Realized by :

Fahmi Chaabane

Table of Contents

General Introduction 3
Architecture 3
Implementation and Technologies 5
Nextjs 5
NGINX 8
NestJS 9
Apollo-Server 11
Passport.js 11
Redis 12
Docker 12
MongoDB 14
GitLab 14
Practical Result 12
General Conclusion 16
References 18

GENERAL INTRODUCTION:

The idea behind the project was to implement a solution where all the video games that have been previously developed within the company, would be unified on one single database and to provide a dashboard for end users to visualize games statistics.

It was also an opportunity 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. 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 containerization and 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 a front-end “dashboard-service” that is implemented by Next.js which represents the entry point of the final user, an NGINX web server “nginx-microservice” that is going to act at the same time as a reverse proxy and a load balancer. The NGINX instance will communicate to the “proxy-gql-service” service that is the entrypoint to our back-end side of the application. This particular microservice has no direct access to the database, however it does communicate to one other service which is the “core-service”, this service is the only responsible for every read/write interaction with the database, it also communicates to another service “mail-service” that is responsible for notifying end users of the application. Each of the back-end microservices are implemented by NestJS.

The application uses several implementations of transport layers:

- The “nginx-service” communicates to the “proxy-gql-service” through the HTTP transport layer.
- The “proxy-gql-service” communicates to the “core-service” through the TCP layer.
- The “core-service” communicates to the “mail-service” via a Redis instance through queues communication channel.

Thanks to the maturity of the containerization of applications, rather than installing each of the required services like NodeJS and NGINX 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, 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:

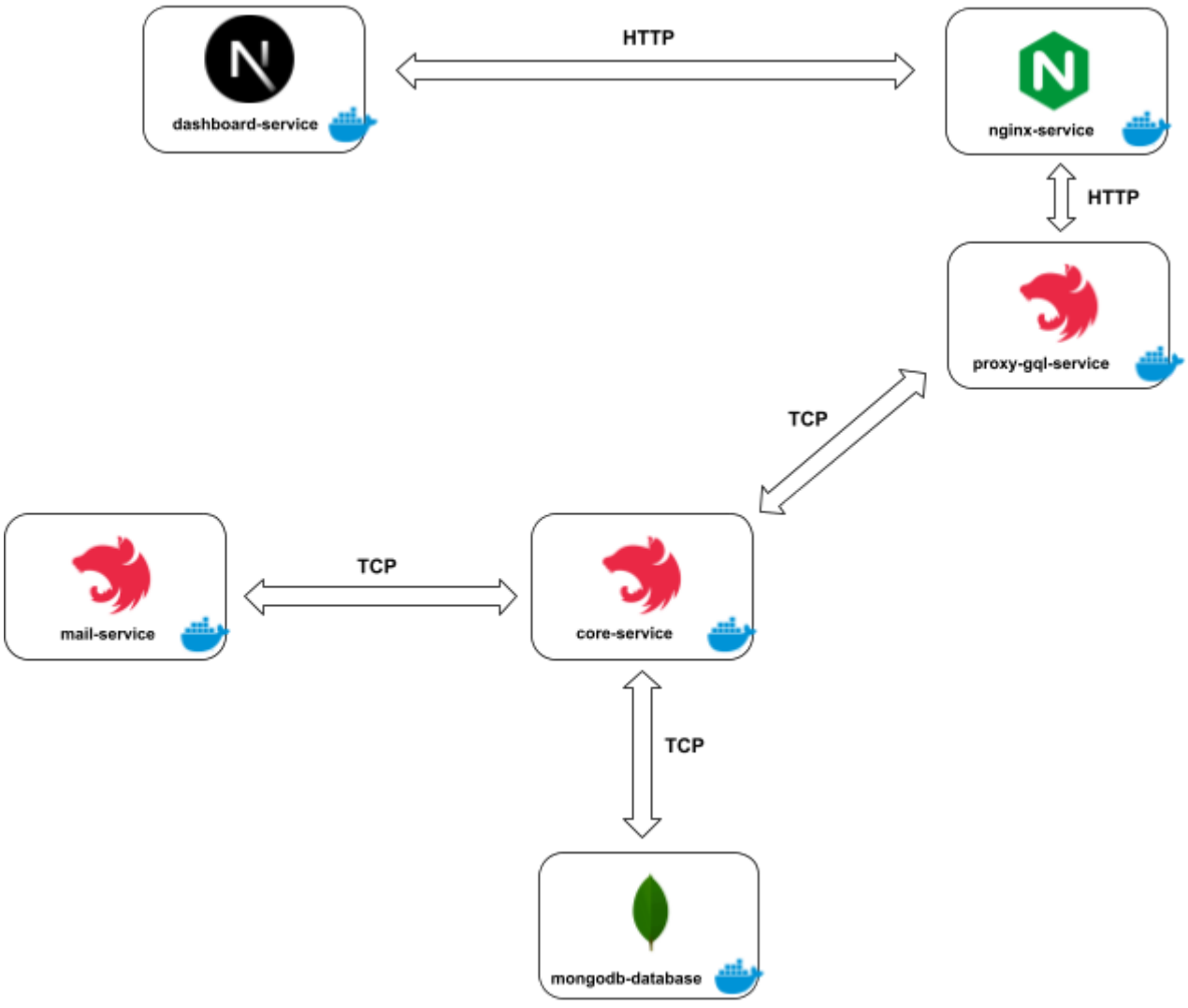

Figure 1 : Project Architecture

IMPLEMENTATION AND TECHNOLOGIES:

In this project, for the business logic, we chose to make it simple and implement a clear database schema. This class diagram should describes the design behind it:

