ADVANCED PROGAMMING TECHNIQUES

FIZZ BUZZ WEB

UNIVERSIDAD DE SAN BUENAVENTURA CALI

Juan Pablo Silvestre González

jpsilvestreg@correo.usbcali.edu.co

30000099127

22/4/2024

CHALLENGE DESCRIPTION

Given the Fizz Buzz problem from past activities, the main goal is to create an API that provides the basic CRUD interface. Using Flask as lite server and Postman as a test tool, we will satisfy all requirements specified.

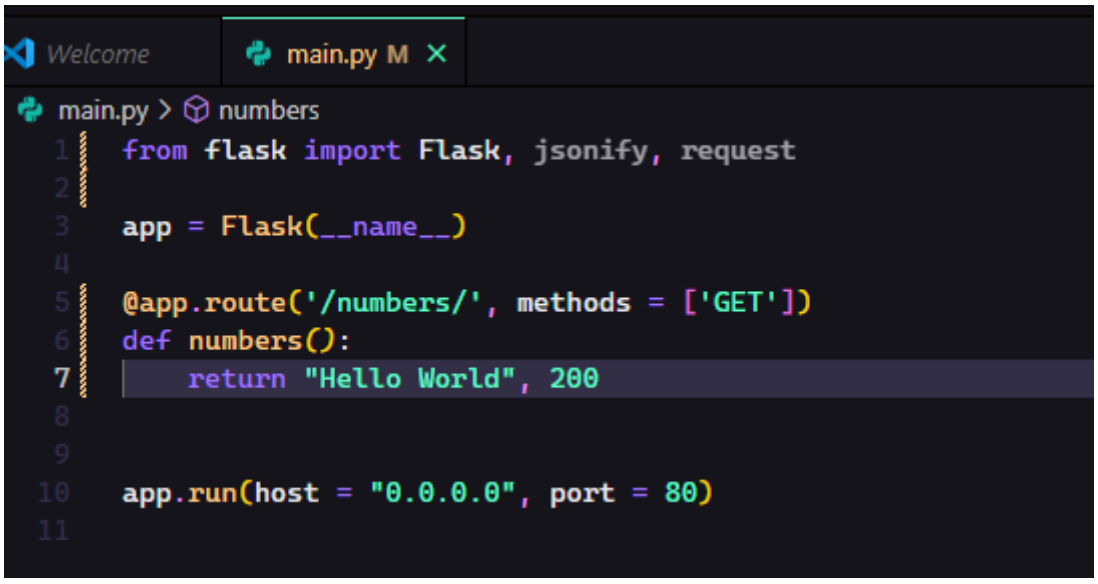
RESTRICTIONS

This report lies its structure on the Test-Driven-Development (TDD) methodology, proving and documenting all the process in each phase (Red, Green and Blue) and what it took to achieve the final product. Also, we will be using the Dependency Inversion Principle (DIP) that guides how we should structure our code to make it more flexible and maintainable.

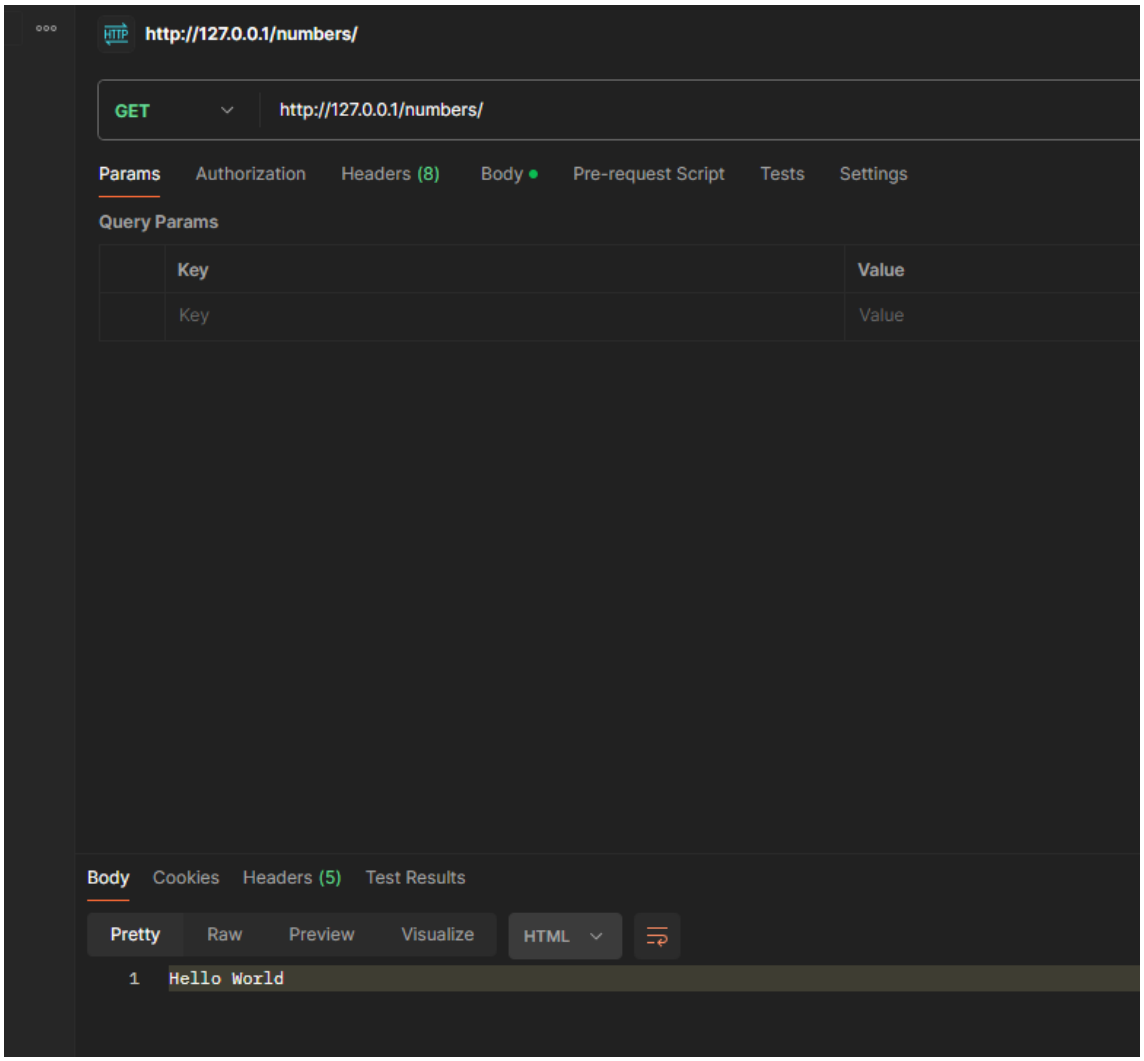
STRATEGY

In response to each requirement pointed in the challenge, we start off with our main Flask application, then I'm going to define a Postman script for each section in this challenge; these scripts will confirm whether the retrieved data matches the expected response or if it correctly handles any errors. Then we will develop our code's structure, which is going to rely on the DIP; after our structure is defined, we can create our database and the logic that will bridge our database with the requests made from Postman.

RED PHASE

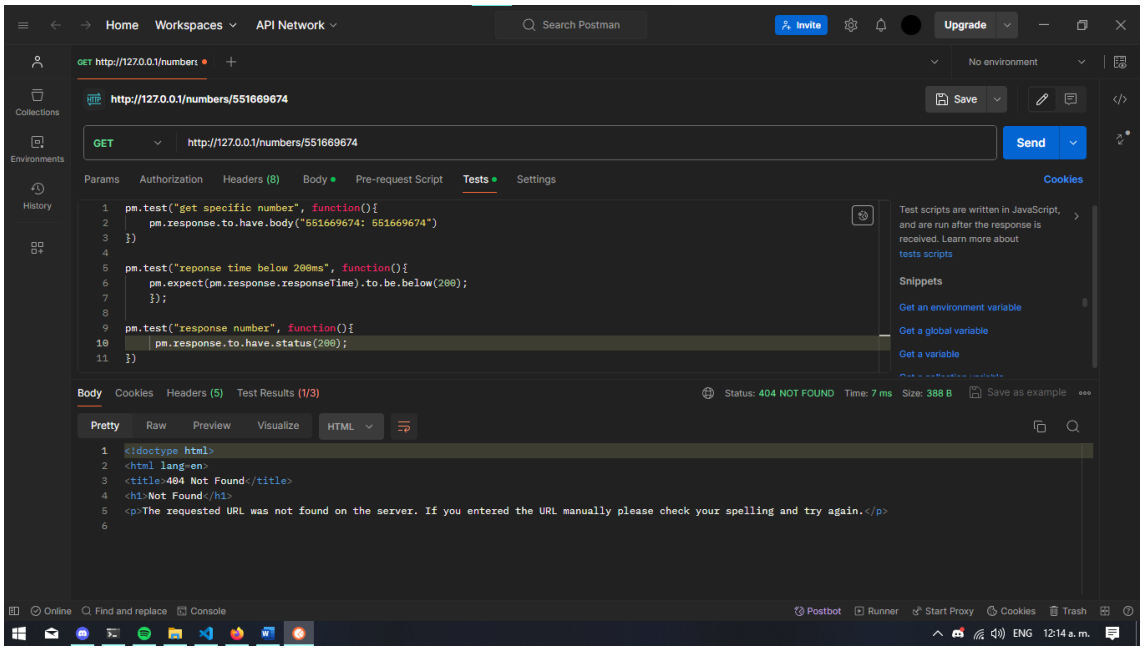


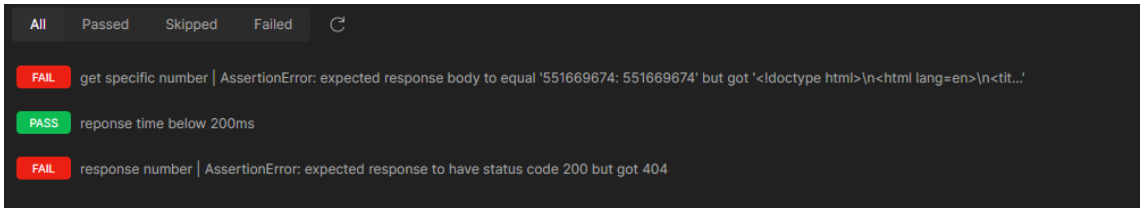
We first create our main application where we'll get the response from, now, all it returns is a "Hello World", with the status 200.



Now, with a basic setup we can start developing the scripts to validate each section provided in the challenge.

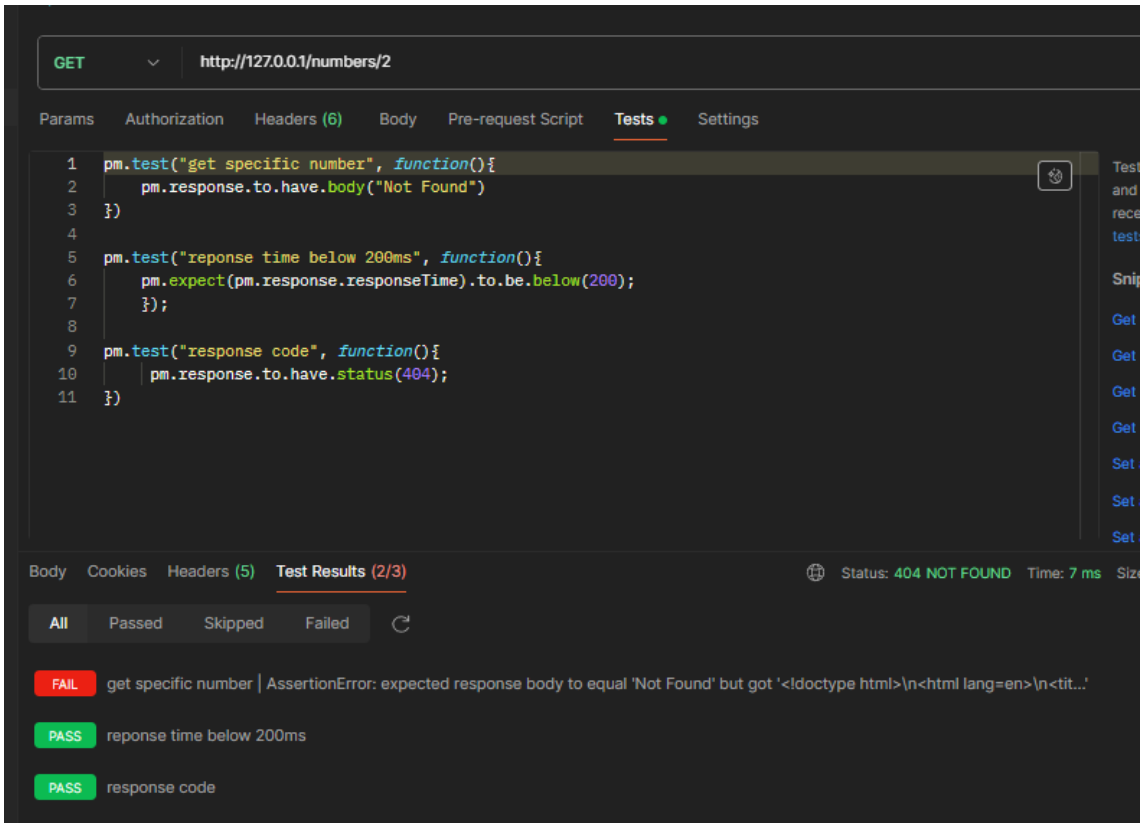
- We start off with 100 numbers stored in the database, those will come from previous results obtained in pasts projects related to FizzBuzz. We can request the API to retrieve one of those numbers and see what we get:





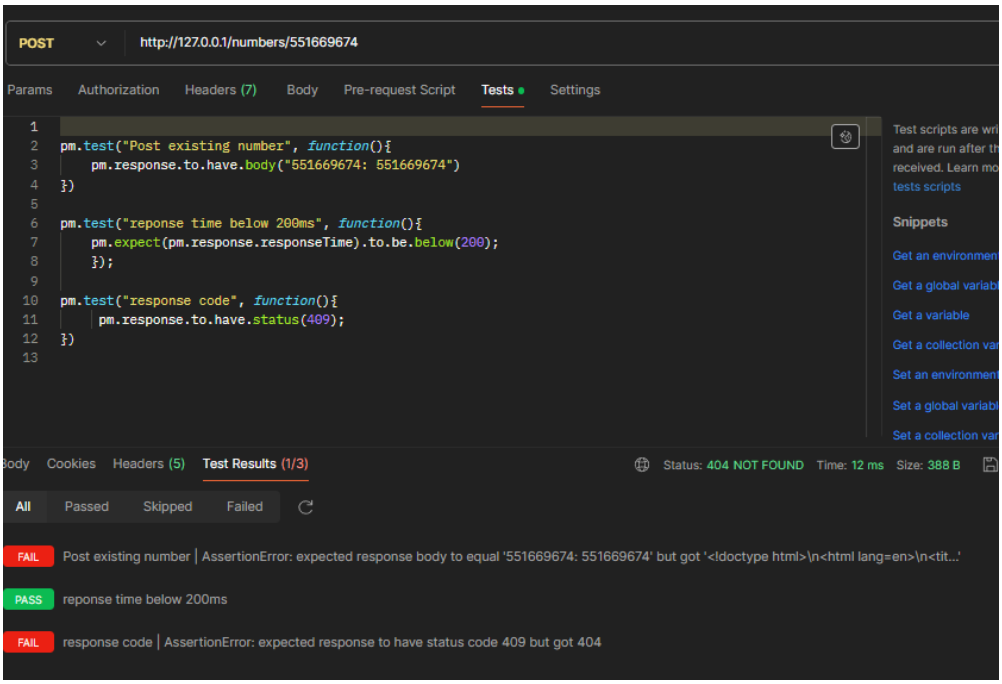
The test result points out the fact that we didn't get the expected response, for the main reason that we haven't created the database and the logic for that request.

