Easy Integration testing using Postman and Newman

20 January 2023 - 8 min. read

CI/CD *DevOps* *Integration Testing* *Newman* *Postman*

Introduction

Welcome back to the **Proud2beCloud** blog. Today we'll be exploring how to implement integration testing for your API directly into your pipelines by leveraging Postman and Newman.

We'll talk about Newman, Pipelines and CodeBuild, Testing, scripting, sharability, and much more.

Let's dive into how to test APIs automatically and... start testing today!

Why should I need to do integration testing?

To be able to safely deploy APIs in production we want to be sure that the expected behavior is going to be followed. **With unit testing alone, we cannot be sure of that.**

Let's put a simple example: imagine having an application unit tested perfectly, hence the code is working as expected given a known input. In front of the application, we'll then have a user calling an API and passing some data to the app. Unit tests are simply not enough; we want to be able to test the effective format of variables passed to the application logic, test headers, and body prior to utilization, timeouts, different status code, and other test cases: write tests for integration.

In real-world applications, behaviors change over time, and modules connected to each other may face changes in how the data is passed onto the other and in API requests.

We may have some third-party applications integrated into our solution where we need to call external API and test using their responses.

Sometimes (*but it should not happen!*) developers deploy applications quickly, maybe without unit testing and that's where integration testing comes in clutch.

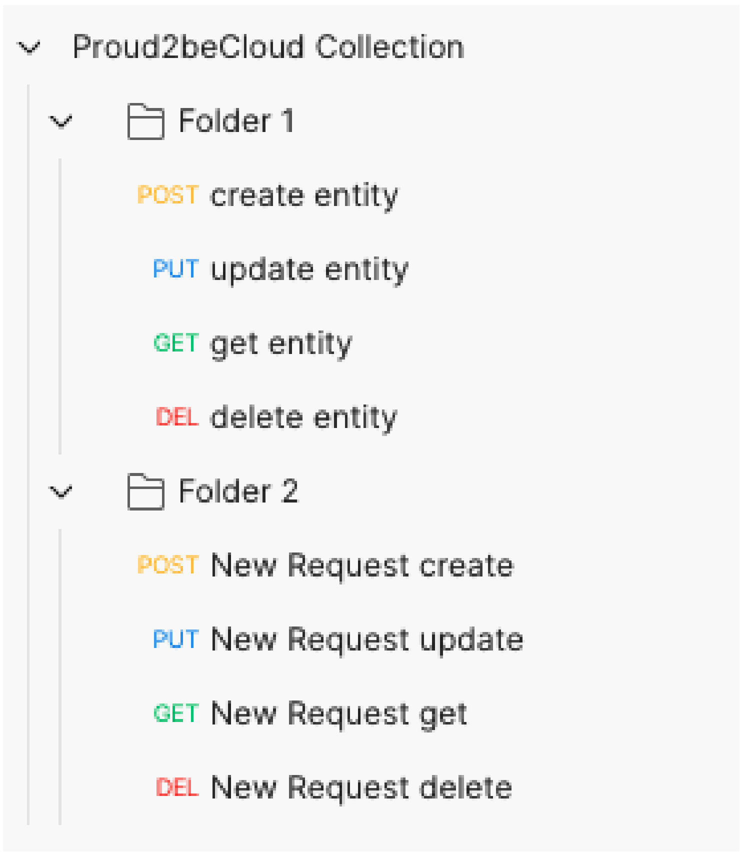
Those are some reasons why we need this integration testing: to **test how various modules (route, validation, application, data layer) interact with each other, after changes and overtime.**

In our case, talking about API Gateway and Lambda, we also want to test the route, decorators, and whatever is in between the user and the application code.

Let's have a quick look at Postman

Best practices:

- Split APIs into separate collections based on the project
- Separate API into subfolders based on entity and/or category
- Write APIs in a logic order POST - GET - PUT - DELETE
- Use variables when possible. It helps to generate random (and more realistic) data in input and greatly helps when switching environments or during modification {{*variabile*}}



Variables

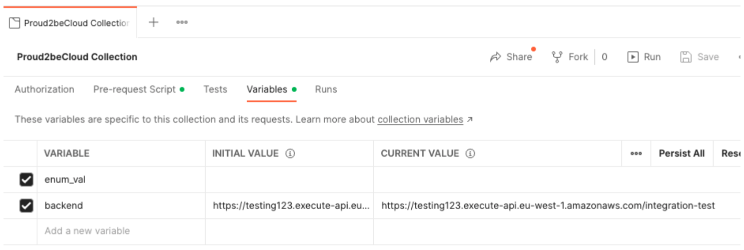
In Postman two different types of variables are available:

1. Collection
2. Environment

Collection:

Those are variables for the whole collection, not depending on what environment we are in.