design

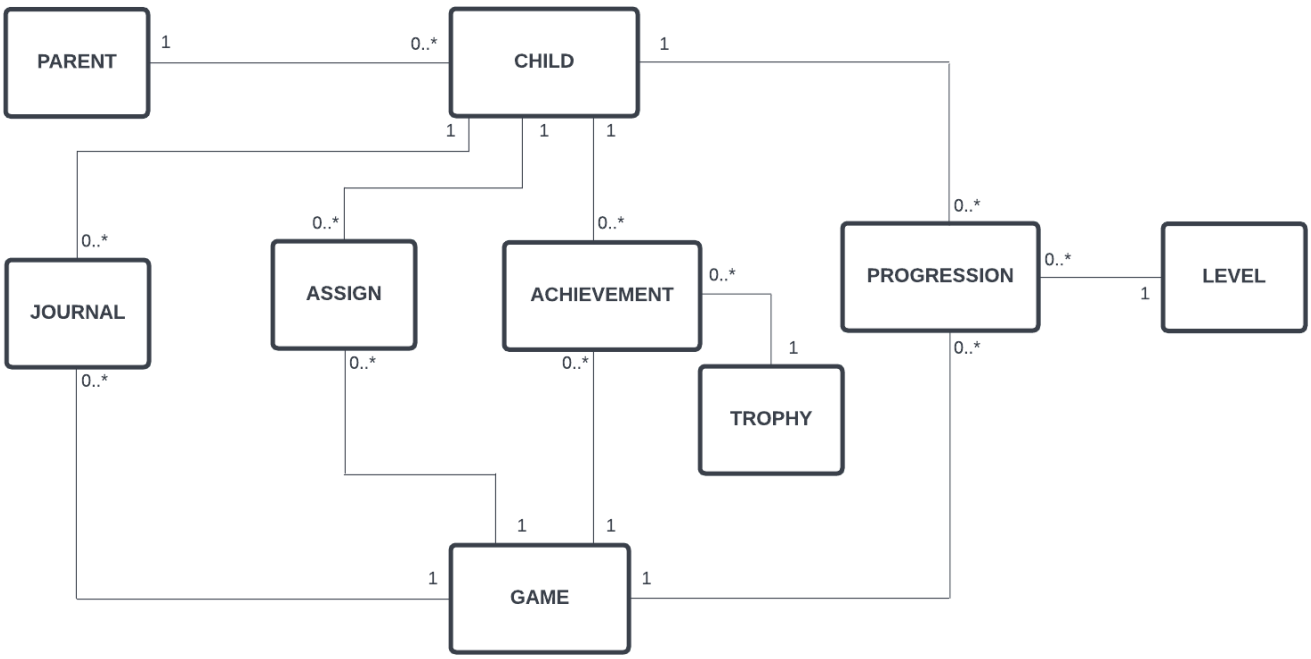

Figure 2 : Class diagram

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 : Nextjs

Nextjs: is an open-source web development framework created by Vercel enabling React-based web applications with server-side rendering and generating static websites. The Nextjs application plays as the end-user entrypoint where a user can login into his account and check out the game statistics of his children. Both Apollo-client and Axios were used to consume data coming from the back-end side.

Examples of implemented interfaces:

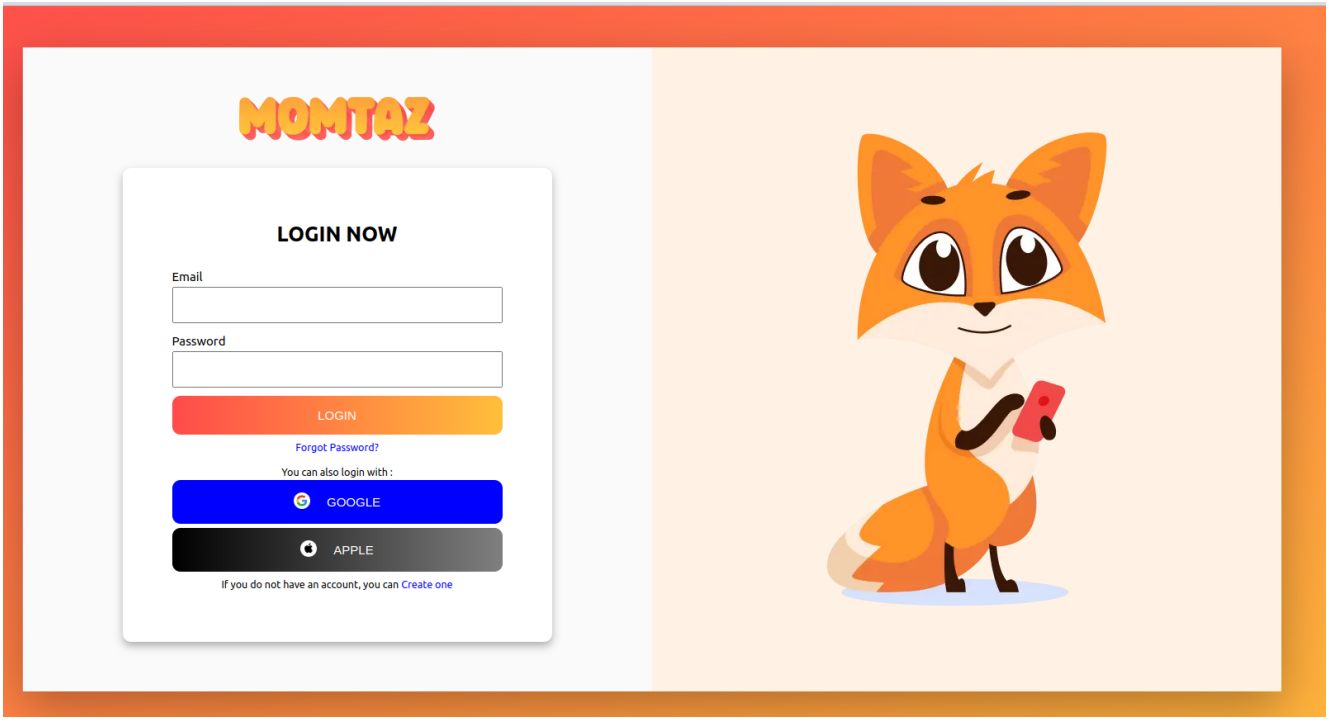

Figure 4 : Login page

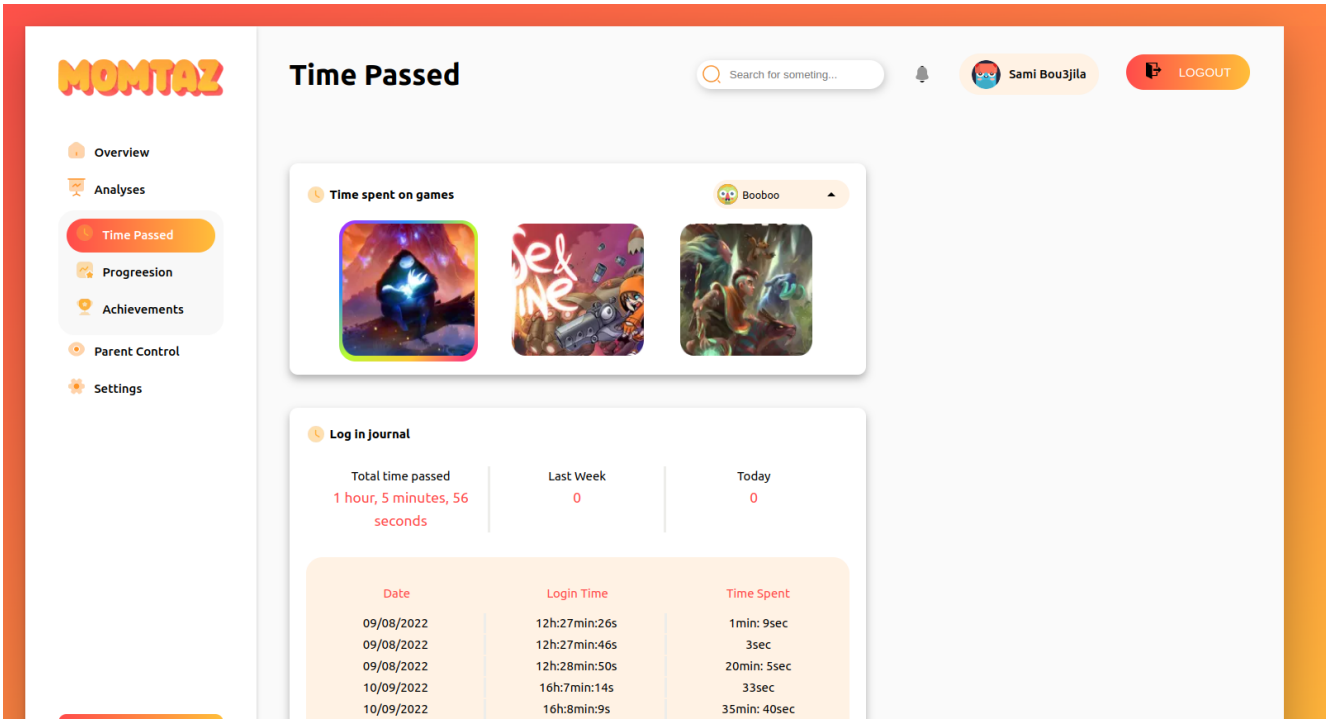

Figure 5 : Journal page

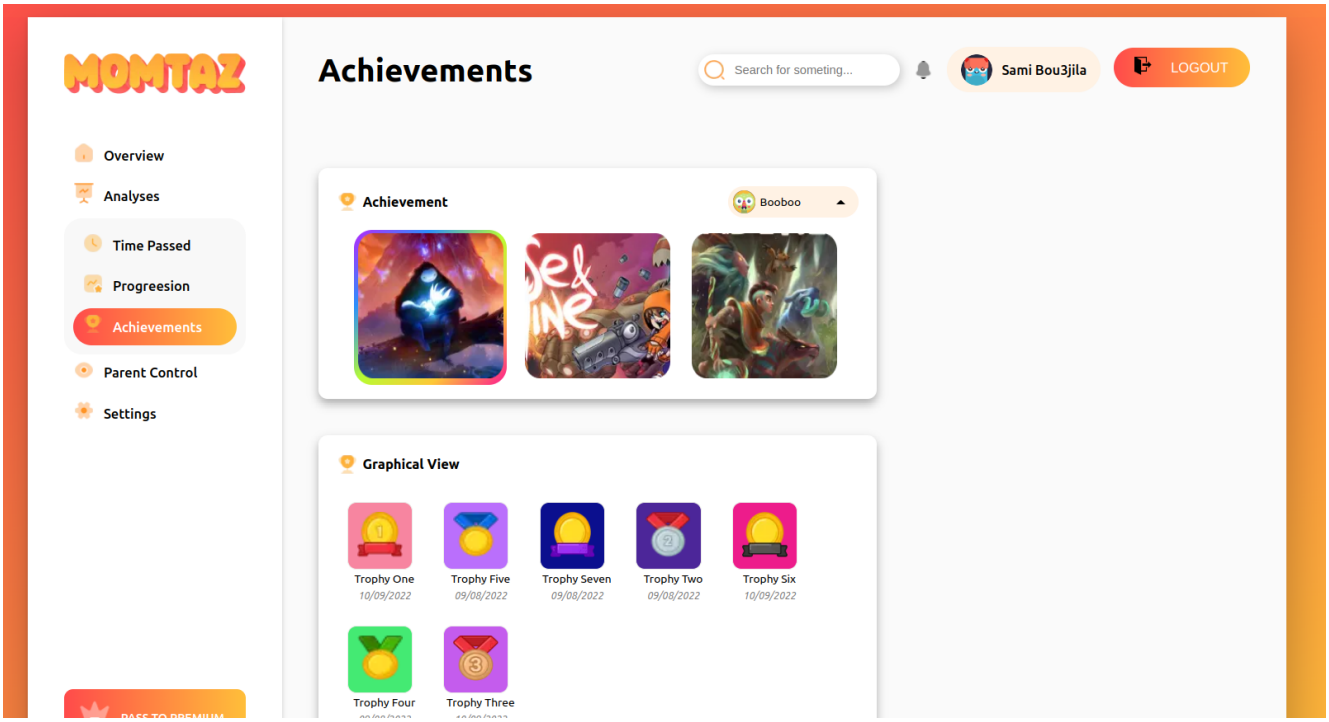

Figure 6 : Achievements page

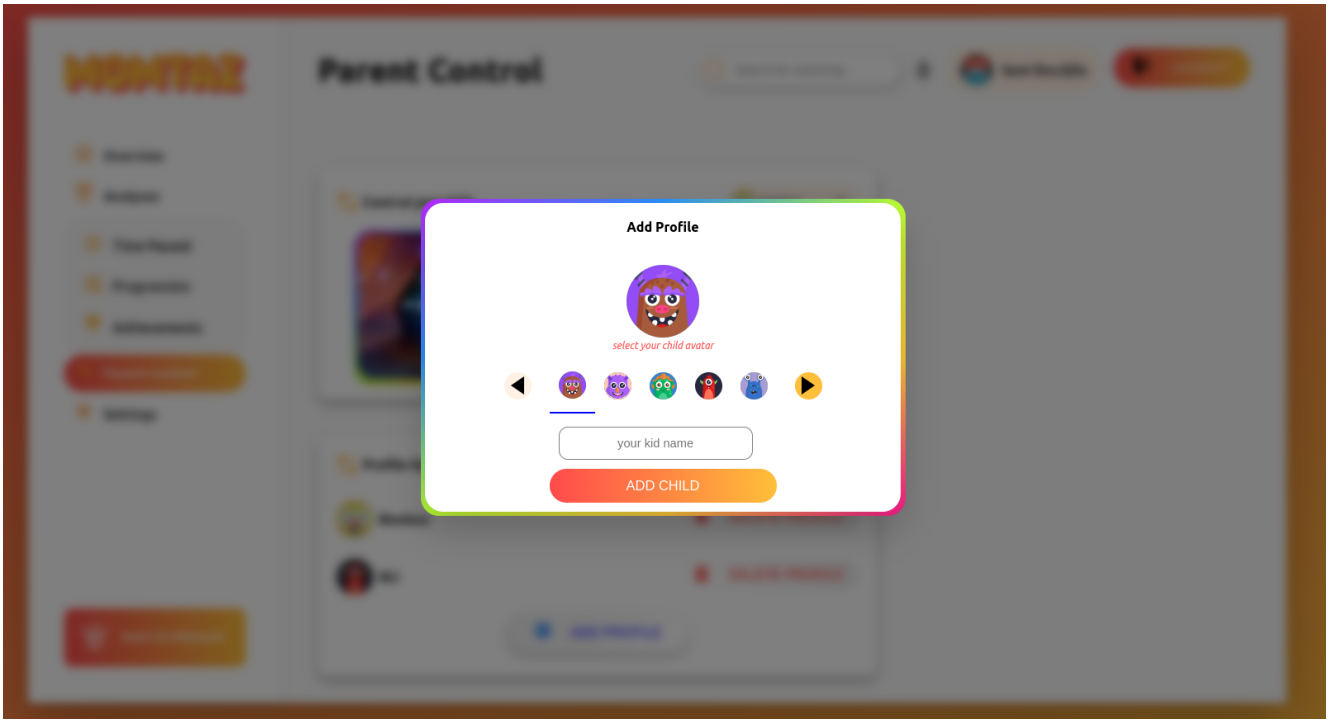

Figure 7 : Parent Control page

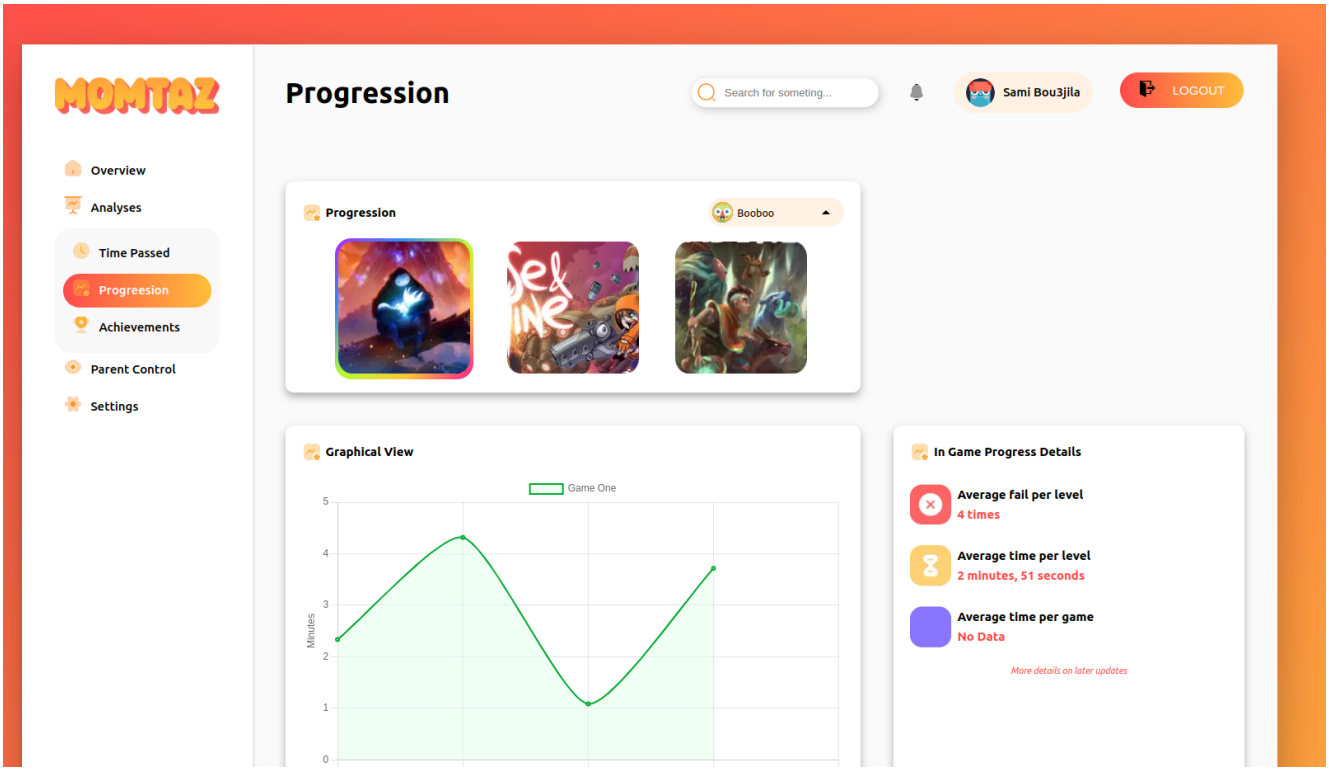

Figure 8 : Progressions page

Figure 9 : 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 back-end by splitting out its 80 port to forward (and load balance) the incoming HTTP requests from the outside world to our back-end. To demonstrate the load balancing process, the proxy-gql-service 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 10 : 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 3 NestJS applications :

- proxy-gql-service: is the entry-point of our back-end, 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 are 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 within this microservice (it was implemented using the Passport.js library [Fig-12]).
- core-service: This is where we define the database entities, the whole business logic functionalities and configure the database connection, which is done thanks to NestJS module Mongoose. [R3]

Figure 11 : Mongoose for database connection

- Mail-service: this service contains the logic of notifying the end users through emails, which is done thanks to NodeMailer API. This service plays as a consumer of data that is provided by the core-service through BullJS queues.

Figure 12 : Mongoose for database connection

We are also able to visualize the data that are provided to the queues by splitting out an endpoint :

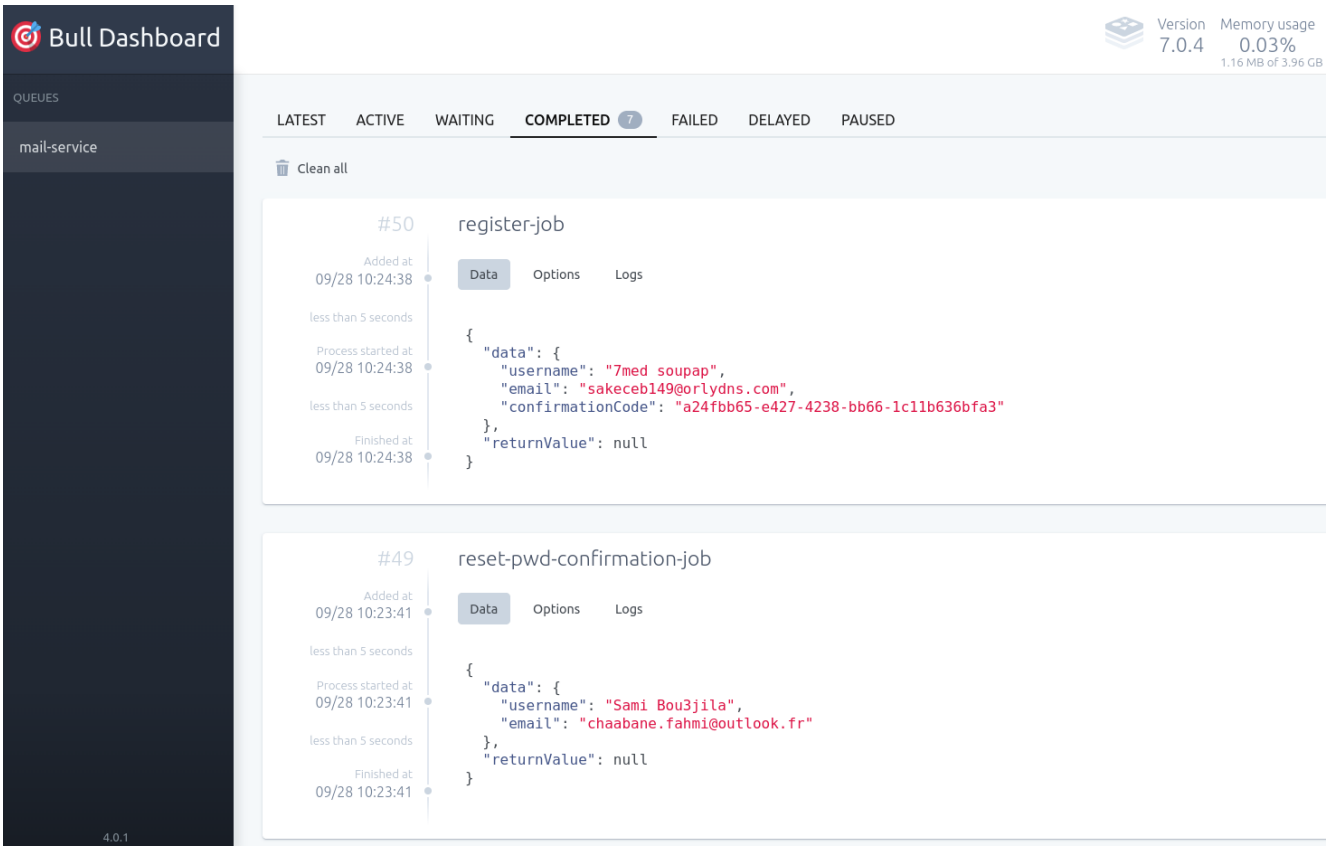

Figure 13 : Bullboard dashboard

Figure 14 : 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 different endpoints in order to fulfill the intended query. For example, in our case, a 'child' has a list of 'games', so whenever we query the child and its corresponding list of games, we first request the child specific getter endpoint in the main query function and then within the field resolver of games, we query the games corresponding endpoint. [R5]

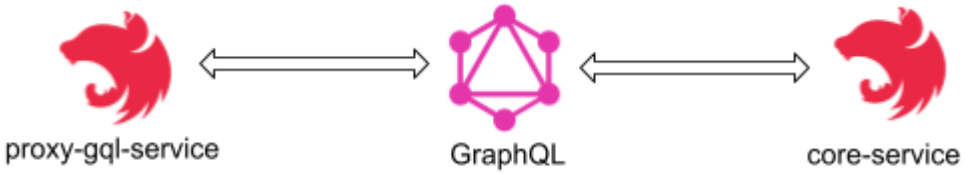

Figure 15 : GraphQL for querying data

Figure 16 : 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 [R6], if the token is invalid, an exception would be thrown back to the client. We also used Google Oauth 2.0 strategy in order to login to the application using google account. [R7]

Figure 17 : Passport as an authentication middleware

Figure 18 : Redis

Redis: is an in-memory data structure store, used as a distributed, in-memory key–value database, cache and message broker, with optional durability. As we mentioned previously, BullJS is a redis-based queue for Node.. [R8]

Figure 19 : Docker

Docker: is a containerization technology. As mentioned in our architecture diagram, every service is running on an isolated docker container within the same docker network. The whole project runs into 6 containers. [R9] Each microservice is also pushed to a docker repository that is DockerHub. [R10]

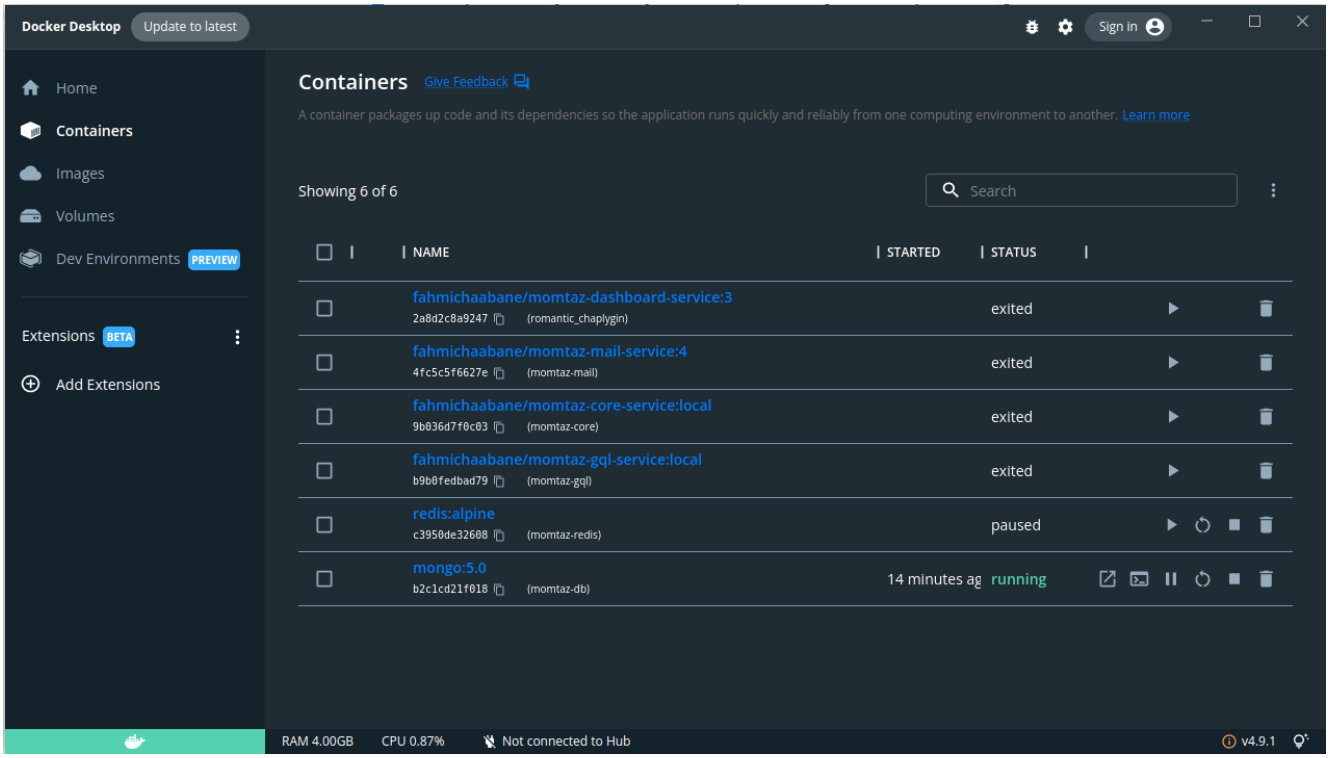

Figure 20 : docker desktop interface

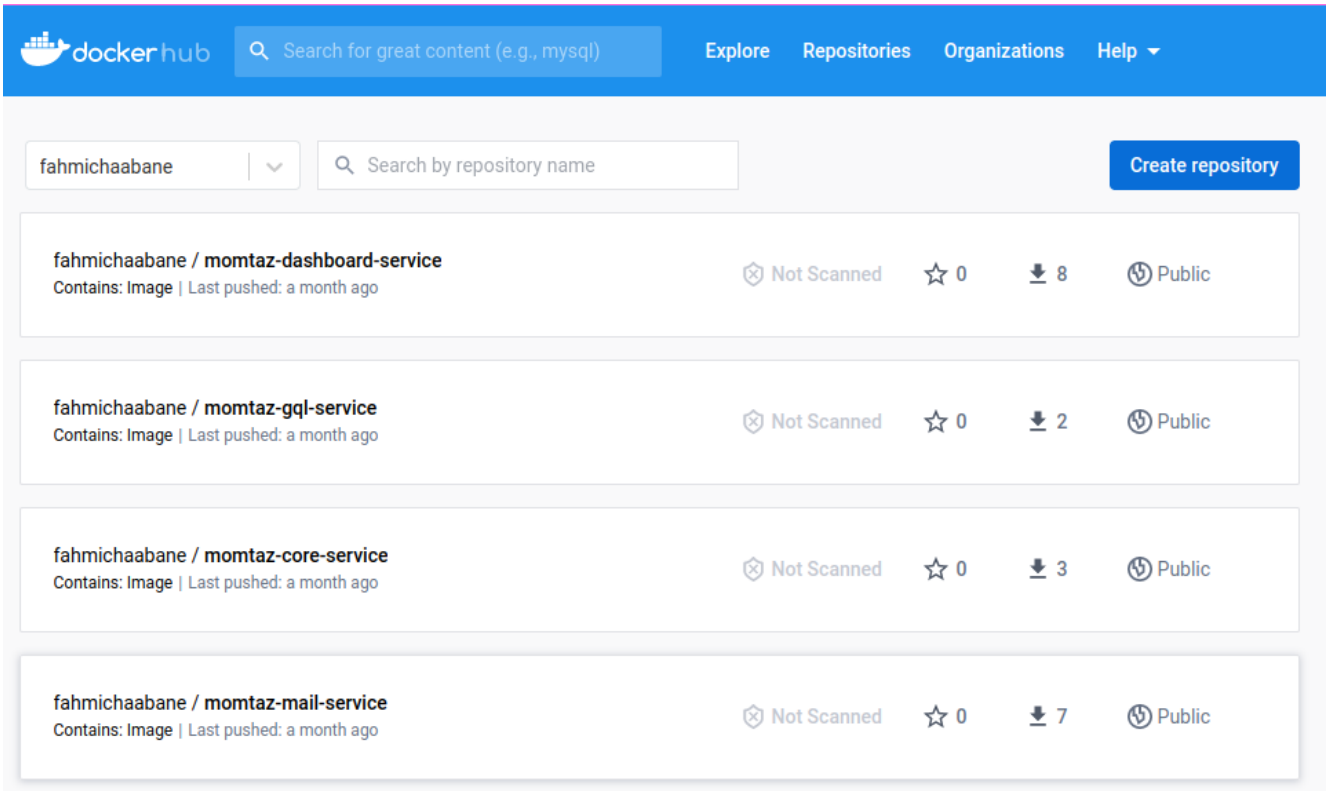

Figure 21 : Deployed images to Dockerhub

Figure 22 : MongoDB

MongoDB: is a source-available cross-platform document-oriented database program. In our application, there is only one instance of a database that the microservices connect to.

Figure 23 : GitLab

GitLab: is an open-source git repository. In our project, we used GitLab as both a local repository for our services [R11] and a CI/CD pipeline executer using the gitlab runners from the cloud [R12].

