Dockerize the Flask application

Let's populate the `Dockerfile` :
Dockerfile:

```
FROM python:3.6-slim-buster

WORKDIR /app

COPY requirements.txt ./

RUN pip install -r requirements.txt

COPY . .

EXPOSE 4000

CMD [ "flask", "run", "--host=0.0.0.0", "--port=4000"]
```

`FROM` sets the base image to use. In this case we are using the python 3.6 slim buster image
`WORKDIR` sets the working directory inside the image
`COPY requirements.txt ./` copies the requirements.txt file to the working directory
`RUN pip install -r requirements.txt` installs the requirements
`COPY . .` copies all the files in the current directory to the working directory
`EXPOSE 4000` exposes the port 4000
`CMD [ "flask", "run", "--host=0.0.0.0", "--port=4000"]` sets the command to run when the container starts

🐳🐳Docker compose

The term "Docker compose" might be a bit confusing because it's referred both to a file and to a set of CLI commands. Here we will use the term to refer to the file.
Populate the `docker-compose.yml` file:

```
version: "3.9"

services:
  flask_app:
    container_name: flask_app
    image: dockerhub-flask_live_app:1.0.0
    build: .
    ports:
      - "4000:4000"
    environment:
      - DB_URL=postgresql://postgres:postgres@flask_db:5432/postgres
    depends_on:
      - flask_db
  flask_db:
    container_name: flask_db
    image: postgres:12
    ports:
```

```
    - "5432:5432"
  environment:
    - POSTGRES_PASSWORD=postgres
    - POSTGRES_USER=postgres
    - POSTGRES_DB=postgres
  volumes:
    - pgdata:/var/lib/postgresql/data

volumes:
  pgdata: {}
```

We just defined 2 services: `flask_app` and `flask_db`
`flask_app` is the Flask application we just dockerized
`flask_db` is a Postgres container, to store the data. We will use the official Postgres image
Explanation:
`version` is the version of the docker-compose file. We are using the verwion 3.9
`services` is the list of services (containers) we want to run. In this case, we have 2 services: flask_app and flask_db
`container_name` is the name of the container. It's not mandatory, but it's a good practice to have a name for the container. Containers find each other by their name, so it's important to have a name for the containers we want to communicate with.
`image` is the name of the image we want to use. I recommend replacing "dockerhub-" with YOUR Dockerhub account (it's free).
`build` is the path to the Dockerfile. In this case, it's the current directory, so we are using `.`
`ports` is the list of ports we want to expose. In this case, we are exposing the port 4000 of the flask_app container, and the port 5432 of the flask_db container. The format is `host_port:container_port`
`depends_on` is the list of services we want to start before this one. In this case, we want to start the flask_db container before the flask_app container
`environment` is to define the environment variables. for the flask_app, we will have a database url to configure the configuration. For the flask_db container, we will have the environment variables we have to define when we wan to use the Postgres container (we can't change the keys here, because we are using the Postgres image, defined by the Postgres team).
`volumes` in the flask_db defines a named volume we will use for persistency. Containers are ephimerals by definition, so we need this additional feature to make our data persist when the container will be removed (a container is just a process).
`volumes` at the end of the file is the list of volumes we want to create. In this case, we are creating a volume called `pgdata`. The format is `volume_name: {}`

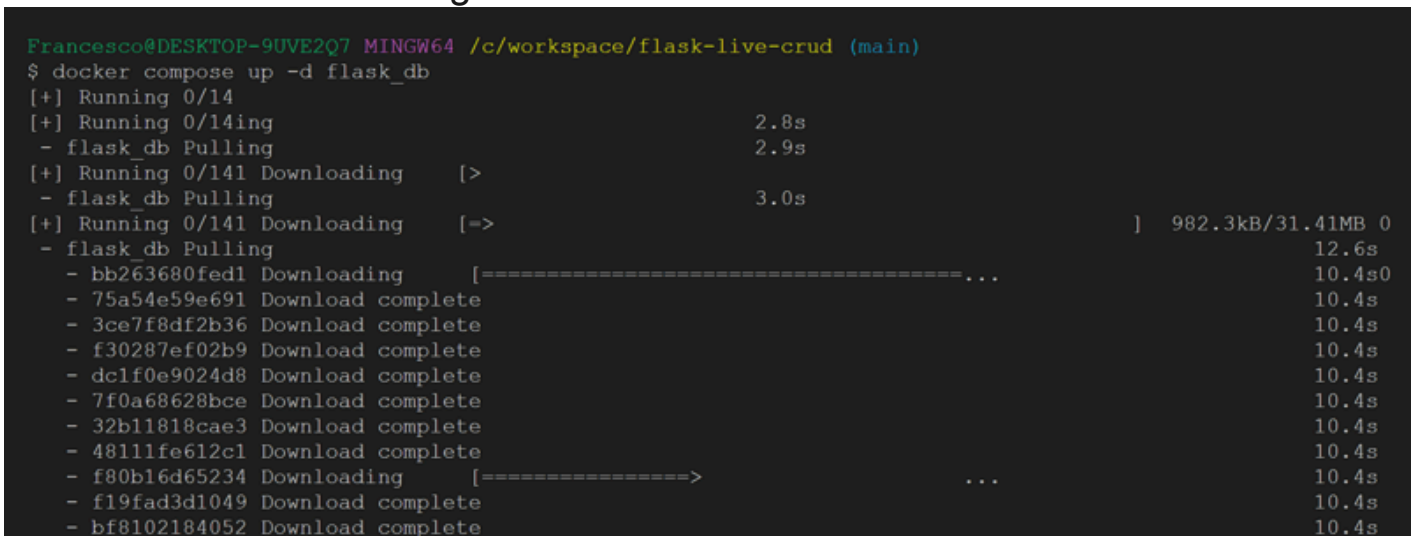
👟 Run the Postgres container and test it with TablePlus

To run the Postgres container, type:

docker compose up -d flask_db

The `-d` flag is to run the container in detached mode, so it will run in the background.

You should see something like this:



Docker is pulling (downloading) the Postgres image on our local machine and it's running a container based on that Postgres image.

