POSTMAN

PILLS

23 DE ENERO DE 2020

ASSEMBLER
Passeig de Joan de Borbó, 101

POSTMAN

Postman is a tool that is used, above all, for API REST testing, although it also supports other features that come out of what is included in the testing of this type of systems.

Thanks to this tool, in addition to testing, consuming and debugging REST APIs, we can monitor them, write automated tests for them, document them, mock them, simulate them, etc.

DIEGO ARTURO SILVA ROJAS

BARCELONA, SPAIN
17 JANUARY 2020

POSTMAN

CONTENT TABLE

POSTMAN

1. General Analysis.

1.1 Install Postman

The first step is to install Postman to start using it. To do this, follow the steps indicated in the official guide:

https://www.getpostman.com/downloads/

1.2 Analyze the interface

Next, review the menus provided by the interface and add in the documentation a section explaining what it is for and how each of the following elements is used:

- *Request.*
- *Collection.*
- *Environment.*
- *API Documentation.*
- *Mock server.*
- *Monitor.*
- *API.*

It is possible that to have access to any of the above elements you must create a Postman account

1.3 Put it into practice

Finally, you must implement the following Postman functionalities using an example online API to make the requests:

- *Create a new POSTMAN Collection (research previously about it).*
- *Create a request for each of the following methods as shown in the image above:*
  - *GET*
  - *POST*
  - *PUT*
  - *DELETE*

The main idea is that you know how to use each type of request and thus be able to test your own or third-party applications. For example, for the request that uses the GET method, you would call "/ employee" for the POST "/ create",…

- *Create a test for each previous request where you will verify that the response meets the following conditions:*
  - *The state code is as expected*
  - *The name of the state code is the corresponding one (OK, Created, ...)*
  - *The answer has body*
  - *The answer is in JSON*
  - *The header has "Content-Type"*
  - *The response time is less than 200ms*
- *Create a monitor that runs the previously created collection every 5 min*
- *Create an API document from the previous collection with the Postman application*

- *Export the created collection and import it back into Postman (you must incorporate the generated file in the project repository.*

You must document all the actions performed previously and take screenshots of all the processes.

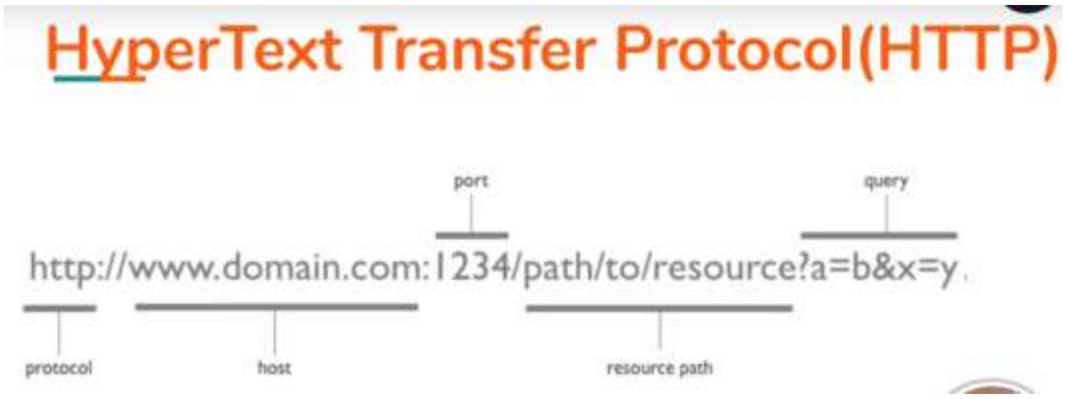
1.4 Search Information.

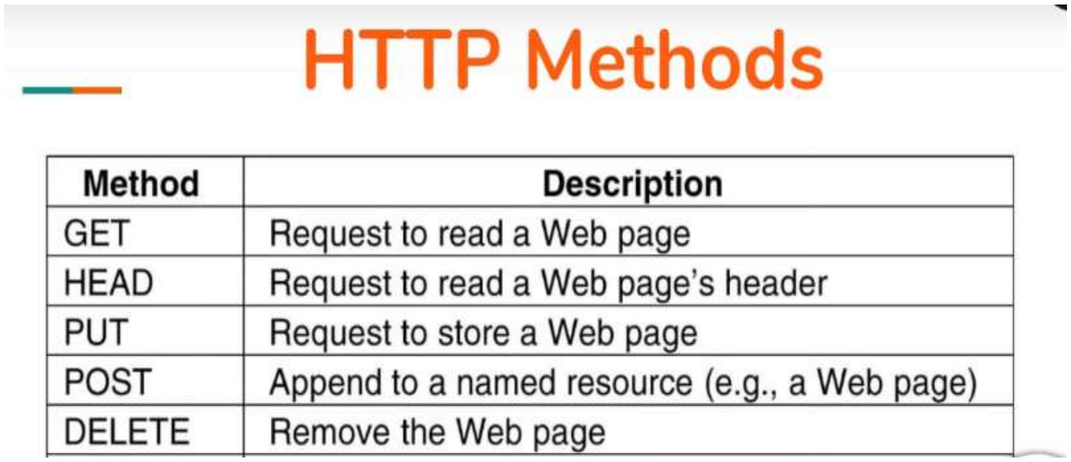
1.4.1 Requests.

You can send requests in Postman to connect to APIs you are working with. Your requests can retrieve, add, delete, and update data. Whether you are building or testing your own API, or integrating with a third-party API, you can try out your requests in Postman. Your requests can send parameters, authorization details, and anybody data you require.

For example, if you're building a client application (e.g. a mobile or web app) for a store, you might send one request to retrieve the list of available products, another request to create a new order (including the selected product details), and a different request to log a customer in to their account.

When you send a request, Postman will display the response received from the API server in a way that lets you examine, visualize, and if necessary troubleshoot it.


HyperText Transfer Protocol(HTTP)
port
query
http://www.domain.com:1234/path/to/resource?a=b&x=y
protocol
host
resource path



HTTP Methods

Method	Description
GET	Request to read a Web page
HEAD	Request to read a Web page's header
PUT	Request to store a Web page
POST	Append to a named resource (e.g., a Web page)
DELETE	Remove the Web page

1.4.2 Collections.

Postman Collections are a group of saved requests you can organize into folders. Every request you send in Postman appears under the History tab of the sidebar.

On a small scale, reusing requests through the history section is convenient. However, as your Postman usage scales, it can be time consuming to find a particular request in your history. Instead of combing through your history section, you can save all your requests as a group for easier access.