They should be utilized for the value of parameters that are environment-independent like custom application names, business names, etc.

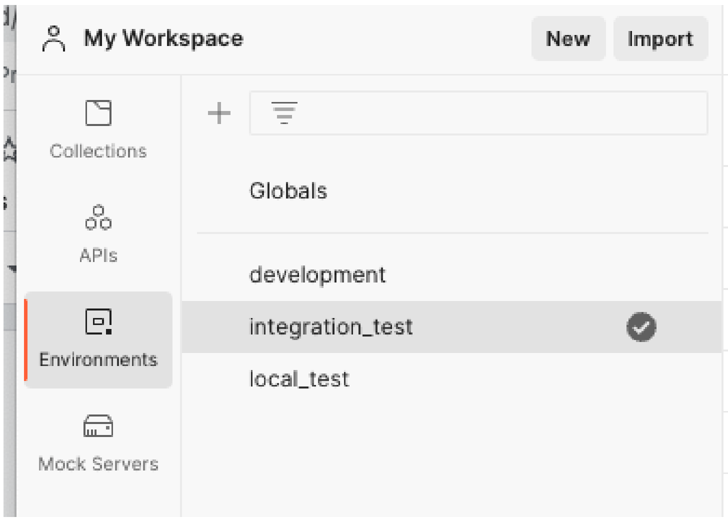


These variables are specific to this collection and its requests. Learn more about collection variables ↗

	VARIABLE	INITIAL VALUE	CURRENT VALUE
✓	enum_val		
✓	backend	https://testing123.execute-api.eu...	https://testing123.execute-api.eu-west-1.amazonaws.com/integration-test
	Add a new variable		

Environment:

In a collection it is also possible to create multiple environments; we can define variables in these environments too.



If the same variable key is present in an environment *and* in the collection, the environment wins and the collection variable will be overridden.

Best practices, in my opinion:

Create at least a local env, an integration test one, and/or a development one. Switching between environments is really useful and quick and setting up a variable like {{base_url}} with various values is gonna become handy.

Note: Postman already has some variables by default, like {{randomEmail}}, {{randomInt}, {{$randomAlphaNumeric}} which are really useful to randomize body in POST and PUT API calls.

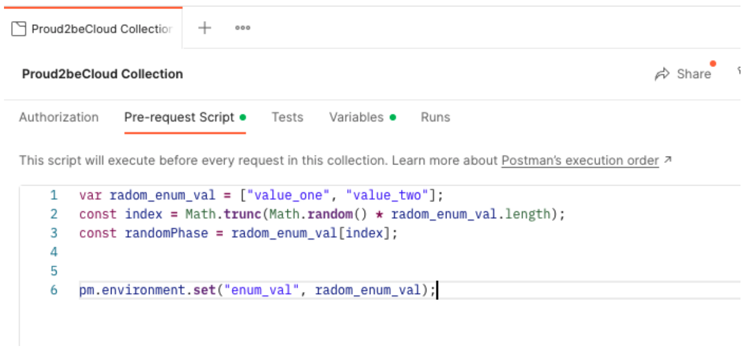
Pre-request script

Both for the whole collection and for a single API call, we have the possibility to use a **pre-Request script,** which is a js-like script to do various stuff. Here's some example:

1. Set up env variables corresponding to ENUM data type and we want to randomize the actual value.

Thanks to this script, it will be possible to have access to the *enum_val* variable that will have a random value between 'value_one' and 'value_two'.

This is useful and should be added to the Collection pre-request script.



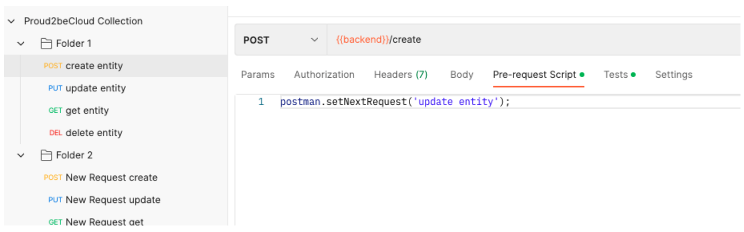
This script will execute before every request in this collection. Learn more about Postman's execution order

```
var radom_enum_val = ["value_one", "value_two"];
const index = Math.trunc(Math.random() * radom_enum_val.length);
const randomPhase = radom_enum_val[index];


pm.environment.set("enum_val", radom_enum_val);
```

2. Set up the next request. Postman by default will run the collection in the order in which APIs are written, if we want to change that order (to leave DELETE last for example), we can do that by using the pre-request script of the single API call:

```
postman.setNextRequest('update entity');
```



```
postman.setNextRequest('update entity');
```

To stop the collection, we can use the same function and set the parameter to null:

```
postman.setNextRequest(null).
```

Is good practice to have a folder structure in our collection as stated before, but lots of times we have dependencies in between entities, so we want to set the *setNextRequest* like this:

Entity1.POST → Entity1.GET → Entity1.PUT → Entity2.POST → Entity2.GET → Entity2.PUT → … → … → … → EntityN.Delete → EntityN-1.DELETE → … → Entity2.DELETE → Entity1.DELETE

To be able to create, update and access objects until we need them and, at last, delete everything back to the main entity.

