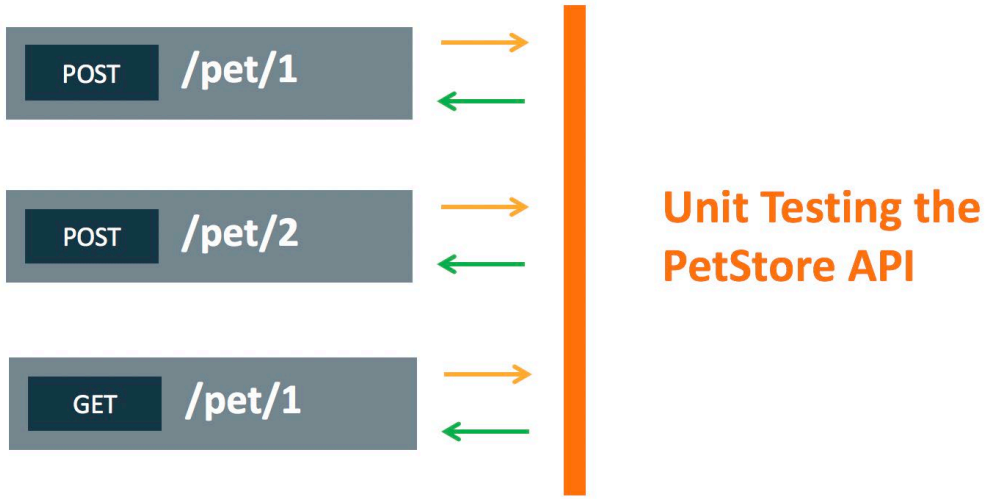
API Testing

The Types of API Testing

- Functionality testing — the API works and does exactly what it's supposed to do.
- Reliability testing — the API can be consistently connected to and lead to consistent results
- Load testing — the API can handle a large amount of calls
- Creativity testing — the API can handle being used in different ways.
- Security testing — the API has defined security requirements including authentication, permissions and access controls. See some API security tips for protecting vital data
- Proficiency testing — the API increases what developers are able to do.
- API documentation testing — also called discovery testing, the API documentation easily guides the user.
- Negative Testing — checking for every kind of wrong input the user can possibly supply

Unit Testing

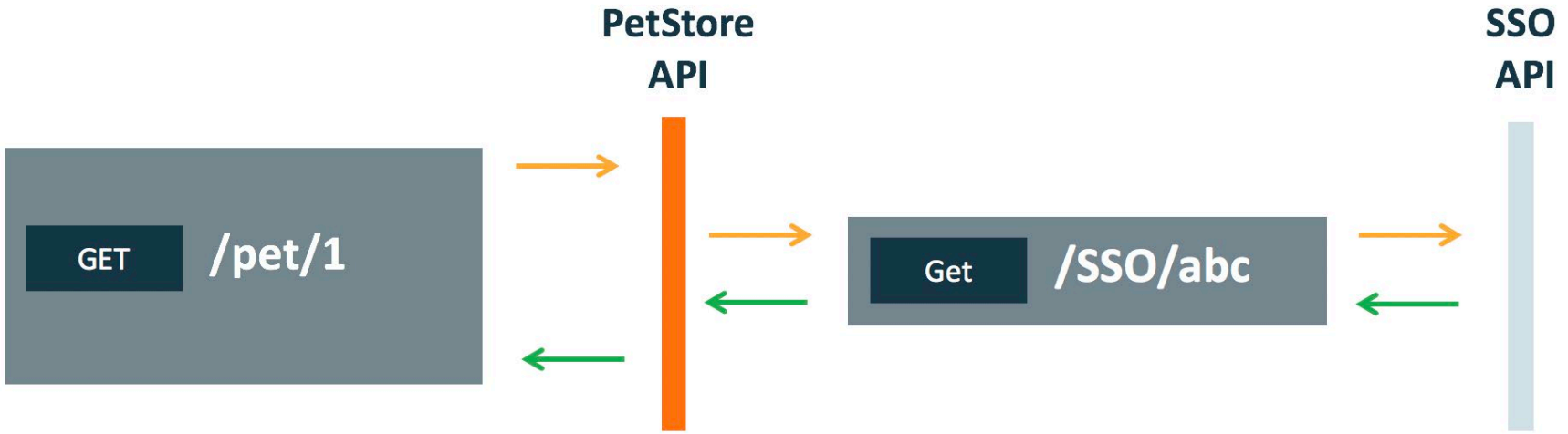
- The easiest way to think about a "unit test" and APIs is testing a single endpoint, with a single request, looking for a single response or set of responses. Many times, this type of testing can be done manually via the command line and something like a URL command or with lightweight tools like SoapUI.



https://smartbear.com/learn/api-testing/what-is-api-testing/

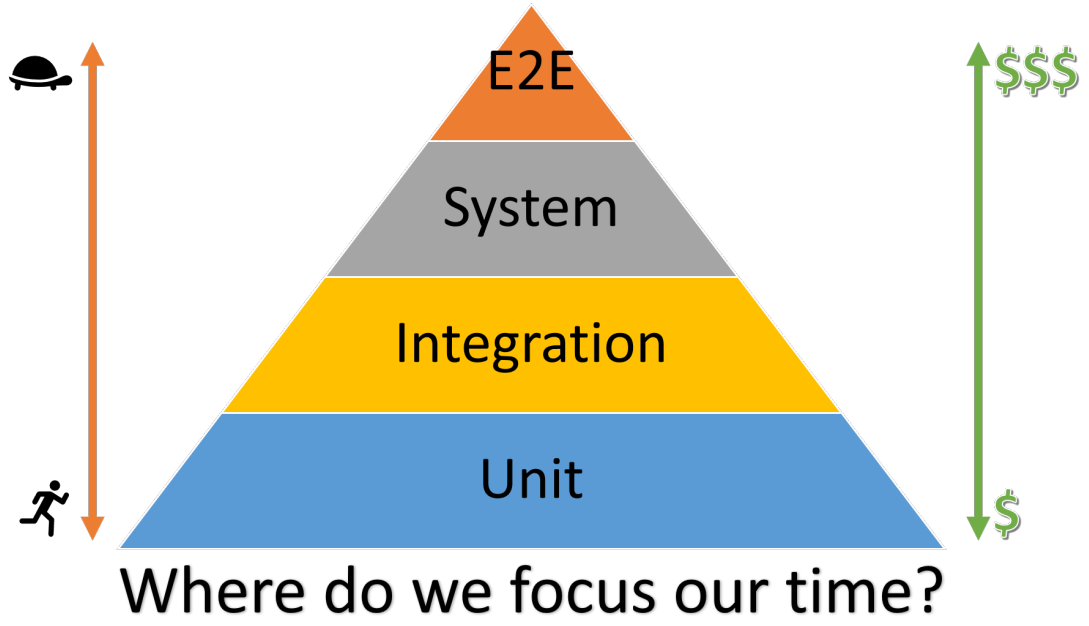
Integration Testing

- Integration testing is the most often used form of API testing, as APIs are at the center of most integrations between internal or third-party services



End-to-End Testing

- End-to-end testing can help us validate the flow of data and information between a few different API connections



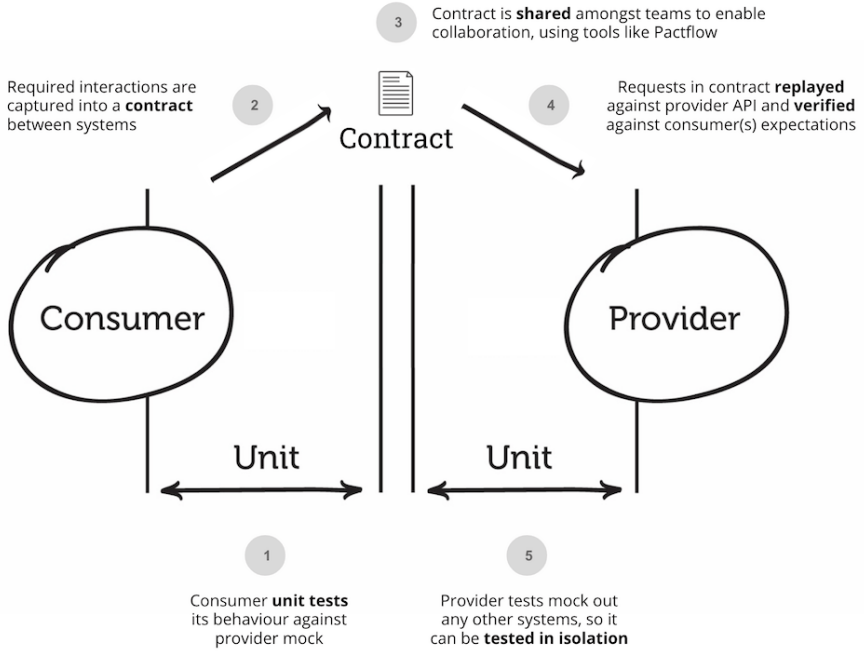
https://kentcdodds.com/blog/write-tests

Performance Testing

- We are trying to change the paradigm **of load testing** and shift it left into every commit. Previously, load testing was kept in the hands of the few and was difficult to execute in a CI/CD environment. ReadyAPI is a performance testing tool for RESTful, SOAP, and other web services that enables nearly any team member to embed performance tests into their CI/CD pipeline.

