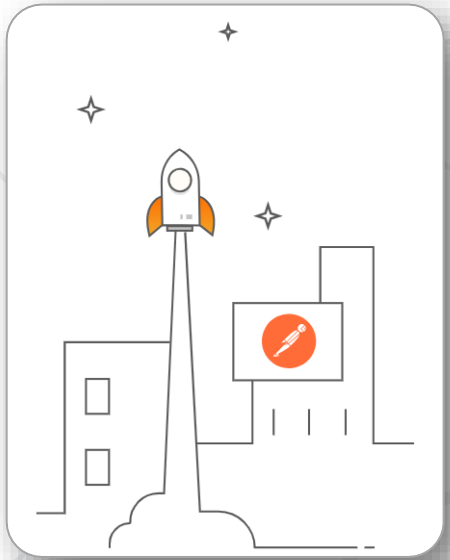
API Testing with Postman

HTTP Requests and Automated API Tests



SoftUni Team

Technical Trainers

Software University

SoftUni

Software University

https://softuni.bg

You Have Questions?

Software University

sli.do

#QA-BackEnd

Table of Contents

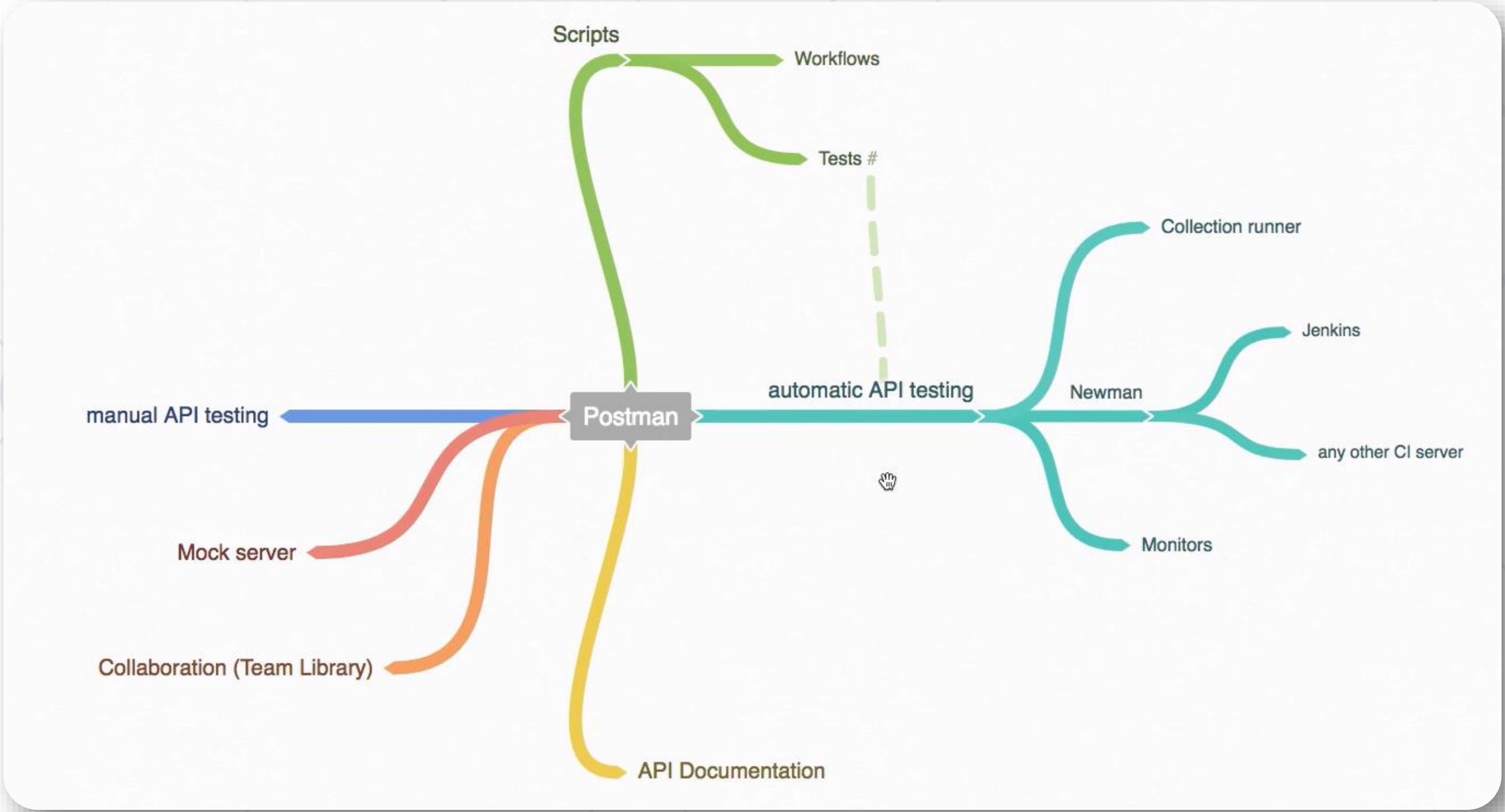
Postman Recap

Usage and Capabilities

Postman Capabilities

- Popular tool used by developers and testers for **API testing** and **development**
- It allows for the **sending HTTP requests** to web servers and the **inspection** of the **responses**
- Used for **various tasks**:
  - **API testing**
  - **Building and consuming APIs**
  - **Automating tests**
  - **Creating documentation** for API services

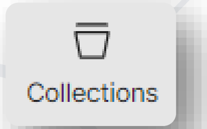
Postman Pipeline



Key Terms

Request, Collection, Variable, Environment, Parameter

Requests and Collections

- **Requests:**
GET POST PUT PATCH DELETE
  - A single HTTP request to an API
  - It can include the HTTP method (**GET, POST, PUT, DELETE,** etc.), the URL, headers, query parameters, and the request body
  - Requests can be shared to collections for reuse and organization
- **Collections:** 

  - A collection is a **group of API requests**
  - Used to **organize API requests** into folders and subfolders, making it easier to manage and share API calls

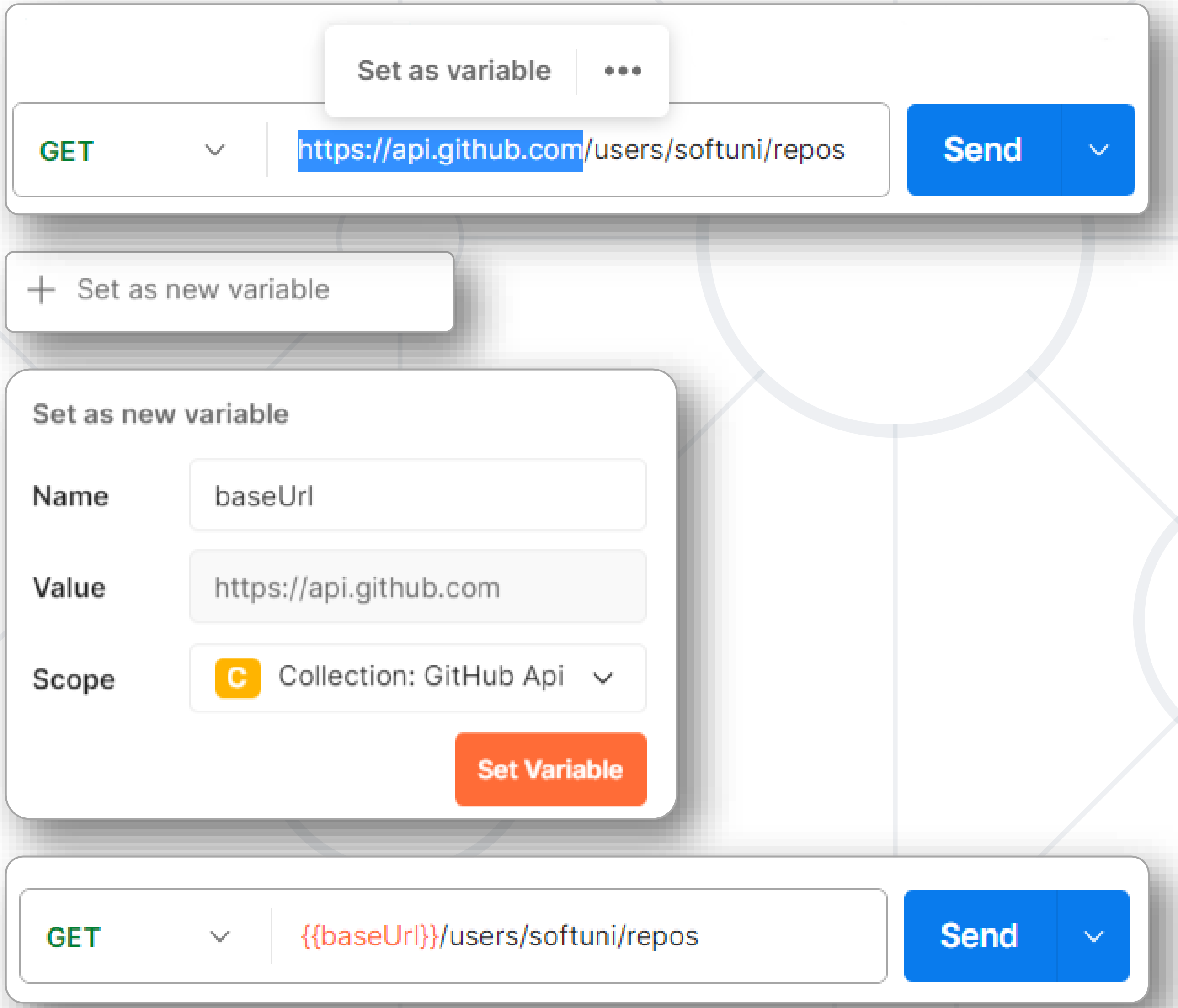
Variables

- Named **placeholders for values**
- Representation of data that enables to **access a value** without having to enter it manually
- Useful for using the **same values** in **multiple places**
- Make **requests** more **flexible** and **readable**, by abstracting the detail away
- Can be defined at **various scopes**:
  - **Global** (accessible in any request)
  - **Collection** (accessible within a specific collection)
  - **Environment** (accessible within a specific environment)

Variables Example

- Same URL in more than one request, but the URL might change later
- Store the URL in a variable "**base_url**" and reference it the requests using **{{base_url}}**
- If the URL changes, change the variable value and it will be reflected throughout the entire collection, wherever the variable name is used

Storing Configuration in Collection Variables



https://api.github.com/users/softuni/repos

Set as new variable

Name	baseUrl
Value	https://api.github.com
Scope	C Collection: GitHub Api

Set Variable

{{baseUrl}}/users/softuni/repos

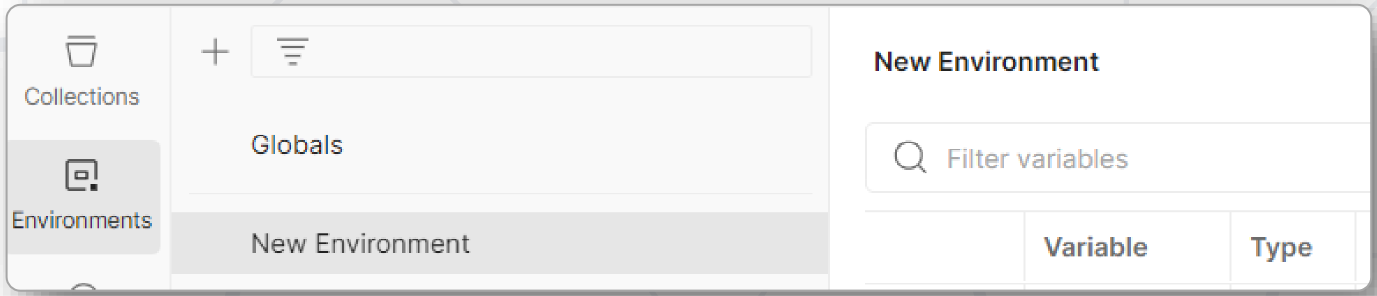
- Highlight the part of the URL
- Click on "Set as variable"
- Choose "Set as new variable"
- Add appropriate name
- Select a scope (collection)
- Click "Set Variable"

