpoint. This is a method used to add information within the request body in the server. It is commonly used for passing delicate information.

Once we send some the request body via POST method, the API in turn yields certain information to us in Response. Thus, a POST request is always accompanied with a body in a proper format.

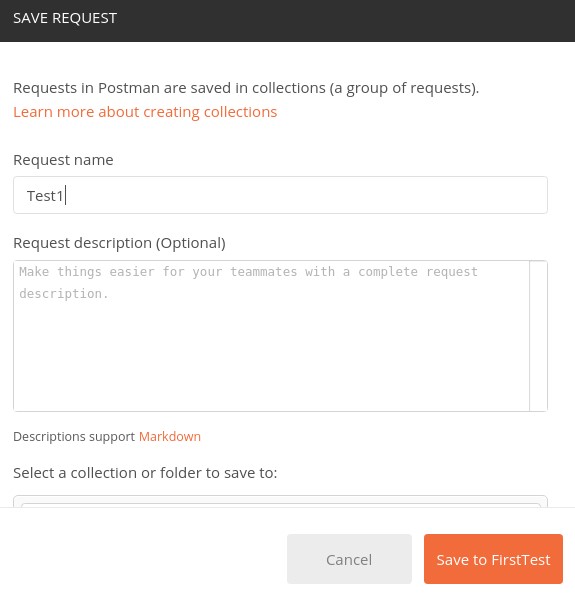
## Create a POST Request

Follow the steps given below to create a POST request successfully in Postman:

**Step 1**: Click on the **New** menu from the Postman application. The **Create New** pop-up comes up. Then, click on the Request link.

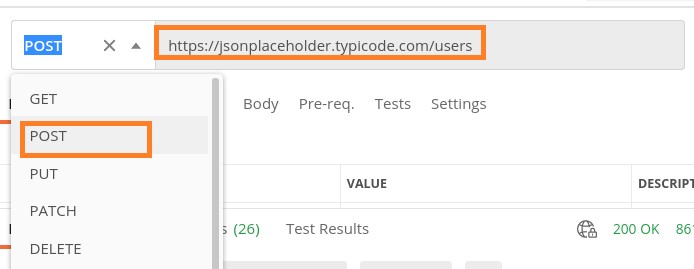


**Step 2**: **SAVE REQUEST** pop-up comes up. Enter the Request name then click on **Save**.

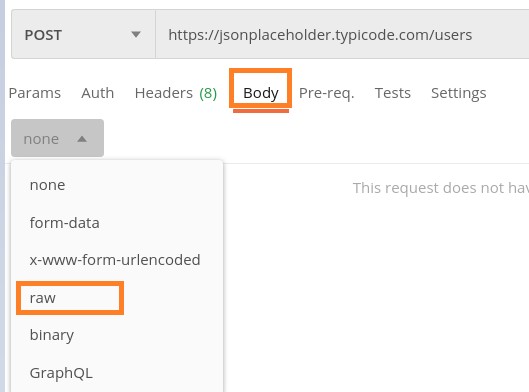


**Step 3**: The **Request name (Test1)** gets reflected on the Request tab. Also, we shall select the option **POST** from the HTTP request dropdown.

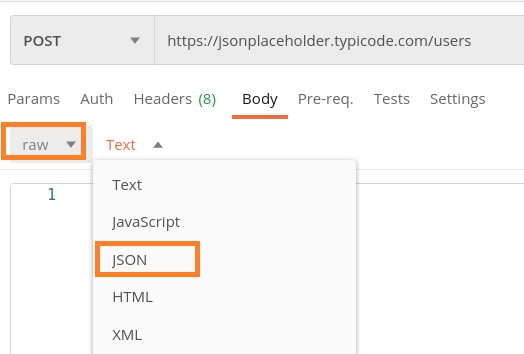
Then, enter an URL https://jsonplaceholder.typicode.com/users in the address bar.



**Step 4**: Move to the Body tab below the address bar and select the option **raw**.



**Step 5**: Then, choose **JSON** from the Text dropdown.



**Step 6**: Copy and paste the below information in the Postman Body tab.

{

"id": 11,

"name": "Tutorialspoint", "username": "Test1",

"email": "abc@gmail.com",

"address": {

"street": "qa street",

"suite": "Apt 123",

"city": "Kochi",

|  |
| --- |
| "zipcode": "49085", "geo": {  "lat": "-3.3155",  "lng": "94.156"  }  },  "phone": "99599125",  "website": "Tutorialspoint.com",  "company": {  "name": "Tutorialspoint",  "catchPhrase": "Simple Easy Learning",  "bs": "Postman Tutorial"  }  } |

The above data that is being sent via POST method is only applicable to the endpoint: [https://jsonplaceholder.typicode.com/users.](https://jsonplaceholder.typicode.com/users)

To pass the data in the correct JSON format, we can use the Jsonformatter available in the below link:

<https://jsonformatter.curiousconcept.com/>

**Step**

**7**

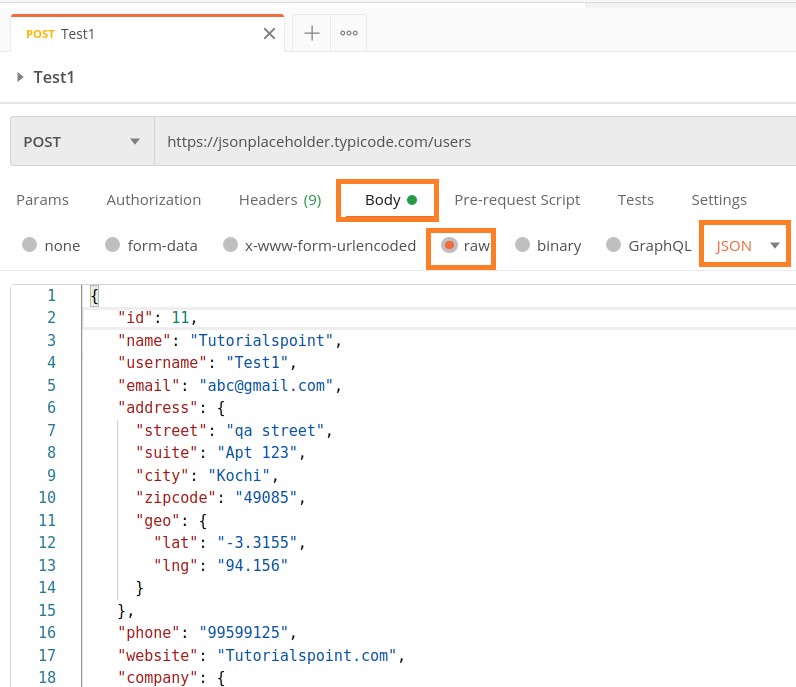
:

Click on the

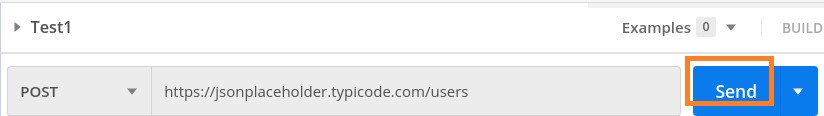
**Sen**

**d**

button.