- The next section is to retrieve the record of a number that is not stored in the database, for instance we'll try retrieving the number 2.



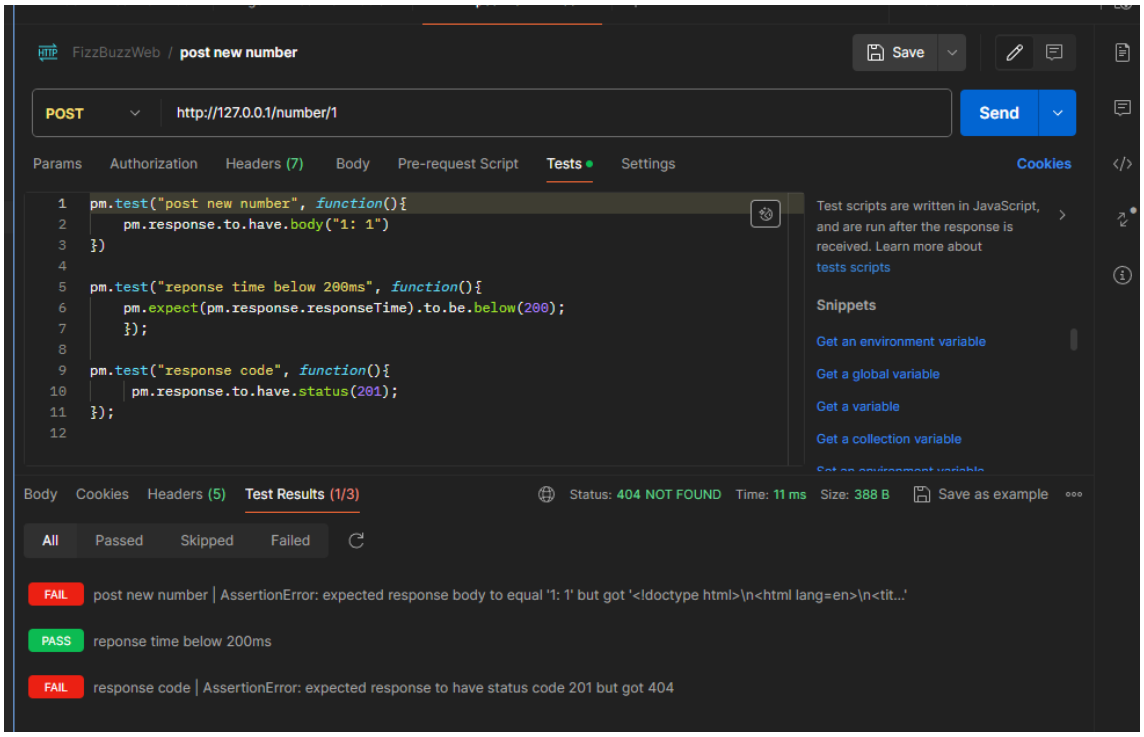
As shown in the screenshot, 2 out of 3 tests passed, but since there's no actual database we are retrieving our requested number from, it will return the 404 anyway.

- The next section tackles inserting the record of a new number into the database; in case the number is already stored, it will return the HTTP code 409. Otherwise, if the number is present but not active, it will set its status back to active and make it available for future requests, besides returning the HTTP code 200. And finally, if the number is not present, it will return the HTTP code 201.



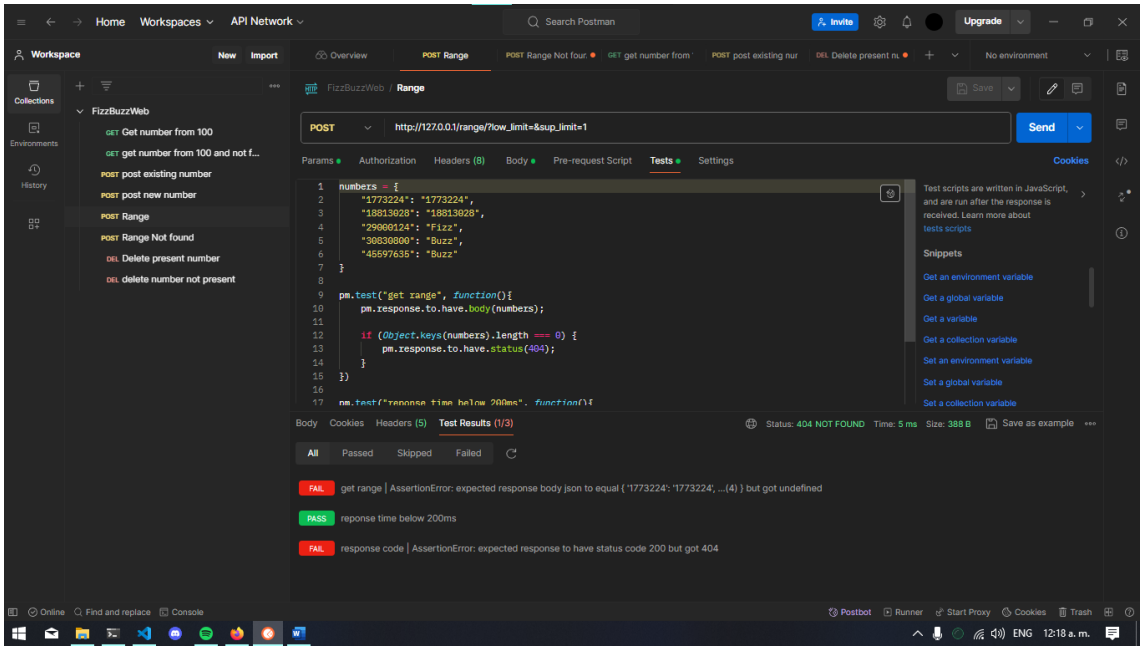
First, we try to insert a number already stored (551669674), which response is 404, again, we're getting the same response since there's no database we are inserting or retrieving our data from.

We will get 404 from now on, until we create our first instance of the database.

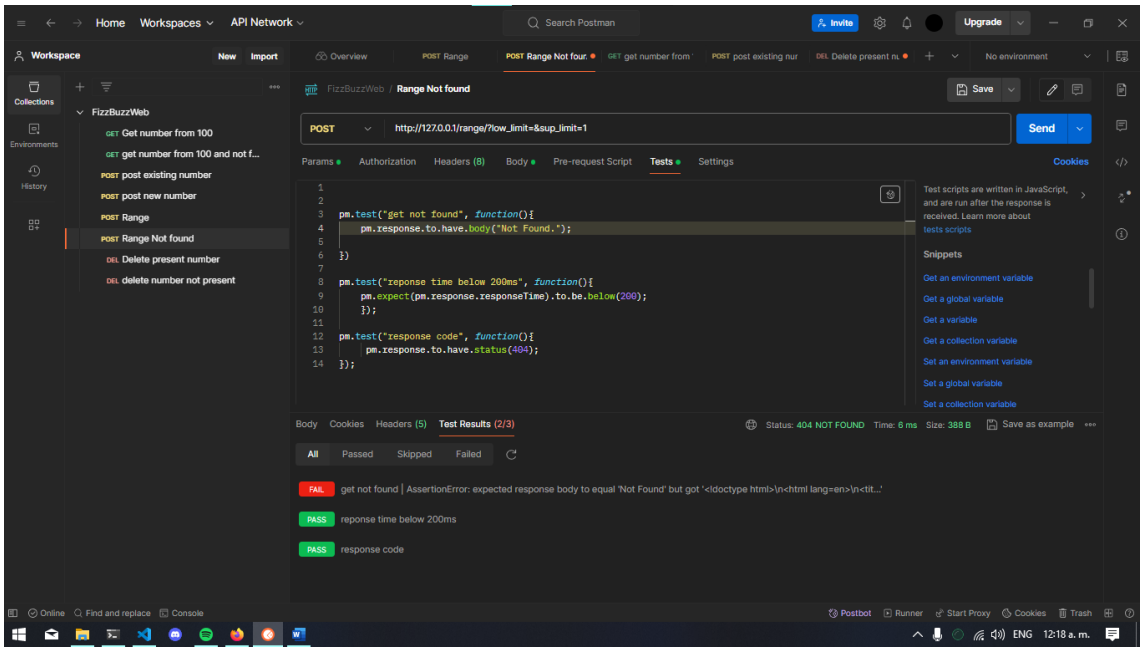


Same result goes for a number that is not present in the database, in this case, we try the number 1.

- The next section consists on retrieving a range of numbers from the database, which is set by sending both inferior and superior limit as parameters; in this test, we set the range to be between 1773224 and 45597635.

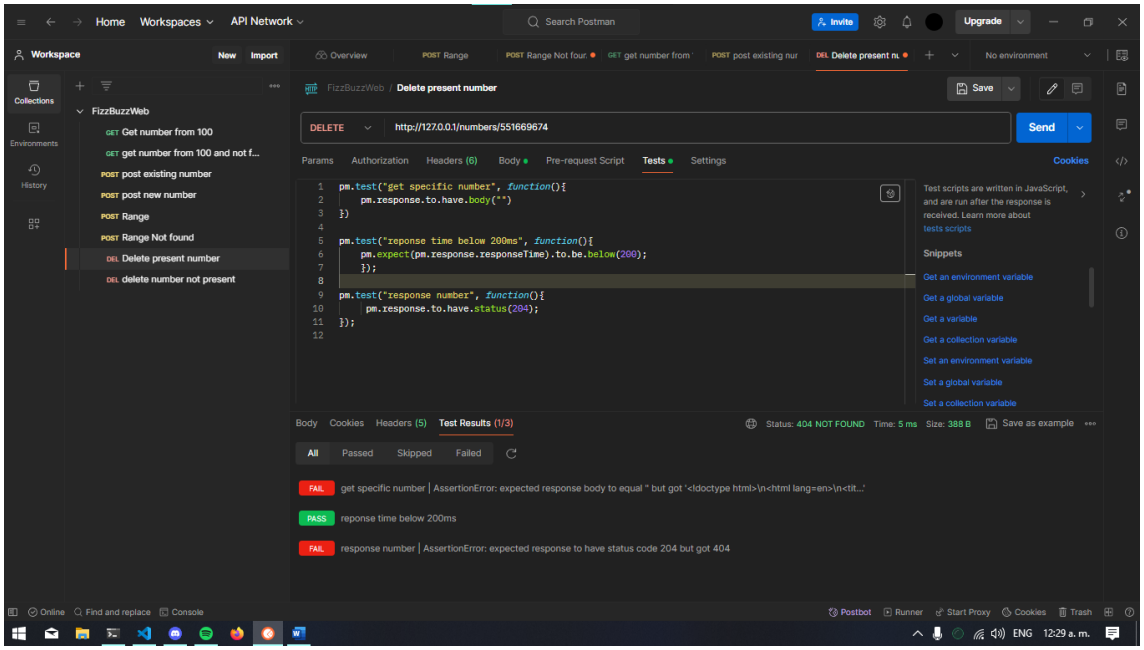


As expected, no response whatsoever, the status code we get 404 since there's not an initial state of the database.



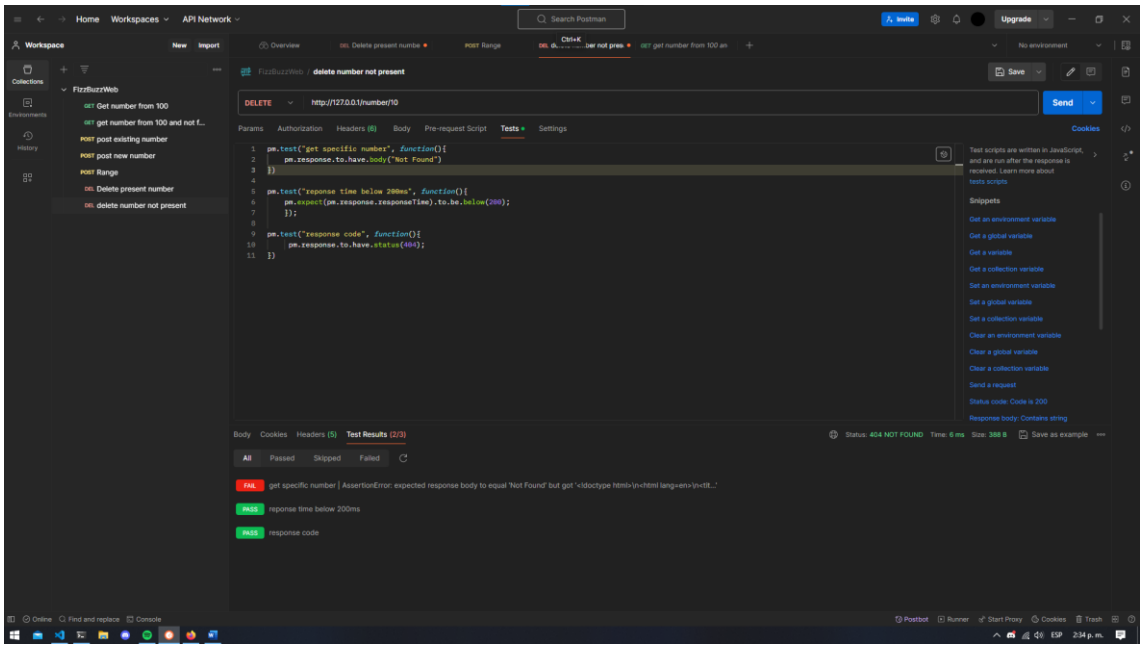
In this test, we pass the response code for HTTP code 404, which makes sense for the reason there's no database to take the range's information from.

- The next section we are assigned to delete an existing number stored in the database, the expected response is the HTTP code 204 if the number is not stored.



