API Testing with Postman Lab Guide

Nagaraj Ravinuthala

Oct 2021

Introduction

Sample applications that can be used

JPetStore

During this course we will be using a sample application called JPetStore for doing the hands on. understanding various concepts we will be discussing in the theory. JPetStore is a sample web application which was originally created for learning Java Spring and exploring JVM tools.

Later several performance testing tool vendors started providing this application for testing their tools as part of their documentation.

If you want to set up JPetStore locally on your desktop you can go to GitHub - mybatis/jpetstore-6: A web application built on top of MyBatis 3, Spring 3 and Stripes and follow the instructions given there.

If you want to check the site without installing you can go to JPetStore Demo (octoperf.com).

API layer of JPetStore can be accessed using Swagger UI

Or else if you have access to your own application, you can use the same.

NetBox Dev

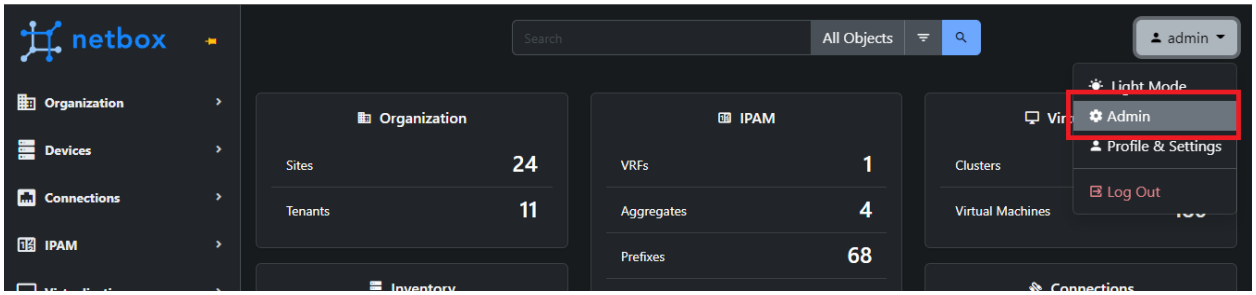
URL: https://demo.netbox.dev/
Credentials: admin/admin
API Documentation: NetBox API
API Browser: API Root | NetBox REST API

Hands-on tasks -- Requests

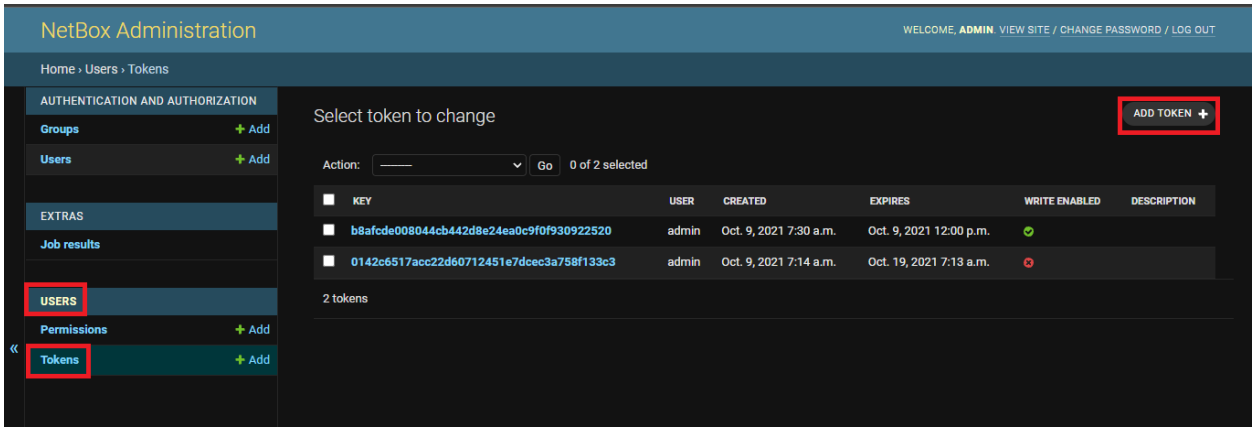
The following tasks demonstrate creating GET, POST, PUT, PATCH and DELETE requests and using variables and correlation between requests by setting collection variables.

Task 1: Finding the token in NetBox:

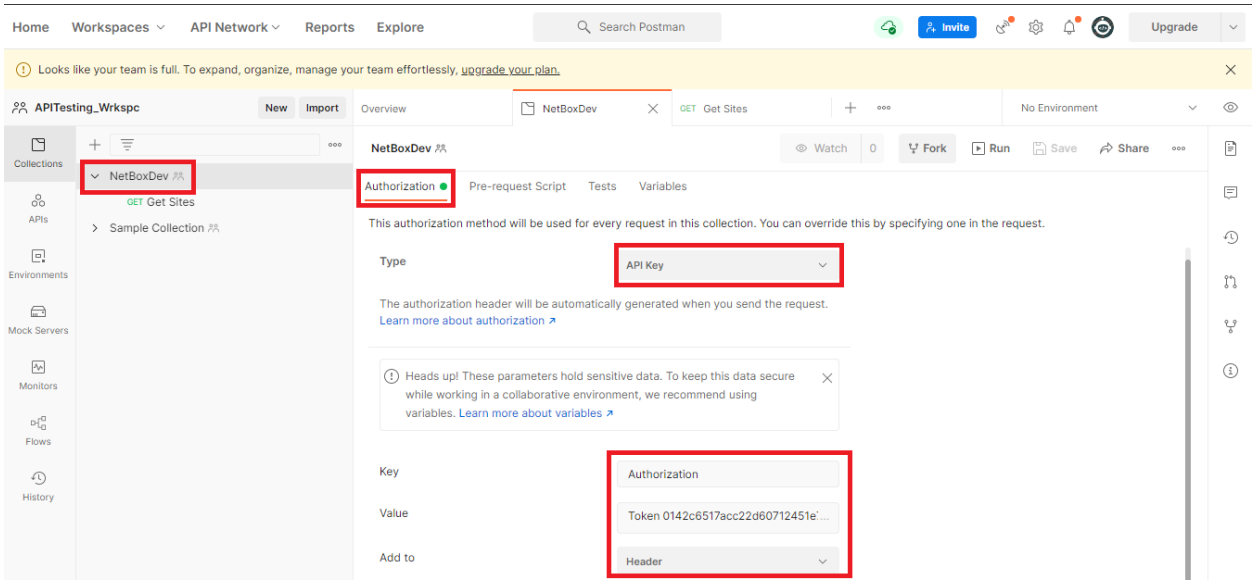
1. Login using the above credentials. Click on Profile.