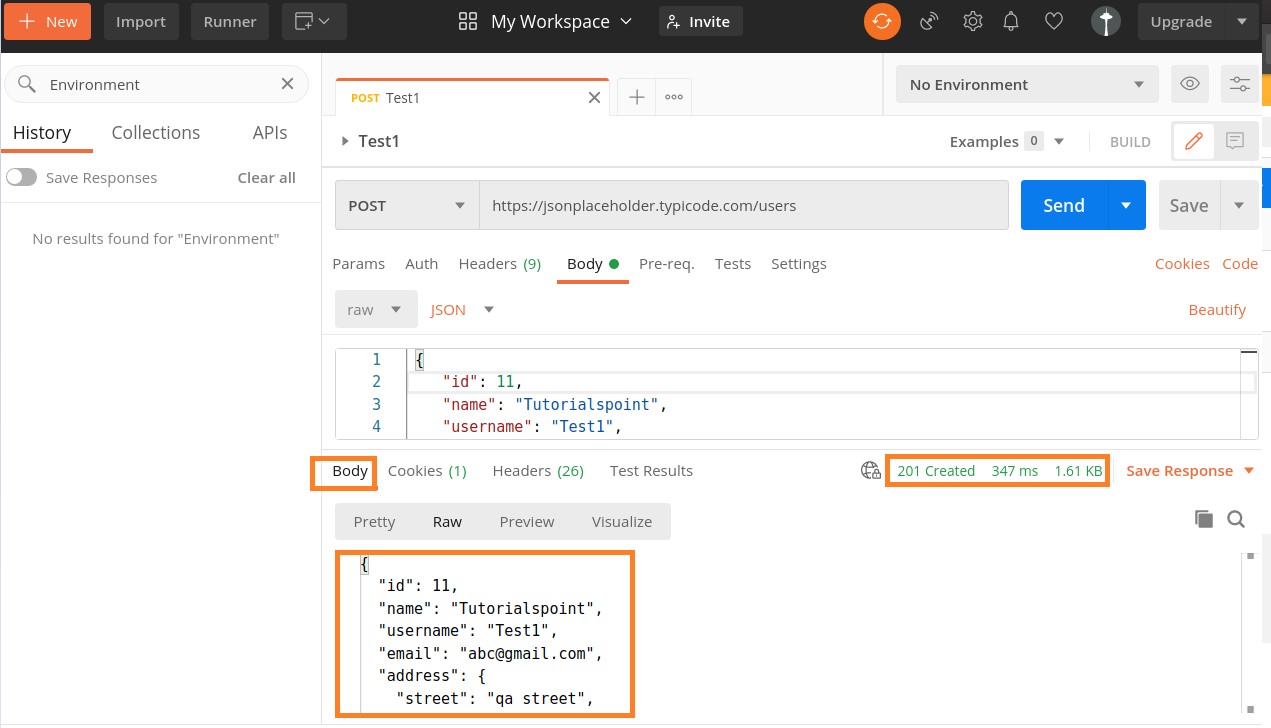


**Response**

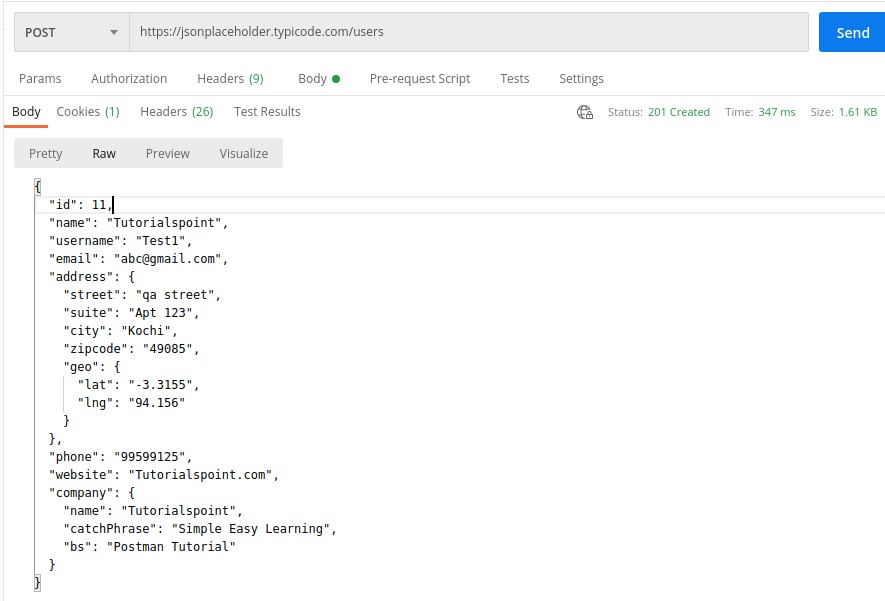


Once a request has been sent, we can see the response code **201 Created** populated in the Response. This signifies a successful request and the request we have sent has been accepted by the server.

Also, information on the time consumed to complete the request (347 ms) and payload size (1.61 KB) are populated.



We can see that the Response body is the same as the request body which we have sent to the server.



