**API Testing using Postman Tool:**

**Terminologies:**

* URL-uniform resource locator
* URI-uniform resource identifier
* URN-Uniform resource name

Example: http://www.google.com/articles/articlename

scheme-https host-google.com path-articles/articlename.

* Resource-something which is available in server which we request.
* Feature: In manual testing to test some functionality and similarly resource used in API automation referring some functionality.
* Payload:(request and response payload) The data which we sending along with request and data we getting along with response.
* Example: train ticket and flight ticket booking.
* Endpoint will be given by developer and also, they provide the location where api’s are deployed accordingly the url will change.

**Postman tools:**

* Create, rename and delete Workspace-Where we maintain files that means test cases. Workplace will be created in google itself.
* Collection-Collections in Postman provide a convenient way to organize and manage your API requests.
* They are especially useful when dealing with multiple requests related to a particular feature or functionality.

**Different types of HTTP request:**

1. Get-Retrieve the resource from database.
2. Post- Create resource from database.
3. Put-update existing resource in database.
4. Patch-Update partial details of resource.
5. Delete-Delete existing resource from database.

Let see how to create HTTP request under collection.

**Validation:**

* + status code
  + Timers
  + size of data
  + Response body which normally in Json or xml
  + cookies
  + Header

HTTP Status Codes:

3levels:

* 200
* 300
* 500

**To create own API, follow below steps:**

Step 1: Node JS

Npm-node package manager

Node –version

Npm –version

Step 2: json-server

**Authentication (both interrelated concept)**

Is to verifying who someone is

Eg: You need to wear ID card for your office or college.

**Authorization:**

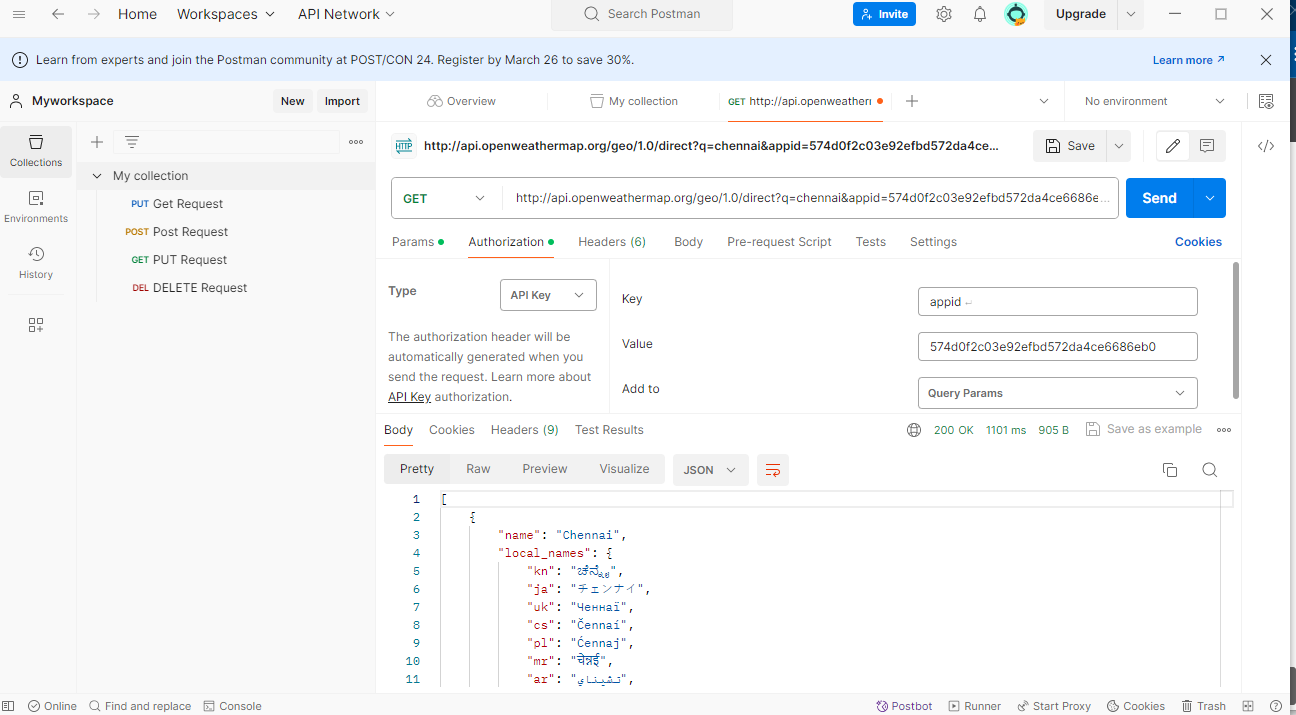
Access level restriction (i.e: privilege level restriction)