1.4.3 Environments.

Environments enables you to create robust requests that you can reuse. You also can use environments in the Collection Runner.

1.4.4 API Documentation.

You can automatically generate documentation for your Postman APIs. You can share your documentation privately or publish it on the web. Postman generates and hosts documentation based on collections, synced in real time and accessible via the browser. You can use documentation to collaborate with team members and partners, or to support developer adoption for your public APIs.

1.4.5 Mock Server.

Mock Servers in Postman let you simulate APIs. You can create mock servers from the Postman app, from the web dashboard, and using the Postman API. You will need a Postman account to set up a mock server.
Mocks in Postman are tied to a collection. Postman matches requests and generates responses for mocks from the Examples in the requests of a collection. You can create a mock server even if you don't have an existing collection.

1.4.6 Monitors.

Postman Monitoring helps you to stay up to date on the health and performance of your APIs. Within a matter of seconds, you can set up Postman's monitoring service and integrate it into your API development pipeline.

1.4.7 API.

The Postman API allows you to programmatically access data stored in Postman account with ease.

The easiest way to get started with the API is to click the Run in Postman button present at the top of the documentation page and use the Postman App to send requests.

2. Pill requirements.

- *“Create a clear and orderly directory structure.*
- *Both the code and the comments must be written in English.*
- *Use the camelCase code style to define variables and functions.*
- *In the case of using HTML, never use online styles.*
- *In the case of using different programming languages always define the implementation in separate terms.*
- *Remember that it is important to divide the tasks into several sub-tasks so that in this way you can associate each particular step of the construction with a specific commit.*

- *You should try as much as possible that the commits and the planned tasks are the same.*
- *Delete files that are not used or are not necessary to evaluate the project."*

3. Pill Planning.

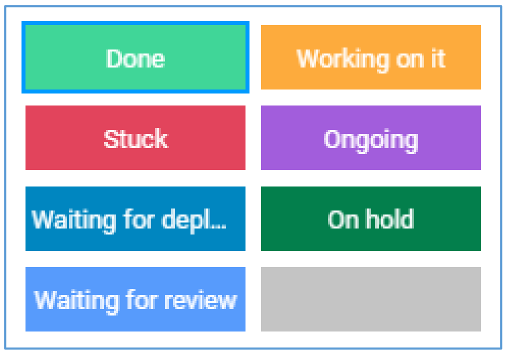
3.1 Pill organization.

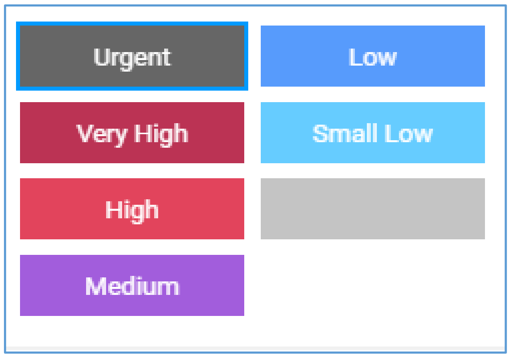
"Next you will have to create a document where you can explain in detail how the current pill is organized. It is important that it be updated throughout the life of the pill. The document must include at least:

- *Requirements documentation.*
- *List of tasks to be performed.*
  - *Priority of each task*
  - *Title and description of each of them*
  - *Difficulty level*
  - *Estimated time for each task.*
- *Record of incidents that were detected during the execution of the pill.*
- *Documentation about the GIT WORKFLOW you are going to use*
- *Documentation about the tools used in the project*
- *Record of lessons learned."*

3.2 Priority of each task.

A total of 5 SPRINTS of the project were defined with each of its tasks, assigning the priority of execution and the state of the task.





8 scenarios with their respective colors were proposed for the status of each task:

- *Done* (Light Green), for task have been finished
- *Working on it* (Orange), tasks are underway and almost finished.
- *Stuck* (Red), tasks are stagnant for some reason and could not be finished.
- *Waiting for review* (Blue), tasks are waiting for a review and approval.
- *Waiting for deployment* (Dark Blue), tasks are pending for status update and repository update.
- *Ongoing* (Purple), tasks are on development.
- *On hold* (Dark Green), tasks are waiting for something resource or search information or depend on another tasks to start.
- (Gray), tasks without assign.

6 scenarios with their respective colors were proposed for assign the tasks priority. The priority upper level is *High* (Red) and *Small Low* (Light Blue) is the priority lower level. For tasks with null scenario (it doesn't have priority) it will be assigned with

Nothing (Gris). The *Urgent (Dark Gray)* scenario was established for tasks with extreme and excessive importance.

3.3 List Task.

POSTMAN				
Name	**Status**	**Priority**	**Date**	**Time**
Discovery				**9,1**
Sprint backlog		Urgent	2020-01-23	0,5
General Analysis		Very High	2020-01-23	2
Planing		Very High	2020-01-23	3,2
Daily Scrum		Medium	2020-01-23	0,5
Project requirements		High	2020-01-23	0,5
Define quality control managment		Medium	2020-01-23	0,2
Define Risk and Incidents managme		Medium	2020-01-23	0,2
Sprint review meeting		Low	2020-01-23	1
Product Backlog		High	2020-01-23	1
General Analysis				**2**
Install Postman		Low	2020-01-23	0,2
Analyze the interface		Low	2020-01-23	0,3
Put it into practice		Low	2020-01-23	0,5
Search Information		Low	2020-01-23	1
Planning				**3,2**
Pill organization		Urgent	2020-01-23	0,7
List Task		Urgent	2020-01-23	1
Estimate time		Medium	2020-01-23	0,5
Workflow Git	Done	Urgent	2020-01-23	0,1
Structure Project Folder	Done	Very High	2020-01-23	0,2
Software requirements	Ongoing	High	2020-01-23	0,4
Tools used.	Ongoing	High	2020-01-23	0,3
Development				**7**
New collection		Very High	2020-01-24	1
Testings		Very High	2020-01-24	3
Monitor		Very High	2020-01-24	2
Document API		Medium	2020-01-24	1
Export Collection		Very High	2020-01-24	1.5
Monitoring Control				**5**
Check requirements acomplished		Medium	2020-01-24	1
Redact document		High	2020-01-24	3
Check deliverables		Medium	2020-01-24	1
Total				**21,1**

3.4 Total approximate project estimate.

Using a tool for project management, called monday.com. Tasks are reflected within sprints or stages. Adding the hours each task for each sprint we have that the project will be executed in 21.1 hours in 4 days, it will be 5.5 hours average per day for the project deadline.

