

# Technical Documentation

This Automated Postman Collection is helpful to run the smoke tests for the rest API in a short period of time, easily, and having all the test cases tested.

Each request has test assertions in order to verify if all the expected results are what we expect to be.

Some of the requests have pre-request scripts that sets all the variables dynamically and ease our work.

By setting all the variables dynamically, we can use the collection runner and test all the requests with a few clicks.

If all the tests are passing the assertions, we know that the rest API is fine and it wasn't affected by the new changes made during the sprint.

- Why creating a new environment when you run the collection for the first time ?
- Automatically follow redirects must be set as OFF
- How requests are written
- Collection runner, why are some settings required when you use it ?
- Why are the two requests separated in another collection ?

When you run the collection for the first time you need to create an environment. Why ?

The variables are created in the environment. That is why we need to set up one. There are a lot of variables that are saved in the environment.

When a request is run, some values needed for that request are obtained from other requests. If we save them in the environment, we can use and reuse them in other requests, that follow the one which created them.

We also create variables in the pre-requested scripts then we bind them into the request itself using curly braces. Those variables are created in the environment.

Why creating variables at environment level and not at collection level ?

Having the variables created at collection level, makes them visible only to that specific collection. This is not what we want. We need the variables for the second collection also.

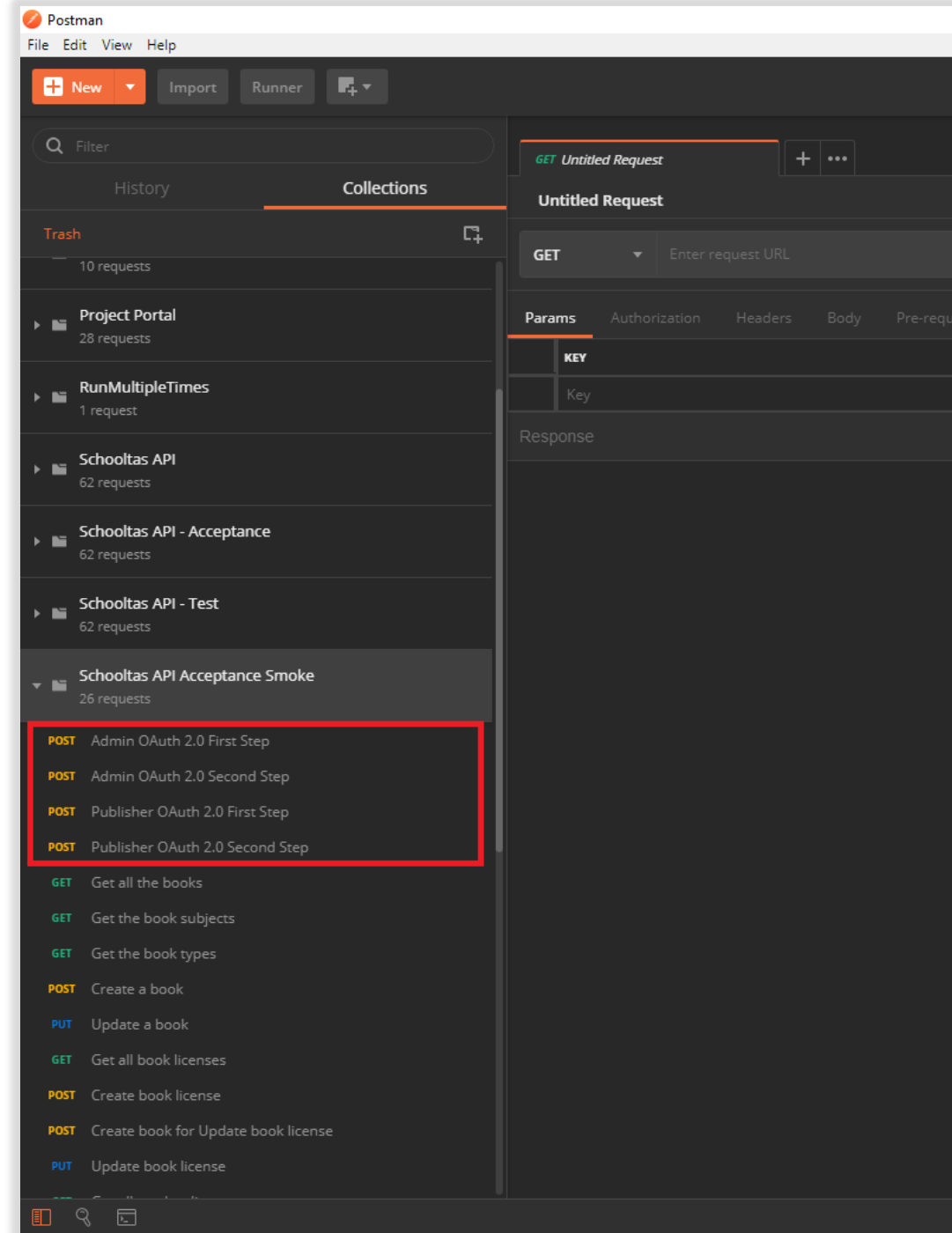
By creating them at the environment level, we have visibility for them on the second collection also having the right environment selected.

In the first collection “Schooltas API Acceptance/Production Smoke” are four POST requests, those requests are for authenticate one Publisher account and one School account (It is called Admin OAuth ..., because the account used for authentication is an admin account also).

The first two requests are for the School authentication, and it is split into two steps (this is because of the OAuth2 principles, it requires a two steps authentication process).

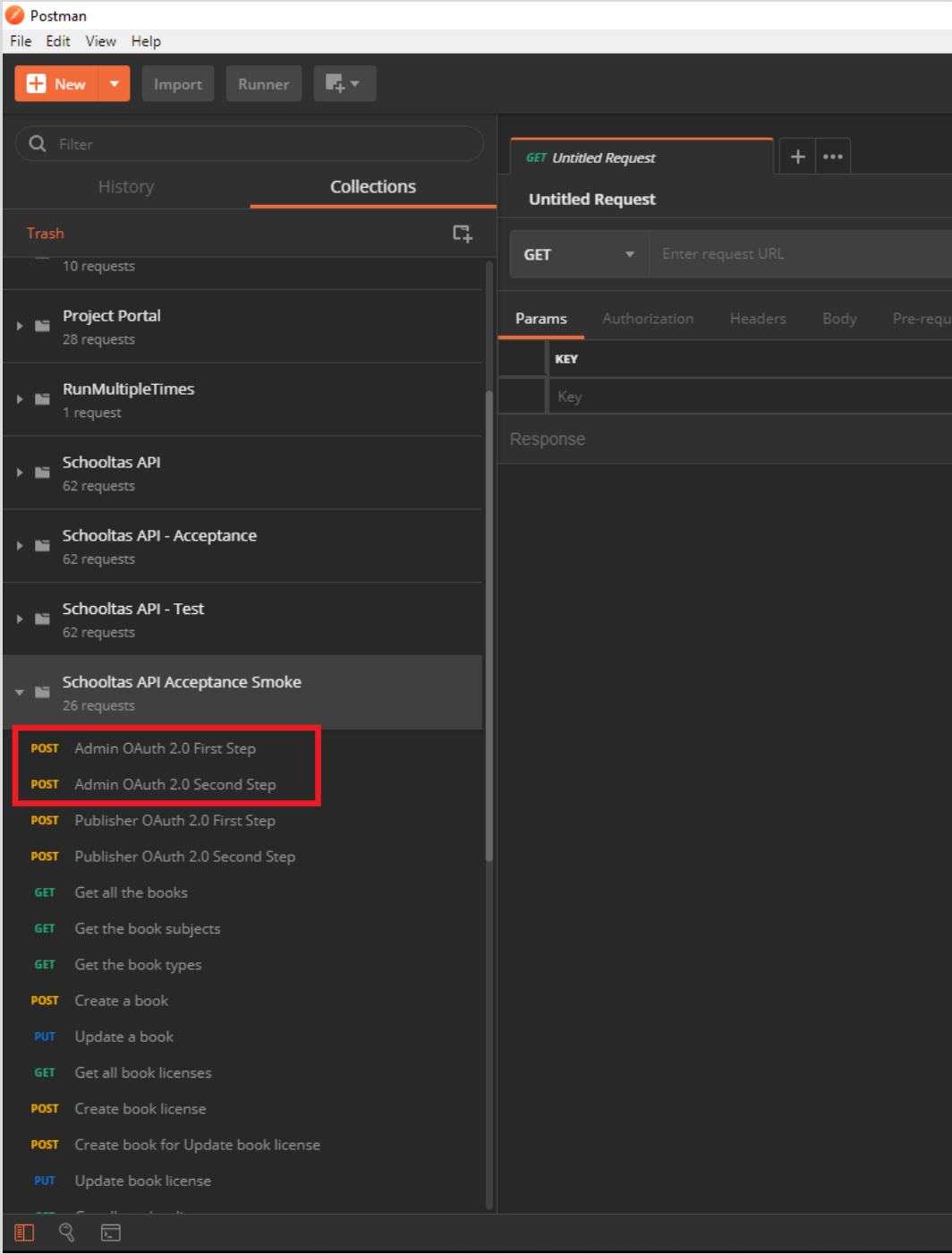
The first step, provides us the code for the second step. The second step, provides us the Access Token which we use later when we make requests to the backend services.

All the requests under the first four ones, require an access token, based on their needs (if the request is made by a publisher, it needs the publisher access token, same for school).

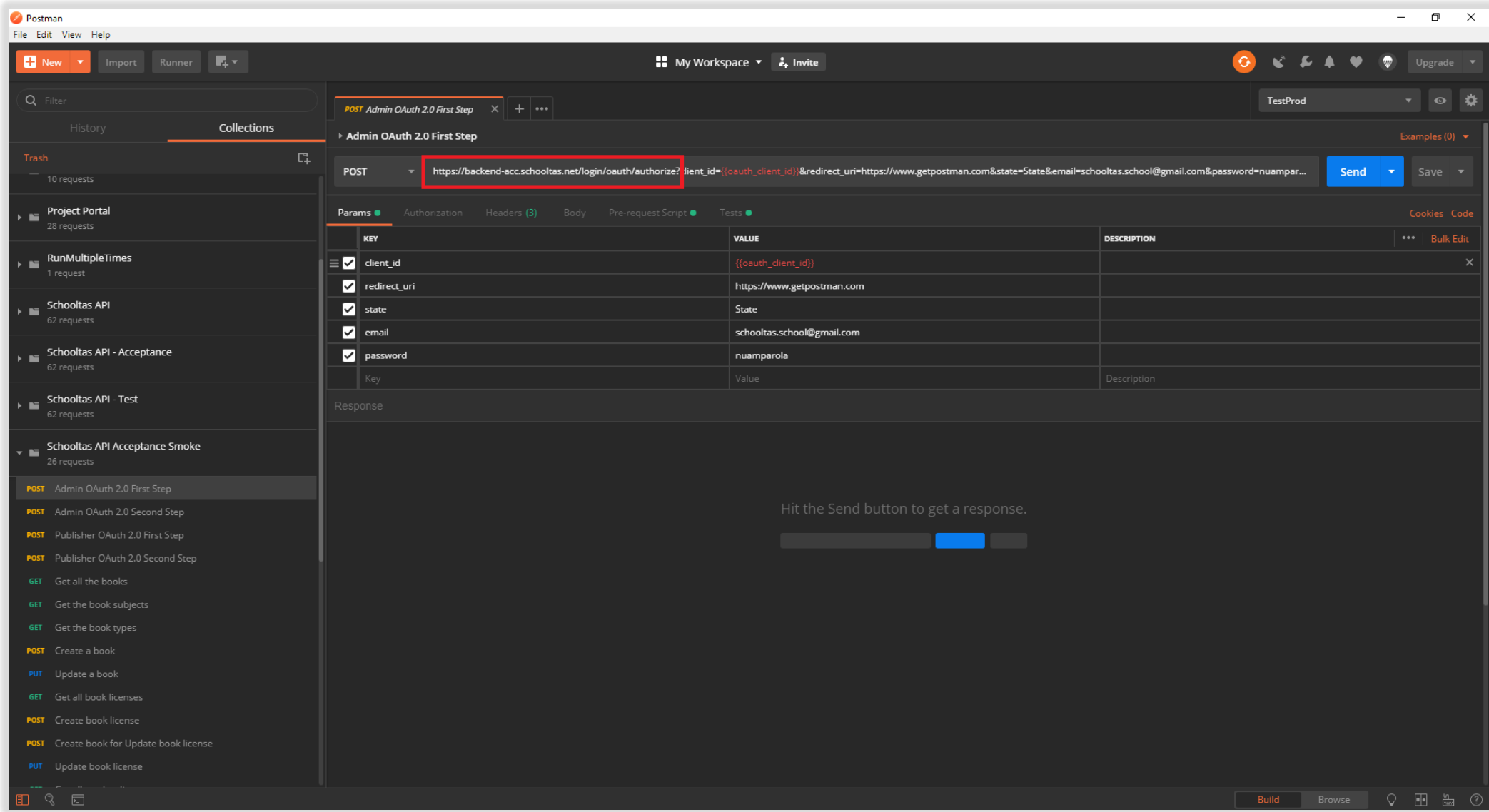


In order to do so, the “Automatically follow redirects” must be set to OFF. By setting this to OFF, we can capture the code in the first request, and pass it to the second request in the authentication process.

