Vapor and Swift

The Backend for beginners

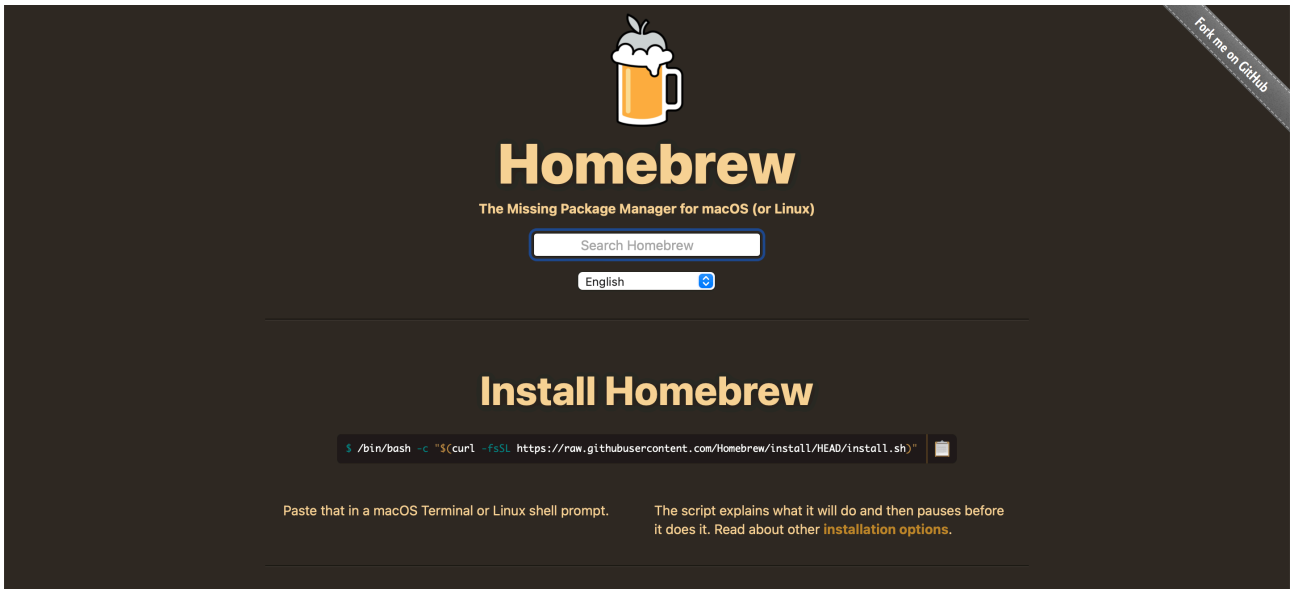
Tutorial

Introduction

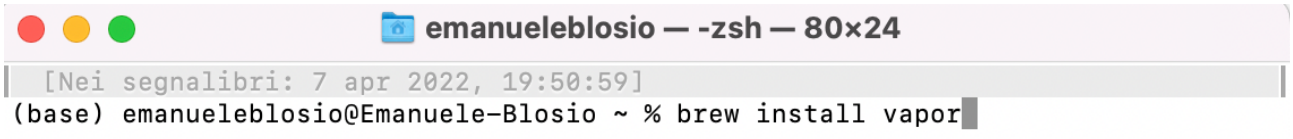
Usually, when a beginner approaches programming, he or she prefers the 'front-end' side and is comfortable because they have a way to actually see what they are creating with the language in code. As in the case of the simulator or the canvas in Xcode.
Every time we add a command, function or anything else we can test what we've just created.
When it comes to the back end, however, it seems that everything is much more "obscure" than what the front end can provide. A beginner who has practiced well with the front-end may feel lost in an environment where they don't think they can actually see what they are doing.
In fact, the back-end with Swift and Vapor is a concept that is easily approached thanks to this framework and some programs that you can use to make it easier to create your own API.
The programs you need to install are HomeBrew to manage the packages, Vapor to build your API, Postman to test the API without having a working iOS app, DBeaver or Azure Data Studio to interact with the Database and Docker to activate your database and make it working.

To demonstrate this, I will guide you through a small "hands-on" tutorial that will take you through creating your own API. I made this tutorial right after using vapor for the first time.

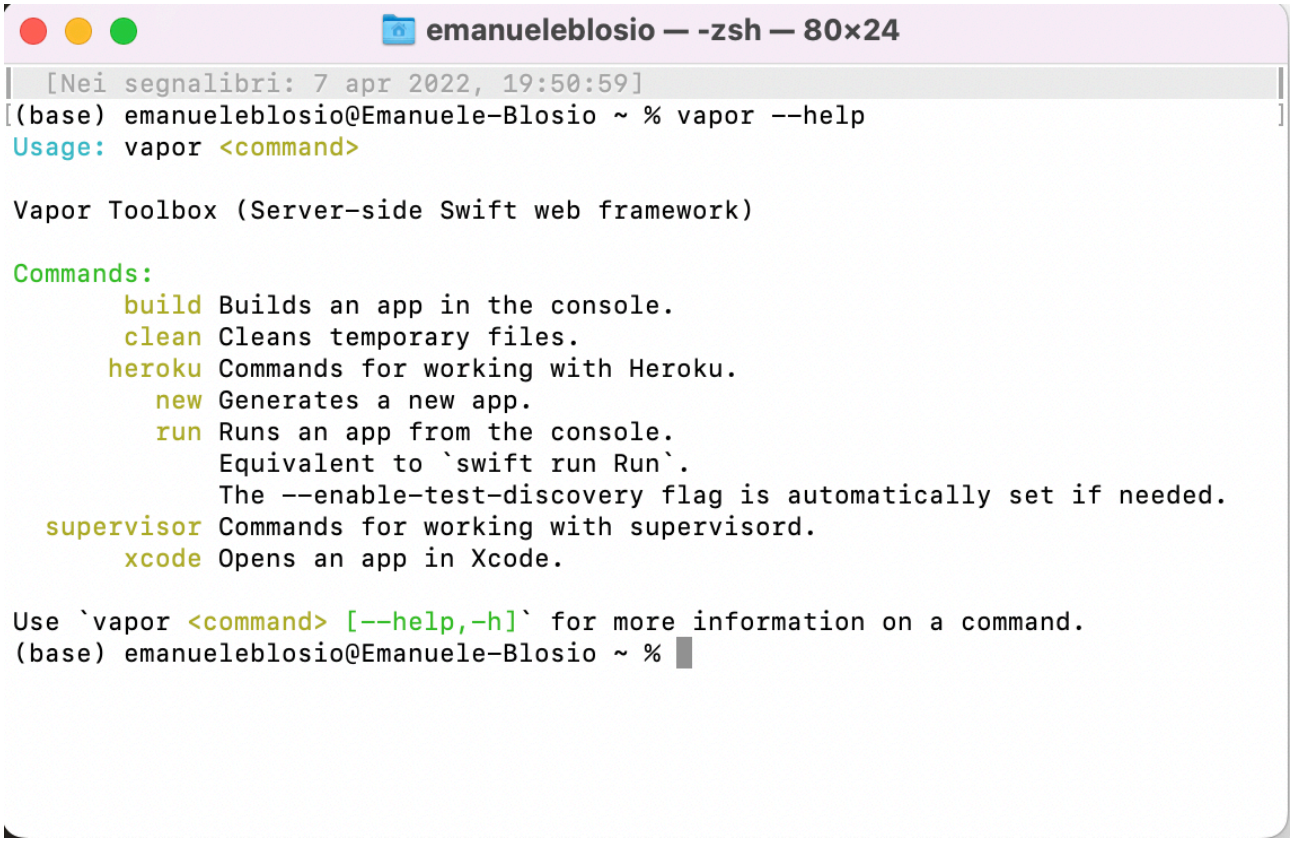
To get started, you will first need to install HomeBrew by pasting the command given on the site into your terminal. Homebrew is a package manager for macOS,



