Introduction

Get started

Sending requests

Writing scripts

Scripting in Postman

Writing pre-requests scripts

Writing tests

Running tests with monitors

Script references

Test script examples

Dynamic variables

Postman JavaScript reference

Using collections

Using Postman Flows

Using the Postman CLI

Collaborating in Postman

Designing and developing your API

Documenting your API

Monitoring your API

API Governance

Administration

Using reports

Developer resources

Integrations

Home / **Writing scripts** / **Script references**

Postman JavaScript reference

Postman provides JavaScript APIs that you can use in your request scripts. The `pm` object provides functionality for testing your request and response data, with the `postman` object providing additional workflow control.

Contents

The pm object

Using variables in scripts

Environment variables

Collection variables

- Global variables
- Data variables
- Scripting with request and response data
  - Request data
  - Response data
  - Request info
  - Cookies
- Sending requests from scripts
- Scripting workflows
- Scripting Postman Visualizations
- Building response data into Postman Visualizations
- Writing test assertions
- Using external libraries

The pm object

You will carry out most of the Postman JavaScript API functionality using `pm.*`, which provides access to request and response data, and variables.

Using variables in scripts

You can access and manipulate **variables** at each scope in Postman using the `pm` API.

> You can use **dynamic variables** to generate values when your requests run.

Postman supports a variety of variable **scopes**. The `pm` object provides methods for accessing global, collection, and environment variables specifically, and `pm.variables` methods for accessing variables at different scopes and setting local variables.

Check if there is a Postman variable in the current scope:

```
pm.variables.has(variableName:String):function → Boolean
```

Get the value of the Postman variable with the specified name:

```
pm.variables.get(variableName:String):function → *
```

Set a local variable with the specified name and value:

```
pm.variables.set(variableName:String, variableValue:*):function
```

Return the resolved value of a dynamic variable inside a script using the syntax `{{$variableName}}`:

```
pm.variables.replaceIn(variableName:String):function: → *
```

For example:

```
const stringWithVars = pm.variables.replaceIn("Hi, my name is {{$randomFirstName}}");
console.log(stringWithVars);
```

Return an object containing all variables with their values in the current scope. Based on the order of precedence, this will contain variables from multiple scopes.

```
pm.variables.toObject():function → Object
```

Variable scope determines the precedence Postman gives to variables when you reference them, in order of increasing precedence:

Global

Collection

Environment

Data

Local

The variable with the closest scope overrides any others. For example, if you have variables named `score` in both the current collection and the active environment, and you call `pm.variables.get('score')`, Postman will return the current value of the environment variable. When you set a variable value using `pm.variables.set`, the value is local and will only persist for the current request or collection run.

```
//collection var 'score' = 1
//environment var 'score' = 2

//first request run
console.log(pm.variables.get('score'));//outputs 2
console.log(pm.collectionVariables.get('score'));//outputs 1
console.log(pm.environment.get('score'));//outputs 2

//second request run
pm.variables.set('score', 3);//local var
console.log(pm.variables.get('score'));//outputs 3

//third request run
console.log(pm.variables.get('score'));//outputs 2
```

See the **Postman Collection SDK Variables reference <https://www.postmanlabs.com/postman-collection/Variable.html>** for more detail.

You can also access variables defined in the individual scopes with **pm.environment**, **pm.collectionVariables**, and **pm.globals**.

Using environment variables in scripts

Your scripts can use the `pm.environment` methods to access and manipulate variables in the active (currently selected) environment.

The name of the active environment:

```
pm.environment.name:String
```

Check whether the environment has a variable with the specified name:

```
pm.environment.has(variableName:String):function → Boolean
```

Get the variable with the specified name in the active environment:

```
pm.environment.get(variableName:String):function → *
```

Set the variable with the specified name and value in the active environment:

```
pm.environment.set(variableName:String, variableValue:*):function
```

Return the resolved value of a dynamic variable inside a script using the syntax `{{$variableName}}`:

```
pm.environment.replaceIn(variableName:String):function → *
```

For example:

```
//environment has vars firstName and age
const stringWithVars = pm.environment.replaceIn("Hi, my name is {{firstName}} and I am
{{age}}.");
console.log(stringWithVars);
```

Return all variables with their values in the active environment in a single object:

```
pm.environment.toObject():function → Object
```

Remove a variable from the active environment, specifying the variable by name:

```
pm.environment.unset(variableName:String):function
```

Clear all variables in the active environment:

```
pm.environment.clear():function
```

Note that your ability to edit variables depends on your **access level** in the workspace.

Using collection variables in scripts

Your scripts can use the `pm.collectionVariables` methods to access and manipulate variables in the collection.