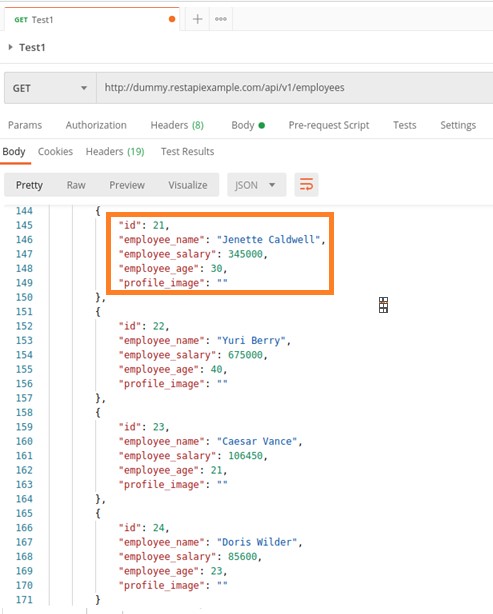
**PUT REQUEST**

A Postman PUT request is used to pass data to the server for creation or modification of a resource. The difference between POST and PUT is that POST request is not idempotent.

This means invoking the same PUT request numerous times will always yield the same output. But invoking the same POST request numerous times will create the similar resource more than one time.

Before creating a PUT request, we shall first send a GET request to the server on an endpoint: [http://dummy.restapiexample.com/api/v1/employees.](http://dummy.restapiexample.com/api/v1/employees) The details on how to create a GET request is explained in detail in the Chapter – Postman GET Requests.

On applying the GET method, the Response body obtained is as follows:

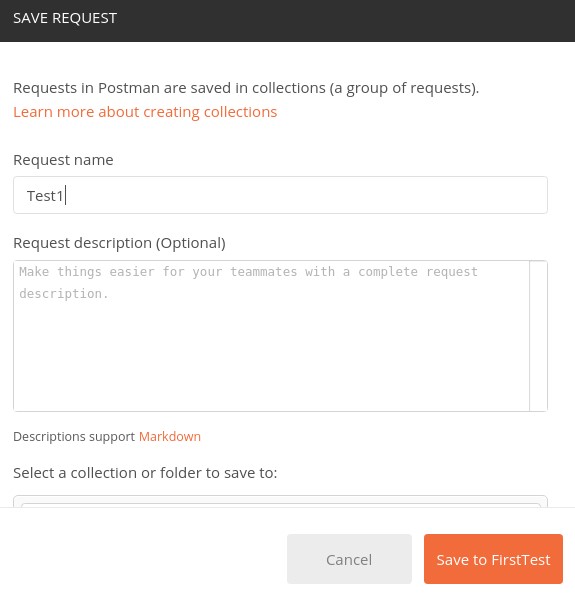


Now, let us update the **employee\_salary** and **employee\_age** for the id 21 with the help of the PUT request.

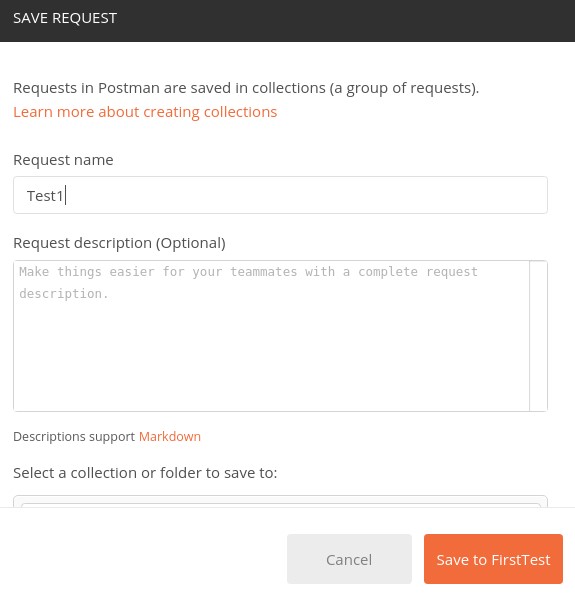
## Create a PUT Request

Follow the steps given below to create a PUT request in Postman successfully:

**Step 1**: Click on the **New** menu from the Postman application. The **Create New** pop-up comes up. Then, click on the Request link.



**Step 2**: **SAVE REQUEST** pop-up comes up. Enter the Request name then click on **Save**.

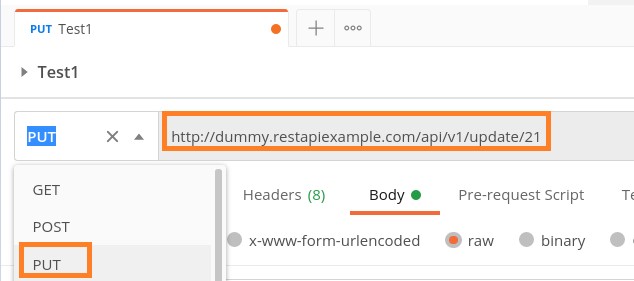


**Step 3**: The Request name (Test1) gets reflected on the Request tab. We shall select the option PUT from the HTTP request dropdown.

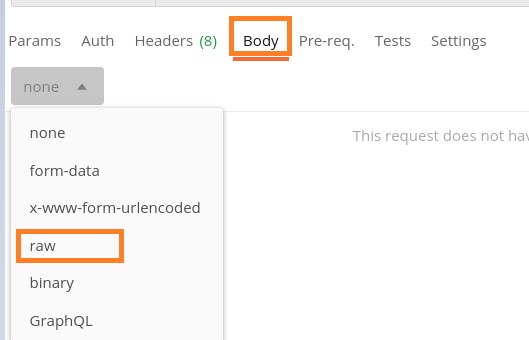
Then enter the URL - [http://dummy.restapiexample.com/api/v1/update/21](http://dummy.restapiexample.com/api/v1/update/21/) (endpoint for updating the record of id 21) in the address bar.

It must be noted that in a PUT request, we have to mention the id of the resource in the server which we want to update in the URL.

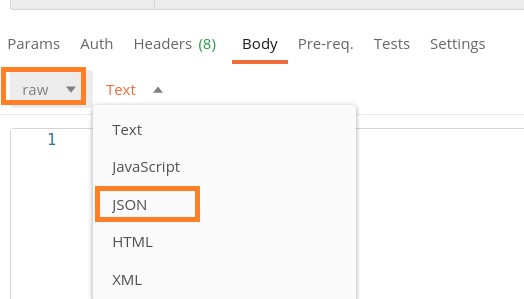
For example, in the above URL we have added the id 21.



