

Learning Objectives

Learners will be able to...

- **Identify the purpose of Postman**
- **Use Postman to interact with an API**
- **Differentiate between GET and POST requests**
- **Use Parameters in the request**
- **Create Postman tests**

info

Make Sure You Know

Learners should be familiar with the how the client-server architecture works, as well as the basics of JavaScript.

Limitations

Postman is a client that runs outside of Codio. It is both a standalone application or a web service. We will be using the website. You will need to make an account in order to follow along with the assignment.

Intro to Postman

What is Postman?

People use Postman to facilitate the development, testing, and documentation of an API (Application Programming Interface). It can be used to send data in a request and check the response. We can save requests into folders and collections, and conveniently parametrize them.

With Postman, we can send http and https requests to services and get answers from them. Using this approach, we can test backend services and make sure that they work correctly.

We can install the Postman client for Linux, Mac, Windows, or use the web version. For this assignment, we will be using the [web client](#). Be sure to create an account. You do not need to download the desktop application. The web interface is more than enough for our needs.

Why use Postman?

Postman is free and easy to use. Because of this, it has a large community of users. Postman's popularity also comes from a long list of sophisticated features. For example, it supports different types of requests to any API (REST, SOAP, GraphQL). Developers can integrate test suites into their favorite CI/CD tool with [Newman](#). This allows developers to run Postman collections at the command line.

Requests

In Postman, we create a request, click the send button, and get a response from the API. Sending requests and receiving responses is done over the internet using the HTTP protocol. The application which sends the request is called the client.

A request always contains URL of API endpoint and HTTP request method. The endpoint is the address that we refer to when sending a request. The most commonly used methods are:

- **GET** requests are used to get data.
- **POST** requests are used to send new data.
- **PUT** requests are used to update existing data.
- **PATCH** requests (like PUT) are used to update existing data. The difference is that PATCH requests can update multiple records at once.

- **DELETE** requests are used to delete existing data.

For more information on Postman, take a look at their [documentation](#).

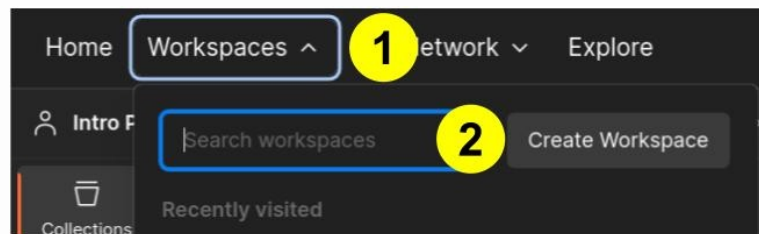
Interface

Setup

First, you need to log in or make a new account if you don't already have a Postman profile. Use the link below to open Postman in a new tab. Log in to your profile.

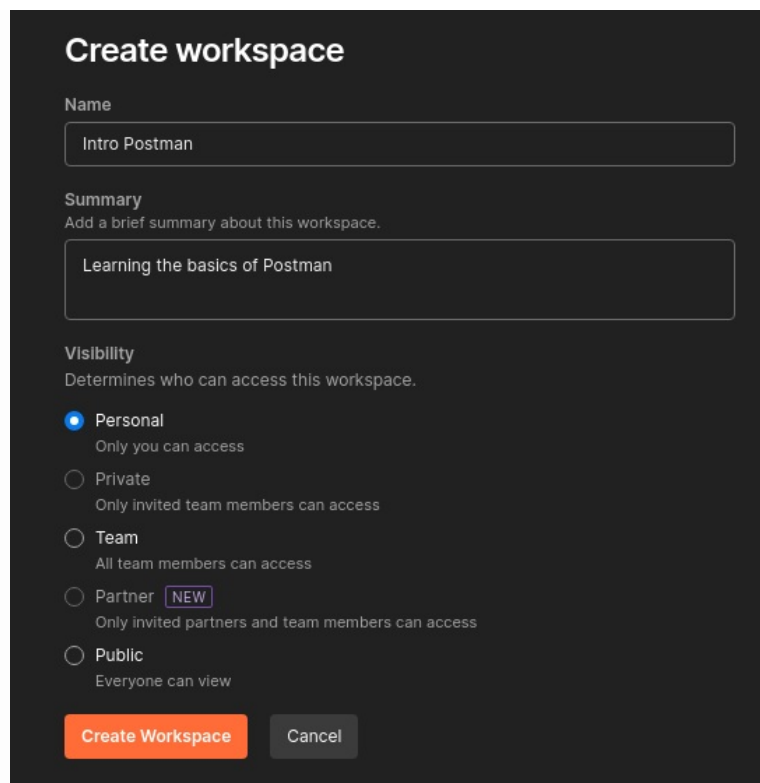
[Postman Login](#)

Then, we need to create a new Workspace. In the top menu, select Workspace and click **Create Workspace**.



The image depicts the buttons to create and name a new workspace.

In the window, enter Intro Postman as the name of our workspace. In the Summary field we can add a short description like "Learning the basics of Postman". Then specify the access level. For example, choose the Personal setting. This means that the created workspace will be available only to me. And click the **Create Workspace** button.



Create workspace

Name

Intro Postman

Summary

Add a brief summary about this workspace.

Learning the basics of Postman

Visibility

Determines who can access this workspace.

☒ Personal
Only you can access

☐ Private
Only invited team members can access

☐ Team
All team members can access

☐ Partner **NEW**
Only invited partners and team members can access

☐ Public
Everyone can view

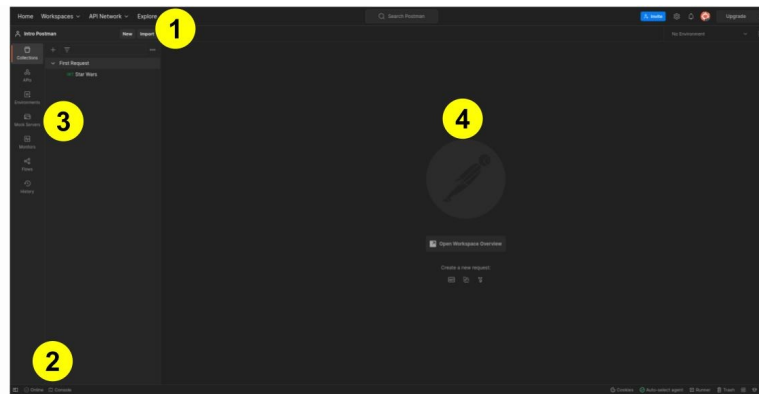
Create Workspace Cancel

The image depicts the name of the workspace, a field for a summary, and the different choices for visibility.

Postman Interface

Now “Intro Postman” workspace has been created and we can work with it. The main parts of application window are:

1. **Top menu** - It allows to create Workspaces and access various Network APIs. In the Workspaces tab we can create a new workspace, select one of the recently visited ones, or search through existing ones. The Home tab contains your home page. On it we can also see our recent activity.
2. **Bottom menu** - Here, we can open Console, manage Cookies, open Runners and many other functions.
3. **Left menu** - The main functional elements in Postman are available in the left menu. The most popular are Collections and History. Collections is for more convenient storage, grouping and searching of requests. In History can see previously made requests.
4. **Main working area** - Most of work in Postman will take place here. The main parts are the tabs, the drop-down list for choosing your environment, and the right-side menu. We need tabs for better organization of our requests.



The image depicts the Postman web application. There are four circles with a number in them. The first circle indicates the top menu. The second circle indicates the bottom menu. The third circle indicates the left menu. The fourth circle indicates the main working area.

Collections and Variables

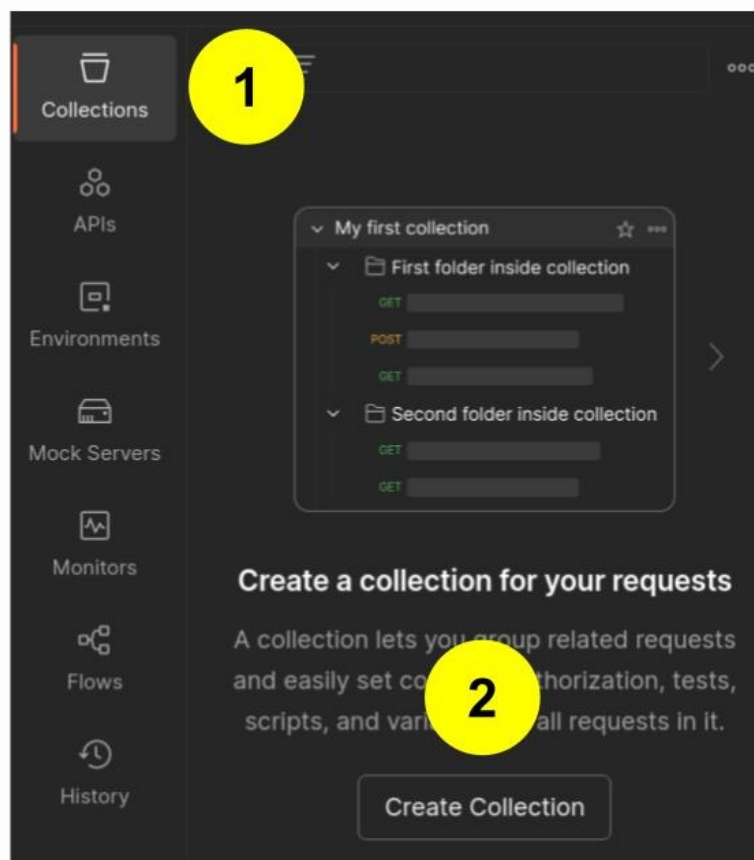
info

Postman Account

Be sure that you have created an [account](#) with Postman and are logged in before continuing.

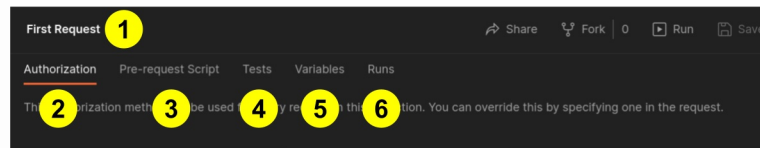
Create a Collection

Before we can create a request, we first need to create a collection inside our workspace. Click the “Collections” button in the left menu (1). Then click “Create Collection” (2).



The image shows the collections button in the left menu and the button to create a new collection.

Label the collection “First Request” (1).



The image shows the name of the collection and the five tabs for different information for each request - Authorization, Pre-request Script, Tests, Variables, and Runs.

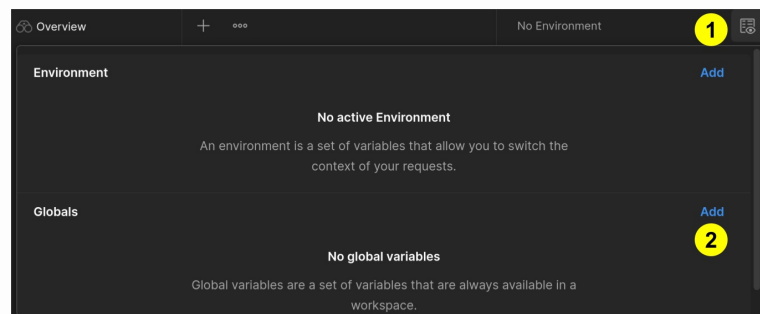
Each collection has a set of common settings you can use:

- The Authorization tab(2) allows you to configure authorization for requests from this collection. For the purposes of these examples, we will not use any authorization.
- The Pre-request Script tab(3) configures the scripts that are run before sending requests from this collection to the server.
- The Tests tab(4) contains the test scripts for this collection, which are run after receiving responses from the API.
- The Variables tab(5) allows you to define the values of the variables used within the collection.
- The Runs tab(6) shows the results from running the collection.

Creating Variables

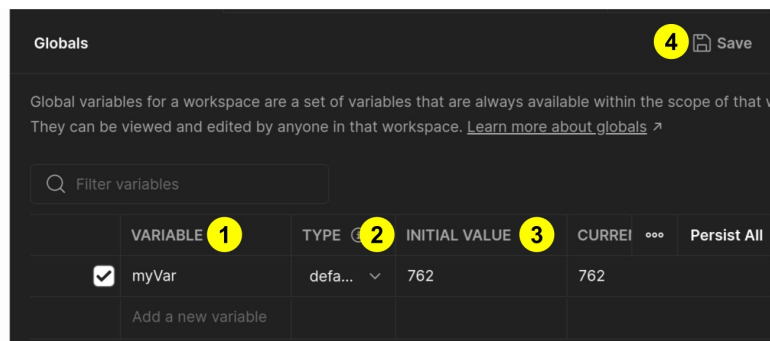
Variables can also be used to pass values between requests and tests. Variables can store not only parameters but also part of a URL.

