Criação de um sistema de consolas de monitorização

UNIVERSIDADE DE ÉVORA
ESCOLA DE CIÊNCIAS E TECNOLOGIA
DEPARTAMENTO DE INFORMÁTICA

Licenciatura em Engenharia Informática
Estágio-Projecto 2021-2022

André da Silva Rato (45517)

Orientador na empresa: Danilo Andrade
Orientador no departamento: Pedro Salgueiro

Trabalho desenvolvido na empresa DECSIS, Sistemas de Informação S.A. no âmbito da disciplina de Estágio-Projeto da Licenciatura em Engenharia Informática.

Évora, 16 de junho de 2022

Conteúdo

1 Introdução

O presente documento enquadra-se no âmbito da Licenciatura em Engenharia Informática na Universidade de Évora, e tem por objetivo dar a conhecer o Estágio Curricular realizado no 6.º Semestre na empresa DECSIS, para completar o grau de Licenciado do curso referido.

O estágio tem como finalidade a concretização dos objetivos descritos posteriormente no subcapítulo 1.2, aplicando alguns conceitos adquiridos no decorrer do curso e durante o próprio estágio.

1.1 Enquadramento

O estágio foi realizado na empresa DECSIS, Sistemas de Informação S.A., mais concretamente na Divisão de Inovação e Desenvolvimento (DID).

A DID está dividia em várias equipas, tais como a equipa de *front-end* onde eu me inseri. O trabalho que desempenhei durante o estágio insere-se maioritariamente na área de *front-end*, passando também por fases de *back-end*, para a criação dos servidores, e de *DevOps*, em momentos de *deployment* de serviços.

O estágio foi realizado em regime misto, tendo a possibilidade de decidir quando trabalhar remota ou presencialmente.

Dado que não foi a primeira vez que realizei um estágio na DECSIS, a comunicação com a equipa sempre foi fácil, acabando assim por ter um alto nível de facilidade em ter todas as minhas dúvidas esclarecidas.

1.2 Objetivos

O estágio realizado teve como objetivo implementar um sistema de consolas de monitorização de um sistema de recolha de dados de fontes externas, utilizando React.js para as componentes de *front-end*. Este sistema teria de comunicar com os *back-end*s dos vários serviços de recolha de dados, que incluem tecnologias de *streaming* e *messaging* (Kafka e Redis), bases de dados (PostgresSQL e MongoDB), sistemas de gestão documental (Alfresco) e agregadores de *logs* (Loki).

Como objetivos secundários, foi proposta a documentação de todo o sistema, explicando as várias componentes que o constituem, como estas comunicam entre si, e outros aspetos que eu considerasse relevantes, tal como a configuração dos *back-end*s a serem utilizados pelo sistema.

1.3 Contribuições

Como foi referido anteriormente, o sistema teria de comunicar com todos os serviços. Estes serviços já haviam sido implementados, sendo que cada um possuía, pelo menos, uma API com todos os *endpoints* necessários para a utilização do serviço, e documentação da mesma. Isto facilitou o meu trabalho, tendo apenas sido necessário a leitura da documentação das APIs dos serviços, para poder começar a utilizar as mesmas.

1.4 Estrutura do documento

O presente relatório é constituído por 6 capítulos.

O primeiro capítulo introduz o contexto do estágio, fazendo um enquadramento do mesmo na Licenciatura de Engenharia Informática e apresentando os seus objetivos e contribuições.

O capítulo seguinte apresenta o contexto empresarial onde o estágio decorreu. Para além de um breve resumo histórico da empresa DECSIS, é também apresentada a localização das cinco sedes existentes em Portugal e da sede existente em Espanha.

O terceiro capítulo detalha as duas ferramentas-chave utilizadas no decorrer do estágio, explicitando quais as razões que as levaram a ser escolhidas.

No quarto capítulo, todo o ambiente técnico, mais precisamente o *hardware* e *software* utilizados, é apresentado. Além disso, todo o ambiente aplicacional em que o trabalho realizado se inseriu e a metodologia de trabalho utilizada são introduzidos.

O quinto capítulo apresenta, não só uma explicação de todo o projeto, mas também uma descrição detalhada dos processos-chave do mesmo, tais como o protocolo utilizado para tratar da segurança do servidor principal (autenticação e autorização), o processo de criação da API do mesmo, e até mesmo a estrutura do próprio sistema.