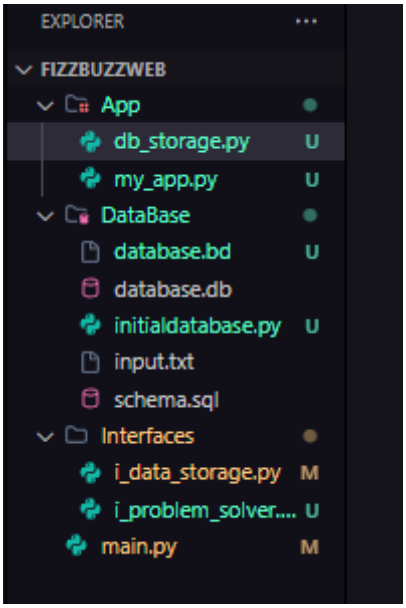
Same problem as before, the HTTP code returned is 404 due to the absence of a database from which the record can be deleted.

Just like the previous section, the last section involves deleting a non-existing number from the database.



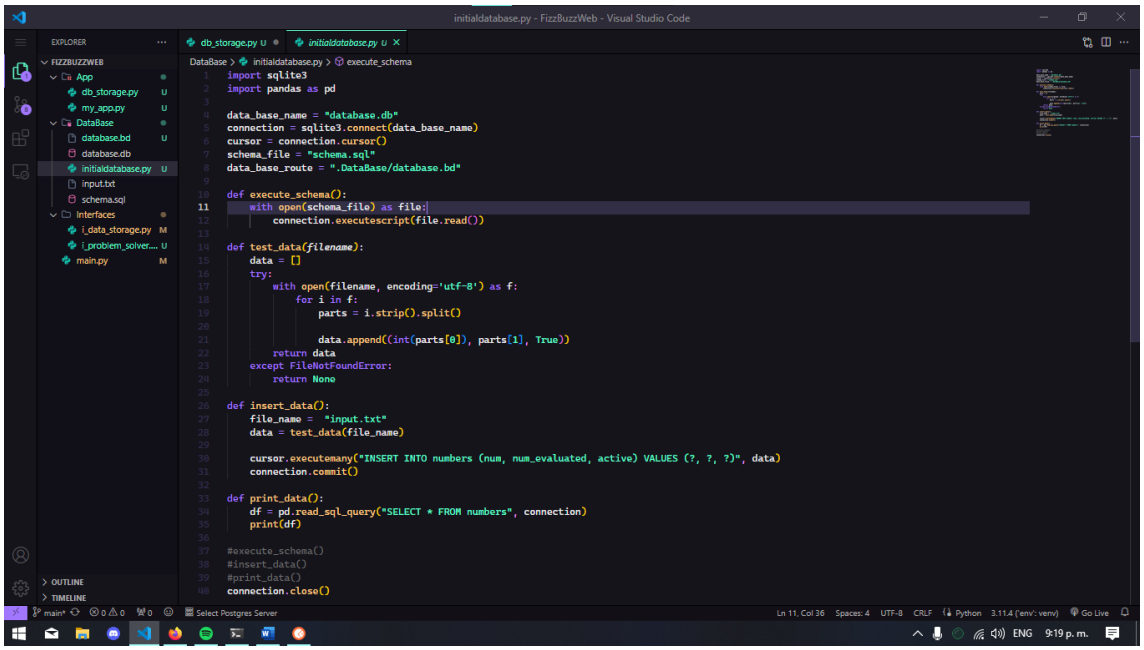
In this section we pass the HTTP code test, but as before, this is a placeholder since the database is not defined.

GREEN PHASE

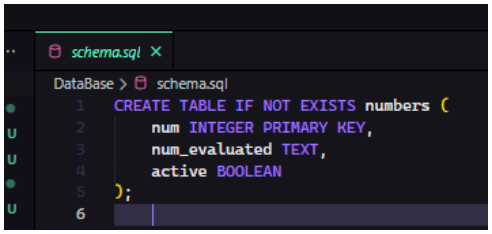


We start off with 3 directories:

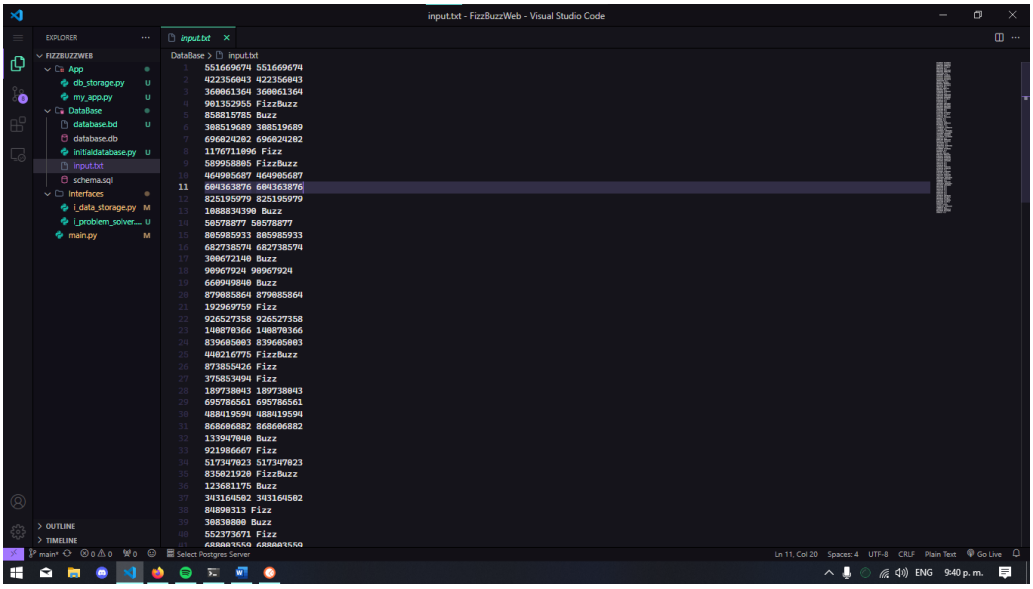
- DataBase: Here, we define the first instance of our data base with the record of the first 100 numbers.



The goal here is, create our database with the table “numbers” that will come from the “schema.sql” file, then we insert the data from the “input.txt” file, that contains the first 100 numbers and their respective result.

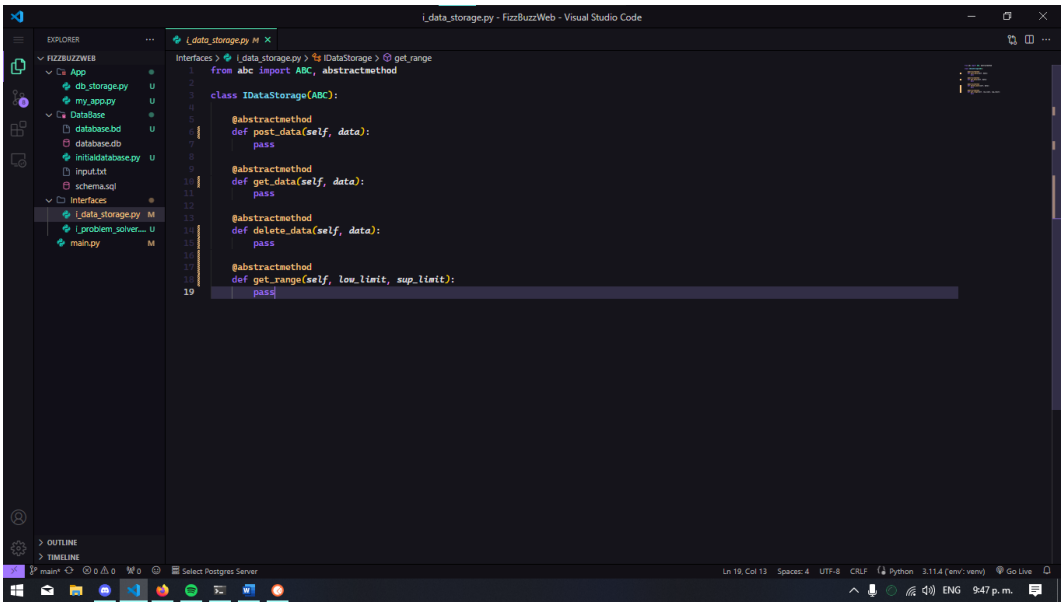


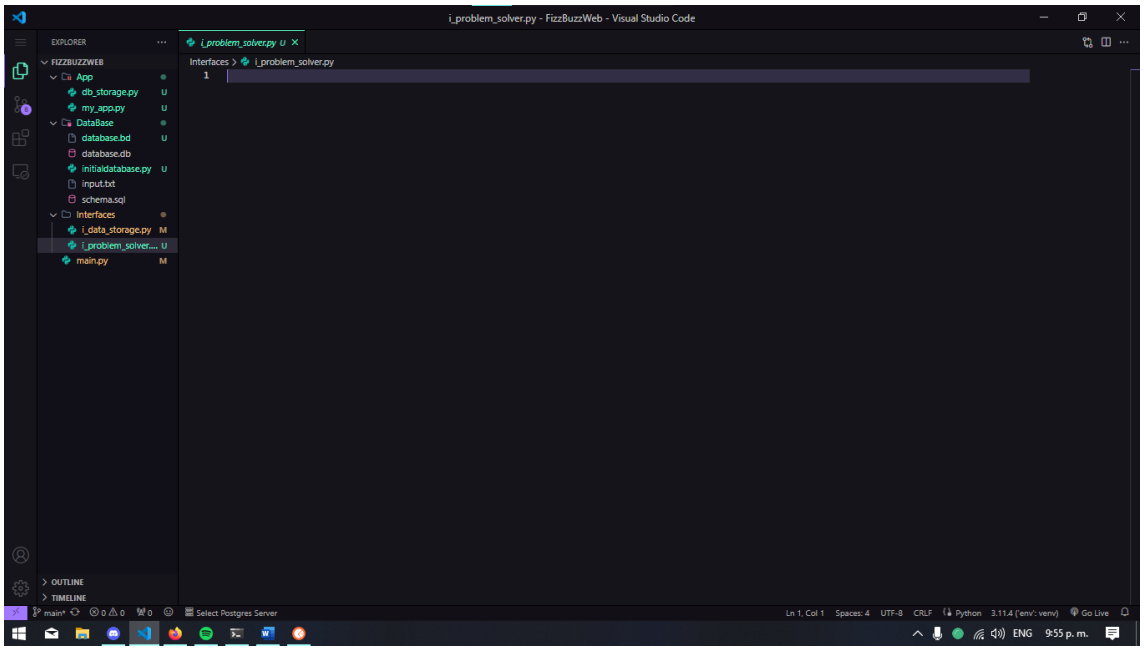
Here's the schema file that creates the table numbers.



Here's the input file where the 100 numbers and their results come from.

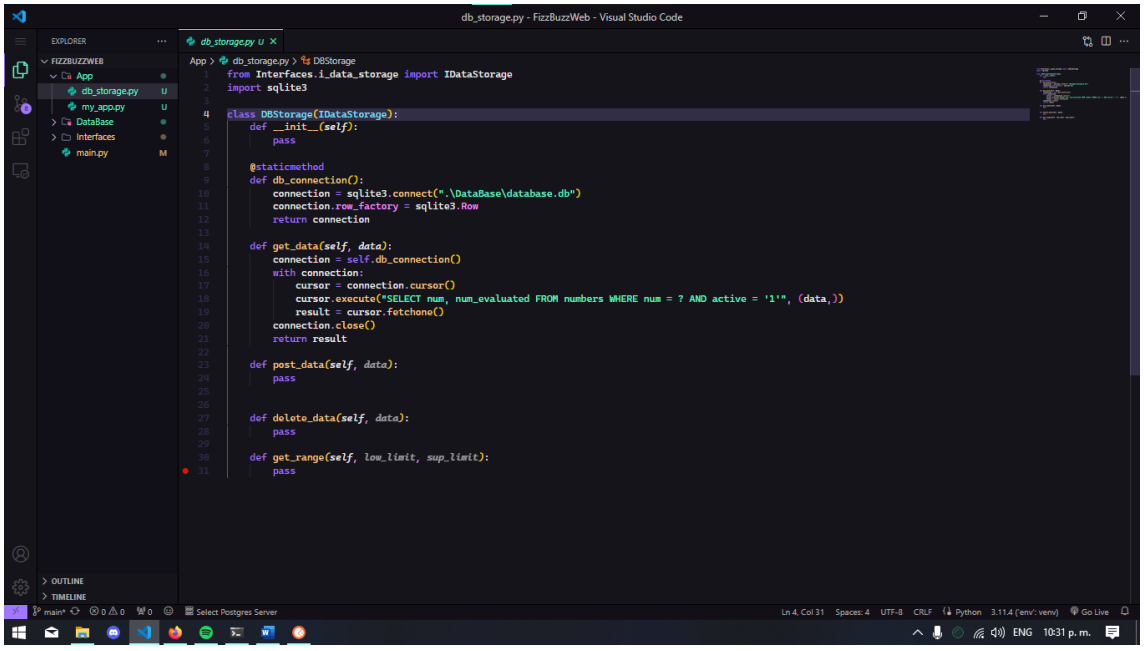
- Interfaces: Here we define the interfaces for the data storage type (i_data_storage) and the problem solver type (i_problem_solver). For the Storage type, we define the methods that will allow us get responses from either a data base or a file in general (if necessary, we will add more methods in the future).



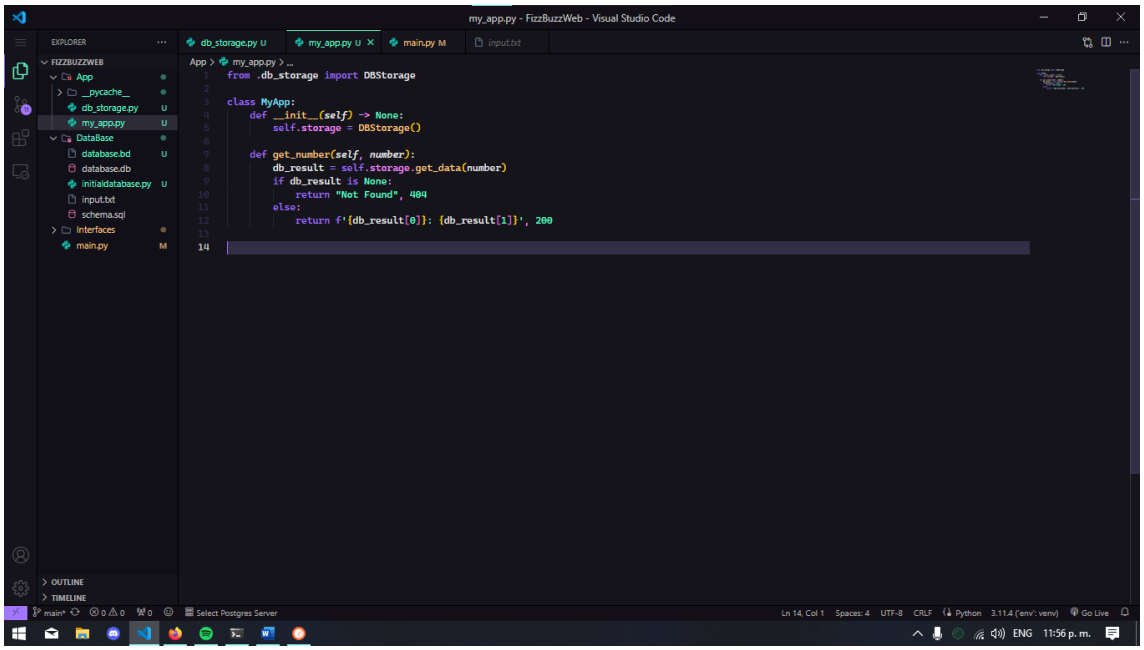


The i_problem_solver interface will remain empty until we have the need of calculating the result of any given number.

