Realized by :

Fahmi Chaabane

Table of Contents

GENERAL INTRODUCTION:

The idea behind the project is to try several approaches of decomposing a whole application into multiple microservices, each one of these microservices is independent and focuses on a particular task, whether it is an interaction with the database or it is a business logic to be fulfilled.

The technologies used in this project did influence the way of approaching the result. Nestjs framework specifically offers a lot of generalized features that enables and facilitates the communication between microservices through what is called “transport layers”. We will go through each one of those features later.

Beside the decomposition of the application, we did also follow the best practices of delivering a ready running application through implementing a CI/CD pipeline for each of our microservices. We will go through each one of the pipeline stages later.

ARCHITECTURE:

In the description of this chapter, we will be using the microservices names to demonstrate how the communication is established between them.

Basically, the architecture starts from an NGINX web server “nginx-reverse-proxy” that is going to act at the same time as a reverse proxy and a load balancer. The NGINX instance will communicate to the “graphql-gateway” service that is the entrypoint to our application. This particular microservice has no direct access to the database, however it does communicate to three others services which are the “user-microservice”, the “movie-microservice” and the “anime-microservice”, these services are the only responsible for every read/write interaction with the database.

Each of the microservices are implemented by NestJS. Thanks to NestJS, it offers several built-in implementations of transport layers:

- The “nginx-reverse-proxy” communicates to the “graphql-microservice” through the HTTP transport layer.
- The “graphql-gateway” communicates to the “user-microservice” through the TCP layer.
- The “graphql-gateway” communicates to the “movie-microservice” and the “anime-microservice” via a RabbitMQ instance through the RMQ transport layer.

Thanks to the maturity of the containerization of applications, rather than installing each of the required services like NodeJS and RabbitMQ for example on our machines, we simply dockerized

them, meaning each one of the mentioned services are running within its own docker container, All of them are running within the same bridge docker network Having all of them in the same docker network, meaning that none of the containers has to expose an application TCP port to be seen by other containers, the container name should be enough, thanks to the build-in discovery service that docker daemon has. (except the nginx instance because it is the only service that will have an external communication, a browser in our test case, so it will expose a port. We will cover that part later).
This diagram should describe everything mentioned above:

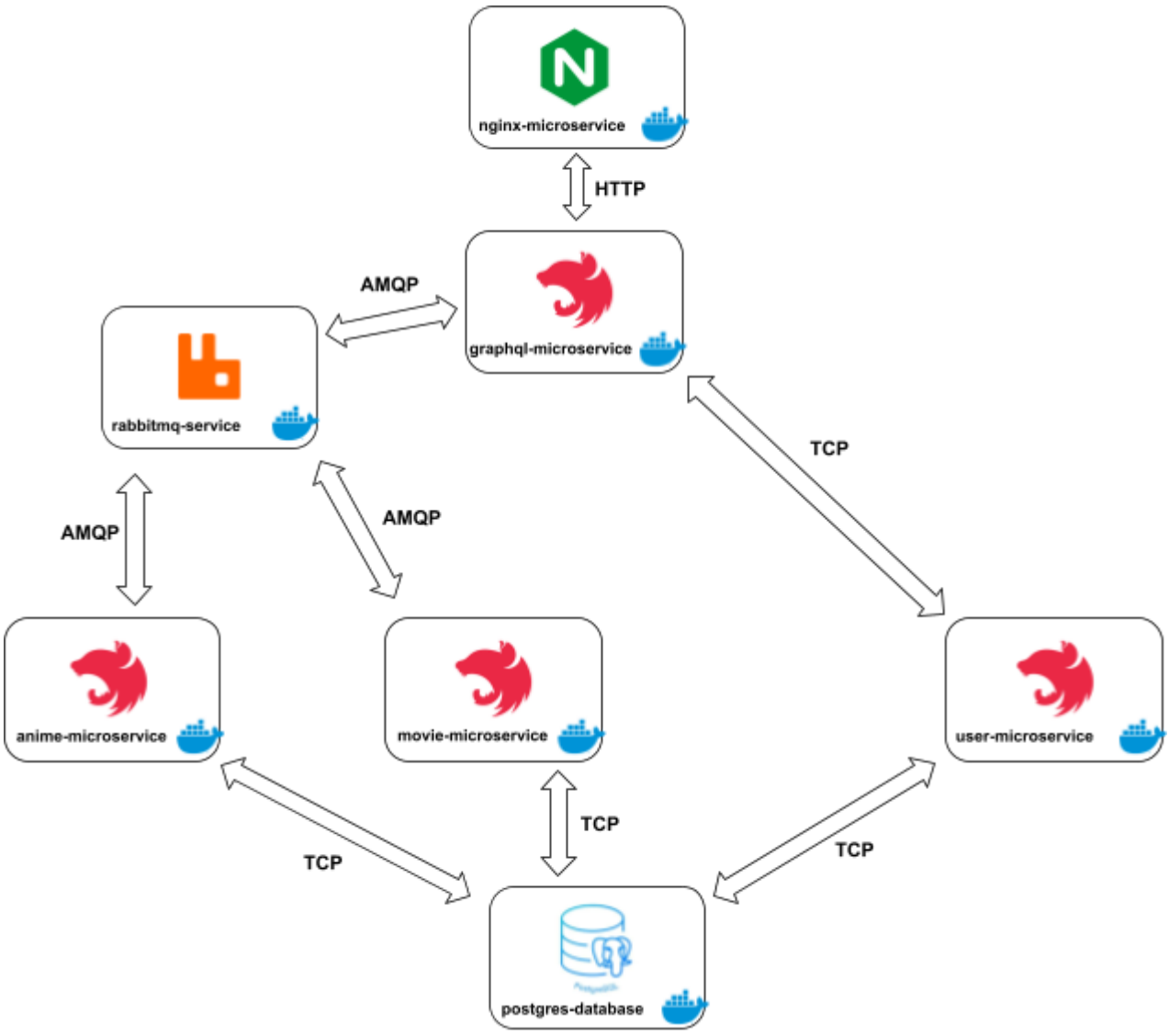

Figure 1 : Project Architecture

IMPLEMENTATION AND TECHNOLOGIES:

In this project, for the business logic, we chose to make it simple and implement a small CRUD operation application. This class diagram describes the design behind it:

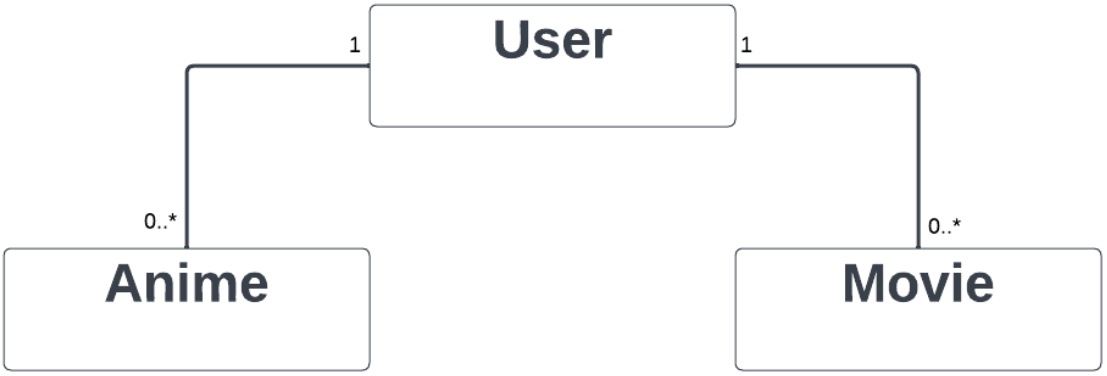

Figure 2 : Class diagram

To implement that, we chose to use several technologies, not only because they are trendy, but also because they provide flexibility and multiple best practices.

Figure 3 : NGINX

NGINX: is a web server that can also be used as a reverse proxy, load balancer, and other cases. It is a free and open-source software. In our case, NGINX plays the role of a portal to our application by splitting out its 80 port to forward (and load balance) the incoming HTTP requests from the outside world to our application. To demonstrate the load balancing process, the graphql-gateway is set to run on two instances (2 docker containers), and because they are within the same docker network, we are able to just mention the instances' names inside the nginx configuration files, to prove that, we did implement an interceptor (which is a built-in feature in NestJS) to log the IP address of the triggered instance to the console. [R1]

Figure 4 : NestJS

NestJS: is a framework for building Node.js server-side applications. It fully supports Typescript. Nest provides an out-of-the-box application architecture which allows developers to create highly testable, scalable, loosely coupled, and easily maintainable applications. The architecture is heavily inspired by Angular. [R2]

Our project contains 4 NestJS applications :

- graphql-gateway: is the entry-point of our application, it receives forwarded requests from the reverse proxy and depending on the type of request, it performs the right business logic. As the name suggests, this microservices contains a graphql server, it will play the role of the orchestrer for all HTTP requests and responses, meaning every request and response will have to go through this microservice.

