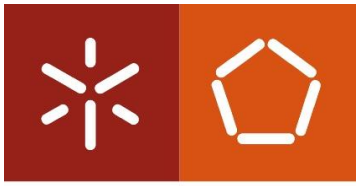

Universidade do Minho

Mestrado Integrado de Engenharia Informática

Laboratório de Engenharia Informática (2ºSemestre - 2020/21)

Relatório LEI - Api Clav

Grupo Nº 33

A85813 António Alexandre Carvalho Lindo

A85400 Nuno Azevedo Alves da Cunha

A85919 Pedro Dias Parente

Braga

28 de junho de 2021

Resumo

Este relatório tem como objetivo descrever a solução criada pelo grupo para atender à proposta de projecto "CLAV - À procura de uma API de dados alternativa"sumbmetida pelo professor José Carlos Ramalho

A meta do trabalho consistiu em procurar alternativas em diversas partes da arquitetura da API atualmente implementada, desde arquitetura até implementação de cache e passando pela persistencia dos dados. Foi ainda desenvolvido um pequeno front end para a API a título de exemplo da utilização da mesma.

Para implementação da API de dados o grupo optou por uma arquitetura GraphQL ao invés de Rest recorrendo à framework *apollo-express*, como base de dados foi utilizado o ArangoDB, uma base de dados NoSQL mutiparadigma, para o qual o grupo migrou um dataset ontológico.

O foco principal deste trabalho passou pela exploração das características destas tecnologias e de que forma estas poderiam ser aplicadas ou não da melhor forma ao problema apresentado.

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

A plataforma CLAV é destinada à classificação e avaliação da informação publica, ou seja é destinada a ser aplicada a qualquer processo público em portugal, desde contratos de construção até inscrições em universidades públicas.

Esta plataforma serve entidades da Administração Pública, a empresas públicas e a outras entidades de Portugal Continental e permite a interação entre estas e a Direção-Geral do Livro, dos Arquivos e das Bibliotecas.

É igualmente destinada ao cidadão, particularmente através da disponibilização do catálogo de processos de negócio da Administração Pública.

Este trabalho tem por objectivo replicar a API de dados existente que implementa todas as funcionalidades necessárias para interagir com a plataforma CLAV usando tencnologias que procuram resolver as principais dificuldades da plataforma original.

Capítulo 2

Modelo de dados a implementar

2.1 Plataforma CLAV

Como foi dito anteriormente, a plataforma CLAV, é destinada à classificação e avaliação da informação publica. O componente principal desta plataforma é a Lista Consolidada, um referencial que engloba todas as funções e actividades da função publica em portugal.

Estas funcões e actividades estão codificadas e organizadas num sistema de classes com diferentes niveis. Os niveis 1 e 2 são apenas classes de organização para estruturar a informação, os processos de negócio propriamente ditos estão descritos nas classes de nivel 3 e os seus sub-processos no nivel 4, por exemplo:

- **Nivel 1 - C600**: Administração da Justiça
- **Nivel 2 - C600.10**: Prevenção e investigação criminal
- **Nivel 3 - C600.10.506**: Ação de proteção de testemunhas
- **Nivel 4 - C600.10.506.02**: Ação de proteção de testemunhas: Ação de proteção de testemunhas: destruição judiciária dos documentos de identificação

Associados a estes processos aparecem também no sistema Entidades Publicas que participam nestes processos bem como as legislações a eles associados. Existem para cada classe ainda uma série de notas e termos de indice que descrevem a aplicação do processo, e um campo sobre o destino final a dar a cada processo.

A plataforma clav disponibiliza ainda a funcionaliade de tabelas de seleção que consiste em subsets da lista consolidade, embora esta faceta do projeto naõ seja implementada na nossa réplica por questões de tempo.

