



# Postman

**tutorialspoint**

SIMPLY EASY LEARNING

[www.tutorialspoint.com](http://www.tutorialspoint.com)



<https://www.facebook.com/tutorialspointindia>



<https://twitter.com/tutorialspoint>

## About the Tutorial

---

Postman is an Application Programming Interface (API) testing tool. This tutorial shall provide you with a detailed understanding on Postman and its salient features. The tutorial contains a good amount of examples on all important topics in Postman.

## Audience

---

This tutorial is designed for professionals working in software testing who want to improve their knowledge on API testing.

## Prerequisites

---

Before going through this tutorial, you should have some insight on how an API works. Also, an understanding on API testing is needed to start with this tutorial.

## Copyright & Disclaimer

---

© Copyright 2021 by Tutorials Point (I) Pvt. Ltd.

All the content and graphics published in this e-book are the property of Tutorials Point (I) Pvt. Ltd. The user of this e-book is prohibited to reuse, retain, copy, distribute or republish any contents or a part of contents of this e-book in any manner without written consent of the publisher.

We strive to update the contents of our website and tutorials as timely and as precisely as possible, however, the contents may contain inaccuracies or errors. Tutorials Point (I) Pvt. Ltd. provides no guarantee regarding the accuracy, timeliness or completeness of our website or its contents including this tutorial. If you discover any errors on our website or in this tutorial, please notify us at [contact@tutorialspoint.com](mailto:contact@tutorialspoint.com)

## Table of Contents

---

About the Tutorial .....	i
Audience.....	i
Prerequisites.....	i
Copyright & Disclaimer .....	i
Table of Contents .....	ii
<b>1. Postman – Introduction .....</b>	<b>1</b>
Need of Postman .....	1
Working with Postman .....	2
<b>2. Postman — Environment Setup .....</b>	<b>7</b>
Standalone Application .....	7
Chrome Extension .....	11
<b>3. Postman — Environment Variables.....</b>	<b>16</b>
Create Environment .....	16
Environment Variables Scope.....	18
<b>4. Postman — Authorization.....</b>	<b>20</b>
Types of Authorization .....	22
Authorization at Collections .....	27
<b>5. Postman — Workflows .....</b>	<b>30</b>
<b>6. Postman — GET Requests .....</b>	<b>35</b>
Create a GET Request .....	35
<b>7. Postman — POST Requests .....</b>	<b>41</b>
Create a POST Request .....	41
<b>8. Postman — PUT Requests .....</b>	<b>47</b>
Create a PUT Request .....	48
<b>9. Postman — DELETE Requests.....</b>	<b>51</b>
Create a DELETE Request.....	51

<b>10. Postman — Create Tests for CRUD .....</b>	<b>54</b>
Tests in Postman .....	54
<b>11. Postman — Create Collections .....</b>	<b>57</b>
Create a New Collection .....	57
<b>12. Postman — Parameterize Requests .....</b>	<b>60</b>
Create a Parameter Request .....	60
<b>13. Postman — Collection Runner .....</b>	<b>63</b>
Execute Tests with Collection Runner .....	63
<b>14. Postman — Assertion.....</b>	<b>66</b>
Writing Assertions .....	66
Assertion for Object Verification .....	68
Assertion Types .....	69
<b>15. Postman — Mock Server.....</b>	<b>71</b>
Benefits of Mock Server .....	71
Mock Server Creation .....	71
Example Request .....	77
<b>16. Postman — Cookies .....</b>	<b>83</b>
Cookies Management.....	83
Access Cookies via Program .....	85
<b>17. Postman — Sessions .....</b>	<b>87</b>
<b>18. Postman — Newman Overview .....</b>	<b>89</b>
Newman Installation .....	89
<b>19. Postman — Run Collections using Newman .....</b>	<b>91</b>
Run Collections .....	91
Common command-line arguments for Newman.....	93
<b>20. Postman — OAuth 2.0 Authorization .....</b>	<b>95</b>

# 1. 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.

Thus, an API is a collection of agreements, functions, and tools that an application can provide to its users for successful communication with another application. We require an API whenever we access an application like checking news over the phone, Facebook, and so on.

Postman was designed in the year 2012 by software developer and entrepreneur Abhinav Asthana to make API development and testing straightforward. It is a tool for testing the software of an API. It can be used to design, document, verify, create, and change APIs.

Postman has the feature of sending and observing the Hypertext Transfer Protocol (HTTP) requests and responses. It has a graphical user interface (GUI) and can be used in platforms like Linux, Windows and Mac. It can build multiple HTTP requests – POST, PUT, GET, PATCH and translate them to code.

## Need of Postman

---

Postman has a huge user base and has become a very popular tool because of the reasons listed below:

- Postman comes without any licensing cost and is suitable for use for the teams with any capacity.
- Postman can be used very easily by just downloading it.
- Postman can be accessed very easily by logging into your own account after installation on the device.
- Postman allows easy maintenance of test suites with the help of collections. Users can make a collection of API calls which can have varied requests and sub-folders.
- Postman is capable of building multiple API calls like SOAP, REST, and HTTP.
- Postman can be used for test development by addition of checkpoints to HTTP response codes and other parameters.
- Postman can be integrated with the continuous integration and either continuous delivery or continuous deployment (CI/CD) pipeline.
- Postman can be integrated with Newman or Collection Runner which allows executing tests in much iteration. Thus we can avoid repeated tests.
- Postman has big community support.
- The Postman console allows debugging test steps.

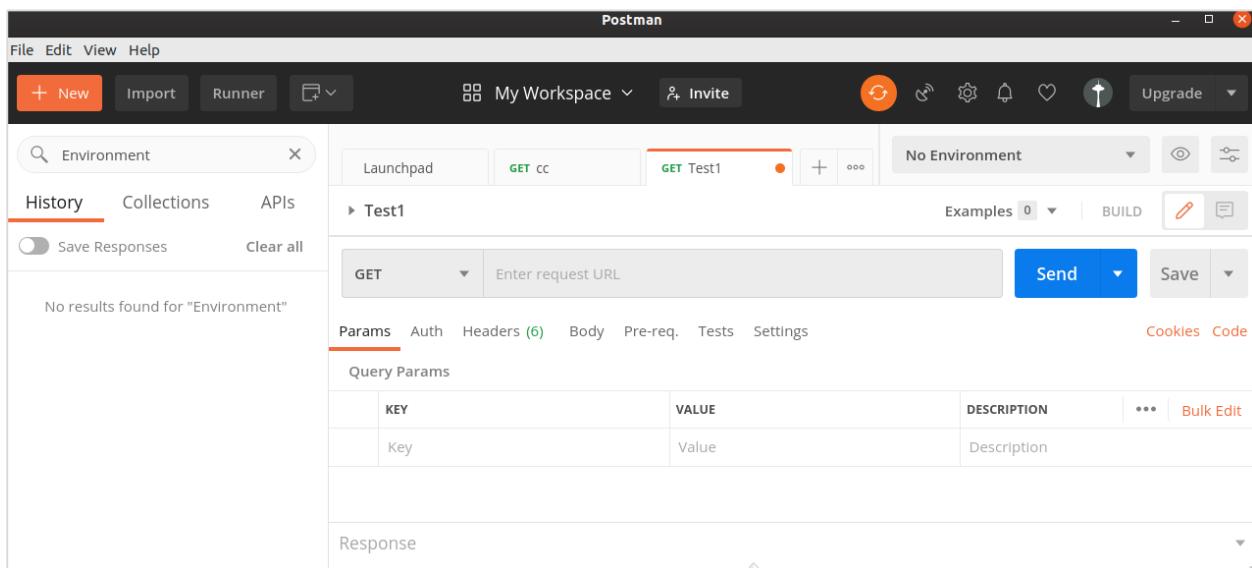
- With Postman, we can create more than one environment. Thus, a single collection can be used with various configurations.
- Postman gives the option to import/export Environments and Collections, enabling easy sharing of files.

## Working with Postman

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

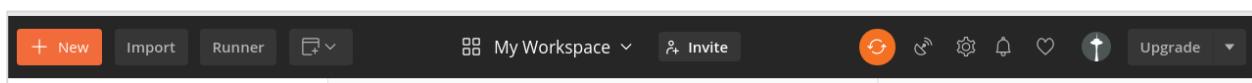
- Header
- Response
- Sidebar
- Builder

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

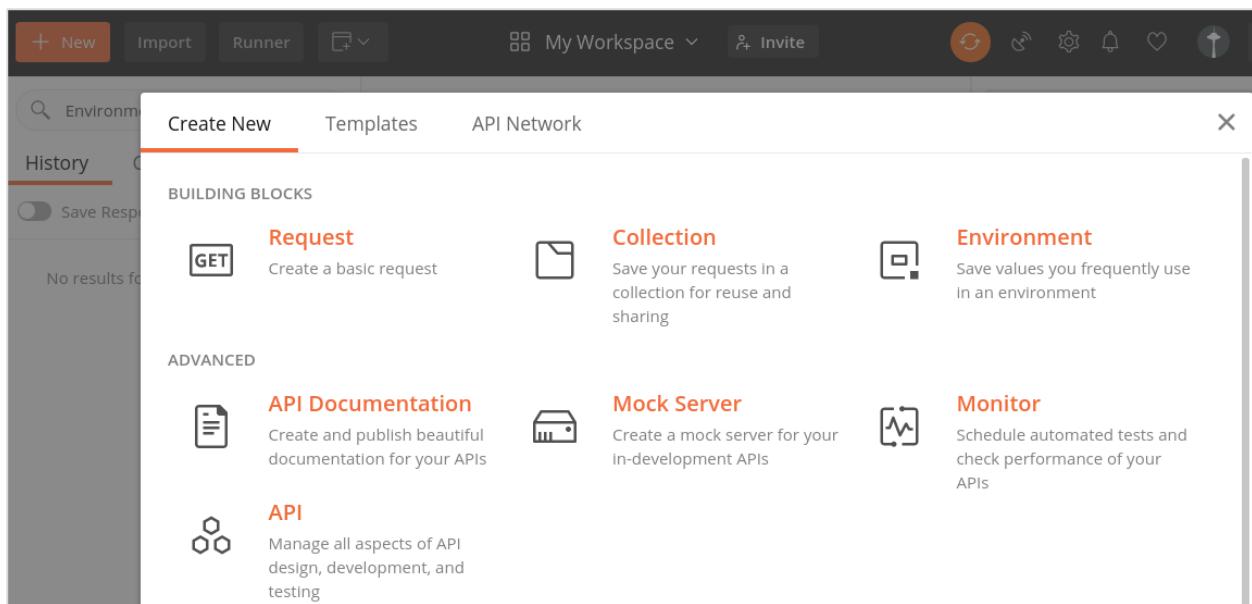


### Header

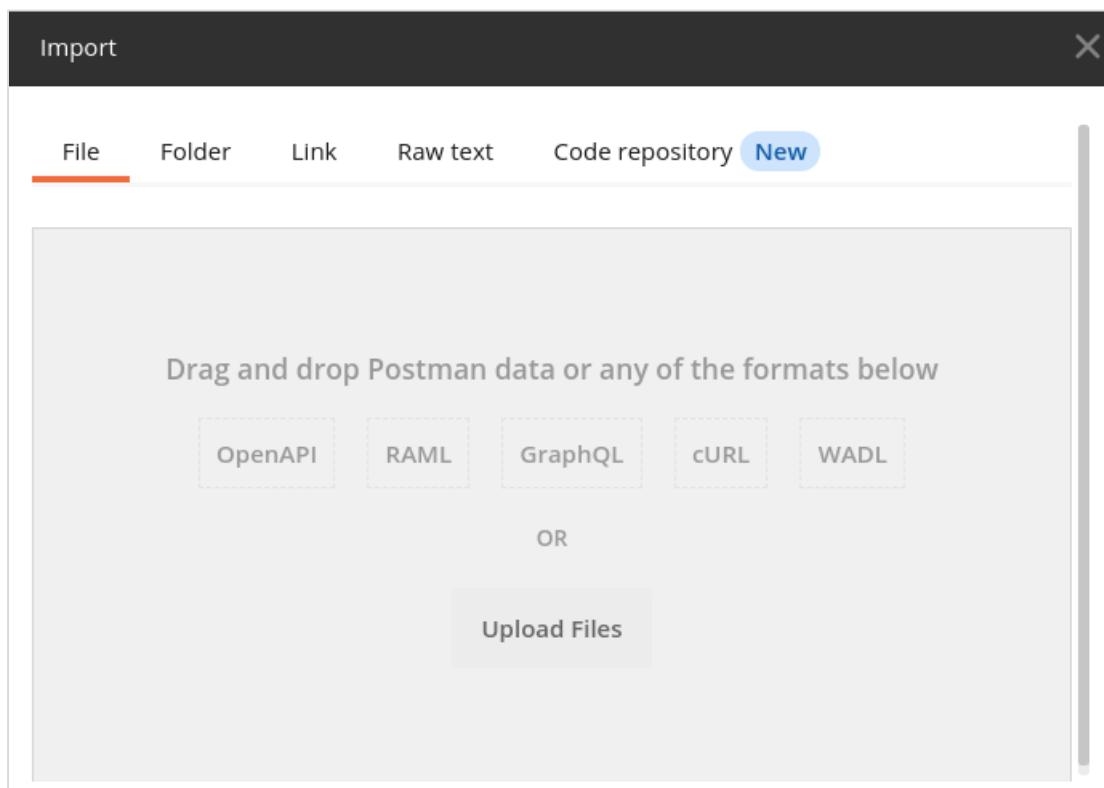
Postman consists of New, Import, Runner (used to execute tests with Collection Runner), Open New, Interceptor, Sync menus, and so on. It shows the workspace name – My Workspace along with the option for Invite for sharing it among teams.



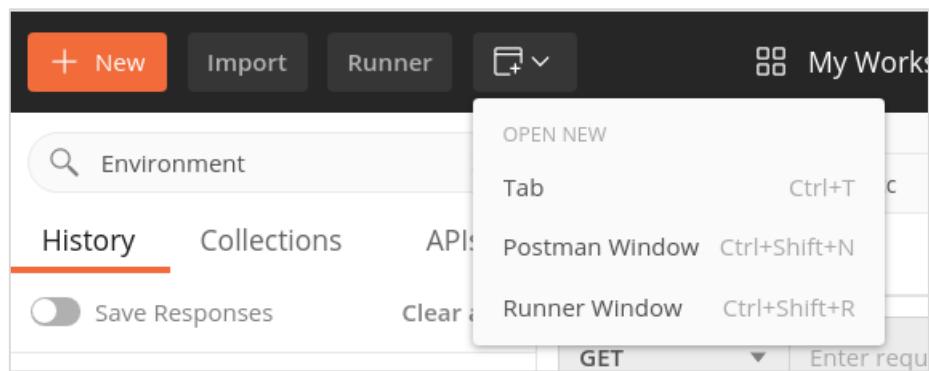
New menu is used to create a new Environment, Collection or request. The Import menu helps to import an Environment/Collection.



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.



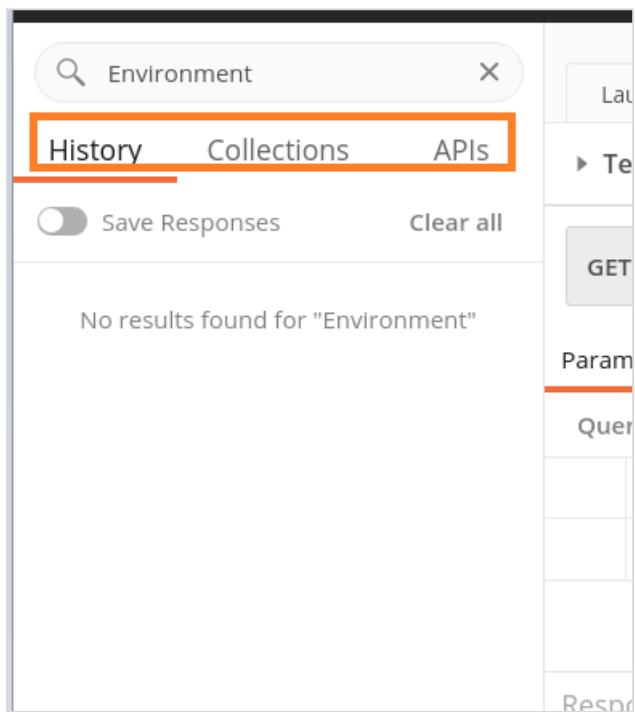
## Response

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

This screenshot shows the 'Test1' collection in Postman. At the top, it displays 'Launchpad', 'GET cc', 'GET Test1' (which is currently selected and highlighted in orange), and 'No Environment'. Below the tabs are sections for 'Examples' (0) and 'BUILD'. The main area is titled 'Test1'. It includes a 'Send' button and a 'GET' dropdown with a placeholder 'Enter request URL'. The 'Params' tab is active, showing a table for 'Query Params' with columns 'KEY', 'VALUE', 'DESCRIPTION', and an ellipsis button. The table has one row with 'Key' and 'Value' fields. At the bottom of the collection view, there's a large 'Response' section which is currently empty and highlighted with an orange border. In the center of the response section is a placeholder icon featuring a rocket and a spaceman.

## 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.

KEY	VALUE	DESCRIPTION	...	Bulk Edit
Key	Value	Description		

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.

The screenshot shows the Postman interface with a GET request to `https://jsonplaceholder.typicode.com/users`. The 'Headers' tab is active, displaying 7 items. Below the header section is a table for 'Query Params' with one row: Key (Value) and Value (Description).

KEY	VALUE	DESCRIPTION
Key	Value	Description

## 2. Postman — Environment Setup

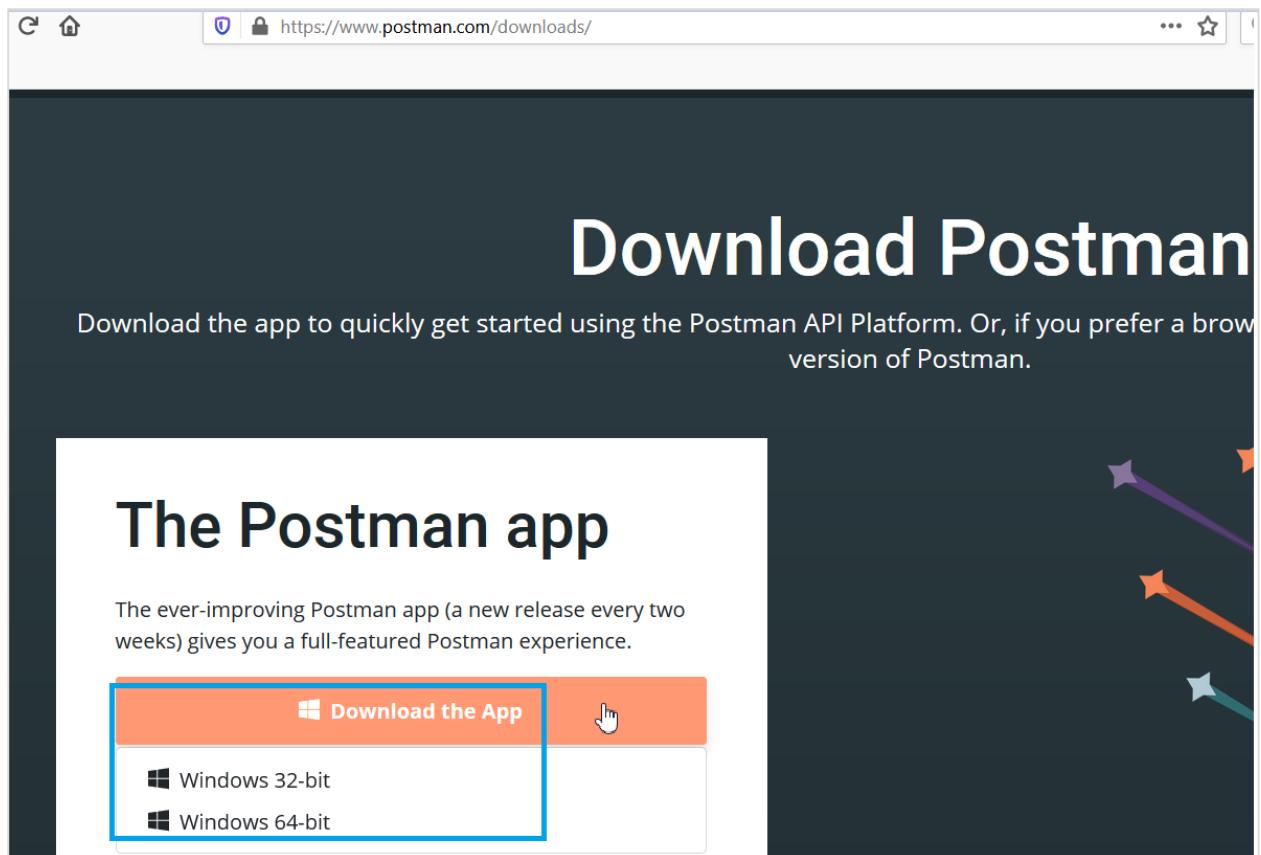
Postman can be installed in operating systems like Mac, Windows and Linux. It is basically an independent application which can be installed in the following ways:

- Postman can be installed from the Chrome Extension (will be available only in Chrome browser).
- It can be installed as a standalone application.

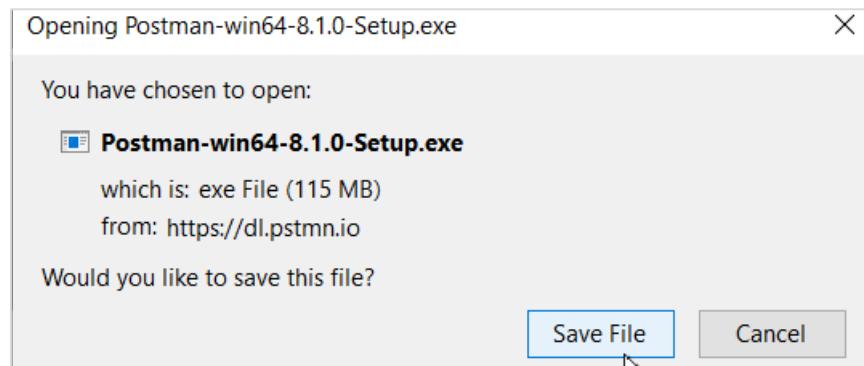
### Standalone Application

To download Postman as a standalone application in Windows, navigate to the following link <https://www.postman.com/downloads/>

Then, click on **Download the App button**. As per the configuration of the operating system, select either the **Windows 32-bit** or **Windows 64-bit** option.



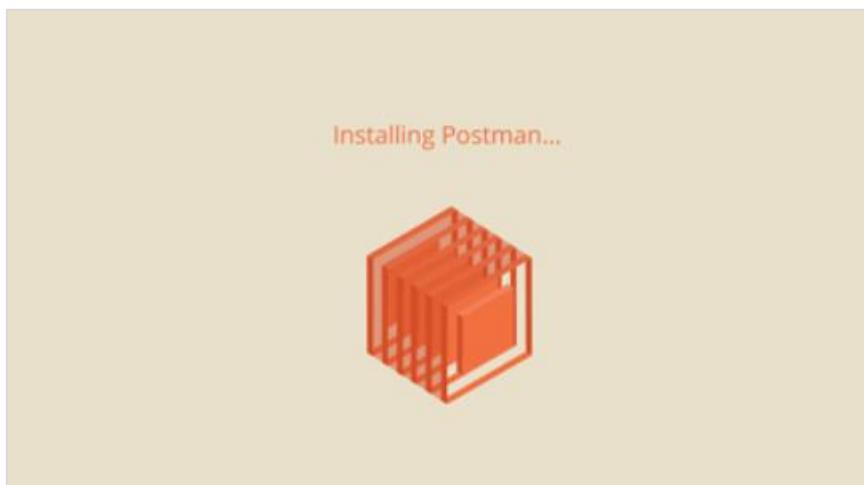
The pop-up to save the executable file gets opened. Click on **Save File**.



As the download is completed successfully, the executable file gets generated.

Name	Date modified	Type	Size
Postman-win64-8.1.0-Setup.exe	4/6/2021 2:19 PM	Application	118,136 KB

Double-click on it for installation.



After installation, the Postman landing screen opens. Also, we have to sign up here. There are two options to **create a Postman account**, which are as follows:

- Click on the Create free account.
- Use the Google Account.

The image shows the Postman account creation interface. On the left, there's a dark background with a white 'POSTMAN' logo and a circular icon containing a pen. Below it, the heading 'Why Sign Up?' is displayed next to a bulleted list of benefits:

- Organize all your API development within Postman Workspaces
- Sync your Postman data across devices
- Backup your data to the Postman cloud
- It's free!

