

# Web API testing

## 1. Contents

1. Contents .....	1
1.1. Table of figures.....	2
2. Introduction .....	3
3. What is a Web API? .....	3
3.1. REST API.....	3
3.2. The key elements of a RESTful implementation.....	3
3.3. Restful Methods .....	4
4. The chosen API .....	4
5. Testing methodology .....	5
5.1. Postman.....	5
6. Testing.....	5
6.1. Parameters of the request .....	8
6.2. Variables in Postman .....	8
6.3. Flows (request chains).....	10
6.4. Results .....	11
7. Some insight.....	13
8. Conclusions.....	13
9. References.....	13

## 1.1. Table of figures

Figure 4.1: An example response from the API.....	4
Figure 6.1: Overview screen of Postman. ....	6
Figure 6.2: Collections and subfolders in Postman. ....	6
Figure 6.3: Creating a request in Postman. ....	7
Figure 6.4: Example response displayed in Postman. ....	7
Figure 6.5: Postman test's execution order. ....	7
Figure 6.6: Example test script. ....	8
Figure 6.7: Request parameters. ....	8
Figure 6.8: Variable scopes. ....	9
Figure 6.9: Example usage of variables. ....	9
Figure 6.10: Mock data using Faker library. ....	9
Figure 6.11: An example flows. ....	10
Figure 6.12: Example SQL Injection. ....	10
Figure 6.13: Example for destructive SQL Injection. ....	11
Figure 6.14: Check if data base is empty after SQL Injection. ....	11
Figure 6.15: Test results overview. ....	12
Figure 6.16: Newman results in the terminal. ....	12
Figure 6.17: GitHub worklow Yamle file. [6].....	13

## 2. Introduction

The term Web Service is representing a service offered by an electronic device to another electronic device while communicating with each other via the World Wide Web. These services usually run on a computer or a big server farms and their main job [1] is to serv web documents (like HTML, CSS, JSON, XNL, etc.) or solving big computational problems offloaded by other lower powered devices.

Nowadays everything is done on the so called "Cloud". In today technology stack web services are essential part of the system, and because of this the testing of these services is rather important.

In this document we are going to discuss how to test web APIs using postman.

## 3. What is a Web API?

There are multiple standards for creating a web service or a web API, like SOAP, WSD, etc. In this document we are going to focus mainly on the REST standard, but the principles are going to be the same with some minimal differences. [2]

In 2004, the web consortium also released the definition of an additional standard called RESTful. Over the past couple of years, this standard has become quite popular. And is being used by many of the popular websites around the world which include Facebook and Twitter. [2]

### 3.1. REST API

Restful Web Services is a lightweight, maintainable, and scalable service that is built on the REST architecture. Restful Web Service expose API from your application in a secure, uniform, stateless manner to the calling client. The calling client can perform predefined operations using the Restful service. **REST** stands for **representational state transfer**. The underlying protocol for REST is HTTP, this means that it uses the same GET, POST, PUT methods and status codes like 200 and 404. [2]

### 3.2. The key elements of a RESTful implementation

1. **Resources** – The most important part of a REST API is the resource we are trying to access. This can be a database of employee information, a collection of photos or even a bunch of printers and scanners we are trying to access remotely.
2. **Request Verbs** - These are the access methods of the HTTP protocol; they describe what you want to do with the resource. A browser issues a GET verb to instruct the endpoint it wants to get data. However, there are many other verbs available including things like POST, PUT, and DELETE.
3. **Request Headers** – These are additional instructions sent with the request. These might define the type of response required or the authorization details.
4. **Request Body** - Data is sent with the request. Data is normally sent in the request when a POST request is made to the REST web services. In a POST call, the client tells the REST web services that it wants to add a resource to the server. Hence, the request body would have the details of the resource which is required to be added to the server.
5. **Response Body** – This is the main body of the response, containing the data of our request. This may be in multiple formats like XML, JSON or even plain text.

6. **Response Status codes** – These codes are the general codes which are returned along with the response from the web server, like code 200 which is normally returned if there is no error, 404 if the resource is not found or 503 if the server ran into a problem while fetching the resource.

### 3.3. Restful Methods

1. **POST** – This would be used to add a new resource.
2. **GET** - This would be used to get a list of all resources or a specific resource.
3. **PUT** - This would be used to update the resource.
4. **DELETE** - This would be used to delete a resource.

## 4. The chosen API

The REST API we are going to test is made by me for an Android Project<sup>1</sup> and has a public source code found on GitHub: <https://github.com/hammasattila/Sapientia-2020-Android-RestAPI>. Right now, you can access it at: <https://ratpark-api.imok.space/>.

The resource that we can access through this API is a dummy database of restaurants. The format used in the response body is JSON and for simplicity we only can use GET methods.

An example response can be seen below:

```
{
  "id": 107257,
  "name": "Las Tablas Colombian Steak House",
  "address": "2942 N Lincoln Ave",
  "city": "Chicago",
  "state": "IL",
  "area": "Chicago / Illinois",
  "postal_code": "60657",
  "country": "US",
  "phone": "7738712414",
  "lat": 41.935137,
  "lng": -87.662815,
  "price": 2,
  "reserve_url": "http://www.opentable.com/single.aspx?rid=107257",
  "mobile_reserve_url": "http://mobile.opentable.com/opentable/?restId=107257",
  "image_url": "https://www.opentable.com/img/restimages/107257.jpg"
}
```

Figure 4.1: An example response from the API

This response shown in Figure 4.1 is given for the following request:

[GET] <https://ratpark-api.imok.space/restaurants/107257>

---

<sup>1</sup> <https://github.com/hammasattila/Sapientia-2020-Android-Project>

## 5. Testing methodology

API testing is a type of software testing that involves testing application programming interfaces (APIs) directly and as part of integration testing to determine if they meet expectations for functionality, reliability, performance, and security. Since APIs lack a GUI (Graphical User Interface), API testing is performed at the message layer. API testing is now considered critical for automating testing because APIs now serve as the primary interface to application logic [3].

