Lab #8

Course Title: Secure Software Design and Engineering

Lab Title: API Testing and Exploration Using Postman

What is an API?

Definition

API stands for **Application Programming Interface**. It is a set of rules and protocols that allows different software applications to communicate with each other.

In simpler terms, an API acts as a **bridge** between two systems, enabling one program to request specific information or services from another.

Why Are APIs Important?

APIs are essential because they:

- Allow software systems to exchange data in a structured, predictable way.
- Enable developers to build complex applications more efficiently by reusing existing functionality.
- Separate internal system logic from the way external users interact with it, which improves security and maintainability.

Real-World Example

Imagine you are using a weather application on your phone. When you check the current temperature:

- Your app sends a request to a weather service API.
- The weather API processes this request on the server.
- It returns the temperature and other weather data to your app.
- The app displays the information to you.

You never interact directly with the weather service's database or servers. The API handles that interaction in the background.

Types of APIs

1. **Web APIs (or HTTP APIs)**
   These are the most common and are accessed via the internet using HTTP protocols. Examples include:
   - REST APIs
   - SOAP APIs
2. **Library or Framework APIs**
   Provided by libraries like jQuery or frameworks like .NET, these APIs give developers access to pre-built functions and components.
3. **Operating System APIs**
   These allow software applications to interact with the operating system. For example, Windows API or Android API.

Common Use Cases of APIs

- Retrieving data from a database or third-party service (e.g., getting stock prices, news, weather updates).
- Sending data to another system (e.g., submitting a form, posting a tweet).
- Authenticating users via services like Google or Facebook.
- Automating workflows between multiple systems (e.g., syncing data between a CRM and an email marketing tool).

How APIs Work: Basic Flow

1. **Client**: Sends a request to the API (for example, asking for a list of users).
2. **API**: Receives the request, processes it, and interacts with the server or database.
3. **Server**: Sends the requested data back to the API.
4. **API**: Formats the response and sends it back to the client.

What is Postman?

Postman was created in 2012 by software engineer **Abhinav Asthana**. He wanted to make API testing easier, so he built Postman as a side project. Today, it’s a full platform used to design, build, and test APIs. It helps teams work together more easily and build better APIs faster.

Postman lets users store and manage everything related to APIs in one place — like documentation, test cases, results, and metrics. It’s now used by over **20 million people** worldwide.

How to Get Postman

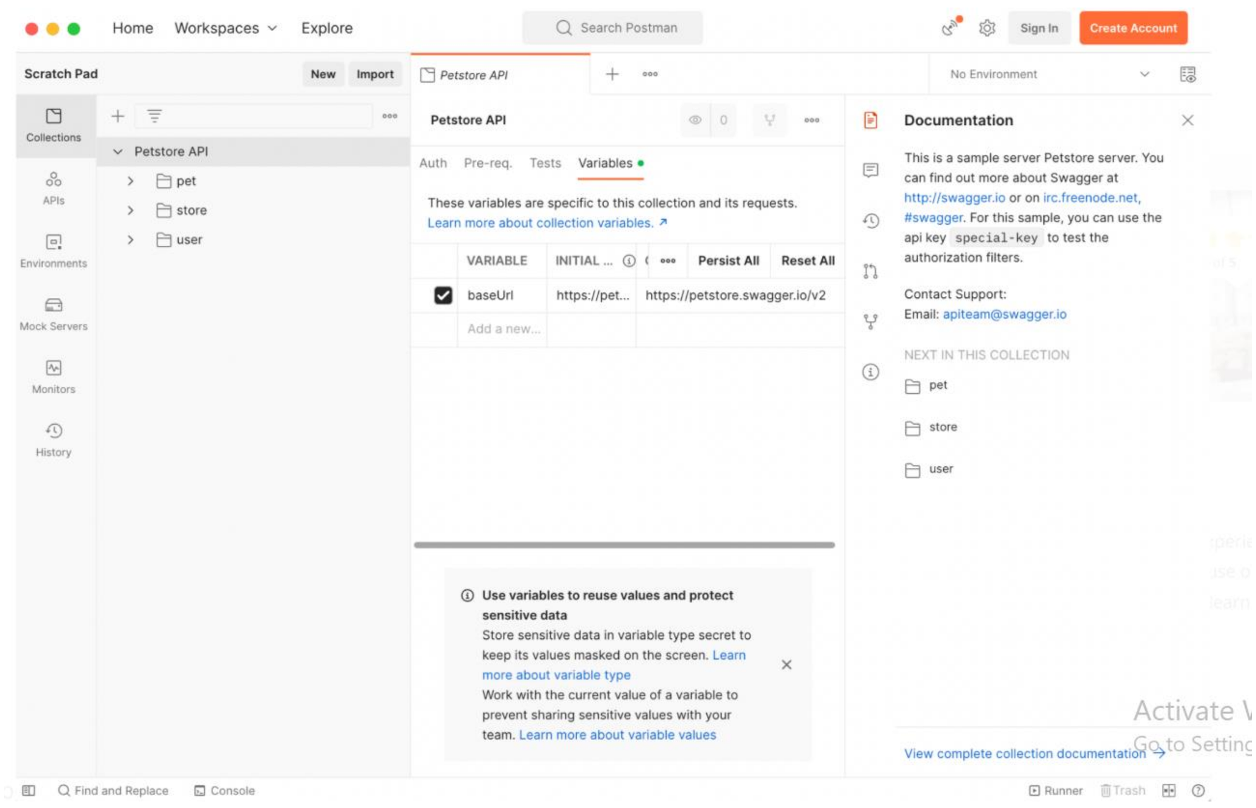
You can **download Postman** from its official website. Just choose the version that works with your computer’s operating system. For Mac users, make sure to download the version that matches your laptop’s chip.

If you don’t want to install it, you can use the **web version** instead — but you’ll need to sign up for a free Postman account.

For the **desktop app**, you can use it without signing in, but some features will be limited.

Getting started with Postman

Postman’s central view is the **workspace** where all the things we’re going to use are positioned. Workspaces help us organize our API work and collaborate with teams across the organization. Inside our workspace we can access collections, environments, mock servers, monitors and other Postman features. **Collection** is a group of saved API requests, while **environments** is a set of variables that we can use in the requests.



These variables are specific to this collection and its requests.
Learn more about collection variables. ↗

	VARIABLE	INITIAL ...	...
✓	baseUrl	https://pet...	https://petstore.swagger.io/v2
	Add a new...		

Persist All Reset All

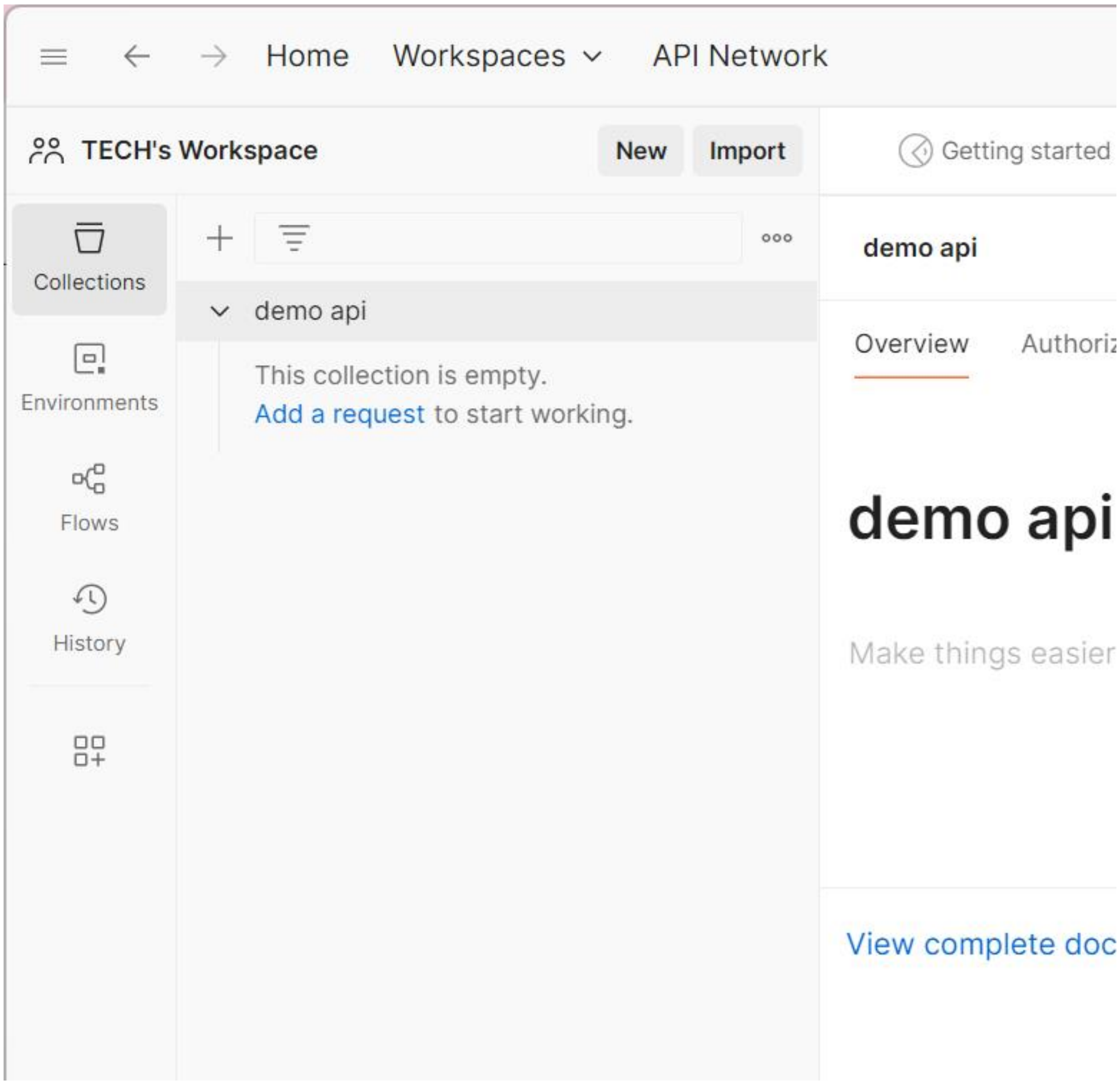
Use variables to reuse values and protect sensitive data
Store sensitive data in variable type secret to keep its values masked on the screen. Learn more about variable type
Work with the current value of a variable to prevent sharing sensitive values with your team. Learn more about variable values

This is a sample server Petstore server. You can find out more about Swagger at http://swagger.io or on irc.freenode.net, #swagger. For this sample, you can use the api key `special-key` to test the authorization filters.