You can then instruct your terminal to install vapor with the command "brew install vapor".



Vapor is a web framework, specifically used to create the backend of your iOS app.
To see if the installation was successful, type vapor -help in the terminal and you should get a result like this.



Then you need to install postman, which will serve to test the API without having to have the iOS app already running. You can easily download the official app from here:

https://www.postman.com/downloads/

After that, you just need to install azure Data Studio, which will be used to read the data from our database without having to have everything working.
Azure Data Studio is not compatible with M1, but don't worry: to replace it, you can easily install DBeaver. Below are the links to the two programmes.

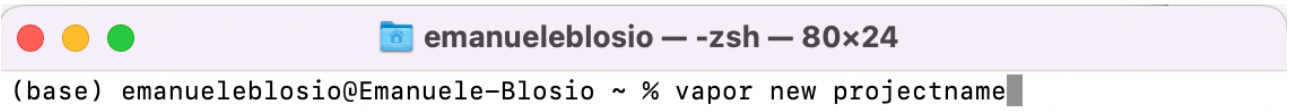
https://docs.microsoft.com/it-it/sql/azure-data-studio/download-azure-data-studio?view=sql-server-ver15
https://dbeaver.io/download/

And finally, you need to install Docker.

You can easily download and install docker desktop from here:
https://www.docker.com/products/docker-desktop/
In case you have Azure Data Studio installed, you will have to download PostgreSQL as an extension, while in case you have DBeaver, you don't have to do anything.

Let's get started!

To create a new project in vapor, just type in terminal vapor new *project name*.

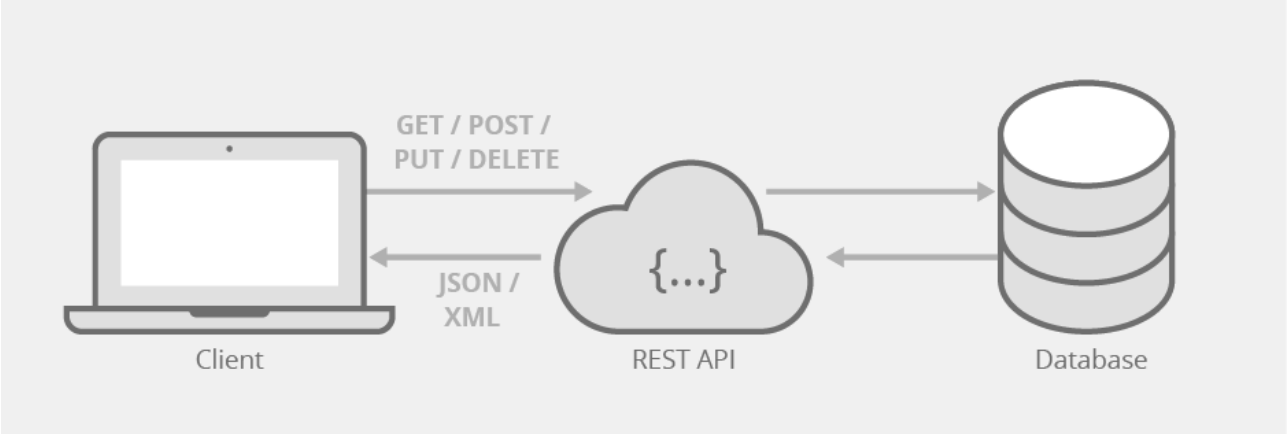


It will ask you if you want to use fluent, which is an "object relational mapper", you say yes, because it will be crucial in order to be able to relate to your database.. Next, it will ask you to use a database type, here you must select Postgres.
For Leaf, select no, as its use is mainly for websites.

Go to the root directory of your computer and open the "package.swift" project.
At first, it will take a while for it to retrieve all the packages.

In the meantime, let's start by explaining the main operations that we can exploit with an API.
There are four operations, known under the acronym CRUD: Create, Read, Update and Delete.
Correspondingly in code, POST, GET, PUT, DELETE.