During API testing the main goal is to check whether the API gives the correct response to a given input (including edge cases such as failures and unexpected and extreme inputs). Another use for API testing is to check the performance (deliver responses in an acceptable amount of time) and the security of the API (whether the service is vulnerable for certain attacks, for example: SQL Injection<sup>2</sup> in case of SQL database resource).

### 5.1. Postman

A well-known framework for testing web APIs which use HTTP requests is Postman<sup>3</sup>. It allows the user to execute all kinds of requests including GET, POST, PUT and DELETE and much more. This tool allows the user to create automated tests. Postman uses [Chai.js](https://www.npmjs.com/package/chai)<sup>4</sup> assertion library which has several interfaces including a BDD<sup>5</sup> styles which provide an expressive language & readable style and a more classical TDD<sup>6</sup> style. As for mocking data, Postman includes the [Faker.js](https://www.npmjs.com/package/faker)<sup>7</sup> library to generate dummy data, this can generate random names, addresses, email addresses, and much more.

This framework is usable straight from the browser (requires an account) or it is available as a native app on Windows, Mac, and Linux. [4]

## 6. Testing

On the first launch of Postman, it greets you with a login screen. If it is used in a web browser an account is required, meanwhile in the native app this can be skipped. After login or skipping login Postman shows an “Overview screen”, which looks like the screen shown in Figure 6.1 (if you skipped the login, then some buttons and features maybe are not shown in the app). This overview page shows some statistics about the workspace and on your past activities. On the left are few quick shortcuts for creating new requests or sharing the existing ones.

---

<sup>2</sup> [https://en.wikipedia.org/wiki/SQL\\_injection](https://en.wikipedia.org/wiki/SQL_injection)

<sup>3</sup> <https://www.postman.com/>

<sup>4</sup> <https://www.npmjs.com/package/chai>

<sup>5</sup> [https://en.wikipedia.org/wiki/Behavior-driven\\_development](https://en.wikipedia.org/wiki/Behavior-driven_development)

<sup>6</sup> [https://en.wikipedia.org/wiki/Test-driven\\_development](https://en.wikipedia.org/wiki/Test-driven_development)

<sup>7</sup> <https://www.npmjs.com/package/faker>

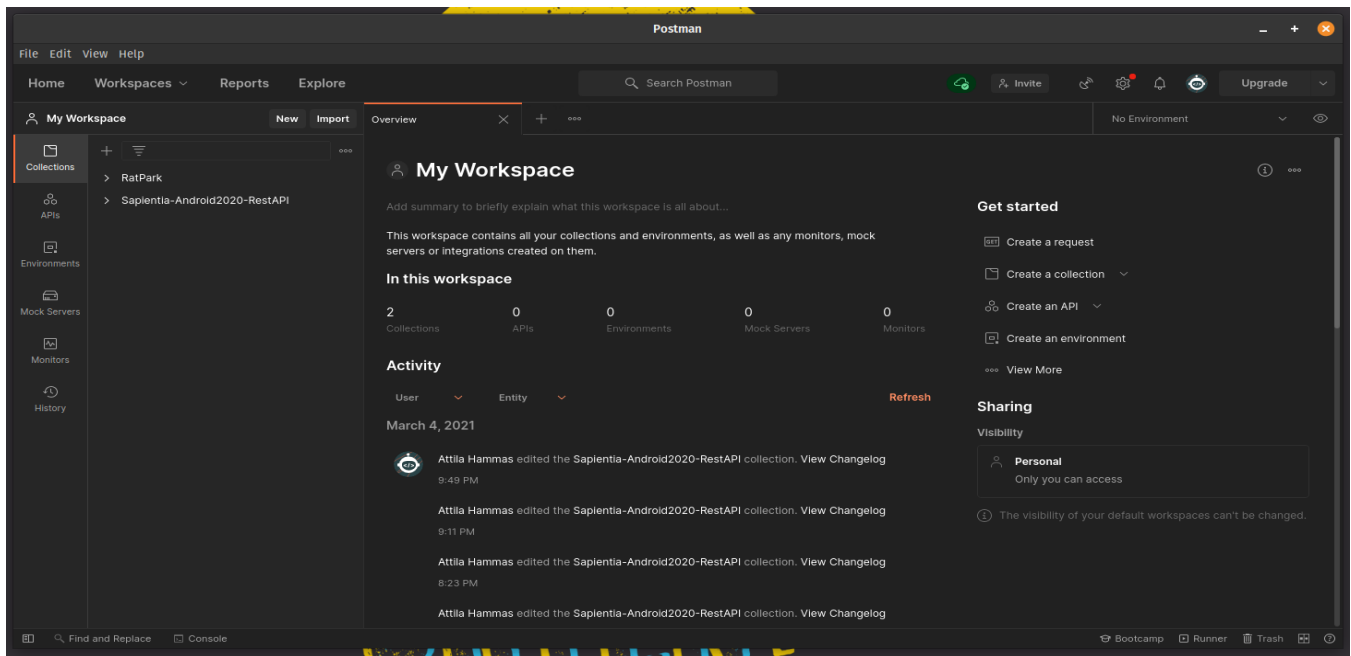


Figure 6.1: Overview screen of Postman.

In postman your request/tests are organized in collections. You can think about it as a folder containing your requests that need to be tested. As seen in the image below (Figure 6.2) there is a “Sapiientia-Android2020-RestAPI” collection containing requests organized in subfolders.

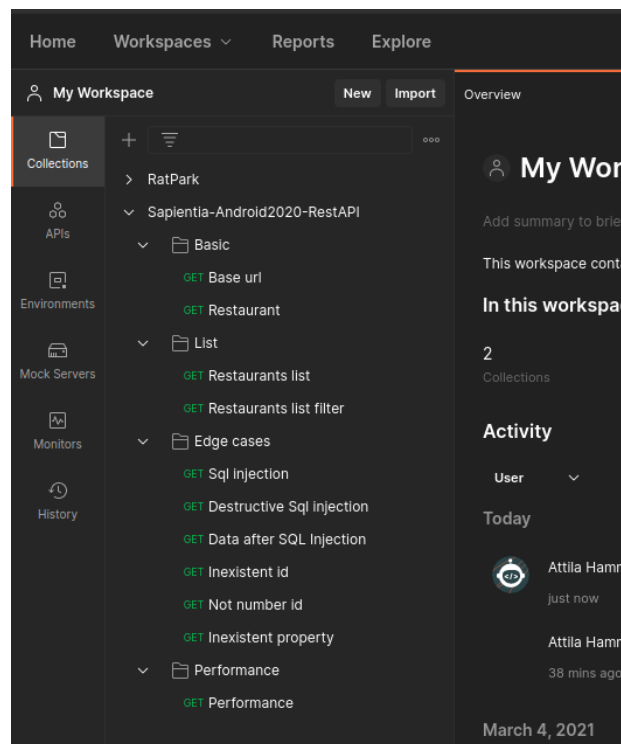


Figure 6.2: Collections and subfolders in Postman.

