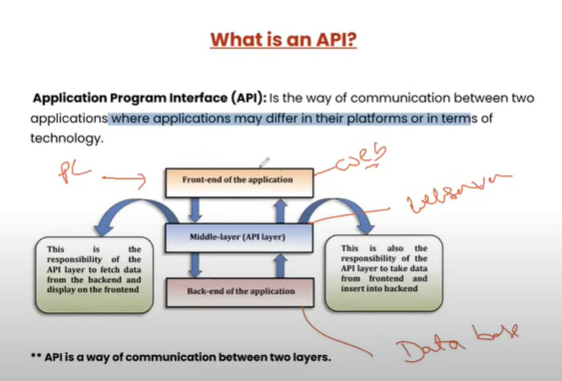
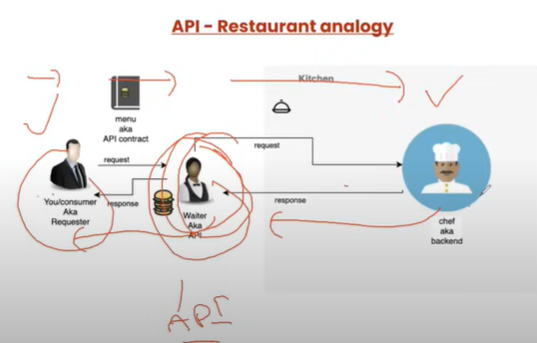
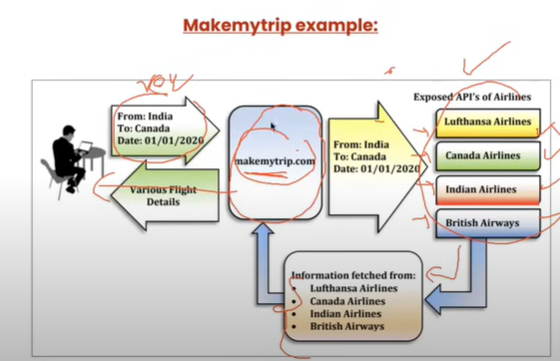
**API Testing: Application Program Interface**

It is way of communication between 2 applications where application may differ in their platforms or in terms of technology.









**Types of API:**

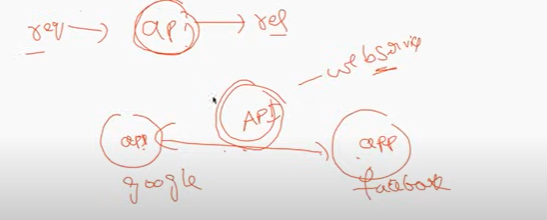
There re 2 types of API’s.

1. Simple Object Access Protocol (SOAP).
2. REST (Representation State Transfer).

Both are the web services.

SOAP => used by old applications.

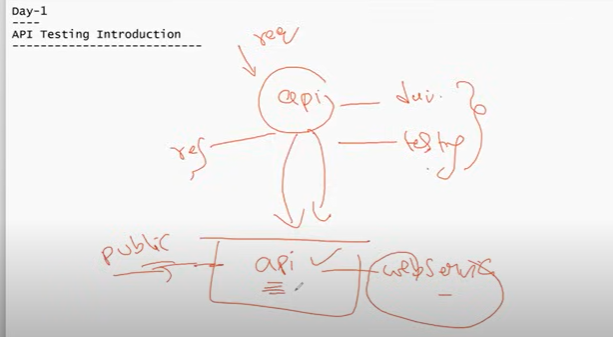
REST => used by new/current applications.



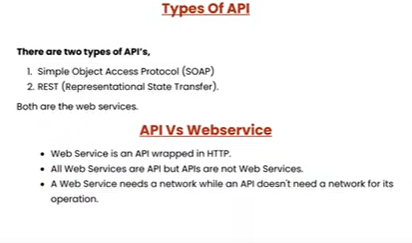
All Web Services are API. But all API are not Web Service.

When API is put in internet then it is called as Web Service.

For development & testing we use Api. And once its available to public over internet its called Webservice.



**Important: API Vs Webservice**



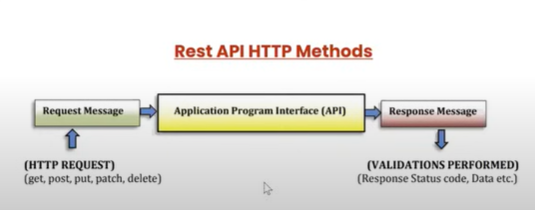
**Rest API methods:** http request

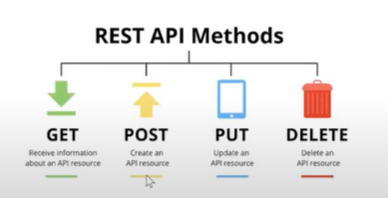
Get: we get data from the server.

Post: we are sending data to server to store.

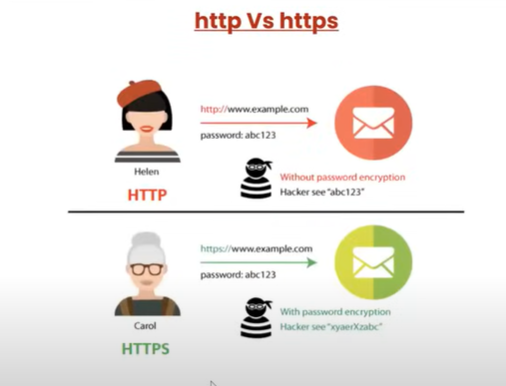
Put: we can create/edit/update data in the server.

Delete: we delete the data from server.

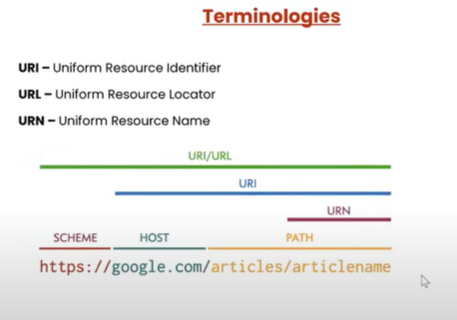




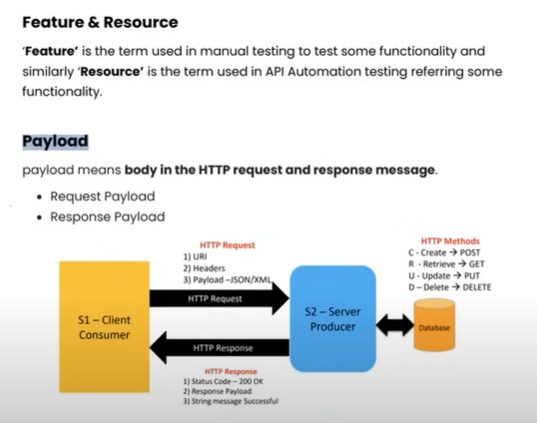
HTTP Vs HTTPS



Terminology:

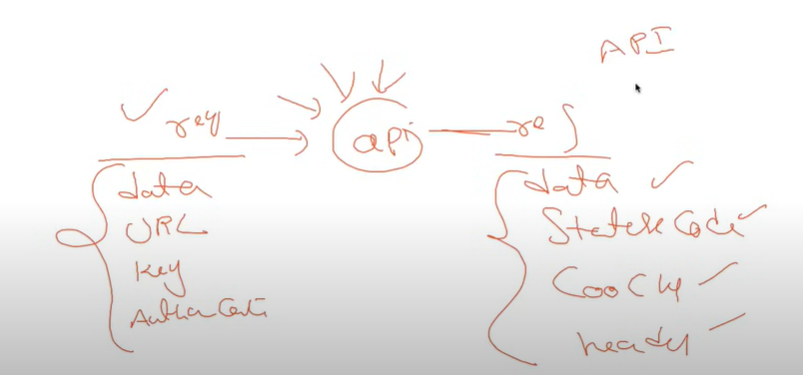


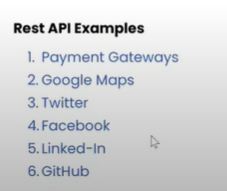
Payload:



Ex: <https://reqres.in/>

Ex Get: <https://reqres.in/api/users?page=2> => <https://reqres.in/api/users?page=2>





Postman – API testing

Desktop/web

Workspace: area where we maintain files and saved.

Workspace – create workspace, rename, delete.

Creating collection = contains number of folders and http requests. Create, rename, delete, run the collection.

We can create any number of collections under workspace.

Request ----🡪 API -------🡪 Response

http Request:

Get => retrieve the resource from database.

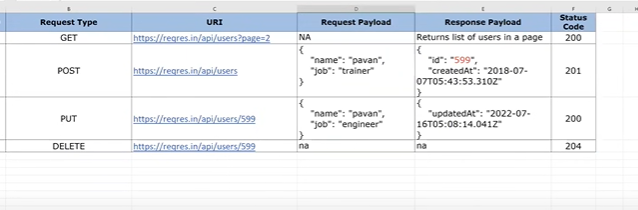
Post => create resource on database.

Put => update existing resource on database.

Patch => update partial details of resource.

Delete => delete existing resource from database.

Sample APIs: <https://reqres.in/>





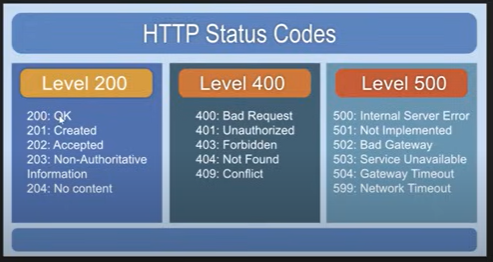
Validations:

1. Status code
2. Time
3. Size data
4. Response body(json/xml)
5. Cookies
6. Headers

HTTP Status code:

3 levels

1. 200
2. 400
3. 500



**How to create own API: JSON**

Create our own API’s

Steps:

1. NodeJS
2. Npm-node package manager

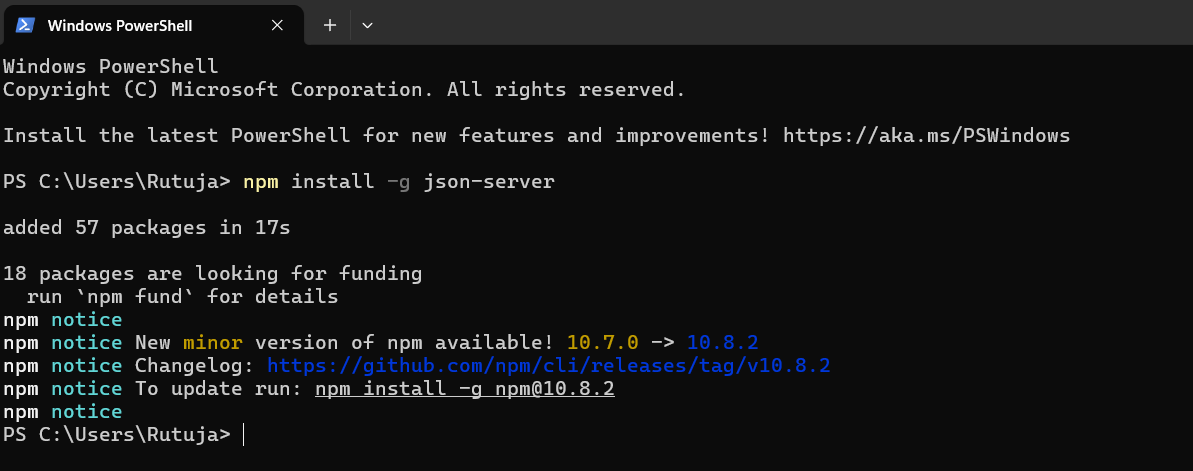
node - -version

npm --version

1. Json-server
2. Install json-server:

Run below command in cmd/terminal

npm install -g json-server



1. Create students.json file with following data.

{

    "students":[

        {

            "id": 1,

            "name": "John",

            "location": "India",

            "phone": "1234567890",

            "courses": [

                "Java",

                "Selenium"

            ]

        },

        {

            "id": 2,

            "name": "Kim",

            "location": "US",

            "phone": "2345678901",

            "courses": [

                "Python",

                "Appium"

            ]

        },

        {

            "id": 3,

            "name": "Smith",

            "location": "Canada",

            "phone": "3456789012",

            "courses": [

                "C#",

                "RestAPI"

