Laboratório de Engenharia Informática
UMbook

Luís Filipe[1], Luís Martins[1], and Luís Braga[1]

Universidade do Minho, Departamento de Informática, 4710-057 Braga, Portugal
{a83099,a82298,a82088}@alunos.uminho.pt

Abstract. O presente relatório foi elaborado no âmbito da Unidade Curricular de Laboratório de Engenharia Informática do 4º ano do Mestrado Integrado em Engenharia Informática. Este apresenta a contextualização, planeamento e implementação do UMbook, uma rede social destinada à partilha de dados na comunidade estudantil.

Keywords: Rede social · Vue · Mongo · Docker · GraphDB.

1 Introdução

No âmbito da unidade curricular de Laboratório de Engenharia Informática foi proposto que se realizasse um projecto maior em dimensão e em trabalho, elevando a fasquia em relação a todos trabalhos pretéritos. Para o efeito, o grupo propôs que se desenvolvesse, com algumas diferenças e em maior profundidade, o trabalho prático de uma outra unidade curricular do primeiro semestre do presente ano denominada Desenvolvimento de Aplicações WEB.

O trabalho prático consiste na construção de uma rede social, cujo objectivo é promover não só uma maior acessibilidade e partilha de dados tal como a coesão entre alunos e a aprendizagem no ambiente estudantil. Contudo, o foco da plataforma é a partilha de dados. Deste prisma, foram criadas funcionalidades como a criação de pastas, directamente relacionadas com unidades curriculares, onde se poderão depositar ficheiros relevantes ao estudo e que contribuam para o melhor desempenho dos alunos que utilizem esta rede social. Tal como foi explicitado anteriormente, o projecto consiste numa rede social, o que por si implica que sejam disponibilizadas formas de comunicar e participar numa comunidade. Para tal, são cedidos serviços como o *chat*, possibilidade de publicar em grupos e criar eventos.

2 Arquitectura

Neste capítulo apresentar-se-á a arquitectura planeada para resolver o problema em mão. Considerando-se um dos passos mais importantes, se não o mais importante, de todo o processo que foi a realização deste projecto.

A seguinte figura representa sucintamente a arquitectura do projecto.

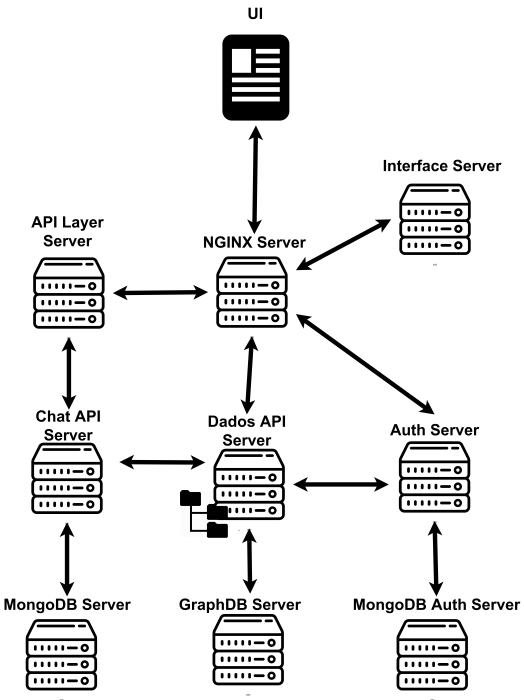

Fig. 1. Arquitectura do sistema.

Explicar-se-á de seguida, de cima para baixo, da esquerda para a direita e de uma maneira geral a função dos componentes representados na figura 1.

As funcionalidades do *Interface Server* são, como o nome indica servir a *interface* necessária ao utilizador para este interagir com o sistema desenvolvido e usufruir das suas funcionalidades.

A *API Layer* serve como camada de segurança e como *façade* de todos os servidores que se encontram ao serviço arquitecturalmente abaixo.

O componente UI (*user interface*) é o conteúdo servido pelo *Interface Server* na forma de uma página web, com a qual o utilizador poderá interagir e realizar ações.

O *NGINX Server* é usado com a finalidade de servir de *proxy*, sendo esta a sua utilização na arquitectura.

Como seria expectável que a funcionalidade de *chat* vá gerar grande volume de dados e de tráfego, criou-se o *Chat API Server*. Que está incumbido de servir a funcionalidade de comunicação entre utilizadores.

O *Dados API Server* é um serviço fulcral neste sistema, uma vez que por ele passa e são tratados grande parte dos dados da aplicação, fazendo ponte entre o servidor *GraphDB* e o detendo em si um sistema de ficheiros que armazena os documentos da aplicação.

O *Auth Server* ou, na forma não abreviada, *authentication server* tem um papel central na autenticação de utilizadores.

O *MongoDB Server* que foi criado com o intuito de manter não só a persistência de dados altamente escaláveis, como os dados relativos a conversas entre utilizadores, como também dados relativos a autenticação.

O servidor de base de dados primordial e principal será o *GraphDB Server*, que trata de informações centrais à aplicação como dados sobre utilizadores.

Por último, o *MongoDB Auth Server* ou, sem abreviaturas, *MongoDB Authentication Server* guarda dados relativos à autenticação dos clientes da aplicação.

3 Backend

No *backend* encontra-se a camada de acesso aos dados e a parte da lógica da aplicação que não é possível colocar do lado do cliente, principalmente devido a questões de segurança. Neste capítulo apresentar-se-ão mais detalhadamente os serviços e servidores elaborados neste projecto.

3.1 Base de Dados GraphDB

De forma a manter a persistência dos dados ao longo do tempo elaboraram-se bases de dados, sendo que a primeira a ser elaborada foi a base de dados em *GraphDB*.

O *GraphDB* é uma base de dados de grafos que utiliza estruturas de grafos e propriedades para representar e armazenar informação. A representação em grafo faz com que informação seja guardada em nodos e que as arestas se manifestem como relações entre os nodos. Esta estrutura, aparentemente simples, traz consigo múltiplas propriedades e benefícios para quem as utilizar, tais como:

- As relações entre dados permitem que os mesmos estejam relacionados directamente, o que permite que simples *queries* retornem grandes quantidades de informação.
- Realizar *queries* sobre relações é rápido, pois as relações também estão guardadas na base de dados.
- O *GraphDB*, particularmente, permite que as perguntas sejam feitas em *SPARQL*, que é uma linguagem relativamente descomplicada.
- As relações entre dados mostram explicitamente as dependências entre os dados, enquanto que bases de dados relacionais ligam os dados de uma forma implícita.

