INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

HOMI

Hybrid Ontology Mapping Interface

Ana Carolina Baptista
41487@alunos.isel.ipl.pt
960314580

Eliane Almeida
41467@alunos.isel.ipl.pt
960271968

Projeto e Seminário

Licenciatura em Engenharia Informática e Computadores

Relatório Final

Orientadores:
Cátia Vaz, ISEL, cvaz@cc.isel.ipl.pt
José Simão, ISEL, jsimao@cc.isel.ipl.pt
Alexandre P. Francisco, IST, aplf@ist.utl.pt

Julho de 2018

Instituto Superior de Engenharia de Lisboa
Licenciatura em Engenharia Informática e de Computadores

Hybrid Ontology Mapping Interface

Ana Carolina Grangeiro Pinto Baptista, 41487
Eliane Socorro de Almeida, 41467

Orientadores: Cátia Vaz (ISEL)
José Simão (ISEL)
Alexandre P. Francisco (IST)

Relatório final realizado no âmbito de Projeto e Seminário,
Do curso de licenciatura em Engenharia Informática e de Computadores
Semestre de Verão 2017/2018

14 de Julho de 2018

Resumo

....resumo

Índice

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Atualmente, com o grande crescimento e propagação de dados na internet, surge a necessidade de que a informação seja descrita e transmitida por meio de uma linguagem *standard*, sendo esta de fácil entendimento tanto para computadores quanto para humanos.

Uma das técnicas de descrição de informação que se está a tornar muito popular é baseada em ontologias [1]. Esta permite especificar explicitamente uma conceptualização ou um conjunto de termos de conhecimento para um domínio particular. Apesar da popularidade das ontologias, há em geral dificuldade em transformar o conhecimento pré-definido num caso concreto.

Na área da bioinformática, existem recursos científicos que necessitam de ser partilhados entre a comunidade científica por meio de ontologias. Sendo as ontologias normalmente definidas através de OWL [2] (*Web Ontology Language*), em várias situações poderá não ser uma tarefa simples para os bioinformáticos representar o seu conhecimento do domínio através das ontologias.

Atualmente existem algumas ferramentas de edição de ontologias que permitem ao utilizador inserir um ficheiro referente a uma ontologia e criar novos dados de acordo com este ficheiro, como por exemplo *Protégé* [3]. Existem também algumas bibliotecas para Java que fazem o mapeamento de XML para OWL, que podem ser utilizadas por *developers*. Contudo, não temos conhecimento da existência de uma ferramenta que combine estes dois aspetos: a criação de novas instâncias através de uma ontologia bem como o mapeamento de um caso concreto escrito noutra linguagem numa instância de uma ontologia.

Desta forma, de modo a ajudar os utilizadores – como por exemplo, os biólogos - o objetivo do nosso trabalho é desenvolver uma aplicação que tenha uma interface intuitiva que permita esta transformação de dados semiestruturados em dados anotados com ontologias definidas em OWL. Nesta interface também teria a possibilidade de anotar valores aos vários conceitos da ontologia ou apenas editar os existentes.

1.1 Sinopse

Este relatório está dividido em 8 capítulos.

No Capítulo 2 iremos explicar sucintamente o que é uma ontologia e OWL.

No Capítulo 3 iremos clarificar qual é o contexto do nosso projeto bem como que ferramentas já existem.

No Capítulo 4 iremos descrever o sistema *Chaos Pop* que fornece uma API que iremos utilizar no nosso programa.

No Capítulo 5 iremos descrever a nossa arquitetura, bem como cada componente e como estes se relacionam entre si.

No Capítulo 6 iremos resumir a implementação do nosso projeto nos aspetos de *front-end*, *back-end* e base de dados.

Capítulo 2

Ontologias e OWL

2.1 Ontologia

Uma ontologia, na área da ciência da computação, é um modelo de dados que representa um grupo de ideias ou conceitos dentro de um determinado domínio e as relações que existem entre eles. Estas são usadas em várias áreas da ciência da computação, tais como inteligência artificial e semântica web, como uma forma de representar conhecimento sobre essa área, ou sobre um subconjunto dessa área. Uma ontologia descreve indivíduos (objetos básicos), classes (conjuntos, coleções ou tipo de objetos), atributos (propriedades, características ou parâmetros) e relacionamentos (entre objetos). [4]

Na Figura 2.1 podemos observar a representação em grafo de um exemplo de uma ontologia, composta por indivíduos, identificados pelos retângulos "Person", "Father", "Son", "Mother"; e por relações, identificados pelas setas a tracejado ("hasFather", "hasMother", "hasSon").

Na Figura 2.2 podemos observar uma instância da ontologia ilustrada pela Figura 2.1, onde o indivíduo "João" tem uma relação com o indivíduo "José" denominado "temPai" e com o individuo "Maria" denominada de "". Ambos estes indivíduos são do tipo "Person"

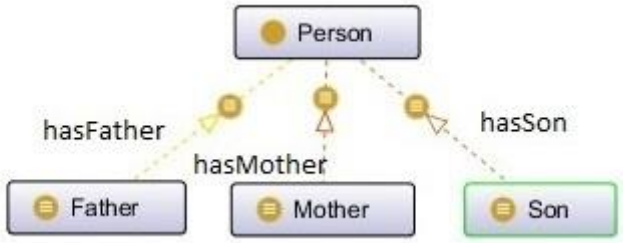

Figura 0.1 - Exemplo de uma ontologia

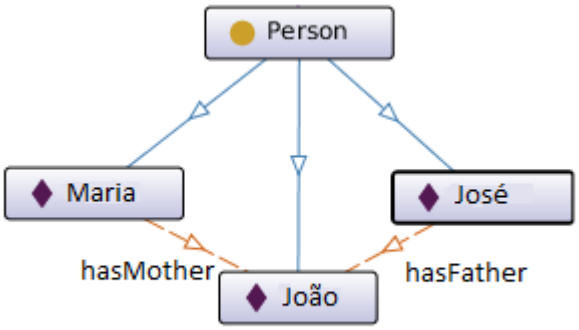

Figura 0.2 - Exemplo de uma instância de uma ontologia

2.2 OWL

OWL (*Ontology Web Language*) é uma linguagem utilizada para definir e instanciar ontologias em Web. Em OWL conseguimos definir as 4 propriedades de ontologias descritas acima (indivíduos, classes, atributos e propriedades). Esta linguagem foi desenhada para processar informação facilitando assim a sua interpretação por máquinas. Quando comparado com outras linguagens (XML, RDF, etc.), OWL destaca-se por fornecer um vocabulário adicional com uma semântica formal. É a linguagem recomendada da W3C (*World Wide Web Consortium*) para definir ontologias em Web.