POSTMAN				
Name	Status	Priority	Date	Time
Discovery				9,1 h
	General Analysis			2,0 h
	Planning			3,2 h
Development				7,0 h
Monitoring Control				5,0 h
Total				**21,1 h**

The first sprint called Discovery were divided in two main sub-sprints: General analysis and Project Planning.

3.5 GIT Workflow documentation.

The GIT repository is located at: https://github.com/diegosilva91/Postman and was cloned into the computer to create the corresponding static portal files. The master BRANCH was used to carry out all the loads with their respective commit.

3.6 Software requirements.

It must be installed for develop the pill:

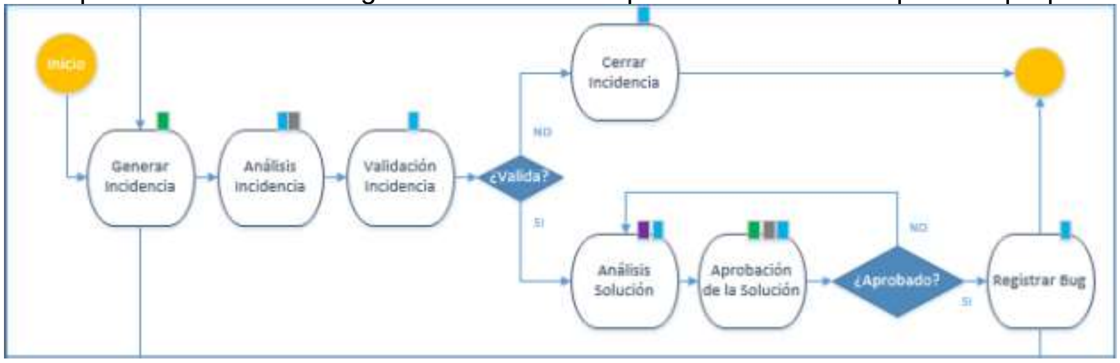
- Postman.

3.7 Tools used in the project.

- Project management was carried out with Monday.com, to plan tasks, sprints, calendar and schedule.
- Visual Studio Code.

4. Incident management.

The procedure to manage the incidents presented in the pill is proposed.



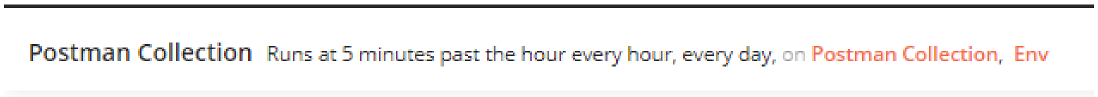
The incidents will be recorded in the updates of each task of the support tool for project management used in its development. This section will record the incident detected, if validated, the solution found for this solution and if it was correctly solved. The incidents will be reviewed again in the partial and final meetings of each sprint, to review the pending incidents and generate actions so that they are not recurring.

4.1 Incidents solved.

- Obtain one record by method GET.
- Status code response was no authorized in PUT and GET methods.

4.2 Incidents no-solved.

- The test check if the "Response time is less than 200ms" failed with GET Request by id.
- It was impossible set up successfully the monitor time execution test every 5 minutes. After set up it, it was not executed at that time.



Postman Collection Runs at 5 minutes past the hour every hour, every day, on Postman Collection, Env

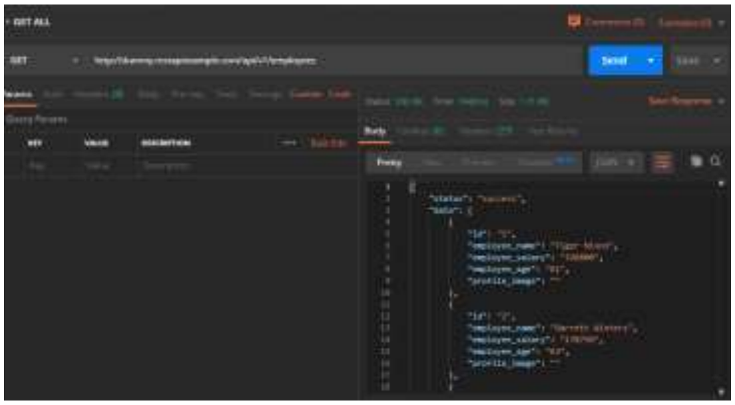
5. Development.

Before the development a little estimate algorithm was made with task and subtasks.

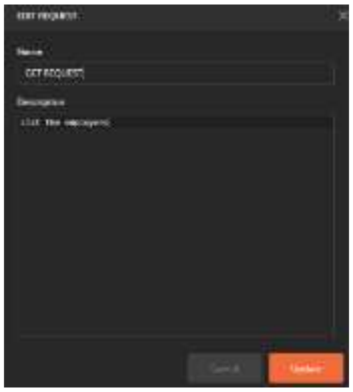
	TASKS	SUBTASKS 1	SUBTASKS 2	SUBTASKS 3
TASK 1	New collection	Name		
		methods	GET	Headers, key
		Network Chrome	POST	Body, keys
		Select an authorization	JSON	
		Enter a pre-request script to execute before the collection runs.		
TASK 2	Testings	Test result	Code scritps	state code
				code is (OK, Created, ...)
				The answer has body
				The answer is in JSON
				The header has "Content-Type"
				The response time is less than 200ms
TASK 3	Monitor	Monitor	Set up	
TASK 4	Document API			
	Export Collection			

5.1 Methods Http.

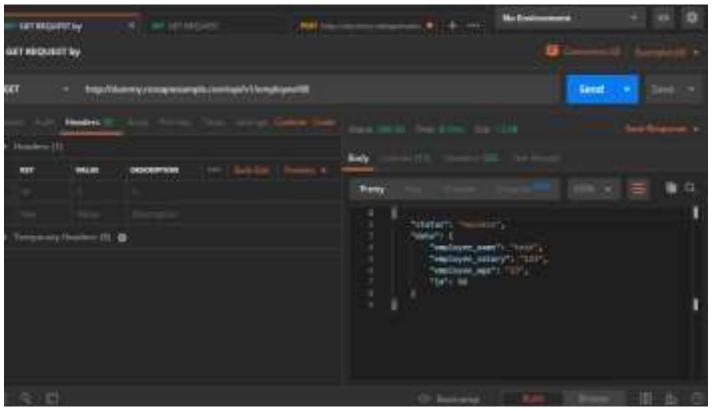
- GET REQUEST.
  List ALL employees from the API.