First thing to do is to create a request by clicking on the “New” button. The request can have a descriptive name like “Base URL” or “Get restaurants”. A request is defined by the HTTP method and the URL. In this example the method is GET and the URL is <https://ratpark-api.imok.space/> as shown below (Figure 6.3).

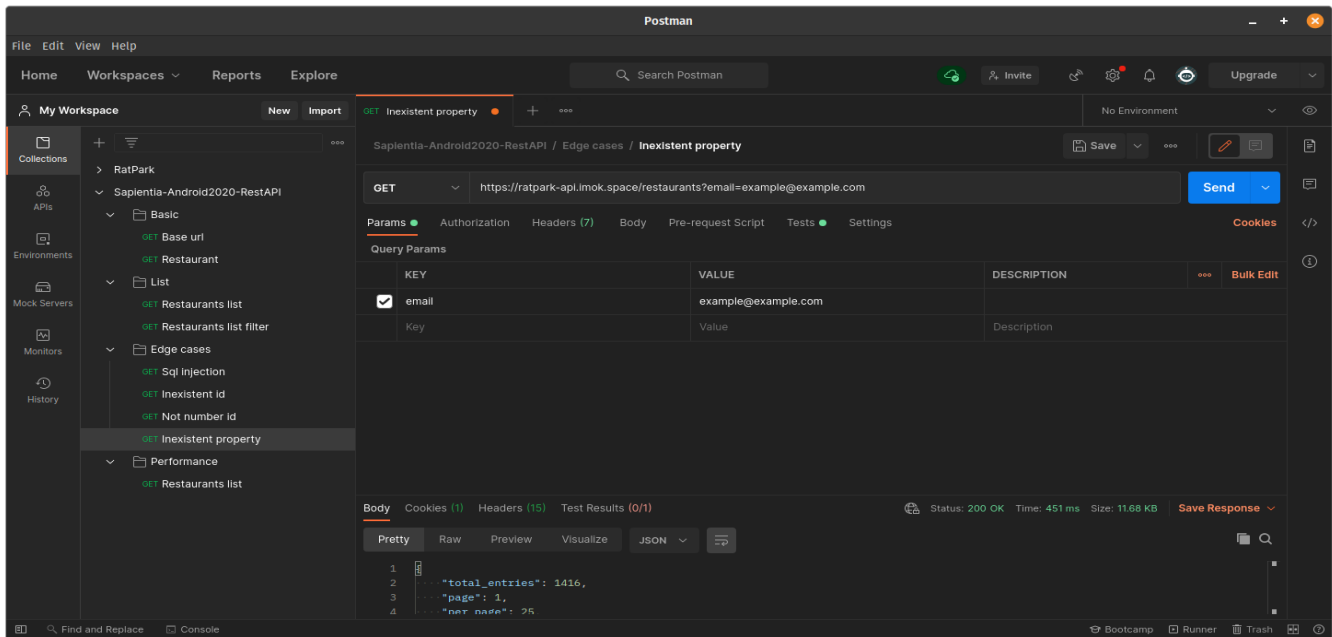


Figure 6.3: Creating a request in Postman.

After sending the request by clicking the “Send” button the response and its main attributes (response status, time, size, body, etc.) are displayed on the bottom of the page (Figure 6.3), here you can check if the web service responded correctly.

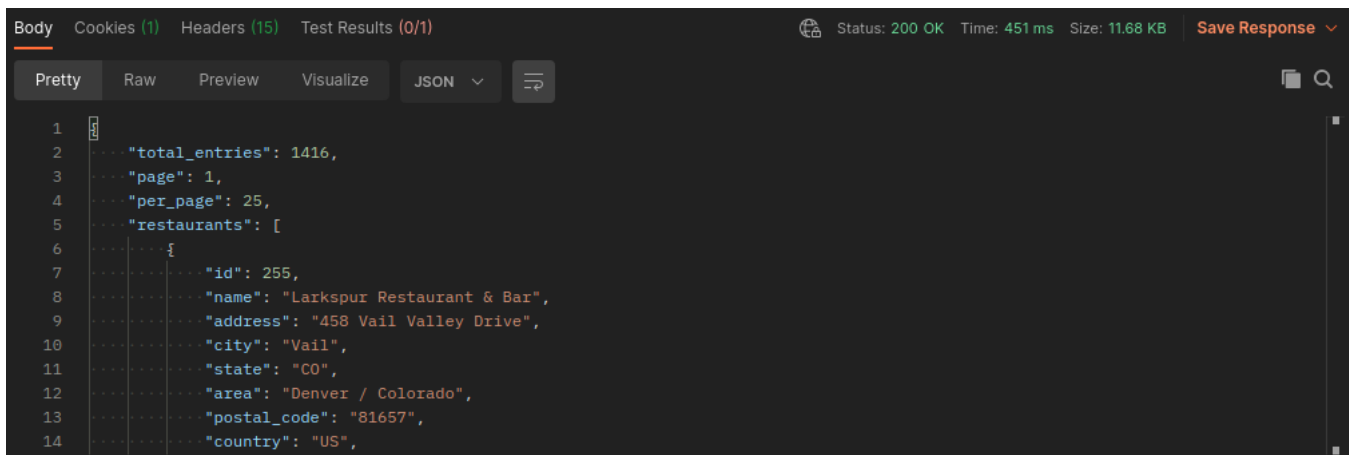


Figure 6.4: Example response displayed in Postman.