2.2 ArangoDB e Import do Dataset

2.2.1 Estrutura da Base de Dados

A base de dados utilizada para armazenar a informação foi o ArangoDB. Trata-se de uma base de dados NoSQL multiparadigma que permite guardar informação de várias formas e cuja linguagem de interrogação se chama Arango Query Language (AQL). A estrutura da base de dados é um grafo com nodos e arestas, tendo os nodos a informação e as arestas o nome do nodo de origem e de destino (que esta liga). A imagem seguinte mostra um exemplo deste tipo de estrutura para as Tipologias:

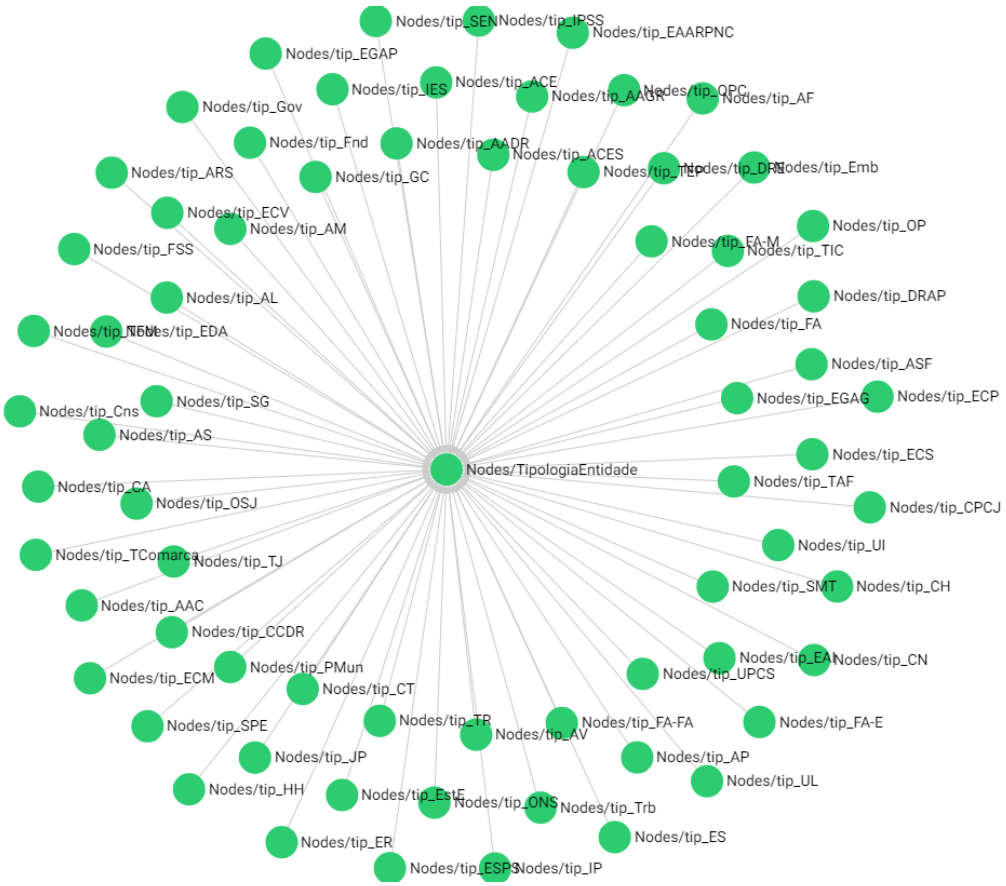

Figura 2.1: Grafo Tipologias

Cada nodo tem a informação de cada tipologia:

_key	_id	_rev	tipEstado	tipSigla	tipDesignacao
tip_AAC	Nodes/tip_AAC	_cKEhM5u--A	Ativa	AAC	Autoridades Administrativas Civis
tip_AADR	Nodes/tip_AADR	_cKEhM56--C	Ativa	AADR	Assembleias de apuramento distrital de resultados
tip_AAGR	Nodes/tip_AAGR	_cKEhM6G--A	Ativa	AAGR	Assembleias de apuramento geral dos resultados
tip_ACE	Nodes/tip_ACE	_cKEhM6S--C	Ativa	ACE	Administração Central do Estado
tip_ACES	Nodes/tip_ACES	_cKEhM6a--C	Ativa	ACES	Agrupamentos de Centros de Saúde
tip_AF	Nodes/tip_AF	_cKEhM6i--E	Ativa	AF	Assembleias de Freguesia
tip_AL	Nodes/tip_AL	_cKEhM6u---	Ativa	AL	Autarquias Locais
tip_AM	Nodes/tip_AM	_cKEhM7K---	Ativa	AM	Assembleias Municipais
tip_AP	Nodes/tip_AP	_cKEhM7----	Ativa	AP	Administração Pública
tip_ARS	Nodes/tip_ARS	_cKEhM7G--A	Ativa	ARS	Administrações Regionais de Saúde
tip_AS	Nodes/tip_AS	_cKEhM7S---	Ativa	AS	Autoridades de Saúde
tip_ASF	Nodes/tip_ASF	_cKEhM7a--C	Ativa	ASF	Autoridades de supervisão e fiscalização
tip_AV	Nodes/tip_AV	_cKEhM7i--C	Ativa	AV	Assembleias de voto
tip_CA	Nodes/tip_CA	_cKEhM7y---	Ativa	CA	Centros de Arbitragem
tip_CCDR	Nodes/tip_CCDR	_cKEhM76---	Ativa	CCDR	Comissões de Coordenação do Desenvolvimento Regional

Figura 2.2: Nodos

Cada aresta tem a origem e destino da mesma:


```
[
  {
    "_key": "1678734",
    "_id": "edges/1678734",
    "_from": "Nodes/tip_AAC",
    "_to": "Nodes/TipologiaEntidade",
    "_rev": "_cKEhM5m--C",
    "rel": "type"
  },
  {
    "_key": "1678748",
    "_id": "edges/1678748",
    "_from": "Nodes/tip_AADR",
    "_to": "Nodes/TipologiaEntidade",
    "_rev": "_cKEhM5y---",
    "rel": "type"
  },
  {
    "_key": "1678762",
    "_id": "edges/1678762",
    "_from": "Nodes/tip_AAGR",
    "_to": "Nodes/TipologiaEntidade",
    "_rev": "_cKEhM6----",
    "rel": "type"
  },
  {
    "_key": "1678776",
    "_id": "edges/1678776",
    "_from": "Nodes/tip_ACE",
    "_to": "Nodes/TipologiaEntidade",
    "_rev": "_cKEhM6K---",
    "rel": "type"
  },
```


Figura 2.3: Arestas

Na base de dados existem, então, duas coleções: Nodes e Edges que guardam os nodos e arestas respetivamente, ambos os nodos e as arestas são documentos json na base de dados, sendo que as arestas possuem obrigatóriamente os campos _from e _to. É depois criado um grafo que consome estas duas coleções para se construir.

2.2.2 Estrutura ontológica nos grafos do arango

Toda a semântica da ontologia está incluida na base de dados, ou seja estão presentes todos os nodos que descrevem classes, data e object properties bem como Named Individuals no entanto esta informação não é utilizada diretamente pelo arango o grupo desenvolveu então as seguintes estratégias para melhor definir a sua representação

Data properties

Para alem nos nodos proprios que descrevem cada data property da ontologia estas são associadas aos Named Individuals através dos documentos dos mesmos. Ou seja se na ontologia estiver definida uma entidade :Peixe :cor "Azul"o resultado no arango será um documento deste género:

```
{
    _key: "Peixe"
    cor: "Azul"
}
```

Object properties

As Object properties da ontologia aparecem no arango através das arestas (Para além de serem descritas individualmente no seu próprio nodo). Na ontologia as object properties não são "pesadas", o que significa que embora as arestas do arango possam conter qualquer informação de um documento json elas são apenas utilizadas para guardar a origem destino e o tipo de relação. Por exemplo numa relacão ontológica de tipo :Peixe2 :FilhoDe :Peixe1 é produzida a seguinte aresta:

```
{
    _from: "Peixe2"
    _to: "Peixe1"
    rel: "FilhoDe"
}
```

