INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

HOMI

Hybrid Ontology Mapping Interface

Projeto Final de Curso

Licenciatura em Engenharia Informática e Computadores

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

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

Relatório Versão Beta

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

Abril de 2018

Índice

Lista de Figuras

Lista de Tabelas

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.

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" denominada "temPai" e com o individuo "Maria" denominada de "". Ambos estes indivíduos são do tipo "Person"

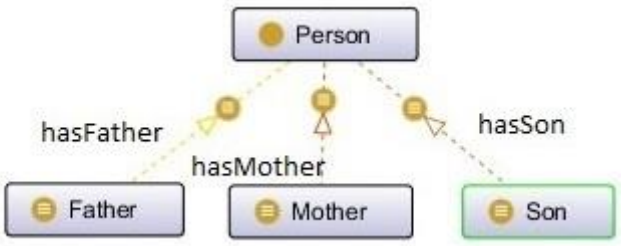

Figura 2.1 - Exemplo de uma ontologia

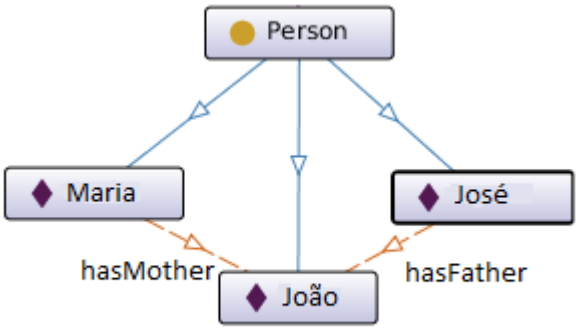

Figura 2.2 - Exemplo de uma instância de uma ontologia

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 carnalidades 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 .
```

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 ficheiros	Não	Não	Não
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 3.1 - Comparação entre várias ferramentas e a nossa aplicação

[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 OXL.

4 Chaos Pop

Chaos Pop é uma biblioteca que fornece um serviço que permite permitir mapear ficheiros semiestruturados de acordo com a descrição de uma ontologia. Para que isto seja possível são indispensáveis 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.

Após a submissão destes ficheiros é possível criar *IndividualMapping's*. Um *IndividualMapping* é um objeto que mapeia um indivíduo específico enquanto que um *Mapping* contém as informações sobre a ontologia e o ficheiro de dados utilizados no mapeamento. Este objeto *Mapping* 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.

Desta forma, o *Chaos Pop* torna-se uma base de dados de mapeamentos já feitos, prontos a ser utilizados pelos seus utilizadores de forma a que estes possam reutilizar *Mapping's* e *IndividualMapping's* já realizado por outros.

4.1 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	

4.2 Armazenamento de dados

Quando são submetidos os ficheiros `DataFile` e `OntologyFile`, a informação presente nestes é guardada numa base de dados remota.

No que diz respeito ao `DataFile` os dados são guardados numa estrutura interna em formato de nodes, onde cada node pode conter children e parent, formando assim uma árvore.

Por outro lado, quando é submetido um `OntologyFile`, a informação que está contida neste é mantida em base de dados da seguinte forma:

- Uma ontologia pode ter *classes, data properties* e *object properties*, sendo assim é possível obter estes dados referentes a uma ontologia com o identificador da mesma;
- Uma *class* pode ter *data properties* e *object properties,* logo é possível aceder a esta informação com base no identificador da ontologia na qual a classe pertence e no identificador da classe em questão.

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* (*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 4.1 - Exemplo de ficheiro de dados semiestruturados

Abaixo é possível observar um dos objetos 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 (capitulo 2.2). Alguns dos mapeamentos a serem realizados são:

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

Tabela 4.1 - 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" .
```

5 Arquitetura do H.O.M.I

Neste capítulo descreveremos detalhadamente cada módulo da aplicação, assim como a forma como estes interagem entre si e as tecnologias utilizadas na implementação de cada um.

5.1 Descrição

A Figura 5.1 mostra a arquitetura da aplicação *web*, na qual está representada a interação entre as camadas existentes. Esta 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 conceitos dos ficheiros semiestruturados (2) com os termos 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 guardar os ficheiros de *input* e *output* numa base de dados remota.