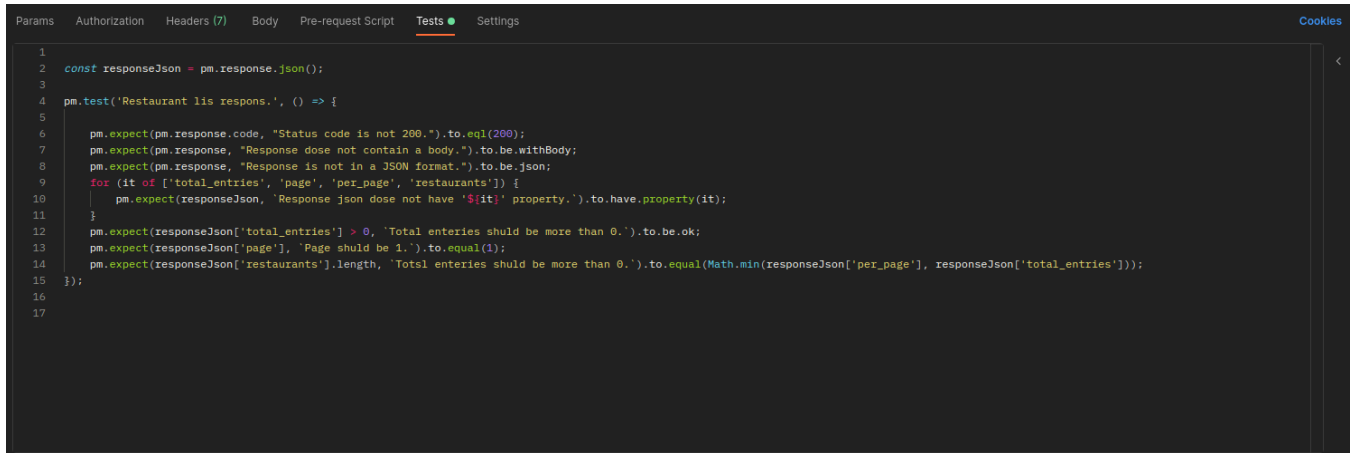
You can automate this checking process by using the “Pre-request script” and “Tests” sections. We can use Chai assertion library’s Behavioral Driven style to define what the response expected to look like. Fortunately, Postman’s editor is based on the [Microsoft’s Monaco Editor](https://microsoft.github.io/monaco-editor/)<sup>8</sup> (which powers Visual Studio Code) and because of that Postman will prompt you with suggestions as you type and you can select one to autocomplete your code. [5]



Figure 6.5: Postman test's execution order.

<sup>8</sup> <https://microsoft.github.io/monaco-editor/>

The execution order is shown above (Figure 6.5). The first is the “Pre-request” script, the dynamic variables are set up here (more on that later), this is followed by the request itself, and in the end the test script is executed. In the image bellow you can see a simple example script (Figure 6.6).



```
1
2 const responseJson = pm.response.json();
3
4 pm.test('Restaurant lis respons.', () => {
5
6     pm.expect(pm.response.code, "Status code is not 200.").toEqual(200);
7     pm.expect(pm.response, "Response dose not contain a body.").toBe.withBody();
8     pm.expect(pm.response, "Response is not in a JSON format.").toBe.json();
9     for (it of ['total_entries', 'page', 'per_page', 'restaurants']) {
10         pm.expect(responseJson, 'Response json dose not have '${it}' property.').toHave.property(it);
11     }
12     pm.expect(responseJson['total_entries'] > 0, 'Total enteries shuld be more than 0.').toBe.ok;
13     pm.expect(responseJson['page'], 'Page shuld be 1.').toEqual(1);
14     pm.expect(responseJson['restaurants'].length, 'Totsl enteries shuld be more than 0.').toEqual(Math.min(responseJson['per_page'], responseJson['total_entries']));
15 });
16
17
```

Figure 6.6: Example test script.

## 6.1. Parameters of the request

The request is defined by a HTTP method and an URL composed from: the base URL, path variables and query parameters. For example in the previous image the request method is GET, the base URL is “<https://ratpark-api.imok.space/>” (the URL of the API), the path variable is “restaurants” (the name of the resource), and the query parameter is “email=example@example.com” (the name of parameter is “email” and its value is “[example@example.com](mailto:example@example.com)”).

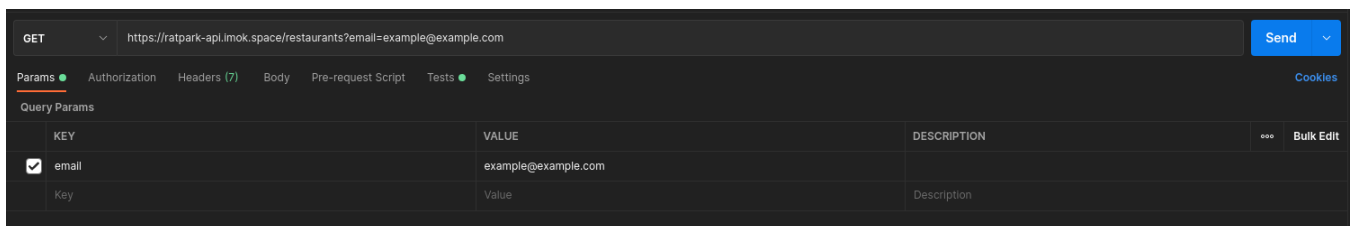


Figure 6.7: Request parameters.

Other HTTP methods, like POST and PUT, can contain a request body. This body is not visible in the URL and in case of secured connection (HTTPS<sup>9</sup>) it is encrypted and not visible by a third parties.