Um documento de OWL consiste num cabeçalho da ontologia (ou seja, um *hyperlink* para o documento que contém informação sobre essa ontologia; este cabeçalho é opcional), inúmeras definições de **classe**, **indivíduos** e **propriedades**. Uma **classe** fornece um mecanismo de abstração para o agrupamento de recursos com características similares. Um **indivíduo** de cada classe é denominado de instância dessa classe. As **propriedades** em OWL podem ser de 2 tipos: ***Object Properties***, que são referências para indivíduos e ***Datatype Properties*** que referenciam *data values*. Uma *property* pode ter relações com outras *properties* dos tipos *equivalentProperty* e *inverseOf,* assim como pode conter restrições globais de cardinalidades que podem ser *FunctionalProperty* ou *InverseFunctionalProperty* [5].

Existem vários formatos que podem ser usados para representar uma ontologia: RDF/XML *Syntax*, *Turtle Syntax*, OWL/XML *Syntax*, OWL *Functional Syntax*, entre outros. Abaixo temos um exemplo da descrição de uma ontologia utilizando o formato *Turtle [13]*.

```
@prefix : <http://sysresearch.org/ontologies/family.owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix owl2xml: <http://www.w3.org/2006/12/owl2-xml#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

:hasFirstGivenName a owl:DatatypeProperty ;
   rdfs:subPropertyOf :hasName ;
   rdfs:domain :Person ;
   rdfs:range xsd:string .

:hasName a owl:DatatypeProperty ;
   rdfs:domain :Person ;
   rdfs:range xsd:string .

:hasFamilyName a owl:DatatypeProperty ;
   rdfs:subPropertyOf :hasName ;
```

```
   a owl:FunctionalProperty ;
   rdfs:domain :Person ;
   rdfs:range xsd:string .

:hasPartner a owl:ObjectProperty ;
   owl:inverseOf :isPartnerIn .


:isPartnerIn a owl:ObjectProperty ;
   rdfs:domain :Person ;
   rdfs:range :Marriage .

:hasSex a owl:ObjectProperty , owl:FunctionalProperty ;
   rdfs:domain :Person ;
   rdfs:range :Sex .

:Male a owl:Class ;
   rdfs:subClassOf :Sex ;
   owl:disjointWith :Female .

:Female a owl:Class ;
   rdfs:subClassOf :Sex ;
   owl:disjointWith :Male .

:Sex a owl:Class ;
   owl:unionOf _: rdf:first: Female .
                        rdf:seconde :Male ;
  rdfs:subClassOf :DomainEntity .

:Person a owl:Class ;
   rdfs:subClassOf _:Class .
   owl:intersectionOf_: rdf:first: DomainEntity .
   owl:Restriction ;
     owl:onProperty :hasSex ;
   owl:someValuesFrom :Sex .
```

Capítulo 3

Descrição do Problema

Como referido anteriormente, com o avançar da tecnologia e, consequentemente, da propagação de dados leva a que esta informação seja descrita em mais que uma linguagem /estrutura, sendo as mais comuns JSON, XML e OWL. [6] Com isto, nesta área, existe a necessidade de transformar dados de um tipo para outro (por exemplo, de XML para OWL ou de JSON para OWL).

3.1 Ferramentas já existentes

Com o crescimento da popularidade de OWL para a descrição de dados, apareceram naturalmente também alguns *softwares* para a edição e manipulação de OWL. Nesta secção iremos falar sobre alguns destes programas e sobre as diferenças entre eles. Iremos comparar 3 programas que têm um maior número de utilizadores: Apollo [7], Protégé [3] e Swoop [8].

3.1.1 Apollo

Apollo é um modelador *user-friendly*, de utilização gratuita e de instalação local. Nesta aplicação, um utilizador pode modelar ontologias com noções básicas de ontologias (classes, relações, instâncias, etc). Também é possível criar novas instâncias a partir das classes presentes nessas ontologias. Alguns dos pontos fortes deste software é o seu validador de tipos que mantém a consistência de tipos durante o processo, bem como o armazenamento das ontologias (em ficheiros). Contudo, este programa é relativamente antigo e carece de uma visualização de dados em grafo, ao contrário dos seus concorrentes. Apollo carece da possibilidade de dar uma experiência multiutilizador aos seus utilizadores, para trabalhos em colaboração com mais do que um utilizador. [9] [10]

3.1.2 Protégé

Protégé é um editor e modelador de ontologias, de utilização gratuita e tem uma vertente local bem como online (WebProtégé) [11]. Tem uma arquitetura baseada em *plug-ins* o que deu origem ao desenvolvimento de inúmeras ferramentas relacionadas com semântica web. Implementa um conjunto de estruturas modeladoras de conhecimento e ações que suportam a criação, modelação e manipulação de ontologias, complementadas com inúmeras formas de visualização desses dados. A customização

proporcionada aos seus utilizadores é uma das características que torna esta aplicação numa das mais populares na área. [9] [10]