Contract Testing

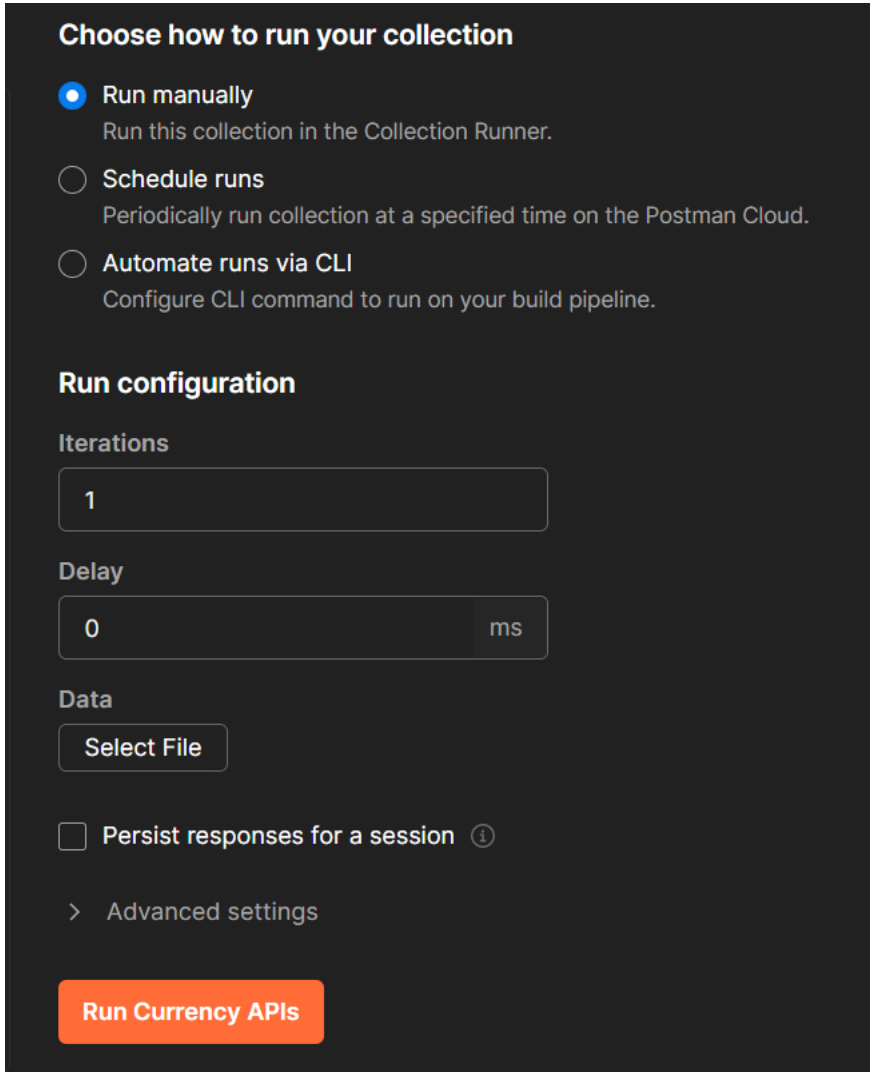
- Contract testing is a technique which validates that two separate systems (such as two microservices) are able to communicate. The interactions exchanged between each service are captured and stored in a contract which is used to validate that both systems adhere to a common agreement. The simplest way to orchestrate contract testing is with Pactflow.



https://smartbear.com/learn/api-testing/what-is-api-testing/

Load Testing

- Load testing is the process of putting simulated demand on software, an application or website in a way that tests or demonstrates it's behavior under various conditions.
  - [Chapter 2 – Types of Performance Testing]

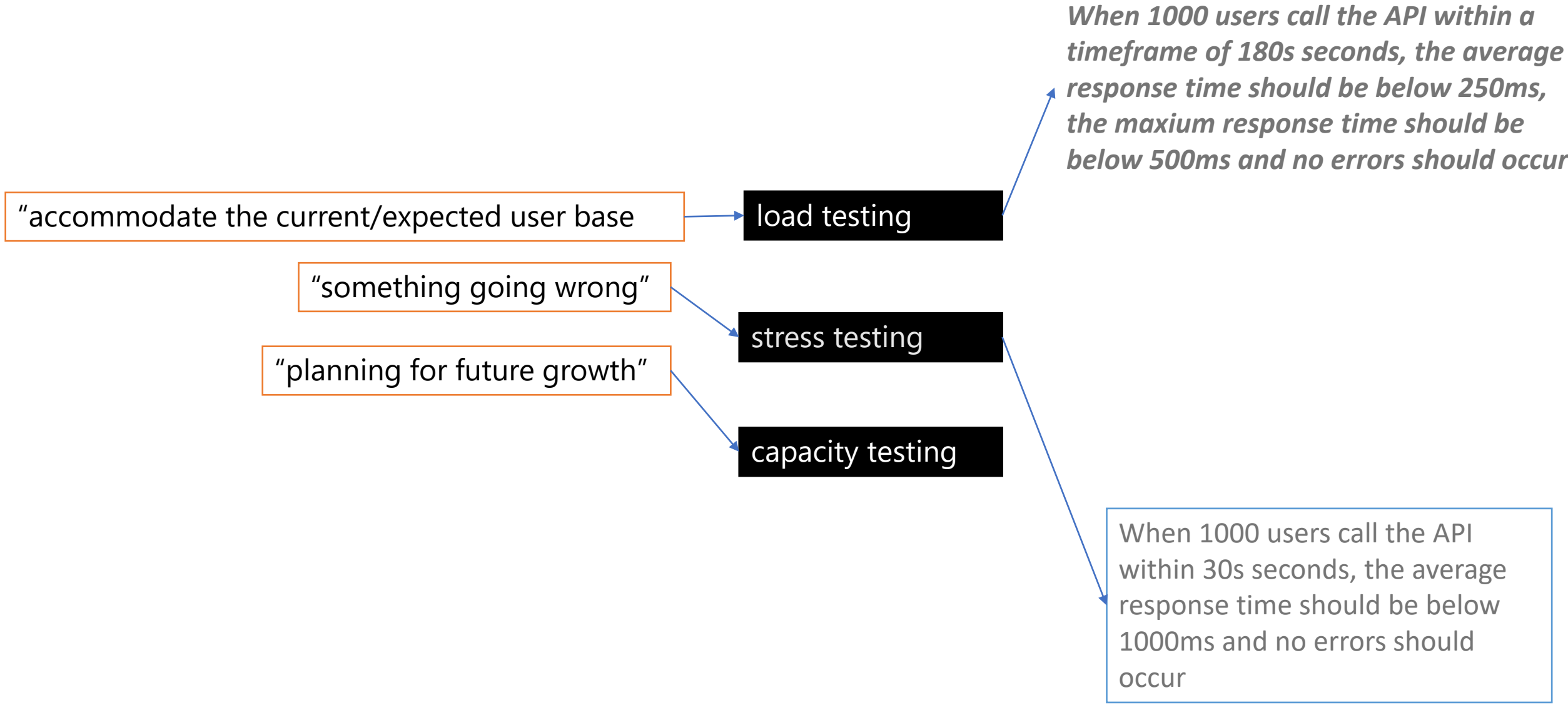


https://learn.microsoft.com/en-us/previous-versions/msp-n-p/bb924357(v=pandp.10)?redirectedfrom=MSDN

Performance Testing

Term	Purpose	Notes
Performance test	To determine or validate speed, scalability, and/or stability.	•A performance test is a technical investigation done to determine or validate the responsiveness, speed, scalability, and/or stability characteristics of the product under test.

The most common performance concerns related to Web applications are
"Will it be fast enough?"
"Will it support all of my clients?"
"What happens if something goes wrong?"
"What do I need to plan for when I get more customers?


When 1000 users call the API within a timeframe of 180s seconds, the average response time should be below 250ms, the maxium response time should be below 500ms and no errors should occur
“accommodate the current/expected user base
load testing
“something going wrong”
stress testing
“planning for future growth”
capacity testing
When 1000 users call the API within 30s seconds, the average response time should be below 1000ms and no errors should occur

Load test	To verify application behavior under normal and peak load conditions.	•Load testing is conducted to verify that your application can meet your desired performance objectives; these performance objectives are often specified in a service level agreement (SLA). •A load test enables you to measure response times, throughput rates, and resource-utilization levels, and to identify your application's breaking point, assuming that the breaking point occurs below the peak load condition. •Endurance testing is a subset of load testing. An *endurance test* is a type of performance test focused on determining or validating the performance characteristics of the product under test when subjected to workload models and load volumes anticipated during production operations over an extended

Stress test	To determine or validate an application's behavior when it is pushed beyond normal or peak load conditions.	•The goal of stress testing is to reveal application bugs that surface only under high load conditions. These bugs can include such things as synchronization issues, race conditions, and memory leaks. Stress testing enables you to identify your application's weak points, and shows how the application behaves under extreme load conditions. •Spike testing is a subset of stress testing. A *spike test* is a type of performance test focused on determining or validating the performance characteristics of the product under test

Capacity test	To determine how many users and/or transactions a given system will support and still meet performance goals.	•Capacity testing is conducted in conjunction with capacity planning, which you use to plan for future growth, such as an increased user base or increased volume of data. For example, to accommodate future loads, you need to know how many additional resources (such as processor capacity, memory usage, disk capacity, or network bandwidth) are necessary to support future usage levels. •Capacity testing helps you to identify a scaling strategy in order to determine whether you should scale up or scale out.

More Concepts

Smoke test	A *smoke test* is the initial run of a performance test to see if your application can perform its operations under a normal load.
Validation test	A *validation test* compares the speed, scalability, and/or stability characteristics of the product under test against the expectations that have been set or presumed for that product.

API Testing Best Practices

1. Test for the typical or expected results first
2. Add stress to the system through a series of API load tests
3. Test for failure. Make sure you understand how your API will fail. Just make sure the API fails consistently and gracefully
4. Group test cases by test category
5. Prioritize API function calls so that it will be easy for testers to test quickly and easily
6. Limit the tests from as many variables as possible by keeping it as isolated as possible
7. See how it handles unforeseen problems and loads by throwing as much as you can at it
8. Perform well-planned call sequencing
9. For complete test coverage, create test cases for *all* possible API input combinations
10. Automate wherever you can

https://www.soapui.org/learn/functional-testing/api-testing-101/

Shift Left Testing

Disadvantages of Traditional Testing

- Testing is at extreme right. A lot of costs are incurred when a bug is identified at the last minute.
- Time consumed in resolving the bug and retesting it before promoting it to production is huge.