Saving the request:

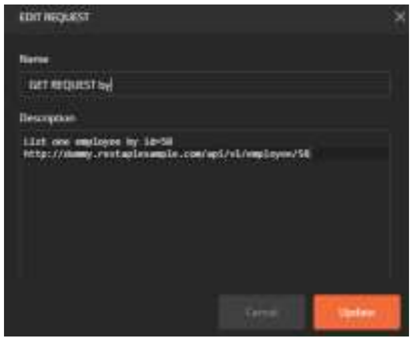


- GET REQUEST BY
  List the employee by id.



```
"status": "success",
"data": {
    "employee_name": "test",
    "employee_salary": "123",
    "employee_age": "23",
    "id": 58
}
```

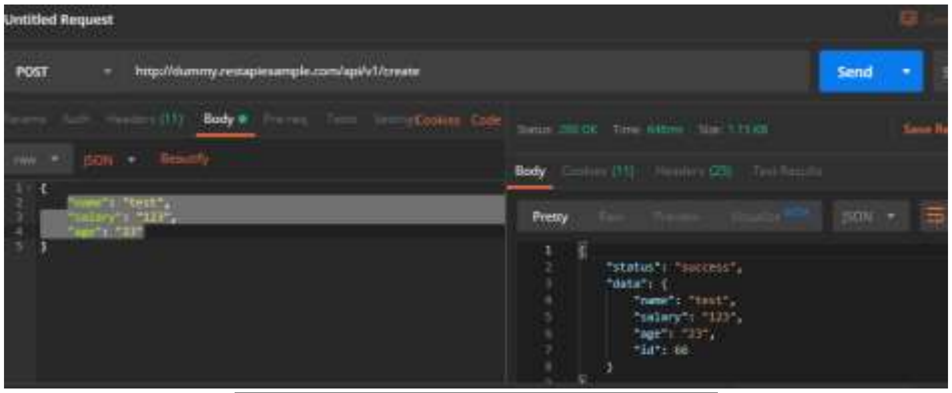
Saving the request



GET REQUEST by

```
List one employee by id=58
http://dummy.restapiexample.com/api/v1/employee/58
```

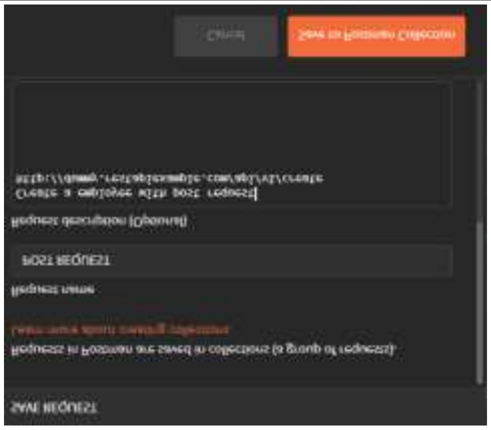
- POST REQUEST.
  Create an employee with a POST method request.


Untitled Request

POST http://dummy.restapiexample.com/api/v1/create

```
{
    "name": "test",
    "salary": "123",
    "age": "23"
}
```

```
{
    "status": "success",
    "data": {
        "name": "test",
        "salary": "123",
        "age": "23",
        "id": 66
    }
}
```


SAVE REQUEST

Requests in Postman are saved in collections (a group of requests).

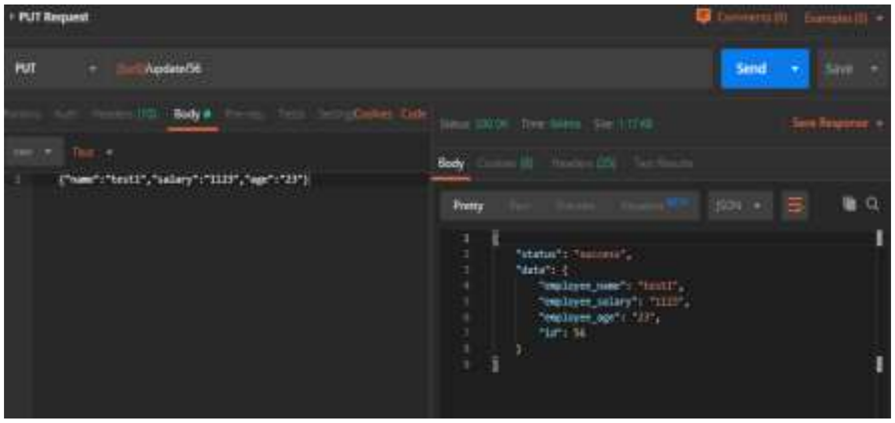
Learn more about creating collections

Request name: POST REQUEST

Request description (Optional):

```
Create a employee with post request
http://dummy.restapiexample.com/api/v1/create
```

- PUT REQUEST

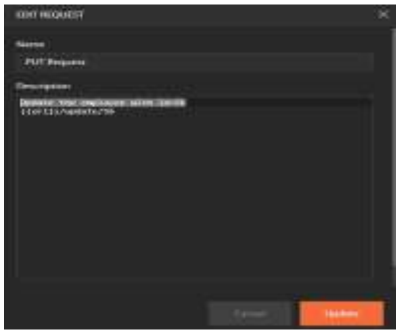

PUT Request

PUT {{url}}/update/56

```
{"name":"test1","salary":"1123","age":"23"}
```

```
{
    "status": "success",
    "data": {
        "employee_name": "test1",
        "employee_salary": "1123",
        "employee_age": "23",
        "id": 56
    }
}
```