2. On the admin page, go to Users → Tokens. Click on Add Token or use an existing token.

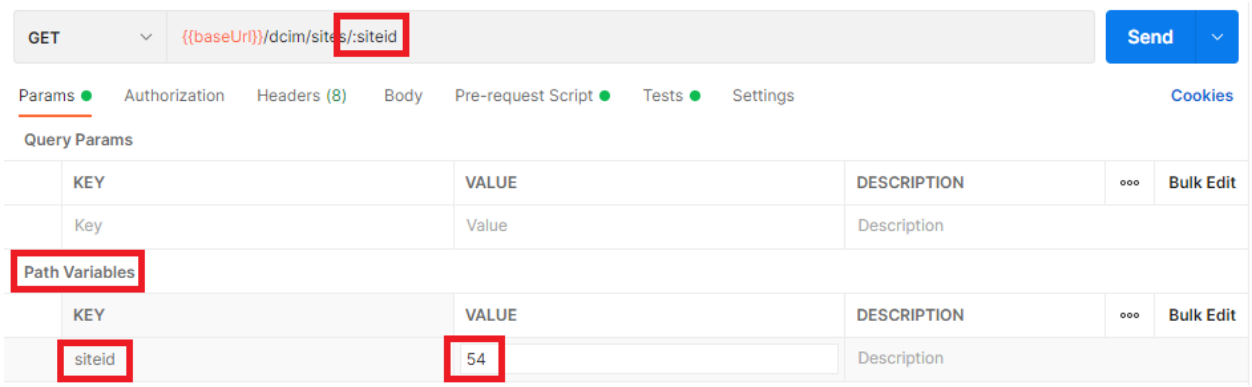


3. Copy the token and keep it handy, as we need to add it in Postman. In Postman go to the collection you just created, click on the Authorization tab, select API Key from the dropdown and add the details as shown below.

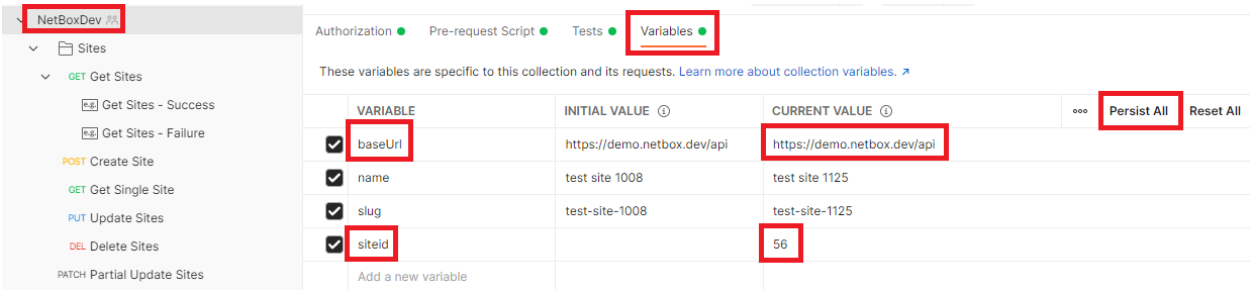


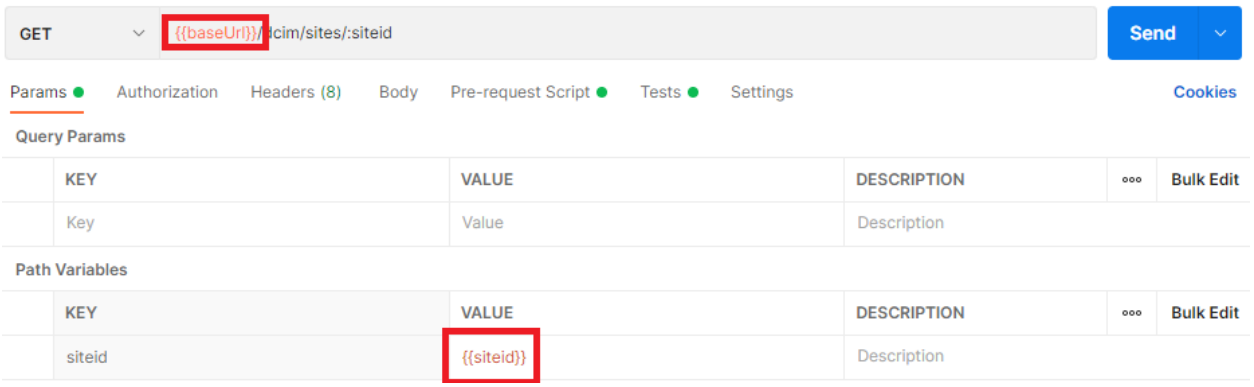
Task 2: Create a collection for NetBox APIs and add a GET request to get all the sites.
Task 3: Get a single site instead of all the sites. This can be done by passing the site id in the endpoint url.

Task 4: Convert the site id passed to “Get Single Site” request into path variable.



Task 5: Create variables for “baseUrl” and “siteid” at collection level. Modify all the requests to refer to the baseUrl variable instead of hard-coded url. Also use siteid variable to pass site id to “Get Single Site” request.