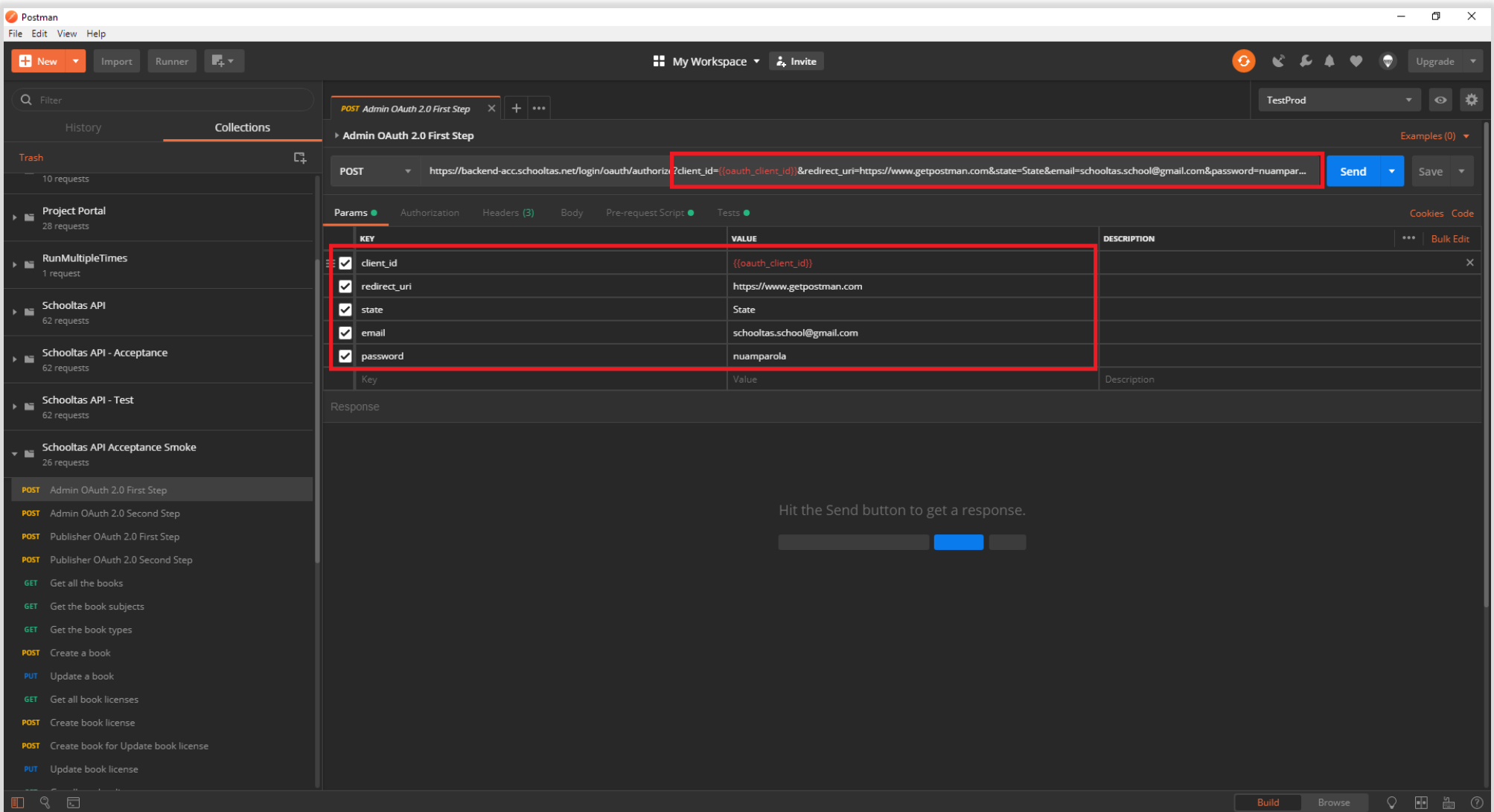
To better understand how the two steps authentication are made, I will explain the authentication for the school, the one for the publisher being similar.



Below you can find the URL marked in red to which the request is sent. When sending the POST method for the request, there are some required parameters that need to be specified. The executed request will generate the code for the next step.

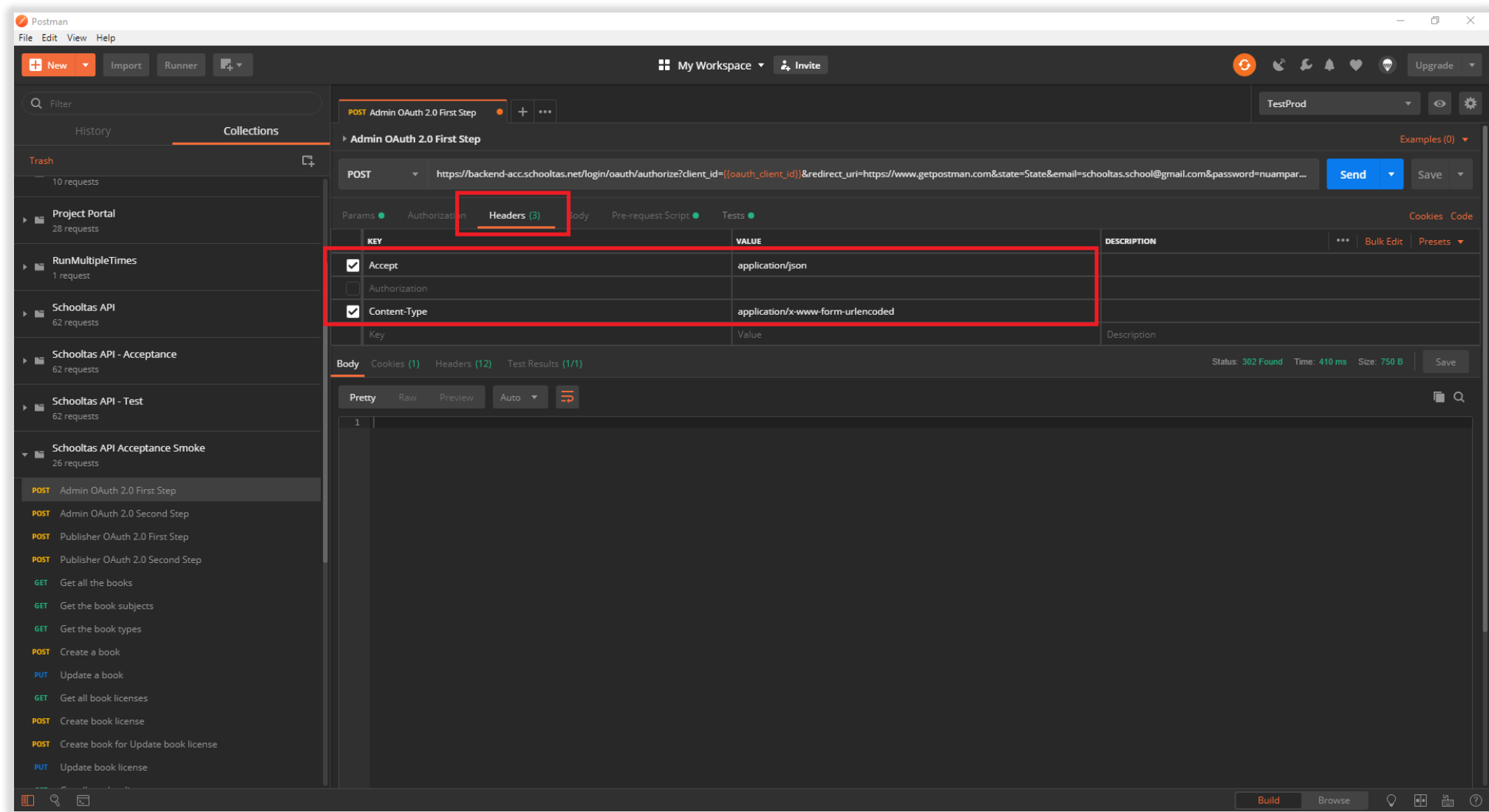


Those are the parameters needed for the request. The value of the `client_id` is in curly braces. This means that this parameter will get the value from somewhere else (from environment, collection, global, or other places. In our case it will take it from the environment place).