**Step 4**: Move to the **Body** tab below the address bar and select the option **raw**.



**Step 5**: Then, choose JSON from the Text dropdown.



**Step 6**: Copy and paste the below information in the Postman Body tab.

{ "name": "Jenette Caldwell","salary": "2000","age": "15"}

The overall parameters to be set for a PUT request are shown below:

**Step**

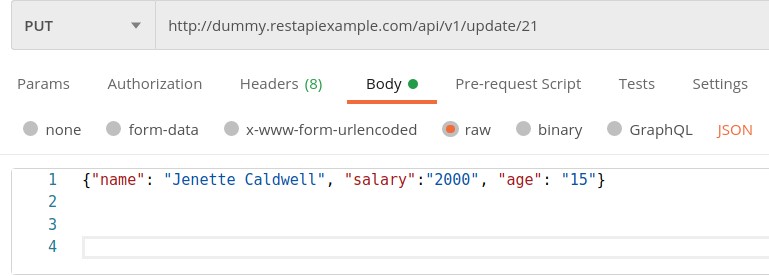
**7**

:

Click on the

**Send**

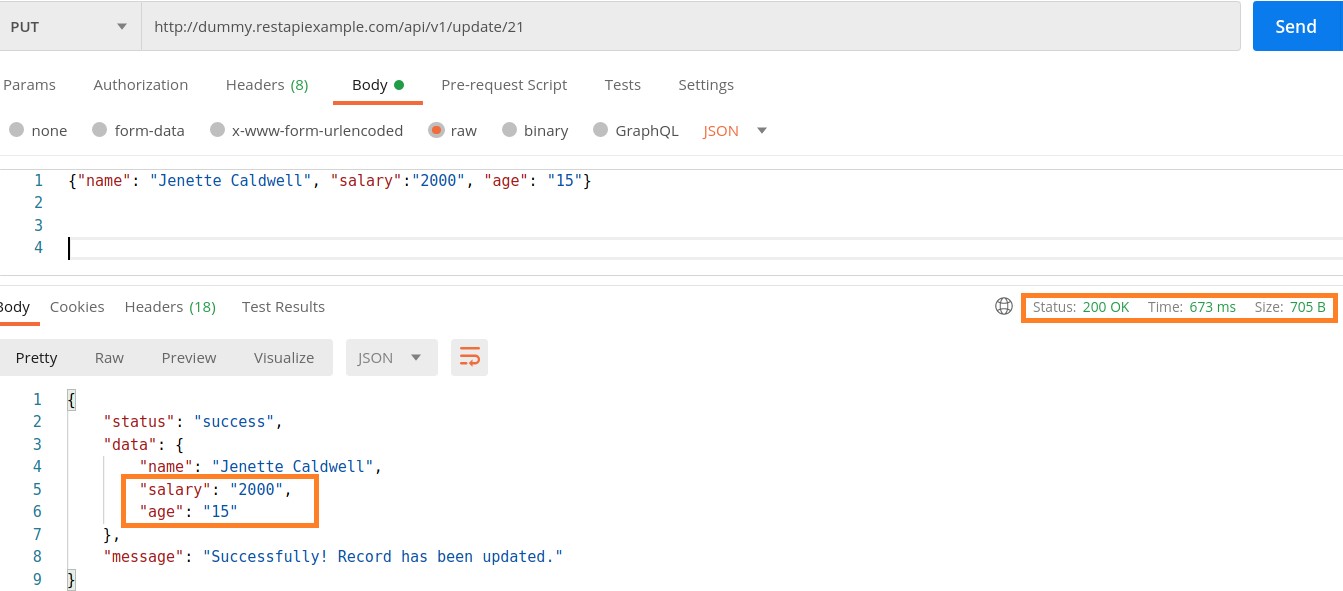
button.



### Response

Once a request has been sent, we can see the response code 200 OK populated in the Response body. This signifies a successful request and the request we have sent has been accepted by the server.

Also, information on the time consumed to complete the request (673 ms) and payload size (705 B) are populated. The Response body shows the salary and age got updated to 2000 and 15 respectively for the employee having id 21.



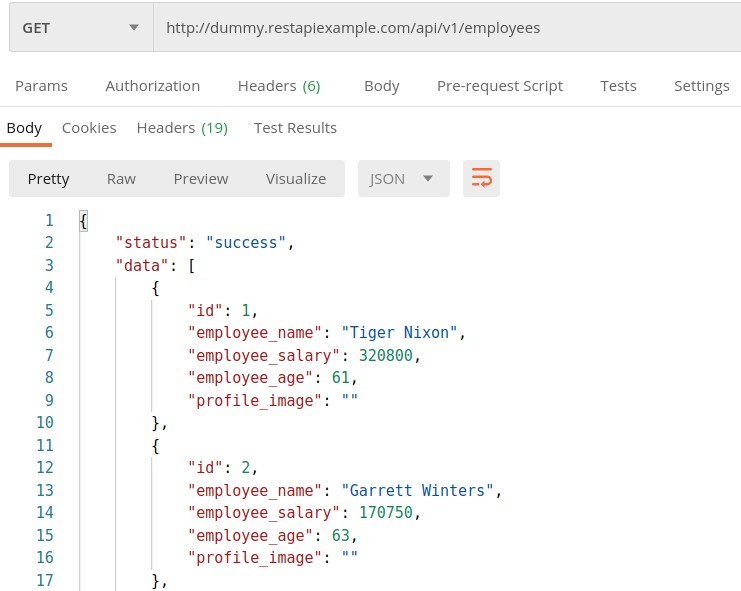
**DELETE REQUEST**

Postman DELETE request deletes a resource already present in the server. The DELETE method sends a request to the server for deleting the request mentioned in the endpoint. Thus, it is capable of updating data on the server.

Before creating a DELETE request, we shall first send a GET request to the server on the endpoint: [http://dummy.restapiexample.com/api/v1/employees.](http://dummy.restapiexample.com/api/v1/employees) The details on how to create a GET request is explained in detail in the Chapter onGET Requests.