Task 6: Create POST request to create a site. URL to be used {{baseUrl}}/dcim/sites/

Request Body:

{

“name”: “site name”,

“slug”: “site-name”,

“status”: “planned”

}

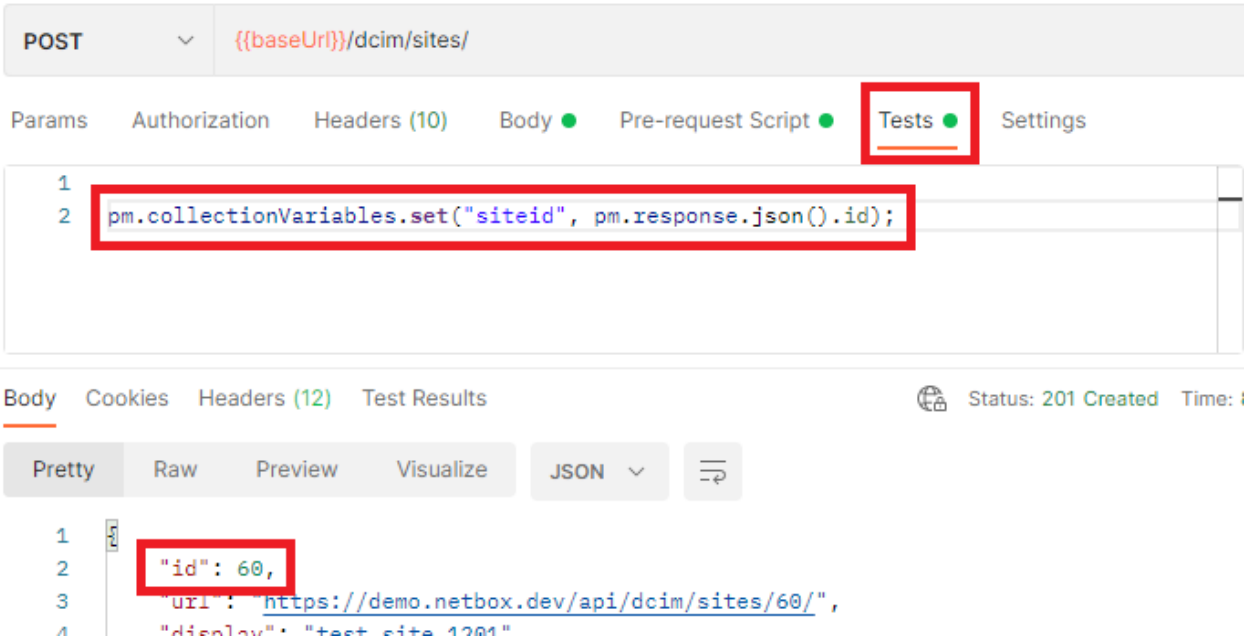
Task 7: Add a pre-request script to the above POST request to generate a random number and concatenate it with a fixed site name and slug to make them unique for each subsequent request. Set the “name” and “slug” as collection variables to be used later.

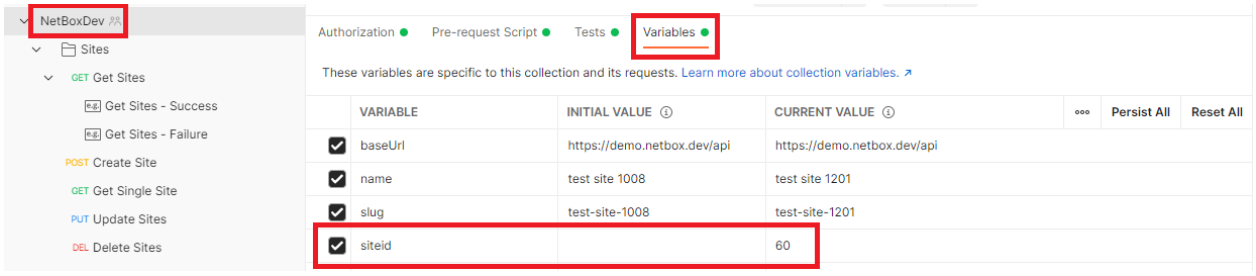
```
let randomPart = Math.floor(Math.random() * 1000 + 1000);
console.log(randomPart);
let name = "test site " + randomPart;
let slug = "test-site-" + randomPart;

pm.collectionVariables.set("name", name);
pm.collectionVariables.set("slug", slug);
```

Request Body now changes as:

```
{
        “name”: “{{name}}”,
        “slug”: “{{slug}}”,
        “status”: “planned”
}
```

Task 8: Add a test script for the above POST request to extract the site id from the response and store it in a collection variable.





Task 9: Create a PUT request to update the status from “planned” to “active”. URL to be used {{baseUrl}}/dcim/sites/ (bulk site update). Use the “siteid” variable stored in the above request.

Request Body:

```
[{
        “id”: {{siteid}},
        “name”: “{{name}}”,
        “slug”: “{{slug}}”,
        “status”: “active”
}]
```

Task 10: Create a PATCH request to do a partial update of the site. The URL to be used is {{baseUrl}}/dcim/sites/ (bulk site partial update).

Request Body:

```
[{
        “id”: {{siteid}},
        “name”: “{{name}}”,
        “slug”: “{{slug}}”,
        “status”: “active”,
        “description”: “site description updated via patch”
}]
```

Task 11: Create a DELETE request to delete the site we created above. URL to be used is {{baseUrl}}/dcim/sites/

Request Body:

```
[{
        “Id”: {{siteid}}
}]
```

Additionals hands-on tasks On Requests

Task 10: Create a PUT request to update a single site.

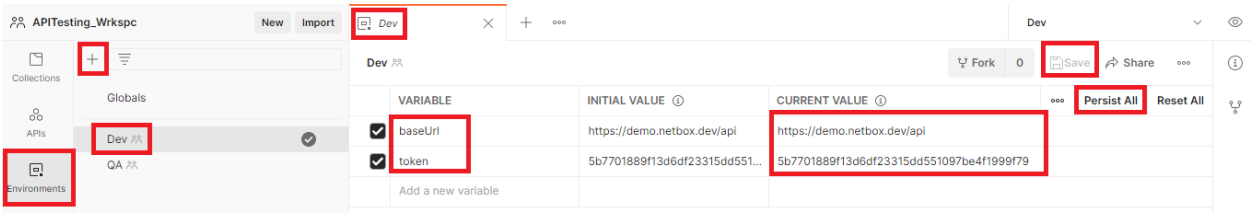
Task 11: Create a PATCH request to update a single site.