To check if the container is running, type:

docker compose logs

If everything is ok, you should see something like this:

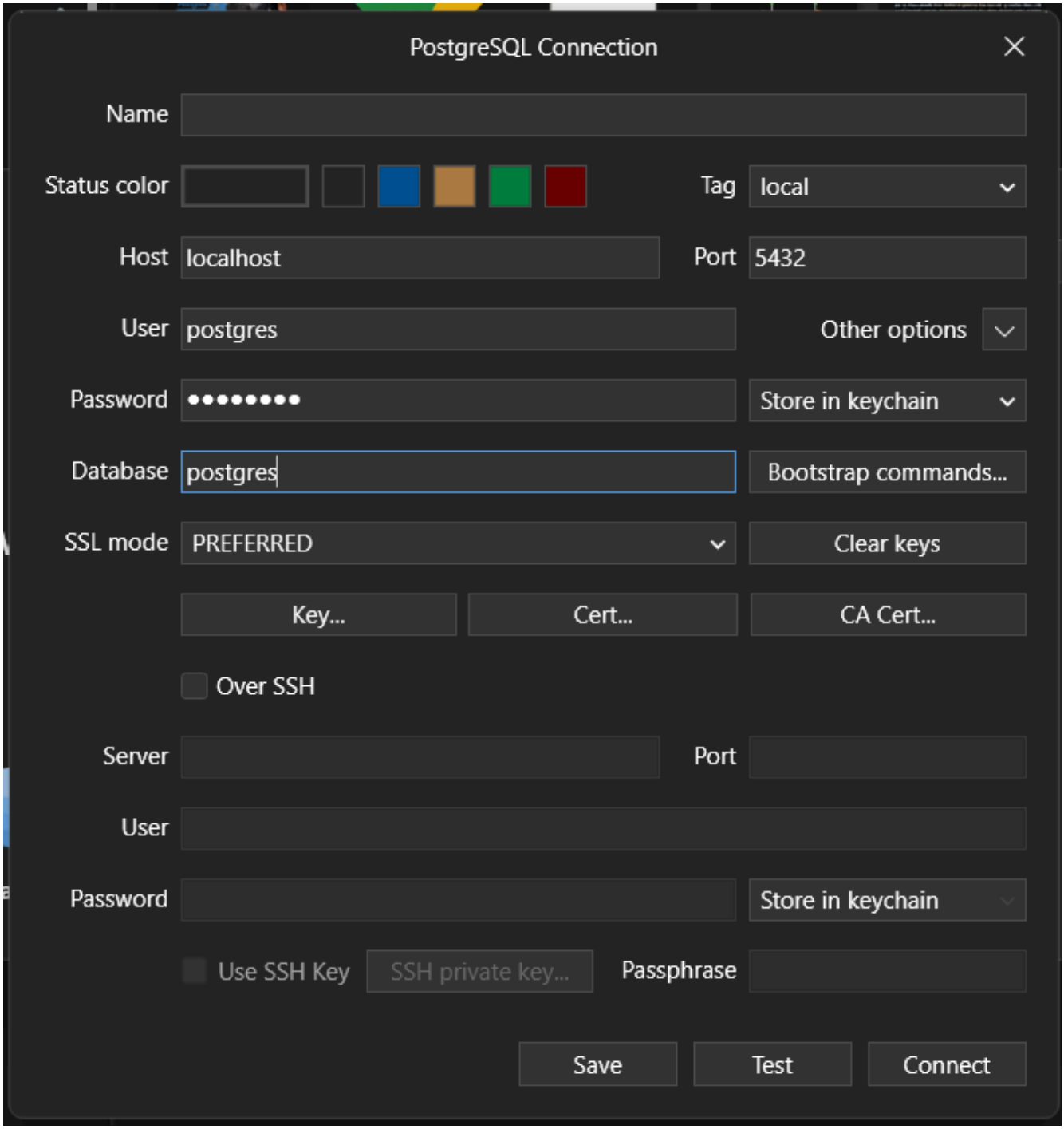
```
Francesco@DESKTOP-9UVE2Q7 MINGW64 /c/workspace/flask-live-crud (main)
$ docker compose logs
flask_db  | The files belonging to this database system will be owned by user "postgres".
flask_db  | This user must also own the server process.
flask_db  |
flask_db  | The database cluster will be initialized with locale "en_US.utf8".
flask_db  | The default database encoding has accordingly been set to "UTF8".
flask_db  | The default text search configuration will be set to "english".
flask_db  |
flask_db  | Data page checksums are disabled.
flask_db  |
flask_db  | fixing permissions on existing directory /var/lib/postgresql/data ... ok
flask_db  | creating subdirectories ... ok
flask_db  | selecting dynamic shared memory implementation ... posix
flask_db  | selecting default max_connections ... 100
flask_db  | selecting default shared_buffers ... 128MB
flask_db  | selecting default time zone ... Etc/UTC
flask_db  | creating configuration files ... ok
flask_db  | running bootstrap script ... ok
flask_db  | performing post-bootstrap initialization ... ok
flask_db  | syncing data to disk ... ok
flask_db  |
flask_db  |
flask_db  | Success. You can now start the database server using:
flask_db  |
flask_db  |     pg_ctl -D /var/lib/postgresql/data -l logfile start
flask_db  |
flask_db  | initdb: warning: enabling "trust" authentication for local connections
flask_db  | You can change this by editing pg_hba.conf or using the option -A, or
flask_db  | --auth-local and --auth-host, the next time you run initdb.
flask_db  | waiting for server to start....2023-02-18 18:07:38.522 UTC [48] LOG:  starting PostgreSQL 12.14 (Debian 12
flask_db  | 2023-02-18 18:07:38.524 UTC [48] LOG:  listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
flask_db  | 2023-02-18 18:07:38.540 UTC [49] LOG:  database system was shut down at 2023-02-18 18:07:38 UTC
flask_db  | 2023-02-18 18:07:38.545 UTC [48] LOG:  database system is ready to accept connections
flask_db  |  done
flask_db  | server started
flask_db  |
flask_db  | /usr/local/bin/docker-entrypoint.sh: ignoring /docker-entrypoint-initdb.d/*
flask_db  | 2023-02-18 18:07:38.650 UTC [48] LOG:  background worker "logical replication launcher" (PID 55) exited wi
flask_db  | 2023-02-18 18:07:38.650 UTC [50] LOG:  shutting down
flask_db  | 2023-02-18 18:07:38.671 UTC [48] LOG:  database system is shut down
flask_db  |  done
flask_db  | server stopped
flask_db  |
flask_db  | PostgreSQL init process complete; ready for start up.
flask_db  |
flask_db  | 2023-02-18 18:07:38.757 UTC [1] LOG:  starting PostgreSQL 12.14 (Debian 12.14-1.pgdg110+1) on x86_64-pc-li
flask_db  | 2023-02-18 18:07:38.757 UTC [1] LOG:  listening on IPv4 address "0.0.0.0", port 5432
flask_db  | 2023-02-18 18:07:38.757 UTC [1] LOG:  listening on IPv6 address "::", port 5432
flask_db  | 2023-02-18 18:07:38.767 UTC [1] LOG:  listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
flask_db  | 2023-02-18 18:07:38.781 UTC [68] LOG:  database system was shut down at 2023-02-18 18:07:38 UTC
flask_db  | 2023-02-18 18:07:38.786 UTC [1] LOG:  database system is ready to accept connections
```