Authorization Types:

1. No auth
2. API key
3. Bearer token
4. Basic auth
5. Digest auth
6. Oauth 1.0
7. Oauth 2.0
8. **API key:**

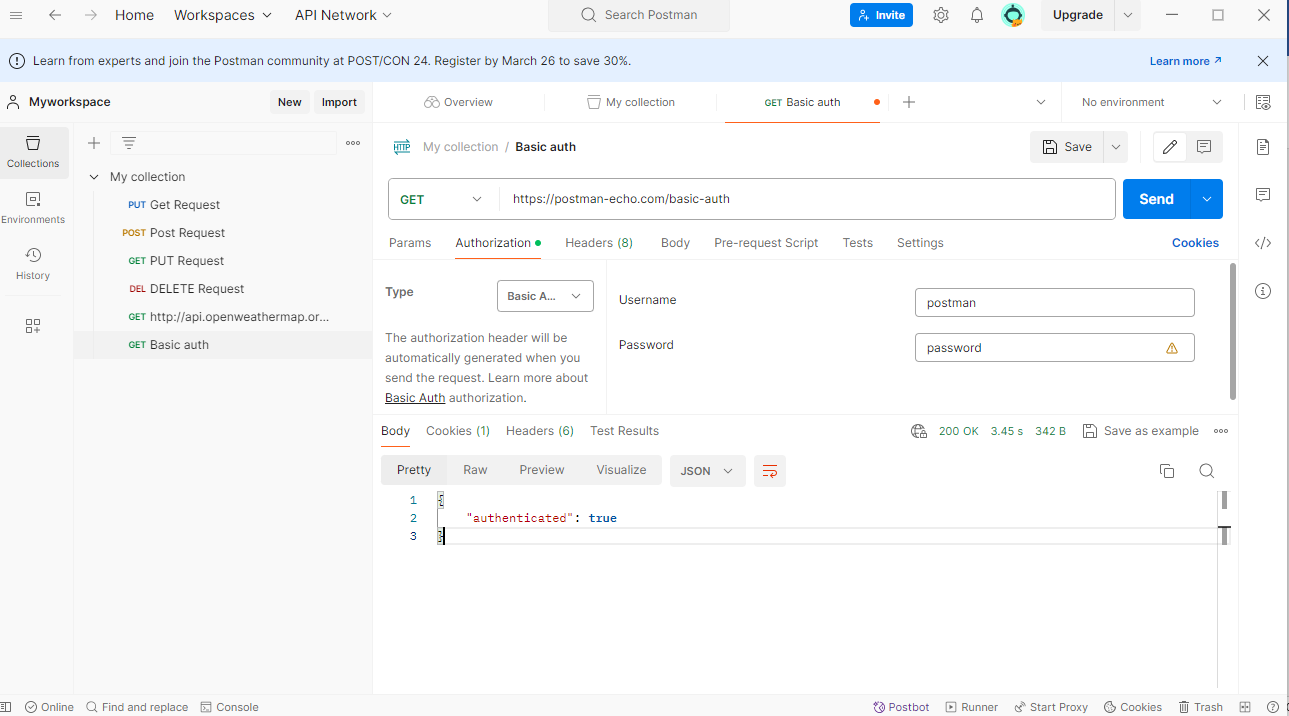
**API testing URL: https://openweathermap.org/**

Check the document whether parameter is considering in header or query param which developed by api developer.