Update the employee with id=56

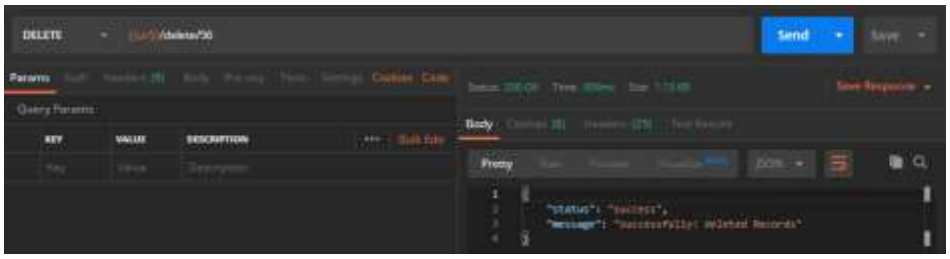

EDIT REQUEST

Name: PUT Request

Description:

```
Update the employee with id=56
{{url}}/update/56
```

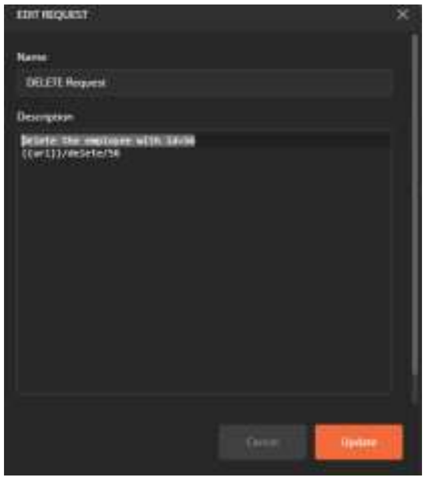
- PUT REQUEST



```
DELETE  {{url}}/delete/56
```

```
{
    "status": "success",
    "message": "successfully! deleted Records"
}
```

Delete the employee with id=56



EDIT REQUEST

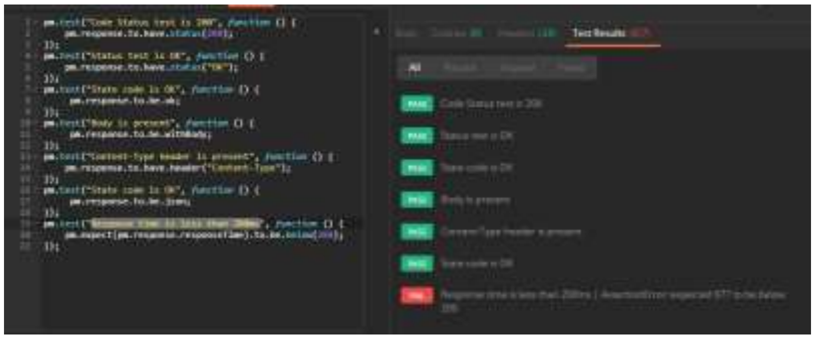
Name: DELETE Request

Description:

```
Delete the employee with id=56
{{url}}/delete/56
```

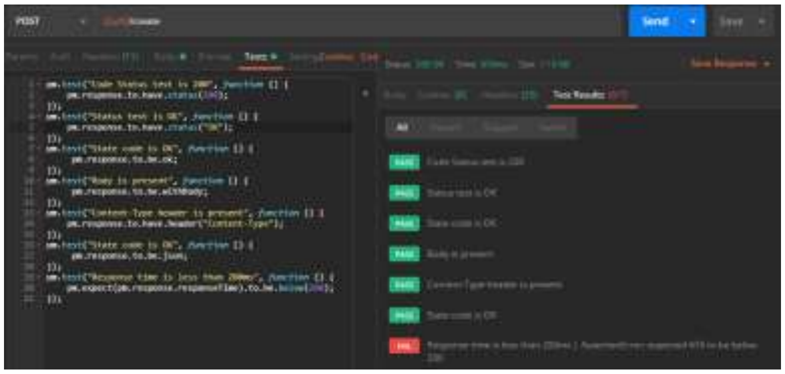
5.2 Test for requests.

For each request the follow tests has been checked:



```
pm.test("Code Status test is 200", function () {
    pm.response.to.have.status(200);
});
pm.test("Status test is OK", function () {
    pm.response.to.have.status("OK");
});
pm.test("State code is OK", function () {
    pm.response.to.be.ok;
});
pm.test("Body is present", function () {
    pm.response.to.be.withBody;
});
pm.test("Content-Type header is present", function () {
    pm.response.to.have.header("Content-Type");
});
pm.test("State code is OK", function () {
    pm.response.to.be.json;
});
pm.test("Response time is less than 200ms", function () {
    pm.expect(pm.response.responseTime).to.be.below(200);
});
```

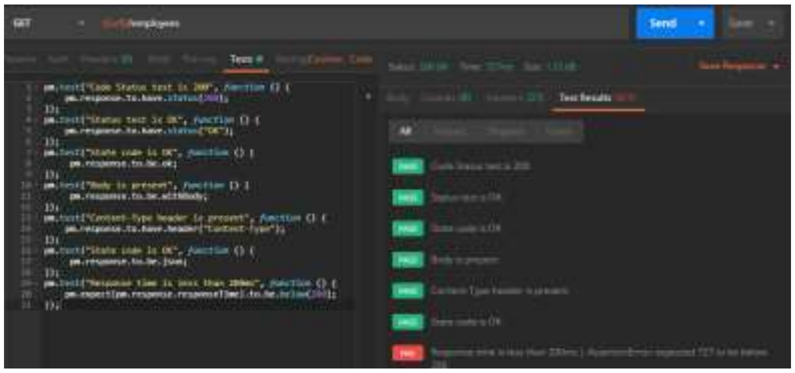
```
PASS  Code Status test is 200
PASS  Status test is OK
PASS  State code is OK
PASS  Body is present
PASS  Content-Type header is present
PASS  State code is OK
FAIL  Response time is less than 200ms | AssertionError: expected 577 to be below
      200
```



```
POST  {{url}}/create
```

```
pm.test("Code Status test is 200", function () {
    pm.response.to.have.status(200);
});
pm.test("Status test is OK", function () {
    pm.response.to.have.status("OK");
});
pm.test("State code is OK", function () {
    pm.response.to.be.ok;
});
pm.test("Body is present", function () {
    pm.response.to.be.withBody;
});
pm.test("Content-Type header is present", function () {
    pm.response.to.have.header("Content-Type");
});
pm.test("State code is OK", function () {
    pm.response.to.be.json;
});
pm.test("Response time is less than 200ms", function () {
    pm.expect(pm.response.responseTime).to.be.below(200);
});
```

```
PASS  Code Status test is 200
PASS  Status test is OK
PASS  State code is OK
PASS  Body is present
PASS  Content-Type header is present
PASS  State code is OK
FAIL  Response time is less than 200ms | AssertionError: expected 619 to be below
      200
```