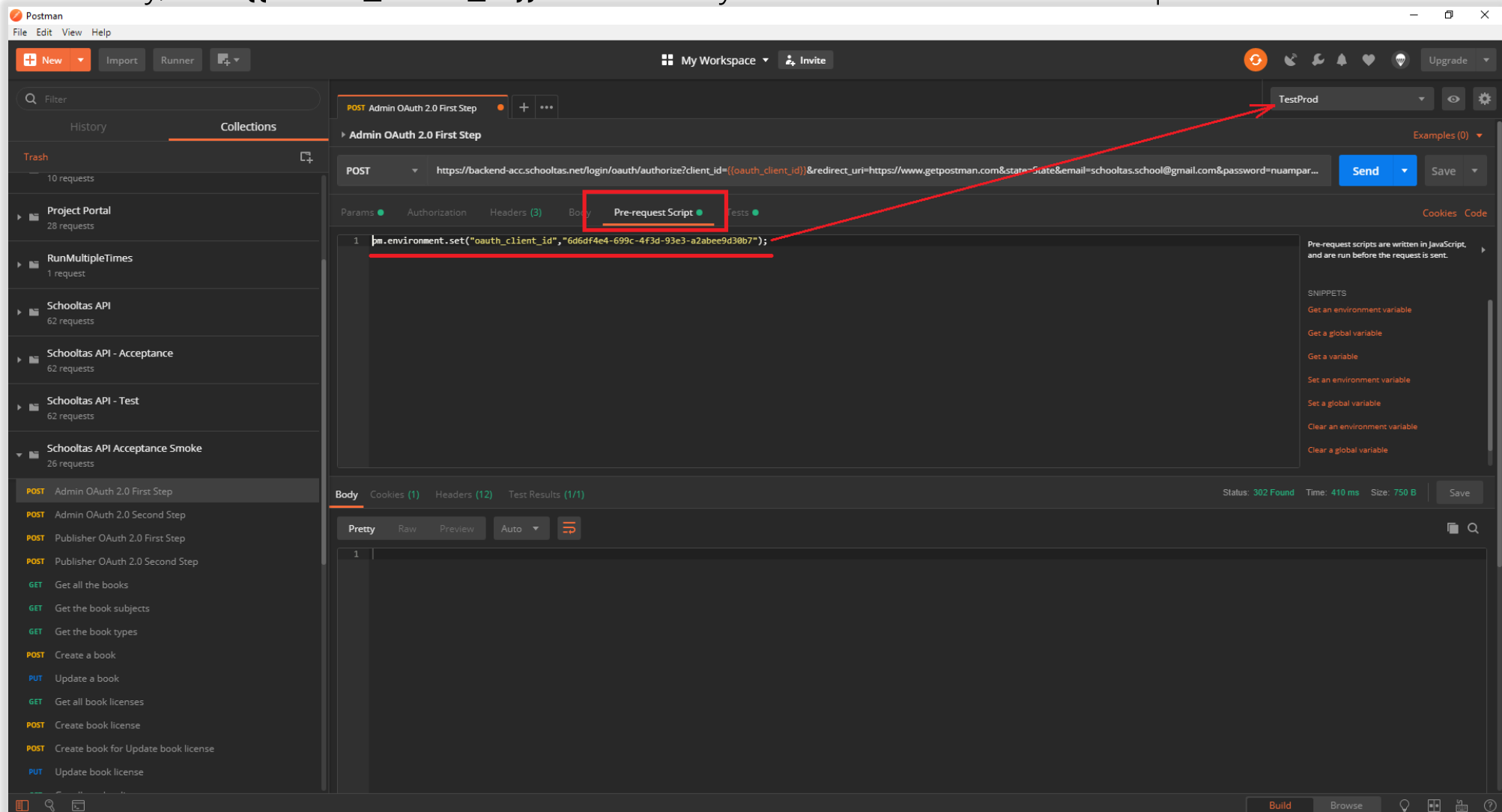




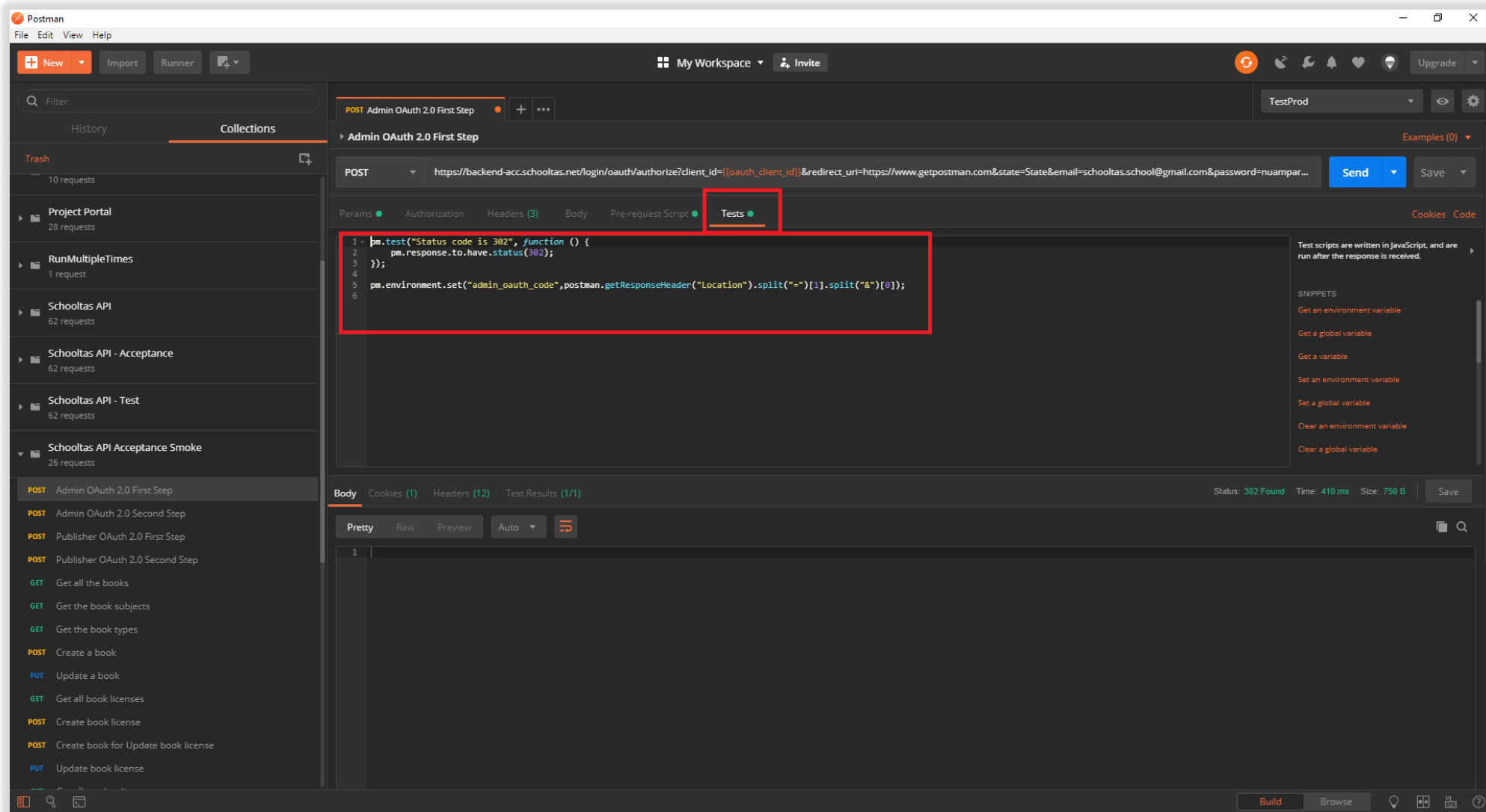
Since we have an empty body, we don't need the keys and the values specified in the headers section. So this means they can be unchecked. I will explain the need of them later on.



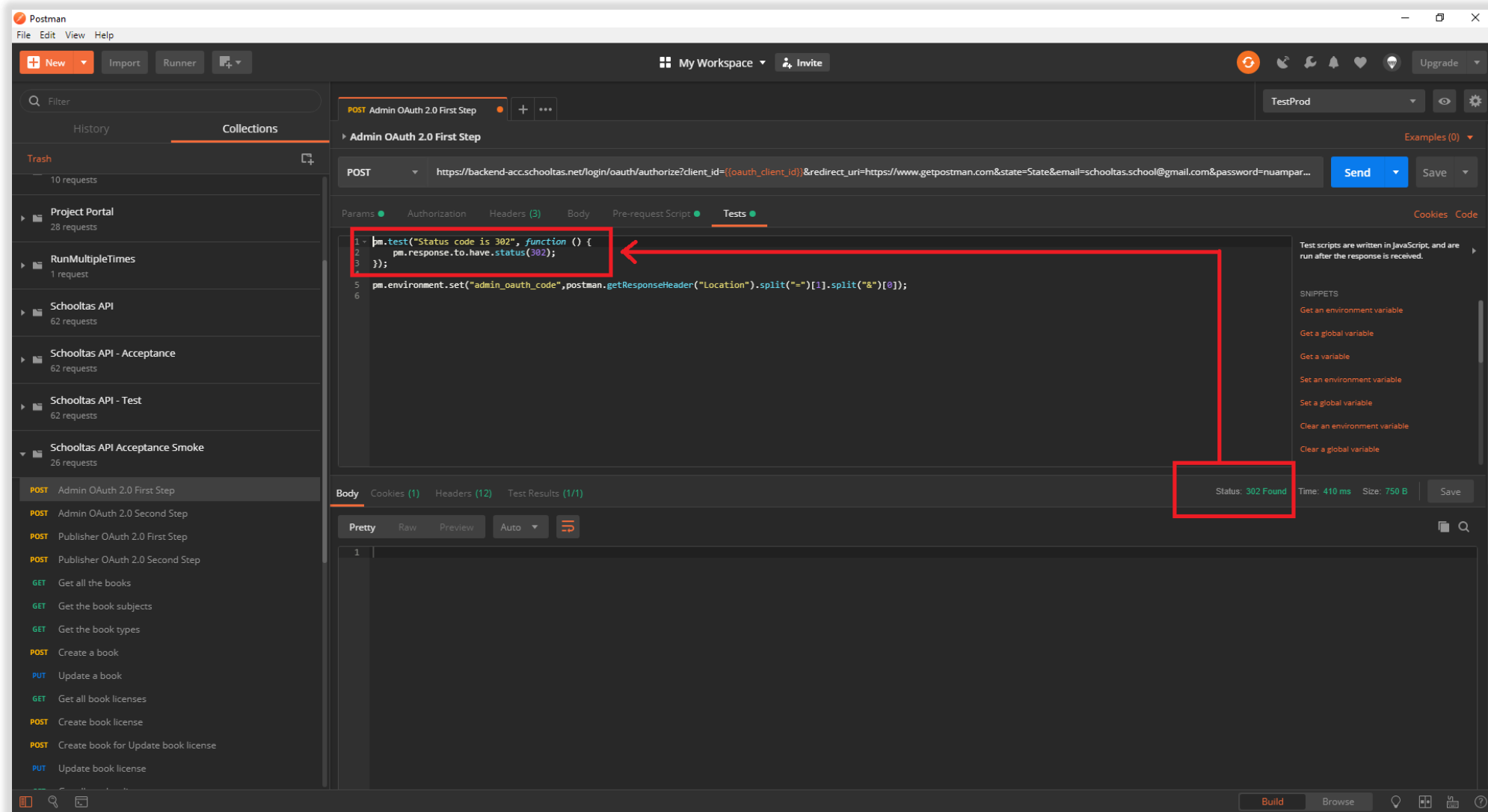
Pre-request Scripts are wrote in Javascript language and they are ran before the request is sent. They are useful for setting the variables which are required in the request call. This line of code sets a variable with the name **oauth\_client\_id** in the selected environment, using the value passed by the **set** function. This environment variable will be created when the request is sent, but before the actual call at the backend service. In this way, the **{{oauth\_client\_id}}** will actually have a value and the request will not crash.



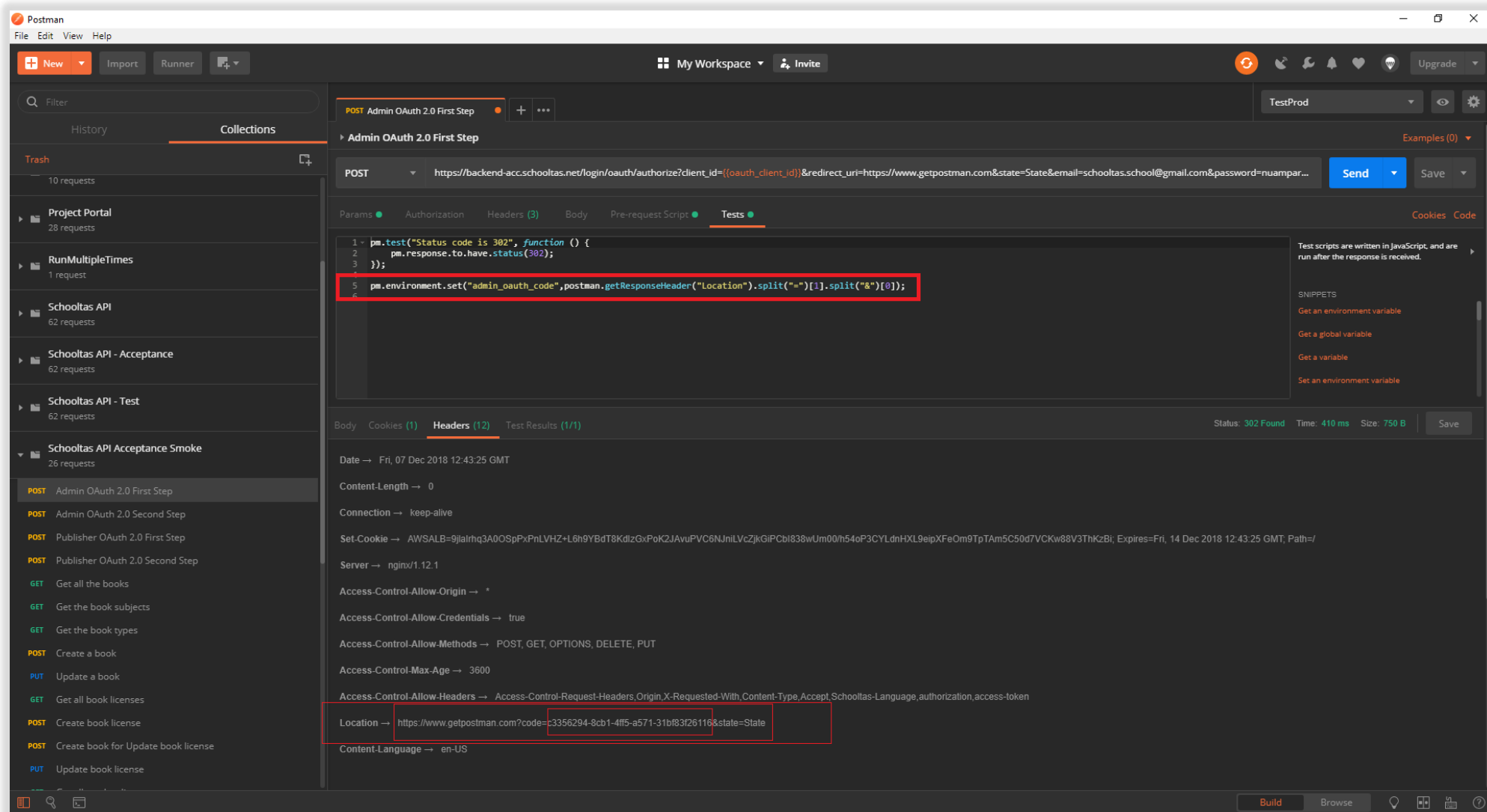
The assertions are made in the Tests tab. This is also a Javascript script which will be ran after the request receives a response. Here we can do all the things that you can do in the pre-request scripts, having a plus, the assertions.



