Universidade do Minho
Escola de Engenharia
Departamento de Informática

Pedro Dias Parente

The role of an API Gateway in a Microservice Architecture

October 2022

Universidade do Minho
Escola de Engenharia
Departamento de Informática

Pedro Dias Parente

The role of an API Gateway in a Microservice Architecture

Master dissertation
Master Degree in Informatics Engineering

Dissertation supervised by
José Carlos Leite Ramalho

October 2022

STATEMENT OF INTEGRITY

ABSTRACT

Nowadays, with the development of bigger and more complex applications, the architectural paradigm for application development is changing from a more traditional Monolithic approach to an architectural style called Microservices. In this more recent, and increasingly popular, style of developing applications, a tool that has also become increasingly more popular is API Gateways. In this thesis I explored these and a few other concepts on various examples, recording my experience, with the intent to create a guide on how to more efficiently implement these tools on to your own projects, facilitating the usually long and arduous process of researching, learning, and implementing new technologies into your work.

Keywords: Microservices, API Gateway

RESUMO

Hoje em dia, com o desenvolvimento de aplicações maiores e mais complexas, o paradigma arquitetural para desenvolvimento de aplicações está a transacionar do estilo Monolítico tradicional para um estilo de arquitetura chamado Microserviços. Neste mais moderno, e cada vez mais popular, estilo de desenvolvimento de aplicações, uma ferramenta que também se tem tornado cada vez mais popular tem o nome de API Gateway. Nesta tese eu explorei estes e outros conceitos em vários exemplos, documentando a minha experiência, com a intenção de criar um guia em como o leitor pode mais eficientemente implementar estas ferramentas nos seus próprios projectos, facilitando o normalmente longo e trabalhoso processo de pesquisa, aprendizagem, e implementação de novas tecnologias no próprio trabalho.

Palavras-Chave: Microserviços, API Gateway

CONTENTS

LIST OF FIGURES

ACRONYMS

A

API	Application Programming Interface.
AWS	Amazon Web Services.

D

DB	Database.

H

HTTP	Hypertext Transfer Protocol.
HTTPS	Hypertext Transfer Protocol Secure.

R

REST	REpresentational State Transfer.

S

SPF	Single point of failure.
SQL	Structured Query Language.

U

URL	Uniform Resource Locator.

1 INTRODUCTION

In 2011, the term "Microservices" was first used in a conference of software architects and was then formally adopted. (Foote) Since then, the style of architecture has become more and more popular. Microservices are described as a type of architecture where your application is divided into several smaller and isolated services, communicating and cooperating through API calls. Each service is independent from each other and is built for specific tasks, allowing complex applications to be built in a much more simple, flexible and scalable way. (Nemer) But with these advantages comes a disadvantage, the complexity of API calls needed to be made for a single request. Because a Microservices architecture is divided into several services, a single request may need to call a large number of services. But there is a tool which facilitates this process. API Gateways address this by being a tool capable of managing these API calls. Acting as a reverse proxy, it redirects all of the calls to the appropriate services and returns the response. It also has many other functions like rate-limiting, user authentication, etc. (API)

1.1 MOTIVATION

As previously stated, this way of building applications is becoming more and more prominent, but it deals with a lot of foreign concepts for someone who hasn't had previous experience in dealing with these concepts and the many tools that I will talk about in the next sections. Learning how to use these tools is not an easy task and usually takes a lot of research and trial-and-error.
This thesis attempts to alleviate that process by constructing a sort of "instruction manual" in how to apply the discussed concepts and tools into your own work. To do this, I will research and apply the tools into many applications that will serve as a "test" of my newly acquired knowledge and record my failings and successes as in to help the reader in their attempts.

1.2 OBJECTIVES

In this dissertation I intend to explore these concepts with the aim of leaving a guide in how to apply an API Gateway in a Microservices architecture. To do this there are a few objectives that are needed to be fulfilled:

- Research on API Gateways and Microservices architecture
- Building a guide on how to apply API Gateways on discussed architecture.
- Implementation of the guide in several case studies, culminating in its application to the Hypatiamat app. (Hypatiamat)

1.3 DEVELOPMENT APPROACH

The following steps are the methodology that I intend to follow in this thesis:

- Experimenting with Kong API Gateway on small, simple applications;
- Using simple applications that are already divided into small services to experiment and understand the communication in a Microservices network;
- Fusing the two previous steps, using Kong as a reverse proxy for the Microservices network;
- Researching and experimenting with other technologies like NGINX, Docker, GraphDB, MongoDB, etc, with the attempt to create more robust applications and get accustomed to these technologies;
- Protecting the connection to the website application by changing from a HTTP protocol to HTTPS protocol;
- Make a guide documenting all of the previous steps;
- Using the Hypatiamat application as a "final test" for the constructed guide.

1.4 DOCUMENT STRUCTURE

This dissertation is divided into several chapters. Chapter 2 "State of the art" is a contextualization on the current state of the several technologies and theory that were used during the writing of this master thesis. Chapter 3 "Guide/Development" and Chapter 4 "Hypatiamat" is where a documentation on the work that was done in the completion of the thesis is written. This is also where the proposed "guide" will be written. Finally, Chapter 5 "Conclusion" is a reflection on the work made.

2 STATE OF THE ART

In this chapter I will contextualize and describe the current state of the main topic of this master thesis, as well as the technologies that I will use that are related to this topic.

2.1 MICROSERVICES

It wasn't always the case that Microservices were a possible route when building an application, in fact, it was only in the 2010's that the concept was introduced and adopted in force. It wasn't long ago when it didn't even make much sense to decouple your services into independent containers. When all you had were desktop applications installed on your computer why have a complex network of communication between services when it can perfectly be run as a single unit on your computer? But as web applications became more popular and the complexity and scale of said applications became bigger and bigger, such that any change to the code base or error that popped up became a huge ordeal for the developers, changes needed to be made. And so the concept of Microservices was introduced.

The Microservices architecture addressed many of the issues that were present in the Monolithic architecture, which was more popular at that point. Some of these are :

- It became easier to scale up projects;
- New additions to the development team didn't need as much training to contribute to the project (instead of understanding the whole application they could focus on the specific service they were working on);
- It facilitated developers working in parallel, not only code-wise but also time-wise (it's fine to work on different services at the same time);
- It is easier to improve performance with less cost (If a service was over-stressed developers can acquire more servers to alleviate the load on that specific service while the others remain with the same amount of service power, which becomes more cost-effective when comparing the Monolithic approach which is to just acquire more servers for everything);

- It severely mitigates the "single point of failure" problem. If a service went down it could be restarted independently while the application as a whole had no down time.

But there was a cost to this architecture: **Complexity in building network communication**. Setting up this network is many times no easy task and can lead to a lot of difficult debugging, but if managed successfully it makes this choice of architecture make a lot of sense for most medium to big applications.

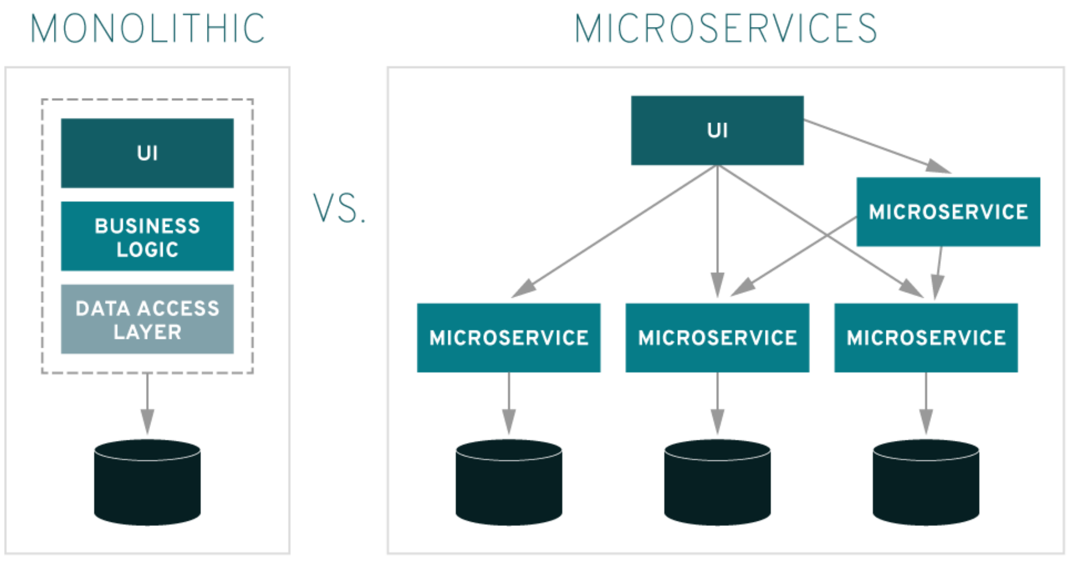


Figure 1: Monolithic vs Microservices (Faria)

As it is depicted by the image, the client can communicate with each service and they may communicate between themselves as well. Another thing that is worth remarking is that it is very common for a service to have its own dedicated database.

2.2 API GATEWAYS

It may be fine to have direct communication from client to the services when dealing with smaller applications, but when these scale up from using a few services to tens or even hundreds of services, a few problems arise:

- Securing your services from threats, as all your services have public endpoints;
- It may become hard to manage connections to each service (when updating or replacing them).

API Gateways address these problems, although not without bringing issues of their own. API Gateways act as a reverse proxy, redirecting all calls to the appropriate services returning the consequent result. (api) In doing so, it resolves the previous problems:

- Because it routes every calls to the appropriate service, these no longer need to have public endpoints to be reached, making them less exposed to possible threats;
- By managing the routing to each service, when a service is updated or replaced, instead of re-configuring every connection to that service you can correct the issue in the routing of the Gateway.

It also may have a few other useful features in form of plugins like:

- Authentication;
- Rate-limiting;
- Analytics on service use;
- etc.

Nevertheless, there is a significant drawback to this approach. The API Gateway introduces a possible **single point of failure**, meaning that if the Gateway goes down the whole application fails until it is back up (if there is nothing in place to prevent or circumvent this, like a second API Gateway for example). Even still, for the purpose of this thesis, I will use an API Gateway: **Kong API Gateway**.

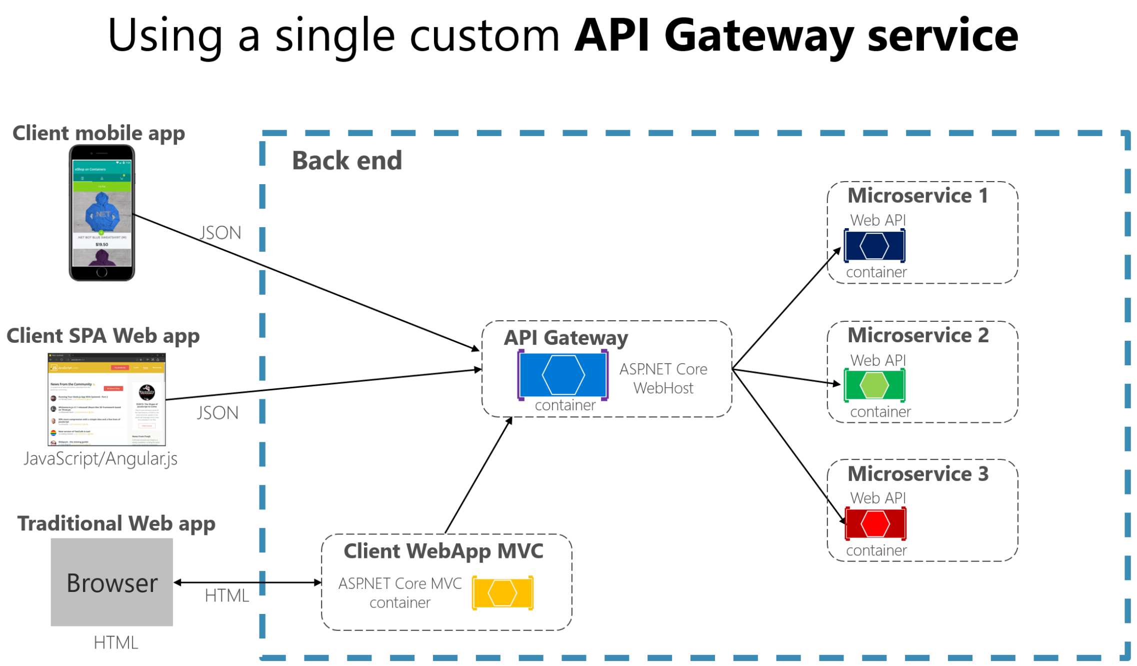


Figure 2: Microservices architecture with an API Gateway (api)

2.3 KONG AND OTHER ALTERNATIVES

Kong API Gateway is not the only possible choice when it comes to managing APIs. A few of the most popular alternatives are: **Amazon API Gateway**, **Apigee** and **WSO2 API Manager**. (There are many more alternatives and which is best is subjective but for the purposes of this thesis I will talk about these in specific).

2.3.1 *Kong*

Kong is an API Management tool that became open-source in 2015. It is widely used by developers all over the world and is a great choice for anyone looking for a free API Gateway tool. It runs on Lua and supports plugins like rate-limiting, authentication, load-balancing, and many others. It also has an enterprise version. (Faren)

2.3.2 *Amazon API Gateway*

AWS API Gateway (or Amazon API Gateway) is an API Management tool provided by Amazon. From their website - "Amazon API Gateway is an AWS service for creating, publishing, maintaining, monitoring, and securing REST, HTTP, and WebSocket APIs at any scale. API developers can create APIs that access AWS or other web services, as well as data stored in the AWS Cloud." (AWS). The service is paid (per API call) but a free 12 month trial is available.

2.3.3 *Apigee*

"Apigee Edge, the self-service API management platform, enables companies to secure, scale, manage, and analyze their digital business, and grow API programs to meet the increase in demand. Edge enables enterprises to design and build the APIs that securely share their services and data." (Apigee) Apigee is a pay-per-API call solution that is mostly used by large companies.

2.3.4 *WSO2 API Manager*

WSO2 API Manager was initially released in 2012, and is a completely open-source and free tool for API management. It has security functionalities with it's WSO2 Identity Server, analytics tools with the WSO2 Data Analytics Server and "workflow management capabilities with human interaction features with WSO2 Business Process Server" (Gunaratne)

2.3.5 *Kong vs alternatives*

All of these solutions are great and are definitely reasonable choices for the purposes of this thesis, but Kong does stand out from the others by being a free alternative with intuitive use and excellent plugin support. It's for those reasons that Kong was chosen for the API Manager role in this dissertation.

3 GUIDE/DEVELOPMENT

Learning new technologies when starting new projects can be quite a difficult task. It involves a lot of research, reading documentation, testing, debugging, etc. Thus, a guide that focuses on explaining how these technologies work in a technical and practical way, with a few case studies as examples, might prove to be helpful to someone who never used said technologies. This thesis is an attempt at creating that guide.

3.1 INTRODUCTION

This section is composed by a guide to a Microservices architecture followed by a detailed accounting of the experience gained with several web applications, the process of transforming them into the Microservices architecture and the problems faced. The guide was constructed along the writing of this thesis and was made with the purpose of assisting those who hope to transform or make web applications that run on a Microservices architecture and API Gateway. It is a generalization of the many areas you can address the matter (from how Microservices work to the set up of the API Gateway) and, as said above, it will end with a couple of examples of web apps, used as case studies, to transform them into the Microservices architecture assisted by API Gateway Tools.

3.2 MICROSERVICES

Before trying to apply an API Gateway in your project, it makes sense to make sure you have a Microservices Architecture. This is because an API Gateway is a way to manage the communication and data transferring between different services, so having your application divided into different services is not only crucial, but almost necessary (if you intend to make the most out of an API Gateway like Kong). To do this, take your app and try to list the many different sectors of functionalities it has and run each of them in a different service. Using a popular example, a website where you buy clothes could have a service dedicated for the commenting system, one to list the available clothes, and one for a rating system.

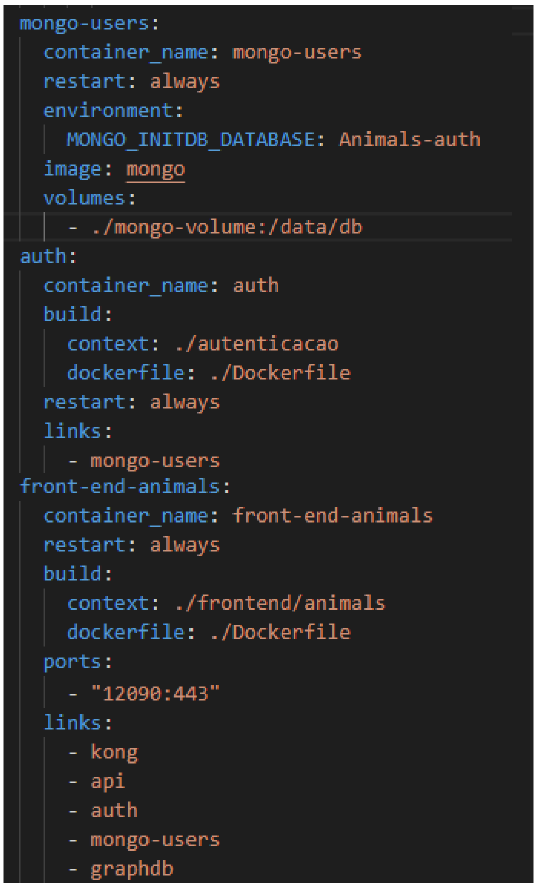
It could even use external APIs, for example, an API to handle the money transactions performed in buying the clothes themselves.

But, it is also important to not divide your app into too many services, as it could make your service network too complex. As a good rule of thumb, if you have two services that only communicate with each other and are performing simple tasks that could be done by only one service, it might be worthwhile to join those two services into only one. It is also useful (but not necessary) to separate your front-end from your back-end services.

3.3 DOCKER

When deploying an application on a machine it is incredibly useful to use Docker, this is because Docker creates **containers**, independent from the machine and OS (Operating System) it runs on, equipped with only the necessary software to fulfill its purpose, which is specified by the developer. This makes it possible for **any** application to run on **any** machine.

Therefore, it is important to run every service in your application in its own container. Here is an example of a docker-compose file:



```yaml
mongo-users:
  container_name: mongo-users
  restart: always
  environment:
    MONGO_INITDB_DATABASE: Animals-auth
  image: mongo
  volumes:
    - ./mongo-volume:/data/db
auth:
  container_name: auth
  build:
    context: ./autenticacao
    dockerfile: ./Dockerfile
  restart: always
  links:
    - mongo-users
front-end-animals:
  container_name: front-end-animals
  restart: always
  build:
    context: ./frontend/animals
    dockerfile: ./Dockerfile
  ports:
    - "12090:443"
  links:
    - kong
    - api
    - auth
    - mongo-users
    - graphdb
```

Figure 3: Docker-compose file example

The docker-compose file is a config file which dictates what containers will be run when running the command **docker-compose up** and how they will run. In this example there are three containers that will be run: a *front-end* container (**front-end-animals**) which a user can communicate with through his/her browser, a *back-end* container (**auth**) which handles authorization, and a *database* container (**mongo-users**) which will store the users.