Task 12: Create a DELETE request to delete the above created site using the single site delete url. URL to be used is {{baseUrl}}/dcim/sites/{id}/

Hands-on tasks -- Environments

Task 1: Go to Environments tab. Create two environments called Dev and QA. As we know, environments are just collections of variables. Add variables like “baseUrl” and “token” which are common for all collections within the workspace.

Since we do not have an additional env to be used as QA or Prod, just use a different token for each env.



Task 2: Run the requests from the above collection by choosing Dev from the environments dropdown. Run them again choosing QA env.

Hands-on tasks -- Order of execution of scripts

Task 1: Go to any GET request like “Get Single Site”. Add the following statement under the pre-request script.

```
console.log("pre - request :: request");
```

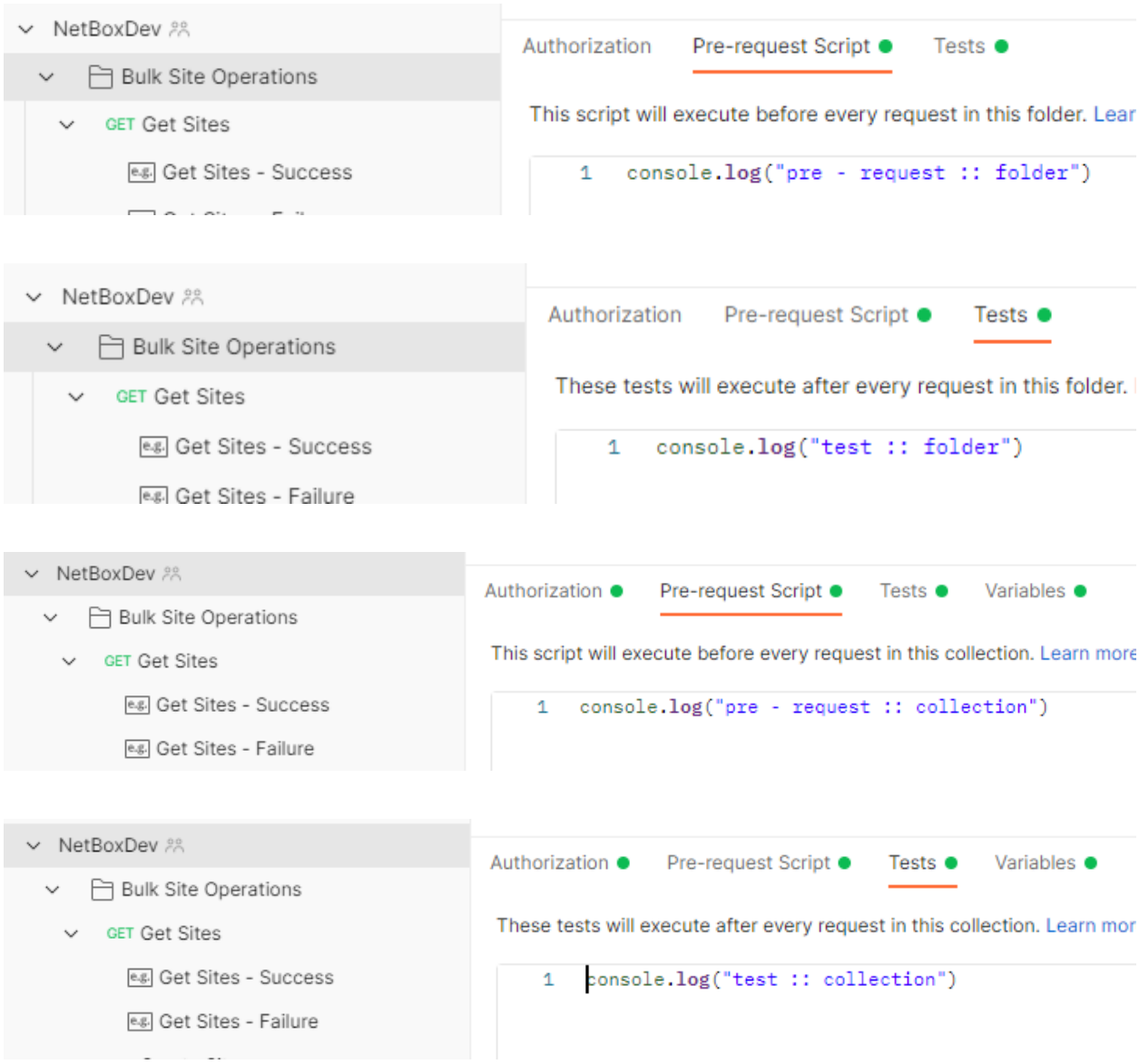
Indicating that this pre-request script is added at the request level.

Go to the Tests tab and add the following command.

```
console.log("test :: request");
```

Indicating that this is a test added at the request level.

Repeat the same at folder and collection level as shown below:

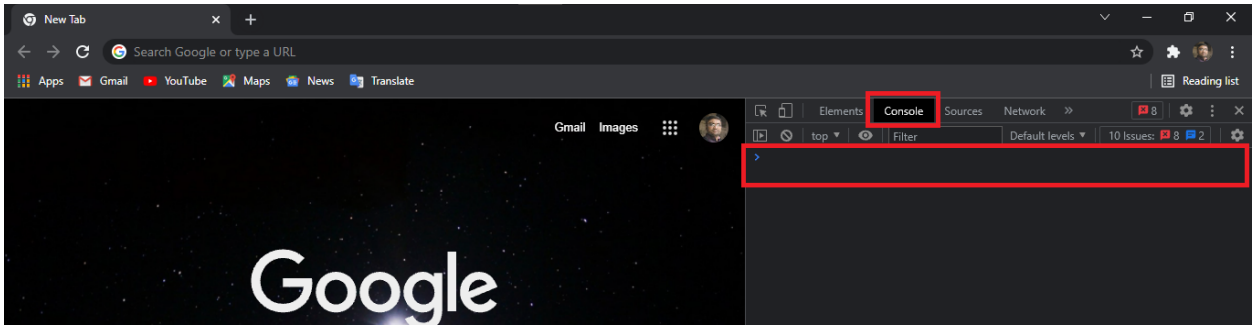


Once done, execute the request and check the console. You should see the following entries.