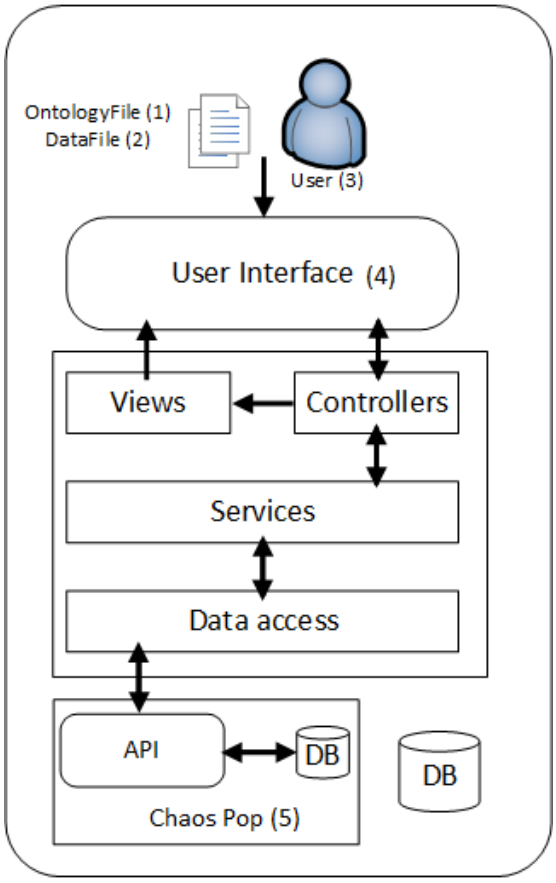

Figura 5.1 - Arquitetura da aplicação

5.2 Componentes

A aplicação foi implementada por camadas (Figura 5.1), 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 diferentes módulos, nestes estão implementadas funções usufruindo de algumas tecnologias das quais temos conhecimento.

5.3.1 Node.js

É um *runtime* de JavaScript, que pelo facto de processar o código JavaScript desvinculando-o do browser, possibilita o desenvolvimento de aplicações estáveis e rápidas.

Uma vez que já tivemos experiências em outras unidades curriculares com esta tecnologia e concluímos que fornece uma maneira fácil de construir uma aplicação, escolhemos esta para desenvolver todo o projeto.

5.3.2 D3.js

Consiste numa ferramenta para JavaScript que associa os dados ao *Document Object Model (DOM)* e permite manipular estes gerando gráficos usando diretamente padrões *web* como o HTML e o CSS.

Decidimos optar por tal pelo facto de suportar comportamentos dinâmicos de interação e animação e grandes conjuntos de dados. Outra característica que nos chamou a atenção foi com facilidade obter gráficos bonitos visualmente. Por isto, utilizamos esta na apresentação dos dados referentes aos ficheiros de entrada, de modo a obter uma representação intuitiva e de fácil entendimento destes por parte do utilizador.

5.3.3 Electron

É um *framework* utilizado para criar aplicações multiplataformas *desktop* com tecnologias *web* (HTML, JavaScript e CSS).

Como a nossa aplicação será desenvolvida na tecnologia JavaScript e esta é a suportada pelo Electron, decidimos utilizá-lo para assim não ter a necessidade de aprender a usar uma nova tecnologia no desenvolvimento do sistema *desktop*.

5.3.4 Express

Equivale a um *framework back-end* em Node.js que cria rotas, *middlewares*, entre outras para facilitar a criação de API's. Este cria e obtém dados a partir do servidor, independentemente da linguagem que irá utilizá-los.

Por já termos experiências com esta tecnologia e pelo facto da sua utilização ser fácil, decidimos implementar todas as rotas disponíveis na camada *controllers* com base nesta.

5.3.5 MongoDb

MongoDB é um banco de dados orientado a documentos que armazena dados em documentos JSON com esquema dinâmico.

Por sua facilidade na obtenção dos documentos existentes, assim como na criação de *queries* para obtê-los, decidimos utilizar este como base de dados.

