Postman

What is Postman?

Postman is a popular collaboration platform for API development. It provides a user-friendly interface that allows developers to design, test, document, and share APIs effortlessly. Some key features of Postman include:

- API Testing: Postman allows developers to create and execute automated tests for APIs, making it easier to ensure the reliability and functionality of APIs.
- API Documentation: Developers can generate comprehensive documentation for APIs within Postman, making it easier for other developers to understand how to use them.
- Mock Servers: Postman allows developers to create mock servers for APIs, enabling them to simulate API behavior without actually implementing the backend logic, which is useful for testing and development purposes.
- API Monitoring: Postman offers monitoring capabilities to track the performance and uptime of APIs, helping developers identify and resolve issues quickly.
- API Collections: Postman allows users to organize APIs into collections, making it easier to manage and share sets of related APIs with team members or the broader community.

With Postman, developers can seamlessly interact with a wide range of API types, including REST, GraphQL, gRPC, WebSockets, and more.
Here are some examples below:

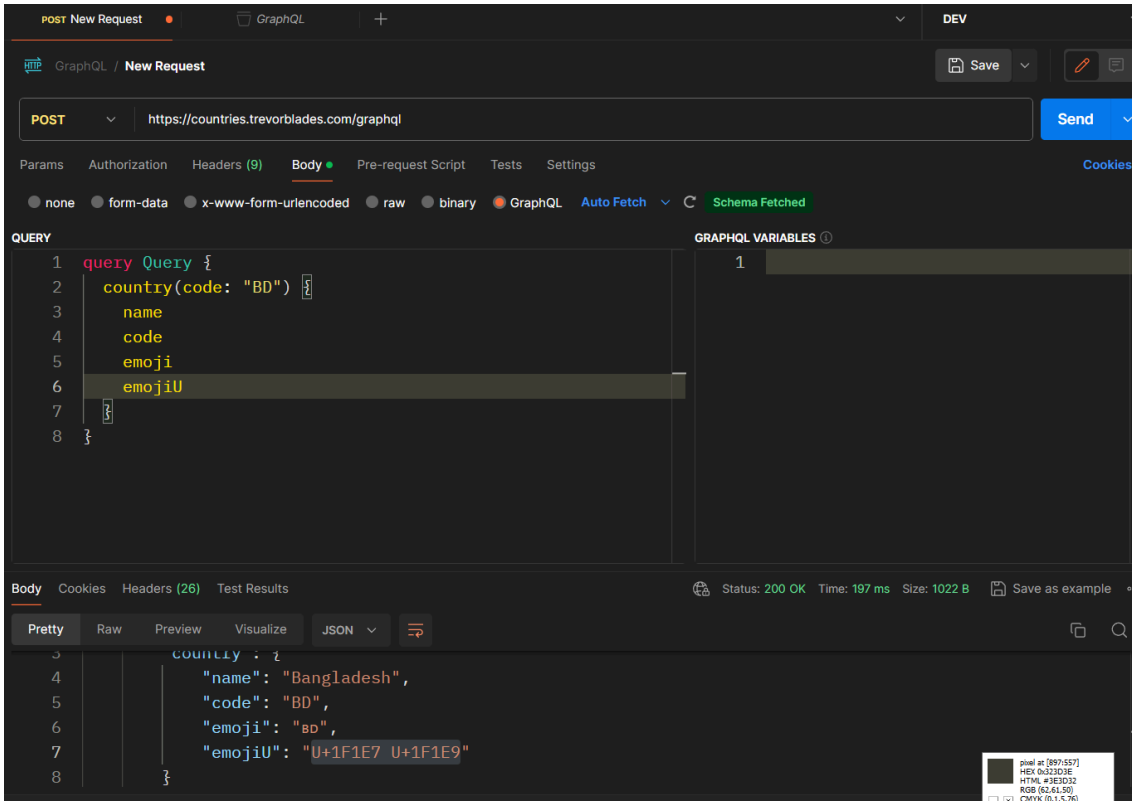
Call GraphQL API:

With the assistance of Postman, calling GraphQL APIs becomes effortless, thanks to its swift schema fetching capabilities. Here, we utilize an online GraphQL API. Simply input the query into the designated Graphql query section within the request body.