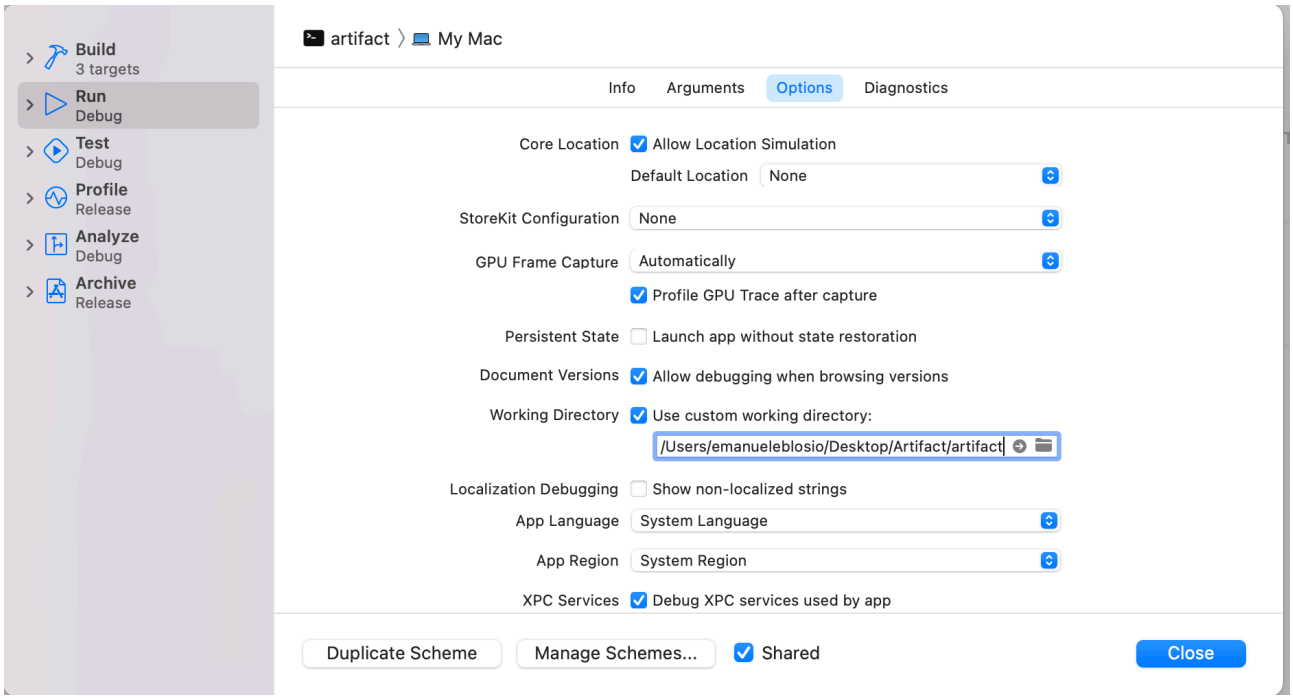
The pattern is simple: the iOS APP will request data from the API, which will then contact the Database. The Database will then return the data to the API, which in turn will return it to the iOS app in the form of a JSON file.
To make this clearer, here is a diagram.



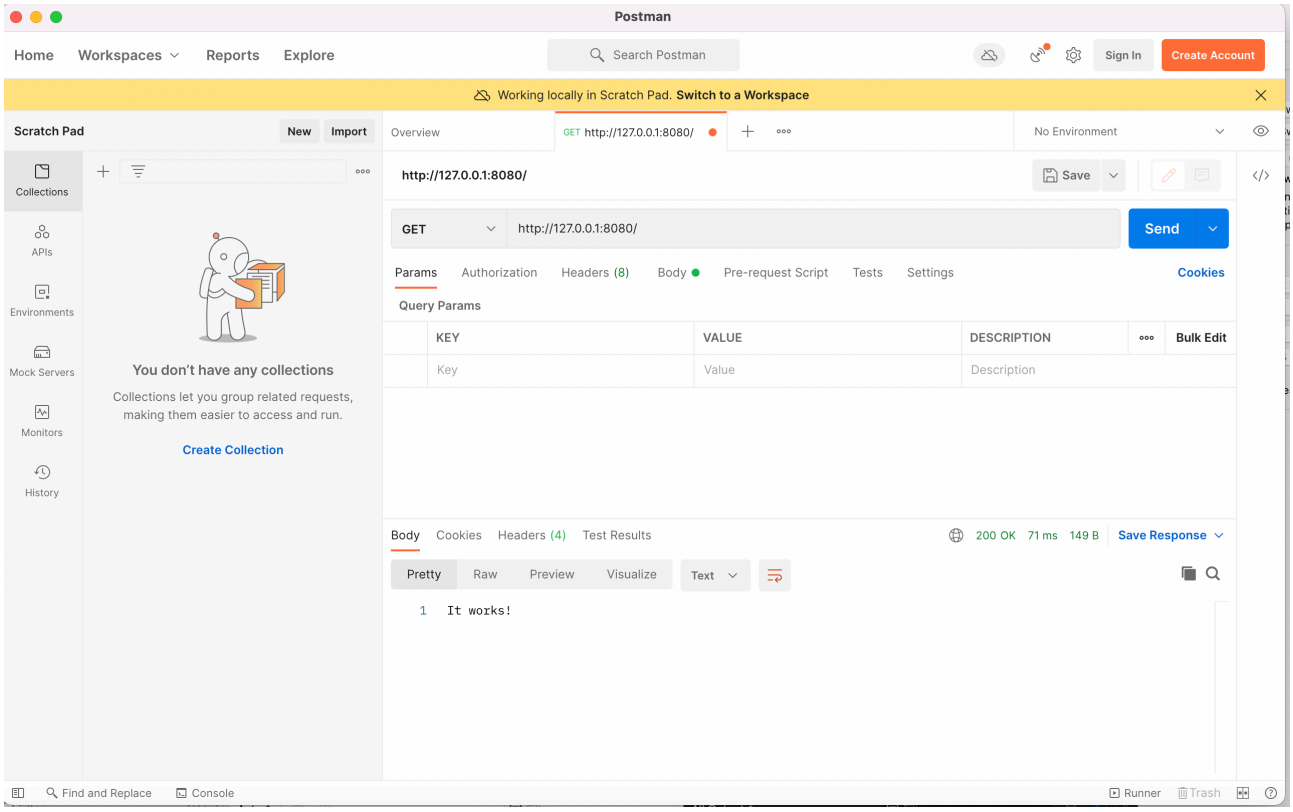
As we said before, Postman will be used to read data from the API without having the iOS app still running, while Dbeaver/azure data studio will be used for the database.

Let's get started for real!