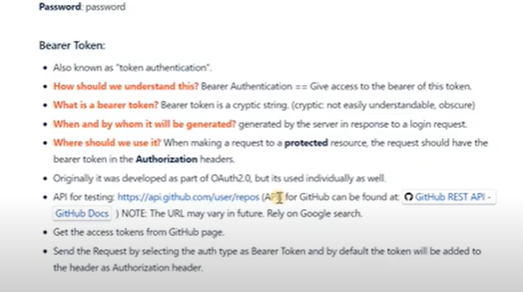


1. **Basic Auth:**

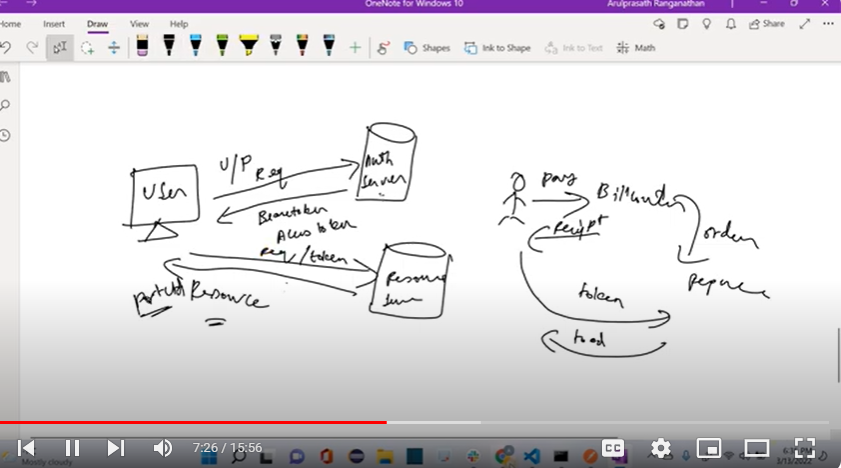
**API testing: https: //postman-echo.com/basic-auth**