https://countries.trevorblades.com/graphql

Query:

```
JavaScript
query Query {
 country(code: "BR") {
  name
  native
  capital
  emoji
  currency
  languages {
   code
   name
  }
 }}
```



Call SOAP API:

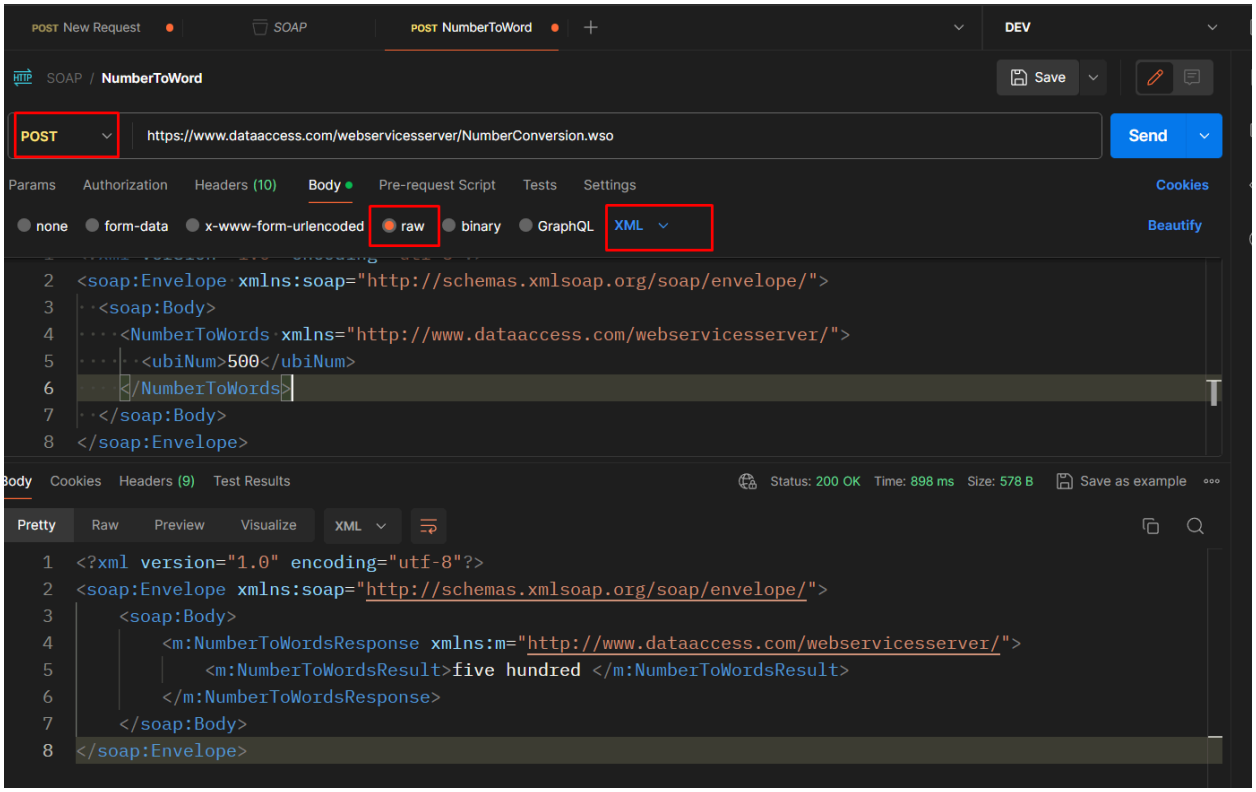
Additionally, we can utilize Postman to call SOAP APIs. In this example, we invoke a SOAP API for number conversion using an XML request body. Upon sending the request, an XML response will be returned. Before clicking the send button, ensure that "Content-Type": "text/xml" is included in the request header.

https://www.dataaccess.com/webservicesserver/NumberConversion.wso

Content-Type: Content-Type: text/xml

Request Body:

```javascript
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
 <soap:Body>
  <NumberToWords xmlns="http://www.dataaccess.com/webservicesserver/">
   <ubiNum>500</ubiNum>
  </NumberToWords>
 </soap:Body>
</soap:Envelope>
```



Postman Collections

Now let's talk about the postman collections. Postman collections are an essential feature of the Postman platform, serving as a convenient way to organize, manage, and share APIs and requests.