The first piece of code is an assertion. This verifies if the response status code is 302. If so, the test result will have the status “passed”, otherwise, it will fail.

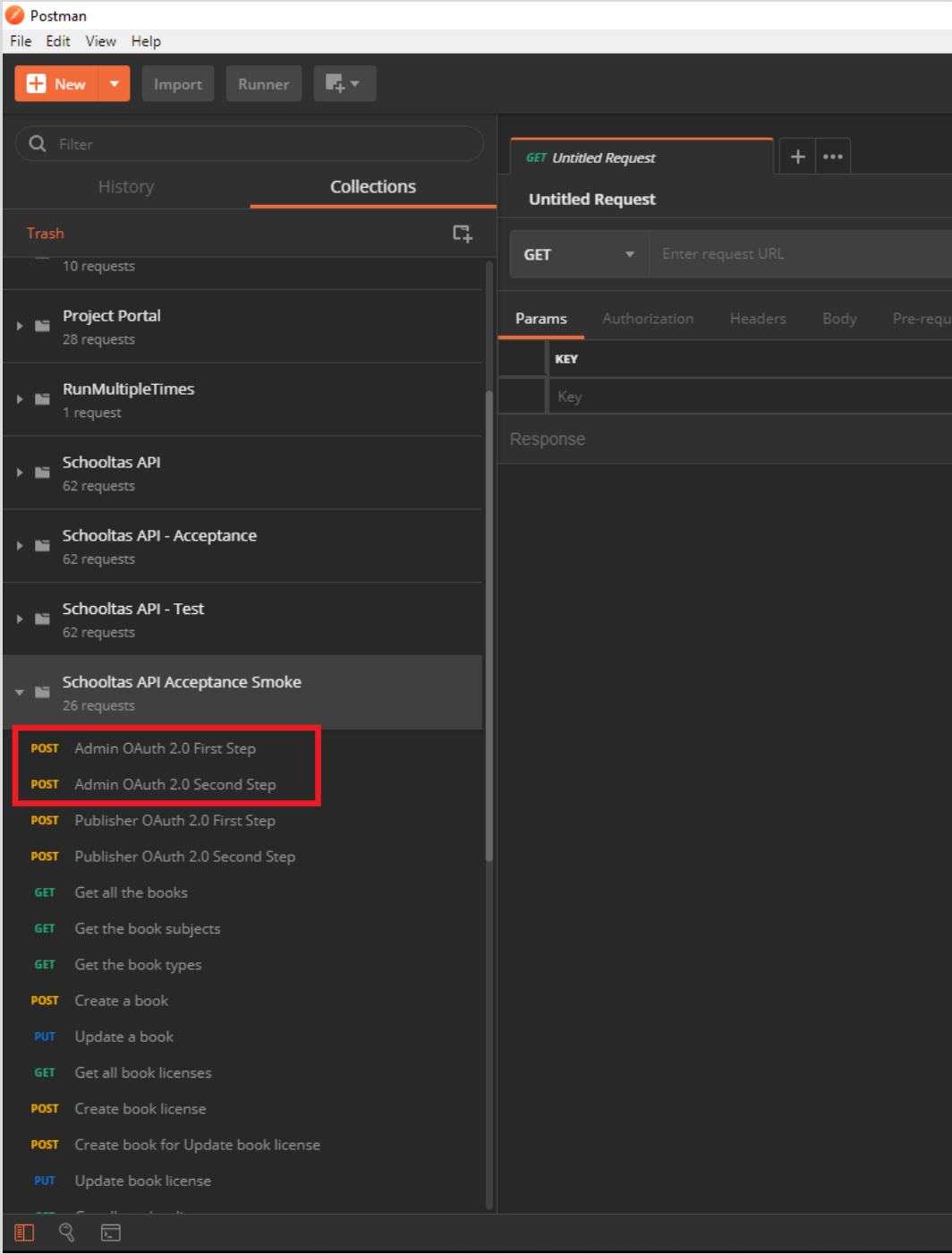


The next piece of code sets an environment variable called **admin\_oauth\_code**. The value of the **admin\_oauth\_code** is taken from the actual header response. When passing the **Location** parameter to the **getResponseHeader** function, a link containing the code and the state will be returned. In order to obtain the code, we split that string. The code itself is passed as value to the environment variable.



Once these two requests passed, we have the access token for the admin ("school" as we discussed earlier) in the **admin\_oauth\_code** variable inside our selected environment.

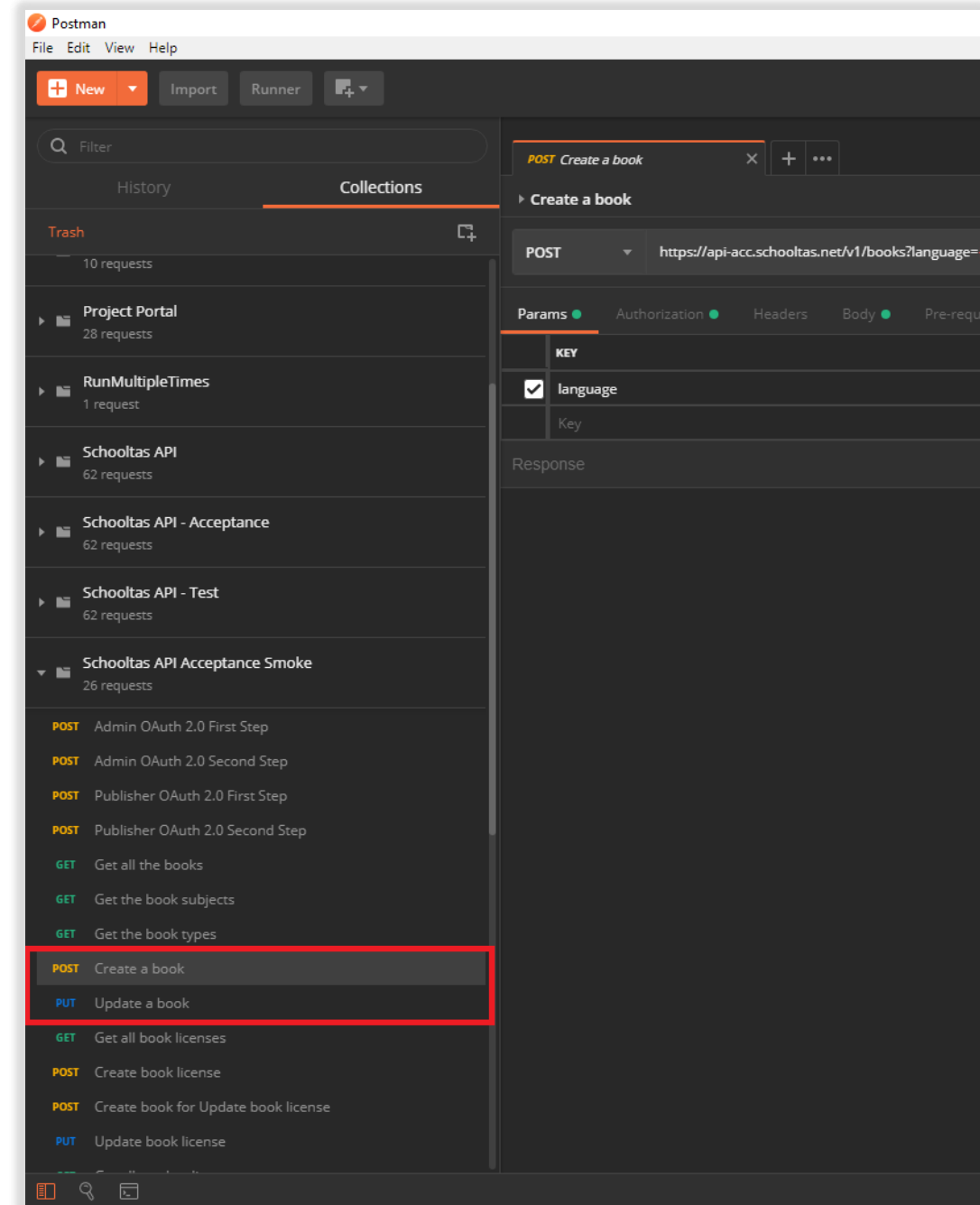
For the Publisher is the same approach. Only the account differs and the **OAuth Code**. With these two access tokens, all the following requests can be ran.



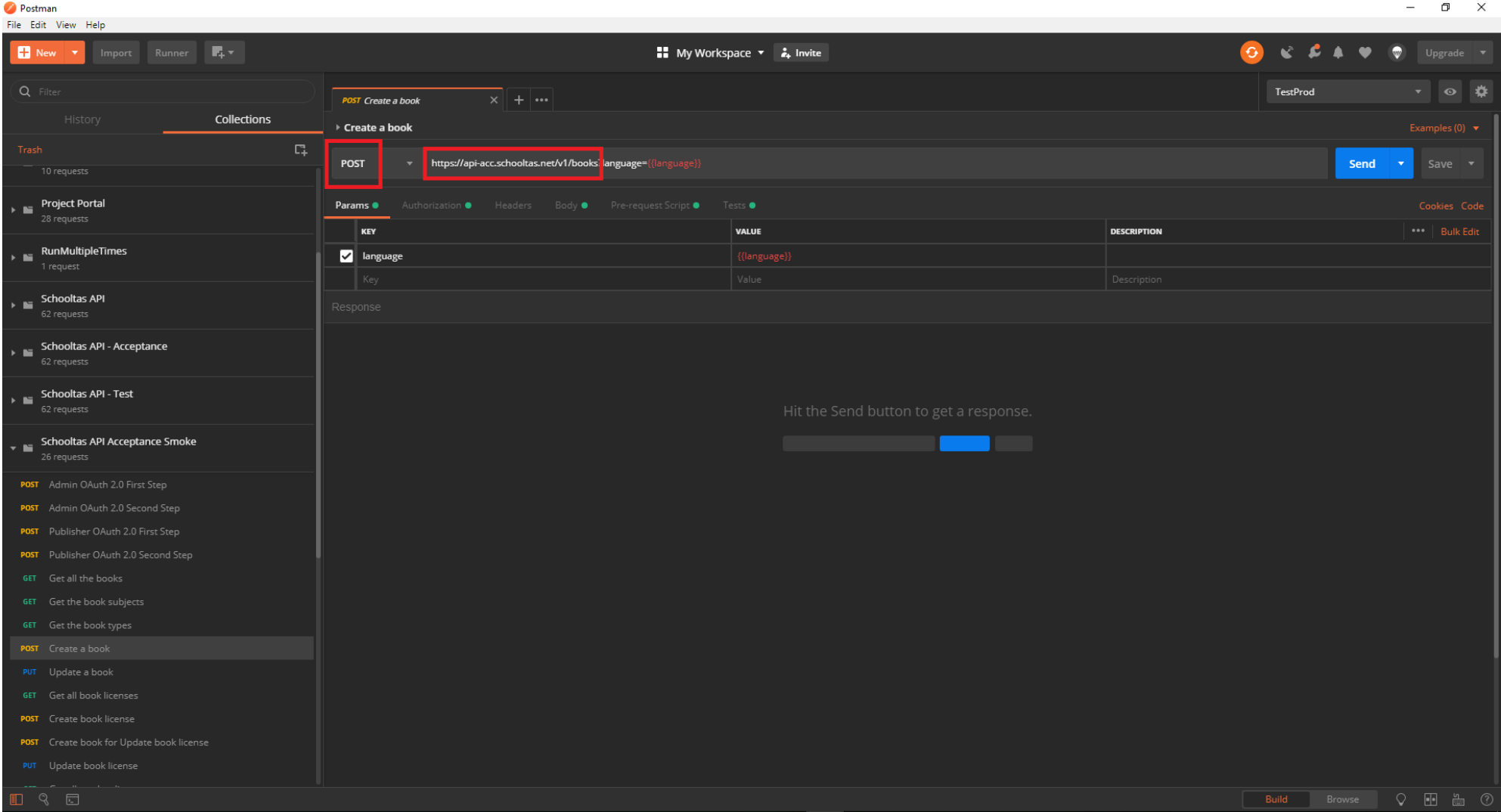
In order to better understand how the requests are working, I will take the **Create a book** and **Update a book** request and explain them.