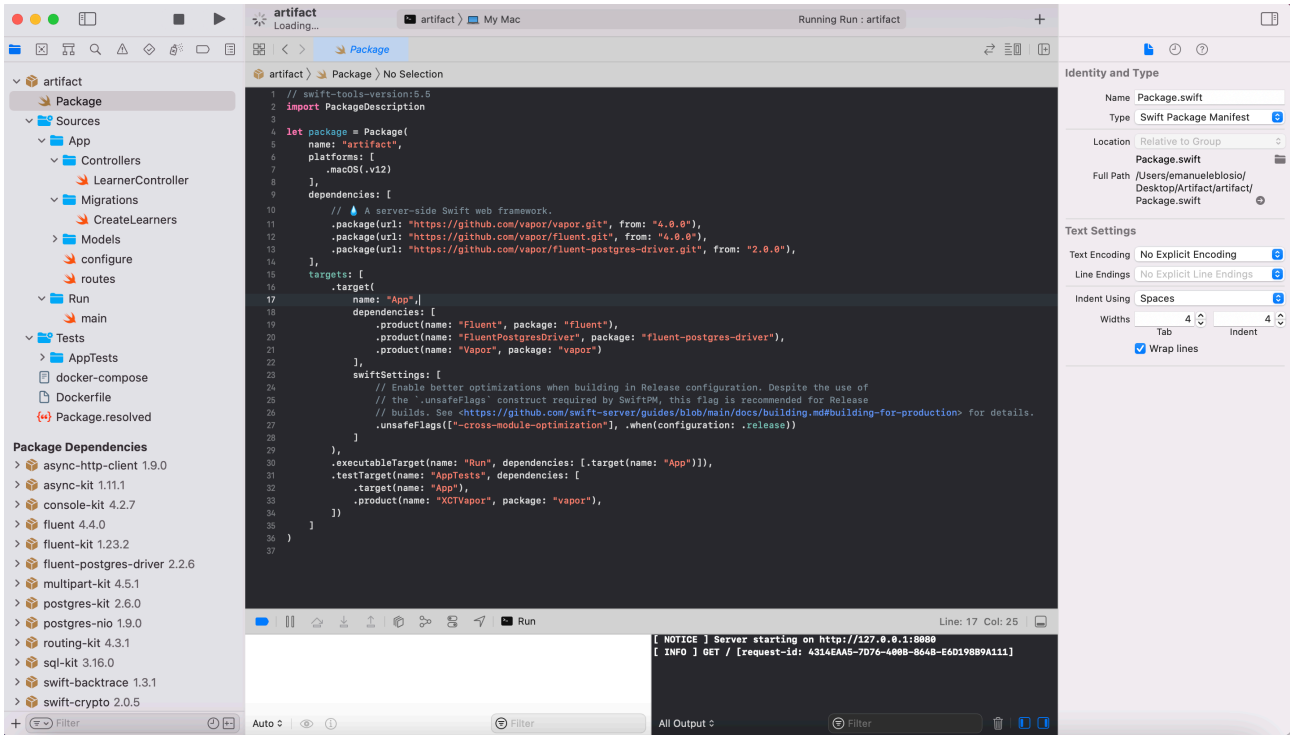
The first thing to do is to select "My Mac" as the target to run the project.
To get started, you will need to click on "edit scheme", go to Opttions and choose "use custom working directory, then find your project directory and click "choose". Then close.



Now, if we run the project with cmd ⌘ + R, and afterwards open postman and click on "send" with "get" request, and we will see that it will return the data set by default in the API created by vapor.



Let's look at the structure.
Package.Swift is our swift package manager. Here you can see the frameworks and the type of database we are using.



Let's look at the controller, it will give us by default a "todo application" (which we will delete because we don't need it), migration will be used for the creation functions of our table and our fields.
Model is where the models for our database are located.
Configure is where the database is configured.
Routes are the routes to which the api will ask for data from the database.
Here we can see the endpoints of our API.
Next we have a docker compose file which is used to configure our database and our API.

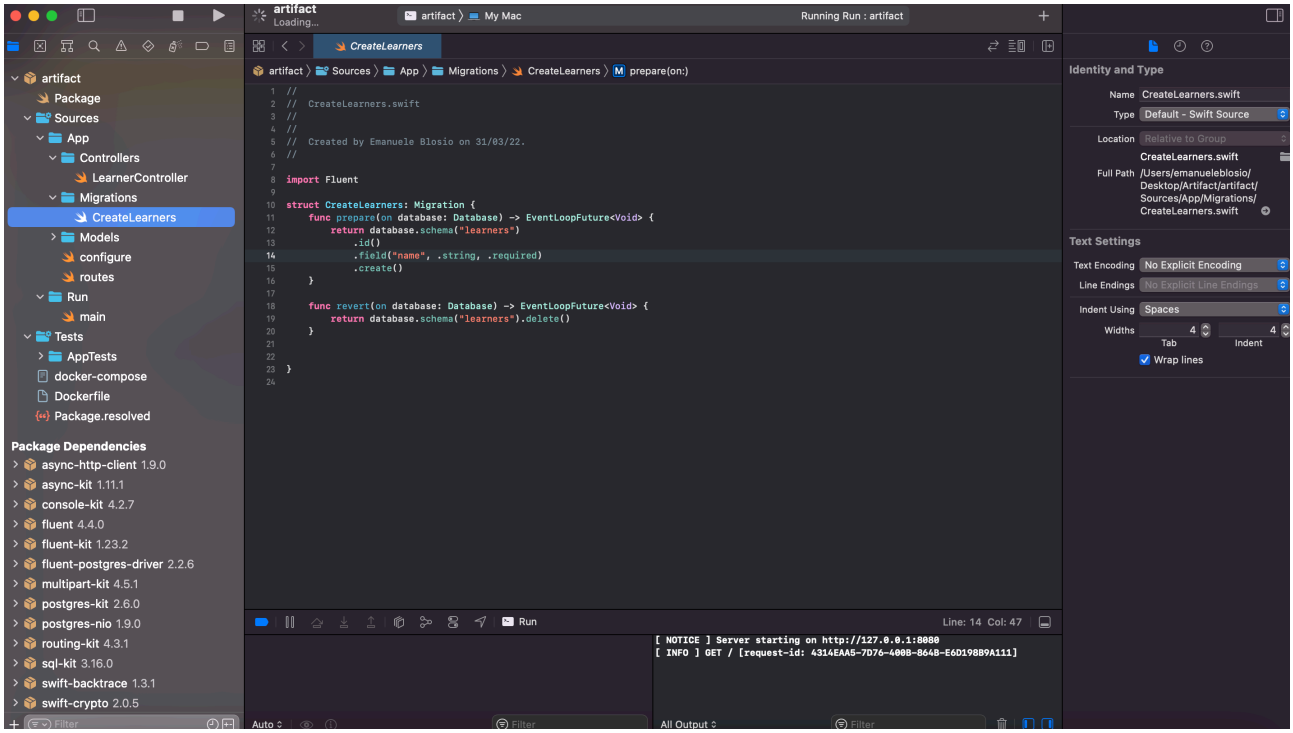
Let's create our project for the first time.

We delete the TodoController file, because we don't need it.
Delete CreateToDo Migration, delete ToDo model, delete the try related to ToDo in the routes.
We delete the migration in the configure.swift that adds the CreateToDo function, we'll add one of our own.

We will now create our own controller.
We're going to write our routes in the controller because if we have a lot of routes, it will help to order the application by not writing them all in the routes.swift (and only inserting the try that we need to execute the routes in the controller).