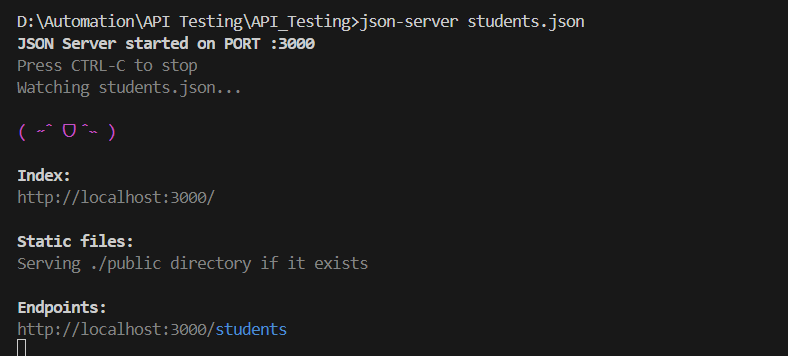
            ]

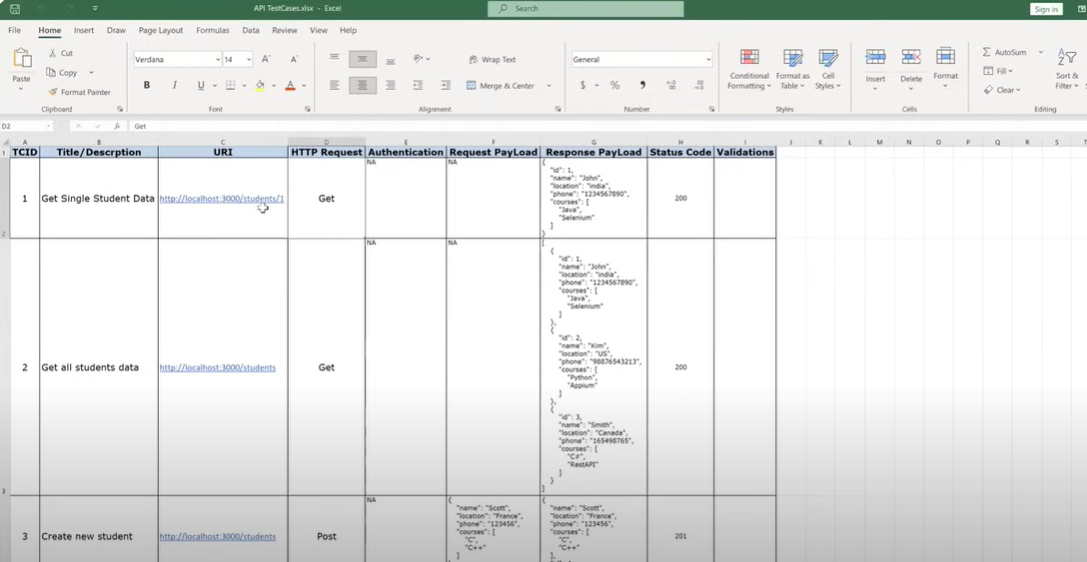
        }

]

}

1. Run using command ‘json-server students.json’



1. <http://localhost:3000/students> 
2. 

**JSON: Java Script Object Notation**

Key value pair

Key: value

**JSON Data Types:**

1. Number
2. String
3. Boolean
4. Null
5. Object
6. Array

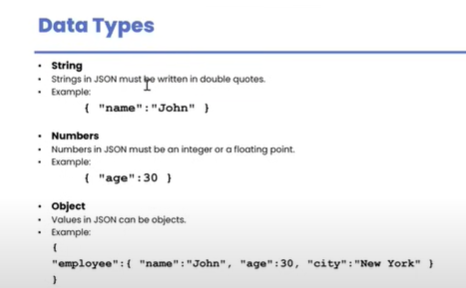
{

“name”: “John”

}

**Data Types:**

****



Key is always included in “ “ quotations.

{

“firstname”:”John”,

“secondname”:null,

“age”:30,

“phone”:[12345,67890],

“status”:true

}

Eg: Student data

Student – sid, sname, grad

{

“students”:[

{

“sid”=101,

“sname”=”John”,

“grad”=”A”

},

{

“sid”=102,

“sname”=”Mark”,

“grad”=”B”

}

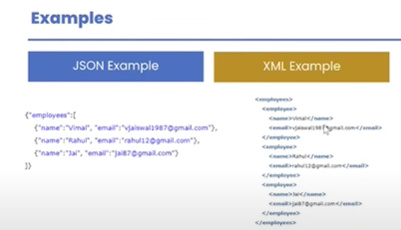
]

}

JSON Path:

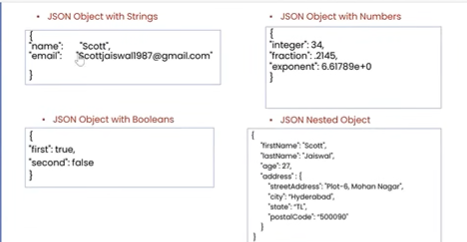
students[0].sname --🡪 John

students[1].sid ---🡪102



**JSON Object & JSON Array:**

**JSON Object:**

****

****

****

**JSON Path Finder:** [**https://jsonpathfinder.com/**](https://jsonpathfinder.com/)

**JSON Path Verify:** [**https://jsonpath.com/**](https://jsonpath.com/)

**API Response Validation | Different types of Assertions:**

**Response Validation:**

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

**Assertion – validation**

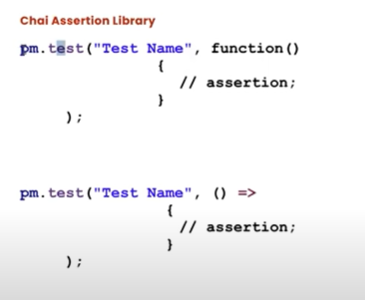
**Postman – library**

Functions: written in JavaScript

Function types/can be written in 2 ways:

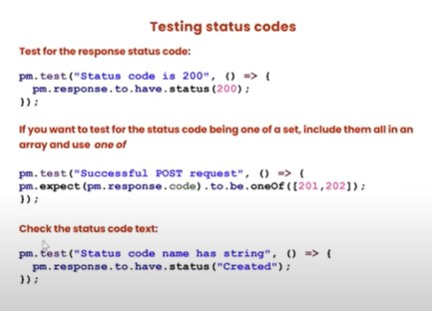
1. Normal function: written using normal keyword i.e. function().
2. Arrow function: written using arrow i.e. () =>.

Library/Framework: Chai



Here pm = postman.

**Testing Status Code:**



**Syntax:**

pm.test(“Test Name”, ()=>{

//assertion;

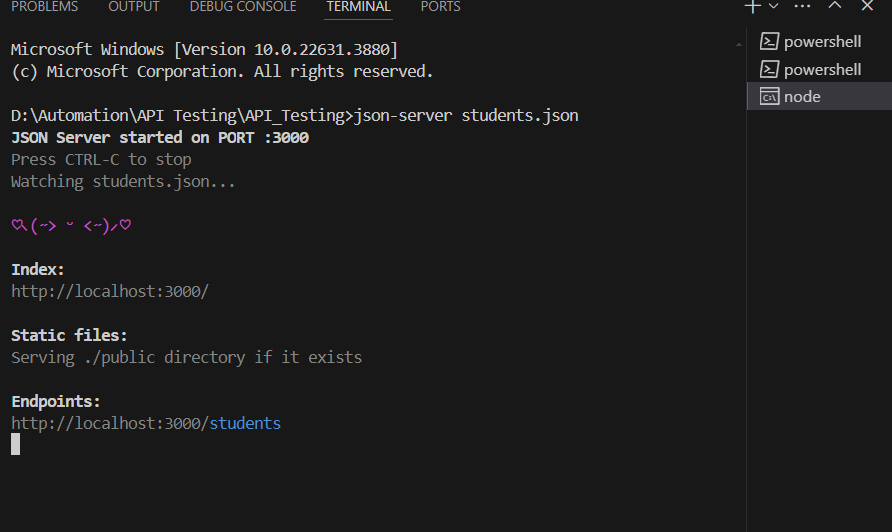
});

Ex:

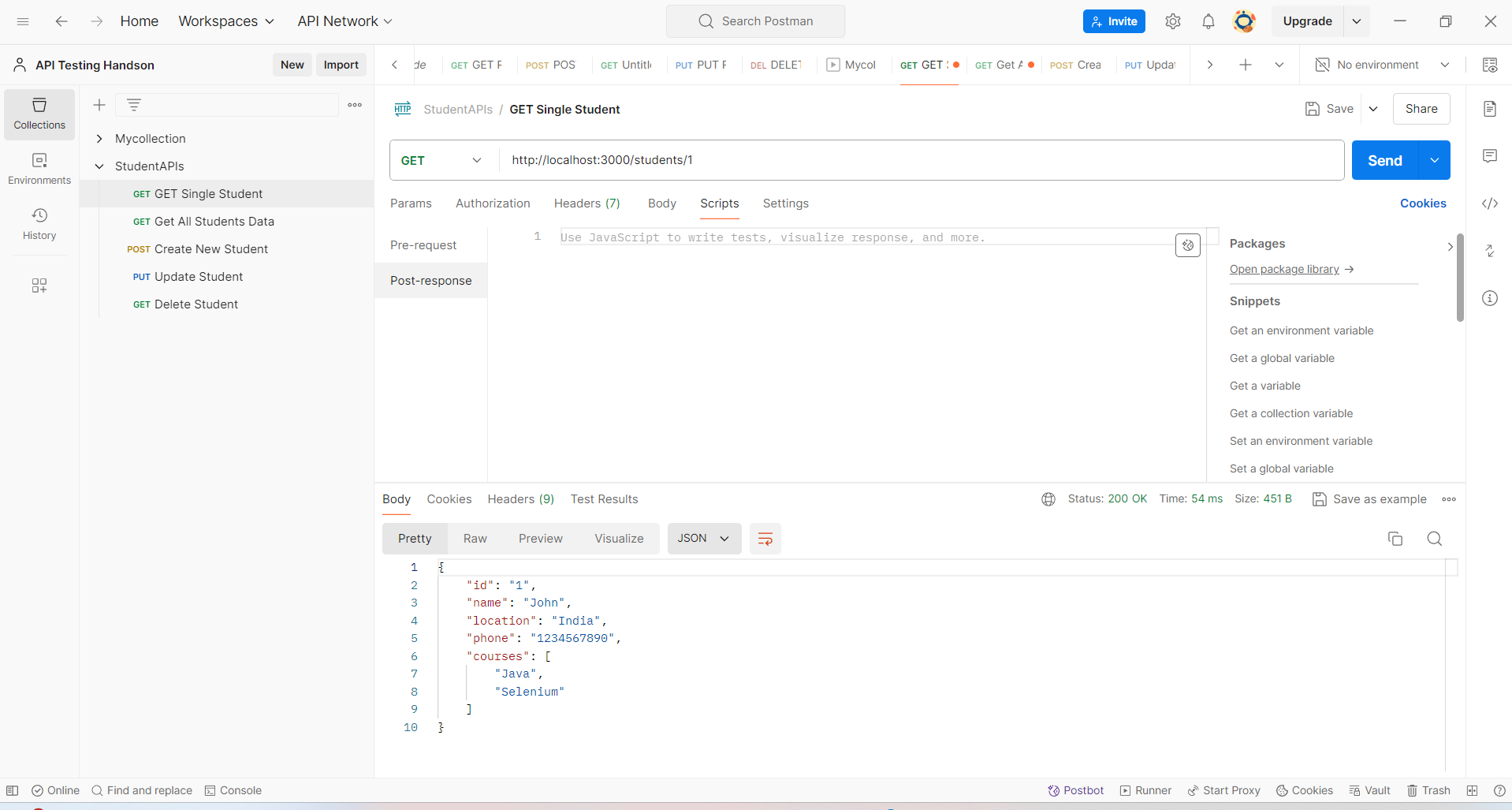
pm.test(“Status code is 200”, () => {

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

});