Here are some key features of Postman collections:

- **Organization:** Collections enable you to organize API requests logically, making it easier to manage and navigate through your APIs.
- **Sharing:** You can share collections with team members or the broader community, facilitating collaboration and knowledge sharing.

- **Documentation:** Collections can include documentation, making it simple to provide detailed information about each API request, including parameters, headers, and response examples.
- **Testing:** Collections allow you to include test scripts, enabling automated testing of APIs as part of your workflow.
- **Environments:** Collections support the use of environments, allowing you to define variables that can be used across multiple requests within the collection. This is particularly useful for managing different environments such as development, staging, and production.

SO, collection is a grouping or container for a set of related API requests. It allows you to organize, structure, and logically manage your API requests. Collections can include multiple requests, scripts, and even folders to help you streamline and categorize your API development and testing workflows

Postman Variables

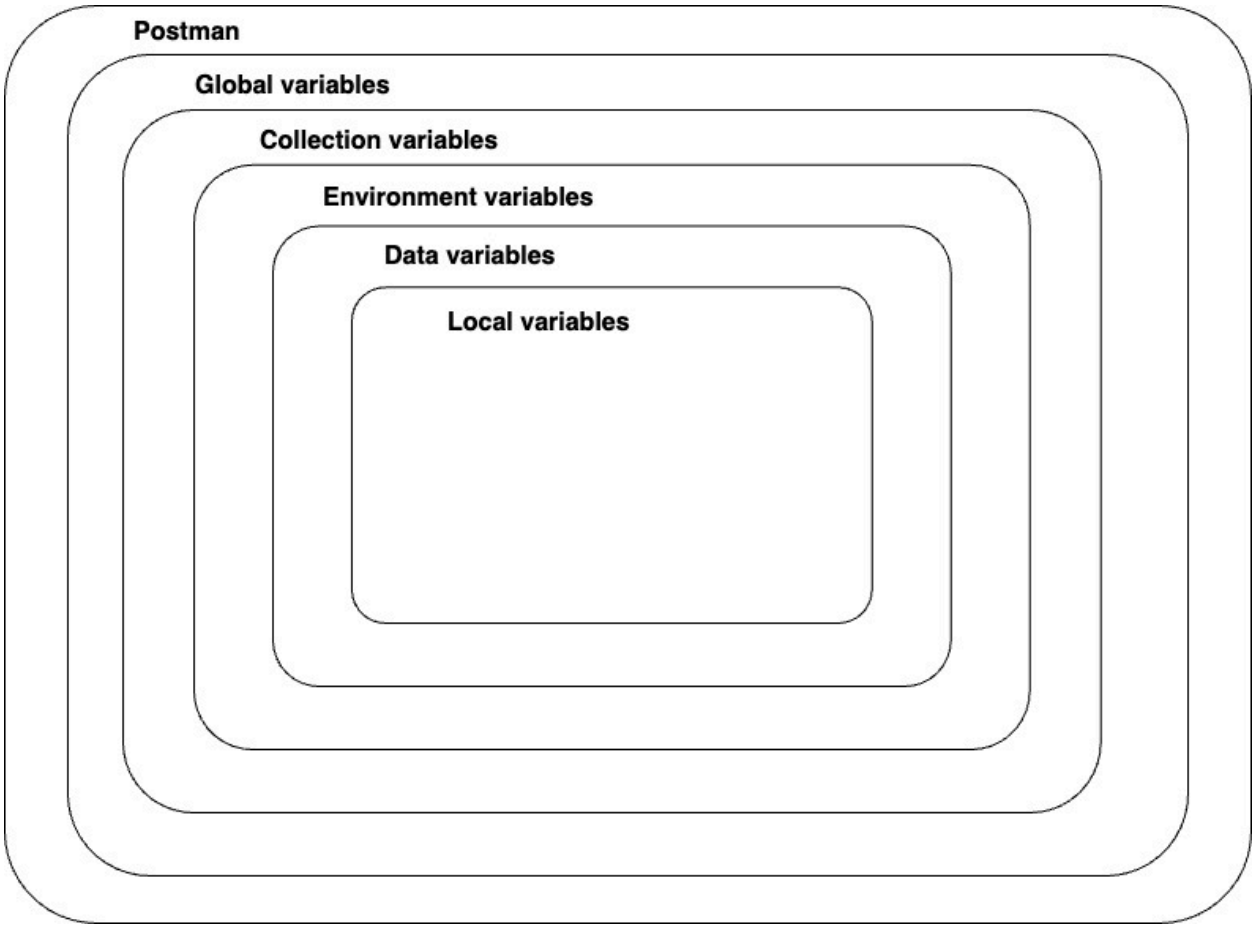
When discussing Postman collections, it's essential to delve into Postman variables. Without a grasp of Postman variables, you can't fully leverage the benefits offered by the Postman platform. These variables are crucial components that enhance flexibility, reusability, and efficiency in managing API requests, scripts, and environments. They empower developers to streamline workflows, simplify maintenance, and ensure dynamic data handling across different testing and development scenarios. In essence, understanding and effectively utilizing Postman variables are fundamental steps toward maximizing the potential of the Postman toolset.

