Universidade do Minho

Mestrado em Engenharia Informática

Processamento e Representação de Conhecimento

Musike - Music Linked

José Carlos Lima Martins
A78821

1 de Junho de 2019

1 Introdução

ENTREGA:
27 de Maio

2 Contextualização

O trabalho a desenvolver passa por escolher uma área/tema de trabalho. A partir desse tema, criar/construir/limpar/extrair um dataset sobre o tema e que esteja disponível na LOD (*Linked Open Data*). Este dataset em Turtle (*Terse RDF Triple Language*) será armazenado em *GraphDB*, onde será possível aceder/realizar *queries* ao dataset localmente.

De seguida, será criado um website/app web de forma a explorar o dataset anteriormente criado. Este website deve possuir autenticação, usando para isso o *MongoDB*.

3 Tema escolhido - Musike

O **Musike**, abreviatura de *Music Linked*, consiste num website 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 website (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 website (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 website
  - número de vezes ouvida pelos utilizadores do website

- 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 website deve permitir aos utilizadores escolher as músicas e albuns que mais gosta, bem como, puder criar playlists.

Cada utilizador deve ter acesso às suas estatísticas, onde deve estar presente as músicas que o utilizador ouve mais, os artistas que ouve mais e os albuns que ouve mais.

Por fim, apresentar estatísticas gerais do website entre as quais:

- músicas mais ouvidas pelos utilizadores
- artistas mais ouvidos pelos utilizadores
- tipos de músicas (tags) mais ouvidos pelos utilizadores

- albuns mais ouvidos pelos utilizadores
- países com mais artistas
- países com mais músicas
- países com mais albuns
- países mais "ouvidos" 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*.

