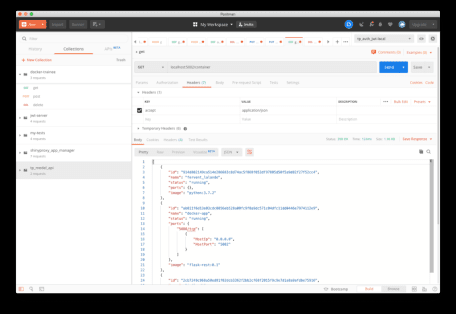
**About Postman**

Postman is a convenient desktop tool handling REST request. Furthermore, Postman gives you the possibility to define test cases (in JavaScript), has a feature to switch environments, and provides you with Pre-Request steps to set up the setting before your calls. In the following, I will give you examples of some interesting features.

**Collection and Requests**

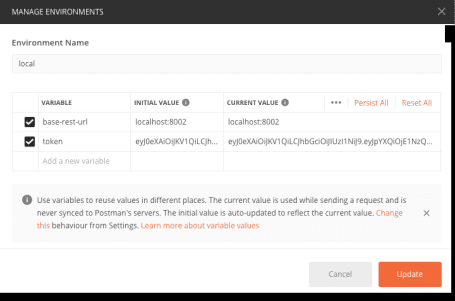
Requests are the basic unit in Postman, and everything else spins around them. As I said previously, Postman’s GUI provides you with a comfortable way to define these: request method can be picked from a drop-down list, header information is presented clearly, there is a helper for authorization, and many more.



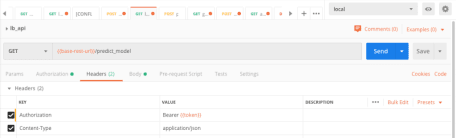
You should have at least one collection per REST interface defined to bundle your requests. At the very end of the definition process, collections can be exported into JSON format. This result will, later on, be exploited for Newman.

**Environments**

Postman also implements the concept of environment variables. This means: Depending on where your requests are fired from, the variables adapt. The API’s hostname is a good example that should be kept variable: In the development stage, it may be just your localhost but could be different in a dockerized environment.



The syntax of environment variables is double-curly brackets. If you want to use the hostname variable hostname put it like this: {{ hostname }}



Like for collections, environments can be exported into JSON files. We should keep this in mind when we move to Newman.

**Tests**

Each API request in Postman should come along with at least one test. I propose the following list as an orientation on what to test:

* **the status code**: Check the status code according to your expectation: regular GET requests are supposed to return *200 OK*, POST requests *201 Created* if successful. On the other hand, authorization should be tested as well as invalid client requests which are supposed to return 40x. – See below a POST request test:

pm.test("Successful POST request", function () {

pm.expect(pm.response.code).to.be.oneOf([201,202]);

});

* **whether data is returned** Test if the response has any data as a first approximation
* **the schema of returned data** Test if the structure of the request data fits the expectations: non-nullable fields, data types, names of properties. Find below an example of a schema validation:

pm.test("Response has correct schema", function () {

var schema = {"type":"object",

"properties":{

"access\_token":{"type":"string"},

"created\_on":{"type":"string"},

"expires\_seconds":{"type":"number"}

}};

var jsonData = pm.response.json();

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

});

* **values of returned data**: Check if the values of the response data are sound; for non-negative values:

pm.test("Expires non negative", function() {

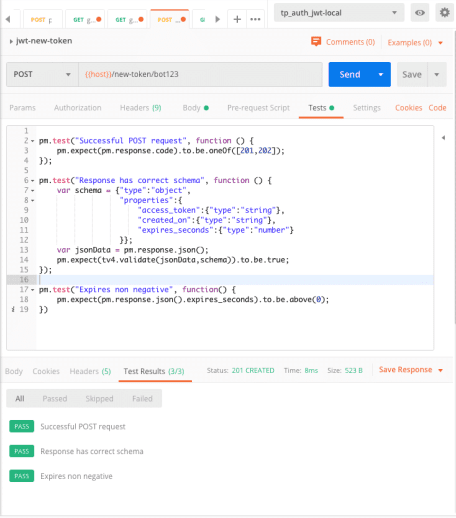
pm.expect(pm.response.json().expires\_seconds).to.be.above(0);

})

* **Header values** Check the header of the response if useful relevant is stored there.

All tests have to be written in JavaScript. Postman ships with its own library and tv4 for schema validation.

Below you find a complete running test:



**Introduction to Newman**

As mentioned before, Newman acts as an executor of what was defined in Postman. To generate results, Newman uses reporters. Reporters can be the command line interface itself, but also known standards as JUnit can be found.  
The simplest way to [install newman](https://www.npmjs.com/package/newman#getting-started) is via [NPM](https://www.npmjs.com/) (Node package manager). There are ready to use docker images of NodeJS on [DockerHub](https://hub.docker.com/_/node/). Install the package via npm install -g newman.

There are two ways to call Newman: command-line interface and within JS code. We will only focus on the first.

**Calling the CLI**

To run a predefined test collections use the command newman run. Please see the example below:

newman run

--reporters cli,junit

--reporter-junit-export /test/out/report.xml

-e /test/env/auth\_jwt-docker.pmenv.json

/test/src/auth\_jwt-test.pmc.json

Let us take a closer look: Recall that we have previously exported the collection and the environment from Postman. The environment can be attached with the -e option. Moreover, two reporters were specified: the *cli* itself which prints into the terminal and *junit* which additional shall export a report to the file report.xml

The CLI reporter prints the following (Note that the first three test cases are those from the test schema proposal):

→ jwt-new-token

POST <http://tp_auth_jwt:5000/new-token/bot123> [201 CREATED, 523B, 42ms]

✓ Successful POST request

✓ Response has correct schema

✓ Expires non negative

→ jwt-auth

POST <http://tp_auth_jwt:5000/new-token/test> [201 CREATED, 521B, 11ms]

GET <http://tp_auth_jwt:5000/auth> [200 OK, 176B, 9ms]

✓ Status code is 200

✓ Login name is correct

→ jwt-auth-no-token

GET <http://tp_auth_jwt:5000/auth> [401 UNAUTHORIZED, 201B, 9ms]

✓ Status is 401 or 403

→ jwt-auth-bad-token

GET <http://tp_auth_jwt:5000/auth> [403 FORBIDDEN, 166B, 6ms]

✓ Status is 401 or 403

**Integration into Jenkins**

Newman functionality can now be integrated into (almost?) any Pipeline tool. For Jenkins, we create a docker image based on NodeJS and with Newman installed. Next, we either pack or mount both the environment and the collection file into the docker container. When running the container, we use Newman as a command-line tool, just as we did before. To use this in a test stage of a Pipeline, we have to make sure that the REST API is actually running when Newman is executed.

In the following example, the functionalities were defined as targets of a Makefile:

* run to run the REST API with all dependencies
* test to run Newman container which itself runs the testing collections
* rm to stop and remove the REST API

After the API has been tested the report from JUnit is digested by Jenkins with the command junit

See below a Pipeline snippet of a test run:

node{

stage('Test'){

try{

sh "cd docker && make run"

sh "sleep 5"

sh "cd docker && make test"

junit "source/test/out/report.xml"

} catch (Exception e){

echo e

} finally {

sh "cd docker && make rm"

}

}

}

**Summary**

Now it’s time to code tests for your REST API. Please also try to integrate it into your build-test cycle and into your automation pipeline because automation and defined processes are crucial to delivering reliable code and packages.