Environments

- Allows to **customize requests** to run in **different contexts** without needing to change the actual requests themselves
- By separating requests into different environments, test run safely against a development server, a staging server, and a production server without the risk of accidentally modifying live data
- Achieved by using variables to **represent parts of the requests** that may change between these contexts, such as URLs, credentials, or other parameters

Setting Up and Using Environments

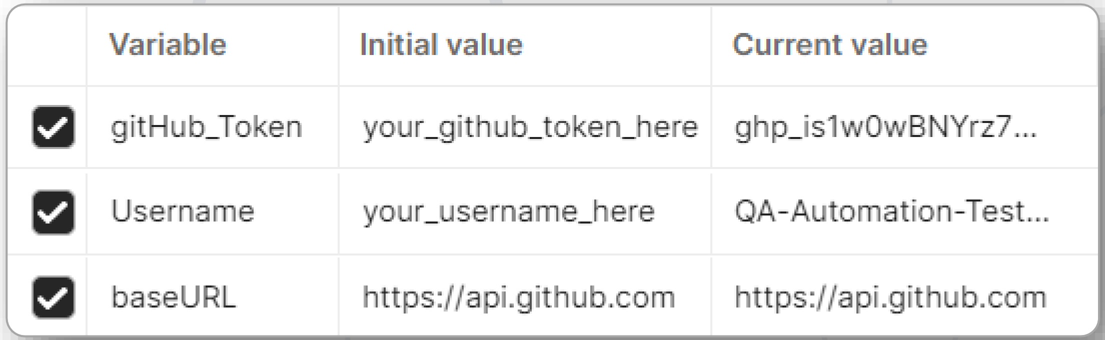
- Environment is created by going to the "Environments" section and adding a **"New Environment"**



- For each environment, **define the needed variables**, such as baseUrl, apiKey, etc. that vary between different setups
- In the requests, similar to the collection variables, use the syntax **{{variableName}}**
- When switching environments, **all variables** in the **requests** are **automatically replaced** with the **values** from the **active environment**

Variables' Initial Value vs. Current Value

- Environment and collection variables can be defined with an Initial Value and a Current Value, serving different purposes:
  - **Initial Value** (**Shared**) - The default value that gets shared when you export your environment or collection
  - **Current Value** (**Local and Private**) - the actual value that Postman uses when executing requests in your local instance. This value is not included when you share or export your environment or collection



	Variable	Initial value	Current value
✓	gitHub_Token	your_github_token_here	ghp_is1w0wBNYrz7...
✓	Username	your_username_here	QA-Automation-Test...
✓	baseURL	https://api.github.com	https://api.github.com

Query Parameters

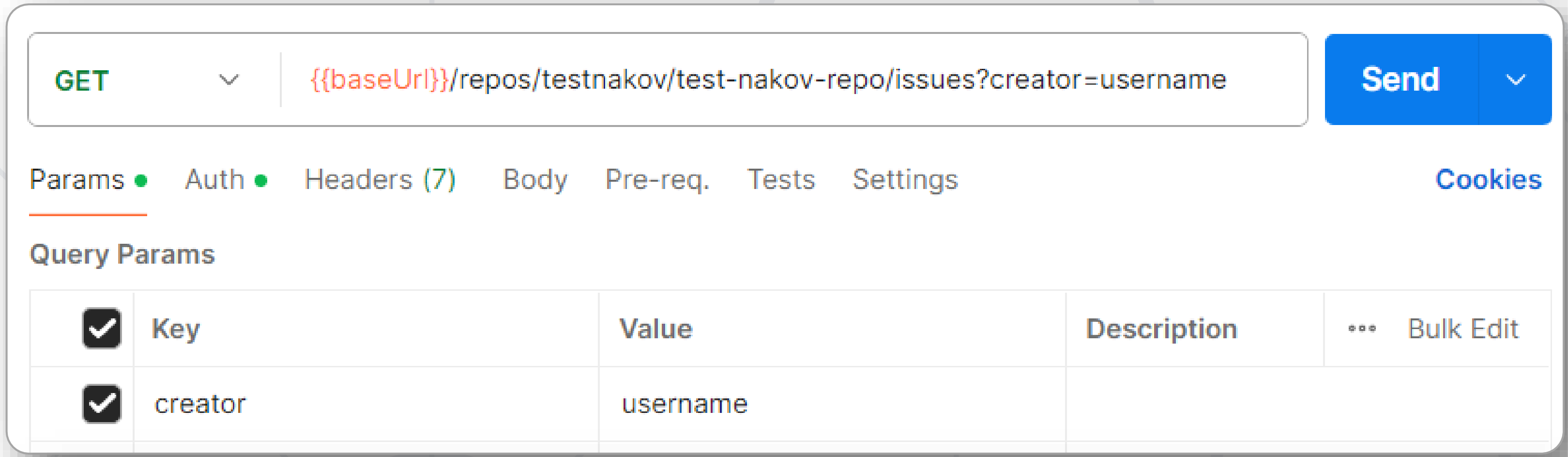
- A key-value pair that is appended to the URL of an HTTP GET request
- They are part of the query string, which is the section of the URL following the ? character
- For example, in the URL:
  https://api.example.com/items?category=book
  The query string is category=books with "**category**" being the **query parameter**
- Each **query parameter** consists of a **key** (category) and a corresponding **value** (book)

Query Parameters Example

- The purpose is to **modify** the **data returned** by the **API call** by:
  - **Filtering**: To return a subset of data based on certain criteria. For instance, **?status=active** could be used to only return active items from an API
  - **Sorting**: To define how the returned data should be ordered. For example, **?sort=price** could be used to sort items by price
  - **Searching**: To perform a search for specific data. For example, **?search=keyword** could be used to search for items containing "**keyword**"
  - **Pagination**: To control the page number and size of the dataset returned. Commonly used parameters include **?page=2** and **?limit=20**
  - **Field Selection**: To specify which fields should be included in the response. For example, **?fields=id,name,price**

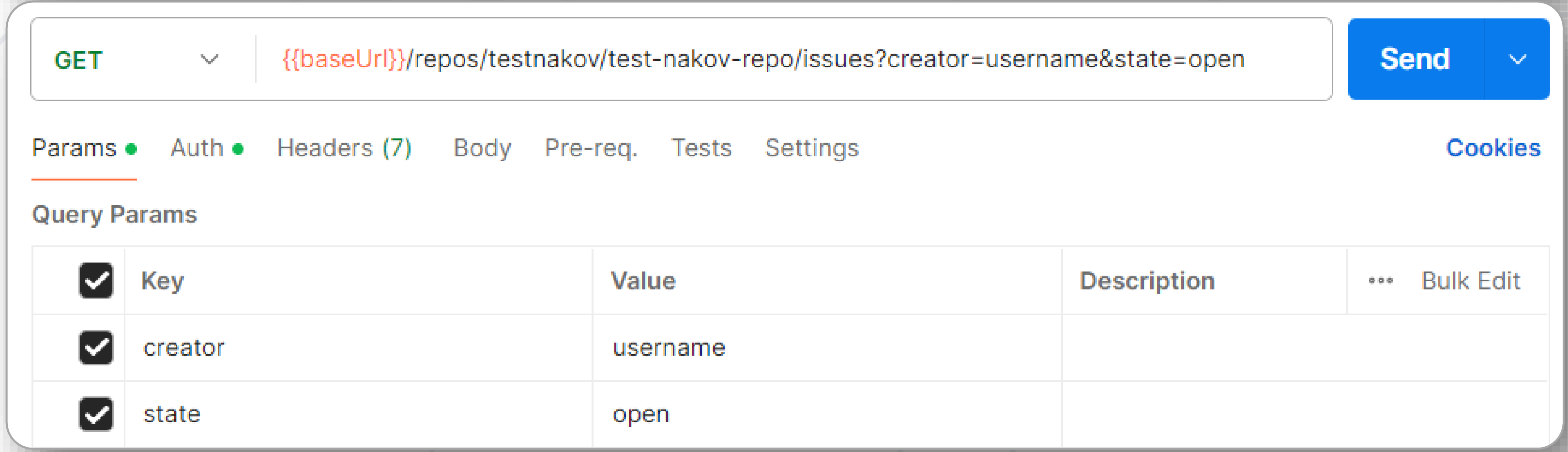
Adding Query Parameters to a Request

- In the request URL field, you can directly append the query parameter by adding a ? followed by the key-value pair



```
GET {{baseUrl}}/repos/testnakov/test-nakov-repo/issues?creator=username
```

	Key	Value	Description
✓	creator	username	



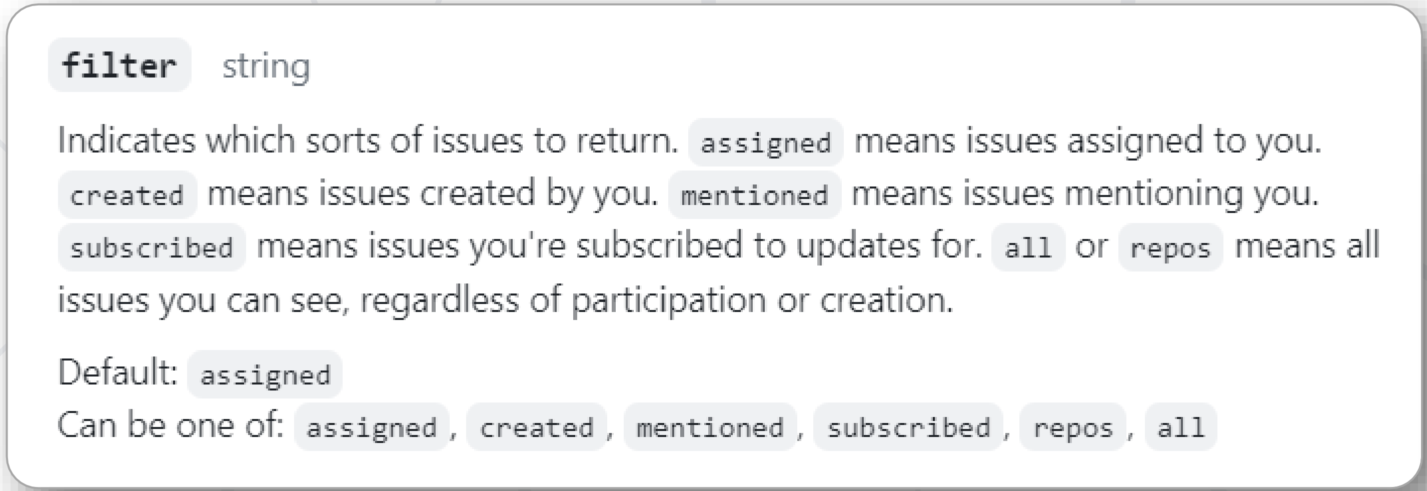
```
GET {{baseUrl}}/repos/testnakov/test-nakov-repo/issues?creator=username&state=open
```

	Key	Value	Description
✓	creator	username	
✓	state	open	

To append multiple query parameters use & symbol

GitHub API Documentation

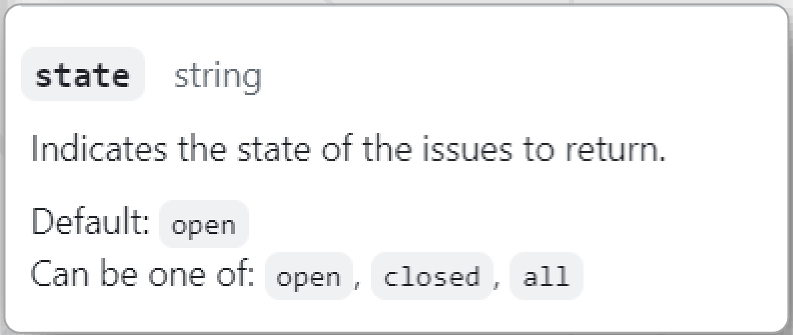
- If you're curious about how we know to use the **creator** and **state parameters** in our API request, this information comes directly from the GitHub API documentation



`filter` string

Indicates which sorts of issues to return. `assigned` means issues assigned to you. `created` means issues created by you. `mentioned` means issues mentioning you. `subscribed` means issues you're subscribed to updates for. `all` or `repos` means all issues you can see, regardless of participation or creation.