If the last line is LOG: database system is ready to accept connections , it means that the container is running and the Postgres server is ready to accept connections.
But to be sure, let's make another test.
To show all the containers (running and stopped ones) type:
docker ps -a
The output should be similar to this:

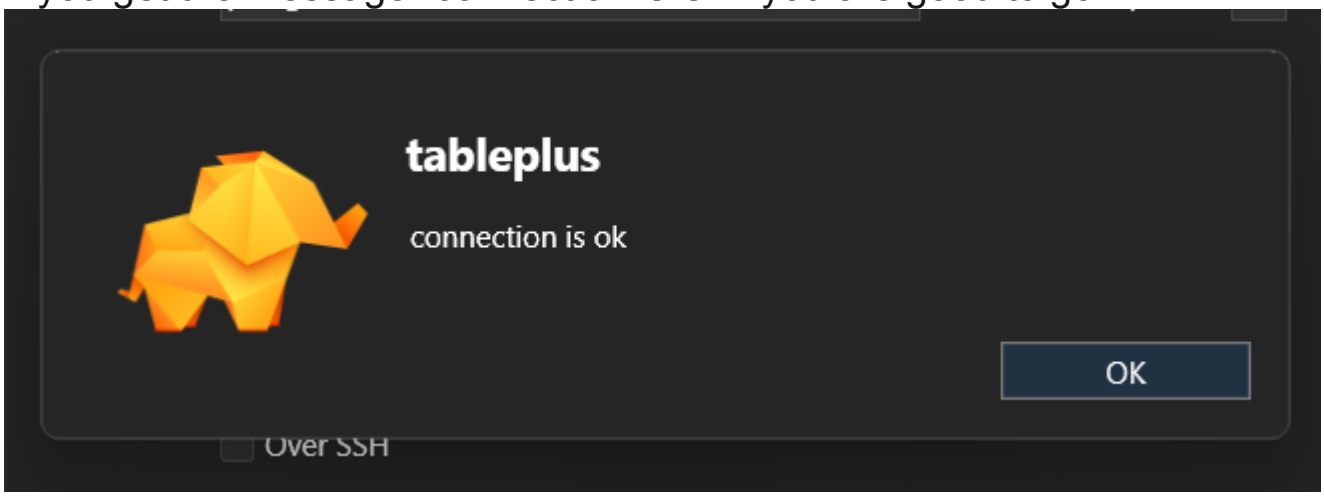
```
Francesco@DESKTOP-9UVE2Q7 MINGW64 /c/workspace/flask-live-crud (main)
$ docker ps -a
CONTAINER ID   IMAGE          COMMAND                  CREATED         STATUS         PORTS                    NAMES
76c515acf924   postgres:12    "docker-entrypoint.s…"   2 minutes ago   Up 2 minutes   0.0.0.0:5432->5432/tcp   flask_db

Francesco@DESKTOP-9UVE2Q7 MINGW64 /c/workspace/flask-live-crud (main)
$
```

Now, to test the db connection, we can use any tool we want. Personally, I use TablePlus.
Use the following configuration:
Host: localhost
Port: 5432
User: postgres
Password: postgres