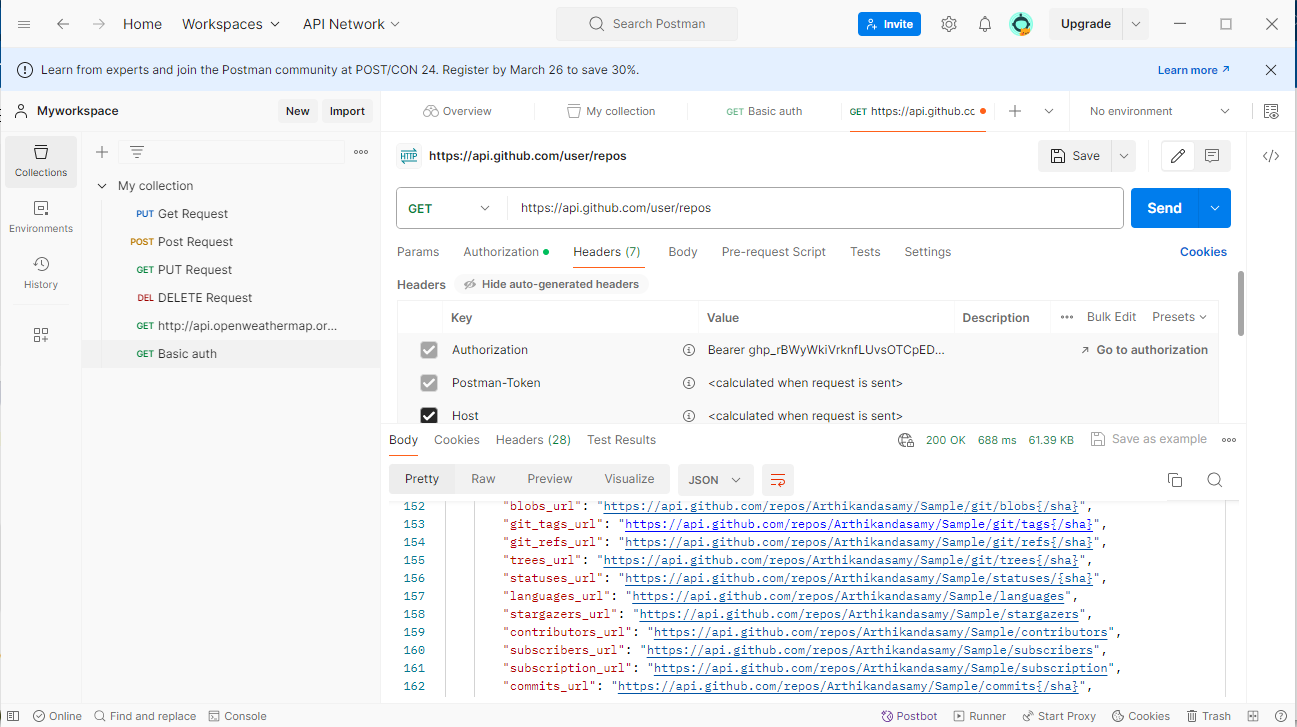


1. **Bearer Token:**



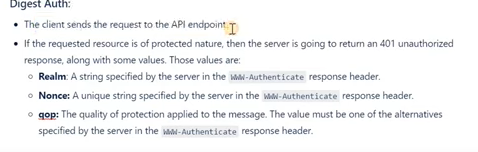
Below screenshot is explanation for bearer token

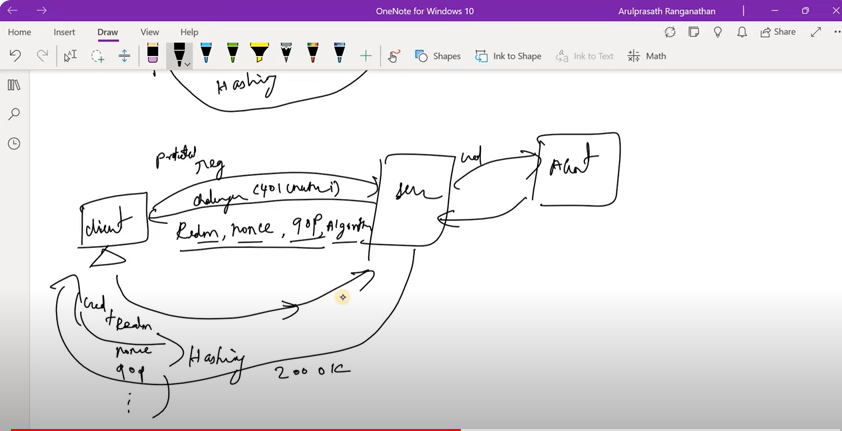




1. **Digest Auth:**

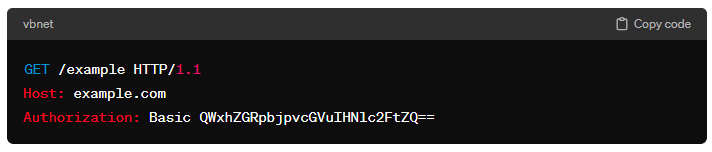
**Api testing url:** [**https://postman-echo.com/digest-auth**](https://postman-echo.com/digest-auth)





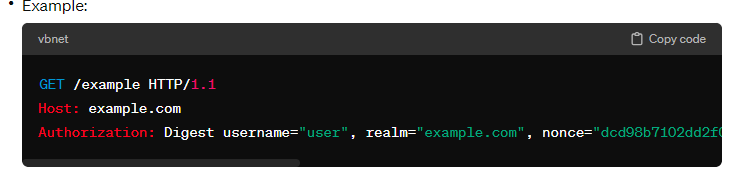
1. **Basic Authentication:**
   * Basic Authentication sends user credentials (username and password) as plaintext in the HTTP request headers.
   * The credentials are base64 encoded but not encrypted, so they can be easily decoded if intercepted.
   * It is a simple authentication mechanism and is widely supported by web servers and browsers.

Example:

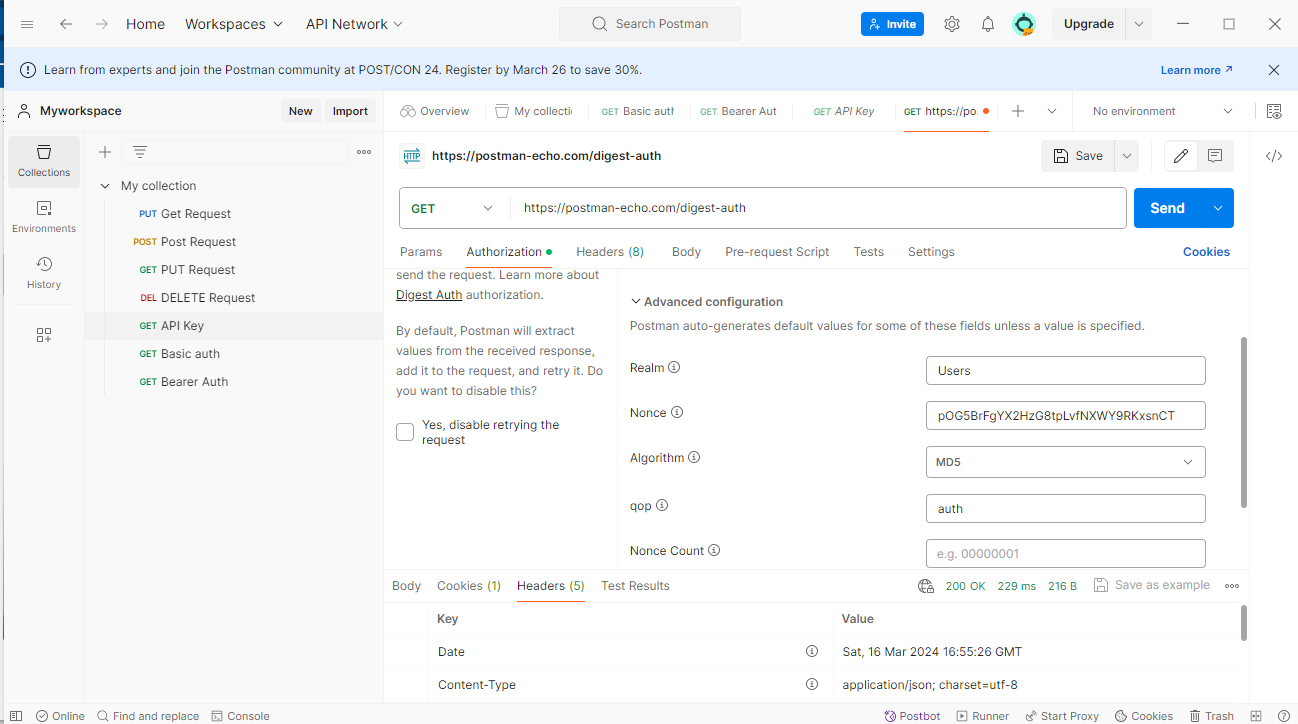


QWxhZGRpbjpvcGVuIHNlc2FtZQ==

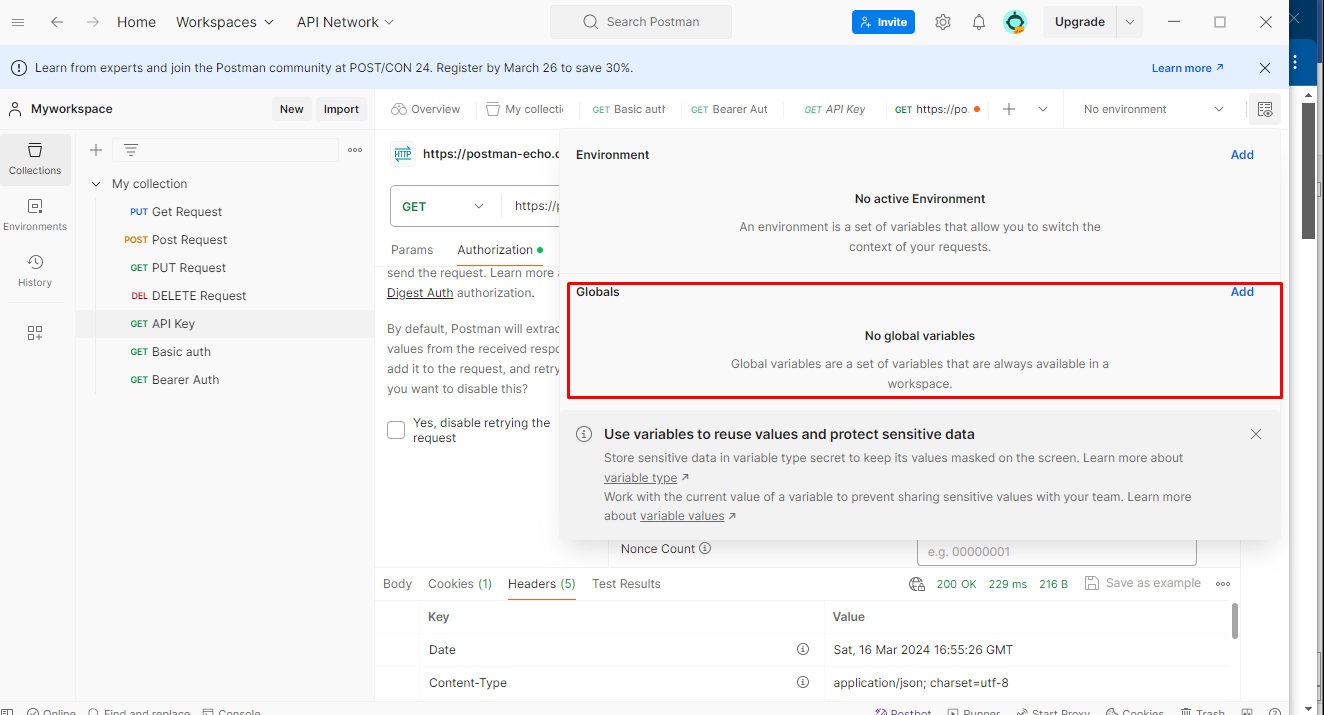
1. **Digest Authentication:**
   * Digest Authentication is more secure than Basic Authentication as it sends hashed credentials rather than plaintext.
   * It uses a challenge-response mechanism where the server sends a challenge containing a nonce (a unique token) to the client, which is then used to calculate a hashed response.
   * The client sends the response along with the username and other required information.