Tests

As the title hints, to do some integration tests, we need tests.

In this case, there's a dedicated tab where we can execute the script AFTER the API call has been done, work with the response, and test that all reflects what we expect.

We'll use this section for 2 main reasons:

1. **Set up variables based on responses**

   USE CASE eg: I'll execute a POST, expecting a new ID to be created. I want to be able to use the ID in a subsequent GET call. I can do that by getting the response, parsing it to get the ID, and put it in an env variable previously created.

In Other API I can then use that ID by calling the variable {{id_entity}}
HOW TO:

- Create an environment variable with a given key and no value (useful also for clarity and readability of the collection)
- Script the get of the response and set up the variable with this new value

```
var jsonData = JSON.parse(responseBody);
postman.setEnvironmentVariable("productId", jsonData.entityId);
```

With that, the variable will definitely have a value and we'll test the calls sequentially with dynamic data.

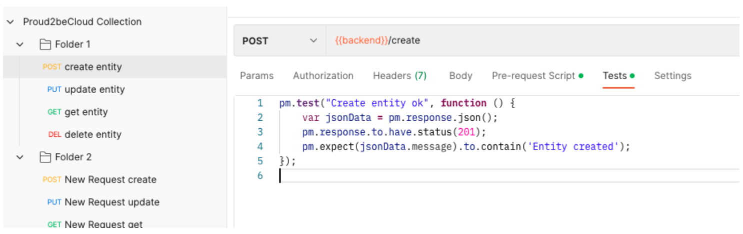
2. **Test the response**

We can invoke a postman function *.test* to actually impose a check on something, based on a boolean evaluation, the test will either fail or succeed.

```
pm.test("Create entity ok", function () {
var jsonData = pm.response.json();
pm.response.to.have.status(201);
pm.expect(jsonData.message).to.contain('Entity created');
});
```

This script gets the response and checks both the body and the status code of the response. If everything inside .test() is correct the test will pass.

A dedicated tab is presented on Postman and, as we're gonna see, Newman will also give a 'report' on how many requests were launched and how many of them failed, and why.



```
pm.test("Create entity ok", function () {
    var jsonData = pm.response.json();
    pm.response.to.have.status(201);
    pm.expect(jsonData.message).to.contain('Entity created');
});
```

Recap: complete Postman collection

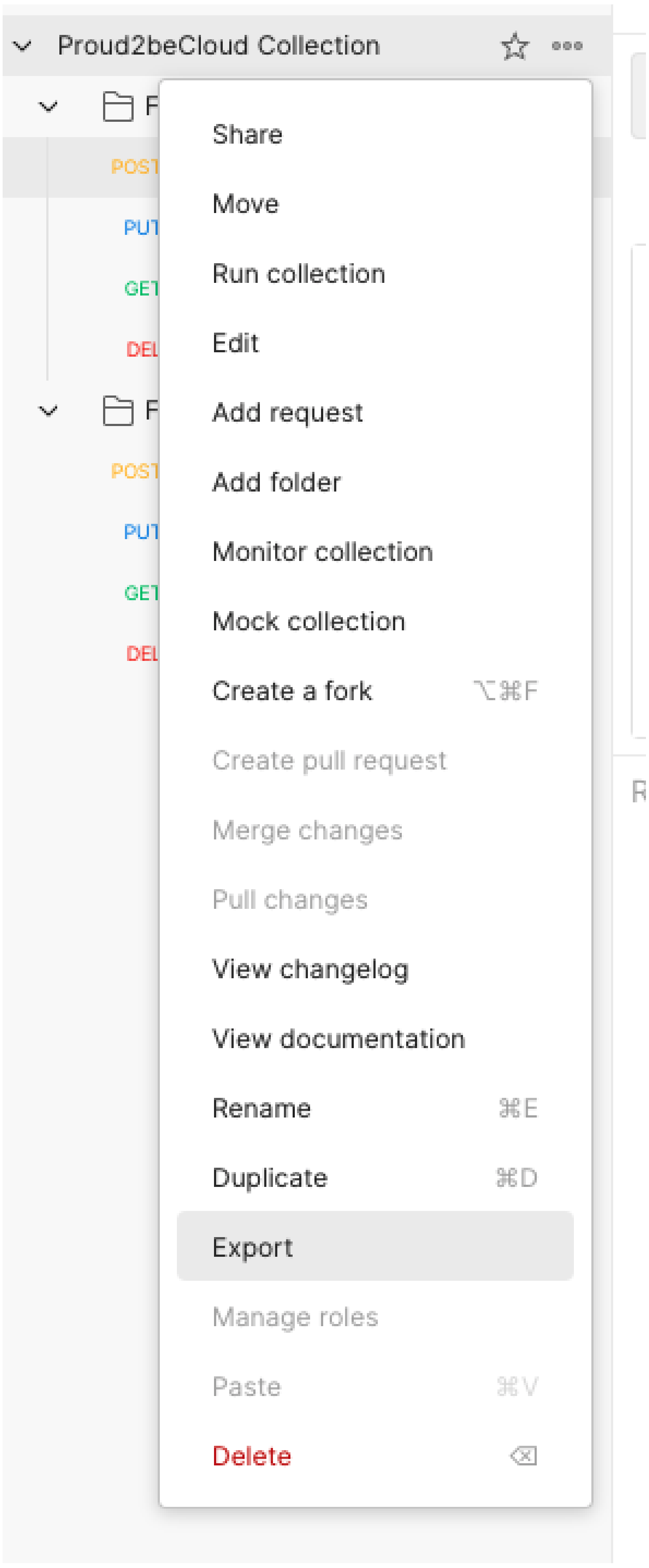
Explained the various steps and features that Postman allows us to use, the process to create a complete collection, ready to be used for integration testing can resemble something like that:

1. Create a workspace
2. Create the collection
3. Create environments if needed
4. Create collections and environment variables
5. Add the pre-request script in the collection if necessary (eg: set custom variables)
6. Create API calls in a logic order (post, put, get, delete)
7. Set .nextRequest in every Pre-request script to orchestrate first hand the order of the APIs.
   Suggested POST, GET, PUT, and after everything, all the DELETEs starting from the smallest entity going up the hierarchy.
8. SetNextRequest(null) at the last API to be sure the collection will stop.
9. Eventually, set dynamic variables in the test tab (es.: ID)
10. Define tests for the single API call

Collection in repository and Newman

Integrating your collection in a pipeline is actually pretty straightforward.

1. Export the collection, which will download the JSON describing all the API calls and collection variables



2. Export, if present, the environment. A JSON with the environment variables will be downloaded

3. Put both files in the repository

4. Create a Code Build step in CodePipeline. It is important to install Newman from npm. The buildspec should look something like this:

```
version: 0.2
phases:
   install:
       runtime-versions:
           nodejs: 14
build:
   commands:
     - npm install -g newman
     - ./run_integration_test.sh
```

5. Create the *run_integration_test.sh* file where we'll actually run the collection like this:

```
#!/bin/bash


newman run postman_collection.json -e integration_test_env.json
```

By doing so, Newman will run the postman collection using the environment variable present in the *integration_test_env* file.

The collection will run in order from the first API and then following the trace created by the .setNextRequest.

Every request with a test written will be evaluated; if even one throws an error or doesn't return what's expected, the collection will have a KO status and the code build step will fail.

Additional notes

Sharability: via workspaces, it is possible to create a workspace for every client/project and share it with the team (free up to 3 users) or share it by a link to viewers. The editors will see the collection updated in real time from other colleagues and won't need to import it every time.

AWS Environment: to be able to actually call APIs we need the code to be deployed prior to testing, this is not feasible for the production environment and often for staging it can be an issue, too.

We found a solution by deploying every time, with every pipeline that needs integration testing (staging & production), on a *shadow env* which is actually present on AWS with API GW, Lambdas, etc.. but is not connected to a CloudFormation dedicated stack and does not have a pipeline either.

With that we can safely deploy on 'integration test env' , test the API and even if the deployment fails we won't have affected real environments.

Conclusion

Integration testing is crucial in application development and will make sure that production will behave as we expect.

To do that, tools like Postman and Newman are super useful and almost completely configurable to best fit business needs; they are even codebase-agnostic as it's just an npm package running a .json so they can be integrated into python projects, node ones, etc..

Integrating them in a dedicated shadow environment on AWS and executing them in the pipeline will make your CI/CD more reliable and will make it easy to control your code which is always a good idea.

Hope this article helped you out! Feel free to leave a comment with questions, or doubts, or tell us your experience with integration testing.

Stay tuned for our next article and, as always, be **#Proud2beCloud**!

Alberto Casadei

DevOps Engineer @ beSharp. I approach problems with a 'lazy' mindset only to find the best and more efficient solution!Cards collector and athlete, "a 'bit' of everything" is the way to go!