## 6.2. Variables in Postman

A variable is a symbolic representation of data. This can be useful especially if you are using the same values in multiple places or some values are generated by the test script. Variables make your requests more flexible and readable, by abstracting some of the detail away. [5]

<sup>9</sup> <https://en.wikipedia.org/wiki/HTTPS>



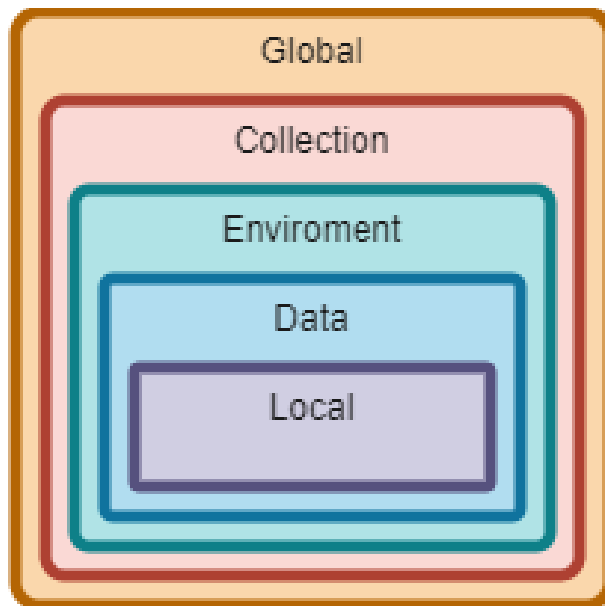


Figure 6.8: Variable scopes.

There are multiple scopes as shown in the image above (Figure 6.8). When multiple variables are defined with the same name, the value stored in the variable with narrowest scope will be used.

```
GET https://ratpark-api.lmok.space/restaurants?country={{country}}&city={{city}}&price={{price}}&page={{page}} Send

Params Authorization Headers (7) Body Pre-request Script Tests Settings Cookies

8
9 var country = pm.environment.get('country');
10 var city = pm.environment.get('city');
11 var price = pm.environment.get('price');
12 var page = pm.environment.get('page');
13
14 pm.test('Simple restaurant list url works.', () => {
15   pm.expect(pm.response.code, 'Status code is not 200.').to.eql(200);
16 });
17
18 pm.test('Simple restaurant list url returns json.', () => {
19   pm.expect(pm.response).to.be.ok;
20   pm.expect(pm.response, 'Response dose not contain a body.').to.be.withBody;
21   pm.expect(pm.response, 'Response is not in a JSON format.').to.be.json;
22 });
23
24 pm.test('Simple restaurant list url returns correct json schema.', () => {
25   const responseJson = pm.response.json();
26   for (it of ['total_entries', 'page', 'per_page', 'restaurants']) {
27     pm.expect(responseJson, 'Response json dose not have '${it}' property.').to.have.property(it);
28   }
29 });
```

Figure 6.9: Example usage of variables.

In the Figure 6.9 the Pre-request script is used to set up a list of parameters. The test script is going to use this list for multiple requests using another feature called request flows (more on that later).

```
GET https://ratpark-api.lmok.space/restaurants/{{${randomUUID}}} Send

Params Authorization Headers (7) Body Pre-request Script Tests Settings Cookies

1 const responseJson = pm.response.json();
2
3 pm.test('Inexistent id.', () => {
4   pm.expect(responseJson, 'The json object should be empty.').to.deep.equal({});
5   pm.expect(pm.response.code, 'Response should be not found (404)').to.eql(404);
6 });
```

Figure 6.10: Mock data using Faker library.

Thanks to Faker library you can generate random names, addresses, email addresses, and much more. You can use these pre-defined variables multiple times to return different values per request. More information and a list of supported values is available in Postman's documentation<sup>10</sup>. In the example shown in Figure 6.10 Each time when a request is sent the “{{\$randomUUID}}” variable is replaced with a valid random UUID.

### 6.3. Flows (request chains)

The postman object provides the “setNextRequest” method for building request workflows when you use the collection runner or Newman. When you run a collection, Postman will run your requests in a default order or an order you specify when you set up the run. However, you can override this execution order using “postman.setNextRequest” to specify which request should run next. [5]

This functionality enables the creation of “flows”. For example, if a web service allows for managing of the resources (for example users) then you can create a flow of actions like shown in



Figure 6.11: An example flows.

The proposed API for testing dose not allows the user to manage the resources, but this does not mean that is not worth trying. The resource in this API is an SQLite<sup>11</sup> Database and because of that may be vulnerable to **SQL Injection**<sup>12</sup>. The request bellow should not succeed.