Check whether there is a variable in the collection with the specified name:

```
pm.collectionVariables.has(variableName:String):function → Boolean
```

Return the value of the collection variable with the specified name:

```
pm.collectionVariables.get(variableName:String):function → *
```

Set a collection variable with the specified name and value:

```
pm.collectionVariables.set(variableName:String, variableValue:*):function
```

Return the resolved value of a dynamic variable inside a script using the syntax `{{$variableName}}`:

```
pm.collectionVariables.replaceIn(variableName:String):function → *
```

For example:

```
//collection has vars firstName and age
const stringWithVars = pm.collectionVariables.replaceIn("Hi, my name is {{firstName}} and I am
{{age}}.");
console.log(stringWithVars);
```

Return all variables with their values in the collection in an object:

```
pm.collectionVariables.toObject():function → Object
```

Remove the specified variable from the collection:

```
pm.collectionVariables.unset(variableName:String):function
```

Clear all variables from the collection:

```
pm.collectionVariables.clear():function
```

Using global variables in scripts

Your scripts can use the `pm.globals` methods to access and manipulate variables at global scope within the workspace.

Check where there is a global variable with the specified name:

```
pm.globals.has(variableName:String):function → Boolean
```

Return the value of the global variable with the specified name:

```
pm.globals.get(variableName:String):function → *
```

Set a global variable with specified name and value:

```
pm.globals.set(variableName:String, variableValue:*):function
```

Return the resolved value of a dynamic variable inside a script using the syntax `{{$variableName}}`:

```
pm.globals.replaceIn(variableName:String):function → String
```

For example:

```
//globals include vars firstName and age
const stringWithVars = pm.globals.replaceIn("Hi, my name is {{firstName}} and I am {{age}}.");
console.log(stringWithVars);
```

Return all global variables and their values in an object:

```
pm.globals.toObject():function → Object
```

Remove the specified global variable:

```
pm.globals.unset(variableName:String):function
```

Clear all global variables in the workspace:

```
pm.globals.clear():function
```

Note that your ability to edit variables depends on your **access level** in the workspace.

Using data variables in scripts

Your scripts can use the `pm.iterationData` methods to access and manipulate variables from **data files during a collection run**.

Check whether a variable with the specified name exists in the current iteration data:

```
pm.iterationData.has(variableName:String):function → boolean
```

Return a variable from the iteration data with the specified name:

```
pm.iterationData.get(variableName:String):function → *
```

Return the iteration data variables in an object:

```
pm.iterationData.toObject():function → Object
```

Convert the iterationData object to JSON format:

```
pm.iterationData.toJSON():function → *
```

Remove the specified variable:

```
pm.iterationData.unset(key:String):function
```

Scripting with request and response data

A variety of methods provide access to request and response data in Postman scripts, including **pm.request**, **pm.response**, **pm.info**, and **pm.cookies**. Additionally you can send requests using **pm.sendRequest**.

Scripting with request data

The `pm.request` object provides access to the data for the request the script is running within. For a **Pre-request Script** this is the request that's about to run, and for a **Test** script this is the request that has already run.

You can use the `pm.request` object pre-request scripts to alter various parts of the request configuration before it runs.

The `pm.request` object provides the following properties and methods:

The request URL:

```
pm.request.url:Url
```

The **list of headers <https://www.postmanlabs.com/postman-collection/HeaderList.html>** for the current request:

```
pm.request.headers:HeaderList
```

The HTTP request method:

```
pm.request.method:String
```

The data in the **request body <https://www.postmanlabs.com/postman-collection/RequestBody.html>** . This object is immutable and can't be modified from scripts:

```
pm.request.body:RequestBody
```

Add a header with the specified name and value for the current request:

```
pm.request.headers.add(header:Header):function
```

For example:

```
pm.request.headers.add({
  key: "client-id",
  value: "abcdef"
});
```

Delete the request header with the specified name:

```
pm.request.headers.remove(headerName:String):function
```

Insert the specified header name and value (if the header doesn't exist, otherwise the already existing header will update to the new value):

```
pm.request.headers.upsert({key: headerName:String, value: headerValue:String}):function)
```

See the Postman **Collection SDK Request reference <https://www.postmanlabs.com/postman-collection/Request.html>** for more detail.

Scripting with response data

The `pm.response` object provides access to the data returned in the response for the current request in scripts added to the **Tests**.

The `pm.response` object provides the following properties and methods:

The response status code:

```
pm.response.code:Number
```

The status text string:

```
pm.response.status:String
```

The **list of response headers <https://www.postmanlabs.com/postman-collection/HeaderList.html>** :

```
pm.response.headers:HeaderList
```

The time the response took to receive in milliseconds:

```
pm.response.responseTime:Number
```

The size of the response received:

```
pm.response.responseSize:Number
```

The response text:

```
pm.response.text():Function → String
```

The response JSON, which you can use to drill down into the properties received:

```
pm.response.json():Function → Object
```

See the Postman **Collection SDK Response reference <https://www.postmanlabs.com/postman-collection/Response.html>** for more detail.