Postman:



pm.**test**("Status code is 200", () **=>** {

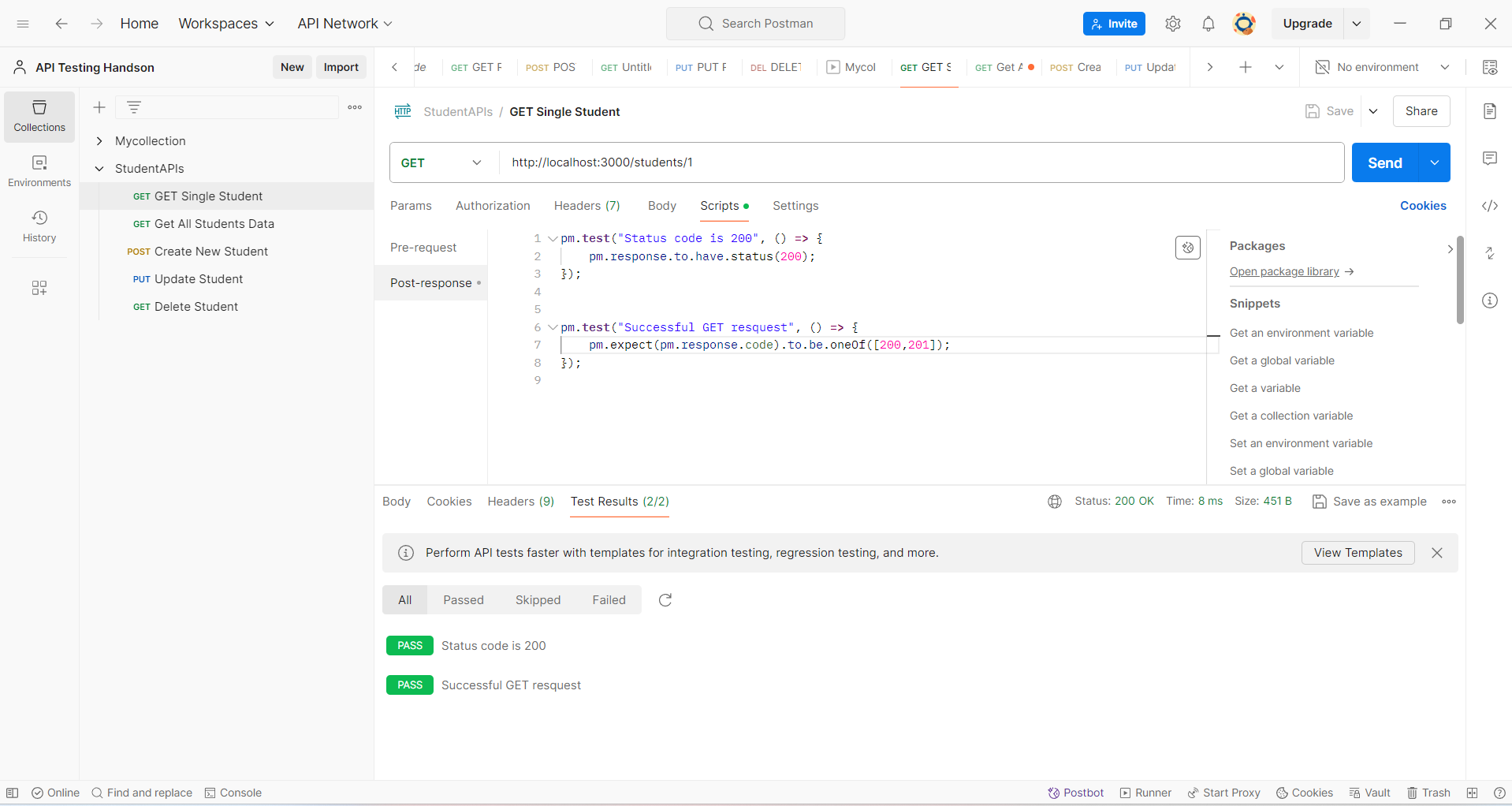
    pm.response.to.have.status(200);

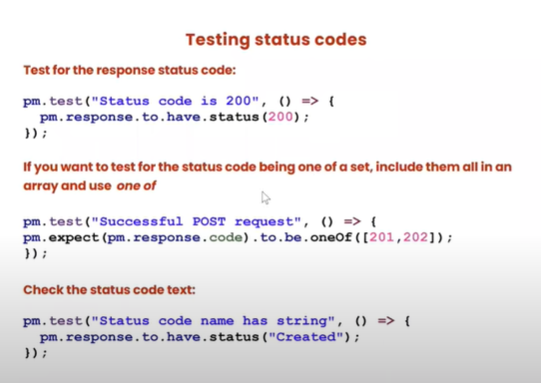
});

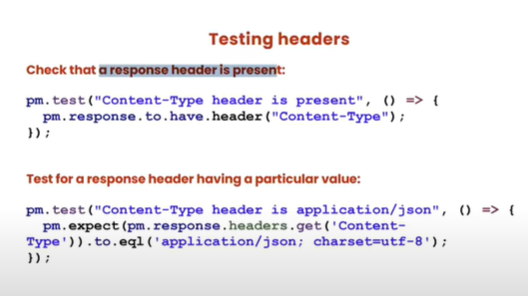
pm.**test**("Successful GET resquest", () **=>** {

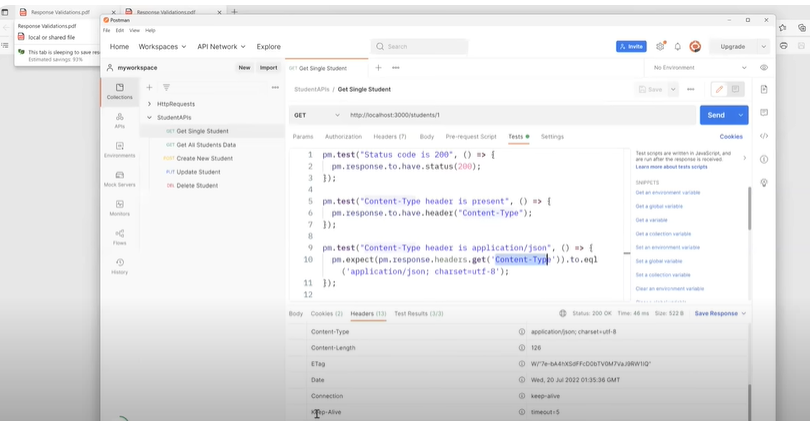
    pm.expect(pm.response.code).to.be.oneOf([200,201]);

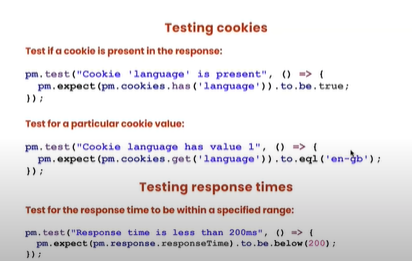
});

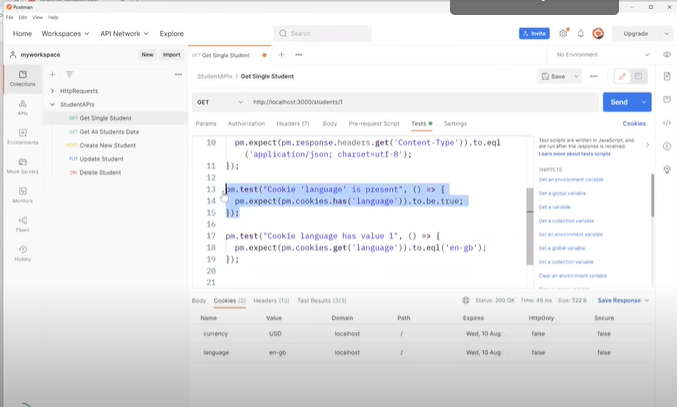


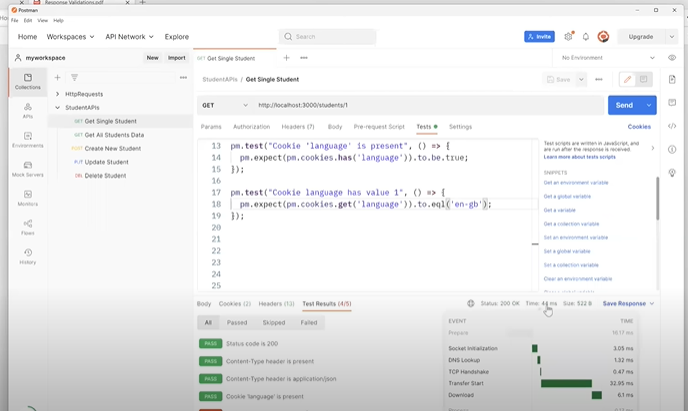


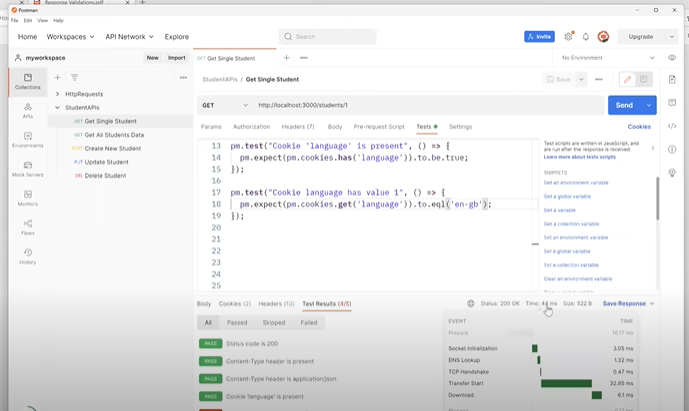




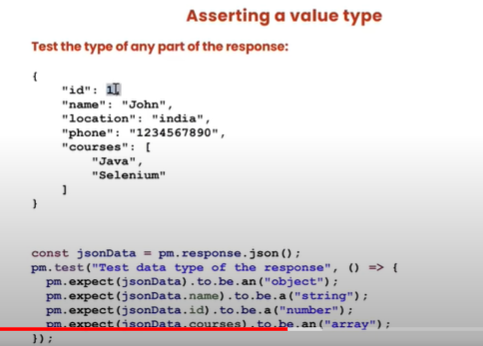


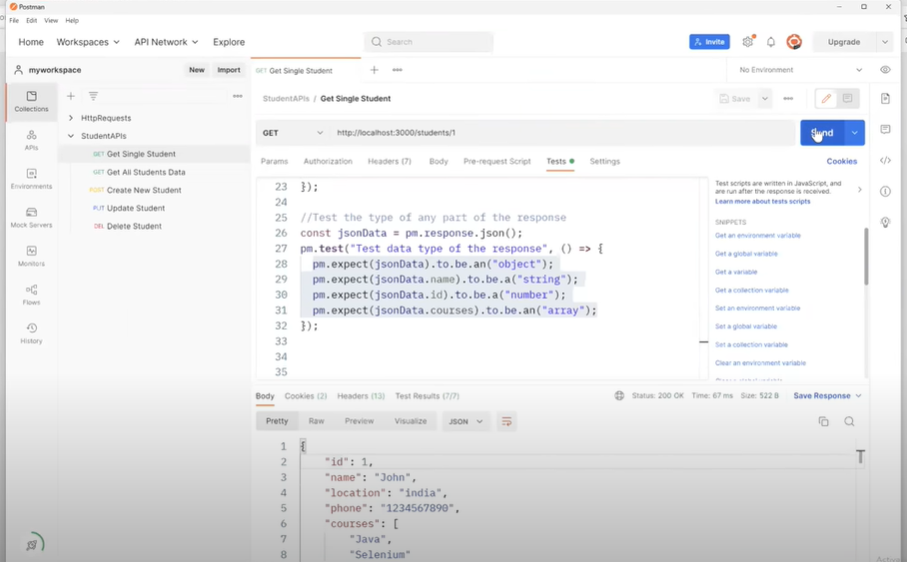






**Response Body:**

****

****

//Test the type of any part of the response

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

pm.**test**("Test data type of the response", () **=>** {

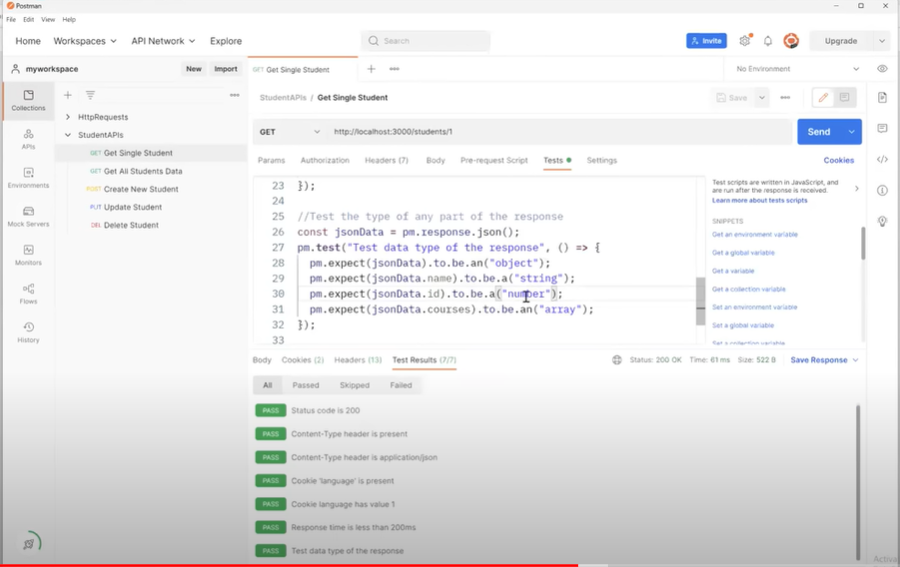
    pm.expect(jsonData).to.be.an("object");

    pm.expect(jsonData.name).to.be.a("string");