Let's call it "learnerController".
Then we create our Migration, "createLearners".

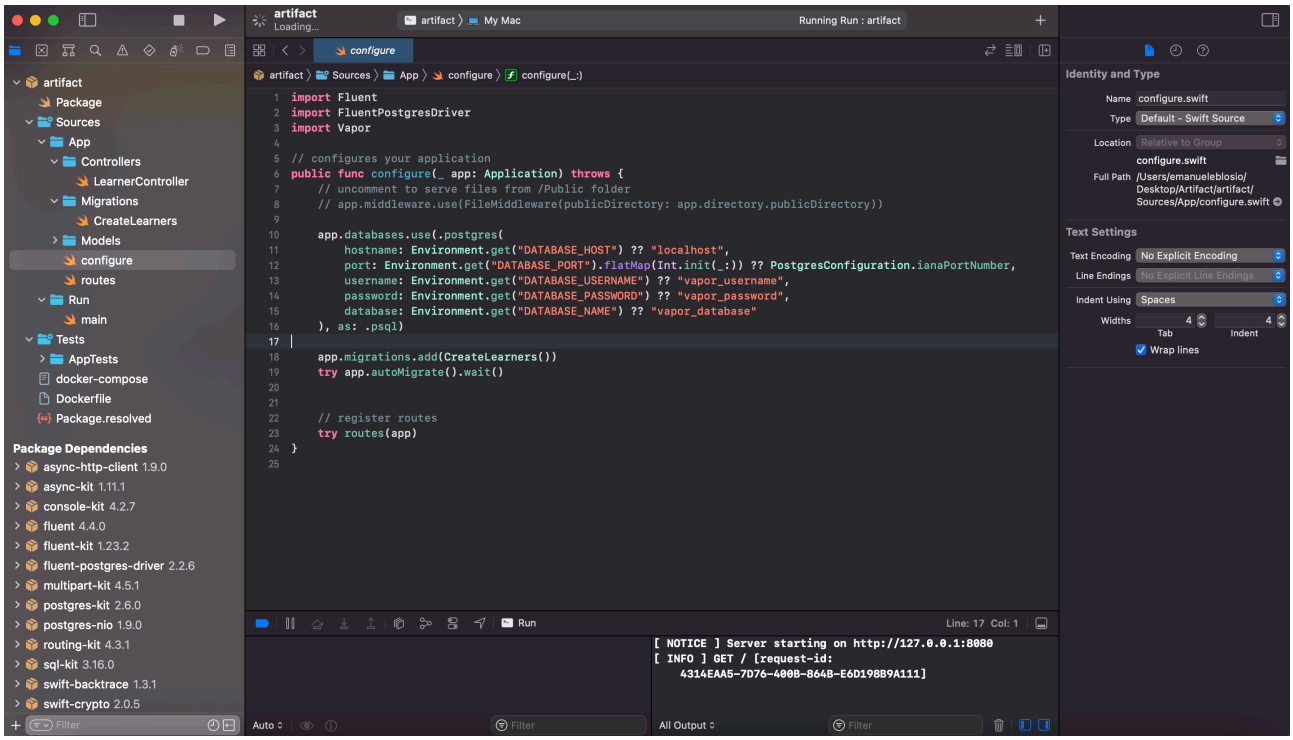
We import fluent into our CreateLearners. Create a struct called "CreateLearners".
We try to build and add the "protocol stubs" suggested by Xcode.
The prepare func is used to make the changes and the other is used to restore them if we want to.
We add the following code



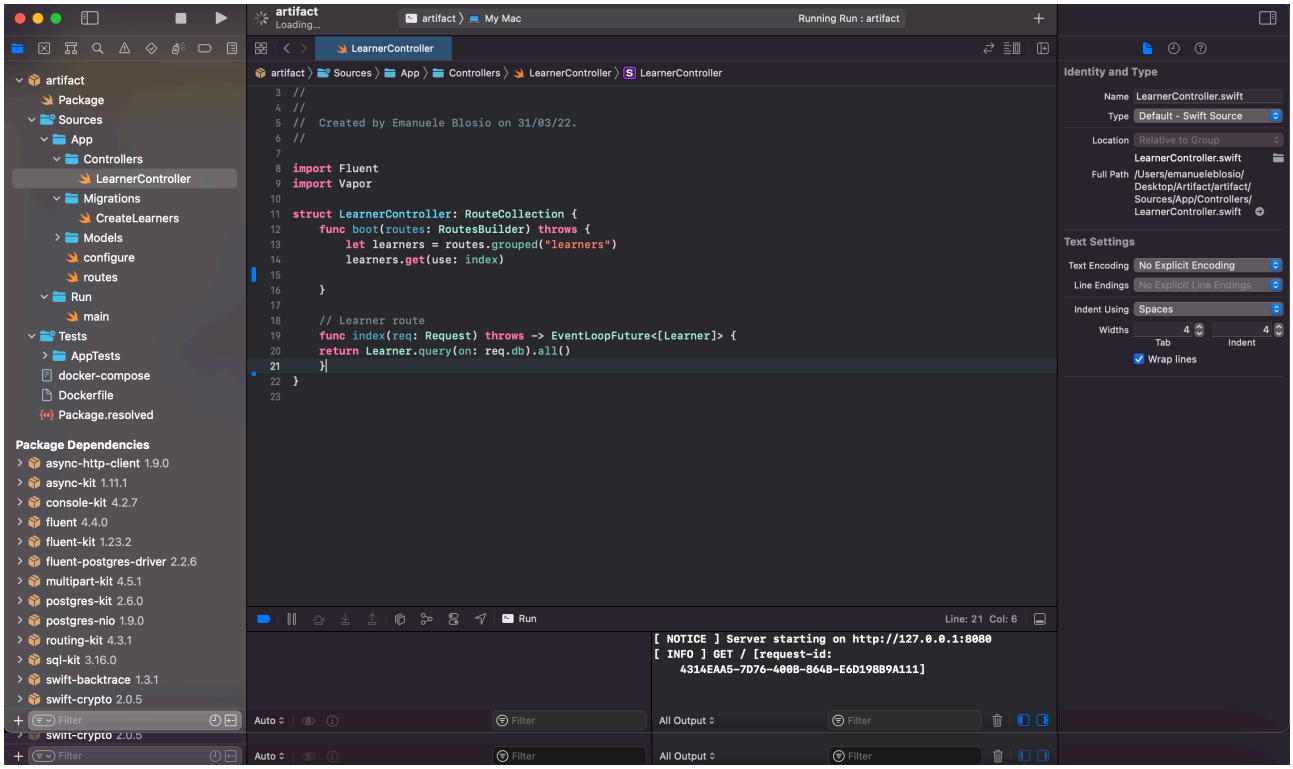
The return database.schema("learners") indicates the name of our table, into which we then insert the characteristics of our elements in the API. In this case, we'll just add an ID and a "name" which is a string type, it will be a required field.
So, we have a table called "learners", a column called ID and a column called "name".
In the revert func part, you have to enter the code.The function will delete the whole table.