Contact Support:
Email: apiteam@swagger.io

Step 1: Create a New Collection

1. Click **“+ New”** → Choose **Collection**
2. Enter the name: Simple Books API or demo api.
3. Save and expand the collection in the sidebar



demo api

This collection is empty.
Add a request to start working.

demo api

Make things easier

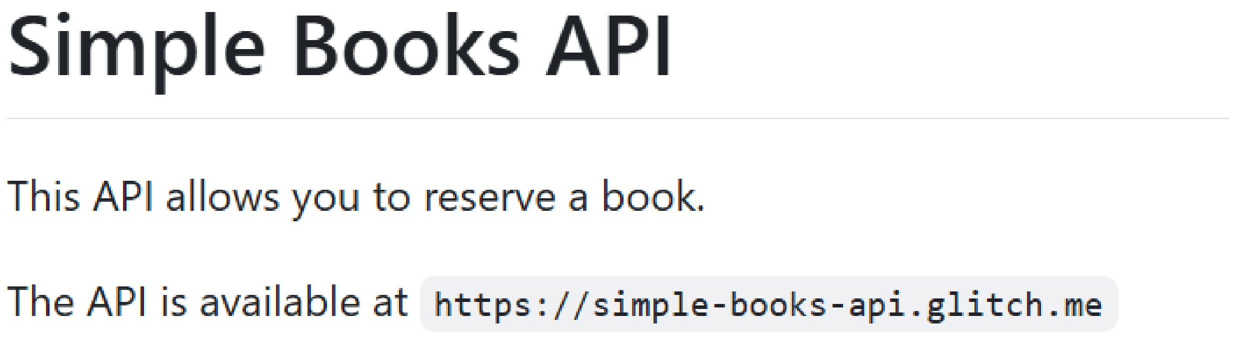
View complete doc

Step 2: Create and Send Your First Request

We'll use the **Simple Books API** from this link:
Documentation: Simple Books API
Base URL: https://simple-books-api.glitch.me

Example 1: GET all books

1. Click **"+"** tab to create a new request.
2. Method: **GET**
3. URL: https://simple-books-api.glitch.me/books
4. Click **Send**
5. Observe the **response body**, **status code**, and **headers**.

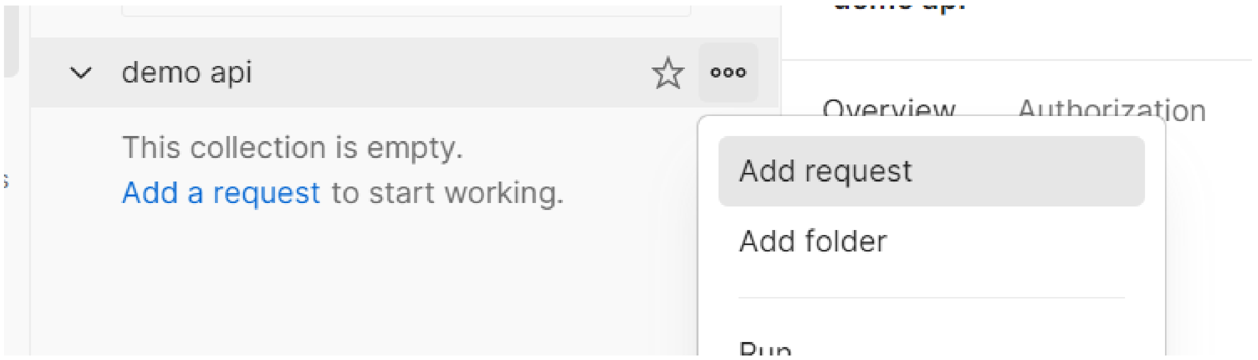


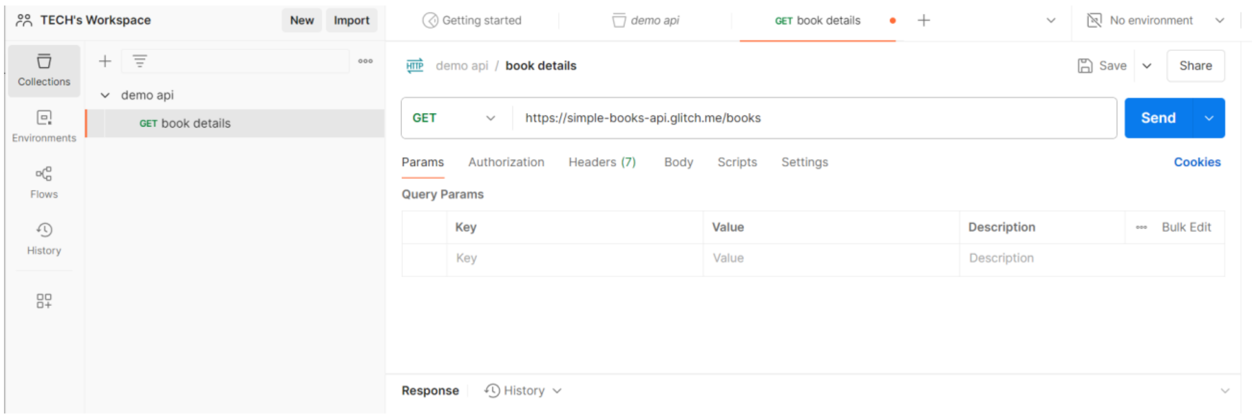
Simple Books API

This API allows you to reserve a book.

The API is available at `https://simple-books-api.glitch.me`

Copy this url https://simple-books-api.glitch.me





Step 3: Use Query Parameters

The endpoint supports two optional parameters:

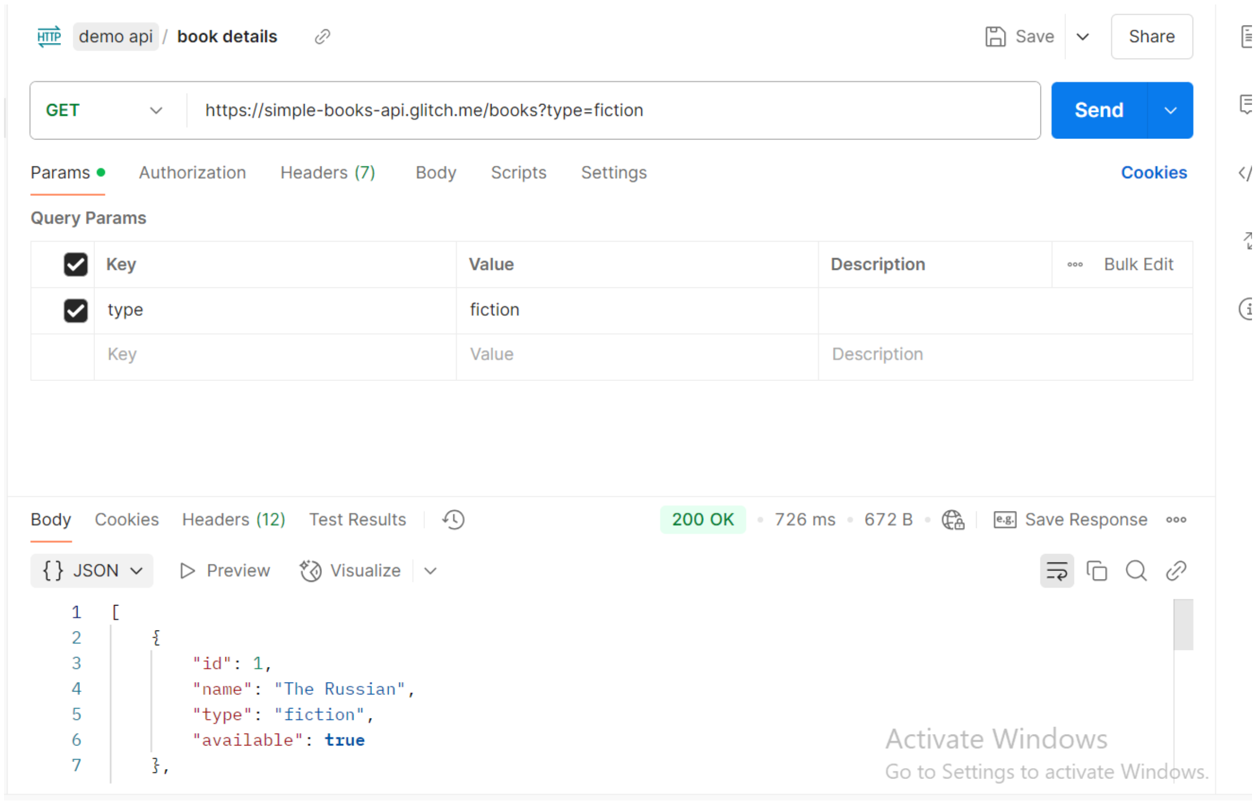
- type: fiction / non-fiction
- limit: any number from 1 to 20

Example:

URL: https://simple-books-api.glitch.me/books?type=fiction&limit=5

Steps:

1. Click on **Params** tab.
2. Add key-value pairs:
   - type → fiction
   - limit → 5
3. Click **Send** and check filtered results.



GET https://simple-books-api.glitch.me/books?type=fiction

Query Params

Key	Value	Description
✓ type	fiction	

```
[
    {
        "id": 1,
        "name": "The Russian",
        "type": "fiction",
        "available": true
    },
```

Step 4: Use Path Parameters

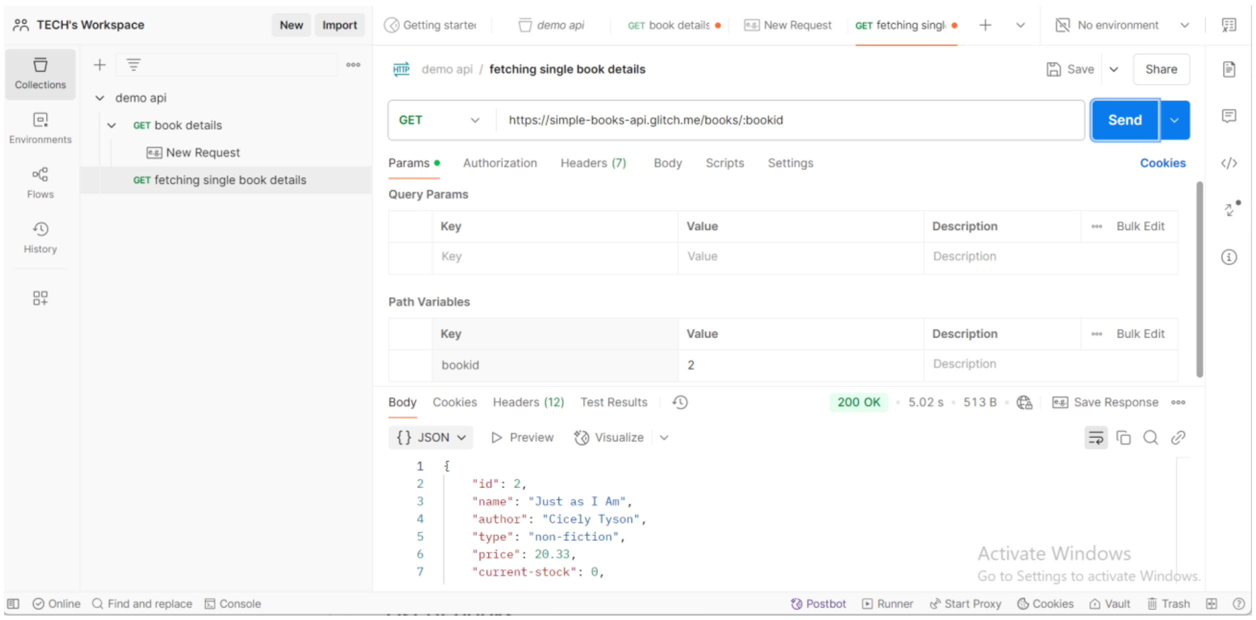
The API allows retrieving a **single book** using its ID.