6 Implementação

6.1 Estruturas de dados

Na nossa aplicação há algumas estruturas de dados relevantes que são armazenadas na base de dados, são elas: *IndividualMapping*, *Populate*, *DataFile* e *OntologyFile*.

6.1.1 Individual Mapping

Esta estrutura de dados representa o mapeamento de um determinado indivíduo e de suas propriedades, entre um `DataFile` e um `OntologyFile`. O nosso *IndividualMapping* é um representante ou um protótipo do *IndividualMapping* utilizado pelo *Chaos Pop*, sendo que existem algumas diferenças entre ambas as estruturas de dados. O *IndividualMapping* utilizado pela HOMI tem a seguinte estrutura:

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

- `_id`: identificador único criado pela base de dados;
- `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 do *Chaos Pop* correspondente ao `OntologyFile` que queremos mapear,
- `dataFileId`: id do *Chaos Pop* 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.

- `annotationProperties`: cada elemento deste *array* é composto por: tipo de anotação (*label*, *comment*, etc) e metadados dos *nodes* (no `DataFile`) envolvidos nesta propriedade.

6.1.2 Populate

Este objeto é um aglomerado de identificadores. Dentro do mesmo estão contidos os identificadores dos ficheiros que o utilizador seleccionou para mapear, tanto de `DataFile's` como de `OntologyFile's`. Este objeto foi criado para facilitar a comunicação entre *front-end* e *back-end* para prevenir a utilização de POST para obter informações, pois assim podemos colocar o ID de um *Populate* no *path* de um GET em vez de colocar uma lista no *body* de um POST.

```
{
        _id: ObjectId,
        dataFiles: [],
        ontologyFiles: []
}
```

6.1.3 DataFile e OntologyFile

Tanto *DataFile* como *OntologyFile* representam um ficheiro selecionado pelo utilizador de `DataFile` e `OntologyFile`, respetivamente. Um objeto de um destes é constituído pelo identificador único gerado pelo *MongoDb* (_id), pelo nome do ficheiro (*name*) e pelo seu identificador no servidor do *Chaos Pop* (*chaosid*).

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

6.2 Base de dados

Adicionamos uma base de dados documental à nossa aplicação, utilizando MongoDb. Esta base de dados está dividida em *collections*, em que cada uma destas aglomera documentos do mesmo tipo. Neste momento temos as seguintes *collections* e os respetivos objetos que elas guardam:

- *IndividualMappings*: [*IndividualMapping*]
- *Populates*: [*Populate*]
- *DataFiles*: [*DataFile*]
- *OntologyFiles*: [*OntologyFile*]

6.3 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 *controllers*, agrupados pela sua finalidade. Por exemplo, todos os *endpoints* relacionados com *IndividualMapping's* poderão ser encontrados no individual-mapping-routes.

index-routes:

GET	/home	

individual-mapping-routes:

POST	/map/individual	body - individualMapping: Object
POST	/map/individual/:id/properties/object	body - objProps: Array
POST	/map/individual/:id/properties/data	body - dataProps: Array
POST	/map/individual/:id/properties/annotation	body - annotationProps: Array
PUT	/map/individual/:id	body - individualMappingObject
DELETE	/map/individual/:id	
DELETE	/map/individual	body - individualMappings: List
GET	/map/individual/:id	
GET	/populate/data/dataprops/:id/view	
GET	/populate/data/objectprops/:id/view	
GET	/populate/data/annotationprops/:id/view	

file-routes:

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

populate-data-routes:

POST	/populate/data	body: Populate
GET	/populate/data/:id	
GET	/populate/data/:id/tree	

6.4 Front-End (Client Side)

A *user interface* visível ao utilizador na página inicial é a apresentada abaixo:

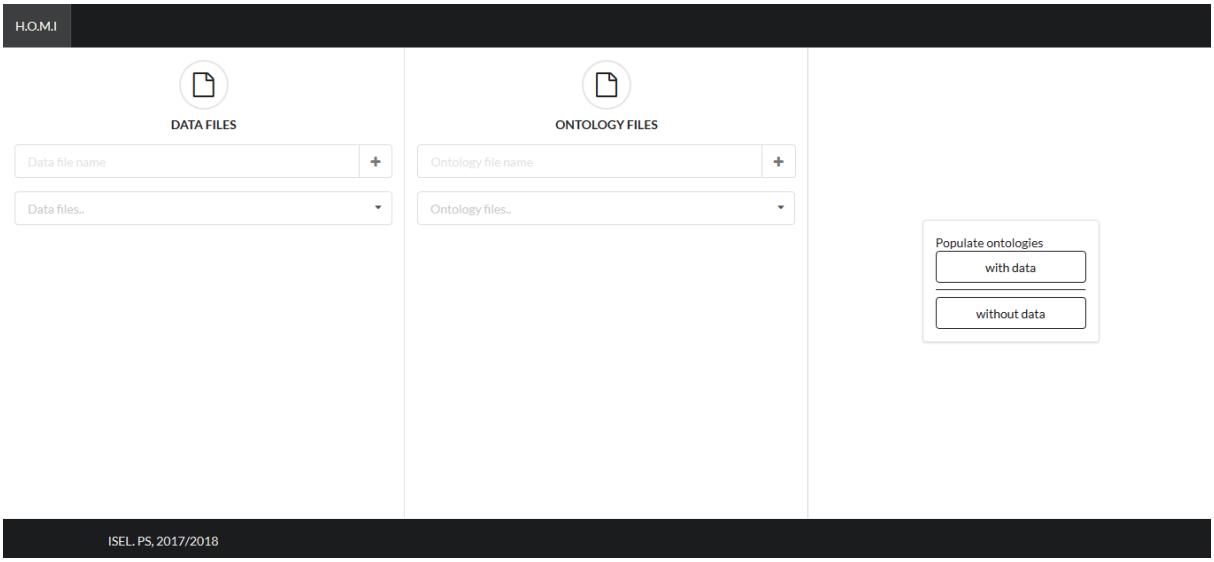

Figura 6.1 - Home page

É nesta página que a interação começa, sendo possível a inserção de ficheiros do tipo `DataFile` e `OntologyFile`, bem como escolher alguns dos ficheiros existentes para realizar um *Populate* com (*with data*) ou sem (*without data*) dados semi-estruturados.

6.4.1 Populate with data

Para realizar um *populate* deste tipo é necessário a seleção de um ou mais ficheiros de `OntologyFile` e `DataFile` (1) e em seguida realizar um pedido para *Populate with data* (2).

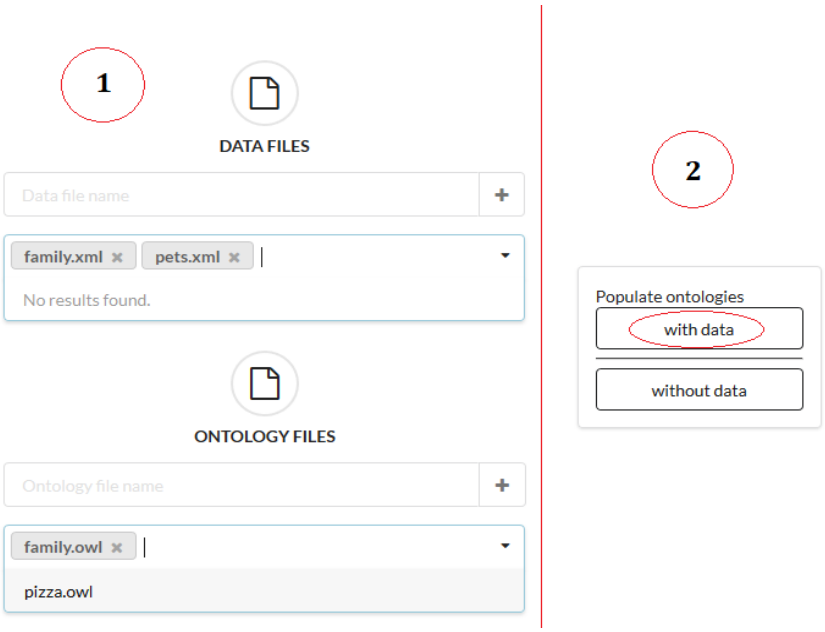

Figura 6.2 - Passo iniciais para a realização de um Populate with data

A página que será apresentada após o pedido, será a indicada na Figura 6.3. Nesta página o utilizador poderá criar *IndividualMapping's*. Este representa o mapeamento de um determinado *node* da árvore (3), na qual será utilizado para mapear todos os *nodes* com aquela *tag* e não apenas aquele *node* em específico.