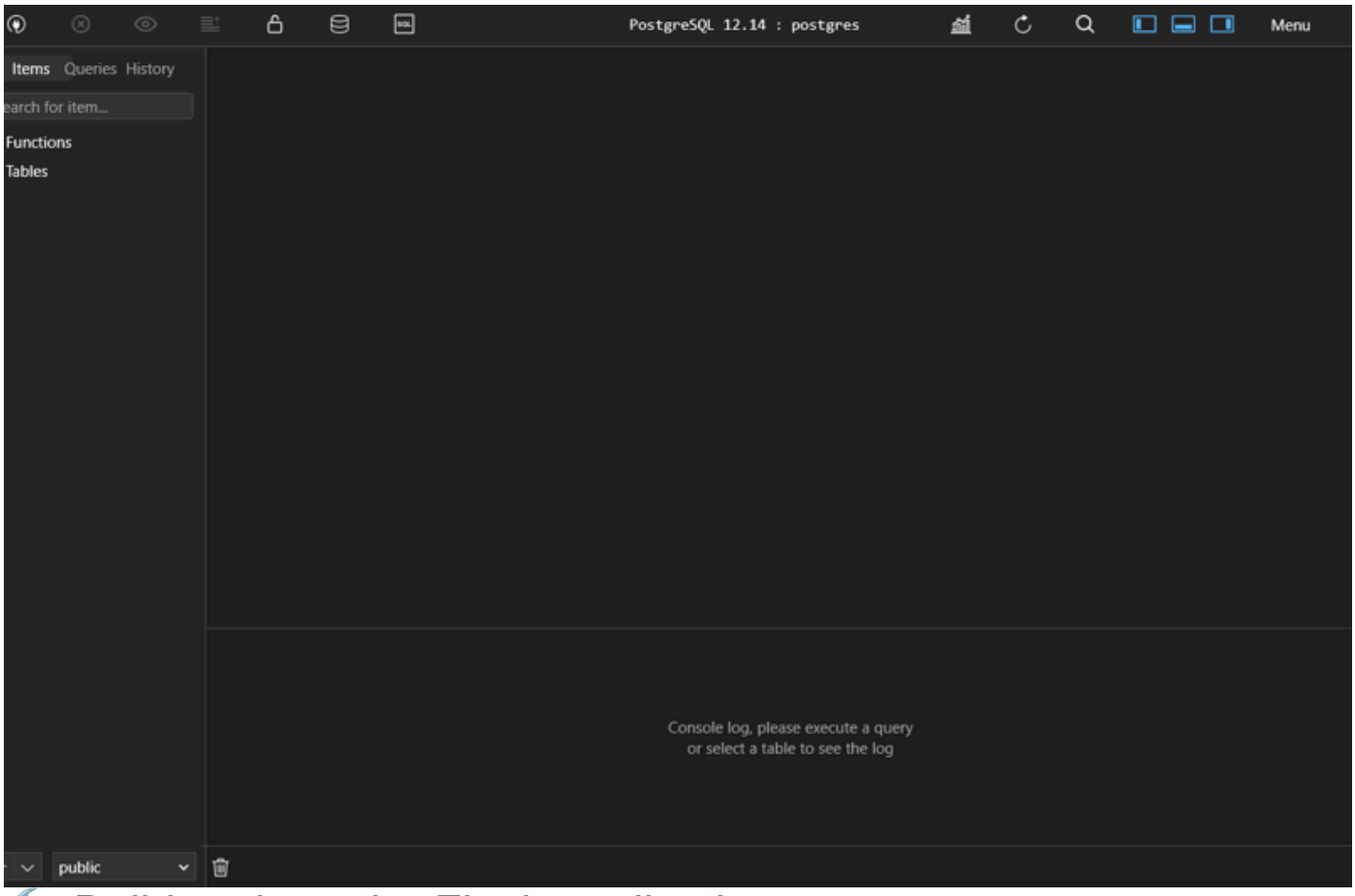
Database: postgres



Then hit "Test" (at the bottom-right).
If you get the message "connection is OK" you are good to go.



You can also click "Connect" and you will see an empty database. This is correct.



🔨 Build and run the Flask application

Now, let's build and run the Flask application.
Let's go back to the folder where the `docker-compose.yml` is located and type:

```
docker compose build
```

This should BUILD the flask_app image, with the name defined in the "image" value. In my case it's francescoxx/flask_live_app:1.0.0 because that's my Dockerhub username. You should replace "francescoxx" with your Dockerhub username.
You can also see all the steps docker did to build the image, layer by layer. You might recognize some of them, because we defined them in the Dockerfile.

```
Francesco@DESKTOP-9UVE2Q7 MINGW64 /c/workspace/flask-live-crud (main)
$ docker compose build
[+] Building 1.5s (11/11) FINISHED
 => [internal] load build definition from Dockerfile
 => => transferring dockerfile: 234B
 => [internal] load .dockerignore
 => => transferring context: 2B
 => [internal] load metadata for docker.io/library/python:3.6-slim-buster
 => [auth] library/python:pull token for registry-1.docker.io
 => [1/5] FROM docker.io/library/python:3.6-slim-buster@sha256:e10aa83604948c6d8d9f72a9a20193d84bb2c
 => [internal] load build context
 => => transferring context: 3.28kB
 => CACHED [2/5] WORKDIR /app
 => CACHED [3/5] COPY requirements.txt ./
 => CACHED [4/5] RUN pip install -r requirements.txt
 => [5/5] COPY . .
 => exporting to image
 => => exporting layers
 => => writing image sha256:9170e8be4524bd327cb956dc6878efc31ce07f2b5b3bca9cce51da451f1f1432
 => => naming to docker.io/francescoxx/flask_live_app:1.0.0

Francesco@DESKTOP-9UVE2Q7 MINGW64 /c/workspace/flask-live-crud (main)
$
```

Now, to check if the image has been built successfully, type:

docker images

We should see a similar result, with the image we just built:

```
Francesco@DESKTOP-9UVE2Q7 MINGW64 /c/workspace/flask-live-crud (main)
$ docker images
REPOSITORY                  TAG       IMAGE ID       CREATED              SIZE
francescoxx/flask_live_app  1.0.0     9170e8be4524   About a minute ago   150MB
postgres                    12        2c278af658a7   7 days ago           373MB

Francesco@DESKTOP-9UVE2Q7 MINGW64 /c/workspace/flask-live-crud (main)
$
```

Run the flask_app service

We are almost done, but one last step is to run a container based on the image we just built.

To do that, we can just type:

docker compose up flask_app

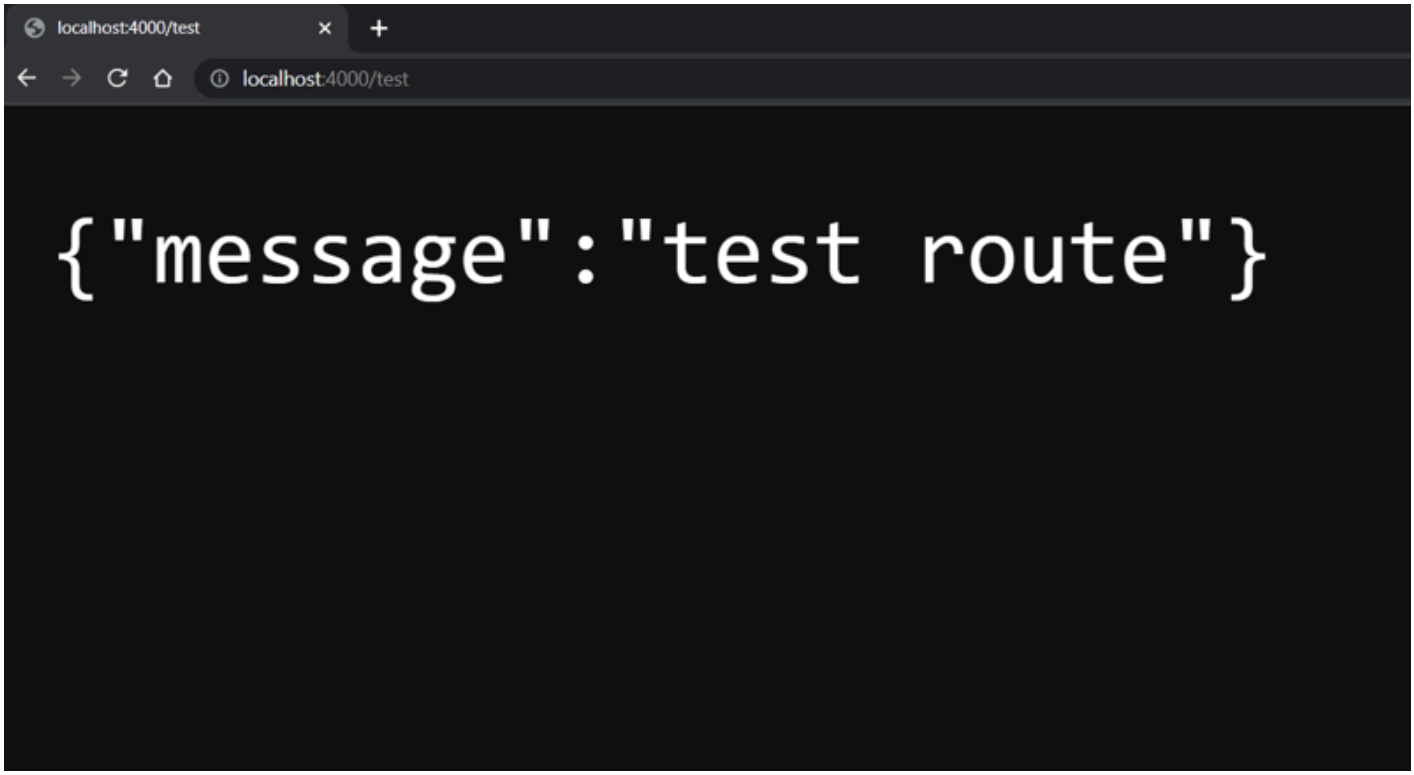
In this case we don't use the -d flag, because we want to see the logs in the terminal.

We should see something like this:

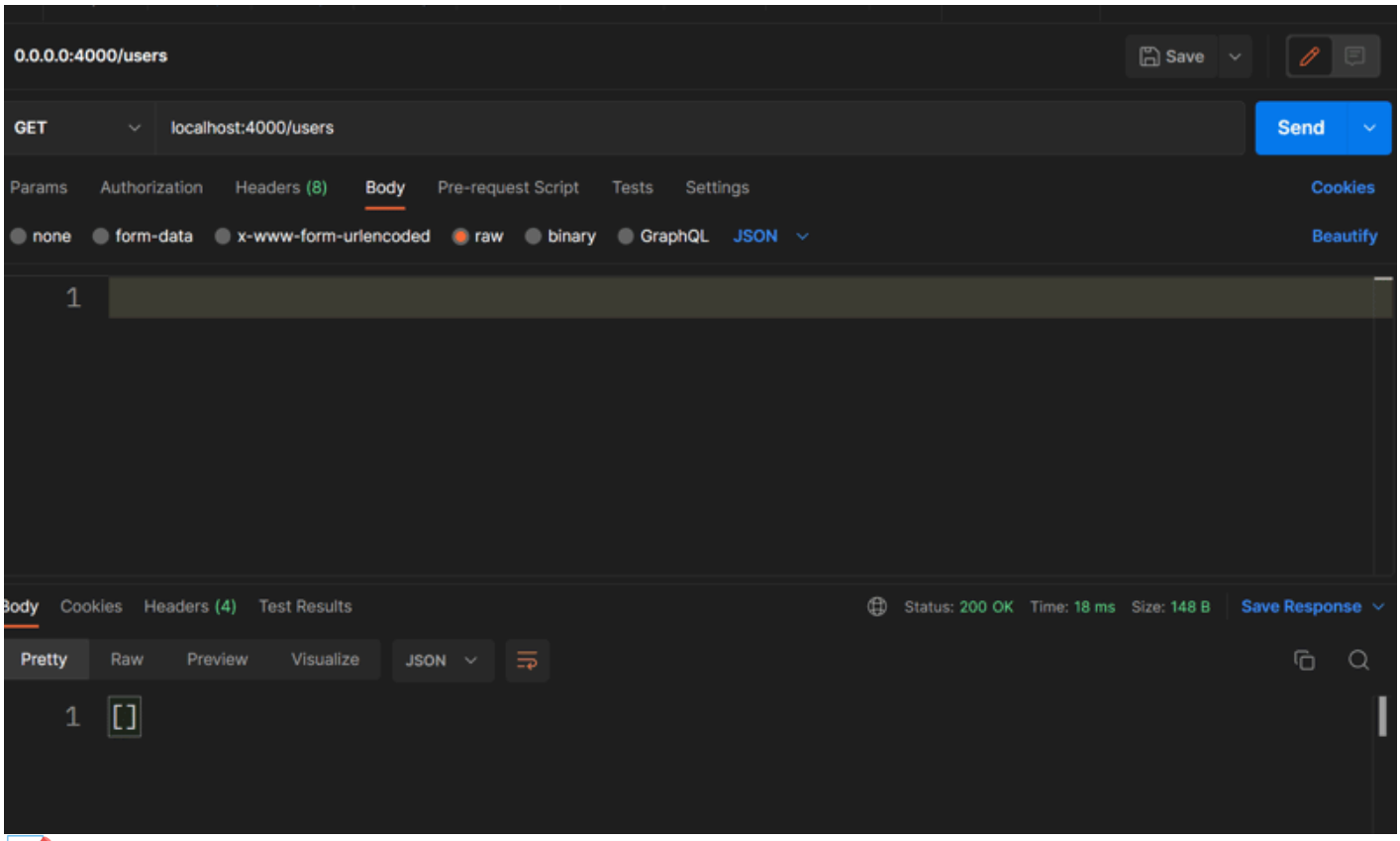
```
Francesco@DESKTOP-9UVE2Q7 MINGW64 /c/workspace/flask-live-crud (main)
$ docker compose up
[+] Running 2/0
 - Container flask_db   Running
 - Container flask_app  Created
Attaching to flask_app, flask_db
flask_app  |  * Environment: production
flask_app  |    WARNING: This is a development server. Do not use it in a production de
flask_app  |    Use a production WSGI server instead.
flask_app  |  * Debug mode: off
flask_app  | /usr/local/lib/python3.6/site-packages/flask_sqlalchemy/__init__.py:873:
adds significant overhead and will be disabled by default in the future.  Set it to Tru
flask_app  |   'SQLALCHEMY_TRACK_MODIFICATIONS adds significant overhead and '
flask_app  |  * Running on all addresses.
flask_app  |    WARNING: This is a development server. Do not use it in a production de
flask_app  |  * Running on http://172.26.0.3:4000/ (Press CTRL+C to quit)
```