To create a variable, click on the Environment quick look icon(1) at the top-right of the screen. In the Globals section, click Add(2).



The image shows two circles with numbers in them. The first circle is next to the environment quick look icon. The second circle is next to the button to add a global variable.

Variables in Postman are key-value pairs. The variable name is the key by which we refer to it to access its value. Enter the name `myVar` for the variable(1), and use 762 as the initial value(3). Be sure to save(4) the variable before continuing.



The image shows four circles with numbers in them. The first circle is next to the column where you enter a variable name. The second circle is next to the column where you set the variable type. The third circle is next to the column where you set the value of the variable. The fourth circle is next to the save icon.

There are two types of global variables — default and secret(2). Default variables display the variable value as text. Secret variables are used to hide values. For example, they are useful for sensitive data such as passwords, keys, and tokens.

We will see later how to create variables that are unique to collections, folders, and requests.

First Request

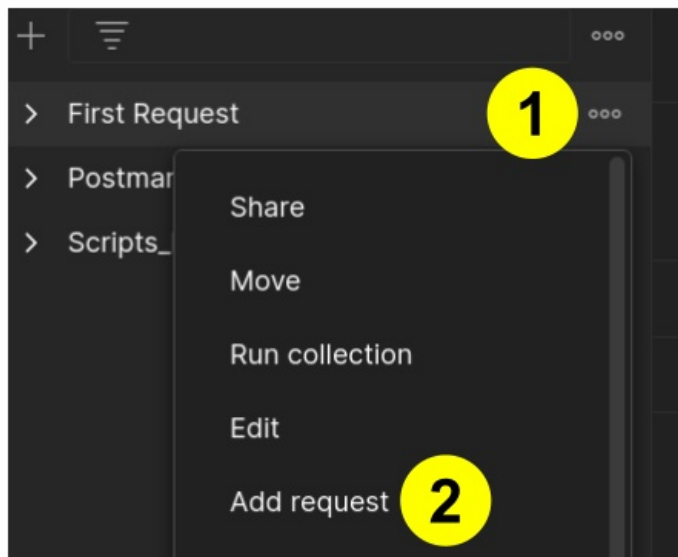
info

Postman Account

Be sure that you have created an [account](#) with Postman and are logged in before continuing.

Create a Request

Now that we have a collection, we can create a request associated with it. Click on the icon with three dots(1) next to the collection name. Then select the option to create a new request(2).



The image shows two numbered circles. The first circle is next to the icon with three dots associated with the collection name. The second circle is next to the button to add a new request to the collection.

Our first request will be a GET request from the [The Star Wars API](#). Use the following URL for the request:

```
https://swapi.dev/api/people/1/
```

Give this request a name. Something like “Star Wars” is sufficient (1). Choose the GET as the method and enter the address above into the blank (2). Click the “Send” button (3) to start our request.

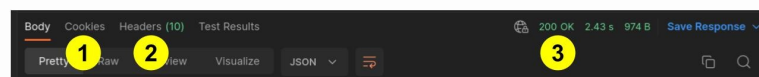


The image shows three numbered circles. The first circle is next to the field where you enter the name of the request. The second circle is next to the dropdown menu where you select the request type. The third circle is next to the button to send the request.

The result will be a response containing the data in JSON format. We should see the following result in the main working area:

```
json -hide-clipboard {  "name": "Luke Skywalker",    "height":
"172",    "mass": "77",    "hair_color": "blond",
"skin_color": "fair",    "eye_color": "blue",    "birth_year":
"19BBY",    "gender": "male",    "homeworld":
"https://swapi.dev/api/planets/1/",    "films": [
"https://swapi.dev/api/films/1/",
"https://swapi.dev/api/films/2/",
"https://swapi.dev/api/films/3/",
"https://swapi.dev/api/films/6/"    ],    "species": [],
"vehicles": [    "https://swapi.dev/api/vehicles/14/",
"https://swapi.dev/api/vehicles/30/"    ],    "starships": [
"https://swapi.dev/api/starships/12/",
"https://swapi.dev/api/starships/22/"    ],    "created": "2014-
12-09T13:50:51.644000Z",    "edited": "2014-12-
20T21:17:56.891000Z",    "url": "https://swapi.dev/api/people/1/" }
```

In Postman we can see the Body we received from the server. Postman automatically recognizes the JSON format and formats it to make it readable. In addition to the Body, we can see the Cookies(1) and Headers(2) of the response. We can also see that the request was executed successfully and we got the response code 200 OK(3).



The image shows three numbered circles. The first circle is below the tab labeled “Cookies”. The second circle is below the tab labeled “Headers”. The third circle is next to the response code for the request.

challenge

Try these variations

Use a GET request with the following request URLs:

- * `https://swapi.dev/api/starships/9`

- * `https://swapi.dev/api/species/9`

Working with GET and POST Requests

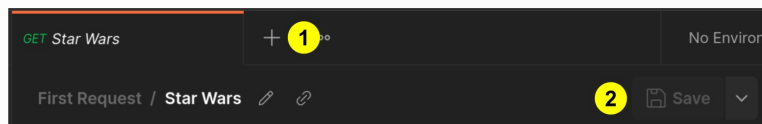
info

Postman Account

Be sure that you have created an account with Postman and are logged in before continuing.

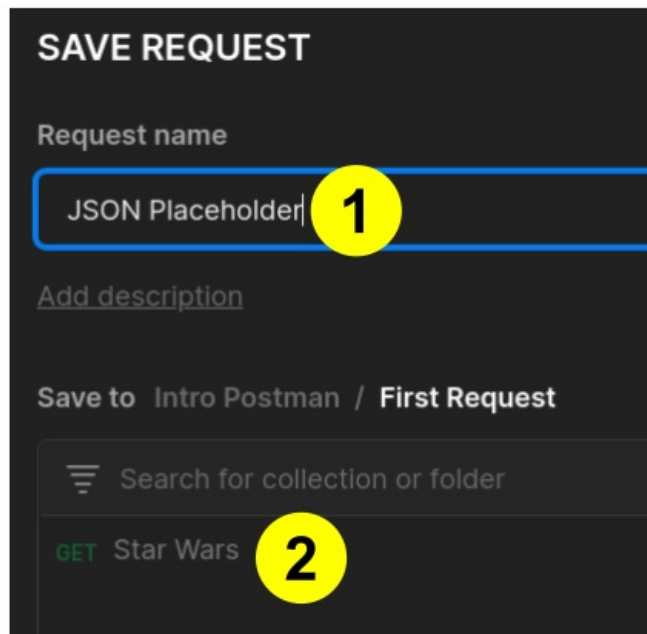
More GET Requests

Let's make another GET request, but this time we are going to use the JSON Placeholder website to generate the response to our API request. Start by clicking the + icon next to the tab with our Star Wars request (1). This will add a new request to our collection. Then click Save(2) and you will be asked to provide additional information about the new request.



The image shows two numbered circles. The first circle is next to the “+” icon. The second circle is next to the save button.

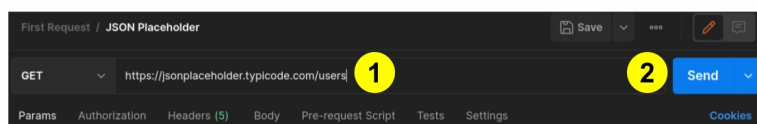
Give this new request the name “JSON Placeholder”. In the box below(2), select the “First Response” collection. Then click save. This will open a new tab with a new request.



The image shows two numbered circles. The first circle is on the field where you enter the name of the new request. The second circle is in the area in which you choose the folder or collection to which the new request belongs.

Leave the type of request as GET. Set the URL(1) to the value below. Then press Send(2).

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



The image shows two numbered circles. The first is on the field where you enter the URL for the GET request. The second is next to the send button.

After performing the request, we should see the data from the server in the Body tab, as well as 200 OK message.

▼ Request Output

Postman will return a list of 10 JSON objects. Each object represents a fictional user with made-up information. Your output should look something like this:

```
[
  {
    "id": 1,
    "name": "Leanne Graham",
```

```
"username": "Bret",
"email": "Sincere@april.biz",
"address": {
  "street": "Kulas Light",
  "suite": "Apt. 556",
  "city": "Gwenborough",
  "zipcode": "92998-3874",
  "geo": {
    "lat": "-37.3159",
    "lng": "81.1496"
  }
},
"phone": "1-770-736-8031 x56442",
"website": "hildegard.org",
"company": {
  "name": "Romaguera-Crona",
  "catchPhrase": "Multi-layered client-server neural-
net",
  "bs": "harness real-time e-markets"
},
{
  "id": 2,
  "name": "Ervin Howell",
  "username": "Antonette",
  "email": "Shanna@melissa.tv",
  "address": {
    "street": "Victor Plains",
    "suite": "Suite 879",
    "city": "Wisokyburgh",
    "zipcode": "90566-7771",
    "geo": {
      "lat": "-43.9509",
      "lng": "-34.4618"
    }
  },
  "phone": "010-692-6593 x09125",
  "website": "anastasia.net",
  "company": {
    "name": "Deckow-Crist",
    "catchPhrase": "Proactive didactic contingency",
    "bs": "synergize scalable supply-chains"
  }
},
{
  "id": 3,
  "name": "Clementine Bauch",
  "username": "Samantha",
```

```
"email": "Nathan@yesenia.net",
"address": {
  "street": "Douglas Extension",
  "suite": "Suite 847",
  "city": "McKenziehaven",
  "zipcode": "59590-4157",
  "geo": {
    "lat": "-68.6102",
    "lng": "-47.0653"
  }
},
"phone": "1-463-123-4447",
"website": "ramiro.info",
"company": {
  "name": "Romaguera-Jacobson",
  "catchPhrase": "Face to face bifurcated interface",
  "bs": "e-enable strategic applications"
}
},
{
  "id": 4,
  "name": "Patricia Lebsack",
  "username": "Karianne",
  "email": "Julianne.OConner@kory.org",
  "address": {
    "street": "Hoeger Mall",
    "suite": "Apt. 692",
    "city": "South Elvis",
    "zipcode": "53919-4257",
    "geo": {
      "lat": "29.4572",
      "lng": "-164.2990"
    }
  },
  "phone": "493-170-9623 x156",
  "website": "kale.biz",
  "company": {
    "name": "Robel-Corkery",
    "catchPhrase": "Multi-tiered zero tolerance productivity",
    "bs": "transition cutting-edge web services"
  }
},
{
  "id": 5,
  "name": "Chelsey Dietrich",
  "username": "Kamren",
  "email": "Lucio_Hettinger@annie.ca",
```



```
    "address": {
      "street": "Skiles Walks",
      "suite": "Suite 351",
      "city": "Roscoevew",
      "zipcode": "33263",
      "geo": {
        "lat": "-31.8129",
        "lng": "62.5342"
      }
    },
    "phone": "(254)954-1289",
    "website": "demarco.info",
    "company": {
      "name": "Keebler LLC",
      "catchPhrase": "User-centric fault-tolerant
solution",
      "bs": "revolutionize end-to-end systems"
    }
  },
  {
    "id": 6,
    "name": "Mrs. Dennis Schulist",
    "username": "Leopoldo_Corkery",
    "email": "Karley_Dach@jasper.info",
    "address": {
      "street": "Norberto Crossing",
      "suite": "Apt. 950",
      "city": "South Christy",
      "zipcode": "23505-1337",
      "geo": {
        "lat": "-71.4197",
        "lng": "71.7478"
      }
    },
    "phone": "1-477-935-8478 x6430",
    "website": "ola.org",
    "company": {
      "name": "Considine-Lockman",
      "catchPhrase": "Synchronised bottom-line interface",
      "bs": "e-enable innovative applications"
    }
  },
  {
    "id": 7,
    "name": "Kurtis Weissnat",
    "username": "Elwyn.Skiles",
    "email": "Telly.Hoeger@billy.biz",
    "address": {
```

