Postman Assertion Library

# Different types of assertions in postman

Postman is the use of assertions, which are validations performed on the responses received from API requests. Assertions help ensure that the API behaves as expected and that the data exchanged is accurate.

* [Status Code Assertions](https://www.geeksforgeeks.org/explain-different-types-of-postman-assertions/#status-code-assertions)
* [Response Body Assertions](https://www.geeksforgeeks.org/explain-different-types-of-postman-assertions/#response-body-assertions)
* [Header Assertions](https://www.geeksforgeeks.org/explain-different-types-of-postman-assertions/#header-assertions)
* [JSON Schema Assertions](https://www.geeksforgeeks.org/explain-different-types-of-postman-assertions/#json-schema-assertions)

## Status Code assertions

To implement a Status Code Assertion:

* 1. Create a request in Postman.
* 2. Go to the “Tests” tab.
* 3. Write the assertion code using the `pm.response` object.

pm.test("Status code is 200", () =>

{

    pm.response.to.have.status(200);

});

## Response body assertions

These assertions validate the content of the response body. You can check for specific values, patterns, or elements within the response.

To implement a Response Body Assertion:

* 1. Access the “Tests” tab.
* 2. Write assertions using the `pm.response` object.

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

pm.test("Response data", () => {

    pm.expect(jsonData).to.be.an("object")

    pm.expect(jsonData.data[0].id).to.eql(1)

     pm.expect(jsonData.data[0].first\_name).to.eql("George")

});

## Validating Headers

Header Assertions validate the presence and values of headers in the API response.

To implement a Header Assertion:

* 1. Navigate to the “Tests” tab.
* 2. Write assertions using the `pm.response` object.

pm.test("Content type validation", () =>

{

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

});

## Validating JSON Schema

Get the schema from a json to schema converter

<https://transform.tools/json-to-json-schema>

Validating schema:

**var** schema = {

  "type": "object",

  "properties": {

    "data": {

      "type": "object",

      "properties": {

        "id": {

          "type": "number"

        },

        "email": {

          "type": "string"

        },

        "first\_name": {

          "type": "string"

        },

        "last\_name": {

          "type": "string"

        },

        "avatar": {

          "type": "string"

        }

      },

      "required": [

        "id",

        "email",

        "first\_name",

        "last\_name",

        "avatar"

      ]

    },

    "support": {

      "type": "object",

      "properties": {

        "url": {

          "type": "string"

        },

        "text": {

          "type": "string"

        }

      },

      "required": [

        "url",

        "text"

      ]

    }

  },

  "required": [

    "data",

    "support"

  ]

}

pm.test("Response body adheres to JSON schema", **function** () {

    pm.response.to.have.jsonSchema(schema);

});

# Chai Assertion library

Chai assertion library is an external javascript library used to write assertions. Compared to what we write directly in javascript, this assertion library needs less time & effort and easy use.

## Writing assertions using Chai

pm.test("If 100 equal to 100", function(){

    pm.expect(100).to.eql(100);

});

--------------------------Include---------------------------------

pm.test("Number included", function(){

pm.expect([100,200,300]).to.include(300);

});

---------------------------Comparing objects--------------

pm.test("Compare Name", function(){

let x= {

"name" : "John"

};

let y= {

"name"  : "John"

};

pm.expect(x).to.eql(y);

});

Postman offers both methods eql() and equal(). But both methods are different. equal() is used to compare the objects, and eql() is used to compare the properties of the objects. In this case, eql() compared the name property of the object.

## Verifying response time using assertion

pm.test("response time is less than 200ms", () => {

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

});

## Verifying response body

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

pm.test ("Test data in resp", ()=>{

pm.expect (jsonData).to.be.an ("Object")

pm.expect (jsonData.data.first\_name).to.be.a ("string")

});

“Object”

“String”

“number”

“array”

## Verifying Arrays using Chai

Endpoint - <https://jsonplaceholder.typicode.com/users>

https://api.github.com/search/repositories?q=bearer&per\_page=10

### To be array

expect([1, 2, 3]).to.be.an(“array”)

### OfSize

pm.expect([1, 2, 3]).to.have.lengthOf(3);

pm.expect([1, 2, 3]).to.not.have.lengthOf(5);

### equalTo

 pm.expect([1,2,3]).to.eql([1, 2, 3]);

 pm.expect([1, 2, 3]).to.not.eql([1, 2, 5]);