Going through each container, starting with **mongo-users**, this container uses an image of MongoDB, a document-oriented NoSQL database. An image is the software in each you will be basing your container. The application that is deployed with this docker-compose file uses a MongoDB database so naturally the image for this particular container will be *mongo*. This container is also using something called *volumes* which, in short, allow for data persistence in a container, even if it goes down and has to be restarted.

Because of how Docker works, when, for whatever reason, Docker fails and has to restart its containers, the data stored while it was running is lost. This because when the container fails and shuts down, the container no longer exists and the data created while it was running does not persist, even if it restarts automatically because it will start as a fresh instance of the container. To combat this we use Docker volumes. Docker volumes create a connection between the containers virtual file system and the hosts file system, so when data is written in the container it is also written in the hosts file system, and vice-versa, therefore making it so when a container is started with a fresh instance of the container, it automatically gets populated with the data it had from the hosts file system, thus having data persistence. The way volumes can be set up in a docker-compose file is like this:

```
  - host_path:container_path
```

Meaning before the colon you have the path to the folder in your host machines file system, and after the colon you have the path in your containers virtual file system.

The next container would be **auth**. In this container it is worth noting the "build" segment. *Context* indicates the path for the folder in which the rest of the "building" settings will be in relation to. *Dockerfile* indicates the path to the Dockerfile.

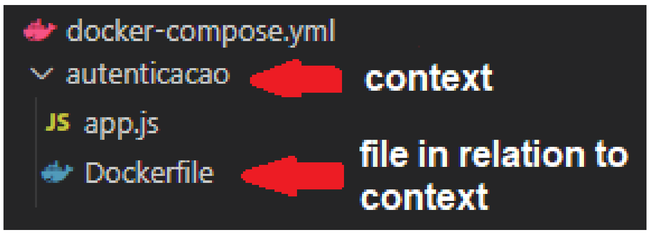
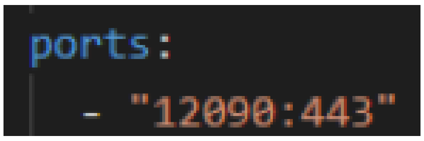


Figure 4: Context and Dockerfile

The Dockerfile is another file used in building the container, besides the docker-compose.yml file. From the official Docker documentation "Docker can build images automatically by reading the instructions from a Dockerfile. A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image." meaning it allows us developers to build our containers with even more detail, facilitating the whole process of deploying an application.

Lastly, in **front-end-animals** it is worth mentioning the "ports" segment. This defines what port to expose our container to in our machine. Since users will only be directly communicating with the front-end container through the browser, this is the only container the needs to expose ports. In this case, since the application is running on port 443 inside the container, and on the client side we want to be able to connect to the application through port 12090, ports were configured this way.



```
ports:
  - "12090:443"
```

Figure 5: Docker-compose front-end ports

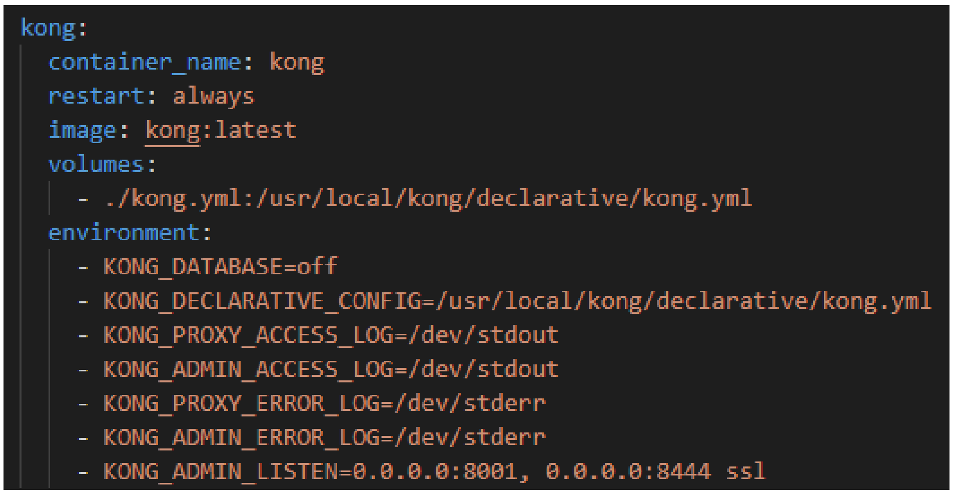
It is also worth mentioning that in all containers the "restart: always" setting was enabled, which makes sure whenever a container goes down for any reason it immediately restarts making sure there are essentially no down-times.

3.4 KONG

Kong is the API Gateway that will be showcased in this guide, but any API Gateway works and they almost all of them work essentially on the same principles.

Kong can be deployed in many different ways, but because we already have Docker running it makes sense to use Docker to deploy Kong as well. When deploying with Docker, you also have the choice to run Kong with two modes: **with a Database** or **without a Database (DB-less mode)**. These choices are up to you, the developer, and you can and should look at the documentation when setting up Kong (Documentation). In this guide, we will be going with DB-less mode.

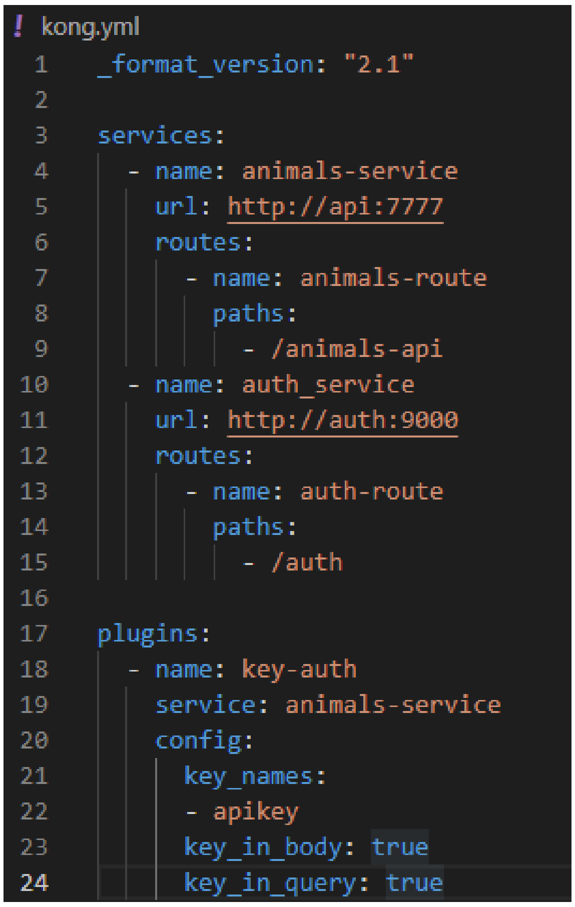
To deploy Kong this way, we will need to set up two documents: the **docker-compose** file, which we already have and will be adding Kong on to it, and a **kong.yml** config file, with the configuration for Kong.



```yaml
kong:
  container_name: kong
  restart: always
  image: kong:latest
  volumes:
    - ./kong.yml:/usr/local/kong/declarative/kong.yml
  environment:
    - KONG_DATABASE=off
    - KONG_DECLARATIVE_CONFIG=/usr/local/kong/declarative/kong.yml
    - KONG_PROXY_ACCESS_LOG=/dev/stdout
    - KONG_ADMIN_ACCESS_LOG=/dev/stdout
    - KONG_PROXY_ERROR_LOG=/dev/stderr
    - KONG_ADMIN_ERROR_LOG=/dev/stderr
    - KONG_ADMIN_LISTEN=0.0.0.0:8001, 0.0.0.0:8444 ssl
```

Figure 6: Docker-compose Kong

There is not much need to change this but the documentation elaborates on why it is set up this way. But it is worth noting that because we are running Kong inside a container, Kong's configuration file must also be inside the container. We could go about this in two ways: copying the file over to the container, or mapping the config file which is the host's file system onto the config file created in the container's file system, through volumes. In this case, its the latter. (Also, the kong.yml file has to be on the same folder that the docker-compose file is, because of how the volume is set up)



```yaml
kong.yml
_format_version: "2.1"

services:
  - name: animals-service
    url: http://api:7777
    routes:
      - name: animals-route
        paths:
          - /animals-api
  - name: auth_service
    url: http://auth:9000
    routes:
      - name: auth-route
        paths:
          - /auth

plugins:
  - name: key-auth
    service: animals-service
    config:
      key_names:
      - apikey
      key_in_body: true
      key_in_query: true
```

Figure 7: kong.yml file

This is configuration for a basic kong.yml file. It lists the **services**, which each have a name (up to the developer) , a URL to the service (in this case the Kong container can communicate with the other two containers through those URLs, but these don't have to be other containers, they can lead to external APIs if you wish), and routes, which also have a name and a *path*. This path is what our front-end will use to communicate with the other services through Kong. For example, when attempting to communicate with the Animals API, instead of communicating directly through HTTP://API:7777, our front-end will communicate through HTTP://KONG:8000/ANIMALS-API.

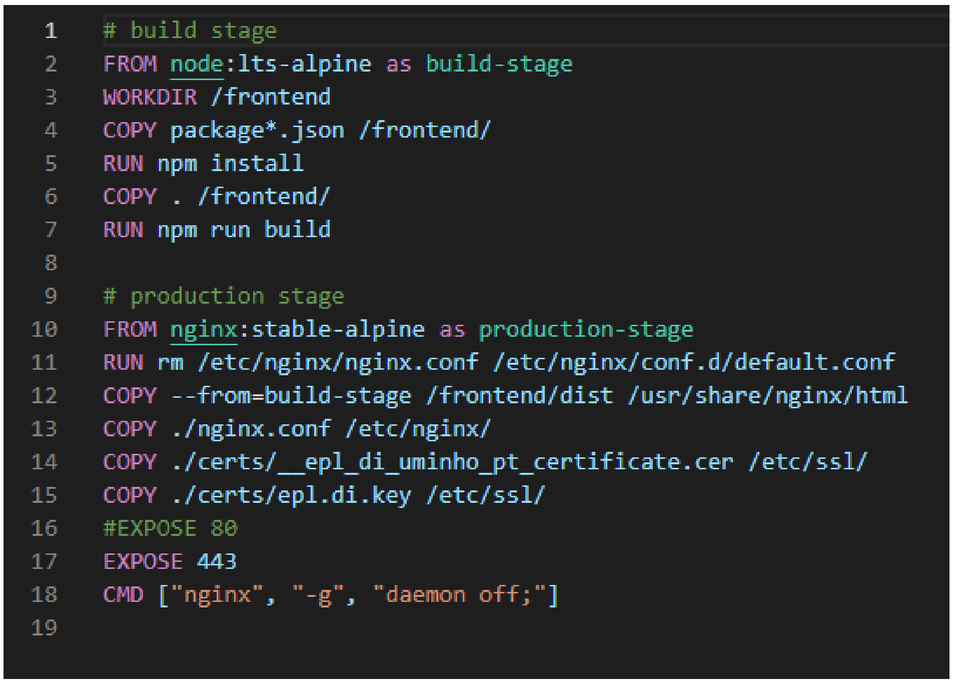
In this configuration file there is also **plugins**, which is a big benefit of Kong because it has a huge library of plugins to choose from (Plugins). In this case, we are using the plugin "Key Authentication", which has the name "key-auth" (this name is **NOT** up to the developer, it depends on the plugin), and are applying it on a specific service, but you can also apply it to a specific route or apply it globally.

3.5 NGINX AND THE FRONT-END DEVELOPMENT

From the NGINX website "NGINX is an open source software for web serving, reverse proxying, caching, load balancing, media streaming, and more." (NGINX-Website). It is not obligatory but is definitely very useful in the building and deploying of any web application.

To run your front-end container with NGINX you need two documents:

- A Dockerfile (like previously discussed, it allows us to specify how we want our container to be built);
- A config file (**nginx.conf**)



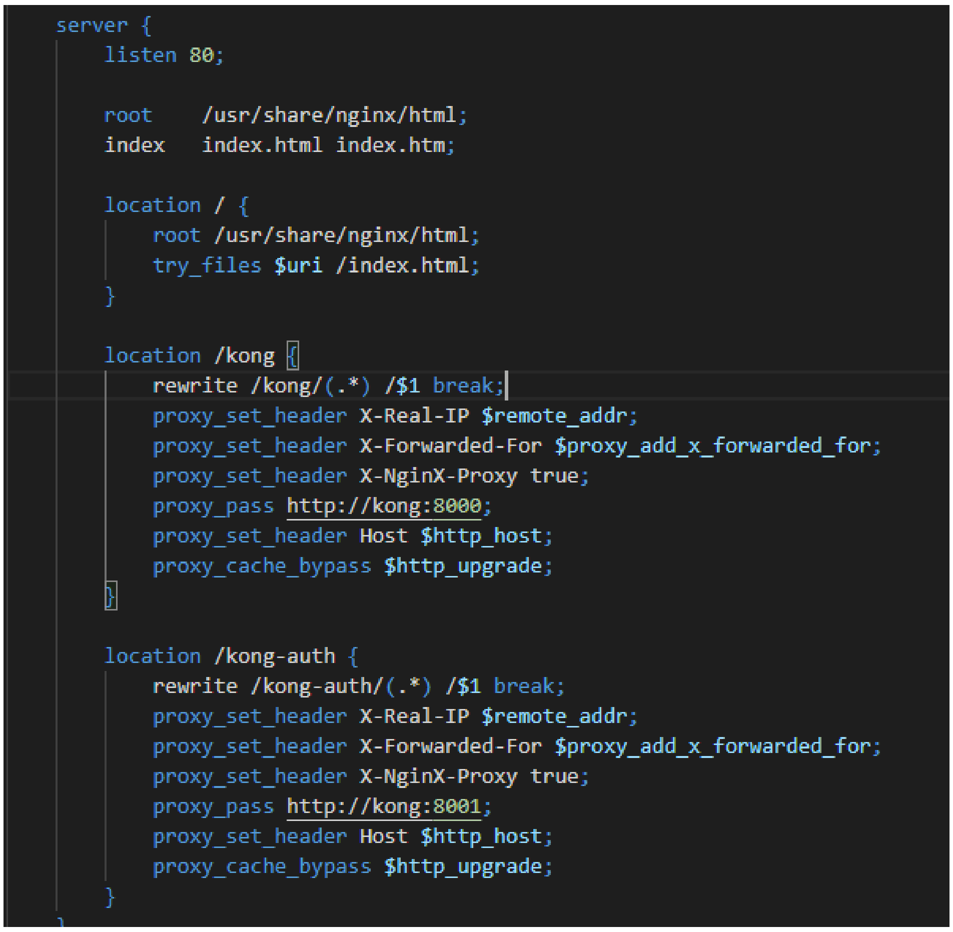
```
# build stage
FROM node:lts-alpine as build-stage
WORKDIR /frontend
COPY package*.json /frontend/
RUN npm install
COPY . /frontend/
RUN npm run build

# production stage
FROM nginx:stable-alpine as production-stage
RUN rm /etc/nginx/nginx.conf /etc/nginx/conf.d/default.conf
COPY --from=build-stage /frontend/dist /usr/share/nginx/html
COPY ./nginx.conf /etc/nginx/
COPY ./certs/__epl_di_uminho_pt_certificate.cer /etc/ssl/
COPY ./certs/epl.di.key /etc/ssl/
#EXPOSE 80
EXPOSE 443
CMD ["nginx", "-g", "daemon off;"]
```

Figure 8: Front-end Dockerfile

The first half of the Dockerfile, the build stage, is just setting up the server as normal, first getting Node.js, then installing dependencies and finally running the server. (It is important to run it as a build for NGINX to work)

On the other hand, the second half is setting up NGINX. First it gets NGINX in production-stage mode and then removes the existing default configuration. Then it copies some files, including the next file we will be addressing, the **nginx.conf** file, which is our custom configuration file. The last two "COPY" lines can be ignored for now, as they are for setting up a HTTPS server (the EXPOSE 443 line is also for HTTPS, here you can write EXPOSE 80, which is the normal port used in an NGINX server). Finally, it runs NGINX.



```
server {
    listen 80;

    root    /usr/share/nginx/html;
    index   index.html index.htm;

    location / {
        root /usr/share/nginx/html;
        try_files $uri /index.html;
    }

    location /kong {
        rewrite /kong/(.*) /$1 break;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-NginX-Proxy true;
        proxy_pass http://kong:8000;
        proxy_set_header Host $http_host;
        proxy_cache_bypass $http_upgrade;
    }

    location /kong-auth {
        rewrite /kong-auth/(.*) /$1 break;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-NginX-Proxy true;
        proxy_pass http://kong:8001;
        proxy_set_header Host $http_host;
        proxy_cache_bypass $http_upgrade;
    }
}
```

Figure 9: nginx.conf

A NGINX config file is bigger than this, but the default values are fine, the server block is where you want to change things. Firstly, the *listen 80* line means that the web application will be run in port 80, meaning you can connect to it omitting the port number (example: instead of connecting to HTTP://WEBSITE:80 you can connect to HTTP://WEBSITE). The *root* block is fine as is. The first *location* block is saying if you go to http://website/ you will just go to the website itself, which is what you normally want. On the other hand, the next *location* block is saying that if you send a request to HTTP://WEBSITE/KONG, that request will be routed to HTTP://KONG:8000. In these *location* blocks is where you would normally establish communication with other services, but, as we are using Kong, having those last two *location* blocks is enough, as our front-end will never communicate directly with other services, only with Kong. You may also note the "rewrite" line inside the *location* blocks. That makes it so our request doesn't send the "/kong/" part in a request like "/kong/animals-api", only the "/animals-api" part, as that is what Kong is expecting for on its side.

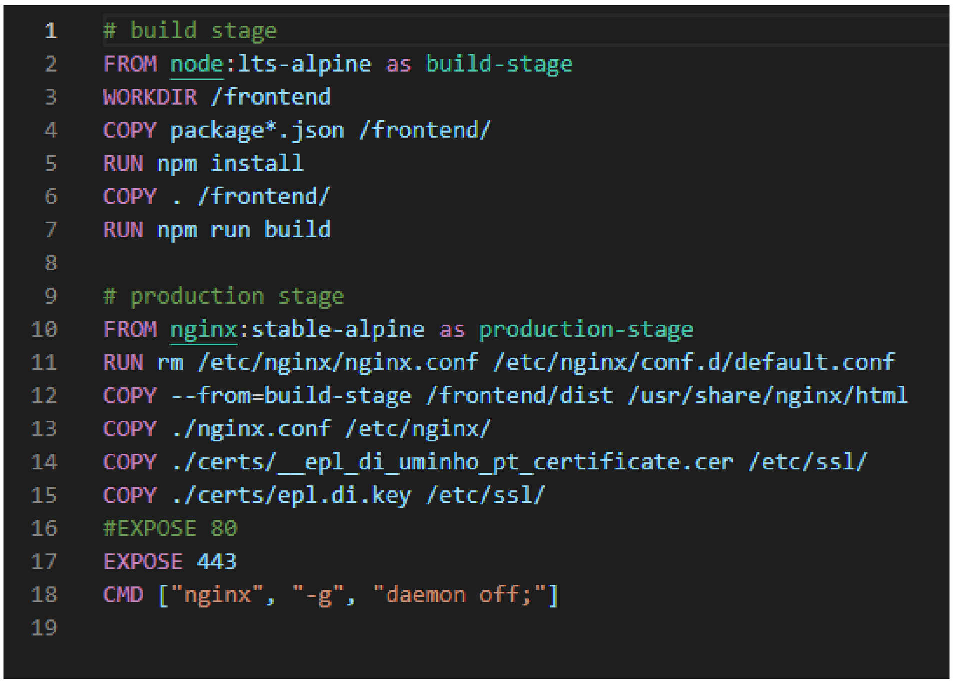
3.6 HTTPS

If you decide to host your web app on a dedicated server, it might be a good step to change from HTTP to HTTPS. With HTTPS, the data sent to and from the server will be encrypted and, therefore, more secure to potential attacks. Doing this change is relatively straightforward but at first can be confusing, so hopefully this part of the guide can help in this process.

The way to do this is by getting a SSL certificate from a trusted source, one that can say your website is trustworthy, which will then make it so accessing your website is done in a secure way by encrypting each data transfer with a public private key pair encryption. One such trusted source is the free certificate authority **Let's Encrypt** (Lets-Encrypt).

After hosting your web application on your dedicated server you can request a certificate from Let's Encrypt and then, simply tell NGINX where that certificate is and your connection will now be made through HTTPS.

It is worth remembering that NGINX is running in a container so when you obtain the SSL certificate (usually a few files that end with cert, crt, key, etc) you need to copy these into the file system of the container, so, backtracking, that is what the final two "COPY" lines were doing in this Dockerfile. (Also, port 443 is the usual port reserved for HTTPS connections, as port 80 is the usual port for HTTP connections)



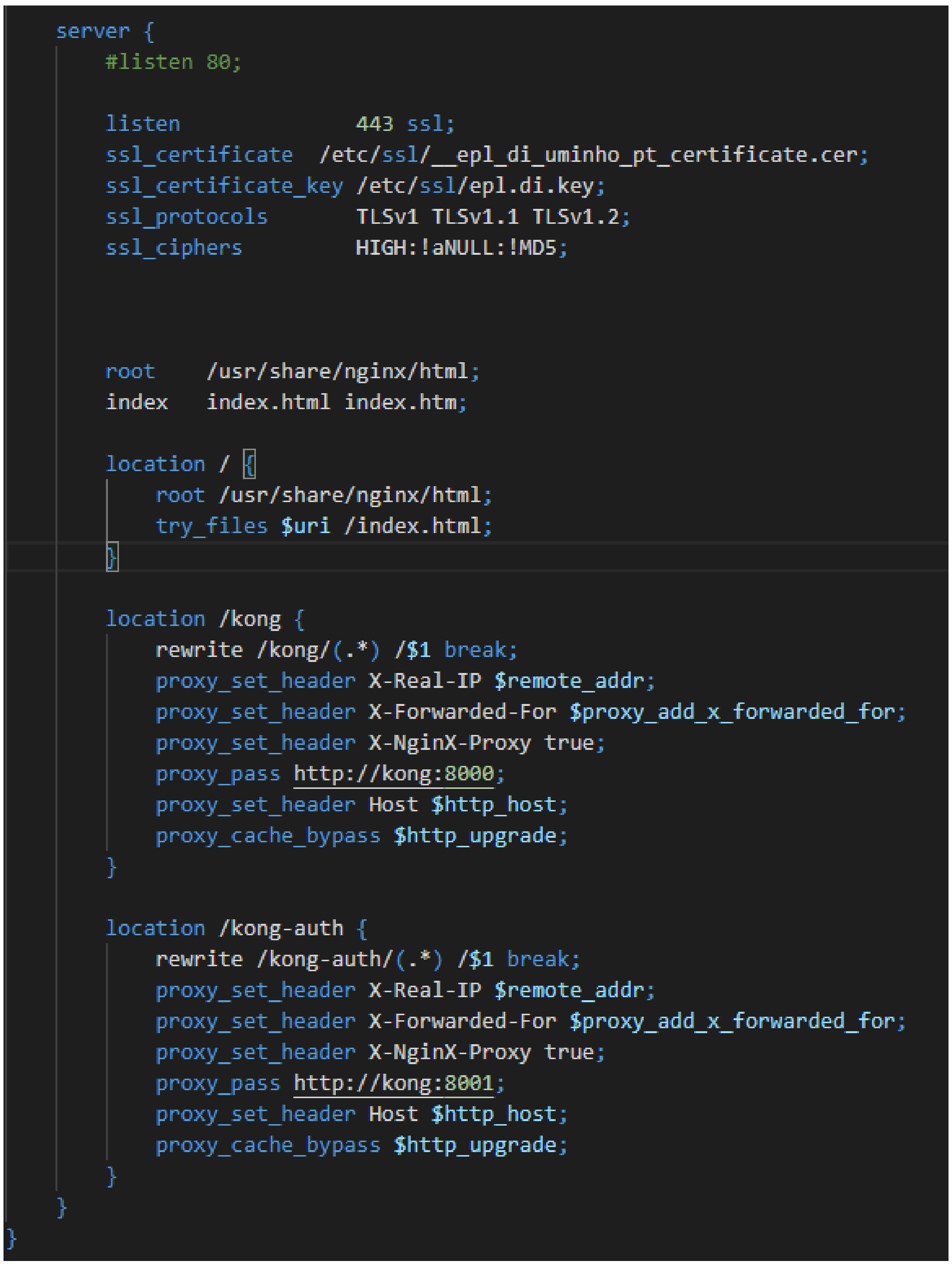
```
# build stage
FROM node:lts-alpine as build-stage
WORKDIR /frontend
COPY package*.json /frontend/
RUN npm install
COPY . /frontend/
RUN npm run build

# production stage
FROM nginx:stable-alpine as production-stage
RUN rm /etc/nginx/nginx.conf /etc/nginx/conf.d/default.conf
COPY --from=build-stage /frontend/dist /usr/share/nginx/html
COPY ./nginx.conf /etc/nginx/
COPY ./certs/__epl_di_uminho_pt_certificate.cer /etc/ssl/
COPY ./certs/epl.di.key /etc/ssl/
#EXPOSE 80
EXPOSE 443
CMD ["nginx", "-g", "daemon off;"]

```

Figure 10: Front-end Dockerfile

The last step is now telling NGINX where these certificates are located (inside the containers file system).



```
server {
    #listen 80;

    listen              443 ssl;
    ssl_certificate   /etc/ssl/__epl_di_uminho_pt_certificate.cer;
    ssl_certificate_key /etc/ssl/epl.di.key;
    ssl_protocols       TLSv1 TLSv1.1 TLSv1.2;
    ssl_ciphers         HIGH:!aNULL:!MD5;



    root    /usr/share/nginx/html;
    index   index.html index.htm;

    location / {
        root /usr/share/nginx/html;
        try_files $uri /index.html;
    }

    location /kong {
        rewrite /kong/(.*) /$1 break;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-NginX-Proxy true;
        proxy_pass http://kong:8000;
        proxy_set_header Host $http_host;
        proxy_cache_bypass $http_upgrade;
    }

    location /kong-auth {
        rewrite /kong-auth/(.*) /$1 break;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-NginX-Proxy true;
        proxy_pass http://kong:8001;
        proxy_set_header Host $http_host;
        proxy_cache_bypass $http_upgrade;
    }
}
}
```

Figure 11: nginx.conf with HTTPS

As the image indicates, the only differences with this file is the **listen 80** line which was changed to **listen 443 ssl**, and the next four lines, of which the first two indicate the location of the certificate files and the last two just are the default configuration in the documentation for HTTPS with NGINX.

3.7 EXAMPLES

Although these "instructions" on how to set up a Microservices Architecture with an API Gateway may be helpful, a real life example should prove even more useful. To that end, a few case studies were undertook to explore the practicality of this Architecture philosophy. The first two apps, which would serve as an introduction to the new technologies used, were the "Equivalencias" app and the "Animals Wiki" app.

3.7.1 *Equivalencias App*

This first app was a simple web app that ran fully with Express on Node.js and was connected to a database in MongoDB, and its purpose was purely to store the grades of students that were changing courses and/or universities, more specifically, the grades that would have equivalents in the other courses.

3.7.1.1 *Introduction to the app*

The intention with this app was to see if it is possible and/or recommended to have your entire main app in a single service (in this case, both the back-end and front-end were in one service) while having a separate service designed for the authentication of the user, which would be needed to be cleared to access the main app.

In other words, when accessing the app, you are confronted with a login page that runs on an isolated service and after clearing it you would be redirected to the proper web page that also ran on its own service (this approach was later heavily used in the last app worked on, the biggest and most complex of them all). Also, the back-end for the authentication part was managed by Kong but the purpose of that was familiarity could be gained with Kong and not managing the communication with that service (because Kong's purpose is to manage a network of communication with many services, managing only one service just isn't necessary), however, doing this allowed Kong to also be partially responsible for the authentication process itself.

3.7.1.2 *Docker*

Going by the order of the previous section of this guide, to deploy this app Docker was used with the following docker-compose configuration:

```

version: "3.7"
services:
  kong:
    container_name: kong
    restart: always
    image: kong:latest
    volumes:
      - ./kong.yml:/usr/local/kong/declarative/kong.yml
    environment:
      - KONG_DATABASE=off
      - KONG_DECLARATIVE_CONFIG=/usr/local/kong/declarative/kong.yml
      - KONG_PROXY_ACCESS_LOG=/dev/stdout
      - KONG_ADMIN_ACCESS_LOG=/dev/stdout
      - KONG_PROXY_ERROR_LOG=/dev/stderr
      - KONG_ADMIN_ERROR_LOG=/dev/stderr
      - KONG_ADMIN_LISTEN=0.0.0.0:8001, 0.0.0.0:8444 ssl
  mongo-equi:
    container_name: mongo-equi
    restart: always
    environment:
      MONGO_INITDB_DATABASE: equivalencias
    image: mongo
    volumes:
      - ./mongo-volume:/data/db
      - ./bds/equivalencias.js:/docker-entrypoint-initdb.d/mongo-init.js:ro
  equi:
    container_name: equi
    build:
      context: ./app
      dockerfile: ./Dockerfile
    restart: always
    links:
      - kong
      - mongo-equi
    ports:
      - "12091:3018"
    networks:
      default:
        aliases:
          - equivalencias
  auth:
    container_name: auth
    build:
      context: ./auth
      dockerfile: ./Dockerfile
    restart: always
    links:
      - mongo-equi
    networks:
      default:
        aliases:
          - auth
  interface-auth:
    container_name: interface-auth
    restart: always
    build:
      context: ./auth-interface
      dockerfile: ./Dockerfile
    ports:
      - "12090:80"
    links:
      - kong
      - equi
      - auth
      - mongo-equi
    networks:
      default:
        aliases:
          - auth
```

As the file shows, there are **five** components that make up this app:

The main app - **equi**, which contains both the front-end and back-end, as previously stated, an interface for user authentication - **interface-auth**, its back-end - **auth**, which serves as the connection to the database - **mongo-equi**, where both the users for authentication and the data for the main app is stored. Finally, there is - **Kong**, which is both acting as a reverse proxy redirecting the authentication interface's calls to the back-end and is also helping in the authentication process itself by providing a unique **key** in the act of a successful login that the browser saves in a cookie. This makes it so that while in the main app if the user does **NOT** have this key, they will be automatically redirected to the authentication interface.

Going deeper into each component's set-up in the docker-compose file:

KONG Kong's set-up is the default one that has been used up until now.

MONGO-EQUI This is the container for the database, for which MongoDB was used. Here we are setting up the database name as an environment variable, "equivalencias", and also setting up two volumes:

- **mongo-volume** to take care of data persistence on eventual container restarts;
- **equivalencias.js** file which has all of the data to populate the database on the first time the container is run

EQUI This container holds the main app. Here docker is instructed to run the app, which is running on port 3018 on the container, on port 12091 on the machine. Docker is also instructed to wait for Kong and mongo-equi to start running before starting this container. Lastly, Docker is informed that there is a Dockerfile to run when building the container. Here is that Dockerfile:

```

#Image
FROM node:15

#Create folder
WORKDIR /equi

#Copy app and install packages
COPY package.json /equi/
COPY package-lock.json /equi/
RUN npm install
COPY . /equi/

#Expose port
EXPOSE 3018

#Run app
CMD [ "npm", "start" ]
```

Here the image for node is pulled, the app is copied over to the created folder in the container and then the app is run on port 3018.

AUTH This section of the docker-compose file, similarly to the previous container, is only instructing Docker to build this container after mongo-equi is built and also declares the path to the Dockerfile which is inside the "auth" folder. Here is the Dockerfile:

```

#Image
FROM node:15

#Create folder
WORKDIR /auth

#Copy app and install packages
COPY package.json /auth/
COPY package-lock.json /auth/
RUN npm install
COPY . /auth/

#Expose port
EXPOSE 9000

#Run app
CMD [ "npm", "start" ]
```

This Dockerfile works the exact same way as the previous container, which makes sense as it is also a server running on Express.

INTERFACE-AUTH Once again, this container works in a similar way to the previous containers in the way that the docker-compose file is only instructing Docker of the following: the **port** to which app will be exposed to, the **order** in which to built this container, and where the **Dockerfile** is located. However, this time the Dockerfile is different to the previous containers' Dockerfiles:

```

# build stage
FROM node:lts-alpine as build-stage
WORKDIR /front-end
COPY package*.json /front-end/
RUN npm install
COPY . /front-end/
RUN npm run build

# production stage
FROM nginx:stable-alpine as production-stage
RUN rm /etc/nginx/nginx.conf /etc/nginx/conf.d/default.conf
COPY --from=build-stage /front-end/dist /usr/share/nginx/html
COPY ./nginx.conf /etc/nginx/
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

The first half of this Dockerfile works like the other Dockerfiles, getting the image for node, copying the app and installing packages and then running it. However, this particular front-end will also be run using NGINX, so in the second half the following happens:

- The image for NGINX is pulled;
- The default configuration is removed;
- The app is copied from the build stage;
- The custom NGINX configuration is copied over (this particular file will be covered in a later section);
- The port is exposed;
- NGINX is run.

Finally, it is worth noting that in all of the containers the option "restart: always" is enabled, which, as previously mentioned, makes it so in case the container shuts down for any reason it will restart automatically, avoiding down-times.

3.7.1.3 *Kong*

As previously stated, Kong's job in this particular app's architecture is not only to act as a reverse proxy between the authentication interface and it's back-end, but also to try and manage, in some capacity, part of the authentication process itself, by providing a unique key to the client (which then saves that key in his/her cookies) and then checking if the user is in the possession of said key. If the user does ***not*** have the key that means he/she tried to bypass the authentication process.

To this end, the following configuration was implemented for Kong:

```
_format_version: "2.1"

services: <-----
  - name: auth_service
    url: http://auth:9000
    routes:
      - name: auth-route
        paths:
          - /auth


consumers: <-----
  - username: user1
keyauth_credentials: <-----
  - consumer: user1
```

Here exactly two things are happening:

- A **service** was defined, meaning that to reach HTTP://AUTH:9000 (the authentication back-end container) one can use the path "/auth" in conjunction with Kong's url: HTTP://KONG:8000/AUTH
- A **"consumer"** and his **credentials** were created. A consumer is what consumes the service, so adding it to the configuration allows Kong to identify the user, which would in turn make it possible so that by attaching plugins to said consumer Kong becomes able to control the user with said plugins, like more complex authentication or rate-limiting.

In this case though, no plugins are being used, because the app is only looking for the generated **key** which is in the consumers **credentials**. After a successful login a request will be made to HTTP://KONG:8001/CONSUMERS/USER1/KEY-AUTH (user1 being the created consumer) which will return the consumer object with the key inside.

3.7.1.4 *NGINX*

NGINX is absolutely not necessary for the success of this app, but because it would be crucial in the development of the next app, it was used in this much more simple app so it later could be used much more naturally. Nevertheless, NGINX is also working as a reverse proxy, this time between the user's machine and the front-end itself, making it so this is the actual flow of a request call:

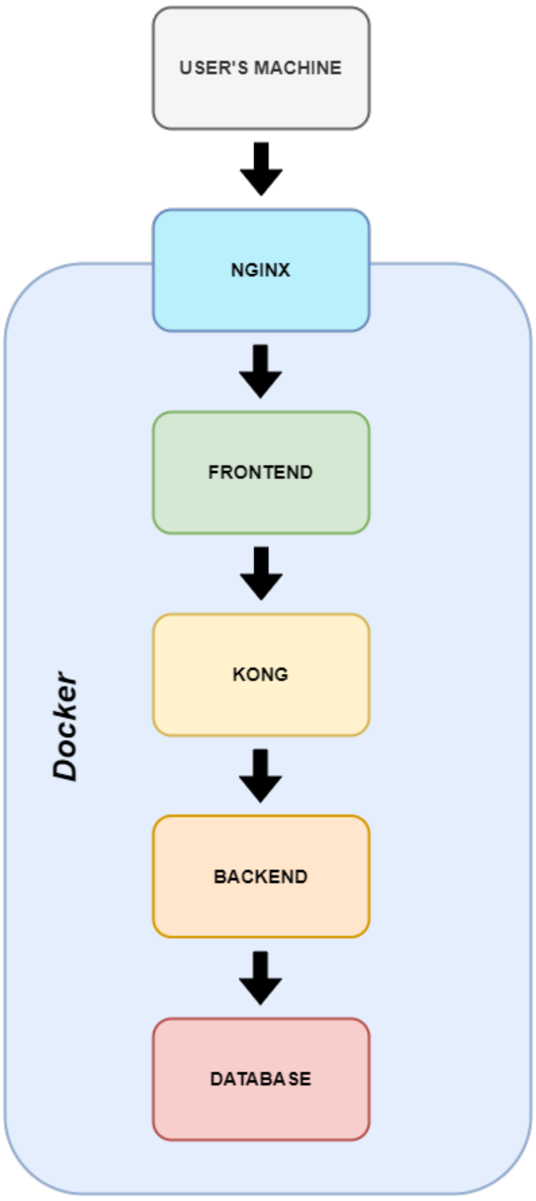


Figure 12: Request flow

Particularly in this web application, NGINX is redirecting all calls made to specific paths to the appropriate services, in this case Kong. This was achieved by modifying the NGINX config file - *nginx.conf*:

```
user  nginx;
worker_processes  auto;

error_log  /var/log/nginx/error.log notice;
pid        /var/run/nginx.pid;


events {
    worker_connections  1024;
}


http {
    include       /etc/nginx/mime.types;
    default_type  application/octet-stream;

    log_format  main  '$remote_addr - $remote_user [$time_local] "$request" '
                      '$status $body_bytes_sent "$http_referer" '
                      '"$http_user_agent" "$http_x_forwarded_for"';

    access_log  /var/log/nginx/access.log  main;

    sendfile        on;
    #tcp_nopush     on;

    keepalive_timeout  65;

    #gzip  on;

    include /etc/nginx/conf.d/*.conf;

    server {
        listen 80;

        root    /usr/share/nginx/html;
        index   index.html index.htm;

        location / {
            root /usr/share/nginx/html;
            try_files $uri /index.html;
        }

        location /kong { <-----
            rewrite /kong/(.*) /$1 break;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
            proxy_set_header X-NginX-Proxy true;
            proxy_pass http://kong:8000;
            proxy_set_header Host $http_host;
            proxy_cache_bypass $http_upgrade;
        }

        location /kong-auth { <-----
            rewrite /kong-auth/(.*) /$1 break;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
            proxy_set_header X-NginX-Proxy true;
            proxy_pass http://kong:8001;
            proxy_set_header Host $http_host;
            proxy_cache_bypass $http_upgrade;
        }
    }
}
```

This configuration file is quite similar to the default configuration for an NGINX server, however it does have a crucial difference: the last two **location blocks**. These two blocks are what redirects the calls made to those specific url paths:

- /KONG will redirect those calls to HTTP://KONG:8000, the actual kong service that acts as a reverse proxy.
- /KONG-AUTH will redirect those calls to HTTP://KONG:8001, which is the admin port for Kong, having many functionalities but in this case it is used to get the consumer's credentials.

3.7.1.5 *The Approach and Difficulties found*

Having visited every part of what makes this web application function in a microservices environment, in this section we will discuss how the idea for the architecture was conceived, the process of implementing it and the difficulties found along the way.

It was known when first experimenting with web applications in this thesis that the end result was a Microservices architecture, so the natural first step in the transformation of this web app was the *Dockerization* of the main app "equivalencias" so it becomes a service capable of running on any machine. At the same time, the other objective was introducing **Kong** into this network of services, so in addition to the main app the default configuration of Kong was added to the *docker-compose* file and a simple service was set up. However, at this point there were no services defined in the *kong.yml* file as there were no services that we **could** define in the *kong.yml* file (except maybe the main app but the reason that was not done will be discussed later in this thesis). After setting up these two services this was the docker-compose file.

```
version: "3.7"
services:
  kong:
    container_name: kong
    restart: always
    image: kong:latest
    volumes:
      - ./kong.yml:/usr/local/kong/declarative/kong.yml
    environment:
      - KONG_DATABASE=off
      - KONG_DECLARATIVE_CONFIG=/usr/local/kong/declarative/kong.yml
      - KONG_PROXY_ACCESS_LOG=/dev/stdout
      - KONG_ADMIN_ACCESS_LOG=/dev/stdout
      - KONG_PROXY_ERROR_LOG=/dev/stderr
      - KONG_ADMIN_ERROR_LOG=/dev/stderr
      - KONG_ADMIN_LISTEN=0.0.0.0:8001, 0.0.0.0:8444 ssl
  equi:
    container_name: equi
    build:
      context: ./app
      dockerfile: ./Dockerfile
    restart: always
    links:
      - kong
    ports:
      - "12091:3018"
    networks:
      default:
        aliases:
```

```
        – equivalencias
```

Although the goal of this thesis was to see how Kong behaves in a Microservices architectural environment, one of the main goals for **this particular experiment** was to see if it was possible to have a separate, isolated, authentication service to the app itself, of which Kong would be used to make sure the user was authenticated for the main app itself. To this end, a web interface with **Vue.js** was created, which was based on another app, one that would also be used later in this thesis - *Animals Wiki*. The homepage for that app with a simple login form was then used as an authentication interface.

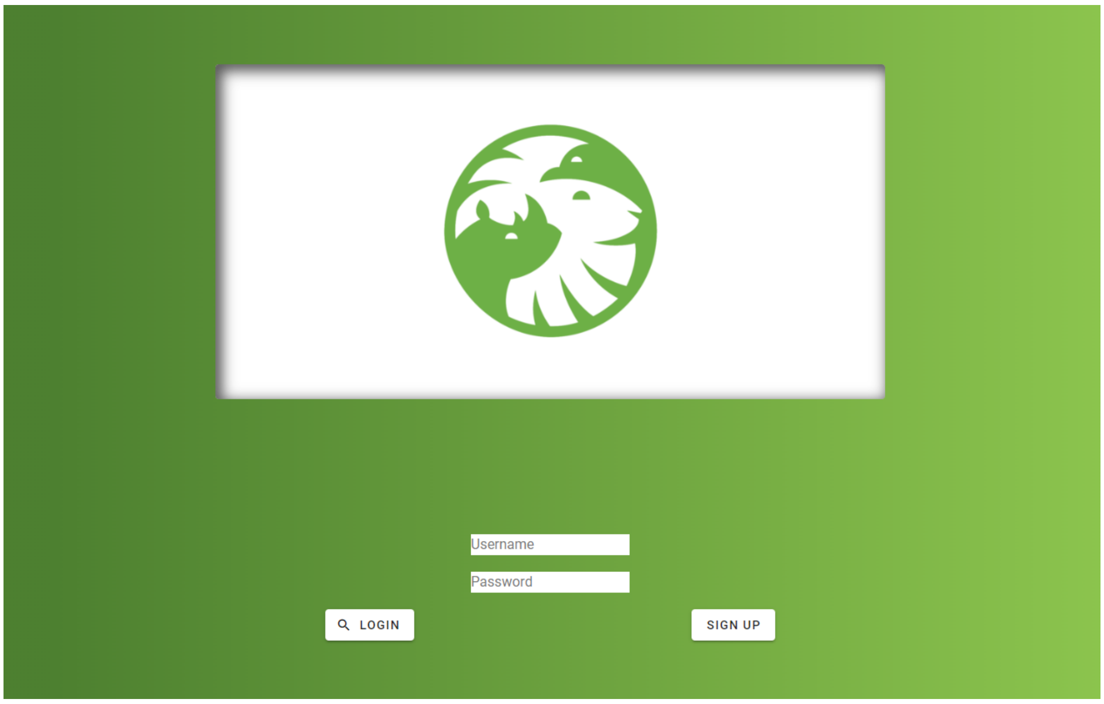


Figure 13: Front-end Equivalencias App

For the back-end, **Express** was used, a Node.js web application framework, to quickly set up a server that handled user authentication. To do this, a **MongoDB** database was set up, it was then connected to the back-end and then the back-end checked if the user existed and the password matched, and if both of those conditions were true it returned a code status 201 (meaning a successful login), and if not it returned a code 403 (meaning the credentials were incorrect).

```
router.post('/login', function(req, res){
  User.consultar(req.body.username)
    .then(dados => {
      if(!dados) res.status(403).jsonp(false)
      else if(req.body.password != dados.password) res.status(403).jsonp(false)
      else return res.status(201).jsonp(true)
    })
    .catch(err => res.status(500).jsonp({err: err}))
})
```

After this, these two services were added to the docker-compose file to introduce them to the network.

However, Kong still had to be used in the authentication process, so two tasks were defined: **First**, define a service for the back-end component, just so Kong would be used as a reverse proxy thus increasing the developer's understanding of how the API Gateway works. **Second**, the consumer credentials would be used to provide a unique key to the user so that only in possession of said key they would be able to access to the main app. This is what the kong.yml file became:

```
_format_version: "2.1"

services:
  - name: auth_service <-----
    url: http://auth:9000
    routes:
      - name: auth-route
        paths:
          - /auth

consumers: <-----
  - username: user1
keyauth_credentials: <-----
  - consumer: user1
```

And this function handles the login on the front-end saving the unique key in a cookie:

```
getkey: function () {
    axios({
        method: 'post',
        url: '/kong/auth/users/login',
        data: {
            username: this.username,
            password: this.password
        }
        })
        .then(data => {
            if (data.data) {
                axios
                    .get('/kong-auth/consumers/user1/key-auth')
                    .then(response => {
                ----->  Vue.$cookies.set("key", response.data.data[0].key, "1d")
                        this.$router.push('/')
                        })
            }
        })
        .catch(err => console.log(err))
```

After setting up the main app and the authentication service, a database system had to be set up for both storing the data that the main app uses, but also the users for the authentication system. Consequently, the configuration for **MongoDB** was added to the docker-compose file:

```
  mongo-equi:
    container_name: mongo-equi
    restart: always
    environment:
      MONGO_INITDB_DATABASE: equivalencias
    image: mongo
    volumes:
      - ./mongo-volume:/data/db
      - ./bds/equivalencias.js:/docker-entrypoint-initdb.d/mongo-init.js:ro
```

However, it is worth nothing the last block of code regarding **volumes**. Volumes in Docker are used for ***data persistence***.

Circling back to the volumes block in the docker-compose file, we then have these two lines:

- ./MONGO-VOLUME:/DATA/DB is responsible for the data persistence in this container because mongo-volume is the folder with all of MongoDB's data in the host system and /data/db is the path in the container where the data is stored by default.
- ./BDS/EQUIVALENCIAS.JS:/DOCKER-ENTRYPOINT-INITDB.D/MONGO-INIT.JS:RO however, is only responsible for the initial import of data if the container has never been run.

When this app was first tackled it already had a lot of data, so a way to import all of it to the container had to be figured out. After some research, it was discovered that the code in the docker-entrypoint-initdb.d folder is only executed if the container has never been run before, so the equivalencias.js file which would import all of the existing data into the database was created.

Subsequently to setting up the database, we had:

- The main app
- The front-end and back-end to the authentication service
- Kong
- The Database MongoDB

Previously we discussed the possibility of defining a service in Kong for the main app, and the initial plan to ensure the user was authenticated was to get the main app to be a service and protecting it with an authentication plugin from Kong, but quickly it became clear that that would be very unpractical. It is very possible to set it up as a Kong service and initially getting on the main app itself would be very simple. But because when traversing the main app it would reference **itself** and not **Kong's service version of itself**, the whole application would have to be restructured so when traversing it one would not go to HTTP://EQUI:12091 but instead to HTTP://KONG:8000/MAIN-APP. As a result, a different approach was taken to resolve this problem. Instead of this initial method, a key would be attributed to a user that passed the login phase, and every time that user attempted to access the main website with any and all requests, the server would contact Kong and check if the user's key was correct. If it failed the check the user would be immediately prohibited from the website and redirected to the login page.

This became then, the definite version of the Microservices Architecture for this web application:

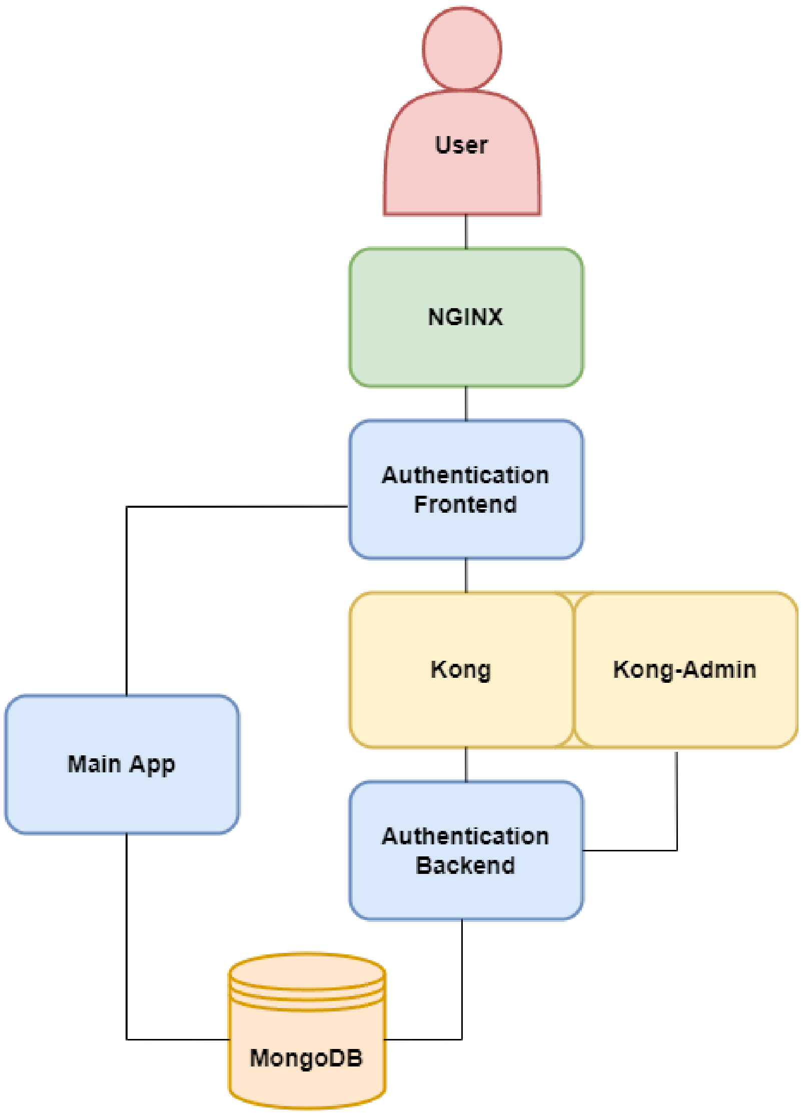


Figure 14: Equivalencias Microservices Architecture

And this is a flowchart of the user experience:

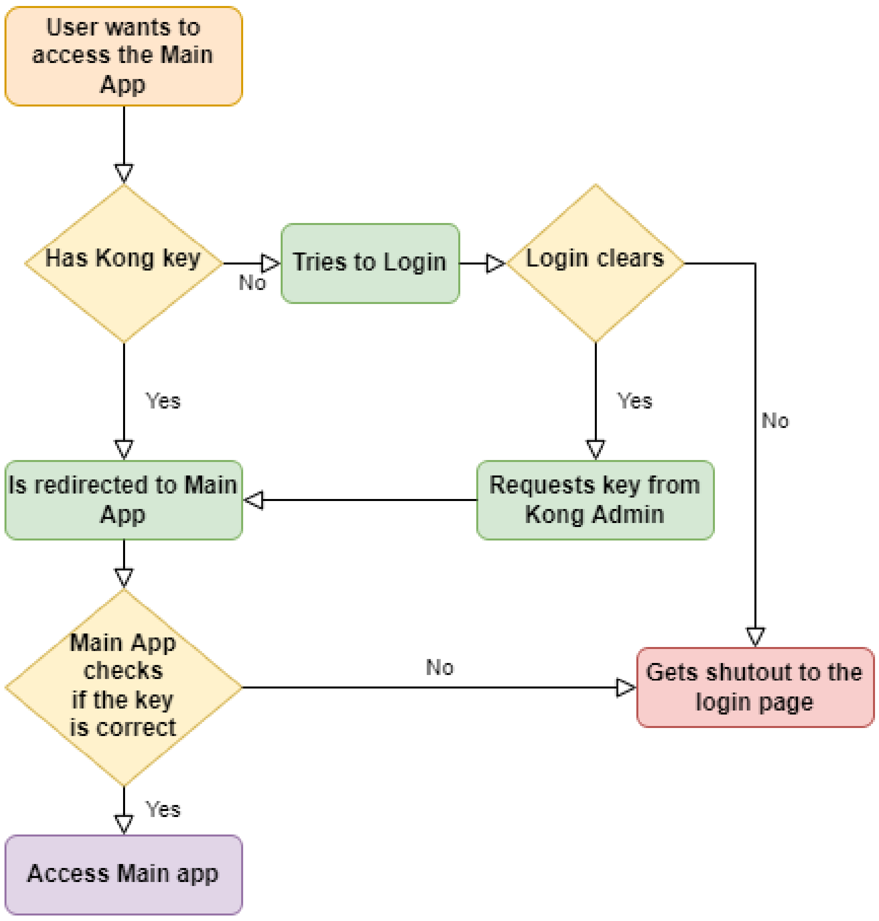

User wants to access the Main App
Has Kong key
No
Tries to Login
Login clears
Yes
Yes
No
Is redirected to Main App
Requests key from Kong Admin
Main App checks if the key is correct
No
Gets shutout to the login page
Yes
Access Main app

Figure 15: Equivalencias Request Flowchart

3.7.2 *Animals Wiki App*

This second app tackled a previously talked about project: Animals-wiki. It consisted on a sort of library of animals, where one could look up their description, where they can be found, their predators and prey, etc. This information was taken from another website, AZ_Animals, and then transformed into Turtle and imported into GraphDB.

3.7.2.1 *Introduction to the app*

This particular web application had a few objectives:

- Using **Kong's** functionalities to a bigger extent than the previous app, mainly having more ***Kong services*** and using ***plugins*** for authentication;
- Try to set up the website with a HTTPS connection, making the connection more secure, and hosting it on a dedicated server.

It is worth noting that, contrary to the last app, this web app runs its front-end in a single Docker container that handles the authentication process and the animals-wiki app itself. It is all a single application. To sum up, there is a single front-end container for the whole app, this container talks to a back-end service that handles authentication and a back-end service that handles the animal data, both of which are reached through Kong. Additionally, the service that handles the animal data can only be reached if the authentication handled with the plugin is cleared.

This website is being run on a dedicated server with HTTPS here: (Animals-Wiki)

3.7.2.2 *Docker*

Once again, the docker-compose configuration used in the deployment of this app is as follows:

```
version: "3.7"
services:
  kong:
    container_name: kong
    restart: always
    image: kong:latest
    volumes:
      - ./kong.yml:/usr/local/kong/declarative/kong.yml
    environment:
      - KONG_DATABASE=off
      - KONG_DECLARATIVE_CONFIG=/usr/local/kong/declarative/kong.yml
      - KONG_PROXY_ACCESS_LOG=/dev/stdout
      - KONG_ADMIN_ACCESS_LOG=/dev/stdout
      - KONG_PROXY_ERROR_LOG=/dev/stderr
      - KONG_ADMIN_ERROR_LOG=/dev/stderr
      - KONG_ADMIN_LISTEN=0.0.0.0:8001, 0.0.0.0:8444 ssl
  mongo-users:
    container_name: mongo-users
    restart: always
    environment:
      MONGO_INITDB_DATABASE: Animals-auth
    image: mongo
    volumes:
      - ./mongo-volume:/data/db
  api:
    container_name: api
    build:
      context: ./API
      dockerfile: ./Dockerfile
    restart: always
    links:
      - graphdb
    networks:
      default:
        aliases:
          - front-end-animals
  auth:
    container_name: auth
    build:
      context: ./autenticacao
      dockerfile: ./Dockerfile
    restart: always
    links:
      - mongo-users
    networks:
      default:
        aliases:
          - auth
  front-end-animals:
    container_name: front-end-animals
    restart: always
    build:
      context: ./front-end/animals
      dockerfile: ./Dockerfile
    ports:
      #- "12090:80"
      - "12090:443"
    links:
      - kong
      - api
      - auth
      - mongo-users
      - graphdb
    networks:
      default:
        aliases:
          - auth
  graphdb:
    container_name: graphdb
    image: khaller/graphdb-free:9.10.0
    ports:
      - "9200:7200"
    build:
      context: ./graphdb
      dockerfile: ./Dockerfile
    restart: always
```

In this app, there are **six** components that make up this app: - **Kong**, acting as a reverse proxy, as per usual, and also being responsible for protecting the core part of the website from being reached by unauthenticated users, - **mongo-users and graphdb**, where the data for the users and animal data is being stored, respectively, - **auth and api**, the back-end services, running on *Express*, for the user authentication and the core app respectively, and lastly - **front-end-animals**, the front-end of the whole web app, running on Vue.js. It is worth noting that, as always, every one of these containers will restart in case of failure.

Diving deeper into each container:

KONG Kong's set-up is the default one that has been used up until now.

AUTH This container is mainly built through the Dockerfile, but it is worth mentioning the *links* block because it is setting up the build so that this container will only start after mongo-users is up and running, as it expresses dependency between the services as well as links them.

Going through the Dockerfile:

```
#Base Image
FROM node:15

#Create work directory
WORKDIR /auth

#Copy the app and install dependencies
COPY package.json /auth/
COPY package-lock.json /auth/
RUN npm install
COPY . /auth/

#Expose port
EXPOSE 9000

#Start the app
CMD [ "npm", "start" ]
```

As is documented, here the app is being copied to the container, dependencies are being installed and finally the app is being run on the exposed port.

API This container functions the exact same way as the previous one.

Here is the Dockerfile:

```
#Base Image
FROM node:15

#Create work directory
WORKDIR /api

#Copy the app and install dependencies
COPY package.json /api/
COPY package-lock.json /api/
RUN npm install
```

```
COPY . /api/

#Expose port
EXPOSE 7777

#Start the app
CMD [ "npm", "start" ]
```

MONGO-USERS This is the default configuration for a MongoDB based container, with the name for the database being Animals-auth and a volume for data persistence being set up through - ./MONGO–VOLUME:/DATA/DB

GRAPHDB GraphDB is not a very common choice for a database, so setting up this container was not very straightforward. Firstly, it is a **graph oriented database** which is already not the most popular type of database. Secondly, even between graph oriented databases GraphDB is not the top choice. This makes it so there might not be as much resources available online, like Docker support. There is an official Docker image for GraphDB provided by Ontotext on Dockerhub, but not for the free version of GraphDB. However, there is a Dockerfile for the free version on their Github, but this Dockerfile was not quite what was being looking for. Instead, Khaller's image for GraphDB was used as it allowed for an easy way to initialize repositories.

Thus, in the docker-compose file the image is being pulled from KHALLER/GRAPHDB-FREE:9.10.0. Also, port 9200 is being exposed in the host machine from port 7200 in the container. This so the Workbench mode could be reached. However, the image was not enough as the repository would have to be set up and the existing data would have to be imported. Therefore, this Dockerfile was configured:

```
FROM khaller/graphdb-free:9.10.0

ENV GDB_HEAP_SIZE=2G

RUN mkdir -p /repository.init/Animals

RUN mkdir -p /temporary-folder

COPY animals-complete.ttl /repository.init/Animals/toLoad/animals-complete.ttl

COPY config.ttl /repository.init/Animals/config.ttl
```

In here the heap size is being set to two gigabytes because the file with the animal data was so big the container would crash while trying to import it. Then, two folders are being created - *repository.init* is of special importance. According to the image documentation from khaller, this folder is where the repository initialization must happen. Inside the /repository.init/Animals folder the config file for the repository must be present, and also another folder called *toLoad* must be present with the *Turtle* file with all of the animal data inside. Hence, the last two lines.

Lastly, looking inside the repository configuration file:

```
# Configuration template for a GraphDB-SE repository
#
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix rep: <http://www.openrdf.org/config/repository#>.
@prefix sr: <http://www.openrdf.org/config/repository/sail#>.
@prefix sail: <http://www.openrdf.org/config/sail#>.
@prefix owlim: <http://www.ontotext.com/trree/owlim#>.

[] a rep:Repository ;
    rep:repositoryID "animals" ; <-----
    rdfs:label "Animals repo" ; <-----
    rep:repositoryImpl [
        rep:repositoryType "graphdb:FreeSailRepository" ;
        sr:sailImpl [
            sail:sailType "graphdb:FreeSail" ;

                        # ruleset to use
                        owlim:ruleset "empty" ;

                        # disable context index(because my data do not uses
    contexts)
                        owlim:enable-context-index "false" ;

                        # indexes to speed up the read queries
                        owlim:enablePredicateList "true" ;
                        owlim:enable-literal-index "true" ;
                        owlim:in-memory-literal-properties "true" ;
        ]
    ].
```

This is a default configuration for a GraphDB repository, everything expect for the repositoryID and label, which define the id and name of the repository, respectively.

FRONT-END-ANIMALS Lastly, the container for the front-end of the app. It is noteworthy to look at the links block as it is set up so front-end-animals will wait for every other container to be up and running before starting. The port is also being exposed to 12090 to the host machine from 443 in the container. It's 443 because that is the default port for websites running on a HTTPS connection, contrary to HTTP servers which run on port 80. Going through the Dockerfile:

```
# build stage
FROM node:lts-alpine as build-stage
WORKDIR /front-end
COPY package*.json /front-end/
RUN npm install
COPY . /front-end/
RUN npm run build

# production stage
FROM nginx:stable-alpine as production-stage
RUN rm /etc/nginx/nginx.conf /etc/nginx/conf.d/default.conf
COPY --from=build-stage /front-end/dist /usr/share/nginx/html
COPY ./nginx.conf /etc/nginx/
COPY ./certs/__epl_di_uminho_pt_certificate.cer /etc/ssl/
COPY ./certs/epl.di.key /etc/ssl/
#EXPOSE 80
EXPOSE 443
CMD ["nginx", "-g", "daemon off;"]
```

The first half is as usual, installing the Vue.js app and running it. Although, in the second half NGINX is being set up. The details for this configuration will be explored further ahead but, aside from the two *COPY ./certs/* lines which will also be explored further ahead in the HTTPS section, this configuration is default and the one used in the previously discussed app.

3.7.2.3 *Kong*

Kong in this web app functions, as usual, as a reverse proxy. But, contrary to the last app - equivalencias, this time it is being used to a bigger extent. Last time it only served as a sort of middle-point between the front-end and the back-end of the authentication service. This time however, it now deals with **two services**, the back-end of the authentication service **and** the back-end of the core app, the service that deals with the retrieval of the animal data. Furthermore, Kong will also use one of the plugins from its extensive library to build a "authentication wall" between the front-end and the back-end of the core app, so a user can only access that information if they are indeed authenticated.

To this end, the following configuration was implemented for Kong:

```
_format_version: "2.1"

services:
  - name: animals-service <-----
    url: http://api:7777
    routes:
      - name: animals-route
        paths:
          - /animals-api
  - name: auth_service <-----
    url: http://auth:9000
    routes:
      - name: auth-route
        paths:
          - /auth

plugins:
  - name: key-auth <-----
    service: animals-service
    config:
      key_names:
      - apikey
      key_in_body: true
      key_in_query: true
consumers:
  - username: user1
keyauth_credentials:
  - consumer: user1
```

As is customary, the services block details Kong's services and, in this case, as was said, the services for the authentication back-end and core app back-end are present. These could be reached by accessing HTTP://AUTH:9000 for the authentication back-end and HTTP://API:7777 for the core app back-end. Instead however, they can now be reached through HTTP://KONG:8000/AUTH and HTTP://KONG:8000/ANIMALS-API respectively.

Also, a new block can be observed: the ***plugins*** block. Here one has to define:

- The **id of the service**, in this case the Key_Authentication plugin was used, which has the id *key-auth*;
- The specific **service** it is targeting (it can also target an individual route or be applied globally to all services and routes);

- The **configuration** for the plugin. In this case, it was defined that the variable name for the key would be *apikey* and when sending a request the key must be either in the body or in the query.

Lastly, exactly as it was in the previous app, there is a consumers and credentials block, which is necessary for a plugin like Key Authentication to work.

3.7.2.4 *NGINX and HTTPS*

As was said in this section on the equivalencias app, NGINX was **crucial** in the development of this app, this because it (or another server like Apache) is absolutely necessary if one intends to set up an HTTPS connection. Evidently, an HTTPS connection is in no way necessary for an app to **function**, but if the intent is for it to be used by anyone, getting a domain and hosting it on a server is a necessary step, as is protecting the connection between the client and the server with an HTTPS protocol. Nevertheless, NGINX still functions as the usual reverse proxy and so, the request call flow is still this:

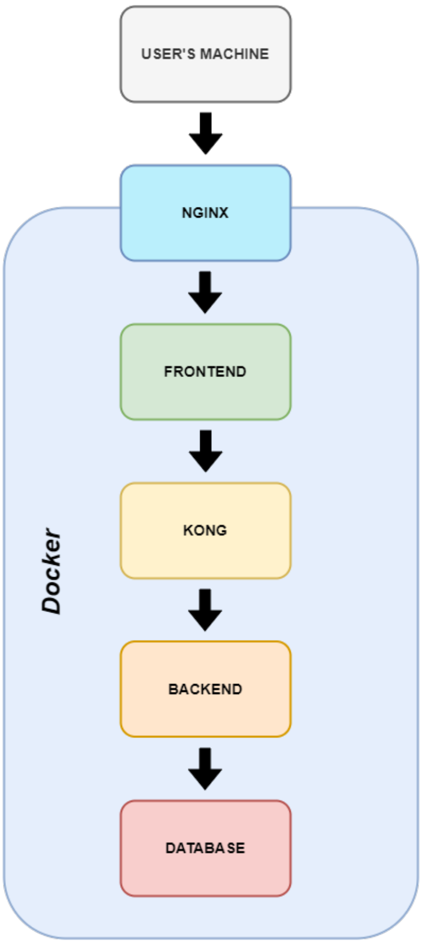


Figure 16: Request flow

Diving deeper into the NGINX configuration file - *nginx.conf*:

```
user  nginx;
worker_processes  auto;

error_log  /var/log/nginx/error.log notice;
pid        /var/run/nginx.pid;


events {
    worker_connections  1024;
}


http {
    include       /etc/nginx/mime.types;
    default_type  application/octet-stream;

    log_format  main  '$remote_addr - $remote_user [$time_local] "$request" '
                      '$status $body_bytes_sent "$http_referer" '
                      '"$http_user_agent" "$http_x_forwarded_for"';

    access_log  /var/log/nginx/access.log  main;

    sendfile        on;
    #tcp_nopush     on;

    keepalive_timeout  65;

    #gzip  on;

    include /etc/nginx/conf.d/*.conf;

    server {
        #listen 80;

  ----> listen              443 ssl;
  ----> ssl_certificate  /etc/ssl/__epl_di_uminho_pt_certificate.cer;
  ----> ssl_certificate_key /etc/ssl/epl.di.key;
  ----> ssl_protocols       TLSv1 TLSv1.1 TLSv1.2;
  ----> ssl_ciphers         HIGH:!aNULL:!MD5;



        root    /usr/share/nginx/html;
        index   index.html index.htm;

        location / {
            root /usr/share/nginx/html;
            try_files $uri /index.html;
        }

  ----> location ~ ^/\.well-known/acme-challenge/([-_a-zA-Z0-9]+)$ {
          default_type text/plain;
          return 200 "$1.lnFOHBDiobOZuZYejG-6m1It5J-xVCzsyL5XoyLJKIc";
        }

        location /kong {
            rewrite /kong/(.*) /$1 break;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
            proxy_set_header X-NginX-Proxy true;
            proxy_pass http://kong:8000;
            proxy_set_header Host $http_host;
            proxy_cache_bypass $http_upgrade;
        }

        location /kong-auth {
            rewrite /kong-auth/(.*) /$1 break;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
            proxy_set_header X-NginX-Proxy true;
            proxy_pass http://kong:8001;
            proxy_set_header Host $http_host;
            proxy_cache_bypass $http_upgrade;
        }
    }
}
```

As it was in the previous app, most of the configuration is the default configuration for an NGINX server, with the addition to the two location blocks: the */kong* and */kong-auth* location blocks. This time however, a few other lines of code or added for the set up of the HTTPS server:

- The *listen 80;* line was commented and substituted by *listen 443 ssl;* as this is the customary port for a HTTPS connection;
- The *ssl_certificate*, *ssl_certificate_key*, *ssl_protocols* and *ssl_ciphers* were added. These lines of code are part of a HTTPS NGINX default configuration and can be consulted here (NGINX_HTTPS), and, most importantly, the ssl_certificate and ssl_certificate_key define where the ***certificate*** and ***certificate key*** are located;

- The *location ~^/\.well-known/acme-challenge/* block was added. In this app, to get HTTPS certificates Lets-Encrypt was used, and *Let's Encrypt* uses something called an **ACME challenge** to figure out if we, the developer, are in control of the domain name we intend to host our server on. To do this, *Let's Encrypt* provides the ACME client with a token and places a file on http://<YOUR_DOMAIN>/.well-known/acme-challenge/<TOKEN>. A validation will then start where Let's Encrypt will try to retrieve this file and, if successful, the HTTPS connection will be established. So, to allow for this validation, this location block was added.

This begs the question however, how to get a certificate from *Let's Encrypt*. In this particular case, the website was being hosted on UMinho's servers which already had certificates issued for *.epl.di.uminho.pt (the website is hosted on gate.epl.di.uminho.pt so this certificate was valid), but new certificates can easily be issued by using the *Certbot ACME Client* (Certbot)

3.7.2.5 *The Approach and Difficulties found*

Just like for the last app, in this section we will dive deeper into how the architecture was conceived, the development process and the problems that had to be overcome. The main goals this time, as previously said, were to take more advantage of Kong's capabilities by using plugins and managing more services. So, like last time, the first step was the *Dockerization* of the app and, with the experience gained from the last app, this was mostly straightforward. As always, the default configuration for Kong was added to the docker-compose file as it is always the same, and a kong.yml file that could later be modified to take full advantage of Kong was also created. The configuration for the already existing app animals-wiki was also added, both the front-end and the back-end (at this point it only consisted of a menu that would take you to the list of animals):

```
version: "3.7"
services:
  kong:
    container_name: kong
    restart: always
    image: kong:latest
    volumes:
      - ./kong.yml:/usr/local/kong/declarative/kong.yml
    environment:
      - KONG_DATABASE=off
      - KONG_DECLARATIVE_CONFIG=/usr/local/kong/declarative/kong.yml
      - KONG_PROXY_ACCESS_LOG=/dev/stdout
      - KONG_ADMIN_ACCESS_LOG=/dev/stdout
      - KONG_PROXY_ERROR_LOG=/dev/stderr
      - KONG_ADMIN_ERROR_LOG=/dev/stderr
      - KONG_ADMIN_LISTEN=0.0.0.0:8001, 0.0.0.0:8444 ssl
  api:
    container_name: api
    build:
      context: ./API
      dockerfile: ./Dockerfile
    restart: always
    networks:
      default:
        aliases:
          - front-end-animals
  front-end-animals:
    container_name: front-end-animals
    restart: always
    build:
      context: ./front-end/animals
      dockerfile: ./Dockerfile
    ports:
      - "12090:80"
    links:
      - kong
      - api
    networks:
      default:
        aliases:
          - auth
```

As expected, the Dockerfiles are consistent as with every other configuration addressed in this thesis. This is the Back-end's Dockerfile:

```
#Base Image
FROM node:15

#Create folder
WORKDIR /api

#Copy app and install packages
COPY package.json /api/
COPY package-lock.json /api/
RUN npm install
COPY . /api/

#Expose port
EXPOSE 7777

#Run app
CMD [ "npm", "start" ]
```

And this is the Front-end's Dockerfile:

```
# build stage
FROM node:lts-alpine as build-stage
WORKDIR /frontend
COPY package*.json /frontend/
RUN npm install
COPY . /frontend/
RUN npm run build

# production stage
FROM nginx:stable-alpine as production-stage
RUN rm /etc/nginx/nginx.conf /etc/nginx/conf.d/default.conf
COPY --from=build-stage /frontend/dist /usr/share/nginx/html
COPY ./nginx.conf /etc/nginx/
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

Obviously though, this website needs a database to store the animal information. This, however, revealed itself to be more problematic than first thought. The database software that was used when the original website was first created was GraphDB, a graph-oriented database with not a lot of online information, besides Ontotext's documentation. This made debugging, especially when having Docker also involved, really difficult. The resources were scarce and every other error encountered had basically no answers to be found on the internet. After a lot of research, a Dockerhub image for the free version of GraphDB was found, which was created by the user Khaller, so that configuration was added to the docker-compose:

```
  graphdb:
    container_name: graphdb
    image: khaller/graphdb-free:9.10.0
    ports:
      - "9200:7200"
    build:
      context: ./graphdb
      dockerfile: ./Dockerfile
    restart: always
```

Dockerfile:

```
FROM khaller/graphdb-free:9.10.0

RUN mkdir -p /repository.init/Animals

RUN mkdir -p /temporary-folder

COPY config.ttl /repository.init/Animals/config.ttl
```

The documentation on the Dockerhub Image's page was quite helpful and allowed for proper initialization of a GraphDB repository using a config.ttl file:

```
#
# Configuration template for a GraphDB-SE repository
#
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix rep: <http://www.openrdf.org/config/repository#>.
@prefix sr: <http://www.openrdf.org/config/repository/sail#>.
@prefix sail: <http://www.openrdf.org/config/sail#>.
@prefix owlim: <http://www.ontotext.com/trree/owlim#>.

[] a rep:Repository ;
    rep:repositoryID "animals" ;
    rdfs:label "Animals repo" ;
    rep:repositoryImpl [
        rep:repositoryType "graphdb:FreeSailRepository" ;
        sr:sailImpl [
            sail:sailType "graphdb:FreeSail" ;

                        # ruleset to use
                        owlim:ruleset "empty" ;

                        # disable context index(because my data do not uses
    contexts)
                        owlim:enable-context-index "false" ;

                        # indexes to speed up the read queries
                        owlim:enablePredicateList "true" ;
                        owlim:enable-literal-index "true" ;
                        owlim:in-memory-literal-properties "true" ;
        ]
    ].
```

However, at this point another problem was encountered, **importing the existing data**. GraphDB has a built-in *Workbench Mode* that allows the user to set up repositories, import data, run SPARQL queries, etc, but when using Docker to set up GraphDB it seems these functions are not as convenient and easy to use. For example, when the container inevitably has to restart for any reason, the container should, ideally, automatically import all the data again, so going into Workbench Mode and importing the data manually just is not practical. Thankfully, this was solved by adding a line to the Dockerfile that copied the .ttl file with all of the data to a specific folder in the container. This because Khaller constructed the image in a way that data files inside the *toLoad* directory inside the corresponding repository folder could be preloaded when the container starts.

```
COPY animals-complete.ttl /repository.init/Animals/toLoad/animals-complete.ttl
```

Yet, this led to the biggest problem faced in this implementation. The container was **crashing** right in the middle of the preload process with seemingly almost no debugging information available. After a lot of trial and error, it was discovered that the file had so much information Docker was crashing due to not having enough Heap Memory available. To fix this another line was added to the Dockerfile:

```
ENV GDB_HEAP_SIZE=2G
```

And also, because the Docker instance being used in the development of this app was being run on Windows and Docker Desktop (Windows' way to run Docker) runs on WSL2, a line to the .wslconfig file had to be added to match the RAM that was allocated to Docker:

```
[wsl2]
memory=2GB
processors=2
```

With these fixes GraphDB was now working properly and the final Dockerfile configuration was as follows:

```
FROM khaller/graphdb-free:9.10.0

ENV GDB_HEAP_SIZE=2G

RUN mkdir -p /repository.init/Animals

RUN mkdir -p /temporary-folder
```

```
COPY animals-complete.ttl /repository.init/Animals/toLoad/animals-complete.ttl

COPY config.ttl /repository.init/Animals/config.ttl
```

At this point, it was time an authentication system to the website was implemented, so one could later utilize Kong to its full potential. To go about implementing this, like it was stated in this section on the previous app, an interface with **Vue.js** that introduced a simple login system was created.

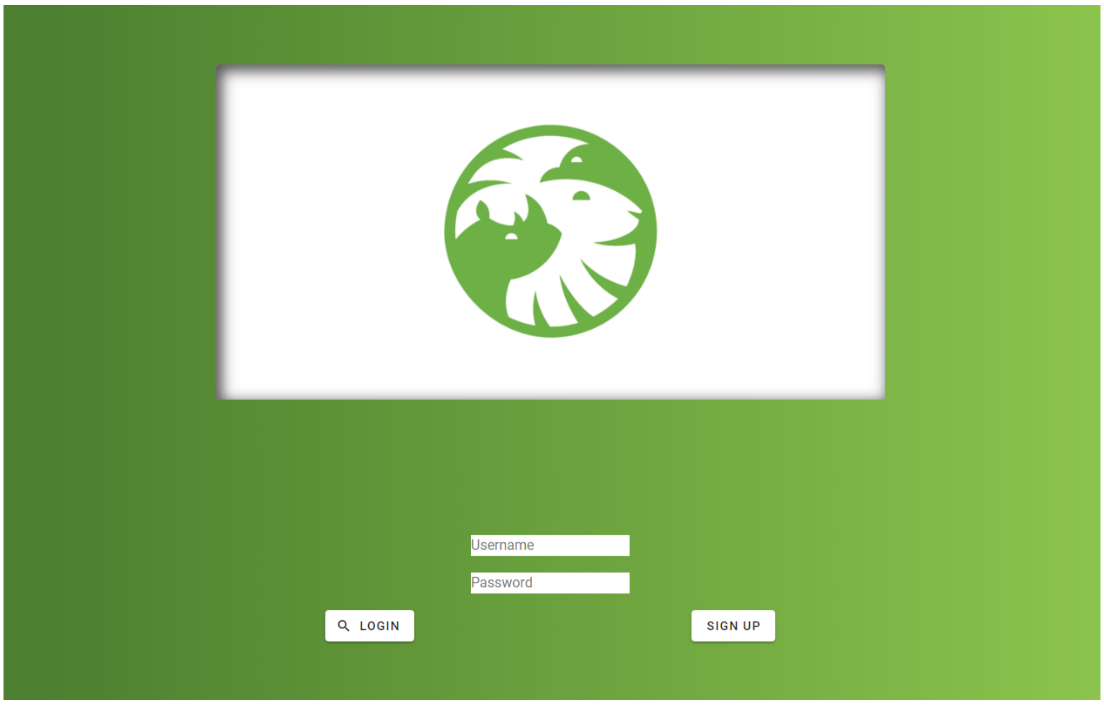


Figure 17: Front-end Animals-Wiki App (same as Equivalencias App)

As for the back-end of this authentication system, again, like the previous app, the **Express** framework was used to set up a server that could handle login requests, returning a success code if the username and password matched or a 403 prohibited if they did not. These users were of course stored in a MongoDB database quickly set up with the configuration from the previous app. Consequently, the configuration for the back-end and MongoDB was added to the docker-compose file.

```
  mongo-users:
    container_name: mongo-users
    restart: always
    environment:
      MONGO_INITDB_DATABASE: Animals-auth
    image: mongo
    volumes:
      - ./mongo-volume:/data/db
  auth:
    container_name: auth
    build:
      context: ./autenticacao
      dockerfile: ./Dockerfile
    restart: always
    links:
      - mongo-users
    networks:
      default:
        aliases:
          - auth
```

After establishing the front-end of the app, the respective back-ends for the core app and authentication system, and the respective databases in GraphDB and MongoDB, it was time to configure Kong. First Kong services were defined for said back-ends and credentials for *consumers* were established. Then, because a relatively simple but solid authentication plugin was preferred, the Key_Authentication plugin was used and the core app back-end service (which would be called animals-service) was protected with said plugin. This is the resulting kong.yml file:

```
_format_version: "2.1"

services:
  - name: animals-service
    url: http://api:7777
    routes:
      - name: animals-route
        paths:
          - /animals-api
  - name: auth_service
    url: http://auth:9000
    routes:
      - name: auth-route
        paths:
          - /auth

plugins:
  - name: key-auth
    service: animals-service
    config:
        key_names:
        - apikey
        key_in_body: true
        key_in_query: true
consumers:
  - username: user1
keyauth_credentials:
  - consumer: user1
```

As stated previously, in the plugin configuration the name *apikey* was chosen for the name of the **key** it must contain in a request to get a successful response and it was also decided that that key must be in either the body of the request or contained in the query. With this implemented, like before, the code of the front-end had to be changed to store the key in the user's cookies.

```
getkey: function () {
    axios({
        method: 'post',
        url: '/kong/auth/users/login',
        data: {
            username: this.username,
            password: this.password
        }
        })
        .then(data => {
            console.log(data.data)
            if (data.data) {
                axios
                    .get('/kong-auth/consumers/user1/key-auth')
                    .then(response => {
                ------> Vue.$cookies.set("key", response.data.data[0].key, "1d")
                        this.$router.push('/')
                        })
            }
        })
        .catch(err => console.log(err))
```

However, contrary to the previous app, when the user gets to the animal listing, i.e. the core app, instead of manually checking with Kong if the key is correct, the requests are simply made with the key in the query, and if the user does not possess the key or the key is incorrect the app does not return the data and the user can not view the animal listing.

```
handler(n) {
                      let v = '&'
                      if (!n) v = '?'
              -----> axios.get('/kong/animals-api/animals' + n + v + 'apikey=' +
    Vue.$cookies.get("key"))
                      .then(dados => {
                          ...
                      })
                      .catch(err => console.log(err))
```

With all of these moving parts now working together, this became the definite version of the Microservices Architecture of this web app:

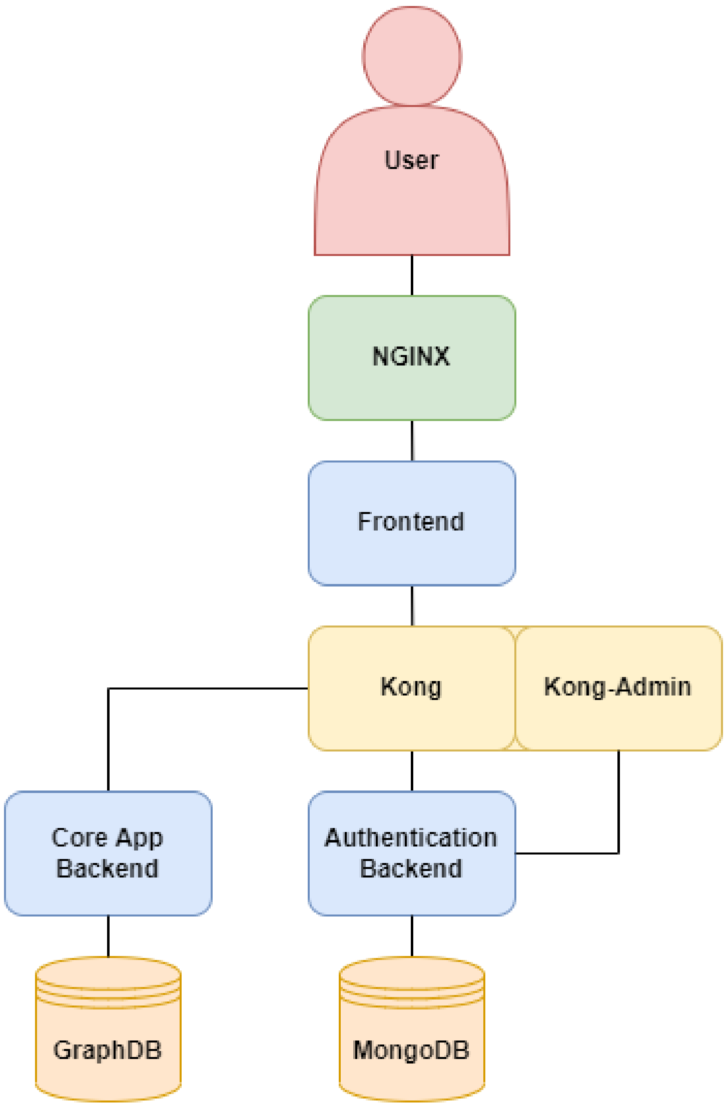


Figure 18: Animals-Wiki Microservices Architecture

And this the flowchart of the user experience:

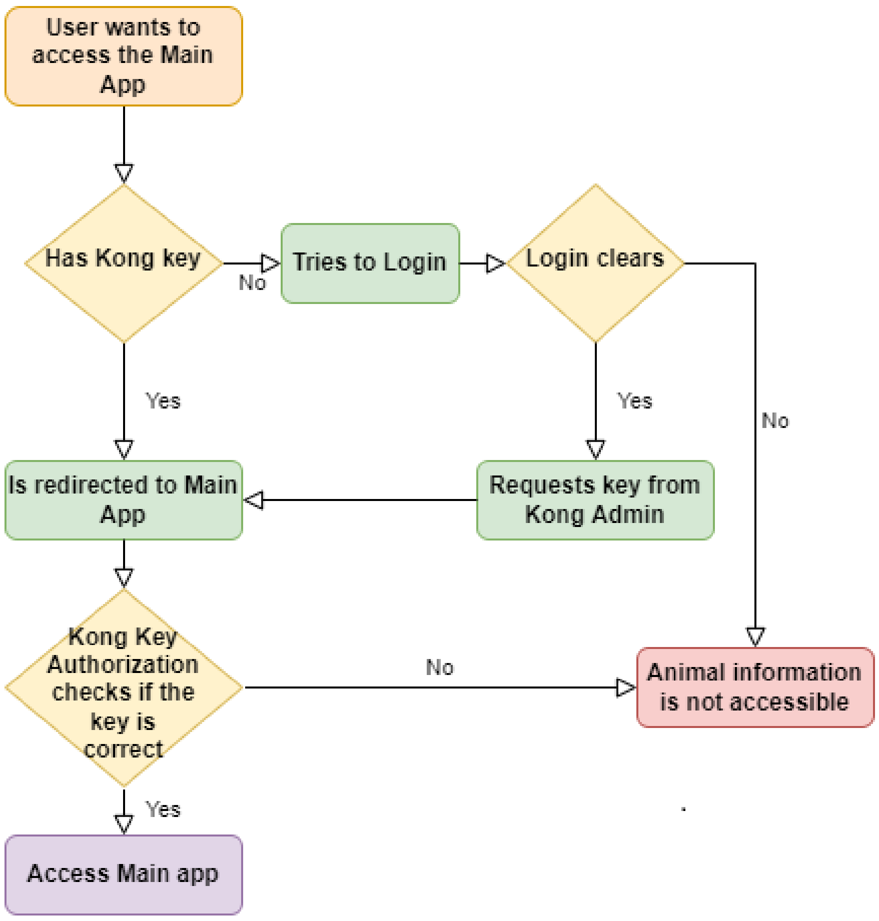

User wants to access the Main App
Has Kong key
No
Tries to Login
Login clears
Yes
Yes
No
Is redirected to Main App
Requests key from Kong Admin
Kong Key Authorization checks if the key is correct
No
Animal information is not accessible
Yes
Access Main app

Figure 19: Animals-Wiki Request Flowchart

4 HYPATIAMAT

Hypatiamat is an education website dedicated to help students of all ages with Mathematics. It contains exercises, mini-games, resources on the curriculum and so much more that could potentially help the student. It is also a platform where a professor is capable of managing their classes, give out homework to their students, etc, and the student can complete their homework or compete in "tournaments", motivating the student to study. It is truly an all encompassing app that helps students and professors alike in the subject of Mathematics. Additionally, because it is composed of several different services, all of them isolated from each other, it became the perfect test to the developed architecture philosophy.

4.1 INTRODUCTION TO THE APP

After experimenting with a few apps, doing a lot of research and recording the whole experience, it was finally time to tackle **Hypatiamat**.

However, the architecture on which the app is built could be, in some ways, considered outdated and in need of a re-work. Because the Hypatiamat "app" is actually several different apps that students would need to log-in to individually and were isolated from one another, this website would actually be, as said above, the perfect app to take everything that was learned up until this point and test if it would work on a large app acting on an even larger scale. Specifically, **two** of the many apps this whole website is built upon, ***Hypatiamat Backoffice*** and ***Hypatiamat TPC*** would be transformed into a Microservices Architecture while Kong would be used to set up a general authentication system, avoiding the individual login per app problem just discussed.

In this next section we will discuss: how the **architecture** was come up with, the process of **implementing** the idea for the app, the **problems** faced and the **solutions** that had to be come up with.

4.2 DEVELOPMENT OF THE WEB APPLICATION

As previously said, two services were tackled, *Hypatiamat Backoffice* and *Hypatiamat TPC* and the idea was to put them on a Microservices network and develop a separate, isolated and *global* Authentication service that allowed users to login to any app by only entering their credentials **once**. So, as the idea was to substitute the already established login systems the apps had, it was decided to first run these apps and see how that system worked, with the intention of making the fewest possible changes to these apps in this architecture transition.

Thus, the first step was to run ***Hypatiamat Backoffice***.

4.2.1 *Hypatiamat Backoffice*

First things first, a docker-compose file was created and added the already existing configuration from the Backoffice app to it:

```
version: '3.0'

services:
  api-dados:
    build:
      context: ./apiDados
    container_name: api-dados
    image: api:latest
    ports:
      - "3050:3050"
  interface-app:
    build:
      context: ./interface
    container_name: interface-app
    image: interface:latest
    depends_on:
      - "api-dados"
    ports:
      - "8080:8080"
```

This file was going to be where the configuration for ALL services was but at that moment it only had Backoffice's configuration because it was important to see how that specific app worked.

Here are the Dockerfiles for the back-end (api-dados) and the front-end (interface-app) respectively:

```
FROM node:alpine
```

```
WORKDIR /app

COPY package*.json ./

RUN npm install

COPY . .

EXPOSE 9000

CMD ["npm","start"]
```

```
FROM node:lts-alpine

WORKDIR /app

COPY package*.json ./

RUN npm install

COPY . .

EXPOSE 8080

CMD ["npm","run","serve"]
```

This is the usual set-up we've been accustomed to, with the only difference being in the front-end's Dockerfile as it does not use NGINX so those lines of code are not present.

Before moving on, it is worth talking about WampServer. WampServer is a Windows software that allows users to create web applications with Apache, PHP and most importantly, a **MySQL database**. This is important because all of the existing data for the Backoffice and TPC app is in SQL files so importing that into WampServer's MySQL database was a necessary step.

Coming back to the Hypatiamat Backoffice app, with this Docker set-up the app was successfully ran but upon experimenting with the login system it was discovered a **massive problem**. The authentication worked with JWT tokens, and these tokens were being stored in ***LocalStorage***. Now, when your app is running on a single domain this is no problem whatsoever, *however*, the authentication system works across **multiple** subdomains (or in this specific case across multiple ports as the whole app was run in localhost): one for the login, one for the Backoffice app, and one for the TPC app. The problem is LocalStorage uses a *same-origin policy* which sandboxes its data, meaning if, for example, our token is on sub1.website.com and we go to sub2.website.com we can not access our token across the subdomains. There is a natural fix for this problem which is storing your JWT tokens on browser **cookies**, which can be accessed across multiple subdomains. Unfortunately though, this meant a refactoring of both Hypatiamat Backoffice and Hypatiamat TPC was needed to store its tokens in cookies. Thankfully, a solution that would make a re-write of the entire code for these apps not necessary was come up with. Instead of **completely** erasing the LocalStorage usage and its entire logic, we would keep that logic as is and change only the part where the app gets the token. In other words, instead of getting the token from LocalStorage, the app would search for it in the cookies (because that is where it would be stored in the login) and only then store it in the LocalStorage from the cookies and let the app check if the token is valid or not. Here is a graph that might help visualize the process:

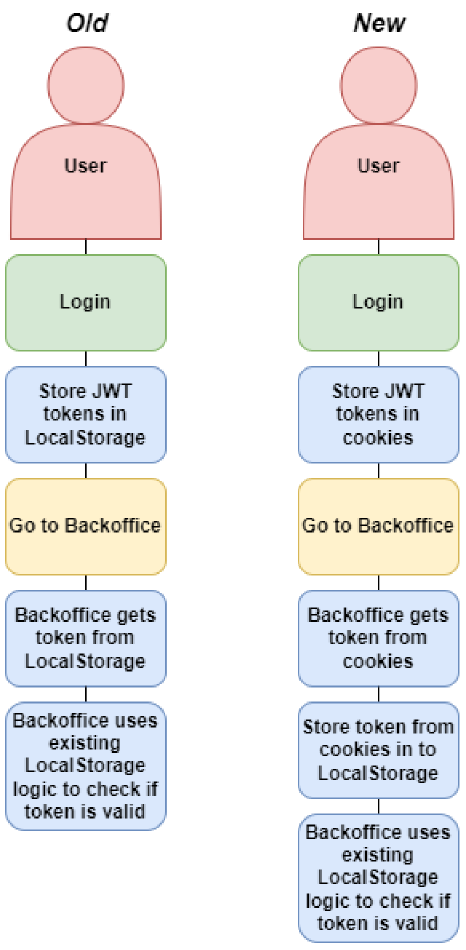

Old
New
User
User
Login
Login
Store JWT tokens in LocalStorage
Store JWT tokens in cookies
Go to Backoffice
Go to Backoffice
Backoffice gets token from LocalStorage
Backoffice gets token from cookies
Backoffice uses existing LocalStorage logic to check if token is valid
Store token from cookies in to LocalStorage
Backoffice uses existing LocalStorage logic to check if token is valid

Figure 20: Old vs New - token storage

After implementing this in the Backoffice app the same was done for the TPC app.