On applying the GET method, the below Response Body is obtained:

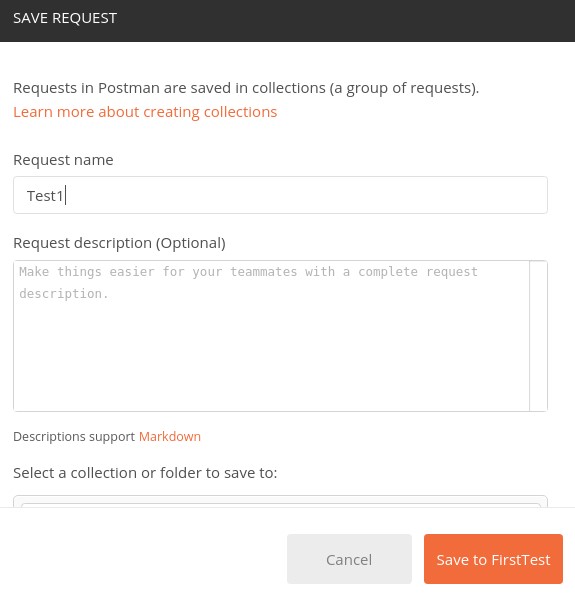
Let us delete the record of the id 2 from the server.



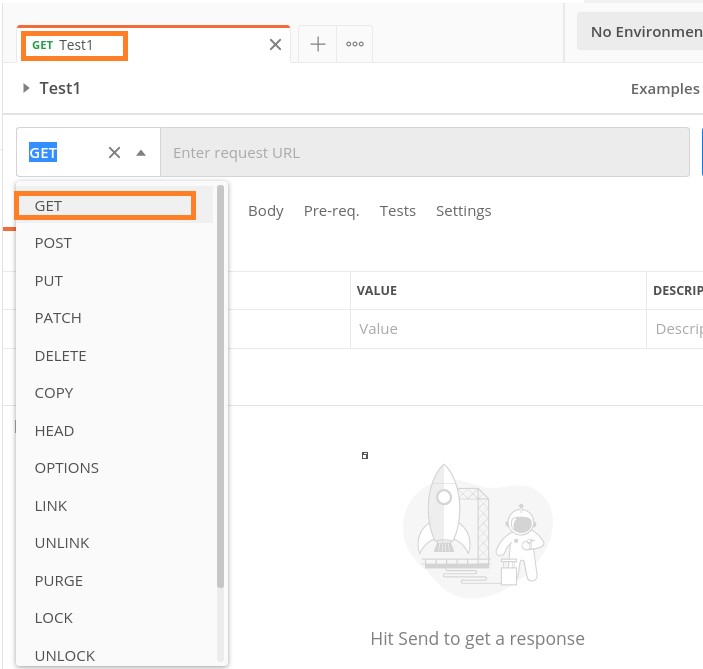
## Create a DELETE Request

Follow the steps given below to create a DELETE request in Postman successfully:

**Step 1**: Click on the **New** menu from the Postman application. The **Create New** pop-up comes up. Then, click on the Request link.



**Step 2**: **SAVE REQUEST** pop-up comes up. Enter the Request name then click on **Save**.



**Step 3**: The Request name (Test1) gets reflected on the Request tab. We shall select the option **DELETE** from the HTTP request dropdown.

Then enter the URL - http://dummy.restapiexample.com/api/v1/delete/2 (endpoint for deleting the record of id 2) in the address bar.

Here, in the DELETE request, we have mentioned the id of the resource in the server which we want to delete in the URL.

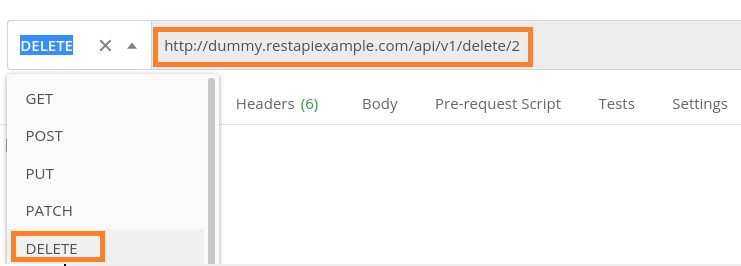
Postman

**Step**

**4**

:

Click on the Send button.



### Response

Once a request has been sent, we can see the Response code **200 OK** populated in the Response. This signifies a successful request and the request we have sent has been accepted by the server.

Also, information on the time consumed to complete the request (734 ms) and payload size (652 B) are populated. The Response shows the status as success. The record id 2 gets deleted from the server.



After deletion of the record with id 2, if we run the GET request on the endpoint: http://dummy.restapiexample.com/api/v1/employee/2, we shall receive 401 Unauthorized status code.

In the previous two lectures we have used APIs available in internet which is not a good approach so here we will create our own API by applying the following steps.

* So in step 1 we need to install Node js. After installation of the nodejs we have to cross check it either it is install or not so we will use the following two commands
  + node –version
  + npm –version
* After that install json server using the following command in cmd
  + npm istall -g json-server
* Now create a file with filename.json and put data in JSON format
* Now go to the created file location and open cmd with that location and start live server by using the following command
  + Json-server filename.json

That’s how our API is created now we can use it on POSTMAN.

**Understanding JSON**

Our response and request body is in JSON format so it is necessary to understand it

JSON stands for Javascript object notation

We use it for the communication between a client and server whenever communication occurs in form of request and response we use the JSON format.

**JSON DATATYPE**

**String:** {“name”:”Usman”}

**Number:** {“age”: 22}

**Object:** { “students”: [ { “name”: “Ali”, “age”: 21 } , { “name”:”Usman”, “age”:23 }]}

**Array:** { “age” : [20,30,40]}

**Null:** { “secondname”: null}

**Boolean:** {“status”: true}

**JSON PATH FINDER:**

To find JSON path we can take help from two websites

* [**https://jsonpathfinder.com/**](https://jsonpathfinder.com/) **(helps to find JSON path)**
* [**https://jsonpath.com/**](https://jsonpath.com/) **(validate that JSON path is correct or not)**

**RESPONSE VALIDATION**

We validate the response by validating the following.

