**Postman Newman**

* Newman is a command line Collection Runner for Postman. It allows you to run and test a Postman Collection directly from the command line. It is built with extensibility in mind so that you can easily integrate it with your continuous integration servers and build systems.
* Newman maintains feature parity with Postman and allows you to run collections the way they are executed inside the collection runner in the Postman app.
* Newman resides in the [NPM registry](https://www.npmjs.org/package/newman) and on [GitHub](https://github.com/postmanlabs/newman).

**Install Newman**

* Newman is built on Node.js. To run Newman, make sure you have Node.js installed.
* You can [download and install](http://nodejs.org/download/) Node.js on Linux, Windows, and Mac OSX.
* After you install Node.js, Newman is just a command away. Install Newman from npm globally on your system, which allows you to run it from anywhere.
* $ npm install -g newman

**Run collection via Newman**

* The easiest way to run Newman is to run it with a collection. You can run any collection file from your file system.
* To learn how to export collections to share as a file
* $ newman run mycollection.json
* You can also pass a collection as a URL.
* Your collection probably uses environment variables. To provide an accompanying set of environment variables, [export the template](https://learning.getpostman.com/docs/postman/environments_and_globals/manage_environments/) from Postman and run them with the -e flag.

$ $ newman run https://www.getpostman.com/collections/cb208e7e64056f5294e5 -e dev\_environment.json

* Use the -n option to set the number of iterations to run the collection.

$ newman run mycollection.json -n 10 # runs the collection 10 times

* To provide a different set of data, such as variables for each iteration, you can use the -d to specify a JSON or CSV file.

For example, a data file such as the one shown below runs 2 iterations, with each iteration using a set of variables.

[{

"url": "http://127.0.0.1:5000",

"user\_id": "1",

"id": "1",

"token\_id": "123123",

},

{

"url": "http://postman-echo.com",

"user\_id": "2",

"id": "2",

"token\_id": "899899",

}]

**$ newman run mycollection.json -d data.json**

Here's an example of the CSV file for the above set of variables:

url, user\_id, id, token\_id

http://127.0.0.1:5000, 1, 1, 123123123

http://postman-echo.com, 2, 2, 899899

Newman, by default, exits with a status code of 0 if everything runs well, such as without any exceptions.

Continuous integration tools respond to these exit codes and correspondingly pass or fail a build.

You can use the --bail flag to tell Newman to halt on a test case error with a status code of 1, which can then be picked up by a CI tool or build system.

$ newman run PostmanCollection.json -e environment.json --bail newman

**Importing test results**

The results of all tests and requests can be exported into a file and later imported into Postman for further analysis. Use the JSON reporter and a file name to save the runner output into a file.

$ newman run mycollection.json --reporters cli,json --reporter-json-export outputfile.json

**Note:** Newman allows you to use all [libraries and objects](https://learning.getpostman.com/docs/postman/scripts/postman_sandbox/) that Postman supports to run tests and pre-request scripts.

Library

Newman has been built as a library from the ground up. It can be extended and used in various ways. You can use it as follows in your Node.js code:

var newman = require('newman'); // require newman in your project

// call newman.run to pass `options` object and wait for callback

newman.run({

collection: require('./sample-collection.json'),

reporters: 'cli'

}, function (err) {

if (err) { throw err; }

console.log('collection run complete!');

});

**Integration with Jenkins**

Postman contains a full-featured [testing sandbox](https://learning.getpostman.com/docs/postman/scripts/postman_sandbox/) that lets you write and execute JavaScript based tests for your API. You can then hook up Postman with your build system using [Newman](https://learning.getpostman.com/docs/postman/collection_runs/command_line_integration_with_newman/), the command line collection runner for Postman.