  NestJS has a graphql module that is built on top of apollo-server. What's the advantages of that ? NestJS empowers the classic apollo-server-express with the ability to catch incoming requests before going through its corresponding resolver, and that enables us to refactor the process of authorization within the guards first.

  We also choose to implement the authorization process only within this microservice (it was implemented using the Passport.js library [Fig-12]), meaning, instead of implementing the process of authorization within each microservice, we only do it one time inside of the entry-point.
- user-microservice: This is where we define the user entity, its business logic functionalities and configure the database connection, which is done thanks to NestJS module TypeORM. [R3]

Figure 5 : TypeORM for database connection

- movie-microservice: similar to the user-microservice in terms of defining the movie entity, its business logic functionalities, the database connection, and also subscribing to the movie_queue within the broker instance.
- Anime-microservice: almost as similar as the movie-microservice.

Figure 6 : Apollo-Server

Apollo-Server: is an open-source GraphQL server, it is the best way to build a production-ready and self documenting GraphQL API that can use data from any source. [R4]

Thanks to the graphql module of NestJS, we are able to enhance the classical way of defining our gql schema and providers files, through the code-first approach, meaning we write TypeScript code to define everything we need that includes the queries type, mutations types and gql schemas. And because of the decomposition design of our application we were able to use the field resolver feature of the graphql specification to query the other microservices in order to fulfill the intended query. For example, in our case, a user has a list of movies, so whenever we query the user and its corresponding list of movies, we first request the user-microservice in the main query function implementation and then within the field resolver of movies, we query the movie-microservice. [R5]

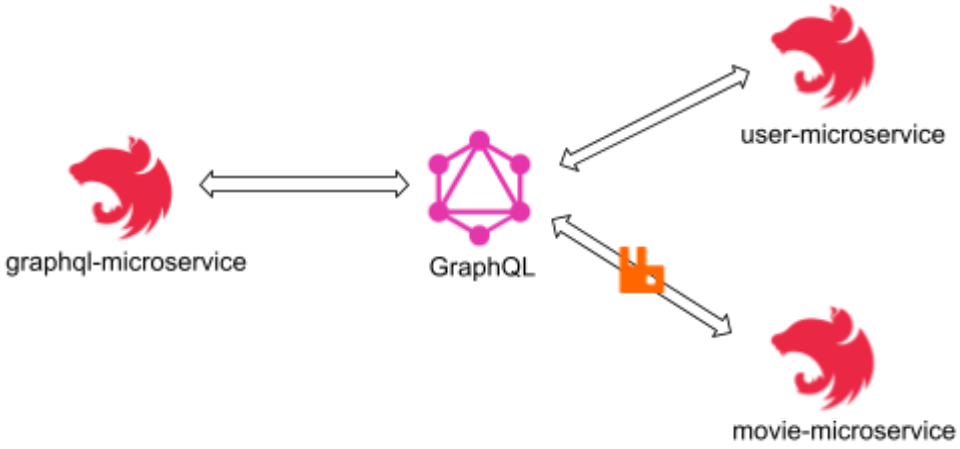

Figure 7 : GraphQL for querying data

Figure 8 : Passport.js

Passport.js: is authentication middleware for Node.js applications. Extremely flexible and modular. It has several sets of strategies. We used in our application the JWT strategy that basically acts as a middleware to verify the token within the request, if the token is invalid, an exception would be thrown back to the client. [R6]

Figure 9 : Passport as an authentication middleware

Figure 10 : Jest

Jest: is a JavaScript testing framework maintained by Meta.