* Status Code
* Headers
* Cookies
* Response time
* Response body

To validate the response it is necessary for us to know the desired output. To validate these things we write assertions using javascript function pm

**Syntax for Chai Assertion library:**

pm.test( ‘Test name’ , ()=>

{

//assertions;

}

)

**Testing Status Code:**

//assertion for validating correct status code

pm.**test**('Status Code should be 200', () **=>**{

    pm.response.to.have.status(200);

});

//assertion for if you dont know the exact staus code sometime it may appear 200 or 201 it mostly use in put request

pm.**test**('Staus Code shobe 200 or 201', ()**=>**{

    pm.expect(pm.response.code).to.be.oneOf([200,201]);

});

//if the status code is in form of text

pm.**test**('Status code name has string', ()**=>**{

    pm.response.to.have.status('Created');

})

**Testing Headers**

//assertion for checking header is present or not

pm.**test**('Test for checking Content Type header is present', ()**=>**{

    pm.response.to.have.header('Content-Type')

});

//assertion for cheecking header having correct value or not

pm.**test**('Test for checking Content Type header having correct value', ()**=>**{

    pm.expect(pm.response.headers.**get**('Content-Type')).to.eql('application/json; charset=utf-8');

});

**Testing Cookies**

//assertion for cookies are presentt or not

pm.**test**('language cookie is present ', ()**=>**{

    pm.expect(pm.cookies.**has**('Language')).to.be.true;

});

//assertion for cookie value is correct or not

pm.**test**('Cookie value is correct', ()**=>**{

    pm.expect(pm.cookies.**get**('Language')).to.eql('English');

});

**Response Time:**

// assertion for response time

pm.**test**('Response time is less than 200', ()**=>**{

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

});

**Response Body**

//assertion for validating the response body has correct data type or not

const jsonData **=** pm.response.json();

pm.**test**('Test case for data type of json body', ()**=>**{

    pm.expect(jsonData.name).to .be.a('string');

    pm.expect(jsonData.id).to.be.a('string');

    pm.expect(jsonData.courses).to .be.an('array');

    pm.expect(jsonData).to .be.an('object');

});

//assertion for checking the values in an array

pm.**test**('Test case to valide the value of an array', ()**=>**{

    pm.expect(jsonData.courses).to.have.members(['python','appium']);

    pm.expect(jsonData.courses).to.include('python');

});

//assertion to check value in response body is correct

pm.**test**('Test case for validating the value of json body', ()**=>**{

    pm.expect(jsonData.name).to .eql('Kim');

    pm.expect(jsonData.id).to.eql('2');

    pm.expect(jsonData.courses[0]).to .eql('python');

});

var schema **=** {

  "$schema": "http://json-schema.org/draft-04/schema#",

  "type": "object",

  "properties": {

    "id": {

      "type": "string"

    },

    "name": {

      "type": "string"

    },

    "location": {

      "type": "string"

    },

    "phone": {

      "type": "string"

    },

    "courses": {

      "type": "array",

      "items": [

        {

          "type": "string"

        },

        {

          "type": "string"

        }

      ]

    }

  },

  "required": [

    "id",

    "name",

    "location",

    "phone",

    "courses"

  ]

};

pm.**test**('validate json schema', ()**=>**{

    pm.expect(tv4.validate(jsonData, schema)).to.be.true;

});

**Scripts**

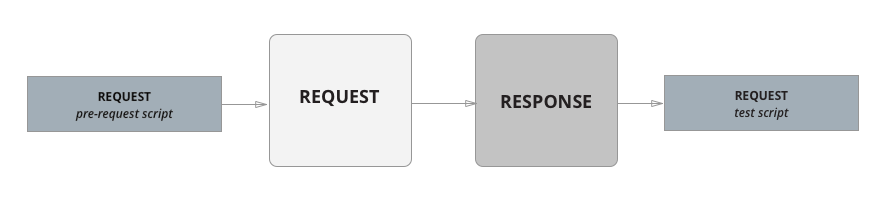
There are two types of scripts

* Pre request script
* Test Script (which we have already seen in previous lecture.

## Execution order of scripts

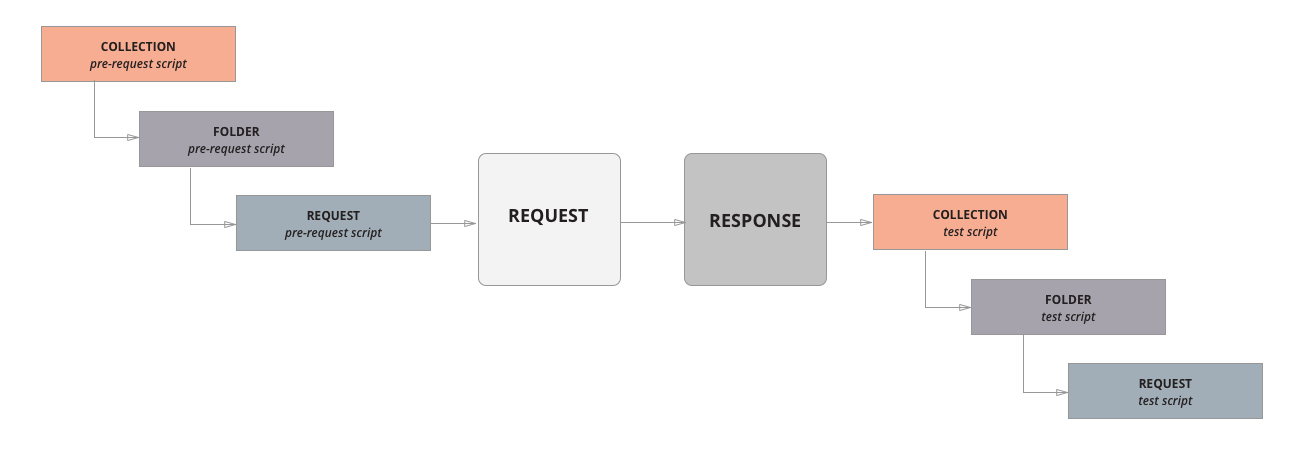
In Postman, the script execution order for a single request looks like this:

* A pre-request script associated with a request will execute before the request is sent
* A test script associated with a request will execute after the request is sent