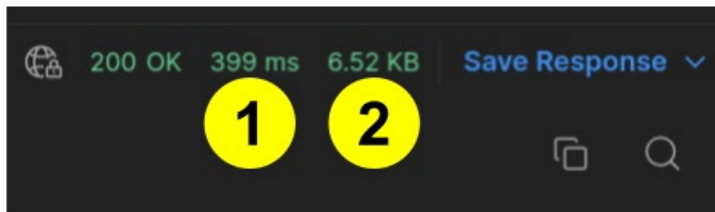
```
        "street": "Rex Trail",
        "suite": "Suite 280",
        "city": "Howemouth",
        "zipcode": "58804-1099",
        "geo": {
            "lat": "24.8918",
            "lng": "21.8984"
        }
    },
    "phone": "210.067.6132",
    "website": "elvis.io",
    "company": {
        "name": "Johns Group",
        "catchPhrase": "Configurable multimedia task-force",
        "bs": "generate enterprise e-tailers"
    }
},
{
    "id": 8,
    "name": "Nicholas Runolfsdottir V",
    "username": "Maxime_Nienow",
    "email": "Sherwood@rosamond.me",
    "address": {
        "street": "Ellsworth Summit",
        "suite": "Suite 729",
        "city": "Aliyaview",
        "zipcode": "45169",
        "geo": {
            "lat": "-14.3990",
            "lng": "-120.7677"
        }
    },
    "phone": "586.493.6943 x140",
    "website": "jacynthe.com",
    "company": {
        "name": "Abernathy Group",
        "catchPhrase": "Implemented secondary concept",
        "bs": "e-enable extensible e-tailers"
    }
},
{
    "id": 9,
    "name": "Glenna Reichert",
    "username": "Delphine",
    "email": "Chaim_McDermott@dana.io",
    "address": {
        "street": "Dayna Park",
        "suite": "Suite 449",
```

```

        "city": "Bartholomebury",
        "zipcode": "76495-3109",
        "geo": {
            "lat": "24.6463",
            "lng": "-168.8889"
        }
    },
    "phone": "(775)976-6794 x41206",
    "website": "conrad.com",
    "company": {
        "name": "Yost and Sons",
        "catchPhrase": "Switchable contextually-based
project",
        "bs": "aggregate real-time technologies"
    }
},
{
    "id": 10,
    "name": "Clementina DuBuque",
    "username": "Moriah.Stanton",
    "email": "Rey.Padberg@karina.biz",
    "address": {
        "street": "Kattie Turnpike",
        "suite": "Suite 198",
        "city": "Lebsackbury",
        "zipcode": "31428-2261",
        "geo": {
            "lat": "-38.2386",
            "lng": "57.2232"
        }
    },
    "phone": "024-648-3804",
    "website": "ambrose.net",
    "company": {
        "name": "Hoeger LLC",
        "catchPhrase": "Centralized empowering task-force",
        "bs": "target end-to-end models"
    }
}
]

```

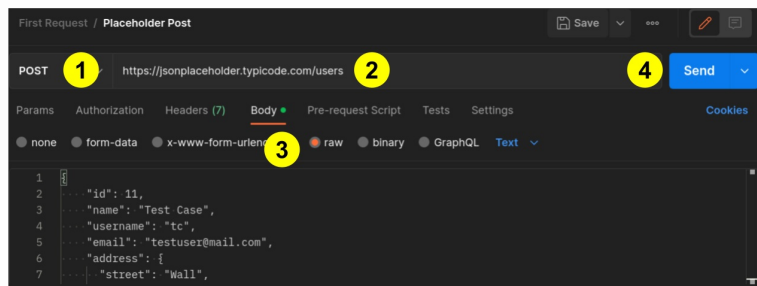
By hovering over Time(1) and Size(2), a pop-up window appears with detailed information.



The image shows two numbered circles. The first is below the text “399ms” which is how long it took the request to complete. The second circle is below the text “6.52kb” which represents the size of the request.

POST Request

Let’s try to use the POST request to add a new user. Using the steps above, create a new request, name it “Placeholder Post”, and save it to the “First Request” collection. Once the new request has been made, we will send the information about the new user in the body of the POST request.

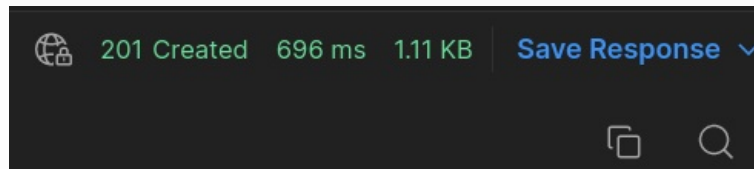


The image shows four numbered circles. The first circle is next to the dropdown that sets the request type. The second circle is on the field for the URL of the request. The third circle is next to the tab labeled “Body”. The fourth circle is next to the send button.

1. Set the request type - POST.
2. Set request URL: `https://jsonplaceholder.typicode.com/users`
3. Click on the Body tab, choose Raw - JSON. Insert data about the user below:

```
{
  "id": 11,
  "name": "Test Case",
  "username": "tc",
  "email": "testuser@mail.com",
  "address": {
    "street": "Wall",
    "suite": "31",
    "city": "New York",
    "zipcode": "10001",
    "geo": {
      "lat": "10.0000",
      "lng": "80.0000"
    }
  },
  "phone": "1-2345-6-7890",
  "website": "mail.com",
  "company": {
    "name": "testuser",
    "catchPhrase": "website for test",
    "bs": "real-time tutorials"
  }
}
```

4. After that press the SEND button and send a POST request. You should see a 201 response code.



The image shows a 201 response code after running the request.

Note: you can check your JSON with [jsonformatter](#) to verify it is in the correct format before sending the request with Postman.

Formats in the Body

info

Postman Account

Be sure that you have created an [account](#) with Postman and are logged in before continuing.

Requests that require information are done with the Body section of the main working area. You can format the body in a variety of ways. We will cover some of the more popular formats.

Examples on this page use the Echo URL in Postman. You can send a request to `postman-echo.com` and Postman will return a JSON response with the details of the request. This provides a convenient way to explore the features of Postman without having to query a proper API.

Form, URL, and Binary Formats

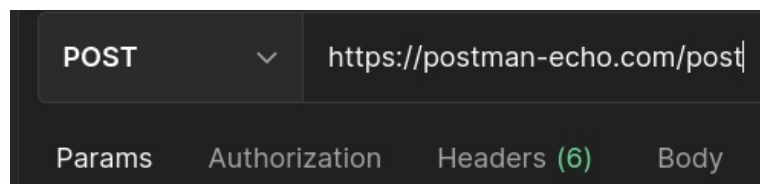
Using the instructions from the previous page, create a new request named “Echo Requests” and save it to the “First Response” collection.



The image shows the new request name as “Echo Requests”.

In the following examples we will use the POST method without any parameters. Use the following URL for the requests:

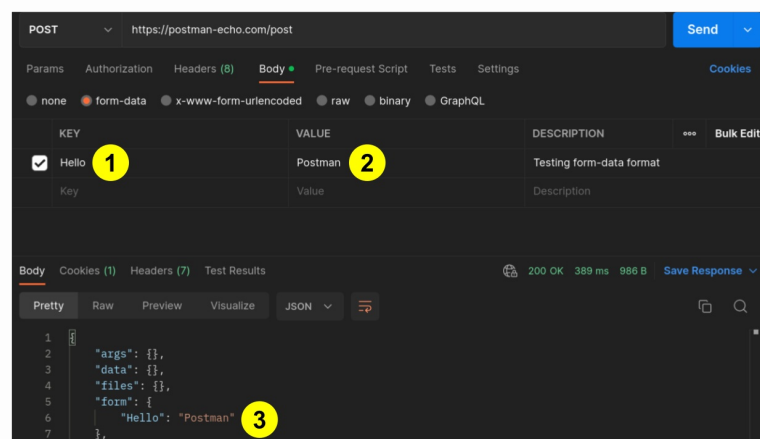
`https://postman-echo.com/post`



The image shows the URL of a POST request, which is “https://postman-echo.com/post”.

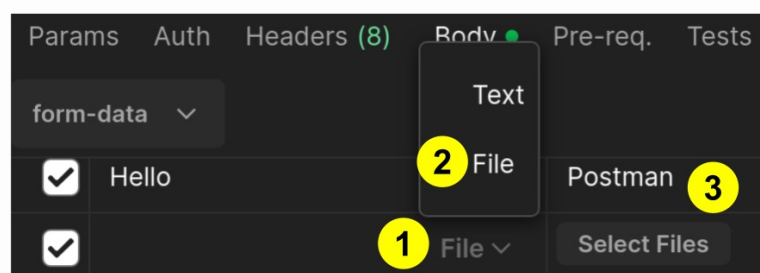
Form Data

The form-data format allows you to send data in key-value pair format. Postman provides a place for you to enter the keys and values. Enter Hello as the key(1) and Postman as the value(2). After sending the request, you can see your key-value pairs in the body of the response(3).



The image shows three numbered circles. The first circle shows you the key with the data “Hello” for the form-data format. The second circles shows the value associated with the previous key with the information “Postman”. The third circle shows the body of the request in JSON format. You can see the key-value pair from circles 1 and 2.

We can also use the form-data format to send files. To do this, place your mouse on the next row for key-value pairs. Click on Text(1) in the key area. Select the File(2) option. You will see a Select File(3) button appear in the value area that allows you to upload a file.

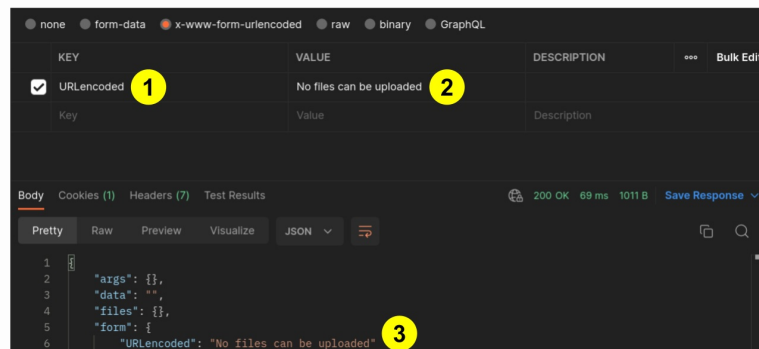


The image shows three numbered circles. First circle shows the “File” option instead of the default text. The second circle shows

the menu that lets you select file or text. The third circle is next to the button that lets you select a file.

URL Encoded

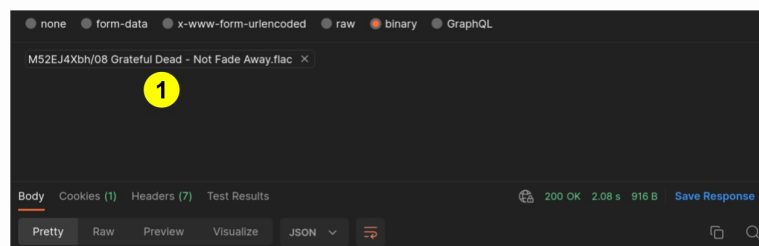
The x-www-form-urlencoded format looks very similar to form-data. Here the data is also transmitted in key-value format. You enter the values for the keys(1) and values(2) just as you would in form-data. You can see these values in the body of the response(3). However, it is not possible to transmit files.



The image shows three numbered circles. The first circle is next to the key data which is “URLencoded”. The second circle is next to the value data which is “No files can be uploaded”. The third circle shows the JSON representation of the body including the key-value pair.

Binary

It is designed to send binary data (pictures, audio, video and text files). Postman provides a **Select File** button for you to select a file. Navigate to the file on your computer and upload it to Postman. You will see it(1) in Postman if everything works as intended.



The image shows a single numbered circle. It is next to the binary file associated with the request. In this case it is a music file in the FLAC format.