Check the order of execution of the scripts above.

Hands-On Javascript Basics

1. How to run Javascript?
    a. Using console tab of web browsers
    b. Using Node JS
    c. By building a web page
    d. Using console tab
        i. Open any browser of your choice
        ii. Right click anywhere and look for “Inspect” or “Inspect Element” menu option and select it.



        iii. This is like your command prompt for Javascript.
    e. Using Node JS
        i. Download and install Node JS.
            1. [Download | Node.js (nodejs.org)]
            2. Once done, open command prompt and give the following command
            3. node --version

4. If you get the version info then node is properly installed

ii. Open command prompt and type node
1. You should get the node js prompt where you can start typing js commands

iii. Type console.log("hello") and enter. If you see "hello" printed on the console, you are all set to ready to roll.

f. By building a web page: This would require some additional knowledge on HTML. If you have one of the above two ways working you are good to go.

2. Working with variables. Variables are placeholders for the data you use during the execution of your program.
   a. At the command prompt type `let age = 20;`
   b. We just declared and initialized a variable called age. Do `console.log(age)` to see the value 20 getting printed.
   c. Create few more variables as follows:
      i. `let name = "javascript" // string variable stores text`
      ii. `let langs = ['en', 'ar', 'af']  // list variables stores list of values`
      iii. `console.log(langs)`
   d. We can change the value stored in a variable at any time.
      i. `age = 50;`
      ii. `console.log(age);`
      iii. You should see 50 printed when you issue above commands at the console.
   e. You use const to declare and initialize constants whose value is not to change.
      i. `const pi = 3.14`
      ii. `console.log(pi)`
      iii. `pi = 7.26 //gives error.`
   f. CHeck the type of data stored in a variable using typeof command.
      i. `console.log(typeof(age))`
3. What we saw above are simple data types. Now let us say we want to store all the details of a student like name, age, class, school etc. together. How do we do it?
4. We use something called an object which is a complex data type for this. It simply means we treat the student as a logical object and store his details in it.

```
let student = {
        'name': 'Raj',
        'age': 13,
        'class': 7,
        'marks':{
                'science': 70,
                'english': 90
        }
}
```

5. Objects are used not to just store data related to the entity it represents, but also to define operations we can perform on the data.
6. For instance in the above object, we can define an operation to calculate the average marks of the student. This operation is called an object method or simply method.
7. Modified object looks like this.

```
let student = {
        'name': 'Raj',
        'age': 13,
        'class': 7,
        'marks':{
                'science': 70,
                'english': 90
        },
        average: function(){
                return (this.marks.science + this.marks.english) / 2;
        }
}
```

8. We can call the method to get the average marks as follows:
    a. console.log(student.average())
9. Now this is something that looks familiar right? Let us extend this a bit and create an object with functions that can be called in a chain.

```
let pm = {
response: {
   code: null,
   set_response_code: function(code){
       this.code = code;
   },
   to: function() {
       return this;
   },
   have: function() {
       return this;
   },
   status: function (code){
       if (this.code == code){
           return true;
       }
       else{
           return false;
```

```
            }
        }
    }
}
pm.response.set_response_code(200);
console.log(pm.response.to().have().status(404));
console.log(pm.response.to().have().status(200));
```

10. Looks complex, but when broken down to small pieces, can be understood easily.
11. We can now call something like we have seen in tests.
    a. pm.response(404).to().have().status(200)
        i. Returns false, since the actual status code is 404 while we expect it to be 200.
    b. pm.response(404).to().have().status(404)
        i. Returns true since the actual status code is 404 and we are expecting it to be 404.

Hands-On: Adding tests

Task 1: Add a test to check the status code for each of the requests.

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

Change the above code for create and delete requests to check for the response code received in them instead of 200.

Also add a common test at the collection level to check for a successful response where response code can be one of the values from a list.

```
pm.test("Successful POST request", function () {
    pm.expect(pm.response.code).to.be.oneOf([200, 201, 204]);
});
```

Task 2: Add a test to check the presence of a string in the response body.

```
pm.test("Body matches string", function () {
    pm.expect(pm.response.text()).to.include("count");
});
```

Task 3: Add a test to check the presence of a json value in the response body.

```
pm.test("Your test name", function () {
    var jsonData = pm.response.json();
```

```
    pm.expect(jsonData.id).to.eql(pm.collectionVariables.get("siteid"));
});
```

Task 4: Add a test to check the presence of a header.

```
pm.test("Content-Type is present", function () {
    pm.response.to.have.header("Content-Type");
});
```

Task 5: Add a test for the response time to be below a certain amount. (Can be used for performance testing)