Scripting with request info

The `pm.info` object provides data related to the request and the script itself, including name, request ID, and iteration count.

The `pm.info` object provides the following properties and methods:

The event, which will be either `prerequest` or `test` depending on where the script is executing within the request:

```
pm.info.eventName:String
```

The value of the current **iteration**:

```
pm.info.iteration:Number
```

The total number of iterations that are scheduled to run:

```
pm.info.iterationCount:Number
```

The saved name of the request running:

```
pm.info.requestName:String
```

A unique GUID that identifies the running request:

```
pm.info.requestId:String
```

Scripting with request cookies

The `pm.cookies` object provides access to the list of cookies associated with the request.

The `pm.cookies` object provides the following properties and methods:

Check whether a particular cookie (specified by name) exists for the requested domain:

```
pm.cookies.has(cookieName:String):Function → Boolean
```

Get the value of the specified cookie:

```
pm.cookies.get(cookieName:String):Function → String
```

Get a copy of all cookies and their values in an object. Returns any cookies that are defined for the request domain and path:

```
pm.cookies.toObject():Function → Object
```

See the Postman **Collection SDK Cookie List reference <https://www.postmanlabs.com/postman-collection/CookieList.html>** for more detail.

You can also use `pm.cookies.jar` to specify a domain for access to request cookies.

To enable programmatic access using the `pm.cookies.jar` methods, first add the cookie URL to the **allowlist**.

Access the cookie jar object:

```
pm.cookies.jar():Function → Object
```

For example:

```
const jar = pm.cookies.jar();
//cookie methods...
```

Set a cookie using name and value:

```
jar.set(URL:String, cookie name:String, cookie value:String, callback(error, cookie)):Function →
Object
```

Set a cookie using **PostmanCookie <https://www.postmanlabs.com/postman-collection/Cookie.html>** or a compatible object:

```
jar.set(URL:String, { name:String, value:String, httpOnly:Bool }, callback(error,
cookie)):Function → Object
```

For example:

```
const jar = pm.cookies.jar();
jar.set("httpbin.org", "session-id", "abc123", (error, cookie) => {
  if (error) {
    console.error(`An error occurred: ${error}`);
  } else {
    console.log(`Cookie saved: ${cookie}`);
  }
});
```

Get a cookie from the cookie jar:

```
jar.get(URL:String, cookieName:String, callback (error, value)):Function → Object
```

Get all the cookies from the cookie jar. The cookies are available in the callback function:

```
jar.getAll(URL:String, callback (error, cookies)):Function
```

Remove a cookie:

```
jar.unset(URL:String, token:String, callback(error)):Function → Object
```

Clear all cookies from the cookie jar:

```
jar.clear(URL:String, callback (error)):Function → Object
```

See the Postman **Collection SDK Cookie reference <https://www.postmanlabs.com/postman-collection/Cookie.html>** for more detail.

Sending requests from scripts

You can use the `pm.sendRequest` method to send a request asynchronously from a **Pre-request** or **Test** script. This allows you to execute logic in the background if you are carrying out computation or sending multiple requests at the same time without waiting for each to complete. You can avoid blocking issues by adding a callback function so that your code can respond when Postman receives a response. You can then carry out any additional processing you need on the response data.

You can pass the `pm.sendRequest` method a URL string, or can provide a complete request configuration in JSON including headers, method, body, **and more <http://www.postmanlabs.com/postman-collection/Request.html#~definition>** .

```
// Example with a plain string URL
pm.sendRequest('https://postman-echo.com/get', (error, response) => {
  if (error) {
    console.log(error);
  } else {
  console.log(response);
  }
});

// Example with a full-fledged request
const postRequest = {
  url: 'https://postman-echo.com/post',
  method: 'POST',
  header: {
    'Content-Type': 'application/json',
    'X-Foo': 'bar'
  },
  body: {
    mode: 'raw',
    raw: JSON.stringify({ key: 'this is json' })
  }
};
pm.sendRequest(postRequest, (error, response) => {
  console.log(error ? error : response.json());
});

// Example containing a test
pm.sendRequest('https://postman-echo.com/get', (error, response) => {
  if (error) {
    console.log(error);
  }

  pm.test('response should be okay to process', () => {
    pm.expect(error).to.equal(null);
    pm.expect(response).to.have.property('code', 200);
    pm.expect(response).to.have.property('status', 'OK');
  });
});
```

See the **Request definition <http://www.postmanlabs.com/postman-collection/Request.html#~definition>** and **Response structure <http://www.postmanlabs.com/postman-collection/Response.html>** reference docs for more detail.

Scripting workflows

The `postman` object provides the `setNextRequest` method for building request workflows when you use the **collection runner** or **Newman**.

> Note that `setNextRequest` has no effect when you run requests using **Send**; it only has an effect when you run a collection.