Example:

URL: https://simple-books-api.glitch.me/books/1

Steps:

1. Create a new request.
2. Method: **GET**
3. Paste the URL with book ID: /books/1
4. Click **Send**
5. Check the JSON response for book details.



demo api / fetching single book details

GET https://simple-books-api.glitch.me/books/:bookid

Query Params

Key	Value	Description
Key	Value	Description

Path Variables

Key	Value	Description
bookid	2	Description

200 OK · 5.02 s · 513 B

```
{
    "id": 2,
    "name": "Just as I Am",
    "author": "Cicely Tyson",
    "type": "non-fiction",
    "price": 20.33,
    "current-stock": 0,
```

Common HTTP Methods Used in APIs

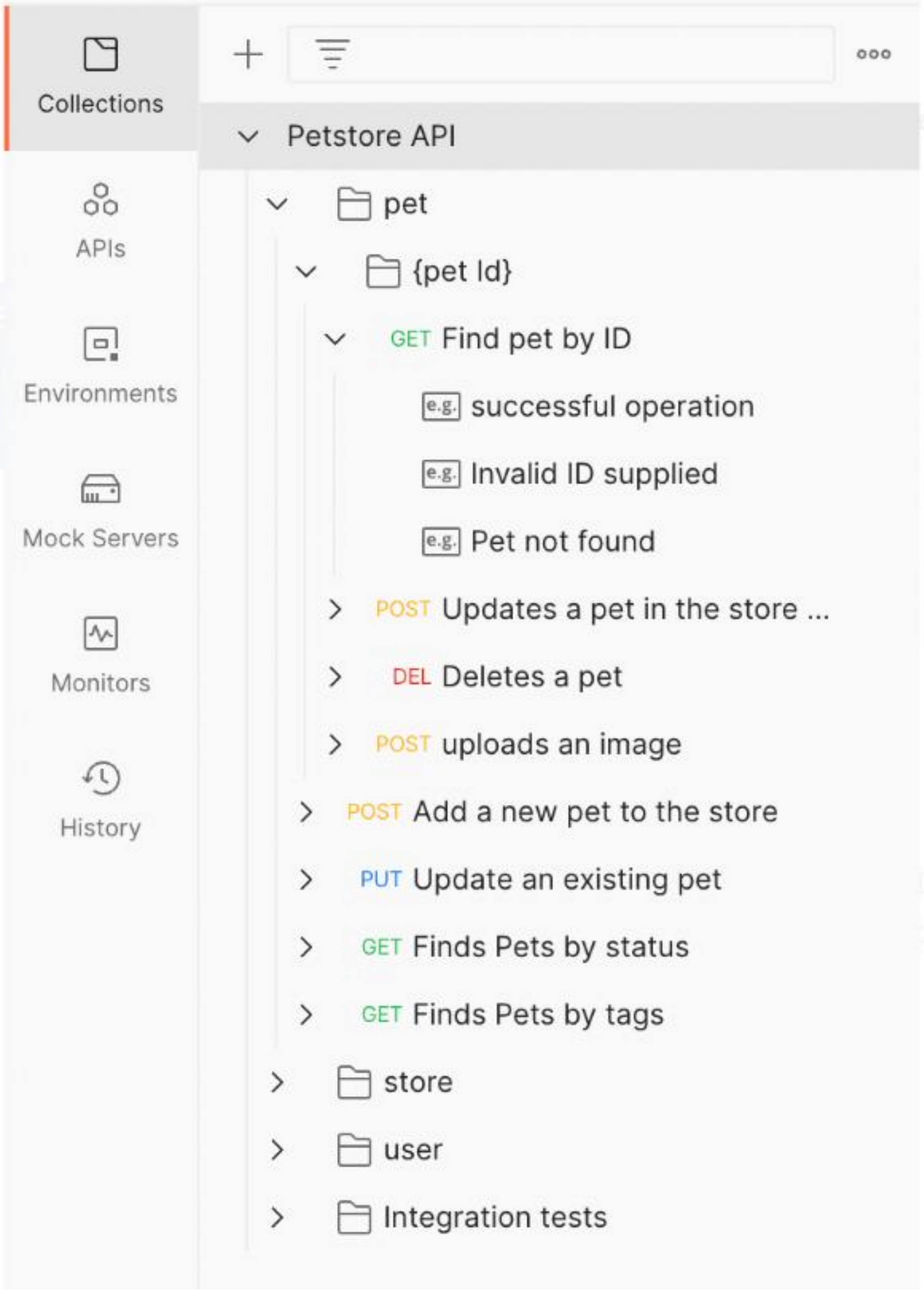
Method	Description	Usage Example
GET	Retrieve data from the server.	Get a list of books.
POST	Send data to the server to create.	Create a new book order.
PUT	Replace existing data completely.	Update a full book entry.
PATCH	Update partial data.	Modify only the book title.
DELETE	Remove data from the server.	Delete a book/order.

How to structure a collection in Postman

As mentioned above, we group all API requests in a collection, but we also group our API tests in a collection. This means that the collection is a root folder of our project. Inside it we can organize our work in folders and subfolders. It is up to you to decide how you would like to structure the collection, however, the most common practice is that:

- Folders are created for every endpoint. The name of the folder is usually the name of the endpoint.

- An *Integration tests* folder can also be created where we can place API tests that test the connection between different API endpoints. Namely, when we want to chain multiple API requests to test, for example, if the data in one endpoint is present in the other related endpoint.



Petstore API
pet
{pet Id}
GET Find pet by ID
e.g. successful operation
e.g. Invalid ID supplied
e.g. Pet not found
POST Updates a pet in the store ...
DEL Deletes a pet
POST uploads an image
POST Add a new pet to the store
PUT Update an existing pet
GET Finds Pets by status
GET Finds Pets by tags
store
user
Integration tests

Practical examples using Postman

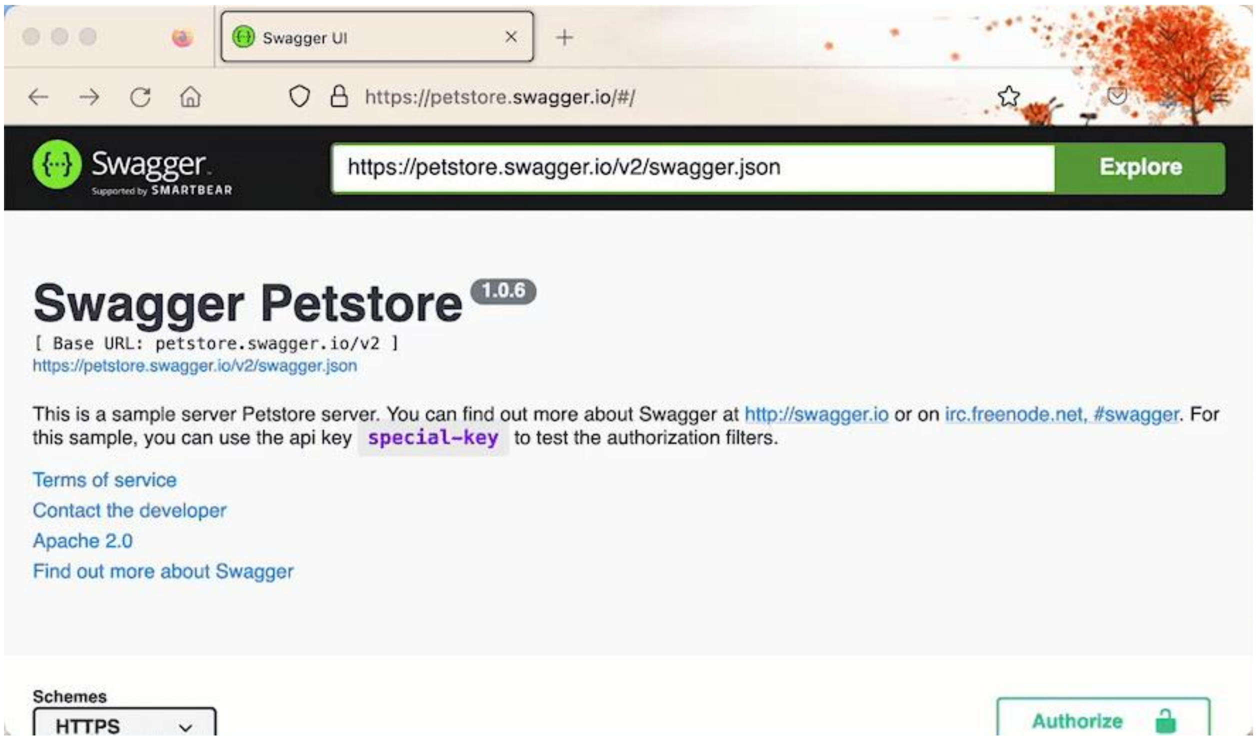
Blog for help **: https://www.testdevlab.com/blog/using-postman-for-api-testing**

I will use the Petstore API from Swagger for the examples. I'd also like to encourage you to take the time to explore the documentation, if available, for every API you use. Exploring the documentation will give you a clear overview of the API, what is expected in each request, and what response you will receive for it. The documentation for Petstore API is available here.

What is Swagger?

Swagger is a set of **open-source tools** built around the **OpenAPI Specification (OAS)**. It helps developers:

- **Design**, **build**, **document**, and **consume** RESTful APIs.
- Automatically generate interactive documentation for APIs.
- Test APIs without writing custom front-end tools.