Now, how do we represent the date in the table?
We need to create a model.
We go to the model folder and create a new file called "Learners.swift".
We need to import Fluent and Vapor.
Let's create a final class to figure out how to represent our date, so type model and content,
We declare a variable schema = "learners", to give the reference of the table created earlier.
We add the reference values, telling fluent that the two properties correspond to the properties we declared in the migration.

We add an empty init and an init with the properties declared in the model, to make it get them from the migration.

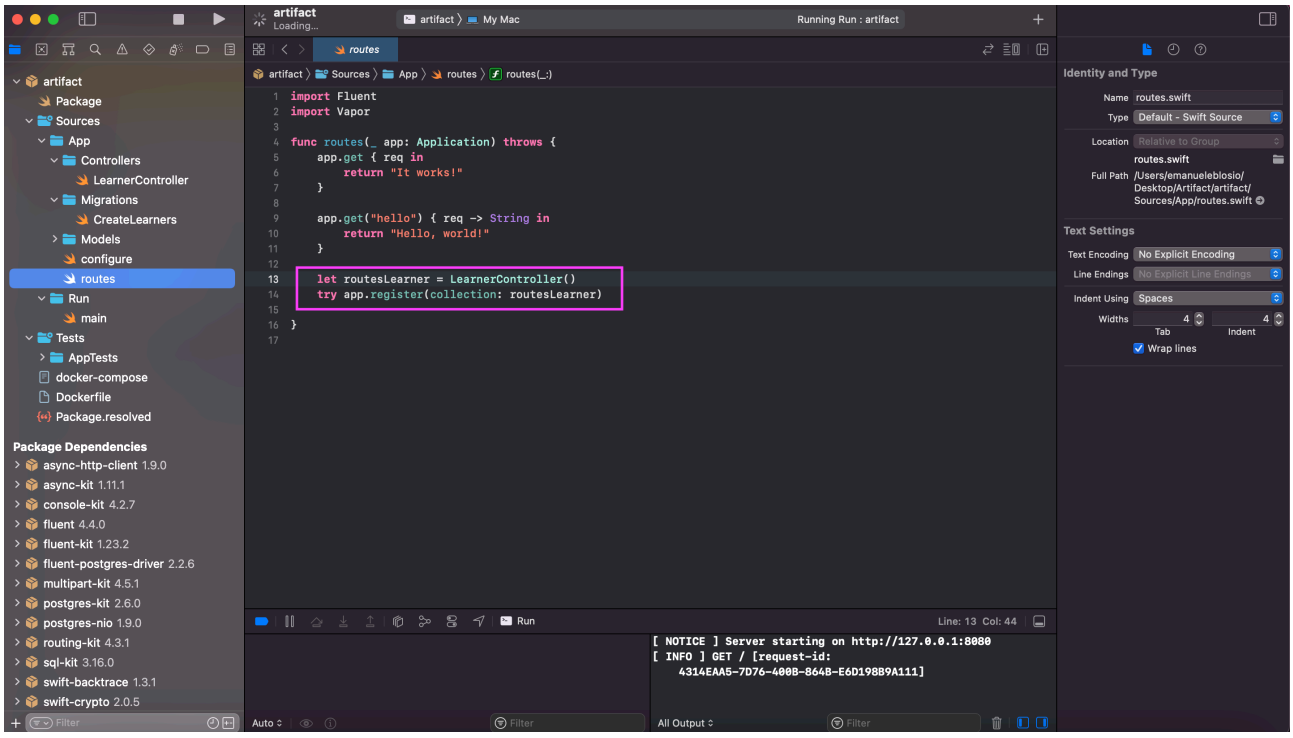


Now let's move on to the controller.
We create a struct called LearnerController of type RouteController, and import Fluent and Vapor. Now we wait for Xcode's "prompt" to add the protocol stubs, adding the boot function. In the boot function we declare a let to say that when it goes to the route with /learners, we will use the get function that will refer to the "index" function that we will create at this time.
The "index" function is the function that will return all the values from the model that we set up earlier thanks to rq.db, which is a request to return EVERYTHING that is in the database.