Default: `assigned`
Can be one of: `assigned`, `created`, `mentioned`, `subscribed`, `repos`, `all`



`state` string

Indicates the state of the issues to return.

Default: `open`
Can be one of: `open`, `closed`, `all`

- The **documentation** is an **invaluable resource** that provides detailed descriptions of all the parameters used to filter and access the data needed. **Learn to read the documentation!**

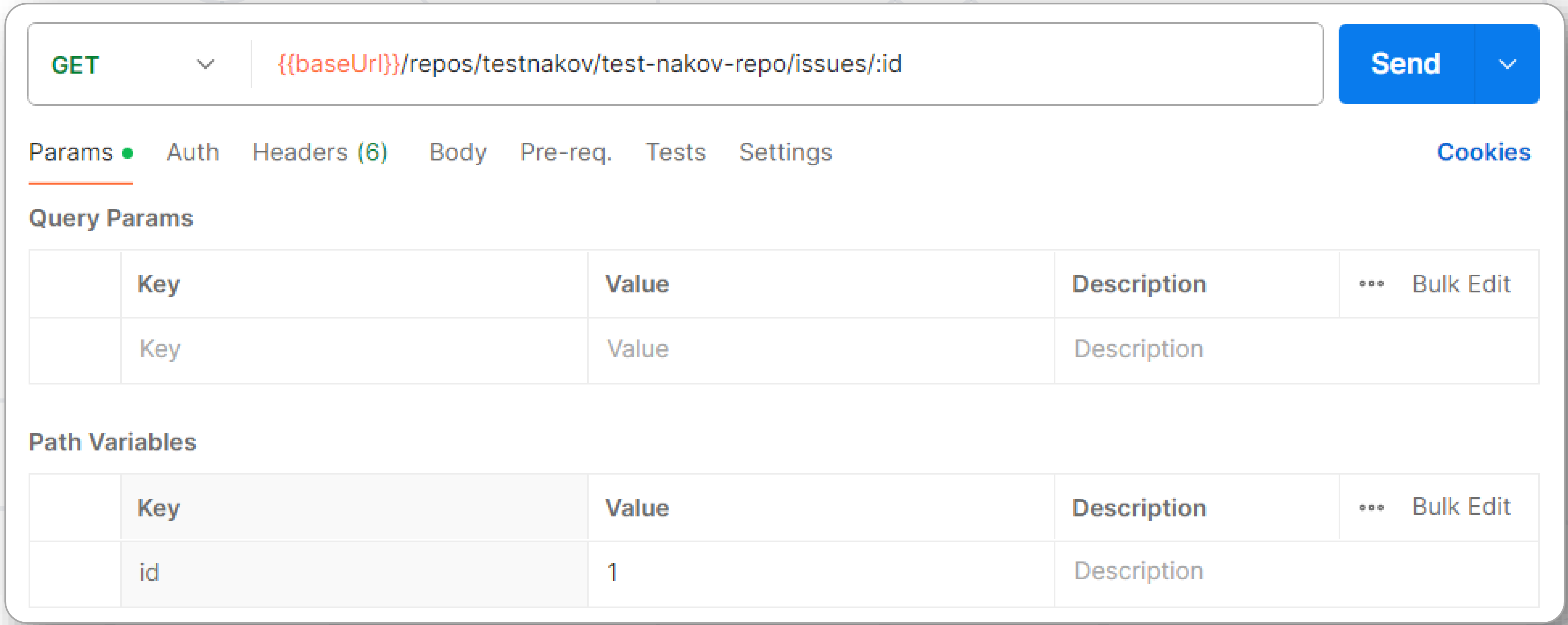
Importance of Documentation

- **Each API** is **unique**
- Documentation provides information of what the **API can do**
- Explains the **correct syntax for requests**, (incl.: the base URL, endpoints, required headers, query parameters, and the expected structure of request and response bodies)
- Outlines the **necessary steps** for **authenticating requests**
- Includes information on how to **handle different response codes**
- Many API documents provide **example requests and responses**
- Provides a **change log** or **release notes** detailing updates, deprecations, and any other modifications

Path Variables

- **Dynamic segments** in the **URL path** that are meant to be replaced with actual values when making a request
- **Placeholders** for parts of the URL that will change, like an ID that specifies a particular resource
- In Postman, a path variable is denoted by a **colon :** preceding the variable name when you define it in the URL
- For example, if you have a URL: **https://api.example.com/items/:itemId**, the **:itemId** is a placeholder for a path variable that you expect to replace with a real value when making the request

Adding Path Variable to a Request


GET {{baseUrl}}/repos/testnakov/test-nakov-repo/issues/:id

Query Params

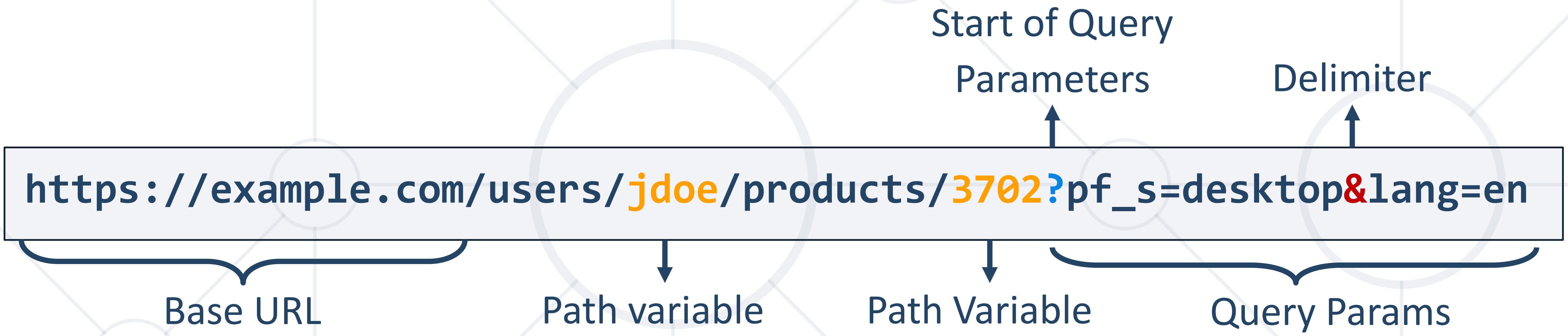
	Key	Value	Description	••• Bulk Edit
	Key	Value	Description	

Path Variables

	Key	Value	Description	••• Bulk Edit
	id	1	Description	

- **:id** in the URL field is set up to be **replaced by the value** entered in the "Path Variables" section of the Params tab
- Postman **will replace :id** with that value in the actual request

Query Parameters vs Path Variables



```
https://example.com/users/jdoe/products/3702?pf_s=desktop&lang=en
```

Path variables	Query parameters
Only value	Key-value pair
Mandatory	Mandatory or optional
Part of the endpoint / path	Start after the question mark

Postman's Scripting API

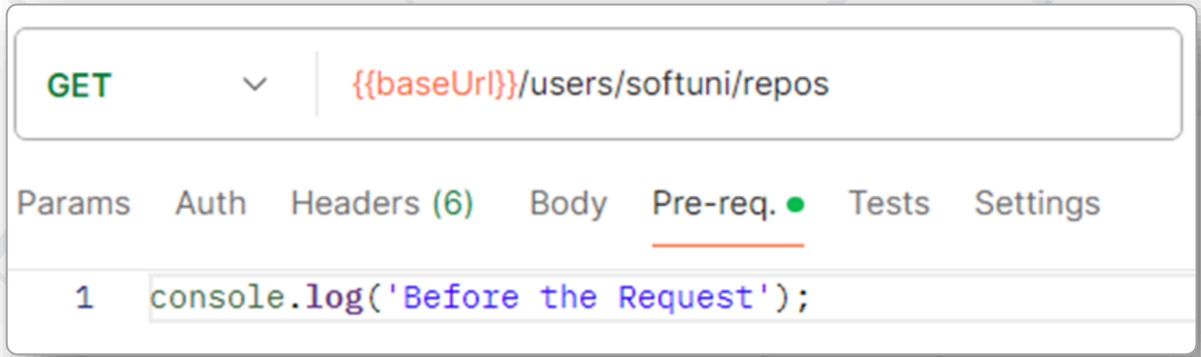
Core Components

Postman and JS

- Postman's scripting capabilities are powered **exclusively** by **JavaScript**
- All the scripting, whether it's for **writing tests**, **pre-request scripts**, or **data processing**, will be in **JavaScript**

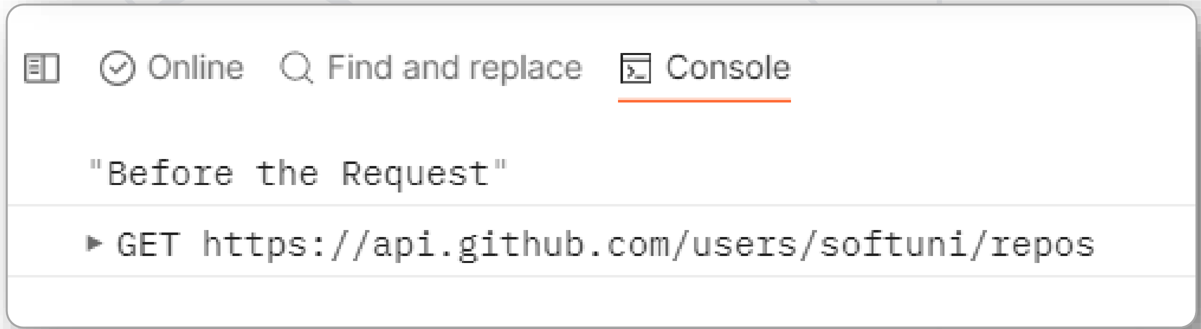
Pre-Request

- **Runs before** an actual **API request**
- To set up **certain aspects** of requests **dynamically**
- Various **purposes**:
  - Set up **environment variables**
  - Create **dynamic parameters**
  - Add **timestamps** / **tokens** to headers
  - Perform **calculations** or **logic** that needs to be included in the request



```
{{baseUrl}}/users/softuni/repos
```

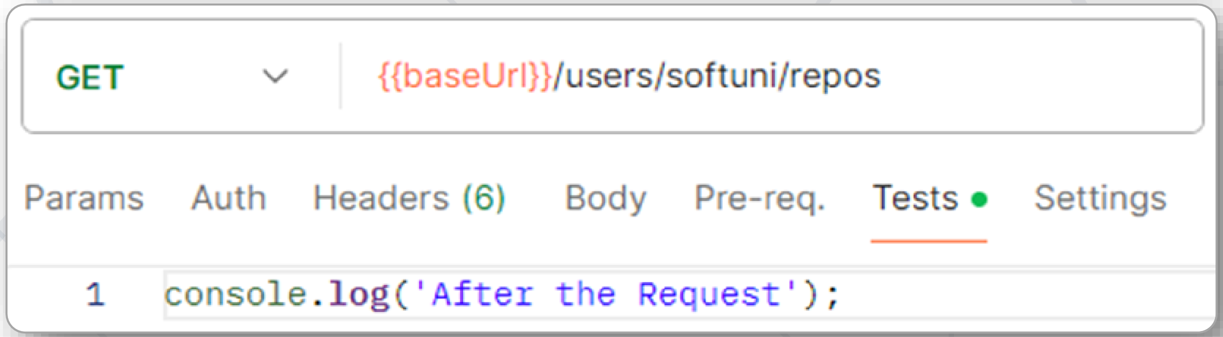
```
console.log('Before the Request');
```



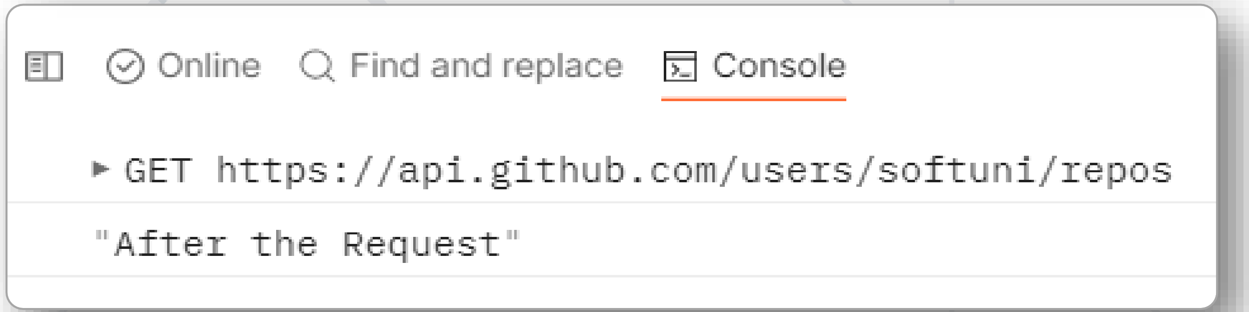
```
"Before the Request"
▸ GET https://api.github.com/users/softuni/repos
```

Tests

- Executed **after the API response** is returned
- Used to **validate the response** to ensure it meets certain conditions:
  - Verifying response **status codes**
  - Ensuring response **body contains specific attributes**
  - Checking the **execution time**
  - Confirming **correct headers**



```
console.log('After the Request');
```



```
▸ GET https://api.github.com/users/softuni/repos
"After the Request"
```

Postman's pm

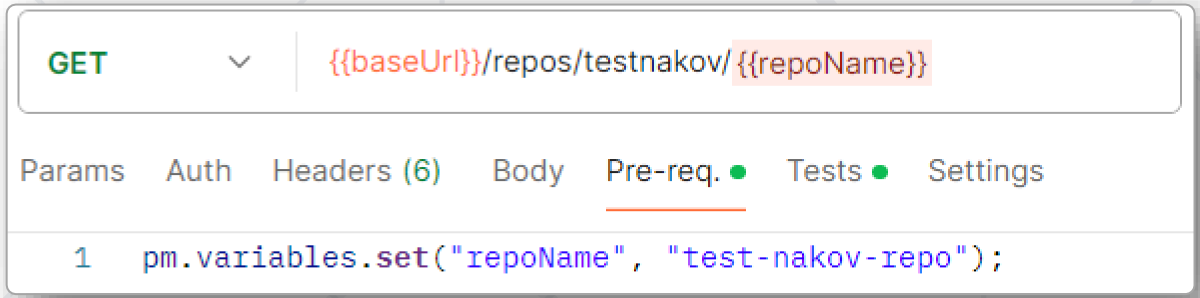
- **pm** object in Postman is a **global namespace** that provides a range of methods and properties
- Used in **pre-request** scripts and **test scripts**
- Stands for "Postman" reflecting its role as a **central component** in **scripting** within the Postman app
- Part of the **Postman Sandbox API**, which enables writing JavaScript code that can enhance and automate different aspects of requests and collections

pm.variables

- pm.variables: Dynamic Data Handlers in Postman
- Typically used to **store data** that may change between executions of requests
- Key to creating **dynamic** and **flexible requests** in Postman
- Allow to:
  - Set Variables: Store data before a request is sent using **pm.variables.set('name', value)**
  - Get Variables: Retrieve stored data to use in requests and tests using **pm.variables.get('name')**

Setting a Variable

- Create a **new variable** called **repoName** and assign the name of the repository you want to check against to
- Use the **{{repoName}}** variable in the request URL

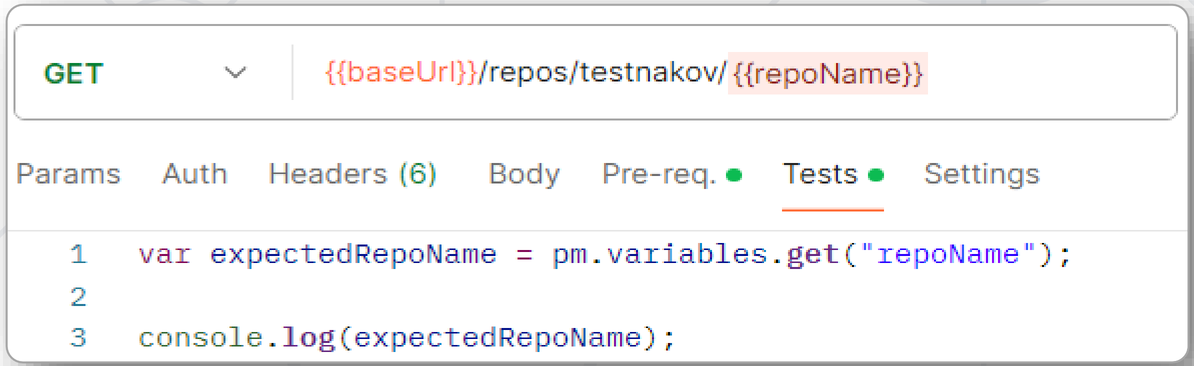


```
{{baseUrl}}/repos/testnakov/{{repoName}}
```

```
pm.variables.set("repoName", "test-nakov-repo");
```

Retrieving a Variable

- When the request is sent, Postman will replace **{{repoName}}** with the actual value of the variable
- Retrieve the value of the variable in the Tests tab to perform assertions



```
var expectedRepoName = pm.variables.get("repoName");

console.log(expectedRepoName);
```

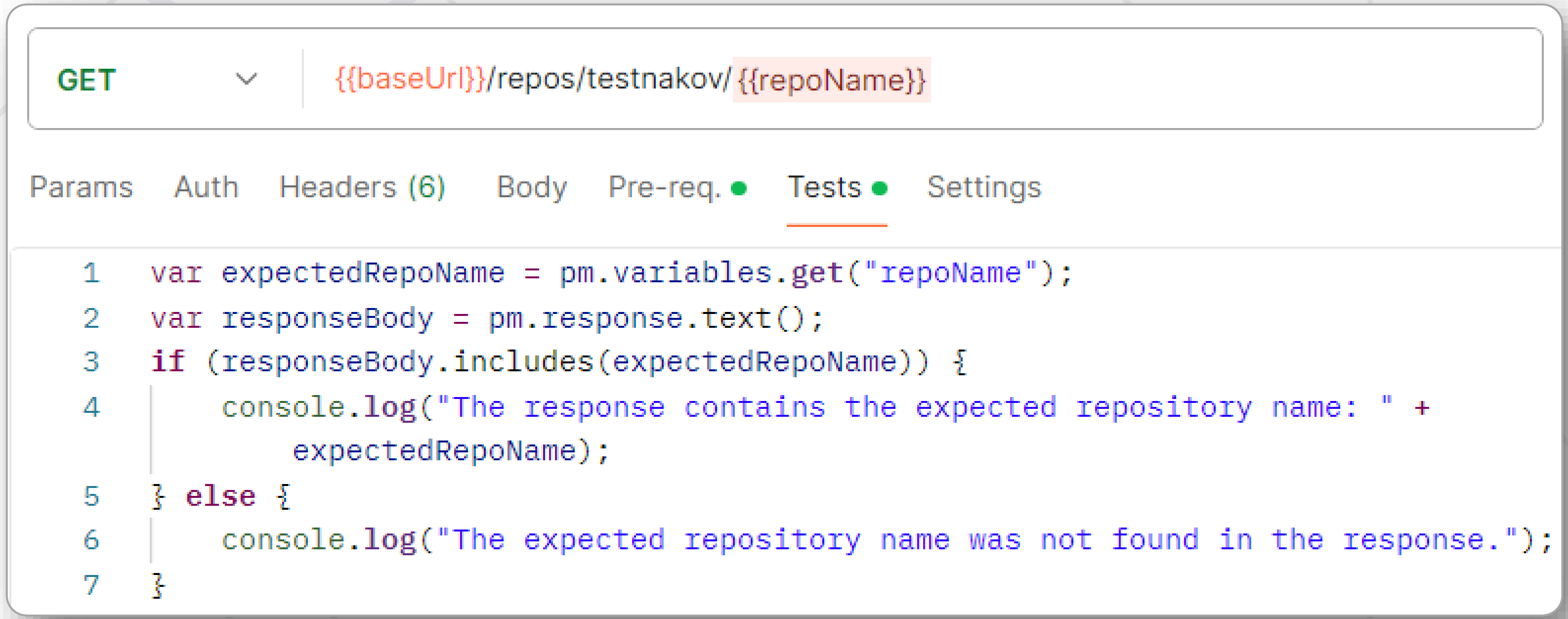
- More on that, later

pm.response

- **pm.response**: Accessing and Inspecting API Responses
- Encapsulates the **details of the response returned** from an API request
- Allows to access the data received:
  - **Check Status Codes**: Verify if the response returned the correct HTTP status code
  - **Examine Headers**: Ensure that the expected HTTP headers are present and correctly formatted
  - **Inspect the Body**: Look at the response body to confirm that it contains the expected information

Using pm.response

- In the **Tests tab, retrieve** the **variable**
- Get the response body as a string
- **Check** if the **Response Body contains** the **Repository Name** and Log the Result



```
var expectedRepoName = pm.variables.get("repoName");
var responseBody = pm.response.text();
if (responseBody.includes(expectedRepoName)) {
    console.log("The response contains the expected repository name: " +
        expectedRepoName);
} else {
    console.log("The expected repository name was not found in the response.");
}
```

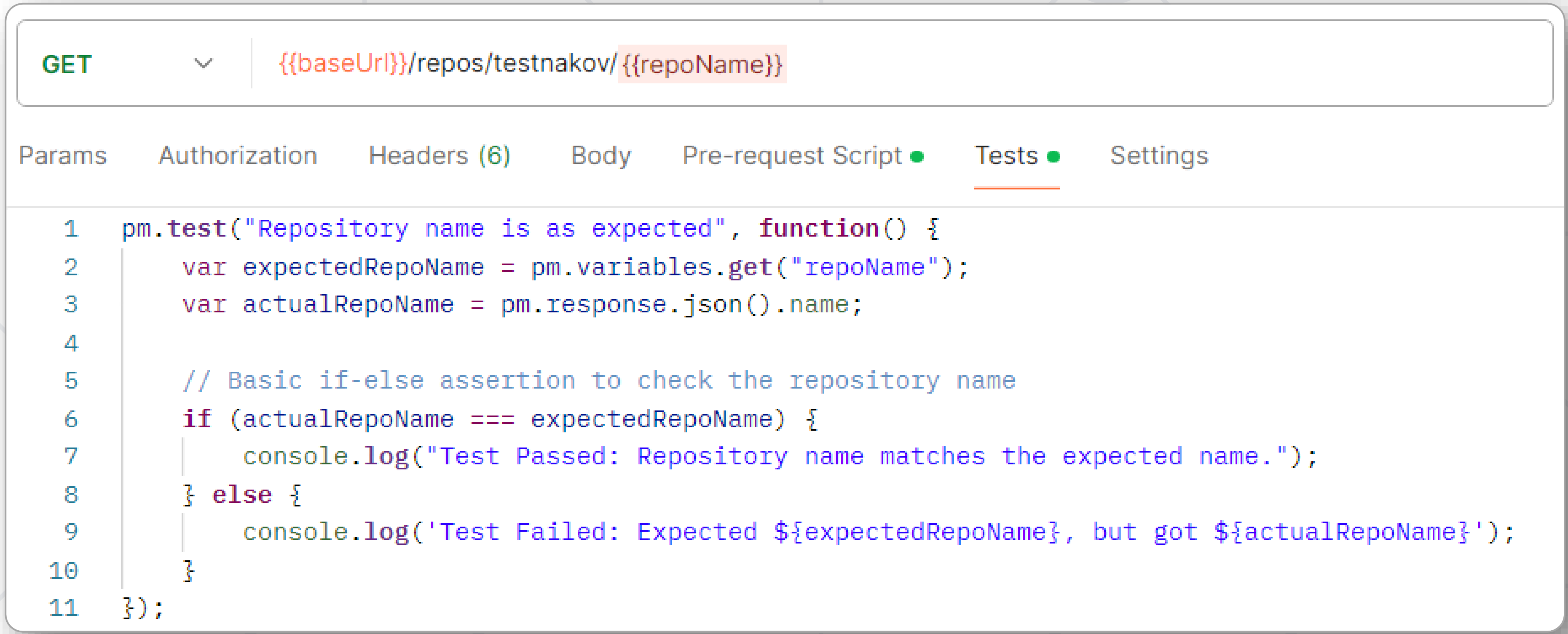
pm.test

- **pm.test**: Grouping and Structuring Tests
- A method in that **allows writing test cases** for verifying the different aspects of an API response
- Each pm.test function **encapsulates assertions** that **evaluate** whether the API response meets **certain conditions**
- Simple structure:
  - The **first arguments** is a **name for the test case**, which describes what the test is checking
  - The **second argument** is a **callback function** that contains one or more assertions to test the response

Using pm.test

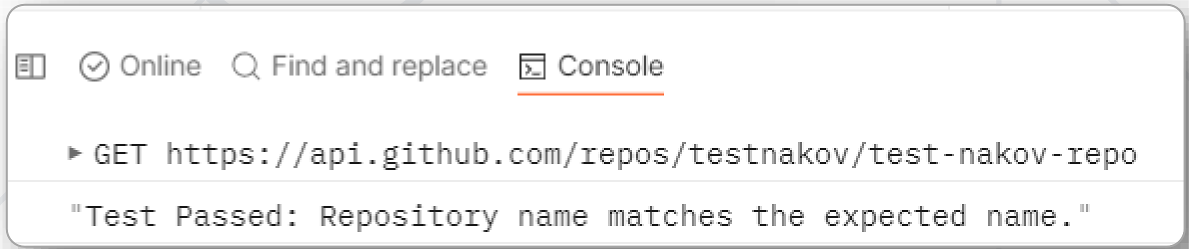
- **Define the test** with a descriptive name: "Repository name is as expected"
- **Retrieve** the expected **repository name** from the variables, using **pm.variables.get**
- **Get the actual repository name** from the JSON response, **pm.response.json().name**
- **Check** if the actual repository name matches the expected name using a **simple if-else statement**
- Based on the condition, **log a message** to the Postman console

Using pm.test



```
{{baseUrl}}/repos/testnakov/{{repoName}}
```

```javascript
pm.test("Repository name is as expected", function() {
    var expectedRepoName = pm.variables.get("repoName");
    var actualRepoName = pm.response.json().name;

    // Basic if-else assertion to check the repository name
    if (actualRepoName === expectedRepoName) {
        console.log("Test Passed: Repository name matches the expected name.");
    } else {
        console.log('Test Failed: Expected ${expectedRepoName}, but got ${actualRepoName}');
    }
});
```



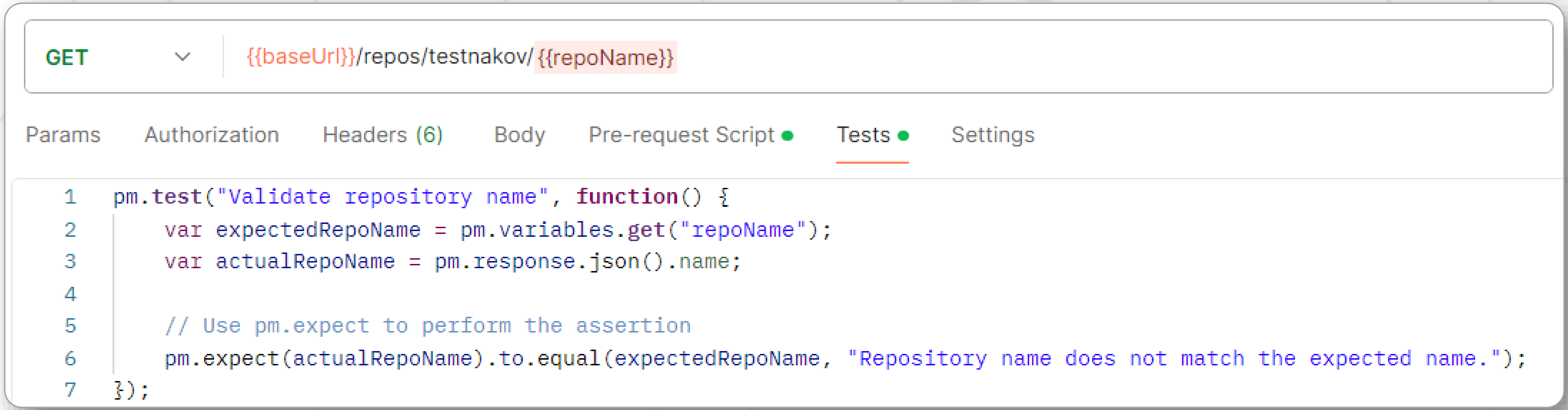
```
▸ GET https://api.github.com/repos/testnakov/test-nakov-repo
"Test Passed: Repository name matches the expected name."
```

pm.expect

- **pm.expect**: Writing Assertive Test Conditions
- An **assertion library** that facilitates writing clear, expressive tests
- Based on Chai's expect BDD library
- Defines the **expected behavior** of an API in a human-readable format
- **Benefits:**
  - **Check** for **specific conditions** within response (value, matching patterns, or data types)
  - **Chainable language** to **construct assertions**, tests are easy to read and write
  - Enables **extensive validation** of API responses against expected values

Using pm.expect

- **pm.expect** is used to assert that the actual repository name (**actualRepoName**) from the response JSON **equals** the expected name (**expectedRepoName**) stored in variables
- If the **repository name does not match**, the **assertion will fail**, and Postman will **report** the provided error message



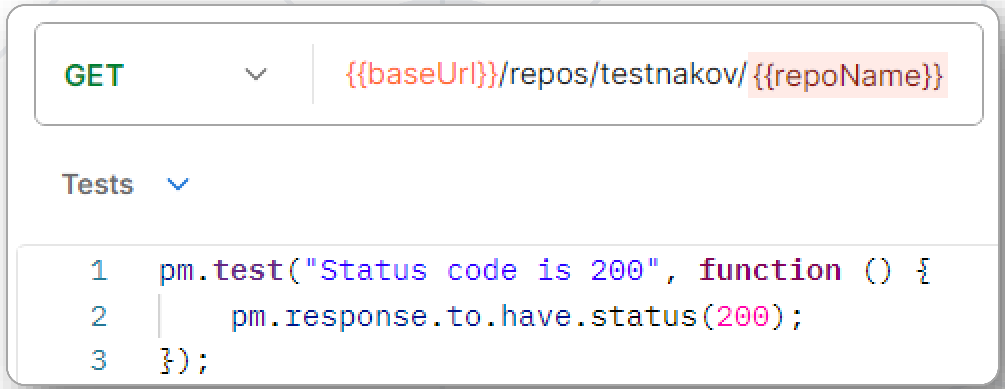
```
pm.test("Validate repository name", function() {
    var expectedRepoName = pm.variables.get("repoName");
    var actualRepoName = pm.response.json().name;

    // Use pm.expect to perform the assertion
    pm.expect(actualRepoName).to.equal(expectedRepoName, "Repository name does not match the expected name.");
});
```

Basic API Tests

Writing your First Tests

Initial focus

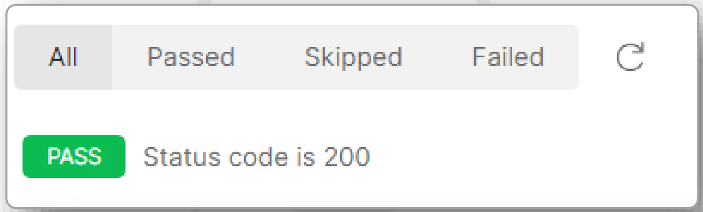
- When testing an API, at a **minimum**, verify the **status code** of **every endpoint**
- From the **Postman snippets** → find the snippet that will test the status code
- Also **importantly**, make sure that the **test will fail if needed**



GET {{baseUrl}}/repos/testnakov/{{repoName}}

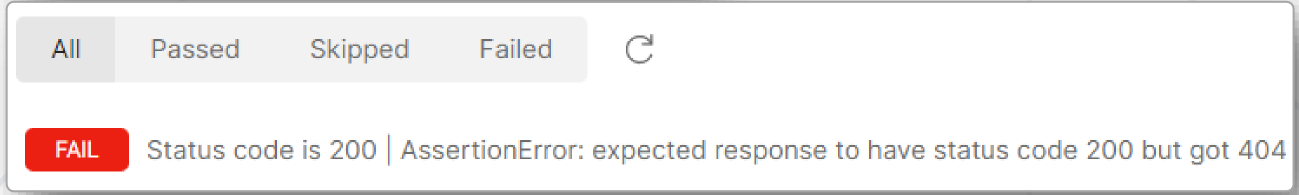
Tests

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



All | Passed | Skipped | Failed

PASS Status code is 200

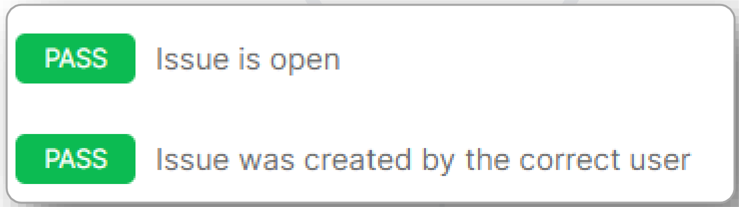


All | Passed | Skipped | Failed

FAIL Status code is 200 | AssertionError: expected response to have status code 200 but got 404

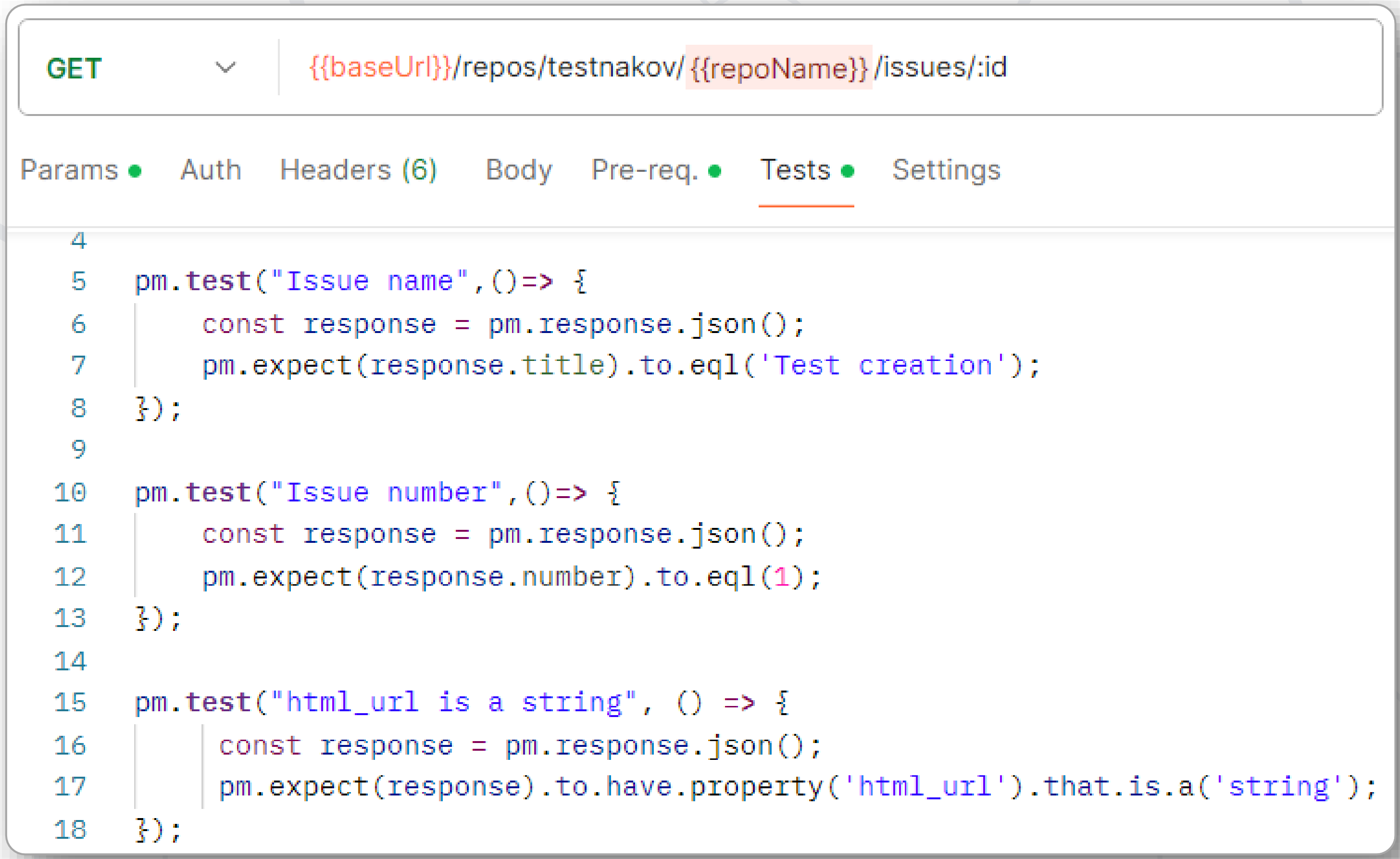
Testing Response Body

- Another area of focus when testing an API is **making assertions against the response body**
- **Test:**
  - The **title** of the issue
  - Issue's **number**
  - Issue's **html_url**
  - Think of other tests



PASS Issue is open

PASS Issue was created by the correct user



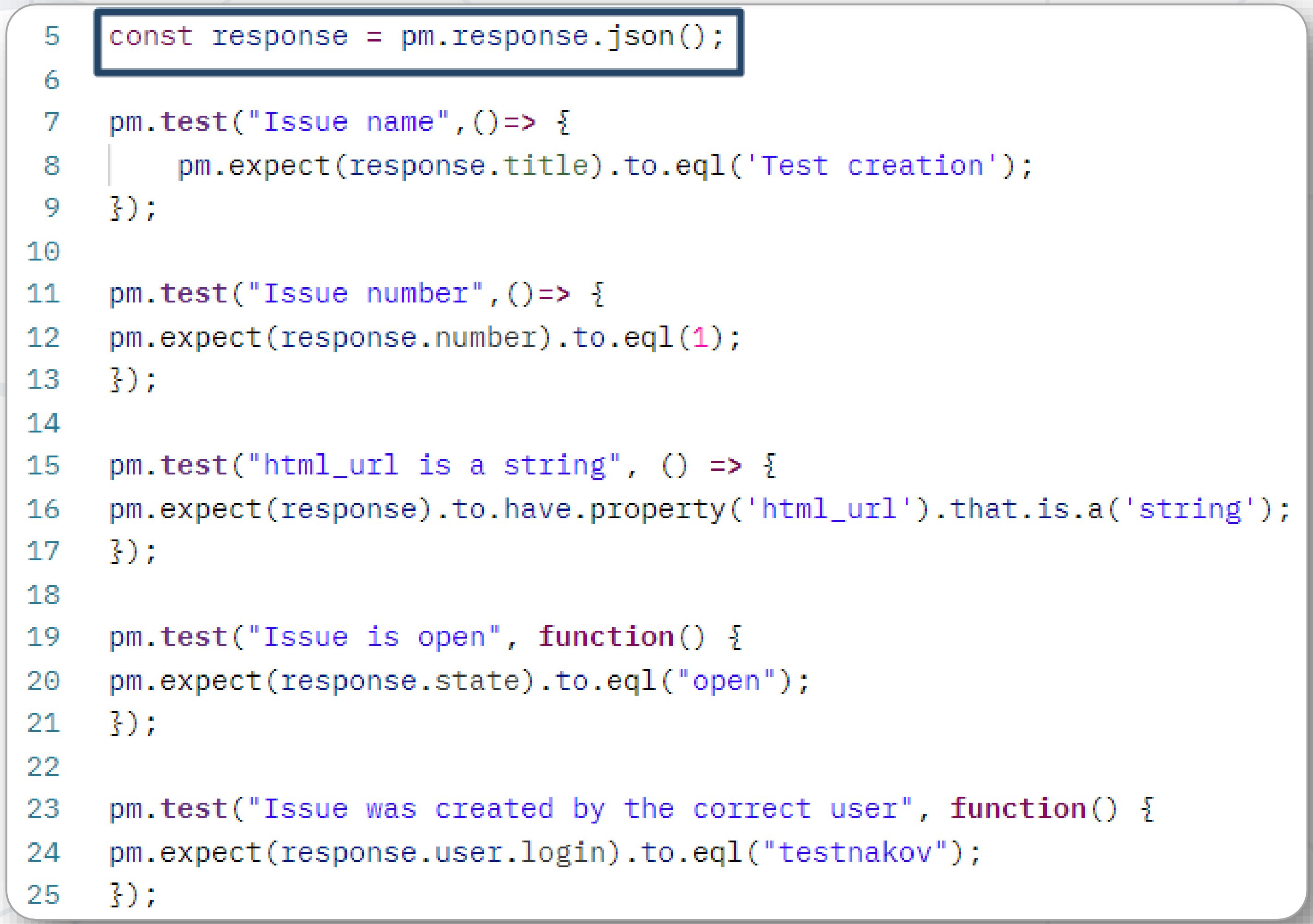
GET {{baseUrl}}/repos/testnakov/{{repoName}}/issues/:id

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

```
pm.test("Issue name",()=> {
    const response = pm.response.json();
    pm.expect(response.title).to.eql('Test creation');
});

pm.test("Issue number",()=> {
    const response = pm.response.json();
    pm.expect(response.number).to.eql(1);
});

pm.test("html_url is a string", () => {
     const response = pm.response.json();
     pm.expect(response).to.have.property('html_url').that.is.a('string');
});
```

Refactoring Tests

- Define the **response** to be in the **global space**
- Outside of the callback function
- **Reuse** it in every test
- Makes tests slightly **smaller**, with **less repetition**

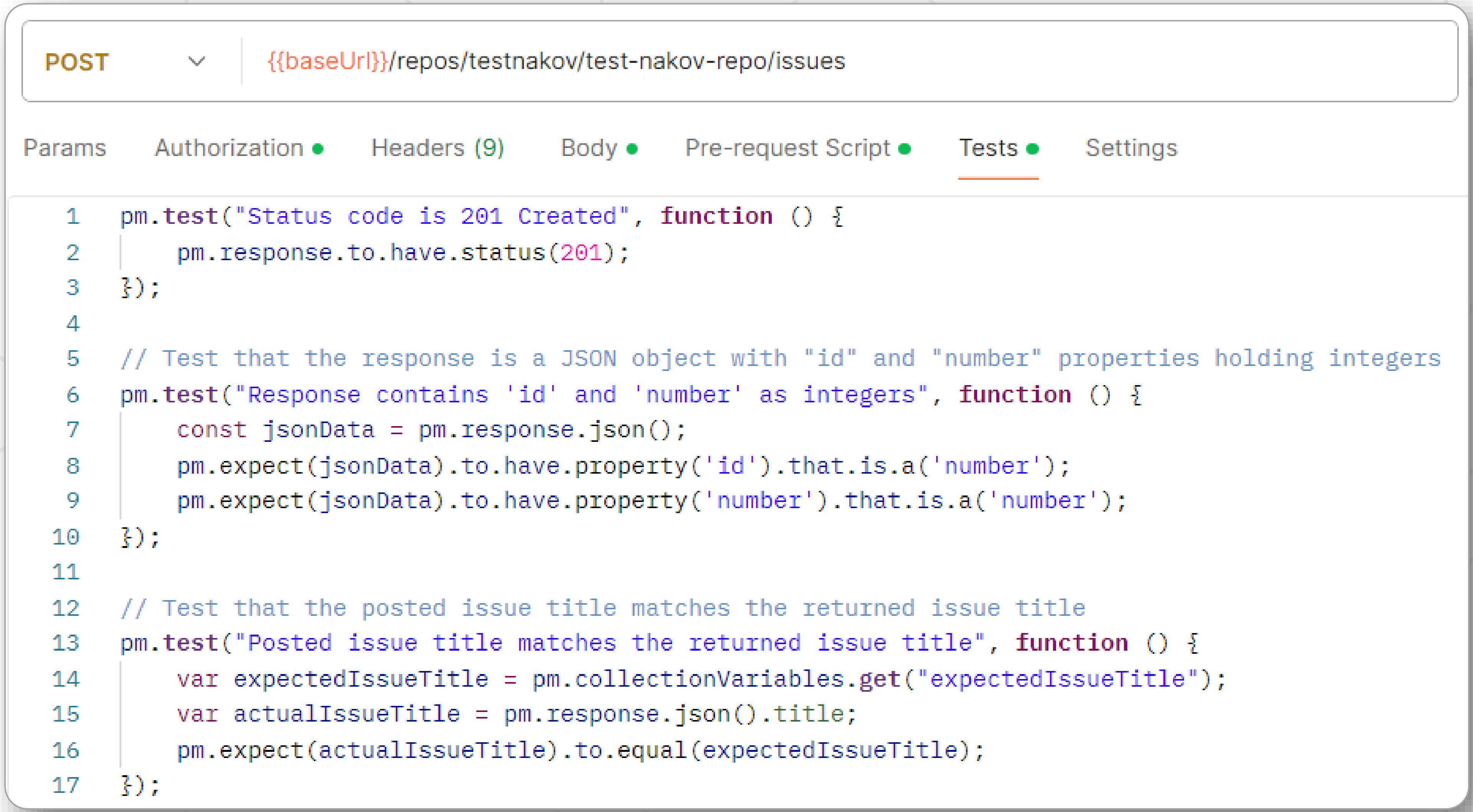


```
const response = pm.response.json();

pm.test("Issue name",()=> {
    pm.expect(response.title).to.eql('Test creation');
});

pm.test("Issue number",()=> {
pm.expect(response.number).to.eql(1);
});

pm.test("html_url is a string", () => {
pm.expect(response).to.have.property('html_url').that.is.a('string');
});

pm.test("Issue is open", function() {
pm.expect(response.state).to.eql("open");
});

pm.test("Issue was created by the correct user", function() {
pm.expect(response.user.login).to.eql("testnakov");
});
```

Problem

- Write Postman API tests for the "**Create new Issue**" HTTP request
- **POST** request with **valid auth data** and **valid JSON body**
- The response should return status **code 201 Created** + the **new issue as JSON object**
- **Assert** that the returned data is a **JSON object, with "id"** and **"number"** properties, which hold **integers**
- Assert that the **posted issue data** (e. g. the issue title) is the same as the **returned issue data**

Solution


POST {{baseUrl}}/repos/testnakov/test-nakov-repo/issues

Params Authorization ● Headers (9) Body ● Pre-request Script ● Tests ● Settings

```
pm.test("Status code is 201 Created", function () {
    pm.response.to.have.status(201);
});

// Test that the response is a JSON object with "id" and "number" properties holding integers
pm.test("Response contains 'id' and 'number' as integers", function () {
    const jsonData = pm.response.json();
    pm.expect(jsonData).to.have.property('id').that.is.a('number');
    pm.expect(jsonData).to.have.property('number').that.is.a('number');
});

// Test that the posted issue title matches the returned issue title
pm.test("Posted issue title matches the returned issue title", function () {
    var expectedIssueTitle = pm.collectionVariables.get("expectedIssueTitle");
    var actualIssueTitle = pm.response.json().title;
    pm.expect(actualIssueTitle).to.equal(expectedIssueTitle);
});
```

Error Handling

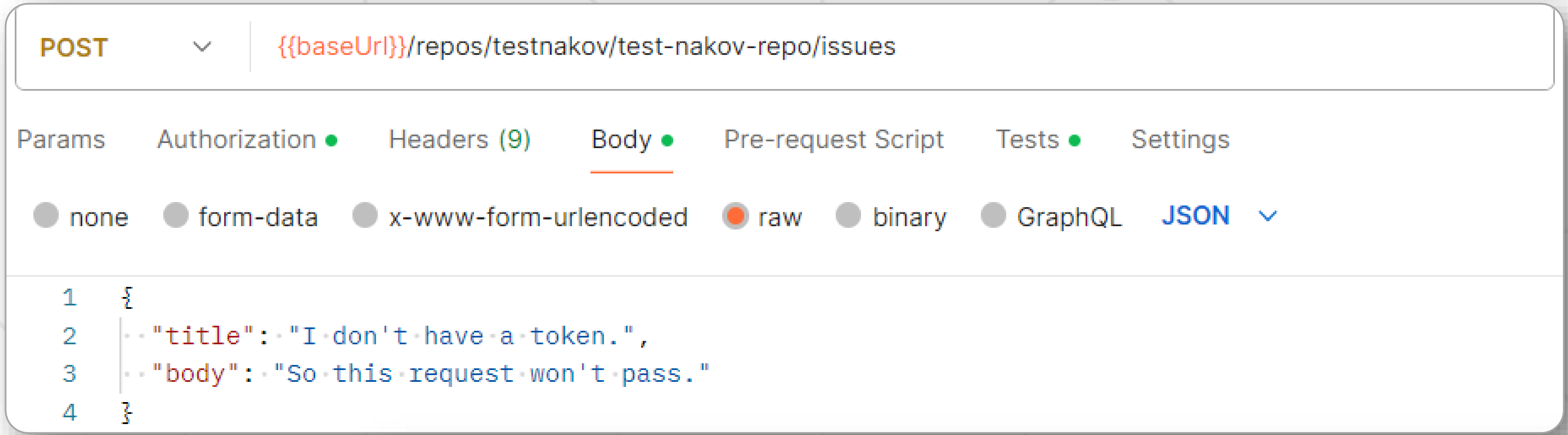
Negative Testing

- So **Happy Path** testing focuses on verifying that the basic functionality of the API **works as expected**
- However, **testing does not end here**
- Some users may experience errors
- The **API should provide the correct error**
- This is called **Negative Testing** and is the **opposite** of **Happy Path** testing
- Negative testing **focuses** on testing **how the API behaves** in **exceptional** or **edge cases**

Problem

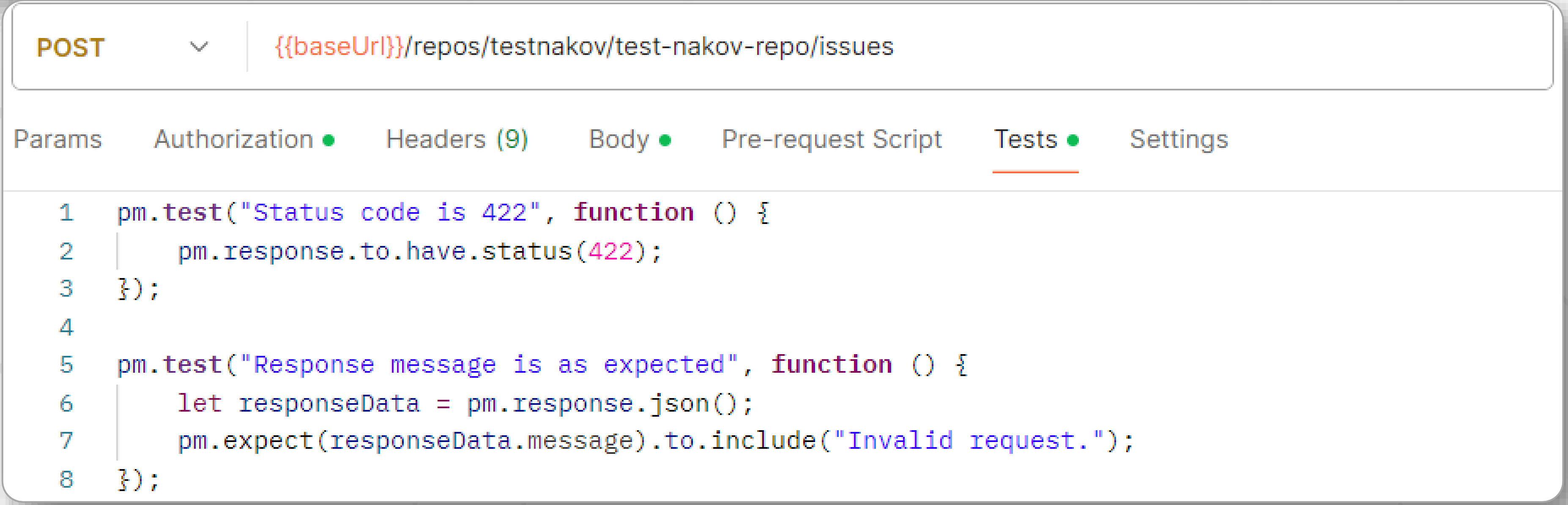
- Write Postman API tests for the "Create new Issue" HTTP request
- **Invalid** authentication data:
  - POST request with **invalid auth data** and **valid JSON body**
  - **Assert** that the response returns **status code 404** Not Found
- Invalid body:
  - POST request with **valid auth data** and **invalid JSON body**
  - **Assert** that the response returns status code **422** Unprocessable Entity
  - **Assert** that the response **message** includes "**Invalid request.**"

Solution



{{baseUrl}}/repos/testnakov/test-nakov-repo/issues

```
{
  "title": "I don't have a token.",
  "body": "So this request won't pass."
}
```



{{baseUrl}}/repos/testnakov/test-nakov-repo/issues

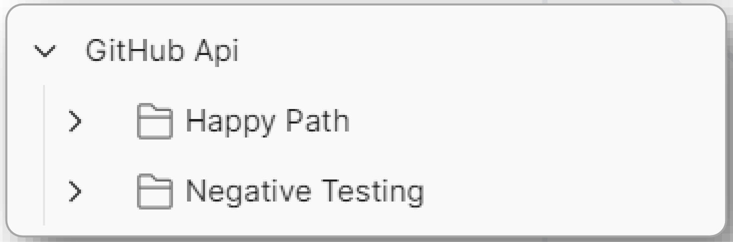
```
pm.test("Status code is 422", function () {
    pm.response.to.have.status(422);
});

pm.test("Response message is as expected", function () {
    let responseData = pm.response.json();
    pm.expect(responseData.message).to.include("Invalid request.");
});
```

Best Practices

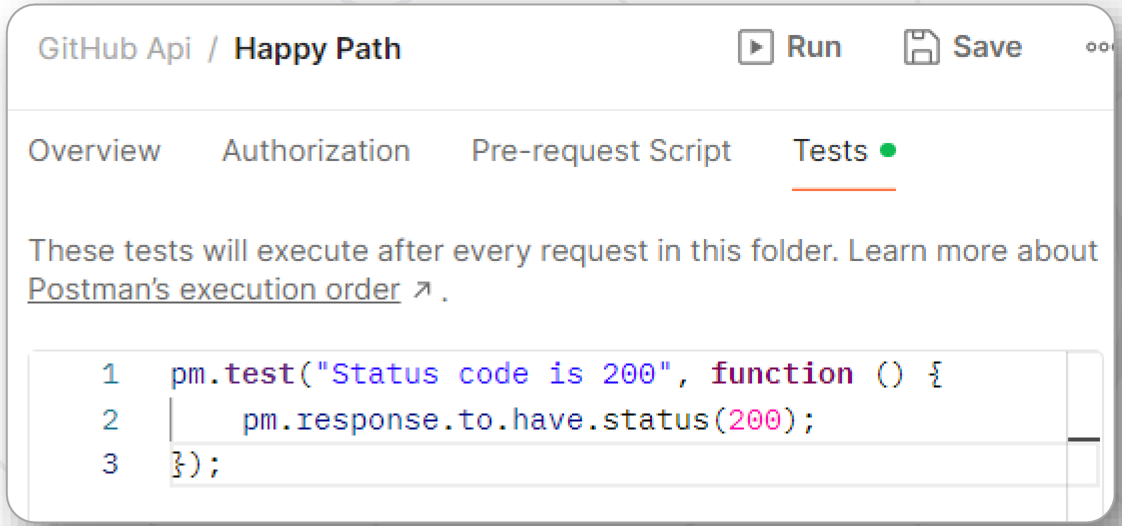
Organizing Tests

Organizing Postman Collections into Folders

- Categorizing requests into logical groups. Easier to locate and understand the purpose of each set
- Organizing tests based on their expected outcomes
- Maintain a tidy workspace, where updates, deletions, or additions are managed more easily
- Allow different members to work on different parts of the API
- Folders can mirror API's versioning system
- Manage the same requests against different environments (Development, Staging, Production)


GitHub Api
Happy Path
Negative Testing

Reusing Tests

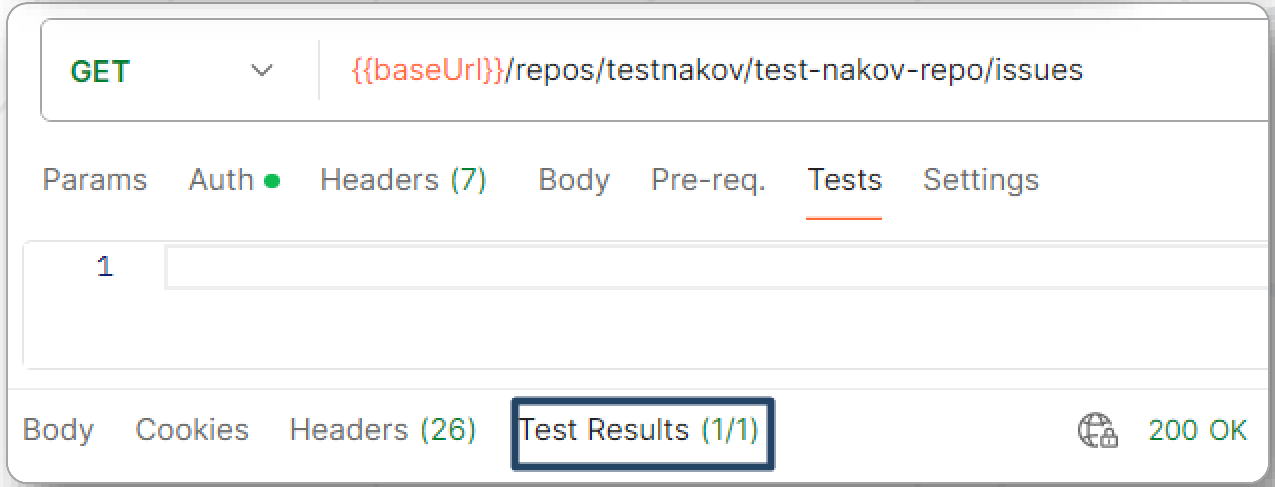


GitHub Api / **Happy Path** Run Save

Overview Authorization Pre-request Script Tests ●

These tests will execute after every request in this folder. Learn more about Postman's execution order ↗ .

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



GET {{baseUrl}}/repos/testnakov/test-nakov-repo/issues

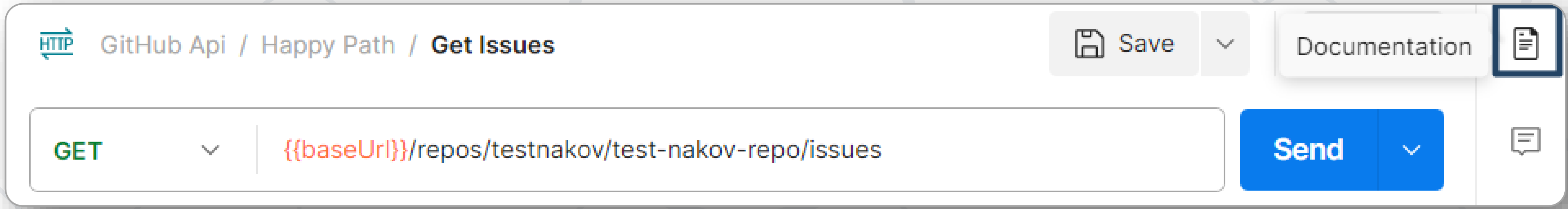
Params Auth ● Headers (7) Body Pre-req. Tests Settings

Body Cookies Headers (26) Test Results (1/1) 200 OK

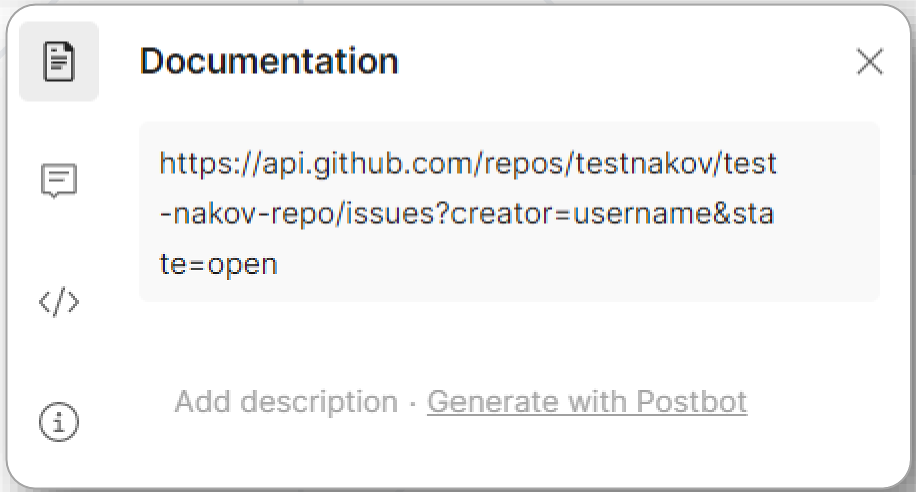
- It's a good practice to **store repeatable tests** in the Folder's Tests tab
- Each Folder can have it's own **Authorization**, **Pre-request Scripts** and **Tests**
- Be careful how you organize your folders

Document Requests

- Use Postman's documentation features to describe what each request does and how it should be used



{{baseUrl}}/repos/testnakov/test-nakov-repo/issues



https://api.github.com/repos/testnakov/test-nakov-repo/issues?creator=username&state=open

Other Good Practices

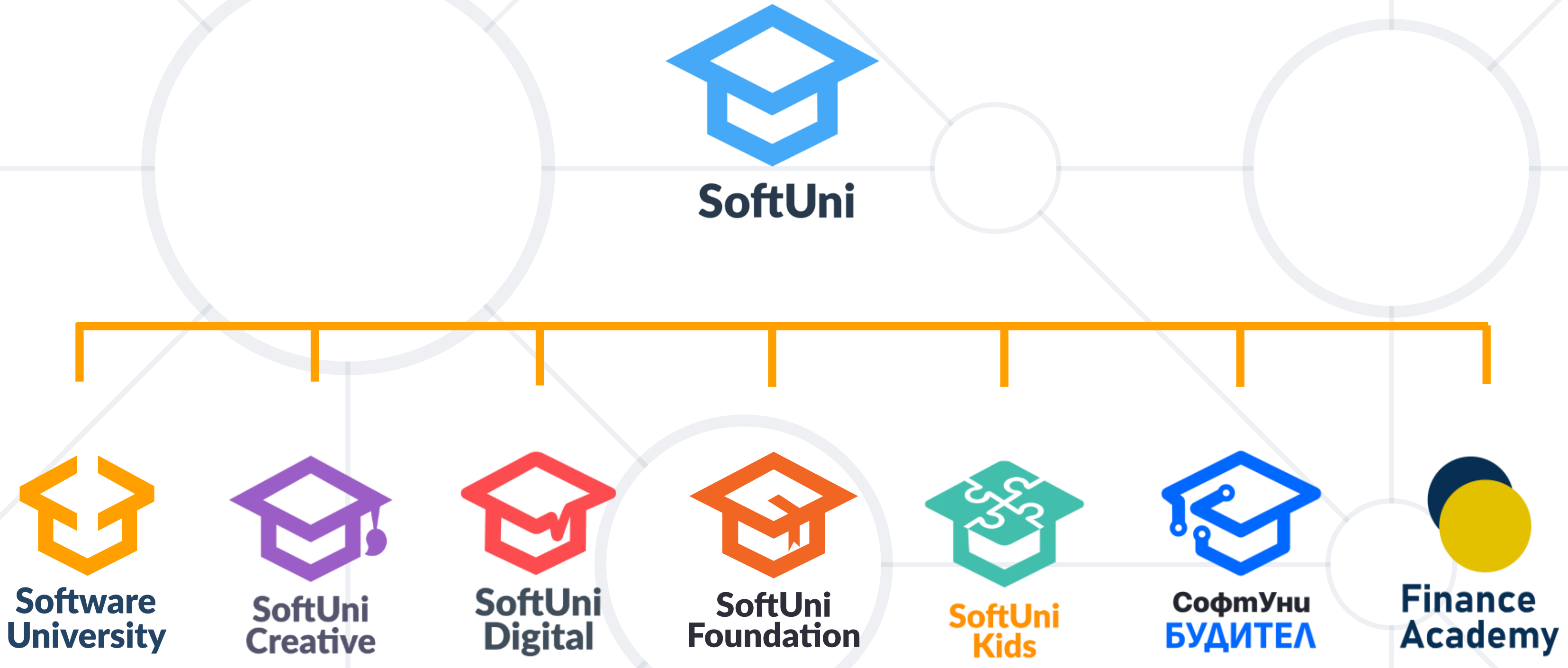
- **Consistent Naming Conventions**: Use clear and consistent naming for collections, requests, and variables
- **Organize Requests by Endpoint or Use Case**: Group requests logically so that it's easy to follow the flow of an application or understand all the operations related to a single endpoint
- **Response Validation**: Always validate the **response structure**, **data type**, and **data content** to ensure that your API is returning the **expected data**

Summary

- Postman – Usage and Capabilities
- Key Terms – What are Collections, Variables, Environments, Requests, Path Parameters
- Core Components – pm.variables, pm.test, pm.response, pm.expect
- Basic API Test – GitHub API
- How to Handle Errors
- Best Practices – How to organize, reuse and document tests

Software University

Questions?



SoftUni Diamond Partners

Software University

SmartIT

createX

VIVACOM

INDEAVR
Serving the high achievers

Coca-Cola HBC
Bulgaria

Postbank
Решения за твоето утре

AMBITIONED

SUPER
HOSTING
.BG

DRAFT
KINGS
THE CROWN IS YOURS

PHAR
VISION

Trainings @ Software University (SoftUni)

- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg, about.softuni.bg
- Software University Foundation
  - softuni.foundation
- Software University @ Facebook
  - facebook.com/SoftwareUniversity

License