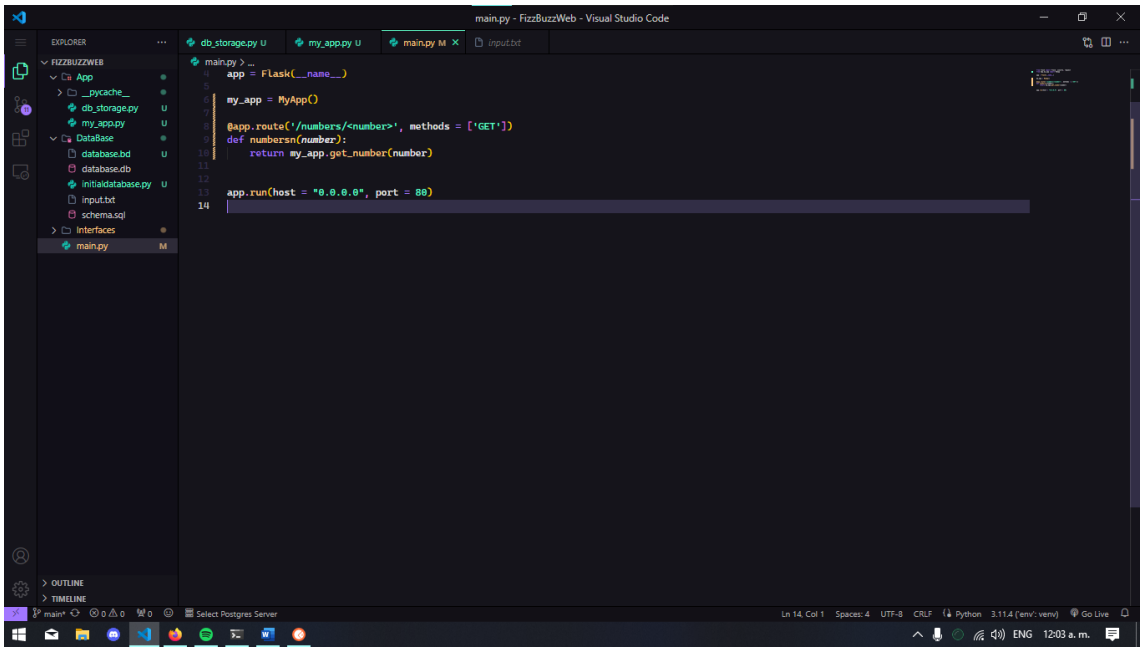
- App: Here we define data base storage type (DBStorage) that implements the interface i_data_storage; where we will make the logic for retrieving and inserting data into the database.



For our first approach, we work on the “get_data” method, because our first Postman test relies on this. We will work on the other methods later on.

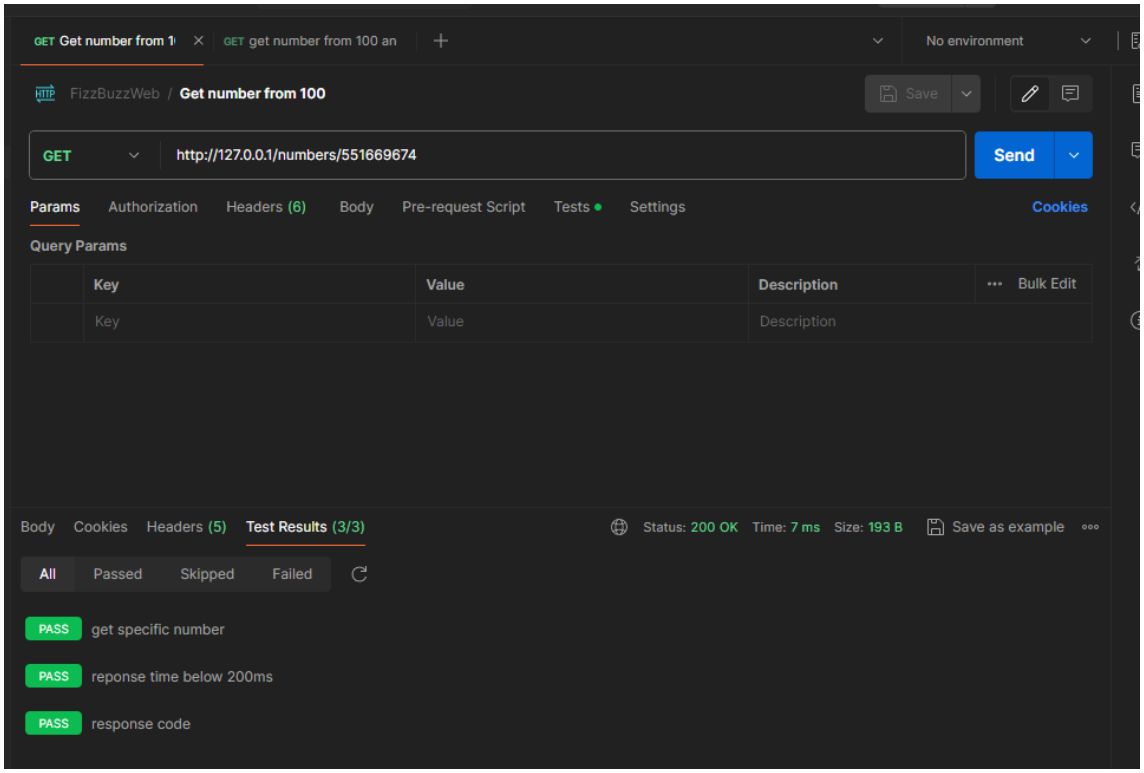


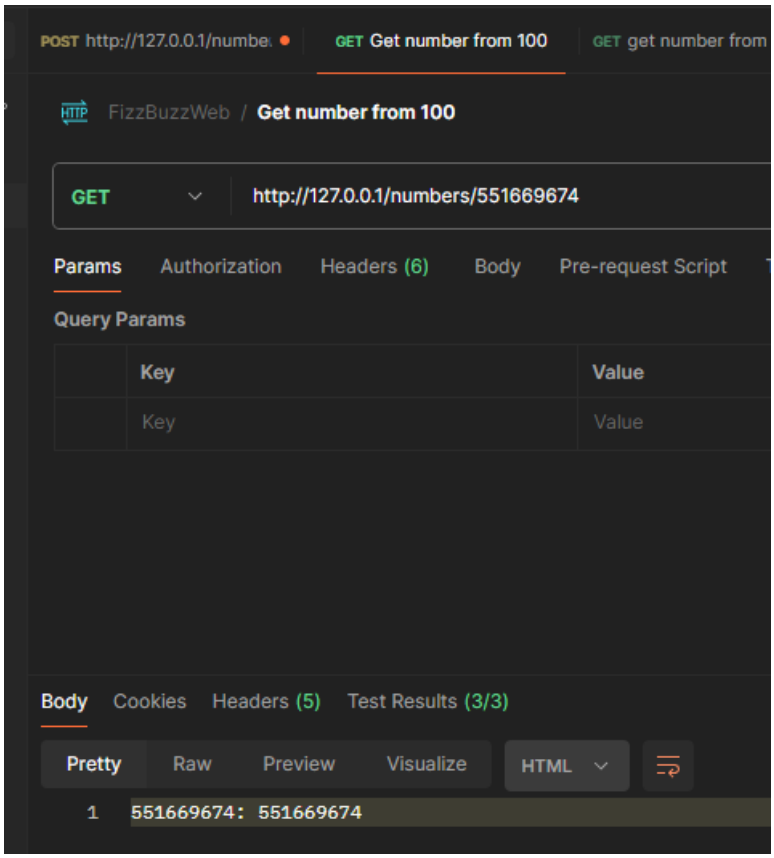
Then we have “my_app”, which is going to bridge the database to the main program, this is where we will get our responses from depending on the current state of the data base.



And last, we have main, where we will run our app and receive the data to validate. Here we instantiate our class MyApp so we can access its methods and do the logic.

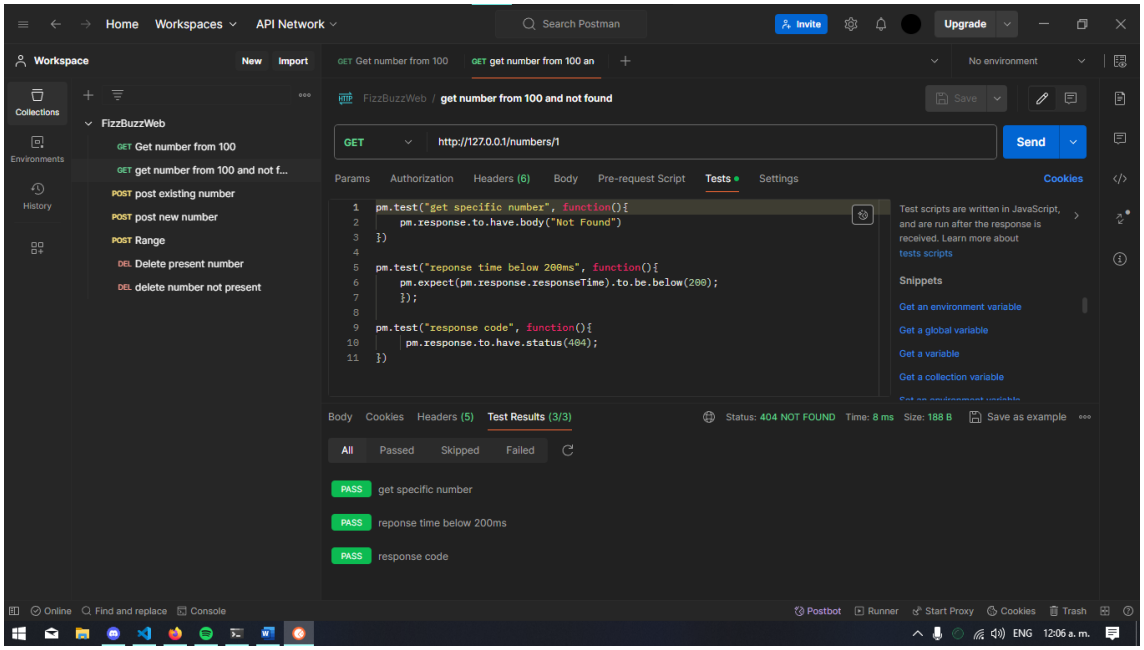
- For the first section we get any number from the 100 first declared in our data base. As we can appreciate from the screenshot, all our tests pass, confirming the expected outcome.





In addition, here's what we get from the API.

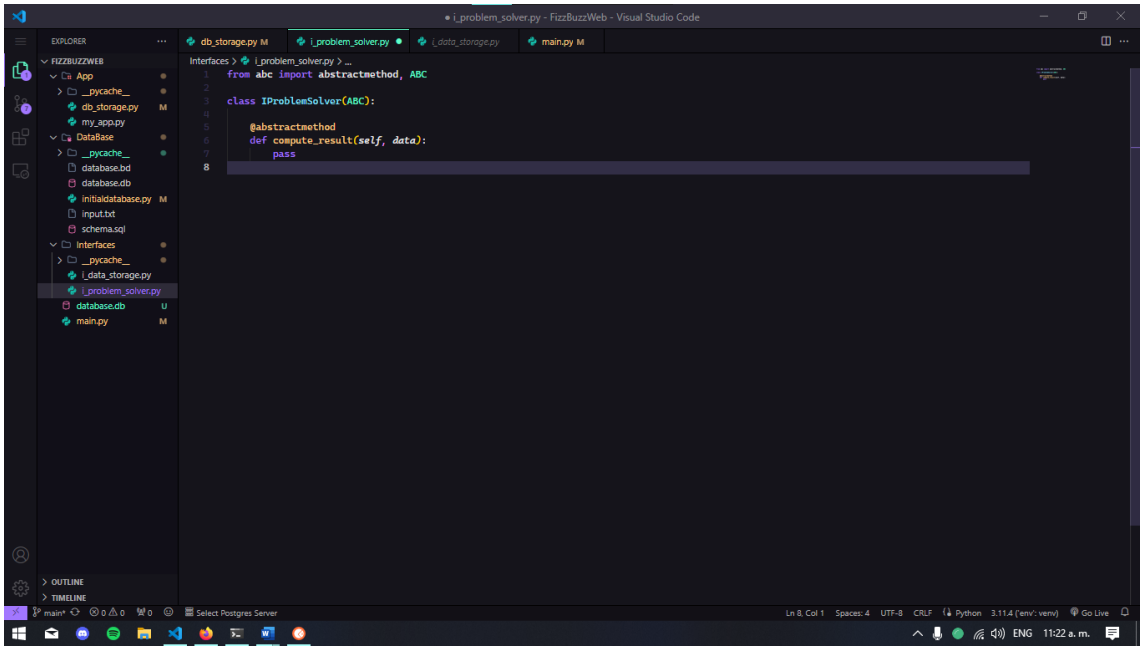
- Next section we try to retrieve the record for the number 1 which is not stored in the database. As we can see from the screenshot, all the tests pass; additionally, the “response code” test passes legitimately since we are getting the HTTP code from our API’s response.