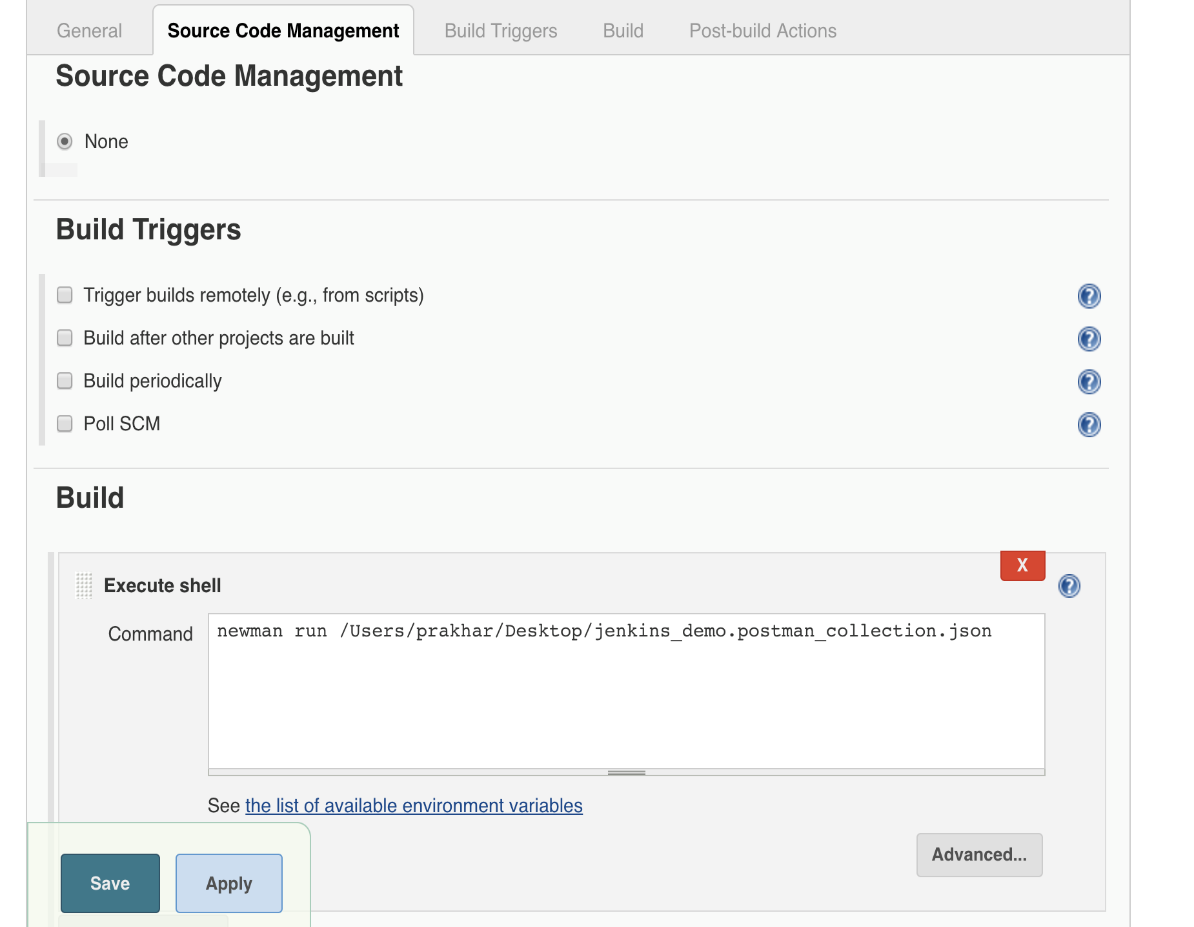
Newman allows you to run and test a Postman Collection. Newman and Jenkins are a perfect match. Let's review these topics to set up this operation.

1. Create a new job by clicking on the “New Item” link on the left sidebar > Select a “Freestyle Project” from the options > Name your project.
2. Add a build step in the project. The build step executes a Shell command.