Named Individuals

Os Named Individuals são representados como Documentos na coleção nodes e em que o campo _key é o identificador do individuo. Ou seja o individuo do tipo :Peixe rdf:type :Animal teria a seguinte configuração no arango:

```
{
    _key: "Peixe"
}
```

Sem esquecer a aresta:

```
{
    _from: "Peixe"
    _to: "Animal"
    rel: "type"
}
```

2.2.3 Import dos dados

Para o import dos dados, foi implementado um script em JavaScript para converter a informação do formato Turtle para os respetivos nodos e arestas já discutidos. Isto porque o ArangoDB não permite o import de ficheiros rdf, uma vez que não se trata de uma base de dados ontológica. O script conecta-se à base de dados criada, lê o ficheiro Turtle e vai criando os nodos e as arestas utilizando queries em AQL (Arango Query Language) de insert, colocando em cada um a respetiva informação. Para isto foi utilizada a biblioteca arangojs do JavaScript.

Capítulo 3

API de dados

3.1 GraphQL

Para a camada da API de dados o grupo deu preferência a uma arquitetura GraphQL ao invés da arquitetura Restful a ser utilizada na API original, para este fim foi utilizada a framework ***Apollo-Server***.

Uma API GraphQL é constituida por duas partes essenciais, o seu schema, onde estão definidos os tipos de queries que pode receber e os tipos dos dados de output e os seus resolvers onde é efectuado o peenchimento do output.

```
const server = new ApolloServer({
    schema,
    resolvers})
```

Figura 3.1: Configuração do servidor GraphQL

3.1.1 GraphQL Schema

A scheema graphql é definida a partir de 2 tipos iniciais Query (Equvalente a um GET) e mutation (Equivalente a um POST), dentro depois destes tipos são designadas as queries que se podem realizar bem como a estrutura do seu output.

```
type Query {
    users: [User!]!
    user(id: String!): User
    entidades: [Entidade!]!
    entidade(_key: String!): Entidade
    legislacoes: [Legislacao!]!
    legislacao(_key: String!): Legislacao
    tipologias: [Tipologia!]!
    tipologia(_key: String!): Tipologia
    classes(tipo: String, nivel: Int, ents: [String!], tips: [String!]): [Classe!]!
    classesTree: [ClasseTree]
    indicadores: Indicadores!
}
```

Figura 3.2: Schema GraphQL

Olhando para a primeira linha podemos ver que a query users retornará uma lista do tipo "User" que está definido da seguinte forma:

```
type User {
    _key: String
    username: String!
    level: Int!
    permissions: Permissions!
    internal: Boolean!
    email: String!
    local: Local!
    entidade: String!
    notificacoes: [String!]!
}
```

Figura 3.3: Schema User

Assim ao executar esta query do lado do cliente sera possivel pedir a query ***users*** fazendo request de qualquer combinação dos campos definidos no tipo ***users***, este processo será exemplificado mais a frente no relatório.

3.1.2 GraphQL Resolvers

Os resolvers são a parte da API GraphQl que geram o output a ser enviado em cada Query. Estes resolvers são implementados de acordo com a query definida no schema.

```
Query: {
    users: (obj, args, context) => {
        return users.list(context)
    },
```

Figura 3.4: Resolver User

Aqui é possivel ver que para o resolver da query users é efectuada uma comunicação com a base de dados através da função *Users.list()* que devolve a lista de todos os utilizadores.

Neste caso a função devolve o objecto User completo e a framework trata depois de o filtrar apenas para os campos pedidos na query, mas é possivel definir resolvers individuais para cada campo do Objecto user, por exemplo um resolver para ir buscar o username, outro o email etc... Nesse caso são acionados apenas os resolvers dos campos que são pedidos na query. Isto não é relevante neste caso, mas em objectos mais extensos como é o caso das classes do CLAV ir buscar apenas um certo numero de campos pode ser um ganho excelente de performance.

3.2 Comunicação com a base de dados

A comunicação com a base de dados ArangoDB é feita utilizando módulo NodeJS ArangoJS. Este módulo estabelece uma ligação com a base de dados e permite depois o envio de queries na linguagem utilizada pelo ArangoDB, o AQL (Arango Query Language). É através do uso de AQL que são enviadas queries para a base de dados a pedir informações. De seguida encontrar-se um exemplo de uma query em AQL:

```
FOR v, e IN 1 INBOUND 'Nodes/Legislacao' GRAPH 'Graph'
                                FILTER e.rel == 'type'
                                RETURN v
```

Como podemos observar, trata-se de uma linguagem de facil compreensão. A query de exemplo, percorre todos os nodos de Legislação do nosso grafo (de nome Graph) e para cada vertice v e aresta e, filtra as arestas que têm a relação type e que se ligam ao vertice em questão. Ou seja, esta query devolve todas as legislações. No AQL também é possível devolver objetos personalizados e ainda utilizar variaveis como no exemplo seguinte:

```
Let n = (FOR v, e IN 1 INBOUND 'Nodes/Legislacao' GRAPH 'Graph'
                                                FILTER e.rel == 'type'
                                                RETURN v)
                                        return {l: LENGTH(n)}
```

Como podemos observar nesta query, é devolvido o tamanho de n, ou seja o número de legislações. LENGTH é uma das várias funções auxiliares do AQL, estando todas elas documentadas no website do ArangoDB. Para inserir informação na base de dados, utilizamos queries insert do AQL que têm o seguinte aspeto:

```
INSERT ${legislacao} INTO Nodes
                                LET inserted = NEW RETURN inserted
```

No exemplo utilizado, legislacao é um objeto passado como argumento, no entanto é este o aspeto das querys insert. Obviamente para inserir informação é necessário que haja inserts nos vértices, como na caso acima, mas também nas arestas do grafo. Para inserir nas arestas, bastava trocar Nodes por Edges na query. De notar que Nodes é o nome da coleção dos vértices e Edges da coleção das arestas, que foram escolhidos por nós.

3.3 Semântica ontológica na camada da API de Dados

Uma vez que a base de dados utilizada neste projeto não é ontológica acabam por existir algumas queries em que a semântica da ontologia tem que ser simulada na camada aplicacional. Para isto foram desenvolvidas 2 funções que o grupo usou para procurar fazer uso da informação sobre a ontologia presente na base de dados. É de notar que estas ferramentas foram criadas numa base de necessidade e não são exaustivas sobre toda a semântica da ontologia incluida na base de dados.

3.3.1 Filtros semânticos AQL

Primeiramente a função ***buildSemanticFilter*** (presente no Anexo A) tem como objectivo construir um filtro AQL que inclui uma relação e todas as suas sub-relações e relações inversas, é possivel usar depois este filtro gerado em várias queries. Esta função é utilizada em casos que o grupo quer por exemplo procurar por um tipo de relação mas como arestas na base de dados estão apenas incluidas as suas sub-relações. Por exemplo, no caso da procura de todos os participantes de uma classe, pretende-se porcurar pelas relações "temParticipante", no entanto na base de dados não há nenhuma aresta referente a essa relação, apenas existem arestas relativas as suas sub-relações como por exemplo "temParticipanteApreciador", a função iria então neste caso construir um filtro que procura não só por arestas com "temParticipante"mas também com qualquer uma das suas sub-relações ou relações inversas das mesmas.

3.3.2 Verificação de Subpropriedades

A funcção ***isSubPropertyOfRel*** (incluida no Anexo B) foi criada para auxiliar num contexto de acreção à base de dados. Dadas duas relações a função determina se a primeira é um sub-relação da segunda. Esta função é util para poder controlar se a inserção da informação na base de dados não é inconsistente com a ontologia presente. Por exempo voltando ao exemplo anterior, ao inserir os participantes de uma classe será necessário inserir arestas com o tipo de relação de cada participante, ao inserir um participante com o tipo de relação "temParticipanteApreciador"é corrida esta função para garantir que esta relação é subPropertyOf "temParticipante"e portanto faz sentido colocar na base de dados neste contexto.

3.4 Queries GraphQL e outputs

Nesta secção, serão exemplificados alguns exemplos da utilização de queries no GraphQL e os respetivos outputs. Para começar será mostrada uma query simples, que permite obter a lista das legislações:
Input:

```
query{
    legislacoes
  {
    _key
    diplomaData
    diplomaLink
    diplomaTipo
    diplomaEstado
    diplomaNumero
    diplomaSumario
  }
}
```

Output:


```
{
  "data": {
    "legislacoes": [
      {
        "_key": "leg_DJrBJtVRKrGlRrtFchnZy",
        "diplomaData": "1941-15-12",
        "diplomaLink": "https://dre.pt/application/file/469131",
        "diplomaTipo": "Decreto",
        "diplomaEstado": "Ativo",
        "diplomaNumero": "31730/41",
        "diplomaSumario": "Regulamento das Alf ndegas"
      },
      {
        "_key": "leg_NmUzXCyAoXL1iQRnPEfh3",
        "diplomaData": "1959-14-11",
        "diplomaLink": "https://dre.pt/application/file/438874",
        "diplomaTipo": "DL",
        "diplomaEstado": "Ativo",
        "diplomaNumero": "42644/59",
        "diplomaSumario": "Atualiza as disposi  es privativas do Registo Comercial"
      },
      {
        "_key": "leg_laLI-a178i5aw6QA6L22N",
        "diplomaData": "1959-14-11",
        "diplomaLink": "https://dre.pt/application/file/438975",
        "diplomaTipo": "DL",
        "diplomaEstado": "Ativo",
        "diplomaNumero": "42645/59",
        "diplomaSumario": "Aprova o Regulamento do Registo Comercial"
      },
      {
      ...
      }
}
```


Como a query é um GET, é necessário indicar isso ao GraphQL (query), depois indica-se o nome da

query (legislacoes) e dentro é indicado um objeto com os campos que pretendemos que sejam retornados. Por exemplo, se so quisessemos as keys de cada legislação, teríamos de executar a seguinte query:

```
query{
    legislacoes
  {
    _key
  }
}
```

O resultado seria o seguinte:


```
{
  "data": {
    "legislacoes": [
      {
        "_key": "leg_DJrBJtVRKrGlRrtFchnZy"
      },
      {
        "_key": "leg_NmUzXCyAoXL1iQRnPEfh3"
      },
      {
        "_key": "leg_laLI-a178i5aw6QA6L22N"
      },
      {
        "_key": "leg_EXtdRCw9frAX5Zw3mM-ns"
      },
      {
        "_key": "leg_Dk8xQTCpUB5HzqT2k8Y10"
      },
      {
        "_key": "leg_yrLTFzyQ8ECVWRnpF2vC0"
      },
      {
        "_key": "leg__Z_ztyqykKAjaYxIeuoYy"
      },
      {
      ...
      }
}
```


Como é possível observar, o output desta query é também uma lista das legislações mas apenas com os campos pedidos na query (neste caso apenas o campo key).
Caso um utilizador pretenda aceder a um determinado objeto, podem ser passados argumentos nas queries. Por exemplo, se pretendermos obter uma dada legislação, teríamos de executar uma querie como a seguinte:

```
query{
  legislacao(_key: "leg_DJrBJtVRKrGlRrtFchnZy"){
    _key
    diplomaData
    diplomaLink
    diplomaTipo
    diplomaEstado
    diplomaNumero
    diplomaSumario
  }
```

```
}
```

Neste caso, a querie devolve a legislação passada como argumento (leg_DJrBJtVRKrGlRrtFchnZy) com todos os campos pedidos:

```
{
  "data": {
    "legislacao": {
      "_key": "leg_DJrBJtVRKrGlRrtFchnZy",
      "diplomaData": "1941-15-12",
      "diplomaLink": "https://dre.pt/application/file/469131",
      "diplomaTipo": "Decreto",
      "diplomaEstado": "Ativo",
      "diplomaNumero": "31730/41",
      "diplomaSumario": "Regulamento das Alf ndegas"
    }
  }
}
```

Se fosse pretendido apenas obter certos campos, bastava indicar os mesmos na query, como foi exemplificado anteriormente.
O outro tipo de queries são as mutations. As mutations são utilizadas para inserir informação. Por exemplo, se quisermos inserir um utilizador na plataforma, terámos de executar a seguinte mutation:

```
mutation {
  registerUser(u: {
    _key: "NunoCunha",
    username: "nuno",
    email: "email@email.com",
    level: 7,
    permissions: {
        LC: true
        AE: true
        ES: true
    },
    local:{
      password: "password"
    },
    internal: true,
    entidade: "ent_NUNO",
    notificacoes: []
  }) {
    _key
    email
    local{password}
  }
}
```

Como podemos observar, é passado como argumento um objeto que contem a informação do utilizador a inserir e depois são, tal como é costume, indicados os campos a receber pela API (_key, email, localpassword). A resposta do servidor a esta mutation seria:

```
{
  "data": {
    "registerUser": {
      "_key": "NunoCunha",
      "email": "email@email.com",
      "local": {
        "password": "$2a$14$78lphTrpGrIBYn9DkyeVJOhuhILr4UPGDIHmoencC2lVdvSVQjgW6"
      }
    }
  }
}
```

3.5 Sistema de Cache

O sistema de cache foi implementado com o módulo "node-cache", que é um simples módulo que guarda uma estrutura de dados em memória, a qual se pode aceder através de uma key. Isto funciona bem para o nosso caso, conseguindo reduzir o fecthing (por exemplo) das classes de cerca de 20/25 segundos para 5 segundos, porém, se as classes não tiverem em cache ainda, a primeira vez que é feito este pedido tem que se fazer este fetch demoroso para depois guardar em cache. A cache é também atualizada quando ocorre uma transação que mude a base de dados.

Porém, existe uma limitação óbvia, não conseguimos dar fetch a campos específicos da cache. Usando outra vez o exemplo das classes, se esta ainda nao estiver em cache, o fetching de campos específicos é possível, mas se já estiver em cache ele vai buscar este objeto por inteiro e só depois seleciona os campos que quer retornar.

3.6 Docker

Para o deployment da API, foi construído um docker-compose. O mesmo contem dois containers, um do arangoDB e outro do NodeJS interligados por uma network. O ficheiro docker-compose.yml encontra-se na imagem seguinte:

```yaml
version: '3.7'
networks:
  isolation-network:
    driver: bridge

services:
  arangodb_db_container:
    image: arangodb:latest
    restart: always
    environment:
      ARANGO_ROOT_PASSWORD: coficofi1
    ports:
      - 8529:8529
      - 4001:4000
    volumes:
      - arangodb_data_container:/var/lib/arangodb3
      - arangodb_apps_data_container:/var/lib/arangodb3-apps
      - ./CLAV-GraphQL-API/povoamento:/home
    networks:
      - isolation-network

  node:
    image: "node:current"
    user: "node"
    working_dir: /home/node/app
    environment:
      - NODE_ENV=production
    volumes:
      - ./CLAV-GraphQL-API:/home/node/app
    ports:
      - 4000:4000
      - 8530:8529
    command: sh -c "npm install && npm start"
    #command after ready: cd povoamento; node arango.js
    networks:
      - isolation-network

volumes:
  arangodb_data_container:
  arangodb_apps_data_container:
```

Figura 3.5: Docker-compose

Para executar o docker-compose é necessário que o ficheiro docker-compose.yml (que no resositório está na pasta docker) esteja na mesma diretoria onde se encontra a diretoria "CLAV-GraphQL-API"(ou seja, uma diretoria acima de onde se encontra). Feito isto, basta executar o comando *docker-compose up -d*. Depois de executado, é necessário abrir um terminal no container do NodeJS e executar os seguintes comandos: *cd povoamento* e *node arango.js* que irão executar o script de povoamento e povoar a base de dados, logo basta executar este dois comandos apenas na primeira vez que for utilizado o docker, uma vez que a base de dados ja vai estar povoada nas próximas. Estas instruções também estão descritas no README contido na pasta docker.

Capítulo 4

Conclusão

Em conclusão, julgamos que nos conseguimos manter fiéis à API original dentro do possível. Como o ArangoDB não se trata de uma base de dados ontológica e como o graphQL tem a vantagem de se poder selecionar apenas certos campos do output, nem sempre foi possível ter a API a funcionar da mesma forma que a original. No entanto, conseguimos que a implementa-la de modo a que tivesse as mesmas funcionalidades que a original.

Durante a execução deste projeto, as nossas maiores dificuldades consistiram no facto de o ArangoDB não ser uma base de dados ontológica e portanto não deduzir informação.

No fim deste trabalho conseguimos concluir também, em luz do que foi escrito anteriormente, que se se pudesse refazer a API de raiz provavelmente conseguiamos algo mais eficiente, pois tentar recriar a API existente, que funcionava por cima de uma base de dados ontológica, com uma query language nao ontológica, fez com fossem criadas dificuldades desnecessárias. Com isto em mente, em futuras iterações do trabalho provavelmente não utilizaríamos o ArangoDB como base de dados. Por outro lado, ficamos muito impressionados com a versatilidade e capacidade de resposta do GraphQL. A sua capacidade de ir buscar apenas a informação necessária com uma query simples (e código simples e intuitivo também) resolvendo os problemas de under e over-fetching presente nas API's RESTful, fez com que esta tecnologia tenha feito uma boa impressão na equipa.

Capítulo 5

Referências

5.1 Referências Eletrónicas

Site da plataforma clav. Disponíel em:
https://clav.dglab.gov.pt/

GraphQl Official tutorials. Disponível em:
https://graphql.org/learn/

Apollo-express GraphQL server. Disponível em:
https://www.apollographql.com/docs/apollo-server/v1/servers/express/

Arango and AQL documentation. Disponível em:
https://www.arangodb.com/documentation/

Ontology web language documentation. Disponível em:
https://www.w3.org/TR/owl-features/

Apêndice A

Função buildSemanticFilter()

```
let buildSemanticFilter = (context,rel) => {
    //FIND Semantic value stored in database
    let aqlrel = aql.literal(rel)
    return context.db.query(aql`LET subproperties = (FOR v,e IN 1 INBOUND 'Nodes/${aqlrel}' GRAPH 'Graph'
                                                        FILTER e.rel == 'subPropertyOf'
                                                    RETURN v)

                                LET inverses = (FOR node IN subproperties
                                                    FOR v,e IN 1 INBOUND node._id GRAPH 'Graph'
                                                    FILTER e.rel == 'inverseOf'
                                                RETURN v)

                                return APPEND(subproperties,inverses)`)
        .then(resp => resp.all()).then((list) => {
            let query = `FILTER rel.rel == '${rel}'`
            list[0].forEach(element => {
                query += ` || rel.rel == '${element._key}'`
            });

            return query
        })
        .catch(err => console.log(err))
}
```

Figura A.1: buildSemanticFilter

Apêndice B

Função isSubPropertyOfRel()

```
let isSubPropertyOfRel = async (context,sub,rel) => {
    let subl = aql.literal(sub)
    let rell = aql.literal(rel)
    return context.db.query(aql`let subProps = (FOR v,e IN 1 INBOUND 'Nodes/${rell}' GRAPH 'Graph'
        FILTER e.rel == 'subPropertyOf'
        RETURN v._key)

        return CONTAINS(subProps,'${subl}')`)
        .then(resp => resp.all()).then(list => list[0])
        .catch(err => console.log(err))
}
```

Figura B.1: isSubPropertyOfRel