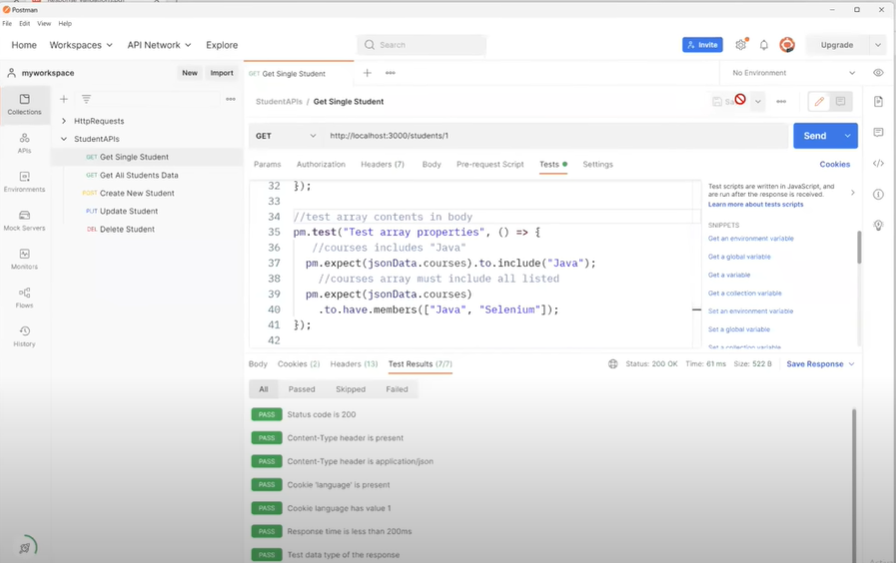
    pm.expect(jsonData.id).to.be.a("number");

    pm.expect(jsonData.courses).to.be.an("array");

});







//Test array contents in body

pm.**test**("Test array properties", () **=>** {

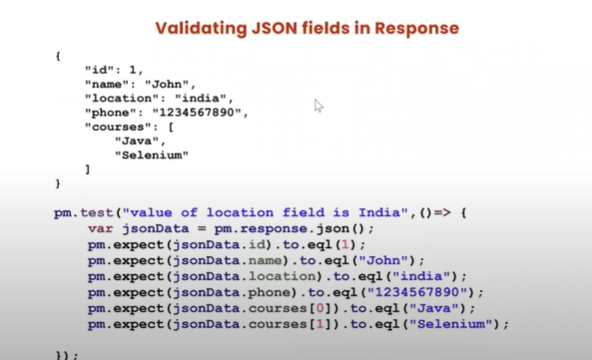
    //course includer "Java"

    pm.expect(jsonData.courses).to.include("Java");

    //courses array must include all listed

    pm.expect(jsonData.courses).to.have.members(["Java", "Selenium"]);

});



//Validating JSON fields in Response

pm.**test**("value of fields in response", () **=>** {

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

    pm.expect(jsonData.id).to.eql(1);

    pm.expect(jsonData.name).to.eql("John");

    pm.expect(jsonData.location).to.eql("India");

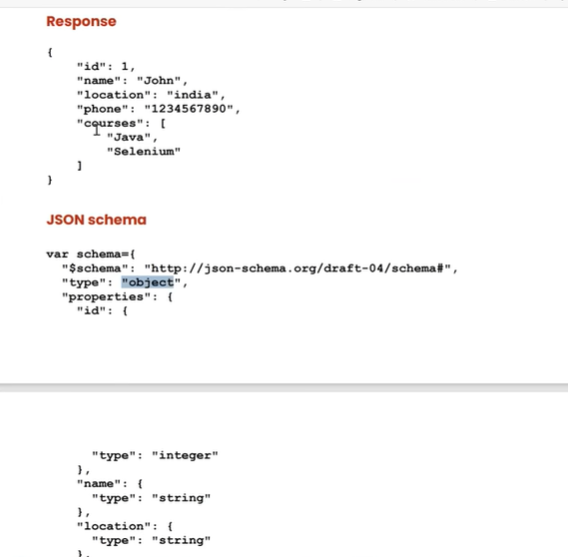
    pm.expect(jsonData.phone).to.eql("1234567890");

    pm.expect(jsonData.courses[0]).to.eql("Java");

});



**JSON Schema**



JSON Schema: <https://www.liquid-technologies.com/online-json-to-schema-converter>

{

"id": 1,

"name": "John",

"location": "India",

"phone": "1234567890",

"courses": [

"Java",

"Selenium"

]

}

Generate Schema:

JSON Schema:

{

"$schema": "http://json-schema.org/draft-04/schema#",

"type": "object",

"properties": {

"id": {

"type": "integer"

},

"name": {

"type": "string"

},

"location": {

"type": "string"

},

"phone": {

"type": "string"

},

"courses": {

"type": "array",

"items": [

{

"type": "string"

},

{

"type": "string"

}

]

}

},

"required": [

"id",

"name",

"location",

"phone",

"courses"

]

}



Ex:

//JSON Schema Validation

var schema **=** {

  "$schema": "http://json-schema.org/draft-04/schema#",

  "type": "object",

  "properties": {

    "id": {

      "type": "string"

    },

    "name": {

      "type": "string"

    },

    "location": {

      "type": "string"

    },

    "phone": {

      "type": "string"

    },

    "courses": {

      "type": "array",

      "items": [

        {

          "type": "string"

        },

        {

          "type": "string"

        }

      ]

    }

  },

  "required": [

    "id",

    "name",

    "location",

    "phone",

    "courses"

  ]

}

pm.**test**('Schema is a valid', **function**() {

    pm.expect(tv4.validate(jsonData, schema)).to.be.true;

});

**Example complete operations:**

pm.**test**("Status code is 200", () **=>** {

    pm.response.to.have.status(200);

});

pm.**test**("Successful GET resquest", () **=>** {

    pm.expect(pm.response.code).to.be.oneOf([200,201]);

});

pm.**test**("Content-Type header is present", () **=>** {

    pm.response.to.have.header("Content-Type");

});

pm.**test**("Content-Type header is application/json", () **=>** {

    pm.expect(pm.response.headers.**get**("Content-Type")).to.eql('application/json');

    // pm.expect(pm.response.headers.get("Content-Type")).to.eql('application/json; charset=utf-8');

});

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

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

});

pm.**test**("Cookie language has value 1", () **=>** {

    pm.expect(pm.cookies.**get**('language').to.eql('en-gb'));

});

pm.**test**("Response time is less than 50ms", () **=>** {

    pm.expect(pm.response.responseTime).to.be.below(50);

});

//Test the type of any part of the response

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

pm.**test**("Test data type of the response", () **=>** {

    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");

});

//Test array contents in body

pm.**test**("Test array properties", () **=>** {

    //course includer "Java"

    pm.expect(jsonData.courses).to.include("Java");

    //courses array must include all listed

    pm.expect(jsonData.courses).to.have.members(["Java", "Selenium"]);

});

//Validating JSON fields in Response

pm.**test**("value of fields in response", () **=>** {

   // var jsonData = pm.response.json();

    pm.expect(jsonData.id).to.eql(1);

    pm.expect(jsonData.name).to.eql("John");

    pm.expect(jsonData.location).to.eql("India");

    pm.expect(jsonData.phone).to.eql("1234567890");

    pm.expect(jsonData.courses[0]).to.eql("Java");

});

//JSON Schema Validation

var schema **=** {

  "$schema": "http://json-schema.org/draft-04/schema#",

  "type": "object",

  "properties": {

    "id": {

      "type": "string"

    },

    "name": {

      "type": "string"

    },

    "location": {

      "type": "string"

    },

    "phone": {

      "type": "string"

    },

    "courses": {

      "type": "array",

      "items": [

        {

          "type": "string"

        },

        {

          "type": "string"

        }

      ]

    }

  },

  "required": [

    "id",

    "name",

    "location",

    "phone",

    "courses"

  ]