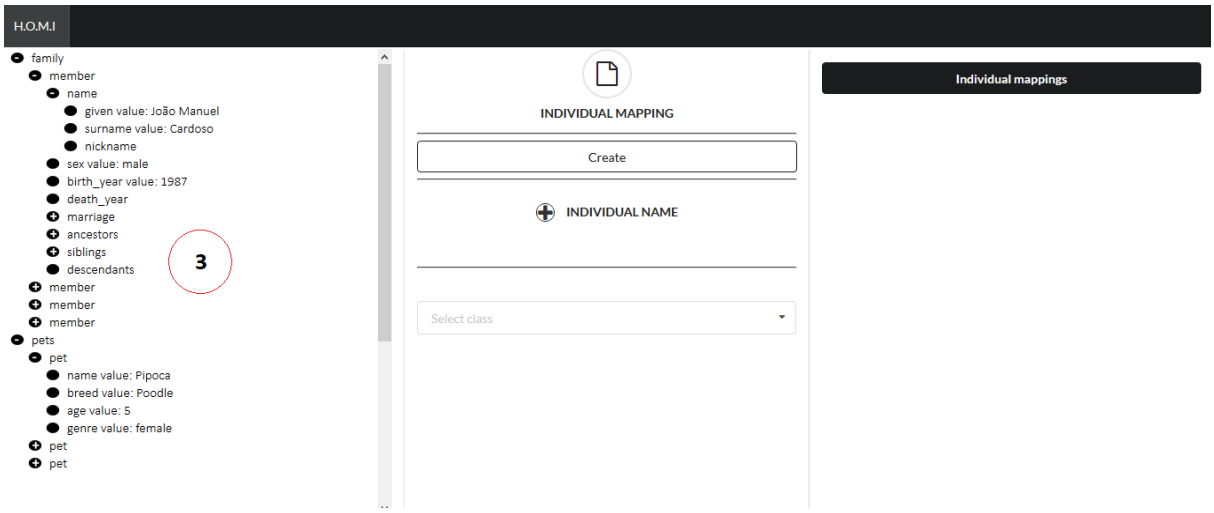

Figura 6.3 - Página de populate with data

A criação de um *IndividualMapping* inicia-se com a escolha de alguns *nodes* presentes na árvore para que os valores destes sejam mapeados como *IndividualName* (4). De seguida, o utilizador seleciona o *node* e a *OWL class* referentes ao *IndividualMapping* que deseja criar. (5)