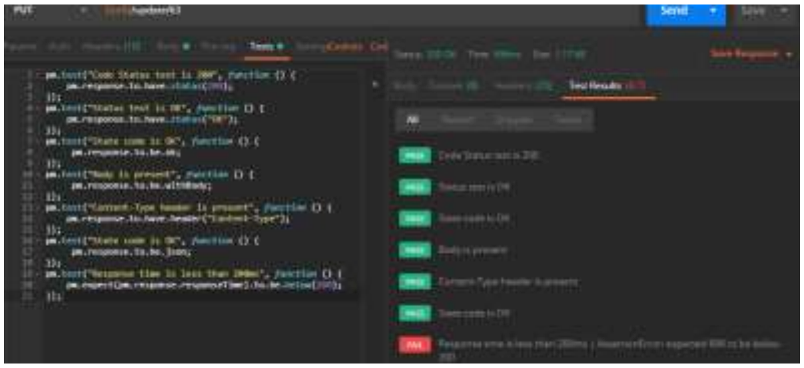


GET [illegible]/employees

```
pm.test("Code Status text is 200", function () {
    pm.response.to.have.status(200);
});
pm.test("Status text is OK", function () {
    pm.response.to.have.status("OK");
});
pm.test("State code is OK", function () {
    pm.response.to.be.ok;
});
pm.test("Body is present", function () {
    pm.response.to.be.withBody;
});
pm.test("Content-Type header is present", function () {
    pm.response.to.have.header("Content-Type");
});
pm.test("State code is OK", function () {
    pm.response.to.be.json;
});
pm.test("Response time is less than 200ms", function () {
    pm.expect(pm.response.responseTime).to.be.below(200);
});
```

Test Results

PASS Code Status text is 200
PASS Status text is OK
PASS State code is OK
PASS Body is present
PASS Content-Type header is present
PASS State code is OK
FAIL Response time is less than 200ms | AssertionError: expected 727 to be below 200

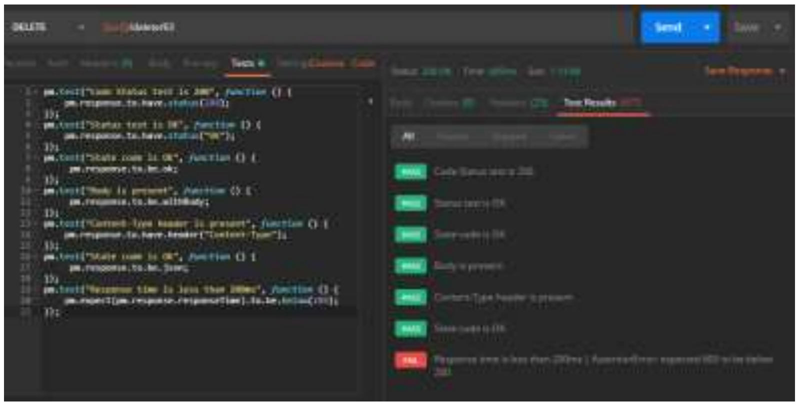


PUT [illegible]/update/63

```
pm.test("Code Status text is 200", function () {
    pm.response.to.have.status(200);
});
pm.test("Status text is OK", function () {
    pm.response.to.have.status("OK");
});
pm.test("State code is OK", function () {
    pm.response.to.be.ok;
});
pm.test("Body is present", function () {
    pm.response.to.be.withBody;
});
pm.test("Content-Type header is present", function () {
    pm.response.to.have.header("Content-Type");
});
pm.test("State code is OK", function () {
    pm.response.to.be.json;
});
pm.test("Response time is less than 200ms", function () {
    pm.expect(pm.response.responseTime).to.be.below(200);
});
```

Test Results

PASS Code Status text is 200
PASS Status text is OK
PASS State code is OK
PASS Body is present
PASS Content-Type header is present
PASS State code is OK
FAIL Response time is less than 200ms | AssertionError: expected [illegible] to be below 200



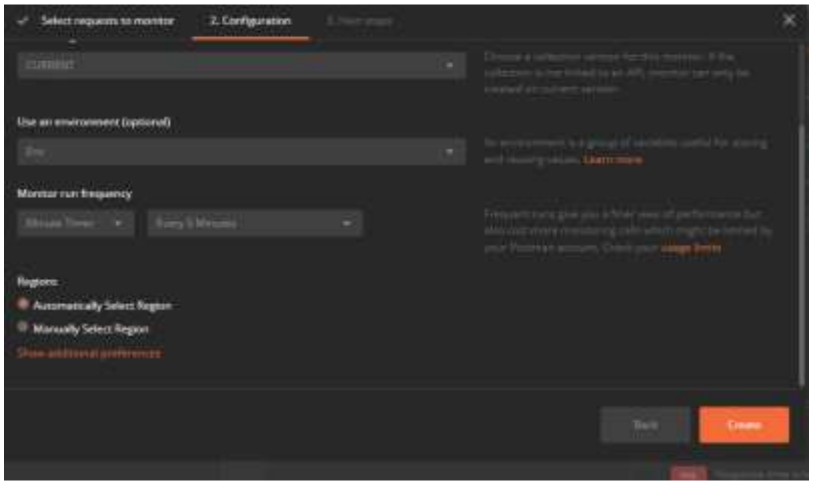
DELETE [illegible]/delete/63

```
pm.test("Code Status text is 200", function () {
    pm.response.to.have.status(200);
});
pm.test("Status text is OK", function () {
    pm.response.to.have.status("OK");
});
pm.test("State code is OK", function () {
    pm.response.to.be.ok;
});
pm.test("Body is present", function () {
    pm.response.to.be.withBody;
});
pm.test("Content-Type header is present", function () {
    pm.response.to.have.header("Content-Type");
});
pm.test("State code is OK", function () {
    pm.response.to.be.json;
});
pm.test("Response time is less than 200ms", function () {
    pm.expect(pm.response.responseTime).to.be.below(200);
});
```

Test Results

PASS Code Status text is 200
PASS Status text is OK
PASS State code is OK
PASS Body is present
PASS Content-Type header is present
PASS State code is OK
FAIL Response time is less than 200ms | AssertionError: expected [illegible] to be below 200

5.3 Set up a monitor.



Select requests to monitor — 2. Configuration — 3. Next steps

CURRENT

Choose a collection version for this monitor. If the collection is not linked to an API, monitor can only be created on current version.