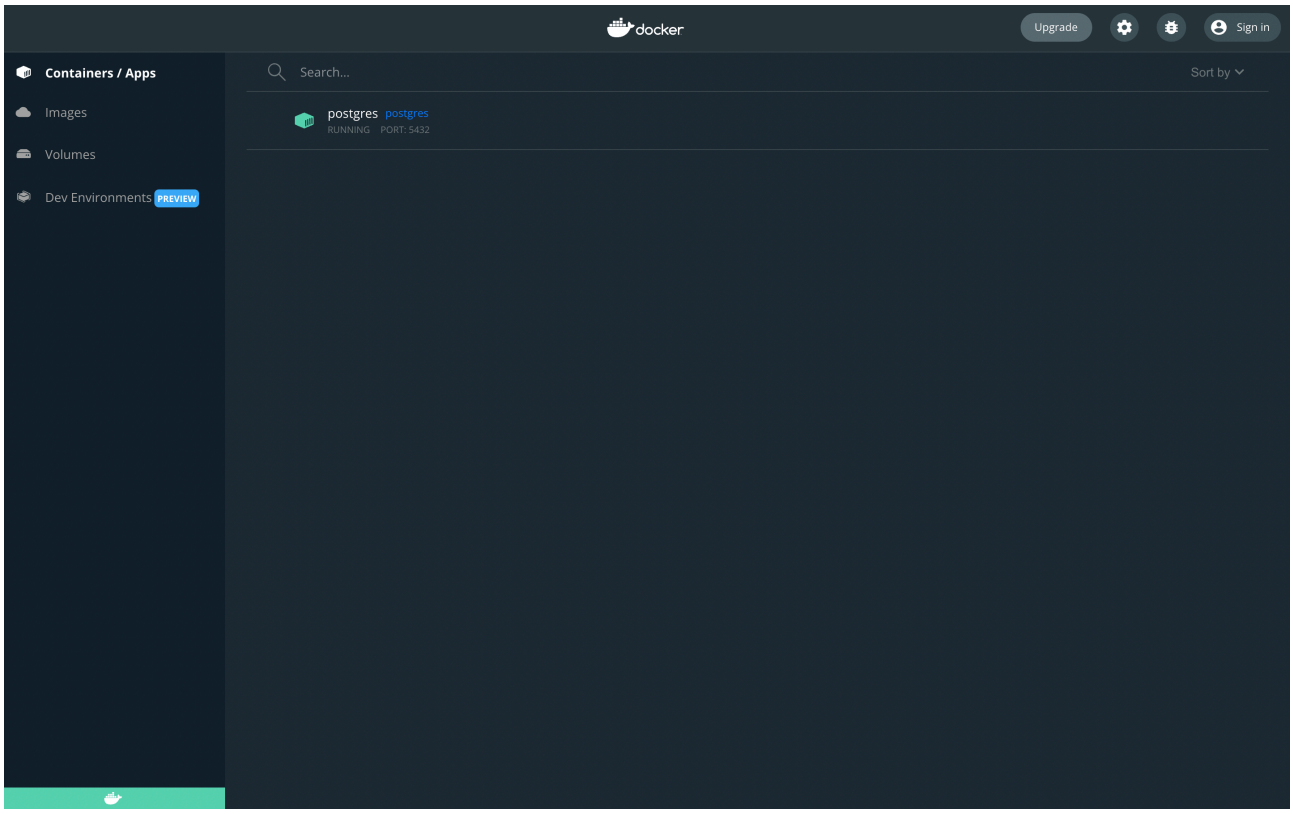


Now, we need to create the database. Let's go to configure.swift. Here are all the values of our database. Here, we need to add the migration that we created earlier. So, let's add the migration and execute it with a try in order to create a database.

Now, moving on to the routes, we need to add our LearnerController.

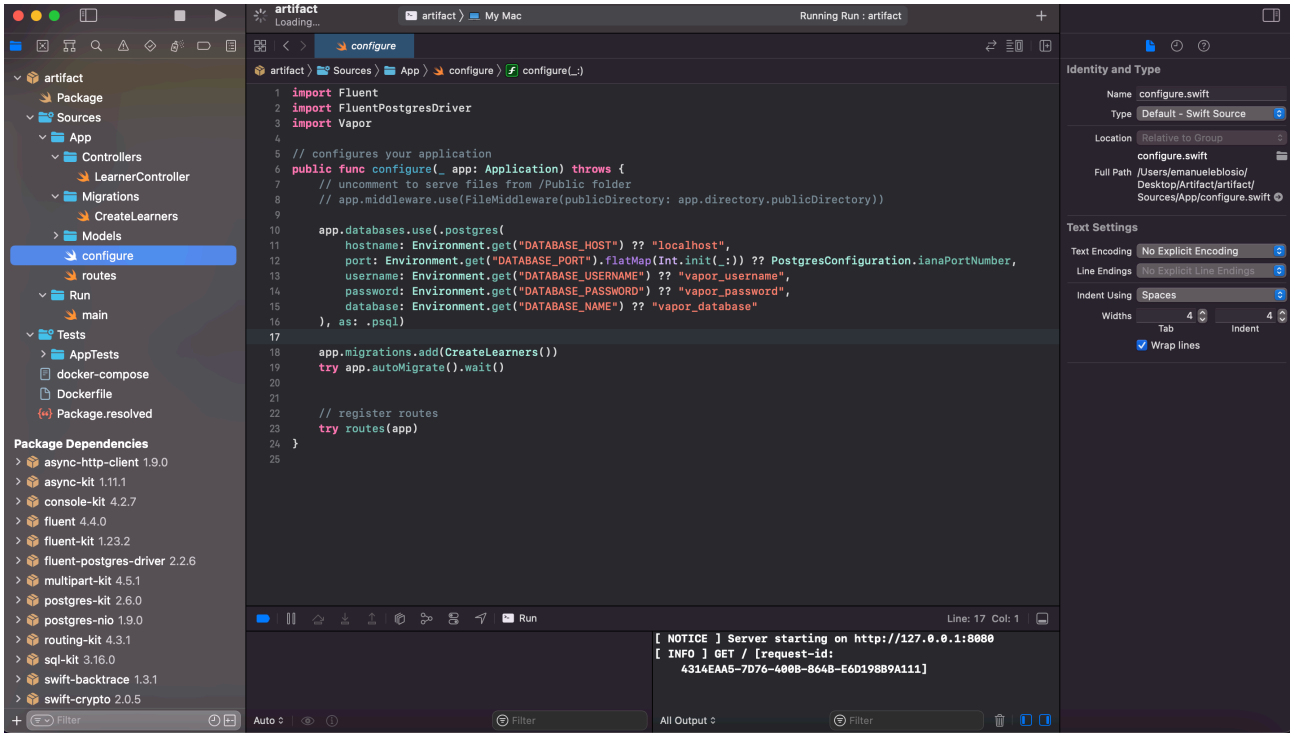


Now, if we run the project, our database will say "connection refused" because the database is not yet active. We stop the project.
We go to the terminal and run the command docker compose up db, it will try to start the database. If it gives you a Connection Refused, you need to open docker. Once we run docker, we run "docker compose up db" again and our database will run correctly. Opening docker desktop, we can see our database added to the available ones.