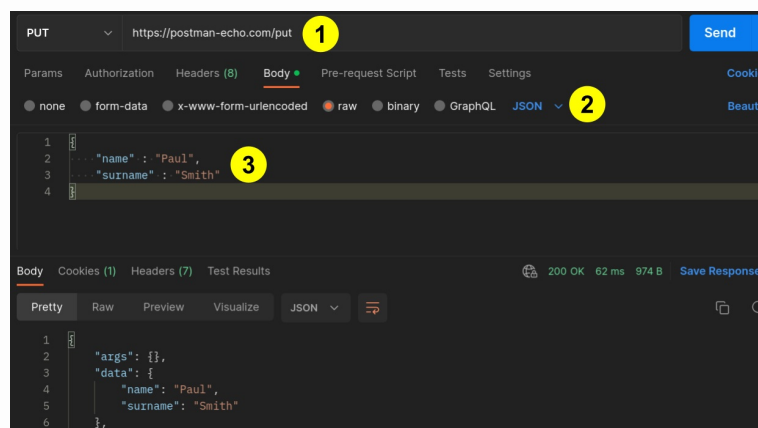
Raw Format

Let's try to send a PUT request. This type of request is usually used to update data. Be sure to change the request type to PUT and add the following request URL(1):

```
https://postman-echo.com/put
```

In the body of the request we can send text data in different formats. For this purpose, the variant of data transmission: `raw` is used. In the drop-down menu we can select the format of the data to be transmitted. Available formats are Text, JavaScript, JSON, HTML and XML. Select JSON as the data format(2). Copy and paste the code below into the request body(3).

```
{
  "name" : "Paul",
  "surname" : "Smith"
}
```



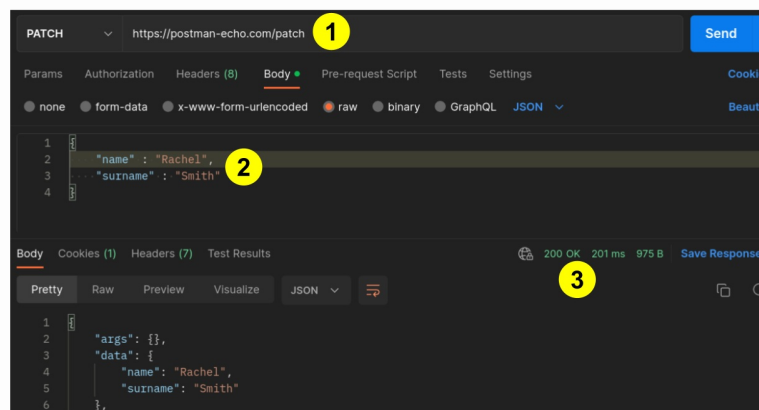
The image shows three numbered circles. The first circle shows the URL for a PUT request, which is “https://postman-echo.com/put”. The second circle shows the format of the raw data, which is JSON. The third circle shows the information associated with the request. Instead of being a table of key-value pairs, it is in JSON.

After sending the request, you should see the following response:

```
{
  "args": {},
  "data": "{
    "name" : "Paul",
    "surname" : "Smith"}"
  },
  "files": {},
  "form": {},
  "headers": {
    .....
  }
}
```

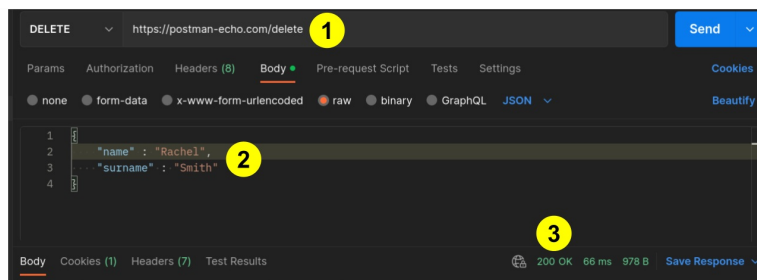
PATCH requests are usually used to partially update data. Let's try to send this type of request in raw as well. Update the request type and request URL to reflect a PATCH request(1). Add some updated information in the request body(2). The response body does not make it clear what changed, but you should see a 200 OK response code(3).

```
{
  "name" : "Rachel",
  "surname" : "Smith"
}
```



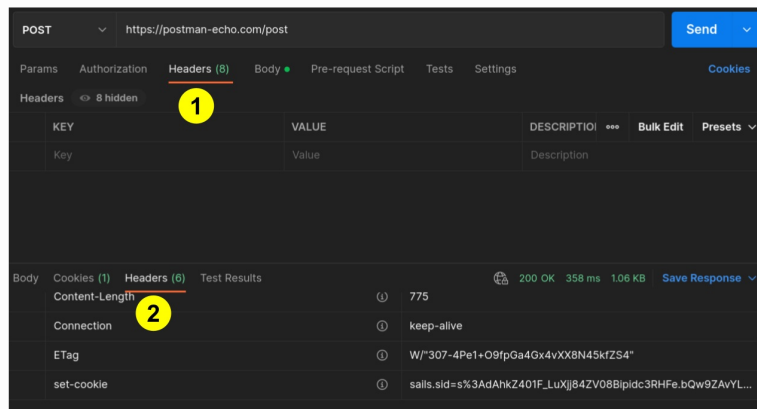
The image shows three numbered circles. The first circle shows the URL for a PATCH request which is “https://postman-echo.com/patch”. The second circle shows the information to be sent in the request in the JSON format. The third circle shows a 200 request response code.

Finally, we can use a DELETE request to delete data. Update the request type and request URL(1). Specify which information should be deleted in the request body(2). Verify that you are getting a 200 OK response code(3).



The image shows three numbered circles. The first circle shows the URL for a DELETE request which is “https://postman-echo.com/delete”. The second circle shows the information to be sent in the request in the JSON format. The third circle shows a 200 request response code.

On the Headers tab, you can add request headers if necessary. You can enter information for the headers of the request(1). After sending the request, we can see the response headers(2) at the bottom of the main screen.



The image shows two numbered circles. The first circle is next to the “Headers” tab. The second circle shows the headers associated with the response.

Working with Responses

info

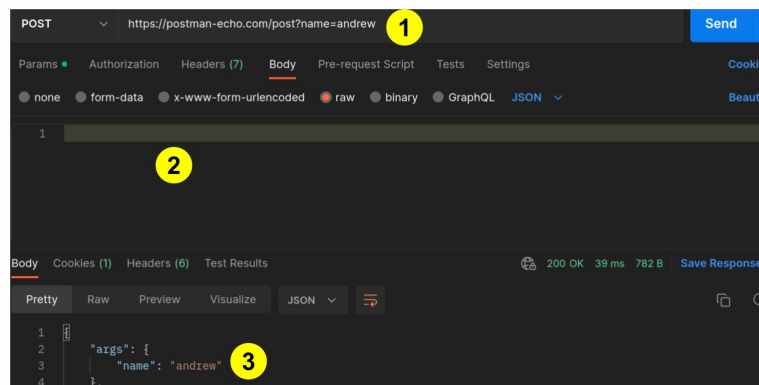
Postman Account

Be sure that you have created an [account](#) with Postman and are logged in before continuing.

Body

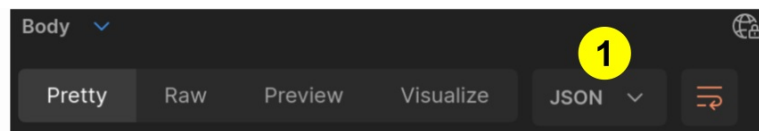
Let's change the request to be a POST with the request URL(1) below. Do not put anything in the body of the request(2). The name=andrew becomes a key-value pair in the "args" portion of the response body(3).

`https://postman-echo.com/post?name=andrew`



The image shows three numbered circles. The first circle shows the URL for the request which is “https://postman-echo.com/post?name=andrew”. The second circle shows the empty body area for the request. The third circle shows a JSON representation of the arguments taken from the URL.

Let's talk a little bit about Pretty, Raw, Preview, Visualize. You have probably seen these options in the body of a response. These options control how you view the response. You can even select the format of the body(1).



The image shows a single numbered circle next to the JSON option for formatting.

- **Pretty** - The request body view for JSON and XML formats. This view automatically formats the received response for easier understanding. A convenient highlighting of different parts of the text is applied.
- **Raw** - Text representation of the answer body, with no additional formatting. This is not a user-friendly way of presenting information.
- **Preview** - This view allows to conveniently view the resulting content in processed form. This can be useful, for example, in the case of standard errors, which come in the form of HTML page.
- **Visualize** - Allows to process the received response in your own way. The code for visualization is added separately.

Headers and Cookies

The Cookies tab allows to view all cookies sent by the server. Each entry will contain the Name, Value, Domain and Path, as well as additional information.

Name	Value	Domain	Path	Expires	HttpOnly	Secure
salls.sid	s%3AhAPYJk...	postman-ech...	/	Session	true	false

The image shows the cookies associated with the request.

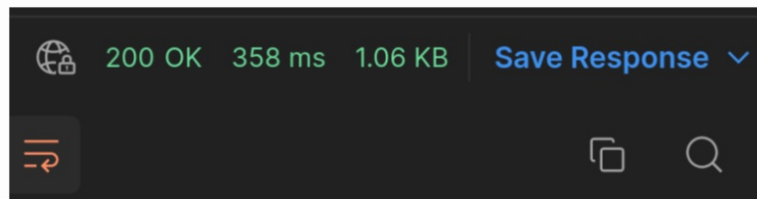
The Headers tab displays information about received headers, in key-value format. If you hover your mouse over the information icon, in the header name field, you can get information about each of it, according to the HTTP protocol specification.

Date	Tue, 17 Jan 2023 16:32:21 GMT
Content-Type	application/json; charset=utf-8
Content-Length	568
Connection	keep-alive
ETag	W/"238-aZUQzQ/QwBSzy3lshOkRuSxS8"
Vary	Accept-Encoding

The image shows the six headers associated with the request.

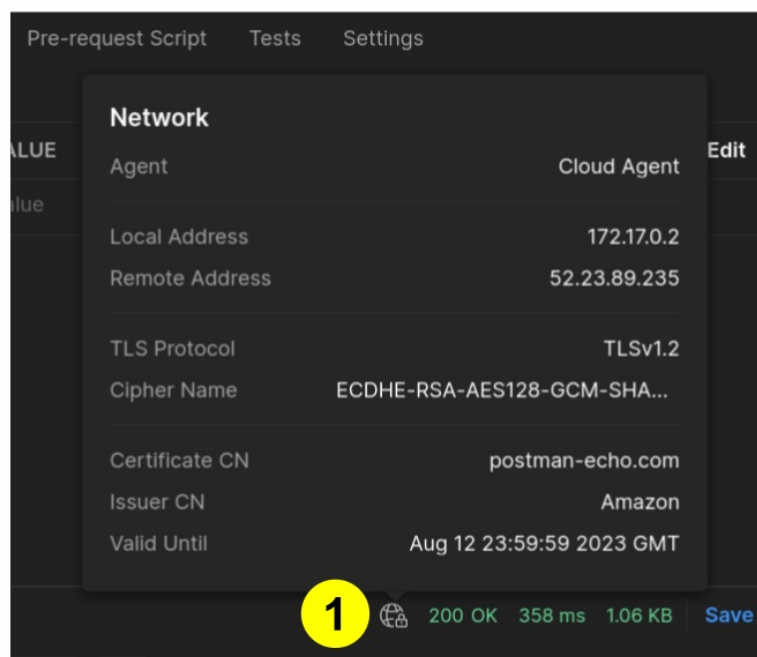
Server Response Info

Postman also displays information when a request is received. This panel has some general information you can get with a quick glance.



The image shows the response information (code, time, and size) from the request.

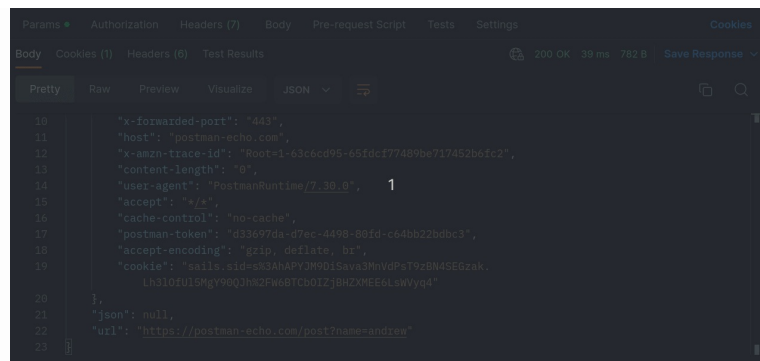
If you are looking for detailed information about the network, hover your mouse over the globe icon(1) next to the status code.