O sexto e último capítulo pretende demonstrar, de modo geral, os conceitos aprendidos e as adversidades encontradas durante o desenvolvimento do projeto, e algumas opções de melhoria do sistema.

No final do presente documento, encontra-se um glossário com a definição de alguns termos e expressões importantes e o conjunto de referências utilizadas para a elaboração do mesmo.

2 Ambiente empresarial

A DECSIS é uma empresa prestadora de serviços nas áreas das Tecnologias de Informação e das Comunicações. Os dois principais fundadores da DECSIS foram o Eng.º Manuel Silva e o Dr. Laurindo Oliveira.

Manuel Silva fez todo o seu percurso profissional trabalhando em multinacionais do setor, tendo iniciado o projeto DECSIS na *Digital Equipment Corporation* (DEC), onde desenvolveu a sua atividade durante 8 anos (1987 – 1994). Já o percurso profissional de Laurindo Oliveira dividiu-se em duas atividades fundamentais - Ensino Superior, nas áreas de Contabilidade e Gestão, e a gestão do grupo de empresa ligado à Expandindústria, empresa vocacionada para estudos, projetos e gestão empresarial.

No dia 1 de julho de 1994, foi criada a DECSIS, Sistemas de Informação Lda., uma empresa com sede na cidade do Porto e com capital social de 24.939,90€, vocacionada para a prestação de serviços TIC para a multinacional DEC [1].

Em 2008, adotou a denominação de DECSIS, Sistemas de Informação S.A., após ter passado a Sociedade Anónima, e tendo aumentado o seu capital social para 50.000,00€. Em 2012, a empresa mudou a sua sede social para Évora.

Figura 1: Localização das sedes da DECSIS

Atualmente, é líder do Grupo DECSIS, grupo constituído por empresas que operam em áreas complementares de especialização em TIC [2]:

Figura 2: Empresas do Grupo DECSIS e os seus anos de criação

3 Estado da arte

Foram utilizadas duas ferramentas-chave no desenvolvimento de todo o sistema, o React.js e o Node.js. Ao longo deste capítulo serão apresentados os aspetos relevantes que levaram à utilização das mesmas.

3.1 Node.js

A primeira ferramenta é o Node.js, um ambiente *open source* de programação de servidores que corre em várias plataformas, como o Windows, Linux, entre outros. Esta ferramenta recorre a JavaScript para a sua implementação [3].

3.1.1 Porquê utilizar Node.js?

Só por utilizar JavaScript, podemos considerar que é uma excelente opção, no que toca a criar servidores que possam comunicar com várias APIs em simultâneo e exportar *front-end*s criados em React.js.

Na figura seguinte, podemos verificar o quão simples é criar um servidor com Node.js e Express.js, uma biblioteca que será referenciada posteriormente, no subcapítulo 5.1.3.

```
1 const express = require('express');
2
3 const api = express();
4 const port = 3000;
5
6 api.listen(port, () => console.log("Server listening on port 3000"));
```

Figura 3: Exemplo de servidor em Node.js

Resumidamente, este é o código básico para a criação de um servidor que ficará em escuta na porta 3000 e que quando iniciado colocará na consola a mensagem `"Server listening on port 3000"`.

É possível afirmar também que o Node.js é uma solução escalável por ser *single-threaded*, conseguindo tratar de um número enorme de conexões em simultâneo com alto rendimento, rápida, devido à programação assíncrona não bloqueante, e de fácil manutenção, visto que o *front-end* e o *back-end* podem ser tratados com apenas uma linguagem, o JavaScript.

É ainda possível utilizar um grande conjunto de pacotes (bibliotecas) feitos pela comunidade, agilizando o processo de desenvolvimento.

Utilizando a simples tarefa de abrir um ficheiro e enviar o seu conteúdo para o cliente como exemplo, podemos obter os seguintes passos se for utilizada outra tecnologia, como PHP ou ASP [3]:

1. o pedido é enviado pelo cliente e recebido pelo servidor;
2. o servidor envia a tarefa para o sistema de ficheiros;
3. o servidor fica em espera, até que o sistema abra e leia o ficheiro;
4. o servidor retorna o conteúdo do ficheiro para o cliente;
5. o servidor fica apto a receber mais pedidos.