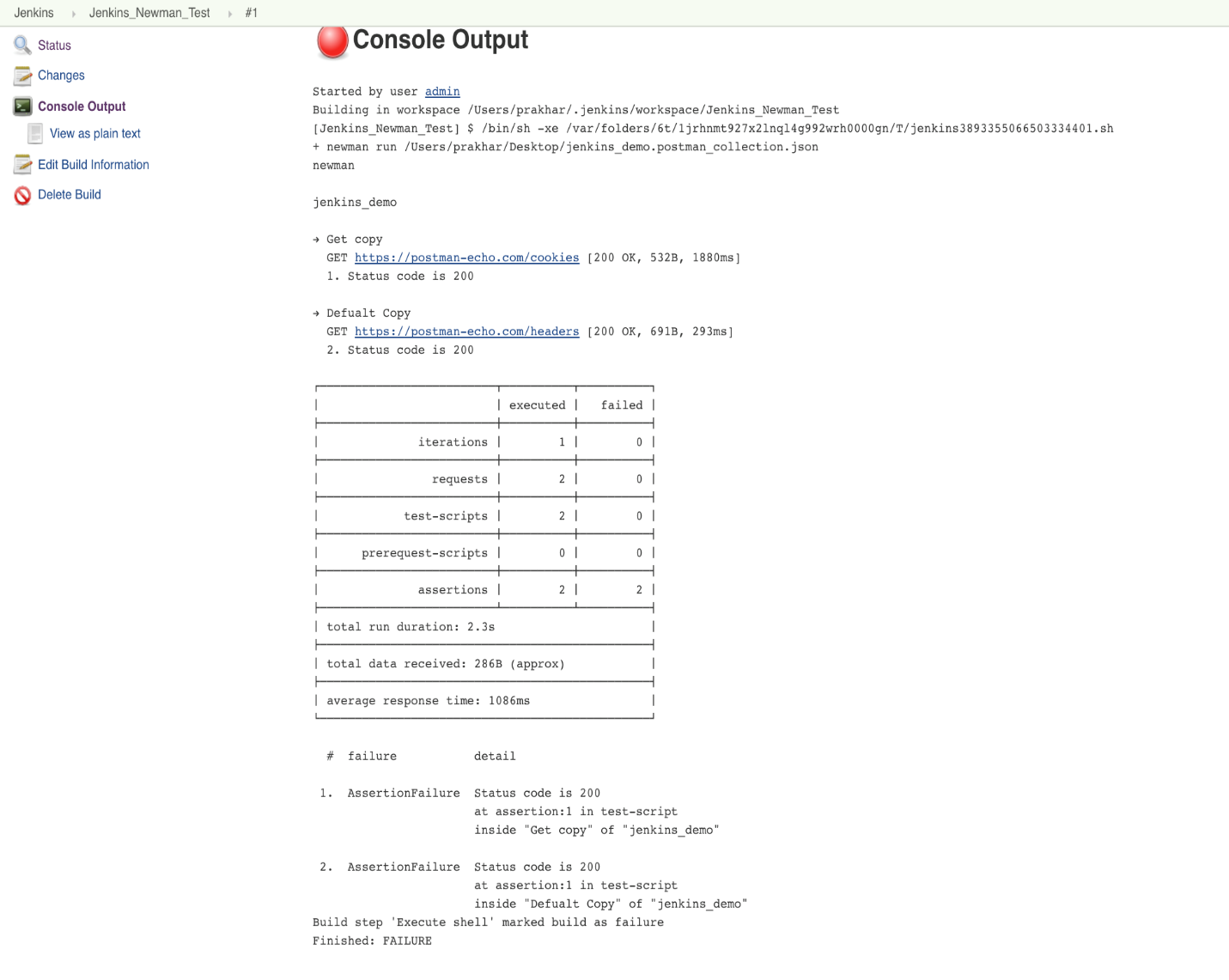
$ newman jenkins\_demo.postman\_collection --exitCode 1

Note here that we are using the Newman command parameter “exitCode” with the value 1. This denotes that Newman is going to exit with this code that will tell Jenkins that everything did not go well.

1. Click the **Save** button to finish creating the project.



1. Run this build test manually by clicking on the “Build Now” link in the sidebar.
2. Jenkins indicates that the build has failed with a red dot in the title. We can check why with the console output from Newman.
3. Click the “Console Output” link in the sidebar to see what Newman returned.



To set up the frequency with which Jenkins runs Newman, click on “Configure project” in the main project window and then scroll down.=. The syntax for setting the frequency is H/(30) \* \* \* \*.

**Note:** 30 can be replaced with another number.

Jenkins will now run Newman at your desired frequency and will tell you whether the build failed or succeeded. In a bigger setup, Newman will be part of your build process and probably not the entire process. You can set up notifications and customize Jenkins as per your needs.

1. **Intro to postman scripts**

Postman contains a powerful runtime based on Node.js that allows you to add dynamic behavior to requests and collections.

You can add JavaScript code to execute during 2 events in the flow:

1. Before a request is sent to the server, as a [pre-request script](https://learning.getpostman.com/docs/postman/scripts/pre_request_scripts/) under the **Pre-request Script** tab.
2. After a response is received, as a [test script](https://learning.getpostman.com/docs/postman/scripts/test_scripts/) under the **Tests** tab.