3.1.3 Swoop

Swoop é um browser e também um editor open-source, de utilização gratuita e local. Esta ferramenta contém validadores de OWL e oferece várias formas de visualização gráfica de ficheiros em OWL. É composto também por um ambiente de edição, comparação e fusão entre múltiplas ontologias. As suas capacidades de hyperlinks proporciona uma interface de navegação fácil aos seus utilizadores. Um utilizador pode também reutilizar dados ontológicos externos ao colocar os links para a entidade externa ou importando a ontologia completa, pois não é possível importar ontologias parciais, mas é possível realizar pesquisas por múltiplas ontologias. [9] [10]

Durante a nossa pesquisa sobre outras ferramentas que já existem nesta área, encontrámos também, para além de outros editores com características diferentes dos apresentados acima, algumas bibliotecas que mapeiam XML para OWL - Ontmalizer[1] e JXML2OWL[2]. Estas bibliotecas foram desenvolvidas em Java e estão disponíveis para *developers* usarem também em Java.
Contudo, apesar da nossa pesquisa, não encontramos nenhuma aplicação que oferecesse todos os pontos mais relevantes das aplicações descritas acima, tais como: versão online como local, mapear XML para OWL[3], etc.

3.2 Como HOMI se compara com estas ferramentas

A nossa ferramenta visa juntar os pontos fortes destas aplicações numa só aplicação. Na nossa ferramenta (H.O.M.I.) iremos ter uma vertente local assim como uma vertente *online*. Ambas terão uma visualização simples, fácil e intuitiva, ou seja, que funcione da forma que o utilizador espera que funcione, bem como a opção de apenas criar uma nova instância a partir de uma dada ontologia ou então, através de ficheiro semiestruturado descrito em XML, realizar o mapeamento de conceitos presentes na ontologia em questão.

Característica - Programa	Criação de novas instâncias	Fácil Utilização	Armazenamento	Interface Intuitiva	Versão Online	Mapeia XML para OWL
Apollo	Sim	Sim	Sim, em	Não	Não	Não

[1] https://github.com/srdc/ontmalizer

[2] http://jxml2owl.projects.semwebcentral.org/index.html

[3] Mapear XML para OWL refere-se ao processo de realizar uma transformação de uma representação em XML para um documento OWL válido, através da associação de *tags* presentes em XML com conceitos de OWL.

			ficheiros			
Protégé	Sim	Sim	Sim	Sim	Sim	Não
Swoop	Sim	Sim	Sim, em modelos HTML	Sim	Não	Não
Ontomalizer & JXML2OWL	Não	Sim	Não	Não contém interface	Não	Sim
HOMI	Sim	Sim	Sim	Sim	Sim	Sim

Tabela 0.1 - Comparação entre várias ferramentas e a nossa aplicação

Capítulo 4

Chaos Pop

Para implementar o projeto descrito anteriormente, seria necessária uma ferramenta que fosse capaz de mapear ficheiro de dados semiestruturados para ontologias, com base nos seus termos e conceitos, respetivamente. A criação de tal ferramenta seria ligeiramente complexa e demorada, o que por si só poderia ser considerado como um projeto final de curso. No entanto, nos foi apresentada a biblioteca do *Chaos Pop* para que pudéssemos utilizá-la.

O *Chaos Pop* é uma biblioteca que fornece um serviço que permite mapear ficheiros semiestruturados de acordo com a descrição de uma ontologia com base em pelo menos dois ficheiros: um OWL referente à uma ontologia (`OntologyFile`) e outro que representa um caso concreto da ontologia em questão (`DataFile`). Neste momento, os ficheiros que são suportados como `DataFile` são de extensões JSON e XML.

Uma vez não usufruímos de todas as funcionalidades disponíveis no serviço fornecido pelo *Chaos Pop,* neste capítulo apenas realizamos uma breve descrição de algumas das estruturadas de dados e dos *endpoints* existentes, assim como um pequeno exemplo de utilização.

4.1 Estruturas de dados

O *Chaos Pop* armazena a informação numa base de dados remota que contém algumas estruturadas. Dentre as estruturas de dados, existem algumas que é essencial ter conhecimento sobre quais dados armazenam e o que representam para que possa usufruir das funcionalidades do serviço fornecido. São elas:

- *DataFile:* representa um ficheiro de dados semiestruturados e armazena a informação referente a este na forma de *nodes*. Se o formato do ficheiro for XML, cada *node* representa um *child element* do *parent*, caso JSON cada *node* representa os objetos presente no ficheiro.
- *OntologyFile:* representa um ficheiro de ontologia, guardando a informação sobre todas as *classes, data properties* e *object properties* presentes na ontologia em questão.
- *IndividualMapping*: representa o mapeamento de um determinado termo presente em `DataFile` para uma classe especifica presente em `OntologyFile.`

- *Mapping*: representa o mapeamento de um ou mais `DataFile's` para um `OntologyFile.` Este é responsável pela criação do ficheiro de *output* e por isso aglomera as informações necessárias para tal, tais como o nome do ficheiro de *output* bem como os identificadores dos *IndividualMapping's* que irão ser utilizados para realizar este mapeamento.
- *Batch*: utilizado quando se deseja popular uma ontologia com base nos mapeamentos realizados e gerar o ficheiro de *output* e armazena os *Mappin's* nas quais deseja executar a fim de popular a ontologia. Para cada *Mapping* é gerado um ficheiro de *output.*

4.2 API

De modo a usufruir das funcionalidades existentes no *Chaos Pop*, seja para submeter ficheiros, obter dados referentes aos ficheiros submetidos ou mapear os vários conceitos é necessário ter conhecimento dos *endpoints* existentes. Todos os *endpoints* apresentados abaixo iniciam com a URL do servidor do *Chaos Pop*: http://chaospop.sysresearch.org/chaos/wsapi