   * Digest Authentication protects against eavesdropping because it doesn't send plaintext passwords over the network.
   * Example:

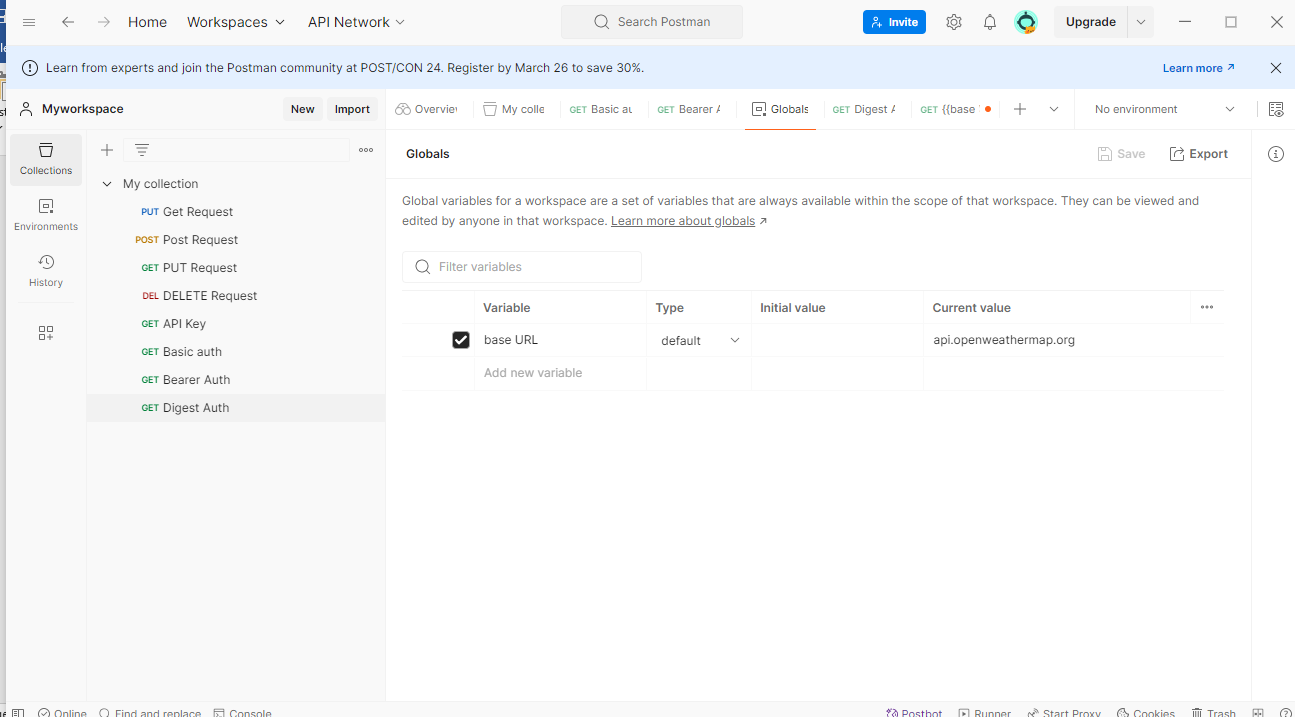
In summary, Basic Authentication is simpler and easier to implement, but it is less secure compared to Digest Authentication, which provides better protection against unauthorized access by using hashed credentials. However, both methods should be used over HTTPS to encrypt the communication and prevent interception of sensitive information.