Swagger UI
https://petstore.swagger.io/#/
Swagger
Supported by SMARTBEAR
https://petstore.swagger.io/v2/swagger.json
Explore

Swagger Petstore 1.0.6

[Base URL: petstore.swagger.io/v2]

https://petstore.swagger.io/v2/swagger.json

This is a sample server Petstore server. You can find out more about Swagger at http://swagger.io or on irc.freenode.net, #swagger. For this sample, you can use the api key `special-key` to test the authorization filters.

Terms of service

Contact the developer

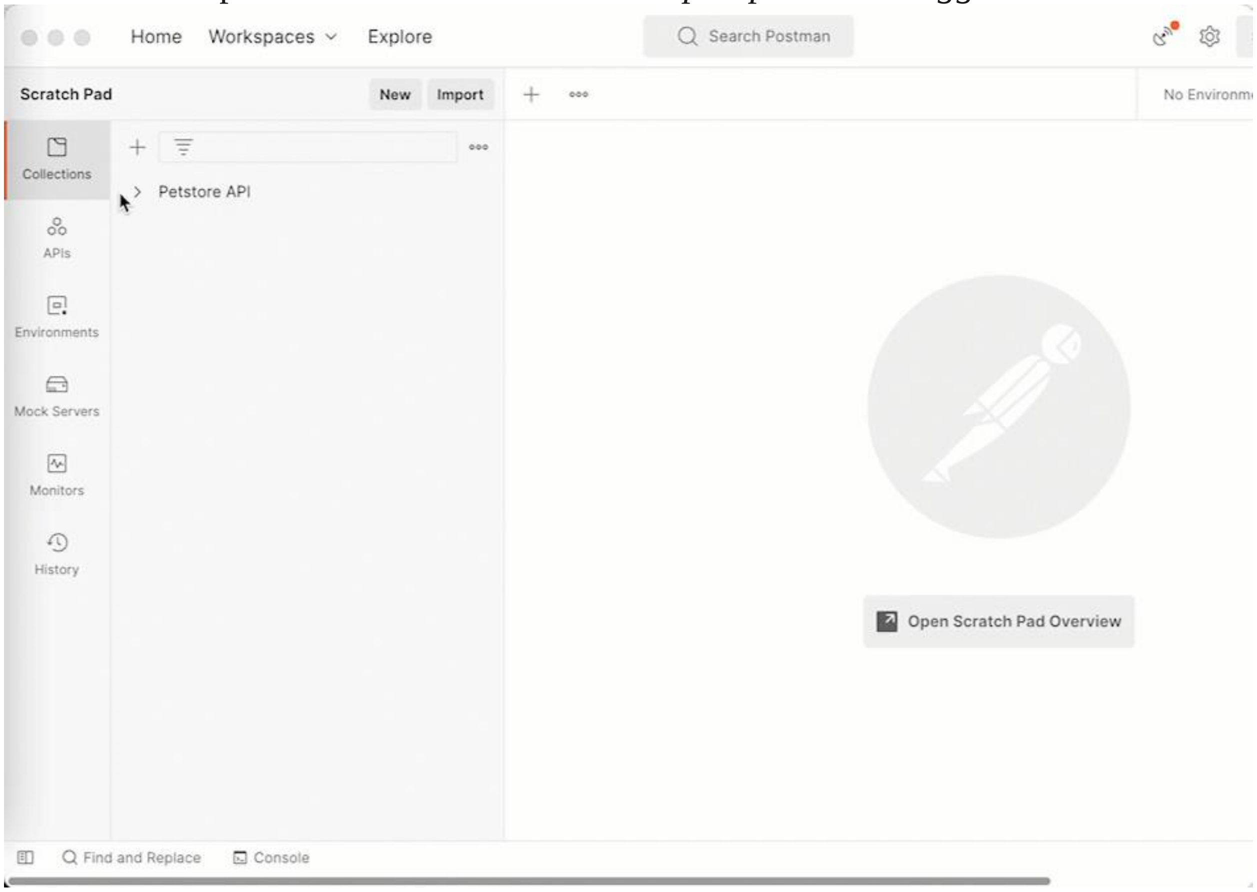
Apache 2.0

Find out more about Swagger

\Exploring Petstore API documentation

In the documentation, the base URL of the API is written as: petstore.swagger.io/v2. Since the base URL is the same for every API request, it makes sense that we store it as a variable in our collection so that we can reuse it. To do that, follow these steps:

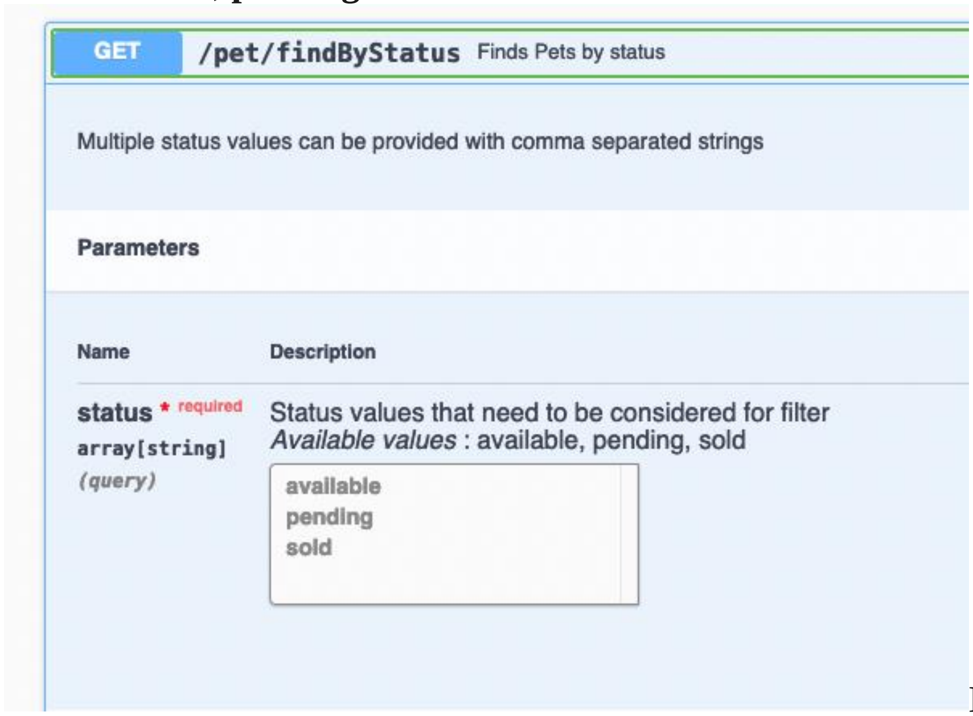
- Select the collection in **“Collections”** tab.
- In the collection’s view, select **“Variables”** tab.
- Populate “**Variable name**”, **“Initial value”** and **“Current value”** columns with respective values: *baseUrl* and *https://petstore.swagger.io/v2*.



Create collection variable

Create GET request and test for it

The first request we’re going to create in Postman is the GET request for finding pets by status. The endpoint for this request is */pet* and the request parameter is */findByStatus*. We can also observe from the documentation that this request requires a **status** query parameter and the values for the status that are accepted are: **available**, **pending** and **sold**.



GET /pet/findByStatus Finds Pets by status

Multiple status values can be provided with comma separated strings

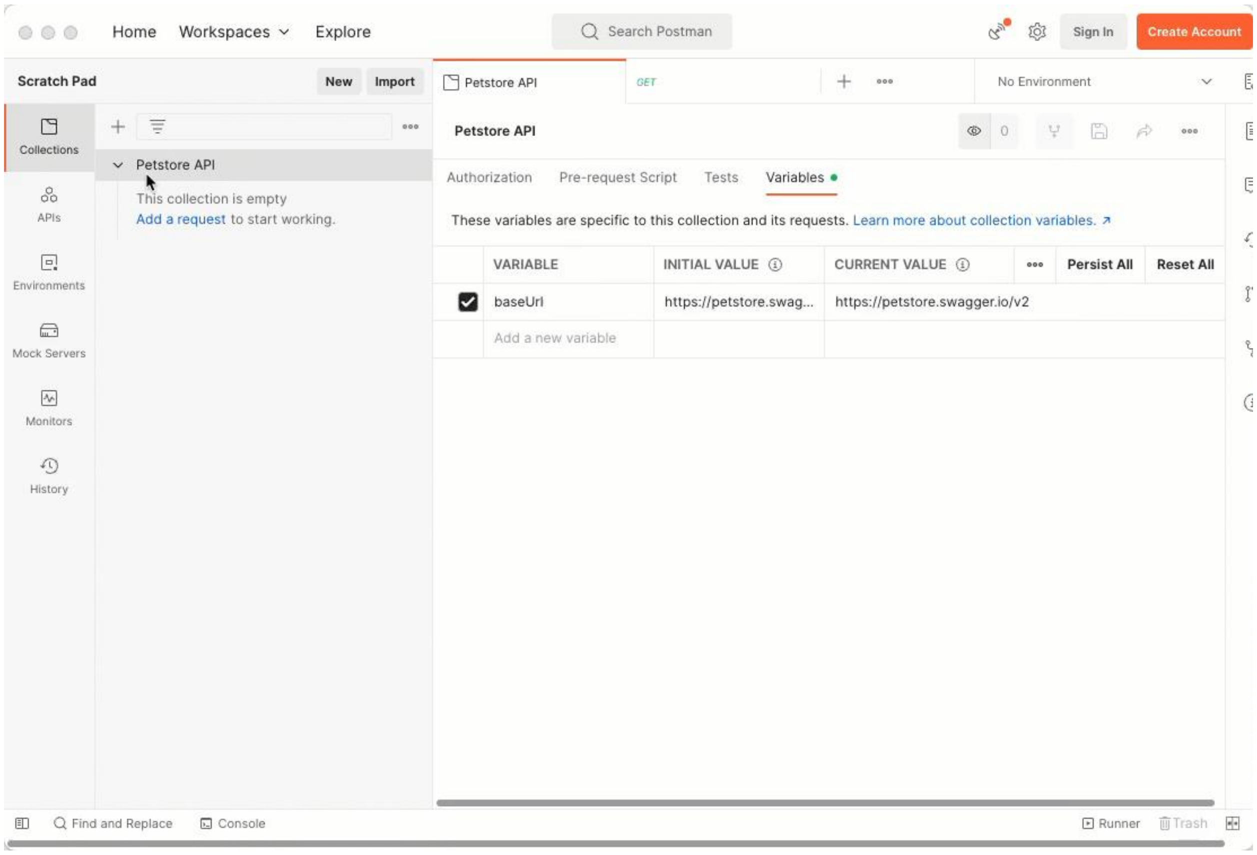
Parameters

Name	Description
status * required array[string] (query)	Status values that need to be considered for filter *Available values* : available, pending, sold available pending sold

Documentation of /pet/findByStatus API request

To add the request in Postman, follow these steps:

- Right click on the collection in the **"Collections"** tab.
- Choose the **"Add folder"** option. Name the folder **"pet"** because we will place all requests for */pet* endpoint in this folder.
- Right click on the **"pet"** folder and choose the **"Add request"** option.
- The request view will be displayed. We should populate all the required information for the request here. For now, we will set just a name for the request.