Requirement ← Shift Left ← Testing Phase

Design ← Shift Left ← Testing Phase

Coding ← Shift Left ← Testing Phase

Production Deployment ← Shift Left ← Testing Phase

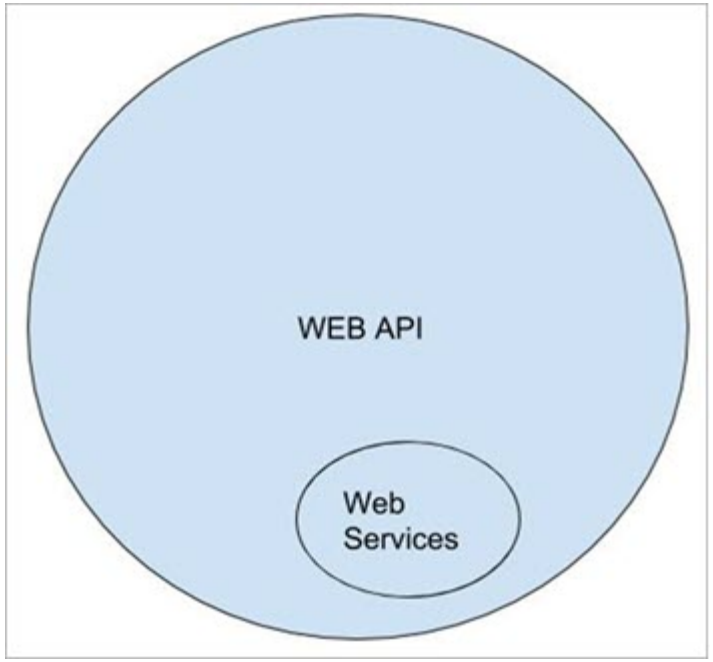
Web API vs Web Services

AP I

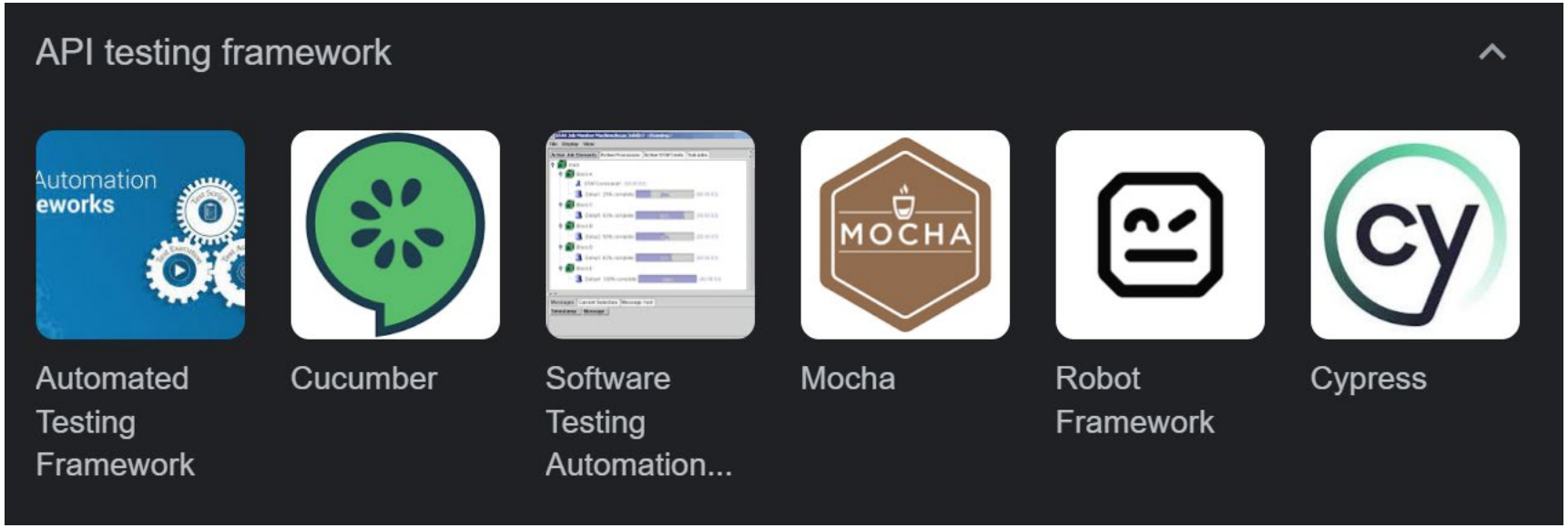
- All APIs are not web services
- API can be hosted within the application or web server
- May use any style of communication

Web Services

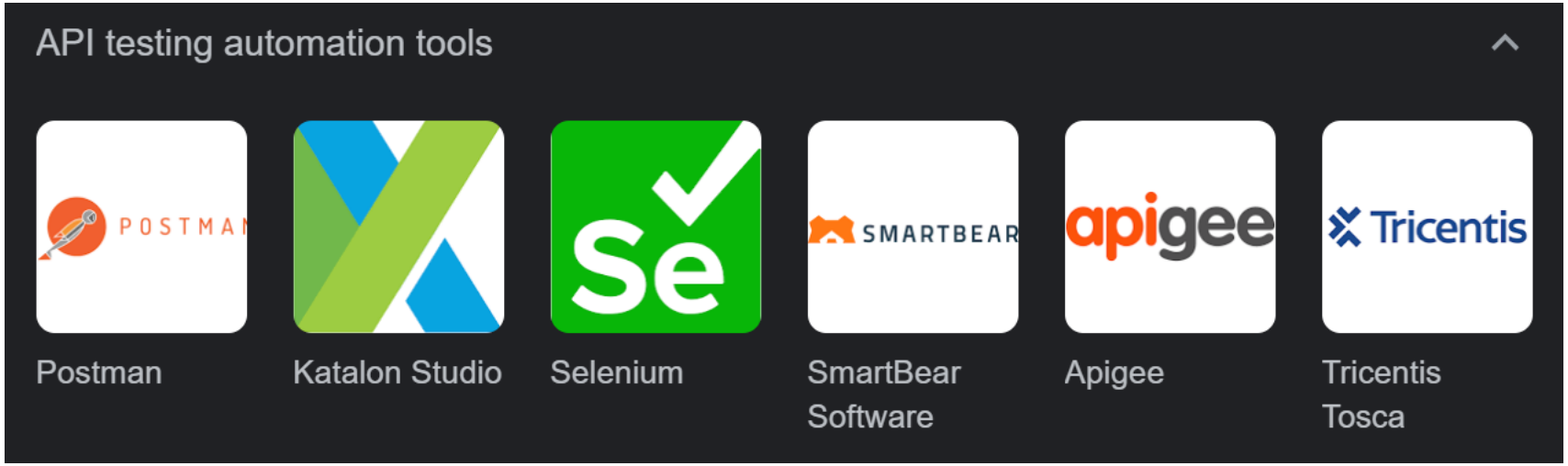
- All web services are APIs
- Can only be hosted in web servers
- Needs network to communicate
- Use limited communications like SOAP, REST,…



API Testing Frameworks



API Testing Tools



Performance testing Tools

- **Apache JMeter™ —** open-source software, Java-based, very popular. It has a GUI but also a command-line tool that can use used to automate the process. The UI is not the most modern or intuitive. Learn more at https://jmeter.apache.org/
- **Gatling** — is slightly newer than JMeter and benefits from a much nicer UI. It also comes from the Java space. Gatling offers a free open-source version but also an enterprise version. Learn more at https://gatling.io/.

Better than postman!!!!!!!!!
Why?

API Testing Using POSTMAN

- POSTMAN is an API client used to develop, test, share and document APIs.
- It is used for backend testing where we enter the end-point URL, it sends the request to the server and receives the response back from the server.