The image shows a single numbered circle next to the globe icon. Clicking this shows additional information about the IP address, TLS connection, and certificate.

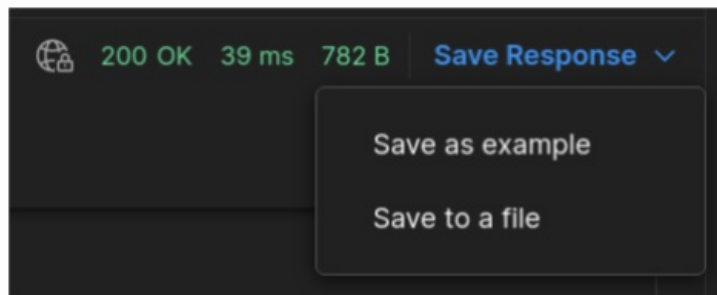
To the right of the globe icon for the network is the status code for the response code that the API returned. Hover over the code itself for a description of what it means.

This is followed by the response **time** and **size**. Postman will automatically calculate the response time from the server in milliseconds. When you hover your cursor over the number, a breakdown of that time by the various response phases is displayed. Next, Postman displays the approximate size of the response. If you hover over the number, you can see the size of the request body and headers.



The image is a gif that shows hovering over the response code, size, or time displays additional information.

Postman also has a function for saving responses from the server. If a request has been saved to a collection, you can also save the response for it. To do this, select **Save Response**, then save as example.



The image shows the options when clicking the “Save Response” button. You can save the example or export it to a file.

Scripts in Postman

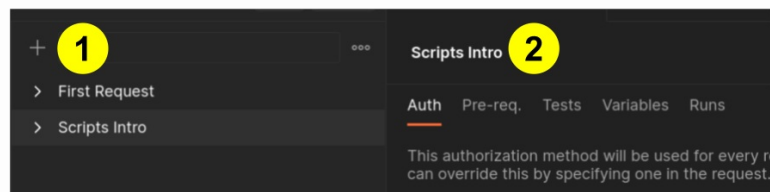
info

Postman Account

Be sure that you have created an [account](#) with Postman and are logged in before continuing.

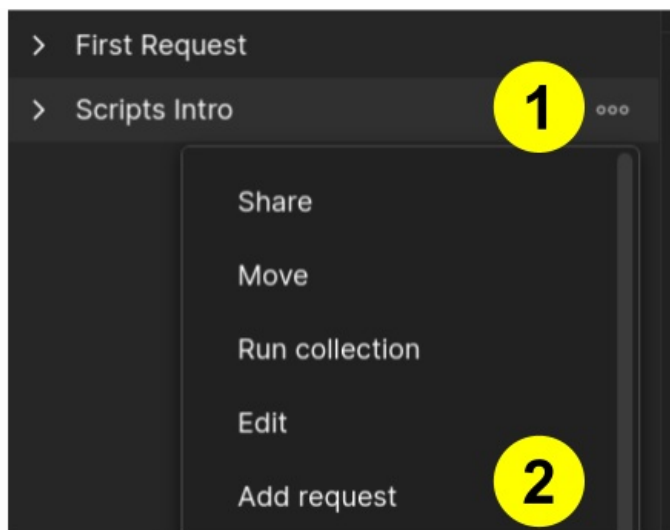
Setup

Before we introduce scripting in Postman, we need to create a collection and some requests. Start by clicking the icon(1) to add a new collection. Then give it the name `Scripts Intro`(2).



The image shows two numbered circles. The first circle is next to the “+” icon which creates a new collection. The second circle is on the field to enter the collection name, which is “Scripts Intro”.

Now, we are going to add a request that will have our scripts. Click the icon with three dots(1) next to the `Scripts Intro` collection. Then select the Add request(2) option. Name this request `Scripts`.

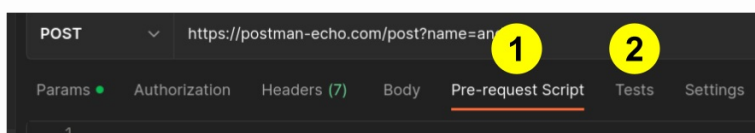


The image shows two numbered circles. The first is the three dot icon next to the collection name. The second circle is next to the button that adds a new request to the collection.

Adding a Script

Postman allows us to add additional logic to requests and collections. It makes it possible to write API tests and create requests that contain dynamic parameters and pass data between requests.

There are two ways to do this by adding JavaScript code. You can add a script before sending a request to the server. This is done on the **Pre-request Script** tab(1). The second way is to add a script that will be executed after receiving a response from the server. It can be added on the **Tests** tab(2).



The image shows two numbered circles. The first circle is next to the tab labeled “Pre-request Script”. The second circle is next to the tab that says “Tests”.

Both types of scripts can be added to individual requests, as well as to folders and collections where they are located.

Working with Cookies

Scripting in Postman relies on an object that starts with `pm`. Since we are working with cookies, the main properties and methods to make cookies work are in the object `pm.cookies`. The main methods are `.toObject()`, `.get()`, and `.has()`.

- The `.toObject()` method returns all cookies as an object. This can be useful for debugging requests.

```
pm.cookies.toObject();
```

- The `.get()` method returns to us the value of a cookie with the given name, which is passed in the input parameter `cookieName`. If there is no cookie with that name, the function will return `undefined`.

```
pm.cookies.get(cookieName);
```

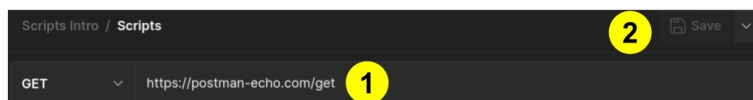
- The `.has()` method checks if there is a cookie with that name. The name is passed in the input parameter `cookieName`. If there is a cookie with that name it returns `true`, otherwise it returns `false`.

```
pm.cookies.has(cookieName);
```

Logging Cookies

Using the above information, let's log information about cookies and a GET request. Make sure that you are using the Scripts request. Use the URL below for the GET request(1). Be sure to save(2) your changes.

```
https://postman-echo.com/get
```

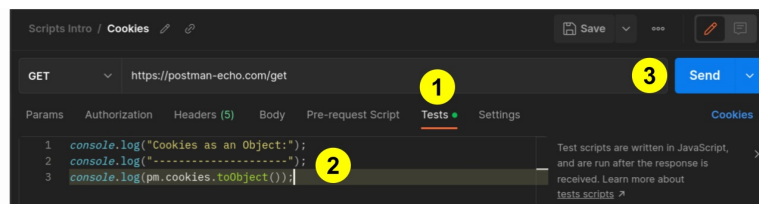


The image shows two numbered circles. The first circle is next to the field for the URL which is "https://postman-echo.com/get".

The second circle is next to the save button.

Open the Tests tab(1) and enter the code sample below(2). The test should log any cookies as an object. Send the request(3) and open the console to see the output.

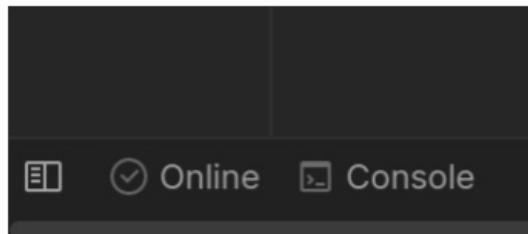
```
console.log("Cookies as an Object:");  
console.log("-----");  
console.log(pm.cookies.toObject());
```



The image shows three numbered circles. The first circle is next to the “Tests” tab. The second circle is in the test script area. It contains code to log information to the console. The exact code can be found above. The third circle is next to the send button.

▼ Where is the console?

You can find a link to the console in the bottom-left corner of Postman. Click this link to open the console.



The image shows the bottom-left corner of the Postman interface. Here, you can see a button labeled “Console”.

You should see the following output:

```
GET https://postman-echo.com/get
Cookies as an Object:
-----
{sails.sid:
"s:3AFYuHEXkiWNA67VxywoKd2pyZGFq4oa.qH5r/AqOUblmd9gGMzp9S/bHIRU8
RXcCe/Qw2bnBBts"}
```

We see that there is a single cookie named `sails.sid`. Now use the `.get()` method to return the value of the cookie. Replace the test code with the code below and send the request again.

```
console.log("Get Cookies 'sails.sid:");
console.log("-----");
console.log(pm.cookies.get("sails.sid"));
```

You should see the following output:

```
GET https://postman-echo.com/get
Get Cookies 'sails.sid':
-----
s:8s21TzSzF_4PveYYE5kQUP6ZiYb6poi.xG9sHMhe4GQTJ7cayCqU06VfTyZ1U
G2nCvBqJgAvygA
```

Finally, we are going to use the `.has()` method to see if the `sails.sid` cookie exists.

```
console.log("Has Cookie 'sails.sid:");
console.log("-----");
console.log(pm.cookies.has("sails.sid"));
```

Send the request once again. You should see the following output:

```
GET https://postman-echo.com/get
Has Cookie 'sails.sid':
-----
true
```

Remove any code from the Tests tab before moving on to the next page.

Variables and Scripts

info

Postman Account

Be sure that you have created an [account](#) with Postman and are logged in before continuing.

Working with Variables

Unlike cookies, the object you work with depends on the scope of the variable. The following table shows the different objects used by variable type.

Variable Type	Postman Object
Local Variable	<code>pm.variables</code>
Collection Variable	<code>pm.collectionVariables</code>
Global Variable	<code>pm.globals</code>
Environment Variable	<code>pm.environment</code>

The most commonly used methods for working with variables are `.has()`, `.get()`, and `.set()`.

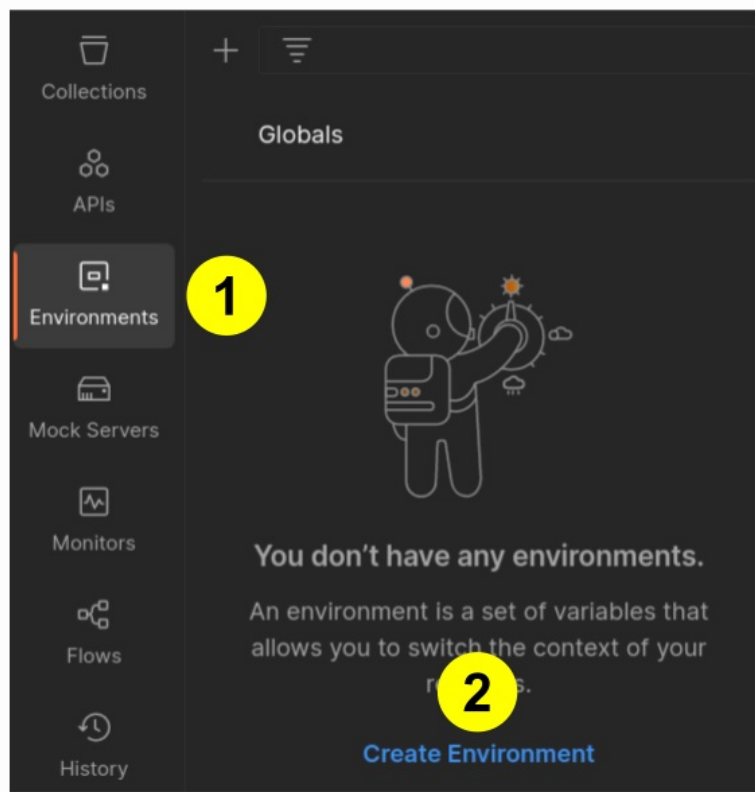
- The `.has()` method checks if the variable exists (returns a boolean).
- The `.get()` method takes the variable name as a parameter and returns its value. If there is no such variable, it will return `undefined`.
- The `.set()` method takes two parameters as input. The first is the variable name and the second is the new value. If there is no such variable, Postman will create it.

Adding Variables

We need to create some variables that have different scopes before we can begin scripting. We have already added a global variable (`myVar`). Follow the steps below to create environment and collection variables.