### Containing

pm.expect([1,2,3]).to.include(2);

pm.expect([1,2,3]).to.not.include(5)

### Containing all of

pm.expect([1,2,3]).to.include.all.members([1,2]);

### Containing any of

pm.expect([1,2,3]).to.include.oneOf([1,3,4]);

## Verifying response data

Create the below validation

1. Store local variables name and job in pre script
2. Pass these variables in request body
3. Verify name and job in response body using post script

----------------------------Pre script----------------

pm.variables.**set**("name","Jonny")

pm.variables.**set**("job","Engg")

----------------------Post Script-----------------

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

pm.test("Response data", () => {

pm.expect(jasonData.name).to.be.eql(pm.variables.**get**("name"))

});

-------------------------Request body-------------------

{

    "name": "*{{name}}*",

    "job": "*{{job}}*"

}

## Verifying nested Json response

Endpoint - <https://jsonplaceholder.typicode.com/users>

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

**const** person = response.find(i=> i.name==="Leanne Graham")

pm.test("testing simply", ()=>{

    pm.expect(person.address.geo.lat).to.eql("-37.3159")

**var** aa = person.address.geo.lat

    console.log(aa)

})

## Verifying cookies

We can add Tests script around cookies and apply Assertions on them for verification.

https://magento.softwaretestingboard.com/ /search/ajax/suggest/?q=bags

### verify cookie present

 pm.test("Verify Cookie value", **function**(){

pm.expect(pm.cookies.has('form\_key')).to.be.**true**;

})

### Verify cookie value

pm.test("Cookie isLoggedIn has value 1", () => {

  pm.expect(pm.cookies.**get**('Cookie\_1')).to.eql('value');

});

# Running collection

You can manually run the requests in a [collection](https://learning.postman.com/docs/collections/use-collections/create-collections/) or a [folder](https://learning.postman.com/docs/collections/use-collections/manage-collections/#add-folders-to-a-collection).

1. Select **Collections** in the sidebar and select the collection or folder you want to run.
2. Click on the three dots on the collection and select ‘Run Collection’
3. Select the requests
4. Select run manually
5. Click on Run <collectionName>

Choose any configuration options:

* **Iterations** - The number of iterations for your collection run.
* **Delay** - An interval delay in milliseconds between each request.
* **Data** - A [data file](https://learning.postman.com/docs/collections/running-collections/working-with-data-files/) for the collection run.
* **Persist responses for a session** - Log the response headers and bodies so you can review them after running the collection.
* **Turn off logs during run** - Turn off logging to the [Postman Console](https://learning.postman.com/docs/sending-requests/response-data/troubleshooting-api-requests/#debugging-in-the-console) during the collection run.

### Changing order of execution

By default, your requests run in the sequence they're listed in the collection. If you need to change the order of execution, select and drag a request to its new location in the order.