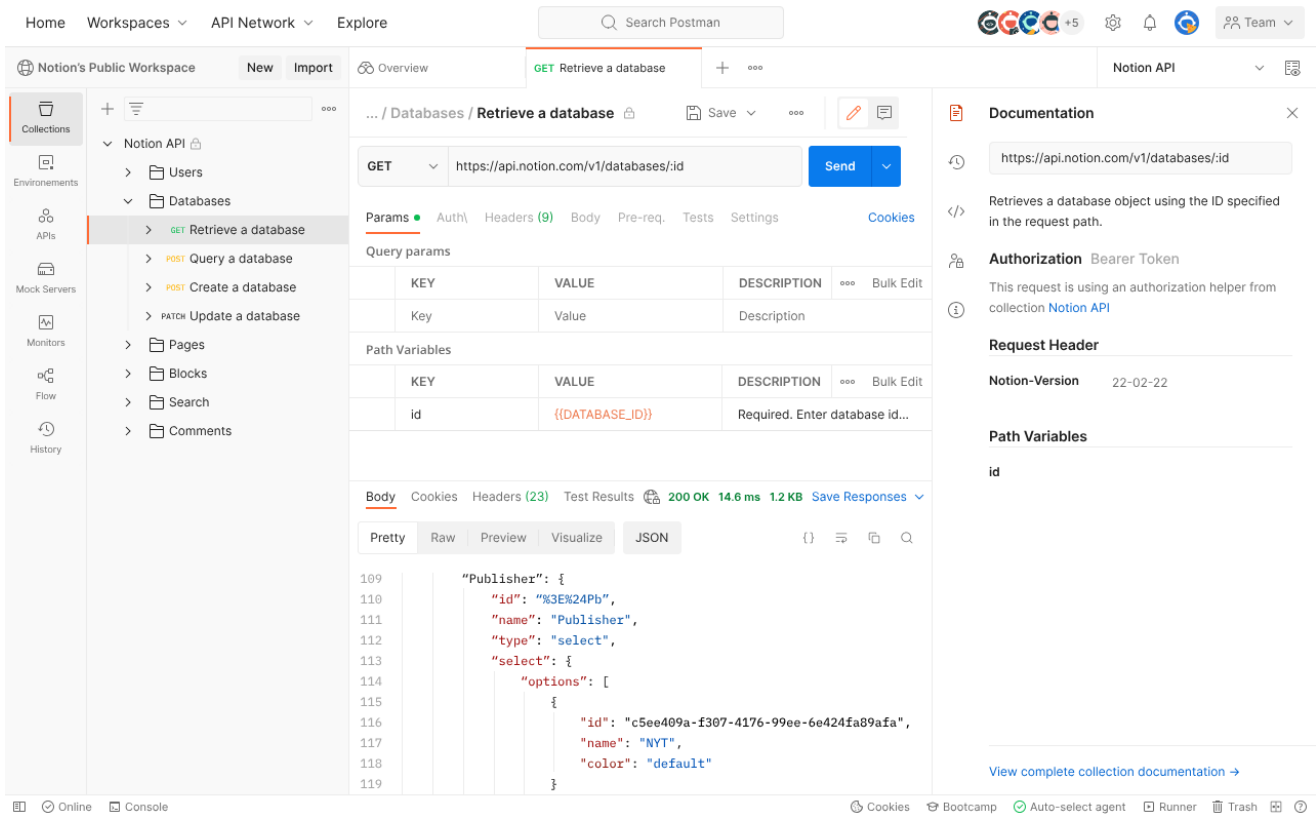
Features of Postman

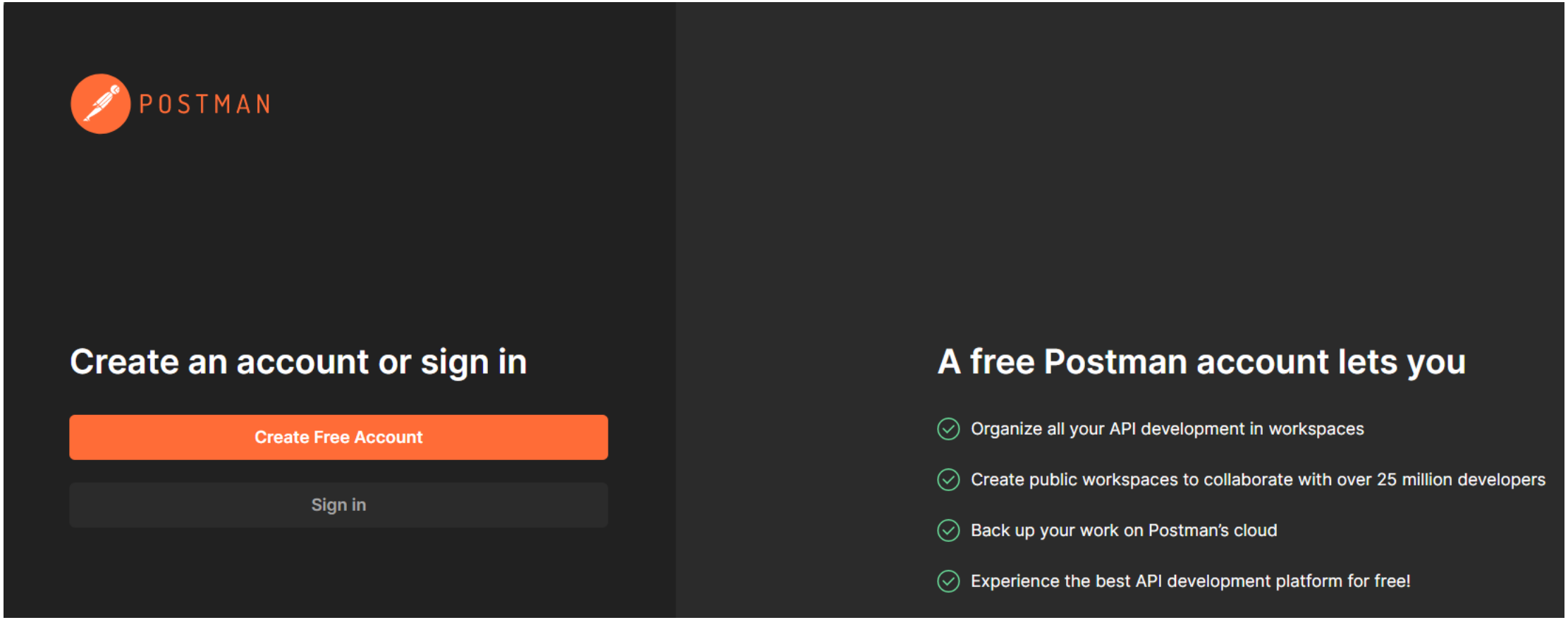
Postman offers a lot of advanced features like:

- API development.
- Setting up Mock endpoints for APIs that are still under development.
- API documentation.
- Assertions for the responses received from API endpoint execution.
- Integration with CI-CD tools like Jenkins, TeamCity, etc.
- Automating API tests execution etc.

Postman Native App

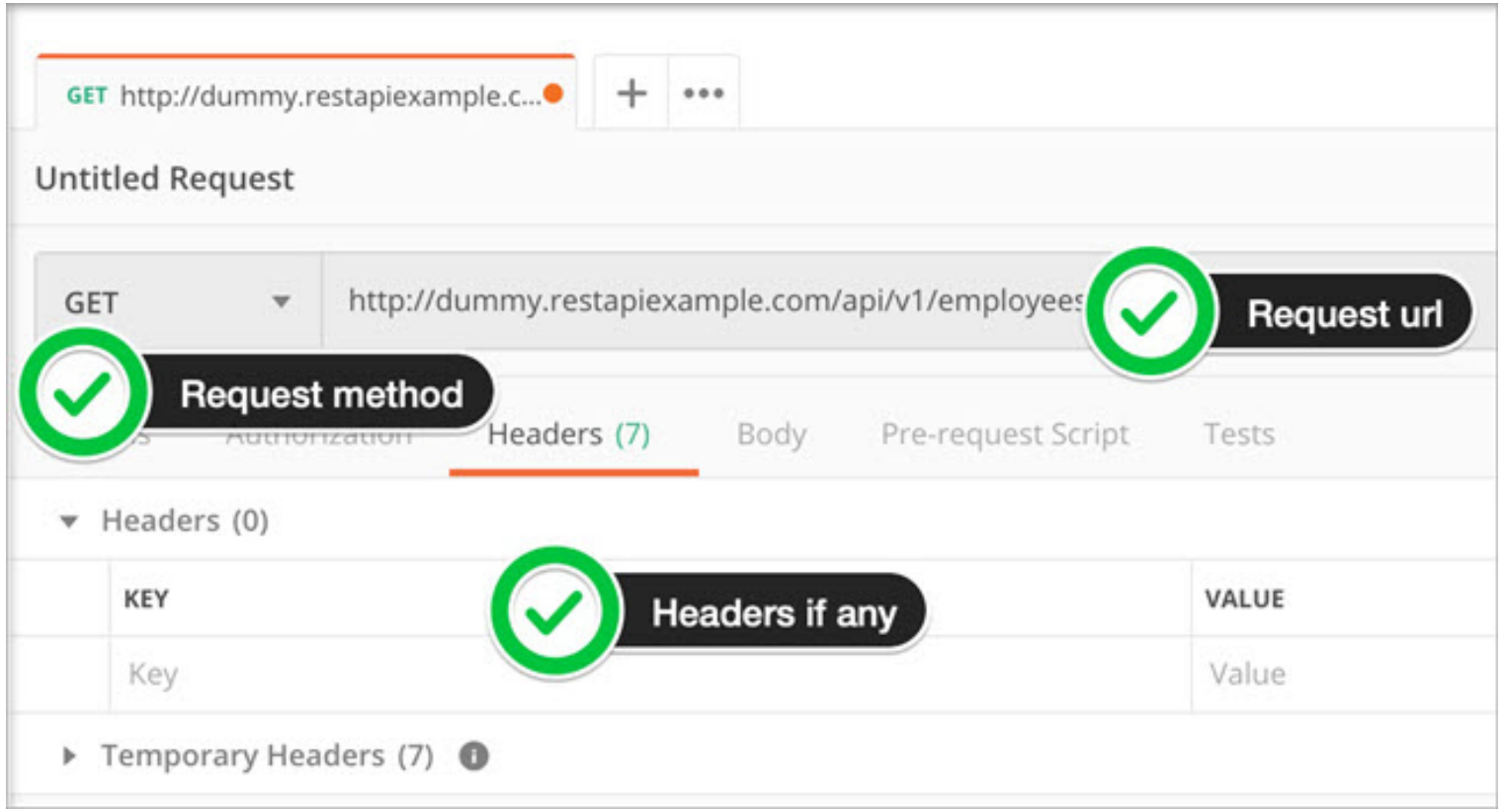
- Download site
  - https://www.postman.com/downloads/

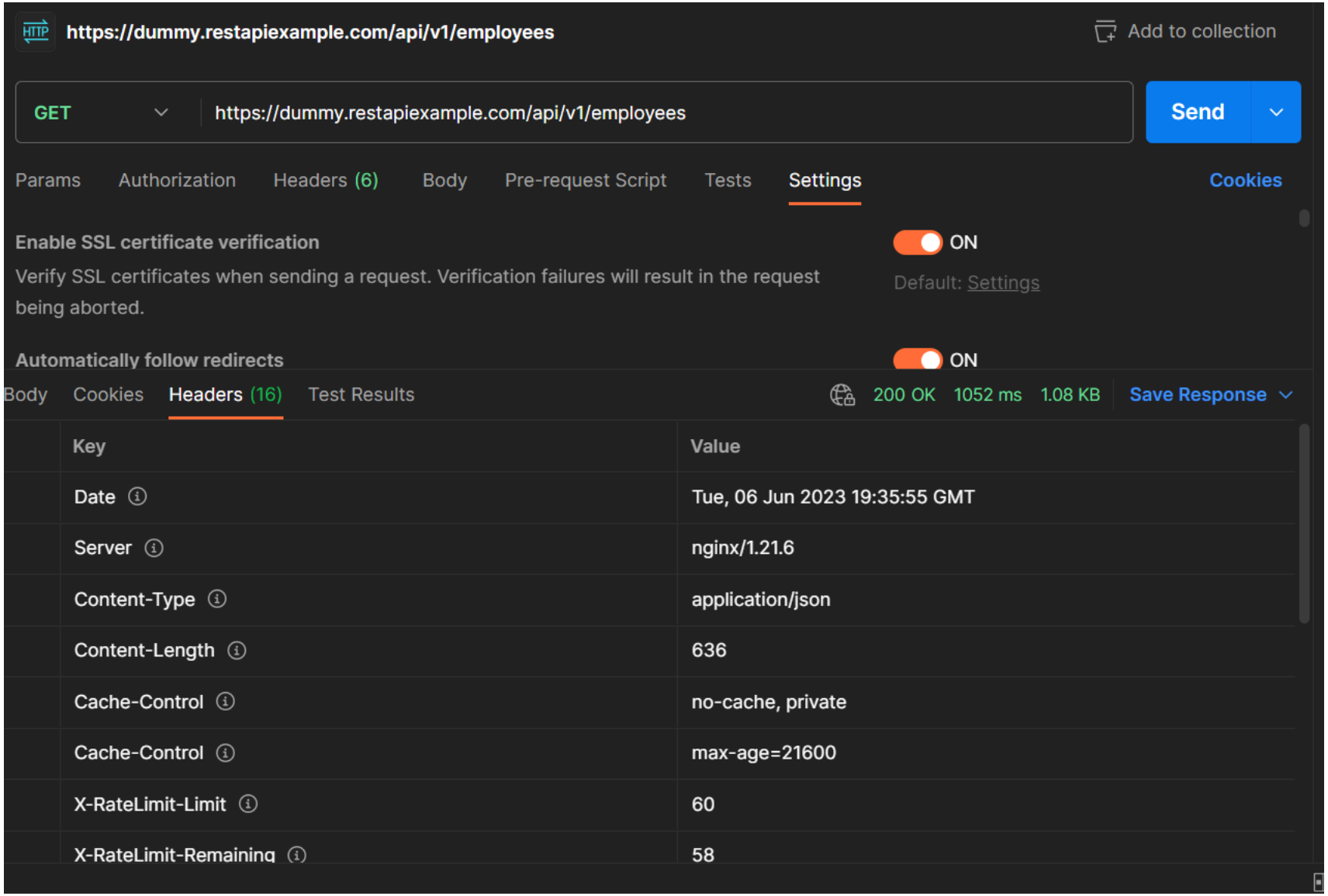



POSTMAN
Create an account or sign in
Create Free Account
Sign in
A free Postman account lets you
Organize all your API development in workspaces
Create public workspaces to collaborate with over 25 million developers
Back up your work on Postman's cloud
Experience the best API development platform for free!