No entanto, o Node.js trata este processo de forma diferente:

1. o pedido é enviado pelo cliente e recebido pelo servidor;
2. o servidor envia a tarefa para o sistema de ficheiros;
3. o servidor fica apto a receber mais pedidos;
4. assim que o sistema de ficheiros abra e leia o ficheiro, o servidor retorna o conteúdo para o cliente.

Outros aspetos importantes que levaram à escolha do Node.js para ferramenta-chave do sistema foram os seguintes:

- consegue gerar páginas de conteúdo dinâmico;
- permite, como já foi referido, criar, abrir, escrever, apagar e fechar ficheiros a partir do servidor;

- consegue obter dados a partir de formulários;
- consegue interagir diretamente com bases de dados, podendo assim adicionar, apagar ou alterar dados das mesmas.

3.2 React.js

A segunda ferramenta-chave é o React.js, uma biblioteca de JavaScript utilizada para criar interfaces de utilizador (UI) através de componentes [4]. Esta biblioteca está descrita no subcapítulo 5.1.2.

3.2.1 Porquê utilizar React.js?

Para além de ser uma ferramenta muito utilizada na DECSIS, o React.js é rápido, flexível e de fácil manutenção.

A utilização desta biblioteca permite que os desenvolvedores utilizem partes individuais das suas *apps*, tanto no lado do servidor, como no lado do cliente, o que aumenta a velocidade do processo de desenvolvimento.

Ao comparar o React.js com outras bibliotecas de *front-end*, é possível verificar que o código é mais fácil de manter por ser mais flexível, devido à sua estrutura divida em componentes.

Por utilizar um programa virtual DOM e uma renderização *server-side*, o React.js permite que aplicações complexas sejam executadas de maneira extremamente rápida.

Como foi referido, o React.js utiliza componentes para criar as suas páginas *web*. Esta característica permite que os desenvolvedores não escrevam o mesmo código várias vezes [5].

Durante o desenvolvimento de uma *app* utilizando React.js, é necessário escolher entre [6]:

- **JSX -** **J**ava**S**cript**X**ML: uma extensão de JavaScript utilizado para React.js que permite ao programador criar e renderizar componentes, utilizar CSS, etc.;
- **TSX -** **T**ype**S**cript**X**ML: uma versão de JSX que adiciona tipos estáticos aos dados no código JavaScript.

No trabalho realizado, foi adotado *JSX* por ser utilizado pela equipa de *front-end*.

Para que esta sintaxe fosse compilada para código JavaScript "regular", foi necessário recorrer à utilização de **Babel.js**, um compilador JavaScript que traduz linguagens de *markup* ou programação em JavaScript [4].

4 Ambiente de desenvolvimento

Nesta secção, será apresentado todo o ambiente de desenvolvimento em que o estágio esteve envolvido, tal como o *hardware* e *software* utilizados, em que ambiente aplicacional o trabalho foi enquadrado e qual foi a metodologia de trabalho utilizada durante o desenvolvimento do trabalho de estágio.

4.1 Ambiente técnico

4.1.1 Hardware

A empresa possui uma instalação local da plataforma GitLab, onde todos os projetos estão guardados, sendo possível fazer a gestão, não só do código dos projetos, mas também das *pipelines* de CI/CD. Foram então criadas as minhas credenciais de acesso, para eu ter acesso à ferramenta e poder documentar toda a informação relativa ao meu trabalho.

Para além disso, foi ainda necessário arranjar uma forma de, quando estivesse a trabalhar remotamente, poder aceder aos serviços através da rede interna da empresa. Com as credenciais que já me tinham sido facultadas, foi-me possível configurar e aceder à VPN da empresa.

4.1.2 Software

Durante o período que estive a estagiar na DECSIS, usei várias ferramentas de software, sendo que algumas delas eram novas para mim.

- **Docker**

  O Docker é uma tecnologia de containerização para criação e uso de *container*s. Com a utilização desta ferramenta, é possível lidar com os *container*s como se fossem máquinas virtuais modulares e extremamente leves.

  Esta ferramenta, à semelhança de outras ferramentas de containerização, fornece um modelo de *deployment* com base em Docker *image*s [7][8].

  No trabalho realizado, foi utilizado para correr, num ambiente isolado, todos os programas e aplicações desenvolvidos.

- **Figma**

  O Figma é uma ferramenta online gratuita para *design* vetorial de interfaces e protótipos. Permite a realização de trabalho colaborativo com alterações em tempo real e a sua utilização torna-se facilitada por ser operada através do próprio *browser*, não existindo assim a necessidade de instalar um novo software [9][10].