Both of them are made by a publisher, so I used the publisher access token for them.

Let's see the **Create a book** request. How the setup for this one looks like.

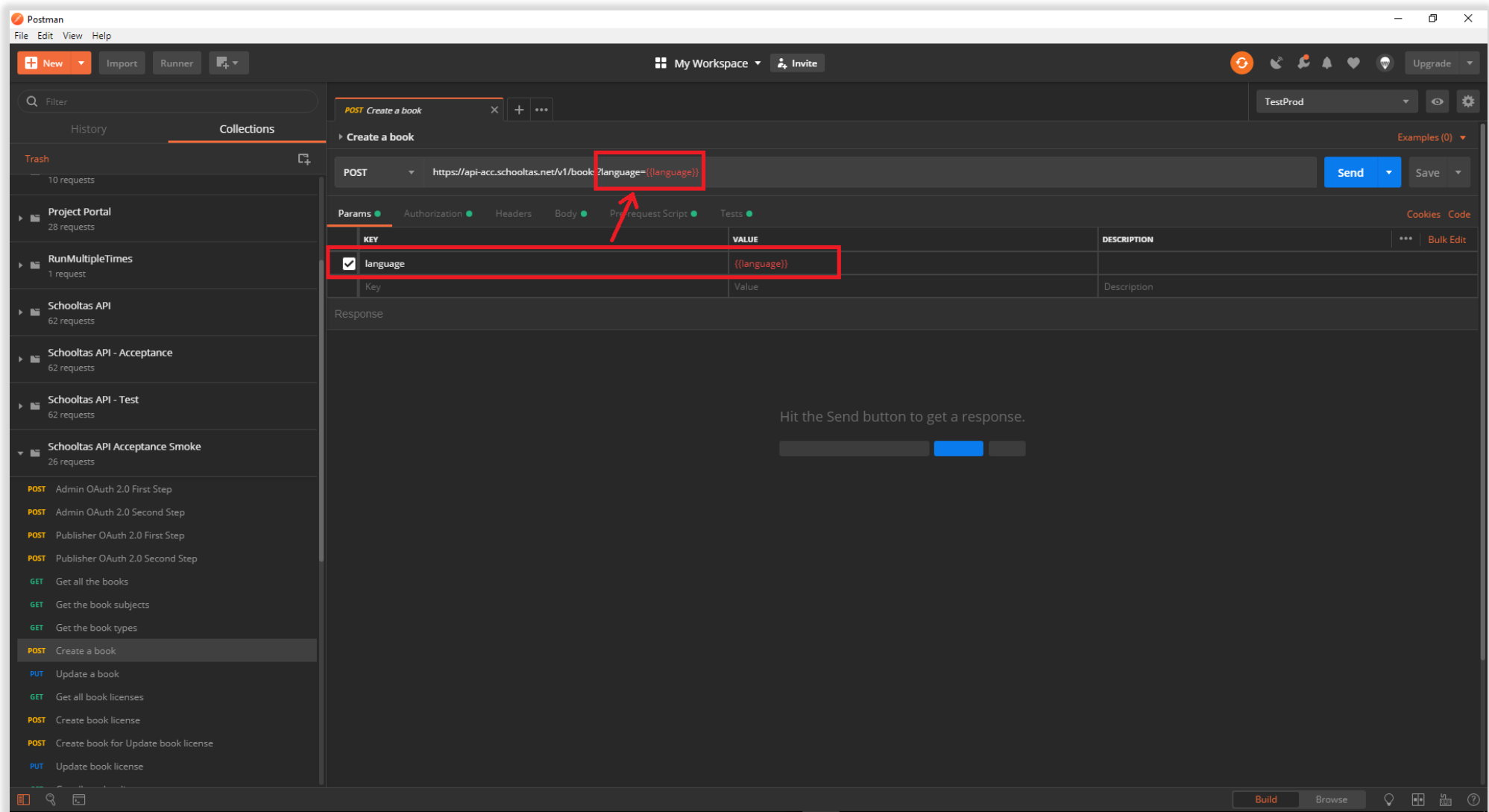


This request is a POST request, made to the **books** endpoint. This can be seen in the URL of the request. For this request we make a call to the backend with some parameters, and a body, in order to create a book in the database.

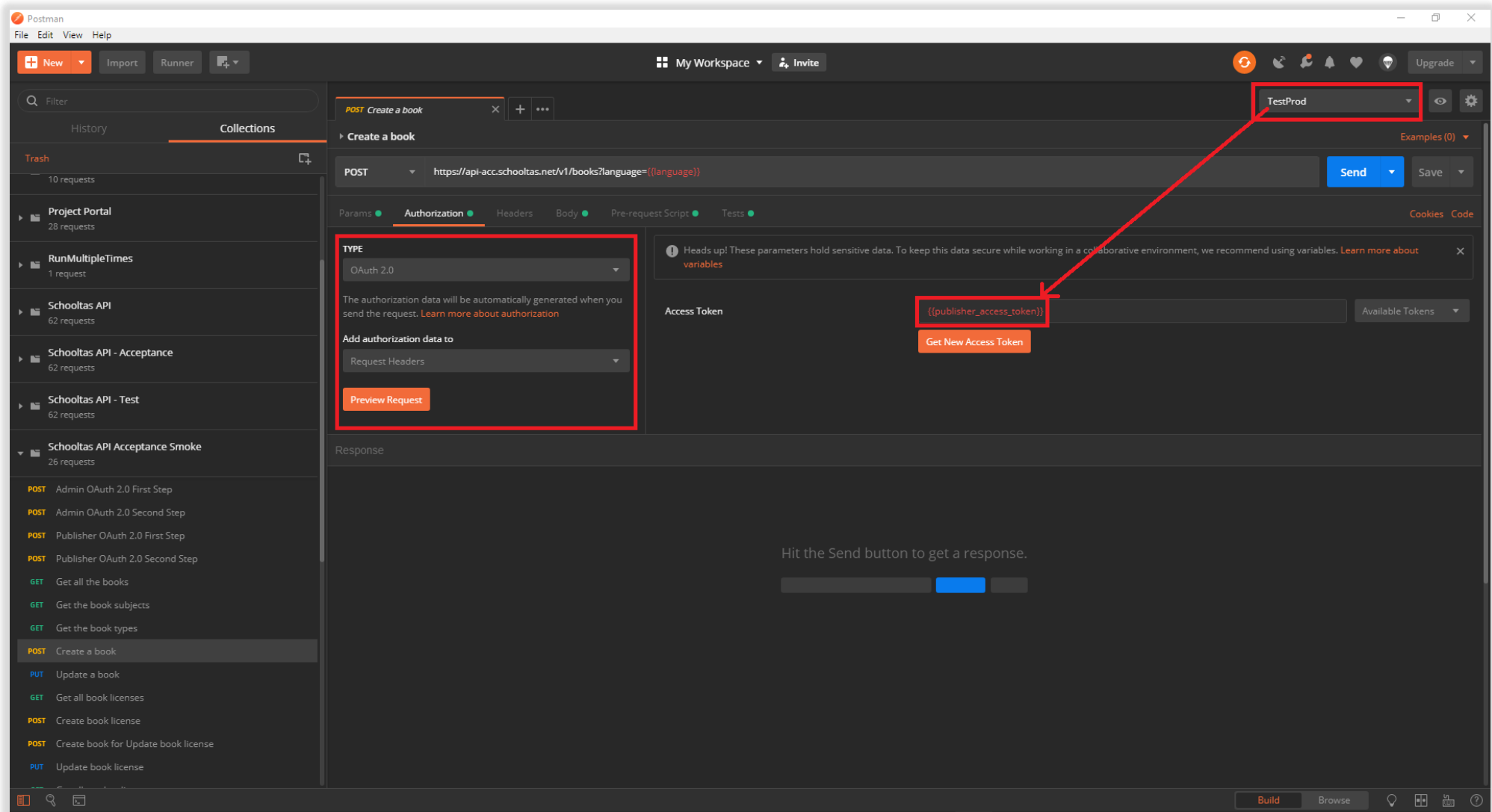




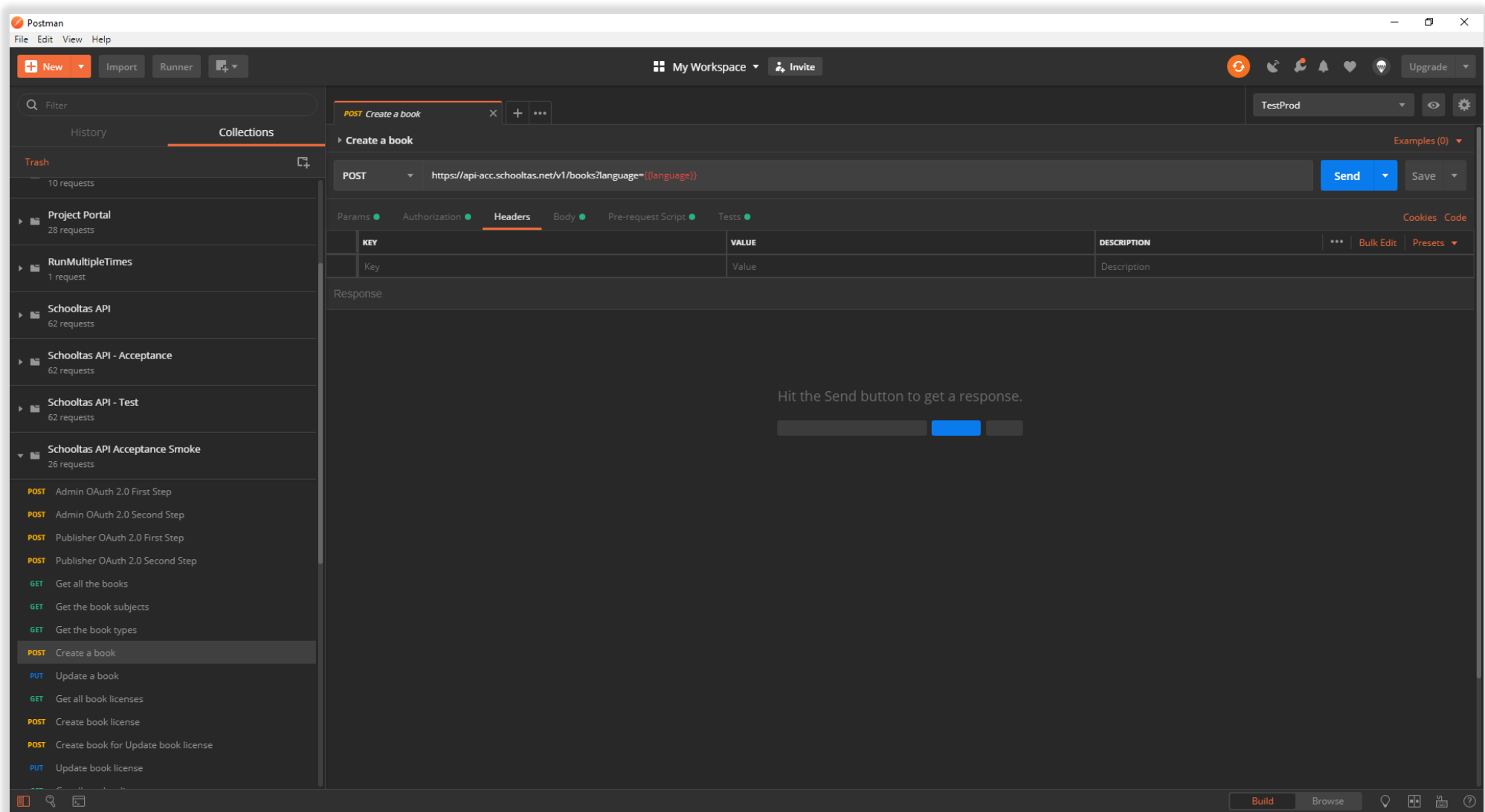
As parameters, this request requires only the language. In the entire URL we can see the parameters and their values after the “?”. They can also be seen in the Params tab.