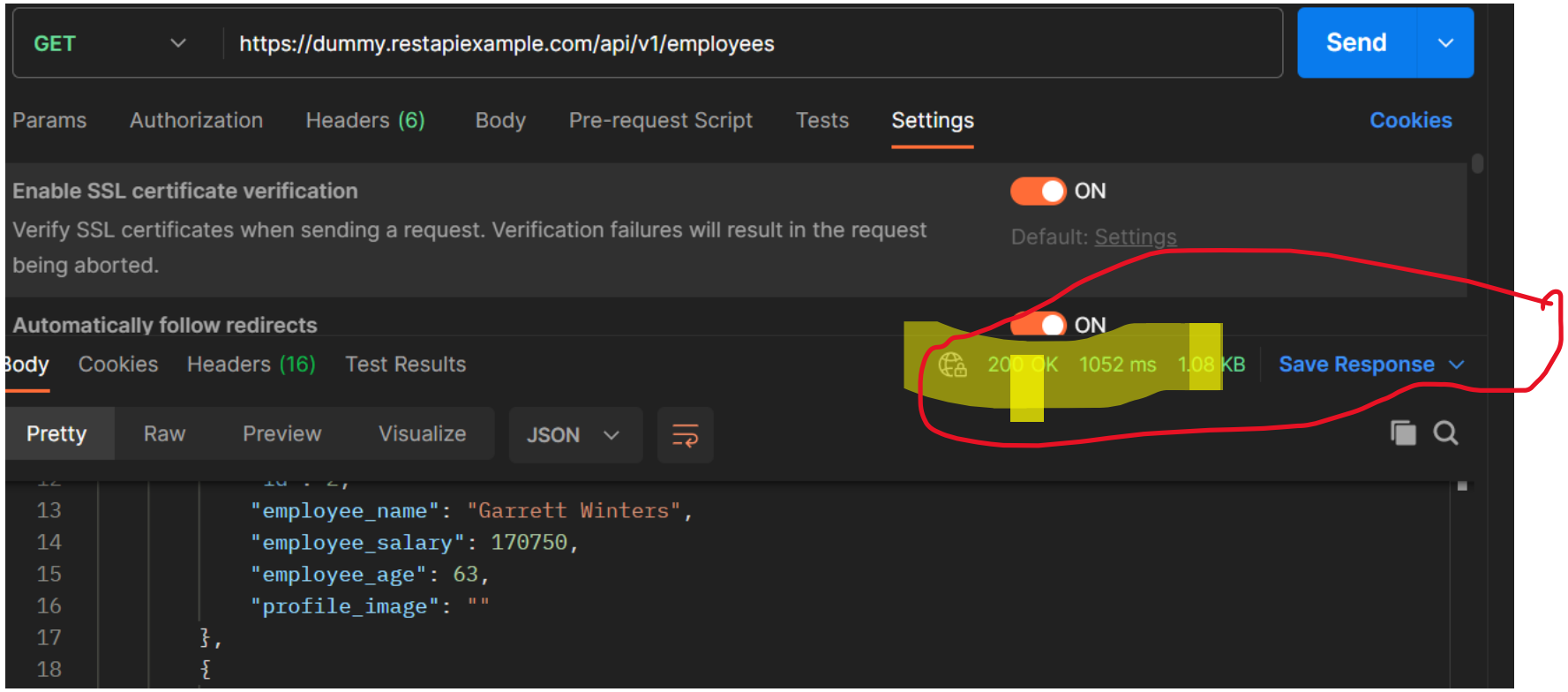
First Experience

- https://dummy.restapiexample.com/api/v1/employees



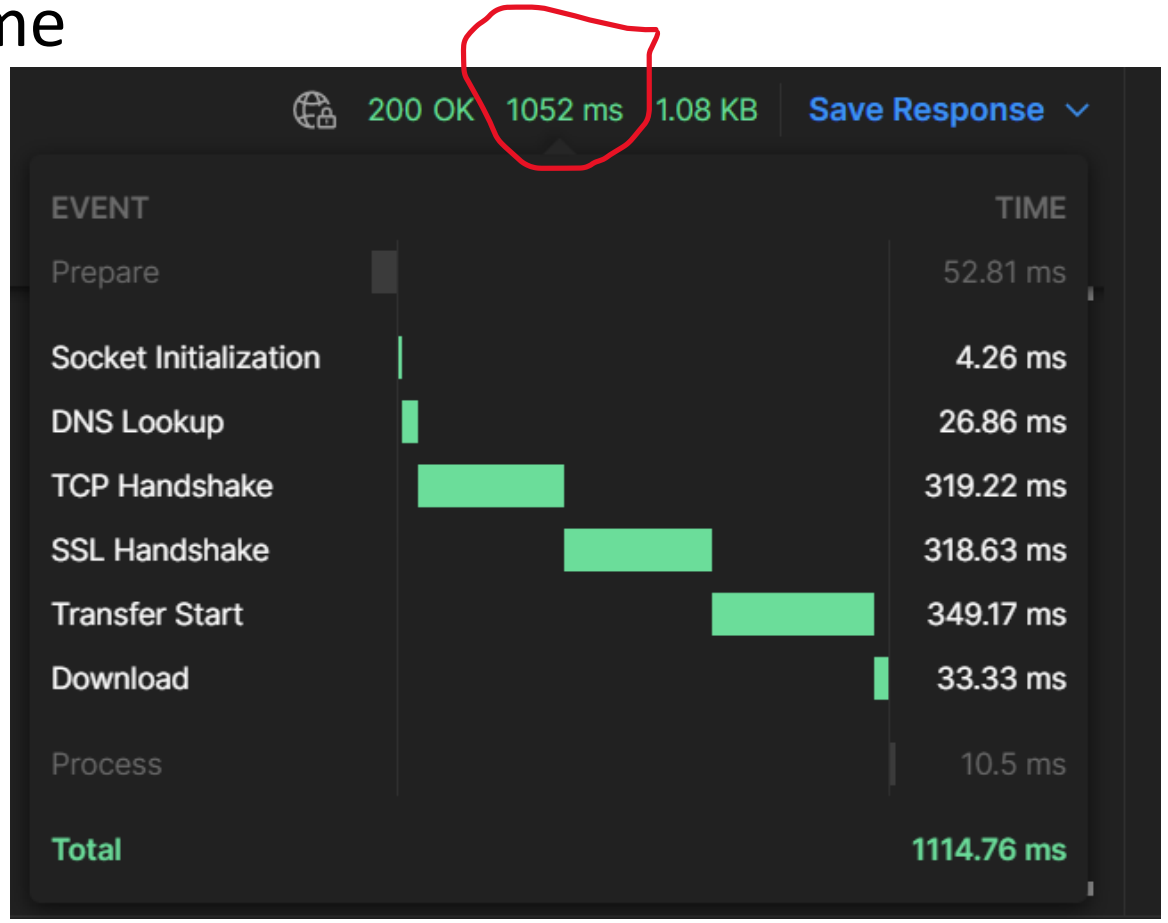

https://dummy.restapiexample.com/api/v1/employees
Add to collection
GET
https://dummy.restapiexample.com/api/v1/employees
Send
Params
Authorization
Headers (6)
Body
Pre-request Script
Tests
Settings
Cookies
Enable SSL certificate verification
ON
Verify SSL certificates when sending a request. Verification failures will result in the request being aborted.
Default: Settings
Automatically follow redirects
ON
Body
Cookies
Headers (16)
Test Results
200 OK
1052 ms
1.08 KB
Save Response
Key
Value
Date
Tue, 06 Jun 2023 19:35:55 GMT
Server
nginx/1.21.6
Content-Type
application/json
Content-Length
636
Cache-Control
no-cache, private
Cache-Control
max-age=21600
X-RateLimit-Limit
60
X-RateLimit-Remaining
58