Because testing your application has become the standard nowadays in the software development process, we have implemented for each of our NestJS microservices unit-tests, which makes a total number of 123 valid tests. [R7]

All files

71.34% Statements 259/363 **100%** Branches 4/4 **40.9%** Functions 27/66 **71.47%** Lines 223/312

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

Filter:

File	Statements	
src	0%	0/23
src/anime	74.66%	56/75
src/anime/input	87.5%	7/8
src/auth	71.42%	55/77
src/auth/guards	75%	6/8
src/auth/input	100%	17/17
src/common	87.87%	29/33
src/config	66.66%	10/15
src/movie	74.62%	50/67
src/movie/input	100%	5/5
src/user	68.57%	24/35

Figure 11 : Jest coverage dashboard

Figure 12 : RabbitMQ

RabbitMQ: is an open-source message-broker software that originally implemented the Advanced Message Queuing Protocol.

As we mentioned previously, the graphql-microservice communicates to the movie-microservice and the anime-microservice through RabbitMQ. Thanks to NestJS that offers a wrapper to manipulate the RabbitMQ client, we are able to either subscribe to a queue or publish to it. In our implementation, we have 2 queues, one for the movies and the other one for the animes. The entry-point is responsible for forwarding the movie/anime registration event, so it is the publisher and the other two would be the subscribers. Other than that, NestJS provides the ability to request-response through RabbitMQ and that is the case when requesting for movie/anime data from its corresponding service. [R8]

Figure 13 : Docker

Docker: is a containerization technology. As mentioned in our architecture diagram (Fig X), every service is running on an isolated docker container within the same docker network. The whole project runs into 11 containers. [R9] Each microservice is also pushed to a docker repository that is DockerHub. [R10]