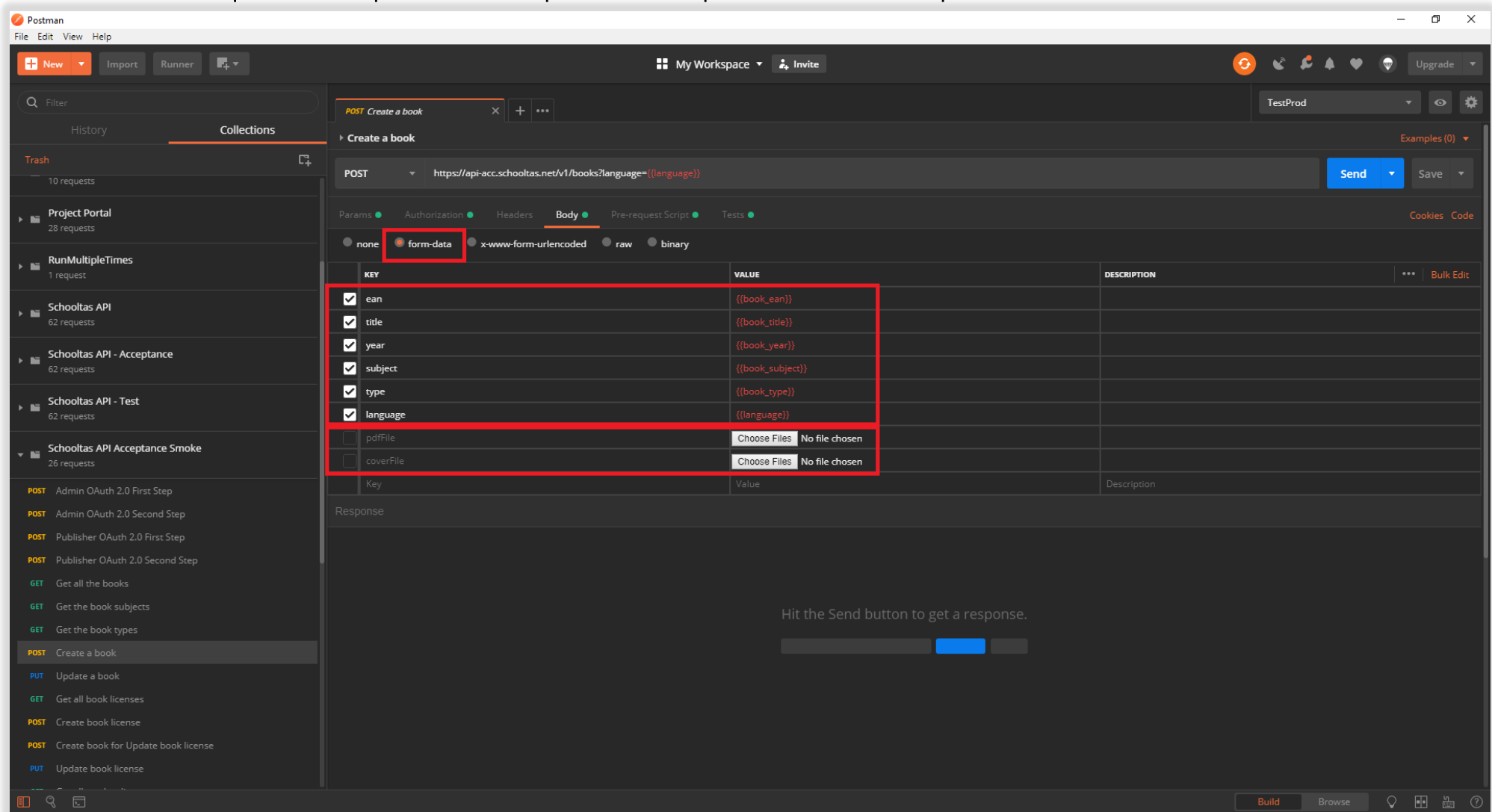
In the Authorization tab, we set the Type of the authorization as OAuth 2.0, and add it to the Request Headers. The Access Token is taken from our environment. This request is ran after the Publisher OAuth 2.0 Steps requests, where we set the **publisher\_access\_token** variable in the environment. Now we can pass the access token for this request, from our environment.



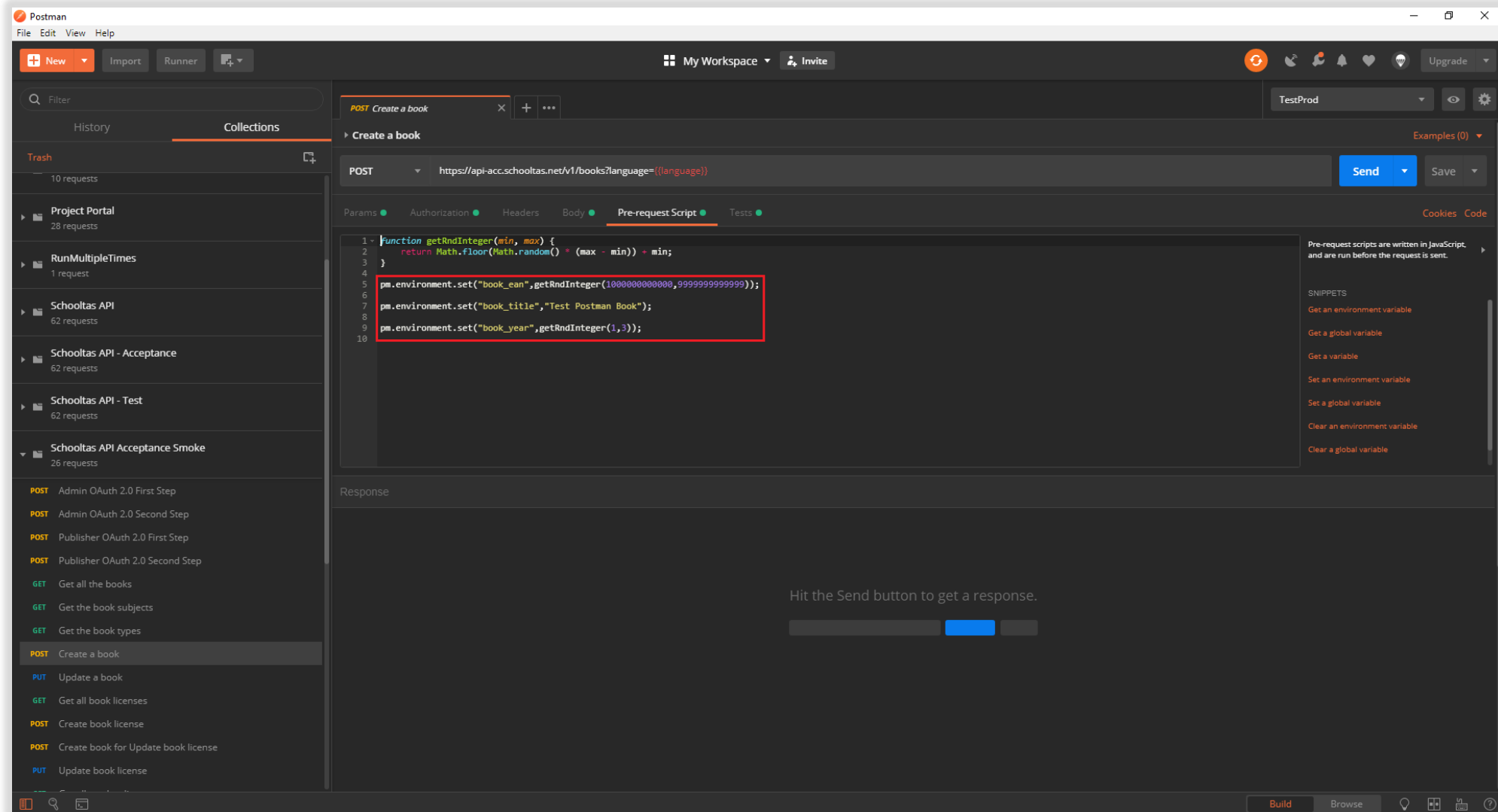
You probably noticed that the Headers tab is empty. Having our body set as form-data we don't need to specify any key and value in the Headers tab. It will automatically be set when we send the request.



In the Body tab, we have the **form-data** checked. In this case our body is filled with keys and values, same as a form. All the keys are required, except pdfFile and coverFile, both of them being unchecked. The values for the required keys are provided from the environment, being set inside the pre-request script or in previous requests.



In the Pre-request Script tab, three of the variables used in body are set. For the **book\_ean** variable, the value is set to an random number given by a function which is declared above it. The same happens for the **book\_year** variable. The **getRndInteger** function returns a random integer between a min and a max value passed as arguments.

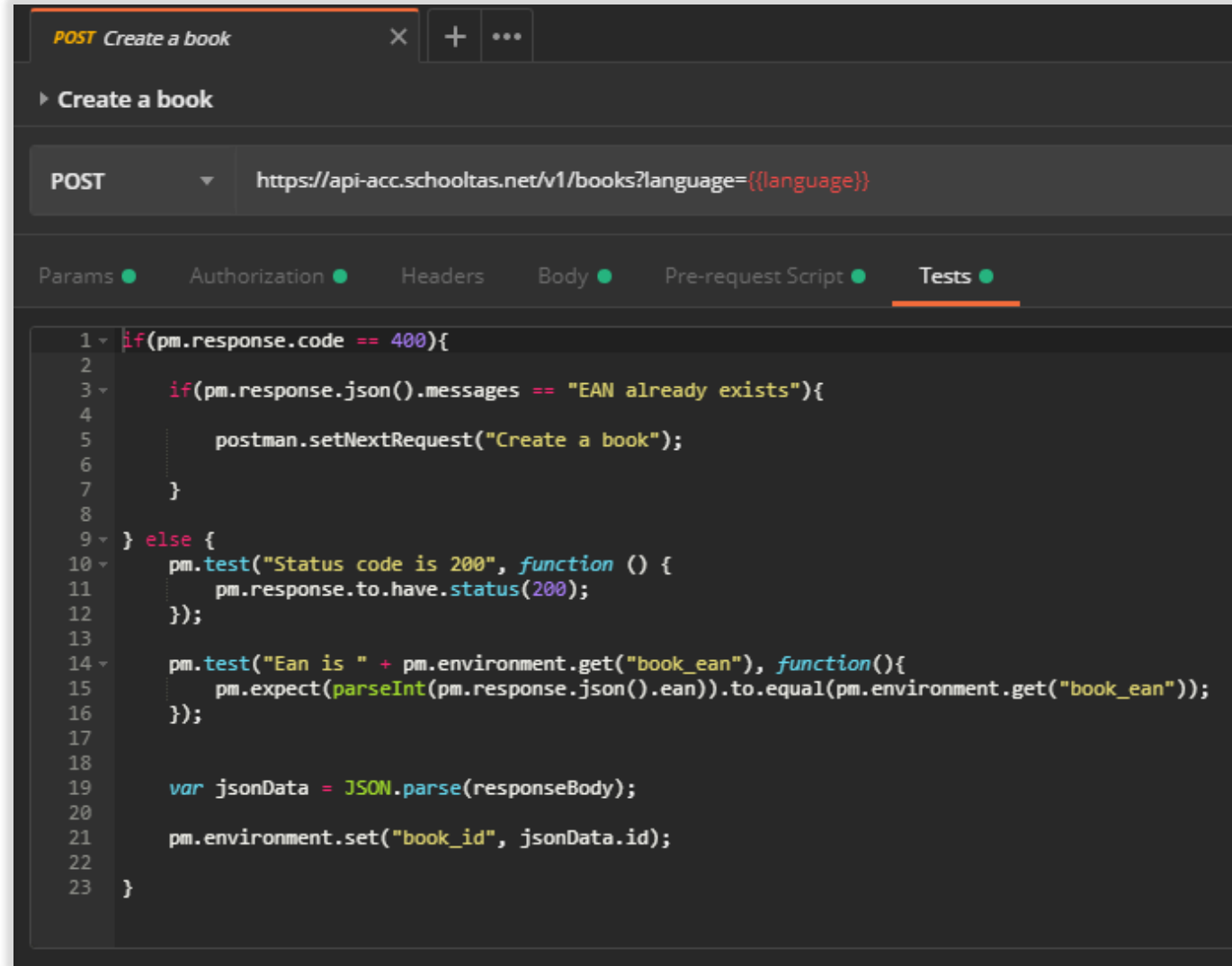


In the Tests tab we have some assertions and an environment variable set.