🔍 Test the application

Let's test our application. First of all, let's just go to any browser and visit `localhost:4000/test`
You shoulds see this result:

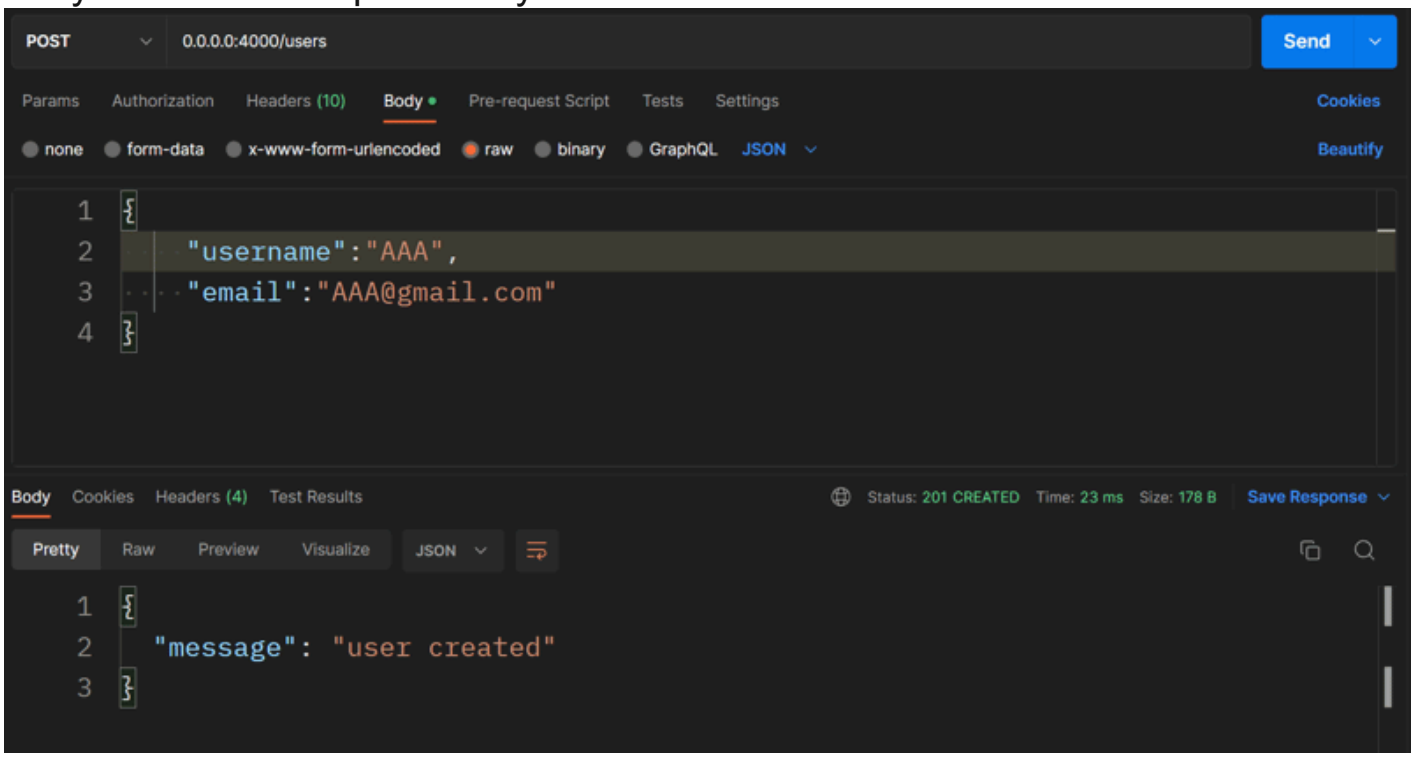


(note that if you visit `localhost:4000` you get an error because there is no route associated to this endpoint, but by getting an error is a good thing, because it means that the server is running!)
Now it's time to test all the endpints using Postman. Feel free to use any tool you want.
If we make a GET request to `localhost:4000/users` we will get an empty array. This is correct