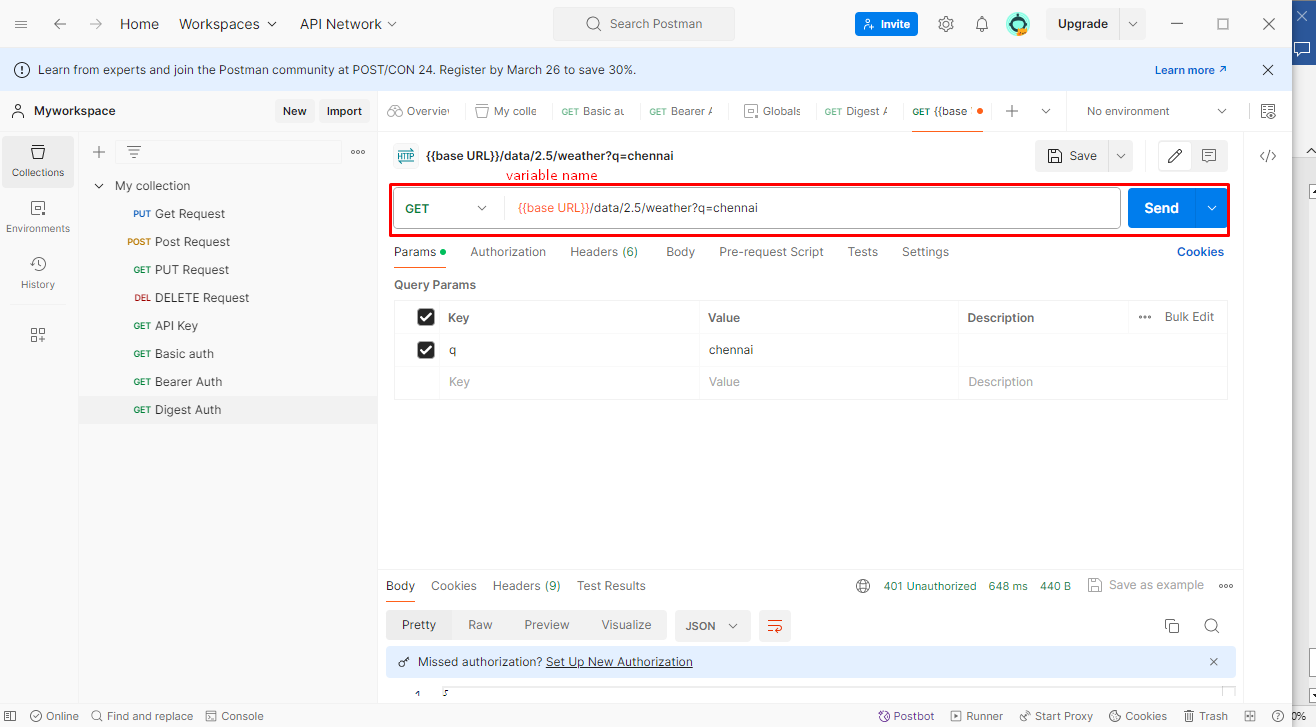
Top of Form



**Global Variable:**







**Environment Variables:**

**API Response Validations:**

**Response Validation:**

**These are the main things to test**

1. Status code
2. Headers
3. Cookies
4. Response Time
5. Response body

**Assertion-validation:**

To add this assertion in postman we have library called ‘pm’ in which it has so many functions. Using functions we can add validation points.