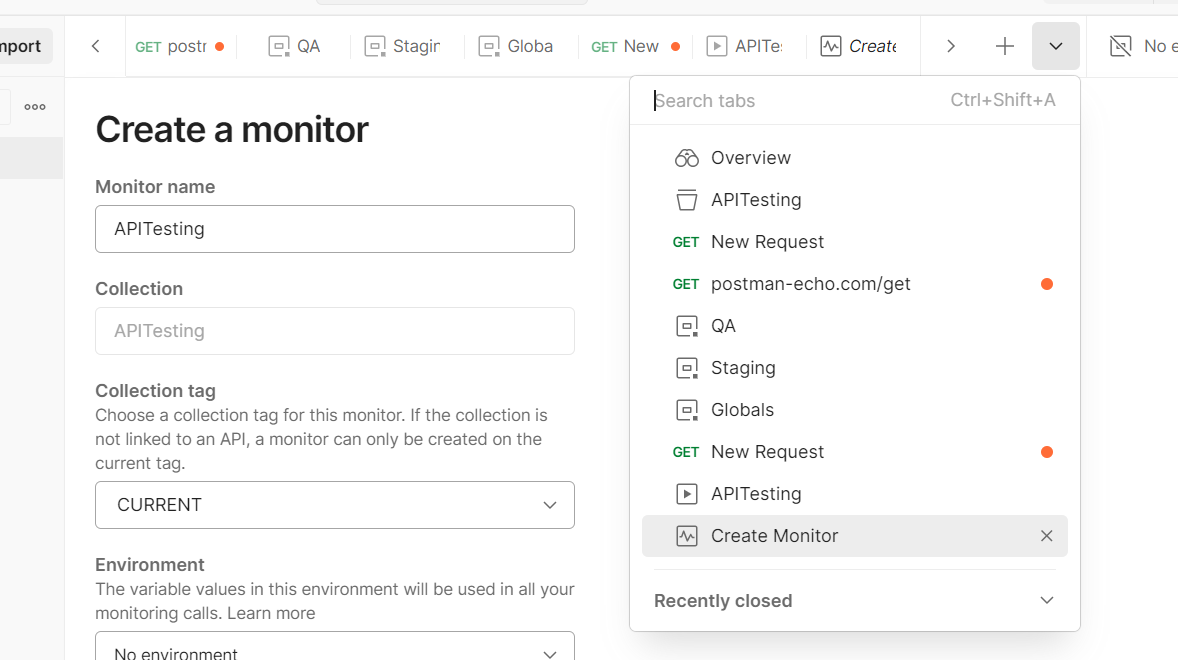
# Postman Monitors

*Postman Monitors* enable you to [continuously check the health and performance of your APIs](https://www.postman.com/api-platform/api-observability/). You can create monitors that run requests in selected collections. Requests can run API test scripts, chain together multiple requests, and more. You can also schedule how often Postman runs monitored collections.

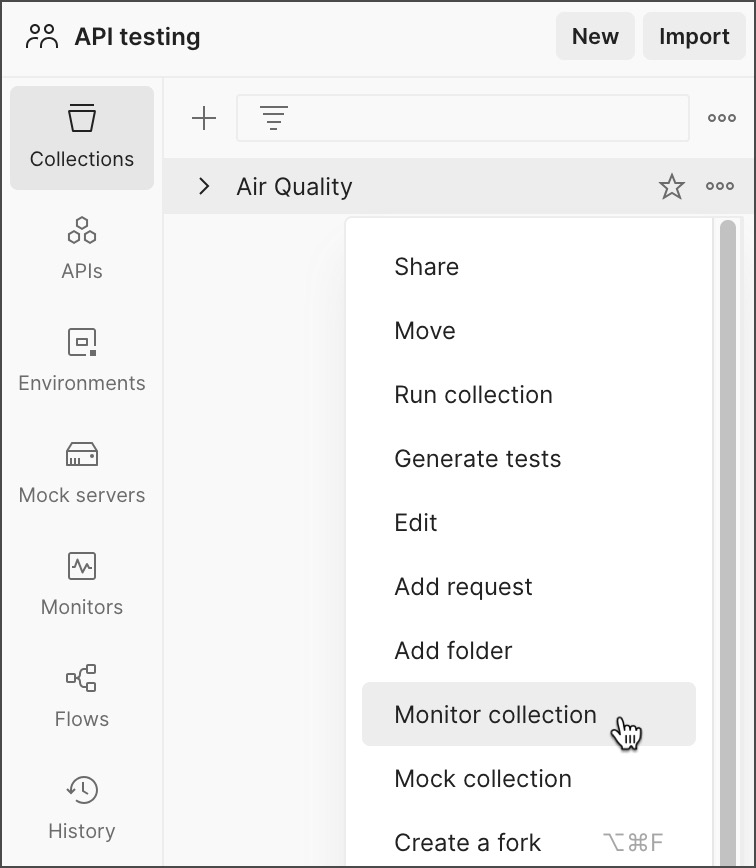
You can [set up monitors](https://learning.postman.com/docs/monitoring-your-api/setting-up-monitor/), enabling you to run a series of requests from the Postman cloud based on a schedule.