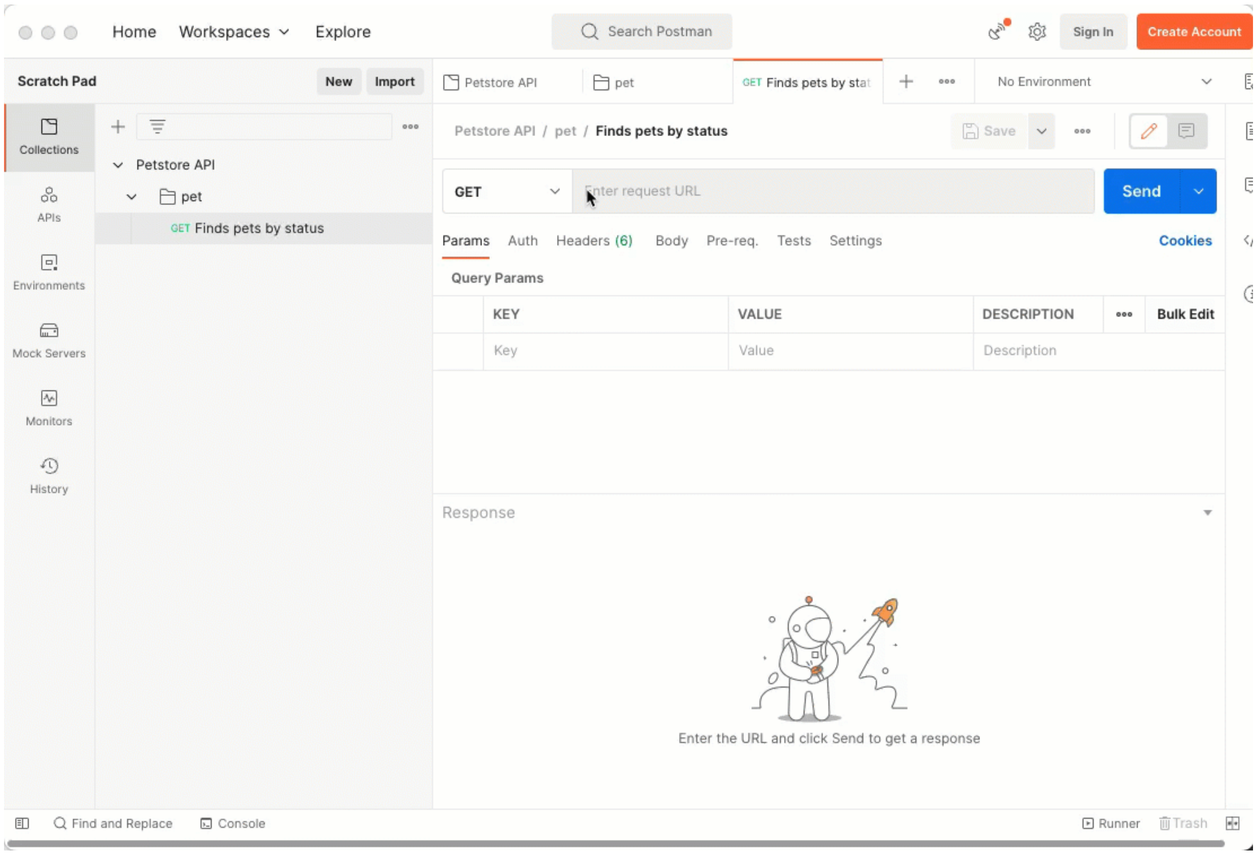


These variables are specific to this collection and its requests. Learn more about collection variables. ↗

	VARIABLE	INITIAL VALUE	CURRENT VALUE
✓	baseUrl	https://petstore.swag...	https://petstore.swagger.io/v2
	Add a new variable		

Adding a folder and request in Postman
GET is already set as the default option in the drop down for request method, therefore we don’t need to change it. We should enter the request URL, here we will insert *{{baseUrl}}/pet/findByStatus*. If you remember, we’ve already added *baseUrl* as a collection variable, so we can use its value by wrapping the variable name in curly brackets. This way, we don’t need to specify the full URL in every request. Under the **“Params”** tab, we can enter query parameters. For our request, we need to enter the **“status”** query parameter and any accepted status as value. We can use **“available”** as a status value for this example.

Now that we have entered all necessary data, we can hit the **“Send”** button to send the request and observe the response we receive from the server. The response is received in the **“Response”** section at the bottom of the view. The data is received in JSON format, and we can easily observe all the information of available pets, like name, photo URLs, categories, etc.



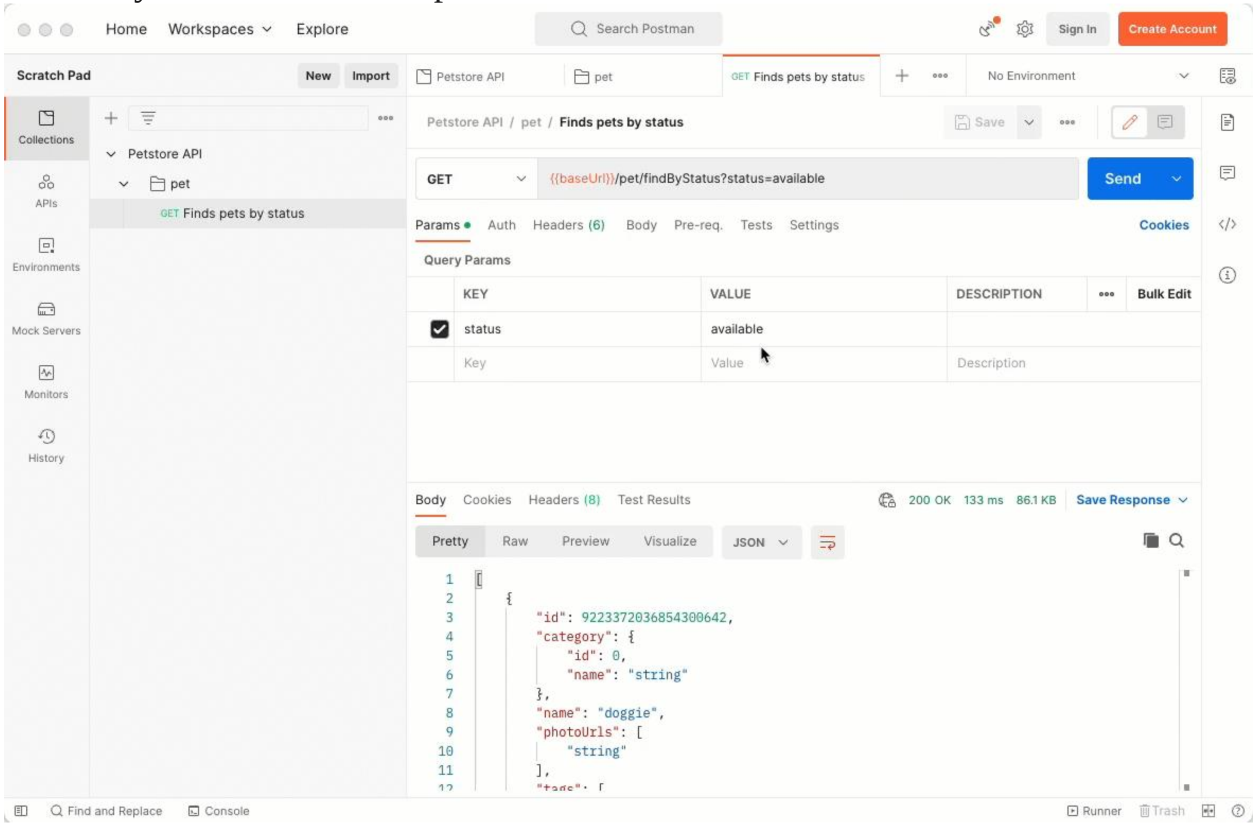
Sending GET request in Postman and observing the response

The next step is to add tests for the response we received. We want to test if we received a successful response (status code is 200) and if the data in the response body contains a pet with the name “doggie”. Tests can be added under the **“Tests”** tab. On the right side, next to the input field, we already have predefined JavaScript code snippets for most common test cases. Scroll down a bit, and look out for the snippets with names: **“Status code: Code is 200”** and **“Response body: JSON value check”**. These snippets can be used for our test cases. If you’re familiar with JavaScript, you can add more code here and test the response more thoroughly.

For our simple test we will modify the second code snippet, so that from the list of all available pets we will check if the first pet has the name “doggie”.

```
pm.test("First pet check name", function () {
  var jsonData = pm.response.json();
  pm.expect(jsonData[0].name).to.eql("doggie");
});
```

To run the tests, we have to send the request again. Test results will be available in the **“Response”** section under the **“Test Results”** tab. If you click on this tab, you can easily check which tests passed or if some tests failed.



GET {{baseUrl}}/pet/findByStatus?status=available

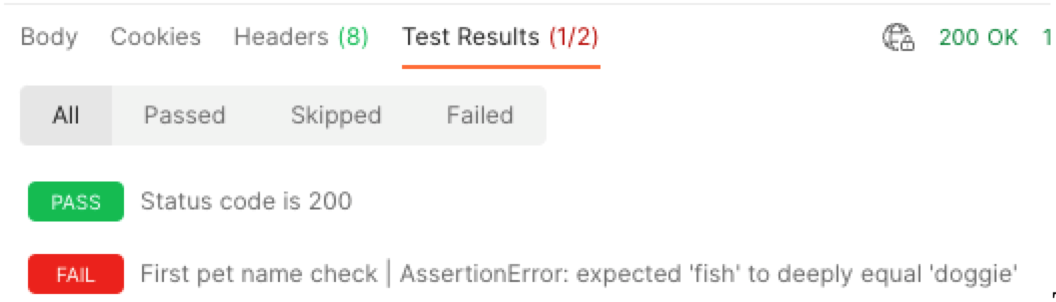
	KEY	VALUE	DESCRIPTION
✓	status	available	
	Key	Value	Description

200 OK 133 ms 86.1 KB

```
[
    {
        "id": 9223372036854300642,
        "category": {
            "id": 0,
            "name": "string"
        },
        "name": "doggie",
        "photoUrls": [
            "string"
        ],
        "tags": [
```

Adding and executing tests, and observing test results

Our second test case will fail sometimes because every time the request is sent, the API responds with available pets at that particular time. This means that “doggie” may not be available the next time we send the request and that’s why the test might fail. We should design our test cases to be better than the simple example above, they should always be consistent with the end use case we want to check and with the API state.