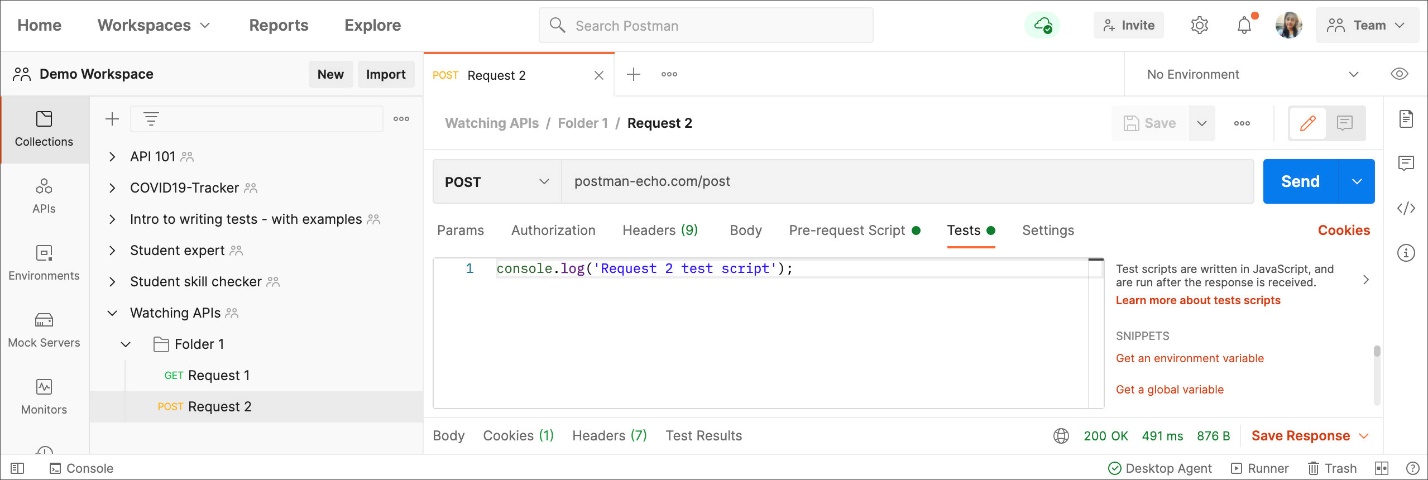
For every request in a collection, scripts will execute in the following order:

* A pre-request script associated with a collection will run prior to every request in the collection.
* A pre-request script associated with a folder will run prior to every direct child request in the folder.
* A test script associated with a collection will run after every request in the collection.
* A test script associated with a folder will run after every direct child request in the folder.

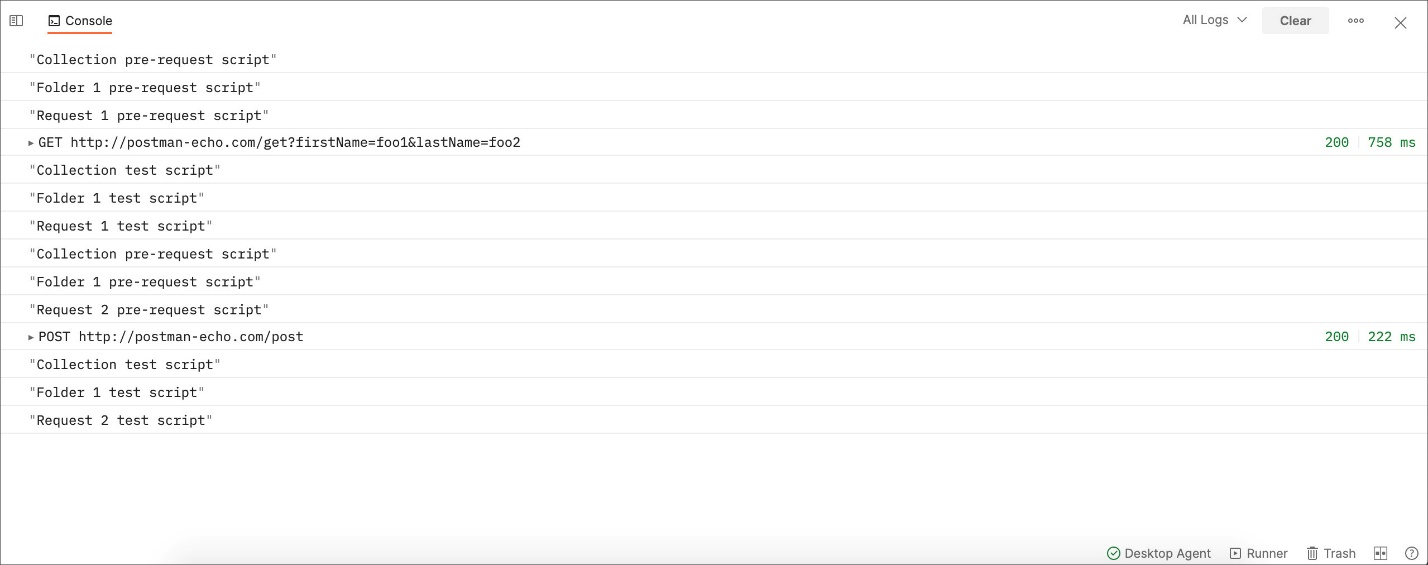


For every request in a collection, the scripts will always run according to the same hierarchy. Collection-level scripts (if any) will run first, then folder-level scripts (if any), and then request-level scripts (if any). Note that this order of execution applies to both pre-request and test scripts.

For example, imagine you had the following collection structured with a single folder and two requests within the folder.



If you created log statements in the pre-request and test script sections for the collection, folder, and requests, the execution order is returned in the [Postman Console](https://learning.postman.com/docs/sending-requests/troubleshooting-api-requests/).