Response Stats

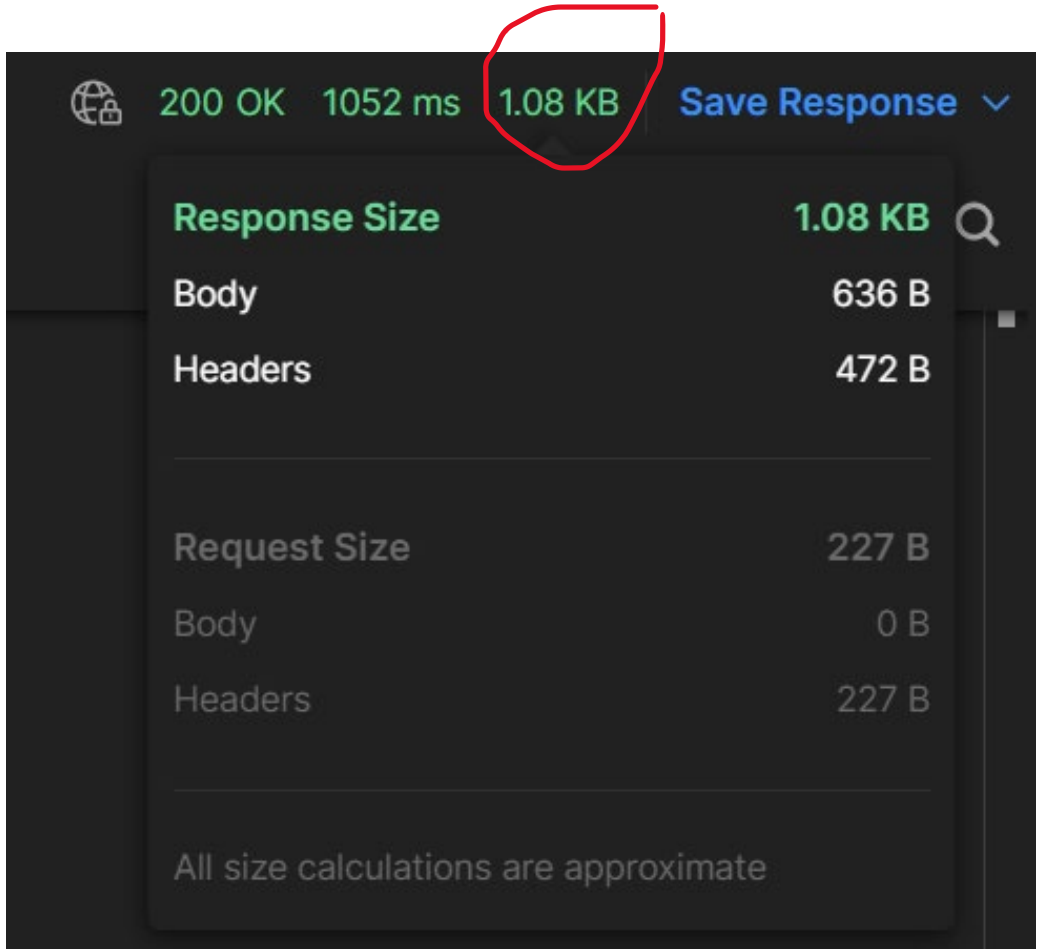


Request time details

- Click on response time



Response size



Building Blocks Of POSTMAN

These three major building blocks are:

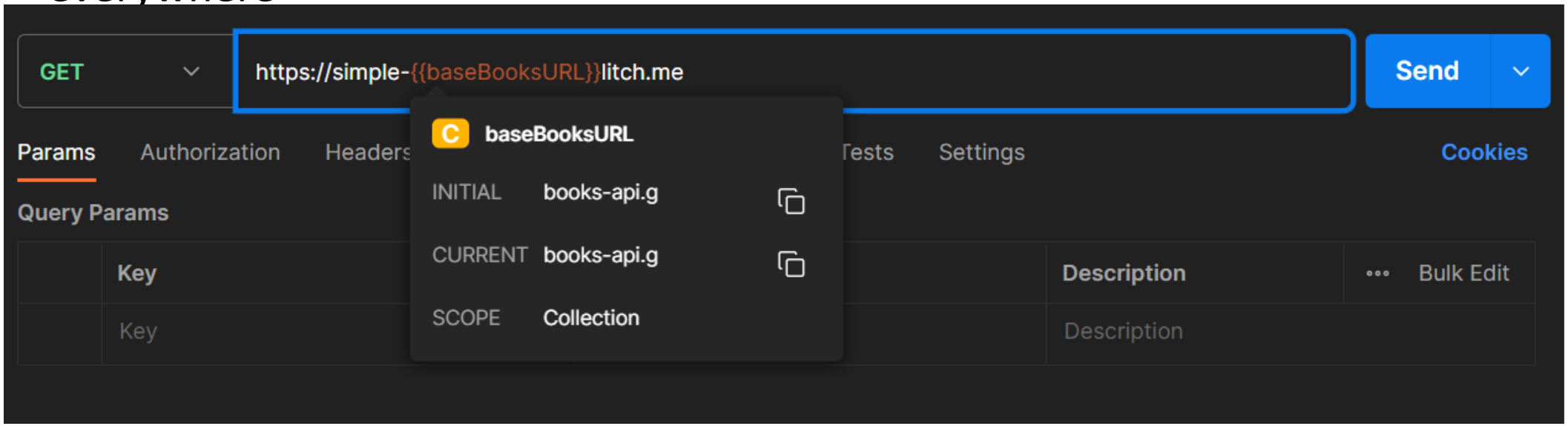
#1) Request

#2) Collection

#3) Environment

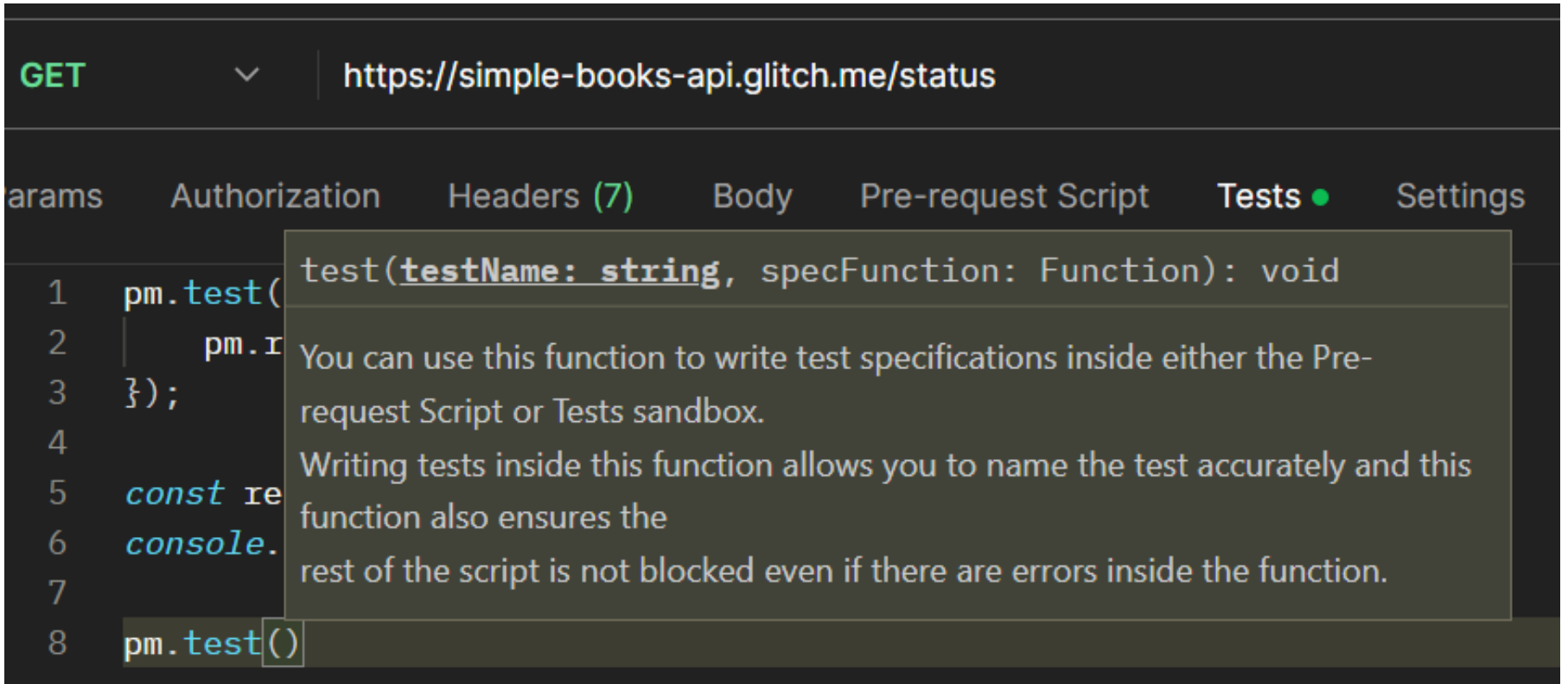
Collection

- A collection of requests
- Variables
  - baseUrl in case we change it in the future, we don’t have to change it everywhere



Postman Automation Testing

- Pm.test(testname, callback function);