These functions are written using JavaScript.

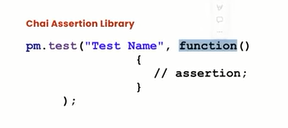
Two different way we can write function:

1.Normal function

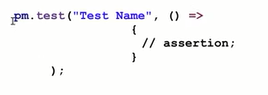
2.Arrow function

**Chai assertion library:**

1.Normal function:



2.Arrow function:



**Response Validation:**

1. **Testing Status code:**

Pm.test(“validate the status code is 200”. Function ()

{

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

}

});

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

Pm.test(“successful POST request”, function()

{

Pm.expect(pm.response.code).to.be.oneof([201,202]);//To check multiple status code

}

});

**Check the status code text:**

Pm.text(“status code name has string”, ()=>

{

Pm.response.to.have.status(“created”);

}});

1. **Testing Headers**

In response headers mostly we check content-type, expires, and other static headers can be validated.

**Check that a response header is present**

Pm.test(“Content-type header is present”, () =>

{

Pm.response.to.have.header(“content-type”);

});

**Test for a response header having a particular value:**

Pm.test(“content-type header is application/json”, () =>)

{

Pm.expect(pm.response.headers.get(‘Content-type’)).to.eql (‘application/json; character=utf-8’);

});

**Testing Cookies**

**Test if a cookie is present in the response**

pm.**test**("Cookie language is present", () **=>**

{

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

});

**Test for a particular cookie value:**

Pm.test(“cookie language has value 1”, () =>

{

Pm.expect(pm.cookies.get(‘language’)).to.eql(‘en-gb’);

});

1. **Testing Response Time:**

**Pm.test(“Response time is less than 50ms ”, () =>**

**{**

**Pm.expect(pm.response.responsetime).to.be.below(50);**

**});**

1. **Response Body:**

**Test the type of any part of the response:**

**{**

**“id”: 1,**

**“name”: “Jhon”,**

**“location”: “india”,**

**“phone”: “8790123478”,**

**“courses”:[**

**“Java”,**

**“Selenium”,**

**“Python”**

**]**

**}**

**Const jsonData = pm.response.json();**

**Pm.test(“Test data type of the resposne”, () =>**

**{**

**Pm.expect(jsonData).to.be.an(“object”);**

**Pm.expect(jsonData.name).to.be.a(“string”);**

**Pm.expect(jsonData.id).to.be.a(“number”);**

**Pm.expect(jsonData.courses).to.be.an(“array”)**

**)**

1. **Asserting array properties:**

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

**{**

**“id”: 1,**

**“name”: “Jhon”,**

**“location”: “india”,**

**“phone”: “8790123478”,**

**“courses”: [**

**“Java”,**

**“Selenium”,**

**“Python”**

**]**

**}**

**Pm.test(“Test that array properties”, () =>**

**{**

**//course include “java”**

**Pm.test(json.courses).to.include(“java”);**

**Pm.test(json.courses).to.have.members([“java”, “selenium”]);**

**});**

1. **Validating JSON fields and response**

**{**

**“id”: 1,**

**“name”: “Jhon”,**

**“location”: “India”,**

**“phone”: “8790123478”,**

**“courses”: [**

**“Java”,**

**“Selenium”,**

**“Python”**

**]**

**}**

**Pm.test(“Value of location field in india”, () =>**

**{**

**Var jsondata = pm. Response.json();**

**Pm.expect(jsonData.id).to.eql(1);**

**Pm.expect(jsonData.name).to.eql(“Jhon”);**

**Pm.expect(jsonData.location).to.eql(“India”);**

**Pm.expect(jsonData.phone).to.eql(“8790123478”);**

**Pm.expect(jsonData.courses[0]).to.eql(“Java”);**

**Pm.expect(jsonData.courses[1]).to.eql(“Selenium”);**

**});**

1. **Validating JSON Schema Validation:**