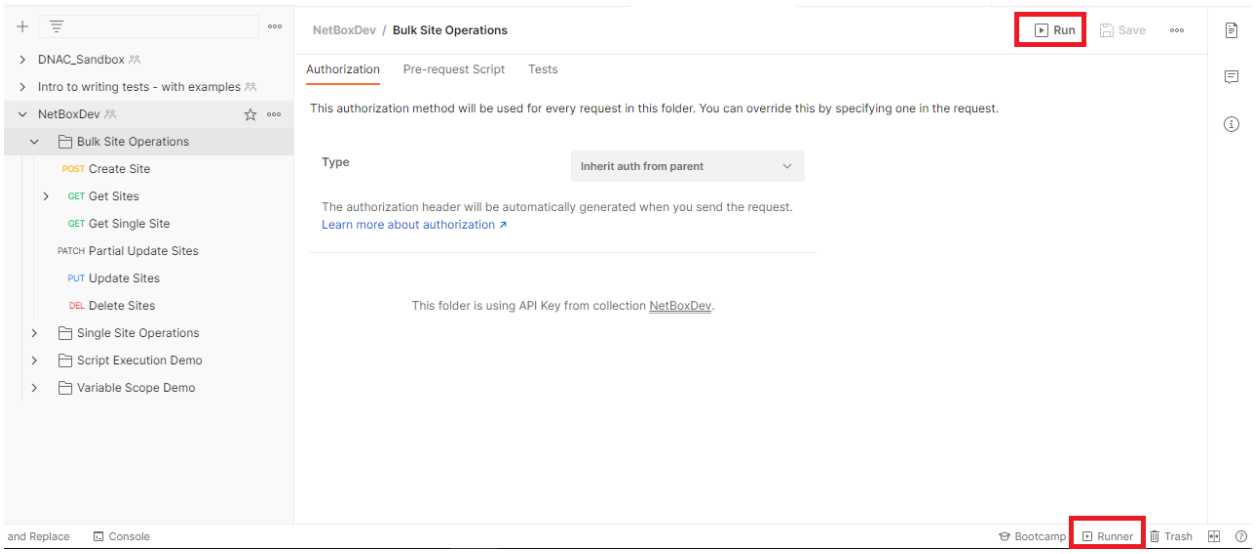
```
pm.test("Response time is less than 1000ms", function () {
    //pm.expect(pm.response.responseTime).to.be.below(200); // this will work
    pm.response.to.have.responseTime.below(1000);  // this will also work
});
```

Task 6: Add a test to check the status message to be as per your ex[ectation.

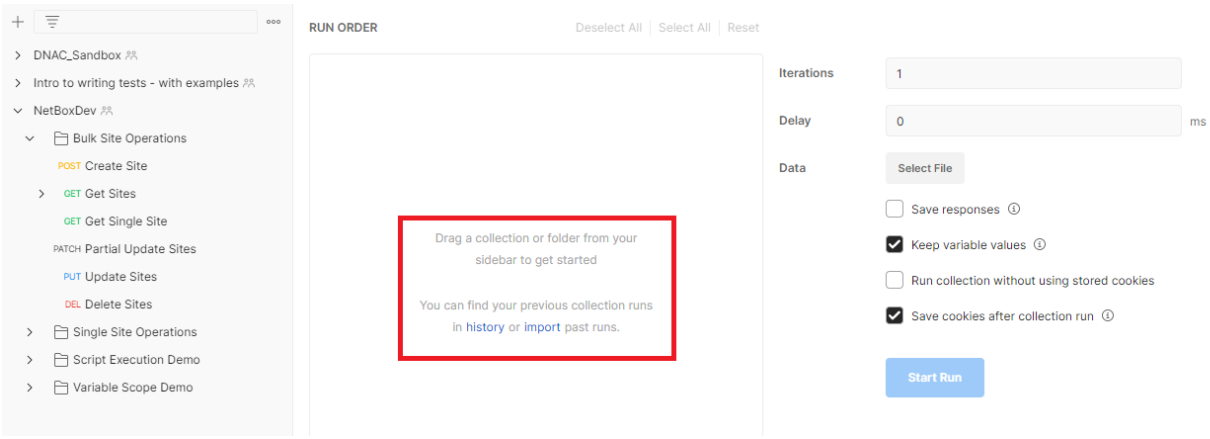
```
pm.test("Status code name has string", function () {
    pm.response.to.have.status("Created");
});
```

Task 7: Execute the entire collection using the Collection Runner.

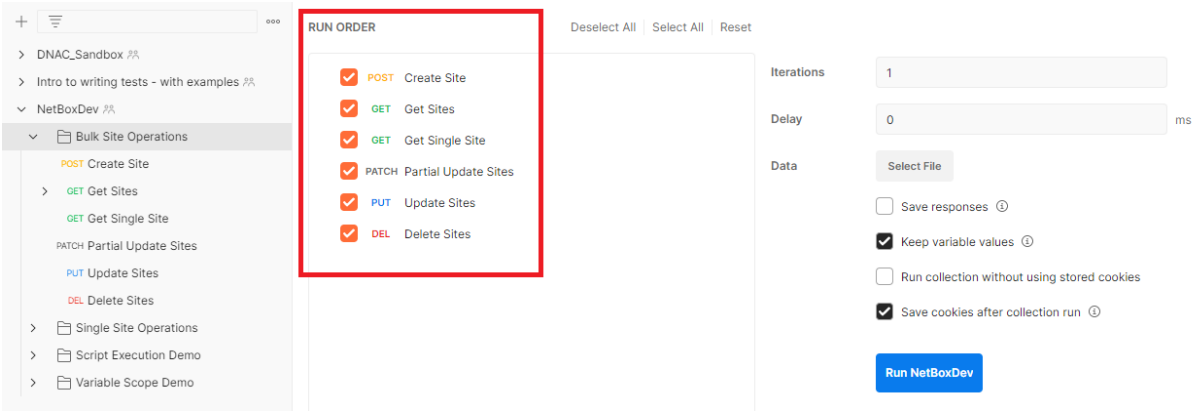
1. Invoke the runner from one of the two options shown below.



2. If you have invoked the runner using the “Runner” button at the bottom, it would be blank. You need to drag the folder or collection into it.



3. If it was invoked using the Run button under folder or collection, the requests under it would be populated.



Task 4: Pass “name” and “slug” from CSV file.

Task 5: Pass “name” and “slug” from JSON file.