Example

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

Example

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

pm.test("Testing status is OK", () =>{

})
```

Variables

Postman offers 5 different types of variable scopes as stated below:

1.Global

2.Collection

3.Environment

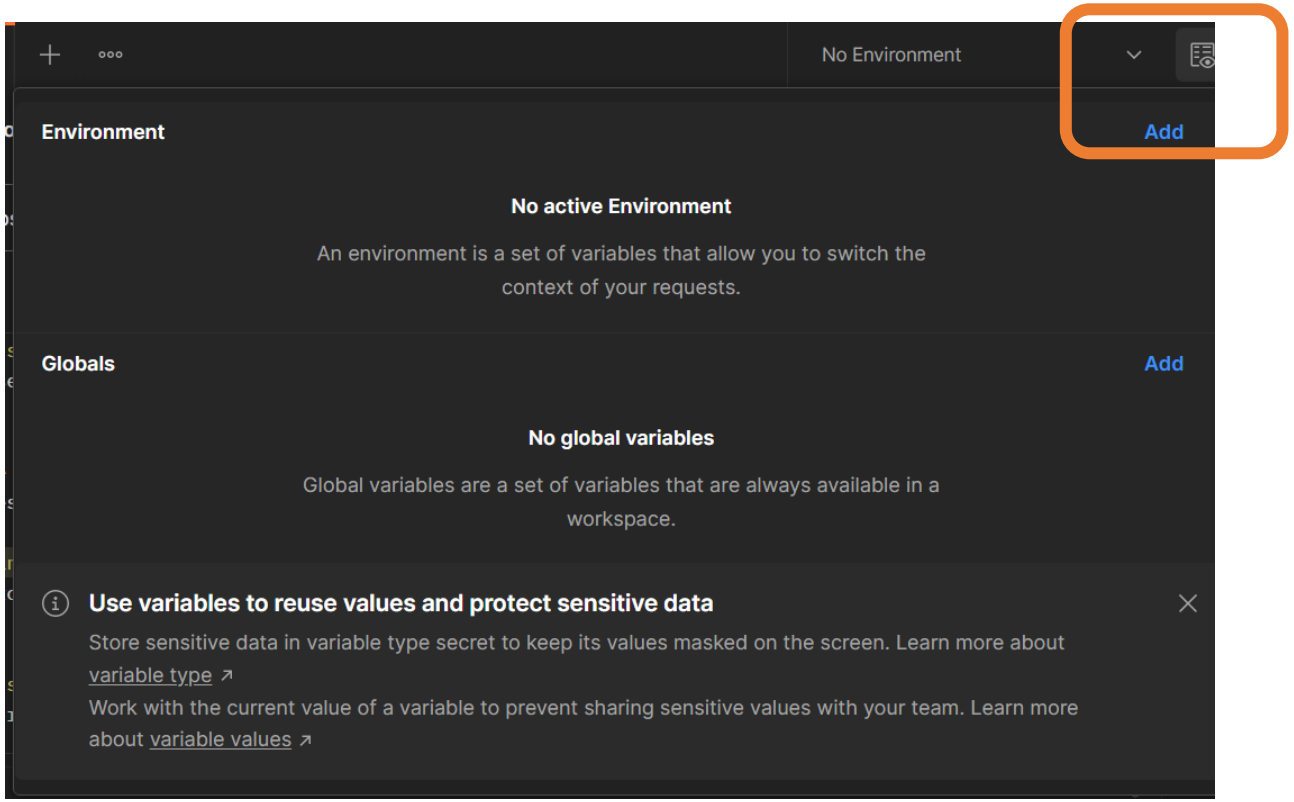
4.Data

5.Local

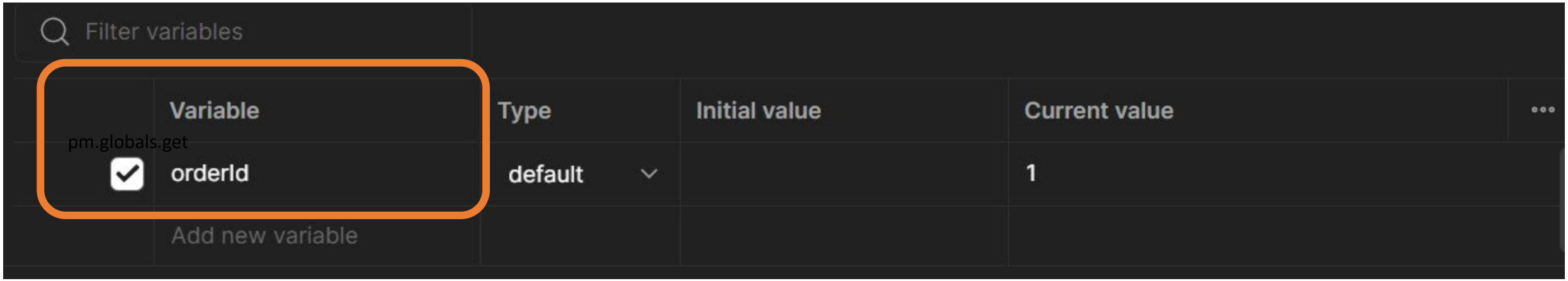
Global

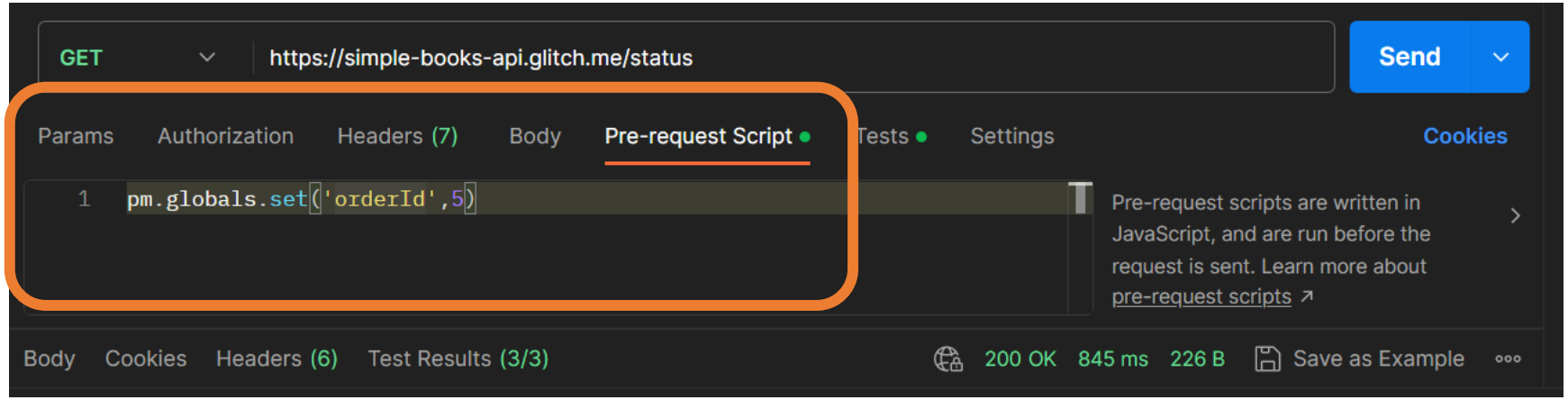
- Global variables are general-purpose variables and should be mostly avoided and used only for quick prototyping requirements.

The initial value is something that is persisted by default for that variable and the current value keeps changing as it is set or updated in the requests that are using these variables.



Global Variables : define and use

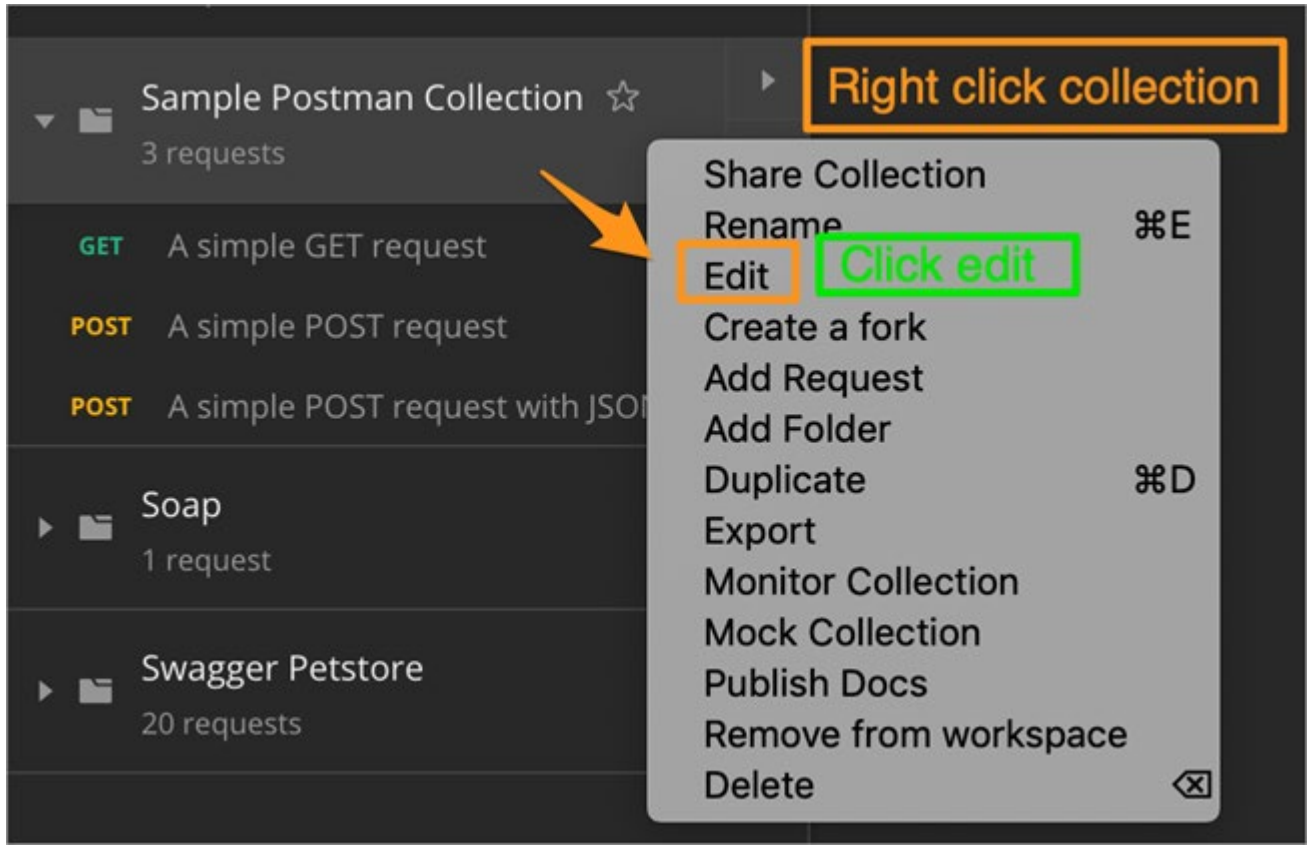




Collection Variables

- Collection variables are used to define variables at the collection scope.
- Collection variables do not change during the execution of a collection or request inside the given collection. Essentially Collection variables could be just retrieved and not updated during request execution

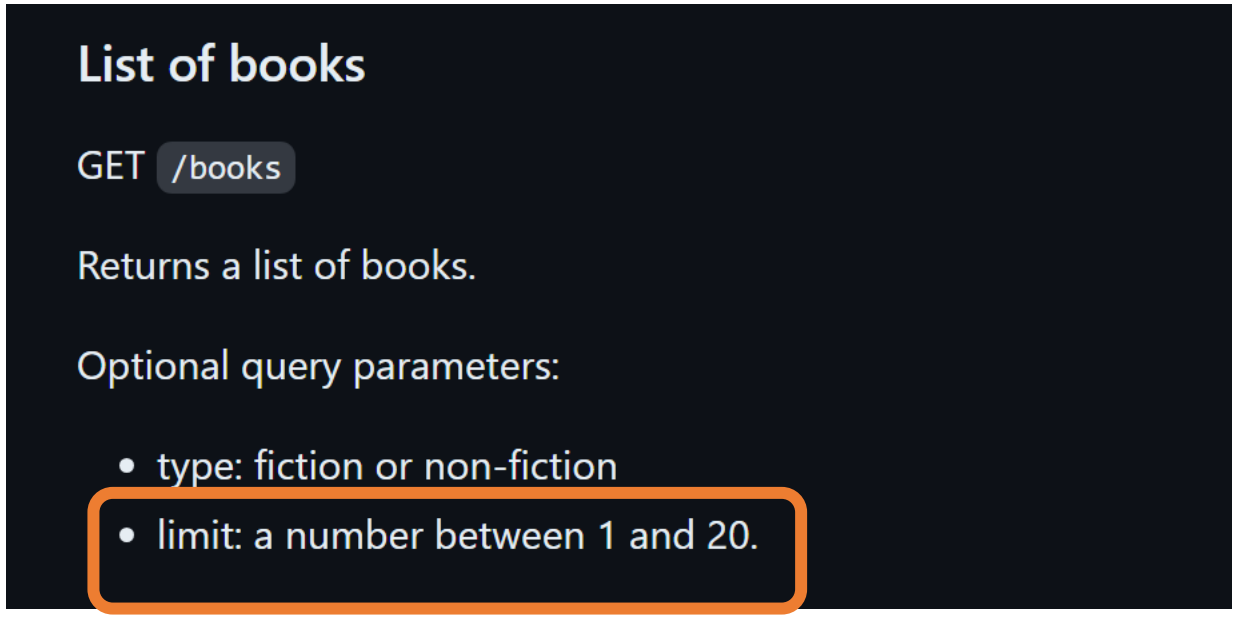
Set

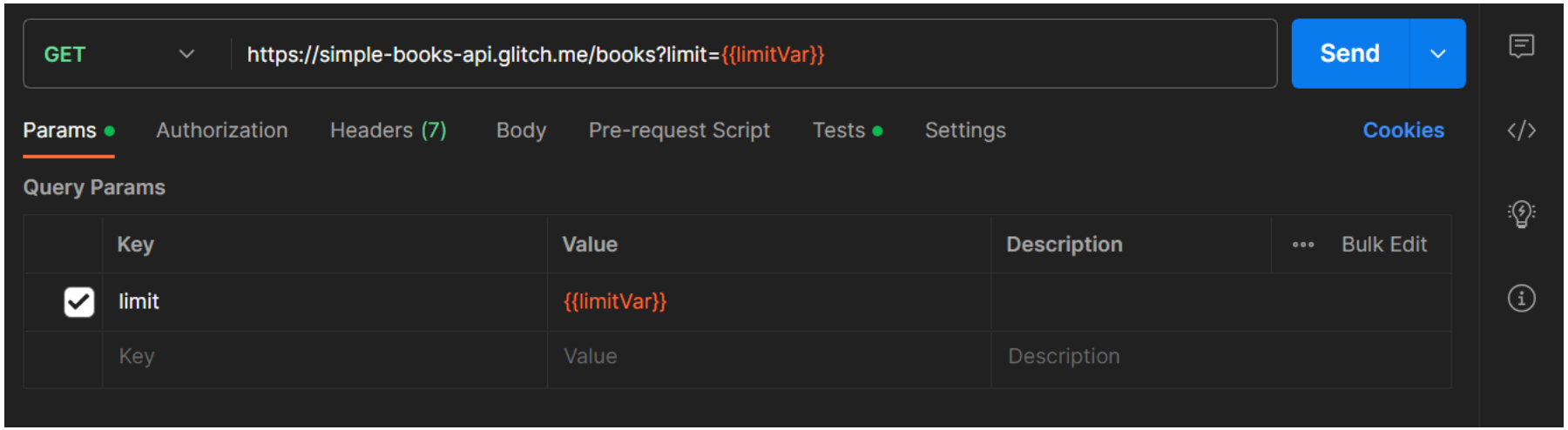

Sample Postman Collection
3 requests
Right click collection
GET A simple GET request
POST A simple POST request
POST A simple POST request with JSO
Soap
1 request
Swagger Petstore
20 requests
Share Collection
Rename ⌘E
Edit
Click edit
Create a fork
Add Request
Add Folder
Duplicate ⌘D
Export
Monitor Collection
Mock Collection
Publish Docs
Remove from workspace
Delete

https://github.com/vdespa/introduction-to-postman-course/blob/main/simple-books-api.md

Exercise

- Use collection variables to set the limit





- https://simple-books-api.glitch.me/books

- https://simple-books-api.glitch.me/books/1

Environment Variables

- Environment variables are the most heavily used kind of variables in Postman.
- They are tied to a selected environment that's being used for executing the request. They have a narrower scope than the Global variables but broader than the Collection variables.

When **there is a need for passing data or information from one request to another, environment variables are a good choice**, as they have a broader scope than the Local variables and narrower scope than Global variables

Using Environment variables

In order to use Environment variables through the script, you can use **pm.environment.get** and **pm.environment.set** to fetch and add/modify environment variables respectively.

Example:

```
pm.environment.get('testEnvVar’)
pm.environment.set('testEnvVar',10)
```

Local Variables

- One important use case of local variables is that they can be used when you want to override the values of a variable that is defined in any other scope like Global, Collection or Environment

- ***pm.variables.get / pm.variables.set***

Data Variables

- Postman allows us to execute requests in a collection through the collection runner and while execution we can provide a data set in the form of JSON or CSV that are used while running the requests inside the collection.
- It’s important to note here that the source of Data variables is the user-supplied data file in the format of JSON or CSV, and during the request execution, the Data variables can only be fetched but not updated/modified or added.

Test script examples

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

The function inside the test represents an assertion. Postman tests can use Chai Assertion Library BDD syntax, which provides options to optimize how readable your tests are to you and your collaborators. In this case, the code uses BDD chains `to.have` to express the assertion.

```
pm.test("Status code is 200", () => {
pm.expect(pm.response.code).to.eql(200); });
```

BDD

The BDD styles are expect and should. Both use the same chainable language to construct assertions, but they differ in the way an assertion is initially constructed..

API Reference

Language Chains

The following are provided as chainable getters to improve the readability of your assertions.

Chains

- to
- be
- been
- is
- that
- which
- and
- has
- have
- with
- at
- of
- same
- but
- does
- still
- also

Using multiple assertions

```
pm.test("The response has all properties", () => { //parse
the response JSON and test three properties const
responseJson = pm.response.json();
pm.expect(responseJson.type).to.eql('vip');
pm.expect(responseJson.name).to.be.a('string');
pm.expect(responseJson.id).to.have.lengthOf(1);
 });
```

- If any of the contained assertions fails, the test as a whole will fail. All assertions must be successful for the test to pass.

Parsing response body data

- To carry out assertions on your responses, you will first need to parse the data into a JavaScript object that your assertions can use.
- To parse JSON data, use the following syntax:

JSON

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

XML

```
const responseJson = xml2Json(pm.response.text());
```

CSV

```
const parse = require('csv-parse/lib/sync');
const responseJson = parse(pm.response.text());
```

HTML

```
const $ = cheerio.load(pm.response.text());
//output the html for testing console.log($.html());
```

Handling responses that don't parse

- If you can't parse the response body to JavaScript because it's not formatted as JSON, XML, HTML, CSV, or any other parsable data format, you can still make assertions on the data.
- Test if the response body contains a string

```
pm.test("Body contains string",() => {
pm.expect(pm.response.text()).to.include("customer_id");
 });
```

Making assertions on the HTTP response

- Your tests can check various aspects of a request response, including the body, status codes, headers, cookies, response times, and more.

Testing response body

Check for particular values in the response body:

```
pm.test("Person is Jane", () => { const responseJson =
pm.response.json();
pm.expect(responseJson.name).to.eql("Jane");
pm.expect(responseJson.age).to.eql(23); });
```

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

```
pm.test("Successful POST request", () => {
pm.expect(pm.response.code).to.be.oneOf([201,202]); });
```

Testing headers

- Check that a response header is present:

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

Testing cookies

- Test if a cookie is present in the response:

```
pm.test("Cookie JSESSIONID is present", () => {
pm.expect(pm.cookies.has('JSESSIONID')).to.be.true; });
```

Testing response times

- Test for the response time to be within a specified range:

```
pm.test("Response time is less than 200ms", () => {
pm.expect(pm.response.responseTime).to.be.below(200); });
```

Asserting a response value against a variable

- Check if a response property has the same value as a variable (in this case an environment variable):

```
pm.test("Response property matches environment variable", function () {
pm.expect(pm.response.json().name).to.eql(pm.environment.get("name"));
});
```

Asserting a value type

```
/* response has this structure: { "name": "Jane",
"age": 29, "hobbies": [ "skating", "painting" ],
"email": null } */
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.age).to.be.a("number");
pm.expect(jsonData.hobbies).to.be.an("array");
pm.expect(jsonData.website).to.be.undefined;
pm.expect(jsonData.email).to.be.null;
});
```

Asserting array properties

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

```
/* response has this structure: { "errors": [], "areas": [ "goods", "services" ], "settings":
[ { "type": "notification", "detail": [ "email", "sms" ] }, { "type": "visual", "detail": [
"light", "large" ] } ] } */
const jsonData = pm.response.json();
pm.test("Test array properties", () => { //errors array is empty
pm.expect(jsonData.errors).to.be.empty; //areas includes "goods"
pm.expect(jsonData.areas).to.include("goods"); //get the notification settings object
const notificationSettings = jsonData.settings.find (m => m.type === "notification");
pm.expect(notificationSettings) .to.be.an("object", "Could not find the setting"); //detail
array must include "sms" pm.expect(notificationSettings.detail).to.include("sms"); //detail
array must include all listed pm.expect(notificationSettings.detail)
.to.have.members(["email", "sms"]); });
```

Asserting that a value is in a set

- Check a response value against a list of valid options

```
pm.test("Value is in valid list", () => {
pm.expect(pm.response.json().type)
.to.be.oneOf(["Subscriber", "Customer", "User"]);
});
```

Asserting the current environment

- Check the active (currently selected) environment in Postman:

```
pm.test("Check the active environment", () => {
pm.expect(pm.environment.name).to.eql("Production");
});
```

Exercise : passing data between requests?

- Use http://httpbin.org/uuid generate a unique id
- Sample response

```
{ "uuid": "d62afd2d-8465-413a-94bf-b687ea8ef581" }
```

- Assign this id to inserting a new book

```
POST /orders/
Authorization: Bearer <YOUR TOKEN>
{
"bookId": 1,
"customerName": "John"
}
```

-

Follow the instruction on this page

https://github.com/vdespa/introduction-to-postman-course/blob/main/simple-books-api.md

References

- Variables:
  - https://www.softwaretestinghelp.com/postman-variables/
- https://manage.exchangeratesapi.io/dashboard

References

- https://www.softwaretestinghelp.com/api-testing-tutorial/