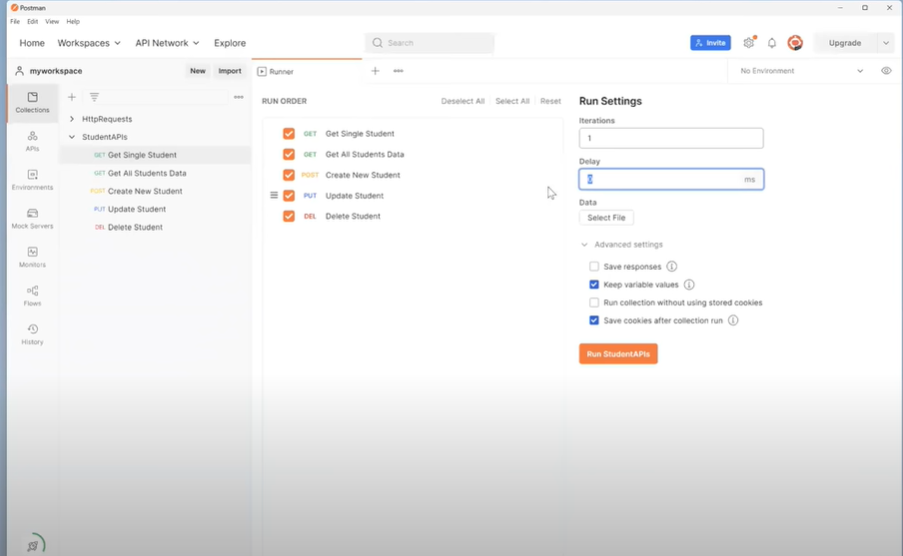
}

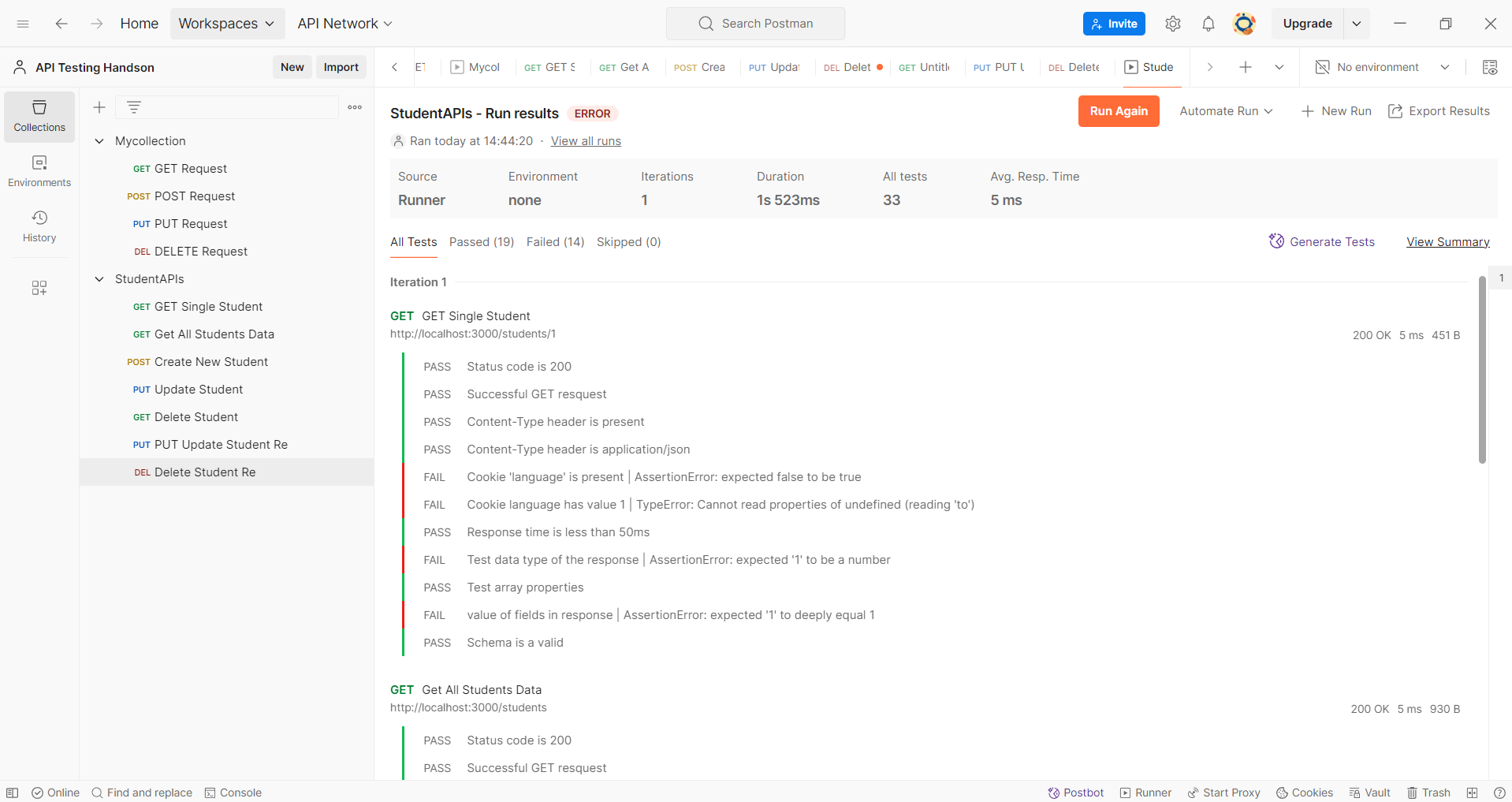
pm.**test**('Schema is a valid', **function**() {

    pm.expect(tv4.validate(jsonData, schema)).to.be.true;

});

**Run Collection:**





**Scripts and Types of variables**

Scripts

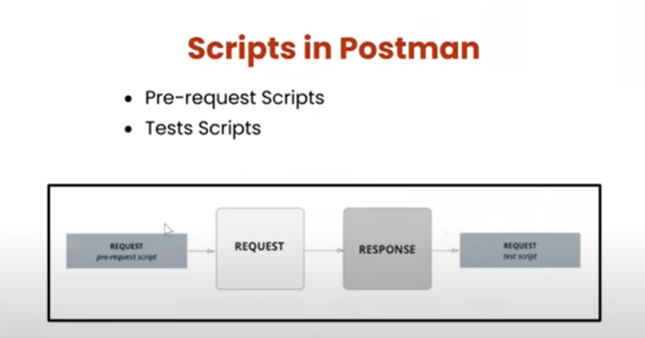
Pre-request scripts

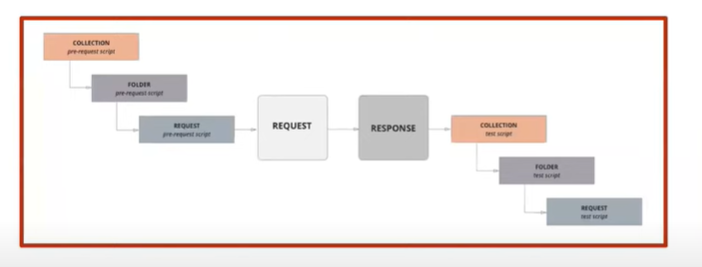
Tests

Pre-requestScript --🡪 Request --🡪 Response --🡪 Tests

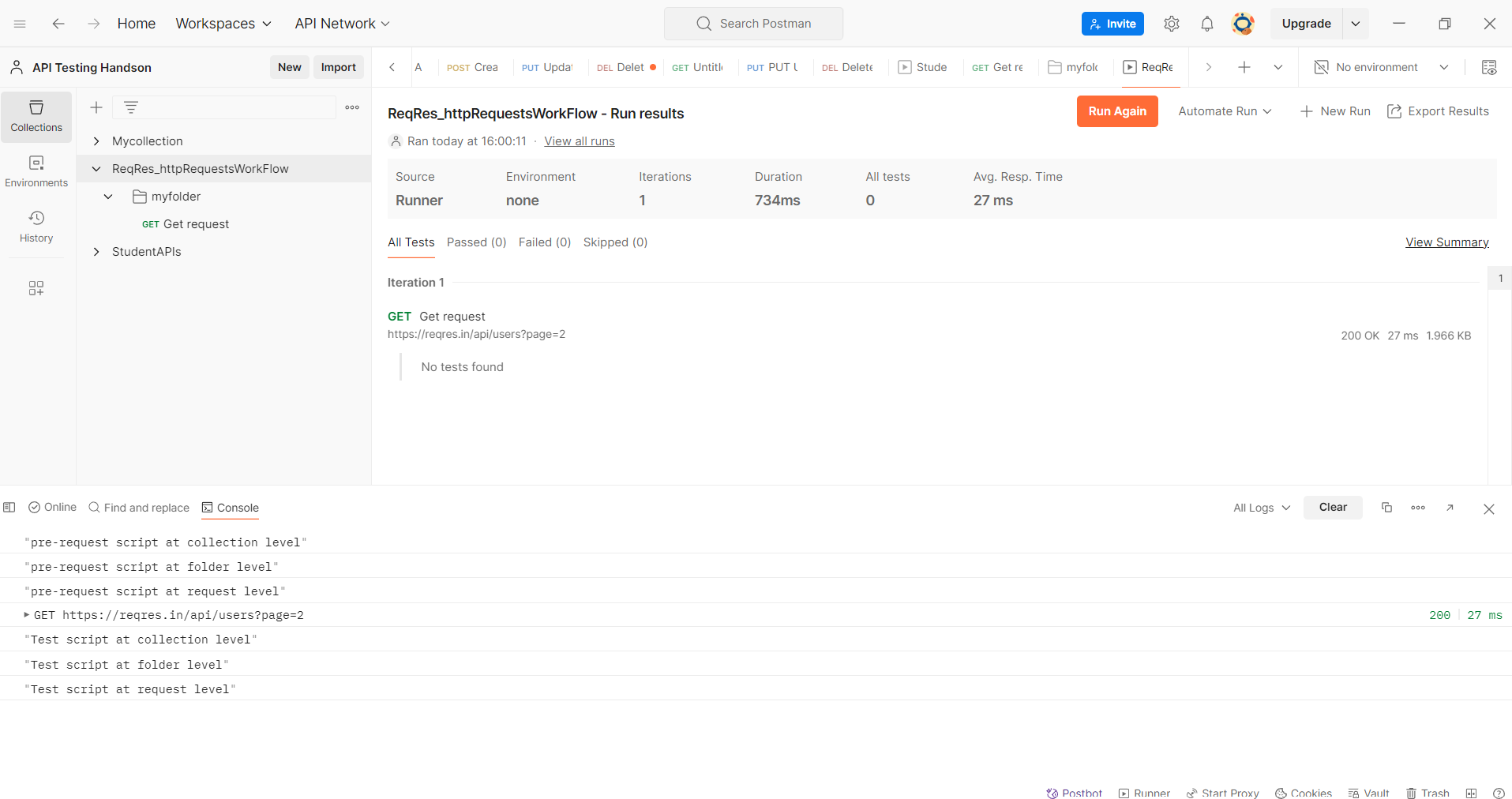
3 levels:

1. Collection
2. Folder
3. Request





Request response order:



**Variables:**

**Scope:**

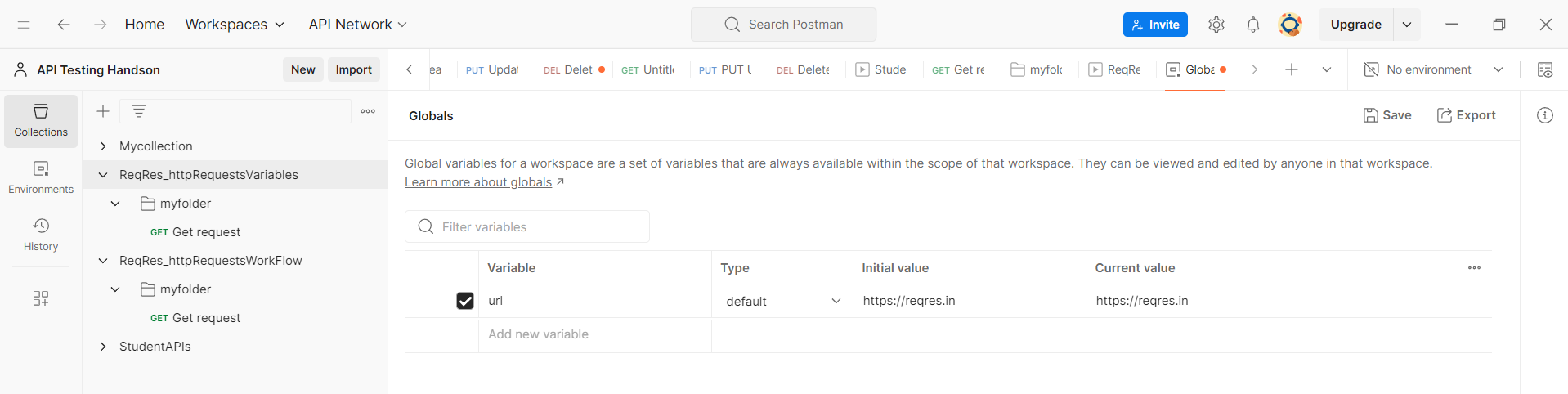
**Workspace -🡪 Collections -🡪 Requests**