On the right, the 'Create Postman Account' form is shown. It includes fields for Email, Username, and Password, along with checkboxes for marketing communications and staying signed in. There are links for Terms and Privacy Policy, and buttons for 'Create free account' and 'Sign up with Google'. A small 'or' is centered between the two main buttons.

Proceed with the steps of account creation and enter relevant details like name, role, and so on.

## Welcome to Postman!

Tell us a bit about yourself so we can help you get the most out of Postman.

What's your name?

[Change profile photo](#)

Which of these roles is closest to yours?

How do you plan to use Postman?

API documentation       Automated testing

Debugging and manual testing       Designing and mocking APIs

Monitoring       Publishing APIs

[Continue](#)

Finally, we shall land to the Start screen of Postman. The following screen will appear on your computer:

The screenshot shows the Postman application's main interface. At the top, there's a navigation bar with File, Edit, View, Help, + New, Import, Runner, and a workspace dropdown set to 'My Workspace'. Below the navigation is a toolbar with various icons like Refresh, Save, and Upgrade. The main workspace area is titled 'Test1' and contains a single GET request labeled 'GET Test1'. The request has a 'Params' tab selected, showing a table with one row: 'Key' (Value: Value). Other tabs include Authorization, Headers (6), Body, Pre-request Script, Tests, and Settings. To the right of the request table are buttons for Examples, BUILD, Send, and Save. Below the request table is a 'Query Params' section with a similar table structure. On the left side of the workspace, there's a message: 'You haven't sent any requests' followed by 'Any request you send in this workspace will appear here.' and a 'Show me how' button. At the bottom right of the workspace is a placeholder for a response with a rocket ship icon and the text 'Hit Send to get a response'.

## Chrome Extension

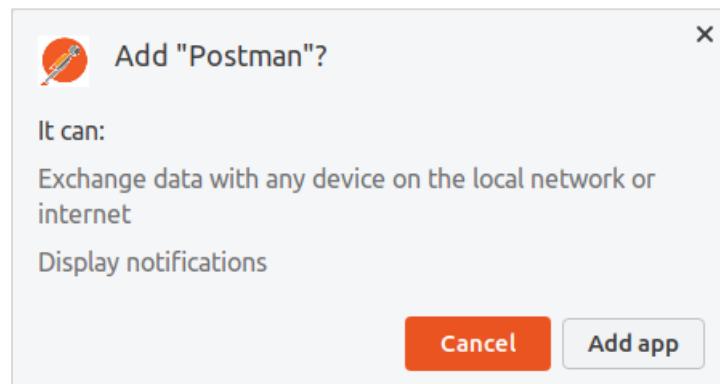
To download Postman as a Chrome browser extension, launch the below link in Chrome:

<https://chrome.google.com/webstore/detail/postman/fhbjgbiflinjbdfgehcddcbnccccdomop?>

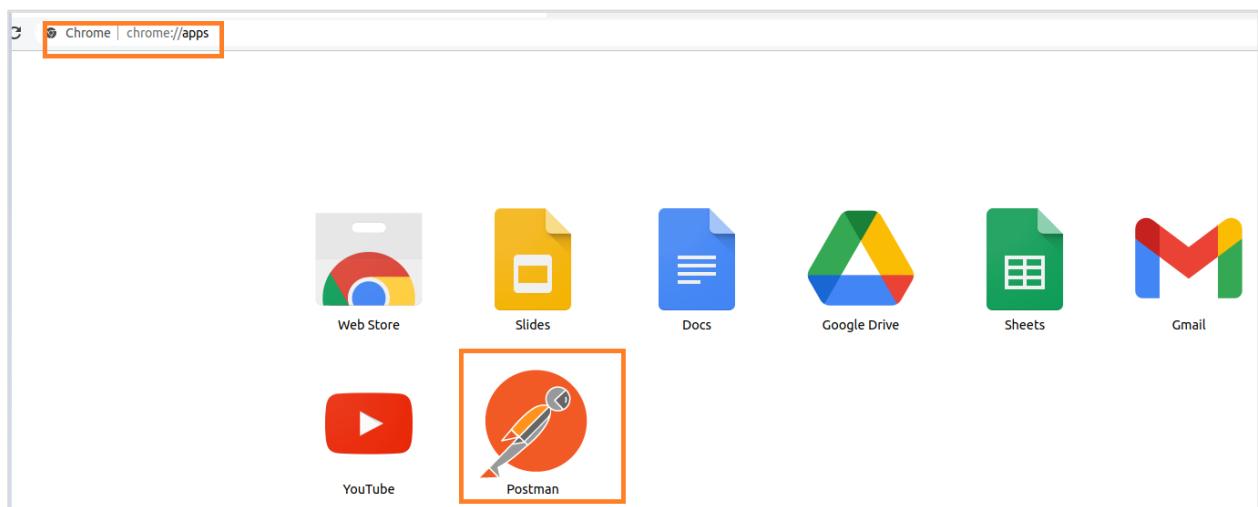
Then, click on **Add to Chrome**.

The screenshot shows the Chrome Web Store page for the Postman extension. The URL in the address bar is 'chrome.google.com/webstore/detail/postman/fhbjgbiflinjbdfgehcddcbnccccdomop?'. The page title is 'chrome web store'. Below the title, it says 'Home > Apps > Postman'. The main card for 'Postman' features the Postman logo (a red circle with a white rocket ship), the name 'Postman', 'Offered by: postman.com', a rating of 4.9 stars (8,966 reviews), a 'Extensions' link, and a user count of '3,000,000+ users'. It also indicates that the extension 'Runs offline'. To the right of the card is a large blue 'Add to Chrome' button, which is highlighted with a red rectangular box.

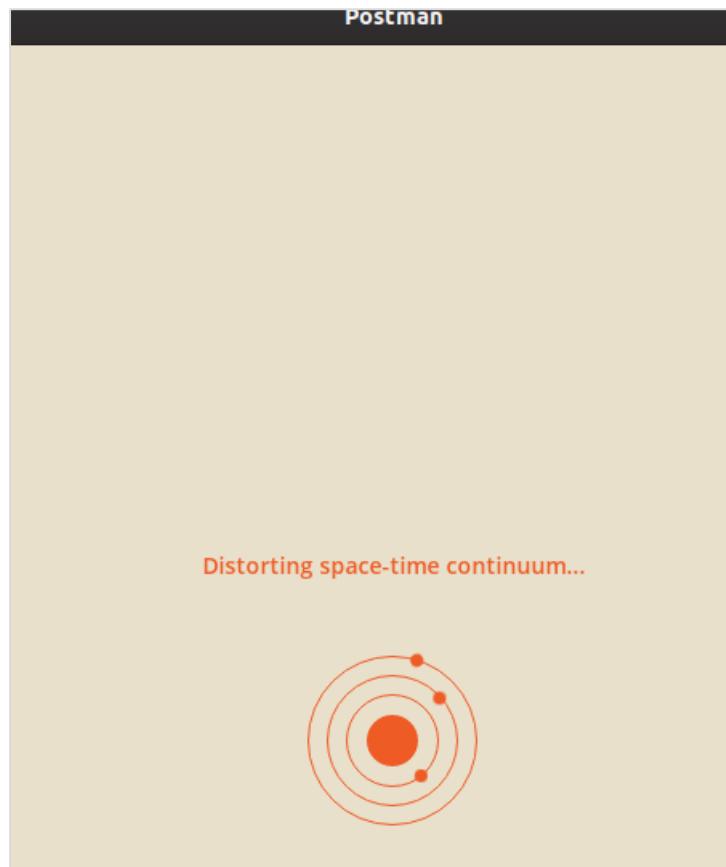
A pop-up gets displayed, click on the **Add app** button.



Chrome Apps page gets launched, along with the Postman icon. Next, we have to click on the Postman icon.

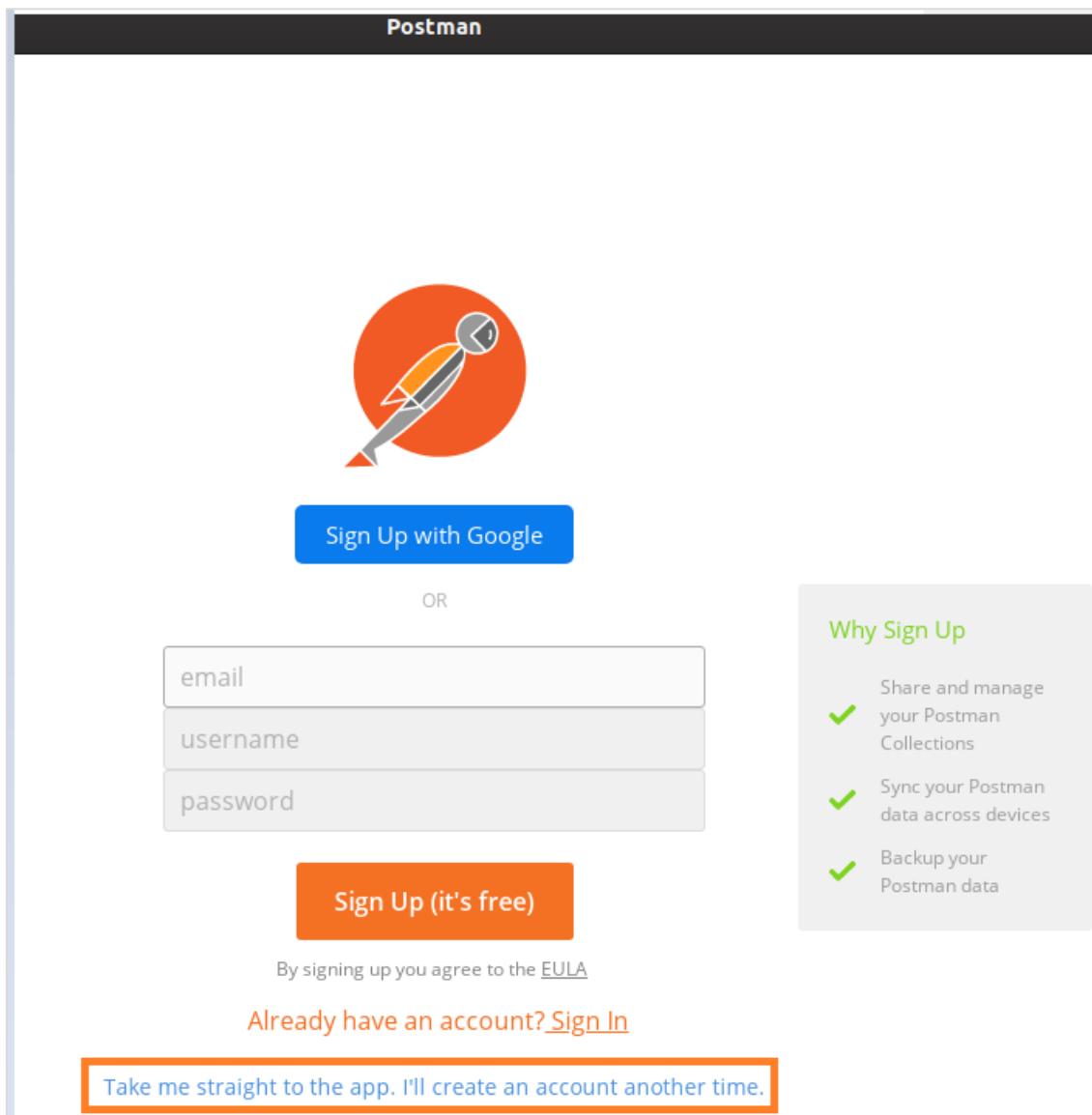


Installation of Postman kicks off.

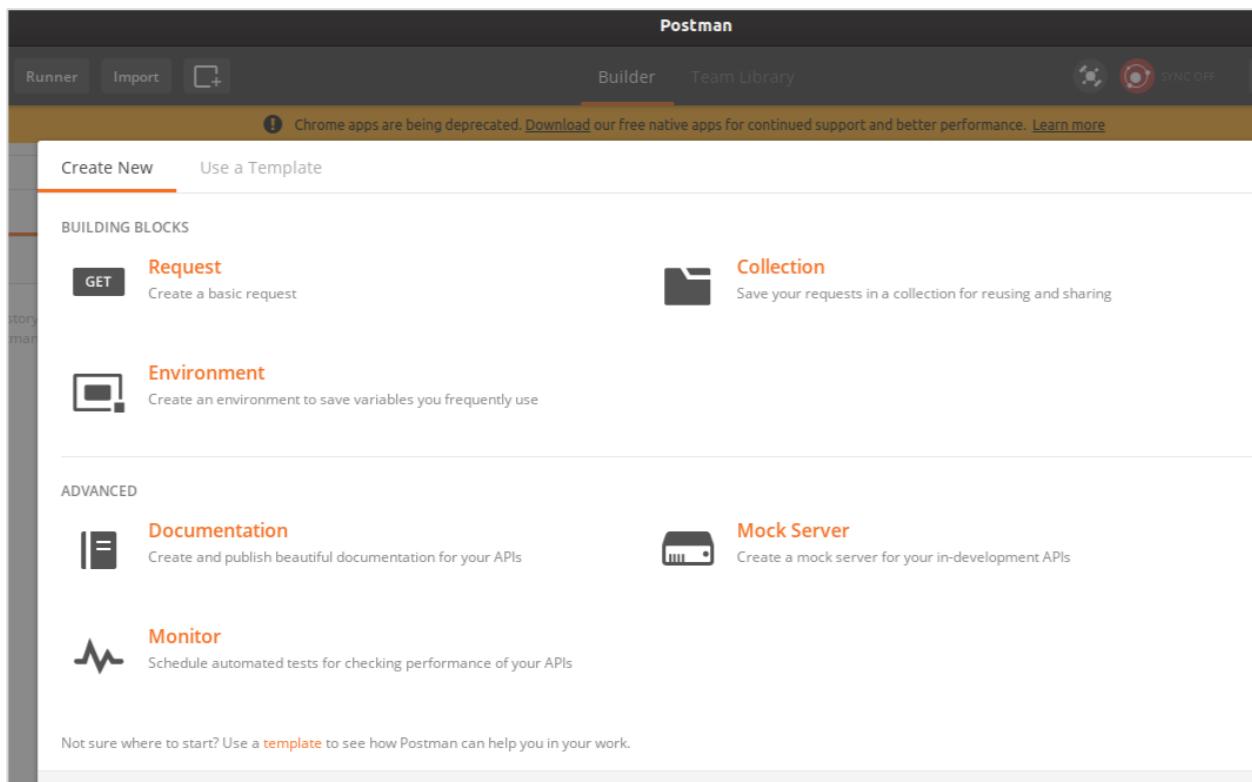


Once the installation is completed, the Postman registration page is opened. We can either proceed with the registration as explained previously (while installing Postman as a standalone application) or skip it by clicking on the link **Take me straight to the app**.

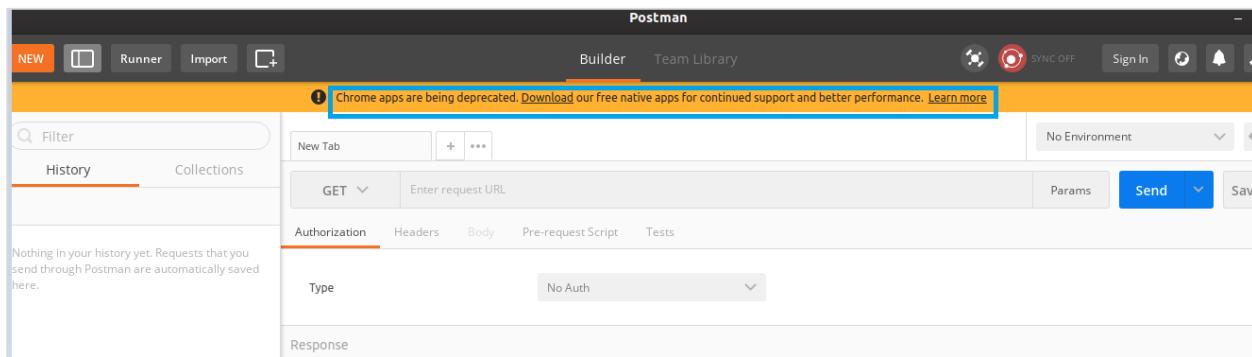
We can create an account later.



Registration is an important step as it enables access to user data from other machines. Next, the Postman welcome page opens up.



Once we close the pop-up and move to the following page, we get the message - **Chrome apps are being deprecated.**



It is always recommended to install Postman as a standalone application rather than a Chrome extension.

### 3. Postman — Environment Variables

Variables give the option to hold and repeat parameters in the requests, collections, scripts and so on. If we need to modify a value, we need to do it in only one place. Thus, the variables help to minimise the chance of errors and increase efficiency.

In Postman, an environment consists of a key-value pair. It helps to identify each request separately. As we create environments, we can modify key-value pairs and that will produce varied responses from the same request.

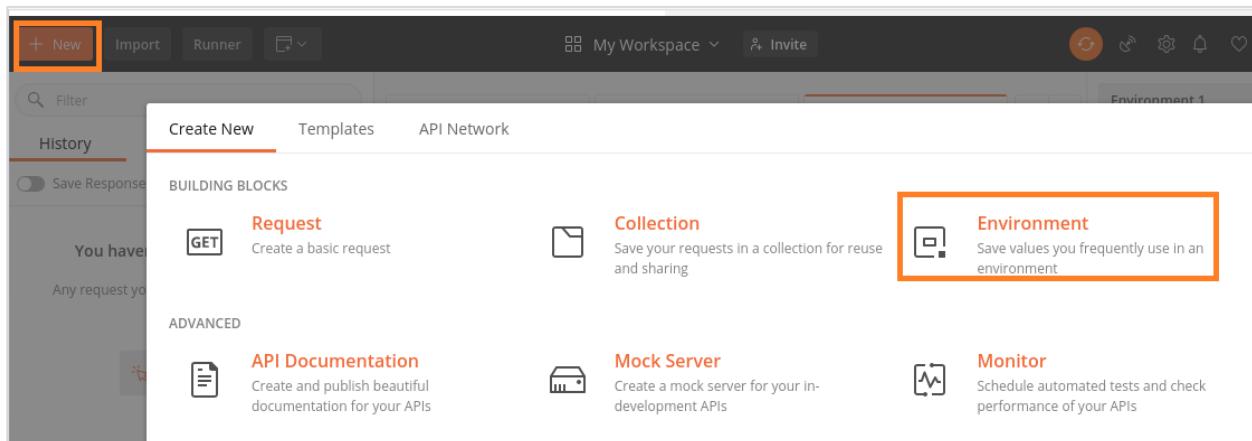
The key in the key-value pair in the environment is known as the Environment variable. There can be multiple environments and each of them can also have multiple variables. However, we can work with a single environment at one time.

In short, an environment allows the execution of requests and collections in a varied data set. We can create environments for production, testing and development. Each of these environments will have different parameters like URL, password, and so on.

#### Create Environment

Follow the steps given below to create an environment in Postman:

**Step 1:** Navigate to the **New menu** and then click on **Environment**.



**Step 2: MANAGE ENVIRONMENTS** pop-up gets opened. We have to enter the Environment name. Then, add a variable name and value.

Here, we have added the variable **u** and the value as <https://jsonplaceholder.typicode.com/users>. Close the pop-up.

MANAGE ENVIRONMENTS

Add Environment

ENV1

VARIABLE	INITIAL VALUE	CURRENT VALUE	...	Persist All	Reset All
<input checked="" type="checkbox"/> u	https://jsonplaceholder.typicode.com/	https://jsonplaceholder.typicode.com/users			
Add a new variable					

**Step 3:** The new Environment (ENV1) gets reflected as one of the items in the No Environment dropdown.

My Workspace ▾ + Invite

Launchpad GET cc GET Test1

No Environment

No Environment

**ENV1**

Environment\_Test

GET {{u}}

**Step 4:** Select the **ENV1 environment** and enter **{{u}}** in the address bar. To utilise an Environment variable in a request, we have to enclose it with double curly braces (**{{{<Environment variable name>}}}**).

**Step 5:** Then, click on **Send**. This variable can be used instead of the actual URL. We have received the Response code **200 OK** (meaning the request is successful).

The screenshot shows the Postman application interface. At the top, there are tabs for '+ New', 'Import', 'Runner', and 'My Workspace'. Below the tabs, there's a search bar for 'Environment' and buttons for 'History', 'Collections', and 'APIs'. A 'Save Responses' toggle and a 'Clear all' button are also present. The main workspace shows a 'Test1' collection with a 'Launchpad' tab and a 'GET Test1' request. The 'ENV1' environment is selected, indicated by an orange box around the 'ENV1' button in the top right. The request method is 'GET' and the URL contains a placeholder '{{u}}', which is highlighted with an orange box. To the right of the URL is a 'Send' button, also highlighted with an orange box. Below the URL, the 'Params' tab is selected, showing a single parameter 'Key' with 'Value' and 'Description' columns. The 'Body' tab is selected, displaying a JSON response with an id of 1. The response body is:

```

1 [ 
2   {
3     "id": 1,
4     "name": "Leanne Graham",
5     "username": "Bret",
6     "email": "Sincere@april.biz",
7     "address": {
8       "street": "Kulas Light",
9       "suite": "Apt. 556",
10      "city": "Gwenborough",
11      "zipcode": "92998-3874",
12      "geo": {
13        "lat": "-37.3159",
14        "lng": "81.1496"
15      }
16    }
17 ]
  
```

The status bar at the bottom indicates 'Status: 200 OK', 'Time: 162 ms', and 'Size: 6.6 KB'. There are 'Save Response' and 'Bulk Edit' buttons to the right.

## Environment Variables Scope

The scope of an Environment variable is within the environment for which it is created. This means it has a local scope confined to that environment. If we select another environment, and try to access the same Environment variable, we shall get an error.

In this chapter, we have created an Environment variable **u** within the **ENV1 environment** and on sending a GET request, we got the desired response.

However, if we try to use the same Environment variable **u** from another Environment, say **Environment\_Test**, we will receive errors.

The following screen shows the error, which we may get if we use the same Environment variable **u** from another environment:

The screenshot shows the Postman application interface. At the top, there's a navigation bar with 'New', 'Import', 'Runner', and other workspace-related options. The main area displays a 'Test1' collection with a single 'GET' request. The URL field contains the placeholder '{{u}}'. A red box highlights the 'Environment' tab in the top right corner of the interface. In the 'Params' tab of the request configuration, there's a table with one row:

KEY	VALUE	DESCRIPTION
Key	Value	Description

The 'Response' section shows a placeholder icon and the message 'Could not send request'. A red box highlights this message. Below it, an error message is displayed in a rounded rectangle: 'Error: getaddrinfo ENOTFOUND {{u}} | View In Console'. The bottom right corner of the interface has a 'Send' button.

## 4. Postman — Authorization

In Postman, authorization is done to verify the eligibility of a user to access a resource in the server. There could be multiple APIs in a project, but their access can be restricted only for certain authorized users.

The process of authorization is applied for the APIs which are required to be secured. This authorization is done for identification and to verify, if the user is entitled to access a server resource.

This is done within the **Authorization tab** in Postman, as shown below:

The screenshot shows the Postman interface with the 'Authorization' tab selected. The URL is set to `http://dummy.restapitutorial.com/api/v1/delete/2`. The 'Authorization' tab is highlighted with an orange border. Below it, there is a 'TYPE' dropdown menu with the option 'Inherit auth from parent' selected. A note below the dropdown states: 'The authorization header will be automatically generated when you send the request. [Learn more about authorization](#)'.

In the **TYPE** dropdown, there are various types of Authorization options, which are as shown below:

The screenshot shows the Postman interface with the 'Authorization' tab selected. A dropdown menu titled 'TYPE' is open, listing various authentication methods: Inherit auth from parent, No Auth, API Key, Bearer Token, Basic Auth, Digest Auth, OAuth 1.0, OAuth 2.0, Hawk Authentication, AWS Signature, NTLM Authentication [Beta], and Akamai EdgeGrid.

Let us now create a POST request with the APIs from GitHub Developer having an endpoint <https://www.api.github.com/user/repos>. In the Postman, click the Body tab and select the option raw and then choose the JSON format.

Add the below request body:

```
{
  "name" : "Tutorialspoint"
}
```

Then, click on **Send**.

The screenshot shows a POST request to <https://api.github.com/user/repos>. The 'Body' tab is selected, and the 'raw' and 'JSON' dropdowns are highlighted. The request body is a JSON object: { "name": "Tutorialspoint" }. The response status is 401 Unauthorized.

The Response code obtained is **401 Unauthorized**. This means, we need to pass authorization to use this resource. To authorize, select any option from the TYPE dropdown within the Authorization tab.