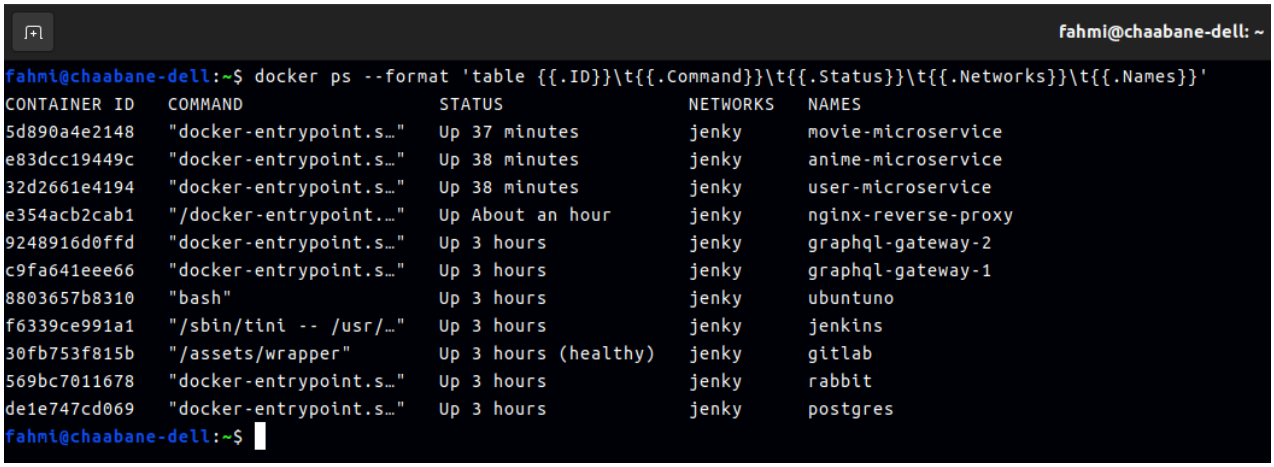

Figure 14 : Running docker containers

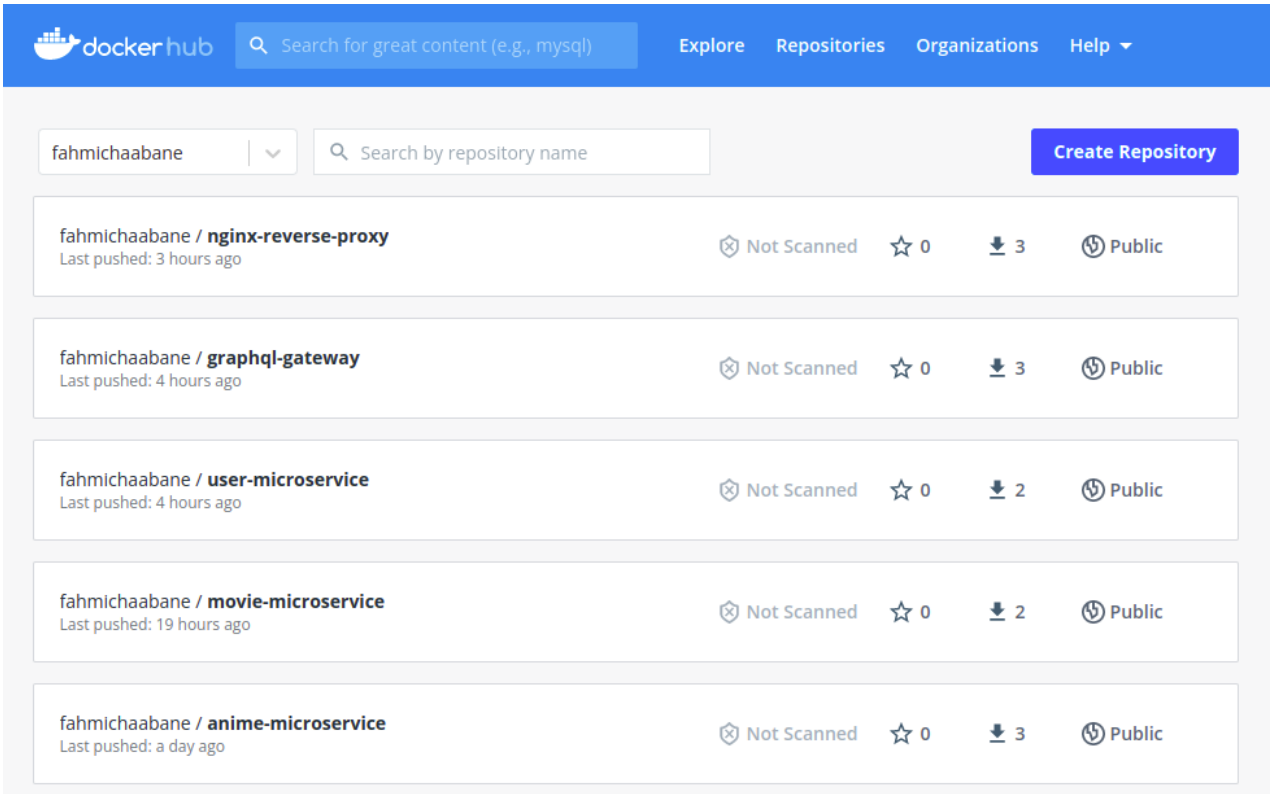

Figure 15 : Deployed images to Dockerhub

Figure 16 : PostgreSQL

PostgreSQL: also known as Postgres, is an open-source relational database management system. In our application, there is only one instance of a database that the microservices connect to.

Figure 17 : GitLab

GitLab: is an open-source git repository. In our project, we used GitLab as both a local repository for our services and a hook container by defining server hooks ‘post-receive’ to trigger jenkins’ jobs after each push to the repository. [R11]