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

4.1 Parte Estrutural da Ontologia

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

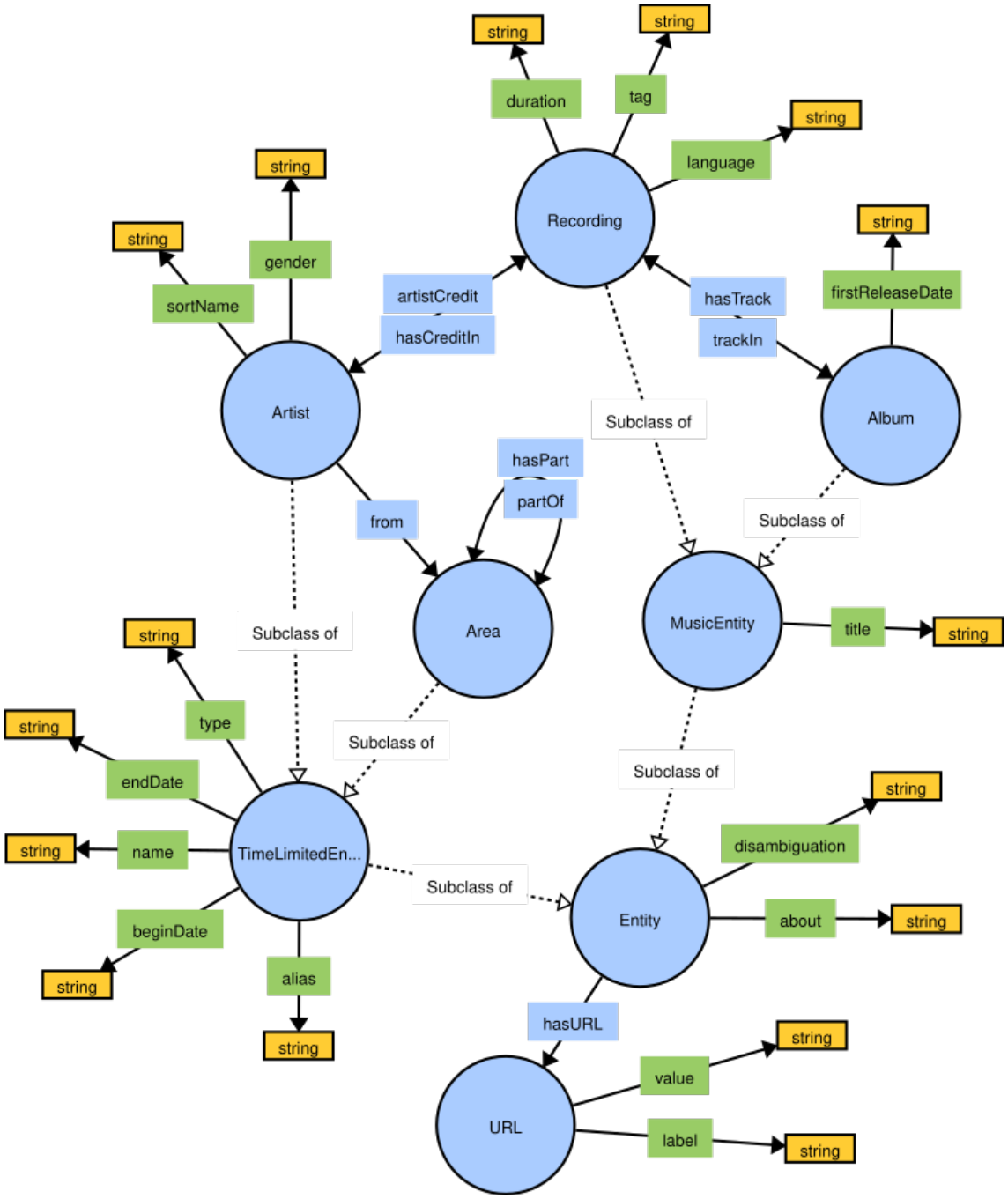

Figura 1: Estrutura da ontologia, imagem obtida através de WebVOWL

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*. *MusicEntity* inclui as subclasses *Recording* e *Album* devido às duas possuírem a propriedade *title*. Por outro lado, *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 website 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, contry, 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 da música
    - ***tag***: Tags (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

Com a estrutura da ontologia e com uma ideia da informação necessária, foi então usado um dataset (`http://ftp.musicbrainz.org/pub/musicbrainz/data/json-dumps/20190403-001001/`) 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 (que em cada linha possui um elemento de tipo igual ao nome do ficheiro a percorrer) com os conversores criados em *Node.js*. Foi criado um conversor por ficheiro a converter. Apresentam se os seguintes conversores, que para cada linha do ficheiro converte a informação para (origem → destino) [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***
- 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***

- 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***
  - *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***
    - *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***
- 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***
- 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 destino do recording → *data property* ***language*** (por cada elemento para o id a recording possui *language* e *languages*)

Ainda nestes conversores, os ids são verificados, visto que, acontece haver "merges" de entidades no **MusicBrainz** e, quando isso acontece passa a existir uma só entidade das duas mas, é possível aceder à mesma através dos dois ids de cada entidade antes do "merge". De tal forma, para obter uma ontologia consistente, os ids quando aparecem nos conversores é acedido o URL `https://musicbrainz.org/ws/2/entidade/id` usando o *axios* e do campo `data` da resposta verificasse o campo `id`. Caso consiga-se 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]. Contudo, como o dataset que vinha com o servidor da máquina virtual era anterior ao dataset em que o trabalho se baseia, foi atualizado o mesmo 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.

Por forma a automatizar os conversores, estes podem ser chamados ao correr a script *convert.sh* que chama cada conversor, redirecionando o *output* para um ficheiro *Turtle* (*.ttl*) por conversor. É importante referir também que não houve uso de dados provenientes de series.json (conjuntos de releases-groups), label.json (editoras de música, discografias, etc), place.json (locais de produção de música), event.json (eventos de música, festivais, etc) e instrument.json (instrumentos).

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

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 e estatísticas (visualizações e ratings). Para além da informação dos utilizadores, quando um utilizador apaga a sua conta, as visualizações e ratings são salvaguardados, sendo agregados pelo identificador da música. A API serve como ponto de ligação para os utilizadores/aplicações à informação. A API está protegida em grande parte das rotas por autenticação, algo que será abordado mais à frente (ver 5.4).

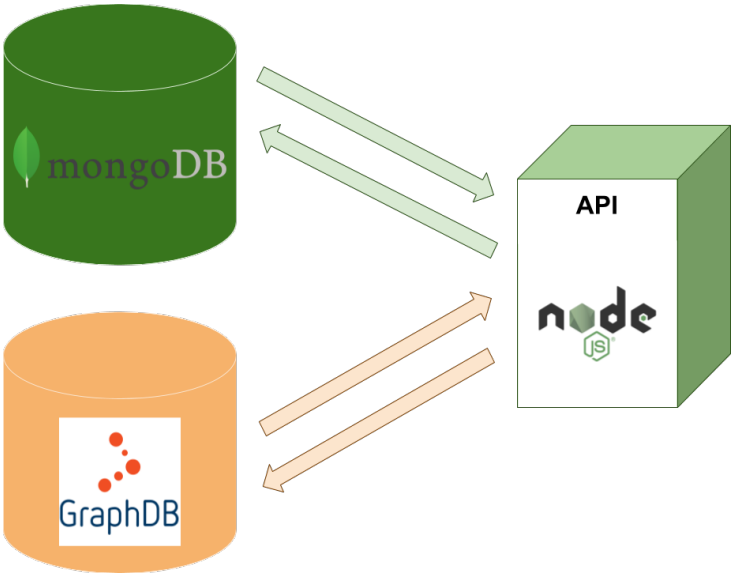

Figura 2: Estrutura do backend

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

Stats: salvaguarda as estatísticas que 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 estatisticas do utilizador (visualizações e classificações);
- `getMoreRecordingsViewsUser`: 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;
- `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 uma música, caso as estatísticas para essa música já existam; caso não exista 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 estatítiscas para esta música ja existam, atribui a nova classificação; caso não exista cria uma nova estatística para a música, com o número de visualizações igual a 0 e a classificação igual ao que foi inserida.

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;
- `getMoreRecordingsViews`: obtém as 10 músicas mais ouvidas;
- `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). Todas as *queries* realizadas são de leitura de informação, sendo estas dividas 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) de todos os albums;
- `listAlbumsByFilter`: dado um *offset* e um filtro, devolve o mesmo que o anterior contudo em vez da lista ordenada 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 website 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) de todas os areas;
- `listAreasByFilter`: dado um *offset* e um filtro, devolve o mesmo que o anterior contudo em vez da lista ordenada 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 website 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.

Artist:

- `listArtists`: dado um *offset*, lista os 50 artistas (id e nome do artista) a partir do *offset* da lista ordenada (pelos nomes) de todas os artistas;
- `listArtistsByFilter`: dado um *offset* e um filtro, devolve o mesmo que o anterior contudo em vez da lista ordenada 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 website 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 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) de todas as músicas;
- `listRecordingsByFilter`: dado um *offset* e um filtro, devolve o mesmo que o anterior contudo em vez da lista ordenada 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;
- `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 website 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/statsMore` - `getMoreRecordingsViewsUser(: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, é gerado 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
  - `/: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 realizado `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`
  - `/more` - `getMoreRecordingsViews()`
  - `/: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`
  - `/: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}`.

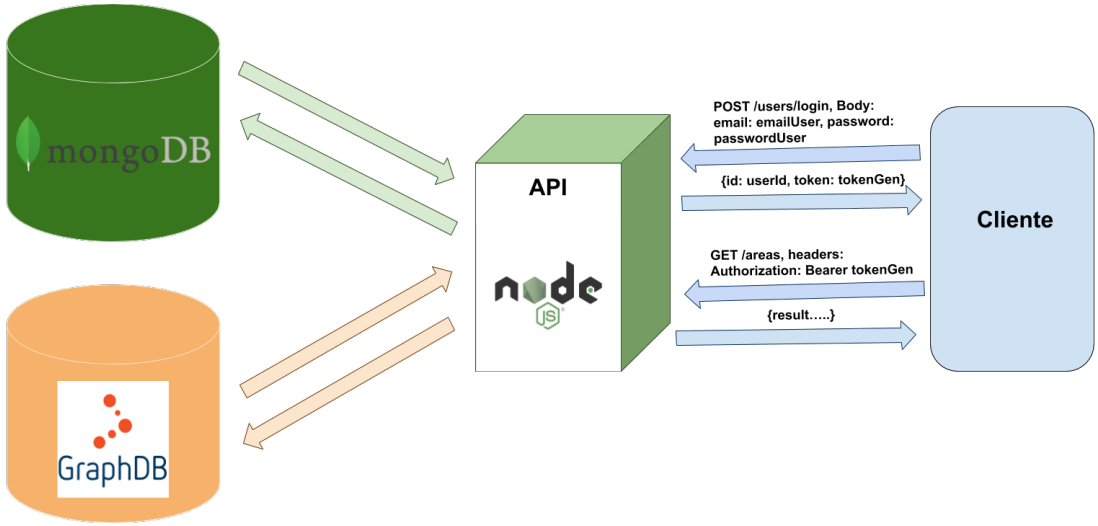

Figura 3: Um exemplo de um login e posterior pedido com envio do *token*

O *token* tem um tempo de validade, pelo que ao fim de 30 minutos expira. 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*.

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/`.

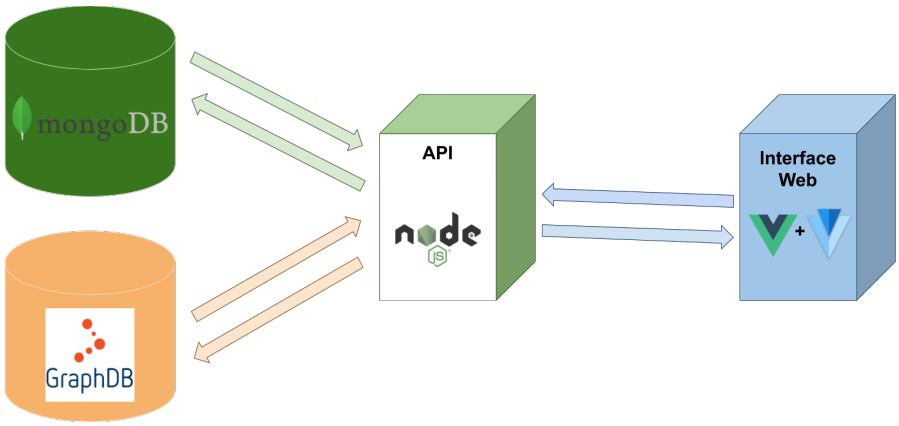

Figura 4: Estrutura do backend e frontend

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 allguns *URLs* importantes, desde API 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 e mais fácil de futuramente ser mantido. 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* existem rotas a definir, indicando para que vista as mesmas 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 `/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` e `UserSettings.vue` verificam se o utilizador está autenticado, em caso afirmativo é feito a renderização da páginas; 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 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.

`request.js`:

Este ficheiro foi criado de forma a possuir funções que simplifiquem o uso da *package* `axios`, 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 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.

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, 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.

`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 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 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* e na página inicial *Index*), esta caixa de texto permite pesquisar por arista 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, 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 os 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 o rating 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 o rating do artista. (ver 3)

`Recording.vue`:

Página que apresenta toda a informação de um determinado individuo da classe *Recording*. Apresenta também a classificação e o rating da música. Para além disso mostra o vídeo da música (quando possível, nem sempre acerta no vídeo correto) e as *lyrics* da música. As *lyrics* da música são obtidas a partid de duas APIs diferentes, *musixmatch* e *Chartlyrics*. (ver 3)

`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).

`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 mais se associa.

7 Instalação

[1]

8 Conclusões e Trabalho Futuro

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.