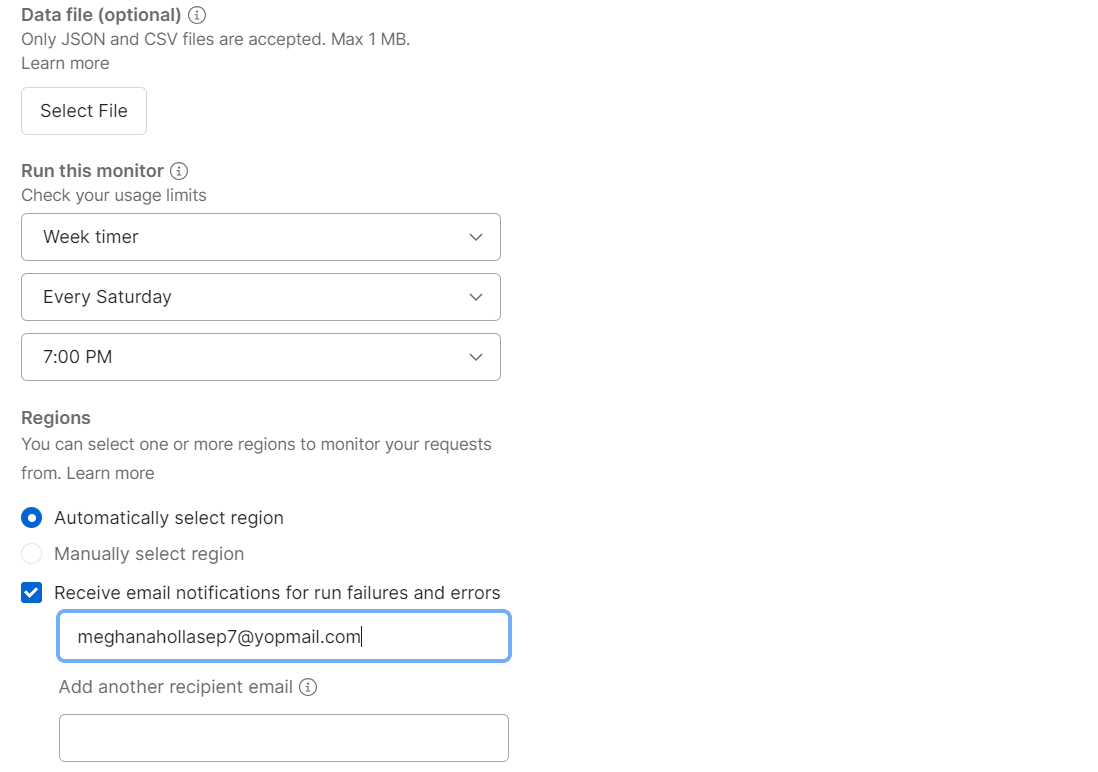
## Setting up a monitor

* To create a monitor from scratch, select **Monitors** in the sidebar, and then select **+**. You will select a collection to run when configuring the new monitor.



* We can also create monitors from collections





## Data Files

Data files can be in CSV or JSON format and must use the following structure:

* **CSV** - The first row of the file must contain the variable names, one variable per column. Each following row is considered a data row, with values for each variable.
* **JSON** - The file must be formatted as an array of objects containing the key-value pairs you want to use as variables. For each pair, the key corresponds to a variable name, and the value will be used for that variable when running the monitored collection.

## View monitor report

You can use the monitor summary to understand how your APIs have performed over time. Each monitor run is represented by a bar in the graph.

[**Individual requests**](https://learning.postman.com/docs/monitoring-your-api/viewing-monitor-results/#individual-requests)

You can select Individual requests to break down your monitor summary into separate requests.

[**Filters**](https://learning.postman.com/docs/monitoring-your-api/viewing-monitor-results/#filters)

You can use filters to identify recurring patterns in your monitoring runs by selecting particular requests, run types, results, and regions

Each run is labelled based on its result:

* **Successful** - Your monitor completed the run with no issues and passed all tests.
* **Failure** - Your monitor completed the run, however one or more tests failed.
* **Error** - Your monitor was unable to complete its run due to an error. An error can occur if there is a syntax error in the code you've written, a network error, or for various other reasons. If you get an error, your [Console Log](https://learning.postman.com/docs/monitoring-your-api/viewing-monitor-results/#console-log) will help you identify what caused it.
* **Abort** - Your monitor timed out because it didn't complete its run within the allotted 10 minutes

# Automating with NewMan

Newman is a command-line Collection Runner for Postman. It enables you to run and test a Postman Collection directly from the command line.

## Installing Newman

Newman is built on Node.js. To run Newman, we should have Node.js installed.