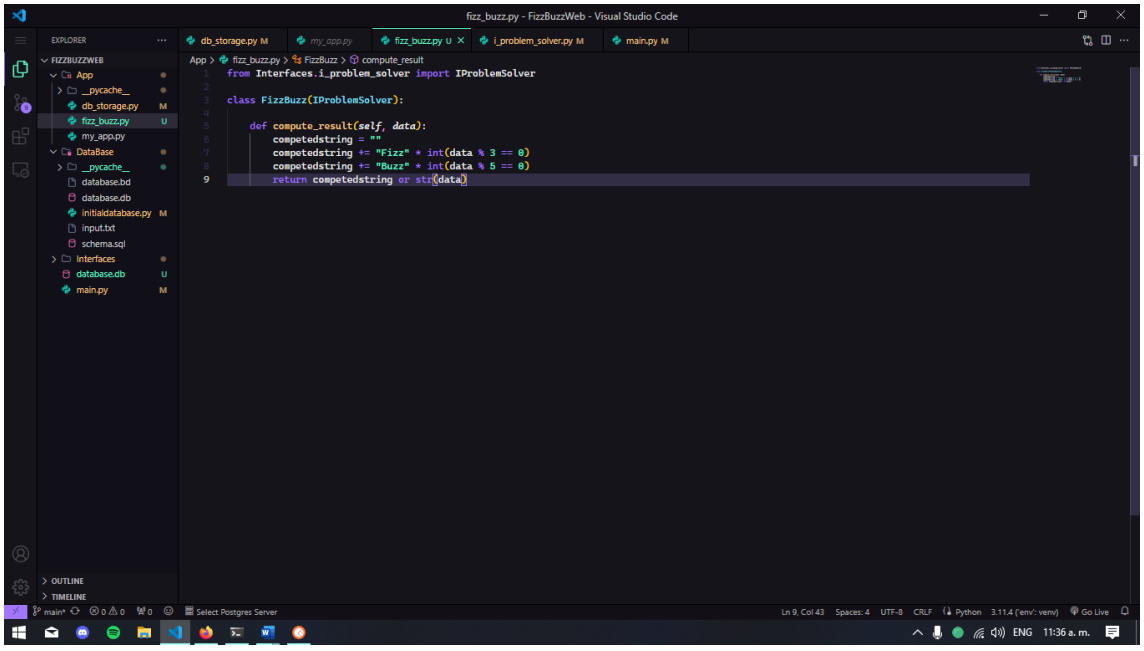
Here we make a slight modification, where we add a new method to the interface. This new method (get_active_data) in addition with “get_data” will make the process easier when it comes to retrieve data that may be active or not.

```
nterfaces > i_data_storage.py > ...
from abc import ABC, abstractmethod

class IDataStorage(ABC):

    @abstractmethod
    def post_data(self, data):
        pass

    @abstractmethod
    def get_data(self, data):
        pass

    @abstractmethod
    def get_active_data(self, data):
        pass

    @abstractmethod
    def delete_data(self, data):
        pass

    @abstractmethod
    def get_range(self, low_limit, sup_limit):
        pass
```

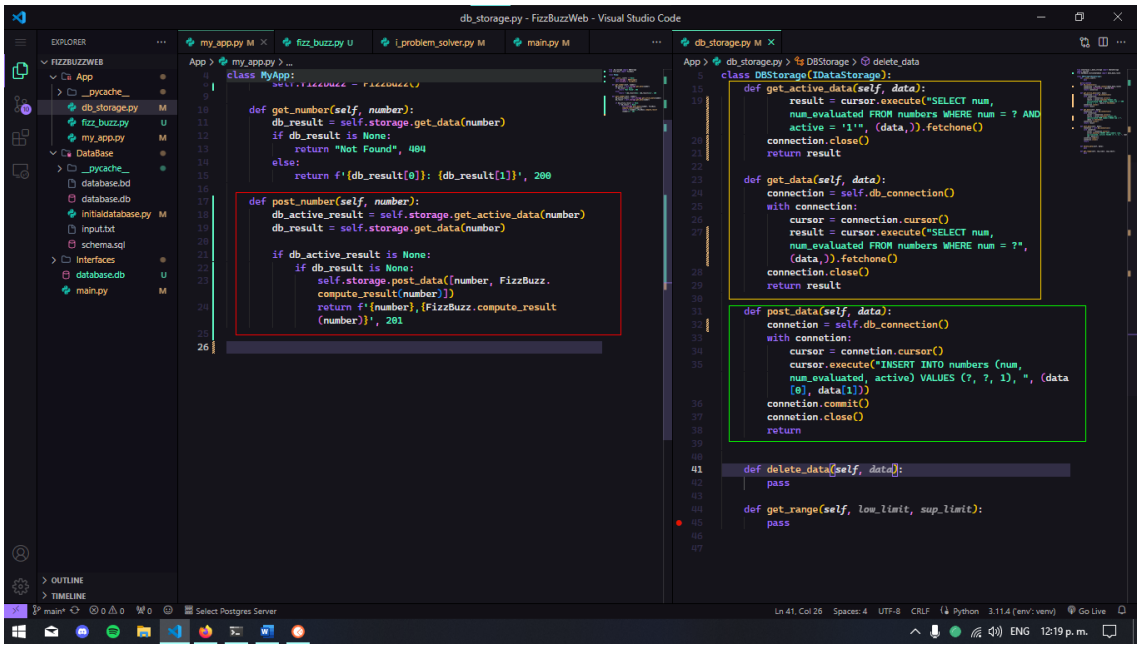
- The next section we try to add a new record of a number into the database, and here's how we are going to achieve it.



First, we define our i_problem_solver interface which holds the abstract method compute_result that will return the evaluation of any given number.



Then, we will create or class FizzBuzz that contains the method to compute the result of any give number.



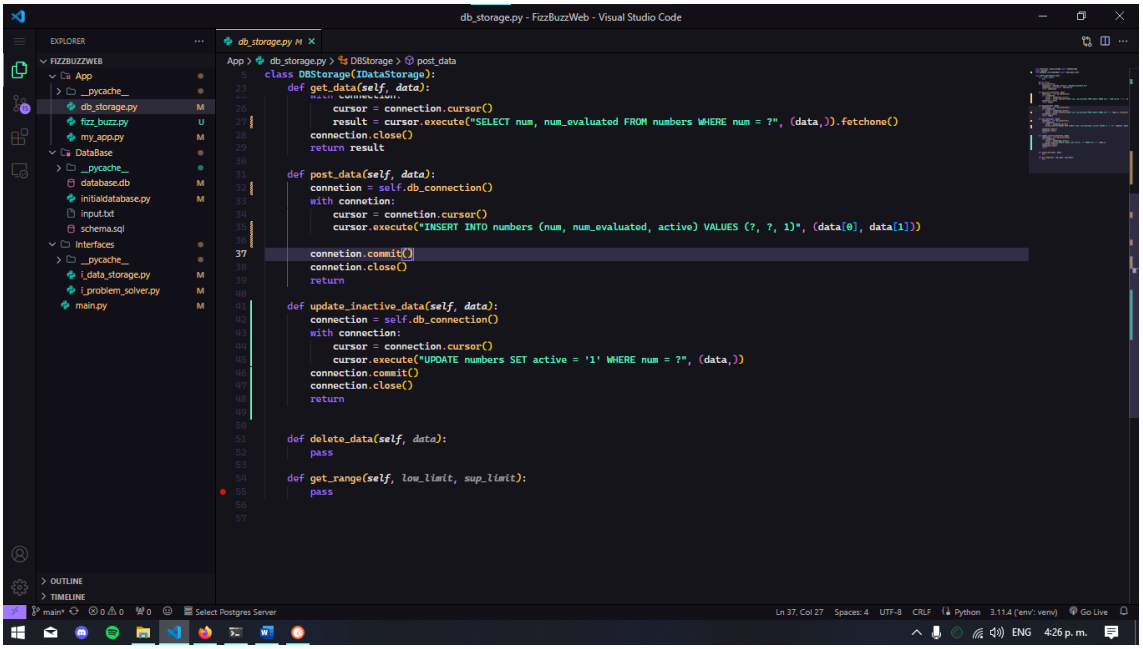
Here, in the red box, we develop the logic to add the new record into the data base, but we have problem, if the number we are trying to add is stored but not active, we haven't declared the method to set its status back to active, so we first need to declare that before proceeding with the logic.

In the Yellow box we implement the modification mentioned earlier, now "get_data "will retrieve a number no matter if it is not active, and "get_active_data" will retrieve the number only if it is active.

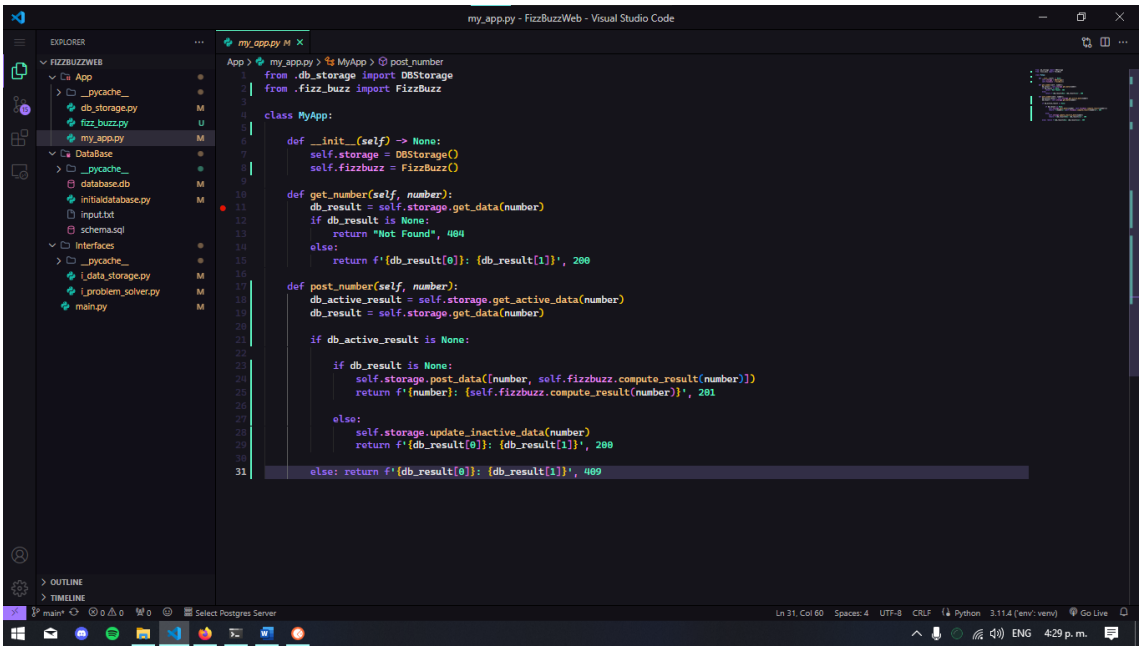
```python
from abc import ABC, abstractmethod

class IDataStorage(ABC):

    @abstractmethod
    def post_data(self, data):
        pass

    @abstractmethod
    def get_data(self, data):
        pass

    @abstractmethod
    def get_active_data(self, data):
        pass

    @abstractmethod
    def delete_data(self, data):
        pass

    @abstractmethod
    def get_range(self, low_limit, sup_limit):
        pass

    @abstractmethod
    def update_inactive_data(self, data):
        pass
```

Now we add the new abstract method to our interface to set the status of any given number back to active.

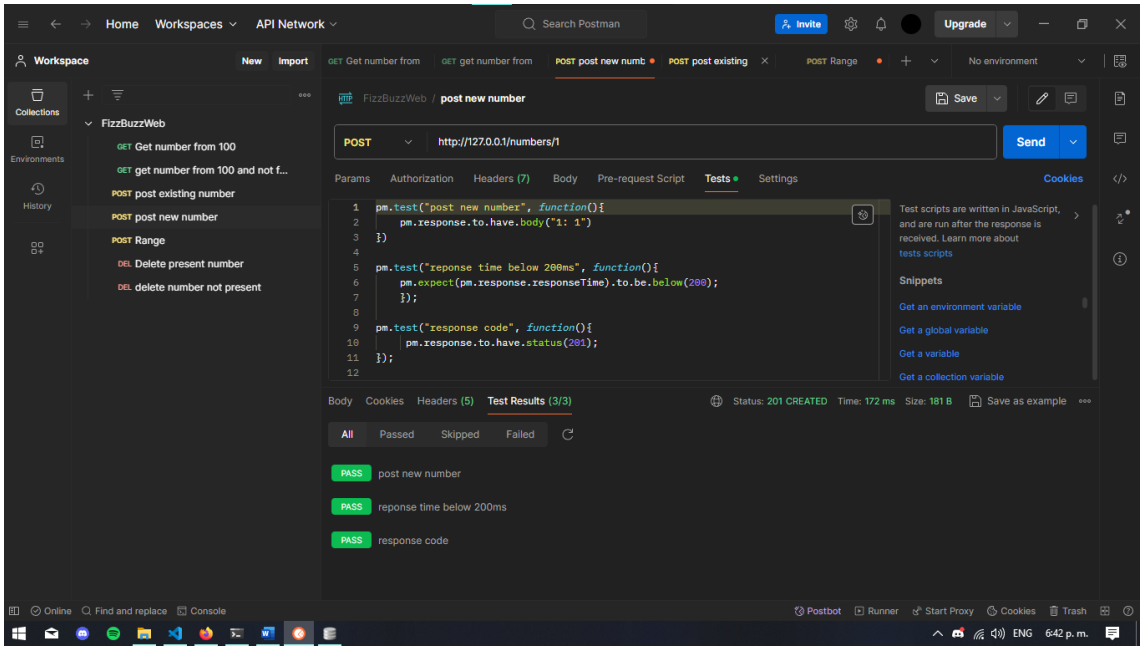
After that we develop the query to set the number's state back to active.



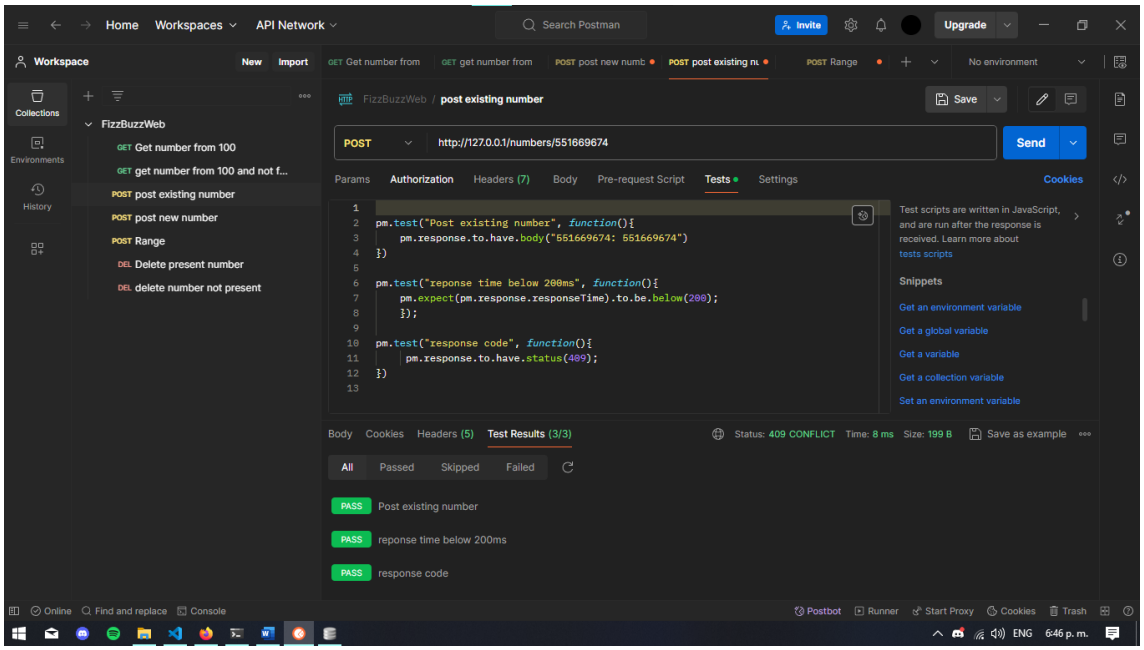
Because we finally developed the logic for updating our number's state, we can continue developing the logic for the post verb, which follows: If the number is not active, we check if it is stored, if not, we add the record into the data base; if it is stored but not active, we set its status back to active, and if it is active, we only return its value.