## Types of Authorization

---

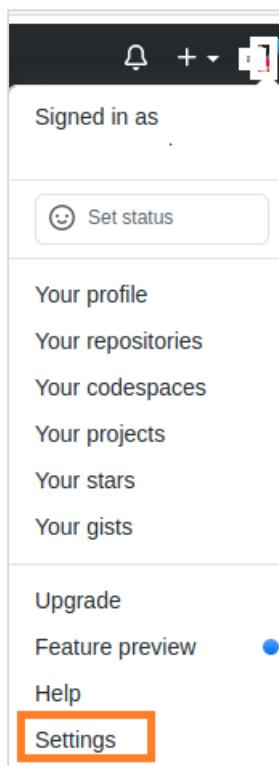
Let us discuss some of the important authorization types namely Bearer Token and Basic Authentication.

### Bearer Token

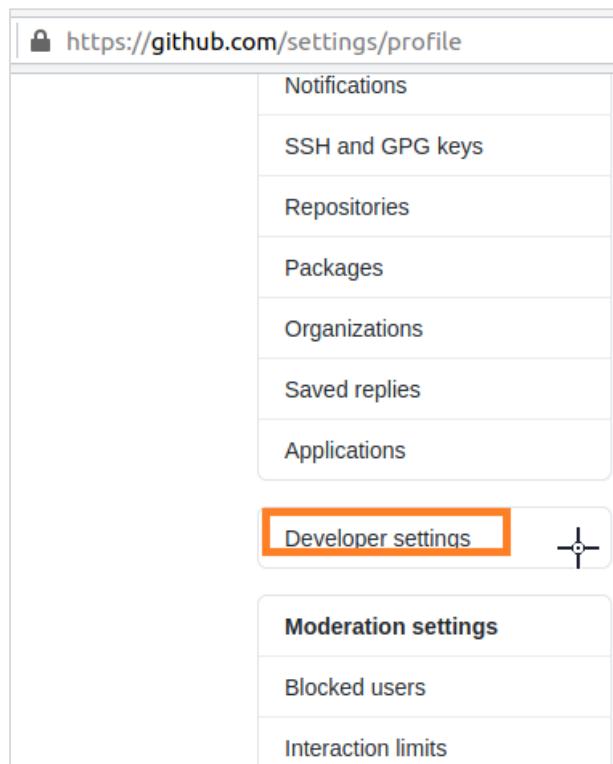
For Bearer Token Authorization, we have to choose the option Bearer Token from the TYPE dropdown. After this, the Token field gets displayed which needs to be provided in order to complete the Authorization.

**Step 1:** To get the Token for the GitHub API, first **login to the GitHub account** by clicking on the link given herewith: <https://github.com/login>.

**Step 2:** After logging in, click on the upper right corner of the screen and select the **Settings** option.



Now, select the option **Developer settings**.



Next, click on **Personal access tokens**.

The screenshot shows the 'Developer settings' page under 'Settings'. The left sidebar has three options: GitHub Apps, OAuth Apps, and Personal access tokens, with 'Personal access tokens' highlighted by an orange box. The main content area is titled 'GitHub Apps' and contains a sub-section 'Personal access tokens'. It includes a note about building GitHub Apps and links to 'Register a new GitHub App' and 'developer documentation'. A 'New GitHub App' button is also visible.

Now, click on the **Generate new token** button.

The screenshot shows the 'Personal access tokens' page under 'Developer settings'. The left sidebar has three options: GitHub Apps, OAuth Apps, and Personal access tokens. The main content area is titled 'Personal access tokens' and contains a note about tokens used for GitHub API access. A 'Generate new token' button is located in the top right corner of this section, highlighted with an orange box.

Provide a Note and select option repo. Then, click on **Generate Token** at the bottom of the page.

Finally, a Token gets generated.

New personal access token

Personal access tokens function like ordinary OAuth access tokens. They can be used instead of a password over HTTPS, or can be used to [authenticate to the API over Basic Authentication](#).

**Note**

Postman Testing

What's this token for?

**Select scopes**

Scopes define the access for personal tokens. [Read more about OAuth scopes.](#)

<input checked="" type="checkbox"/> repo	Full control of private repositories
<input checked="" type="checkbox"/> repo:status	Access commit status
<input checked="" type="checkbox"/> repo_deployment	Access deployment status
<input checked="" type="checkbox"/> public_repo	Access public repositories
<input checked="" type="checkbox"/> repo:invite	Access repository invitations
<input checked="" type="checkbox"/> security_events	Read and write security events

Copy the Token and paste it within the Token field under the Authorization tab in Postman. Then, click on **Send**.

Please note: Here, the Token is unique to a particular GitHub account and should not be shared.

## Response

The Response code is **201 Created** which means that the request is successful.

POST https://api.github.com/user/repos

Authorization

Bearer Token

Token

TYPE

Body

```

1 {
2   "id": 357300003,
3   "node_id": "MDEwOlJlcG9zaXRvcnkzNTczMDAwMDM=",
4   "name": "Tutorialspoint",
5   "full_name": "username:Tutorialspoint",
6   "private": false,
7   "owner": {
8     "login": "username",
9     "id": 52132518,
10    "node_id": "MDQyFollow link (ctrl + click)",
11    "avatar_url": "https://avatars.githubusercontent.com/u/52132518?v=4",

```

Status: 201 Created Time: 1148 ms

## Basic Authentication

For Basic Authentication Authorization, we have to choose the option **Basic Auth** from the **TYPE** dropdown, so that the Username and Password fields get displayed.

First we shall send a GET request for an endpoint (<https://postman-echo.com/basic-auth>) with the option **No Auth selected** from the TYPE dropdown.

Please note: The username for the above endpoint is postman and password is password.

The screenshot shows the Postman interface with a GET request to <https://postman-echo.com/basic-auth>. The Authorization tab is active, and the TYPE dropdown is set to "No Auth". A note indicates that this request does not use any authorization. The response section shows a status of 401 Unauthorized, time 227 ms, and size 293 B. The response body contains the text "1 Unauthorized".

The Response Code obtained is **401 Unauthorized**. This means that Authorization did not pass for this API.

Now, let us select the option **Basic Auth** as the Authorization type, following which the Username and Password fields get displayed.

Enter the postman for the Username and password for the Password field. Then, click on **Send**.

The screenshot shows the Postman interface with a GET request to <https://postman-echo.com/basic-auth>. The Authorization tab is active, and the TYPE dropdown is set to "Basic Auth". The response section shows a status of 200 OK, time 341 ms, and size 358 B. The response body contains the JSON object {"authenticated": true}.

The Response code obtained is now **200 OK**, which means that our request has been sent successfully.

### No Auth

We can also carry out Basic Authentication using the request Header. First, we have to choose the option as **No Auth** from the Authorization tab. Then in the Headers tab, we have to add a key: **value pair**.

We shall have the key as Authorization and the value is the username and password of the user in the format as **basic <encoded credential>**.

The endpoint used in our example is: <https://postman-echo.com/basic-auth>. To encode the username and password, we shall take the help of the third party application having the URL: <https://www.base64encode.org>

Please note: The username for our endpoint here is postman and password is password. Enter postman: password in the edit box and click on Encode. The encoded value gets populated at the bottom.

The screenshot shows a web browser window with the URL <https://www.base64encode.org> in the address bar. The main content area has a green header bar with the text "Encode to Base64 format". Below it, a sub-header says "Simply enter your data then push the encode button." A text input field contains the text "postman:password", which is highlighted with an orange border. Below the input field, there is explanatory text: "To encode binaries (like images, documents, etc.) use the file upload form a little further down on this page." There are two dropdown menus: "UTF-8" for "Destination character set" and "LF (Unix)" for "Destination newline separator". Below these are three checkboxes: "Encode each line separately (useful for when you have multiple entries)", "Split lines into 76 character wide chunks (useful for MIME)", and "Perform URL-safe encoding (uses Base64URL format)". The last checkbox is checked and highlighted with an orange border. A radio button labeled "Live mode OFF" is selected, with the note "Encodes in real-time as you type or paste (supports only the UTF-8 character set)". At the bottom is a large green "ENCODE" button with white text, which is also highlighted with an orange border. Below the button, the encoded output "cG9zdG1hbjpwYXNzd29yZA==" is displayed in a text area, also highlighted with an orange border.

We shall add the encoded Username and Password received as cG9zdG1hbjpwYXNzd29yZA== in the Header in the format - **basic cG9zdG1hbjpwYXNzd29yZA==**. Then, click on **Send**.

GET https://postman-echo.com/basic-auth

Headers (10)

Content-Length	<calculated when request is sent>
Host	<calculated when request is sent>
User-Agent	PostmanRuntime/7.26.8
Accept	/*
Accept-Encoding	gzip, deflate, br
Connection	keep-alive
Authorization	basic cG9zdG1hbjpwYXNzd29yZA==

Body

```
1 {  
2   "authenticated": true  
3 }
```

Status: 200 OK Time: 71 ms Size: 356 B

No Auth selected from the TYPE dropdown.

GET https://postman-echo.com/basic-auth

Authorization

TYPE

No Auth

This request does not use any authorization. [Learn more about authorization](#)

Status: 200 OK

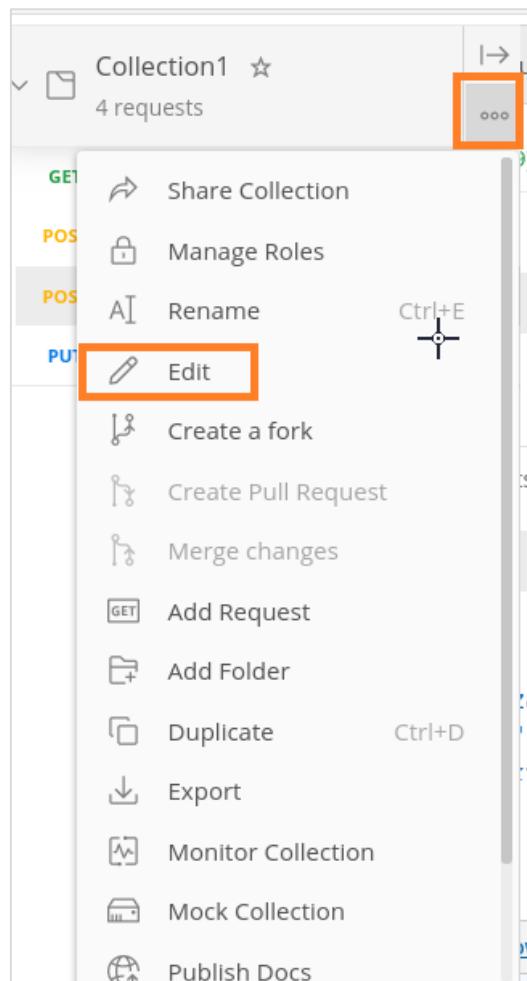
```
1 {  
2   "authenticated": true  
3 }
```

The Response code obtained is **200 OK**, which means that our request has been sent successfully.

## Authorization at Collections

To add Authorization for a Collection, following the steps given below:

**Step 1:** Click on the three dots beside the Collection name in Postman and select the option **Edit**.



**Step 2:** The EDIT COLLECTION pop-up comes up. Move to the Authorization tab and then select any option from the TYPE dropdown. Click on **Update**.

EDIT COLLECTION

Name  
Collection1

Description    **Authorization**    Pre-request Scripts    Tests    Variables

This authorization method will be used for every request in this collection. You can override this by specifying one in the request.

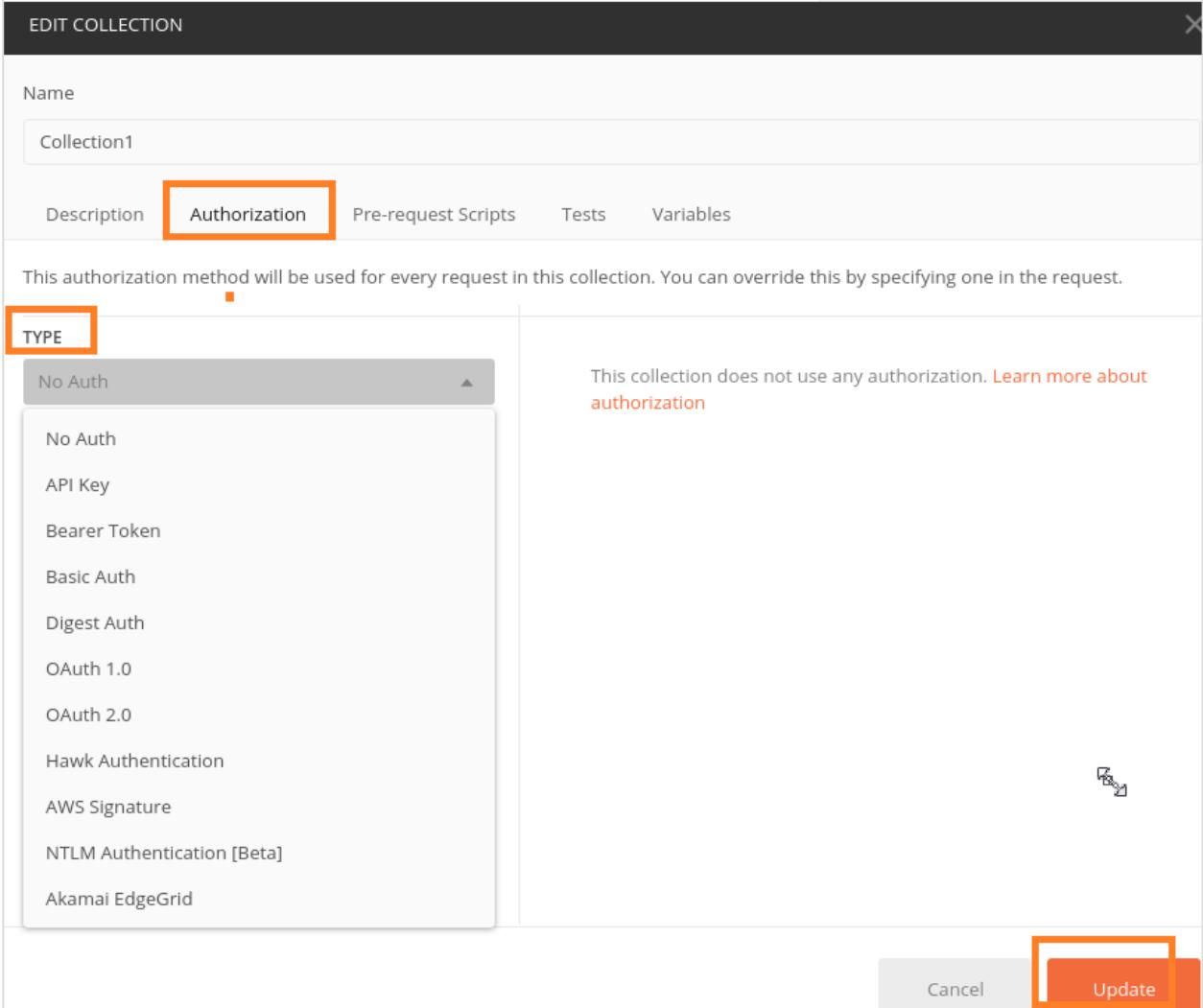
**TYPE**

No Auth

- No Auth
- API Key
- Bearer Token
- Basic Auth
- Digest Auth
- OAuth 1.0
- OAuth 2.0
- Hawk Authentication
- AWS Signature
- NTLM Authentication [Beta]
- Akamai EdgeGrid

This collection does not use any authorization. [Learn more about authorization](#)

Cancel    **Update**

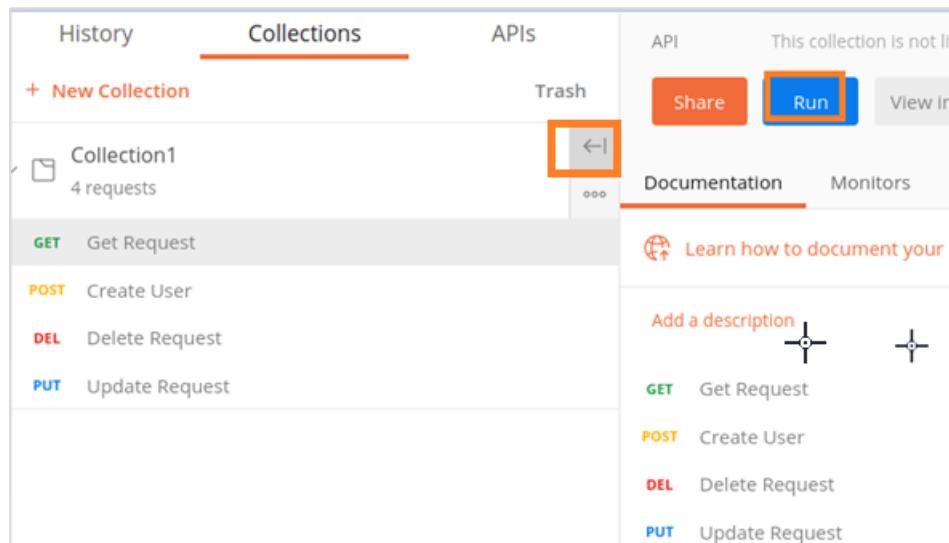


## 5. Postman — Workflows

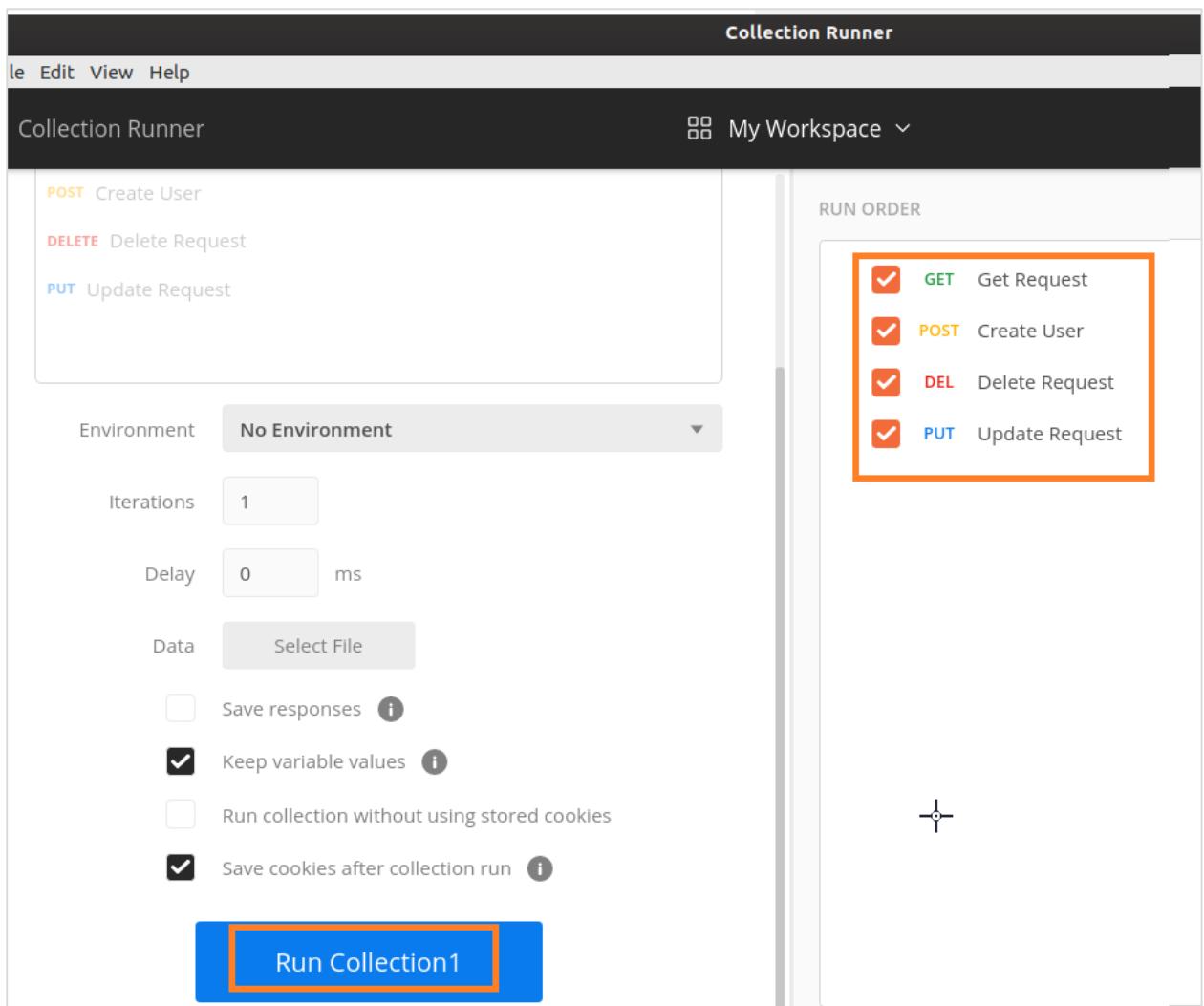
In a Postman Collection, the requests are executed in the order in which they appear. Every request is run first by the order of the folder followed by any request at the Collection root.

Let us create a Collection (Collection1) with four requests. The details on how to create a Collection is discussed in detail in the Chapter about Create Collections.

**Step 1:** Click on the arrow appearing to the right of the Collection name in the sidebar. Then, click on **Run button** to trigger execution of requests within the Collection.



**Step 2:** The Collection Runner pop-up comes up. The RUN ORDER section shows the order in which the requests shall get executed from top to the bottom. (GET->POST->DEL->PUT). Click on the **Run Collection1** button.



**Step 3:** Execution Results show the GET request executed first, followed by POST, then DEL and finally PUT, as mentioned in the RUN ORDER section in the Step 2.

The screenshot shows the Postman Run Summary for Collection1. It indicates 2 PASSED and 0 FAILED tests. The summary table includes columns for status, method, URL, test name, response code, duration, and size. The results are as follows:

Iteration 1	Method	URL	Test Name	Status	Duration	Size
	GET	http://dummy.restaplexa...	/ Get Request	200 OK	94 ms	21.248 KB
			Equality			
	POST	http://dummy.restaplexa...	/ Create User	200 OK	777 ms	659 B
			This request does not have any tests.			
	DELETE	http://dummy.restaplexa...	/ Delete Request	200 OK	708 ms	652 B
			Status Code is 200			
	PUT	http://dummy.restaplexa...	/ Update Request	200 OK	717 ms	652 B
			This request does not have any tests.			

If we want to change the order of the request to be executed (for example, first the Get Request shall run, followed by Create User, then Update Request and finally the Delete Request). We have to take the help of the function **postman.setNextRequest()**.

This function has the feature to state which request shall execute next. The request name to be executed next is passed as a parameter to this function. As per the workflow, we have to add this function either in the Tests or Pre-request Script tab under the endpoint address bar in Postman.

The syntax for execution of a request in Postman is as follows:

```
postman.setNextRequest("name of request")
```

## Implementation of a Workflow

The implementation of a workflow in Postman is explained below in a step wise manner:

**Step 1:** Add the below script under the Tests tab, for the request – Create User.

```
postman.setNextRequest("Update Request")
```

The following screen will appear:

**Step 2:** Add the below script under the Tests tab, for the request – Update Request.

```
postman.setNextRequest("Delete Request")
```

The following screen will appear:

The screenshot shows the Postman interface with a collection named 'Update Request'. In the 'Tests' tab, the following script is written:

```
1 postman.setNextRequest("Delete Request")
```

## Output of Workflow

Given below is the output of the workflow:

The screenshot shows the Postman Collection Runner for 'Collection1'. The 'Iteration 1' section displays the following requests:

- GET / Get Request**: Status 200 OK, 88 ms, 21.248 KB
- Equality**: Status 200 OK, 611 ms, 659 B
- POST / Create User**: Status 200 OK, 593 ms, 652 B
- PUT / Update Request**: Status 200 OK, 593 ms, 652 B
- DELETE / Delete Request**: Status 429 Too Many Requests, 546 ms, 1.181 KB (assertion failed: Status Code is 200 | AssertionError: expected false to be truthy)
- PUT / Update Request**: Status 429 Too Many Requests, 529 ms, 1.181 KB
- DELETE / Delete Request**: Status 429 Too Many Requests, 554 ms, 1.181 KB (assertion failed: Status Code is 200 | AssertionError: expected false to be truthy)

The 'PUT / Update Request' and 'DELETE / Delete Request' steps are highlighted with an orange box.

The output shows that Update Request and Delete Request are running in an infinite loop until we stop it by clicking the Stop Run button.

## Infinite Workflow Loop

If we want to stop the infinite Workflow loop via script, we have to add the below script for the request – Delete Request.

```
postman.setNextRequest(null)
```

The following screen will appear:

The screenshot shows the Postman interface with a collection named "Delete Request". The "Tests" tab is selected, displaying the following code:

```

1 tests["Status Code is 200"] = responseCode.code === 200
2 postman.setNextRequest(null)

```

Again run the same Collection and output shall be as follows:

The screenshot shows the Collection Runner interface with a collection named "Collection1". The summary indicates 1 PASSED and 1 FAILED request. The failed request, "DELETE Delete Request", is highlighted with a red circle. The details for each request are listed below:

Request Type	URL	Method	Status	Time	Size
GET	http://dummy.restaplexample.com/api/v1/index.htm	/ Get Request	200 OK	185 ms	21.248 KB
Equality					
POST	http://dummy.restaplexa...	/ Create User	200 OK	671 ms	659 B
This request does not have any tests.					
PUT	http://dummy.restaplexa...	/ Update Request	200 OK	627 ms	652 B
This request does not have any tests.					
DELETE	http://dummy.restaplexa...	/ Delete Request	429 Too Many Requests	672 ms	1.181 KB
Status Code Is 200   Assertion Error: expected false to be truthy					

The output shows the order of execution as Get Request, Create User, Update Request and finally Delete Request.

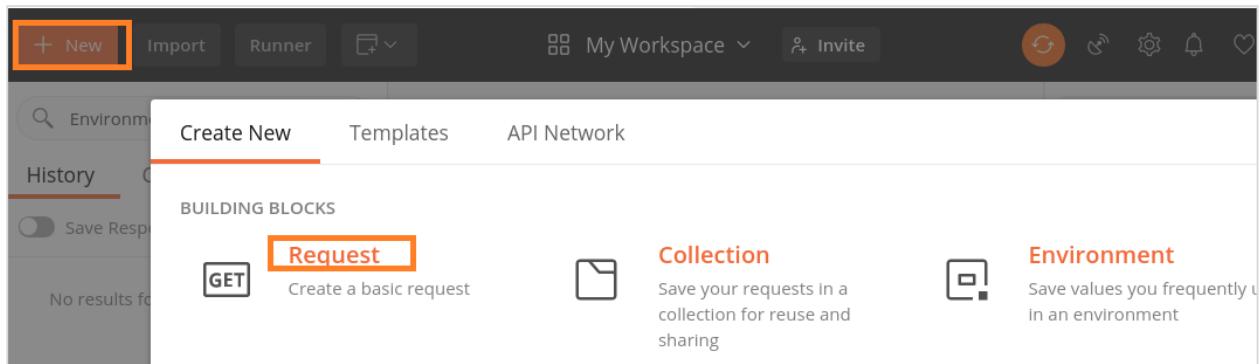
# 6. Postman — GET Requests

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.

SAVE REQUEST

Requests in Postman are saved in collections (a group of requests).  
[Learn more about creating collections](#)

Request name

Request description (Optional)

Make things easier for your teammates with a complete request description.

Descriptions support [Markdown](#)

Select a collection or folder to save to:

Cancel Save to FirstTest

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

The screenshot shows the Postman application interface. At the top, there is a header with the title 'Test1' (highlighted with an orange box), a close button ('X'), a plus sign ('+') button, and a three-dot ('...') button. To the right of the header is a button labeled 'No Environment'. Below the header, the word 'Test1' is followed by a 'Examples' link.

The main workspace has a toolbar on the left with various HTTP method buttons: GET (highlighted with an orange box), POST, PUT, PATCH, DELETE, COPY, HEAD, OPTIONS, LINK, UNLINK, PURGE, LOCK, and UNLOCK. To the right of the toolbar is a large input field labeled 'Enter request URL'. Below the URL input, there are tabs for 'Body', 'Pre-req.', 'Tests', and 'Settings'. A table row is visible under the 'Body' tab, with columns for 'NAME', 'VALUE', and 'DESCRIPTION'. The 'Value' column contains the text 'Value'.

In the center-right area, there is a small icon of a rocket launching from a platform, with a figure standing nearby. Below this icon is the text 'Hit Send to get a response'.

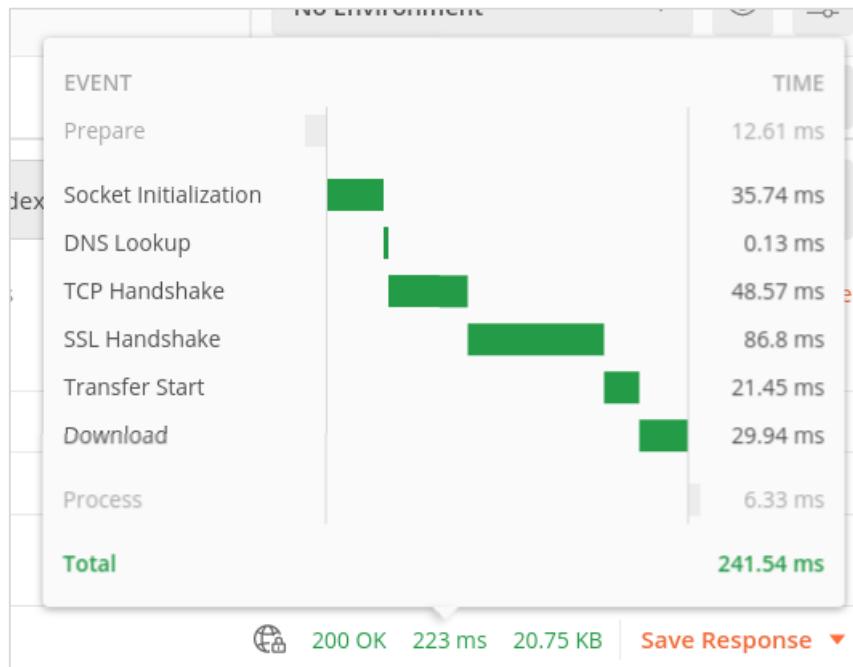
**Step 4:** Enter an URL - <https://www.tutorialspoint.com/index.htm> in the address bar and click on Send.

## Response

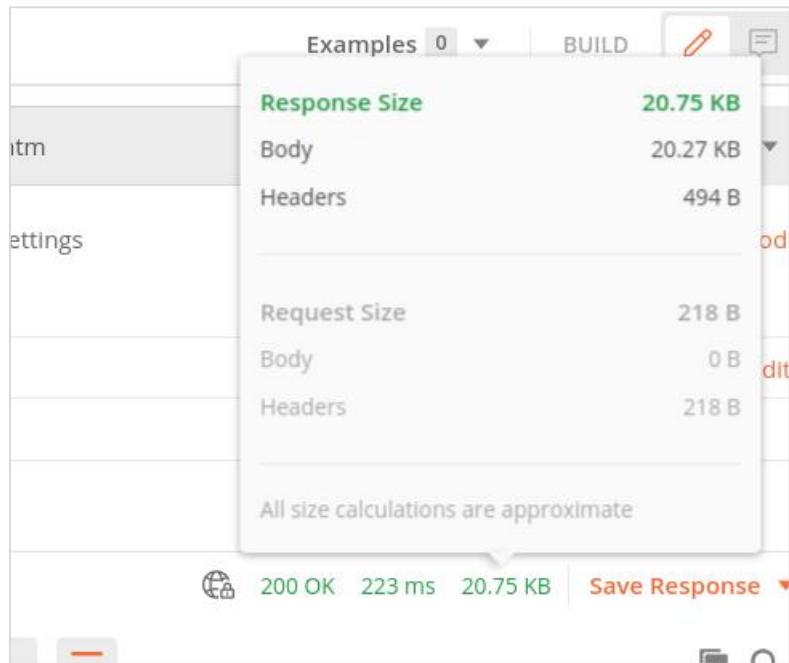
Once a request has been sent, we can see the response code **200 OK** populated in the Response. This signifies a successful request and a correct endpoint. Also, information on the time consumed to complete the request (223 ms) and payload size (20.75 KB) are populated.

The screenshot shows the Postman application interface. At the top, there's a header with 'GET Test1' and a status indicator (red dot). To the right are buttons for '+', 'ooo', 'No Environment', 'Examples 0', 'BUILD', and a settings icon. Below the header, a section titled 'Test1' contains a 'GET' method and a URL field with 'https://www.tutorialspoint.com/Index.htm'. To the right of the URL is a 'Send' button, which is highlighted with an orange box. Further right are 'Save' and other options. Below the URL, tabs include 'Params', 'Auth', 'Headers (6)', 'Body', 'Pre-req.', 'Tests', and 'Settings'. The 'Params' tab is selected. Under 'Query Params', there's a table with columns 'KEY', 'VALUE', 'DESCRIPTION', and 'Bulk Edit'. A single row is present with 'Key' and 'Value' columns empty. To the right of the table are 'Cookies' and 'Code' buttons. At the bottom of the main area, tabs for 'Body', 'Cookies', 'Headers (15)', and 'Test Results' are shown. The 'Test Results' tab is selected and displays a green box containing the response status: '200 OK 223 ms 20.75 KB'. Below this, there are buttons for 'Pretty', 'Raw', 'Preview', 'Visualize', and 'HTML' (with a dropdown arrow), along with a search icon. The response body itself is a block of HTML code, numbered 1 to 7.

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.



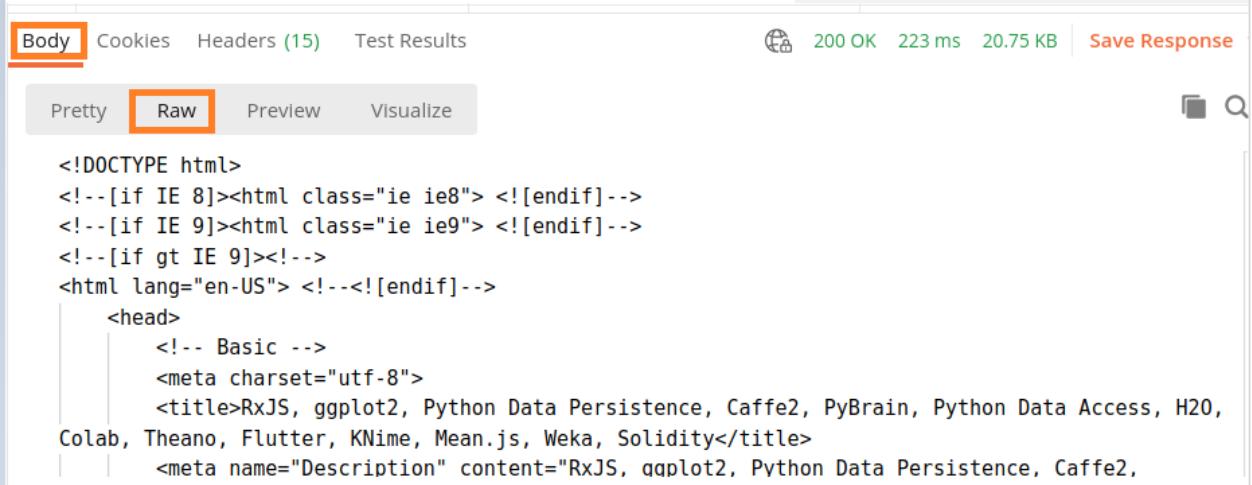
The Response Body contains the sub-tabs – Pretty, Raw and Preview. The Pretty format shows color formatting for keywords and indentation for easy reading. The Raw format displays the same data displayed in the Pretty tab but without any color or indentation.

The Preview tab shows the preview of the page.

KEY	VALUE	DESCRIPTION	...
Body	Cookies Headers (15) Test Results	200 OK 223 ms 20.75 KB	
Pretty	Raw Preview Visualize	HTML	
1	<!DOCTYPE html>		
2	<!--[if IE 8]><html class="ie ie8"> <![endif]-->		
3	<!--[if IE 9]><html class="ie ie9"> <![endif]-->		
4	<!--[if gt IE 9]><!-->		
5	<html lang="en-US">		
6	<!--<![endif]-->		
7			
8	<head>		
9	<!-- Basic -->		
10	<meta charset="utf-8">		
11	<title>RxJS, qplot2, Python Data Persistence, Caffe2, PyBrain, Python Data Access		
3	<!--[if lt IE 9]><html class="ie ie9"> <![endif]-->		
4	<!--[if gt IE 9]><!-->		
5	<html lang="en-US">		

## Raw tab

The following screen will appear:

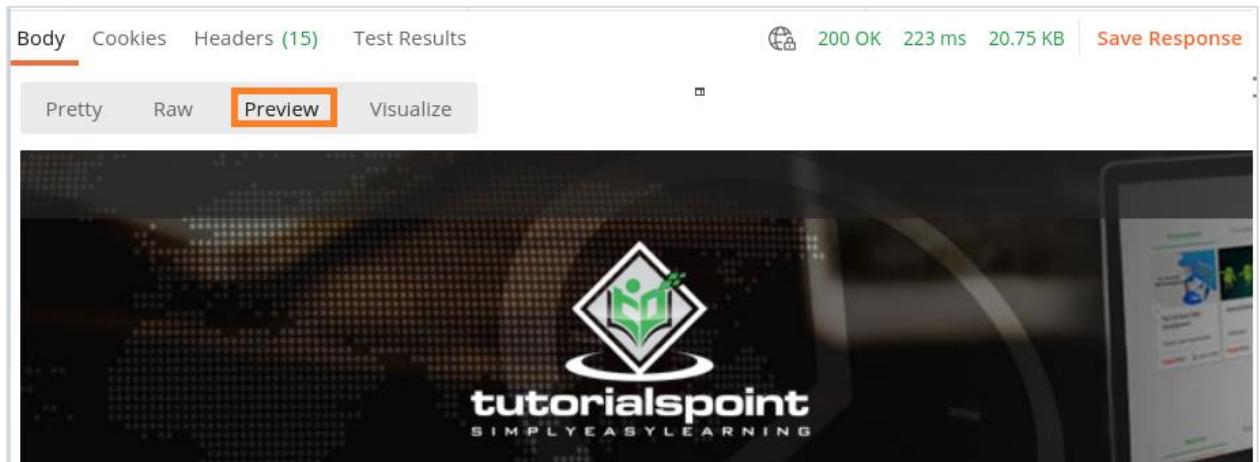


The screenshot shows the Postman interface with the 'Body' tab selected. The 'Raw' tab is highlighted with a red box. The response body is displayed as HTML code:

```
<!DOCTYPE html>
<!--[if IE 8]><html class="ie ie8"> <![endif]-->
<!--[if IE 9]><html class="ie ie9"> <![endif]-->
<!--[if gt IE 9]><!--
<html lang="en-US"> <!--<![endif]-->
    <head>
        <!-- Basic -->
        <meta charset="utf-8">
        <title>RxJS, ggplot2, Python Data Persistence, Caffe2, PyBrain, Python Data Access, H2O, Colab, Theano, Flutter, KNime, Mean.js, Weka, Solidity</title>
        <meta name="Description" content="RxJS, ggplot2, Python Data Persistence, Caffe2,
```

## Preview tab

The following screen will appear:



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

KEY	VALUE	DESCRIPTION
Body	Cookies Headers (15) Test Results	200 OK 223 ms 20.75 KB
Strict-Transport-Security	max-age=63072000; includeSubdomains	
Vary	Accept-Encoding	
X-Cache	HIT	
X-Content-Type-Options	nosniff	
X-Frame-Options	SAMEORIGIN	
X-XSS-Protection	1; mode=block	
Content-Length	20753	

# 7. Postman — POST Requests

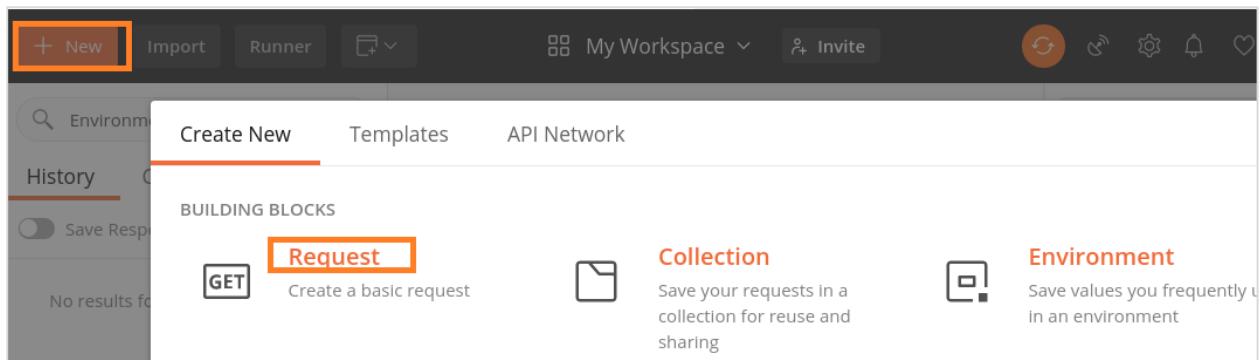
Postman POST request allows appending data to the endpoint. This is a method used to add information within the request body in the server. It is commonly used for passing delicate information.

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.

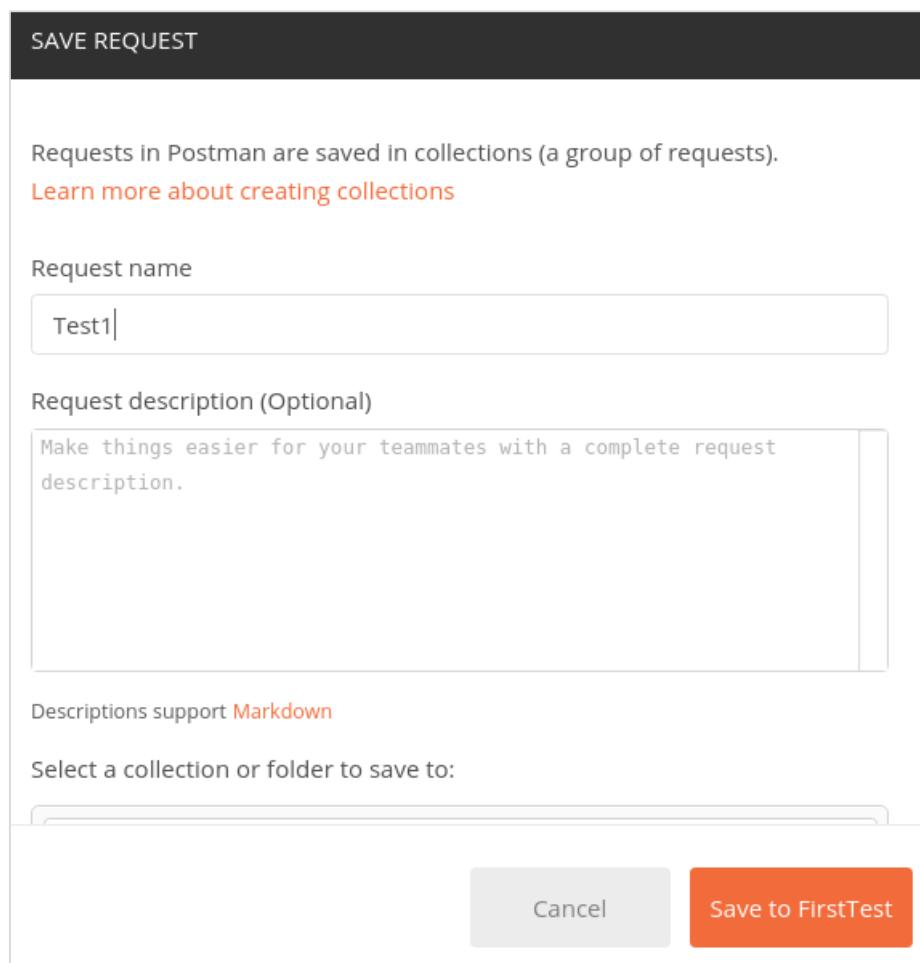
## Create a POST Request

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.

The screenshot shows the Postman Request tab. The method dropdown is set to 'POST', which is highlighted with an orange border. The URL 'https://jsonplaceholder.typicode.com/users' is entered in the address bar. On the left, other methods like GET, PUT, PATCH, and DELETE are listed. To the right, tabs for Body, Pre-req., Tests, and Settings are visible. Below the tabs, there's a table with columns 'VALUE' and 'DESCRIPTION'. At the bottom, it shows '5 (26) Test Results' and a status of '200 OK 86ms'.

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

The screenshot shows the Postman interface with a POST request to <https://jsonplaceholder.typicode.com/users>. The 'Body' tab is selected. A dropdown menu is open under the 'Text' dropdown, with 'raw' selected.

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

The screenshot shows the Postman interface with a POST request to <https://jsonplaceholder.typicode.com/users>. The 'Body' tab is selected. A dropdown menu is open under the 'Text' dropdown, with 'JSON' selected.

**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>.

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

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

The screenshot shows the Postman interface with a POST request named 'Test1' to the URL <https://jsonplaceholder.typicode.com/users>. The 'Body' tab is selected, and the 'raw' type is chosen. The JSON payload is:

```

1  {
2      "id": 11,
3      "name": "Tutorialspoint",
4      "username": "Test1",
5      "email": "abc@gmail.com",
6      "address": {
7          "street": "qa street",
8          "suite": "Apt 123",
9          "city": "Kochi",
10         "zipcode": "49085",
11         "geo": {
12             "lat": "-3.3155",
13             "lng": "94.156"
14         }
15     },
16     "phone": "99599125",
17     "website": "Tutorialspoint.com",
18     "company": {

```

**Step 7:** Click on the **Send** button.

The screenshot shows the Postman interface with the 'Send' button highlighted. The request details are identical to the previous screenshot.

## 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.

POST Test1

Test1

POST https://jsonplaceholder.typicode.com/users

Body (1)

```

1 {
2   "id": 11,
3   "name": "Tutorialspoint",
4   "username": "Test1",
5   "email": "abc@gmail.com",
6   "address": {
7     "street": "qa street",
8     "suite": "Apt 123",
9     "city": "Kochi",
10    "zipcode": "49085",
11    "geo": {
12      "lat": "-3.3155",
13      "lng": "94.156"
14    }
15  },
16  "phone": "99599125",
17  "website": "Tutorialspoint.com",
18  "company": {
19    "name": "Tutorialspoint",
20    "catchPhrase": "Simple Easy Learning",
21    "bs": "Postman Tutorial"
22  }
23 }
```

201 Created 347 ms 1.61 KB

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

POST https://jsonplaceholder.typicode.com/users

Body (1)

```

{
  "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"
  }
}
```

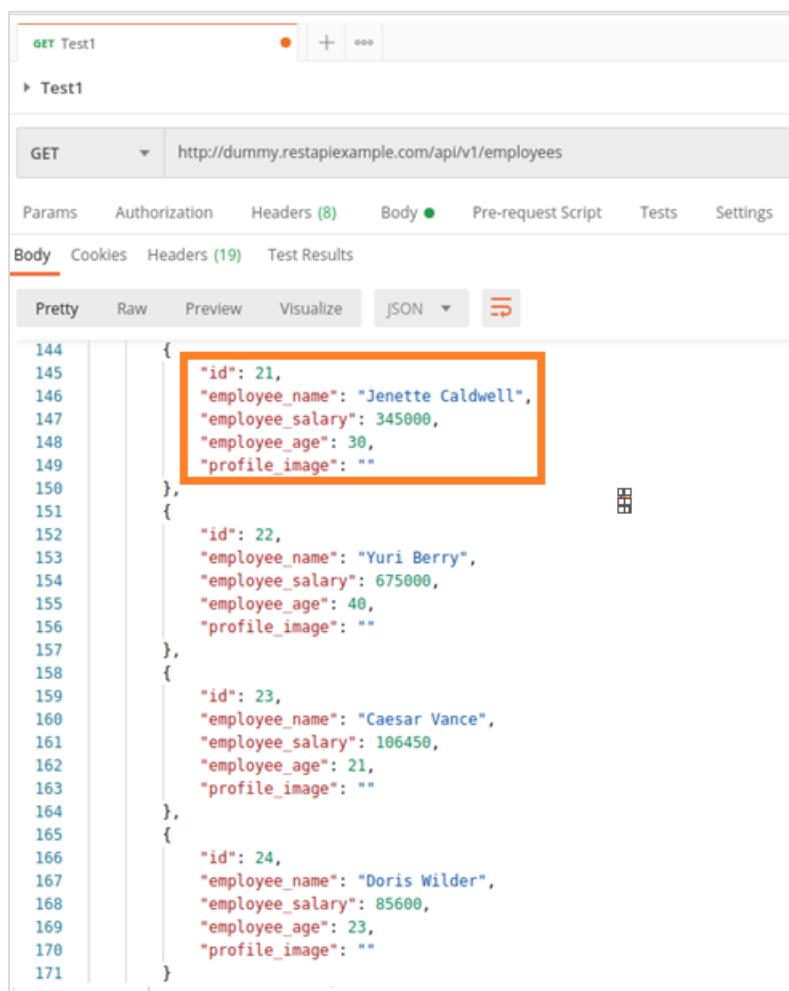
## 8. Postman — PUT Requests

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>. 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:



```
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171 }]
```

The JSON response body is a list of employees. The employee with id 21 is highlighted in the screenshot.

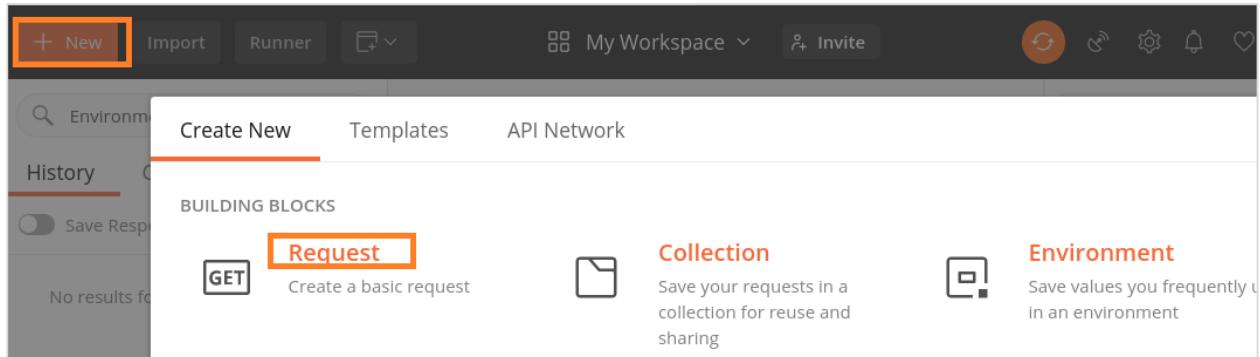
```
{
  "id": 21,
  "employee_name": "Jenette Caldwell",
  "employee_salary": 345000,
  "employee_age": 30,
  "profile_image": ""
}, {
  "id": 22,
  "employee_name": "Yuri Berry",
  "employee_salary": 675000,
  "employee_age": 40,
  "profile_image": ""
}, {
  "id": 23,
  "employee_name": "Caesar Vance",
  "employee_salary": 106450,
  "employee_age": 21,
  "profile_image": ""
}, {
  "id": 24,
  "employee_name": "Doris Wilder",
  "employee_salary": 85600,
  "employee_age": 23,
  "profile_image": ""
}
```

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

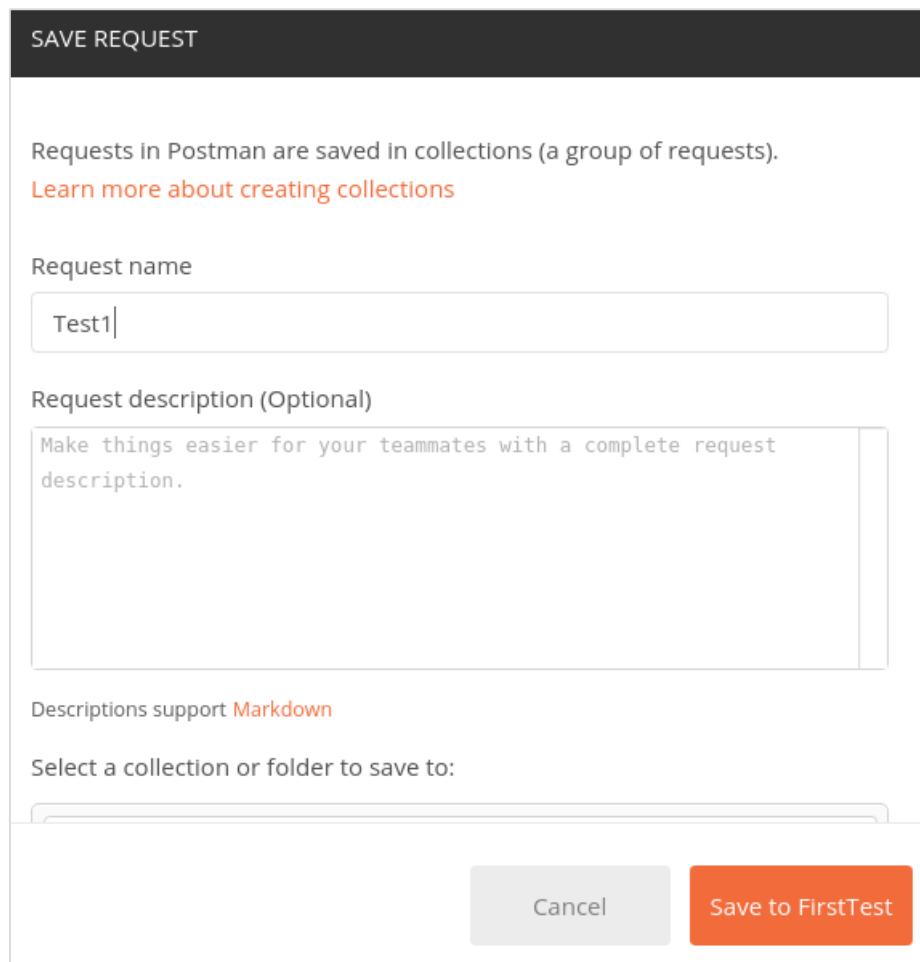
## Create a PUT Request

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> (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.

The screenshot shows the Postman interface with a test named "Test1". A PUT request is selected. The URL "http://dummy.restapitutorial.com/api/v1/update/21" is entered in the address bar. The "Body" tab is highlighted in red, indicating it is active. Below the address bar, there are tabs for GET, POST, and PUT, with PUT being the active one. To the right of the address bar, there are buttons for Headers (8), Body (highlighted in green), Pre-request Script, and Tests.

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

The screenshot shows the "Body" tab selected. A dropdown menu is open, listing "none", "form-data", "x-www-form-urlencoded", "raw" (which is highlighted with a red box), "binary", and "GraphQL".

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

The screenshot shows the "Body" tab selected. A dropdown menu is open, listing "Text", "JavaScript", "JSON" (which is highlighted with a red box), "HTML", and "XML".

**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:

PUT http://dummy.restapiexample.com/api/v1/update/21

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

none form-data x-www-form-urlencoded raw binary GraphQL **JSON**

```
1 {"name": "Jenette Caldwell", "salary": "2000", "age": "15"}  
2  
3  
4
```

**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.

PUT http://dummy.restapiexample.com/api/v1/update/21 **Send**

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

none form-data x-www-form-urlencoded raw binary GraphQL **JSON**

```
1 {"name": "Jenette Caldwell", "salary": "2000", "age": "15"}  
2  
3  
4
```

Body Cookies Headers (18) Test Results Status: 200 OK Time: 673 ms Size: 705 B

Pretty Raw Preview Visualize **JSON**

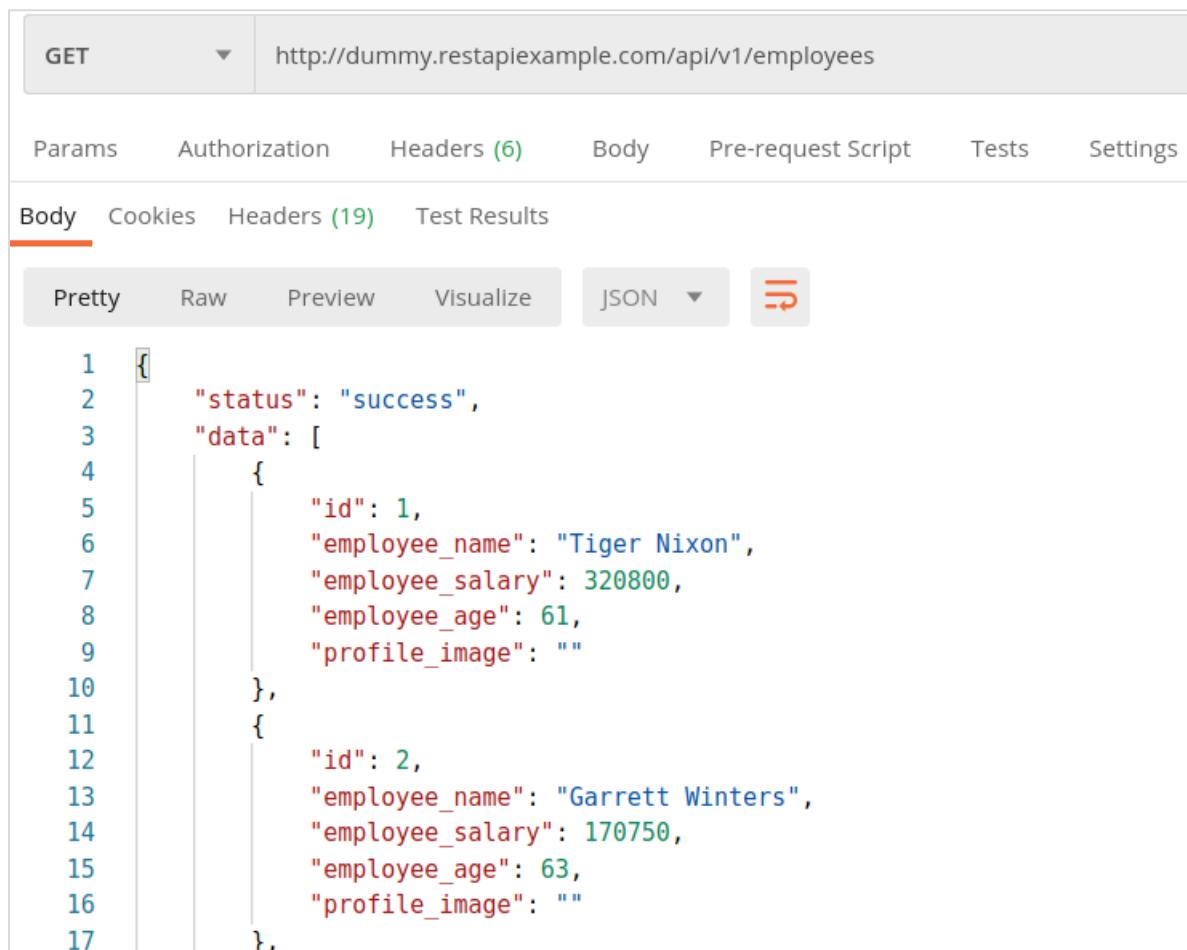
```
1 {  
2   "status": "success",  
3   "data": {  
4     "name": "Jenette Caldwell",  
5     "salary": "2000",  
6     "age": "15"  
7   },  
8   "message": "Successfully! Record has been updated."  
9 }
```

# 9. Postman — DELETE Requests

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>. The details on how to create a GET request is explained in detail in the Chapter on GET Requests.

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



The screenshot shows the Postman interface with a GET request to <http://dummy.restapiexample.com/api/v1/employees>. The Headers tab shows 6 items. The Body tab is selected, displaying a JSON response with two employee records. The response body is:

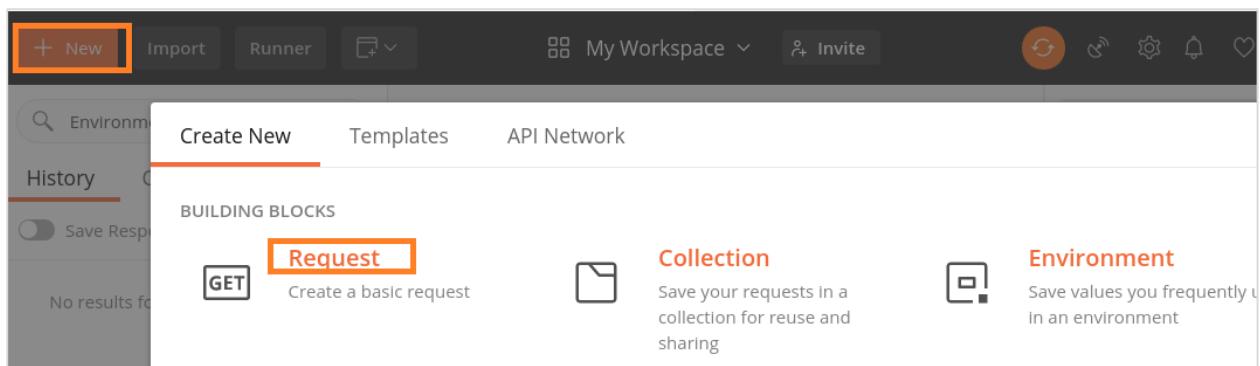
```
1  {
2      "status": "success",
3      "data": [
4          {
5              "id": 1,
6              "employee_name": "Tiger Nixon",
7              "employee_salary": 320800,
8              "employee_age": 61,
9              "profile_image": ""
10         },
11         {
12             "id": 2,
13             "employee_name": "Garrett Winters",
14             "employee_salary": 170750,
15             "employee_age": 63,
16             "profile_image": ""
17         }
    ]
```

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

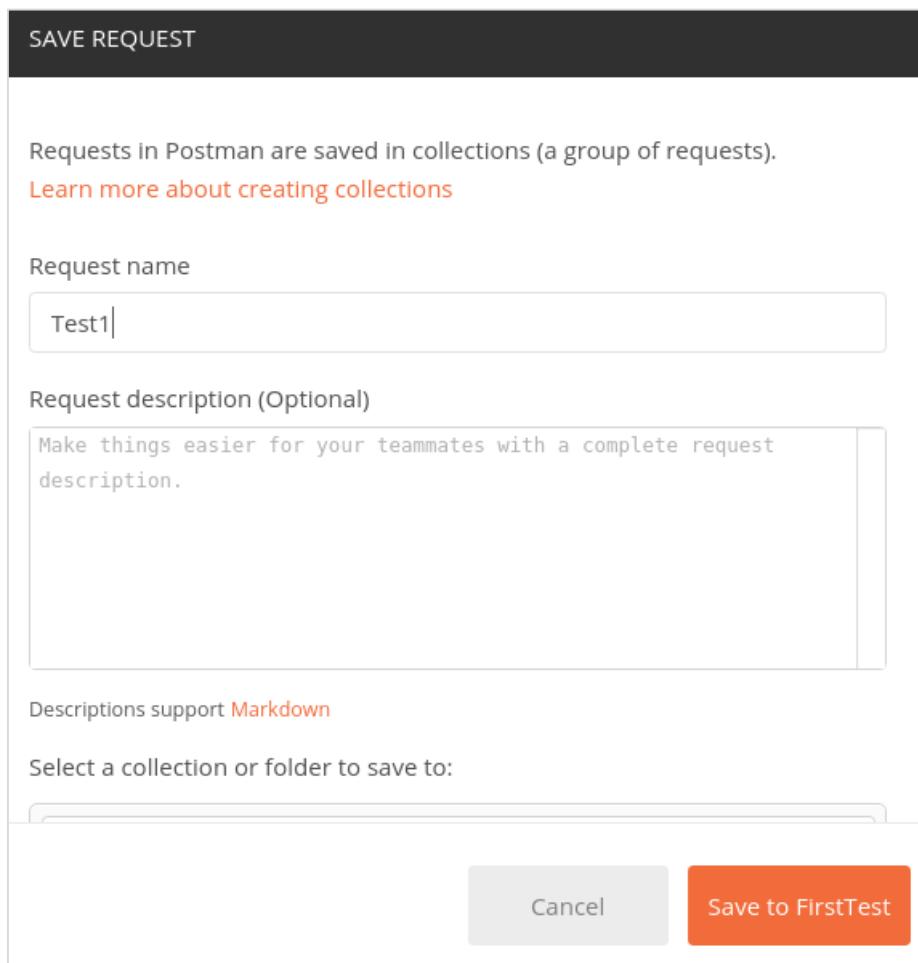
## Create a DELETE Request

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.



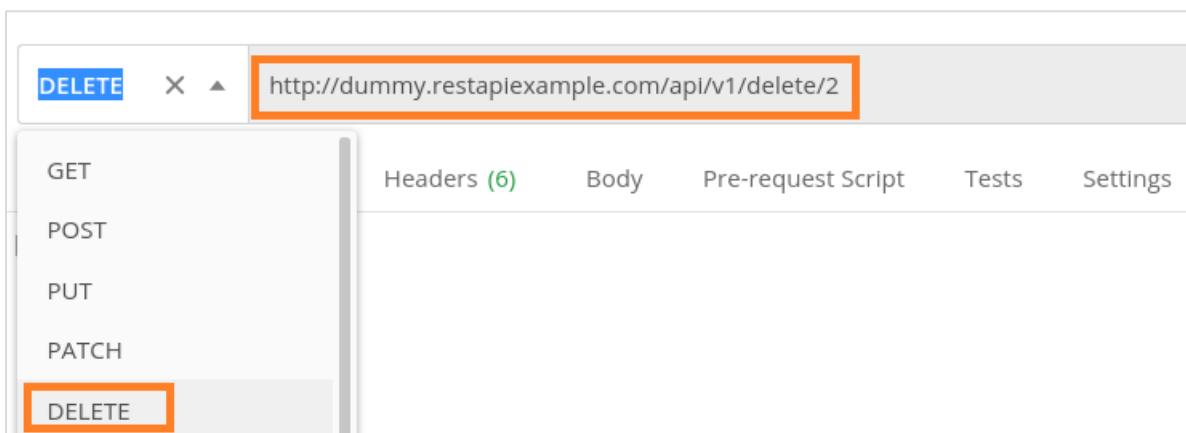
**Step 2: SAVE REQUEST** pop-up comes up. Enter the Request name then click on **Save**.



**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.



**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.

The screenshot shows the Postman interface after sending the DELETE request. The response body is a JSON object:

```

1 [ {
2   "status": "success",
3   "data": "2",
4   "message": "Successfully! Record has been deleted"
5 }
6 ]
    
```

The status bar indicates Status: 200 OK, Time: 734 ms, and Size: 652 B.

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.

# 10. Postman — Create Tests for CRUD

CRUD stands for **Create, Retrieve, Update and Delete** operations on any website opened in a browser. Whenever we launch an application, the retrieve operation is performed.

On creating data, for example, adding a new user for a website, the create operation is performed. If we are modifying the information, for example, changing details of an existing customer in a website, the update operation is performed.

Finally, to eliminate any information, for example, deleting a user in a website, the delete operation is carried out.

To retrieve a resource from the server, the HTTP method – GET is used (discussed in details in the Chapter – Postman GET Requests). To create a resource in the server, the HTTP method – POST is used (discussed in details in the Chapter – Postman POST Requests).

To modify a resource in the server, the HTTP method – PUT is used (discussed in details in the Chapter – Postman PUT Requests). To delete a resource in the server, the HTTP method – DELETE is used (discussed in details in the Chapter – Postman DELETE Requests).

## Tests in Postman

---

A Postman test is executed only if a request is successful. If a Response Body is not generated, it means our request is incorrect and we will not be able to execute any test to validate a Response.

In Postman, tests are developed in JavaScript and can be developed using the JavaScript and Functional methods. Both the techniques are based on the language JavaScript.

### JavaScript Method

Follow the steps given below to develop tests in Javascript:

**Step 1:** Tests developed in the JavaScript method are mentioned within the Tests tab under the address bar.

The screenshot shows the Postman interface with a request titled 'Get Request'. The 'Tests' tab is highlighted with a red box. Below it, there is a code editor containing the following JavaScript:

```
tests["Status Code should be 200"] = responseCode.code === 200
tests["Response time lesser than 10ms"] = responseTime<10
```

**Step 2:** Add the below JavaScript verifications within the Tests tab:

```
tests["Status Code should be 200"] = responseCode.code === 200
tests["Response time lesser than 10ms"] = responseTime<10
```

We can add one or more than one test for a particular request.

Here, tests is a variable of type array which can hold data types- integer, string, Boolean and so on. The Status Code should be 200 and Response time lesser than 10ms are the names of the tests. It is recommended to give meaningful names to test.

The **responseCode.code** is the response code obtained in the Response and the **responseTime** is the time taken to get the Response.

**Step 3:** Select the GET method and enter an endpoint then click on **Send**.

## Response

In the Response, click on the **Test Results** tab:

The screenshot shows the Postman interface with the 'Test Results' tab selected. The code editor contains the same tests as before:

```
1 tests["Status Code should be 200"] = responseCode.code === 200
2 tests["Response time lesser than 10ms"] = responseTime<10
```

The results section shows one test passed and one failed:

- PASS Status Code should be 200
- FAIL Response time lesser than 10ms | Assertion Error: expected false to be truthy

At the bottom right, the status bar shows: Status: 200 OK Time: 129 ms Size: 20.75 KB.

The Test Results tab shows the test which has passed in green and the test which has failed in red. The Test Results (1/2) means one out of the two tests has passed.

Response shows the status as 200 OK and Response time as 129ms (the second test checks if the Response time is less than 10ms).

Hence, the first test got passed and the second one failed along with the Assertion error.

## Functional Method

Follow the steps given below to develop a test in with functional method:

**Step 1:** Tests developed in the Functional method are mentioned within the Tests tab under the address bar.

**Step 2:** Add the below code within the Tests tab:

```
pm.test["Status Code is 401"], function(){
    pm.response.to.have.status(401)
})
```

Here, **pm.test** is the function for the test being performed. Status Code is **401** and it is the name of the test which shall be visible in the Test Result after execution.

The **pm.response** is used for obtaining the response and adding assertions on it to verify the header, code, status, and so on.

**Step 3:** Select the GET method and enter an endpoint then click on Send.

## Response

In the Response, click on the **Test Results** tab:

The screenshot shows the Postman interface with a 'Get Request' selected. In the 'Tests' tab, the following JavaScript code is written:

```
1 pm.test["Status code is 401", function(){
2     pm.response.to.have.status(401)
3 })
```

The 'Test Results' tab is active, showing a single test named 'Status code is 401'. The status is listed as 'FAIL' with the message 'Status code is 401 | Assertion Error: expected response to have status code 401 but got 200'. Other tabs like 'Body', 'Cookies', and 'Headers' are visible at the bottom.

The Test Results tab shows the test in red as the test has failed. The Test Results (0/1) means zero out of the one test has passed. Response shows the status as 200 OK (the test checks if the response code is 401).

Hence the test shows failed along with the Assertion error.

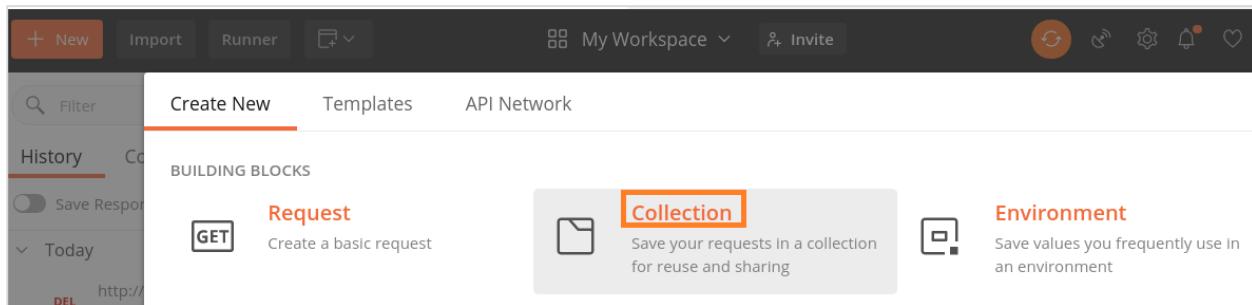
# 11. Postman — Create Collections

A group of requests that have been saved and organized into folders is known as the Collections. It is similar to a repository. Thus, Collections help to maintain the API tests and also split them easily with teams.

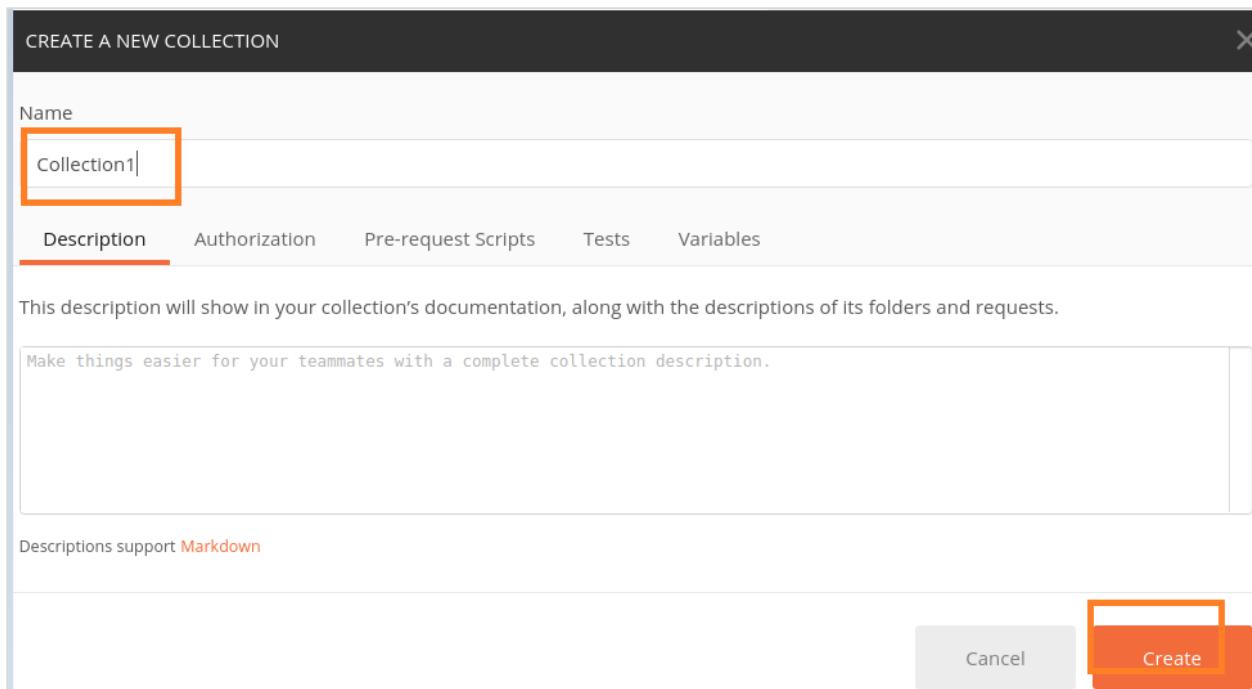
## Create a New Collection

Follow the steps given below to create a new collection in Postman:

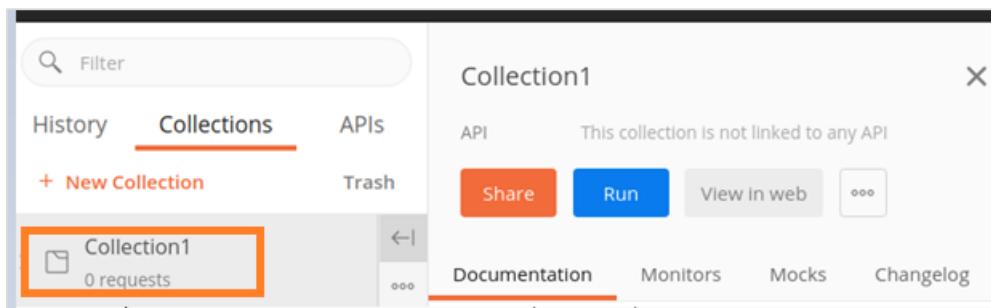
**Step 1:** Click on the **New** icon from the Postman application. The **Create New** pop-up comes up. Then click on the Collection link.



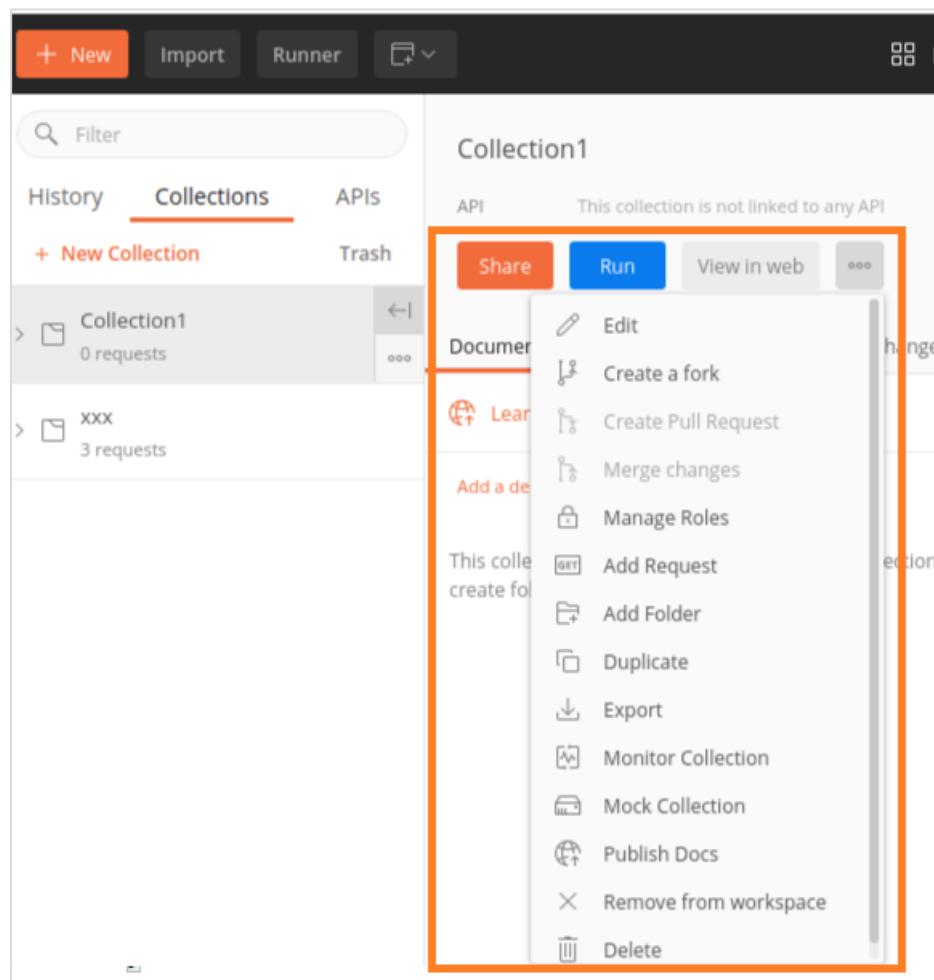
**Step 2:** CREATE A NEW COLLECTION pop-up comes up. Enter a Collection Name and click on the Create button.



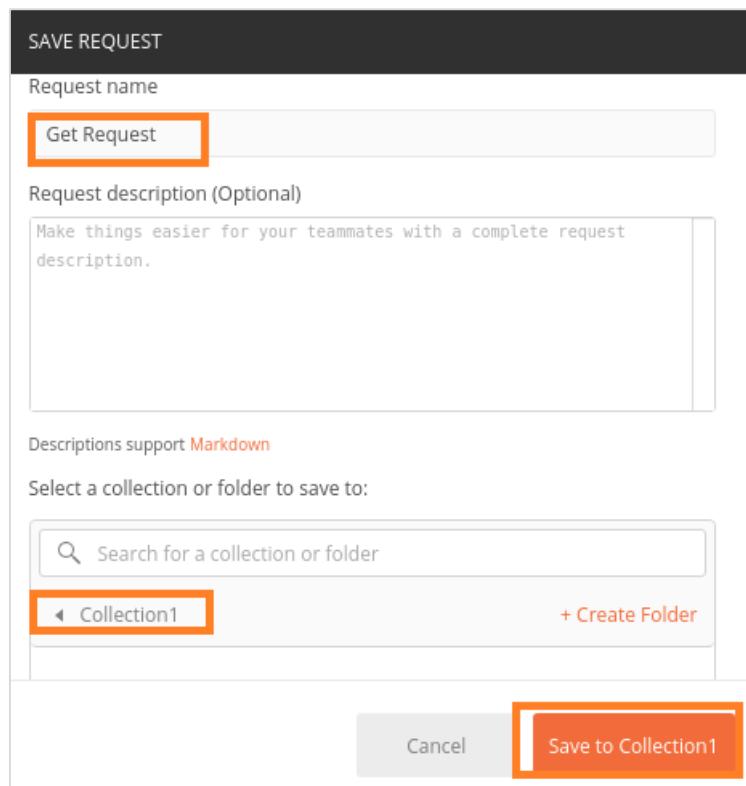
**Step 3:** The Collection name and the number of requests it contains are displayed in the sidebar under the Collections tab.



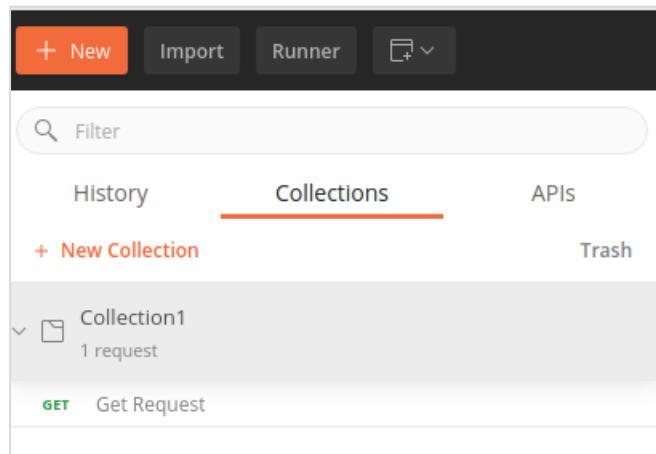
**Step 4:** To the right of the Collection name, we have the options like Share, Run and so on available. Click on the three dots to get more options to select.



**Step 5:** Click on **Add Request**. The SAVE REQUEST pop-up comes up. Enter Request Name and select the Collection we have created. Then, click on the Save to Collection1 button.



**Step 6:** The Collection with its request gets displayed to the side bar under the Collections tab.



# 12. Postman — Parameterize Requests

We can parameterize Postman requests to execute the same request with various sets of data. This is done with the help of variables along with parameters. A parameter is a part of the URL used to pass more information to the server.

The data can be used in the form of a data file or an Environment variable. Parameterization is an important feature of Postman and helps to eliminate redundant tests. Parameters are enclosed in double curly braces `{{parameter}}`.

## Example

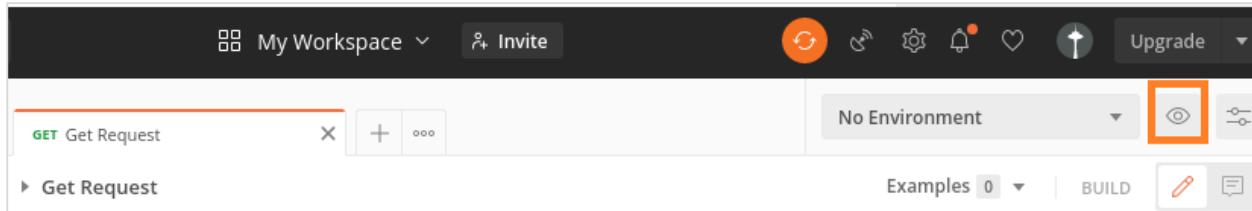
Let us take an example of an URL: <https://www.tutorialspoint.com/index.htm>. We shall create a variable as url then use it for parameterization of request. We can refer to it in the format `{{url}}` in Postman.

A parameter is in the form of a key-value pair. So to point to the URL: <https://www.tutorialspoint.com/index.htm>, we can mention it as `url`:`https://www.tutorialspoint.com`. So here, the url is the key and the value set is `https://www.tutorialspoint.com`.

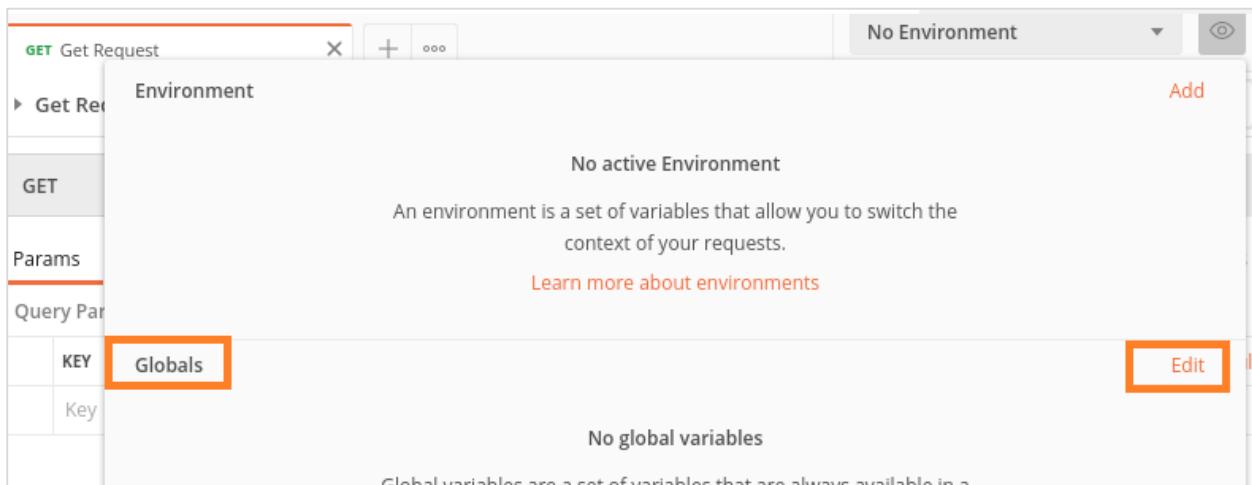
## Create a Parameter Request

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

**Step 1:** Click on the **eye icon** to the right of the Environment dropdown in the top right corner in the Postman application.



**Step 2:** Click on the **Edit link** in the **Globals** section.



**Step 3: MANAGE ENVIRONMENTS** pop-up comes up. Enter URL for the VARIABLE field and <https://www.tutorialspoint.com> for INITIAL VALUE. Then, click on **Save**.

VARIABLE	INITIAL VALUE	CURRENT VALUE	...	Persist All	Reset All
url	https://www.tutorialspoint.com	https://www.tutorialspoint.com			

Use variables to reuse values in different places. Work with the current value of a variable to prevent sharing sensitive values with your team. [Learn more about variable values](#)

Save and Download as JSON    Cancel    **Save**

**Step 4:** Click on close to move to the next screen.

**Step 5:** In the Http Request tab, enter `{{url}}/index.htm` in the address bar. Select the **GET method** and click on **Send**.

GET Get Request    +    No Environment    Examples 0    BUILD

Get Request

GET    {{url}}/index.htm    Send    Save

## 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.

The screenshot shows the Postman application interface. At the top, there's a header with 'GET Get Request' and a status indicator (red dot). To the right are buttons for '+', '...', 'No Environment', 'Examples 0', 'BUILD', and settings. Below the header, a search bar contains 'Get Request'. The main area has tabs for 'GET' and 'POST' with the URL '{url}/index.htm'. Buttons for 'Send' and 'Save' are visible. Below these are tabs for 'Params', 'Auth', 'Headers (6)', 'Body', 'Pre-req.', 'Tests', 'Settings', 'Cookies', and 'Code'. The 'Params' tab is selected, showing a table with one row: 'Key' (Value) and 'Value' (Description). Under the 'Body' tab, a dropdown shows 'Pretty', 'Raw', 'Preview', 'Visualize', and 'HTML' (selected). The response section shows '200 OK' with a globe icon, '147 ms', and '20.75 KB'. A red box highlights this area. To the right are 'Save Response' and other icons. The response body is displayed as a code block:

```
1 <!DOCTYPE html>
2 <!--[if IE 8]><html class="ie ie8"> <![endif]-->
3 <!--[if IE 9]><html class="ie ie9"> <![endif]-->
4 <!--[if gt IE 9]><!-->
5 <html lang="en-US">
6 <!--<![endif]-->
```

# 13. Postman — Collection Runner

Postman Collection Runner is used to execute a Collection having multiple requests together. All the requests within a Collection will be executed simultaneously. The Collection Runner does not produce any Response Body.

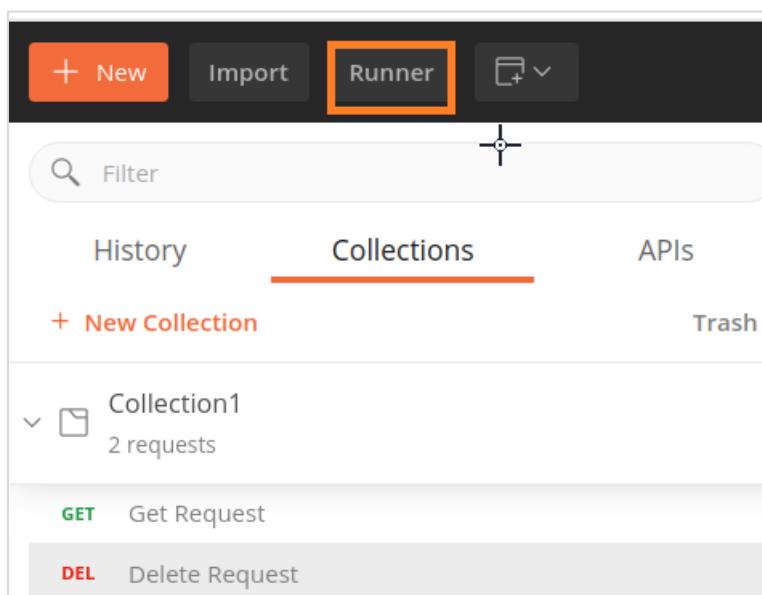
The Collection Runner console displays the test results for individual requests. It is mandatory to have more than one request within the Collection to work with Collection Runner.

The details on how to create a Collection is discussed in detail in the Chapter on Create Collections.

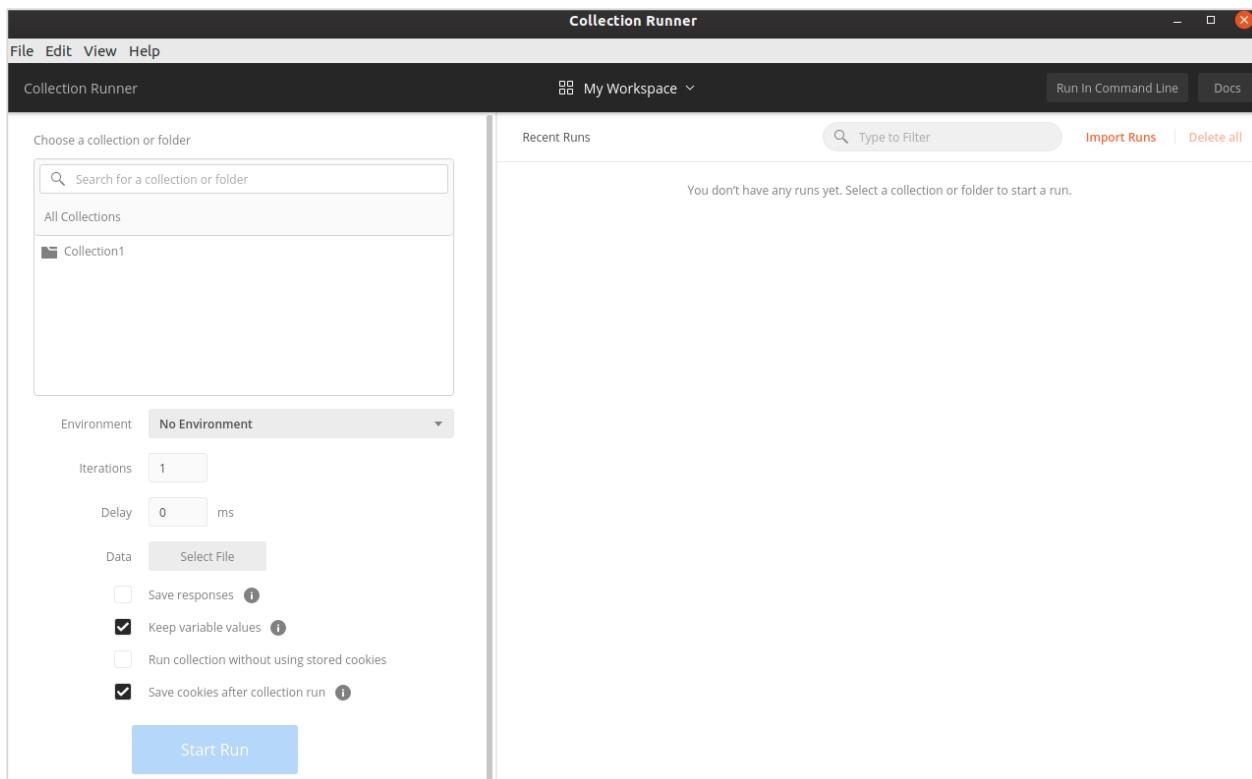
## Execute Tests with Collection Runner

Follow the steps given below to execute the tests with Collection Runner in Postman:

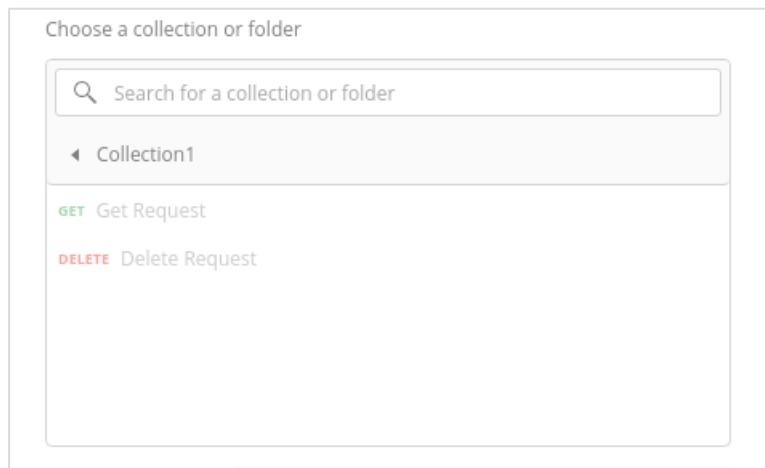
**Step 1:** Click on the **Runner** menu present at the top of the Postman application.



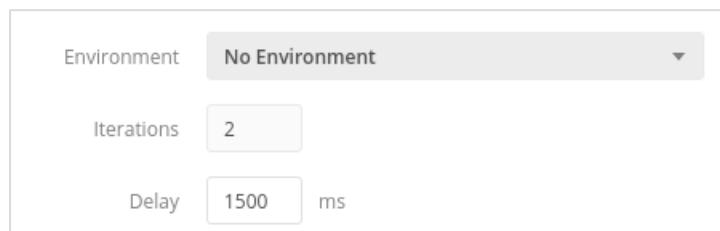
**Step 2:** The **Collection Runner** screen shall appear.



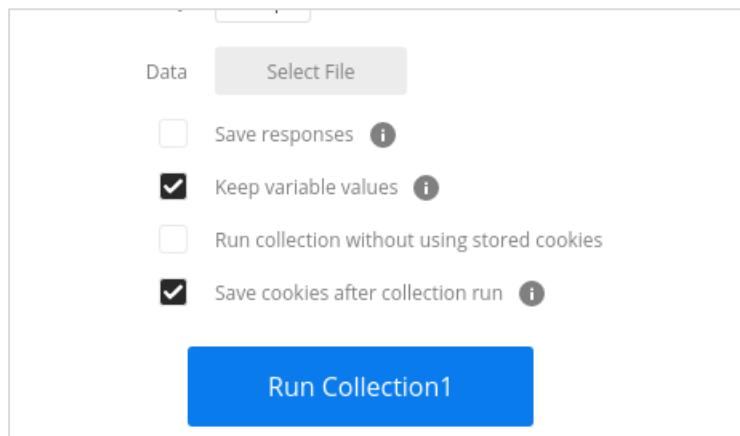
**Step 3:** Select the Collection name from **Choose a collection or folder**.



**Step 4:** Select an environment from the Environment dropdown to run the requests in a particular environment. Then, specify the number of times we need to iterate the request. We can also set a delay time in milliseconds for the requests.



**Step 5:** If we have data in a file, then we have to choose the file type from Data. Then, click on the Run Collection1 button.



**Step 6:** The Run Results page shall come up. Depending on the delay time provided, the tests should get executed.

The test results (Pass/Fail) should be displayed for each iteration. The **pass status** is represented in green and **failed** ones are represented in red. If there is no test implemented for a particular request, then it shall display the message as - **This request does not have any tests.**

This is the environment in which the tests are executed and the Collection names are visible at the top of the Collection Runner. For each request, the status code, time taken, payload size, and test verification are also displayed.

The screenshot shows the 'Collection Runner' interface with the 'Run Results' tab selected. At the top, it displays 'Collection1' (No Environment) with 4 PASSED and 0 FAILED tests, run just now. Below this, the 'Run Summary' and 'Export Results' buttons are visible. The main area shows two iterations of 'Collection1'. Iteration 1 contains three requests: 'Get Request' (status 200 OK, 117 ms, 21.248 KB), 'Status Code is 200' (green checkmark), and 'Delete Request' (status 200 OK, 640 ms, 652 B). Iteration 2 contains the same three requests, with the first two having green checkmarks and the last one being 'Status Code is 200'.

Iteration	Request Type	URL	Status	Time	Size
Iteration 1	GET	/Index.htm	200 OK	117 ms	21.248 KB
	DELETE	http://dummy.restapiex...	200 OK	640 ms	652 B
	Status Code is 200				
Iteration 2	GET	/Index.htm	200 OK	19 ms	21.248 KB
	Status Code is 200				
	DELETE	http://dummy.restapiex...	200 OK	706 ms	652 B
Status Code is 200					

# 14. Postman — Assertion

Assertions are used to verify if the actual and expected values have matched after the execution of a test. If they are not matching, the test shall fail and we shall get the reason for failure from the output of the test.

An assertion returns a Boolean value of either true or false. In Postman, we can take the help of JavaScript Chai Assertion Library to add assertions in our tests. It is available in the Postman application automatically.

The Chai – Assertions are easily comprehensible as they are defined in a human readable format. The Assertions in Postman are written within the Tests tab under the address bar.