Postman supports different types of variables, including *global*, *collection*, *environment*, *data*, and *local*.

Here is the details overview:

- **Global variables** enable you to access data between collections, requests, and test scripts. Global variables are available throughout a workspace. Global variables are a helpful tool for sharing variables across the workspace.

- **Collection variables** are available throughout the requests in a collection. Collection variables are useful for keeping variables private within a collection of APIs and don't want to share variables outside the collection.
- **Environment variables** enable you to scope your work to different environments, for example, Development and testing. One environment can be active at a time.
- **Data Variables** in your data come from external CSV and JSON files. You use these sets of data when running collections with Newman or the Collection Runner. These variables have current values that only last during the request or collection runs.
- **Local variables** are temporary variables that are accessed in your request scripts. Local variable values are scoped to a single request or collection run and are no longer available when the run is complete. Local variables are suitable if you need a value to override all other variable scopes but don't want the value to persist once execution has ended.



If a variable with the same name is declared in two different scopes, the value stored in the variable with the narrowest scope will be used. For example, if there is a global

variable named `username` and a local variable named `username`, the local value will be used when the request runs.

Access variables from scripts

You can access variables by the pm object in Pre-request Scripts and Test Scripts, which is a powerful feature in Postman. Here, I'll elucidate how to access variables from various scopes and their respective use cases:

Method	Use-case	Example
`pm.globals`	Use to define a global variable.	`pm.globals.set("variable_key", "variable_value");`
`pm.collectionVariables`	Use to define a collection variable.	`pm.collectionVariables.set("variable_key", "variable_value");`
`pm.environment`	Use to define an environment variable in the currently selected environment.	`pm.environment.set("variable_key", "variable_value");`
`pm.variables`	Use to define a local variable.	`pm.variables.set("variable_key", "variable_value");`
`unset`	You can use `unset` to remove a variable.	`pm.environment.unset("variable_key");`

You can retrieve the current value of a variable in your scripts using the object representing the scope level and the `.get` method:

```javascript
//access a variable at any scope including local
pm.variables.get("variable_key");

//access a global variable
pm.globals.get("variable_key");

//access a collection variable
pm.collectionVariables.get("variable_key")

;//access an environment variable
pm.environment.get("variable_key");
```

Note: Using `pm.variables.get()` to access variables in your scripts gives you the option to change variable scope without affecting your script functionality. This method will return whatever variable currently has the highest precedence (or narrowest scope).

Using data variables

The Collection Runner lets you import a CSV or a JSON file, and use the values from the data file inside requests and scripts. You can't set a data variable inside Postman because it's pulled from the data file, but you can access data variables inside scripts, for example using

```javascript
pm.iterationData.get("variable_name").
```

Using variables

You can use double curly braces to reference variables throughout Postman. For example, to reference a variable named `username` you would use the following syntax with double curly braces around the name:
{{username}}

When you run a request, Postman will resolve the variable and replace it with its current value.