Ontologia Nas ciências de computação uma ontologia é um modelo de dados que representa um conjunto de conceitos dentro de um domínio e os relacionamentos entre estes. Esta estrutura de dados também permite que se realizem inferências sobre os objectos do domínio. Perante esta definição, e devido às suas semelhanças com um grafo, é possível armazenar estruturas ontológicas

num sistema de base de dados *GraphDB*. O conceito de ontologia também oferece outro grau técnico e de coerência a nível dos dados ao projecto, o que levou ao seu uso pelo grupo de trabalho.

A ontologia foi elaborada com recurso ao *software Protégé* [1] e foram elaboradas as seguintes classes. De forma a elaborar a ontologia no *Protégé* foi necessário definir as classes, as *object properties* (relações) e as *data properties* (propriedades das classes).

Table 1. Classes definidas na ontologia.

Classe	Descrição
Admin	Tipo de utilizador com nível de acesso máximo
Aluno	Tipo de utilizador com acesso normal
Ano	São frequentados por alunos e fazem parte dos cursos
Cadeira	Pertencem a um determinado ano
Comentario	Comentado por alguém à cerca de uma publicação
Curso	Curso que possuí alunos (um grupo ao qual o aluno tem acesso)
Evento	Possui um intervalo de tempo para ocorrer (duas datas) e está relacionado com um parceiro
Ficheiro	Ficheiro que foi publicado numa publicação ou relacionado com uma pasta
Pasta	Faz parte de uma cadeira e possui vários ficheiros
Publicacao	Pode conter ficheiros e os utilizadores podem visualizar/gostar/comentar uma publicação
Responsavel	Tipo de utilizador relacionado com o Ano pelo qual é responsável, com nível de acesso suficiente para editar esse Ano

A fim de tirar proveito máximo da capacidade do *GraphDB* e dos grafos que são possíveis construir ainda se idealizaram as relações entre estas classes, enriquecendo esta rede de conhecimento e as suas propriedades. Abaixo se explicitam as *object properties* (relações) e os seus domínios e *ranges*, bem como as propriedades inversas.

Table 2: *Object Properties* definidas na ontologia.

Object Property	Domínio	Ranges	Obj. Prop. inversa
adoradaPor	Publicacao; Comentario	Aluno; Admin; Responsavel	gostaDe
armazenadoEm	Ficheiro	Cadeira	
comenta	Aluno; Admin; Responsavel	Comentario	éComentadoPor
comentadoEm	Comentario	Publicacao	possuiComentario

fazParteCurso	Ano	Curso	possuiAno
fazPedidoAmizade	Admin; Aluno; Responsavel	Admin; Aluno; Responsavel	pedidoAmizadeFeitoPor
frequenta	Admin; Aluno; Responsavel	Curso; Ano; Cadeira	éFrequentadoPor
gere	Responsavel	Ano	éGeridoPor
gostaDe	Admin; Aluno; Responsavel	Comentario; Publicacao	adoradaPor
guardadoEm	Ficheiro	Publicacao	PossuiFicheiro
leciona	Ano	Cadeira	éLeccionadaEm
organiza	Curso	Evento	parceriaCom
parceriaCom	Evento	Curso	organiza
pedidoAmizadeFeitoPor	Admin; Aluno; Responsavel	Admin; Aluno; Responsavel	fazPedidoAmizade
possuiComentario	Publicacao	Comentario	comentadoEm
possuiPasta	Cadeira; Pasta	Pasta	éPossuido
publica	Admin; Aluno; Responsavel	Publicacao	éPublicadoPor
possuiAno	Curso	Ano	fazParteCurso
PossuiFicheiro	Pasta; Cadeira; Publicacao	Ficheiro	guardadoEm
possuiPublicacao	Ano; Curso; Cadeira	Publicacao	éPublicadaEm
temFoto	Admin; Aluno; Responsavel	Ficheiro	éFotode
temPresenca	Evento	Admin; Aluno; Responsavel	vai
vai	Admin; Aluno; Responsavel	Evento	temPresenca
éAmigoDe	Admin; Aluno; Responsavel	Admin; Aluno; Responsavel	éAmigoDe

éComentadoPor	Comentario	Admin; Aluno; Responsavel	comenta
éFotode	Ficheiro	Admin; Aluno; Responsavel	temFoto
éFrequentadoPor	Curso; Ano; Cadeira	Admin; Aluno; Responsavel	frequenta
éGeridoPor	Ano	Responsavel	gere
éLecionadaEm	Cadeira	Ano	leciona
éPossuidoPor	Pasta	Cadeira	possuiPasta
éPublicadaEm	Publicacao	Curso; Ano; Cadeira	possuiPublicacao
éPublicadoPor	Publicacao	Admin; Aluno; Responsavel	publica

Para simplicar a compreensão destas relações apresenta-se a seguir a tabela com a descrição das mesmas.

Table 3: Descrições das *Object Properties* definidas na ontologia.

Object Property	Descrição
adoradaPor	Uma publicação possuí gostos de utilizadores
armazenadoEm	Um ficheiro está armazenado numa determinada Cadeira
comenta	Um utilizador comenta uma determinada publicação
comentadoEm	Um comentário faz parte de uma determinada publicação
fazParteCurso	Um ano faz parte de um determinado curso
fazPedidoAmizade	Um utilizador faz um pedido de amizade a outro
frequenta	Um curso, ano ou cadeira é frequentado por utilizadores
gere	Um responsável tem a responsabilidade de gerir um determinado ano
gostaDe	Um utilizador pode gostar de uma publicação ou de um comentário
guardadoEm	Um ficheiro guardado e associado a uma determinada publicação
leciona	Um ano tem no seu plano de estudos determinada cadeira

organiza	Um curso organiza ou faz parte dos organizadores de um evento
parceriaCom	Um evento possuí como organizador um ou mais cursos
pedidoAmizadeFeitoPor	Um utilizador recebeu um pedido de amizade feito por outro
possuiComentario	Uma publicação possuí um ou mais comentários
possuiPasta	Uma cadeira ou pasta possuem pastas
publica	Um utilizador fez uma publicação
possuiAno	Um curso possuí determinado ano
PossuiFicheiro	Uma pasta, cadeira ou publicação contem um ficheiro
possuiPublicacao	Um curso, ano ou cadeira contem publicações
temFoto	Um utilizador possuí uma foto de perifl
temPresenca	Um evento possuí a presença de utilizadores
vai	Um utilizador vai marcar presença num evento
éAmigoDe	Um utilizador é amigo de outro utilizador
éComentadoPor	Um comentário foi publicado por um utilizador
éFotode	Um ficheiro é a foto de perfil de um utilizador
éFrequentadoPor	Um curso, ano ou cadeira é frequentada por utilizadores
éGeridoPor	Um ano é gerido/administrado por um utilizador do tipo Responsavel
éLecionadaEm	Uma cadeira é lecionada/possuída por um ano
éPossuidoPor	Uma pasta faz parte de uma cadeira ou pasta
éPublicadaEm	Uma publicação foi realizada num curso, ano ou cadeira
éPublicadoPor	Uma publicação tem como autor um utilizador

Table 4: *Data Properties* definidas na ontologia.

Data Property	Domínio	Descrição
anoLetivo	Ano	Ano letivo referente ao Ano
conteudo	Evento; Mensagem; Publicacao; Comentario	O conteúdo escrito destas classes
conteudosProgramaticos	Cadeira	Conteúdos programáticos ou matéria abordada
data	Ficheiro; Publicacao; Mensagem; Comentario	A data em que foi publicada/enviada
dataFim	Evento	Data em que um evento termina

dataInicio	Evento	Data em que um evento começa
dataNasc	Responsavel; Aluno; Admin	Data de nasicmento de um utilizador
nome	Responsavel; Ficheiro; Pasta; Admin; Ano; Evento; Aluno; Cadeira	As classes possuí um determinado nome
numAluno	Aluno; Admin; Responsavel	Identificador do aluno
numTelemovel	Aluno; Admin; Responsavel	Número de telemóvel do aluno
password	Aluno; Admin; Responsavel	Palavra passe encriptada de um utilizador
path	Ficheiro	Localização/Caminho onde o ficheiro está guardado
sexo	Aluno; Admin; Responsavel	Género do utilizador
size	Ficheiro	Tamanho de um ficheiro
titulo	Publicacao	O título atribuído à publicação
type	Ficheiro	Tipo de ficheiro (png, etc)

3.2 ApiDados

Concebido em *Node*, o servidor da *API* de dados tem a funcionalidade de, tal como o nome indica, servir os dados da aplicação (excepto as mensagens). Para tal, o servidor possui uma conexão com o servidor de *GraphDB*. Este servidor serve como uma camada de acesso sobre os dados, objectos e relações, da base de dados de grafos descrita na subsecção precedente.

3.3 Base de Dados Mongo - UMbook

Com a finalidade de garantir a persistência das conversas efectuadas na aplicação concebeu-se em *MongoDB* uma base de dados. Estes dados poderiam ser guardados na base de dados de grafos, porém, como é esperado um grande aumento no volume de dados relativos às conversas, o grupo enveredou pelo *MongoDB*,

uma vez que os sistemas de bases de dados *MongoDB* permitem uma grande escalabilidade dos dados.

Documentos Para tal elaboraram-se duas colecções de dados: as *Mensagens* e as *Conversas*.

Ambas as colecções possuem um formato de documento associado. Explicitam-se de seguida o campos de ambos os documentos.

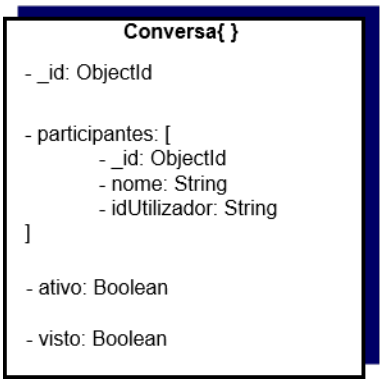

Fig. 2. Estrutura de um documento relativo a uma conversa.

No documento *Conversa* é possível verificar que existem 5 campos. Que podem ser descritos da seguinte maneira:

- **_id**: identificador único da conversa;
- **participantes**: dados relativos aos participantes da conversa;
  - **_id**: identificador único do participante na conversa;
  - **nome**: nome do participante;
  - **idUtilizador**: identificador do utilizador.
- **ativo**: indica se a conversa está ativa ou não;
- **visto**: indica se uma conversa já foi visualizada.

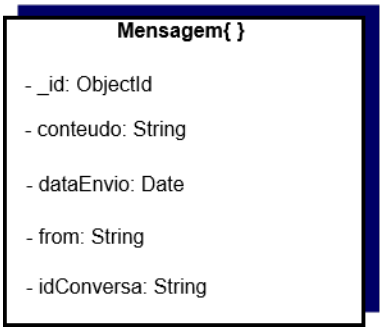

Fig. 3. Estrutura de um documento relativo a uma mensagem.

Por sua vez, no documento *Mensagem* também possui 5 campos:

- **_id**: identificador únicode uma mensagem;
- **conteudo**: conteúdo da mensagem;
- **dataEnvio**: data de envio da mensagem;
- **from**: identificador de quem enviou a mensagem;
- **idConversa**: identificador da conversa a que pertence.

3.4 ApiChat

A *ApiChat* serve os dados relativos às conversas. Este servidor possui uma conexão com o servidor *MongoDB Server* e opera sobre os dados todas as operações *CRUD*.

3.5 ApiLayer

A *ApiLayer* tem uma função simples, contudo, importantíssima na segurança da aplicação, servindo de *facade* e camada de protecção entre a *interface* e todos os outros servidores de *API*. É também neste servidor que se encontram as imagens de perfil associadas a cada utilizador.

3.6 Base de Dados Mongo - UMbook-AuthServer

A implementação da autenticação também foi um passo importante no que diz respeito a elevar a fasquia, melhorar a qualidade do trabalho e a segurança, como seria suposto.

No seguimento desta implementação construiu-se uma base de dados em *MongoDB* com a finalidade de armazenar os dados relativos ao *refresh token* (explicado adiante), enquanto que um utilizador autenticado utiliza a aplicação.

Documentos Esta base de dados possui uma colecção denominada *refreshtoken* que, por sua vez, armazena documentos com o formato explicitado na figura 4.

Fig. 4. Estrutura de um documento relativo a um refresh token.

Este documento compreende 4 campos:

- **_id**: identificador único de um *Refresh Token*
- **token**: *token string* associado ao utilizador autenticado
- **idUtilizador**: identificador do utilizador autenticado
- **estado**: estado do utilizador na aplicação

3.7 Auth Server

À semelhança dos outros servidores que servem dados neste projecto, também este é desenvolvido em *Node*. Este servidor retém uma ligação com o servidor *MongoDB Auth Server* e é nele que guarda os dados necessários. Este servidor não é responsável pela autenticação em si, contudo desempenha um papel vital no *silent refresh* da autenticação do utilizador.

4 Frontend

No *Frontend* encontra-se a parte gráfica e visual da aplicação. O grupo de trabalho compreendeu nesta secção o servidor de *interface*, isto é, o servidor que é responsável por fornecer a *interface* da aplicação quando solicitado.

4.1 InterfaceApp

A *InterfaceApp* é um servidor em *Vue.js* que fornece a *interface* gráfica aos utilizadores da aplicação.

Neste servidor através do consumo dos diversos dados das APIs, são apresentadas as diversas funcionalidades do sistema.

Assim, um utilizador é capaz de se registar (fornecendo as várias informações pedidas pelo sistema) e, de seguida, autenticar-se. Estando autenticado, depende

do tipo de utilizador e assim varia o número de funcionalidades que este pode aceder.

Tal como referido anteriormente, existem três tipos de utilizadores (Admin, Responsável e Aluno). A seguir, iremos enumerar e descrever as funcionalidades que estes utlizadores têm acesso.

Aluno:

- Aceder ao grupo geral da plataforma (UMinho).
- Aceder ao curso a que pertence e respetivos anos e cadeiras, assim como os estudantes e responsáveis.
- Pode aceder e marcar-se como participante nos eventos, sendo que estes estão divididos em três secções (Eventos Gerais, Evento do Curso e os seus Eventos).
- Pode aceder aos pedidos de amizade que recebeu e aceitá-los ou não, assim como pesquisar por outros utilizadores.
- Visualizar o seu perfil de utilizador e também altera as suas informações pessoais. Aqui, também pode alterar a sua foto de perfil e visualizar os seus amigos.
- Quando acede ao perfil de um utilizador, apenas se for amigo, pode visualizar as suas publicações no grupo público (UMinho).
- Realizar pedidos de amizade a outros utilizadores.
- De resto no grupo geral (Uminho) e no seu curso, pode fazer publicações com ou sem ficheiros, comentar publicações, apagar as suas publicações, assim como gostar destas.
- Pode falar através do chat com um amigo, assim como realizar uma vídeochamada com o mesmo.
- Pode pesquisar publicações pelo título.
- Tem permissão para publicar ficheiros nas pastas das cadeiras, assim como fazer download's dos ficheiros.
- O utilizador pode dar *logout*.

Responsável (além das funcionalidades do Aluno):

- Pode editar o ano pelo qual é responsável, como acrescentar/apagar cadeiras, editar a designação do Ano, etc.
- Pode editar as propriedades de uma cadeira, bem como acrescentar/apagar pastas associadas a uma determinada cadeira.
- Pode apagar publicações e comentários do ano que é responsável e também das cadeiras que lhe pertencem.

Admin (além das funcionalidades do Aluno):

- Pode realizar a gestão de Cursos, ou seja, visualizar, apagar, editar ou criar um curso.
- Pode acrescentar/editar/apagar anos a um curso.
- Tem permissão para acrescentar/editar/apagar cadeiras de um ano.
- Pode acrescentar/editar/apagar pastas de uma cadeira
- Pode inserir/editar/eliminar eventos.
- A única maneira de acrescentar novos responsáveis ou admin's é através do Admin.
- Pode apagar utilizadores do sistema.
- Pode apagar publicações e comentários de qualquer grupo ou curso.

5 Autenticação

De modo a proteger o acesso a dados da aplicação foi elaborado um sistema de autenticação de modo a verificar a identidade de cada utilizador. Para além disso, todas as *APIs* de dados foram devidamente protegidas recorrendo a *JSON WEB Tokens* (JWTs).

Efectivamente, a autenticação dos utilizadores é feita através de um email e uma password a qual é encriptada antes de ser guardada na base de dados, de modo a garantir que a password do utilizador fica protegida.

Aquando de um *login* efectuado com sucesso, é gerado um *Access Token (JWT)* associado ao utilizador autenticado. Este *token* guarda dados associados ao utilizador no seu payload, tais como Identificador, nível de acesso e curso tendo como tempo de expiração apenas 10 minutos. Um *token* de acesso deve ser *Short-Lived*, isto é, deve ter um tempo de expiração reduzido, na medida em que, este é guardado na *localStorage* do browser, a qual é acessível através de *JavaScript* ficado assim vulnerável a ataques maliciosos. Assim, um *token* de acesso não deve ter um tempo de expiração elevado, de modo a que, caso seja roubado, este expire rapidamente, minimizando assim os possíveis danos que poderiam ser causados.

De facto, esta politica de *tokens* de acesso promove a segurança do sistema, no entanto, levanta outras questões,como por exemplo, de que forma se pode evitar que o utilizador tenha de se autenticar de 10 em 10 minutos? Para resolver este problema foi elaborada uma solução com *Refesh Tokens* ilustrada no seguinte diagrama:

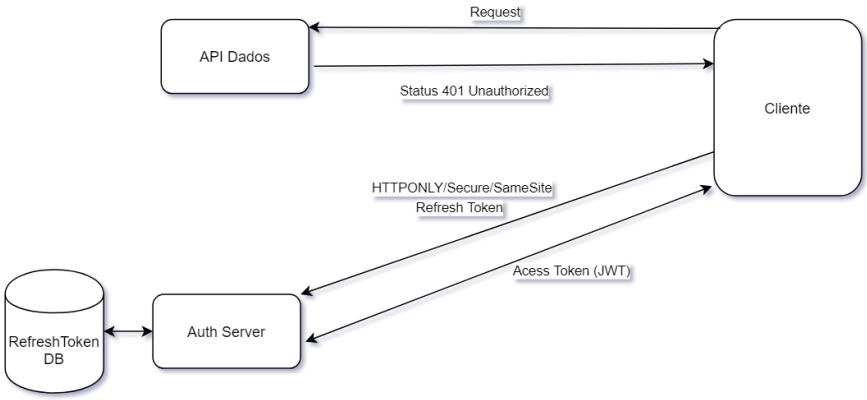

Fig. 5. Arquitetura de sistema de Autenticação.

O principal objectivo que se pretende atingir com esta solução é obter um *Access Token* de forma silenciosa através da utilização de um *Refresh Token*, *Long-Lived*.

Um *Refresh Token*, tal como o nome indica, permite refrescar ou gerar um novo *Access Token* de modo a envia-lo para a aplicação sem que utilizador se aperceba.

Efectivamente, introduziu-se um novo servidor de autenticação que é responsável pela geração dos tokens. Aquando de um login bem sucedido, para além de se gerar um *Access Token* é gerado um *Rerfesh Token* o qual é guardado numa base de dados Mongo. Para além disso, é enviado num *Cookie*, com as flags *HttpOnly*, *Secure* e *SameSite* para o cliente. Estas flags impões restrições aos browsers que recebem estes *Cookies*, sendo que neste caso restringem o acesso dos mesmos através de *JavaScript*, impõe uma requisição criptografada sobre um protocolo Https e restringe o domínio a partir do qual este *Cookie* pode ser enviado.

Resumindo, um cliente autenticado tem no seu LocalStorage um *Access Token*, o qual lhe permite fazer pedidos à API de dados, e um *Cookie* que guarda o *Refresh Token*, o qual é inacessível através de *JavaScript* (proteção contra ataques *XSS*), e não pode ser enviado a partir de outros domínios (proteção contra ataques *CSRF*). Para além disso, este tem de ser enviado obrigatoriamente através de Https tornando o canal de comunicação entre o cliente e o servidor de autenticação seguro.

Assim, quando um dado pedido à API de dados devolve erro um 401 (Unauthorized) sabe-se que o *Access Token* expirou pelo que se faz automaticamente um novo pedido ao servidor de autenticação com o *Refresh Token*. Caso o *Refresh Token* esteja na base de dados e seja válido, é gerado um novo *Access Token*, e enviado para o cliente o qual irá refazer o pedido original à API de dados

com o novo *token* de acesso. Caso o *Refresh Token* seja inválido, o utilizador é desconectado da aplicação em todos os dispositivos que esteja conectado.

Desta forma, atualiza-se o *Access Token* de forma silenciosa e o utilizador pode continuar a utilizar a aplicação sem qualquer interrupção.

6 Chat

Um dos objectivos projectados para o sistema é a facultação de uma forma de comunicação conveniente entre os seus utilizadores. Assim, foi adicionada a funcionalidade de enviar e receber mensagens em tempo real através de um *Chat* de mensagens.

Efectivamente, para enviar uma mensagem de um cliente para outro cliente, em tempo real, é utilizada uma conexão bidirecional *statefull* entre ambos que consiste na utilização de *WebSockets*. Com este protocolo é possível manter uma conexão entre o cliente e o servidor até que um deles se desconecte, mantendo a conexão *Alive* entre diferentes pedidos *Http* ou páginas. A figura abaixo ilustra a arquitectura utilizada.

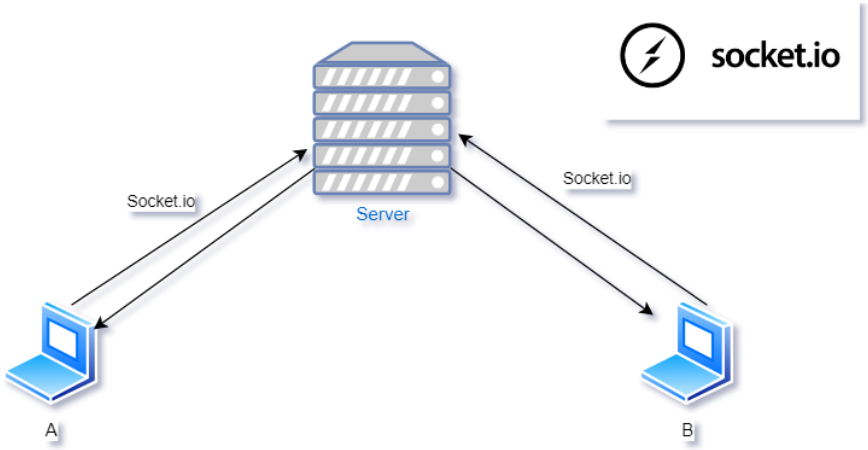

Fig. 6. Arquitetura de sistema de Chat.

Sempre que um utilizador entra na aplicação, é iniciada uma conexão entre ele e o servidor através de *WebSockets* (utilizando a libraria *Socket.io* do *Node.js*) associando a cada *socket* o identificador de utilizador. Quando uma mensagem é enviada, esta é sempre acompanhada com o identificador do destinatário da mensagem. Desta forma, o servidor sabe qual é o *socket* que deve utilizar de modo a enviar a mensagem para o destinatário correto.

Para além disso, durante este processo, o servidor guarda numa base de dados *Mongo* todas as mensagens que estão a ser enviadas bem como o seu remetente e destinatário de modo a guardar histórico de todas as mensagens enviadas.

7 Video Chat

Para além da implementação do *Chat* de mensagens foi ainda implementado um Vídeo *Chat* de modo a permitir aos utilizadores fazerem vídeo chamadas entre si.

Em contraste com o *Chat* de mensagens implementado, transferir áudio e vídeo em tempo real deve ser feito utilizando um protocolo UDP. Para além disso, este tipo de dados requer uma quantidade de recursos bastante elevado comparado a simples mensagens de texto pelo que, ter um servidor a mediar todos os fremes de cada vídeo tornar-se-ia dispendioso e difícil de escalar (deve ser feito em vídeo chamadas com muitos participantes). Isto faz com que a utilização de *WebSockets* não seja adequada para este caso pelo que

Deste modo, para desenvolver um sistema de video chamada utilizou-se a API *WebRTC* (Web Real Time Communication) desenvolvida pela *W3C*. Esta API consiste num projecto *Open Source* a qual permite criar comunicações *Peer to Peer* entre browsers, ou seja, permite a transferência de dados tais como vídeo e áudio entre clientes.

A seguinte figura ilustra a arquitectura que se utilizou.

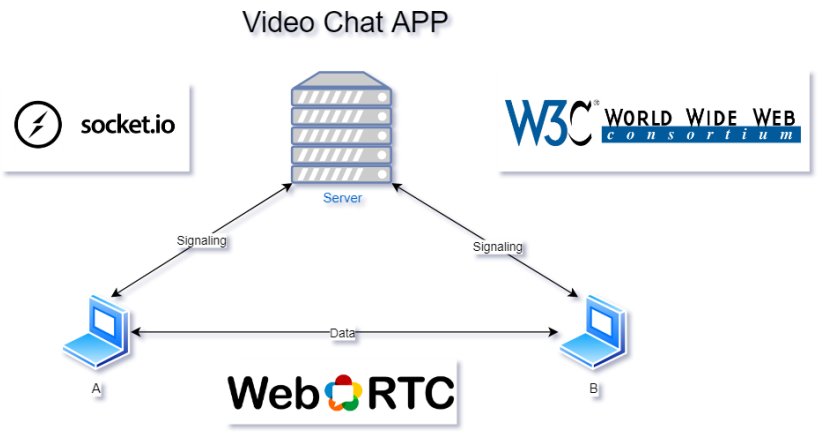

Fig. 7. Arquitetura de Video Chat.

Para se estabelecer uma conexão entre dois clientes utilizando WebRTC é necessário estabelecer um processo de negociação entre os dois clientes denominado *signalling*. Este processo consiste em trocar dados entre os clientes, através de um *signalling server*, os quais permitam ambos os clientes saber como comunicarem entre si. Este processo não é especificado na API do WebRTC pelo

que existe liberdade na forma como é implementado. Neste caso foram utilizados WebSockets.

O seguinte diagrama demonstra como foi realizado o processo de *signalling* neste sistema.

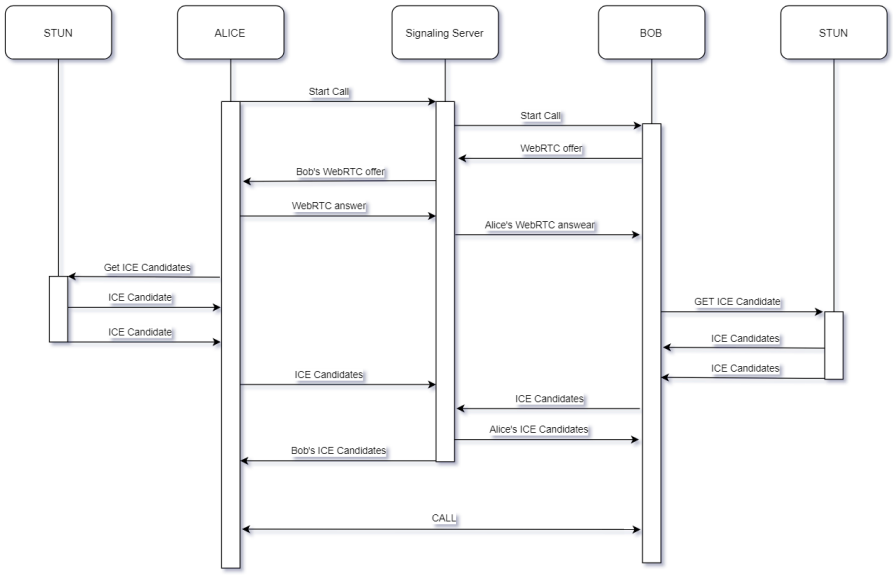

Fig. 8. Arquitetura de Video Chat.

Como se pode ver no diagrama, a Alice envia um pedido de chamada para o *signalling server* o qual é reencaminhado para o socket do Bob. Caso o Bob aceite realizar a video chamada, este cria uma *offer* utilizando o *Session Description Protocol (SDP)* e envia-a para a Alice a qual responde com uma resposta que também contem um *SDP*. Um SDP contém dados sobre o tipo de coddecs de audio e video bem como a topologia da rede e outros dados de cada cliente necessários para negociar de que forma se dará a conexão. Neste ponto, o Bob e a Alice conhecem e negociaram que tipo de tecnologias serão usadas para realizar a transmissão, no entanto, falta agora saber como comunicar diretamente um com o outro, isto é, precisam de saber os endereços IP, *external NATs*, restrições de portas etc.

Um dos desafios do *WebRTC* consiste em lidar com este problema. De facto, uma aplicação que forçasse os seus utilizadores a abrir portas no seu router não seria viável de todo. Para resolver este problema é então necessário aderir ao *ICE* (Interactive Connectivity Establishment).

ICE é um método de comunicão entre *peers* para enviar e receber dados através da WEB. Esta técnica consiste em recolher conexões de rede disponíveis conhecidas como *ICE candidates* e usar protocolos *STUN* (Session Traversal

Utilities for NAT) e *TURN* (Traversal Using Relays around NAT) para passar pela *NAT* e potenciais firewalls.

Na seguinte figura tem-se um esquema deste método.

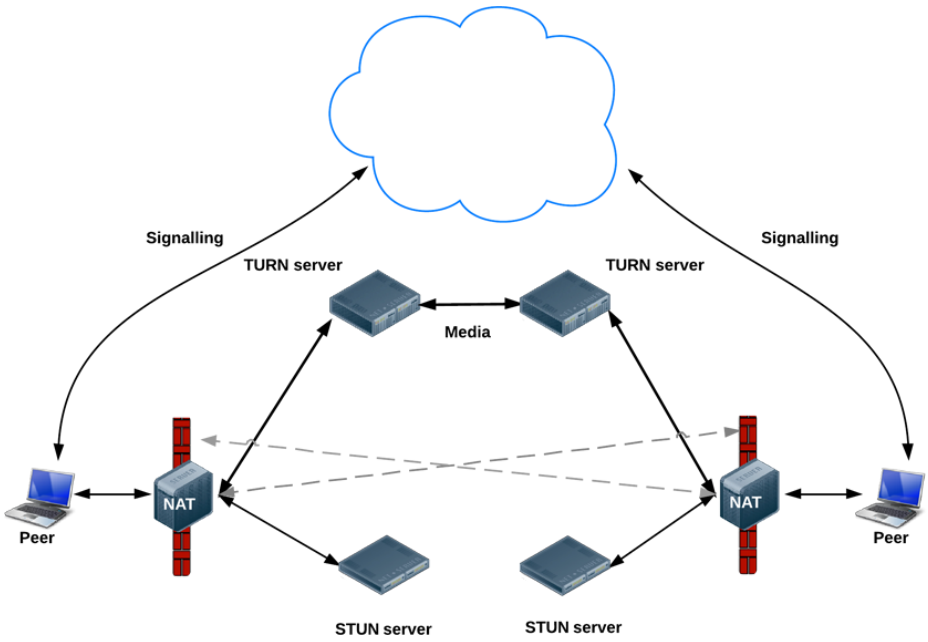

Fig. 9. Interactive Connectivity Establishment.

Em primeiro lugar, o protocolo *STUN* consiste em aceder a servidores *STUN*, (neste caso usaram-se servidores da google) para que estes respondam com informação sobre o endereço de IP público bem como que tipo de *NAT* está à frente do cliente correspondente. Posteriormente, os clientes partilham esta informação entre eles de modo a saberem como estabelecer uma conexão.

Em segundo lugar, tem-se os servidores *TURN*. Estes servidores são usados quando não se consegue estabelecer uma conexão entre os clientes devido a restrições da rede. A sua principal função consiste em dar " *relay* a uma *stream* de dados. De facto, estes funcionam como uma *callback* quando não se consegue estabelecer uma conexão com o método anteriormente descrito. Importante referir que um servidor *TURN* consome grandes quantidades de *bandwidth* e capacidade de processamento pelo que costumam ter um custo elevado. Para além disso, quando se pretende estabelecer uma video-chamada com muitos elementos, deve-se utilizar um servidor *TURN* pois, normalmente, a *bandwidth* disponível para utilizadores normais não aguenta uma transferência de dados deste calibre para muitos clientes em simultâneo.

De modo a estabelecer-se um sistema de vídeo-chamadas fiável é necessário usar estes dois protocolos de modo a que, caso o *STUN* falhe, o servidor *TURN* seja utilizado como *backup*.

Assim, no final de ambos os clientes saberem as tecnologias a utilizar na comunicação e como podem estabelecer uma conexão entre eles é então iniciada a video-chamada.

8 Docker

O *Docker* é um *software* que através de *containers* fornece uma camada de abstracção para a virtualização de sistemas operativos.

Um *container* é a unidade *standard* de *software* que embala o código e todas as suas dependências para que a aplicação execute rapidamente e de uma forma confiável em diferentes ambientes.

A utilização do *Docker* detém vários benefícios, tais como:

- **Portabilidade**: uma aplicação que esteja a correr em *Docker* pode ser executada em qualquer sistema, desde que este tenha o *software Docker*;
- **Performance**: máquinas virtuais podem ser utilizados de forma alternativa aos *containers*, contudo um *container* não possui um sistema operativo, ao contrário de máquinas virtuais, o que faz com que sejam rápidos de criar e mais rápidos a iniciar;
- **Isolamento**: Um *Docker container* que contém uma aplicação também contém as versões de qualquer outro *software* que também seja relevante para o funcionamento da aplicação. Se outra imagem necessitar de versões diferentes do mesmo *software* isso não se apresenta como um problema, pois os *containers* são completamente independentes;
- **Escalabilidade**: é possível criar novos *containers* rapidamente se necessário.

Como se pode afirmar a partir da figura 12 o *Docker* é um programa sobre o qual correm os *containers* com as respectivas aplicações.

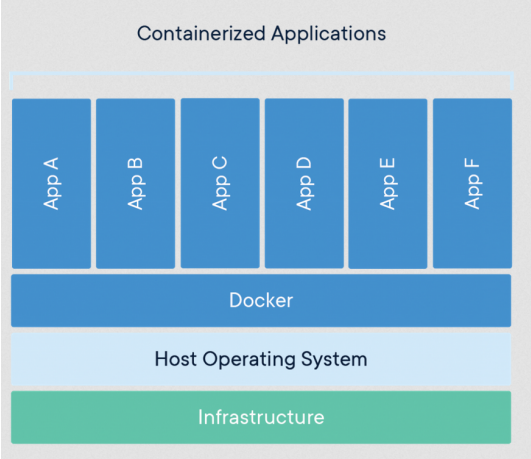

Fig. 10. Funcionamento geral do Docker.

Docker Compose O *Docker compose* é uma ferramenta para definir e executar aplicações multi-container. Com o *compose* é possível configurar todas as aplicações com apenas um ficheiro *YAML*, ulteriormente com apenas um comando é possível criar e executar todos os serviços presentes na configuração.

Estrutura Neste projecto definiram-se tantos serviços como servidores, isto é, oito. Para tal, teve de se construir um ficheiro *YAML* com as definições de todos os serviços.

A título de exemplo apresenta-se na figura seguinte um excerto do ficheiro *YAML* elaborado.

Listing 1.1. "Conteúdo do ficheiro docker-compose."

```
version: '3.3'

services:

    umbook-graphdb:
        build: ./ontologia
        container_name: umbook-graphdb
        image: bragamann/umbook-graphdb:2020-07-03
        ports:
            - "7200:7200"
        networks:
            - graphdb
        volumes:
            - graphdbvol:/opt/graphdb/home

    ...
```

No *listing* acima pode verificar-se que a versão do *compose* utilizada é a 3.3 e que o serviço definido é o *umbook-graphdb* (serviço do servidor de *GraphDB*). Da mesma forma, é possível ver outras definições como: *build*, que é a pasta onde se situa o *dockerfile* do serviço em questão; o *container_name*, que é o nome do

container onde vai correr o serviço; *image*, nome da imagem necessária para o *container* e a conta *dockerhub* onde se situa; *ports* que configura as portas do serviço; *networks* que são as redes que permitem que os vários serviços comuniquem entre eles (especificadas mais adiante) e os *volumes* que são mecanismos para a persistência dos dados (explicados na secção seguinte).

Volumes No *Docker* existe um mecanismo chamado *volume* que garante a persistência de dados e, por isso, é imprescindível a sua utilização numa aplicação que contenha bases de dados. Neste projecto foram utilizados dois tipos de *volume*: *named volume* (especificado no *compose*) e no *host* na forma de uma pasta onde são guardados ficheiros.

Listing 1.2. "Tipos de volumes no ficheiro docker-compose."

```
# named volume
- graphdbvol:/opt/graphdb/home

# volume com caminho definido relativamente ao ficheiro compose
- ./apiDados/ficheiros:/app/ficheiros
```

Networks De forma aplicações comunicarem também se definiu uma rede da seguinte forma no ficheiro *compose*:

Listing 1.3. "Networks no ficheiro docker-compose."

```
...
networks:
    frontend:
    mongo:
    mongoauth:
    backend:
    graphdb:
...
```

No diagrama abaixo é clarificado que servidores estão incluídos em que redes. Foram configuradas 5 redes:

- **frontend**
- **backend**
- **mongo**
- **mongoauth**
- **graphdb**

As redes foram configuradas a manter de forma a não incluir servidores que fossem desnecessários à rede, de maneira a que os servidores comuniquem apenas com quem têm de comunicar.

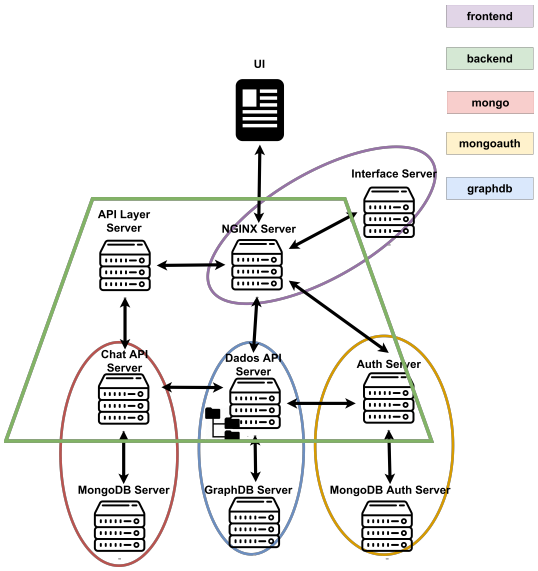

Fig. 11. Redes definidas no compose.

9 Swagger

O **Swagger** é uma ferramenta que permite documentar uma API e, assim, descrevê-la em detalhe a sua estrutura, o que representa uma grande vantagem, dado que, desta forma, consegue-se visualizar todas as rotas, bem como todos os parâmetros e, assim, visualiza-se de forma fácil à especificação de cada rota. Além disso, também pode ser bastante útil para testar se as rotas estão a cumprir o desejado.

Para construirmos a documentação da nossa API, foi necessário construir um ficheiro de formato *JSON* (também podia ter sido construído em *YAML*). Esse ficheiro possuí o título da API e uma breve descrição, assim como o host (endereço e porta da API) para onde os pedidos vão ser realizados. Além disso, possuí uma lista de tags que serão associadas às várias rotas que a API possuí. Também possuí os "paths" que são as rotas e dentro de cada rota, contem uma breve descrição do que a rota faz, os parâmetros necessários, as possiveís respostas e o tipo de conteúdo dessas respostas, assim como se é necessário receber um token de autorização para efetuar o pedido. Possuí também uma definição da segurança utilizada que basicamente é um token de autorização e é especificado onde ele é enviada (no nosso caso é enviada na querystring) e também o nome dela. Por último, está presente um conjunto de definições que é bastante útil para referir no tipo de respostas e também é apresentado, podendo-se visualizar o tipo das estruturas de dados presentes na base de dados.

Na seguinte fração de código, pode-se visualizar algumas definições contidas no ficheiro *JSON*.

Listing 1.4. "Conteúdo do ficheiro swagger.json."

```
1                                       ...
2     "host": "localhost:3049",
3     "basePath": "/api",
4     "tags": [
5         {
6           "name": "utilizadores",
7           "description": "Todas as opera  es    cerca de utilizadores.
      "
8         },
9         {
10          "name": "cursos",
11          "description": "Todas as opera  es    cerca de cursos."
12        },
13        {
14          "name": "anos",
15          "description": "Todas as opera  es    cerca de anos."
16        },
17        {
18            "name": "cadeiras",
19            "description": "Todas as opera  es    cerca de cadeiras."
20        },
21        {
22            "name": "eventos",
23            "description": "Todas as opera  es    cerca de eventos."
24        },
25        {
26            "name": "publicacoes",
27            "description": "Todas as opera  es    cerca de
      publica  es."
28        },
29        {
30            "name": "conversas",
31            "description": "Todas as opera  es    cerca de conversas."
32        }
33      ],
34                                      ...
```

De seguida, na próxima imagem, pode-se verificar o resultado final da construção do ficheiro *JSON* e respetiva integração no servidor.

Fig. 12. Interface gerada pelo Swagger.

10 Conclusão

Durante a resolução deste trabalho vários foram os *stumbling blocks* encontrados e várias vezes o grupo teve de repensar passos feitos anteriormente.

O grupo de trabalho após analisar o problema e examinar qual as tarefas a realizar de uma forma geral, optou por explorar a ontologia por ser uma matéria nova com a qual nenhum dos elementos do grupo possuía experiência. Elaborou-se uma ontologia base da qual resultou a ontologia final depois de vários avanços e retrocessos.

Uma das tarefas iniciais do projecto foi também a realização de um esboço da arquitectura. Numa fase inicial apenas se planeou dois servidores (interfaceapp e apiDados), contudo ao longo do tempo foram-se acrescentando mais servidores que fossem pertinentes e úteis para atingir os objectivos do sistema.

O *Vue* também foi um desafio para a equipa, pois foi uma *framework* nova para os elementos do grupo. Esta *framework* também possui uma curva de aprendizagem um pouco acentuada, pelo que foi necessário que os elementos dedicassem algum tempo para a aprendizagem da mesma.

Uma das fases mais trabalhosas deste projecto foi a autenticação. Primeiramente, a autenticação era feita com um *token* dado pela aplicação ao utilizador aquando do *login*. Este teria um tempo limitado a uma hora, sendo que após este tempo o *token* expirava e o utilizador teria de realizar novamente o *login*. No entanto, o coletivo não ficou satisfeito com a solução e implementou o *refresh token*, o que implicou mais dois servidores (auth server e mongoauth) e que houvesse um *silent refresh* do *token* e. assim sendo, o utilizador não necessita de efectuar novamente o *login*. Outro benefício desta última implementação foi também uma segurança mais robusta.

Numa fase final, decidiu-se por utilizar *docker* e a sua tecnologia de *containers* para aprimorar o projecto.

Assim, as dificuldades durante o desenvolvimento foram várias, mas poder-se-á dizer que as aprendizagens também.

References

1. Protégé, https://protege.stanford.edu/. Last accessed 7 Julho 2020
2. GraphDB, https://www.ontotext.com/products/graphdb/. Last accessed 18 Julho 2020
3. Stack Overflow, https://pt.stackoverflow.com/. Last accessed 7 Julho 2020
4. Vue Material Design Component Framework, https://vuetifyjs.com/en/. Last accessed 7 Julho 2020
5. Vue.js, https://vuejs.org/. Last accessed 7 Julho 2020
6. What is Vuex?, https://vuex.vuejs.org/. Last accessed 7 Julho 2020
7. Docker Homepage, https://www.docker.com/. Last accessed 19 Julho 2020
8. Overview of Docker Compose, https://docs.docker.com/compose/. Last accessed 19 Julho 2020
9. Swagger, https://swagger.io/. Last accessed 19 Julho 2020
10. MongoDB, https://www.mongodb.com/. Last accessed 3 Julho 2020
11. Dropbox Engenharia Informática, https://www.dropbox.com/sh/akiqghi79eo8ti8/AABZiHu8mU3Bu4lblNZQqp0Ha?dl=0. Last accessed 10 Julho 2020.
12. Documentação NGINX, http://nginx.org/en/docs/beginners_guide.html. Last accessed 21 Julho 2020.