End to end Hands On Flow:

1. Create a workspace and invite your team members to the workspace to collaborate with for the below activities.
2. Create a collection using any application that you have access to.
3. Else make use of NetBox Demo site
4. Add requests for all the methods GET, POST, PUT, PATCH, DELETE.
5. Use https://demo.netbox.dev/ demo application for the above examples
6. Login using username/password as admin/admin
7. Set https://demo.netbox.dev/api as baseURL variable
    a. GET
        i. Endpoint - /dcim/sites/
        ii. Typically does not have a request body
        iii. Response code is 200 (OK)

    b. POST
        i. Endpoint - /dcim/sites/
        ii. Requires request body

```
{
   "name": "test site",
   "slug": "test-site",
   "status": "planned"
}
```

        iii. Response code is 201 (Created)

    c. PUT (Bulk Site Update)
        i. Endpoint - /dcim/sites/
        ii. Required request body

```
[
{
   "id": 32,
   "name": "<<sitename for id 32>>",
   "slug": "<<slug for id 32>>",
   "status": "active"
},
{
   "id": 32,
   "name": "<<sitename for id 32>>",
   "slug": "<<slug for id 32>>",
   "status": "active"
}
]
```

iii. Response code is 200 (OK)

d. PUT (Single Site Update)
   i. Endpoint - /dcim/sites/<<siteid>>
   ii. Required request body

```
{
   "name": "<<sitename for id your chosen id>>",
   "slug": "<<slug for id for your chosen id>>",
   "status": "active"
}
```

   iii. Response code is 200 (OK)
e. PATCH (Bulk Site Update)
   i. Endpoint - /dcim/sites/
   ii. Required request body
   iii. Response code is 200 (OK)
f. DELETE (Bulk)
   i. Endpoint - /dcim/sites/
   ii. Requires request body

```
[
   {
      "id": 58
   },
   {
      "id": 59
   }
]
```

   iii. Response code is 204 (No Content)
g. DELETE (Single)
   i. Endpoint - /dcim/sites/<<siteid>>
   ii. No request body
   iii. Response code is 204 (No Content)

8. Parametrize the requests by creating variables for the endpoint URL path and other input values as appropriate.
   a. Click on Environments and go to your environment and create a new variable called “baseUrl” and set its value as “https://demo.netbox.dev/api”.
   b. Click on the “Persist All” button, to share this variable with your team while working in a workspace.
   c. Create Site -- For the site name, fix some part and append a random number or string to make the site name and slug unique for each request.
      i. In the Pre-request script of “Create Site” add the following code.

```
1. let randomPart = Math.floor(Math.random() * 100 + 100);
2. pm.environment.set("name", "test site " + randomPart);
```

3. `pm.environment.set("slug", "test-site-" + randomPart);`

ii. We are setting “name” and “slug” as env variables.

iii. Change your request body as follows to refer to the above variables.

```
{
  "name": "{{name}}",
  "slug": "{{slug}}",
  "status": "planned"
}
```

iv. In the test script, extract the id of the newly created site and set it as an env variable called “siteid” by adding the below code.

1. `pm.environment.set("siteid", pm.response.json().id);`

d. Delete SIngle Site

i. In the above request we stored the id of the site in an env variable called “siteid”. We can use this to delete the site.

ii. If you are using single site delete API, change the endpoint URL as follows:

1. *{{baseUrl}}*/dcim/sites/*{{siteid}}*

e. Delete Site (Bulk)

i. Since Bule Site Delete requires site ids to be passed in body as an array, we can modify the body as follows, by substituting the siteid variable in place of actual site id.

```
[
  {
    "id": {{siteid}}
  }
]
```

f. Update Single Site

i. Update single site URL will change as follows:

1. *{{baseUrl}}*/dcim/sites/*{{siteid}}*

ii. Request body will change as follows:

```
{
  "name": "{{name}}",
  "slug": "{{slug}}",
  "status": "planned"
}
```

g. Bulk Update Site:

i. URL will change as follows after replacing the baseUrl.

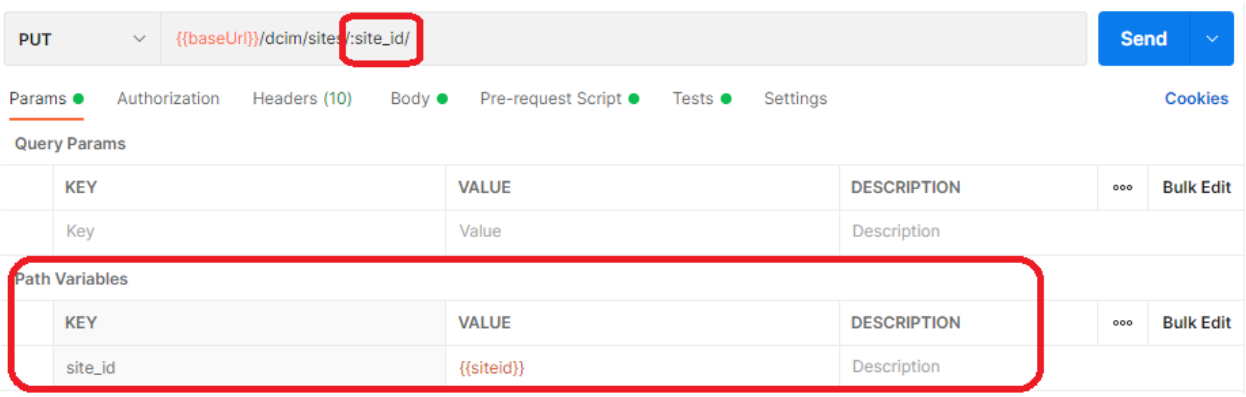
1. {{baseUrl}}/dcim/sites/

ii. Request body will change as follows after replacing it with siteid, name and slug variables.

```
[{
  "id": {{siteid}},
```

```
        "name": "{{name}}",
        "slug": "{{slug}}",
        "status": "planned"
    }]
```

9. Use both query params and path params while parameterizing.
    a. All single site operations like get site by id, update single site and delete single site expect site id to be passed in the URL.
    b. This can be converted into path parameter as follows:
        i. {{baseUrl}}/dcim/sites/:site_id/
        ii. This will add a section called Path Variables to the Params tab