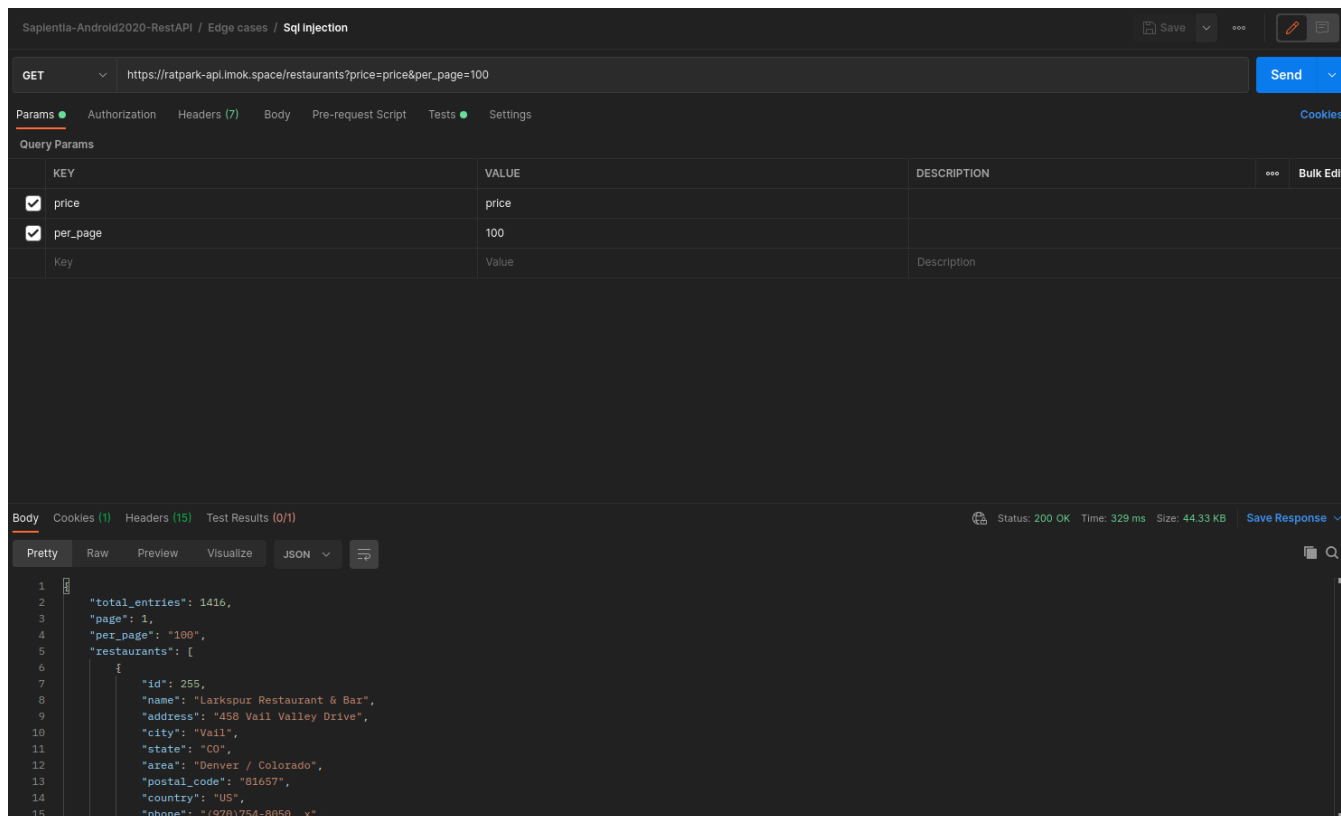


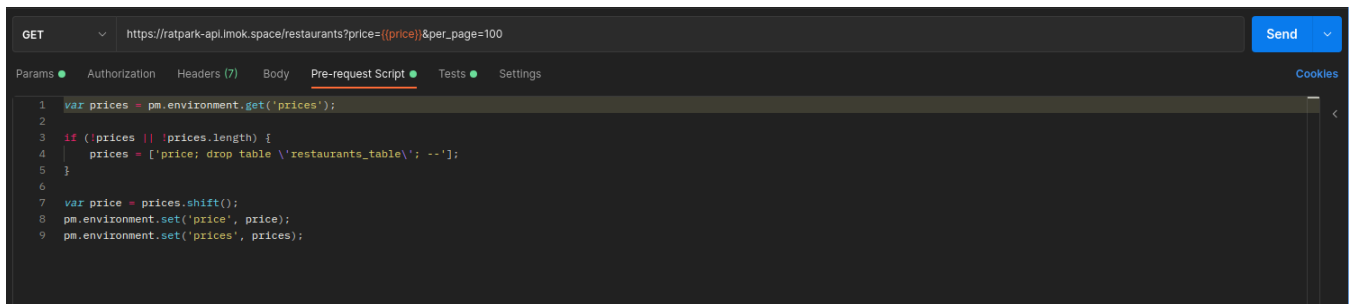
Figure 6.12: Example SQL Injection.

<sup>10</sup> <https://learning.postman.com/docs/writing-scripts/script-references/variables-list/>

<sup>11</sup> <https://en.wikipedia.org/wiki/SQLite>

<sup>12</sup> [https://en.wikipedia.org/wiki/SQL\\_injection](https://en.wikipedia.org/wiki/SQL_injection)

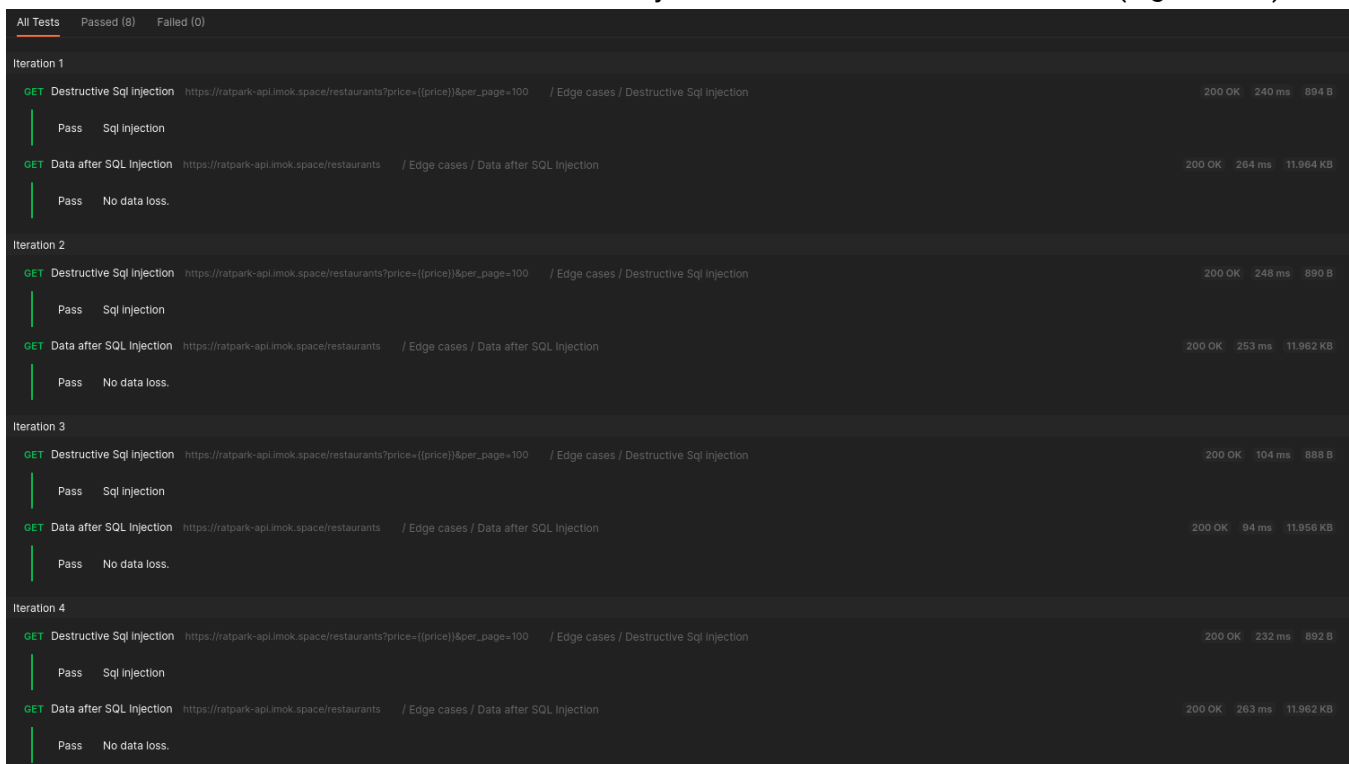
As the Figure 6.12 shows the injection is successful. There is no hidden confidential information in the database, so this is not as big of a deal. But what if we try to execute a more destructive SQL Injection, which may result in data loss.



```
1 var prices = pm.environment.get('prices');
2
3 if (!prices || !prices.length) {
4   prices = ['price; drop table `restaurants_table`; --'];
5 }
6
7 var price = prices.shift();
8 pm.environment.set('price', price);
9 pm.environment.set('prices', prices);
```