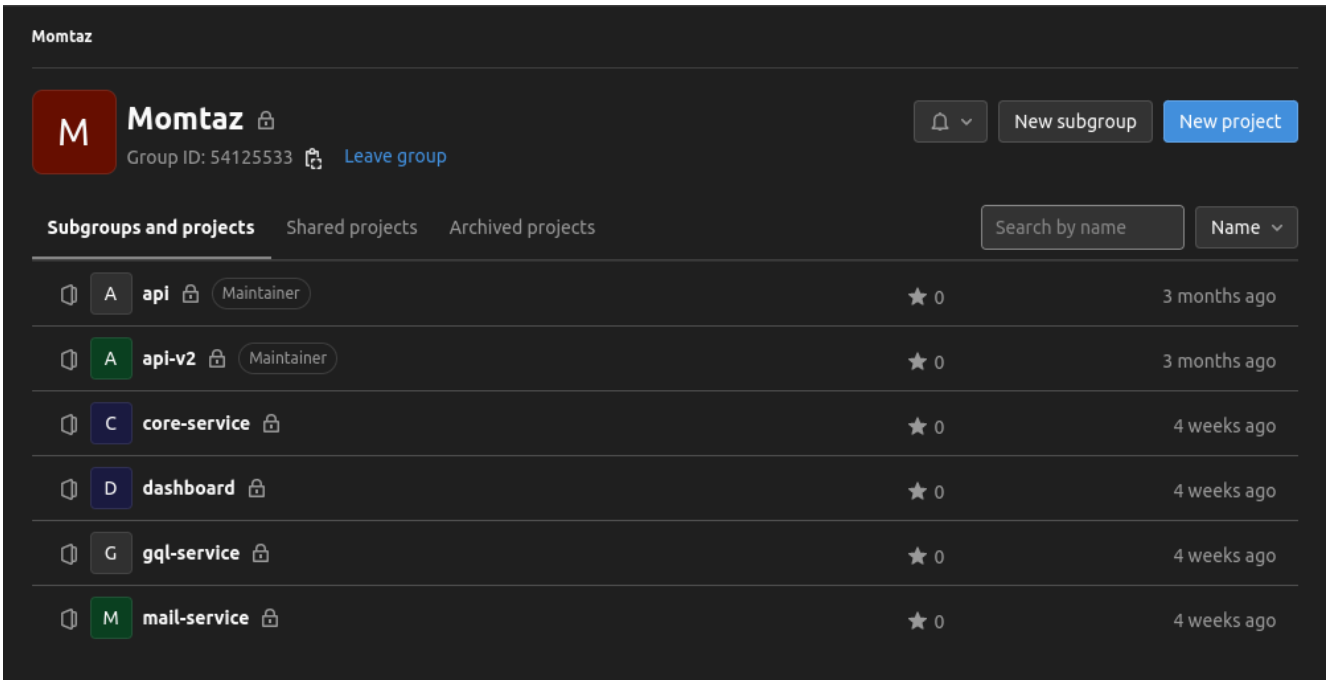

Figure 24 : GitLab Repository

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

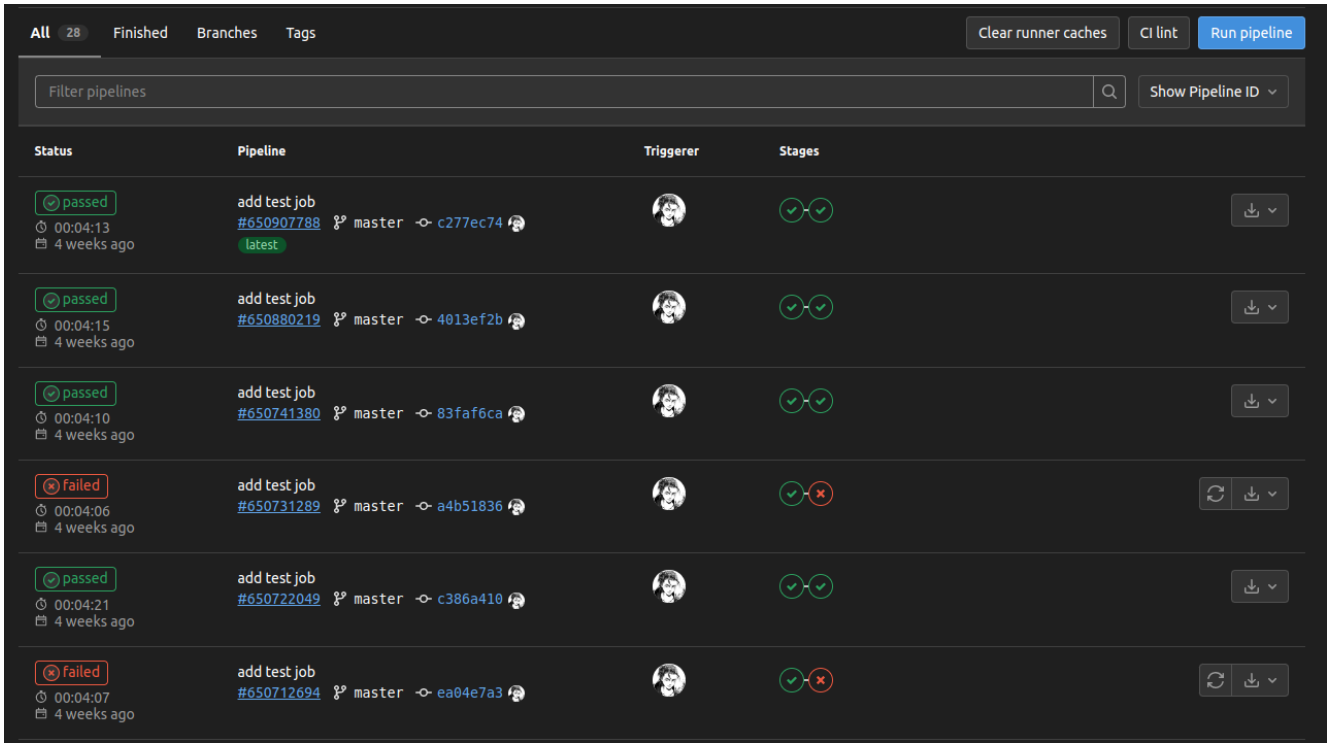

Figure 25 : Gitlab pipelines dashboard for core-service

Almost all services that we implemented share the same stages logic :

- Build: We build a docker image that contains all of the dependencies needed in order for the instance to run regularly.
- Push: We push the built image to dockerhub.
- Smoke test: We run the image locally and check the health endpoint of the service.

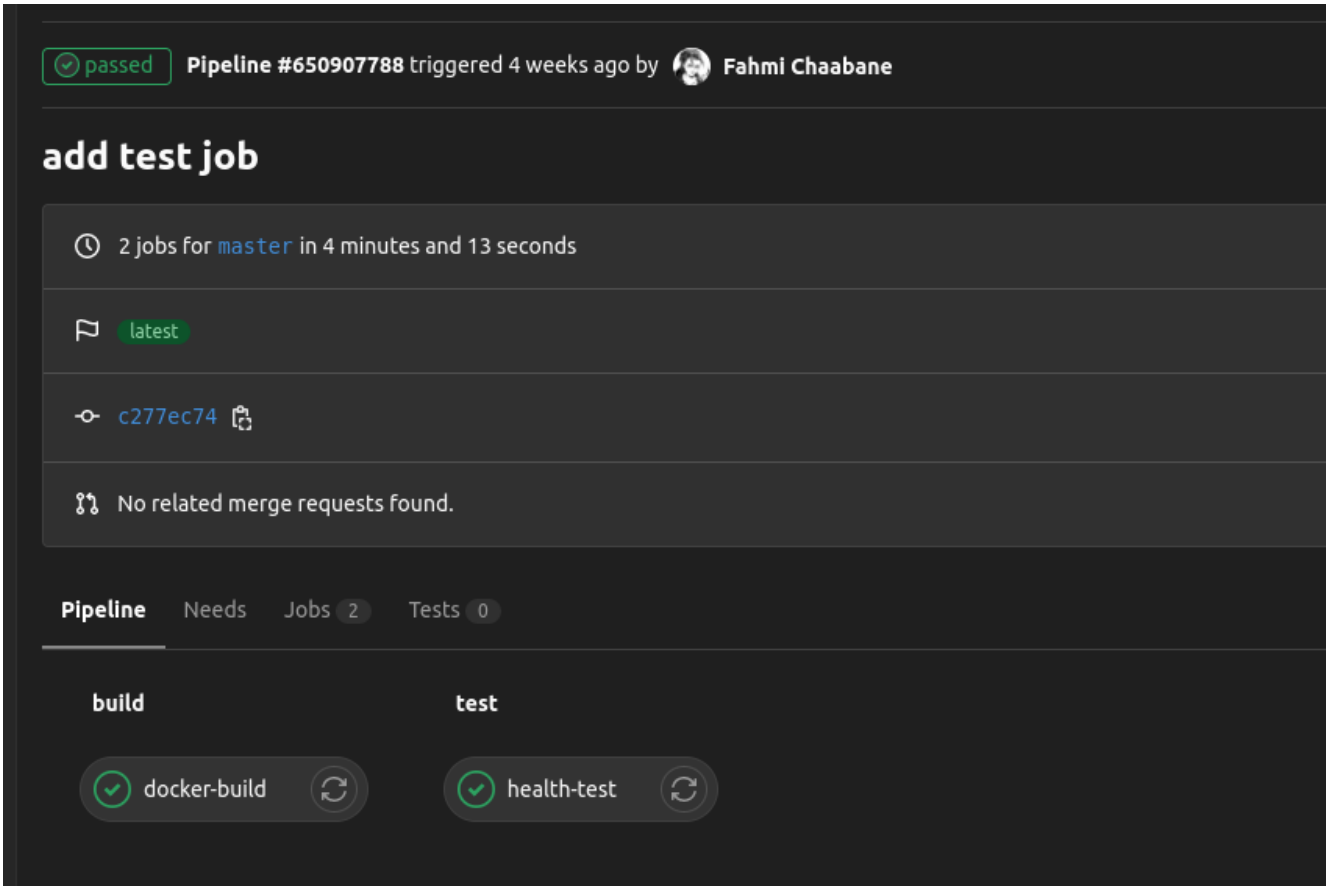

Figure 26 : Pipeline pipeline stage of proxy-gql-service

PRACTICAL RESULT:

Thanks to the apollo-server, we have access to the Apollo Studio playground, where we can configure it to hit our backend endpoints 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.