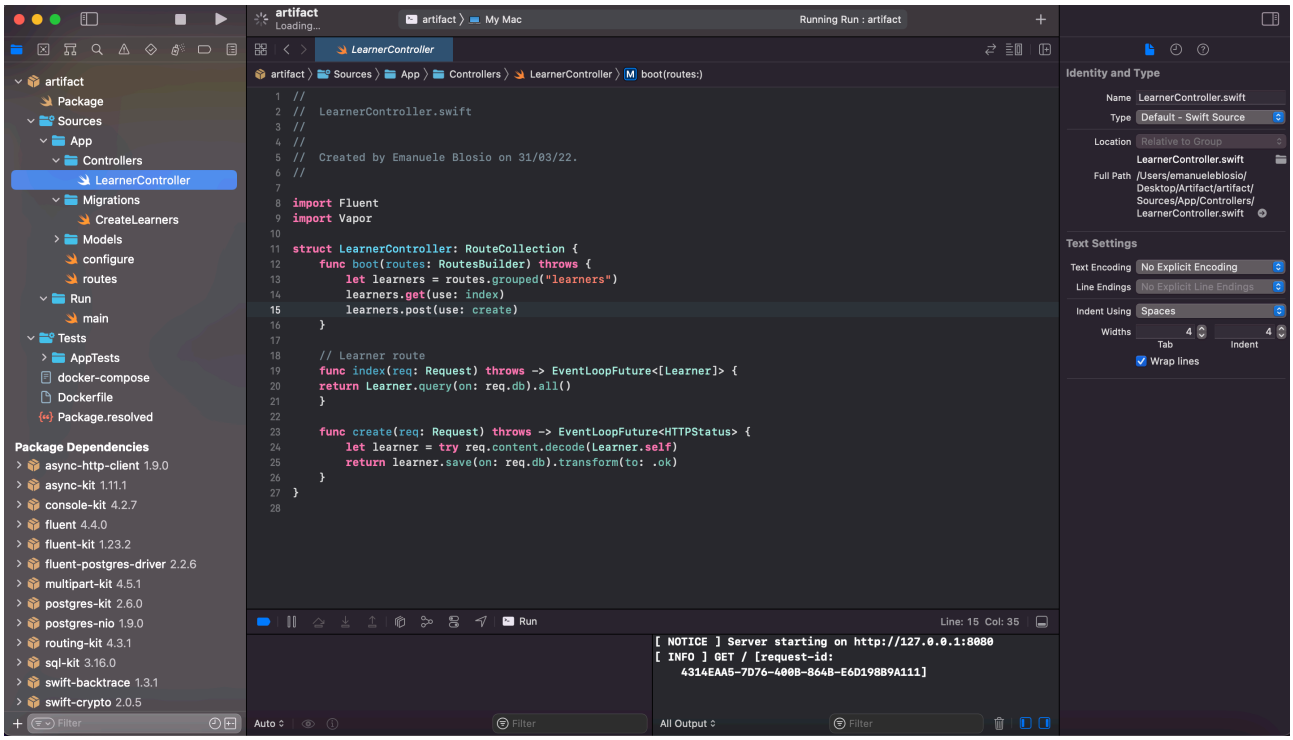


Now, if we run our project in Xcode, we will no longer have the "connection Refused".
The server will be running locally.
Now open Postman and in the get request, click send and you'll see that nothing is returned as we haven't added anything to our database yet.

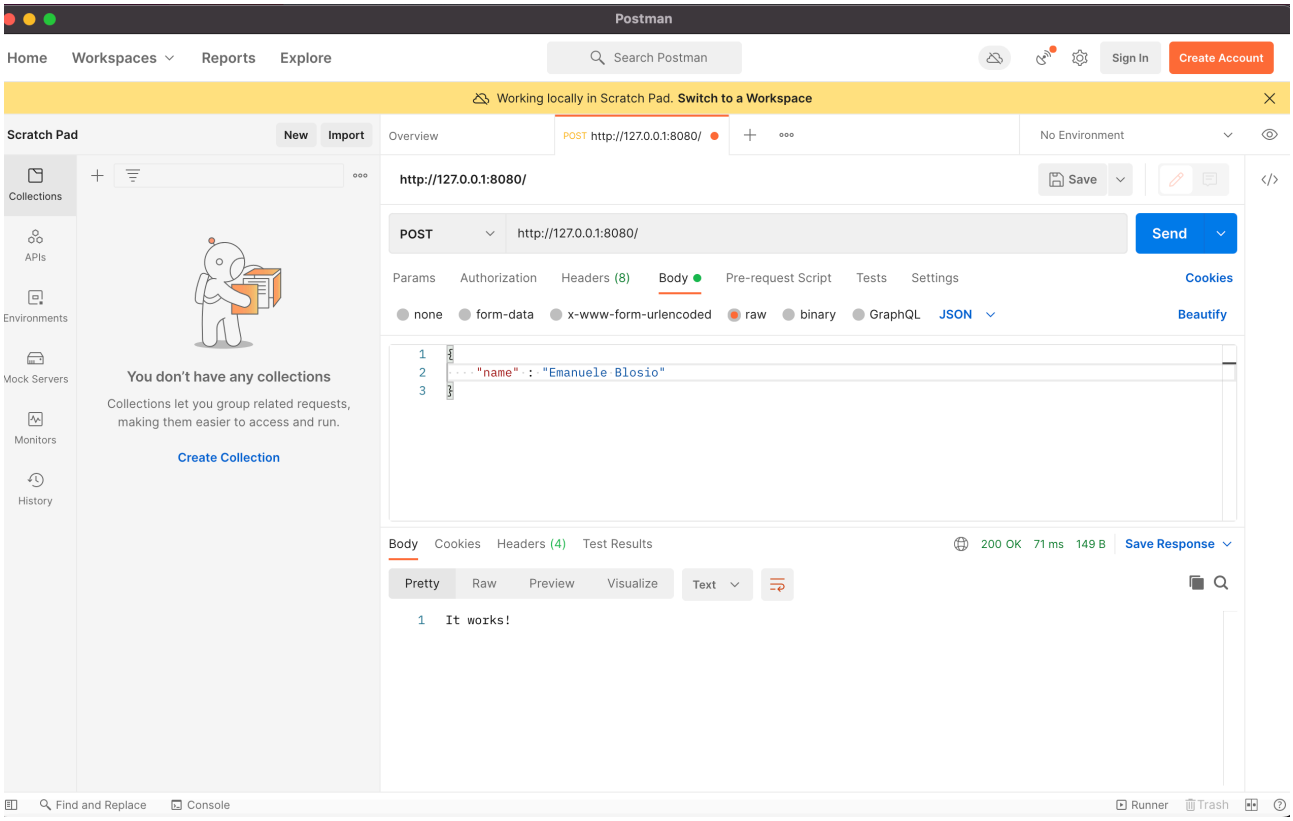
Now open DBeaver or Azure Data Studio and add a connection by setting all the required data. Remember that we use PostgreSQL and we give as our host name, username, password and database name the same as indicated in the Xcode file.



We will realise that we have not added a "post" function i.e. to write data to our database.
We need to create a new route, so let's first create a new function in our LearnerController and then add the command "learners.post" telling it to use the create function.



Now, to test the newly added "post" function, open postman, select "post" on the left, then select under body "Raw" and file type "JSON".
Now we simply need to add a new element based on our template, then write "name" : "learnerName" (the database will create the ID for us, we should not create it ourselves). Now we click on "POST", and our update will be sent to the Database.



If we now try to use the "GET" function, with the path http://127.0.0.1:8080/learners we will see that we will have as a result all the elements entered in the "POST" function.
We have created our first database!