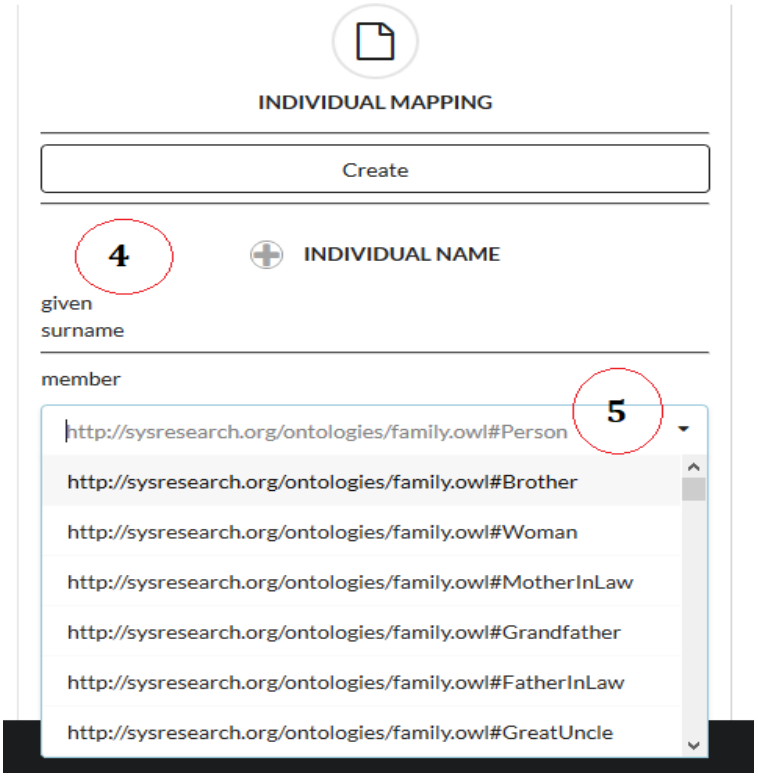

Figura 6.4 - Criação de individual mappgins

Após a criação do *IndividualMapping*, o utilizador pode começar a mapear as propriedades do mesmo, sendo estas do tipo *data*, *object* e *annotation*. Como no exemplo acima foi selecionado o *node member* para mapeá-lo como *Person* (5), sendo assim o mapeamento das *properties* deste só pode ser realizado a partir dos *children* de *member*. (6). As regras de mapeamento para cada tipo de *property* são:

- Para mapear uma *annotation property* é necessário selecionar o *node* (ou os *nodes*, podem ser selecionados mais que um) que serão associados a este mapeamento, bem como o tipo da anotação que está a ser feita (7). São suportadas as seguintes anotações: *Label, Comment, SeeAlso, IsDefinedBy, VersionInfo, BackwardCompability, IncompatibleWith*.
- Para mapear uma *data property* é necessário selecionar o *node* (ou os *nodes*, podem ser selecionados mais que um) que quer associar a este mapeamento bem como a *OWL data* IRI e o tipo desta propriedade (8). São aceites os tipos *Integer, Float, Double, String e Boolean.*
- Para mapear uma *object property* é necessário selecionar o *node* (apenas um) que será associado a essa propriedade e a *OWL object property* IRI a ser mapeada. (9).