First, we have a check. If the request fails with 400 response code, this is a small chance to have the **book\_ean** variable randomly set to an already existing one.

If this happens, we tell postman, to recall this request. This is done with **postman.setNextRequest** with the same request name.

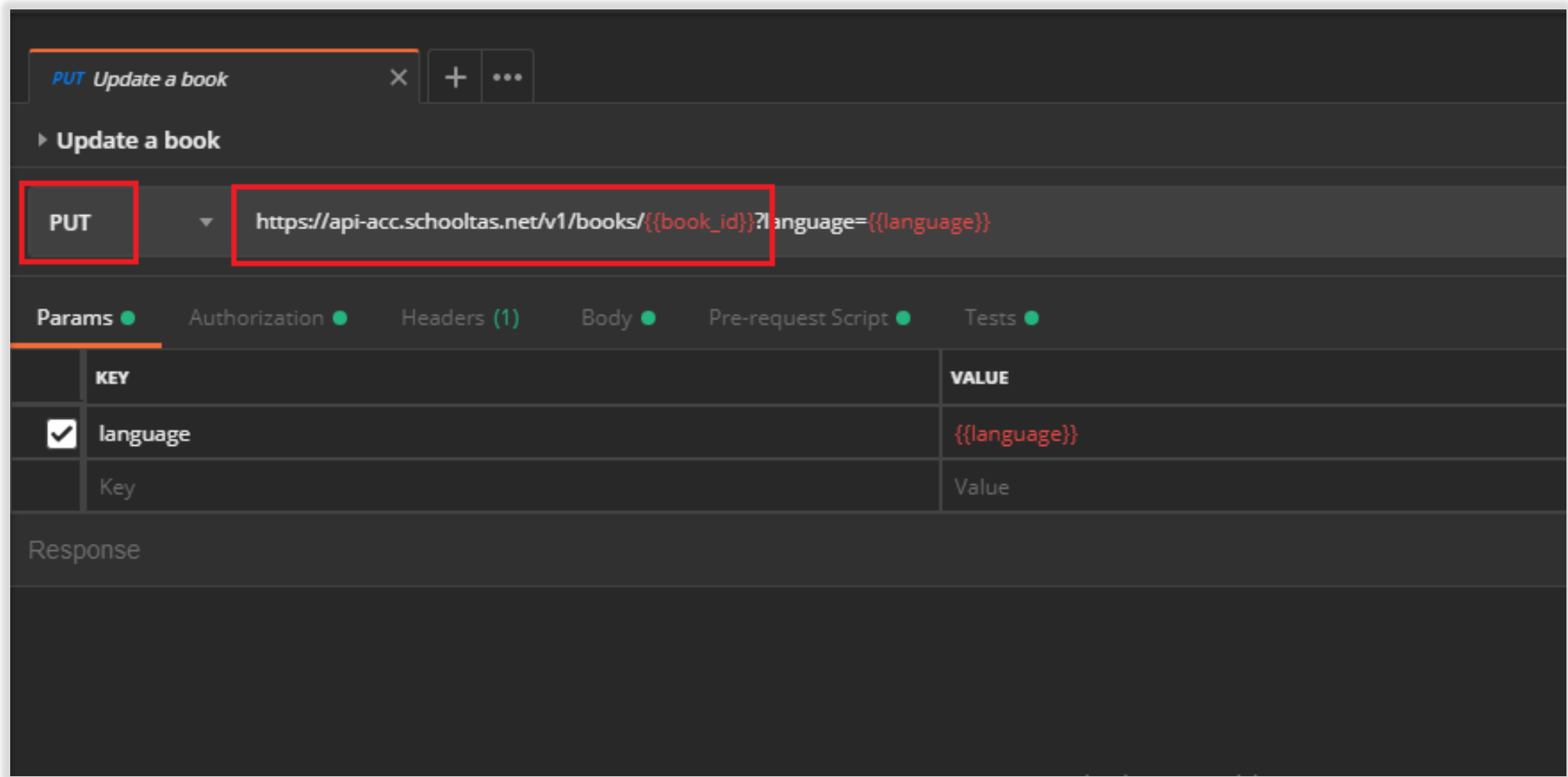
Otherwise, we perform the assertions and we parse the JSON response in order to set the **book\_id** variable with the one received in the response body. This variable will be used in the next request.



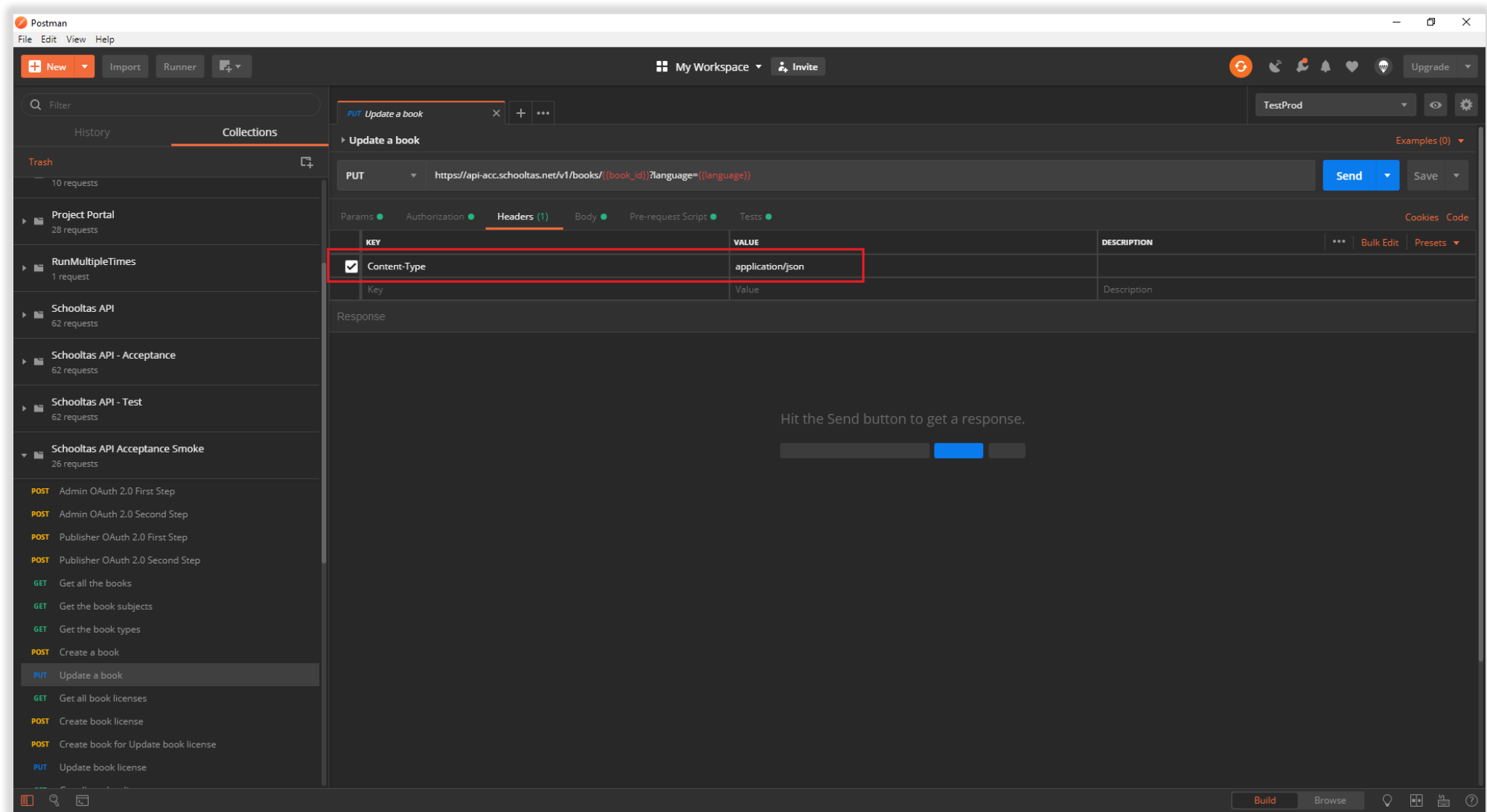
The screenshot shows the Postman interface for a POST request named "Create a book". The URL is `https://api-acc.schooltas.net/v1/books?language={{language}}`. The "Tests" tab is selected, displaying the following JavaScript code for assertions and environment variable management:

```
1 if(pm.response.code == 400){
2
3   if(pm.response.json().messages == "EAN already exists"){
4
5     postman.setNextRequest("Create a book");
6
7   }
8
9 } else {
10  pm.test("Status code is 200", function () {
11    pm.response.to.have.status(200);
12  });
13
14  pm.test("Ean is " + pm.environment.get("book_ean"), function(){
15    pm.expect(parseInt(pm.response.json().ean)).to.equal(pm.environment.get("book_ean"));
16  });
17
18
19  var jsonData = JSON.parse(responseBody);
20
21  pm.environment.set("book_id", jsonData.id);
22
23 }
```

For the **Update a book** request, we need to pass the id of the book, that we want to update in the URL. This id was set in the previous request, **Create a book**. The **Language** parameter is the same as in previous request. Also, the Authorization tab is the same. This time, the used request method is PUT.