POST	/createIndividualMapping	body - individualMappingTO: Object
POST	/replaceIndividualMapping	body - individualMappingTO: Object
POST	/removeIndividualMapping	body - ids: String
GET	/getAllIndividualMappings	
POST	/addFile	body - file: InputStream
POST	/getFile	body - id: String
POST	/removeFile	body - ids: String
GET	/listDataFiles	
POST	/createMapping	body - mappingTO: Object
POST	/removeMapping	body - ids: String
POST	/getAllNodesFromDataFile	body - id: String
POST	/getNode	body - id: String
POST	/getOntologyFile	body - id: String
POST	/removeOntologyFiles	body - ontologyIds: String
POST	/getOWLClasses	body - ontologyId: String
POST	/getObjectProperties	body - ontologyId: String
POST	/getDataProperties	body - ontologyId: String
GET	/listOntologyFiles	
POST	/createBatch	body - jsonBatch: String
POST	/removeBatch	body - batchIds: String

4.3 Exemplo de utilização

Quando se pretende realizar o mapeamento de dados, é necessário primeiro obter as estruturadas de dados que representam os ficheiros que se tenciona mapear, sejam eles um ou mais `DataFile's` e `OntologyFile's`.

Por exemplo, se for submetido o ficheiro XML da Figura 4.2, será gerado uma lista de *nodes* denominada de *nodesTO*.

```
<family>
    <member>
        <name>
            <given>João Antero</given>
            <surname>Cardoso</surname>
        </name>
        <sex>male</sex>
        <birth_year>1960</birth_year>
        <marriage>
            <marriage_year>1986</marriage_year>
            <partner_name>Maria Goretti</partner_name>
        </marriage>
    </member>
    <member>
        <name>
            <given>Maria Goretti</given>
            <surname>Fernandes</surname>
        </name>
        <sex>female</sex>
        <birth_year>1960</birth_year>
        <marriage>
            <marriage_year>1986</marriage_year>
            <partner_name>João Antero Cardoso</partner_name>
        </marriage>
    </member>
</family>
```

Figura 0.1 - Exemplo de ficheiro de dados semiestruturados

Abaixo é possível observar um dos objetos pertencentes ao *nodesTO* gerados quando se inseriu o ficheiro mencionado em Figura 4.2.


```
"nodesTO": [
        {
            "childrenIDs": {
                "childID": [
                  "5b019a9c4f0caa89a2acf961",
                  "5b019a9c4f0caa89a2acf96a"
                ]
            },
            "_id": "5b019a9c4f0caa89a2acf960",
            "dataFileId": "5b019a9c4f0caa89a2acf960",
            "hasAttributes": false,
            "tag": "family"
```

```
        },
        ...
]
```

Num segundo momento, o utilizador deve realizar as correspondências dos vários termos existentes no ficheiro semiestruturado aos conceitos da ontologia. Alguns dos mapeamentos a serem realizados são:

Semiestruturado termos	Ontologia conceitos
member	Person
given	hasFirsGivenName
surname	hasFamilyName
partner_name	hasPartner

Tabela 0.1 - Alguns mapeamentos do ficheiro semiestruturado para ontologia

Após realizar todos os mapeamentos do ficheiro semiestruturado, irá ser gerado o ficheiro do caso concreto da ontologia. Abaixo está apresentado o ficheiro gerado no formato *turtle*.

```
@prefix : <http://sysresearch.org/ontologies/cardosofamily.owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix family: <http://sysresearch.org/ontologies/family.owl#> .

<http://sysresearch.org/ontologies/cardosofamily.owl#> a owl:Ontology ;
   owl:imports <http://sysresearch.org/ontologies/family.owl#> .

<http://sysresearch.org/ontologies/family#Person> a owl:Class .

:JoãoAnteroCardoso a owl:NamedIndividual ,
<http://sysresearch.org/ontologies/family#Person> ;
   family:hasPartner :MariaGorettiFernandes ;
   family:hasFamilyName "Cardoso" ;
   family:hasFirstGivenName "João Antero" .

:MariaGorettiFernandes a owl:NamedIndividual ,
<http://sysresearch.org/ontologies/family#Person> ;
   family:hasPartner :JoãoAnteroCardoso ;
   family:hasFamilyName "Fernandes" ;
   family:hasFirstGivenName "Maria Goretti" .
```

Capítulo 5

Arquitetura do H.O.M.I

Neste capítulo apresentamos a arquitetura da aplicação *web*, descrevemos qual a funcionalidade de cada módulo presente nesta, assim como a forma como estes interagem entre si e as tecnologias utilizadas na implementação da aplicação no geral.

5.1 Descrição

A Figura 5.1 mostra a arquitetura da aplicação *web*, na qual está representada a interação entre todas as camadas existentes. Existem quatro camadas de acesso:

1. Views
2. Routes
3. Services
4. Data access

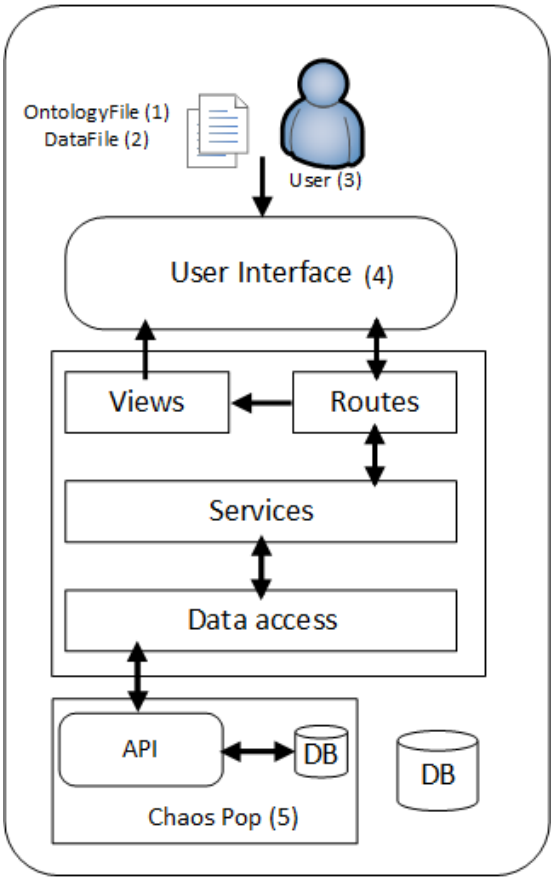

Figura 0.1 - Arquitetura da aplicação

A interação inicia-se quando o utilizador (3) insere um ou mais ficheiros com a definição de uma ontologia (1) e, opcionalmente, um ou mais ficheiros de dados semiestruturados (2). Estes ficheiros irão ser submetidos à API *Chaos Pop* (5). De seguida irá ser gerada uma interface gráfica (4) onde o utilizador poderá anotar novos dados aos vários conceitos presentes nos ficheiros que descrevem uma ontologia (1) ou mapear os termos dos ficheiros semiestruturados (2) com os conceitos dos ficheiros da ontologia (1). No final deste processo, é gerado um novo ficheiro OWL que contém um caso concreto dos dados descritos nos `OntologyFile's`. Iremos também dar a opção ao utilizador de manter os ficheiros de *input* e *output* numa base de dados remota.

5.2 Componentes

A aplicação foi implementada por camadas (Figura 0.1), na qual cada camada apenas comunica com a superior ou inferior. Esta implementação proporciona uma maior facilidade na manipulação das funcionalidades. As camadas existentes são:

- ***controllers:*** contém todos os *endpoints* possíveis de serem acedidos através da *user interface* e comunica com *views* e *services*;
- ***services:*** camada intermédia entre *controllers* e *data access*, contém toda a lógica necessária para a execução das operações disponíveis;
- ***data access:*** engloba todo o acesso aos dados. Seja este acesso na API *Chaos Pop* ou na nossa própria base de dados;
- ***views:*** inclui todas as representações visuais utilizadas em *user interface.*

5.3 Tecnologias

Em cada uma das camadas existem na arquitetura apresentada anteriormente existem diferentes módulos, nestes estão implementadas funções usufruindo de algumas tecnologias. Todas as tecnologias utilizadas foram previamente escolhidas tendo em conta as funcionalidades de que cada uma delas disponibilizam assim como o objetivo final do projeto.

5.3.1 Node.js

Uma plataforma de software que é usada para construir aplicativos de redes escaláveis. O Node.js usa o JavaScript como linguagem de *script* e alcança rendimentos altos através de um *loop* de eventos, na qual interpreta, uma única *thread*, as requisições de forma assíncrona em vez de sequenciais e não permitindo bloqueios [14].

Já tivemos experiências em outras unidades curriculares com esta tecnologia e concluímos que fornece uma maneira fácil de construir uma aplicação *web*, assim como descobrimos a existência de outras tecnologias que utilizam esta para criar aplicações *desktop*, o que nos facilitaria no desenvolvimento do projeto e optamos por utilizá-la.

5.3.2 Express

Equivale a um *framework* rápido, flexível e minimalista para Node.js. *Express* fornece um conjunto robusto de recursos para desenvolver aplicações *web*, assim como um grande número de métodos utilitários HTTP e *middleware* para criar uma API rápido e fácil [15].

Por já termos experiências com esta tecnologia e pelo facto da sua utilização ser fácil, decidimos implementar a nossa aplicação *web* e todas *endpoints* disponíveis com base nesta tecnologia.

5.3.3 D3.js

Consiste numa biblioteca JavaScript que permite vincular dados arbitrários ao *Document Object Model (DOM)* e em seguida aplicar transformações nos dados usando diferentes padrões *web* como HTML, SVG e CSS [16].

Decidimos optar por tal pelo facto de existirem vários exemplos de utilização na documentação da mesma que podiam ser adaptados para a nossa necessidade de obter de uma forma simples e prática uma representação em árvore dos dados referentes ao ficheiro de entrada `DataFile`.

5.3.4 Electron

É uma biblioteca *open source* utilizada para criar aplicações *desktop* multiplataformas com tecnologias *web* (HTML, JavaScript e CSS). *Electron* combina *Chromium* e *Node.js* em um único tempo de execução e as aplicações podem ser empacotadas das Mac, Windows e Linux [17].

Uma vez que a nossa aplicação foi projetada para ter duas vertentes, uma *web* e outra *desktop*, tínhamos de optar por alguma tecnologia que nos permitisse aproveitar o máximo possível do código desenvolvido para a aplicação *web* na criação da versão *desktop.* O que o *Electron* nos proporciona é exatamente isso, reutilizar o que já tem implementado em JavaScript sem a necessidade de aprender a usar uma nova tecnologia no desenvolvimento da aplicação *desktop*.

5.3.5 MongoDb

MongoDB é um banco de dados que armazena dados em documentos flexíveis semelhantes a JSON, o que significa que os campos de cada documento podem variar de um para outro e a estrutura de dados pode ser alterada ao longo do tempo [18].

Apesar de podemos desfrutar da base de dados do *Chaos Pop,* todos os acessos a essa teriam um grande custo em tempo de execução decorrente das várias operações que são necessárias realizar. Por isso decidimos criar a nossa própria base de dados.

Pelo facto do MongoDb permitir ser flexível na criação dos documentos e a execução de *queries* para obtê-los, optamos por usá-lo na como base de dados.

Capítulo 6

Implementação

Neste capítulo descrevemos todas estruturas de dados existentes na base de dados documental e o seu respetivo intuito, os *endpoints* presentes na API que permite a comunicação entre *client-side* e *server-side,* a organização do código em vários packages e a interação com o utilizador na aplicação *web.*

6.1 Estruturas de dados

A nossa base de dados é documental e está dividida em quatro *collections*, onde cada uma armazena dados estruturados de forma distinta. São as *collections:*

1. DataFiles
2. OntologyFiles
3. IndividualMappings
4. Populates

Cada documento presente nestas *collections* contém o campo `_id` que é o identificador único gerado pelo *MongoDb*, para além de outros campos que são descritos abaixo.

6.1.1 Data Files

Esta *collection* armazena dados referentes aos ficheiros de *input* `DataFile`. A constituição de um ficheiro presente nesta é:

```
{
      _id: ObjectId,
      name: String,
      chaosid: String,
      nodes: Array
}
```

- `name:` nome do ficheiro;
- `chaosid`: identificador no ficheiro do servidor do *Chaos Pop*;
- `nodes`: contém todos os nodes que constituem a árvore representativa dos dados.

6.1.2 Ontology Files

Esta *collection* armazena dados referentes aos `OntologyFile's`. A constituição de um ficheiro presente nesta é:

```
{
      _id: ObjectId,
      name: String,
      chaosid: String,
      classes: Array,
      dataProperties: Array,
      objectProperties: Array
}
```

- `name:` nome do ficheiro;
- `chaosid`: identificador no ficheiro do servidor do *Chaos Pop*;
- `classes`: classes que pertencem ficheiro de ontologia;
- `dataProperties`: propriedades do tipo *data* que pertencem ao ficheiro de ontologia;
- `objectProperties`: propriedades do tipo *object* que pertencem ao ficheiro de ontologia.

6.1.3 Individual Mappings

Esta *collection* contém dados que representam o mapeamento de um determinado indivíduo e de suas propriedades, entre um `DataFile` e um `OntologyFile`, que são denominados de *IndividualMapping*. Também é utilizada para armazenar a criação de indivíduos apenas a partir de uma `OntologyFile`. O nosso *IndividualMapping* é semelhante ao utilizado pelo *Chaos Pop* e tem a seguinte estrutura:

```
{
      _id: ObjectId,
      tag: String,
      nodeId: String,
      owlClassIRI: String,
      ontologyFileId: String,
      dataFileId: String,
      specification: Boolean,
      objectProperties: Array,
      dataProperties: Array,
      annotationProperties: Array
}
```

- `tag`: *tag* do *node* presente no `DataFile` que estamos a mapear;
- `nodeId`: id do *Chaos Pop* correspondente ao *node* do `DataFile` que estamos a mapear;
- `owlClassIRI`: IRI da classe que estamos a mapear, contida no `OntologyFile`;

- `ontologyFileId`: id presente na *collection OntologyFiles* correspondente ao `OntologyFile` que queremos mapear;
- `dataFileId`: id presente na *collection DataFiles* correspondente ao `DataFile` que queremos mapear,
- `specification`: uma *flag* que indica se este *IndividualMapping* irá alterar algum outro já existente;
- `individualName`: metadados que indicam qual a propriedade de dados que irá ser utilizada para o nome deste indivíduo;
- `objectProperties`: cada elemento deste *array* é composto por: IRI da propriedade no `OntologyFile` e metadados dos *nodes* (no `DataFile`) envolvidos nesta propriedade;
- `dataProperties`: cada elemento deste *array* é composto por: IRI da propriedade no `OntologyFile`, um *pair* composto pelos metadados dos *nodes* (no `DataFile`) envolvidos nesta propriedade, bem como o tipo da mesma (*string, integer, etc*);
- `annotationProperties`: cada elemento deste *array* é composto por: tipo de anotação (*label*, *comment*, etc) e metadados dos *nodes* (no `DataFile`) envolvidos nesta propriedade.

No geral, a constituição das estruturas presentes nesta *collection* é a mencionada acima. Porém quando o *IndividualMapping* se diz respeito a um mapeamento e não a criação de um individuo apenas, este contém mais um campo denominado `indTree`, que representa a subárvore extraído do `DataFile` correspondente ao *IndividualMapping* em questão.

6.1.4 Populates

Nesta *collection* estão presentes estruturas que podem ser consideradas como uma sessão, uma vez que em cada uma delas estão presentes um aglomerado de identificadores indispensáveis em toda a interação com o utilizador. As estruturas podem ser usadas numa população de ontologia com (*populate with data*) ou sem dados semiestrurados(*populate without data*). Os campos em comum nos dois casos são:

```
{
      _id: ObjectId,
      ontologyFiles: Array,
      indMappings: Array,
      chaosid: String,
      batchId: String,
      outputFileId: String
}
```

- `ontologyFiles`: *array* com dados sobre todos os ficheiros de ontologia que deseja popular;
- `indMappings`: *array* de *IndividualMappings* pertencentes ao mapeamento;

- `chaosid`: identificador do servidor do *Chaos Pop* referente ao *Mapping;*
- `batchId`: identificador do servidor do *Chaos Pop* referente ao *Batch*;
- `outputFileId`: identificador do servidor do *Chaos Pop* referente ao ficheiro de *output,* quando esse é gerado.

Quando se trata de um *populate with data*, além da descrição acima também contém o campo `dataFiles` que representa todos os ficheiros de dados semiestruturados envolvidos no mapeamento.

