MastersPlat

Filipe Ribeiro[1], Luís Castro[2] and José Torres[3]

1 a15061@alunos.ipca.pt
2 a15060@alunos.ipca.pt
3 a14487@alunos.ipca.pt

Resumo. Este relatório descreve todo o processo de criação de uma API de agregação de dissertações de mestrado e de uma interface gráfica que suporta todas as funcionalidades da aplicação. Ao analisar o problema foram definidas todas as rotas que a API irá de suportar, bem como os diferentes tipos de pedidos para cada uma das mesmas. De seguida foi elaborado um modelo de dados capaz de dar suporte a toda a API nas diversas operações.
A solução desenvolvida suporta as operações CRUD, bem como um sistema de paginação, projeção, pesquisa simples, avançada e ordenação de todos os dados obtidos e inseridos na base de dados.

1 Introdução e Objetivos

Com a necessidade de centralizar toda a informação, o objetivo é desenvolver uma API seguindo a filosofia RESTful para agregação e descoberta de dissertações de mestrado dos diferentes repositórios disponíveis à comunidade.

Numa visão mais detalhada, um utilizador tem de ser capaz de introduzir uma nova fonte de dados, carregar as teses dessa mesma fonte quando desejar e posteriormente consultar todas as teses introduzidas. Para todos os dados resultantes poderão ser aplicadas várias opções pertinentes como limite de itens a visualizar e offset aos mesmos, criando assim um sistema de paginação, aplicar ordenação, projeção e diversos filtros os dados obtendo assim o resultado pretendido.

Após desenvolver a API, é necessário implementar uma interface gráfica, neste caso utilizando a Framework Angular, que irá dar suporte a todas as funcionalizes da API. Ao pensar numa componente gráfica, é necessário considerar algumas noções básicas de design além de analisar o diferente publico que irá consumir a aplicação.

2 Descrição da Solução

2.1 Arquitetura

A API suporta várias rotas, cada uma delas com tipos de pedidos distintos. Toda a API suporta JSONP para troca de mensagens entre cliente e servidor, a Fig. 1. apresenta a arquitetura global da API bem como cada tipo de pedido que cada rota suporta.

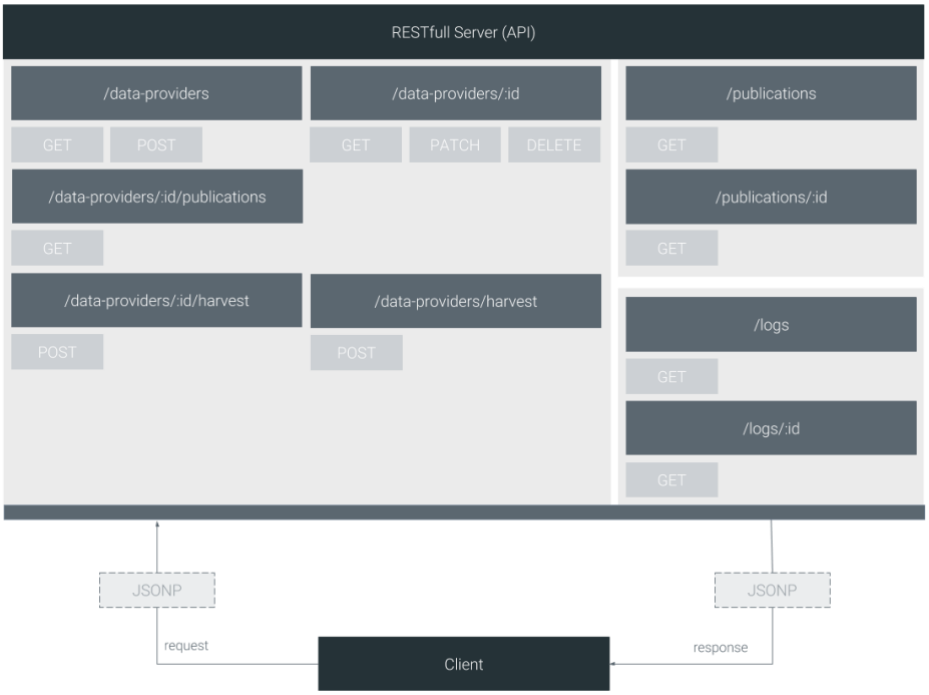

Fig. 1. Arquitetura da API

2.2 Descrição da API

Ao analisar os objetivos foram implementados um conjunto de funcionalidades que são essenciais para a interação do utilizador com a API.

A API e constituída por três conjuntos de rotas, para os Data Providers, publicações e para os logs, cada uma delas com pedidos específicos.

Para os Data Providers foram implementadas as operações CRUD, podendo adicionar, editar, consultar e eliminar uma fonte de informação. Para uma visualização mais detalhada é possível consultar um Data Provider específico ao utilizar o seu identificador.

Ao inserir um Data Provider, as respetivas publicações não são carregadas automaticamente, podendo ser carregadas quando o utilizador desejar. No caso de existir vários Data Providers para carregar é possível executar a mesta ação para todos invocando um único método.

Após as publicações serem carregadas, todos os Data Providers podem ser atualizados. Internamente esse processo pode ser realizado de duas formas distintas configurável num ficheiro de configuração na API. Um Data Provider pode ser atualizado com base na data da última atualização e na data em que a ação de atualização está a ser executada, neste caso é criado um intervalo de tempo para verificar se existem novas publicações. Uma outra possibilidade é uma atualização total, removendo todas as publicações e de seguida voltar a carregar o Data Provider.

A operação de carregar e atualizar um ou múltiplos Data Providers podem ser operações bastante demoradas, para otimizar esse processo o carregamento é feito de forma assíncrona, controlando posteriormente o estado do Data Provider, se o carregamento já foi concluído ou ainda está em processo através de uma variável de estado atribuída a cada um dos Data Providers.

A API suporta um sistema de Logs que para cada Data Provider que é carregado, é criado um registo com a data de início e de fim da operação, as informações do Data Provider bem como o total de publicações foram registadas.

Ao agregar várias publicações, é possível consultar todas as publicações presentes na aplicação ou de um determinado Data Provider ao identificar o identificador do Data Provider pretendido.

Para todas as operações que listagem (GET), como listar todos os Data Providers, publicações ou registo de Logs, a API suporta várias opções que podem ser utilizadas de modo o obter o resultado pretendido. É possível limitar e atribuir um offset aos itens, criando assim um sistema de paginação, bem como, filtrar, projetar ou ordenar o conteúdo resultante da AP, garantindo assim que o utilizador tem um maior controlo do resultado conseguindo-o personalizar de acordo com o seu objetivo.

Cada método pode retornar de duas formas, ou retorna com sucesso ou erro caso tenha ocorrido. Cada pedido retorna o código HTTP mais adequado, sendo que em caso de erro segue uma estrutura composta por um código e por uma mensagem com uma descrição do erro ocorrido. Os códigos de erro são compostos por três grupos de dígitos, o primeiro dígito identifica o tipo do erro: Servidor (1), Validação (2) ou Lógica (3). O segundo grupo identifica o módulo e o último identifica o erro dentro desse mesmo modulo.

```
{
  error: {
    code: 'x.xxx.xxxx',
    message: 'Error Message!'
  }
}
```

Fig. 2. Estrutura da mensagem de erro

No decorrer do desenvolvimento da API foram utilizadas diversas bibliotecas suportadas pelo NPM, cada uma delas com um objetivo concreto e de auxílio o desenvolvimento da API, entre as quais:

- **express**[1]: Framework Node utilizada criar a estrutura base da solução desenvolvida.
- **swagger-ui-express**[2]: Utilizada para gerar a documentação da API.
- **mongoose**[3]: Ao utilizar o MongoDB como base de dados, esta biblioteca facilita a modelação de objetos e a interação com os mesmos.
- **api-query-params**[4]: Converte parâmetros do URL para poderem ser passados como consultas para o MongoDB, facilitando o processo de consulta, paginação e ordenação.
- **compression**[5]: Utilizada na fase de deploy da API, é um middleware que possibilita a compressão dos pedidos do servidor.

2.3 Modelo de dados

Como já referido foi utilizado MongoDB para armazenar os dados. Foram criadas três coleções, cada uma delas para suportar os requisitos definidos.

1 https://www.npmjs.com/package/express
2 https://www.npmjs.com/package/swagger-ui-express
3 https://www.npmjs.com/package/mongoose
4 https://www.npmjs.com/package/api-query-params
5 https://www.npmjs.com/package/compression

Fig. 3. Modelo de dados da API

Data Providers: Cada Data Provider ao ser inserido na aplicação é constituído por um nome e um URL. Automaticamente ao ser inserido é registado a data atual, o total de publicações, por default zero e um status, para identificar o estado de um Data Provider, sendo que '-1' indica que ainda não foi carregado.

Um Data Provider após ser carregado, fica guardado a data em que o processo de carregamento finalizou como data da última atualização, atualiza a informação do total de publicações e troca o valor da variável status para, em caso de sucesso '0', ou em cado de erro '2'. Durante o carregamento esta variável fica com o valor '1' sendo assim possível saber quais os Data Providers que estão a carregar.

Foi criado um índice em determinados elementos do tipo texto para posteriormente ser possível executar operações de pesquisa sobre os dados.

```
const dataProviderSchema = mongoose.Schema({
  _id: mongoose.Schema.Types.ObjectId,
  name: {type: String, required: true },
  url: { type: String, required: true },
  total: { type: Number, default: 0 },
  inserted_date: { type: Date, default: Date.now },
  updated_date: { type: Date },
  status: { type: Number , default: -1 }
});

dataProviderSchema.index( {
  name: "text",
  url: "text"
 }, { name: 'dataproviders_index' });

module.exports = mongoose.model('DataProvider', dataProviderSchema);
```

Fig. 4. Modelo de Dados - Data Provider

Publicações: Para cada publicação é armazenado a sua Metadata mais relevante bem como a informação do seu Data Provider. Foi criado novamente um índice para suportar a operação de pesquisa.

```
const publicationSchema = mongoose.Schema({
  _id: mongoose.Schema.Types.ObjectId,
  identifier: { type: String, required: true },
  datestamp: { type: Date, required: true },
  metadata: {
    title: { type: String },
    creator: [ { type: String } ],
    subject: [ { type: String } ],
    description: [ { type: String } ],
    identifier: [ { type: String } ],
    publisher: [ { type: String } ],
    contributor: [ { type: String } ],
    date: [ { type: String } ],
    type: { type: String },
    format: { type: String },
    lang: { type: String },
    rights: { type: String },
  },
  data_provider: {
    id: { type: mongoose.Schema.Types.ObjectId, required: true },
    name: { type: String, required: true },
    url: { type: String, required: true }
  }
});

publicationSchema.index( {
  "identifier": "text",
  "metadata.creator": "text",
  "metadata.title": "text",
  "metadata.subject": "text",
  "metadata.description": "text",
  "metadata.identifier": "text",
  "metadata.publisher": "text",
  "metadata.contributor": "text",
  "metadata.date": "text",
  "metadata.type": "text",
  "metadata.format": "text",
  "metadata.lang": "text",
  "metadata.rights": "text",
  "data_provider.name": "text",
  "data_provider.url": "text"
},{ name: 'publication_index' });

module.exports = mongoose.model('Publication', publicationSchema);
```

Fig. 5. Modelo de dados - Publicações

Logs: Para cada processo de carregamento ou atualização de um Data Provider é criado um registo em que é guardada a informação do mesmo, a data de início e de fim da operação bem como o total de publicações que foram registadas.

```
const logSchema = mongoose.Schema({
  _id: mongoose.Schema.Types.ObjectId,
  start_date: { type: Date, default: Date.now },
  end_date: {type: Date, required: true },
  data_provider: {
    id: { type: mongoose.Schema.Types.ObjectId, required: true },
    name: { type: String, required: true },
    url: { type: String, required: true }
  },
  total_records: { type: Number, required: true }
});

logSchema.index( {
  "data_provider.name": "text",
  "data_provider.url": "text"
 },{ name: 'logs_index' });

module.exports = mongoose.model('Log', logSchema);
```

Fig. 6. Modelo de dados – Logs

2.4 Modelo de interação (GUI)

2.4.1 Funcionalidades a Implementar

Após desenvolver a API foi implementado uma interface gráfica que irá dar suporte a toda a aplicação.

Deverá permitir a um utilizador realizar diversas pesquisas sobre as publicações presentes, além de, como componente administrativa, permitir realizar todas as operações CRUD dos Data Providers que a API suporta bem como consultar todos os logs existentes.

Como já referido foi utilizado a Framework Angular para desenvolver toda a aplicação, no entanto no decorrer do desenvolvimento foram utilizadas algumas bibliotecas e ferramentas auxiliares, entre as quais:

- **Bootstrap**[6]: Framework usada para desenvolver todo o layout base da aplicação;
- **Fontawesome**[7]**:** Todos os icons utilizados na aplicação.

[6] https://getbootstrap.com/

[7] https://fontawesome.com/

- **ngx-bootstrap**[8]: Todos os componentes disponíveis na Framework Bootstarp com uma fácil integração com o Angular.
- **ng-select**[9]: Utiliza para implementar todos as listas drop-down presentes na aplicação.
- **ngx-infinite-scroll**[10]: Utilizado para facilitar a implementação de scroll infinito.

2.4.2 Abordagem Visual

Na parte destinada ao utilizador comum, a abordagem utilizada consiste num design simples onde dispõem de uma secção para pesquisa, quer em todos os campos dos dados quer solicitando a pesquisa em campos específicos.

O resultado obtido consiste numa listagem utilizando um sistema de cartões, para apresentar cada uma das publicações. É possível também caso seja pretendido, visualizar mais conteúdo de uma determinada publicação, apresentando todos os dados da mesma.

Para uma questão de eficácia e de usabilidade, o sistema de paginação dos dados foi implementado aplicando Infinite Scroll.

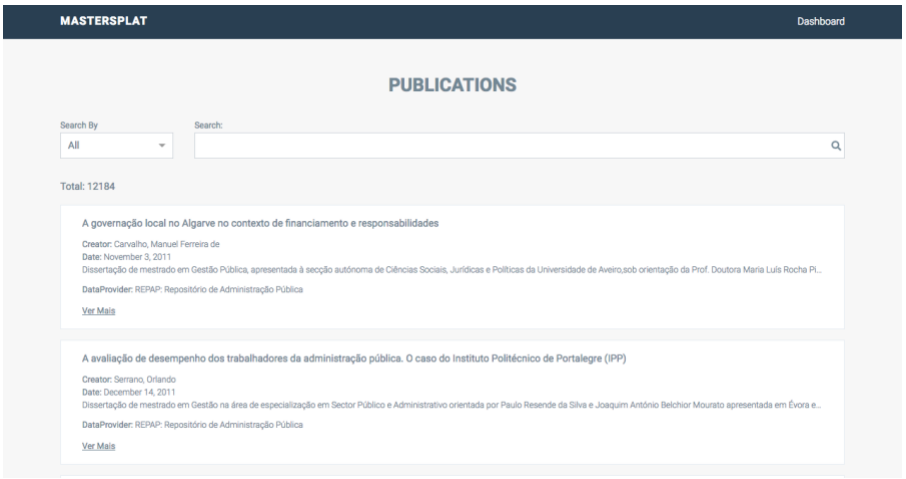

Fig. 7. Layout página principal

8 https://valor-software.com/ngx-bootstrap/#/
9 https://ng-select.github.io/ng-select#/data-sources
10 https://www.npmjs.com/package/ngx-infinite-scroll

A secção administrativa foi pensada como uma Dashboard onde dispõe de todas as operações CRUD aplicadas aos Data Providers e todas as restantes operações que a API suporta relacionadas com as publicações e os logs. É possível também efetuar pesquisa sobre os diferentes dados, mas apenas na forma de pesquisa simples, ao contrário da página inicial.

Nesta área também é possível visualizar um conjunto de dados estatísticos gerais como o total de Data Providers, publicações e de logos.

Fig. 8. Layout da Dashboard - Estatísticas

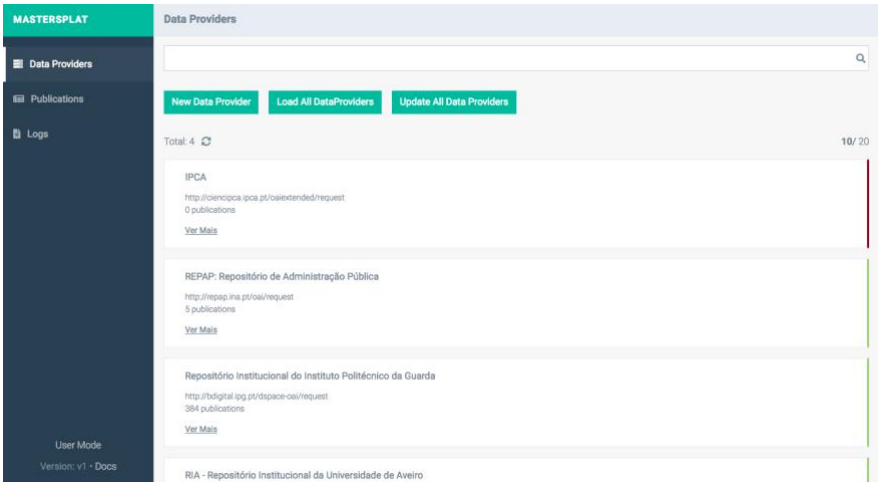

Fig. 9. Layout da Dashboard – Data Providers

3 Infraestrutura

3.1 Servidor Cloud

Para finalizar a API foi necessário alojar a mesma utilizar uma plataforma Cloud e configurar um cliente de Base de Dados. A API foi alojada na plataforma Heroku e utiliza o MongoDB Atlas como serviço de base de dados.

A API e toda a documentação gerada pelo swagger pode ser encontrada em:
- https://pw-mastersplat.herokuapp.com/

3.2 Docker Container

De forma a facilitar a processo de distribuição da aplicação independentemente de qualquer infraestrutura, foi criado utilizado a ferramenta Docker um ambiente de execução que garanta todo esse processo.

Para a criação do container, foi criado um ficheiro Dockerfile utilizando uma image do Docker Hub contendo a versão do node utilizada. De seguida foi explicitado todos os comandos sequencialmente para uma correta integração entre a aplicação cliente e servido.

```
# base image
FROM node:8.11.1
LABEL description="PW-Mastersplat" version="1.0"
# Expose Port
EXPOSE 3000 8080
# Create a new directory for apllication
RUN mkdir -p /home/node/app
# Copy server and client folders
COPY ./mastersplat_app/ /home/node/app

# Change working directory: Client
WORKDIR /home/node/app/client
# Install angular CLI
RUN npm install -g @angular/cli@6.0.7
# Install all the dependencies in the package.json in
RUN npm install
# Build angular to production
RUN ng build --prod
# Copy compiled angular for correct the folder
RUN cp -r ./dist/mastersplat-layout/* /home/node/app/server/public

# Change working directory: Server
WORKDIR /home/node/app/server
# Install all the dependencies: --unsafe-perm: for run harvest package.json
RUN npm install --unsafe-perm

CMD ["npm","start"]

```

Fig. 10. Dockerfile - Mastersplat

Posteriormente foi criado um docker-compose com um serviço que constrói o container com base nas configurações do dockerfile e definindo a ambiente de execução:

```yaml
version: "3"
services:
  mastersplay_app:
    build: .
    container_name: mastersplat_app
    restart: always
    user: "root"
    working_dir: /home/node/app/server
    environment:
      - NODE_ENV=production
      - PORT=3000
    ports:
      - "3000:3000"

```

Fig. 11. Docker-compose Mastersplat

4 Conclusões e melhorias futuras

Este projeto revelou-se um verdadeiro desafio durante o seu processo, desde de encontrar todas as rotas necessárias, a melhor forma dos dados serem armazenados a base de dados, deixando um pouco de lado a pensamento da estrutura de uma base dados relacional, até pensar numa interface gráfica que suporte toda a API pensando um pouco num design adequado ao tipo de utilizador

Uma funcionalidade que deveria ser realizada é a implementação de um sistema de autenticação. A divisão da aplicação entre dashboard e página principal foi pensada para suportar esta funcionalidade mas não foi possível implementar.