Use an environment (optional): Env

An environment is a group of variables useful for storing and reusing values. Learn more

Monitor run frequency: Minute Timer — Every 5 Minutes

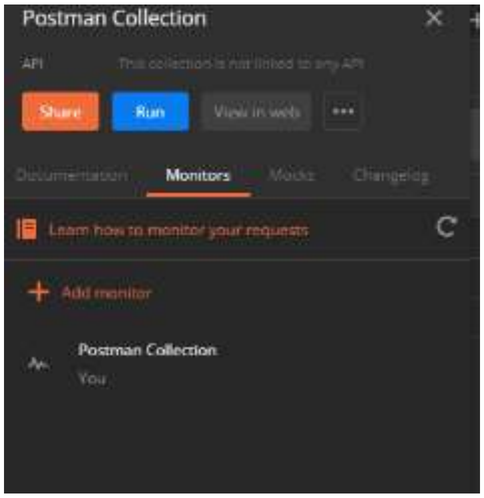
Frequent runs give you a finer view of performance but also use more monitoring calls which might be limited by your Postman account. Check your usage limits

Regions

Automatically Select Region

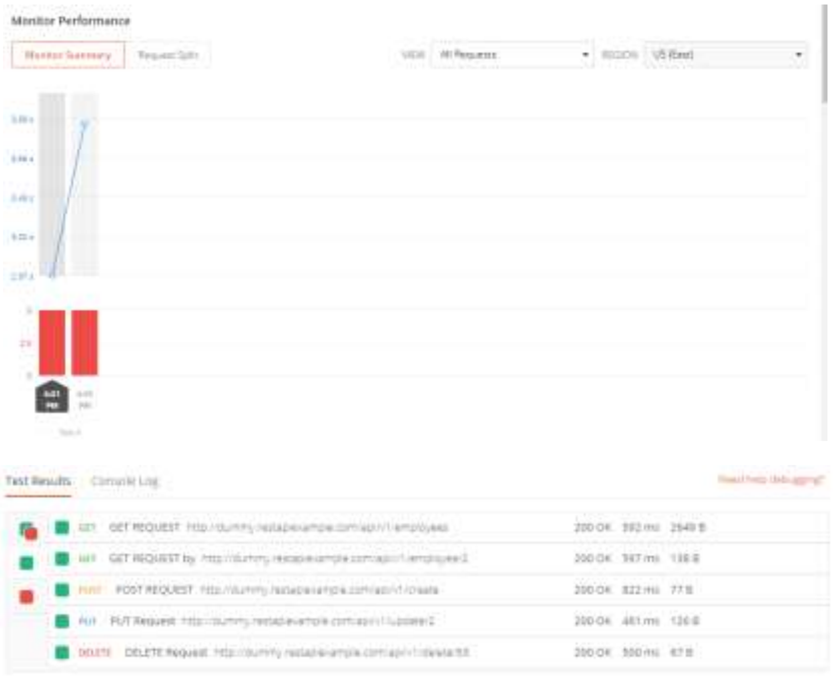
Manually Select Region

Show additional preferences



Postman Collection

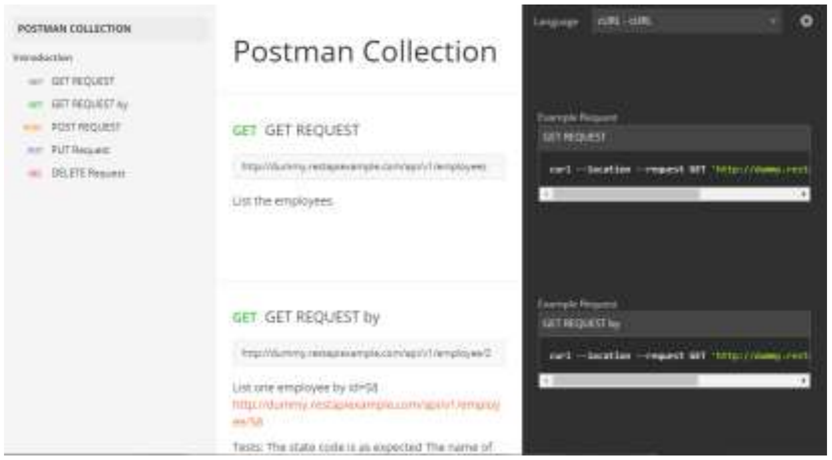
API This collection is not linked to any API



Monitor Performance

```
GET     GET REQUEST     http://dummy.restapiexample.com/api/v1/employees      200 OK  392 ms  2649 B
GET     GET REQUEST by  http://dummy.restapiexample.com/api/v1/employee/2     200 OK  387 ms  136 B
POST    POST REQUEST    http://dummy.restapiexample.com/api/v1/create         200 OK  822 ms  77 B
PUT     PUT Request     http://dummy.restapiexample.com/api/v1/update/2       200 OK  481 ms  126 B
DELETE  DELETE Request  http://dummy.restapiexample.com/api/v1/delete/58      200 OK  500 ms  67 B
```

5.4 API Document.



Postman Collection

GET GET REQUEST

http://dummy.restapiexample.com/api/v1/employees

List the employees

```
curl --location --request GET 'http://dummy.rest
```

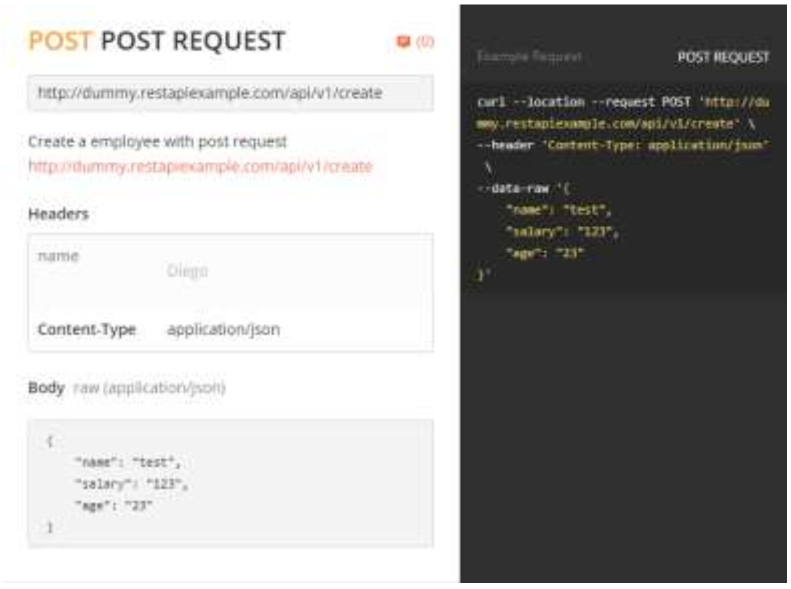
GET GET REQUEST by

http://dummy.restapiexample.com/api/v1/employee/2

List one employee by id=58
http://dummy.restapiexample.com/api/v1/employee/58

Tests: The state code is as expected The name of

```
curl --location --request GET 'http://dummy.rest
```