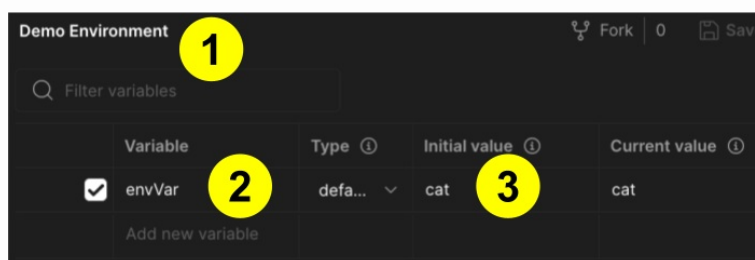
Environment Variable

On the far left, click on Environments(1). Since we do not currently have any environments, click the button at the bottom(2) to make a new one.



The image shows two numbered circles. The first circle is next to the button that says “Environments”. The second circle is next to the button to create a new environment.

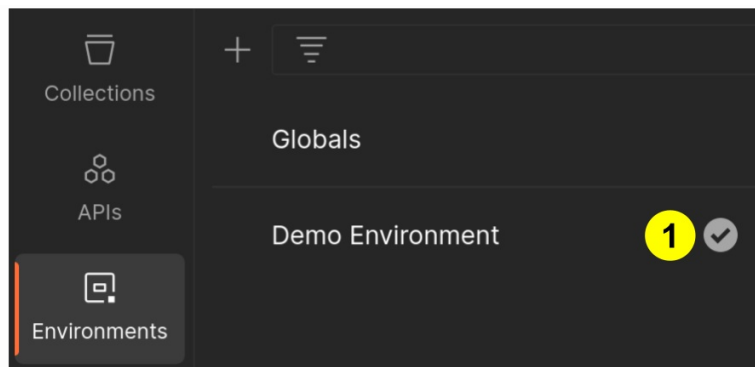
Give the new environment the name Demo Environment(1). In the section below, create the variable envVar(2) and give it an initial value of cat(3).



The image shows three numbered circles. The first circle shows the new environment name “Demo Environment”. The second circle shows the name of the environment variable “envVar”.

The third circle shows the initial value of the variable as “cat”.

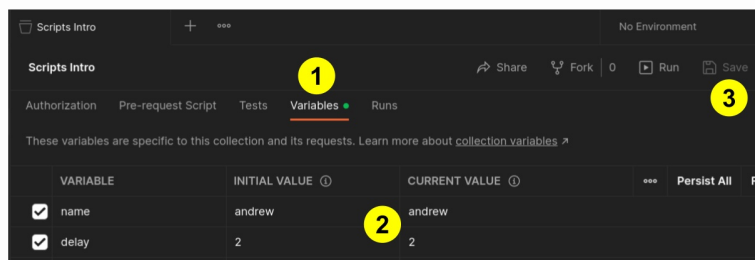
In order to use these newly created environment variables, we need to make sure that Demo Environment is set to active. Under our list of environments, verify that the check mark button(1) is set to active.



The image shows the Demo Environment as being set to active. The check mark button is colored white.

Collection Variables

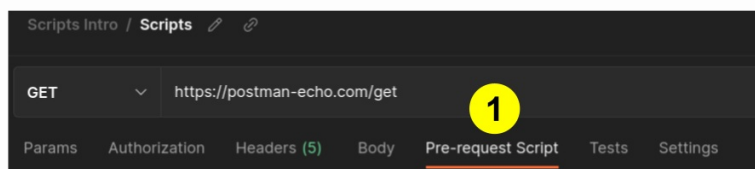
Click on the Script Intro collection. Then click on the Variables(1) tab. Create the variable name that has the value andrew as well as the variable delay that has the value 2(2). Be sure to save(3) your changes.



The image shows three numbered circles. The first circle is next to the “Variables” tab. The second circle shows information about two variables. One is named “name” and has the value “andrew”. The second variable is named “delay” and has the value 2. The third circle is next to the save button.

Logging Variables

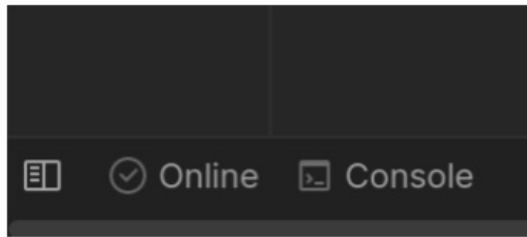
The following code examples show how to work with the different kinds of variables with a script. The code itself will be entered in the Pre-requisite Script tab(1) for the Scripts request. Clicking the Send button will execute the script and log the output to the console.



The image shows a single numbered circle. It is next the “Pre-request Scripts” tab.

▼ Where is the console?

You can find a link to the console in the bottom-left corner of Postman. Click this link to open the console.



The image shows the bottom-left corner of the Postman interface. Here, you can see a button labeled “Console”.

Environment Variables

Accessing environment variables is done with the `pm.environment` object. Copy and paste the code below into the Pre-requisite Script tab and send the request.

```
console.log(pm.environment.has("envVar"));
console.log(pm.environment.get("envVar"));
pm.environment.set("envVar", "dog");
console.log(pm.environment.get("envVar"));
```

You should see the following output:

```
true
"cat"
"dog"
GET https://postman-echo.com/get
```

▼ Why does the request come last?

The scripts that worked with cookies used the Tests tab, while these scripts use the Pre-requisite Script tag. The Pre-requisite script runs before the request, which is why the request came last.

Global Variables

Global variables use the `pm.globals` object.

```
console.log(pm.globals.has("myVar"));
console.log(pm.globals.get("myVar"));
pm.globals.set("myVar", 5);
console.log(pm.globals.get("myVar"));
```


You should see the following output:

```
true
"762"
5
POST https://postman-echo.com/post
```

Collection Variables

```
console.log(pm.collectionVariables.has("name"));
console.log(pm.collectionVariables.get("name"));
pm.collectionVariables.set("name", "rachel");
console.log(pm.collectionVariables.get("name"));
```

You should see the following output:

```
true
"andrew"
"rachel"
POST https://postman-echo.com/post
```

Local Variables

Local variables are different in that they are scoped to a single request. That means you cannot access these variables from another request. Once the request is done, the local variable is no longer available. You do not create local variables through the Postman UI. Instead, we use the `.set()` method to declare the local variable.

```
pm.variables.set("localVar", 42);
console.log(pm.variables.has("localVar"));
console.log(pm.variables.get("localVar"));
pm.variables.set("localVar", false);
console.log(pm.variables.get("localVar"));
```

You should see the following output:

```
true
42
false
POST https://postman-echo.com/post
```

Remove any code from the Pre-request Script tab before moving on to the next page.

Response Data and Scripts

info

Postman Account

Be sure that you have created an [account](#) with Postman and are logged in before continuing.

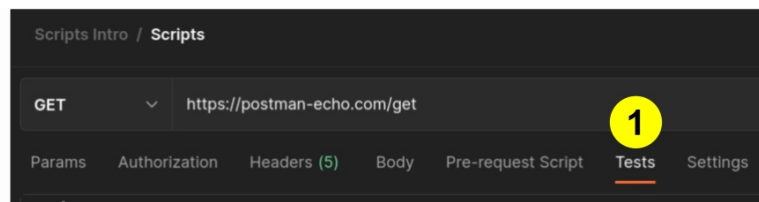
Receiving Response Data

We can handle the response from the server in scripts with the object `pm.response`. Just as on the previous pages, there are several common properties and methods used with this object.

- The `pm.response.code` property contains the response code. For example 200 or 404.
- The `pm.response.status` property contains a description of the status. It can be, for example, OK or Not found.
- The `pm.response.headers` property records all the headers received in the response.
- The `pm.response.responseSize` property contains the size of the response (in bytes) from the server.
- The `pm.response.responseTime` property contains the server response time in milliseconds.
- The `pm.response.text()` method returns the body of the response in text format.
- The `pm.response.json()` method returns the body of the response in JSON format.

Working with Response Data

Keep using the `Scripts` request for the following examples. Next, open the `Tests` tab(1) and remove any previous code. We have to use this tab because we are working with response data. That means the request must be sent after we get a response. Putting the code in the `Pre-request Script` tab means it would execute before we get a response.



The image shows a single numbered circle. It is located next to the Tests tab.

Log each of the properties and methods mentioned above. Add a short description so we can better understand the output.

```
console.log("response code: " + pm.response.code);  
console.log("response status: " + pm.response.status);  
console.log("response headers: " + pm.response.headers);  
console.log("response size: " + pm.response.responseSize);  
console.log("response time: " + pm.response.responseTime);  
console.log("response text: " + pm.response.text());  
console.log("response json: " + pm.response.json());
```

Send the request and open the console (bottom-left corner of Postman). Your output should look something like this:

```
GET https://postman-echo.com/get
response code: 200
response status: OK
response headers: Date: Fri, 20 Jan 2023 17:02:47 GMT
Content-Type: application/json; charset=utf-8
Content-Length: 497
Connection: keep-alive
ETag: W/"1f1-rEvhy+4cP5Vl+pb0IuRHGcv7w"
Vary: Accept-Encoding
set-cookie:
sails.sid=s%3APCmHnifDeEYwsUKm5Yvq4kZ0viy3BU81.JOZkwNK8V8J7LHRQ9
e03qrPJkYLjOi6grvSsqMvX9Ag; Path=/; HttpOnly
response size: 497
response time: 79
response text: {"args":{},"headers":{"x-forwarded-
proto":"https","x-forwarded-port":"443","host":"postman-
echo.com","x-amzn-trace-id":"Root=1-63cac937-
18b8d9886a507f7b5f73b2fc","content-length":"0","user-
agent":"PostmanRuntime/7.30.0","accept":"*/*","cache-
control":"no-cache","postman-token":"aaa25b72-5902-483e-9dc5-
4547f81f2427","accept-encoding":"gzip, deflate,
br","cookie":"sails.sid=s%3A_6ZRYHD0PAVJVfXZddA1BgP2CDotzS0S.Xwg
e9Rvr1h%2FRIRaiPNFBbVZF13q0rpJqSiXoJ4MXbpo"},"url":"https://post
man-echo.com/get"}
response json: [object Object]
```

Notice how the response headers are printed out each one on their own line. In addition, the JSON version of the response is listed simply as [object Object]. Change the code in the script area with the example below. We are only going to look at the headers and the body in JSON format. Importantly, we are going to remove the description string.

```
console.log(pm.response.headers);
console.log(pm.response.json());
```

Send the request once more. You should now see some output similar to the image below. The headers are now represented by a list of JSON objects, and the JSON format of the body is a JSON object. You can click on the ► icons to expand the list and objects.

```
► GET https://postman-echo.com/get
► (7) [{...}, {...}, {...}, {...}, {...}, ...]
  {args: {}, headers: {...}, url: "https://postman-echo.com/get"}
```

The image shows three lines of output in the console. The first line shows a GET request and the URL used. The second line shows a list of objects. There is an arrowhead icon to the left of

the output. The third line shows the body of the request as a truncated JSON object. There is an arrowhead icon to the left of the output.

Remove any code from the `Tests` tab before moving on to the next page.

Request Data and Scripts

info

Postman Account

Be sure that you have created an [account](#) with Postman and are logged in before continuing.

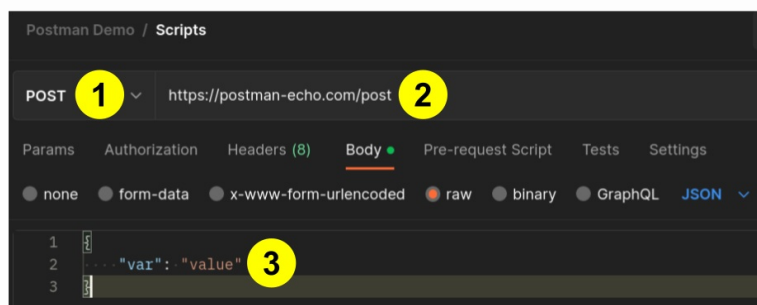
Requesting Data

Similar to response, we can use the `pm.request` object to work with request data in scripts. Change the Scripts request so that it is now a POST(1). Update the request URL(2) to the following:

```
https://postman-echo.com/post
```

Then add the following information as the body of the request(3).

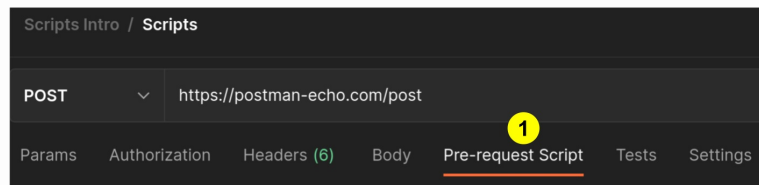
```
{
  "var": "value"
}
```



The image shows three numbered circles. The first circle is next to the request type. The second circle is next to the field for the URL used for the request. The third circle is in the part of Postman used for the body of the request.