You can add pre-request and test scripts to a collection, a folder, a request within a collection, or a request not saved to a collection.

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

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

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

**How does this work?**

Is this magic? No, it's the [Postman Sandbox](https://learning.getpostman.com/docs/postman/scripts/postman_sandbox/). The Postman Sandbox is a JavaScript execution environment that is available to you while writing pre-request and test scripts for requests (both in Postman and Newman). Whatever code you write in these sections is executed in this sandbox.

## Debugging scripts

Debugging scripts can be written under either the **Pre-request Script** tab or the **Tests** tab, with helpful messages logged in the [Postman Console](https://learning.getpostman.com/docs/postman/sending_api_requests/debugging_and_logs/).

# **Pre-request scripts**

Pre-request scripts are snippets of code associated with a collection request that are executed before the request is sent. This is perfect for use-cases like including the timestamp in the request headers or sending a random alphanumeric string in the URL parameters.

For example, to include a timestamp in the request headers, you can set an environment variable with the value returned from a function.

**pm.environment.set(“timeStampHeader”, new Date());**

you can then access the **timestampHeader** variable in the header data editor by typing {{timestampHeader}}. When the request is sent, your pre-request script will be executed, and the value of timestampHeader will be sent in place of {{timestampHeader}}.

Pre-request scripts are written in JavaScript, and the syntax is similar to the [test scripts](https://learning.getpostman.com/docs/postman/scripts/test_scripts/) except that the response object is not present.

You can add pre-request scripts to a collection, a folder, or a single request within a collection. This allows you to reuse commonly executed code prior to every request.

Collection and folder scripts can be updated in the collection or folder details respectively. Click on the ellipsis (...) next to the collection or folder name, and select “Edit” to open the modal.

console.log(“folder1 prerequest script”)

**Test scripts**

**W**ith Postman you can write and run tests for each request using the JavaScript language. A Postman test is essentially JavaScript code executed after the request is sent, allowing access to the pm.response object.

**Pm.test(“Response time should be less than 200ms”, function() {**

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

**}):**

**Pm.test(“response code is 200”, function() {**

**Pm.response.to.have.status(200);**

**});**

// example using pm.expect()

pm.test("environment to be production", function () {

pm.expect(pm.environment.get("env")).to.equal("production");

});

// example using response assertions

pm.test("response should be okay to process", function () {

pm.response.to.not.be.error;

pm.response.to.have.jsonBody("");

pm.response.to.not.have.jsonBody("error");

});

// example using pm.response.to.be\*

pm.test("response must be valid and have a body", function () {

// assert that the status code is 200

pm.response.to.be.ok; // info, success, redirection, clientError, serverError, are other variants

// assert that the response has a valid JSON body

pm.response.to.be.withBody;

pm.response.to.be.json; // this assertion also checks if a body exists, so the above check is not needed

})

You can add as many tests as needed, depending on how many things you want to test for. Check out some [examples](https://learning.getpostman.com/docs/postman/scripts/test_examples/) of Postman tests.

### **Setting an environment variable**

pm.environment.set("variable\_key", "variable\_value");

### //**Getting an environment variable**

var value = pm.environment.get("variable\_key");

### **Setting a nested object as an environment variable**

var array = [1, 2, 3, 4];

pm.environment.set("array", JSON.stringify(array, null, 2));

var obj = { a: [1, 2, 3, 4], b: { c: 'val' } };

pm.environment.set("obj", JSON.stringify(obj));

**If the value is a stringified JSON:**

// These statements should be wrapped in a try-catch block if the data is coming from an unknown source.

var array = JSON.parse(pm.environment.get("array"));

var obj = JSON.parse(pm.environment.get("obj"));

### **Clear an environment variable**

pm.environment.unset("variable\_key");

## Globals

### **Set a global variable**

pm.globals.set("variable\_key", "variable\_value");

### **Get a global variable**

pm.globals.get("variable\_key");

### **Clear a global variable**

pm.globals.unset("variable\_key");

## Variables

This function searches for the variable across globals and the active environment.

var value = pm.variables.get("variable\_key");

## Response handling

### **Check if response body contains a string**

pm.test("Body matches string", function () {

pm.expect(pm.response.text()).to.include("string\_you\_want\_to\_search");

});

### **Check if response body is equal to a string**

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

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

});

### **Check for a JSON value**

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

var jsonData = pm.response.json();

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

});

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

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

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

});

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

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

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

});

### **Status code is 200**

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

pm.response.to.have.status(200);

});

### **Code name contains a string**

pm.test("Status code name has string", function () {

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

});

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

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

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

});

## Assertion library examples

Following is a list of some of the most common assertion tests used in the Postman test scripts.