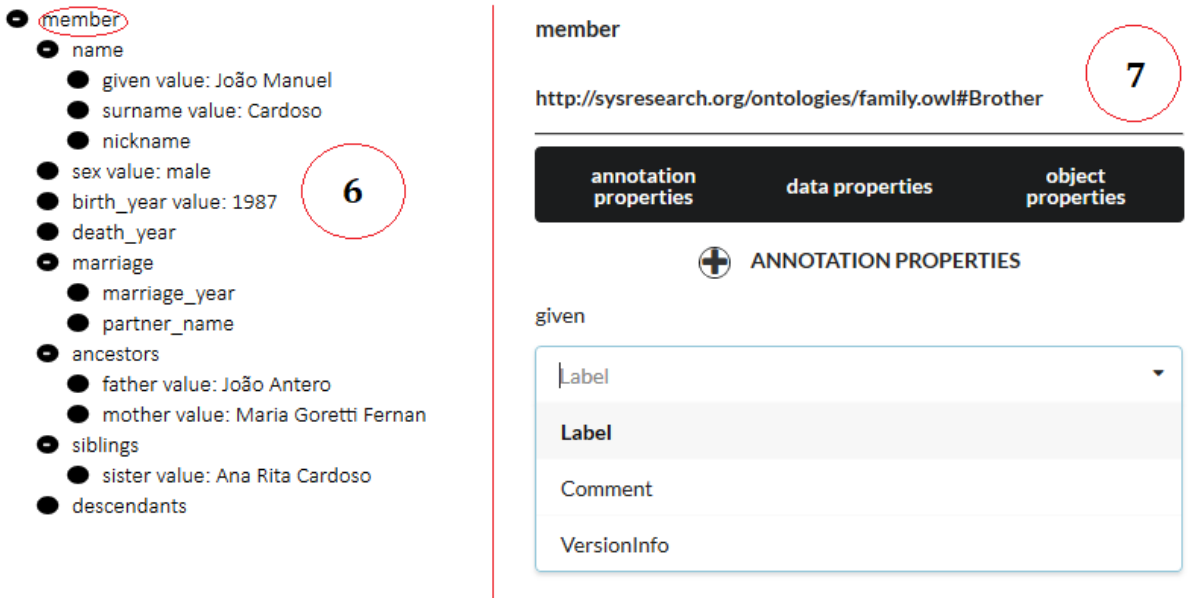

Figura 6.5 - Mapeamento da properties de um individual mapping

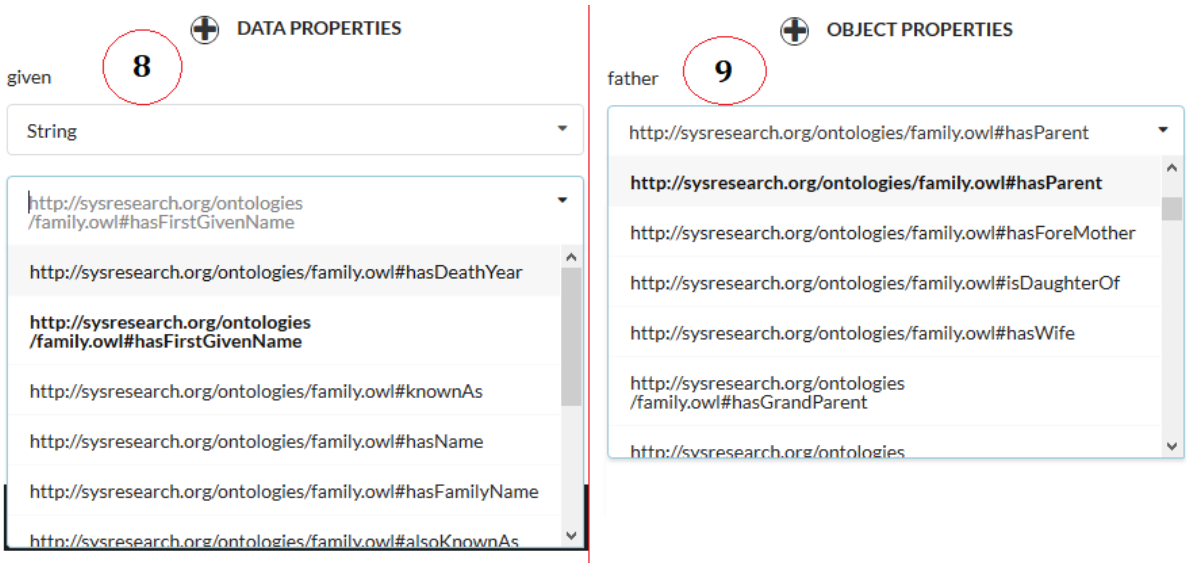

Figura 6.6 - Mapeamento das properties de um individual mapping

Depois do utilizador definir as propriedades que quer mapear, é necessário carregar no botão *Finish individual mapping*. Ao carregar neste botão, o *IndividualMapping* criado será enviado para o *Chaos Pop*. Podem ser criados tantos *IndividualMapping's* quanto os *nodes* diferentes. Quando o utilizador criar todos os *IndividualMapping's* necessários, estes serão aglomerados num *Mapping*, na qual posteriormente será enviado para o *Chaos Pop* para ser criado o ficheiro de *output*.

6.5 Back-End (Server Side)

Como mencionado em no Capítulo 5.1, o servidor está dividido em 3 camadas: *controllers*, *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:

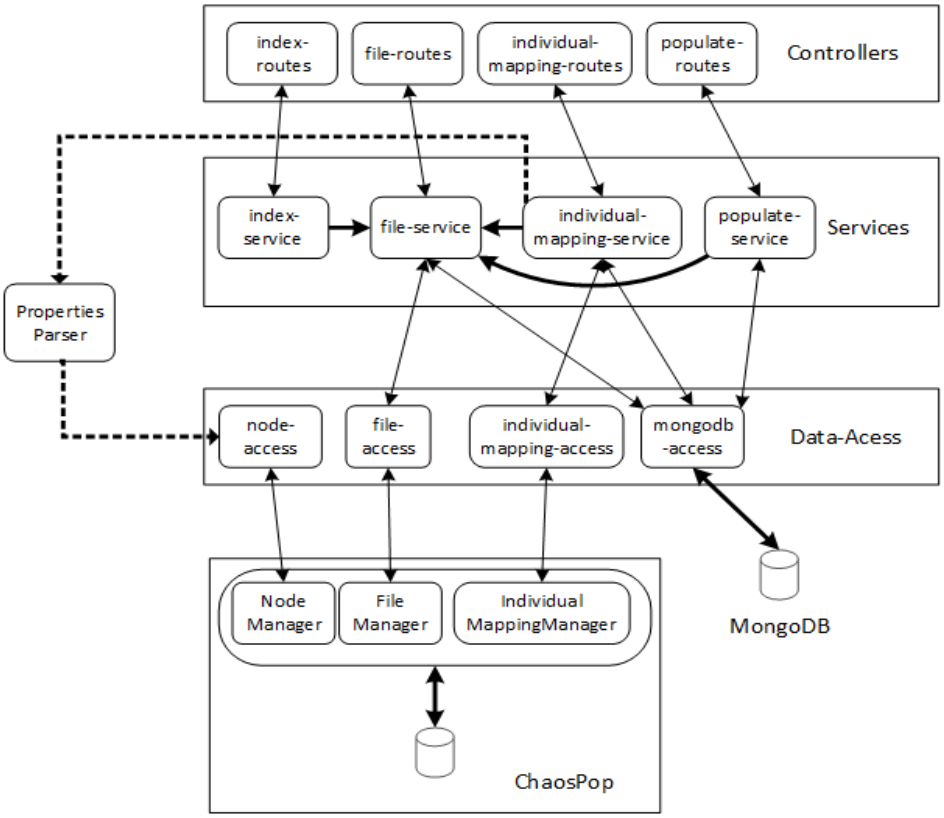

Figura 6.7 - Camadas de acesso e seus respetivos módulos

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.

6.5.1 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.

7 Progresso do projeto

Apesar de termos avançado bastante desde o relatório de projeto ainda estamos atrasadas em relação à calendarização apresentada no último relatório. Este atraso foi devido ao facto de termos adicionado outras características que achamos importantes para a utilização da nossa aplicação, como por exemplo, a utilização de múltiplos ficheiros do tipo `OntologyFile` e `DataFile`. Tivemos também que implementar a base de dados bem como o acesso à mesma mais cedo de forma a conseguirmos melhorar a experiência de utilização. Encontramos também alguns problemas com a API do *Chaos Pop* o que levou à alteração da mesma e, consequentemente, tivemos que refazer algumas partes do nosso código, o que também contribuiu para este atraso.
Na Tabela 6.1 está apresentada a nova calendarização do projeto com as mudanças necessárias.

Data de início	Semana	Descrição
19/02/2018	1-2	- Compreensão da necessidade da ferramenta nos dias atuais - Estudo do Chaos Pop
05/03/2018	3-4	- Estudo da ferramenta Electron - Desenvolvimento da proposta
19/03/2018	5-7	- Entrega da proposta do projeto - Esclarecimentos sobre a API Chaos Pop e utilização do mesmo em alguns exemplos
09/04/2018	8-10	- Início da implementação dos níveis de acesso (*controllers, services, data-access*) - Início do desenvolvimento da *User Interface - Populate with data*
30/04/2018	11	- Apresentação individual e entrega do relatório de progresso
07/05/2018	12-14	- Continuação do desenvolvimento da *User Interface - populate with data* e corrigir os níveis de acesso
28/05/2018	15	- Entrega do cartaz e da versão beta
28/05/2018	16	- Finalização da *User Interface - Populate With Data* -Início da *User Interface - Populate Without Data*
11/06/2018	17	-Finalização da User-Interface - Populate without data - Desenvolvimento da aplicação *desktop*
18/06/2018	18	- Definição da descrição de ferramentas em OWL
18/06/2018	19	- Testes de escalabilidade
25/06/2018	10-21	- Finalização do relatório e entrega da versão final

Tabela 7.1 - Calendarização atual do projeto

8 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.

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