Some of the more commonly used properties are `method`, `url`, `headers`, and `body`. Let's add the following code to our script on the Pre-request Script tab(1):

```
console.log("Request method: " + pm.request.method);
console.log("Request url: " + pm.request.url);
console.log("Request headers: " + pm.request.headers);
console.log("Request body: " + pm.request.body);
```



The image shows a single numbered circle. It is next to the tab for pre-request scripts.

After clicking on the Send button in the console, we will see the output below. Since we did not add any headers, Postman returned an empty string.

```
Request method: POST
Request url: https://postman-echo.com/post
Request headers:
Request body: Body:
{
  "var": "value"
}
```

Changing Request Parameters

We can also programmatically change request parameters with a script with the `.addQueryParams()` and `.removeQueryParams()` methods.

The `addQueryParams()` method adds a parameter to the request string. It takes a string containing the new parameter and its value.

```
pm.request.addQueryParams("param1=value1");
```

The `removeQueryParams()` method removes a parameter from the query string. The passed value is the name of the parameter that we want to remove.

```
pm.request.removeQueryParams("param1");
```

Let's use these methods to modify our request. But first, we need to change our request a little. Add two parameters to the request URL:


```
https://postman-echo.com/post?param1=value1&param2=value2
```

Then on the Pre-request Script tab remove any existing code and replace it with the following:

```
pm.request.addQueryParams("param3=value3");  
pm.request.removeQueryParams("param1");
```

After clicking Send, we will see the following in the console:

```
POST https://postman-echo.com/post?param2=value2&param3=value3
```

challenge

Try this variation:

Update the script in the Pre-requisite Script tab to the code below. Then send the request.

```
pm.request.addQueryParams("param3=value3");  
pm.request.removeQueryParams("param1");  
pm.request.addQueryParams("param1=value1");
```

▼ Did you notice?

The parameter param1 is now at the end of the output. That is because the parameter was first deleted. After adding the parameter back again, it then goes to the end of the list.

```
POST https://postman-echo.com/post?  
param2=value2&param3=value3&param1=value1
```

Remove any code from the Pre-request Script tab before moving on to the next page.

Combining Scripts

info

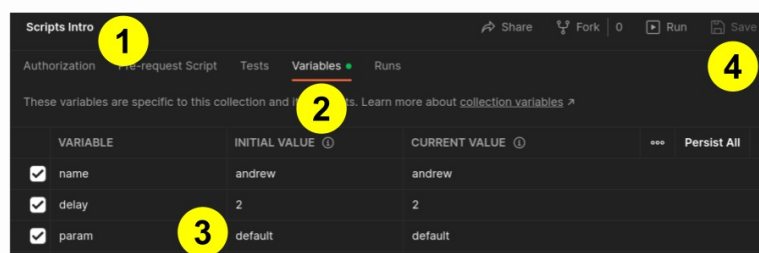
Postman Account

Be sure that you have created an [account](#) with Postman and are logged in before continuing.

Example

Imagine you have a collection of requests. Run it sequentially. The first request returns a value, which you then need to pass to the second request, but before sending it, we need to first process that value. In this case, after executing the first request in the test scripts, we will save the resulting value in a variable that will also be available to the second request. In the Pre-request Scripts of the second request, we get the value from the variable, process it and store it in this variable again. Then in the second request, we use the processed value when sending the request.

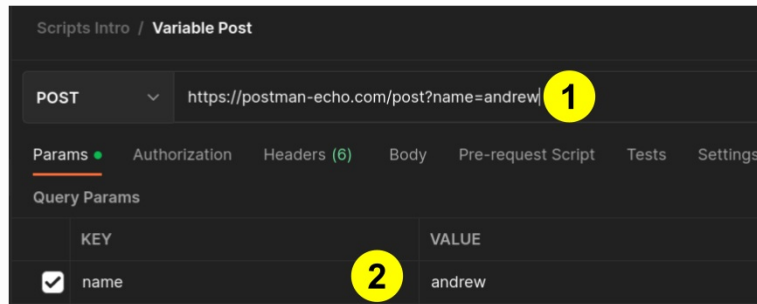
Let's try to write such a script. Open the collection named `Scripts Intro`(1). Click on the `Variables` tab(2) to create a variable. Add a collection-level variable named `param` with a value of `default`(3). Save the changes (4).



The image shows four numbered circles. The first circle is next to the tab with the name of the collection. The second circle is next to the tab labeled “Variables”. The third circle is in the variable declaration area. The fourth circle is next to the save button.

Create a new request named `Variable Post`. Make sure that it is saved to the `Scripts Intro` collection. The newly created `POST` request should use the value below for the request URL(1). This URL will automatically add `name` and `andrew` as query parameters(2).

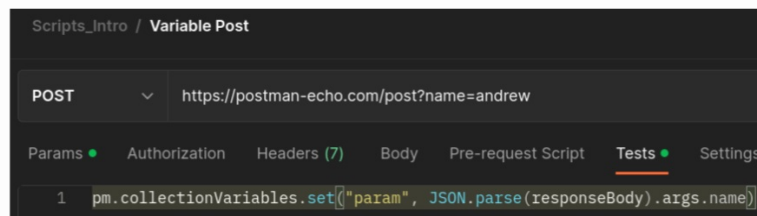
```
https://postman-echo.com/post?name=andrew
```



The image shows two numbered circles. The first circle is next to the field for the URL of the request. The URL is “https://postman-echo.com/post?name=andrew”. The second circle is in the query parameter area. The key name is “name” and the value is “andrew”.

Open the Tests tab. This is where we will add the code that saves the value of the parameter from the response body into the param variable. The code will be executed after receiving the response from the server. Be sure to save this request.

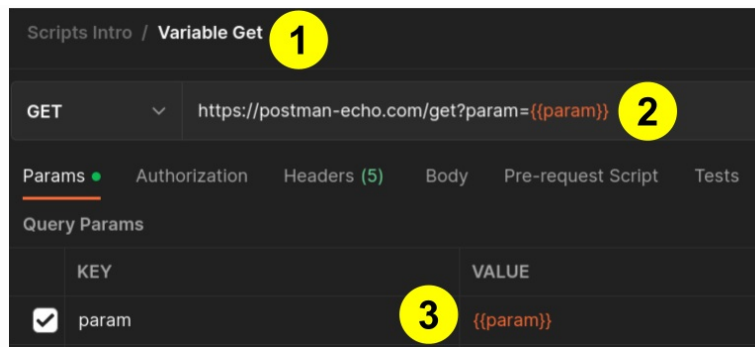
```
pm.collectionVariables.set("param",  
    JSON.parse(responseBody).args.name);
```



The image shows the test script area. The code in the test script is “pm.collectionVariables.set(“param“,
JSON.parse(responseBody).args.name);”.

Now, create a separate GET request named Variable Get(1). We are still working inside the Script Intro collection. The request should use the request URL below(2). After entering the URL, Postman should automatically come up with the query parameters(3).

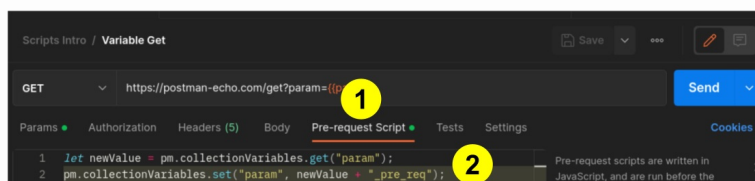
```
https://postman-echo.com/get?param={{param}}
```



The image shows three numbered circles. The first circle is next to the new request name which is “Variable Get”. The second circle is in the field for the URL which is “https://postman-echo.com/get?param={{param}}”. The third circle is in the query parameter area. The key is “param” and the value is “{{param}}”.

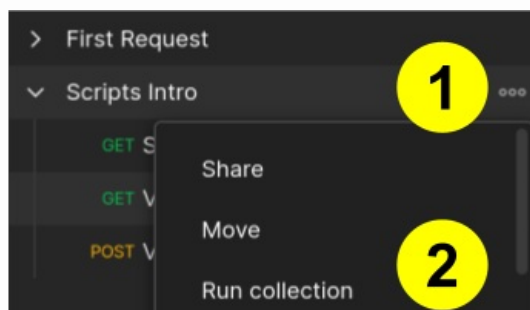
Open the Pre-request Script tab(1), and add the code below(2). The value of "param" is stored in newValue. Then the value for "param" is set to newValue concatenated with "_pre_req". Be sure to save this request.

```
let newValue = pm.collectionVariables.get("param");
pm.collectionVariables.set("param", newValue + "_pre_req");
```



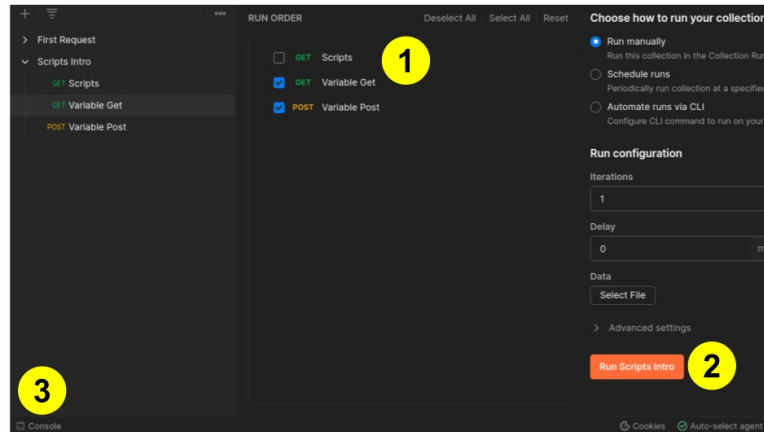
The image shows two numbered circles. The first is next to the tab labeled “Pre-request Script”. The second circle is in the script area. The code for the scripts is found above.

Run collection and open the Postman console. We do this by clicking on the icon with three dots next to the Script Intro collection(1). Select Run collection(2).



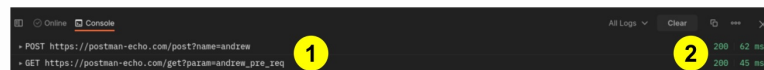
The image shows two numbered circles. The first is next to the three dots icon for the collection. The second circle is next to the button to run the collection.

In the “Runner” tab, un-check the Scripts request(1) as we only want to run the scripts associated with the other two requests. Click the button that says Run Scripts Intro(2). Then open the Postman console(3), which is near the bottom-left corner.



The image shows three numbered circles. The first circle is next to an un-checked box for the Scripts request. The second box is next to an orange button that runs the scripts in the collection. The third circle is next to the button for the console which is in the bottom-left corner of the screen.

You should see the following results(1) as well as the 200 response code(2) that everything worked as expected.



The image shows two numbered circles in the console. The first circle shows the request type and its associated URL. The second circle is to the right of the screen and shows 200 response codes for each request.

Remove any code from the Pre-request Script and Test tabs before moving on to the next page.

Request-Level Test Scripts

info

Postman Account

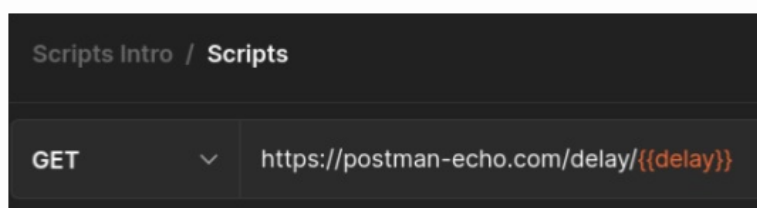
Be sure that you have created an [account](#) with Postman and are logged in before continuing.

Test Scripts for a Request

Now that we have a basic understanding of how scripts work in Postman, we can use them for testing. We can check if the API works correctly by comparing the result to an expected value. In addition to individual requests, test scripts can be added to collections or folders, as well as pre-request scripts.

Let's continue working with the Scripts request in the Scripts Intro collection. Modify the request so that it is a GET request with the following URL:

```
https://postman-echo.com/delay/{{delay}}
```



The image shows a GET request with the address “https://postman-echo.com/delay/{{delay}}”.

If you recall, the variable `delay` was previously created. It is a collection-level variable with the initial value of 2. Open the Tests tab(1) and add the following code(2):

```
pm.test("Check status", function() {
    pm.response.to.have.status(200);
});

pm.test("Check delay", function() {

    pm.expect(pm.response.json().delay).to.equal(pm.collectionVariables.get("delay"));

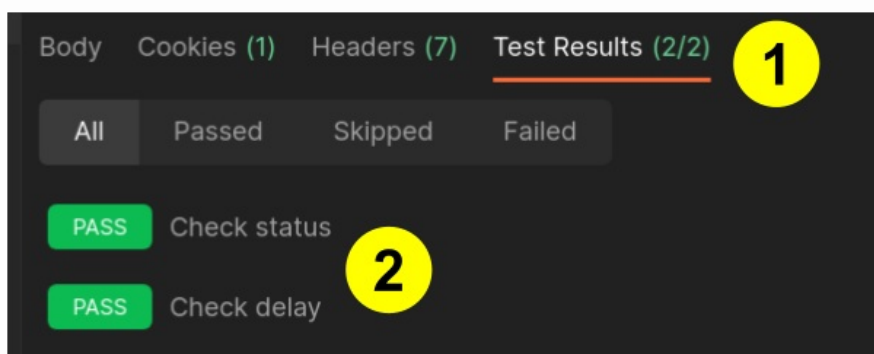
});
```



The image shows two numbered circles. The first circle is above the tab that says “Tests”. The second circle is in the area for writing tests. The code for the two functions in the test is listed above.

In the first call to `pm.test`, we check that the response code is 200. In the second, we check that the value of `delay` in the body of the response is the same as the value of the collection variable we used when we sent the request.

Let’s click Send and execute our request. Then open the Test Results tab(1). Near the name of the tab we see the number of tests passed successfully(2), as well as the total number of tests.



[Click here to see a larger version of the image](#)

challenge

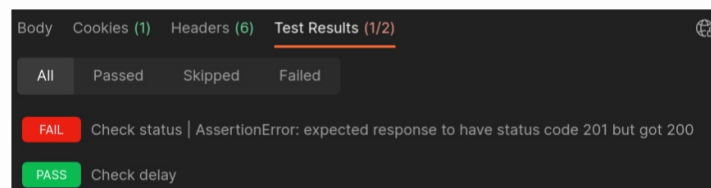
Try this variation:

Change the expected value in the first test to 201 and run the request again.

```
pm.test("Check status", function() {  
  pm.response.to.have.status(201);  
});
```

▼ Why is there an error?

As we expected, the first check failed because the expected response code does not match the one we received. The check itself is now marked in red. Details of the error are available, next to the description.



The image shows the test results after the change. This time, the Check status test failed. The error message says, “AssertionError: expected response to have a status code 201 but got 200”.

Remove any code from the Tests tab before moving on to the next page.

Collection & Folder-Level Test Scripts

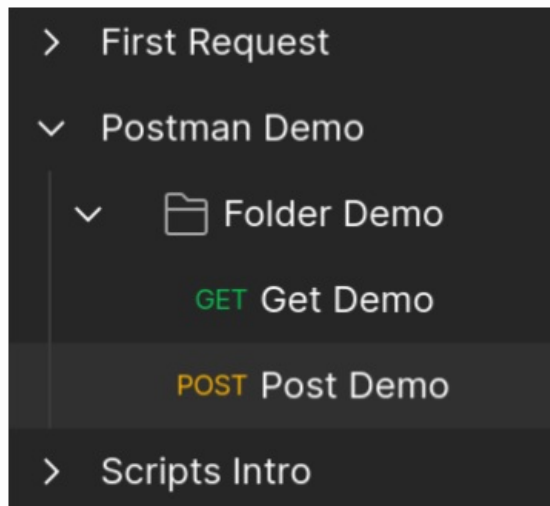
info

Postman Account

Be sure that you have created an account with Postman and are logged in before continuing.

Test Scripts for Folders and Collections

For this example, we are going to create test scripts that work across an entire collection. Start by creating a new collection called `Postman Demo`. In it, create a new folder named `Folder Demo`. Inside the folder we will add two requests, a `GET` request named `Get Demo` and a `POST` request named `Post Demo`. Use the three-dots icon next to the `Folder Demo` name to create requests for the folder.



The image shows the structural hierarchy of the “Postman Demo” collection. Inside this collection is a folder called “Folder Demo”. In this folder is a request called “Get Demo” and another request called “Post Demo”.

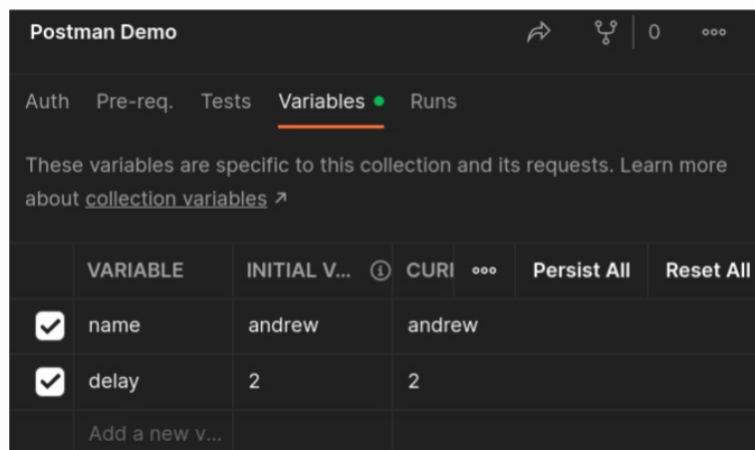
For the `GET` request, use the following request URL:

```
https://postman-echo.com/get?name={{name}}
```

And for the POST request, use the following request URL:

```
https://postman-echo.com/post?name={{name}}
```

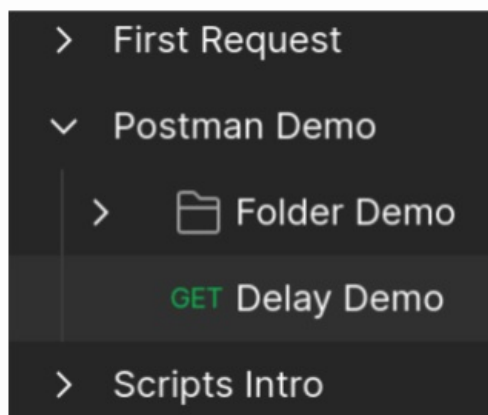
At the collection level, create the variable name and give it the value andrew. Create another variable delay whose value is 2. Be sure to save your work.



The image shows the variables associated with the Postman Demo collection. The first variable is “name” and the value of “andrew”. The second variable is “delay” and has the value of 2.

Inside the collection, add another GET request named Delay Demo. Use the three-dots icon next to the Postman Demo name to create a request for the collection. Use the following URL for the request:

```
https://postman-echo.com/delay/{{delay}}
```



The image shows the Post Demo collection with the folder “Folder Demo” collapsed so you cannot see any of the requests in the folder. Outside of the folder is another request called “Delay

Demo”.

The delay parameter will be set from a collection variable of the same name. Now let's add the script at the level of the last Delay Demo request(1). Add the code below in the test area(2). Be sure to save your work.

```
pm.test("Check delay", function() {  
    pm.expect(pm.response.json().delay).to.equal(pm.collectionVariables.get("delay"));  
});
```



The image shows two numbered circles. The first circle is above the tab labeled “Tests”. The second circle is in the test script area where the function “Check delay” is defined with the code sample above.

Next, let's add another test script that will be at the folder level. It will run only for the two requests that are in that folder. Click on Folder Demo on left. This will open a tab for the folder. Select the Tests tab(1) and add the following code(2):

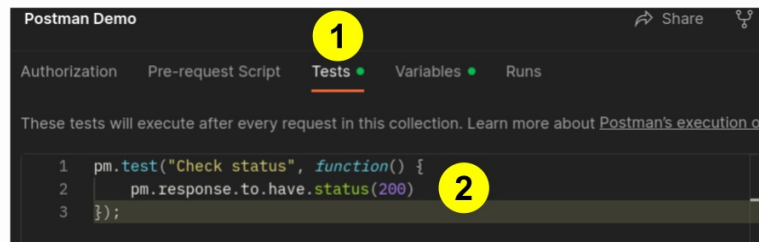
```
pm.test("Check name", function() {  
    pm.expect(pm.response.json().args.name).to.equal("andrew");  
});
```



The image shows two numbered circles in the settings for the folder called “Folder Demo”. The first circle is above the tab labeled “Tests”. The second circle is in the test script area. The code for the “Check name” function is found above.

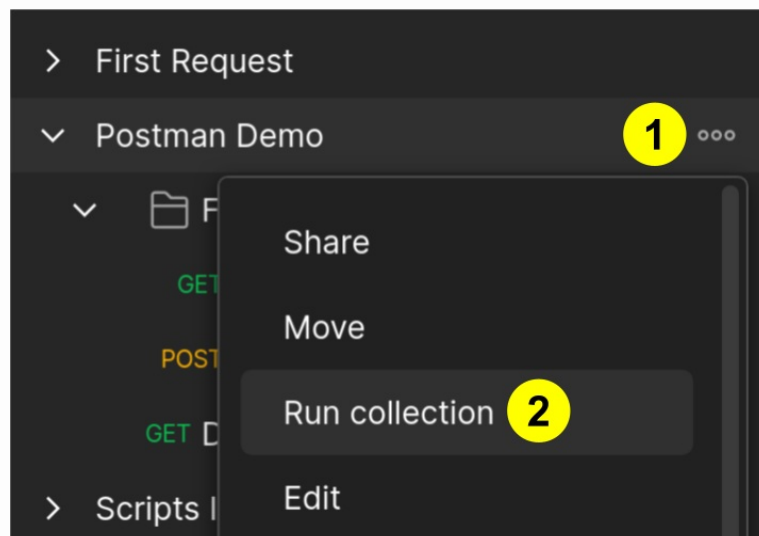
The last test script will be added at the collection level. It will run for all requests in the collection. Click on Postman Demo on left. This will open a tab for the collection. Select the Tests tab(1) and add the following code(2):

```
pm.test("Check status", function() {  
  pm.response.to.have.status(200)  
});
```



The image shows two numbered circles in the settings for the collection called “Postman Demo”. The first circle is above the tab labeled “Tests”. The second circle is in the test script area. The code for the “Check status” function is found above.

Make sure that you have saved all of the tests. To run the collection, click on the three-dots icon next to Postman Demo collection(1). Then select Run collection(2).



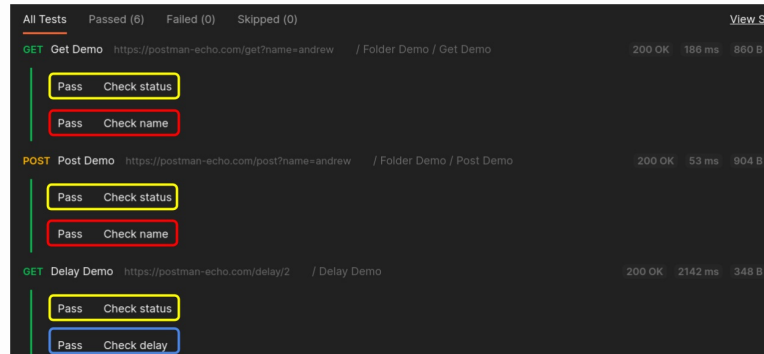
The image shows two numbered circles in the left menu. The first circle is next to the icon of three dots for the Postman Demo collection. The second circle is next to the button to run the collection.

Postman will display the results of the tests. Even though we wrote three tests, we see results for six different tests. Here is how Postman executed the tests:

- Check status (yellow) - This is a collection-level test, which means it

runs for the three requests.

- Check name (red) - This is a folder-level test, which means it only runs on those requests in the folder — Get Demo and Post Demo.
- Check delay (blue) - This is a request-level test, which means it only runs on the Delay Demo request.



The image shows the results from all of the above test scripts. The results are outlined in yellow, red, or blue. The Check status test is outlined in yellow and appears three times. The Check name test is outlined in red and appears twice. The Check delay test is outlined in blue and appears once.