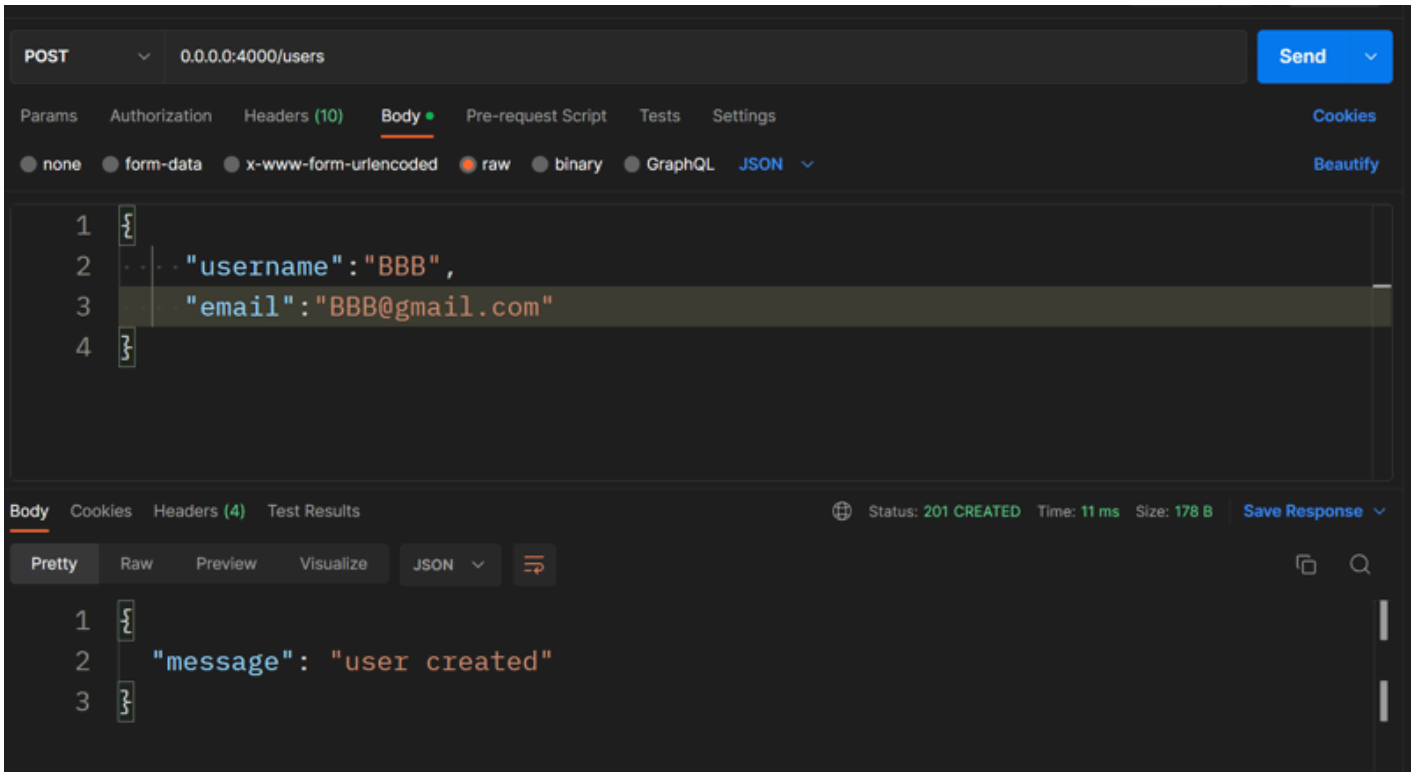


📝 Create a user

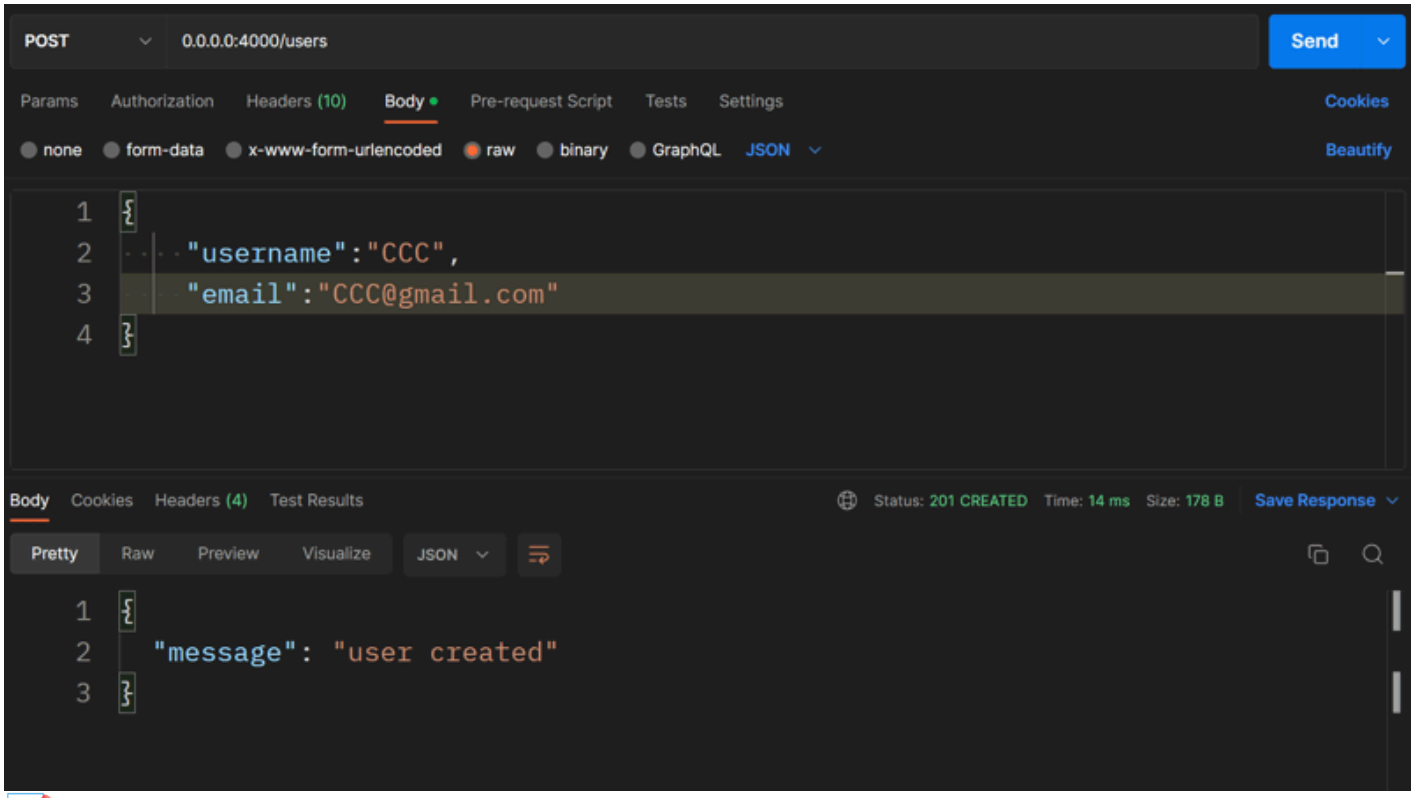
Now let's create a user, making a POST request to `localhost:4000/users` with the body below as a request body:



Let's create another one:

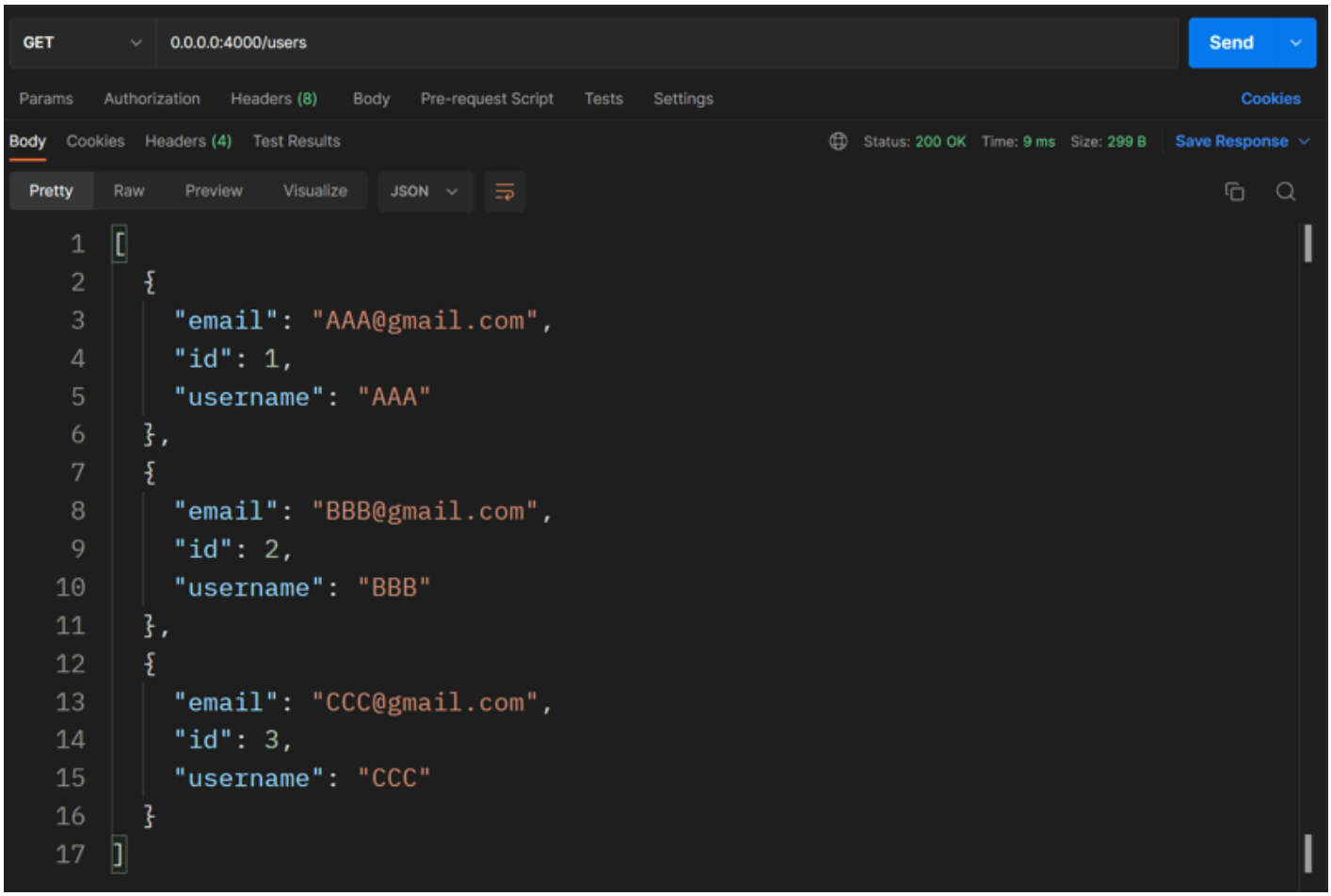


One more:



📝 Get all users

Now, let's make a GET request to `localhost:4000/users` to get all the users:

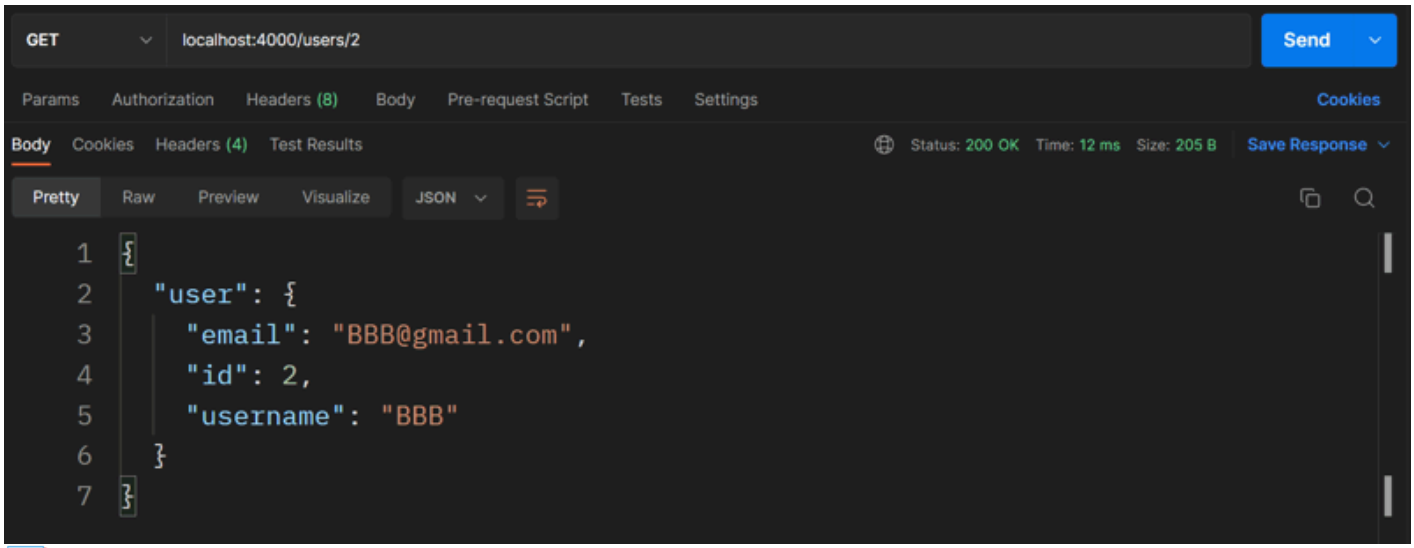


We just created 3 users.

📝 Get a specific user

If you want to get a specific user, you can make a GET request to localhost:4000/users/<user_id> .

For example, to get the user with id 2, you can make a GET request to localhost:4000/users/2



📝 Update a user

If you want to update a user, you can make a PUT request to localhost:4000/users/<user_id> .

For example, to update the user with id 2, you can make a PUT request to `localhost:4000/users/2` with the body below as a request body: