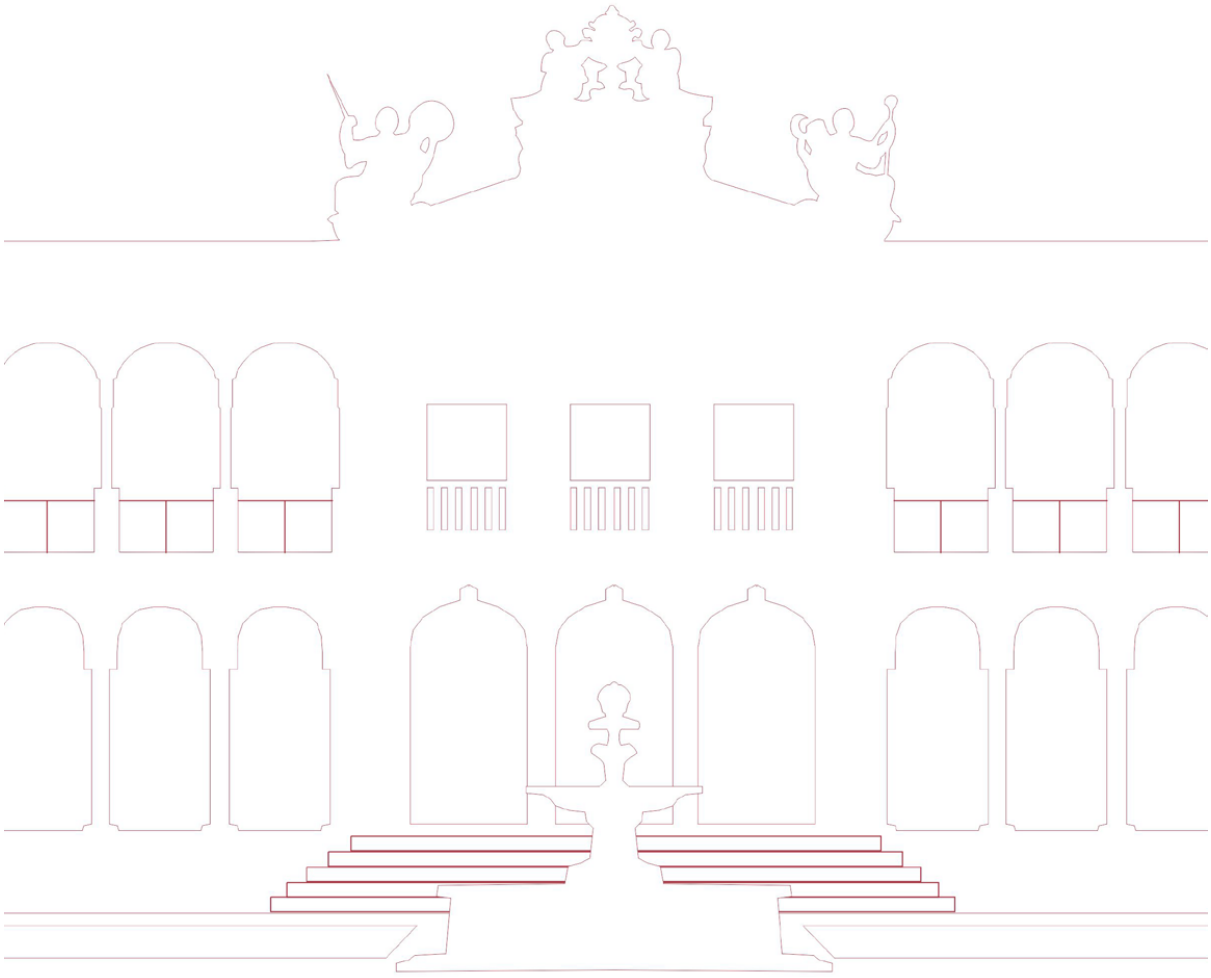
Dep. de Informática, Universidade de Évora: Recolha e visualização de dados climáticos

UNIVERSIDADE DE ÉVORA
ESCOLA DE CIÊNCIAS E TECNOLOGIA
DEPARTAMENTO DE INFORMÁTICA

Licenciatura em Eng. Informática
Estágio-Projecto 2020-2021



Tomás Dias

Orientador na empresa: José Saias
Orientador no departamento: Vítor Nogueira

Trabalho desenvolvido no Departamento de Informática da Universidade de Évora no âmbito da disciplina de Estágio-Projeto da Licenciatura em Eng. Informática.