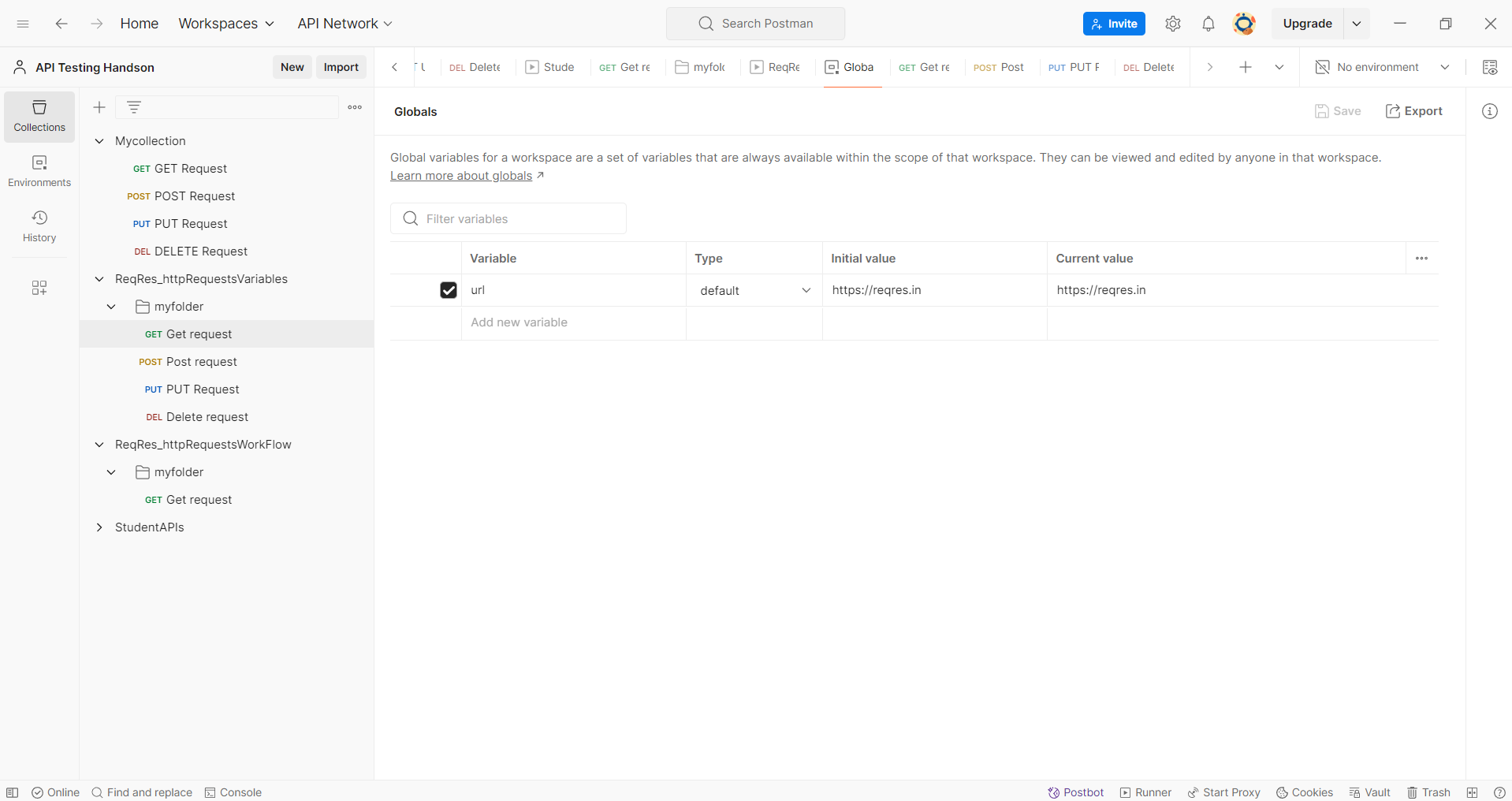
Types of variables as per Scope of access of variables:

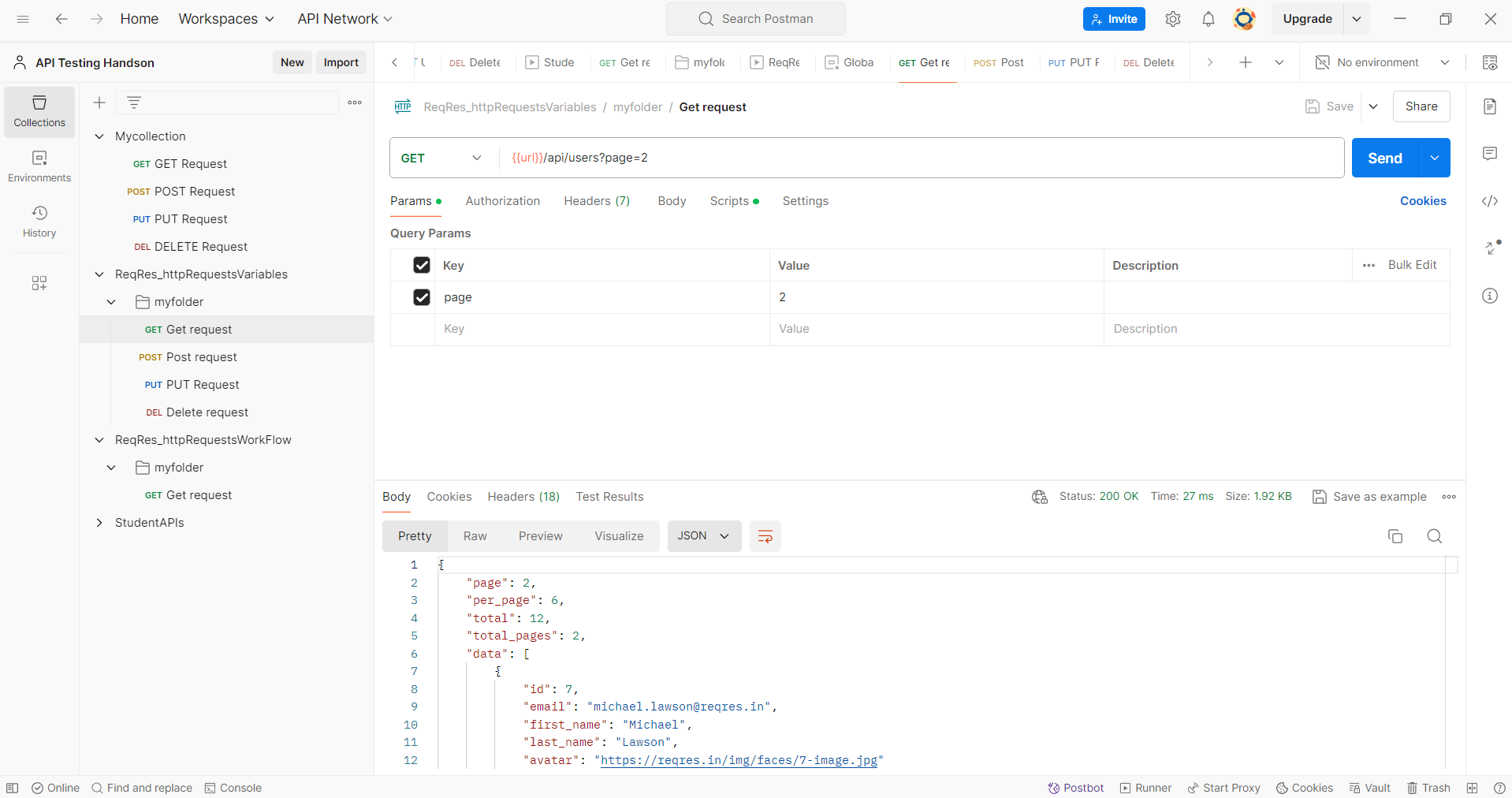
1. Global => accessible in workspace.
2. Collection => accessible within collection.
3. Environment => accessible in all collections, but we need to switch to environment.
4. Local => accessible only within request(specific to request). Declared inside Pre-request Script i.e. Request -> Scripts -> Pre-request.
5. Data => external files csv/text.

Referring variable: {{variable}}

Global variables:

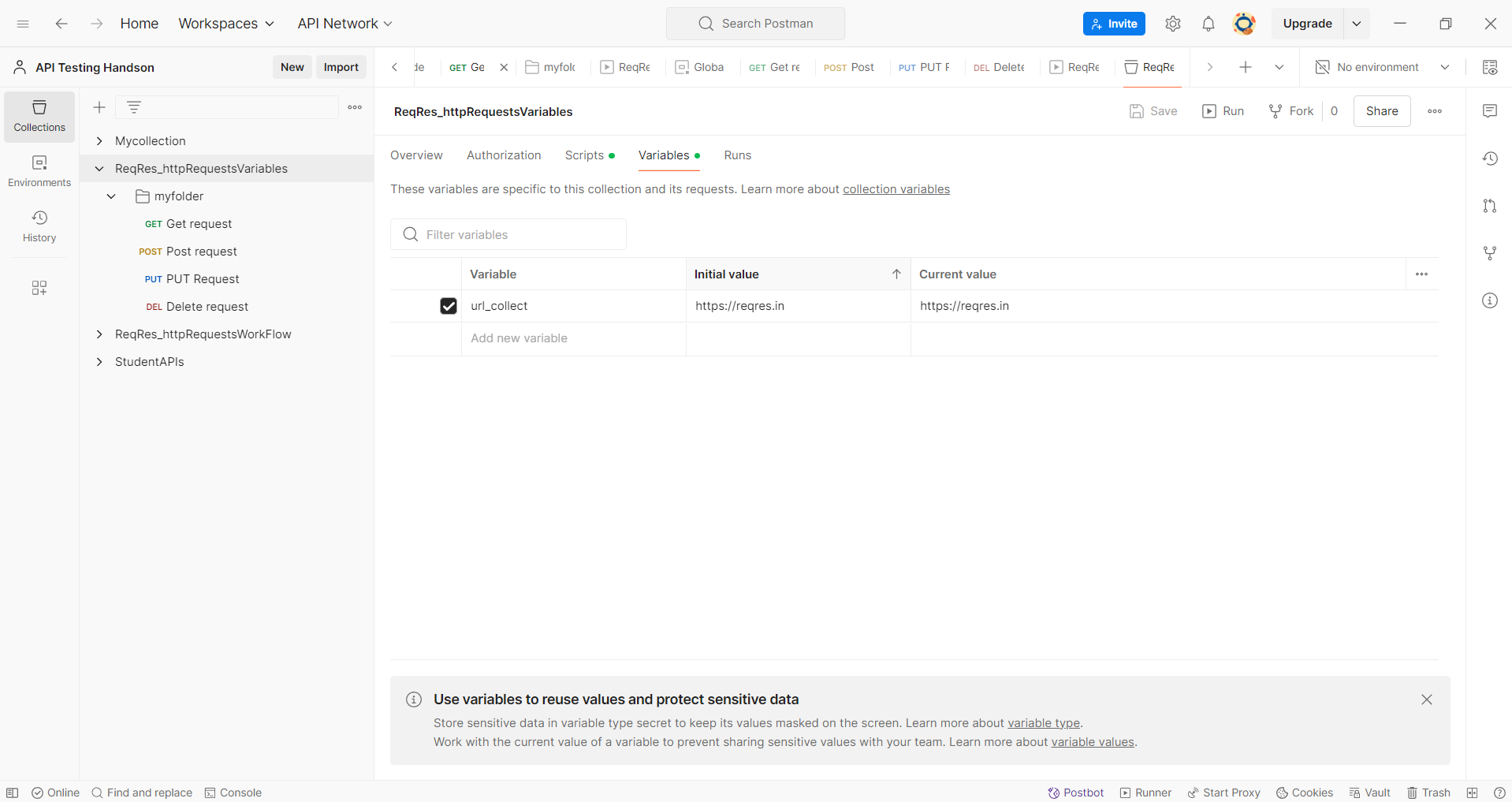


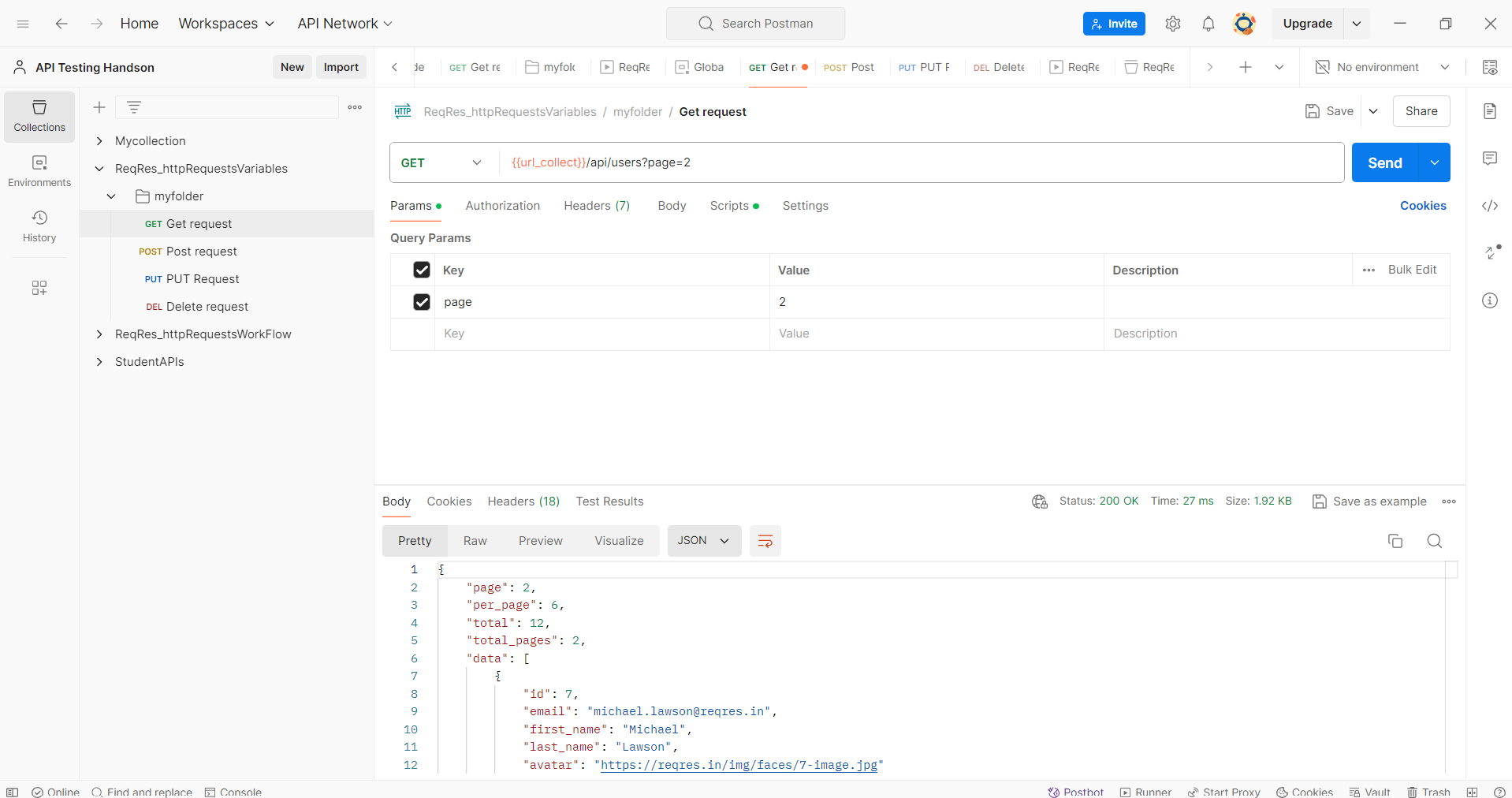




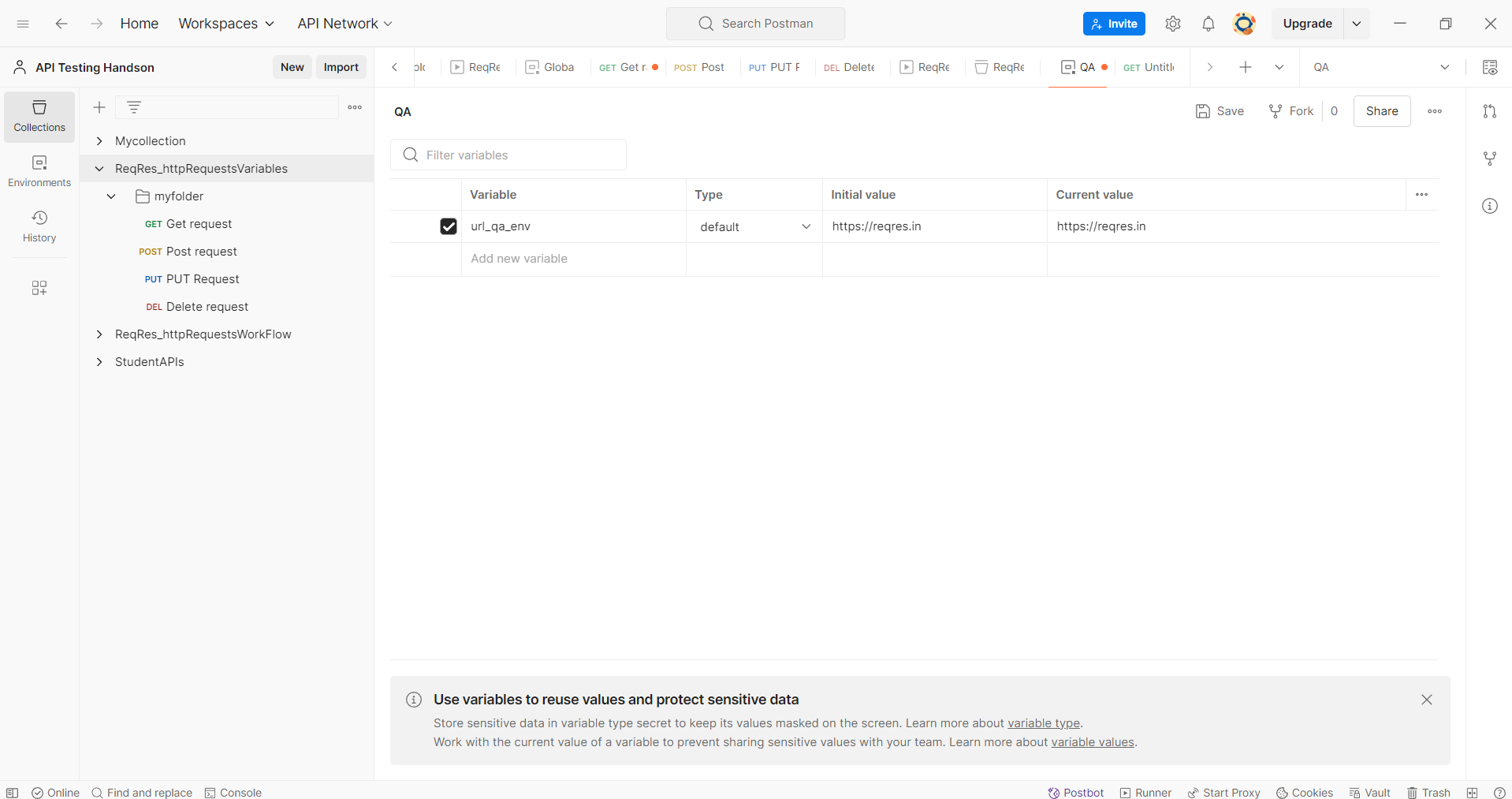


Collection variable:

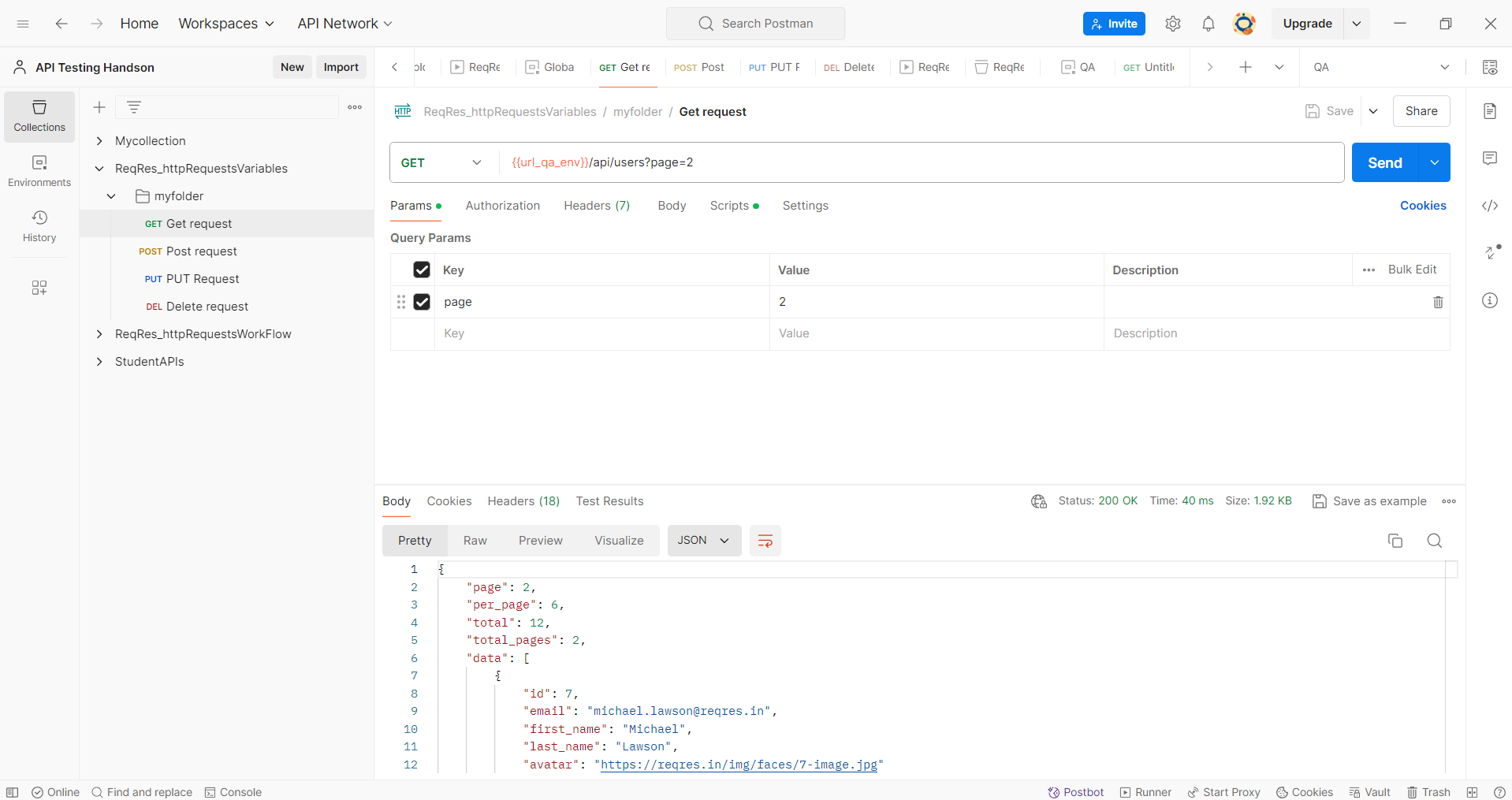




Environment variables:



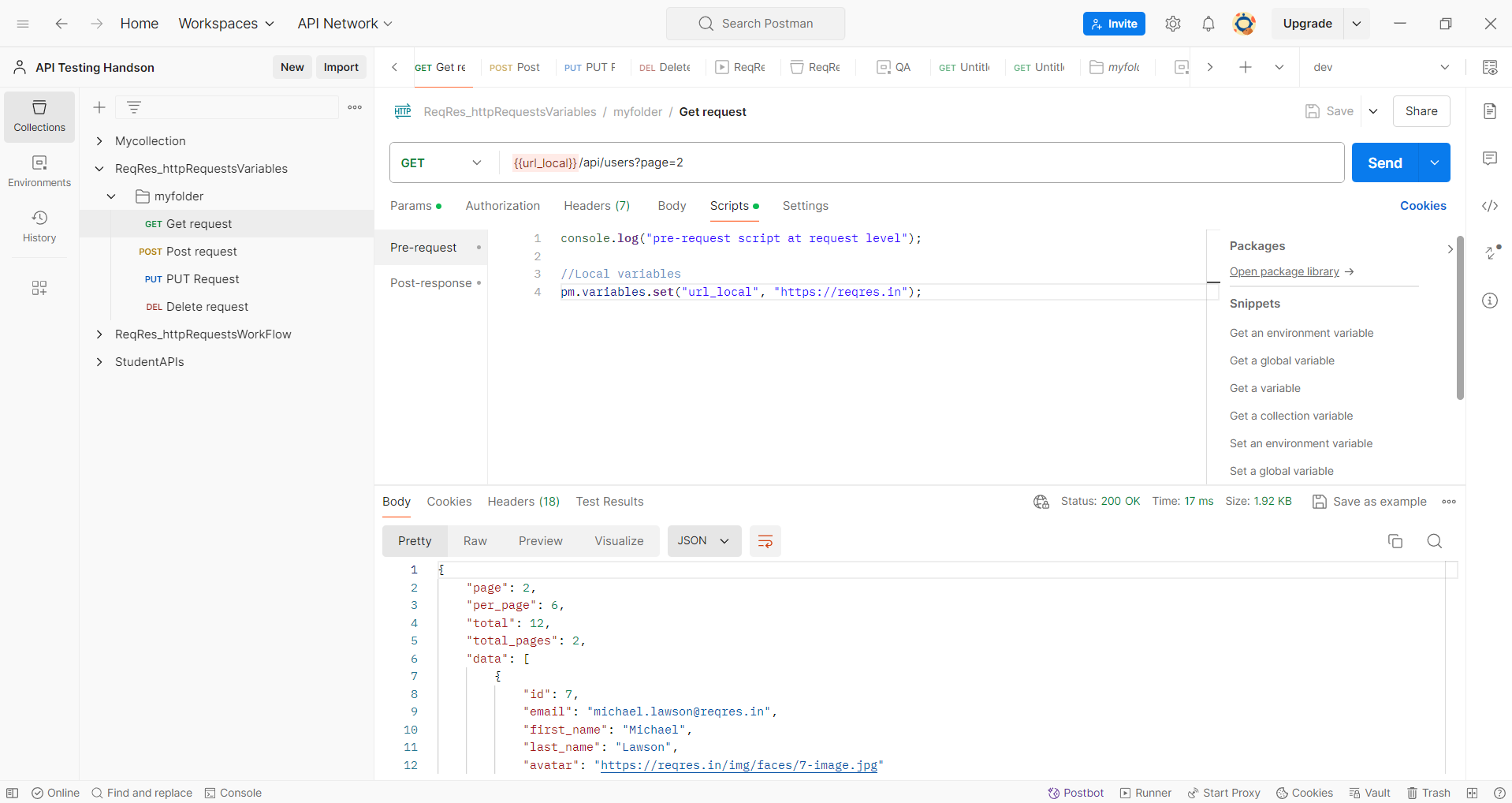
Select environment as QA.