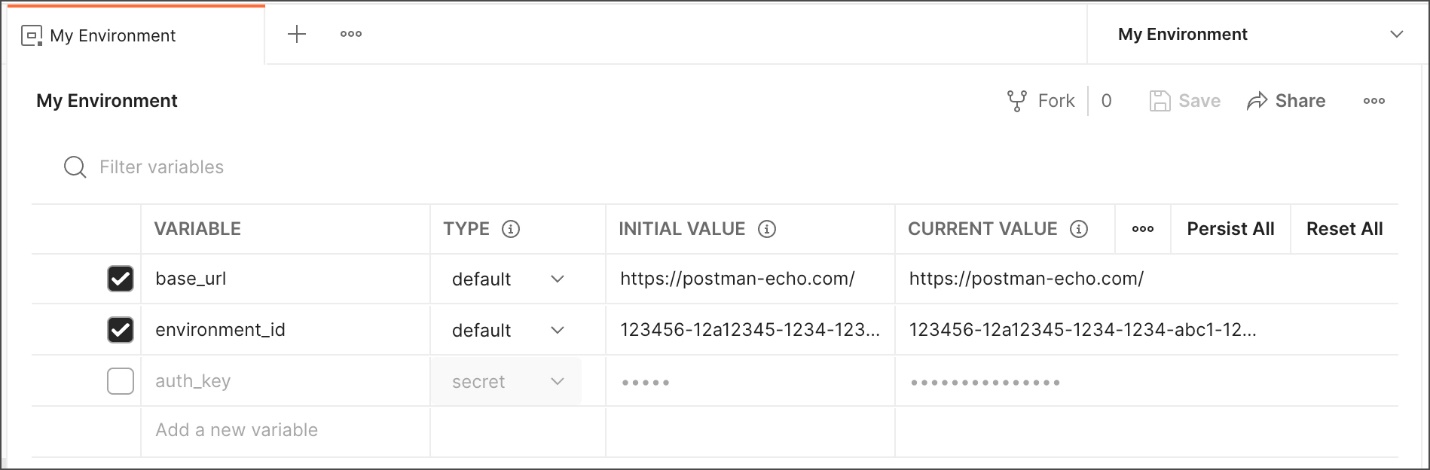


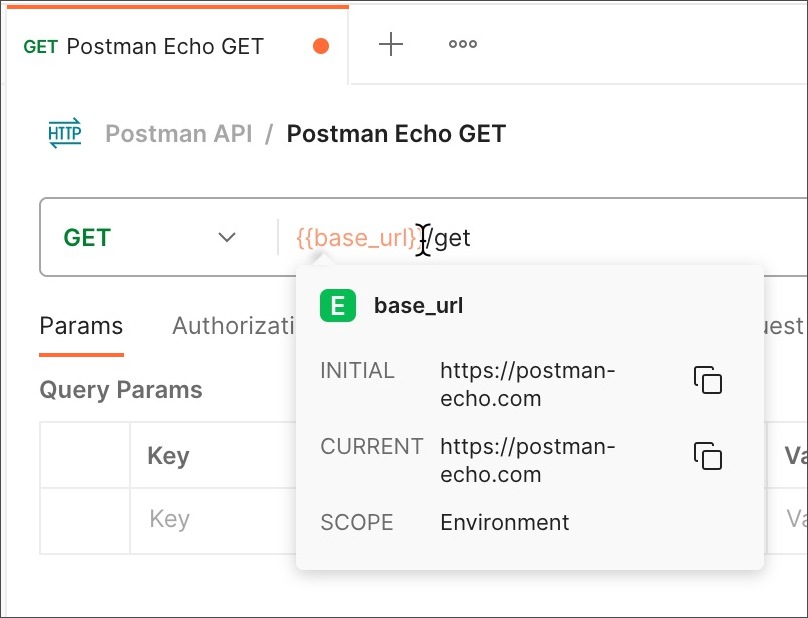
**Variables in Postman**

A variable is a symbolic representation of data that enables you to access a value without having to enter it manually wherever you need it. This can be useful if you are using the same values in multiple places. Variables make your requests more flexible and readable, by abstracting the detail away.

For example, if you have the same URL in more than one request, but the URL might change later, you can store the URL in a variable base\_url and reference it in your requests using {{base\_url}}. If the URL changes, you can change the variable value and it will be reflected throughout your collection, wherever you've used the variable name.

The same principle applies to any part of your request where data is repeated. Whatever value is stored in the variable will be included wherever you've referenced the variable when your requests run. If the base URL value is https://postman-echo.com, and is listed as part of the request URL using {{base\_url}}/get, Postman will send the request to https://postman-echo.com/get.





Variables in Postman are key-value pairs. Each variable name represents its key, so referencing the variable name enables you to access its value.

You can use variables to pass data between requests and tests, for example if you are [chaining requests](https://www.postman.com/postman/workspace/postman-team-collections/collection/1559645-81122f8b-5e07-4760-9504-f4387f45d2bc) in a collection.

You can use environments to group sets of variables together and share them with collaborators, for example if you use one set of config details for your production server and another for testing. See [Managing environments](https://learning.postman.com/docs/sending-requests/managing-environments/) for more on how you can incorporate environments into your team workflows.

## Variable scopes

Postman supports variables at different scopes, allowing you to tailor your processing to a variety of development, testing, and collaboration tasks. Scopes in Postman relate to the different contexts that your requests run in, and different variable scopes are suited to different tasks.

In order from broadest to narrowest, these scopes are: global, collection, environment, data, and local.

* **Global variables** enable you to access data between collections, requests, test scripts, and environments. Global variables are available throughout a [workspace](https://learning.postman.com/docs/collaborating-in-postman/using-workspaces/creating-workspaces/). Since global variables have the broadest scope available in Postman, they're well-suited for testing and prototyping. In later development phases, use more specific scopes.
* **Collection variables** are available throughout the requests in a collection and are independent of environments. Collection variables don't change based on the selected environment. Collection variables are suitable if you're using a single environment, for example for auth or URL details.
* **Environment variables** enable you to scope your work to different environments, for example local development versus testing or production. One environment can be active at a time. If you have a single environment, using collection variables can be more efficient, but environments enable you to specify [role-based access levels](https://learning.postman.com/docs/sending-requests/managing-environments/#working-with-environments-as-a-team).
* **Data variables** come from external CSV and JSON files to define data sets you can use when running collections with [Newman](https://learning.postman.com/docs/collections/using-newman-cli/command-line-integration-with-newman/) or the [Collection Runner](https://learning.postman.com/docs/collections/running-collections/intro-to-collection-runs/). Data variables have current values, which don't persist beyond request or collection runs.
* **Local variables** are temporary variables that are accessed in your request scripts. Local variable values are scoped to a single request or collection run, and are no longer available when the run is complete. Local variables are suitable if you need a value to override all other variable scopes but don't want the value to persist once execution has ended.

### Defining variables in scripts

You can set variables programmatically in your request scripts.

| **Method** | **Use-case** | **Example** |
| --- | --- | --- |
| pm.globals | Use to define a global variable. | pm.globals.set("variable\_key", "variable\_value"); |
| pm.collectionVariables | Use to define a collection variable. | pm.collectionVariables.set("variable\_key", "variable\_value"); |
| pm.environment | Use to define an environment variable in the currently selected environment. | pm.environment.set("variable\_key", "variable\_value"); |
| pm.variables | Use to define a local variable. | pm.variables.set("variable\_key", "variable\_value"); |
| unset | You can use unset to remove a variable. | pm.environment.unset("variable\_key"); |