Évora, 25 de junho de 2021

Conteúdo

1 Introdução

Neste relatório, abordarei toda a evolução do estágio, especificando todas as suas etapas, fases e consequências.

1.1 Enquadramento

O estágio foi realizado no Departamento de Informática da Universidade de Évora, situado na Rua Romão Ramalho 59, em Évora, remotamente.

Não fui integrado em qualquer equipa, existindo apenas o acompanhamento dos tutores em relação ao trabalho efetuado.

1.2 Objetivos

O estágio tinha como principal objetivo a implementação de uma plataforma que permitisse recolher os dados climáticos da estação meteorológica do Instituto de Ciências da Terra da Escola de Ciências e Tecnologias da Universidade de Évora e que conseguisse disponibilizar estes dados de forma programática através de uma *API (Application Programming Interface)*, para que pudessem ser utilizados por outras aplicações.

No âmbito do estágio, também foi planeado o desenvolvimento de uma aplicação móvel que permitisse a visualização dos dados recolhidos, usando a API implementada, que acabou por não acontecer.

1.3 Contribuições

Como resultado do trabalho desenvolvido no decorrer do estágio, foi implementada a API que permite a disponibilização dos dados climáticos recolhidos pela estação meteorológica do ICT da Universidade de Évora para outras aplicações.

1.4 Estrutura do documento

O relatório encontra-se estruturado pelas seguintes secções principais:

- **Estado da arte**: É descrito todo o processo de levantamento e análise das possíveis soluções, ferramentas e metodologias que pudessem fazer sentido ser utilizadas nas tarefas do estágio.
- **Ambiente de desenvolvimento**: É descrito o ambiente técnico, nomeadamente o hardware e as ferramentas de software utilizadas. Também é especificada a metodologia de trabalho adotada.
- **Trabalho desenvolvido**: É resumido o trabalho realizado durante o estágio, incluindo as tarefas efetuadas com os respetivos intervalos de tempo a elas associadas.
- **Avaliação crítica**: É apresentado o que aprendi com a experiência, tanto do ponto de vista técnico como não técnico.

2 Estado da arte

2.1 Ferramentas para Web Scraping

Para a implementação da ferramenta de recolha de dados, procurou-se soluções para recolher os dados climáticos disponibilizados pelo *website* do ICT da Universidade de Évora. Para o efeito, era necessário aplicar o processo de *Web Scraping* e nesse sentido as opções exploradas foram:

- ***Scrapy*** [1]: *Framework* de alto-nível escrito em *Python* especializado em *Web Scraping* e *Web Crawling* que, entre outras funções, é utilizado principalmente para extração de dados de forma estruturada, monitorização de dados e testes automatizados. Estes processos são feitos através de *spiders*, classes que definem como o *website* (ou um grupo de *websites*) é rastreado e como a informação é estruturada. Destaca-se pela rapidez e eficiência na recolha de grande quantidade de dados, suporte *built-in* de várias linguagens de programação como *CSS* e *XPath* para a recolha de dados, e extensibilidade. Contudo, a robustez de programação que apresenta acaba por não justificar a sua utilização dado à simplicidade dos dados a serem recolhidos.
- ***Selenium*** [2]: *Framework* que funciona fundamentalmente para criação de testes de aplicações *web*. Devido à sua versatilidade e capacidade de se estender em várias formas, pode ser utilizado para *Web Scraping*. No entanto, como não é especificado para este tipo de tarefa, torna-se ineficiente e suscetível a possíveis erros, sendo o seu uso apropriado apenas no caso de existência de conteúdo dinâmico (*scripts*). Sendo que esta situação não se verifica, acaba por não ser a melhor opção.
- ***Beautiful Soup*** [3]: Biblioteca *Python* para extrair dados de ficheiros *HTML* e *XML*. O seu setup é simples, é de fácil aprendizagem e masterização, destacando-se a existência de métodos descritivos para as ações que efetuam, e a documentação existente é vasta e compreensível. Ainda assim, requer bibliotecas adicionais como a *requests* para aceder ao conteúdo do *website*, e pode apresentar problemas ao recolher conteúdo dinâmico e em grande quantidade. Não obstante, como as características dos dados a serem extraídos não vão ao encontro destas falhas, acaba por se tornar na opção mais adequada.