And with that define, we can now add our new record of the number 1.



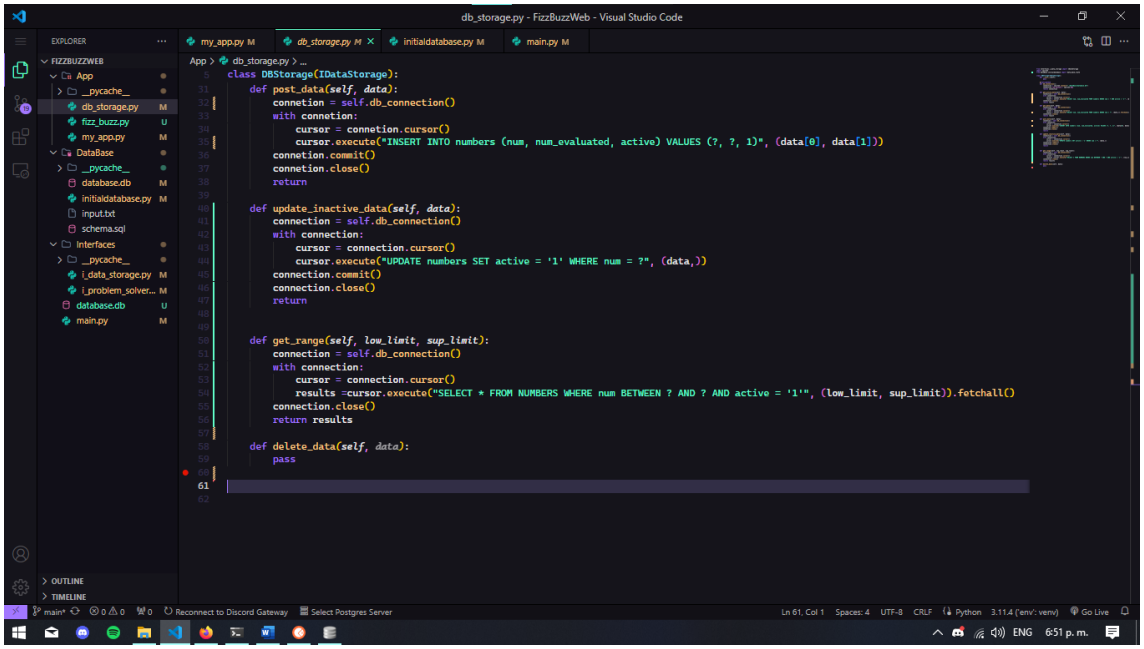
- In this section we try adding a number that is already stored in the data base, in this case, 551669674 is our test subject.



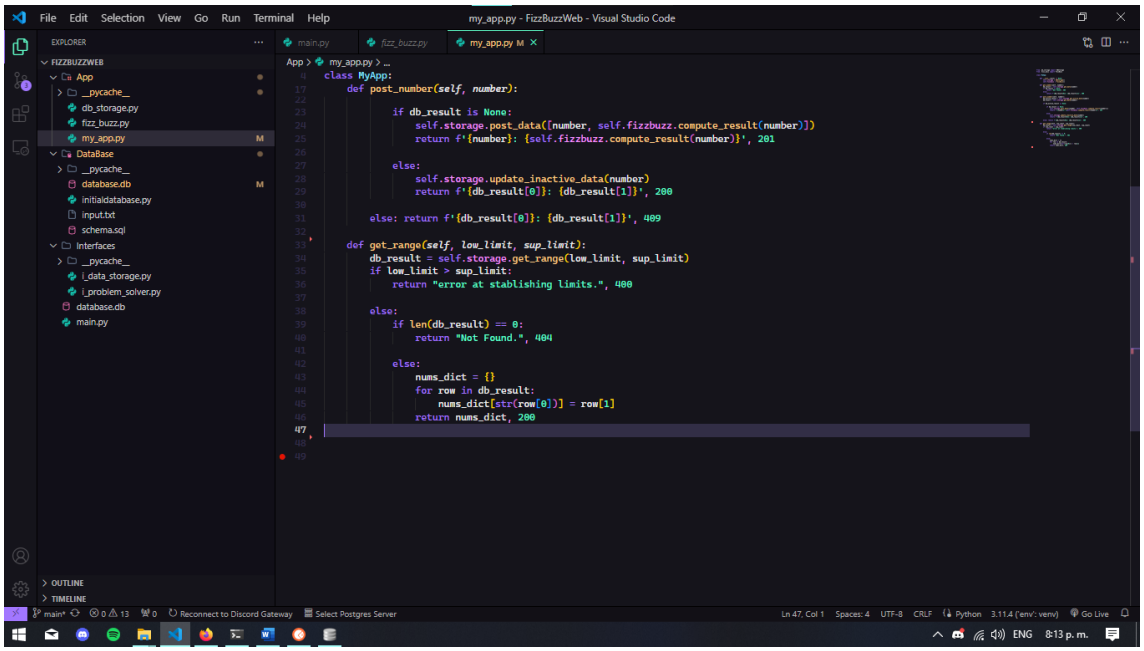
We could try adding a number that is not active, but since we haven't yet defined the logic for that, we will test that out later on.

- In this section we try retrieving the range of the records for the limits established back in the report.

First, we define the query logic to get the range.



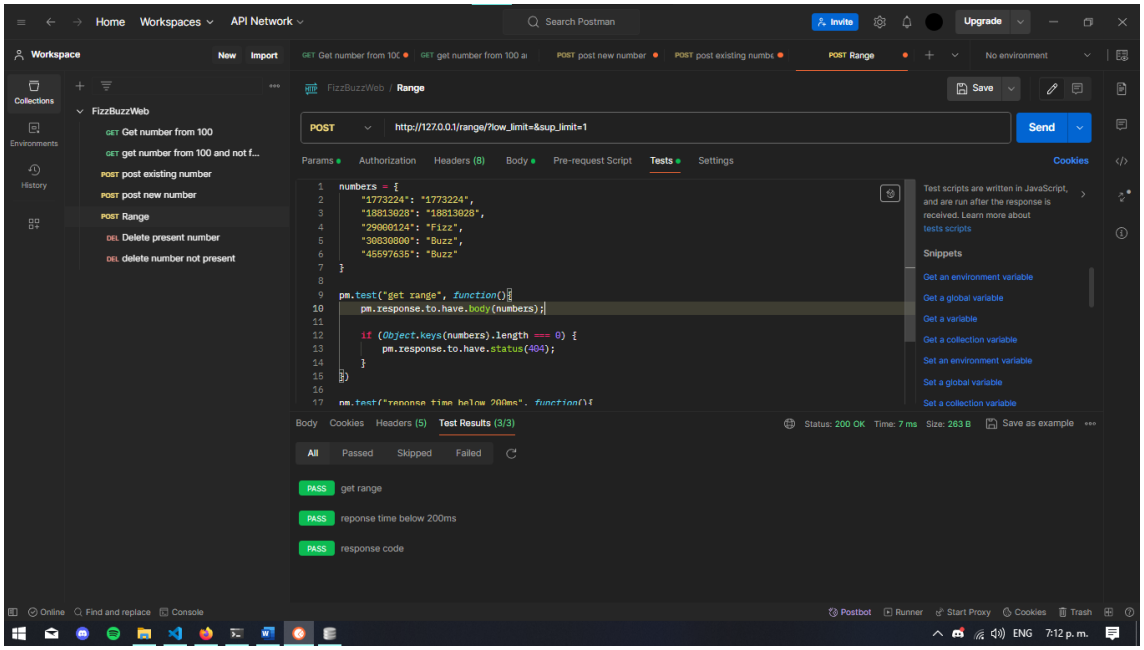
Then we define our API's response depending on the result retrieved from the database.



The final part consists on developing the logic to receive the request data from Postman and sending it into our API.

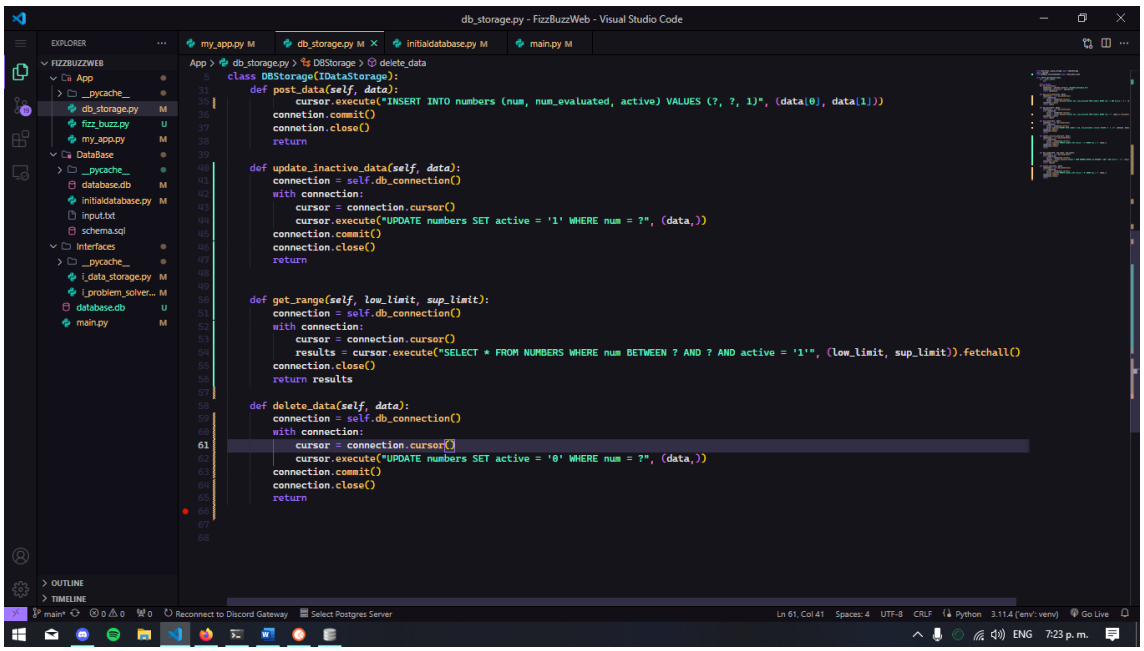
```python
from flask import Flask, jsonify, request
from App.my_app import MyApp

app = Flask(__name__)

my_app = MyApp()

@app.route('/numbers/<number>', methods = ['GET', 'POST', 'DELETE'])
def numbers(number):
    if request.method == "GET":
        return my_app.get_number(number)

    if request.method == "POST":
        return my_app.post_number(int(number))


@app.route('/range/', methods = ['POST'])
def get_range():
    limits = request.get_json()
    low_limit = limits["lowlimit"]
    sup_limit = limits["suplimit"]
    return my_app.get_range(low_limit, sup_limit)


app.run(host = "0.0.0.0", port = 80)
```

Here's what we get: all the tests pass and we are ready to go to the next section.

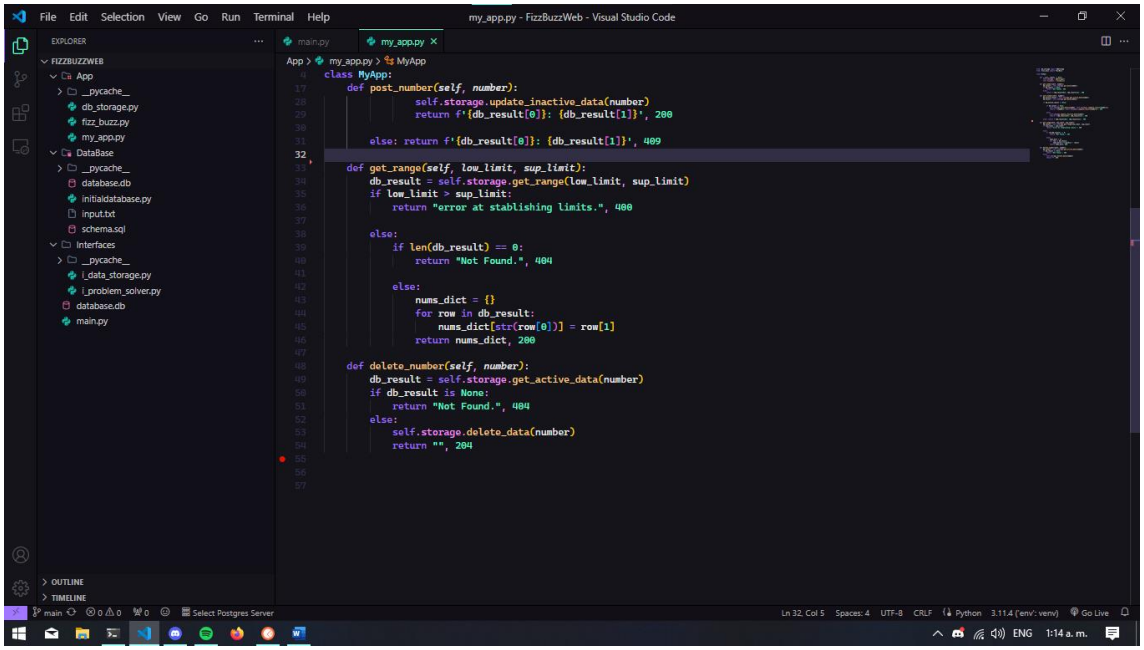


- In this section we try to “delete” the record of a stored number from our database.

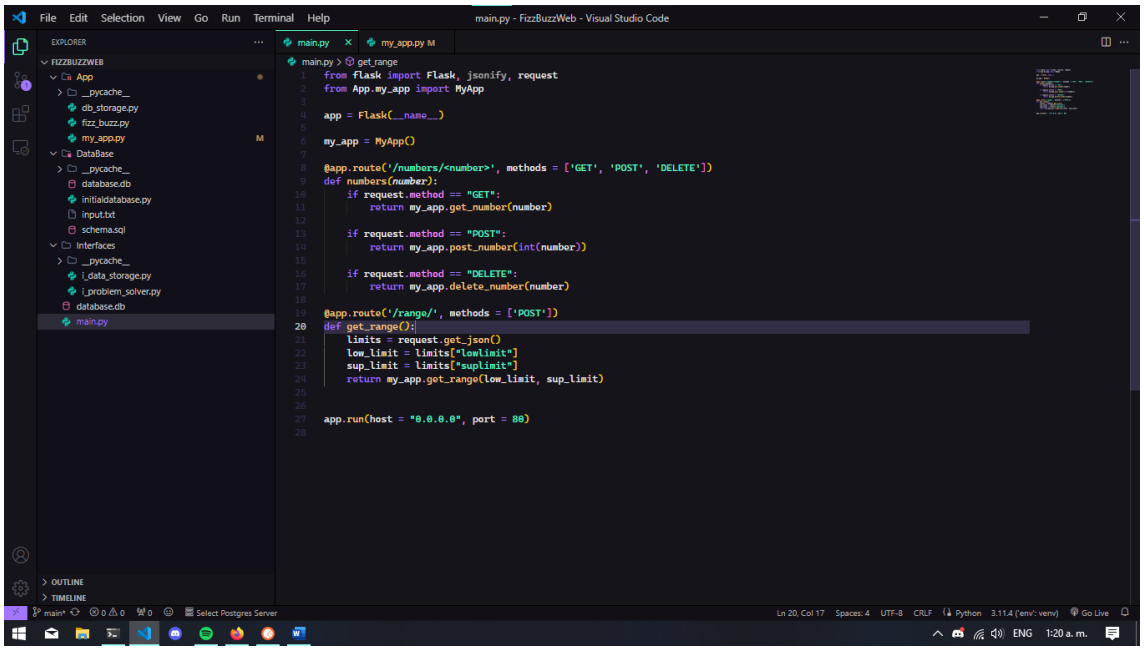
First, we define the logic for our SQL query, which is going to perform a soft deletion to our requested number.