6.2 Endpoints

A comunicação entre o *client-side* e o *server-side* é feita através de pedidos HTTP. Desta forma, o servidor disponibiliza uma lista de *endpoints* para facilitar esta comunicação. Estes *endpoints* estão definidos nos *routes*, agrupados pela sua finalidade. Por exemplo, todos os *endpoints* relacionados com *Populate's* poderão ser encontrados no populates-routes.

file-routes:

POST	/dataFile	body: File
POST	/ontologyFile	body: File
GET	/dataFile	
GET	/ontologyFile	
DELETE	/dataFile/:id	
DELETE	/ontologyFile/:id	

index-routes:

GET	/	

individual-mapping-routes:

POST	/map/individual	body - individualMapping: Object
PUT	/map/individual/:id	body – individualMapping: Object
PUT	/map/individual/:id/name	body – individualName: Array
PUT	/map/individual/:id/properties/annotation	body - annotationProps: Array
PUT	/map/individual/:id/properties/data	body - dataProps: Array
PUT	/map/individual/:id/properties/object	body - objectrops: Array
GET	/map/individual/:id/properties/annotation/view	
GET	/map/individual/:id/properties/data/view	
GET	/map/individual/:id/properties/object/view	
DELETE	/map/individual/:id	
DELETE	/map/individual/:id/properties/:pid	

individual-routes:

POST	/individual	body - individual: Object
PUT	/individual/:id/properties/annotation	body - annotationProps: Array
PUT	/individual/:id/properties/data	body - dataProps: Array
PUT	/individual/:id/properties/object	body - objectProps: Array
GET	/individual/:id/properties/annotation/view	
GET	/individual/:id/properties/data/view	
GET	/individual/:id/properties/object/view	
DELETE	/individual/:id	
DELETE	/individual/:id/properties/:pid	

maping-routes:

POST	/map	body - mapping: Object
GET	/map/:id	
DELETE	/map/:id	

populate-routes:

POST	/populate	body – populate: Object
PUT	/populate/:id/:output	
GET	/populate	
DELETE	/populate/:id	
GET	/populate/data/:id	
GET	/populate/data/:id/tree	
GET	/populate/data/:id/mapping	
GET	/populate/data/:id/individual/:ind	
GET	/populate/data/:id/individual/:ind/tree	
PUT	/populate/nondate/:id/finalize	body: indMappings: List
GET	/populate/nondata/:id	
GET	/populate/nondata/:id/mapping	
GET	/populate/nondata/:id/individual/:ind	

6.3 Organização do código

Como mencionado anteriormente, o servidor está dividido em 3 camadas: *routes*, *services* e *data-access*. Apresentamos agora um esquema mais completo que indica as relações entre estas camadas, bem como os módulos presentes em cada uma delas:

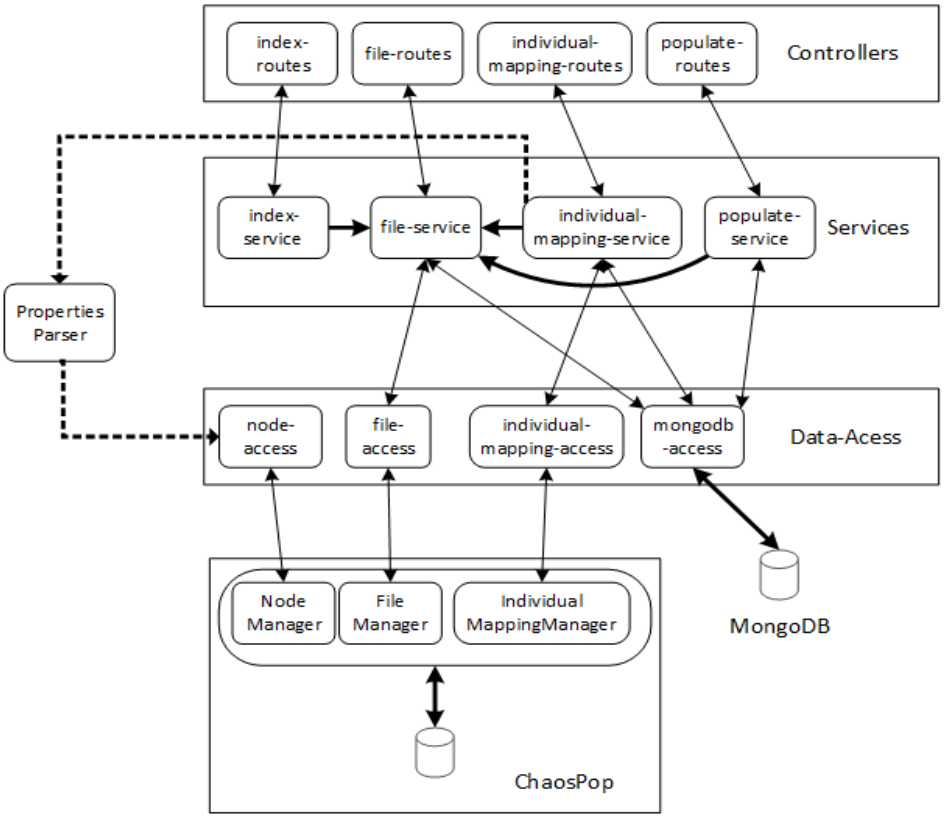

Figura 0.1 - Camadas de acesso e seus respetivos módulos

Com isto garantimos que toda a lógica da aplicação está nos *services*. Cada *controller* apenas comunica com o seu serviço e os serviços comunicam entre si bem como com vários *data-access* dependendo da ação a ser realizada. Os *controllers* apenas respondem a pedidos HTTP, retiram a informação necessária dos mesmos (caso se trate de um POST, PUT ou DELETE). De seguida é chamado o *service* correspondente para processar aquele pedido que, por sua vez pode precisar de outros *services* e de um ou vários *data-access*.

Um exemplo, quando é chamado o *endpoint /home*, este está localizado no *index-routes*. Para realizar o *render* da página inicial é necessário ir buscar os ficheiros já existentes para o utilizador ter a opção de selecionar um ficheiro que já submeteu anteriormente. Então o *index-router* irá comunicar com o *index-service* que por sua vez irá precisar do *file-service*. O *file-service* é responsável por processar toda a informação relacionada com ficheiros, então será utilizado o *file-access* para ir buscar os ficheiros já existentes, que os devolve ao *index-service*, que responde ao *index-routes* e assim este já pode renderizar a página inicial com toda a informação necessária.