Figure 6.13: Example for destructive SQL Injection.

As the result shows the request was executed successfully. But if we check the resource by getting the list of “restaurants” than the data is there. This means the injection was not successful after all (Figure 6.14).



All Tests		Passed (8)	Failed (0)
Iteration 1			
GET	Destructive Sql Injection	https://ratpark-api.imok.space/restaurants?price={{price}}&per_page=100 / Edge cases / Destructive Sql Injection	200 OK 240 ms 894 B
Pass	Sql Injection		
GET	Data after SQL Injection	https://ratpark-api.imok.space/restaurants / Edge cases / Data after SQL Injection	200 OK 264 ms 11.964 KB
Pass	No data loss.		
Iteration 2			
GET	Destructive Sql Injection	https://ratpark-api.imok.space/restaurants?price={{price}}&per_page=100 / Edge cases / Destructive Sql Injection	200 OK 248 ms 890 B
Pass	Sql Injection		
GET	Data after SQL Injection	https://ratpark-api.imok.space/restaurants / Edge cases / Data after SQL Injection	200 OK 253 ms 11.962 KB
Pass	No data loss.		
Iteration 3			
GET	Destructive Sql Injection	https://ratpark-api.imok.space/restaurants?price={{price}}&per_page=100 / Edge cases / Destructive Sql Injection	200 OK 104 ms 888 B
Pass	Sql Injection		
GET	Data after SQL Injection	https://ratpark-api.imok.space/restaurants / Edge cases / Data after SQL Injection	200 OK 94 ms 11.956 KB
Pass	No data loss.		
Iteration 4			
GET	Destructive Sql Injection	https://ratpark-api.imok.space/restaurants?price={{price}}&per_page=100 / Edge cases / Destructive Sql Injection	200 OK 232 ms 892 B
Pass	Sql Injection		
GET	Data after SQL Injection	https://ratpark-api.imok.space/restaurants / Edge cases / Data after SQL Injection	200 OK 263 ms 11.962 KB
Pass	No data loss.		

Figure 6.14: Check if data base is empty after SQL Injection.

## 6.4. Results

The simplest way to check the written tests results is by opening a post man runner and giving it parameters like iteration time and delay between iteration and pressing “Run” button. This will begin to send the request one by one, will run the test cases afterwards and will display the result in a detailed way or in an overview page.

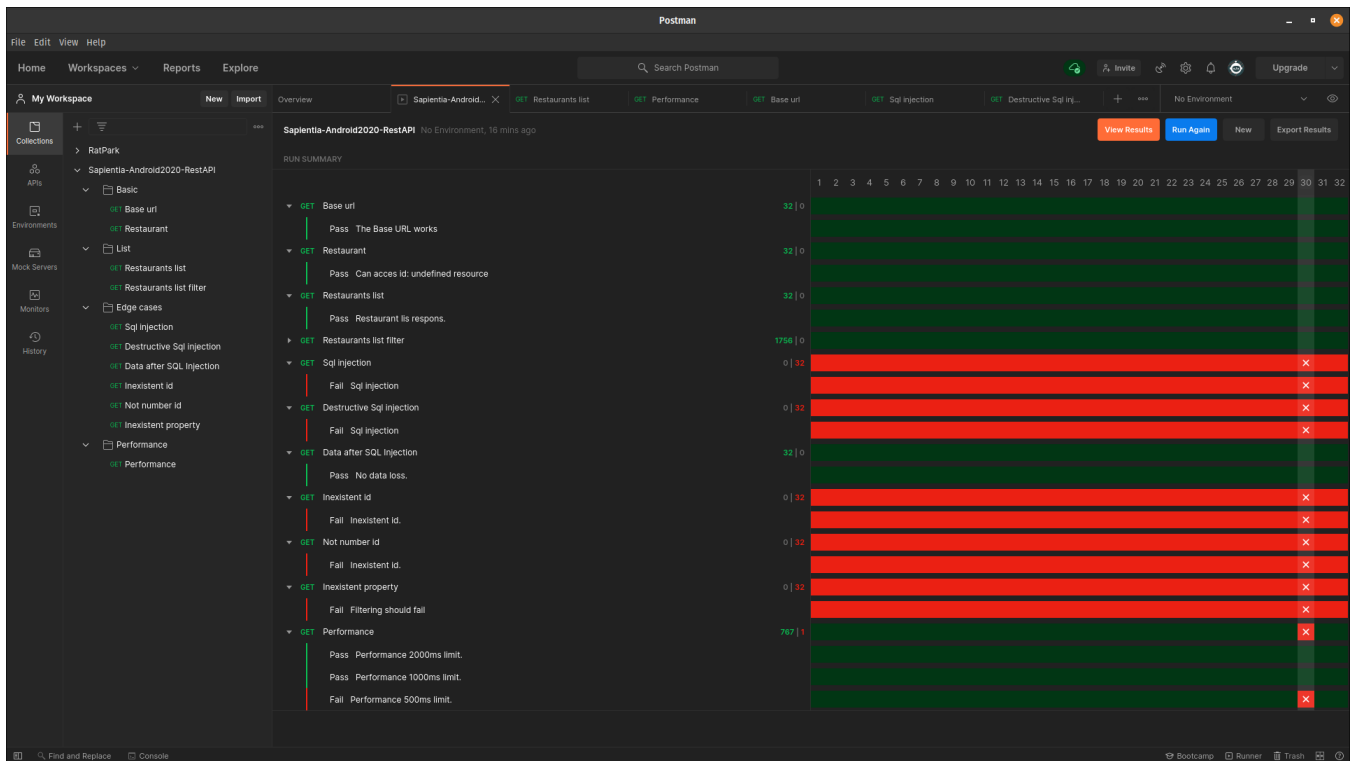


Figure 6.15: Test results overview.