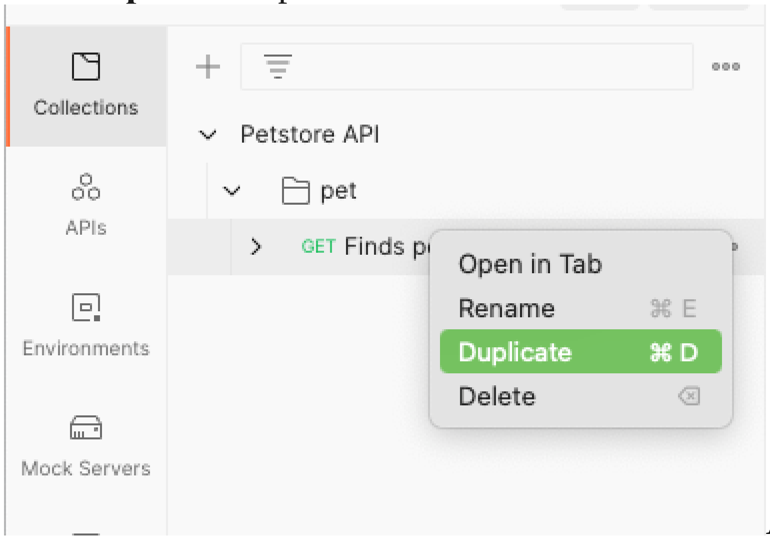


PASS Status code is 200

FAIL First pet name check | AssertionError: expected 'fish' to deeply equal 'doggie'

Test might fail if you run it again after some time has passed

We can also check and test what happens when we pass invalid status value. The documentation states that in this case we should receive the 400 response status code. We can add this example in our request, also we can test this expectation. We can duplicate our existing request by right clicking on it and choosing the **“Duplicate”** option.



Adding a duplicate request

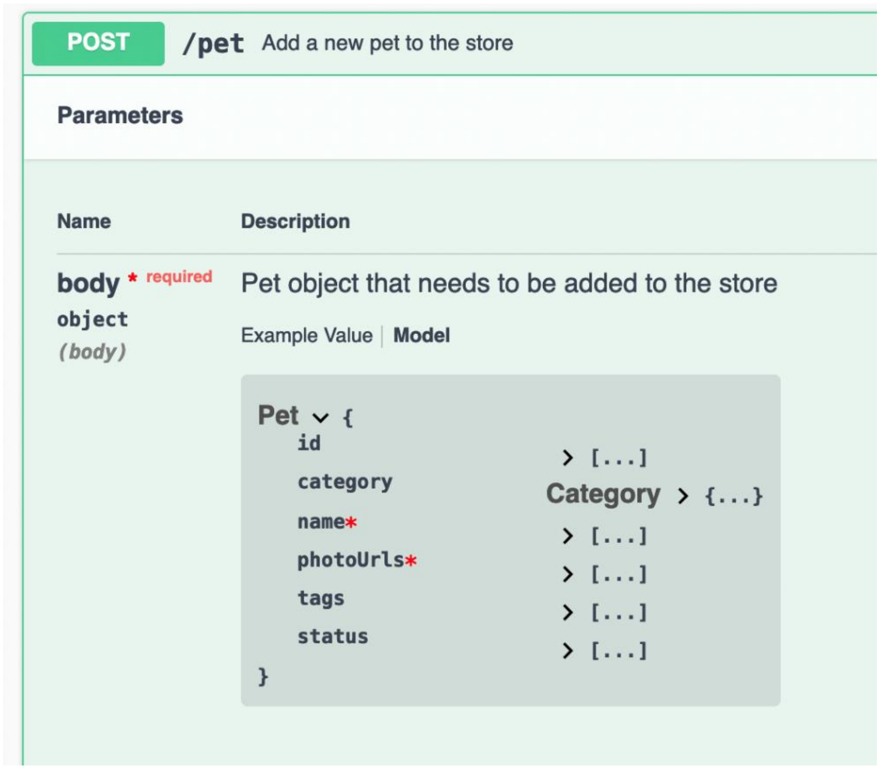
We set a new name for the request so that we distinguish that it is for an invalid status code. Also, we modify the value of the “status” query parameter and add an arbitrary value. In the tests, we remove the second test case because we don’t need it for this request, but we also modify the first test case to check if we receive the 400 response status code.

```
pm.test("Status code is 400", function () {
  pm.response.to.have.status(400);
});
```

If we send the request, the test should pass.

Create POST request and test for it

How about we add a new pet with the name Dogo to the pet store? For this we can use the POST request, which is the only request that changes the server by adding a new object. From the documentation, we can see that the endpoint for adding a new pet is just **/pet** without any request parameter. A request body is mandatory, as is with every POST request. We can see from the documentation that the body should be given in JSON format and the required data that should be provided is pet’s **name** and **photoUrls**.


POST /pet Add a new pet to the store

Parameters

Name	Description
body * required object (body)	Pet object that needs to be added to the store

Example Value | Model

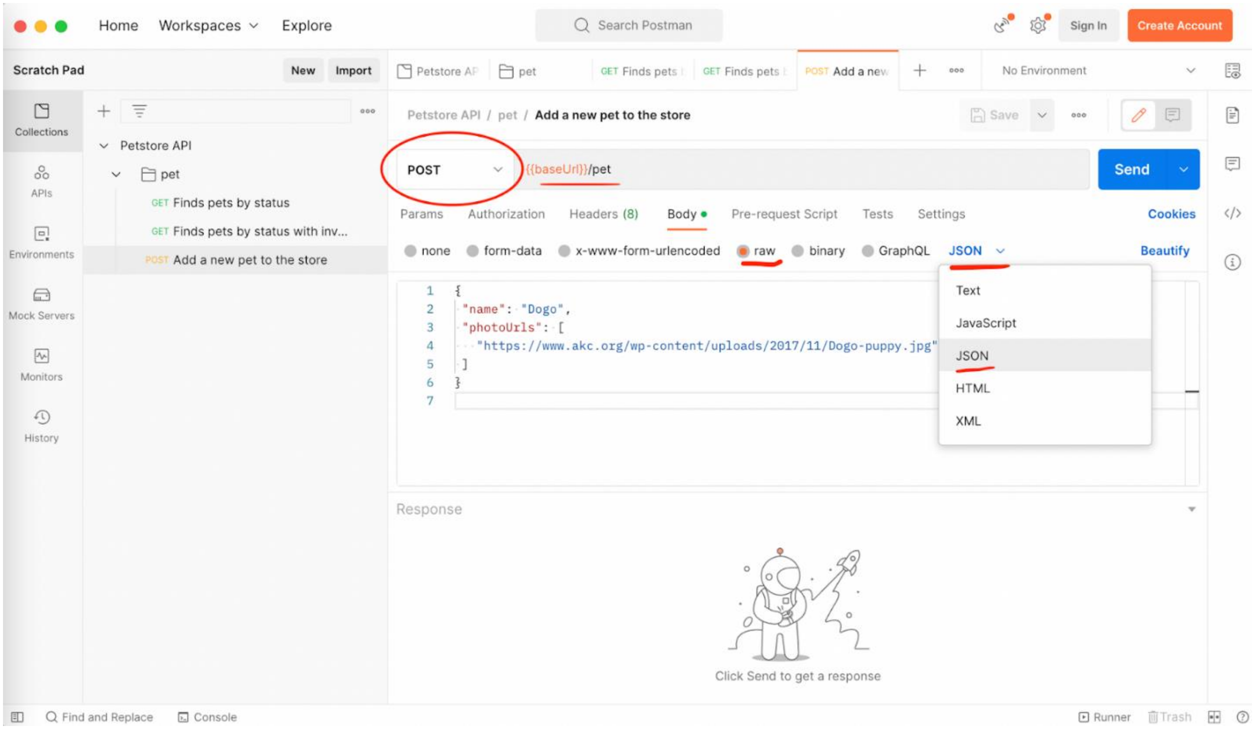
```
Pet ∨ {
   id                  > [...]
   category            Category > {...}
   name*               > [...]
   photoUrls*          > [...]
   tags                > [...]
   status              > [...]
}
```

Documentation of POST /pet API request

Similarly, as we did with the GET request, we will add the POST request under the **“pet” folder**:

- Select **“POST”** from the drop down menu.
- Enter *{{baseUrl}}/pet* in the request URL input field.
- Select the **“Body”** tab below the request URL input field to add the JSON request body.
- Choose the **“raw”** option and select **JSON** from the last drop down view in the row.
- In the input field below the body format options, we should enter the following body:

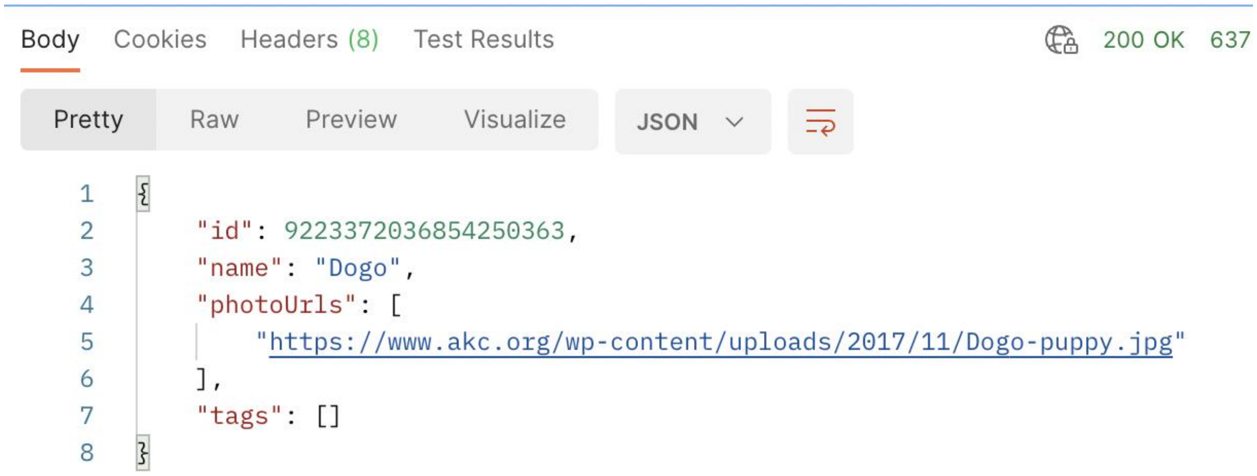
```
{
 "name": "Dogo",
 "photoUrls": [
   "https://www.akc.org/wp-content/uploads/2017/11/Dogo-puppy.jpg"
 ]
}
```



```
{
"name": "Dogo",
"photoUrls": [
  "https://www.akc.org/wp-content/uploads/2017/11/Dogo-puppy.jpg"
]
}
```

Adding POST request in Postman

If we execute this request by pressing the **"Send"** button, we should receive a successful response. The response should contain data we already provided with additional information, like ID.