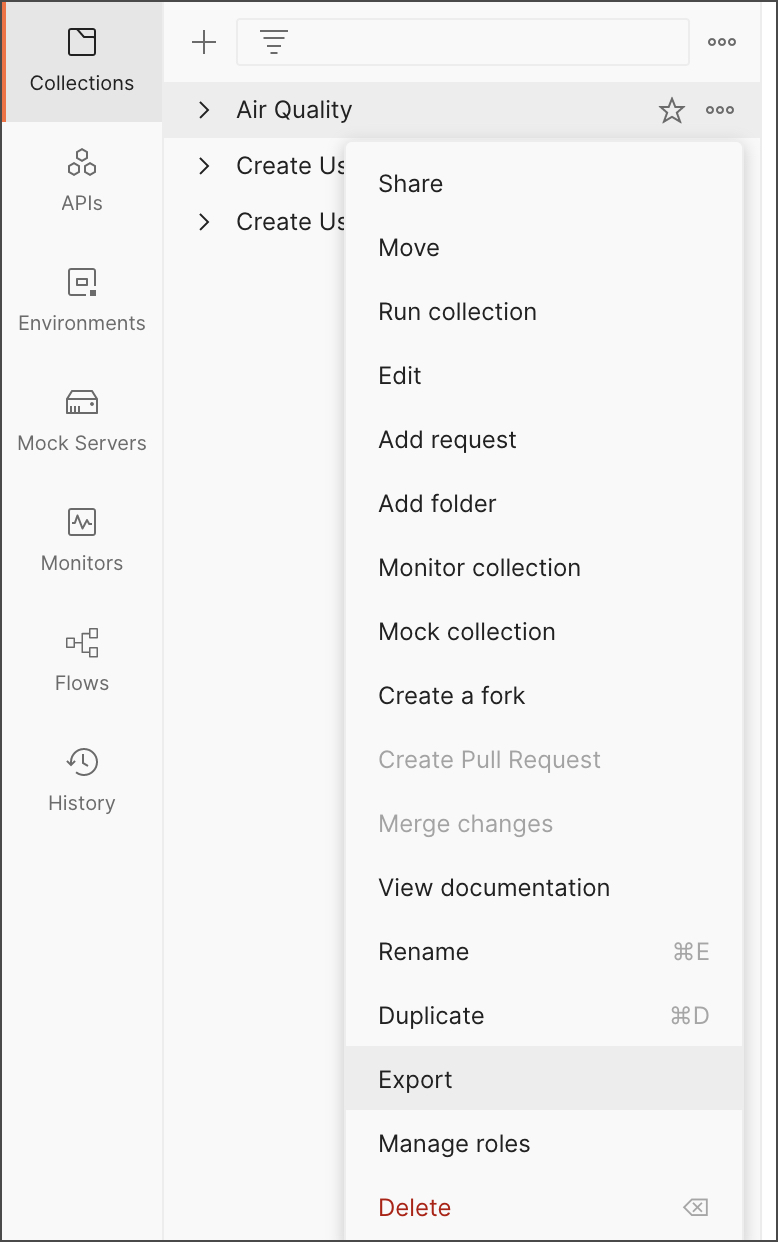
Use the below command to install newman

npm install -g newman

To run the collection using Newman, first we have to export the collection.

[**Export collections**](https://learning.postman.com/docs/getting-started/importing-and-exporting/exporting-data/#export-collections)