The postman team created a tool called [Newman<sup>13</sup>](https://www.npmjs.com/package/newman). this is a CLI tool which can run the test cases defined in Postman, this way we do not need to use web app to run our tests. Newman can be installed as an NPM package, this way we can use it in the terminal (Figure 6.16) and in JavaScript scripts as well.

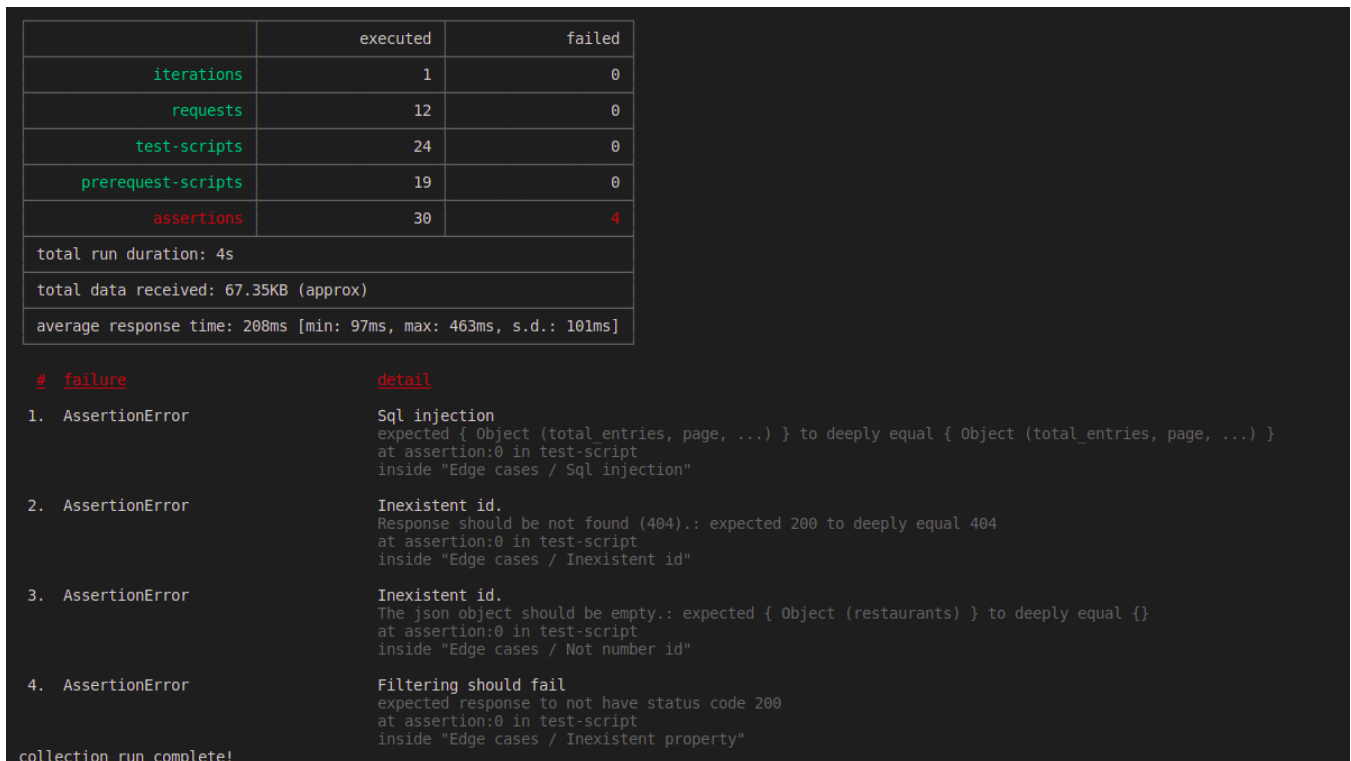


Figure 6.16: Newman results in the terminal.

<sup>13</sup> <https://www.npmjs.com/package/newman>

By using Newman, we can integrate the test collection into staging environments, for example if we use GitHub workflows, we can define it as shown below (Figure 6.17).

```
name: Newman Run

on:
  pull_request:
    branches:
      - master

jobs:
  newman:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@master
      - uses: matt-ball/newman-action@master
        with:
          apiKey: ${ secrets.postmanApiKey }
          collection: postman_collection.json
          environment: 5922408-228c7edd-fc15-4f68-9665-a35d7df6945b
```

Figure 6.17: GitHub workflow Yamle file. [6]

## 7. Some insight

As some of the test result show the API's performance and security could be better. However, this Web API was created with educational purposes, so performance is not as a big deal. At the time of finding out that the API is vulnerable to nondestructive SQL Injections, I thought that "this ain't a bug it's a feature", the user can access the data as he or she pleases (can filter the dataset in a way that is not implemented as a feature).

Furthermore, because this API is not needed anymore, it is not maintained, and these issues shown by the tests are not going to be fixed.

## 8. Conclusions

In conclusion, Postman Team created a powerful set of tools that can help developers to create testing pipelines for their Web API projects. It is user friendly by using ChaiJs, versatile because of FakerJs, helpful due to the use of Microsoft's Monaco editor, and can be used for testing, monitoring and in CI/CD pipelines as well.

## 9. References

1. [https://en.wikipedia.org/wiki/Web\\_service](https://en.wikipedia.org/wiki/Web_service)
2. <https://www.guru99.com/restful-web-services.html>
3. [https://en.wikipedia.org/wiki/API\\_testing](https://en.wikipedia.org/wiki/API_testing)
4. <https://postman.com/>
5. <https://learning.postman.com/docs>
6. <https://github.com/marketplace/actions/newman-action>