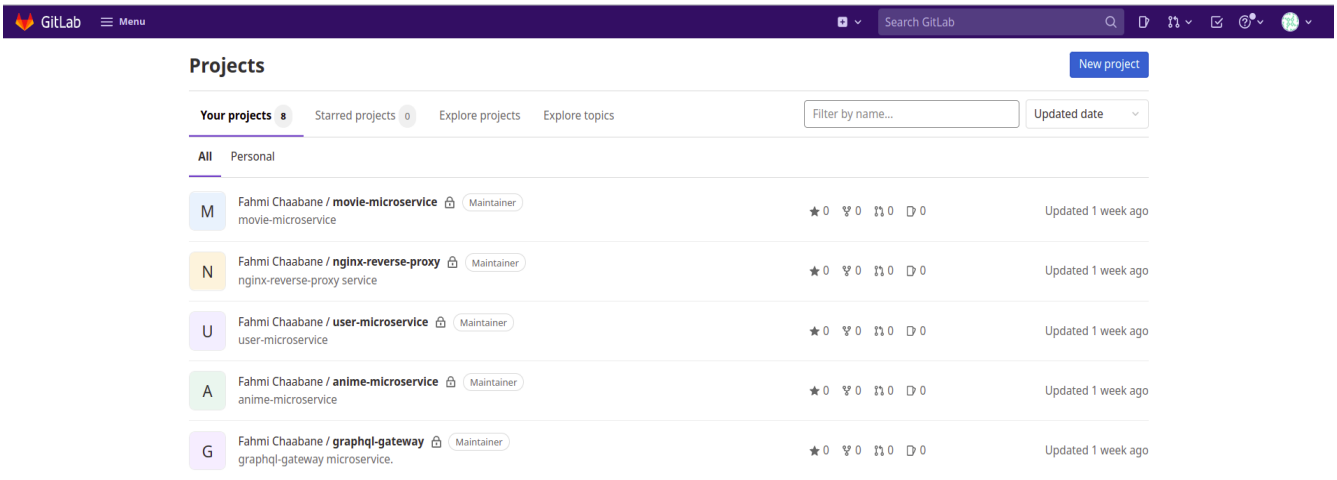

Figure 18 : GitLab Repository

Figure 19 : Jenkins

Jenkins: is an open-source devops automatisation server.

We have established a Jenkins image that contains docker inside of it, meaning, the Jenkins container is able run docker containers too thanks to the Docker-In-Docker approach, we implemented that feature within our jobs stages.

What we have done by jenkins is basically create a CI/CD pipeline for each microservice following the pipeline as code best practice by implementing the jenkinsfile script (that contains the stages) within our applications code. [R12]

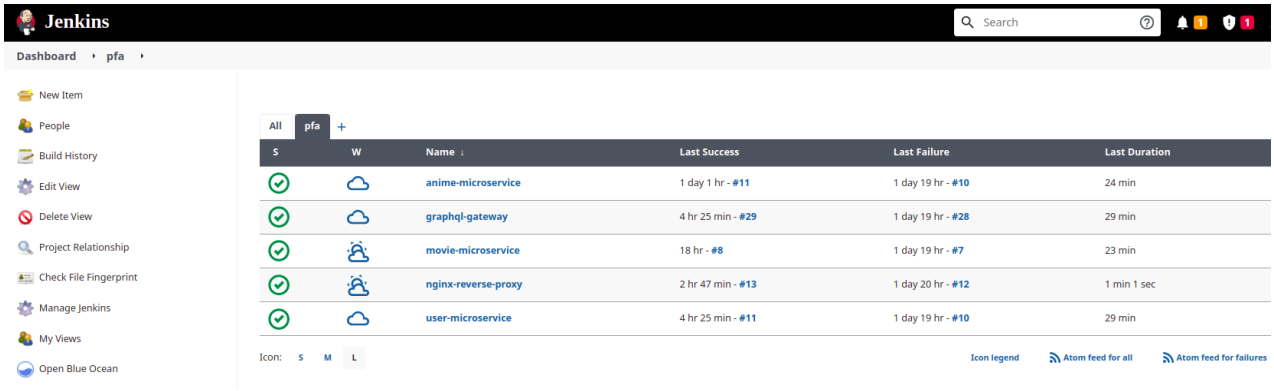

Figure 20 : Jenkins dashboard

For the NestJS instances that are the graphql-microservice, user-microservice, movie-microservice and anime-microservice, we implemented 4 stages that are:

- Build: We build a docker image that contains all of the dependencies needed in order for the instance to run regularly.
- Test: We run our test scripts using Jest.
- Push: We push the built and tested image to dockerhub.
- Deploy: we connect to a ubuntu container that we run on localhost and ssh-server installed to it to simulate the behavior of deploying into a real virtual machine.

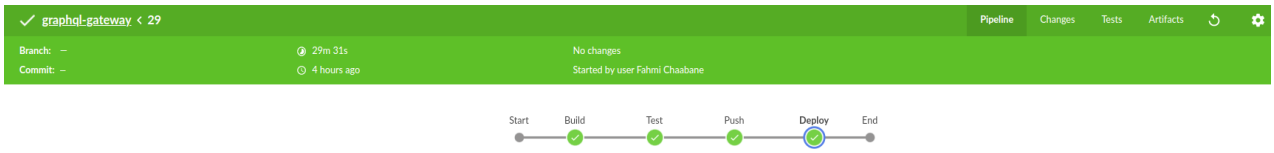

Figure 21 : Pipeline dashboard of graphql-microservice

For the NGINX instance it only contains the Build and Push and Deploy stages.