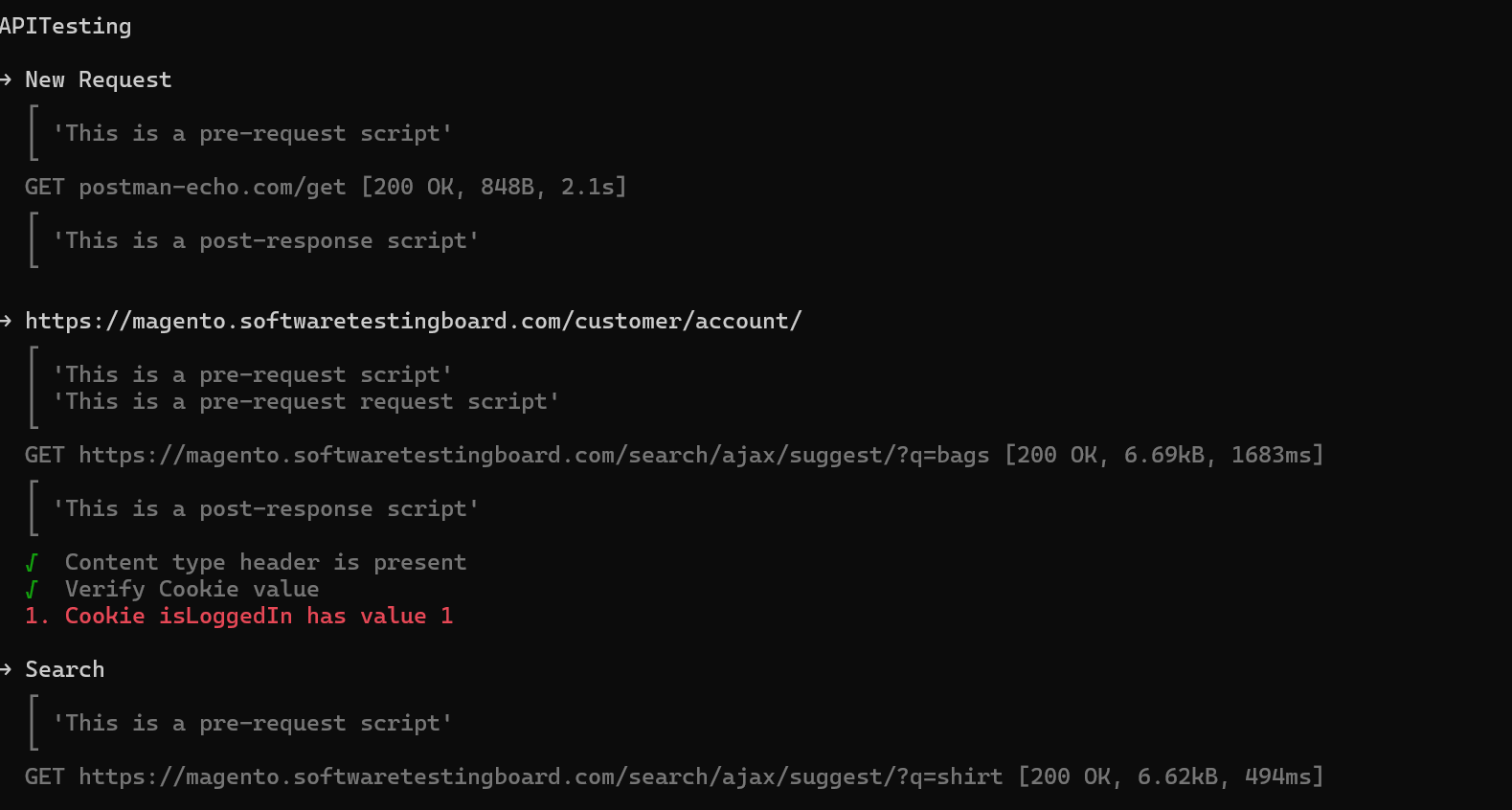
1. Select **Collections** in the sidebar.
2. Select the more actions icon More actions icon next to a collection, then select **Export**.



1. Select the export format for your collection.
2. Select **Export** to download the generated JSON file.

**To run the exported collection**

newman run mycollection.json



## Newman options

| **Option** | **Details** |
| --- | --- |
| -e, --environment [file|URL] | Specify the file path or URL of environment variables. |
| -g, --globals [file|URL] | Specify the file path or URL of global variables. |
| -d, --iteration-data [file] | Specify a data file, either JSON or CSV, to use for iteration as a path to a file or as a URL. |
| -n, --iteration-count [number] | Specify the number of times for the collection to run. Use with the iteration data file. |
| --delay-request [number] | Specify a delay (in milliseconds) between requests. |
| --timeout [number] | Specify the time (in milliseconds) to wait for the entire collection run to complete execution. |
| --timeout-request [number] | Specify the time (in milliseconds) to wait for requests to return a response. |
| --timeout-script [number] | Specify the time (in milliseconds) to wait for scripts to complete execution. |

Below command gives list of all the options available

newman run -h

# Integration of Newman with Jenkins

[Jenkins](https://www.jenkins.io/) is an open source automation server that can act as a continuous integration (CI) server or a continuous delivery (CD) hub. You can integrate your [Postman tests](https://learning.postman.com/docs/tests-and-scripts/write-scripts/test-scripts/) with Jenkins using Newman. This enables you to run a Postman Collection and its API tests directly from Jenkins.

To install NodeJS in Jenkins, do the following:

1. Select **Manage Jenkins**.
2. Under **System Configuration**, select **Plugins**.
3. Select **Available plugins**, and search for "NodeJS".
4. Select the checkbox next to **NodeJS**, and select **Install**.