Using dynamic variables

Postman provides dynamic variables you can use in your requests.
Examples of dynamic variables include:

- `{{$guid}}` : A v4-style GUID
- `{{$timestamp}}`: The current Unix timestamp in seconds
- `{{$randomInt}}`: A random integer between 0 and 1000

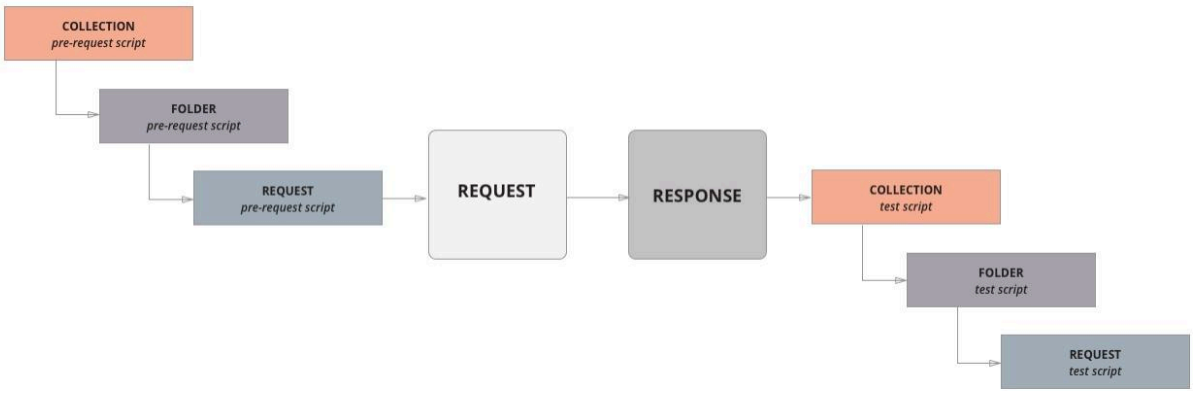
Scripts (pre-scripts, test scripts)

Now let’s talk about the Scripts in Postman. Postman scripts are used to automate and customize API requests and tests. They allow you to write code that will be executed before or after a request is sent, giving you the ability to manipulate data, set up test conditions, or perform other tasks.

Before diving into Postman scripts, it's crucial to comprehend their execution order. Understanding this order ensures effective utilization of Postman's scripting capabilities.

For every request in a collection, scripts will execute in the following order:

- A pre-request script associated with a collection will run before every request in the collection.
- A pre-request script associated with a folder will run before every direct child request in the folder.
- A test script associated with a collection will run after every request in the collection.
- A test script associated with a folder will run after every direct child request in the folder.



So, for every request in a collection, the scripts will always run according to the same hierarchy. Collection-level scripts (if any) will run first, then folder-level scripts (if any), and then request-level scripts (if any). Note that this order of execution applies to both pre-request and test scripts.

Now Let's embark on some hands-on practice by writing test scripts for our Auth and Inventory APIs. Below are the details along with the provided test scripts:
Note: {{auth_base_url}} is the base URL of auth API e.g., (http://localhost:4005). {{inventory_base_url}} is the base URL of Inventory API e.g, (http://localhost:4001)

Auth API

POST : {{auth_base_url}}/api/v1/signup

Request Body:

```javascript
{
  "name":"{{name}}",
  "email":"{{$randomEmail}}",
  "Password":"{{password}}"
}
```

Response Body:

```javascript
{
  "message": "User created successfully",
  "links": {
    "signIn": "${BASE_URL}/api/v1/signin"
  }
}
```

Test-Scripts:

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

pm.test("it should respons 201 status code", () => {
  pm.response.to.have.status(201)
})

pm.test("it should response expected properties if user created", () => {

  pm.expect(response.message).to.be.equal("User created successfully")
```

```
pm.expect(response.links.signIn).to.be.equal("${BASE_URL}/api/v1/signin"
)

  pm.expect(response).to.have.all.keys("message", "links")
  pm.expect(response.links).to.have.property("signIn")

  pm.expect(response.message).to.be.a("string"),
  pm.expect(response.links).to.be.a("object"),
  pm.expect(response.links.signIn).to.be.a("string")

})
```

POST: {{auth_base_url}}/api/v1/signin

Request Body:

JavaScript

```
{
    "email":"{{email}}",
    "password":"{{password}}"
}
```

Response Body:

JavaScript

```
{
    "token":
"eyJhbGciOiJIUzI1NiIsInR5cCJpZCI6IjlhNDMxMDQ1LThiYmMtNDllZC05N2QwLTcyODA
5MjM0MTIwOCIsIm5hbNlbCBIb3NzYWluIiwiZW1haWwiOiJyYXNlbEBnbWFpbC5jb20iLCJp
YXQiOjE3MDY2OTgzNjYsImV4cgwMjM2Nn0.iH2sjzuraXHqfnaGaSStrGBgJqT_2Y1v8yECS
So9BA8"
}
```

Test-Scripts:

JavaScript

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

pm.test("it shuld respons 200 or 404 stats code", () => {
  pm.expect(pm.response.code).to.be.oneOf([200, 404, 401])
})

pm.test("it should response token", () => {
  pm.expect(response.token).to.be.a("string")
  pm.expect(response).to.be.have.property("token")
})

pm.response.code === 404 && pm.test("it should response 'User not found!' message
if user send wrong email", () => {

  pm.expect(response.message).to.be.a("string"),
    pm.expect(response.message).to.be.equal("User not found!")
  pm.expect(response).to.be.have.property("message")
  pm.expect(response).to.not.be.have.property("token")

})

pm.response.code === 401 && pm.test("it should response 'Password is incorrect'
message if user send wrong password", () => {

  pm.expect(response.message).to.be.a("string"),
    pm.expect(response.message).to.be.equal("Password is incorrect!")
  pm.expect(response).to.be.have.property("message")
  pm.expect(response).to.not.be.have.property("token")

})
```

POST: {{auth_base_url}}/api/v1 /checkpoint

Request Header:

```javascript
"Authorization": "Bearer your_jwt_token"
```

Response Body:

```javascript
{
    "message": "Access granted"
}
```

Test-Scripts:

```javascript
const response = pm.response.json()
const authHeader = pm.request.headers.get("Authorization")

pm.test("it should return 200 status code", () => {
  pm.response.to.have.status(200)
})

pm.test("it should response `Access granted` message", () => {
  pm.expect(response).to.be.have.property("message")
  pm.expect(response.message).to.be.a("string")
  pm.expect(response.message).to.be.eql("Access granted")
})

!authHeader && pm.test("it should respons `Access denied. No token
provided.` if user doesn't send token", () => {
  pm.response.to.have.status(401)
  pm.expect(authHeader).not.to.exist
```

```
  pm.expect(response.message).to.be.equal("Access denied. No token
provided.")
})
```

Inventory API

GET: {{inventory_base_url}}/api/v1/health

Response Body:

JavaScript

```
{
    "message": "Your server is healthy"
}
```

Test-Scripts:

JavaScript

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

pm.test("it should response 200 response",()=>{

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

})

pm.test("it should response 'Your server is healthy' message",()=>{

  pm.expect(response).to.be.have.property("message")

  pm.expect(response.message).to.be.a("string")

  pm.expect(response.message).to.be.equal("Your server is healthy")

})
```

POST: {{inventory_base_url}}/api/v1/products

Request Body:

```javascript
{
    "name": "{{$randomProduct}}",
    "price": {{$randomPrice}},
    "quantity": {{$randomInt}},
    "userID": "{{$guid}}"
}
```

Request Header:

```javascript
"Authorization": "Bearer your_jwt_token"
```

Response Body:

```javascript
{
    "message": "Success",
    "product": {
        "id": "b5d9abd8-746c-44fd-9b97-28d9d77ab25f",
        "name": "Salad",
```

```
        "quantity": 923,
        "price": 662.06,
        "createdBy": "15fb3527-434d-4d94-ad34-77de01ad8693",
        "createdAt": "2024-02-02T12:48:00.772Z",
        "updatedAt": "2024-02-02T12:48:00.772Z"
    }
}
```


Test-Scripts:

JavaScript

```
const response=pm.response.json()
const authHeader=pm.request.headers.get("Authorization")

pm.test("it should return status code 201 Created", function () {
  pm.response.to.have.status(201);
  pm.collectionVariables.set("productId",response.product.id)
});

pm.test("response has a success message", function () {
  pm.expect(response.message).to.equal("Success");
});

pm.test("product object is present", function () {
  pm.expect(response).to.have.property("product");
});

pm.test("response type should be object",()=>{
  pm.expect(response).to.be.a("object")
})

pm.test("product shuld have expected properties",()=>{
  pm.expect(response.product).to.have.all.keys("id", "name", "quantity",
"price", "createdBy", "createdAt", "updatedAt")
})
```

```
!authHeader && pm.test("it should respons `Access denied. No token
provided.` if user doesn't send token", () => {
  pm.response.to.have.status(401)
  pm.expect(authHeader).not.to.exist
})
```

GET:
{{inventory_base_url}}/api/v1/products/174ba87d-7c5b-4f76-a245-b608c1fc7b39

Response Body:

JavaScript


```
{
    "message": "Success",
    "product": {
        "id": "b5d9abd8-746c-44fd-9b97-28d9d77ab25f",
        "name": "Salad",
        "quantity": 923,
        "price": 662.06,
        "createdBy": "15fb3527-434d-4d94-ad34-77de01ad8693",
        "createdAt": "2024-02-02T12:48:00.772Z",
        "updatedAt": "2024-02-02T12:48:00.772Z"
    }
}
```

Test-Scripts:

JavaScript

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

pm.test("it should response 200 or 404 status code", function () {
  pm.expect(pm.response.code).to.be.oneOf([200, 404])
});

pm.test("it should have product property if status code 200", () => {
  pm.expect(response).to.have.property("product")
})

pm.response.code ===404 && pm.test("it should not have product propery if
status code 404", () => {
  pm.expect(response.message).to.be.eql("Product Not Found")
    pm.expect(response).to.not.have.property("product")

})

pm.test("it should have message property", () => {
  pm.expect(response).to.have.property("message")
})

pm.test("message should be string type", () => {
  pm.expect(response.message).to.be.a("string")
})
```

GET: {{inventory_base_url}}/api/v1/products

JavaScript

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

pm.test("it Should have status code 200 OK", () => {
  pm.response.to.have.status(200)
})

pm.test("it should response success message", () => {
  pm.expect(response.message).to.equal("Success")
})

pm.test("it should response with product property", () => {
  pm.expect(response).to.have.property("products")
})

response.products.length && pm.test("it should have expected products
properties if products are present", () => {

  const product = response.products[0]
  pm.expect(product).to.have.property("id")
  pm.expect(product).to.have.property("name")
  // -----
  pm.expect(product).to.have.all.keys("id", "name", "quantity", "price",
"createdBy", "createdAt", "updatedAt")

})

!response.products.length && pm.test("it should have empty array if not
products found", () => {
  console.log("products")
  pm.expect(response.products).to.be.empty
})


// Visualizer
var template = `
<table>
    <tr style="background:blue; color: #fff">
        <th>ID</th>
```

```
        <th>Name</th>
        <th>Quantity</th>
        <th>Price</th>
        <th>CreatedBy</th>
        <th>CreatedAt</th>
        <th>UpdatedAt</th>
    </tr>

    {{#each response}}
        <tr>
            <td>{{id}}</td>
            <td>{{name}}</td>
            <td>{{quantity}}</td>
            <td>{{price}}</td>
            <td>{{createdBy}}</td>
            <td>{{createdAt}}</td>
            <td>{{updatedAt}}</td>
        </tr>
  {{/each}}

</table>
`

pm.visualizer.set(template, {response: pm.response.json().products});
```

PUT: {{inventory_base_url}}/api/v1/products/:productId

Request Body:

```javascript
{
    "name": "{{$randomProduct}}",
    "price": {{$randomPrice}},
    "quantity": {{$randomInt}}
}
```

Request Header:

```javascript
"Authorization": "Bearer your_jwt_token"
```

Response Body:

```javascript
{
  "message": "Success",
  "product": {
    "id": "b5d9abd8-746c-44fd-9b97-28d9d77ab25f",
    "name": "Gloves",
    "quantity": 264,
    "price": 239.11,
    "createdBy": "15fb3527-434d-4d94-ad34-77de01ad8693",
    "createdAt": "2024-02-02T12:48:00.772Z",
    "updatedAt": "2024-02-02T12:53:15.218Z"
  }
}
```

Test-Scripts:

JavaScript

```
const response=pm.response.json()
const authHeader=pm.request.headers.get("Authorization")

pm.test("it should response 200 status code",()=>{
  pm.response.to.have.status(200)
})

pm.test("it should have expected properties",()=>{
  pm.expect(response).to.have.all.keys("message","product")

pm.expect(response.product).to.have.all.keys("id","name","quantity","pri
ce","createdBy","createdAt", "updatedAt")
})

pm.response.code===404 && pm.test("it should response 'Product not found'
message if there are no product",()=>{
  pm.expect(response).not.to.have.property("products")
  pm.expect(response.message).to.be.a("string")
  pm.expect(response.message).to.be.equal("Product not found")
})

!authHeader && pm.test("it should respons `Access denied. No token
provided.` if user doesn't send token", () => {
  pm.response.to.have.status(401)
  pm.expect(authHeader).not.to.exist
  pm.expect(response.message).to.be.equal("Access denied. No token
provided.")
})
```

Delete: {{inventory_base_url}}/api/v1/products/:productId

Request-Header:

```javascript
"Authorization": "Bearer your_jwt_token"
```

Response Body:

```javascript
{
  "message": "Success",
  "product": {
    "id": "b5d9abd8-746c-44fd-9b97-28d9d77ab25f",
    "name": "Gloves",
    "quantity": 264,
    "price": 239.11,
    "createdBy": "15fb3527-434d-4d94-ad34-77de01ad8693",
    "createdAt": "2024-02-02T12:48:00.772Z",
    "updatedAt": "2024-02-02T12:53:15.218Z"
  }
}
```

Test-Scripts:

```javascript
const response=pm.response.json()
const authHeader=pm.request.headers.get("Authorization")

pm.test("it should response 200 status code",()=>{
  pm.response.to.have.status(200)
})

pm.test("it should have expected properties",()=>{
  pm.expect(response).to.have.all.keys("message","product")
```

```
pm.expect(response.product).to.have.all.keys("id","name","quantity","pri
ce","createdBy","createdAt", "updatedAt")
})

pm.response.code===404 && pm.test("it should response 'Product not found'
message if there are no product",()=>{
  pm.expect(response).not.to.have.property("products")
  pm.expect(response.message).to.be.a("string")
  pm.expect(response.message).to.be.equal("Product not found")
})

!authHeader && pm.test("it should respons `Access denied. No token
provided.` if user doesn't send token", () => {
  pm.response.to.have.status(401)
  pm.expect(authHeader).not.to.exist
  pm.expect(response.message).to.be.equal("Access denied. No token
provided.")
})
```

Visualizer

Now, let's explore one of the most intriguing features of Postman: the ability to visualize API responses. With Postman Visualizer, you can transform your API responses into interactive visual representations such as tables, bar charts, pie charts, and more. This functionality adds a whole new dimension to API testing and development
You can use this to model and highlight the information that's relevant to your project, instead of having to read through raw response data. When you share a Postman Collection, other people on your team can also understand your visualizations within the context of each request.

You can enable the visualizer from the test scripts using Visualizer API . You can access the Visualizer from the Postman API. The `pm.visualizer.set()` method takes three parameters:

- `layout` (required): The first parameter is a Handlebars HTML template string.
- `data` (optional): The second parameter is data you can bind to the template. The properties of this object can be accessed in the template.
- `options` (optional): The third argument is an `options` object for `Handlebars.compile()`. You can use this to control how Handlebars compiles the template.

Postman uses the information you pass to `pm.visualizer.set()` to render an HTML page in the sandbox for the Visualizer. Select the **Visualize** tab for the rendered HTML page. The `layout` string is inserted into the `<body>` of the rendered page, including any JavaScript, CSS, and HTML that the template has.
sample response

```javascript
[
  {
    "name": "RaselOfficial",
    "email": "rasel@gmail.com"
  },
  {
    "name": "HM Nayem",
    "email": "nayem@gmail.com"
  },
]
```

The Visualizer code creates a Handlebars template to render a table displaying the names and email addresses by looping over an array. Handlebars can do this with the `{{#each}}` tag. This script runs in the request **Tests**:

```javascript
var template = `
    <table bgcolor="#FFFFFF">
        <tr>
            <th>Name</th>
            <th>Email</th>
        </tr>

        {{#each response}}
            <tr>
                <td>{{name}}</td>
                <td>{{email}}</td>
            </tr>
        {{/each}}
    </table>
`;
```

The variable names inside the double curly braces in the template will be substituted by the data passed to the `pm.visualizer.set()` method. To apply the template, the following code completes the **test** script:

```
JavaScript
pm.visualizer.set(template, {response: pm.response.json()});
```

The `template` variable is the template string created earlier. The second argument passed is an object defined as the `response` property—this is the variable that the template expects in the `{{#each response}}` loop. The value assigned to the `response` property is the response JSON data from the request parsed as an object.