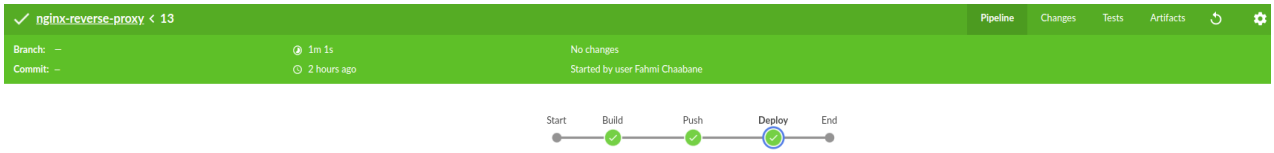

Figure 22 : Pipeline dashboard of nginx-reverse-proxy

PRACTICAL RESULT:

Thanks to the apollo-server, we have access to the Apollo Studio playground, where we can configure it to hit our backend endpoint in order to identify all resolvers that we have implemented. Inside of the Apollo Studio, we can also find documentation for our GraphQL schema to figure out the return types and fields.

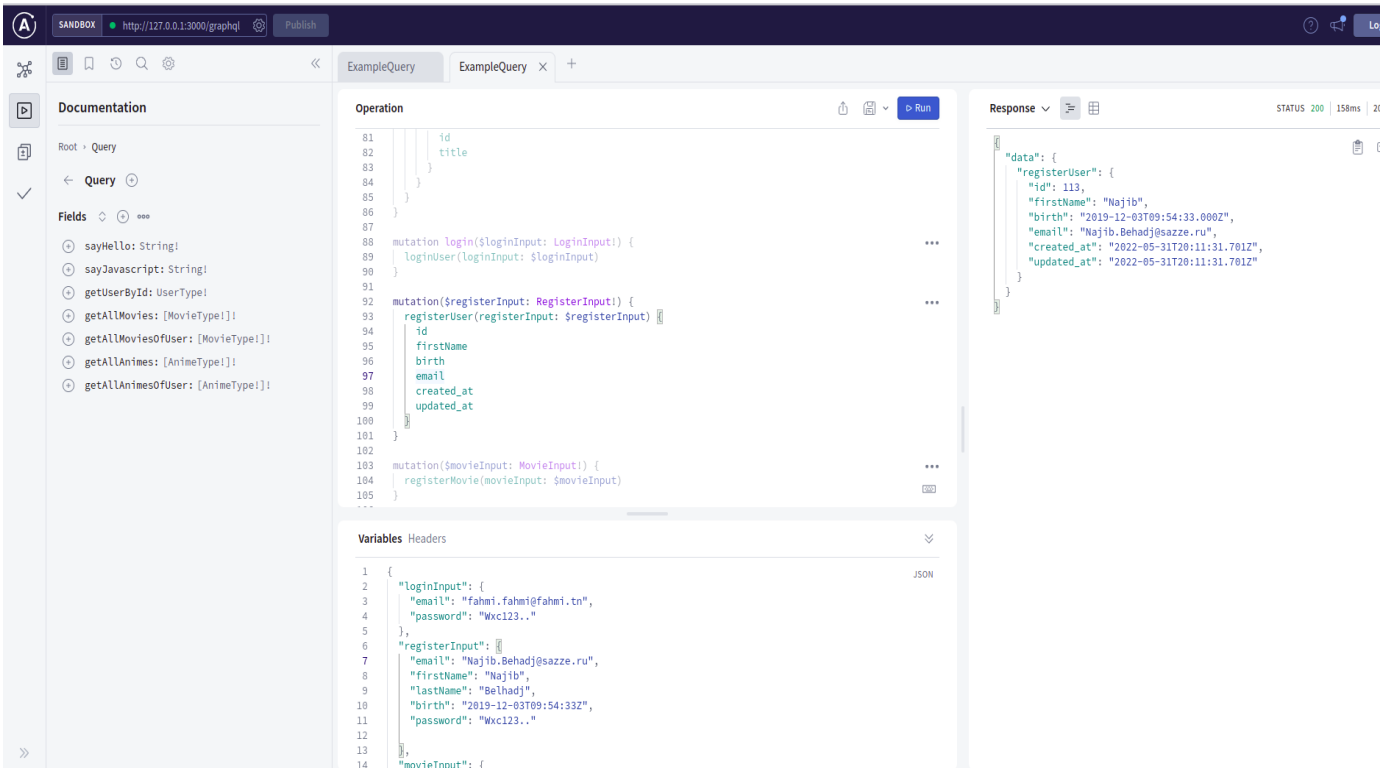

Figure 23 : Apollo Studio

Consequently, we are able to execute all of the defined mutations and queries and check out the returning results.

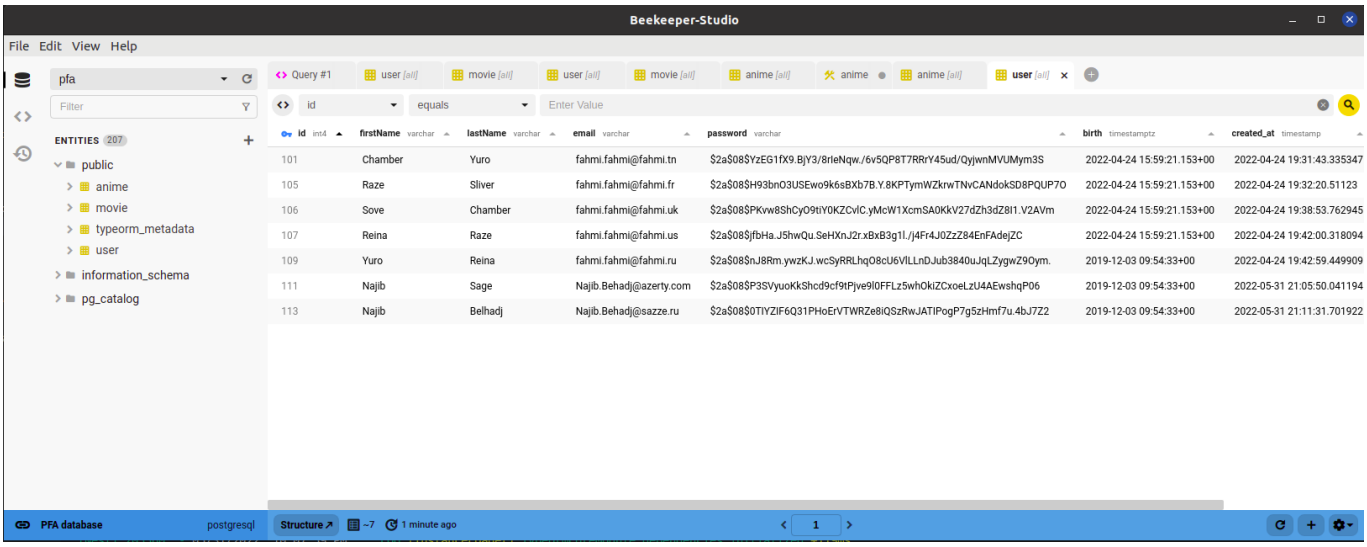

Figure 24 : Beekeeper Studio client for Postgres

GENERAL CONCLUSION:

In this project, we have implemented a small business logic on top of a heavy technical background that includes GraphQL based APIs, code best practices (configuration management, data transfer objects validation, unit tests, etc..) and CI/CD pipelines through an automatisation server. The backend was split into microservices to achieve loosely-coupled, scalable and testable applications.

The code is accessible publicly in : https://github.com/FahmyChaabane/Nest-microservices.boilerplate

REFERENCES

[R1]: https://nginx.org/

[R2]: https://docs.nestjs.com/

[R3]: https://typeorm.io/

[R4]: https://graphql.org/learn/

[R5]: https://www.apollographql.com/docs/apollo-server/

[R6]: https://www.passportjs.org/docs/

[R7]: https://jestjs.io/docs/getting-started

[R8]: https://www.rabbitmq.com/getstarted.html

[R9]: https://docs.docker.com/get-started/overview/

[R10]: https://hub.docker.com/

[R11]: https://docs.gitlab.com/

[R12]: https://www.jenkins.io/doc/