To install Newman in Jenkins, do the following:

1. Select **Manage Jenkins**.
2. Under **System Configuration**, select **Tools**.
3. Under **NodeJS installations**, select **Add NodeJS**.
4. Enter a name for the NodeJS installation.
5. In **Global npm packages to install**, enter newman.
6. Select **Save**.

To configure Jenkins to run Newman, do the following:

1. On the Dashboard page, select + New Item to create a new job.
2. Select **Freestyle project** from the options, name your project, and select OK.
3. Go to the build Step section.
4. Select **“Windows batch Command”**
5. Enter the below command

newman run “path of json file” --disable-unicode

# Workflow in postman

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.

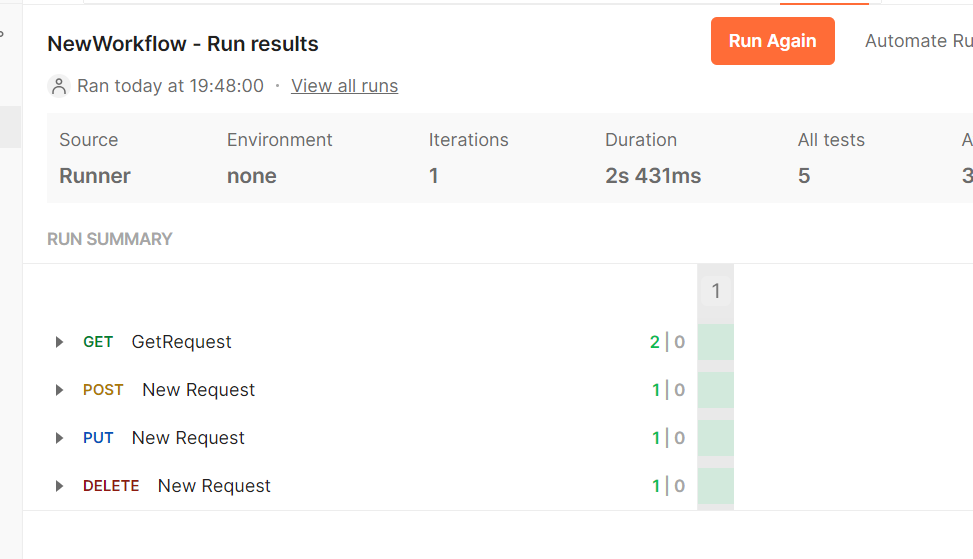
Let us create a Collection (Collection1) with four requests with some post test scripts

Run the collection

Select the request

Click on Run button

 Execution Results show the GET request executed first, followed by POST, then PUT and finally DELETE, as mentioned in the RUN ORDER section



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.

pm.execution.setNextRequest("name of request")

Rerun the collection and check the order of execution.

Use the below line to stop the execution:

pm.execution.setNextRequest(null);

Add below line to skip an execution in pre request script

pm.execution.skipRequest()

## Things to keep in mind when using setNextRequest

* Use setNextRequest() when you run an entire collection
* Use setNextRequest() in pre-request or post-response scripts
* setNextRequest() always executes last
* setNextRequest() scope is limited to the collection

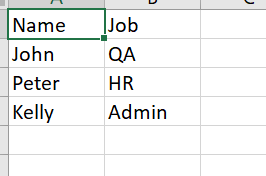
# Running collection using imported data

Postman enables you to import a CSV or JSON file, and use the values from the data file in Collection Runner.

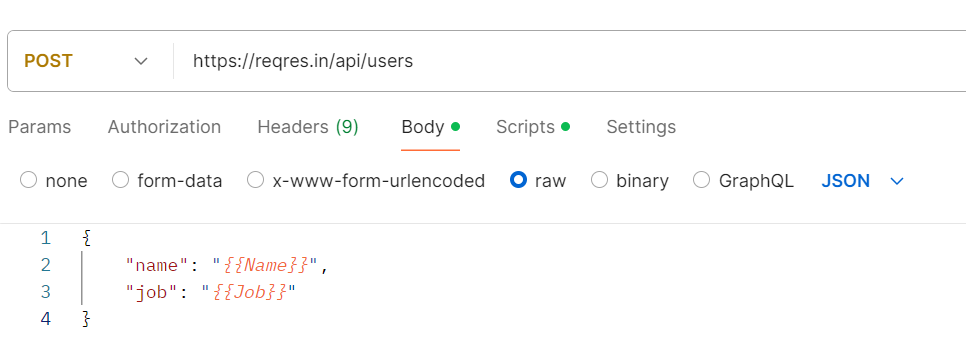
Format the CSV file so that the first row has the variable names you want to use inside the requests. After that, every row will be used as a data row. Each row must have the same number of columns.

