Universidade do Minho

Mestrado em Engenharia Informática

Processamento e Representação de Conhecimento

Musike - Music Linked

José Carlos Lima Martins
A78821

13 de Junho de 2019

1 Introdução

O seguinte trabalho, apresenta a construção de uma ontologia, ou seja, um dataset em *RDF*, a partir de um dataset em *JSON* do `MusicBrainz` bem como, a criação de uma aplicação que faz uso da informação proveniente desta ontologia, recorrendo a *Noje.js* e a *Vue.js*.

Portanto, na secção 2, realiza-se uma pequena contextualização do trabalho, enquanto que na secção 3, descreve-se o tema escolhido e os objetivos com a realização do trabalho. Por outro lado, na secção 4.1, apresenta-se a estrutura na ontologia, para de seguida, na secção 4.2, povoá-la com o dataset *JSON* do `MusicBrainz`.

Por outro lado, na secção 5 refere-se a construção da `API`, parte integrante da aplicação, para seguidamente, na secção 6, apresentar a interface que comunica com esta `API`, por forma a obter a informação necessária a apresentar.

Por fim, refere-se vários métodos de instalação da secção 7 e apresenta-se trabalho a realizar futuramente na secção 8.

Conteúdo

2 Contextualização

O trabalho a desenvolver passa por escolher uma área/tema de trabalho. A partir desse tema, criar/construir/limpar/extrair uma ontologia sobre o tema e que pode estar disponível na LOD (*Linked Open Data*). Esta ontologia, em Turtle (*Terse RDF Triple Language*, sintaxe e formato de ficheiros para expressar dados em `RDF - Resource Description Framework`), será armazenada em *GraphDB*, base de dados de semântica baseada em grafos, onde será possível aceder/realizar *queries* à ontologia.

De seguida, será criado um web site/app de forma a explorar a ontologia anteriormente criada. Este web site deve possuir autenticação, usando para isso o *MongoDB*, base de dados baseada em documentos.

3 Tema escolhido - Musike

O **Musike**, abreviatura de *Music Linked*, consiste num web site com a informação de artistas, bem como, das suas músicas e dos seus albuns. Em termos de informação a apresentar, pretendesse o seguinte:

- Artista
  - nome
  - alias
  - tipo de artista (Grupo, Pessoa, etc)
  - data de nascimento/inicio
  - data de falecimento/fim
  - sexo (não é aplicável a todos os tipos de artista)
  - nacionalidade (obtido através da área)
  - descrição
  - urls para página pessoal, redes sociais, etc
  - classificação dos utilizadores do web site (média das classificações das músicas)
  - soma das visualizações das várias músicas
- Album
  - título
  - data do primeiro "release"
  - artista(s)
  - descrição
  - músicas do album
  - tipos (tags) do album (clássica, rock, etc)
  - urls sobre o album
  - classificação dos utilizadores do web site (média das classificações das músicas)
  - soma das visualizações das várias músicas

- Música
  - título
  - artista(s)
  - duração
  - descrição
  - língua(s)
  - tipos (tags) da música (clássica, rock, etc)
  - urls sobre a música
  - classificação dos utilizadores do web site
  - número de vezes ouvida pelos utilizadores do web site
- Área
  - nome
  - tipo (país, cidade, etc)
  - alias
  - data de criação
  - data de extinção
  - descrição
  - urls sobre a área
  - relações com outas áreas, pois uma área pode ser parte de outra e, vice-versa, uma área pode incluir várias áreas

Para além disso, para cada música o objetivo é ter na sua página o vídeo presente no *YouTube* bem como a letra da música. Com isto, pretendesse que por cada visualização do vídeo se conte que a música foi ouvida uma vez.

Por outro lado, o web site deve permitir aos utilizadores escolher as músicas que mais gosta, bem como, puder criar *playlists*.

Cada utilizador deve ter acesso às suas estatísticas, onde deve estar presente as músicas, artistas e albums que o utilizador ouve mais, bem como, os artistas, albuns e músicas com melhor classificação dada pelo utilizador.

Por fim, devem ser apresentadas estatísticas gerais do web site entre as quais:

- músicas mais ouvidas pelos utilizadores
- músicas com melhor classificação dada pelos utilizadores
- artistas mais ouvidos pelos utilizadores
- artistas com melhor classificação dada pelos utilizadores
- tipos de músicas (tags) mais ouvidos pelos utilizadores
- tipos de músicas (tags) com melhor classificação dada pelos utilizadores
- albuns mais ouvidos pelos utilizadores
- albuns com melhor classificação dada pelos utilizadores
- países com mais artistas
- países com mais músicas
- países com mais albuns
- países mais "ouvidos" pelos utilizadores
- países com melhor "classificação" dada pelos utilizadores

De forma adicional, seria interessante dar sugestões ao utilizador de músicas a ouvir a partir das estatísticas do utilizador.

4 Ontologia baseada no `MusicBrainz`

Para o tema escolhido é necessário um dataset que possua a informação de albuns, artistas, músicas e áreas. Como tal foi escolhido o dataset JSON do `MusicBrainz`. O `MusicBrainz` é uma enciclopédia de música que coleciona metadados de músicas e torna-as disponíveis para o público. É mantida pela comunidade e, para além do dataset, possui uma API e uma plataforma online para aceder aos dados, bem como, máquinas virtuais caso se pretenda hospedar a informação. Esta plataforma é usada por várias outras grandes plataformas online/empresas, tais como, `last.fm`, `Amazon`, `Universal Music`, `Spotify`, `BBC`, `Google`, etc.

A partir deste dataset (do `MusicBrainz`) em JSON bem como das características do tema escolhido, foi construído a estrutura da ontologia e, posteriormente, foi povoada com a informação presente no dataset.

4.1 Parte Estrutural da Ontologia

O primeiro passo foi a construção da estrutura da ontologia baseada no dataset JSON do `MusicBrainz` [5]. O desenho da estrutura pode ser visualizado de seguida:

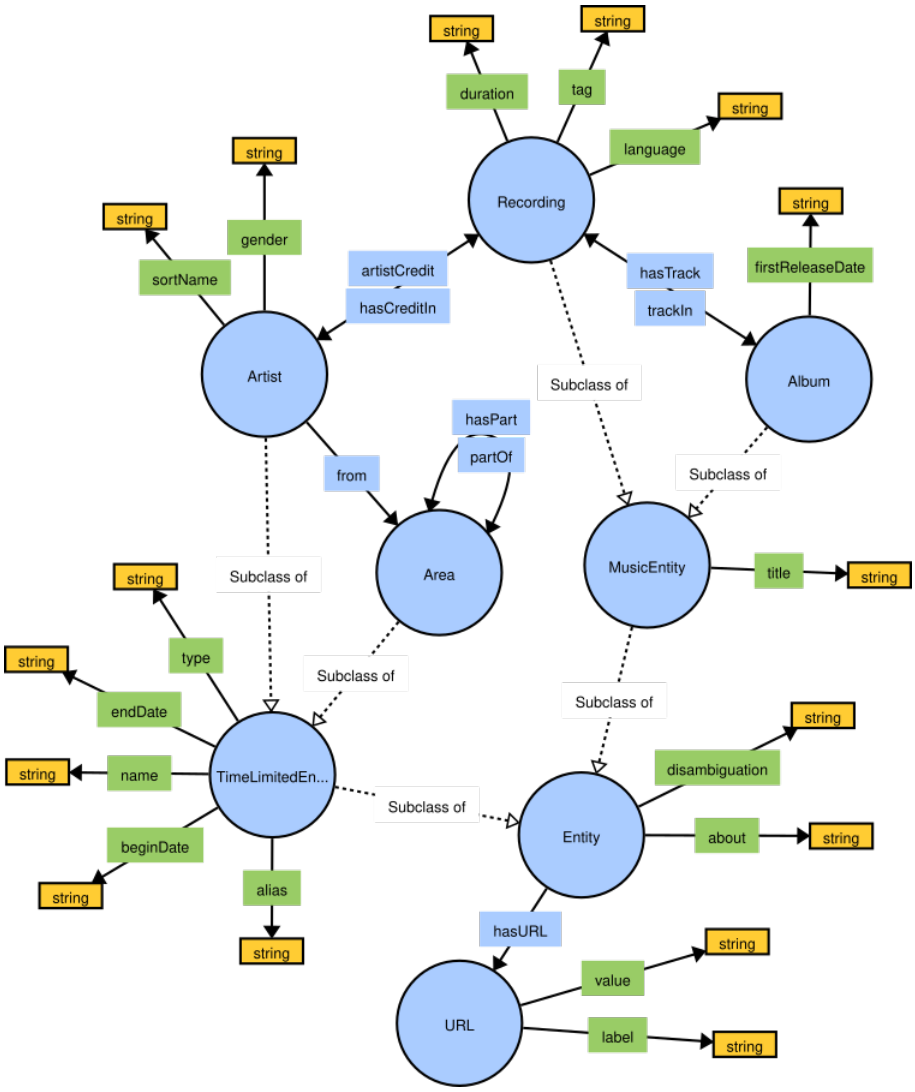

Figura 1: Estrutura da ontologia, imagem obtida através de WebVOWL

As classes principais da ontologia são *Artist*, *Recording*, *Album* e *Area* que representam respetivamente Artista,

Música, Album e Área. Uma classe também importante mas secundária, é a classe *URL* que representa URL's. Por forma a organizar melhor as propriedades, visto que as subclasses herdam as propriedades/relações das superclasses, foram criadas as superclasses *Entity*, *MusicEntity* e *TimeLimitedEntity*. A classe *MusicEntity* inclui as subclasses *Recording* e *Album* devido às duas possuírem a propriedade *title*. Por outro lado, a classe *TimeLimitedEntity* tem como subclasses *Artist* e *Area*, visto que estas duas classes possuem uma data de inicio e de fim. Já a classe *Entity* inclui as subclasses *MusicEntity* e *TimeLimitedEntity*, ou seja, é a superclasse das quatro principais classes, até porque estas quatro classes possuem as propriedades *about* e *disambiguation*.

Quanto às propriedades/relações, a ontologia apresenta as seguintes:

- Propriedades (*Data Properties*)
  - Classe ***Entity***
    - ***about***: Descrição do elemento
    - ***disambiguation***: Forma de desambiguar entre elementos com o mesmo nome
  - Classe ***URL***
    - ***label***: o nome do web site a que se refere o URL (exe: wikipédia)
    - ***value***: URL propriamente dito
  - Classe ***TimeLimitedEntity***
    - ***name***: Nome do elemento
    - ***alias***: nomes alternativos e nomes com erros ortográficos, permitindo uma melhor busca quando o utilizador introduz o nome com erros
    - ***type***: Tipo do elemento (em *Area*: district, country, city, etc, já em *Artist*: person, group, choir, etc)
    - ***beginDate***: Data de inicio
    - ***endDate***: Data de fim
  - Classe ***MusicEntity***
    - ***title***: Título
  - Classe ***Album***
    - ***firstReleaseDate***: Data da primeira "release" do Album
  - Classe ***Recording***
    - ***duration***: Duração da música
    - ***language***: Língua(s) da música
    - ***tag***: Tag(s) (tipos) da música
  - Classe ***Artist***
    - ***gender***: Género do artista
    - ***sortName***: Nome de modo a ordenar o artista numa lista
- Relações (*Object Properties*)
  - ***from***: De *Arist* para *Area*, indica que um artista é da área
  - ***hasURL***: De *Entity* para *URL*, indica que uma entidade tem o URL
  - ***hasPart***: De *Area* para *Area*, indica que uma área inclui a outra
  - ***partOf***: Inverso de *hasPart*, indica que uma área é parte de outra
  - ***artistCredit***: De *Recording* para *Artist*, indica que a música tem como crédito o artista
  - ***hasCreditIn***: Inverso de *artistCredit*, indica que um artista tem crédito na música
  - ***hasTrack***: De *Album* para *Recording*, indica que um album tem a música
  - ***trackIn***: Inverso de *hasTrack*, indica que uma música pertence a um album

É importante voltar a referir que as subclasses herdam as propriedades das superclasses.

4.2 De JSON para Turtle - Povoamento da Ontologia

A partir da estrutura da ontologia e da informação necessária, foi então usado um dataset (datasets JSON do `MusicBrainz` disponíveis em `http://ftp.musicbrainz.org/pub/musicbrainz/data/json-dumps/`) para popular a ontologia. Este dataset é proveniente do **MusicBrainz** estando o mesmo em JSON. O seu timestamp é de 2019-04-03, portanto um dataset bastante recente possuindo cerca de 240GB de tamanho. Visto o mesmo estar em JSON é necessário então convertê-lo para Turtle.

Num primeiro passo, percorresse os ficheiros JSON (possui em cada linha um elemento de tipo igual ao nome do ficheiro a percorrer) com os conversores criados em *Node.js*. Foi, então, criado um conversor por ficheiro a converter. Apresentam-se os seguintes conversores, que para cada linha do ficheiro converte a informação para (assuma: a converter → conversão) [5, 8]:

- Conversor para area.json (*jsonTOturtle/area.js*) [2]:
  - *id* → "area_" + *id* de forma a identificar o indivíduo
  - *name* → *data property* ***name***
  - *type* → *data property* ***type***
  - *aliases* (lista em que de cada elemento usasse o *name* e o *locale*) → *data property* ***alias*** (um por elemento da lista no seguinte formato: "*name*(*locale*)")
  - *life-span.begin* → *data property* ***beginDate***
  - *life-span.end* → *data property* ***endDate***
  - *annotation* → *data property* ***about***
  - *disambiguation* → *data property* ***disambiguation***
  - *relations* (lista) em que os elementos que possuírem:
    - *type* igual a "part of" ou item *area*: são transformados em relações entre indivíduos áreas usando *area.id* para saber qual o id da área destino; para além disso, observasse a *direction* por forma a saber a direção da relação → *object property* ***partOf*** (*direction* "backward") ou ***hasPart*** (*direction* "forward")
    - item *url*: guardasse o *url.id* (identificador do url: "url_" + *url.id*), *type* (***label*** de URL) e *url.resource* (***value*** de URL) por forma a posterior criação do indivíduo URL e usasse *url.id* para criar a relação entre area e URL a ser criado → *object property* ***hasURL*** + Indivíduos da classe *URL*
- Conversor para artist.json (*jsonTOturtle/artist.js*) [3]:
  - *id* → "artist_" + *id* de forma a identificar o indivíduo
  - *area.id* → *object property* ***from***
  - *name* → *data property* ***name***
  - *type* → *data property* ***type***
  - *aliases* (lista em que de cada elemento usasse o *name* e o *locale*) → *data property* ***alias*** (um por elemento da lista no seguinte formato: "*name*(*locale*)")
  - *life-span.begin* → *data property* ***beginDate***
  - *life-span.end* → *data property* ***endDate***
  - *annotation* → *data property* ***about***
  - *disambiguation* → *data property* ***disambiguation***
  - *sort-name* → *data property* ***sortName***
  - *gender* → *data property* ***gender***
  - *relations* (lista) em que os elementos que possuirem:
    - item *url*: guardasse o *url.id* (identificador do url: "url_" + *url.id*), *type* (***label*** de URL) e *url.resource* (***value*** de URL) por forma a posterior criação do indivíduo URL e usasse *url.id* para criar a relação entre area e URL a ser criado → *object property* ***hasURL*** + Indivíduos da classe *URL*

- Conversor para recording.json (*jsonTOturtle/recording.js*) [7]:
  - *id* → "recording_" + *id* de forma a identificar o indivíduo
  - *title* → *data property* ***title***
  - *length* → *data property* ***duration***
  - *annotation* → *data property* ***about***
  - *disambiguation* → *data property* ***disambiguation***
  - *tag* (lista, em que para cada elemento usasse *name*) → *data property* ***tag*** (um por cada elemento da lista)
  - *artist-credit* (lista, para cada elemento usasse *artist.id* de forma a associar ao artista) → *object property* ***artistCredit*** (um por cada elemento da lista)
  - *relations* (lista) em que os elementos que possuirem:
    - item *url*: guardasse o *url.id* (identificador do url: "url_" + *url.id*), *type* (***label*** de URL) e *url.resource* (***value*** de URL) por forma a posterior criação do indivíduo URL e usasse *url.id* para criar a relação entre area e URL a ser criado → *object property* ***hasURL*** + Indivíduos da classe *URL*
    - *type* igual a "performance": usasse o *language* e *languages* (lista) → *data property* ***language*** (por cada elemento)
- Conversor para release-group.json (*jsonTOturtle/release-group.js*) [10]:
  - *id* → "album_" + *id* de forma a identificar o indivíduo
  - *title* → *data property* ***title***
  - *first-release-date* → *data property* ***firstReleaseDate***
  - *annotation* → *data property* ***about***
  - *disambiguation* → *data property* ***disambiguation***
  - *relations* (lista) em que os elementos que possuirem:
    - item *url*: guardasse o *url.id* (identificador do url: "url_" + *url.id*), *type* (***label*** de URL) e *url.resource* (***value*** de URL) por forma a posterior criação do indivíduo URL e usasse *url.id* para criar a relação entre area e URL a ser criado → *object property* ***hasURL*** + Indivíduos da classe *URL*
- Conversor para release.json (*jsonTOturtle/release.js*) [9]:
  - *release-group.id*
  - *media* (lista, em que *tracks* (lista) possui recordings e para cada recording usar *recording.id* de forma a criar a relação entre album (*release-group.id*) e recording; para além disso para cada recording realizar o mesmo que em recording.json, ou seja criar um indivíduo *Recording*) → *object property* ***hasTrack*** + Indivíduos da classe *Recording*
- Conversor para work.json (*jsonTOturtle/work.js*) [11]:
  - *language*
  - *languages* (lista)
  - *relations* (lista) em que os elementos que possuírem:
    - *type* igual a "performance": usasse o *recording.id* para o id do recording → *data property* ***language*** (por cada elemento, o recording com "id" tem como línguas *language* e *languages*)

Nestes conversores, grande parte dos campos do dataset JSON são verificados se são nulos ou vazios antes de converter, por forma a garantir que é inserido na ontologia informação válida e útil.

Ainda nestes conversores, os ids são verificados, visto que, às vezes há "merges" de entidades no **MusicBrainz** e, quando isso acontece, passa a existir uma só entidade que representa as duas mas, é possível aceder à mesma através dos dois ids de cada entidade do "merge". De tal forma, para obter uma ontologia consistente, quando aparecem ids nos conversores acedesse o URL `https://musicbrainz.org/ws/2/entidade/id` usando o *axios* e, do campo

`data` da resposta, verificasse o campo `id`. Caso consigasse obter resposta usasse o id obtido, caso contrário (pode acontecer por exemplo quando a entidade não existe num dataset mais recente), mantém-se o atual. Contudo, o acesso através deste URL tem uma grande limitação, apenas se pode realizar um pedido a cada segundo, devido a restrições impostas pelo **MusicBrainz** [12].

Portanto, por forma a contornar este obstáculo, obteve-se a máquina virtual disponibilizada em `ftp://ftp.eu.metabrainz.org/pub/musicbrainz-vm/musicbrainz-server-2018-08-14.ova` ou através do *torrent* disponibilizado em `ftp://ftp.eu.metabrainz.org/pub/musicbrainz-vm/musicbrainz-server-2018-08-14.ova.torrent`. Esta máquina virtual possui o servidor do `MusicBrainz` já pronto a correr, bastando apenas iniciar a máquina virtual e autenticar-se com username e password igual a `vagrant` [6]. Porém, como o dataset que vinha com o servidor da máquina virtual era anterior ao dataset em que o trabalho se baseia, atualizou-se o dataset da máquina virtual ao correr os seguintes comandos na máquina virtual [4]:

```
cd musicbrainz/musicbrainz-docker
docker-compose exec musicbrainz /recreatedb.sh -fetch
```

Assim, por forma a usar a máquina virtual em vez do servidor oficial do `MusicBrainz` e evitando assim a limitação do mesmo, basta aceder o URL `http://localhost:5000/ws/2/entidade/id` em vez do anteriormente referido, substituindo entidade por `area`, `artist`, `recording` ou `release-group` e id pelo id a verificar.

De seguida, para verificar se os ids estão corretos, bem como se todos os id's referenciados existem, foi criado uma *script* também em *Node.js* (`getIdsFromTTL.js`) que obtém dos ficheiros *Turtle* os ids de indivíduos, bem como, os ids referenciados. Depois, também com recurso a uma *script* criada (`compare.js`), obtém-se quais os ids referenciados em que o indivíduo não existe. Ao correr esta script, verificou-se que faltavam alguns artistas que eram referenciados e estão apenas presentes em `release.json`, só que sem possuírem toda a informação (não possuem *URLs* por exemplo). Como tal, foi criado um outro conversor (`remainArtistsFromRelease.js`), que a partir dos ids que faltam obtém a informação de cada artista (de forma semelhante ao já referido para o ficheiro `artist.json` mas, sem ter em conta *URLs*).

Por fim, concatenasse os vários ficheiros gerados e o ficheiro com a estrutura da ontologia, resultando num ficheiro Turtle final, passível de ser carregado no *GraphDB*.

Por forma a automatizar todo este processo de conversão (sem contar com a instalação da máquina virtual nem com a descompressão do dataset), foi criada a script *convert.sh*. É importante referir também que não houve uso de dados provenientes de series.json (conjuntos de releases-groups), de label.json (editoras de música, discografias, etc), de place.json (locais de produção de música), de event.json (eventos de música, festivais, etc) e de instrument.json (instrumentos).

5 Servidor API

O servidor com a API foi desenvolvido recorrendo ao *Node.js*, usando a *framework* de desenvolvimento *Express.js*. O servidor está em constante comunicação com duas bases de dados, *MongoDB* e *GraphDB*. No *GraphDB*, base de dados de semântica baseada em grafos, foi carregado a ontologia já povoada referida anteriormente (ver 4). Já no *MongoDB*, base de dados baseada em documentos, é onde se encontra a informação dos utilizadores, o seu nome, email, password, estatísticas (visualizações e classificações) e músicas favoritas. Para além da informação dos utilizadores, quando um utilizador apaga a sua conta, as visualizações e classificações são salvaguardados, sendo agregados pelo identificador da música. A API serve como ponto de ligação para os utilizadores/aplicações à informação. Esta API está protegida em grande parte das rotas por autenticação, algo que será abordado mais à frente (ver 5.4).

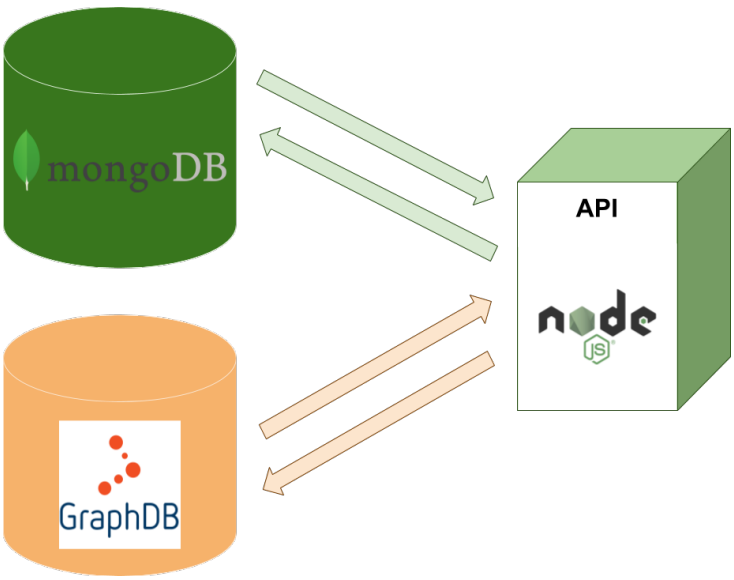

Figura 2: Estrutura do backend

Esta estrutura, demonstrada na figura acima (fig. 2), é modular, ou seja, cada um dos três componentes identificados pode estar num servidor/máquina/*container* diferente, o que permite escalar facilmente a API.

Nas próximas secções será aprofundado os vários constituintes e camadas da API.

5.1 Modelos

Nos `Models` define-se os *schemas* das coleções de documentos da base de dados *MongoDB*. Cada ficheiro `.js` corresponde a uma coleção, sendo que cada `schema` permite definir a estrutura de cada tipo de documento. Foram definidos os seguintes *Schemas*/Documentos:

User: representa um utilizador

- `name:` String obrigatória referente ao nome do utilizador;
- `email:` String obrigatória, única, ou seja sem repetição na base de dados, portanto pode servir de identificação, representado o email do utilizador;
- `password:` String obrigatória encriptada referente à password do utilizador;
- `stats:` lista com as estatísticas do utilizador;
  - `id:` id (String) do *Recording* a que se associa os valores seguintes;
  - `views:` número de visualizações do utilizador deste *Recording*;
  - `rating:` classificação dado pelo utilizador a este *Recording*.
- `favs:` lista de identificadores dos *Recordings* favoritos do utilizador

Stats: salvaguarda as estatísticas de antigos utilizadores (utilizadores que apagaram a sua conta)

- `id:` id (String) do *Recording* a que se associa os valores seguintes;
- `views:` número de visualizações do *Recording* salvaguardadas;
- `avgRating:` classificação média do *Recording* salvaguardada;
- `nRating:` número de utilizadores que classificaram o *Recording*.

5.2 Controladores

Os `controllers` possuem várias funções que permitem manusear os documentos e os seus valores. As coleções de documentos são estruturados por `Schemas` recorrendo ao `Mongoose`, sendo nos `Controllers` onde se encontram definidos a criação, atualização, destruição, listagem e obtenção de estatísticas dos mesmos. Apresentasse de seguida os *controllers* dos *models User* e *Stats* (quando se refere música assuma o mesmo que *Recording*):

User:

- `findOne`: Dado um email obtém a informação do utilizador com esse email associado;
- `getUser`: Dado um id obtém a informação do utilizador com esse id associado;
- `isValidPassword`: Dadas duas passwords, compara-as, verificando se são iguais;
- `createUser`: Dada a informação de um utilizador, encripta a password, criando, por fim, o utilizador;
- `updateUser`: Dada a informação a atualizar (*name* e/ou *email*) e o id do utilizador, atualiza a informação deste; esta função restringe o acesso, ou seja, apenas o próprio utilizador pode atualizar a sua informação;
- `updatePassword`: dado o id de um utilizador, a password atual e a nova password, verifica se a password atual está correta (se é igual à presente na base de dados), e em caso afirmativo, encripta a nova password e passa a ser a password atual do utilizador;
- `deleteUser`: dado um id de um utilizador, apaga-o;
- `getRecordingsUser`: dado um id de um utilizador, obtém uma lista com todas as estatísticas do utilizador (visualizações e classificações);
- `getMostRecordingsViewsUser`: dado um id de um utilizador, obtém uma lista das 10 músicas mais ouvidas/vistas pelo utilizador; cada elemento desta lista tem o id da música e o número de visualizações desta;
- `getMostRecordingsRatingUser`: dado um id de um utilizador, obtém uma lista das 10 músicas com maior classificação dada pelo utilizador; cada elemento desta lista tem o id da música e a classificação atribuída;
- `getRecordingUserRating`: dado um id de um utilizador e um id de uma música, devolve a classificação dada pelo utilizado à música; caso o utilizador não a tenha classificado, devolve 0;
- `updateViews`: dado um id de um utilizador e o id de uma música, soma mais uma visualização a essa música, caso as estatísticas para essa música já existam; caso não existam cria uma, com número de visualizações igual a 1 e classificação igual a 0;
- `updateRating`: dado um id de um utilizador, o id de uma música e a classificação a atribuir, caso as estatísticas para esta música já existam, atribui a nova classificação; caso não existam cria uma nova estatística para a música, com o número de visualizações igual a 0 e a classificação igual à que foi inserida.
- `getFavs`: dado um id de um utilizador, obtém a lista de favoritos do utilizador
- `addFav`: dado um id de um utilizador e o id da música, adiciona a música à lista de favoritos do utilizador
- `removeFav`: dado um id de um utilizador e o id da música, remove a música da lista de favoritos do utilizador
- `isFav`: dado um id de um utilizador e o id da música, verifica se a música está na lista de favoritos do utilizador

Stats:

- `createOrUpdate`: dado estatísticas de uma música (id, *views*, *rating*), agrega estes valores com os existentes, caso já exista o documento para o id desta música; caso ainda não exista é criado um;
- `getStats`: obtém todas as estatísticas, estejam elas presentes nos documentos *Stats* ou nos utilizadores, agregando as estatísticas pelos ids das músicas;
- `getMostRecordingsViews`: obtém as 10 músicas mais ouvidas;
- `getMostRecordingsRating`: obtém as 10 músicas com maior classificação;
- `getRecordingStats`: dado o id de uma música, obtém as estatísticas para esta música;
- Nota: Para obter as estatísticas, foi necessário, como já referido, aceder aos documentos dos utilizadores a partir deste `controller`.

Por outro lado, também é aqui nos `controllers` que são realizadas as *queries* à base de dados *GraphDB*. Por forma, a modularizar o código, no ficheiro *execQuery.js* é definida a função `execQuery`, onde apenas recebe a *query* a ser realizada, e devolve o resultado já "limpo", ou seja, uma lista de resultados, onde cada elemento, tem como atributos as variáveis devolvidas pela *query* e como valor o seu valor obtido. A função `execQuery` já inclui o prefixo da ontologia pelo que na *query* não é necessário indicar a totalidade do *URI* bastando apenas colocar ":" quando se refere a um *URI* da ontologia definida neste trabalho (ver 4). Contudo, antes de fazer qualquer pedido com esta função é necessário definir o `hostname:porta` através da função `set` (basta apenas definir uma vez). Esta função (`set`) foi criada por forma a facilitar o *deployment* do servidor bem como do *GraphDB*. Todas as *queries* realizadas são de leitura de informação, sendo estas divididas em ficheiros pela classe que questionam. Sendo assim, existem os seguintes `controllers` que questionam o *GraphDB* de forma a obter informação de indivíduos da classe:

Album:

- `listAlbums`: dado um *offset*, lista os 50 albums (id e título do album) a partir do *offset* da lista (ordenada (pelos títulos), foi removido ordenação por questões de performance) de todos os albums;
- `listAlbumsByFilter`: dado um *offset* e um filtro, devolve o mesmo que o anterior contudo em vez da lista (ordenada, foi removido ordenação por questões de performance) ser de todos os albums, é apenas dos albums que começam pelo valor presente no filtro;
- `getAlbum`: dado o id de um album, obtém o título, e caso exista, data da primeira *release*, sobre (*about*) e desambiguação;
- `getURLs`: dado o id de um album, obtém os *URLs* sobre o album, devolvendo uma lista onde cada elemento possui o nome do web site e o *URL*;
- `getTracks`: dado o id de um album, obtém as músicas pertencentes ao album, devolvendo uma lista, em que cada elemento tem o id, título, duração e, caso exista, desambiguação de uma música;
- `getTags`: dado o id de um album, obtém as tags do album, através das tags pertencentes às músicas pertencentes ao album;
- `getArtistsCredit`: dado o id de um album, obtém os artistas do album através dos artistas associados às músicas pertencentes ao album.

Area:

- `listAreas`: dado um *offset*, lista as 50 areas (id e nome da area) a partir do *offset* da lista (ordenada (pelos nomes), foi removido ordenação por questões de performance) de todas os areas;
- `listAreasByFilter`: dado um *offset* e um filtro, devolve o mesmo que o anterior contudo em vez da lista (ordenada, foi removido ordenação por questões de performance) ser de todas as areas, é apenas das areas que começam pelo valor presente no filtro;

- `getArea`: dado o id de uma area, obtém o nome, o tipo, e caso exista, sobre (*about*) e desambiguação;
- `getAliases`: dado o id de uma area obtém os nomes alternativos para a area;
- `getPartOf`: dado o id de uma area obtém o id, nome e tipo das areas a que pertence;
- `getParts`: dado o id de uma area obtém o id, nome e tipo das areas que contém;
- `getURLs`: dado o id de uma area, obtém os *URLs* sobre a area, devolvendo uma lista onde cada elemento possui o nome do web site e o *URL*;
- `getArtists`: dado o id de uma area, obtém os artistas que são dessa area, devolvendo para cada artista o seu id e nome.
- `countriesWithMostArtists`: obtém uma lista de todos os países, ordenados do país com mais artistas para o país com menos
- `countriesWithMostRecordings`: obtém uma lista de todos os países, ordenados do país com mais músicas para o país com menos
- `countriesWithMostAlbums`: obtém uma lista de todos os países, ordenados do país com mais albums para o país com menos

Artist:

- `listArtists`: dado um *offset*, lista os 50 artistas (id e nome do artista) a partir do *offset* da lista (ordenada (pelos nomes), foi removido ordenação por questões de performance) de todas os artistas;
- `listArtistsByFilter`: dado um *offset* e um filtro, devolve o mesmo que o anterior contudo em vez da lista (ordenada, foi removido ordenação por questões de performance) ser de todos os artistas, é apenas dos artistas que começam pelo valor presente no filtro;
- `getArtist`: dado o id de um artista, obtém o nome, e caso exista, o tipo, o nome para ordenar, a data de nascimento/início, a data de falecimento/fim, o sexo, o id da area, o nome da area, sobre (*about*) e desambiguação;
- `getAliases`: dado o id de um artista obtém os nomes alternativos para o artista;
- `getURLs`: dado o id de um artista, obtém os *URLs* sobre o artista, devolvendo uma lista onde cada elemento possui o nome do web site e o *URL*;
- `getRecordings`: dado o id de um artista, devolve o id e o título das músicas no qual o artista fez parte;
- `getTags`: dado o id de um artista, obtém as tags do artista, através das tags pertencentes às músicas no qual o artista fez parte;
- `getAlbums`: dado o id de um artista, obtém os albums do artista, ou seja, onde o artista possui músicas nas quais fez parte.

Recording:

- `listRecordings`: dado um *offset*, lista as 50 músicas (id e título da música) a partir do *offset* da lista (ordenada (pelos títulos), foi removido ordenação por questões de performance) de todas as músicas;
- `listRecordingsByFilter`: dado um *offset* e um filtro, devolve o mesmo que o anterior contudo em vez da lista (ordenada, foi removido ordenação por questões de performance) ser de todas as músicas, é apenas das músicas que começam pelo valor presente no filtro;
- `getRecording`: dado o id de uma música, obtém o título, e caso exista, a duração, sobre (*about*) e desambiguação;
- `getLanguages`: dado o id de uma música obtém as línguas usadas na música;

- `getTags`: dado o id de uma música, obtém as tags da música;
- `getURLs`: dado o id de uma música, obtém os *URLs* sobre a música, devolvendo uma lista onde cada elemento possui o nome do web site e o *URL*;
- `getArtistsCredit`: dado o id de uma música, devolve o id e o nome dos artistas que fizeram parte da música;
- `getAlbums`: dado o id de uma música, devolve os albums no qual a música faz parte;
- `searchForRecordings`: dados dois filtros, um para o nome de um artista e outro para um título de uma música, obtém o id, nome do artista e título de músicas em que o nome do artista comece pelo seu filtro e o título da música comece também pelo seu filtro.

5.3 Rotas

As rotas encaminham os pedidos dos clientes de forma a realizarem a correta chamada de funções presentes nos `Controllers`, sendo que a rota para onde o utilizador é encaminhado varia consoante o `URL` e o método do pedido. De seguida, apresentam-se as rotas, a função que é chamada para cada rota e é referido alguns passos adicionais que são realizados em alguns casos.

***Controller* User: rota `/users` concatenada com:**

- método `GET`
  - `/isAuthenticated` - devolve "Authenticated" se o utilizador estiver autenticado, caso contrário devolve "Unauthorized", permite saber se um utilizador está autenticado
  - `/:id/favs` - `getFavs(:id)` se o ":id" for igual ao id do utilizador que está autenticado, senão devolve uma mensagem de erro
  - `/:id/isFav/:idMusic` - `isFav(:id,:idMusic)` se o ":id" for igual ao id do utilizador que está autenticado, senão devolve uma mensagem de erro
  - `/:id/statsMostViews` - `getMostRecordingsViewsUser(:id)` se o ":id" for igual ao id do utilizador que está autenticado, senão devolve uma mensagem de erro
  - `/:id/statsMostRating` - `getMostRecordingsRatingUser(:id)` se o ":id" for igual ao id do utilizador que está autenticado, senão devolve uma mensagem de erro
  - `/:id/stats?idRec=idRecI` - `getRecordingUserRating(:id, idRecI)` se o ":id" for igual ao id do utilizador que está autenticado, senão devolve uma mensagem de erro
  - `/:id/stats` - `getRecordingsUser(:id)` se o ":id" for igual ao id do utilizador que está autenticado, senão devolve uma mensagem de erro
  - `/:id` - `getUser(:id)` se o ":id" for igual ao id do utilizador que está autenticado, senão devolve uma mensagem de erro
- método `POST`
  - `/login` - permite a autenticação de um utilizador, caso o utilizador exista e a password seja a correta, gerando um *token* que é enviado ao cliente. Esta parte será vista mais à frente (ver 5.4)
  - `/` - `createUser(body)`
- método `PUT`
  - `/views/:id` - `updateViews(:id, body.idMusic)` se o ":id" for igual ao id do utilizador que está autenticado, senão devolve uma mensagem de erro
  - `/rating/:id` - `updateRating(:id, body.idMusic, body.rating)` se o ":id" for igual ao id do utilizador que está autenticado, senão devolve uma mensagem de erro
  - `/updPass/:id` - `updatePassword(:id, body.prevPass, body.newPass)` se o ":id" for igual ao id do utilizador que está autenticado, senão devolve uma mensagem de erro

  - `/addFav/:id` - `addFav(:id, body.idMusic)` se o ":id" for igual ao id do utilizador que está autenticado, senão devolve uma mensagem de erro
  - `/removeFav/:id` - `removeFav(:id, body.idMusic, body.rating)` se o ":id" for igual ao id do utilizador que está autenticado, senão devolve uma mensagem de erro
  - `/:id` - `updateUser(:id, body)` se o ":id" for igual ao id do utilizador que está autenticado, senão devolve uma mensagem de erro
- método `DELETE`
  - `/:id` - `deleteUser(:id)` e para cada sub documento de estatísticas do utilizador apagado (`stat`) é a chamada a função `createOrUpdate(stat)` do `controller` Stats, se o ":id" for igual ao id do utilizador que está autenticado, senão devolve uma mensagem de erro

***Controller* Stats: rota `/stats` concatenada com:**

- método `GET`
  - `/mostViews` - `getMostRecordingsViews()`
  - `/mostRating` - `getMostRecordingsRating()`
  - `/:id` - `getRecordingStats(:id)`
  - `/` - `getStats()`
- método `POST`
  - `/` - `createOrUpdate(body)`

***Controller* Album: rota `/albums` concatenada com:**

- método `GET`
  - `/:id/tags` - `getTags(:id)`
  - `/:id/tracks` - `getTracks(:id)`
  - `/:id/artistsCredit` - `getArtistsCredit(:id)`
  - `/:id/urls` - `getURLs(:id)`
  - `/:id` - `getAlbum(:id)`
  - `?offset=offsetI&filter=filterI` - `listAlbumsByFilter(offsetI, filterI)`
  - `?filter=filterI` - `listAlbumsByFilter(0, filterI)`
  - `?offset=offsetI` - `listAlbums(offsetI)`
  - `/` - `listAlbums(0)`

***Controller* Area: rota `/areas` concatenada com:**

- método `GET`
  - `/countriesWithMostAlbums?cache=true` - o mesmo que `countriesWithMostAlbums()` mas a partir da cache, visto que o resultado é estático
  - `/countriesWithMostRecordings?cache=true` - o mesmo que `countriesWithMostRecordings()` mas a partir da cache, visto que o resultado é estático
  - `/countriesWithMostArtists?cache=true` - o mesmo que `countriesWithMostArtists()` mas a partir da cache, visto que o resultado é estático
  - `/countriesWithMostAlbums` - `countriesWithMostAlbums()`
  - `/countriesWithMostRecordings` - `countriesWithMostRecordings()`

- /countriesWithMostArtists - countriesWithMostArtists()
- /:id/aliases - getAliases(:id)
- /:id/artists - getArtists(:id)
- /:id/urls - getURLs(:id)
- /:id/parts - getParts(:id)
- /:id/partOf - getPartOf(:id)
- /:id - getArea(:id)
- ?offset=offsetI&filter=filterI - listAreasByFilter(offsetI, filterI)
- ?filter=filterI - listAreasByFilter(0, filterI)
- ?offset=offsetI - listAreas(offsetI)
- / - listAreas(0)

***Controller* Artist: rota /artists concatenada com:**

- método GET
  - /:id/tags - getTags(:id)
  - /:id/albums - getAlbums(:id)
  - /:id/recordings - getRecordings(:id)
  - /:id/urls - getURLs(:id)
  - /:id/aliases - getAliases(:id)
  - /:id - getArtist(:id)
  - ?offset=offsetI&filter=filterI - listArtistsByFilter(offsetI, filterI)
  - ?filter=filterI - listArtistsByFilter(0, filterI)
  - ?offset=offsetI - listArtists(offsetI)
  - / - listArtists(0)

***Controller* Recording: rota /recordings concatenada com:**

- método GET
  - /search?name=nameI&title=titleI - searchForRecordings(nameI, titleI)
  - /:id/albums - getAlbums(:id)
  - /:id/artistsCredit - getArtistsCredit(:id)
  - /:id/urls - getURLs(:id)
  - /:id/languages - getLanguages(:id)
  - /:id/tags - getTags(:id)
  - /:id - getRecording(:id)
  - ?offset=offsetI&filter=filterI - listRecordingsByFilter(offsetI, filterI)
  - ?filter=filterI - listRecordingsByFilter(0, filterI)
  - ?offset=offsetI - listRecordings(offsetI)
  - / - listRecordings(0)

5.4 Autenticação

As rotas da API estão quase todas protegidas à exceção de duas, `GET` de `/users/login` e `POST` de `/users`. A primeira rota não está protegida porque é onde os utilizadores podem realizar *login*, e que em caso de autenticação com sucesso é gerado um `token` que é enviado para o utilizador. Já a segunda rota permite a criação (registo) de utilizadores.

De forma a aceder às rotas protegidas, os clientes tem de enviar o `token`, que lhes foi fornecido, nas `headers` do pedido, mais precisamente na *header* `Authorization`, sendo o valor a colocar neste campo igual a `Bearer ${token}`.

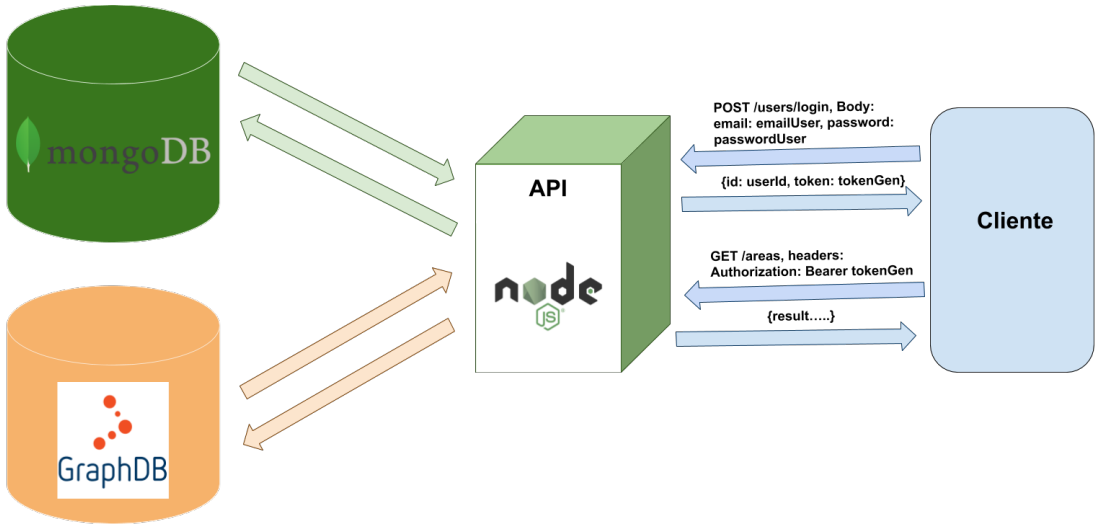

Figura 3: Um exemplo de um login e posterior pedido com envio do *token*

O *token* tem um tempo de validade, pelo que ao fim de 30 minutos expira, sendo necessária nova autenticação pelo utilizador.

A autenticação é realizada com a ajuda do `passport-local`. De forma a gerar o *token* recorre-se ao `jsonwebtoken` usando uma chave privada para gerar o *token* através do algoritmo RS256 (*RSA Signature with SHA-256*). Já na verificação do *token* é utilizado o `passport-jwt` com a tática de extração `fromAuthHeaderAsBearerToken`, daí a necessidade de colocar o *token* nas `headers` dos pedidos como indicado. Esta verificação recorre à chave pública para confirmar a validade do *token*. Por fim, é importante referir que no *deployment* deste servidor é de extrema importância, trocar a chave privada e a chave pública, visto este trabalho estar disponível publicamente.

6 Interface

A interface foi desenvolvida através da *framework* `Vue.js` e a *framework* de material `Vuetify.js` baseada no *Material Design* da *Google*. Esta interface estará no lado do utilizador, enquanto que a API está presente num servidor. Claro, contudo, esta interface tem de ser armazenada num servidor, de forma aos clientes/utilizadores fazerem o download desta interface, para posteriormente acederem diretamente à API através da interface.

Quanto a imagens usadas na interface, estas estão armazenadas em `public/static/`.

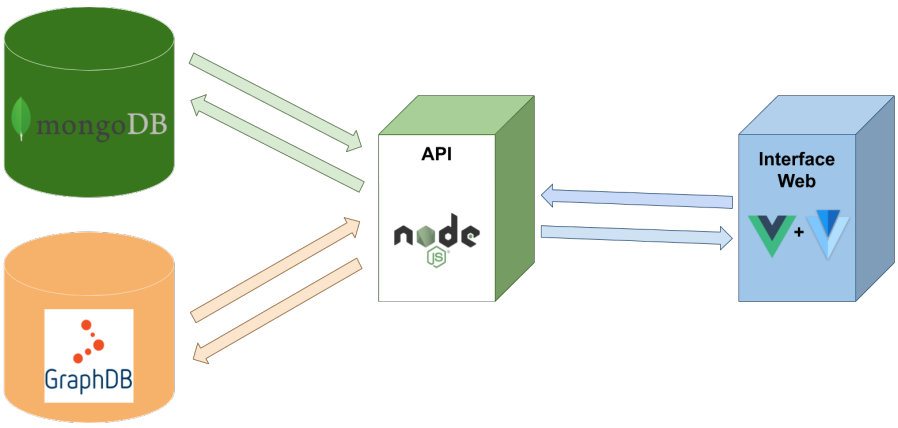

Figura 4: Estrutura do backend e frontend

Já em relação ao aspeto visual da interface, ela possui um *footer*, que aparece em todas as páginas, e como tal, foi definida na `App.vue`. Este *footer* possui alguns *URLs* importantes, desde APIs externas usadas, ao *URL* onde se pode verificar o código produzido.

Já na `main.js` é definida uma variável global (`$urlAPI`) onde está presente o *URL* da API, de forma a tornar o código mais limpo. O valor desta variável toma o valor da variável ambiente `VUE_APP_SERVER_URL` definida em `.env` por forma a uma maior facilidade de *deployment* da interface. É também aqui importado uma *package wrapper* da API do *iframe* do *Youtube*.

De seguida, serão apresentadas as várias camadas da interface.

6.1 Rotas

Visto que foi usada um versão do *Vue.js* com um *router* é necessário definir rotas, indicando para que vista as mesmas (rotas) redirecionam. Apresenta-se, de seguida, as rotas definidas (presentes no ficheiro *router.js*):

- Rota `/recordingsSearch` → vista `RecordingsSearch.vue`
- Rota `/recordings/:id` → vista `Recording.vue`
- Rota `/areas/:id` → vista `Area.vue`
- Rota `/artists/:id` → vista `Artist.vue`
- Rota `/albums/:id` → vista `Album.vue`
- Rota `/userSettings/:id` → vista `UserSettings.vue`
- Rota `/recordings` → vista `Recordings.vue`
- Rota `/areas` → vista `Areas.vue`
- Rota `/artists` → vista `Artists.vue`
- Rota `/albums` → vista `Albums.vue`
- Rota `/favs` → vista `Favs.vue`
- Rota `/index` → vista `Index.vue`
- Rota `/signup` → vista `SignUp.vue`
- Rota `/` → vista `Login.vue`

6.2 Vistas

As vistas definidas são bastante parecidas umas com as outras, tendo apenas algumas pequenas diferenças.

Todas as vistas importam o componente com o mesmo nome que a vista, ou seja, por exemplo a vista `Areas.vue` importa o componente `Areas.vue`.

As vistas `Index.vue`, `Recordings.vue`, `Areas.vue`, `Artists.vue`, `Albums.vue`, `RecordingsSearch.vue`, `Recording.vue`, `Area.vue`, `Artist.vue`, `Album.vue`, `UserSettings.vue` e `Favs.vue` verificam se o utilizador está autenticado, em caso afirmativo é feito a renderização da página; em caso negativo o utilizador é redirecionado para a página de *login* (rota `/`).

Em adição, as vistas `Recording.vue`, `Area.vue`, `Artist.vue` e `Album.vue`, passam um argumento para o componente. Este argumento é o id do individuo (*Recording*, *Area*, *Artist* ou *Album* respetivamente). A vista `UserSettings.vue` também passa um argumento para o componente, contudo este argumento é o id do utilizador.

Por fim, as vistas `Login.vue` e `SignUp.vue` verificam se o utilizador está autenticado, em caso afirmativo, o utilizador é redirecionado para a página inicial (vista `Index.vue`, rota `/index`); em caso negativo a página é renderizada.

6.3 Componentes

Os componentes permitem a modularização das páginas web, fomentando uma melhor codificação.

Visto existirem várias funções que são usadas em vários componentes, foram criados os seguintes ficheiros:

`auth.js`:

Possui duas funções, `isAuthenticated` e `logout`. A primeira, pergunta à API se o utilizador está autenticado, caso não esteja, apaga da *local storage* o *token* e o id do utilizador. Já o `logout` apaga da *local storage* o *token* e o id do utilizador, sem verificar se ainda está autenticado. Ou seja, como foi dado a entender, durante a sessão de um utilizador, é guardado o seu id bem como o seu *token* na *local storage*.

`request.js`:

Este ficheiro foi criado de forma a possuir funções que simplifiquem o uso da *package* `axios`, ou seja, a realização de pedidos *web* com a colocação em cabeçalho do *token* do utilizador. Neste ficheiro está presente três funções, *getAPI*, *putAPI* e *deleteAPI*. A função *getAPI* recebe como argumento um *url* para onde é feito um pedido `GET`. Já a função *putAPI* para além de um *url*, recebe também dados a enviar como argumento, realizando um pedido `PUT` desses dados para o *url* fornecido. Por fim, o *deleteAPI* recebe como argumento um *url* para onde é feito um pedido `DELETE`. Como já referido estas três funções colocam nas *headers* o *token* do utilizador da forma já descrita na secção 5.4.

Antes de apresentar os componentes é importante referir que os componentes `Album.vue`, `Area.vue`, `Artist.vue` e `Recording.vue`, que apresentam um campo chamado *about*, realizam o processamento deste campo, visto o mesmo possuir uma sintaxe própria, passível de ser representada em `HTML`. [1]. Esta sintaxe, foi convertida da seguinte forma:

- `''pal''` (itálico) → `<i>pal</i>`
- `'''pal'''` (bold) → `<b>pal</b>`
- `----` (linha horizontal) → `<hr>`
- `=title=` (título 1) → `<h4>title</h4>`
- `==title==` (título 2) → `<h5>title</h5>`
- `===title===` (título 3) → `<h6>title</h6>`
- `[url]` (url) → `<a href=url>url</a>`
- `[url|desc]` (url) → `<a href=url>desc</a>`

De seguida apresentam-se os componentes criados.

`Login.vue`:

Página onde se realiza o login, possui um formulário para preencher o email e a password. Possui ainda dois botões, um para autenticar e outro para ser redirecionado para a página de registo.

`SignUp.vue`:

Página onde os utilizadores podem se registar. Possui um formulário, onde o utilizador necessita de indicar nome, email, password, repetir a password e depois de tudo ser validado (de forma a validar foi usado `vuelidate`) o botão para submeter o registo fica disponível. Quando o utilizador carregar para se registar, é redirecionado para a página de login, sendo mostrado uma mensagem de sucesso se o registo for efetuado com sucesso, ou pelo contrário se não tiver sucesso, aparece uma mensagem de erro. Possui também um botão *Back to Login*, para voltar à página de login.

`Toolbar.vue`:

Componente usado por quase todos os componentes, com a exceção dos componentes `Login.vue` e `SignUp.vue`. Este componente representa a barra de ferramentas superior que se encontra em grande parte das páginas. Nesta barra está presente vários links para diferentes páginas da interface, o primeiro onde está o nome da aplicação e o seu símbolo, redireciona para a página inicial (`/index`). Depois por ordem, links para *Recordings* favoritos do utilizador, listagem de *Albums*, *Recordings*, *Artists* e *Areas*, para a página de configuração do utilizador e por fim *Logout*. Quando o utilizador está numa destas páginas, a barra de ferramentas sinaliza a mesma ao usar uma cor diferente para o botão associada a essa página.

Para além disso, esta barra, por vezes, apresenta uma caixa de texto por forma a filtrar os resultados disponíveis ou para pesquisar. Quando se está nas páginas de listagem de *Albums*, *Recordings*, *Artists* e *Areas* esta caixa de texto permite filtrar os resultados. Nas restantes páginas em que aparece (página de um individuo *Album*, *Recording*, *Artist* ou *Area*, na página inicial *Index* e na página de favoritos do utilizador *Favs*), esta caixa de texto permite pesquisar por artista mais música com o formato "*Artist-Recording*".

`Index.vue`:

Página inicial da interface. Apresenta estatísticas do utilizador e da aplicação, como as músicas mais ouvidas e com melhor classificação, entre outras (ver 3). Ao carregar numa das músicas, albums, artistas ou areas, o utilizador é redirecionado para a página desse individuo.

`Albums.vue`:

Lista todos os albums, mas por partes de 50 albums, devido à grande quantidade de albums. Ao fim dos 50 albums, apresenta-se um botão, de forma a obter os próximos 50 albums. Todos os albums renderizados, são clicáveis de forma a obter mais informação (redirecionamento para a página do album).

`Areas.vue`:

Lista todos as areas, mas por partes de 50 areas, devido à grande quantidade de areas. Ao fim das 50 areas, apresenta-se um botão, de forma a obter as próximas 50 areas. Todas as areas renderizadas, são clicáveis de forma a obter mais informação (redirecionamento para a página da area).

`Artists.vue`:

Lista todos os artistas, mas por partes de 50 artistas, devido à grande quantidade de artistas. Ao fim dos 50 artistas, apresenta-se um botão, de forma a obter os próximos 50 artistas. Todos os artistas renderizados, são clicáveis de forma a obter mais informação (redirecionamento para a página do artista).

`Recordings.vue`:

Lista todos as músicas, mas por partes de 50 músicas, devido à grande quantidade de músicas. Ao fim das 50 músicas, apresenta-se um botão, de forma a obter as próximas 50 músicas. Todas as músicas renderizadas, são clicáveis de forma a obter mais informação (redirecionamento para a página da música).

`RecordingsSearch.vue`:

Página para onde o utilizador é redirecionado quando pesquisa na caixa de texto com o formato "*Artist-Recording*". Lista as músicas que obedecem à pesquisa, sendo possível obter mais informação da música ao clicar nela (o utilizador é redirecionado para a página da música).

`Album.vue`:

Página que apresenta toda a informação de um determinado individuo da classe *Album*. Apresenta também a classificação e a classificação do album. (ver 3)

`Area.vue`:

Página que apresenta toda a informação de um determinado individuo da classe *Area*. (ver 3)

`Artist.vue`:

Página que apresenta toda a informação de um determinado individuo da classe *Artist*. Apresenta também a classificação e a classificação do artista, bem como as 10 músicas mais ouvidas e mais bem classificadas do artista. (ver 3)

`Recording.vue`:

Página que apresenta toda a informação de um determinado individuo da classe *Recording*. Apresenta também o número de visualizações e a classificação da música. Para além disso mostra o vídeo da música (quando possível, nem sempre apresenta o vídeo correto) e as *lyrics* da música. As *lyrics* da música são obtidas a partir de duas `APIs` diferentes, *musixmatch* e *Chartlyrics*. A primeira API, apenas apresenta 30% da música, visto estar a ser utilizado uma versão gratuita. Futuramente é necessário alterar o *token* usado para aceder a esta API (*musixmatch*) seja por questões de segurança ou para usar uma versão paga que apresenta toda a letra da música. Por fim, apresenta também um botão por forma ao utilizador adicionar/remover a música aos/dos favoritos.

`UserSettings.vue`:

Nesta página, estão presentes dois formulários onde o utilizador pode, num deles atualizar a sua password e no outro atualizar a sua informação (nome e email). Os campos são igualmente verificados com `vuelidate`.

`Favs.vue`:

Página que apresenta a listagem de todas as músicas favoritas do utilizador. É possível, através de dois botões, remover uma música dos favoritos ou aceder à página da música.

`Youtube.vue`:

Componente usado pela página onde se apresenta um individuo da classe *Recording*. Permite pesquisar vídeos do *YouTube* por uma *string* e mostrar o vídeo que possui o título que mais se assemelha a essa *string*.

7 Instalação

Antes de explicar a instalação, é importante referir que ao correr o servidor API, o mesmo, aceita dois parâmetros, `hostname + porta` do `MongoDB` e do `GraphDB`. Permite que na altura de instalação se possa agilizar mais facilmente o escalonamento da aplicação. Também nesse sentido, na interface existe um ficheiro `.env`, que possui a variável de ambiente `VUE_APP_SERVER_URL`, por forma a indicar à interface qual é o `hostname + porta` do servidor API.

Todo o código produzido pode ser descarregado a partir do repositório disponível no `GitHub` em `https://github.com/jcm300/Musike`.

Em relação à instalação propriamente dita, pode ser realizada de várias formas, seja criando a ontologia de raiz (envolvendo conversão do *dataset JSON*) com posterior instalação da aplicação; seja usar a ontologia já criada com posterior instalação da aplicação; ou, por fim, através da utilização de *containers* (*Docker*).

7.1 Instalação com criação da ontologia com posterior instalação da aplicação

Podemos dividir esta instalação em duas partes, criação da ontologia e instalação da aplicação.

Criação da ontologia:

Tem como dependências `git` e `Node.js`.

```
# clonar repositório por ssh
git clone git@github.com:jcm300/Musike.git
# ou por https
git clone https://github.com/jcm300/Musike.git

# entrar no diretório
cd Musike

# descomprimir dataset em JSON
cd datasets/JSON
cat artist_aa artist_ab artist_ac artist_ad artist_ae artist_af artist_ag \
    artist_ah artist_ai > artist.tar.xz
cat release-group_aa release-group_ab release-group_ac > release-group.tar.xz
cat release_aa release_ab release_ac release_ad release_ae release_af release_ag \
    release_ah release_ai release_aj release_ak release_al release_am release_an \
    release_ao release_ap release_aq release_ar release_as release_at release_au \
    release_av release_aw release_ax release_ay release_az release_ba release_bb \
    release_bc release_bd release_be release_bf release_bg release_bh release_bi \
    release_bj release_bk release_bl release_bm release_bn release_bo release_bp \
    release_bq release_br release_bs release_bt release_bu release_bv release_bw \
    release_bx release_by release_bz release_ca release_cb release_cc release_cd \
    release_ce release_cf release_cg release_ch release_ci release_cj release_ck \
    release_cl release_cm release-group_aa release-group_ab \
    release-group_ac > release.tar.xz
cat work_aa work_ab work_ac work_ad > work.tar.xz
tar xf area.tar.xz
tar xf artist.tar.xz
tar xf recording.tar.xz
tar xf release.tar.xz #CUIDADO: este ocupa 230GB
tar xf release-group.tar.xz
tar xf work.tar.xz
```

```
# preparar ficheiros para converter
mv mbdump/area area.json
mv mbdump/artist artist.json
mv mbdump/recording recording.json
mv mbdump/release release.json
mv mbdump/release-group release-group.json
mv mbdump/work work.json

# converter ficheiros json, criando finalmente a ontologia
cd ../../jsonTOturtle
./convert.sh

# a ontologia criada estará em:
cd ../datasets/Turtle
# com o nome igual a MusicBrainz.ttl
```

Instalação da aplicação:

Por forma a instalar a aplicação é necessário instalar, `Java 8` (dependência do `GraphDB`), `GraphDB`, `MongoDB` e `Node.js`.

```
# criar uma pasta graph-import na home do utilizador
mkdir ~/graph-import

# mover ontologia para a pasta
mv MusicBrainz.ttl ~/graph-import

# Correr GraphDB e no GraphDB Workbench criar um repositório com id igual
# a "musicbrainz" e importar a ontologia para esse repositório

cd ../../src # mover para a pasta Musike/src

# alterar forma como é iniciado o GraphDB na script install.sh, caso seja necessário,
# visto que assume que o GraphDB está instalado em Musike/src/graphdb-free-8.9.0

# instalar dependências do node.js e iniciar GraphDB, MongoDB, servidor API e interface
./install.sh i

# depois de correr uma primeira vez o comando anterior, sempre que se pretender iniciar,
# ou seja, iniciar GraphDB, MongoDB, servidor API e interface basta correr
./install.sh r
```

A interface será acessível a partir de `http://localhost:8080/`, enquanto que a `API` será acessível a partir de `http://localhost:3000/`, o `MongoDB` a partir de `http://localhost:27017/` e o *GraphDB* a partir de `http://localhost:7200/`.

7.2 Instalação usando ontologia já criada com posterior instalação da aplicação

Esta instalação também pode ser dividida em duas partes, importação da ontologia e instalação da aplicação. Esta instalação pode incluir uma terceira parte opcional, a criação de *containers*.

Importação da ontologia:

Em primeiro lugar, realiza-se a clonagem do repositório:

```
# clonar repositório por ssh
git clone git@github.com:jcm300/Musike.git
# ou por https
git clone https://github.com/jcm300/Musike.git

cd Musike
```

De seguida, instalasse `Java 8` (dependência do `GraphDB`).

O próximo passo pode ser realizado de duas formas, importação da ontologia ou usar o `GraphDB` incluído no repositório que já inclui a importação da ontologia. No primeiro método segue-se os passos iniciais da instalação anterior ou seja:

```
#Instalar GraphDB

# descomprimir ontologia já criada
cd datasets/Turtle
cat MusicBrainz.ttl.gz_aa MusicBrainz.ttl.gz_ab MusicBrainz.ttl.gz_ac \
    MusicBrainz.ttl.gz_ad MusicBrainz.ttl.gz_ae MusicBrainz.ttl.gz_af \
    MusicBrainz.ttl.gz_ag MusicBrainz.ttl.gz_ah MusicBrainz.ttl.gz_ai \
    MusicBrainz.ttl.gz_aj MusicBrainz.ttl.gz_ak MusicBrainz.ttl.gz_al \
    MusicBrainz.ttl.gz_am MusicBrainz.ttl.gz_an MusicBrainz.ttl.gz_ao \
    MusicBrainz.ttl.gz_ap MusicBrainz.ttl.gz_aq MusicBrainz.ttl.gz_ar \
    MusicBrainz.ttl.gz_as MusicBrainz.ttl.gz_at MusicBrainz.ttl.gz_au \
    MusicBrainz.ttl.gz_av MusicBrainz.ttl.gz_aw MusicBrainz.ttl.gz_ax \
    MusicBrainz.ttl.gz_ay MusicBrainz.ttl.gz_az MusicBrainz.ttl.gz_ba \
    MusicBrainz.ttl.gz_bb MusicBrainz.ttl.gz_bc MusicBrainz.ttl.gz_bd \
    MusicBrainz.ttl.gz_be MusicBrainz.ttl.gz_bf MusicBrainz.ttl.gz_bg \
    MusicBrainz.ttl.gz_bh MusicBrainz.ttl.gz_bi MusicBrainz.ttl.gz_bj \
    MusicBrainz.ttl.gz_bk MusicBrainz.ttl.gz_bl MusicBrainz.ttl.gz_bm \
    MusicBrainz.ttl.gz_bn > MusicBrainz.ttl.gz
gunzip -k MusicBrainz.ttl.gz

# criar uma pasta graph-import na home do utilizador
mkdir ~/graph-import

# mover ontologia para a pasta
mv MusicBrainz.ttl ~/graph-import

# Correr GraphDB e no GraphDB Workbench criar um repositório com id igual
# a "musicbrainz" e importar a ontologia para esse repositório
```

```
cd ../../src # mover para a pasta Musike/src

# alterar forma como é iniciado o GraphDB na script install.sh, caso seja necessário,
# visto que assume que o GraphDB está instalado em Musike/src/graphdb-free-8.9.0
```

O segundo método é usar o `GraphDB` incluído no repositório no qual, já inclui a ontologia importada bastando seguir os seguintes passos:

```
# descomprimir GraphDB
cd src/graphdb-free-8.9.0
cat graph.tar.gz_aa graph.tar.gz_ab graph.tar.gz_ac graph.tar.gz_ad graph.tar.gz_ae \
    graph.tar.gz_af graph.tar.gz_ag graph.tar.gz_ah graph.tar.gz_ai graph.tar.gz_aj \
    graph.tar.gz_ak graph.tar.gz_al graph.tar.gz_am graph.tar.gz_an graph.tar.gz_ao \
    graph.tar.gz_ap graph.tar.gz_aq graph.tar.gz_ar graph.tar.gz_as graph.tar.gz_at \
    graph.tar.gz_au graph.tar.gz_av graph.tar.gz_aw graph.tar.gz_ax graph.tar.gz_ay \
    graph.tar.gz_az graph.tar.gz_ba graph.tar.gz_bb graph.tar.gz_bc graph.tar.gz_bd \
    graph.tar.gz_be graph.tar.gz_bf graph.tar.gz_bg graph.tar.gz_bh graph.tar.gz_bi \
    graph.tar.gz_bj graph.tar.gz_bk graph.tar.gz_bl graph.tar.gz_bm graph.tar.gz_bn \
    graph.tar.gz_bo graph.tar.gz_bp graph.tar.gz_bq graph.tar.gz_br graph.tar.gz_bs \
    graph.tar.gz_bt graph.tar.gz_bu graph.tar.gz_bv graph.tar.gz_bw graph.tar.gz_bx \
    graph.tar.gz_by graph.tar.gz_bz graph.tar.gz_ca graph.tar.gz_cb graph.tar.gz_cc \
    graph.tar.gz_cd graph.tar.gz_ce graph.tar.gz_cf > graph.tar.gz
tar -xf graph.tar.gz

cd .. # mover para a pasta Musike/src
```

Instalação da aplicação:

Por forma a instalar a aplicação é necessário instalar, `MongoDB` e `Node.js`.

```
# instalar dependências do node.js e iniciar GraphDB, MongoDB, servidor API e interface
./install.sh i

# depois de correr uma primeira vez o comando anterior, sempre que se pretender iniciar,
# ou seja, iniciar GraphDB, MongoDB, servidor API e interface basta correr
./install.sh r
```

A interface será acessível a partir de `http://localhost:8080/`, enquanto que a `API` será acessível a partir de `http://localhost:3000/`, o `MongoDB` a partir de `http://localhost:27017/` e o *GraphDB* a partir de `http://localhost:7200/`.

Criação dos *containers*:

Uma terceira fase opcional é a criação de *containers*. Usando o segundo método da importação da ontologia, ou seja usar o `GraphDB` já incluído no repositório, é possível criar os *containers* usados na instalação através de *containers* e, fazer *push* dos mesmos (*containers*) posteriormente, executando os seguintes comandos:

```
# clonar repositório por ssh
git clone git@github.com:jcm300/Musike.git
# ou por https
git clone https://github.com/jcm300/Musike.git

cd Musike

# descomprimir GraphDB
cd src/graphdb-free-8.9.0
cat graph.tar.gz_aa graph.tar.gz_ab graph.tar.gz_ac graph.tar.gz_ad graph.tar.gz_ae \
    graph.tar.gz_af graph.tar.gz_ag graph.tar.gz_ah graph.tar.gz_ai graph.tar.gz_aj \
    graph.tar.gz_ak graph.tar.gz_al graph.tar.gz_am graph.tar.gz_an graph.tar.gz_ao \
    graph.tar.gz_ap graph.tar.gz_aq graph.tar.gz_ar graph.tar.gz_as graph.tar.gz_at \
    graph.tar.gz_au graph.tar.gz_av graph.tar.gz_aw graph.tar.gz_ax graph.tar.gz_ay \
    graph.tar.gz_az graph.tar.gz_ba graph.tar.gz_bb graph.tar.gz_bc graph.tar.gz_bd \
    graph.tar.gz_be graph.tar.gz_bf graph.tar.gz_bg graph.tar.gz_bh graph.tar.gz_bi \
    graph.tar.gz_bj graph.tar.gz_bk graph.tar.gz_bl graph.tar.gz_bm graph.tar.gz_bn \
    graph.tar.gz_bo graph.tar.gz_bp graph.tar.gz_bq graph.tar.gz_br graph.tar.gz_bs \
    graph.tar.gz_bt graph.tar.gz_bu graph.tar.gz_bv graph.tar.gz_bw graph.tar.gz_bx \
    graph.tar.gz_by graph.tar.gz_bz graph.tar.gz_ca graph.tar.gz_cb graph.tar.gz_cc \
    graph.tar.gz_cd graph.tar.gz_ce graph.tar.gz_cf > graph.tar.gz
tar -xf graph.tar.gz

cd .. # mover para a pasta Musike/src

# criar containers, pode ser necessário sudo
docker-compose -f docker-compose.yml build

# push dos containers, pode ser necessário sudo
docker push jcm300/mb_graphdb:tag
docker push jcm300/musike_api:tag
docker push jcm300/musike_interface:tag
```

7.3 Instalação através de containers (Docker)

A instalação através de *containers* é a menos trabalhosa e mais simples contudo, normalmente possui pior performance em relação a uma instalação sobre um sistema operativo. Logo é deixado à escolha do utilizador/administrador o método que pretende usar para instalar.

Portanto, para instalar tudo através de *containers*, o primeiro passo é instalar o `docker` e o `docker-compose`. De seguida, os passos a realizar são os seguintes:

```
# obter o ficheiro docker-compose para proceder à criação dos containers
wget https://raw.githubusercontent.com/jcm300/Musike/master/src/docker-compose_without_build.yml
# ou
curl https://raw.githubusercontent.com/jcm300/Musike/master/src/docker-compose_without_build.yml \
    > docker-compose_without_build.yml

# criar e iniciar os containers, pode ser necessário sudo
docker-compose -f docker-compose_without_build.yml up
```

Este procedimento irá ocupar, no fim, cerca de 20 GB de espaço de armazenamento, estando a interface acessível a partir de `http://localhost:8080/`, enquanto que a `API` é acessível a partir de `http://localhost:3000/`, o `MongoDB` a partir de `http://localhost:27017/` e o *GraphDB* a partir de `http://localhost:7200/`.

8 Conclusões e Trabalho Futuro

Em forma de conclusão, após todo o trabalho realizado, a partir do dataset em *JSON* do `MusicBrainz`, com um tamanho de cerca de 240GB, converteu-se esse dataset numa ontologia com cerca de 9GB de tamanho. Para além disso, apresenta-se também uma aplicação funcional que utiliza a informação da ontologia, permitindo pesquisar a mesma (informação), para além de permitir a visualização adicional de estatísticas e dos vídeos e letras das músicas.

Contudo, o trabalho carece de algumas melhorias, desde evitar a repetição de uma ou duas propriedades para certos indivíduos na ontologia (algo complicado devido à estrutura do dataset *JSON* do `MusicBrainz`, este *dump* carece de melhorias por parte do `MusicBrainz`). Para além disso, em termos de *API*, as *queries* de pesquisa por grandes quantidades bem como, a listagem de grandes quantidades (quando com ordenação), carecem de algumas melhorias de performance. Já na interface ficou por acrescentar, a criação e gestão de *playlists* por parte de utilizador, bem como, de forma adicional a sugestão de músicas ao utilizador a partir das estatísticas desse mesmo utilizador. Por fim, ainda na interface, em termos de apresentação de conteúdo, seria útil que as listagens de indivíduos, fossem também possíveis de serem visualizadas por paginação.

Referências

[1] *Annotation*. Access in 29/03/2019. URL: https://musicbrainz.org/doc/Annotation.

[2] *Area*. Access in 29/03/2019. URL: https://musicbrainz.org/doc/Area.

[3] *Artist*. Access in 29/03/2019. URL: https://musicbrainz.org/doc/Artist.

[4] *How to use database-dumps with the virtual machine image*. Access in 25/04/2019. URL: https://community.metabrainz.org/t/how-to-use-database-dumps-with-the-virtual-machine-image/390132.

[5] *MusicBrainz Database/Schema*. Access in 29/03/2019. URL: https://musicbrainz.org/doc/MusicBrainz_Database/Schema.

[6] *MusicBrainz Server/Setup*. Access in 25/04/2019. URL: https://musicbrainz.org/doc/MusicBrainz_Server/Setup.

[7] *Recording*. Access in 29/03/2019. URL: https://musicbrainz.org/doc/Recording.

[8] *Relationships*. Access in 29/03/2019. URL: https://musicbrainz.org/relationships.

[9] *Release*. Access in 29/03/2019. URL: https://musicbrainz.org/doc/Release.

[10] *Release Group*. Access in 29/03/2019. URL: https://musicbrainz.org/doc/Release_Group.

[11] *Work*. Access in 29/03/2019. URL: https://musicbrainz.org/doc/Work.

[12] *XML Web Service/Rate Limiting*. Access in 25/04/2019. URL: https://wiki.musicbrainz.org/XML_Web_Service/Rate_Limiting.