2.2 Escolha da base de dados

Com a finalidade de guardar os dados recolhidos, averiguou-se possíveis soluções nesse sentido. O principal propósito era determinar de que forma a base de dados deveria ser esquematizada, seguindo-se a escolha da base de dados em si. Deste procedimento conclui-se que:

- **Base de dados relacional** [4, 5, 6]: A utilização de uma base de dados deste tipo assegura diversos benefícios como a eficácia na normalização dos dados, permitindo a remoção de redundâncias e a sua otimização; as transações sendo compatíveis com o conjunto de propriedades *ACID (Atomicity, Consistency, Isolation, Durability)* garantem segurança e estabilidade; permite um acesso rápido e fácil aos seus *records*; é extensível verticalmente (melhoramento da capacidade do servidor ou do *hardware*). Por outro lado, o processo de construção da interface é complexo e são envolvidos custos elevados na gestão de grandes quantidades de dados e na extensibilidade.
- **Base de dados não-relacional** [7, 8]: Por não utilizar esquemas ou tabelas oferece um alto nível de flexibilidade com modelos de dados; poupa-se tempo e esforço visto que não existe necessidade de construir um modelo de base de dados detalhado; apresenta um bom custo de extensão. Todavia, utiliza as propriedades *ACID* de modo "relaxado"o que poderá afetar a sua consistência; não é muito eficaz aquando da realização de *queries* dinâmicas e complexas; é de evitar se a aplicação precisar tempo de execução flexível; menos documentação e suporte existente.
- **Bases de dados *time series*** [9, 10]: Otimizada para dados *time-stamped* ou séries cronológicas. Os dados das séries cronológicas são simplesmente medições ou eventos que são rastreados, monitorizados, amostrados e agregados ao longo do tempo. Podem ser métricas de servidor, monitorização do desempenho de aplicações, dados de rede, dados de sensores, eventos, cliques, transacções de mercado, e outros tipos de dados analíticos. As propriedades que tornam os dados das séries cronológicas muito diferentes de outros dados são a gestão do ciclo de vida dos dados, resumos, e *range scans* de grande alcance de muitos registos (*records*).

2.3 Análise da arquitetura de microsserviços

No que diz respeito à arquitetura da plataforma, analisou-se a arquitetura de *microsserviços* [11]. É um tipo de arquitetura que estrutura uma aplicação como um conjunto de serviços que são altamente sustentáveis e testáveis, "loosely coupled" e em que o deployment ocorre de forma independente. Algumas das conclusões retiradas foram:

- Permite a entrega e implementação contínua de aplicações grandes e complexas.
- Maior capacidade de manutenção pois cada serviço é relativamente pequeno, pelo que é mais fácil de compreender e alterar.
- Melhor testabilidade pois os serviços são mais pequenos e mais rápidos de testar.
- Melhor deployment pois este é feito independentemente para cada serviço.
- Permite organizar o esforço de desenvolvimento em torno de equipas múltiplas e autónomas, sendo que cada equipa possui e é responsável por um ou mais serviços, podendo desenvolver, testar, distribuir e escalar os seus serviços independentemente de todas as outras equipas.
- Cada microsserviço é relativamente pequeno sendo mais fácil de entender para o desenvolvedor.
- As implementações são feitas mais rapidamente.
- Melhor isolamento de falhas (por exemplo, se houver uma fuga de memória num serviço, então apenas esse serviço será afetado) em comparação com uma arquitetura monolítica, pois um componente mal-comportado nesta pode fazer cair todo o sistema.
- É necessário implementar um mecanismo de comunicação entre serviços que está sujeito a falhas.
- A implementação de requests que abrangem vários serviços é mais difícil.
- Testar as interações entre os serviços é mais complexo.
- As ferramentas de desenvolvimento são orientadas para a construção de aplicações monolíticas e não fornecem apoio explícito para o desenvolvimento de aplicações distribuídas.
- O deployment é mais complexo quando comparado com arquiteturas monolíticas.
- O consumo de memória é alto.

2.4 Escolha da arquitetura da *API*

Foram analisadas dois tipos de arquiteturas para a *API*.

2.4.1 *REST (Representational state transfer)*

Os serviços web do tipo *REST* [12] permitem aos sistemas aceder e manipular as representações textuais dos recursos web utilizando um conjunto predefinido de operações stateless (incluindo GET, POST, PUT, e DELETE). Além disso, a implementação do cliente e do servidor é frequentemente feita de forma independente. Isto significa que o código do lado do cliente pode ser alterado sem afetar a forma como o servidor funciona e vice-versa. Desta forma, são mantidos modulares e separados. A ideia central do *REST* é que tudo é um recurso que é identificado por um URL. Na sua forma mais simples, seria possível recuperar um recurso através de um pedido GET ao URL do recurso e obter uma resposta. Foram tiradas a seguintes conclusões:

- É facilmente extensível devido à dissociação existente entre cliente e servidor
- É flexível uma vez que os dados não estão ligados a recursos ou métodos, sendo capaz de lidar com diferentes tipos de pedidos e devolver diferentes formatos de dados.
- No caso de aplicações web que requerem grandes conjuntos de dados que combinem recursos com eles relacionados, seriam necessários múltiplos pedidos e respostas para aceder a esses dados de forma a extrair toda a informação necessária
- A existência de over-fetching e under-fetching. Isto acontece pois os clientes apenas podem enviar requests de dados a "endpoints" que retornam estruturas de dados fixas, descarregando mais informação do que aquela que a aplicação necessita (over-fetching) ou quando o "endpoint" não

fornece toda a informação necessária obrigando o cliente a enviar vários pedidos até a obter (under-fetching).

2.4.2 *GraphQL*

Diferencia-se de *REST* por não lidar com recursos dedicados. Em vez disso, no *GraphQL* [13] tudo é considerado como um grafo, implicando que tudo se encontra ligado entre si. Com esta arquitetura é possível adaptar um request de forma a corresponder exatamente aos seus requisitos. Foi possível concluir que:

- Permite alterações no lado do cliente (por exemplo, UI) de forma a não modificar a estrutura do servidor.
- É possível obter informações acerca dos dados pedidos e de como estes estão a ser utilizados, pois cada cliente especifica exatamente a informação que necessita. Isto permite depreciar os campos que os clientes já não utilizam, melhorando o desempenho da API.
- Utiliza um único "endpoint" em vez de seguir a especificação HTTP para a cache (ao nível da rede é importante, pois pode reduzir a quantidade de tráfego para um servidor ou manter os dados acedidos frequentemente perto do cliente).
- Não é a melhor solução para aplicações simples, pois acrescenta complexidade (como types ou queries) a coisas que poderiam ser feitas de forma muito mais fácil com REST.

2.5 Ferramentas para a criação da *API*

Sendo todos os frameworks analisados de arquitetura *MVC (Model–view–controller)* [14] que permitem um mapeamento de dados utilizando o *ORM (Object–relational mapping)* [15], foram apuradas os principais aspetos de cada um.

2.5.1 *Rails*

Referente ao *Rails* [16], pode-se concluir que:

- Aprendizagem simples.
- Consistente (código simples e limpo).
- Existência de um grande número de ferramentas e bibliotecas.
- Elevada escalabilidade.
- Eficiente a nível de segurança.
- Pouca documentação.
- Baixa velocidade de execução a quando comparado com outros frameworks.
- Arranque demoroso dependendo do número de recursos utilizados

2.5.2 *Laravel*

Referente ao Laravel [17], pode-se concluir que:

- Permite a utilização das funcionalidades mais recentes do PHP.
- Boa documentação.
- Abundância de pacotes e recursos.
- Bom uso do ORM.
- Pouca continuação entre as versões do framework.
- Algumas atualizações podem tornar-se problemáticas.
- Por se tratar de um framework pequeno tem poucas funcionalidades built-in.

- Alguns componentes não se encontram bem desenhados (injeção de dependências é por vezes complexa).

2.5.3 *Django Rest Framework*

Referente ao *Django Rest Framework* [18, 19], pode-se concluir que:

- A configuração e o desenho de arquitetura do framework permite um desenvolvimento rápido.
- Grande número de bibliotecas e "batteries" incluídas.
- Boa documentação.
- Elevada escalabilidade.
- Gestão de base de dados simples.
- Bom uso do ORM.
- Flexível em comparação com outros frameworks.
- Curva de aprendizagem relativamente grande.
- Inexistência de convenções que tornam a configuração do framework mais demorada do que o esperado.

2.5.4 *Spring*

Referente ao *Spring* [20], pode-se concluir que:

- A implementação em POJO (Plain Old Java Object) torna o framework leve.
- Elevada escalabilidade.
- Flexível.
- Configuração consistente.
- Ideal para microsserviços.
- Automatização de testes facilitada devido ao setup predefinido existente.
- É rápido.
- É seguro.
- Boa documentação.
- Mais complexo em comparação com outros frameworks.
- Curva de aprendizagem grande.
- Mecanismo paralelo, ou seja, são oferecidas demasiadas opções ao desenvolvedor podendo originar confusões em relação a que funcionalidades usar e não usar.

3 Ambiente de desenvolvimento

3.1 Ambiente técnico

3.1.1 Hardware

Para a realização deste estágio, foi-me facilitado uma máquina de desenvolvimento com um processador Intel(R) Core(TM) i3-4330 CPU @ 3.50GHz (2 cores, 4 threads) com 8GB de RAM e um disco rígido Samsung SSD 840 EVO 120GB.

3.1.2 Software

No âmbito do estágio, tive a oportunidade de aperfeiçoar e estender o meu conhecimento em ferramentas de software já abordadas no curso e de interagir com outras que eram desconhecidas por mim até à data. Excluindo as ferramentas testadas mas não utilizadas no desenvolvimento da plataforma, que podem ser encontradas na secção anterior, foram utilizadas:

- *Visual Studio Code* como *IDE (Integrated development environment)* onde toda a plataforma foi desenvolvida.
- As bibliotecas *Python Beautiful Soup* e *requests* para *Web Scraping*.
- As bases de dados TimescaleDB, para guardar os dados recolhidos, e *Elasticsearch*, para guardar os *logs* do *scraper*.
- O *framework Django Rest Framework* para a criação da *API*.
- *Docker* para o *deployment* da plataforma.

3.2 Metodologia de trabalho

Não tendo sido aplicado diretamente qualquer processo de desenvolvimento de software, o mais aproximado foi o modelo em cascata, ou *Waterfall*, na medida em que existiu uma sequência cronológica para a implementação dos requisitos da plataforma que foram inicialmente definidos. Por outro lado, com o decorrer do estágio foram analisadas e testadas ferramentas e metodologias que não estavam primeiramente planeadas assim como implementadas novas funcionalidades. A calendarização das tarefas acabou também por ser flexível, dependendo do progresso feito em cada semana e do que era discutido em cada reunião semanal com os tutores.

4 Trabalho desenvolvido

No dia 01/03/2021, sucedeu-se a primeira reunião com os tutores. Nesta reunião foi feito um breve resumo do trabalho a ser realizado durante o estágio, seguindo-se a especificação dos requisitos da plataforma. Nesta fase, os requisitos da plataforma não eram finais, pelo que algumas alterações e adições acabaram por ser exercidas com o decorrer do estágio. A primeira tarefa foi definir qual as ferramentas existentes para o desenvolvimento da primeira parte do sistema, o *scraper*. Este componente, como o nome indica, seria o responsável pela extração dos dados climáticos necessários. As ferramentas encontradas por sua vez teriam de ser alvo de testes de modo a perceber qual a mais indicada para o pretendido. Os resultados da pesquisa e análise foram discutidos na semana seguinte.

No dia 08/03/2021, foi debatido com os tutores os resultados da primeira análise e experiência com as ferramentas de software destinadas à criação do *scraper*. Apesar de ter existido uma certa preferência pelo *Beautiful Soup* no fim da primeira análise, a semana seguinte serviria para aprofundar mais o conhecimento sobre as ferramentas referidas e para se realizarem mais testes de forma a chegar-se a uma conclusão. A página web a ser rastreada, relativa à estação meteorológica do ICT da Universidade de Évora, também deveria ser foco de estudo.

No dia 15/03/2021, após o trabalho feito durante a semana anterior, chegou-se a um consenso relativamente à escolha da ferramenta a ser utilizada na composição do *scraper* optando-se pelo *Beautiful Soup*. Com esta ferramenta, foi possível gerar um esboço inicial do planeado para o *scraper*, em que se conseguiu obter os dados desejados. Até à reunião da semana seguinte, ficou acordado com os tutores a elaboração de uma pesquisa sobre o tipo de base de dados a ser utilizado para a persistência dos dados. Também deveria começar a ser visto o tipo de arquitetura da plataforma, nomeadamente se faria sentido a implementação de microsserviços.

No dia 22/03/2021, foi discutido com os tutores a pesquisa a respeito das bases de dados. Esta pesquisa abrangia essencialmente características de bases de dados relacionais e não relacionais e as suas diferenças. Após o *feedback* recebido, foi determinado que, dada a natureza dos dados, uma solução *Time Series* para o tipo de base de dados poderia ser mais adequada. O trabalho da semana seguinte seria centrado em investigar esta solução de modo a chegar a uma conclusão. Além disso, foi debatido o uso de uma arquitetura de microsserviços. Dado ao número de componentes pensados para a plataforma e o desejo de a manter o mais modular possível, decidiu-se avançar para a criação do sistema utilizando microsserviços. No entanto, teriam de ser ponderados nas semanas seguintes os serviços a serem implementados ficando um diagrama da arquitetura por fazer, de modo a ser apresentado na semana seguinte. Por último, a arquitetura da *API* que disponibiliza os dados também deveria ser cogitada.

No dia 29/03/2021, foi decidida a base de dados a ser utilizada, elegendo-se a *TimescaleDB*. Quanto à arquitetura da *API*, analisou-se as arquiteturas do tipo *REST (Representational state transfer)* e *GraphQL*. Apesar de ambas serem opções muitíssimo válidas, houve uma ligeira inclinação para a implementação da arquitetura *REST* dada à sua simplicidade em relação ao *GraphQL* e tendo em vista a clareza dos dados a serem disponibilizados. Contudo, uma decisão final foi adiada para a semana seguinte. Ainda foi debatido o diagrama de microsserviços da plataforma. Ficou marcado para até à reunião da semana seguinte retificar algumas imperfeições do diagrama e efetuar uma investigação mais completa sobre microsserviços, procurando modelos semelhantes ao que se intencionava implementar.

No dia 05/04/2021, foi determinado o uso da arquitetura *REST* para o desenvolvimento da *API*. O desenho da arquitetura de microsserviços também foi finalizada.

Após a fase de levantamento de soluções que decorreu durante as semanas referidas, iniciou-se a implementação do *scraper*, o primeiro serviço da plataforma. A primeira parte da implementação durou duas semanas. A segunda parte da implementação ficou dependente do inicio do desenvolvimento da *API*, o segundo serviço da plataforma.

Passou-se portanto para a criação do modelo de dados. Esta tarefa durou uma semana.

A próxima tarefa desempenhada foi a conceção da *API RESTful* para a disponibilização dos dados. Foi então necessário escolher qual a ferramenta de software a ser utilizada. Para tal, foi acordado com os tutores a realização de uma análise dos *frameworks* existentes para o efeito. A ideia era de estes possibilitarem o mapeamento dos dados por *ORM (Object–relational mapping)* e que seguissem os princípios da arquitetura *MVC (Model–view–controller)*. Esta tarefa durou duas semanas.

Ocorrida a análise dos *frameworks* para a criação da *API*, incialmente optou-se pelo *Spring* dada a

sua rapidez, eficiência e por funcionar bem com o *JPA (Java Persistence API)*, contribuindo para uma programação mais simples no que diz respeito à interação com a base de dados. Porém, uma característica peculiar da base de dados escolhida, a *TimescaleDB*, que era a necessidade de criação de uma *hypertable*, tornava o uso do *JPA* inviável dado a questões de falta de compatibilidade. Desse modo, optou-se por trabalhar com o *Django Rest Framework*, especializado em criação de *APIs*. A implementação da *API* demorou duas semanas.

Finalizada a implementação da *API*, realizou-se os ajustes necessários ao *scraper*, de forma a permitir o envio dos dados para a própria *API*.

O *deployment* da plataforma foi efetuado utilizando *Docker* [21]. Através do *docker-compose*, foi possível executar os serviços *scraper* e *API*, e as respetivas bases de dados, cada um num *container*.

4.1 Descrição detalhada

Nesta secção, é explicado ao detalhe o processo de implementação e o funcionamento dos dois principais serviços da plataforma: o *Scraper* e a *API*.

4.1.1 *Scraper*

A implementação do scraper foi relativamente simples, sendo extraídos os dados reais de temperatura, velocidade do vento, direção do vento (em graus e cardinal) e a percentagem de humidade das localidades Évora, Mitra e Portel.

Sempre que ocorre um *scrap* ao website, é enviado um log para a base de dados *Elasticsearch* [22], uma base de dados especializada em guardar documentos. Cada log é composto por uma mensagem que indica se o *scrap* foi ou não efetuado com êxito, a data em que occoreu e o *url* do site rastreado.

Visto que o scraper deveria funcionar periodicamente, de 10 em 10 minutos neste caso dado que é o intervalo de tempo que o site atualiza os dados, utilizou-se a biblioteca *schedule* do *Python* para a implementação desta funcionalidade.

4.1.2 Modelo de dados e *API*

O modelo de dados, definido utilizando o *Django Rest Framework*, é composto por duas entidades:

- ***Sensor***: Contém a informação de um sensor. Tem como campos o tipo de dado climático medido (temperatura, humidade,...), a localização do sensor e a unidade da medição.
- ***Medição***: Contém a informação de uma medição. Tem como campos o valor da medição, a data da medição e o sensor que lhe está associada.

A relação entre Sensor e Medição é de *One-to-Many*, ou seja, um sensor pode ter várias medições sendo que uma medição é exclusiva de um sensor.

No que diz respeito à *API*, foi implementado os seguintes *endpoints* de acesso aos dados recolhidos:

- ***'sensors/'***: Suporta a operação stateless GET. Disponibiliza todos os sensores existentes.
- ***'sensors/localizations'***: Suporta a operação stateless GET. Disponibiliza as localizações existentes.
- ***'sensors/[localization]'***: Suporta a operação stateless GET. Disponibiliza todos os sensores existentes em [localization].
- ***'sensors/[localization]/[type]'***: Suporta a operação stateless GET. Disponibiliza todos os sensores do tipo [type] existentes em [localization].
- ***'sensors/[localization]/[type]/measurements'***: Suporta as operações stateless GET e POST, esta última sujeita a autenticação. Disponibiliza todas as medições dos sensores do tipo [type] existentes em [localization] quando o request é do tipo GET. Adiciona medições referentes ao sensor ao sistema quando o request é do tipo POST.

5 Avaliação crítica

Neste estágio foi me possível tanto solidificar e aplicar os conhecimentos que adquiri ao longo do curso de Engenharia Informática como aprender e obter experiência em novas metodologias, arquiteturas e ferramentas de software. A constante análise em cada etapa do desenvolvimento da plataforma bem como o levantamento de soluções para cada funcionalidade contribuiu fortemente para esta aprendizagem.

Apesar do estágio não me ter conseguido proporcionar a experiência de trabalhar em contexto empresarial ou de trabalhar numa equipa, acredito que as discussões semanais com os tutores sobre cada fase do projeto tenha emulado esse ambiente da forma possível.

A necessidade de ter cumprir prazos curtos, principalmente semanais, foi desafiante, exigindo uma melhor gestão do tempo da minha parte.

Como nota final, tenho a certeza que este estágio contribuiu para o meu enriquecimento pessoal e profissional, dando-me boas bases para o futuro.

Referências

[1] Scrapy. https://docs.scrapy.org/en/latest/.

[2] Selenium. https://www.selenium.dev/documentation/en/.

[3] Beautiful Soup. https://www.crummy.com/software/BeautifulSoup/bs4/doc/.

[4] AWS Amazon. https://aws.amazon.com/relational-database/.

[5] Oracle. https://www.oracle.com/pt/database/what-is-a-relational-database/.

[6] PostgreSQL. https://www.postgresql.org/docs/current/index.html.

[7] MongoDB. https://www.mongodb.com/non-relational-database.

[8] AWS Amazon. https://aws.amazon.com/nosql/.

[9] TimescaleDB. https://docs.timescale.com/.

[10] InfluxDB. https://www.influxdata.com/time-series-database/.

[11] Microservices. https://microservices.io/.

[12] Red Hat. https://www.redhat.com/en/topics/api/what-is-a-rest-api.

[13] GraphQL. https://graphql.org/.

[14] Microsoft. https://docs.microsoft.com/en-gb/aspnet/core/mvc/overview?view=aspnetcore-5.0.

[15] Hibernate. https://hibernate.org/orm/what-is-an-orm/.

[16] Ruby on Rails. https://rubyonrails.org/.

[17] Laravel. https://laravel.com/docs/8.x.

[18] Django. https://docs.djangoproject.com/en/3.2/.

[19] Django Rest Framework. https://www.django-rest-framework.org/.

[20] Spring. https://spring.io/.

[21] Docker. https://docs.docker.com/samples/django/.

[22] Elasticsearch. https://www.elastic.co/guide/en/elasticsearch/reference/current/index.html.