**Get started with tests**

To write your first test script, open a request in Postman, then select the **Tests** tab. Enter the following JavaScript code:

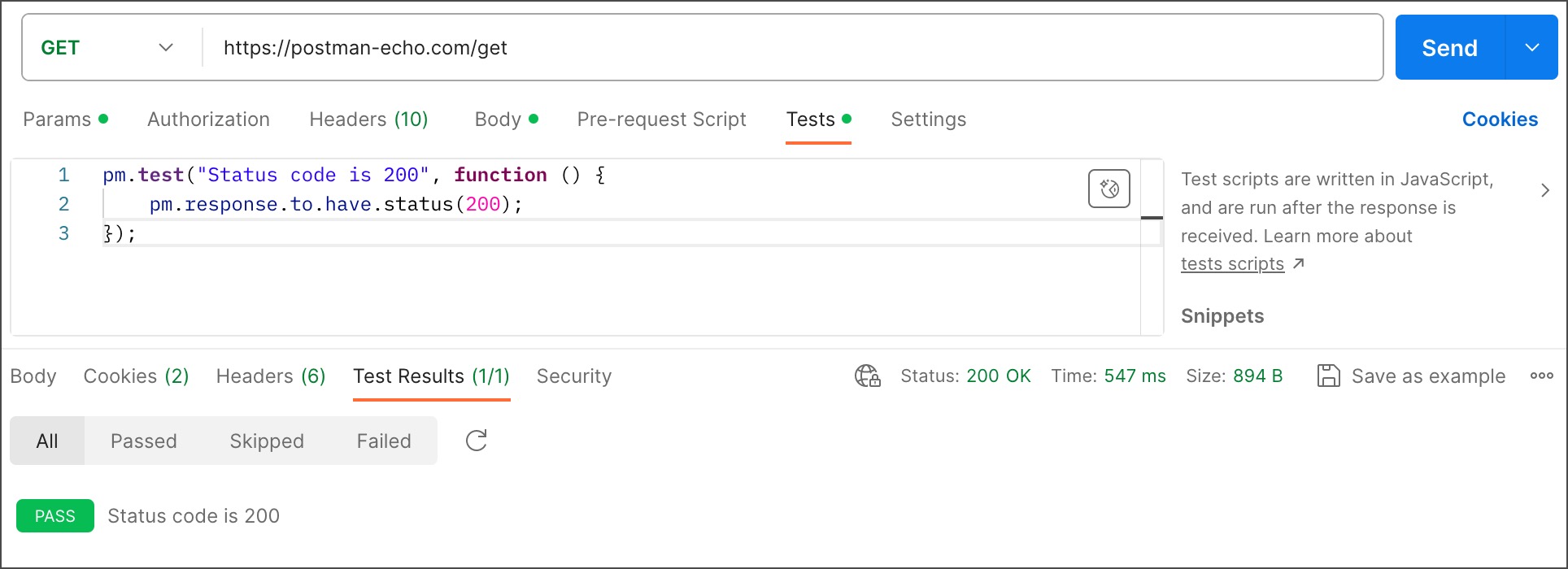
pm.test("Status code is 200", function () {

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

});

This code uses the pm library to run the test method. The text string will appear in the test output. The function inside the test represents an assertion. Postman tests can use [Chai Assertion Library BDD](https://www.chaijs.com/api/bdd/) syntax, which provides options to optimize how readable your tests are to you and your collaborators. In this case, the code uses BDD chains to.have to express the assertion.

This test checks the response code returned by the API. If the response code is 200, the test will pass, otherwise it will fail. Select **Send** and go to the **Test Results** tab in the response area.



To learn what test results look like when they pass or fail, change the status code in the assertion code and send the request again.

You can structure your test assertions in a variety of ways, depending on how you want the results to output. The following code is an alternative way of achieving the same test as the one above using the expect syntax:

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

pm.expect(pm.response.code).to.eql(200);

});

Refer to the [Chai Assertion Library Docs](https://www.chaijs.com/api/bdd/) for a complete overview of assertion syntax options.

**Use multiple assertions**

Your tests can include multiple assertions as part of a single test. Use this to group together related assertions:

pm.test("The response has all properties", () => {

//parse the response JSON and test three properties

const responseJson = pm.response.json();

pm.expect(responseJson.type).to.eql('vip');

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

pm.expect(responseJson.id).to.have.lengthOf(1);

});

If any of the contained assertions fails, the test as a whole will fail. All assertions must be successful for the test to pass.