4.2.2 *Kong and JWT tokens*

When this point was reached, Kong was introduced into the mix and, because the app was using JWT tokens, the JWT plugin was opted for. This is the kong.yml file:

```
_format_version: "2.1"

services:
  - name: backoffice-api
    url: http://host.docker.internal:3050
    routes:
      - name: backoffice-api-route
        paths:
          - /backoffice-api
  - name: tpc-api
    url: http://host.docker.internal:3052
    routes:
      - name: tpc-api-route
        paths:
          - /tpc-api

plugins:
  - name: jwt
    service: backoffice-api
    enabled: true
    config:
      secret_is_base64: false
      run_on_preflight: true
      key_claim_name: kid
      claims_to_verify:
        - exp
  - name: jwt
    service: tpc-api
    enabled: true
    config:
      secret_is_base64: false
      run_on_preflight: true
      key_claim_name: kid
      claims_to_verify:
        - exp
  - name: cors
    config:
      origins:
        - "*"
      methods:
        - GET
        - POST
```

```
          - PUT
          - DELETE
        headers:
          - Accept
          - Accept-Version
          - Content-Length
          - Content-MD5
          - Content-Type
          - Date
          - X-Auth-Token
          - Authorization
        exposed_headers:
          - X-Auth-Token
        credentials: true
        max_age: 3600
        preflight_continue: false

consumers:
  - username: user123

jwt_secrets:
  - consumer: user123
    secret: YOUR_SECRET
```

Unpacking this config file one block at a time, first we have the *services*: here is where the Kong services were defined for the back-ends of the Backoffice app and the TPC app. Next, the *plugins*: here we have two instances of the JWT plugin, one for each back-end service, and the CORS plugin. In the JWT plugin block there are a few things happening: obviously the *name* is the name of the plugin and *service* is the targeted service. *enabled* just means the plugin is active and *config* contains a few options you might decide to enable depending on your application. In this case, namely *key_claim_name* and *claims_to_verify* are important: the second one defines what fields the plugin will check that are valid, in this case *exp* or, in other words, the expiration date, but the ***first*** one, *key_claim_name*, will define the name of the unique key that **every** token must be signed with to be valid. It works very similarly to the key in *Key Authorization* from the previous app, in the way that it tracks which consumer, or user, is associated with the token, so it can be verified. Another thing that the token must be signed with is the **secret**. This is obviously up to the developer and should not be shared. It can be defined in the last block of the kong.yml config file, jwt_secrets.

Here is the function that signs the tokens (this function would only be created later but it shows how the signing of the token works):

```
const secret = YOUR_SECRET

generateToken = async function(user, time){
    try {
        ...
        const kid = await axios.get('http://host.docker.internal:8001/consumers/
    user123/jwt');
        const token = jwt.sign({ user, kid: kid.data.data[0].key}, secret, {
            algorithm: "HS256",
            expiresIn: time,
        })
        return token
    } catch (err) {
        // Handle Error Here
        console.error(err);
    }
}
```

Circling back to the plugins, the **CORS** plugin is only present because the kong requests were being made across sub-domains, so CORS policy errors were obviously encountered. Thankfully, Kong developed a plugin to help with that.

When testing sending requests to the Backoffice app through Postman, an API platform capable of creating and sending complex API requests to any url as well as get a detailed response, the request call succeeded if a signed token was sent through the **Bearer Authentication** header. However, both the Backoffice app and the TPC app already had their API calls defined, so, to not refactor **every single request** the front-end makes to the back-end of these applications, the code was changed so when the app first obtains the token it sets it as the Bearer Authentication token **globally**:

```
if (Vue.$cookies.isKey("token")) {
    let tokenTemp = Vue.$cookies.get("token")
    localStorage.setItem("token", tokenTemp)
    axios.defaults.headers.common['Authorization'] = 'Bearer '+ localStorage.
    getItem('token');
}
```

This seemed to work, although at this point another problem appeared. When storing the token from the cookies into LocalStorage the app thought the token was incorrect, deleted it and kicked the user from the app. This was because the order of instructions for the transition from cookies to LocalStorage was wrong and not very robust, so then it was changed so first the app would check if the token cookie *exists* and if it did store it in LocalStorage and then set the global header. Also, when logging out of the app, it would delete the token cookie and empty the LocalStorage. This because when logging back into the app or even going from Backoffice to TPC or vice-versa the app would throw an error and kick out the user. After re-working and cleaning up that logic the app worked perfectly.

4.2.3 *Login interface*

At this point, both apps were working, one could traverse them and go from one to the other without problems, sending every request call through Kong with the token in the Bearer Authorization header. **However**, there were no means of logging in and signing a token easily, they were being signed manually. Therefore, it was time for the final piece of this Microservices Architecture "puzzle": The authentication interface.

Following the pattern of previous apps, the front-end was designed in **Vue.js** and the Backoffice's back-end, made with **Express**, was used as a mold for the Login system's back-end, as it already defined the routes and configuration for its original login system. To accomplish this, the configuration for these two services was added to the docker-compose file, making this the final version of that file:

```
version: '3.0'

services:
  #KONG
  kong:
    container_name: kong
    restart: always
    image: kong:latest
    volumes:
      - ./kong.yml:/usr/local/kong/declarative/kong.yml
    environment:
      - KONG_DATABASE=off
      - KONG_DECLARATIVE_CONFIG=/usr/local/kong/declarative/kong.yml
      - KONG_PROXY_ACCESS_LOG=/dev/stdout
      - KONG_ADMIN_ACCESS_LOG=/dev/stdout
      - KONG_PROXY_ERROR_LOG=/dev/stderr
      - KONG_ADMIN_ERROR_LOG=/dev/stderr
      - KONG_ADMIN_LISTEN=0.0.0.0:8001, 0.0.0.0:8444 ssl
    ports:
      - "8001:8001"
      - "8000:8000"

  #BACKOFFICE BACK-END
  api-dados:
    build:
      context: ./Hypatiamat-BackOffice/apiDados
    container_name: api-dados
    restart: always
    image: api:latest
    ports:
      - "3050:3050"

  #BACKOFFICE FRONT-END
  interface-app:
    build:
      context: ./Hypatiamat-BackOffice/interface
    container_name: interface-app
    restart: always
    image: interface:latest
    depends_on:
      - "api-dados"
    ports:
      - "8080:8080"

  #AUTH BACK-END
  api-auth:
    build:
      context: ./Auth/backend
    container_name: api-auth
    restart: always
    ports:
      - "9000:9000"

  #AUTH FRONT-END
  auth-app:
    build:
      context: ./Auth/frontend
    container_name: auth-app
    restart: always
    depends_on:
      - "api-auth"
    ports:
      - "12090:12090"

  #TPC FRONT-END
  tpc-app:
    build:
      context: ./Hypatiamat-TPC/frontend
    container_name: tpc-app
    restart: always
    depends_on:
      - "strapi"
    ports:
      - "8081:8081"

  #TPC BACK-END
  strapi:
    container_name: strapi
    restart: always
    build:
      context: ./Hypatiamat-TPC/backend
    ports:
      - "3052:3052"
```

For this system's back-end, as was just stated, the back-end of the Backoffice app was copied over, and every part of the API that did not correspond to the login and register features was deleted. This was so the original login system's logic could be kept as much as possible, so a transition from the original system to the new system would be as seamless as possible.

As for the front-end, when designing the login interface it was known there were two services one could go to, so this main menu was devised, where one could **register**, or choose the Backoffice or TPC app to login to.:

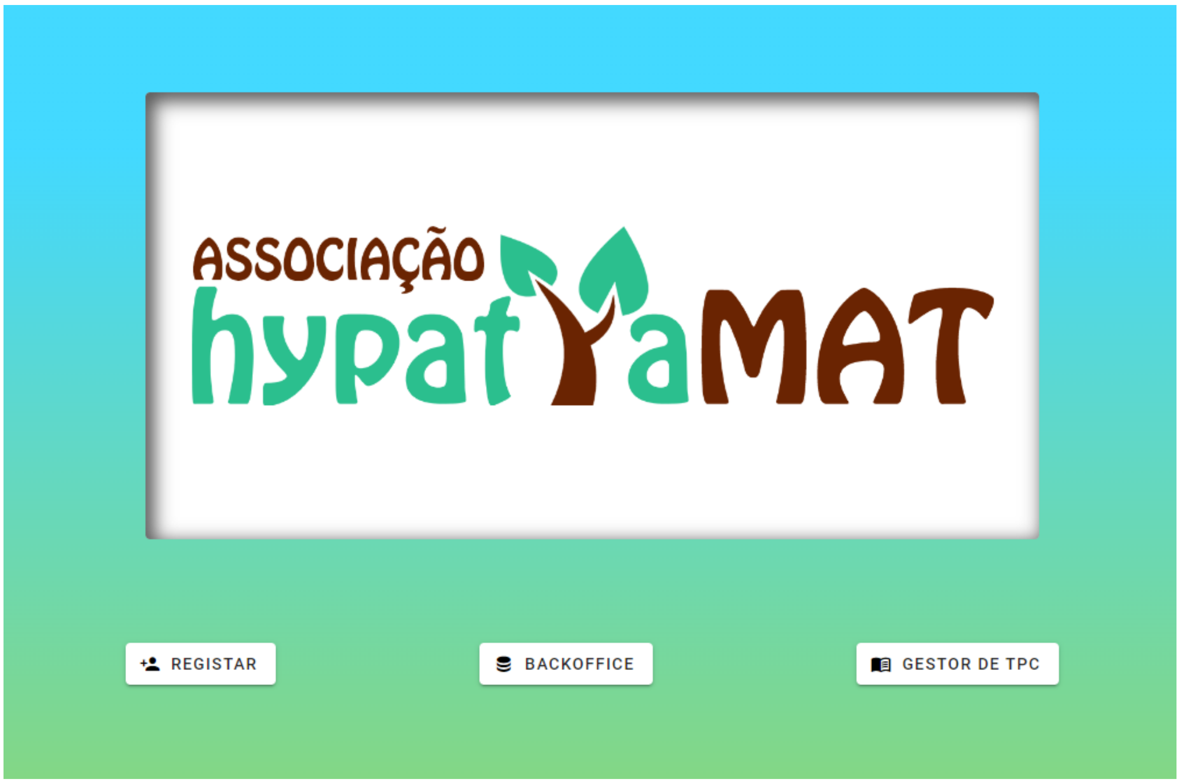


Figure 21: Main Menu

This is the register menu:

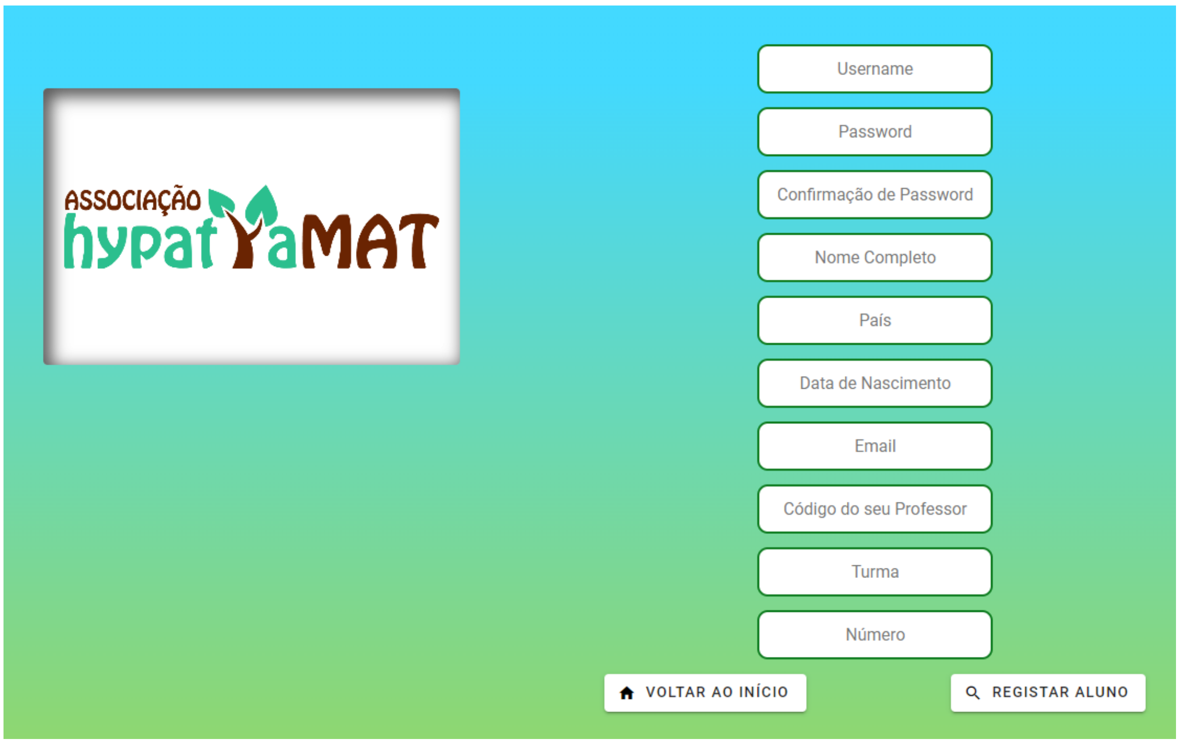


Figure 22: Register Menu

Originally there was a login menu and only then the user would choose if they wanted to go the Backoffice app or TPC app, which would make sense as the login is supposed to be global, i.e. independent from which app they want to access, however, the Backoffice has two extra login options: logging in as a **temporary student** or as a **temporary professor**. This is so a user can explore the app without having an account. But these options do not exist in the TPC app so it was decided that the design of the main menu would be as it is now. This is not to say the login is not global, it is and a user can freely and effortlessly go from Backoffice to TPC or vice-versa by logging in only once, but it was felt that this distinction in the menus had to be made for a more intuitive experience.

Here is the Backoffice login menu:

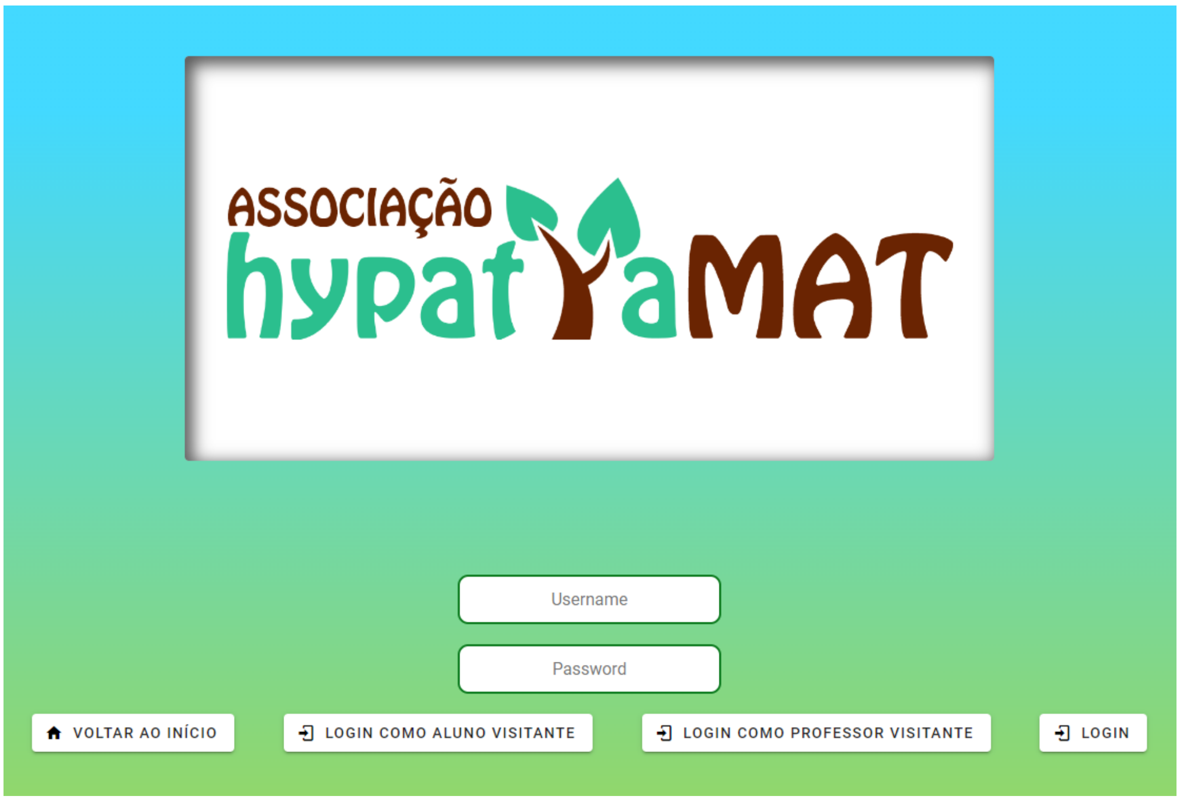


Figure 23: Backoffice login Menu

And here is the TPC login menu:

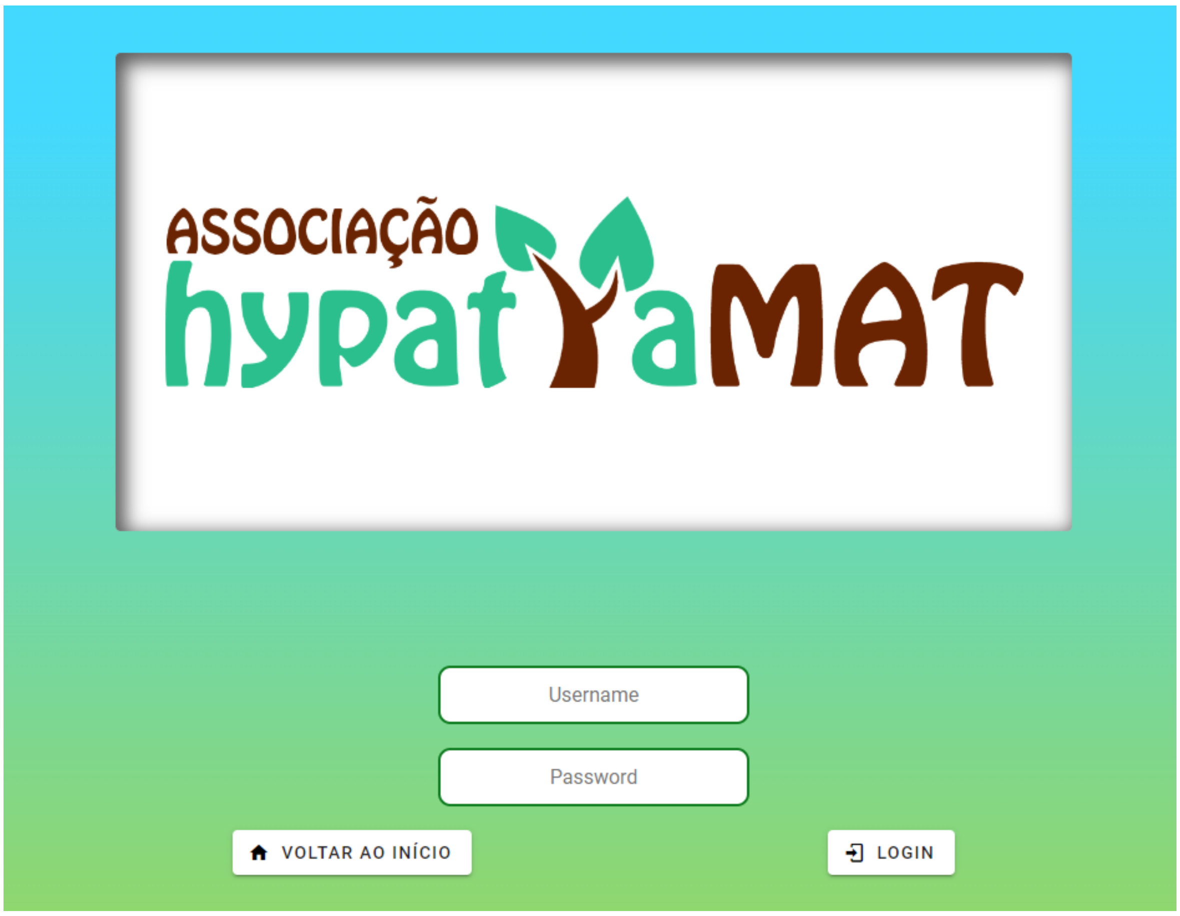
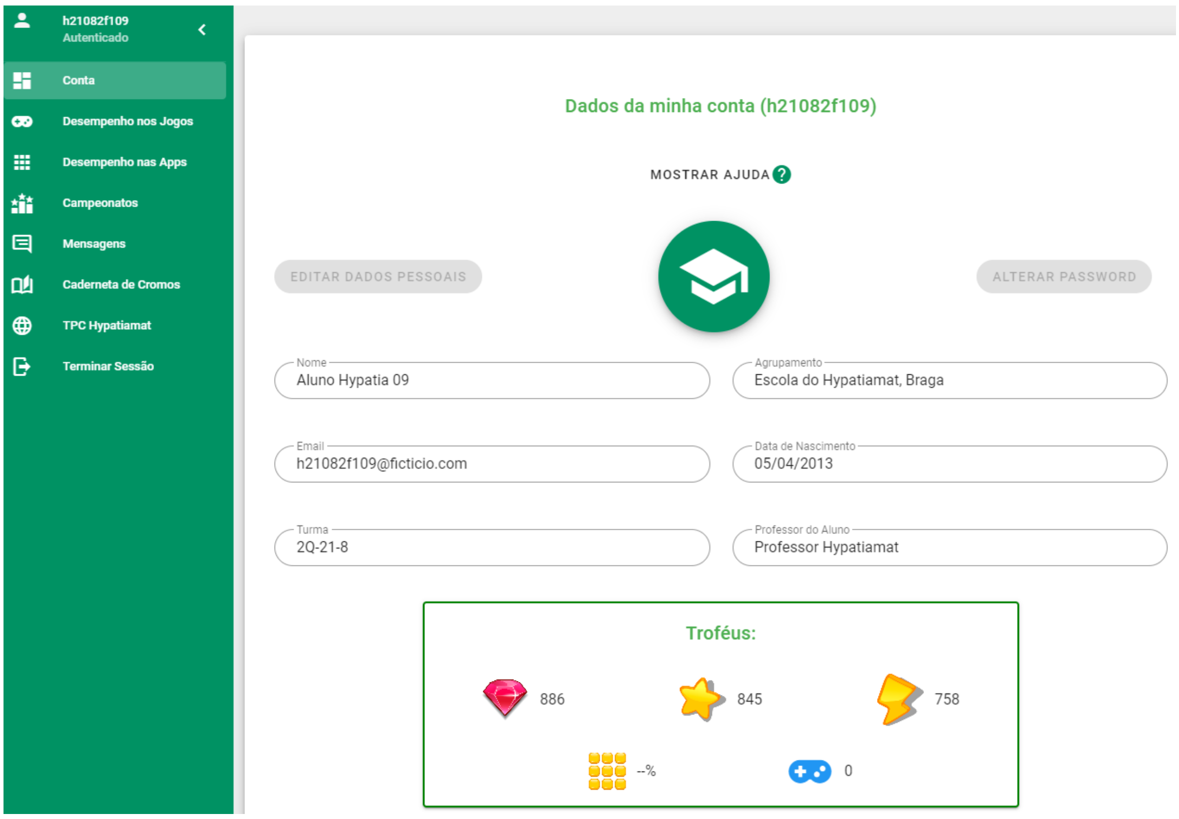


Figure 24: TPC login Menu

Upon entering the Backoffice app this is what a user would see:



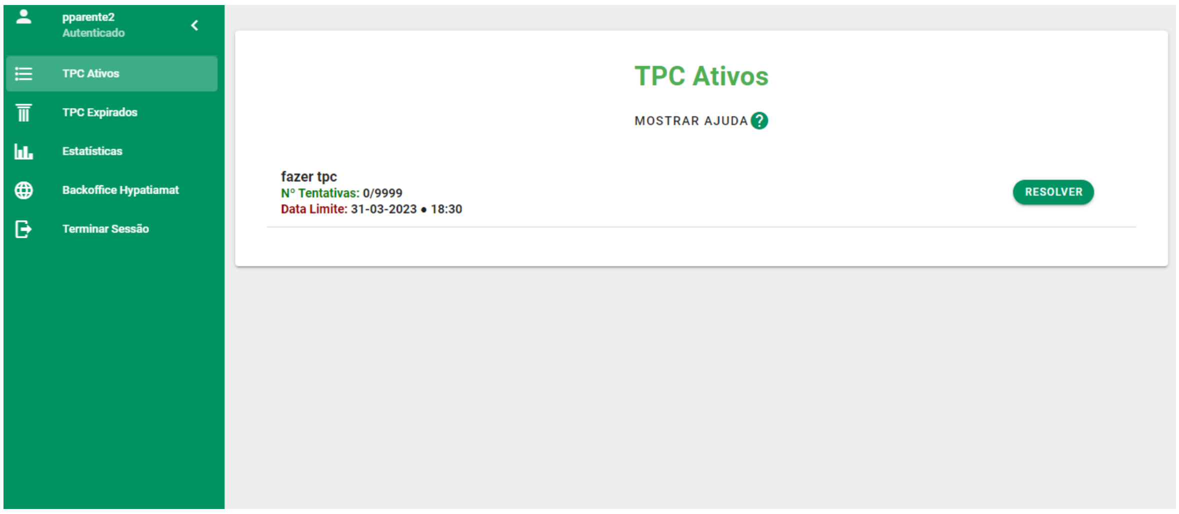
Dados da minha conta (h21082f109)

Nome: Aluno Hypatia 09
Agrupamento: Escola do Hypatiamat, Braga
Email: h21082f109@ficticio.com
Data de Nascimento: 05/04/2013
Turma: 2Q-21-8
Professor do Aluno: Professor Hypatiamat

Troféus:
886 845 758
--% 0

Figure 25: Backoffice App

Entering the TPC app this is what a user would see:



TPC Ativos

fazer tpc
Nº Tentativas: 0/9999
Data Limite: 31-03-2023 • 18:30

Figure 26: TPC App

With the login interface done the app was now completely functional. This is the final version of the architecture:

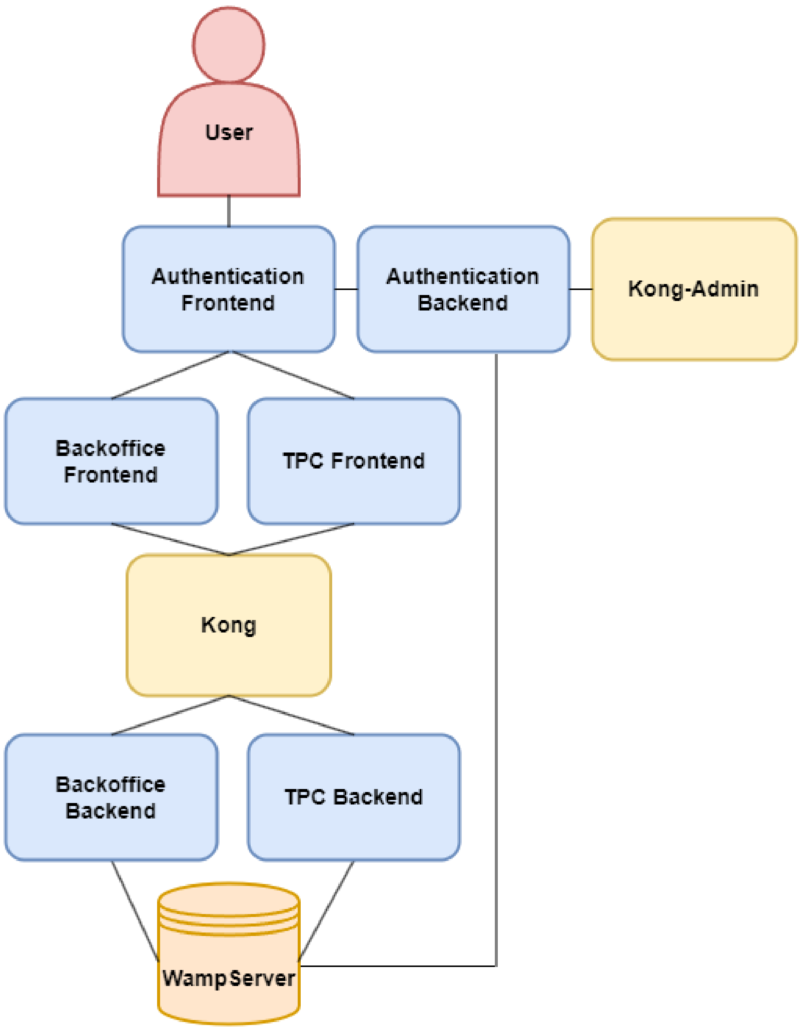


Figure 27: Hypatiamat Architecture

And this is a flowchart of the user experience:

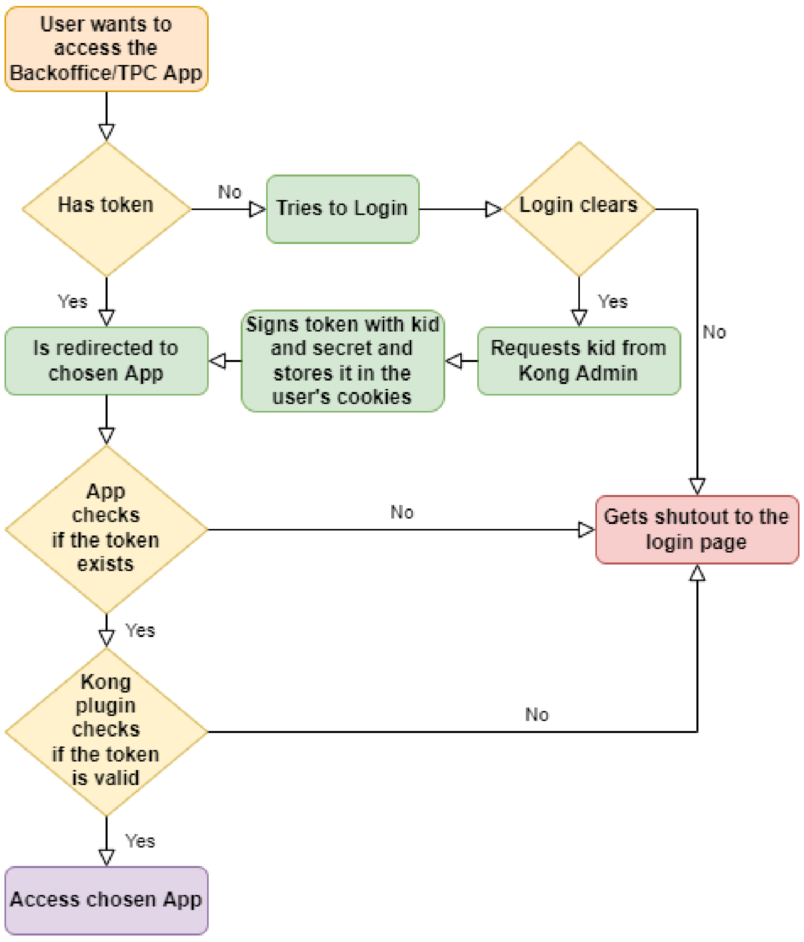

User wants to access the Backoffice/TPC App
Has token
No
Tries to Login
Login clears
Yes
Is redirected to chosen App
Signs token with kid and secret and stores it in the user's cookies
Yes
Requests kid from Kong Admin
No
App checks if the token exists
No
Gets shutout to the login page
Yes
Kong plugin checks if the token is valid
No
Yes
Access chosen App

Figure 28: User experience flowchart

It is worth noting that the new authentication system is very scalable and if one wishes to add more apps to the system they would only need to add a button and a login form and the login should work right away, assuming the new app stores its tokens in the cookies. If not, a similar refactoring done to Backoffice and TPC would need to be done to that app as well.

4.3 OUTCOME

After undertaking these case studies it is possible to say that the changes made to the original apps definitely are beneficial the functionality of the app as a whole and the knowledge acquired in the process of implementing these changes most certainly helped in the understanding of Kong, Microservices Architectures and the logic of login systems, as well as several other technologies and paradigms addressed in the making of this thesis.

5 CONCLUSION

With the completion of this thesis the main goal was achieved: constructing a guide that could help the reader applying some or all of the technologies talked about in their own work. A few other goals were also achieved, like creating a general login system for every and any frontend service in the application, as well as attaining a deep understanding of API Gateways, Microservices, HTTPS passport configuration and authentication token handling.

To accomplish this, we experimented on several case studies, first two smaller apps to create familiarity with the technologies and later a real life case with a considerably bigger and more complex application called "Hypatiamat".

Reflecting on the progress made, it is possible to assert that Kong as an API Gateway is definitely one of the better options, as it is free and has many features, such as a huge library of plugins that can be easily and seamlessly implemented into any project, and it is also possible to assert that in huge applications with many services Kong can certainly be a strong asset because it allows for a better management of the service-to-service communication. **However**, it is worth noting that in small applications it might not be worth it to have this extra step in communication because managing that communication tends do be quite simple, or even big applications it might not be worth it if the plugin functionality is not being used, as something like NGINX is also capable of managing the communication network as Kong does.

In conclusion, Kong can most certainly be a huge help in many applications, but, as most things, it really depends on the app and its function in that app.

In future iterations of the work done in the Hypatiamat application we could add more applications to the login system as it is done in a scalable way that could easily introduce similar apps to its communication network. We could also take more advantage of the plugin system because, as of the writing of this document, only the JWT plugin is being used. We could take advantage of a rate-limiting plugin or an analytics plugin to gather data on the website's users.

BIBLIOGRAPHY

What does an api gateway do? URL https://www.redhat.com/en/topics/api/what-does-an-api-gateway-do.

What is amazon api gateway? URL https://docs.aws.amazon.com/apigateway/latest/developerguide/welcome.html.

The api gateway pattern versus the direct client-to-microservice communication. URL https://docs.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/direct-client-to-microservice-communication-versus-the-api-gateway-pattern.

Animals-Wiki. URL https://gate.epl.di.uminho.pt/.

Apigee. Apigee reviews, rating and features 2022. URL https://www.peerspot.com/products/apigee-reviews.

AZ_Animals. URL https://a-z-animals.com/.

ACME Certbot. URL https://letsencrypt.org/docs/client-options/.

Kong's CORS. URL https://docs.konghq.com/hub/kong-inc/cors/.

Docker. URL https://www.docker.com/.

Kong Documentation. URL https://docs.konghq.com/gateway/latest/.

Express. URL https://expressjs.com/.

Faren. Kong — the microservice api gateway. URL https://medium.com/@far3ns/kong-the-microservice-api-gateway-526c4ca0cfa6.

André Faria. Melhores livros sobre microservices (microserviços). URL https://blog.andrefaria.com/melhores-livros-sobre-microservices-microservicos.

Keith D. Foote. A brief history of microservices - dataversity. URL https://www.dataversity.net/a-brief-history-of-microservices/#.

GraphDB Dockerfile Github. URL https://github.com/Ontotext-AD/graphdb-docker.

GraphDB. URL https://graphdb.ontotext.com/.

Imesh Gunaratne. Wso2 api manager in a nutshell. URL https://medium.com/scalable/wso2-api-manager-in-a-nutshell-eaac20812f0c.

Hypatiamat. Hypatiamat website. URL https://www.hypatiamat.com/.

Official Docker image for GraphDB. URL https://hub.docker.com/r/ontotext/graphdb/.

Kong's JWT. URL https://docs.konghq.com/hub/kong-inc/jwt/.

Kong's Key_Authentication. URL https://docs.konghq.com/hub/kong-inc/key-auth/.

GraphDB Khaller. URL https://hub.docker.com/r/khaller/graphdb-free.

Lets-Encrypt. URL https://letsencrypt.org/.

Lua. URL https://www.lua.org/.

MongoDB. URL https://www.mongodb.com/.

Joe Nemer. Advantages and disadvantages of microservices architecture. URL https://cloudacademy.com/blog/microservices-architecture-challenge-advantage-drawback/.

NGINX. URL https://www.nginx.com/.

NGINX-Website. URL https://www.nginx.com/resources/glossary/nginx/.

NGINX_HTTPS. URL http://nginx.org/en/docs/http/configuring_https_servers.html.

Kong Plugins. URL https://docs.konghq.com/hub/.

Postman. URL https://www.postman.com/.

Syntax Turtle. URL https://en.wikipedia.org/wiki/Turtle_(syntax).

WampServer. URL https://www.wampserver.com/en/.

Docker WSL2. URL https://docs.docker.com/desktop/windows/wsl/.