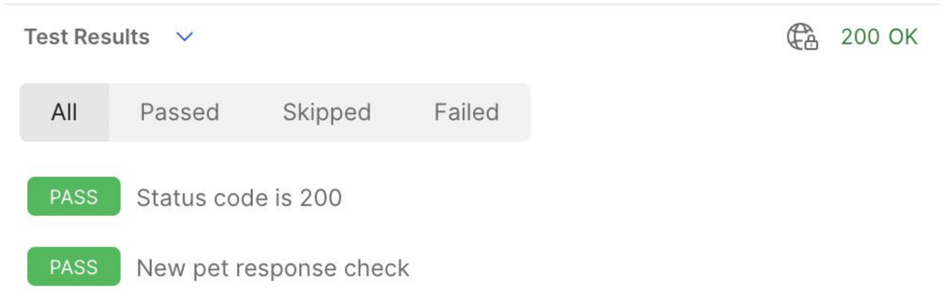
```
{
    "id": 9223372036854250363,
    "name": "Dogo",
    "photoUrls": [
        "https://www.akc.org/wp-content/uploads/2017/11/Dogo-puppy.jpg"
    ],
    "tags": []
}
```

The response from the request
It's fairly simple to construct a test for this response, we want to make sure that the added pet is indeed with the name Dogo and with the correct photo URL. Similarly as with the GET request, we can use already existing snippets for checking the

response status code and values from the JSON response. Our tests should look like this:

```
pm.test("Status code is 200", function () {
  pm.response.to.have.status(200);
});
pm.test("New pet response check", function () {
  var jsonData = pm.response.json();
  pm.expect(jsonData.name).to.eql("Dogo");
pm.expect(jsonData.photoUrls[0]).to.eql("https://www.akc.org/wp-content/uploads/2017/11/Dogo-puppy.jpg");
});
```

If we run the tests, they should pass.



PASS Status code is 200

PASS New pet response check

Test results for adding new pet request

We can also play around with the request body and try to send invalid data or data without mandatory values. It's recommended to construct tests with expected response codes, trying to cover as many negative and positive scenarios as possible.

Create PUT request and test for it

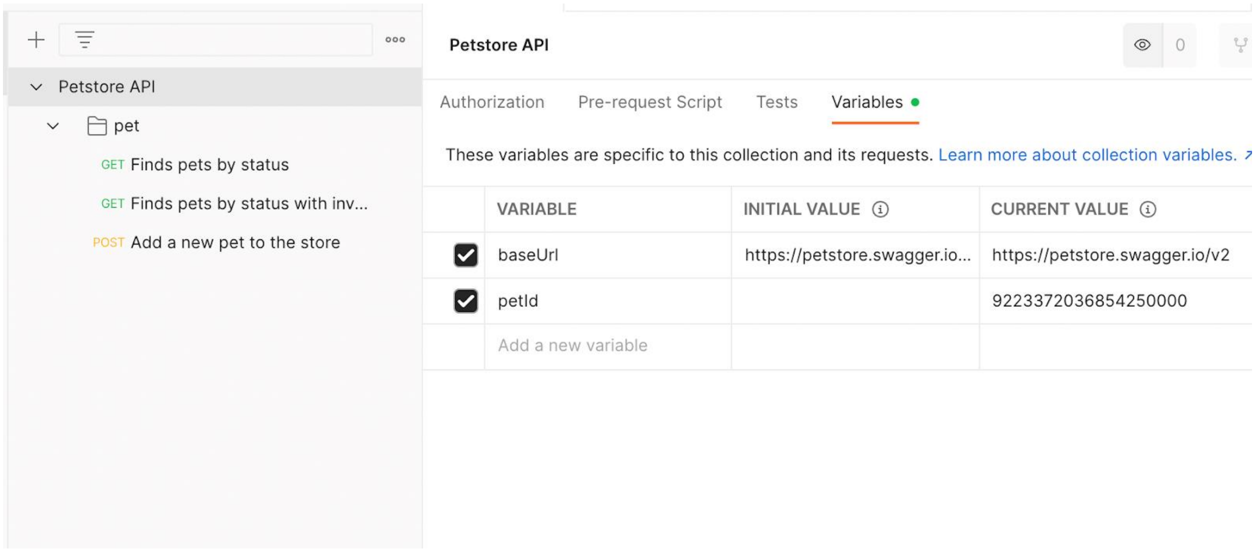
What if you want to change the name of your pet and update the status to sold? We know that the PUT request can modify the server so that an existing object can be updated with new data. But from all the pets in the store, how will the server know which one is your pet? If you observe the response from above closely, where we added the pet to the store with the POST request, you'll notice that there is an ID in the data. This ID is the identification of the pet whose data we want to update. We can use this ID in our request for updating the name of our pet and the status. However, it's not optimal to copy and paste the ID. We want this ID to be stored as a variable and then used in our update request. Luckily, Postman allows us to store data from responses as variables too, just like we stored the base URL in collection variables. Let’s store the ID of the previously created pet. In Postman, follow these steps:

- Open the POST request you created previously.
- Go to the **“Tests”** tab.
- From the snippets on the right, choose the one with the name **“Set a collection variable”**.
- The snippet should be copied at the end of the test. Move the snippet in **“New pet response check”** test and modify it like so:

```
pm.collectionVariables.set("petId", jsonData.id);
```

- Run the tests again. This is important so that with the new code snippet, the ID can be saved to collection variables.

You can check if your ID is stored by clicking on the collection, then selecting the **“Variables”** tab. The *petId* variable should be listed below the *baseUrl* variable.



These variables are specific to this collection and its requests. Learn more about collection variables.

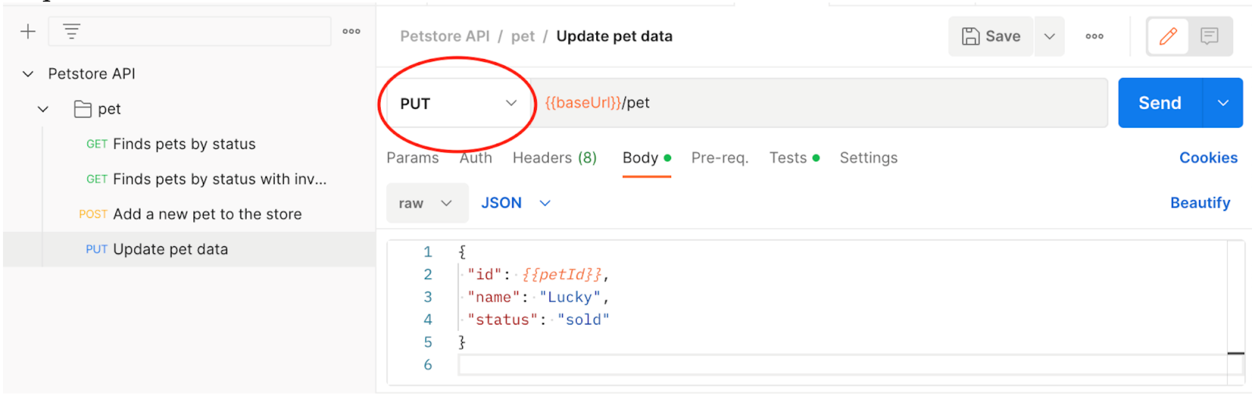
	VARIABLE	INITIAL VALUE	CURRENT VALUE
✓	baseUrl	https://petstore.swagger.io...	https://petstore.swagger.io/v2
✓	petId		9223372036854250000
	Add a new variable		

petId saved as collection variable after test execution of adding new pet

Now that we have the identification of our pet we can update the name and status with the PUT request. From the documentation, we can see that everything is the same as with the POST request, the only difference is in the request method. Repeat the same steps as with the POST request, but make sure to select **“PUT”** from the drop down menu for the request method and send the following request body:

```
{
"id": {{petId}},
"name": "Lucky",
"status": "sold"
}
```

We wrapped our saved pet ID with curly brackets in the request body. The created request in Postman should like this:

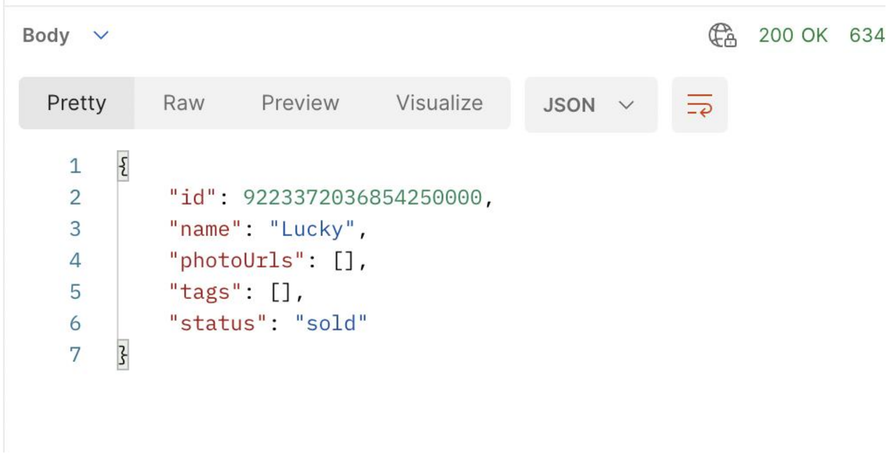

Petstore API
pet
GET Finds pets by status
GET Finds pets by status with inv...
POST Add a new pet to the store
PUT Update pet data
Petstore API / pet / Update pet data
Save
PUT
Send
Params Auth Headers (8) Body Pre-req. Tests Settings
Cookies
raw JSON
Beautify

{{baseUrl}}/pet

```
{
 "id": {{petId}},
 "name": "Lucky",
 "status": "sold"
}
```

PUT request to update the data of our pet

Run the request and observe the response. New data should be visible in the response.



```
{
    "id": 9223372036854250000,
    "name": "Lucky",
    "photoUrls": [],
    "tags": [],
    "status": "sold"
}
```

The response of the PUT request

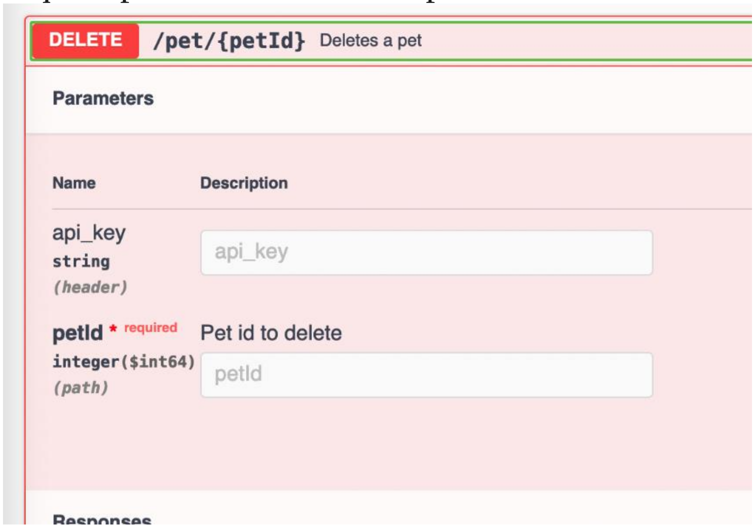
We can add similar tests as with the POST request to check if the pet’s name and status are updated. The tests should look like this:

```
pm.test("Status code is 200", function () {
  pm.response.to.have.status(200);
});
pm.test("Pet update check", function () {
  var jsonData = pm.response.json();
  pm.expect(jsonData.name).to.eql("Lucky");
  pm.expect(jsonData.status).to.eql("sold");
});
```

When we run the tests, they should pass.

Create DELETE request and test for it

Now, let’s imagine that the pet store has sold the dog Lucky and he is now happily living with me. I can safely delete the data of my pet from the server by using the DELETE request. The documentation states that the pet ID should be provided as a request parameter to the endpoint.



DELETE `/pet/{petId}` Deletes a pet

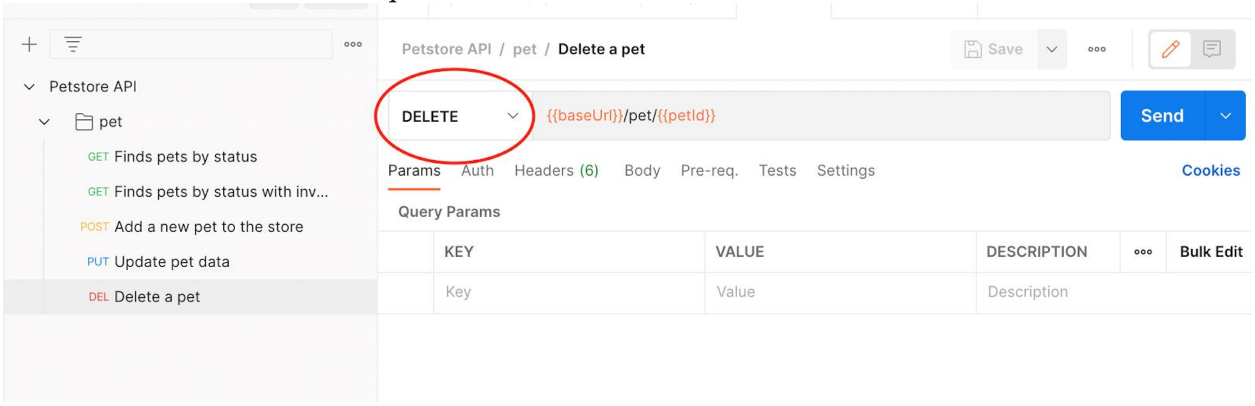
Parameters

Name	Description
api_key string (header)	api_key
petId * required integer($int64) (path)	Pet id to delete petId

Responses

Documentation of DELETE /pet API request

We already have the ID stored in the collection variables, therefore we can use it for the DELETE request as well by wrapping it in curly brackets in the request URL. Here is what the request in Postman should look like:



Petstore API / pet / Delete a pet

Petstore API
- pet
  - GET Finds pets by status
  - GET Finds pets by status with inv...
  - POST Add a new pet to the store
  - PUT Update pet data
  - DEL Delete a pet

DELETE `{{baseUrl}}/pet/{{petId}}` Send

Params Auth Headers (6) Body Pre-req. Tests Settings Cookies

Query Params

KEY	VALUE	DESCRIPTION
Key	Value	Description

DELETE request to delete the data of a pet

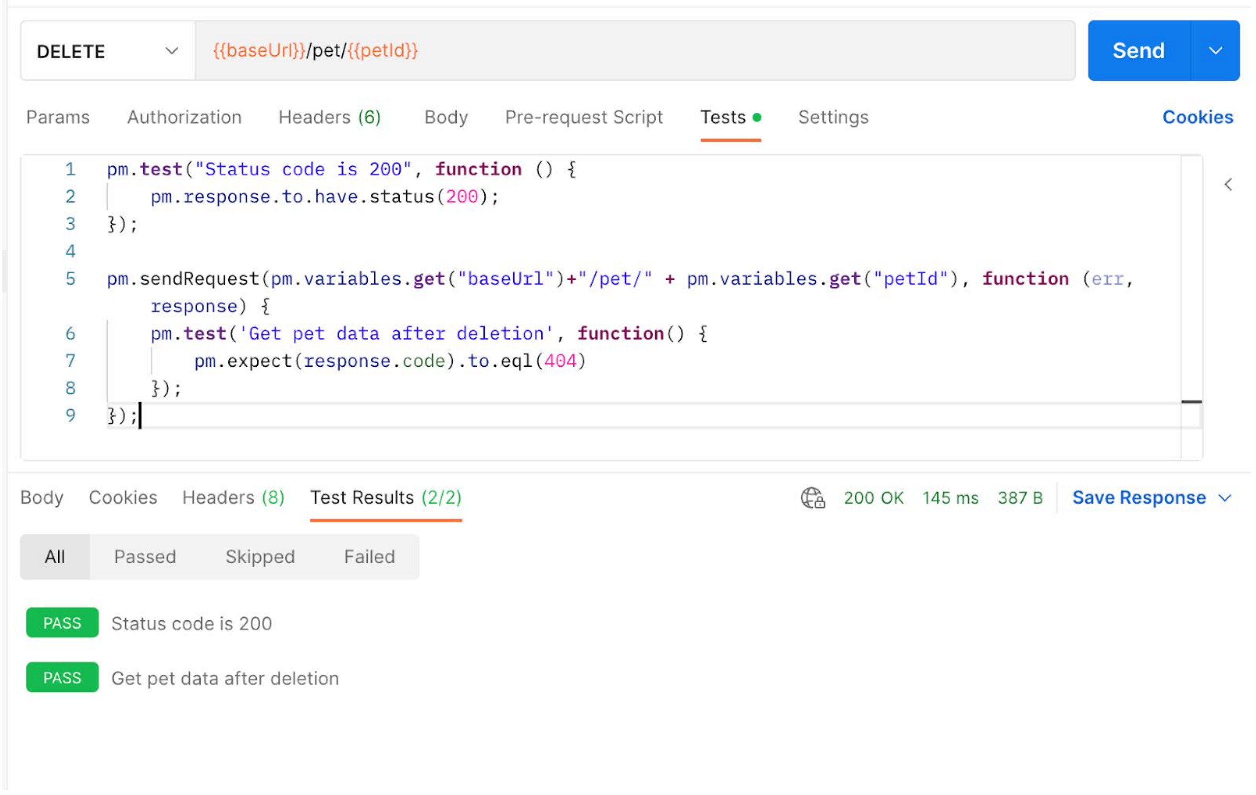
Before executing this request with the “Send” button, we should add tests first. This is because the DELETE request tests are more informative than the response body we receive. We can add one test to check if we receive the successful response status code 200. However, we can also check if the pet is actually deleted by using the GET request with the */pet/{petId}* endpoint, which gives us information about a given pet with a specific ID.

For this request we should receive a 404 (not found) response status code because we previously deleted the pet with the stored ID and it should no longer exist. We can check this by sending the request in code via an already existing code snippet. This code snippet can be found under **“Send a request”** name. In the test, for this request, we expect that the response has a 404 (not found) status code, so it should look like this:

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


pm.sendRequest(pm.variables.get("baseUrl")+"/pet/" + pm.variables.get("petId"),
function (err, response) {
  pm.test('Get pet data after deletion', function() {
     pm.expect(response.code).to.eql(404)
  });
});
```

Now we can execute the request and observe test results. Tests should have passed.



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

pm.sendRequest(pm.variables.get("baseUrl")+"/pet/" + pm.variables.get("petId"), function (err,
    response) {
    pm.test('Get pet data after deletion', function() {
        pm.expect(response.code).to.eql(404)
    });
});
```

PASS Status code is 200

PASS Get pet data after deletion

Lab Task 1: Testing a GET Request with Query Parameters

Objective:

Students will test the `/pet/findByStatus` GET endpoint using Postman and validate the response using tests.

Instructions:

1. Open **Postman**.
2. Import or create a new **Collection** and set a **collection variable**:
   - Name: `baseUrl`
   - Value: `https://petstore.swagger.io/v2`
3. Create a **folder** named `pet` inside the collection.
4. Inside the `pet` folder, create a **GET request**:
   - URL: `{{baseUrl}}/pet/findByStatus`
   - Under the **Params** tab, add:
     - `status` = `available`
5. Click **Send** and observe the JSON response.
6. Under the **Tests** tab, add the following two test cases:

javascript

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

pm.test("First pet name is 'doggie'", function () {
   var jsonData = pm.response.json();
   pm.expect(jsonData[0].name).to.eql("doggie");
});
```

✓ Submission Requirements:

- Screenshot of the request with status and query parameter.
- Screenshot of test results (pass/fail).
- Comment on why the test might fail even if the request is correct.

Lab Task 2: Adding a New Pet Using POST Request

Objective:

Students will use the POST method to add a new pet to the store and write tests to confirm the request was successful.

Instructions:

1. In the same `pet` folder, create a **new POST request**.
    - URL: `{{baseUrl}}/pet`
    - Method: `POST`
2. Under the **Body** tab:
    - Select `raw` and choose `JSON` format.
    - Use the following JSON body:

```
json
{
  "name": "Dogo",
  "photoUrls": ["https://www.akc.org/wp-content/uploads/2017/11/Dogo-
puppy.jpg"]
}
```

3. Add the following tests:

```
javascript
pm.test("Status code is 200", function () {
   pm.response.to.have.status(200);
});

pm.test("New pet response check", function () {
   var jsonData = pm.response.json();
   pm.expect(jsonData.name).to.eql("Dogo");
   pm.expect(jsonData.photoUrls[0]).to.eql("https://www.akc.org/wp-
content/uploads/2017/11/Dogo-puppy.jpg");
});
```

Submission Requirements:

- Screenshot of the request body.
- Screenshot of the test results (pass/fail).
- Comment on how you would improve or extend this test.