The documentation for Chai is available in the following link:

<https://www.chaijs.com/>



A screenshot of the Postman application interface. At the top, there is a header with a dropdown menu set to 'GET' and a URL field containing '{{url}}/Index.htm'. Below the header, there are several tabs: 'Params', 'Authorization', 'Headers (6)', 'Body', 'Pre-request Script', 'Tests' (which is highlighted with a red box), and 'Settings'. The 'Tests' tab contains a code editor with the following Chai assertion script:

```
1 pm.test["Status code is 401", function(){
2     pm.response.to.have.status(401)
3 })
```

## Writing Assertions

Let us write an assertion to check **if a particular text - Postman is within an array of strings**.

```
pm.test["Text is present"], function(){
    pm.expect(['Java', 'Postman']).to.include('Postman')
}
```

## Output

The output is as follows:

The screenshot shows the Postman interface with the 'Tests' tab selected. The test script contains a single assertion:

```
1 pm.test("Text is present", function(){
2     pm.expect(['Java', 'Postman']).to.include('Postman')
3 }
4 )
```

The 'Test Results' section shows 1/1 test passed, labeled 'Text is present'.

Let us write an Assertion to check **if an array is empty**.

```
pm.test["Array contains element"], function(){
    pm.expect(['Java', 'Postman']).to.be.an('array').that.is.not.empty
})
```

### Output

The output is given below:

The screenshot shows the Postman interface with the 'Tests' tab selected. The test script contains a single assertion:

```
1 pm.test("Array contains element", function(){
2     pm.expect(['Java', 'Postman']).to.be.an('array').that.is.not.empty
3 }
4 )
```

The 'Test Results' section shows 1/1 test passed, labeled 'Array contains element'.

## Assertion for Object Verification

Let us write an **Assertion for object verification** with eql. It is used to compare the properties of the object i and j in the below example.

```
pm.test("Equality", function(){
let i = {
"subject" : "Postman"
};
let j= {
"subject" : "Cypress"
};
pm.expect(i).to.not.eql(j);
```

### Output

The output is mentioned below:

The screenshot shows the Postman interface with the 'Tests' tab selected. The test script is displayed in the 'Tests' section:

```
1 pm.test("Equality", function(){
2   let i = {
3     "subject": "Postman"
4   }
5   let j = [
6     "subject": "Cypress"
7   ]
8   pm.expect(i).to.not.eq(j)
9 })
10
```

The 'Test Results' section shows 1/1 test passed. The results table includes columns for 'All', 'Passed', 'Skipped', and 'Failed'. One result row is shown with status 'PASS' and label 'Equality'.

All	Passed	Skipped	Failed
PASS	Equality		

The property defined for object i is Postman while the property defined for j is Cypress. Hence, **not.eql** Assertion got passed.

## Assertion Types

In Postman, we can apply assertions on different parts of Response. These are explained below:

### Status Code

```
pm.test["Status Code is 401"], function(){
    pm.response.to.have.status(401)
})
```

The above assertion passes if the Response status code obtained is 401.

```
pm.test["Status is Forbidden"], function(){
    pm.response.to.have.property('status', 'Forbidden')
})
```

The above assertion is applied on the Response property – status having the value Forbidden.

### Time taken by Response

The assertion for time taken by response is as follows:

```
pm.test("Response time above 500 milliseconds", function () {
    pm.expect(pm.response.responseTime).to.be.above(500)
})
```

The above assertion passes if the Response time is above 500ms.

### Type of Response Format

The assertion for type of response format is as follows:

```
pm.test("Response type is JSON", function(){
    pm.response.to.be.json;
})
```

The above assertion passes if the Response is of JSON type.

### Header of Response

The assertion for header of response is as follows:

```
pm.test("Header Content-Encoding is available", function () {
    pm.response.to.have.header("Content-Encoding")
})
```

The above assertion passes if the Response has a header Content-Encoding.

## Text of Response

The assertion for text of response is as follows:

```
pm.test("Response Text", function () {  
    pm.expect(pm.response.text()).to.include("Tutorialspoint")  
})
```

The above assertion passes if the Response text contains the text Tutorialspoint.

# 15. Postman — Mock Server

A mock server is not a real server and it is created to simulate and function as a real server to verify APIs and their responses. These are commonly used if certain responses need to be verified but are not available on the web servers due to security concerns on the actual server.

## Purpose of Mock Server

A Mock Server is created for the reasons listed below:

- A Mock Server is created if the APIs to be used in Production are still in development.
- A Mock Server is used if we want to avoid sending requests on real time data.

## Benefits of Mock Server

---

The benefits of Mock Server are listed below:

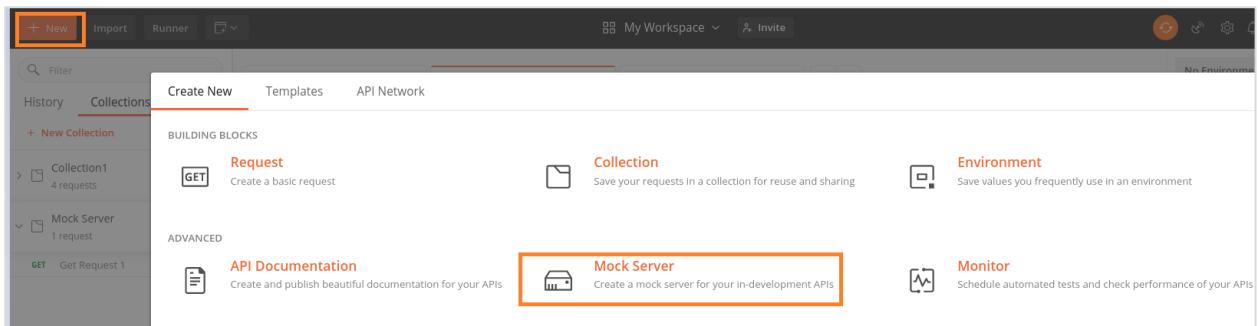
- Simulation of real API features with examples.
- Mock server can be appended to a Collection.
- Verify APIs with mocking data.
- To identify errors and defects early.
- To identify dependencies in API before it is released for actual usage.
- It is used by engineers to build a prototype for a concept and showcase it to higher management.
- While developing the front end of an application, the developer should have some idea on the response features that shall be obtained from the real server on sending a request. A Mock Server can be really helpful at this time.

## Mock Server Creation

---

Follow the steps given below for creation of mock server in Postman:

**Step 1:** Click on the **New** icon from the Postman application. Then, click on **Mock Server**.



**Step 2:** Select **GET** from the Method dropdown, enter a Request Path as **/user/home**, Response Code as **200**, and a Response Body. Then, click on **Next**.

Method	Request Path	Response Code	Response Body
GET	{{url}}/ /user/home	200	This is Postman Tutori in TutorialsPoint
GET	{{url}}/ Path	200	Response Body

Learn how mock servers and examples can [speed up your API development](#).

[Back](#) [Next](#)

**Step 3:** Enter a Mock Server name and click on the Create Mock Server button.

Name the mock server  
Mock Server

Select an environment (optional)  
No Environment

Environments are groups of variables. The mock server will use the variable values from the selected environment in the request and response.

Note: To call a private mock server, you'll need to add an `x-api-key` header to your requests. See how to [generate a Postman API key](#).

Note: This will create a new environment containing the URL.

Make this mock server private

Save the mock server URL as an environment variable

Add a delay before sending response

Simulate a network delay

[Back](#) [Create Mock Server](#)

**Step 4:** The Mock Server gets created along with the Mock URL. The Copy Mock URL button is used to copy the Mock link. Click on the Close button to proceed.

The screenshot shows the Postman interface at the '3. Next steps' stage. It displays a success message: 'Mock server, collection and environment created. To keep things easy, they're all named Mock Server.' Below this, it says 'Check out your mock server's call logs here: [Mock Call Logs](#)'. A 'Try it out:' section is present, followed by instructions to 'To call the mock server:'. A bulleted list includes: 'Add examples responses to each request that the mock server will return. [Learn what examples are and how to use them](#)' and 'Then just send a request to the following URL, followed by the request path: <https://05303abe-b842-4c47-ab8c-db2af9334f57.mock.pstmn.io/>. Or select the Mock Server environment and enter {{url}} followed by the request path.' A 'Copy Mock URL' button is shown. A note at the bottom states: 'Note: Calls to a mock server count against your plan's monthly call limits. Be sure to check your [usage limits](#).'

**Step 5:** Select **Mock Server** as the Environment from the No Environment dropdown and click on Send. The Response code obtained is **200 OK** which means that the request is successful.

Also the Response Body shows the message – **This is Postman Tutorial for TutorialsPoint** which is the same as we passed as a Response Body in the Step 2.

The screenshot shows the Postman interface with a successful API call. The URL is 'GET /user/home'. The 'Send' button is highlighted. The 'Status: 200 OK' message is displayed. The response body is '1 This is Postman Tutori in Tutoraispoint'.

**Step 6:** The value of URL can be obtained by clicking on the eye icon at the right upper corner of the screen.

The screenshot shows the Postman interface with the title "Mock Server". A table lists variables with their initial and current values. The "url" variable is highlighted with an orange border.

VARIABLE	INITIAL VALUE	CURRENT VALUE
url	https://05303abe-b842-4c47-ab8c-db2af9334f57.mock.pstmn.io	https://05303abe-b842-4c47-ab8c-db2af9334f57.mock.pstmn.io

So the complete request Mock URL should be: <https://05303abe-b842-4c47-ab8c-db2af9334f57.mock.pstmn.io/user/home>(represented by {{url}}/user/home in the address bar in Step 5). We have appended /user/home at the end of the url value since it is the Request Path we have set for the Mock Server in Step2.

**Step 7:** We have seen that the Response Body is in text format. We can get the response in JSON format as well. To achieve this select the option Save as example from the Save Response dropdown.

The screenshot shows the Postman interface with the "Body" tab selected. The response body contains the text "1 This is Postman Tutor in Tutoraispoint". In the top right corner, there is a "Save Response" dropdown menu. The "Save as example" option is highlighted with an orange border.

**Step 8:** Provide an Example name and select JSON from the Response Body section.

The screenshot shows the Postman interface with the following details:

- Header Bar:** Shows the URL `GET /user/home` and a placeholder `e.g. /user/home Example`.
- Collection Name:** `/user/home Example` (highlighted with an orange box).
- Example Request:** A GET request to `{{url}}//user/home`. The "Params" tab is selected.
- Query Params:** A table with one row containing a KEY "Key".
- Content Type Selection:** A dropdown menu showing options: **JSON** (highlighted with an orange box), XML, HTML, Text, and Auto.
- Example Response:** The "Body" tab is selected. It contains the text "1 This is Postman Tutori in Tutoraispoint".
- Preview Options:** Buttons for Pretty, Raw, Preview, and HTML.

**Step 9:** Add the below Response Body in JSON format. Then click on Save Example.

```
{  
  "name": "Tutorialspoint",  
  "subject": "Postman"  
}
```

The screenshot shows the Postman interface with a 'Mock Server' tab selected. A 'GET /user/home Example' tab is open. The 'Body' tab is selected, showing a JSON response with four lines of code. The second line, containing the key 'name' and its value 'Tutorialspoint', is highlighted with an orange box.

```

1 [
2   "name": "Tutorialspoint",
3   "subject": "Postman"
4 ]
    
```

**Step 10:** Finally, send the GET request on the same endpoint, and we shall receive the same Response Body as we have passed in the Example request.

The below image shows Response is in HTML format:

The screenshot shows the Postman interface with a 'Mock Server' tab selected. A 'GET /user/home' request is made. The 'Body' tab is selected, showing a JSON response with four lines of code. The second line, containing the key 'name' and its value 'Tutorialspoint', is highlighted with an orange box.

```

1 [
2   "name": "Tutorialspoint",
3   "subject": "Postman"
4 ]
    
```

The below image shows Response is in JSON format:

The screenshot shows the Postman interface with the following details:

- Request URL:** GET /user/home
- Headers:** Authorization (set to Bearer token)
- Query Params:** Key: Value, Description: Description
- Body:** JSON response (Pretty, Raw, Preview, Visualize) showing:
 

```

1 [
2   {
3     "name": "Tutorialspoint",
4     "subject": "Postman"
5   }
6 ]

```
- Status:** 200 OK, Time: 525 ms, Size: 498 B

## Example Request

Follow the steps given below for Mock Server Creation by example request:

**Step 1:** Create a Collection and add a request to it.

The details on how to create a Collection is discussed in detail in the Chapter – Postman Create Collections.

The screenshot shows the Postman Collections view with the following structure:

- Collections:**
  - Collection1 (4 requests)
  - Mock Server (1 request)
    - Get Request 1

**Step 2:** Add the endpoint <https://postman-echo.com/get?test=123> and send a GET request.

The screenshot shows the Postman interface with a successful GET request to `https://postman-echo.com/get?test=123`. The response status is 200 OK, time is 215 ms, and size is 828 B. The response body is a JSON object:

```

1  {
2    "args": {
3      "test": "123"
4    },
5    "headers": {
6      "x-forwarded-proto": "https",
7      "x-forwarded-port": "443",
8      "host": "postman-echo.com",

```

**Step 3:** From Response Body, select the option **Save as Example** from the Save Response dropdown.

The screenshot shows the Postman interface with the 'Save Response' dropdown open, highlighting the 'Save as example' option.

**Step 4:** Give an **Example name** and click on the **Save Example** button.

The screenshot shows the Postman interface with an example named 'Example Get Request1' created. The example name is highlighted with a red box. The example details show a GET request to `https://postman-echo.com/get?test=123` with a parameter 'test' set to '123'.

**Step 5:** Click on the Collection name **Mock Server** (that we have created) and click on the **Mock tab**. Then, click on Create a mock server.

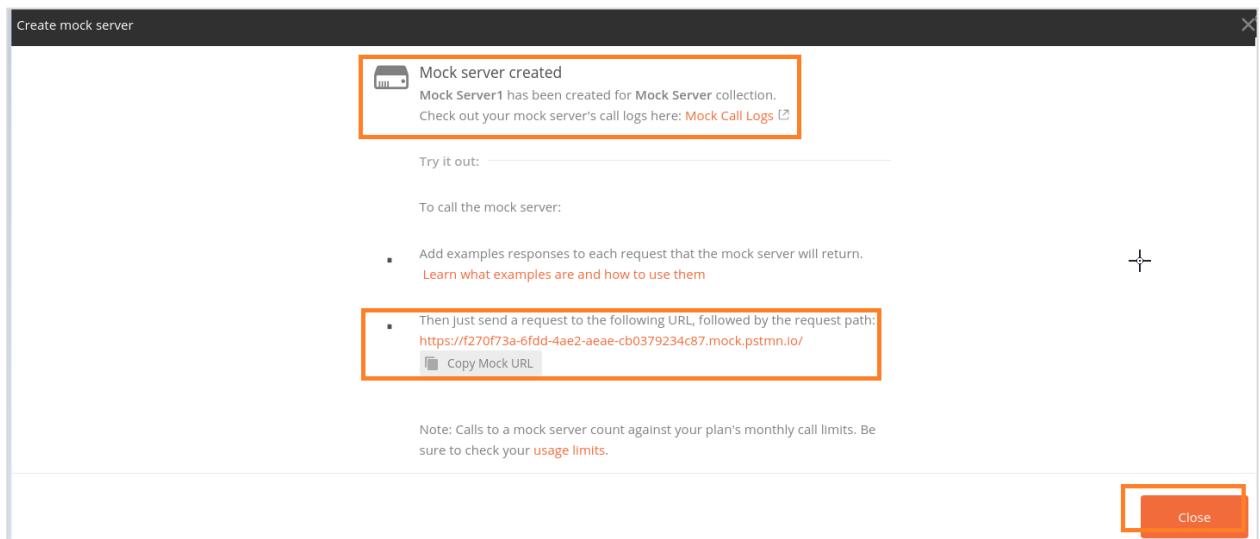
The screenshot shows the Postman interface. On the left, there's a sidebar with tabs for History, Collections (which is selected and highlighted in orange), APIs, and Trash. Below these are buttons for '+ New Collection' and 'Trash'. Under the Collections tab, there's a list of collections: 'Collection1' (4 requests) and 'Mock Server' (1 request). The 'Mock Server' item is highlighted with an orange border. On the right, the main area is titled 'Mock Server' and displays the message 'This collection is not linked to any API'. It has buttons for Share, Run, View in web, and a more options menu. Below these are tabs for Documentation, Monitors, Mocks (which is also highlighted in orange), and Changelog. A call-to-action button 'Learn how to mock your requests' is shown with a globe icon. The central text 'This collection is not being mocked' is displayed above a detailed explanation: 'A mock lets you simulate endpoints and their corresponding responses in a Collection without actually spinning up a back end.' At the bottom right of this section is a button labeled 'Create a mock server'.

**Step 6:** The Create mock server pop-up comes up. Provide a name to the Mock Server and then click on the Create Mock Server button.

Please note: We can make a Mock Server private or public. To make a Mock Server private, we have to check the checkbox **Make this mock server private**. Then, we need to utilise the Postman API key.

The screenshot shows the 'Create mock server' dialog box. At the top, it says 'Name your mock server' with a text input field containing 'Mock Server1'. Below that is a dropdown for 'Select a version tag' with 'CURRENT' selected. A note says 'Version tags allow you to link the mock sever to a specific API version.' Next is a dropdown for 'Select an environment (optional)' with 'No Environment' selected. A note says 'Environments are groups of variables. The mock server will use the variable values from the selected environment in the request and response.' Below these is a checkbox 'Make this mock server private' which is checked and highlighted with an orange border. A note next to it says 'Note: To call a private mock server, you'll need to add an x-api-key header to your requests. See how to [generate a Postman API key](#).' At the bottom left is a checkbox 'Add a delay before sending response'. At the bottom right are buttons for 'Cancel' and 'Create Mock Server' (which is highlighted in orange).

**Step 7:** The message – Mock server created shall come up. Also, we shall get the Mock URL. We can copy it with the Copy Mock URL button. Then, click on **Close**.



**Step 8:** The Mock Server which we have created gets reflected under the Mock tab in the Collections sidebar. Click on the same.

**Step 9:** We shall add a new request and paste the URL we have copied in Step 7. To send a GET request, we shall append the value - /get at the end of the pasted URL.

For example, here, the Mock URL generated is: <https://f270f73a-6fdd-4ae2-aaeae-cb0379234c87.mock.pstmn.io>.

Now to send a GET request, the endpoint should be: <https://f270f73a-6fdd-4ae2-aaeae-cb0379234c87.mock.pstmn.io/get>.

The screenshot shows the Postman interface with three requests listed in the header:

- GET Get Request 1
- Example Get Request1
- GET https://f270f73a-6fdd-4ae2-aea... (highlighted)

The selected request is titled "Untitled Request". The URL is https://f270f73a-6fdd-4ae2-aea... (highlighted). The "Params" tab is selected. The response status is 200 OK, and the time taken is 480 ms.

The response body is displayed in JSON format:

```

1 {
2   "args": {
3     "test": "123"
4   },
5   "headers": {
6     "x-forwarded-proto": "https",
7     "x-forwarded-port": "443",
8     "host": "postman-echo.com",
9     "x-amzn-trace-id": "Root=1-6075e520-7fdc330909193c6a019deeee",
10    "user-agent": "PostmanRuntime/7.26.8",
11    "accept": "*/*",
12    "postman-token": "d604ef72-904f-4554-9568-2e885bfea674",
13    "accept-encoding": "gzip, deflate, br",
14    "cookie": "Cookie_4=value; sails.sid=s%3Aqnptcm2qhx24FszU-Vbb14ojoTtUkWk.co234SEVx6eQIEcNmfpL52mxXkDUdeKHMPDNbiyICY"
15 }

```

The Response Body received by mocking the server is the same as the Example request.

Response obtained in the Example request is as follows:

The screenshot shows the Postman application interface. At the top, there are tabs for 'Get Request 1' and 'Example Get Request1'. The 'Example Get Request1' tab is active, indicated by an orange border around its title bar and content area.

**EXAMPLE REQUEST:**

A GET request is defined with the URL <https://postman-echo.com/get?test=123>. The 'Params' tab is selected, showing a table with one row where 'test' has a value of '123' and another row for 'Key' with a 'Value' column.

**EXAMPLE RESPONSE:**

The response body is displayed in JSON format, highlighted with an orange box. The JSON object contains the following structure:

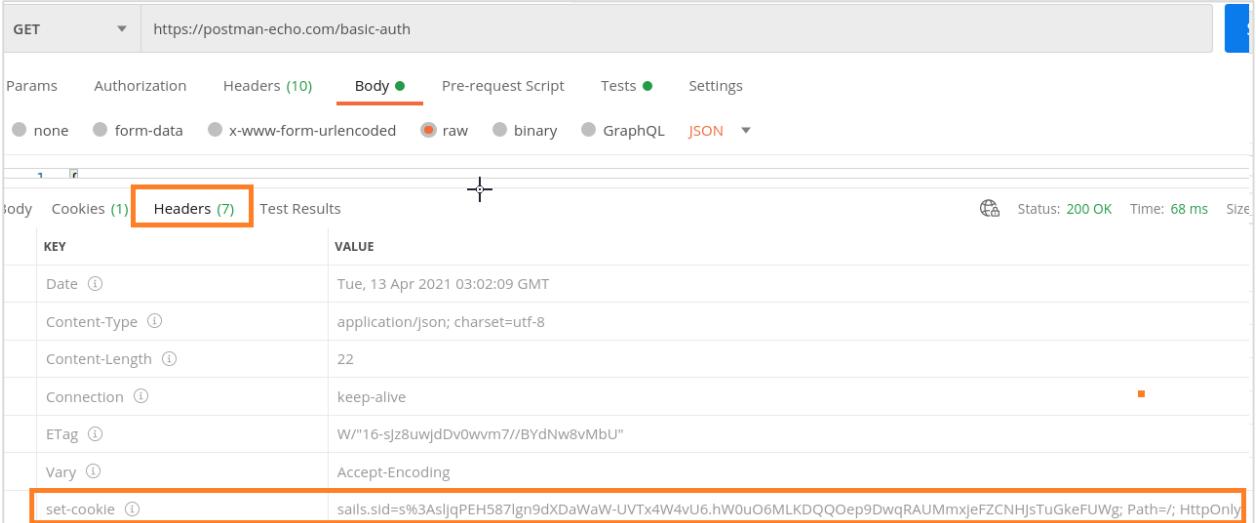
```
1 {  
2   "args": [  
3     "test": "123"  
4   ],  
5   "headers": {  
6     "x-forwarded-proto": "https",  
7     "x-forwarded-port": "443",  
8     "host": "postman-echo.com",  
9     "x-amzn-trace-id": "Root=1-6075e520-7fdc330909193c6a019deeee",  
10    "user-agent": "PostmanRuntime/7.26.8",  
11    "accept": "*/*",  
12    "postman-token": "d604ef72-904f-4554-9568-2e885bfea674",  
13    "accept-encoding": "gzip, deflate, br",  
14    "cookie": "Cookie_4=value; sails.sid=s%3AqwnptcM2qhx24FszU-Vbb14ojoTtUkWK.co234SEVx6eQIEcNmfpL52mxXkDUdeKHMPDNbiyiCY",  
15  },  
16  "url": "https://postman-echo.com/get?test=123"  
17 }
```

# 16. Postman — Cookies

The cookies are information sent by the server and stored in the browser. As soon as a request is sent, the cookies are returned by the server. In Postman, the cookies are mentioned under the Headers and Cookies tab in the Response.

Let us apply a GET request on an endpoint and find the cookies.

In the Headers tab, the cookie sent by the server is set with the key - set-cookie.

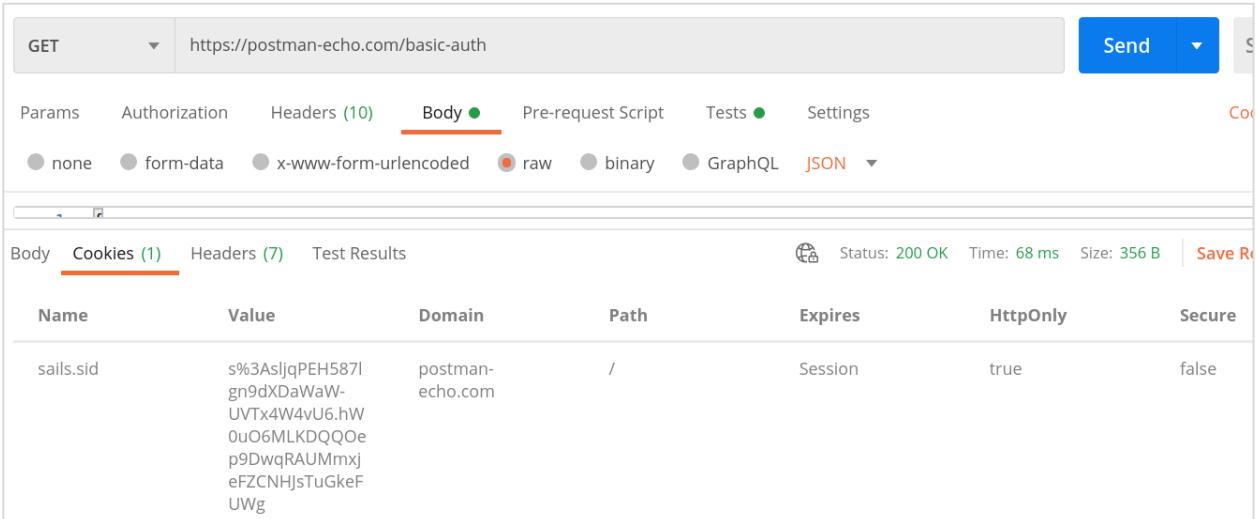


The screenshot shows the Postman interface with a GET request to <https://postman-echo.com/basic-auth>. The 'Headers' tab is selected, displaying the following headers:

KEY	VALUE
Date	Tue, 13 Apr 2021 03:02:09 GMT
Content-Type	application/json; charset=utf-8
Content-Length	22
Connection	keep-alive
ETag	W/"16-sjz8uwjdDv0wvm7//BYdNw8vMbU"
Vary	Accept-Encoding
set-cookie	sails.sid=s%3AsljqPEH587Iggn9dXDaWaW-UVTx4W4vU6.hW0uO6MLKDQQOep9DwqRAUMmxjeFZCNHjsTuGkeFUWg; Path=/; HttpOnly

The 'set-cookie' row is highlighted with an orange border.

In the Cookies tab, the same cookie details will also get displayed.



The screenshot shows the Postman interface with the same GET request. The 'Cookies' tab is selected, displaying the following cookie:

Name	Value	Domain	Path	Expires	HttpOnly	Secure
sails.sid	s%3AsljqPEH587Iggn9dXDaWaW-UVTx4W4vU6.hW0uO6MLKDQQOep9DwqRAUMmxjeFZCNHjsTuGkeFUWg	postman-echo.com	/	Session	true	false

## Cookies Management

In Postman, we can manage cookies by addition, deletion, and modification of cookies. Under the Params tab, we have the Cookies link to perform operations on cookies.

The screenshot shows the Postman interface with the 'Cookies' tab highlighted in orange at the top right. Below it, the 'Params' tab is active, showing a table for 'Query Params' with one row: 'Key' under 'KEY' and 'Value' under 'VALUE'. There are also tabs for 'Authorization', 'Headers (10)', 'Body', 'Pre-request Script', 'Tests', and 'Settings'. A 'Bulk Edit' button is visible in the top right corner of the table.

Click on the Cookies link. MANAGE COOKIES pop-up shall open where all the available cookies are present with the option to add and delete a cookie.

The 'MANAGE COOKIES' pop-up window is shown. It has a search bar at the top with placeholder text 'Type a domain name' and an 'Add' button. Below it, there's a message 'Sync cookies directly from your browser with Interceptor' with a 'Start Lesson' button. The main area lists cookies from different domains:

- typicode.com**: 1 cookie. One cookie named '\_cfduid' is listed with an 'X' button and a '+ Add Cookie' button.
- postman-echo.com**: 1 cookie. One cookie named 'sails.sid' is listed with an 'X' button and a '+ Add Cookie' button.

## Cookies Addition

Follow the steps given below for adding cookie in Postman:

**Step 1:** Click on the Add Cookie button. A text box shall open up with pre-existing values inside it. We can modify its values and then click on Save.

The 'Cookie' configuration dialog box is shown. It lists two cookies: 'sails.sid' and 'Cookie\_3'. Below them is a large text input field containing the following cookie value:

```
Cookie_Postman=value; Path=/; Domain=.postman-echo.com; Expires=Wed, 13 Apr 2022  
03:23:46 GMT;
```

At the bottom right are 'Cancel' and 'Save' buttons, with 'Save' being highlighted in orange.

**Step 2:** Send the request again to the server.

The Response code obtained is 200 OK. Also, the Cookies tab in the Response now shows the newly added cookie – Cookie\_Postman.

Name	Value	Domain	Path	Expires	HttpOnly	Secure
Cookie_Postman	value	postman-echo.com	/	Wed, 13 Apr 2022 03:23:46 GMT	false	false
sails.sid	s%3A_GDZNBL7UHTIb9z_7B54w36QTWnu2YGk.A48l1Zqczv%62Bqb%2BxML%2B8Ql4LIXOU3rl820c%2FDi	postman-echo.com	/	Session	true	false

## Access Cookies via Program

Cookies can be handled programmatically without using the GUI in Postman. To work with cookies, we have to first generate a Cookie jar. It is an object which has all the cookies and methods to access them.

### Cookie Jar Creation

The syntax for Cookie Jar Creation is as follows:

```
const c = pm.cookies.jar();
```

### Cookie Creation

We can create a cookie with the `.set()` function. It accepts URL, name of cookie, value of cookie as parameters.

The syntax for cookie creation is as follows:

```
const c = pm.cookies.jar();
c.set(URL, name of cookie, value of cookie, callback(error, cookie));
```

### Get Cookie

We can get a cookie with the `.get()` function. It accepts the URL, name of cookie as parameters. It yields the cookie value.

The syntax for getting a cookie is as follows:

```
const c = pm.cookies.jar();
c.set(URL, name of cookie, value of cookie, callback(error, cookie));
c.get(URL, name of cookie, callback(error, cookie));
```

## Get All Cookies

We can get all cookies of a specific URL within a Cookie jar with the .getAll() function. It accepts a URL as a parameter. It yields all the cookie values for that URL.

The syntax for getting all cookies is as follows:

```
const c = pm.cookies.jar();
c.set(URL, name of first cookie, value of first cookie, callback(error,
cookie));
c.set(URL, name of second cookie, value of second cookie, callback(error,
cookie));
c.getAll(URL, callback(error, cookie));
```

## Delete Cookie

We can delete a cookie with the .unset() function. It accepts the URL, name of cookie to be deleted as parameters.

The syntax for deleting cookie is as follows:

```
const c = pm.cookies.jar();
c.set(URL, name of cookie, value of cookie, callback(error, cookie));
c.unset(URL, name of cookie, callback(error, cookie));
```

## Delete All Cookies

We can delete all cookies of a specific URL with the .clear() function. It accepts a URL as a parameter. It removes all the cookie values for that URL.

The syntax for deleting all cookies is as follows:

```
const c = pm.cookies.jar();
c.set(URL, name of first cookie, value of first cookie, callback(error,
cookie));
c.set(URL, name of second cookie, value of second cookie, callback(error,
cookie));
c.clear(URL, callback(error, cookie));
```

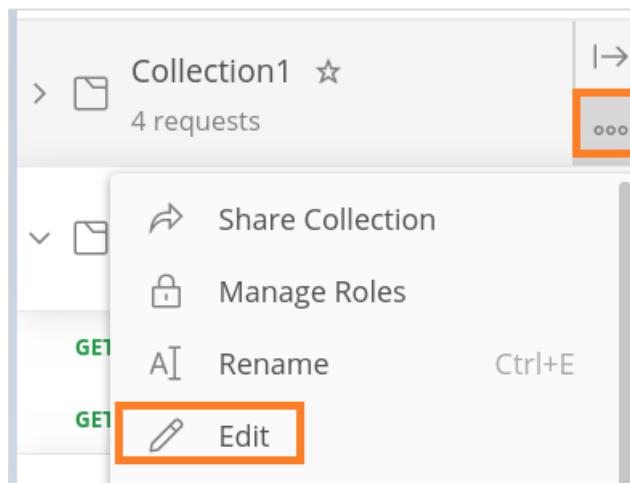
# 17. Postman — Sessions

A session is a temporary fold that stores values of variables. They are used for the present instance and have a local scope. In Postman, we can modify the session variable value to share workspace among teams.

Postman gives the feature of local session share. Even if a Collection can be shared among teams, the sessions are never shared. Different tokens have to be generated while a task is to be carried out in a team structure.

A session has a local scope for a user within his Workspace and any modifications he makes shall not be reflected in the server. In Postman, a session can store Environment variables, global variables and so on.

We can assign current values to Collection variables and to the global and Environment variables. To assign a current value to the Collection, click on the three dots appearing beside the Collection name, then click on Edit.



In the EDIT COLLECTION pop-up, move to the Variables tab.

A screenshot of the 'EDIT COLLECTION' dialog box. At the top, it says 'EDIT COLLECTION'. Below that is a 'Name' field containing 'Collection1'. Underneath are tabs for 'Description', 'Authorization', 'Pre-request Scripts', 'Tests', and 'Variables'. The 'Variables' tab is selected and highlighted with an orange box. Below the tabs, it says 'These variables are specific to this collection and its requests. [Learn more about collection variables.](#)' There is a table with three columns: 'VARIABLE', 'INITIAL VALUE', and 'CURRENT VALUE'. The 'CURRENT VALUE' column is also highlighted with an orange box. At the bottom right of the table are buttons for '... Persist All' and 'Reset All'.

The CURRENT VALUE is local to the user and never in sync with the server of Postman. We can also replace or modify the INITIAL VALUE with CURRENT VALUE.

Also, it must be remembered that the INITIAL VALUE gets impacted only if we apply the option Persist on a variable. After that, it gets in sync with the server of Postman.

# 18. Postman — Newman Overview

Newman is a potential command-line runner used in Postman. We can execute and verify a Postman Collection from the command-line as well. Newman has features which are consistent with Postman.

We can run the requests within a Collection from Newman in the same way as in the Collection Runner. Newman can occupy both the NPM registry and GitHub. We can also perform Continuous Integration or Deployment with Newman.

A status code of 0 is thrown by Newman if all the execution is done without errors. The Continuous Integration tools read the status code and accordingly fail/pass a build.

We can add the flag --bail to the Newman to pause on an error encountered in a test with a status code of 1. This can be interpreted by the CI tools. Newman is based on node.js and uses npm as a package manager.

## Newman Installation

The installation of Newman requires Node.js and npm. Follow the steps given below to install Newman:

**Step 1:** Navigate to the link: <https://nodejs.org/en/download/current/> for downloading the Node.js.

The screenshot shows the Node.js download page. At the top, there's a navigation bar with links for HOME, ABOUT, DOWNLOADS, DOCS, GET INVOLVED, SECURITY, CERTIFICATION, and NEWS. Below this is a search bar and a progress bar indicating 67% completion. The main content area is titled 'Downloads' and shows the 'Latest Current Version: 15.14.0 (includes npm 7.7.6)'. It features two main sections: 'LTS Recommended For Most Users' (green background) and 'Current Latest Features' (dark green background). Under 'LTS', there are links for 'Windows Installer (.msi)', 'Windows Binary (.zip)', 'macOS Installer (.pkg)', 'macOS Binary (.tar.gz)', 'Linux Binaries (x64)', 'Linux Binaries (ARM)', and 'Source Code'. Under 'Current', there are links for '32-bit' and '64-bit' versions of the Windows, macOS, and Linux installers. A table at the bottom provides a detailed comparison of the available binary formats for each platform.

	32-bit	64-bit
Windows Installer (.msi)		
Windows Binary (.zip)	32-bit	64-bit
macOS Installer (.pkg)		64-bit
macOS Binary (.tar.gz)		64-bit
Linux Binaries (x64)		64-bit
Linux Binaries (ARM)	ARMv7	ARMv8
Source Code	node-v15.14.0.tar.gz	

**Step 2:** Once the download is completed, execute the below command to verify that the Node.js is installed properly.

The command for verifying the installation in **Windows** is as follows:

```
node --v
```

The command for verifying the installation in **Linux** is as follows:

```
node --version
```

The below image shows the version v10.15.2 of the Node.js is installed in the system.

```
osboxes@osboxes:~/Desktop$ node --version
v10.15.2
osboxes@osboxes:~/Desktop$
```

**Step 3:** The npm is allocated with Node.js so once we download the Node.js then npm gets downloaded by default. To verify if npm is available in our system, run the below command:

The command for verifying the installation in **Windows** is as follows:

```
npm --v
```

The command for verifying the installation in **Linux** is as follows:

```
npm --version
```

The below image shows the version 5.8.0 of the npm installed in the system:

```
osboxes@osboxes:~/Desktop$ node --version
v10.15.2
osboxes@osboxes:~/Desktop$ npm --version
5.8.0
osboxes@osboxes:~/Desktop$
```

**Step 4:** For installation of Newman, run the below mentioned command:

```
npm install -g newman.
```

**Step 5:** To verify the version of newman, run the below commands:

The command for verifying the installation in **Windows** is as follows:

```
newman --v
```

The command for verifying the installation in **Linux** is as follows:

```
newman --version
```

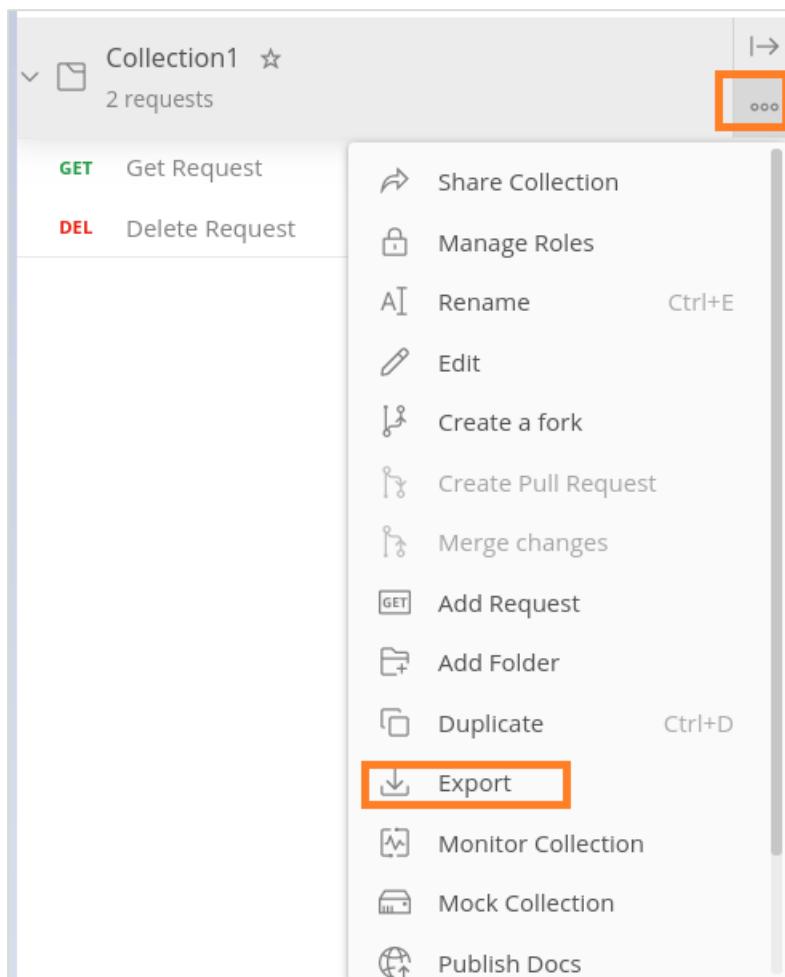
# 19. Postman — Run Collections using Newman

To run Collections using Newman, we have to first launch the Postman application and click on the three dots available beside the Collection name. The details on how to create a Collection are discussed in detail in the Chapter – Postman Create Collections.

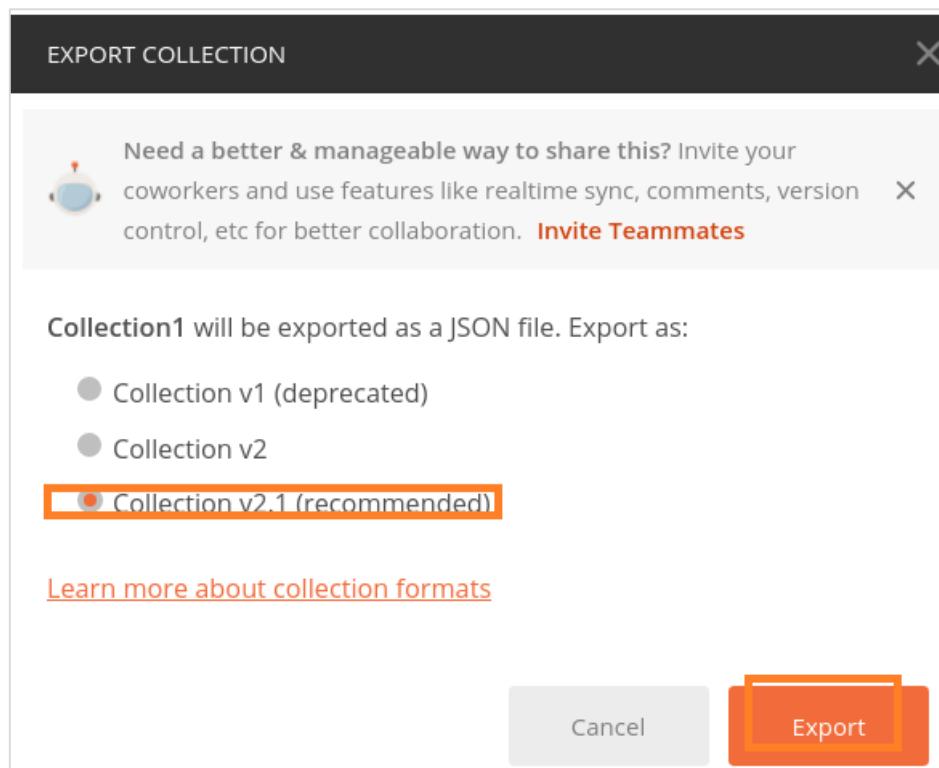
## Run Collections

Follow the steps given below to run collections using Newman:

**Step 1:** Click on Export.

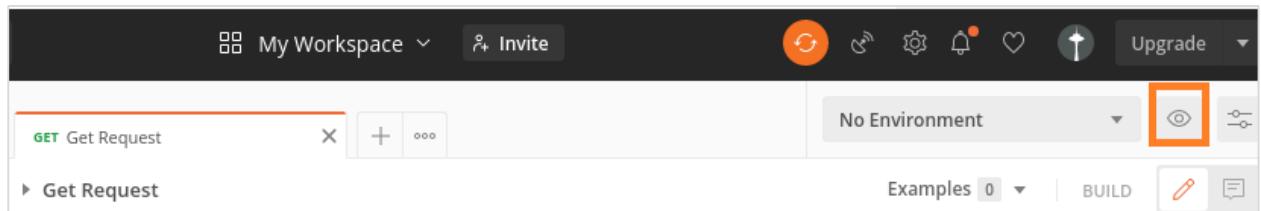


**Step 2:** Select the option **Collection v2.1(recommended)** from the EXPORT COLLECTION pop-up. Click on **Export**.

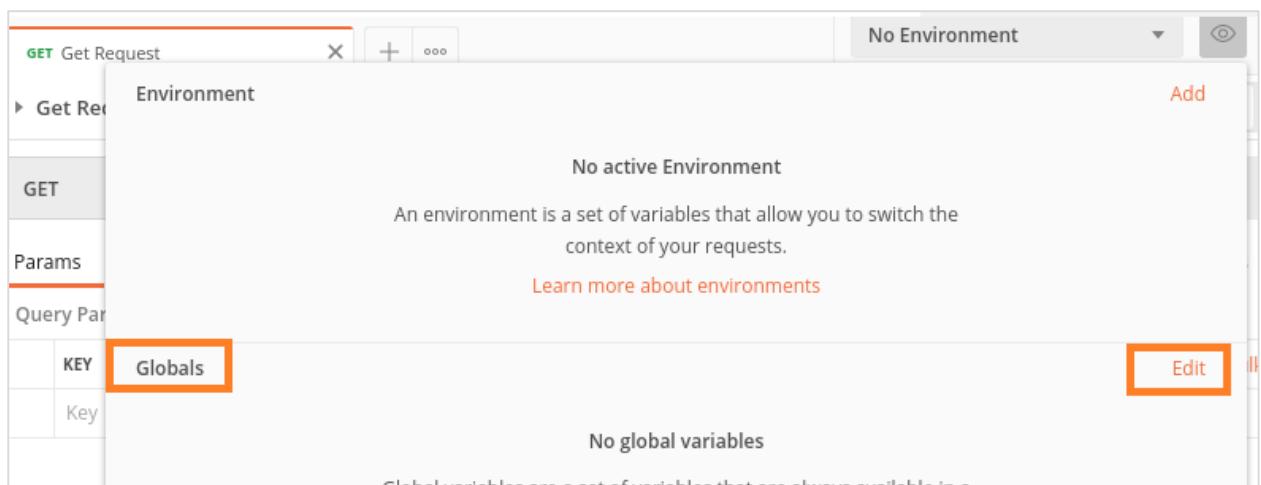


**Step 3:** Choose a location and then click on **Save**.

**Step 4:** Next, we shall export the Environment. Click on the **eye icon** to the right of the No Environment dropdown.

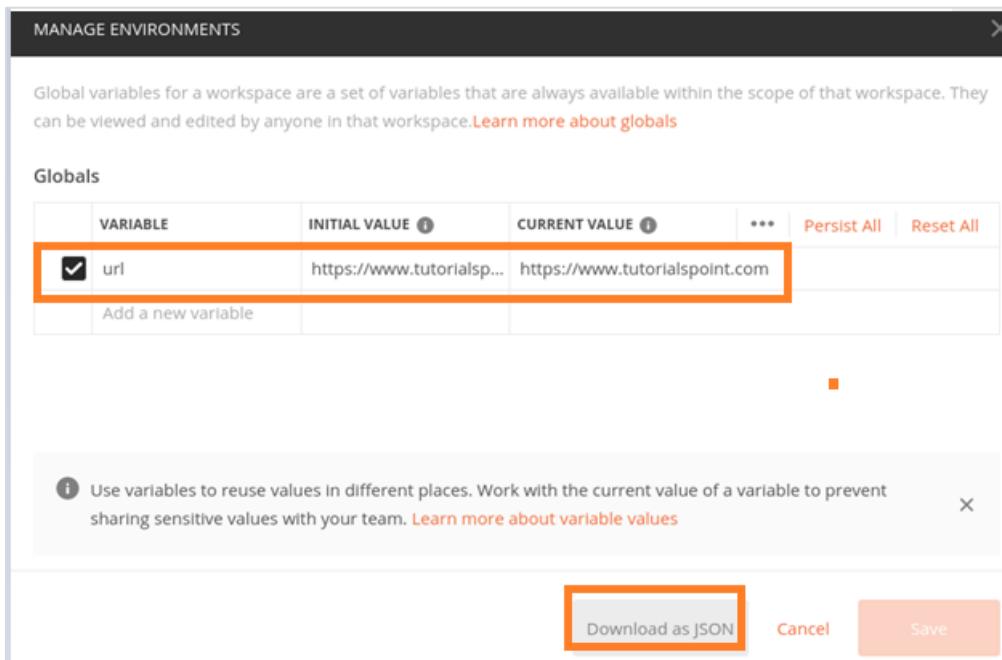


**Step 5:** Click on the Edit link in the Globals section.



**Step 6: MANAGE ENVIRONMENTS** pop-up comes up. Enter URL for the VARIABLE field and <https://www.tutorialspoint.com> for INITIAL VALUE. Then, click on Download as JSON.

Finally, choose a preferred location and click on **Save**.



**Step 7:** Export the Environment to the same location where the Collection resides.

**Step 8:** From the command-line move to the directory path where the Collection and the Environment is stored. Then, execute the command given below:

```
newman run <"file name">.
```

The file name should always be in inverted quotes; else it shall be taken as a directory name.

## Common command-line arguments for Newman

The common command-line arguments for Newman are given below:

- To execute a Collection in an Environment, the command is as follows:

```
newman run <name of Collection> -e <name of Environment>
```

- To execute a Collection for a number of iterations, the command is as follows:

```
newman run <name of Collection> -n <iteration count>
```

- To execute a Collection with data file, the command is as follows:

```
newman run <name of Collection> --data <name of file> -n <iteration count> -e <name of Environment>
```

- Configure delay time in between requests, the command is as follows:

```
newman run <name of Collection> -d <time of delay>
```

## 20. Postman — OAuth 2.0 Authorization

The OAuth 2.0 is an authorization technique available in Postman. Here, we first obtain a token for accessing the API and then utilise the token to authenticate a request. A token is used to ensure that a user is authorised to access a resource in the server.

If we make an attempt to access a secured URL without the token, a Response code **401 Unauthorized** shall be obtained. To start with, the application passes an authorization request for the end user to access a resource.

As the application allows the user access, it asks for an access token from the server by providing user information. In turn, the server yields an access token. The client can then access the secured data via the access token.