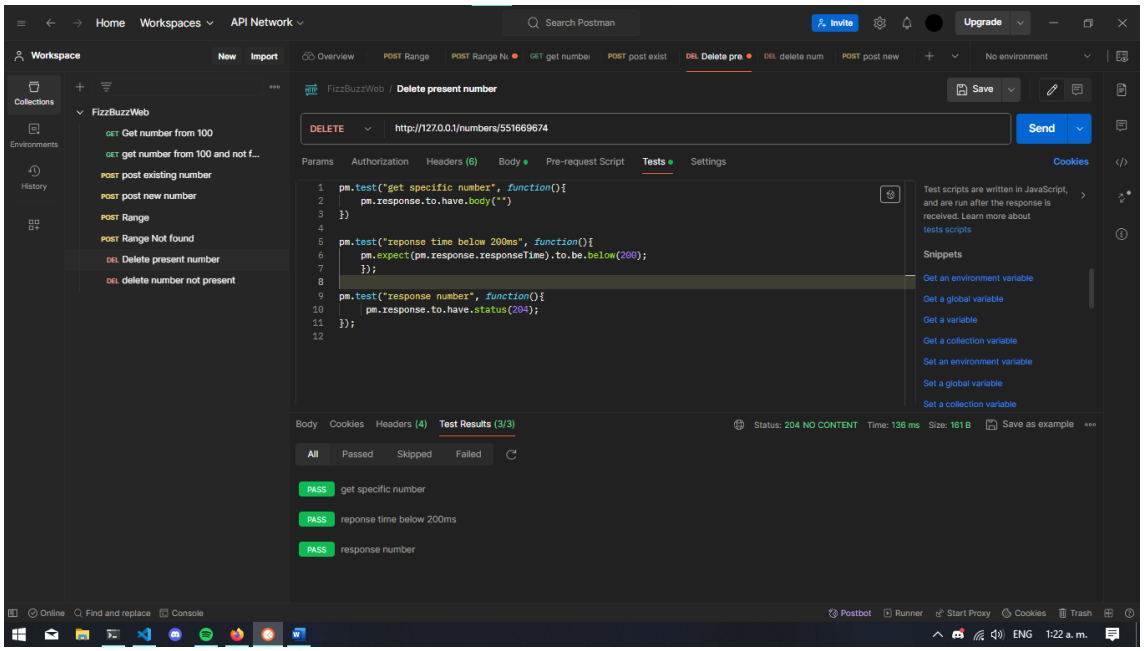
Then we define our API’s response depending on what we retrieve from the database.



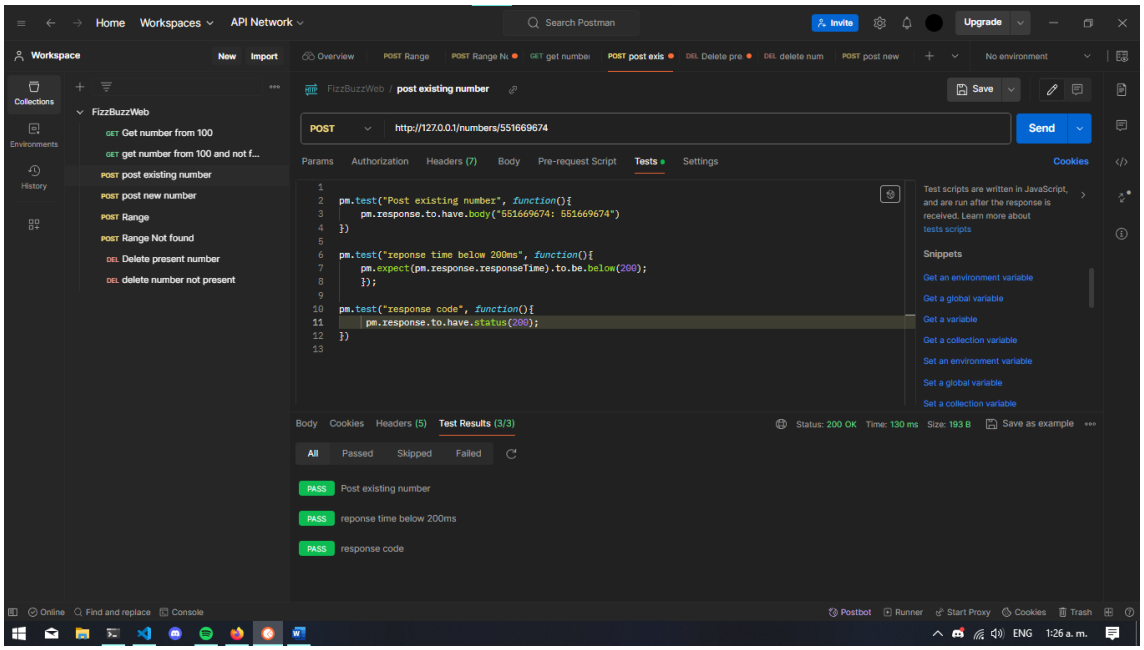
Last but not least, we define the logic to receive our data.



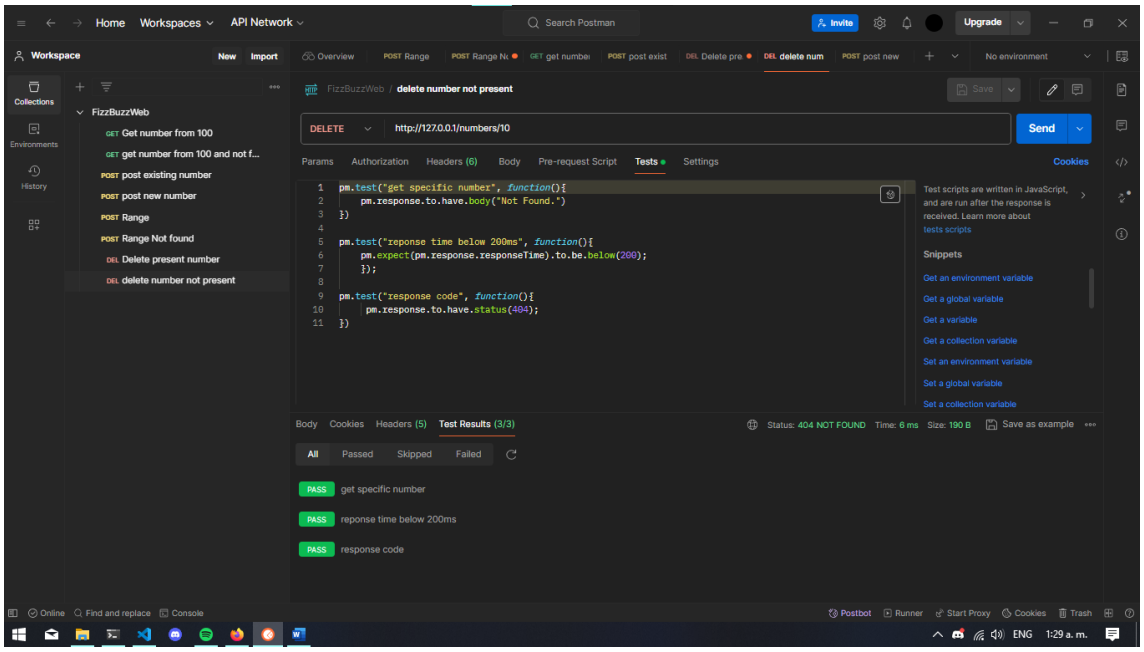
With all the logic defined, we can continue with our tests, first we try deleting the record of a number stored in our data base, and see what we get.



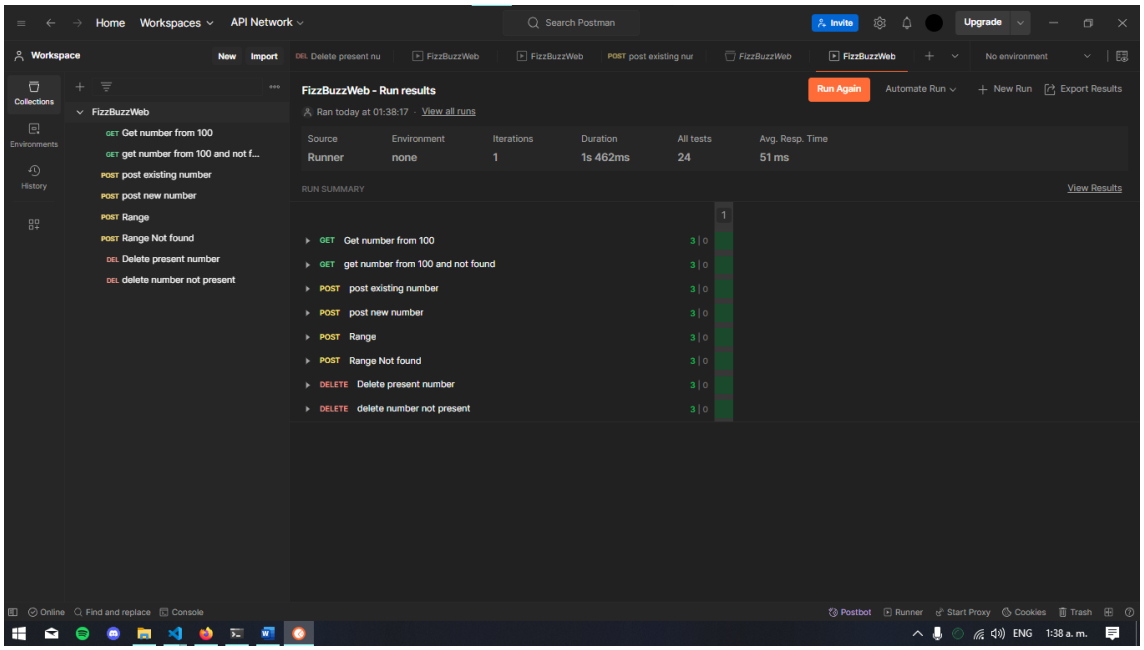
Our number 551669674 is now soft deleted, which is what we expected; now we can try setting its status back to active and see what we get:



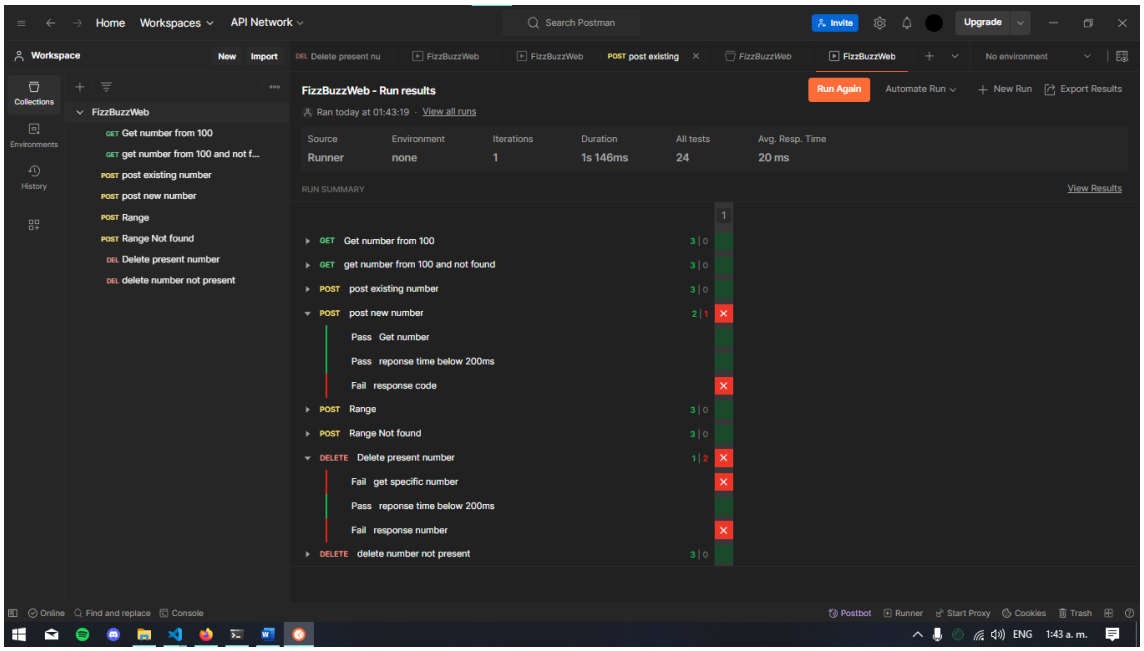
Now the number is active once again!



Our last test involves deleting the record for a number that isn’t stored in our database. As shown in the screenshot, all of our tests pass, confirming the desired outcome.



Now, we can run our collection of tests, with a fresh instance of our data base (the first 100 numbers) all our tests pass; let's try running it one more time.



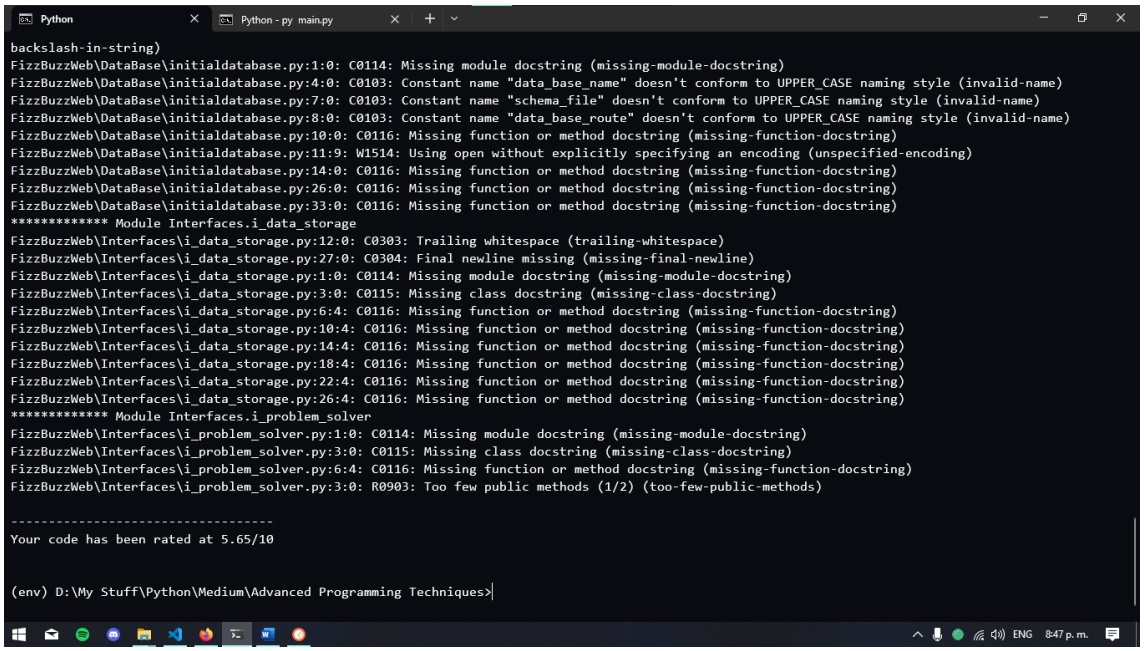
After running it once again, we get some errors, which makes sense, for instance:

The "Post new number" test will add a new record of a number into the database, if we run it twice, our API will return a different HTTP code making the test fail, same goes for "Delete present number", besides the fact if the number is no longer active, our API will return a different HTTP code.