**Parse response body data**

To carry out assertions on your responses, you will first need to parse the data into a JavaScript object that your assertions can use.

To parse JSON data, use the following syntax:

const responseJson = pm.response.json();

To parse XML, use the following:

const responseJson = xml2Json(pm.response.text());

If you're dealing with complex XML responses you may find [Console logging](https://learning.postman.com/docs/sending-requests/troubleshooting-api-requests/#debugging-in-the-console) useful.

To parse CSV, use the [CSV parse (csv-parse/lib/sync)](https://csv.js.org/parse/) utility:

const parse = require('csv-parse/lib/sync');

const responseJson = parse(pm.response.text());

To parse HTML, use [cheerio](https://cheerio.js.org/):

const $ = cheerio.load(pm.response.text());

//output the html for testing

console.log($.html());

**Handle responses that don't parse**

If you can't parse the response body into JavaScript because it's not formatted as JSON, XML, HTML, CSV, or any other parsable data format, you can still make assertions on the data.

Test if the response body contains a string:

pm.test("Body contains string",() => {

pm.expect(pm.response.text()).to.include("customer\_id");

});

This doesn't tell you where the string was encountered because it carries out the test on the whole response body. Test if a response matches a string:

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

pm.response.to.have.body("whole-body-text");

});

**Make assertions on the HTTP response**

Your tests can check various aspects of a request response, including the [body](https://learning.postman.com/docs/writing-scripts/script-references/test-examples/#test-response-body), [status codes](https://learning.postman.com/docs/writing-scripts/script-references/test-examples/#test-status-codes), [headers](https://learning.postman.com/docs/writing-scripts/script-references/test-examples/#test-headers), [cookies](https://learning.postman.com/docs/writing-scripts/script-references/test-examples/#test-cookies), [response times](https://learning.postman.com/docs/writing-scripts/script-references/test-examples/#test-response-times), and more.

**Test response body**

Check for particular values in the response body:

/\* Response has the following structure:

{

"name": "Jane",

"age": 23

},

\*/

pm.test("Person is Jane", () => {

const responseJson = pm.response.json();

pm.expect(responseJson.name).to.eql("Jane");

pm.expect(responseJson.age).to.eql(23);

});

**Test status codes**

Test for the response status code:

pm.test("Status code is 201", () => {

pm.response.to.have.status(201);

});

If you want to test for the status code being one of a set, include them all in an array and use oneOf:

pm.test("Successful POST request", () => {

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

});

Check the status code text:

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

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

});

**Test headers**

Check that a response header is present:

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

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

});

Test for a response header having a particular value:

pm.test("Content-Type header is application/json", () => {

pm.expect(pm.response.headers.get('Content-Type')).to.include('application/json');

});

**Test cookies**

Test if a cookie is present in the response:

pm.test("Cookie isLoggedIn is present", () => {

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

});

Test for a particular cookie value:

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

pm.expect(pm.cookies.get('isLoggedIn')).to.eql('1');

});

**Test response times**

Test for the response time to be within a specified range:

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

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

});

**Common assertion examples**

The following examples of common assertions might help you write your test scripts.

For a more comprehensive overview of what you can include in your assertions, refer to the [Chai Assertion Library Docs](https://www.chaijs.com/api/bdd/).

**Assert a response value against a variable**

Check if a response property has the same value as a variable (this example uses an environment variable):

pm.test("Response property matches environment variable", function () {

pm.expect(pm.response.json().name).to.eql(pm.environment.get("name"));

});

See [Using variables](https://learning.postman.com/docs/sending-requests/variables/) to learn more about using variables in your test scripts.

**Assert a value type**

Test the type of any part of the response:

/\* Response has the following structure:

{

"name": "Jane",

"age": 29,

"hobbies": [

"skating",

"painting"

],

"email": null

},

\*/

const jsonData = pm.response.json();

pm.test("Test data type of the response", () => {

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

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

pm.expect(jsonData.age).to.be.a("number");

pm.expect(jsonData.hobbies).to.be.an("array");

pm.expect(jsonData.website).to.be.undefined;

pm.expect(jsonData.email).to.be.null;

});

**Assert array properties**

Check if an array is empty, and if it contains particular items:

/\* Response has the following structure:

{

"errors": [],

"areas": [ "goods", "services" ],

"settings": [

{

"type": "notification",

"detail": [ "email", "sms" ]

},

{

"type": "visual",

"detail": [ "light", "large" ]

}

]

},

\*/

const jsonData = pm.response.json();

pm.test("Test array properties", () => {

//errors array is empty

pm.expect(jsonData.errors).to.be.empty;

//areas array includes "goods"

pm.expect(jsonData.areas).to.include("goods");

//get the notification settings object

const notificationSettings = jsonData.settings.find

(m => m.type === "notification");

pm.expect(notificationSettings)

.to.be.an("object", "Could not find the setting");

//detail array must include "sms"

pm.expect(notificationSettings.detail).to.include("sms");

//detail array must include all listed

pm.expect(notificationSettings.detail)

.to.have.members(["email", "sms"]);

});

The order in .members doesn't affect the test.

**Assert object properties**

Assert that an object contains keys or properties:

/\* Response has the following structure:

{

"a": 1,

"b": 2

},

\*/

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

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

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

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

pm.expect({a: 1, b: 2}).to.be.a('object')

.that.has.all.keys('a', 'b');

Target can be an object, set, array or map. If .keys is run without .all or .any, the expression defaults to .all. As .keys behavior varies based on the target type, it's recommended to check the type before using .keys with .a.

**Assert that a value is in a set**

Check a response value against a list of valid options:

/\* Response has the following structure:

{

"type": "Subscriber"

},

\*/

pm.test("Value is in valid list", () => {

pm.expect(pm.response.json().type)

.to.be.oneOf(["Subscriber", "Customer", "User"]);

});

**Assert that an object is contained**

Check that an object is part of a parent object:

/\* Response has the following structure:

{

"id": "d8893057-3e91-4cdd-a36f-a0af460b6373",

"created": true,

"errors": []

},

\*/

pm.test("Object is contained", () => {

const expectedObject = {

"created": true,

"errors": []

};

pm.expect(pm.response.json()).to.deep.include(expectedObject);

});

Using .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. While .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 doesn't.

**Assert the current environment**

Check the [active environment](https://learning.postman.com/docs/sending-requests/managing-environments/#selecting-an-active-environment) in Postman:

pm.test("Check the active environment", () => {

pm.expect(pm.environment.name).to.eql("Production");

});

**Troubleshoot common test errors**

When you encounter errors or unexpected behavior in your test scripts, [the Postman Console](https://learning.postman.com/docs/sending-requests/troubleshooting-api-requests/) can help you to identify the source. By combining console.log(), console.info(), console.warn(), and console.error() debug statements with your test assertions, you can examine the content of the HTTP requests and responses, and Postman data items such as variables. You can also use the console.clear() method to clear information from the console. Select Console icon **Console** from the Postman footer to open it.



Log the value of a variable or response property:

console.log(pm.collectionVariables.get("name"));

console.log(pm.response.json().name);

Log the type of variable or response property:

console.log(typeof pm.response.json().id);

Use Console logs to mark code execution, sometimes known as "trace statements":

if (pm.response.json().id) {

console.log("id was found!");

// do something

} else {

console.log("no id ...");

//do something else

}

**Assertion deep equality error**

You might encounter the AssertionError: expected <value> to deeply equal '<value>' error. For example, this would arise with the following code:

pm.expect(1).to.eql("1");

This happens because the test is comparing a number to a string value. The test will only return true if both the type and value are equal.

**Variable not defined error**

You might encounter the ReferenceError: <variable> is not defined error. This typically happens when you're attempting to reference a variable that hasn't been declared or is outside the scope of your test code.

In the following example, a JSON object is the value of a variable in the first test. The second test is attempting to reference the variable, but it can't because the variable is outside the scope of the second test's code.

/\* Response has the following structure:

{

"name": "John",

"age": 29

},

\*/

pm.test("Test 1", () => {

const jsonData = pm.response.json();

pm.expect(jsonData.name).to.eql("John");

});

pm.test("Test 2", () => {

pm.expect(jsonData.age).to.eql(29); // ReferenceError: jsonData is not defined

});

Make sure variables are available at the global scope if test functions needs to reference it. In the previous example, moving const jsonData = pm.response.json(); before the first pm.test would make it available to both test functions.