A comunicação com a base de dados é feita apenas na camada *data-access*, no *mongodb-access*, através do *package mongodb* do npm, onde a interação é feita através de um

objeto *MongoClient*, através do nome da base de dados e da *collection* que pretendemos aceder.

A comunicação com o *Chaos Pop* é feita através dos restantes módulos presentes em *data-access* (*Node*, *File* e *IndividualMapping*), sendo que cada um destes *data-access* comunica com o seu *Manager* correspondente no *Chaos Pop*, com excepção do *file-access* que comunica com *FileManager* e *OntologyManager* pois achamos que não valia a pena separar esse acesso do nosso lado.

PropertyParser e ListToTree

Em *utils* criamos um *parser* para transformar as propriedades numa metalinguagem definida pelo *Chaos Pop* e que o mesmo consiga interpretar. Desta forma, o que é enviado do *front end* quando o utilizador quer mapear uma propriedade é o sempre uma *key* (IRI ou anotação) e um conjunto de nós que irá ser utilizado para mapear essa propriedade (adicionalmente é também enviado o tipo, caso se trate de uma *data property*). Estas informações são enviadas ao *PropertyParser* pelo *individual-mapping-service*, bem como o *nodeId* do *IndividualMapping* em questão. A ideia deste *parser* é escrever o caminho desde o nó que está presente no *IndividualMapping* (doravante denominado de `indMapId`) até ao nó ou nós presentes na propriedade a ser mapeada. Abaixo apresentamos um exemplo, onde os objetos são a informação enviada:

Object Properties:
```
{
    owlIRI: '#hasChild',
    toMapNodeId: '147'
}
```

DataProperties:
```
{
   owlIRI:
'#hasFirstGivenName',
    toMapNodeIds:
['123','456'],
    type: 'String'}
```

AnnotationProps :
```
{
    annotation: 'label',
toMapNodeIds: ['118','748']
 }
```

Bem como um *IndividualMapping*, onde é necessário o *nodeId* deste objeto pois indica o início do *path*. O *Parser* irá utilizar o *toMapNodeId* fornecido para ir buscar o nó ao *Chaos Pop* e irá comparar o *parent* desse *node* com o *nodeId* do *IndividualMapping*. Caso seja igual, retorna a *string* que descreve esses metadados, caso contrário irá buscar o *parentNode* do nó que analisamos. Irá realizar esta pesquisa até que o nó retornado seja o do *IndividualMapping*. No fim, o *IndividualMapping* terá o seguinte aspecto (os dados a *bold* são os metadados gerados pelo *parser*):

```
Individual Mapping : {
    tag : 'member',
    dataFileIds : '169',
    individualName : '.inspecificchild-name-given;.inspecificchild-name-surname',
    owlClassIRI : '#Person',
    specification : false,
    annotationProperties : {
        label : '.inspecificchild-name-nickname'
    },
    objectProperties : {
        #hasChild : '.inspecificchild-descendants-son'
    },
    dataProperties : {
```

```
        #hasFirstGivenName :  ['.inspecificchild-name-given','String']
    }
}
```

Uma das funcionalidades do *Chaos Pop*, como mencionada no Capítulo 4, é a habilidade desta ferramenta transformar um `DataFile` em *nodes*. Por sua vez, conseguimos aceder a estes nodes através do *endpoint* /getAllNodesFromDataFile. Este *endpoint* retorna todos os *nodes* existentes com aquele *fileId* e seus respectivos *children*. Para facilitar a visualização para o utilizador, criamos um *parser* (List-To-Tree) que torna esta lista de *nodes* numa árvore n-ària.

6.4 Interação com o utilizador

Referências

[1] "Ontology," [Online]. Available: https://www.w3.org/standards/semanticweb/ontology. [Acedido em 19 03 2018].

[2] "OWL," [Online]. Available: https://www.w3.org/OWL/. [Acedido em 09 03 2018].

[3] "Protege," [Online]. Available: https://protege.stanford.edu/. [Acedido em 15 03 2018].

[4] "Ontologias," [Online]. Available: https://pt.wikipedia.org/wiki/Ontologia_(ci%C3%AAncia_da_computa%C3%A7%C3%A3o). [Acedido em 22 04 2018].

[5] "OWL-ref," [Online]. Available: https://www.w3.org/TR/owl-ref/. [Acedido em 24 04 2018].

[6] "Meta Formats," [Online]. Available: https://www.w3.org/standards/webarch/metaformats. [Acedido em 26 04 2018].

[7] "Apollo," [Online]. Available: http://apollo.open.ac.uk/. [Acedido em 22 04 2018].

[8] "Swoop," [Online]. Available: https://github.com/ronwalf/swoop. [Acedido em 23 04 2018].

[9] "Editores 1," [Online]. Available: https://www.w3.org/wiki/Ontology_editors. [Acedido em 21 04 2018].

[10] "Artigo sobre editores de ontologias," [Online]. Available: http://www.ef.uns.ac.rs/mis/archive-pdf/2013%20-%20No2/MIS2013-2-4.pdf. [Acedido em 21 04 2018].

[11] "WebProtege," [Online]. Available: https://protege.stanford.edu/products.php#web-protege. [Acedido em 23 04 2018].

[12] "Editores 2," [Online]. Available: https://www.w3.org/wiki/SemanticWebTools#Semantic_Web_Development_Tools: _Introduction. [Acedido em 21 04 2018].

[13] "Turtle", [Online]. Available: https://www.w3.org/TR/turtle/. [Acedido em 27 05 2018]

Jamie Taylor, Colin Evans, Toby Segaran. (2009). Programming the Semantic Web.

] Jim R. Wilson. (2013). Node.js the Right Way: Practical, Server-side JavaScript that Scales.

[14] https://nodejs.org/en/about/

[15] http://expressjs.com/

[16] https://d3js.org/

[17] https://electronjs.org/

[18] https://www.mongodb.com

Logo feito por: Joana Antunes (joanasantosantunes@hotmail.com)