## Steps to run using data driven approach

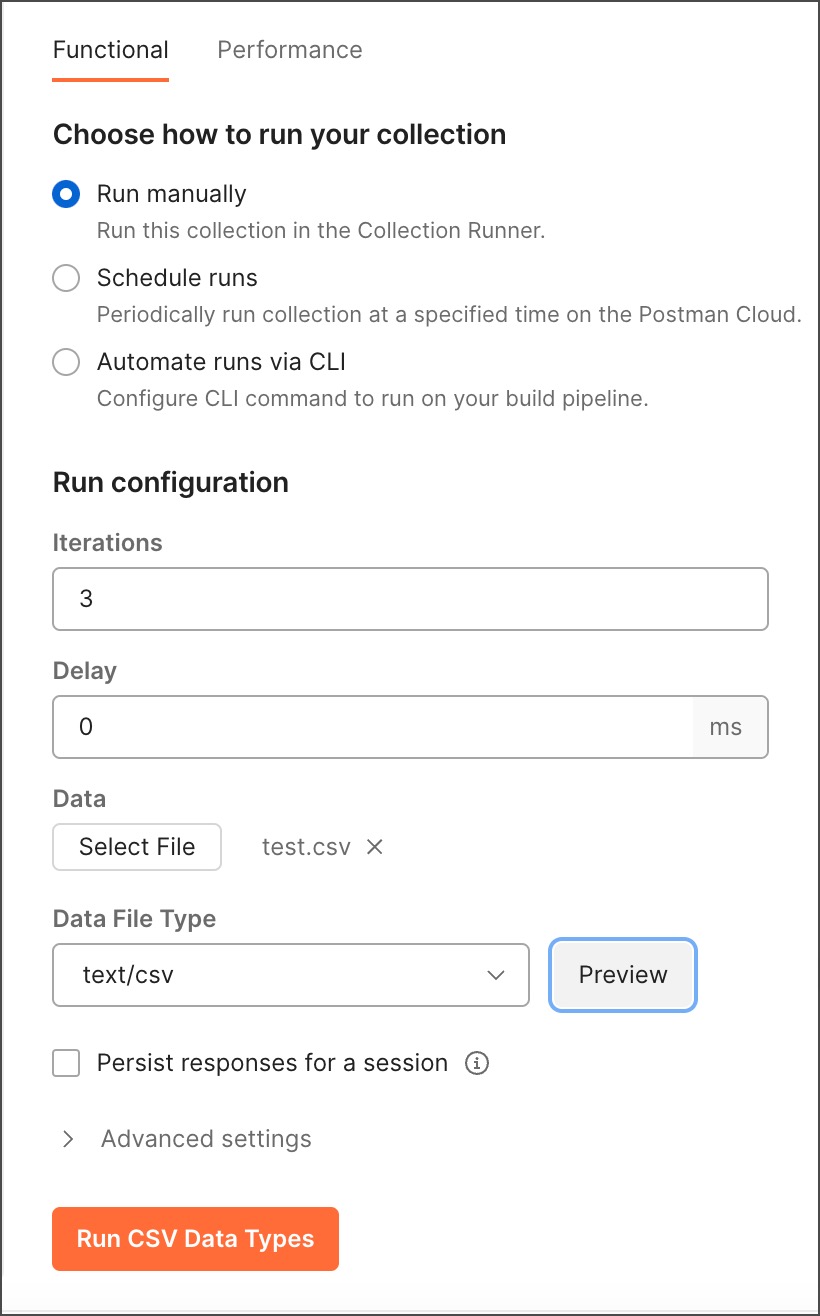
1. Create a test CSV file with multiple data sets



1. Parameterize the request to take data from data table



1. Run the collection
2. Select Run Manually
3. Set the number of iterations
4. Select the CSV file
5. Select the check box ‘persist responses for a session’ to save the response data after execution.



1. After you select your data file, select **Preview** to inspect the data in the file before you start the run.
2. Click on Run