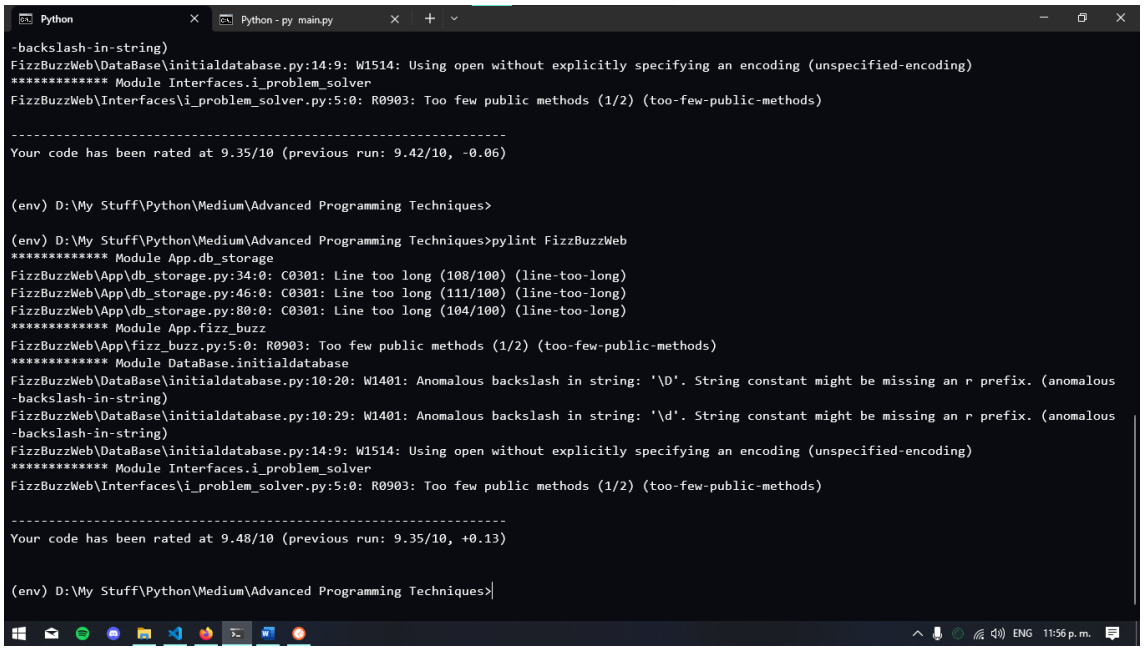
BLUE PHASE (REFACTORY)

We have succeeded on creating our program, now it's time to make it a little bit pleasant. To check our code's indentation and readability, we will use a static code analysis tool for python (Pylint) what will ensure those factors.

Running the command to check one the files, we get 5.65/10 score which is pretty bad. We will try to get at least a 9, so this is going to take a while.

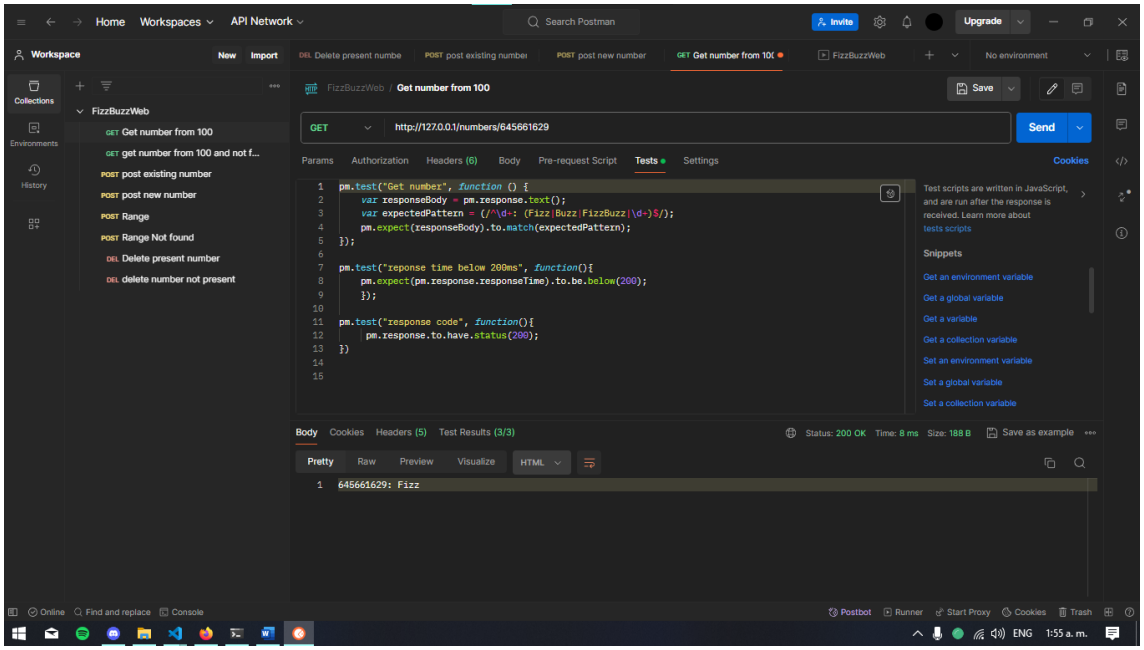


This is the initial evaluation for every file in our project.

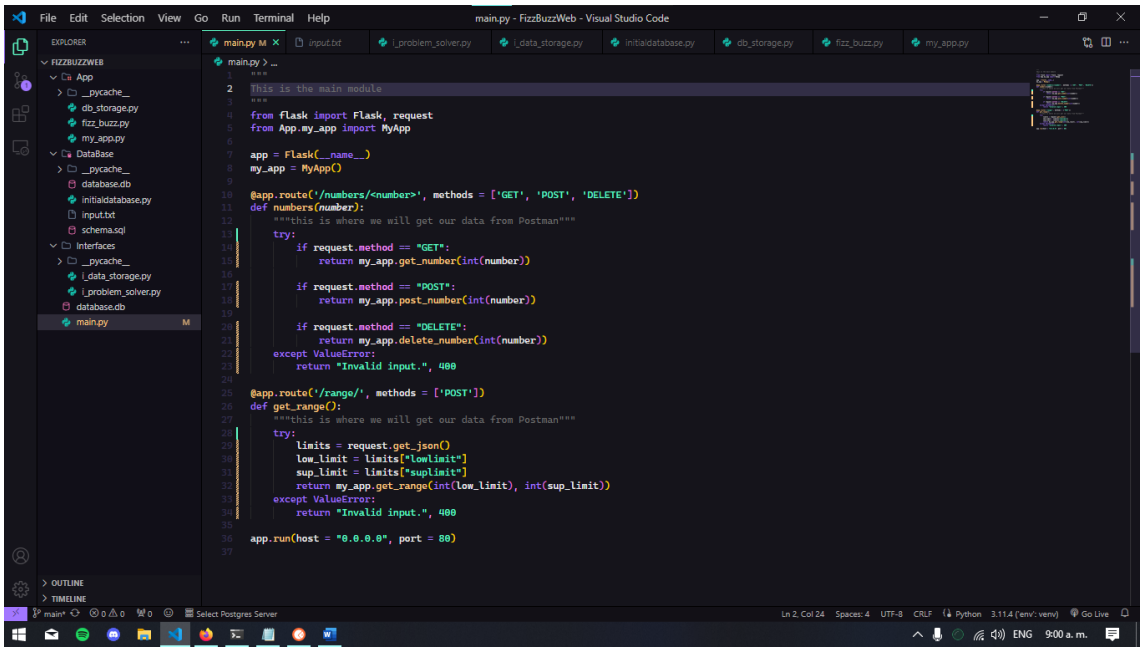


After some corrections, the max score we get is 9.48/10 which is fairly satisfactory.

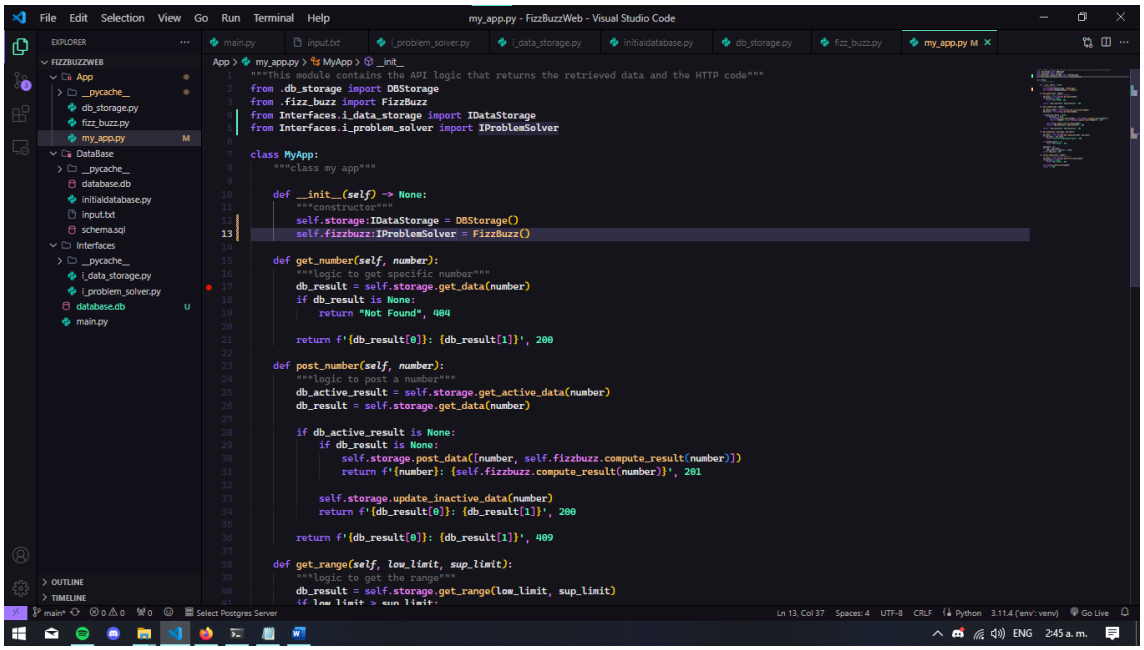
AUTOMATIZING TESTS



This slight improvement will make our life easier when it comes to testing different numbers, instead of manually switching the test, now we only need to enter the number in the URL and call it a day.



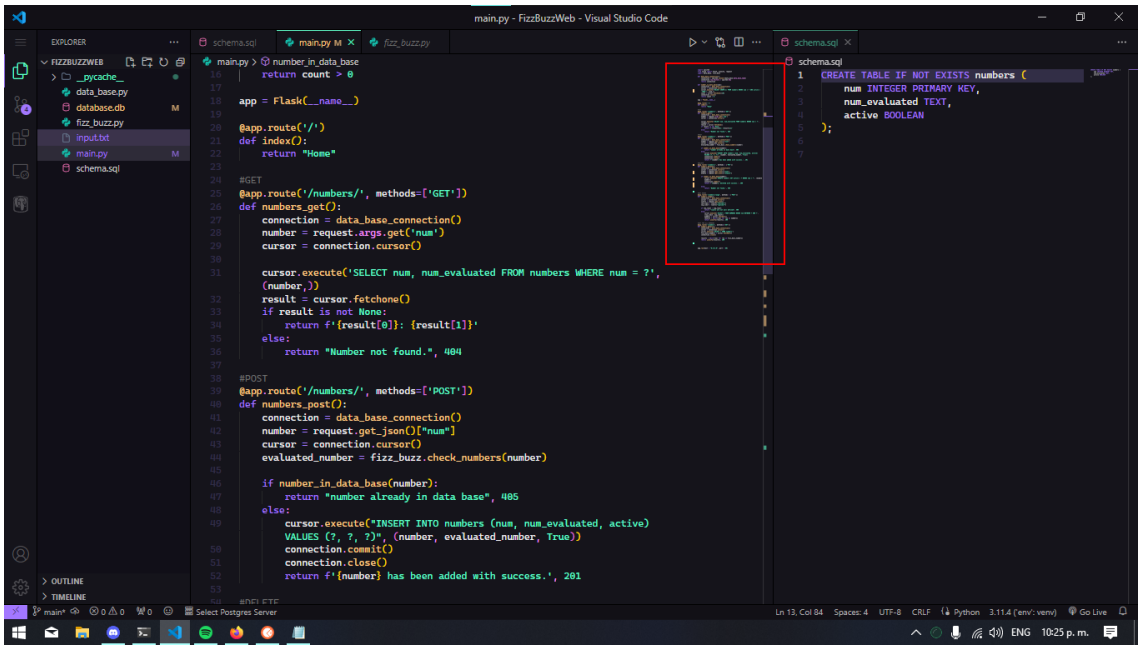
To handle any non-expected input error, we use a try except block in order to display a message for any bad request.



Last modification will be in our “MyApp” constructor, we set our instances of “DBSorage” and “FizzBuzz” to be their interface type, since they both satisfy their respectively interface.

CONCLUTIONS:

So, we are finally done with our project, this challenge really thought me a lot about HTTP response status code, API’s (I wasn’t really aware of APIs until this very semester), besides applying the DIP, which at the beginning I was hard struggling to abstract the structure, but the more I spent time on it, the more I got used to it, so at the end I was comfortable working with it, and it really makes the code way cleaner (when I was first messing around with it, it was lots of repetitive code).



Here's a screenshot from 4/14/2024, and as shown in the red box, the code wasn't really that pleasant to see.

The use of AI like ChatGPT helped me when it comes to correcting my English grammar, setting the first instance of the data base and visualizing it using pandas, connecting to the data base with less lines of code using "with", automatizing the Postman tests, among others.

Watching videos, consulting on the official documentation and Stack Overflow also helped me a lot when I was with no direction to follow.