**Local Variable:**

//Local variables

pm.variables.**set**("url\_local", "https://reqres.in");



We can also initialize global variable under request

Ex:

GET => {{url\_local}}/api/users?page={{userid\_globals}}

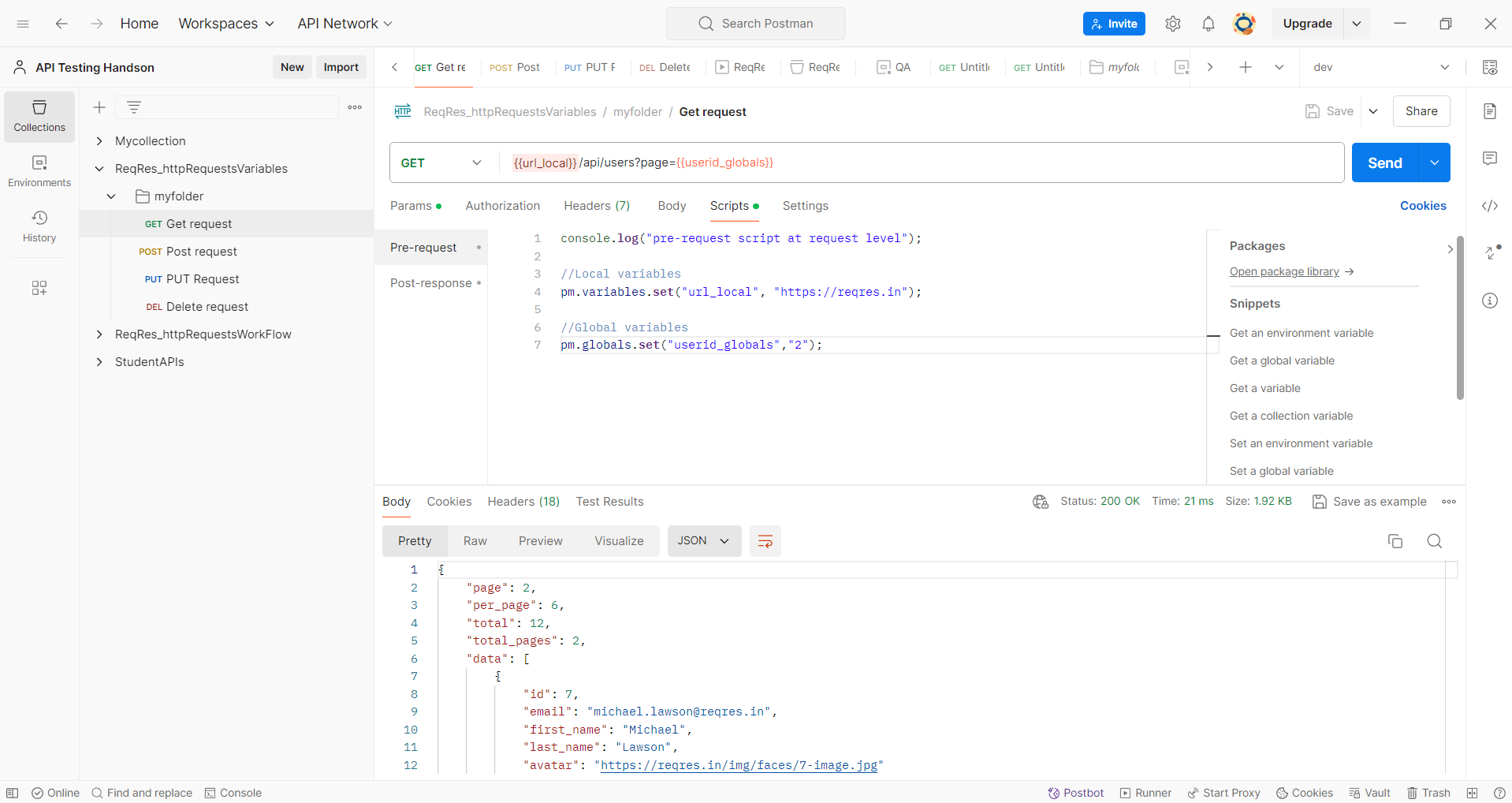
Script=>

//Local variables

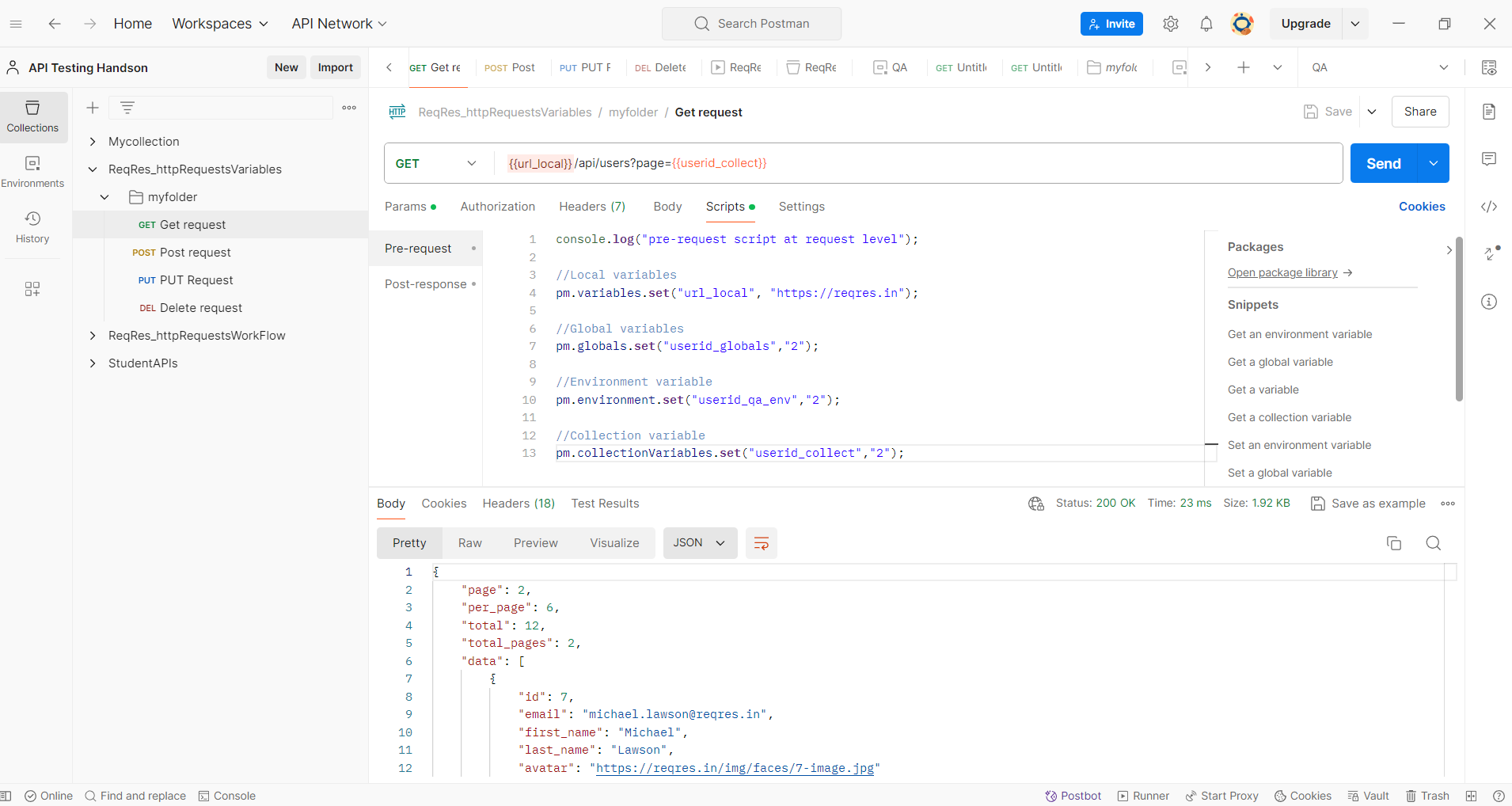
pm.variables.**set**("url\_local", "https://reqres.in");

//Global variables

pm.globals.**set**("userid\_globals","2");



A global variable created in a request can also be accessible everywhere.



//Local variables

pm.variables.**set**("url\_local", "https://reqres.in");

//Global variables

pm.globals.**set**("userid\_globals","2");

//Environment variable

pm.environment.**set**("userid\_qa\_env","2");

//Collection variable

pm.collectionVariables.**set**("userid\_collect","2");

**Creating variables using pre-request scripts:**

//Local variables

pm.variables.**set**("url\_local", "https://reqres.in");

//Global variables

pm.globals.**set**("userid\_globals","2");

//Environment variable

pm.environment.**set**("userid\_qa\_env","2");

//Collection variable

pm.collectionVariables.**set**("userid\_collect","2");

**Unset/Delete: in Post-response Scripts**

Ex:

//Global variables

pm.globals.unset("userid\_globals");

**Capture the values from variables:**

console.**log**(pm.globals.**get**("userid\_globals"));

console.**log**(pm.environment.**get**("userid\_qa\_env"));

console.**log**(pm.collectionVariables.**get**("userid\_collect"));

console.**log**(pm.variables.**get**("url\_local"));

**Methods:**

Set, unset, get => global, env, collection, local.

**API Chaining**

Ex: <https://gorest.co.in/>

GET: <https://gorest.co.in/public/v2/users>

**GoRest API :**

<https://gorest.co.in/>

Generate Token using github account

url: <https://gorest.co.in/>

endpoint: /public/v2/users

REST API Http Response Codes

* 200: OK. Everything worked as expected.
* 201: A resource was successfully created in response to a POST request. The Location header contains the URL pointing to the newly created resource.
* 204: The request was handled successfully and the response contains no body content (like a DELETE request).
* 304: The resource was not modified. You can use the cached version.
* 400: Bad request. This could be caused by various actions by the user, such as providing invalid JSON data in the request body etc.
* 401: Authentication failed.
* 403: The authenticated user is not allowed to access the specified API endpoint.
* 404: The requested resource does not exist.
* 405: Method not allowed. Please check the Allow header for the allowed HTTP methods.
* 415: Unsupported media type. The requested content type or version number is invalid.
* 422: Data validation failed (in response to a POST request, for example). Please check the response body for detailed error messages.
* 429: Too many requests. The request was rejected due to rate limiting.
* 500: Internal server error. This could be caused by internal program errors.

Request Body:

{

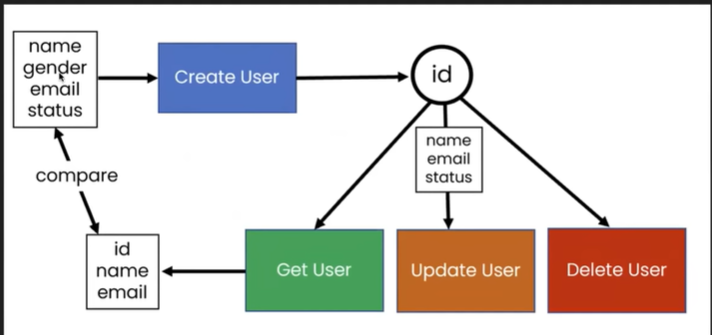
"name": "rutuja",

"email": "test@gmail.com",

"gender": "female",

"status": "active"

}



abcd