When you run a collection (using the collection runner or Newman), Postman will run your requests in a default order or an order you specify when you set up the run. However, you can override this execution order using `postman.setNextRequest` to specify which request to run next.

Run the specified request after this one (the request name as defined in the collection, for example "Get customers"):

```
postman.setNextRequest(requestName:String):Function
```

Run the specified request after this one (the request ID returned by `pm.info.requestId`):

```
postman.setNextRequest(requestId:String):Function
```

For example:

```
//script in another request calls:
//pm.environment.set('next', pm.info.requestId)
postman.setNextRequest(pm.environment.get('next'));
```

Scripting Postman Visualizations

Use `pm.visualizer.set` to specify a template to **display response data in the Postman Visualizer**.

```
pm.visualizer.set(layout:String, data:Object, options:Object):Function
```

`layout` **required**

Handlebars <https://handlebarsjs.com/> HTML template string

`data` *optional*

JSON object that binds to the template and you can access it inside the template string

`options` *optional*

Options object <https://handlebarsjs.com/api-reference/compilation.html> for `Handlebars.compile()`

Example usage:

```
var template = `<p>{{res.info}}</p>`;
pm.visualizer.set(template, {
    res: pm.response.json()
});
```

Building response data into Postman Visualizations

Use `pm.getData` to retrieve response data inside a Postman Visualizer template string.

```
pm.getData(callback):Function
```

The callback function accepts two parameters:

`error`

Any error detail

`data`

Data **passed to the template** by `pm.visualizer.set`

Example usage:

```
pm.getData(function (error, data) {
  var value = data.res.info;
});
```

Writing test assertions

`pm.test(testName:String, specFunction:Function):Function`

You can use `pm.test` to write test specifications inside either the **Pre-request** or **Tests** scripts. Tests include a name and assertion—Postman will output test results as part of the response.

The `pm.test` method returns the `pm` object, making the call chainable. The following sample test checks that a response is valid to proceed.

```
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');
});
```

An optional `done` callback can be passed to `pm.test`, to test asynchronous functions.

```
pm.test('async test', function (done) {
  setTimeout(() => {
    pm.expect(pm.response.code).to.equal(200);
    done();
  }, 1500);
});
```

Get the total number of tests executed from a specific location in code:

```
pm.test.index():Function → Number
```

The `pm.expect` method allows you to write assertions on your response data, using **ChaiJS expect BDD <https://www.chaijs.com/api/bdd/>** syntax.

```
pm.expect(assertion:*):Function → Assertion
```

You can also use `pm.response.to.have.*` and `pm.response.to.be.*` to build your assertions.

See **Test examples** for more assertions.

Using external libraries

```
require(moduleName:String):function → *
```

The `require` method allows you to use the sandbox built-in library modules. The list of available libraries is listed below with links to the corresponding documentation.

- **ajv <https://www.npmjs.com/package/ajv>**
- **atob <https://www.npmjs.com/package/atob>**
- **btoa <https://www.npmjs.com/package/btoa>**
- **chai <https://www.chaijs.com/>**
- **cheerio <https://cheerio.js.org/>**
- **crypto-js <https://www.npmjs.com/package/crypto-js>**
- **csv-parse/lib/sync <https://csv.js.org/parse/>**
- **lodash <https://lodash.com/>** (The built-in _ object v3.10.1 exists in the sandbox by default. Use `require` to load the latest version.)
- **moment <https://momentjs.com/docs/>**
- **postman-collection <http://www.postmanlabs.com/postman-collection/>**
- **tv4 <https://github.com/geraintluff/tv4>**
- **uuid <https://www.npmjs.com/package/uuid>**
- **xml2js <https://www.npmjs.com/package/xml2js>**

The following NodeJS modules are also available to use in the sandbox:

- **path <https://nodejs.org/api/path.html>**
- **assert <https://nodejs.org/api/assert.html>**
- **buffer <https://nodejs.org/api/buffer.html>**
- **util <https://nodejs.org/api/util.html>**
- **url <https://nodejs.org/api/url.html>**
- **punycode <https://nodejs.org/api/punycode.html>**
- **querystring <https://nodejs.org/api/querystring.html>**
- **string-decoder <https://nodejs.org/api/string_decoder.html>**
- **stream <https://nodejs.org/api/stream.html>**
- **timers <https://nodejs.org/api/timers.html>**
- **events <https://nodejs.org/api/events.html>**

To use a library, call the `require` method, pass the module name as a parameter, and assign the return object from the method to a variable.

Next steps

You can use tests to build Postman into your development projects in a variety of ways using **Postman utilities**.

Last modified: 2023/08/23

[← Dynamic variables]

Communication
<https://www.postman.com/category/communication>

Artificial Intelligence
<https://www.postman.com/category/artificial-intelligence>