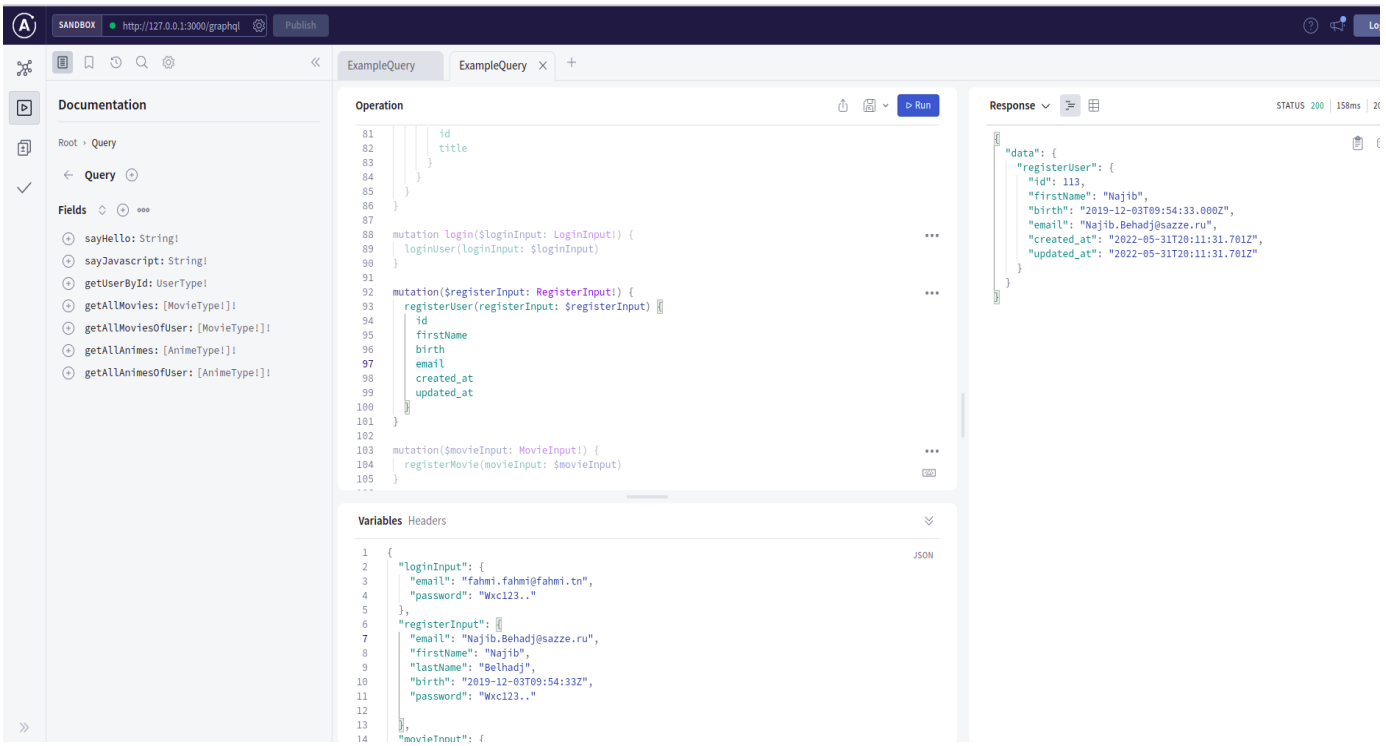

Figure 27 : Apollo Studio

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

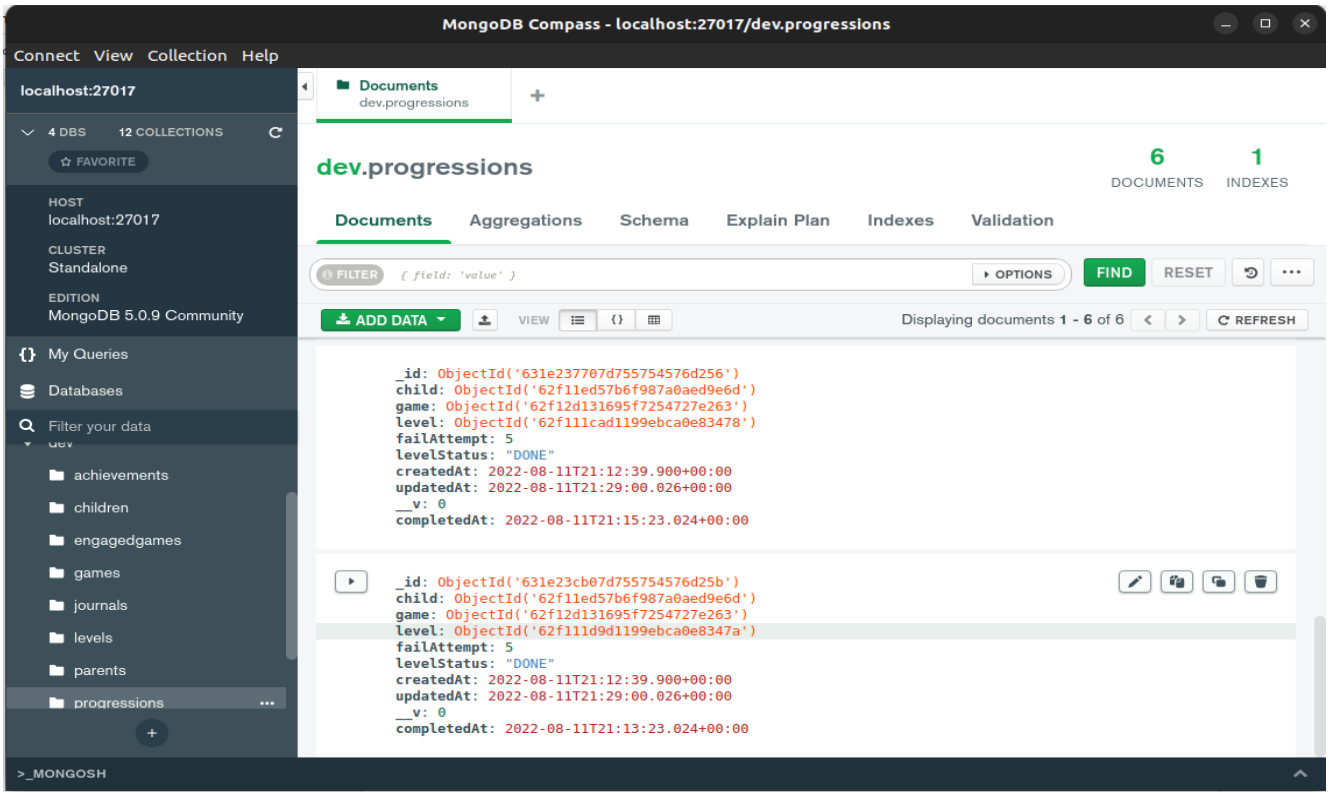

Figure 28 : MongoDB Compass

GENERAL CONCLUSION:

In this project, we have implemented a business logic on top of a 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/momtaz

REFERENCES

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

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

[R3]: https://mongoosejs.com/

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

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

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

[R7]: https://www.passportjs.org/concepts/authentication/strategies/

[R8]: https://github.com/OptimalBits/bull

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

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

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

[R12]: https://docs.gitlab.com/ee/ci/