10. Create multiple environments.
    a. Create an environment and call it “Dev Env”.
    b. Add 2 variables “baseUrl” and “token” and set the values which are appropriate for the Dev env.
    c. Create another environment called “Prod Env”.
    d. Add the same 2 variables which we have added above to the Dev Env.
    e. But this time the values of these variables will change as appropriate to the prod env.
    f. When we add a new environment, we need to add only those variables which we have added manually to the other environment.
    g. Variables which are getting set at the environment level, through the script, will be added to the new env, when the script runs against that env.
11. Create same variable at the following locations with different values to understand the variable scope
    a. Create a dummy collection. Add any request.
        i. In the pre-request script ass the following variable statements
            1. `let myvar = "available only in this script";`
            2. `pm.variables.set("myvar", "available to this request");`
            3. `pm.environment.set("myvar", "available only when environment is selected");`
            4. `pm.collectionVariables.set("myvar", "available to all requests in the collection");`

5. pm.globals.set("myvar", "available when the variable is not available at any of the lower scopes");

b. In the body set the request body as:

```
{
   "var":"{{myvar}}"
}
```

c. Run the request:
   i. By selecting an environment
   ii. Without selecting any environment

d. Create another dummy request.
   i. In its pre-request script add the following commands:
      1. let myvar = pm.variables.get("myvar");
      2. console.log(myvar);
   ii. Run this request again with and without selecting an environment.

12. Randomize the above variables using the dynamic variables option provided by Postman.
    a. Under Create Site request pre-request script add the following commands:
       i. let randomPart = pm.variables.replaceIn("{{$randomAbbreviation}}");
       ii. pm.environment.set("name", "test site " + randomPart);
       iii. pm.environment.set("slug", "test-site-" + randomPart);
    b. Run the request and check the response.
13. Set some variables in pre-request script at environment and global level and verify the same in Test script using get methods.
    a. Add the following commands in the pre-request script.
       i. let myvar = "available only in this script";
       ii. pm.environment.set("myvar", "available only when environment is selected");
       iii. pm.collectionVariables.set("myvar", "available to all requests in the collection");
       iv. pm.variables.set("myvar", "set as variables");
       v. pm.globals.set("myvar", "available when the variable is not available at any of the lower scopes");
    b. Add the following commands in the test script.
       i. console.log(pm.environment.get("myvar"));
       ii. console.log(pm.collectionVariables.get("myvar"));
       iii. console.log(pm.variables.get("myvar"));
       iv. console.log(pm.globals.get("myvar"));
    c. Run the script and check the console output.
14. Add basic tests for request code, message, response time etc.
    a. Add a test to validate response code to Create Site request under Test tab

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

b. Add a test to validate response time. Time is measured in milli sec. So if you expect the response to be under 1 sec, you can use the below code.

```
pm.test("Response time is less than 1 sec", function () {
    pm.expect(pm.response.responseTime).to.be.below(1000);
});
```

15. Run the tests against multiple environments.
    a. Just before sending the request, from the environment dropdown, select an environment against which you intend to run the test.
16. Save responses in a file as well as examples.
    a. Responses can be saved as json files. In the response section, look for Save Response dropdown which shows 2 options.
        i. Save as example
        ii. Save to a file
    b. Try both the above options.
    c. “Save as example” option stores the static response under that request. This will help as documentation for future team members.
17. Run the entire collection.
    a. Select Collection and click the Run button.
    b. Or optionally, click the Runner button at the bottom right side of the pane, to bring up the runner and drag a collection into it.
    c. Both the above options populate the runner with the methods under the collection.
    d. Once the requests are added to the collection, we can drag them up or down to change the order in which they would be executed.
    e. We can also uncheck any request, if we want to omit it from the current run.
    f. Click the “Run <<collection name>>” button to start the collective execution.
18. Automate collection run using monitors.
    a. Go to the Monitors tab and click Create Monitor.
    b. Enter a name and select a collection from the dropdown.
    c. Select the Environment against which the collection has to be run.
    d. Select the schedule as either Week Timer or Hour Timer.
    e. Leave other fields as default and click Create Monitor.
    f. Graph shows all the test runs.
    g. Click on a specific test run to see its results and the console log.
19. Make your tests data driven.
    a. Open Collection Runner choosing a collection.
    b. Click on the Select File button and choose the data files in csv or json format.
    c. E.g. In the Create Site request, we can pass data for the variables “name” and “slug” from the csv file.
    d. So csv for this would look like:
    name,slug

Chicago,chicago

e. Once the file is selected use the Preview button to ensure data is read properly.
f. By default the number of iterations would be the same as the number of data rows in the file.
g. We can increase or decrease the number of iterations.

20. Add additional tests to check whether response data is as per the data sent from file.
    a. Use pm.iterationData methods to extract data that was passed from file.
    b. Some useful methods are :
        i. pm.iterationData.has(variableName) → returns true or false
        ii. pm.iterationData.get(variableName) → returns the data associated with the variable name.
        iii. pm.iterationData.toJSON() converts the data object into json.
21. Create a workflow around this collection.
    a. Decide the order in which you want the requests to be executed.
    b. The first request that you want to run, should be kept at the top of the collection runner.
    c. In the pre-request or test script of the first request you put the following command:
        i. postman.setNextRequest(<<request name>>);
    d. Open Create Site and go to Test script and add the following command.
        i. postman.setNextRequest("Get Sites");
    e. Open Get Sites request test script and add the following command.
        i. postman.setNextRequest("Delete Single Site");
    f. Open Collection Runner on this collection.
    g. Leave only Create Site, Get SIte and Delete Single Site requests checked and uncheck others.
    h. Make sure that Create Site is at the top of the run order.
    i. Drap Delete Single SIte to second place.
    j. Leave Get Sites at third place.
    k. Go to Delete Single Site request Test scripts and add the following command.
        i. `postman.setNextRequest(null);`
22. Create a mock server which provides dummy GET, POST, PUT, DELETE requests with the required request body and expected response body and appropriate response codes.
23. Run it from the command prompt using Newman.
    a. Export the collection as a json file.
    b. Export the environment as a json file.
    c. Open a command prompt and give the following command.
    d. Newman run <<collection json file name with path if required>> -e <<environment json file name with path if required>> -d <<data file name with path if required>>
24. Run tests from command prompt with data files.
25. Run tests from command prompt with workflow.
26. Trigger it from Jenkins.