Note that this list is not exhaustive. For the complete reference, see the documentation at: [ChaiJS expect BDD library](http://chaijs.com/api/bdd/" \t "_blank)

### **Assert if substring exists in target**

pm.test("Check if pattern is in target string",function () {

pm.expect('foobar').to.have.string('bar');

});

### **Strict Comparison**

const TEN = 10;

pm.test('Check if number is equal to 10', function () {

pm.expect(TEN).to.equal(10);

});

### **Loose comparison**

pm.test("Our JSON is loosely equal to the provided JSON", function () {

pm.expect(data1).to.deep.equal(data2);

});

**Note:**

1. .deep causes all .equal, .include, .members, .keys, and .property assertions that follow in the chain to use deep equality(loose equality) instead of strict (===) equality.
2. While the .eql also compares loosely, .deep.equal causes deep equality comparisons to also be used for any other assertions that follow in the chain while .eql does not.

### **Assert the value of response**

pm.test("Check response value", function () {

var jsonData = pm.response.json();

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

});

### **Assert the current environment**

pm.test("Check if environment is production", function () {

pm.expect(pm.environment.get('env')).to.equal('production');

});

### **Assert the type of the target is equal to the given string type**

pm.test("Check if target is string", function () {

pm.expect('Postman').to.be.a('string');

});

pm.test("Check if target is an object", function () {

pm.expect({a: 1}).to.be.an('object');

});

pm.test("Check if target is undefined", function () {

pm.expect(undefined).to.be.an('undefined');

});

**Note:**

1. It’s often best to use .a to check a target’s type before making more assertions on the same target.
2. Types are case insensitive.

### **Assert if the target is empty**

pm.test("Check if array is empty", function () {

expect([]).to.be.empty;

});

pm.test("Check if string is empty", function () {

pm.expect('').to.be.empty;

});

This can be combined with .a to check if the target is empty but has a type, say for example an array or an object.

Example:

pm.test("Check if array is empty", function () {

pm.expect([]).to.be.an('array').that.is.empty;

});

### **Assert that the target contains the keys passed**

pm.test("Check if object contains all provided keys", function () {

pm.expect({a: 1, b: 2}).to.have.all.keys('a', 'b');

});

pm.test("Checking if object contains any ONE of the keys", function () {

pm.expect({a: 1, b: 2}).to.have.any.keys('a', 'b');

});

pm.test("Check if object contains any NONE of the provided keys", function () {

pm.expect({a: 1, b: 2}).to.not.have.any.keys('c', 'd');

});

### **Assert that the target contains said property**

pm.test("Check if object contains the property", function () {

pm.expect({a: 1}).to.have.property('a');

});

**Note:**

1. Target can be an object, set, array or map.
2. If .keys is run without .all or .any, the expression defaults to .all.
3. As .keys does different things based on the target’s type, it’s recommended to check the target’s type before using .keys using .a.

pm.test("Check if object contains all the keys", function () {

pm.expect({a: 1, b: 2}).to.be.an('object').that.has.all.keys('a', 'b');

});

### **Assert the length of target**

pm.test("Check the length of the target", function () {

pm.expect('foo').to.have.lengthOf(3);

});

pm.test("Check the size of the target", function () {

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

});

### **Assert that the target array has the same members as the given array set**

pm.test("Check if the target has same members as the array set", function () {

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

});

**Note:**

1. By default, .members makes strict comparison.
2. The order of members is irrelevant.

### **Assert that the target contains the provided item**

pm.test("Check if the target array includes the number provided", function () {

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

});

pm.test("Check if the target object includes the properties provided", function () {

pm.expect({a: 1, b: 2, c: 3}).to.include({a: 1, b: 2});

});

**Note:** It is advised to first assert the type of target, as .include operates on various types. Hence it is advised to chain .a when using .include.

**Example:**

pm.test("Check if the target is an array that includes the number specified", function () {

pm.expect([1, 2, 3]).to.be.an('array').that.includes(2);

});

## Sandbox

Postman tests run in a sandboxed environment, which is separate from the execution environment of the app.