POST POST REQUEST

http://dummy.restapiexample.com/api/v1/create

Create a employee with post request
http://dummy.restapiexample.com/api/v1/create

Headers

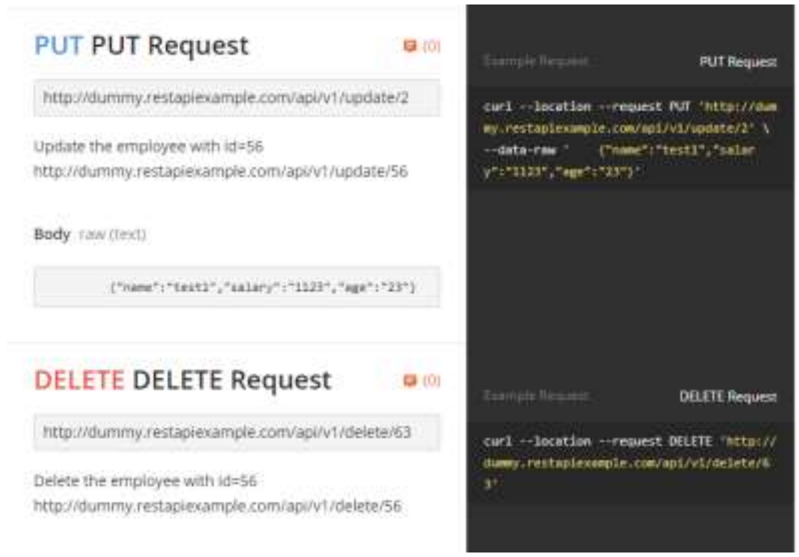
name	Diego
Content-Type	application/json

Body raw (application/json)

```
{
    "name": "test",
    "salary": "123",
    "age": "23"
}
```

Example Request POST REQUEST

```
curl --location --request POST 'http://du
mmy.restapiexample.com/api/v1/create' \
--header 'Content-Type: application/json'
 \
--data-raw '{
    "name": "test",
    "salary": "123",
    "age": "23"
}'
```


PUT PUT Request

http://dummy.restapiexample.com/api/v1/update/2

Update the employee with id=56
http://dummy.restapiexample.com/api/v1/update/56

Body raw (text)

```
        {"name":"test1","salary":"1123","age":"23"}
```

Example Request PUT Request

```
curl --location --request PUT 'http://dum
my.restapiexample.com/api/v1/update/2' \
--data-raw '    {"name":"test1","salar
y":"1123","age":"23"}'
```

DELETE DELETE Request

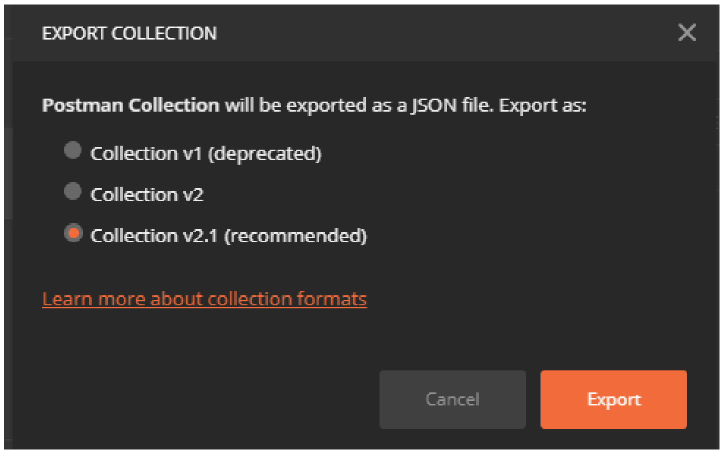
http://dummy.restapiexample.com/api/v1/delete/63

Delete the employee with id=56
http://dummy.restapiexample.com/api/v1/delete/56

Example Request DELETE Request

```
curl --location --request DELETE 'http://
dummy.restapiexample.com/api/v1/delete/6
3'
```

5.5 Export & Import collection.

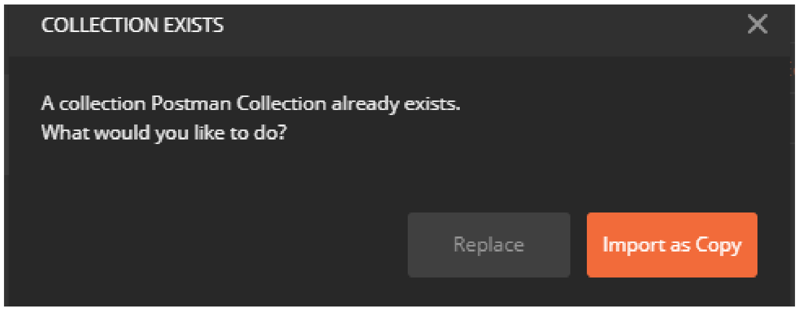

EXPORT COLLECTION

Postman Collection will be exported as a JSON file. Export as:

- ○ Collection v1 (deprecated)
- ○ Collection v2
- ● Collection v2.1 (recommended)

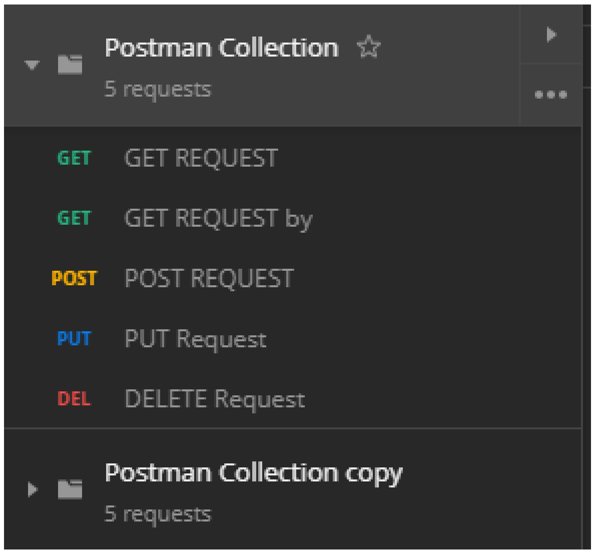
Learn more about collection formats

After export and import the collection, a message appear display this:


COLLECTION EXISTS

A collection Postman Collection already exists.
What would you like to do?

Then, the collection was created like a copy.



6. Monitoring Control.

The requirements were join into a list for last checking and avoid delivery issues. The table was created for compare each requirement with the development.