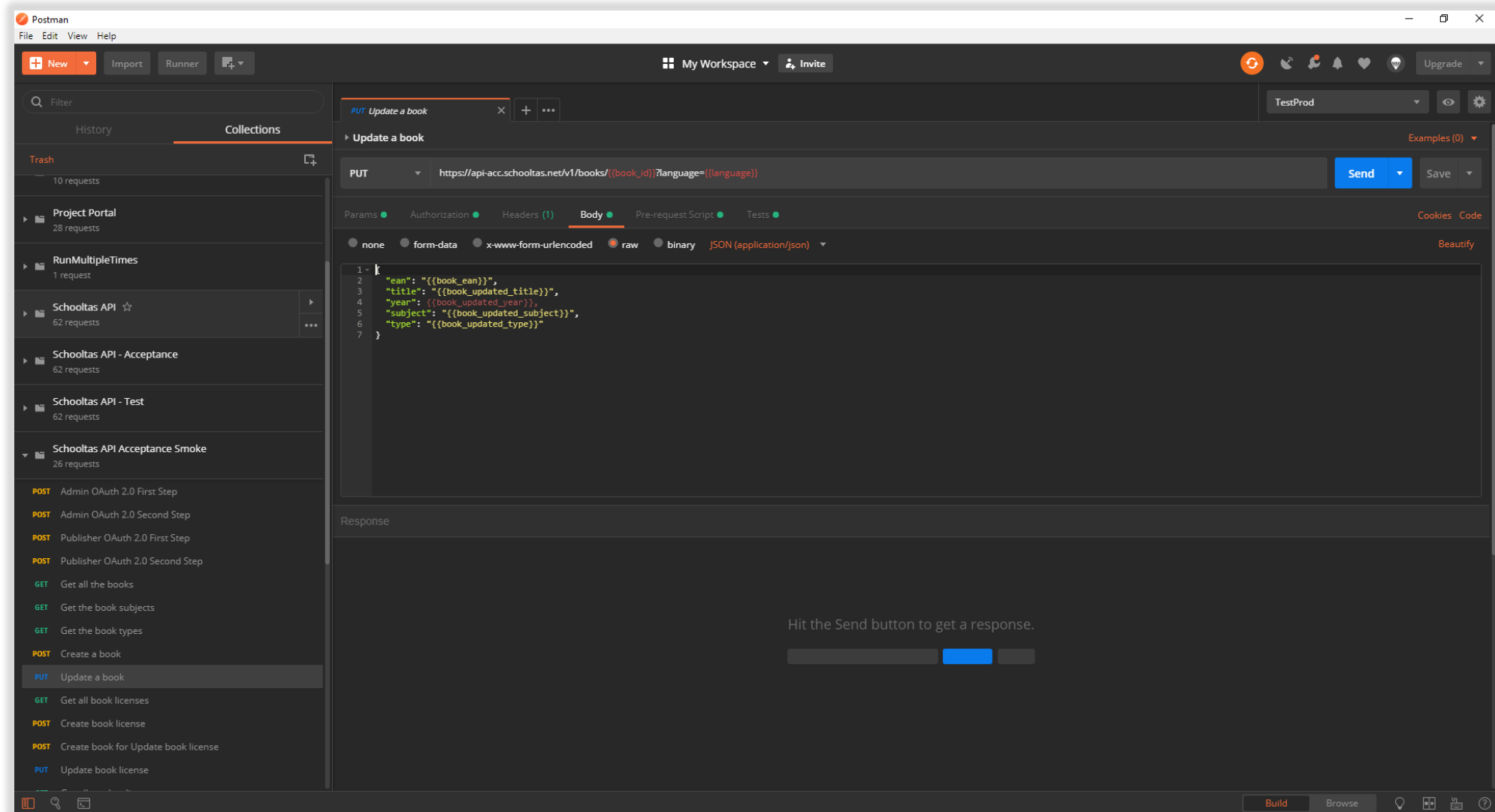


For this request, we have a key and a value set for the Headers tab. The *key* is **Content-Type**, and the *value* is **application/json**. In this case, we specify that the Body of our request is a JSON type.

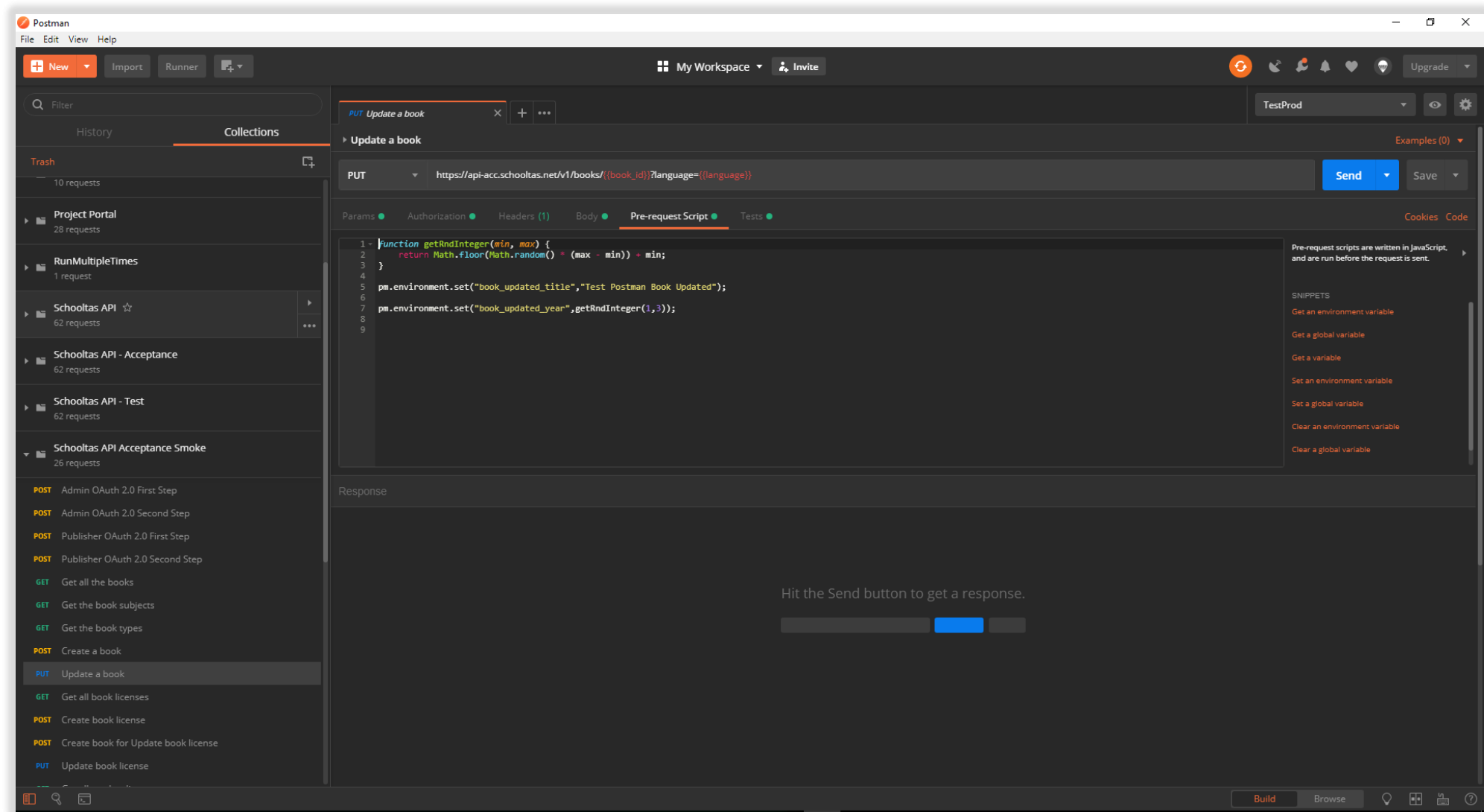




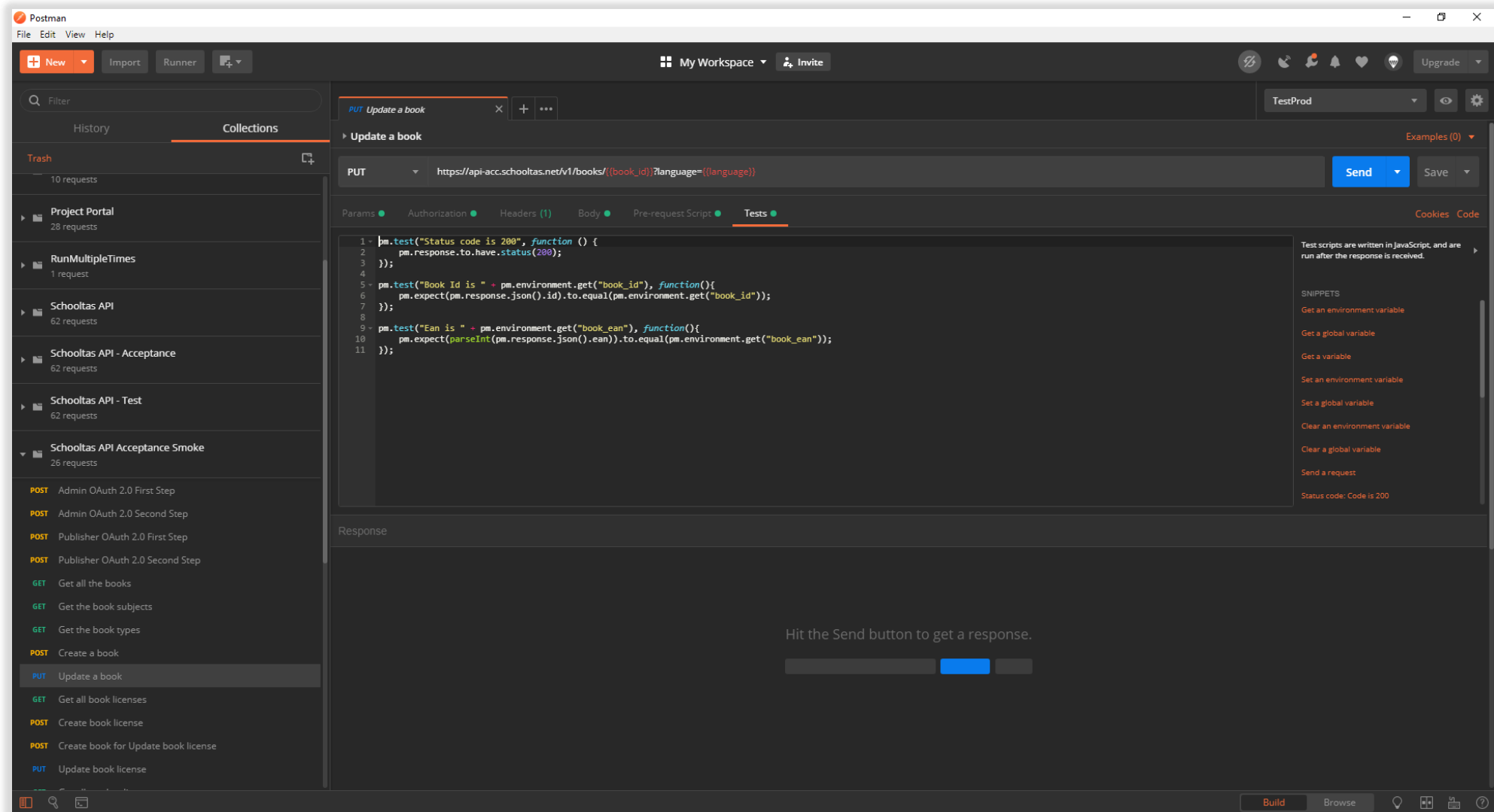
The content of the body being a json, inside the Body tab, the raw option is checked, and the type is set to JSON. The same as in the previous request, the variables have been set in the previous requests or are set in the pre-request script.



Inside Pre-request Script tab, two of the variables used in this call are set here. The function used in this script, is the same as the one used in the **Create a book** request.



In the Tests tab, for this request we created only assertions. In order to see if the updated book is the right one, we are checking the **book id** inside the response of this call. The **pm.expect(A).to.equal(B)** verify if the **A** equals **B**, if so, then it is a pass, otherwise it is a fail. We compare the id inside the json response, with the id saved in the environment variables.



Collection runner, why are some settings required when you use it ?

First of all, the first setting is the **Environment** field. This is required to be set to the environment that you created, in order to save the variables inside it.

In order to save the variables inside the environment, we need to make sure that the **Keep variable values** field is *checked*.

Having the **Keep variable values** field checked, helps us keeping the variables set in the environment and avoiding their deletion.

We need to save the variables and their values inside the environment in order to perform the two tests inside the *second collection* manually.

We could perform them without the environment, but we should set all the values for the variables used in that requests. It is easier to have all the values in place, and just press the Send button.

## Why are the two requests separated in another collection ?

The first request, **Create a new reader**, requires a pdf file in the Body. This pdf file can only be assigned manually each time you want to run this request. If the user clicks on another request, this field gets cleared.

Postman does not persist the value picked for the files. Those files cannot be assigned in the collection runner. This means that we cannot perform the automated request because of the mandatory pdf file. This issue was investigated and was reported to the postman team.

The values of the variables for the second request, **Update a reader**, depend on the **Create a new reader** request. This means that the second request needs to be ran after the first request. Due to its dependencies, the request needs to be ran manually.

The Production collections have the same approach. The only difference are the values of the variables, and the endpoints. There are the same tests performed on both collections.