**Assertion undefined error**

You might encounter the AssertionError: expected undefined to deeply equal <value> error. Typically this happens when you are referring to a property that doesn't exist or is out of scope.

const jsonData = pm.response.json();

pm.expect(jsonData.name).to.eql("John");

In this example, if you get the error AssertionError: expected undefined to deeply equal 'John', this indicates that the name property isn't defined in the jsonData object.

**Test not failing**

There may be occasions where you expect a test to fail, and it doesn't. Make sure your test code is syntactically correct, then resend your request.

In the following example, the test is expected to fail because true doesn't equal false. The test actually passes because the pm.test function isn't properly defined. The pm.test function is missing the first parameter, which is a text string that displays in the test result output. You can learn more about [defining tests using the pm.test function](https://learning.postman.com/docs/writing-scripts/test-scripts/#validating-responses).

pm.test( function () {

pm.expect(true).to.eql(false);

});

**Validate response structure**

You can validate your JSON schema with [Tiny Validator V4 (tv4)](https://github.com/geraintluff/tv4):

const schema = {

"items": {

"type": "boolean"

}

};

const data1 = [true, false];

const data2 = [true, 123];

pm.test('Schema is valid', function() {

pm.expect(tv4.validate(data1, schema)).to.be.true;

pm.expect(tv4.validate(data2, schema)).to.be.true;

});

You can also validate your JSON schema with the [Ajv JSON schema validator](https://www.npmjs.com/package/ajv):

const schema = {

"properties": {

"alpha": {

"type": "boolean"

}

}

};

pm.test('Schema is valid', function() {

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

});

**Send an asynchronous request**

You can send a request from your test code and log the response:

pm.sendRequest("https://postman-echo.com/get", function (err, response) {

console.log(response.json());

});

**Previous style of writing Postman tests (deprecated)**

**This section refers to deprecated script syntax used in earlier versions of Postman. If you are writing new scripts, use the current syntax.**

The previous style of writing Postman tests relies on setting values for the tests object. Set a descriptive key for an element in the object and then assert if it's true or false. For example, the following will check if the response body contains the user\_id string:

tests["Body contains user\_id"] = responsebody.has("user\_id");

Add as many keys as needed, depending on how many things you want to test for. View your test results in the response viewer under the **Tests** tab. The tab header shows how many tests passed, and the keys that you set in the tests variable are listed there. If the value evaluates to true, the test passed.

//Set an environment variable

postman.setEnvironmentVariable("key", "value");

//Set a nested object as an environment variable

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

postman.setEnvironmentVariable("array", JSON.stringify(array, null, 2));

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

postman.setEnvironmentVariable("obj", JSON.stringify(obj));

//Get an environment variable

postman.getEnvironmentVariable("key");

//Get an environment variable whose value is a stringified object

//(Wrap in a try-catch block if the data is coming from an unknown source)

const array = JSON.parse(postman.getEnvironmentVariable("array"));

const obj = JSON.parse(postman.getEnvironmentVariable("obj"));

//Clear an environment variable

postman.clearEnvironmentVariable("key");

//Set a global variable

postman.setGlobalVariable("key", "value");

//Get a global variable

postman.getGlobalVariable("key");

//Clear a global variable

postman.clearGlobalVariable("key");

//Check if response body contains a string

tests["Body matches string"] = responseBody.has("string\_you\_want\_to\_search");

//Check if response body is equal to a string

tests["Body is correct"] = responseBody === "response\_body\_string";

//Check for a JSON value

const data = JSON.parse(responseBody);

tests["Your test name"] = data.value === 100;

//Content-Type is present (Case-insensitive checking)

tests["Content-Type is present"] = postman.getResponseHeader("Content-Type");

tests["Content-Type is present"] = postman.getResponseHeader("Content-Type");

//getResponseHeader() method returns the header value, if it exists

//Content-Type is present (Case-sensitive)

tests["Content-Type is present"] = responseHeaders.hasOwnProperty("Content-Type");

//Response time is less than 200ms

tests["Response time is less than 200ms"] = responseTime < 200;

//Response time is within a specific range

//(lower bound inclusive, upper bound exclusive)

tests["Response time is acceptable"] = \_.inRange(responseTime, 100, 1001);

//Status code is 200

tests["Status code is 200"] = responseCode.code === 200;

//Code name contains a string

tests["Status code name has string"] = responseCode.name.has("Created");

//Successful POST request status code

tests["Successful POST request"] = responseCode.code === 201 || responseCode.code === 202;