  No trabalho realizado, o Figma foi a ferramenta que permitiu a criação das *mock*s para todo o conjunto de *front-end*s, facilitando a semelhança entre eles. Esta ferramenta permitiu ainda agilizar o trabalho de implementação, pois, através da utilização do *plugin* *Overlay*, foi possível exportar os *designs* criados para código, em ficheiros JSX e SCSS.

- **GitLab**

  O GitLab é um gestor de repositórios de *software* baseado em git. Possui suporte a *Wiki*, gestão de tarefas e *pipelines* de CI/CD [11].

  No trabalho realizado, foi utilizado para guardar o código do sistema, reportar o trabalho desenvolvido, gerar tags, entre outros.

- **WebStorm e PhpStorm**

  O WebStorm e o PhpStorm são duas ferramentas da JetBrains, uma empresa fornecedora de *softwares* de ponta para desenvolvedores e gestores de projeto. Estes *softwares* são altamente inteligentes e permitem um fluxo de trabalho bastante eficiente.

  O WebStorm é um ambiente de desenvolvimento integrado para JavaScript e tecnologias relacionadas. Como outros IDEs da JetBrains, o WebStorm torna o desenvolvimento mais agradável, automatizando a rotina de trabalho e ajudando o programador a lidar com tarefas complexas com facilidade [12].

O PhpStorm é um IDE para PHP. Fornece prevenção de erros *on-the-fly* (durante o desenvolvimento), um sistema de *autocomplete* e *refactoring* de código altamente inteligente, funcionalidades de *debugging* e um editor estendido de HTML, CSS e JavaScript [13].

No trabalho realizado, estas ferramentas foram utilizadas em conjunto de modo a ter vários projetos abertos e a correr em simultâneo.

- **Postman**

  O Postman é uma ferramenta que dá suporte à documentação das requisições feitas por APIs. Possui um ambiente para a documentação, execução de testes de APIs e requisições em geral [14].

  No trabalho realizado, o Postman permitiu-me, não só testar e ter melhor conhecimento das APIs dos *back-end*s dos serviços associados aos diversos *front-end*s, mas também testar os próprios *endpoints* da API criada para servir de servidor central do sistema.

- **Rancher**

  O Rancher é uma plataforma *open source* de gestão da infraestrutura de Docker e Kubernets em produção. Permite efetuar o *deployment* de *apps* através do Docker [15].

  No trabalho realizado, permitiu que o processo de *deployment* e gestão das tags utilizadas fosse simplificado, possibilitando assim uma agilização da configuração e do *deployment* dos *front-end*s e servidor.

4.2 Ambiente aplicacional

Como referido anteriormente, o trabalho de estágio teve como objetivo criar um sistema de *front-end*s para alguns dos serviços internos da empresa.

Sabendo que cada um dos serviços tinha uma finalidade diferente (alguns para configuração de processos de recolha de dados, outros apenas com o intuito de transformar dados já recolhidos, e até mesmo para exportá-los para plataformas de armazenamento), o sistema teria que conseguir comunicar com todos eles, de forma transparente aos utilizadores dos vários clientes.

Na figura seguinte, é possível ver com que APIs já existentes o sistema comunica:

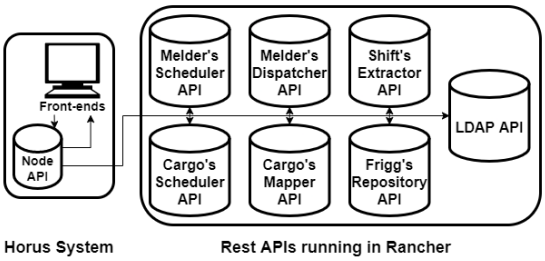

Figura 4: Esquema simplificado das conexões do sistema implementado com as APIs já existentes

Cada API desempenhava uma função essencial em cada serviço:

- **Melder's Scheduler API:** responsável por receber os pedidos relacionados com configuração da extração de dados, por parte do serviço Melder;
- **Melder's Dispatcher API:** responsável por fazer a gestão dos processos subscritos provenientes do Melder;
- **Shift's Extractor API:** responsável por receber pedidos de configuração da extração de dados realizada pelo serviço Shift;
- **Cargo's Scheduler API:** responsável por receber pedidos com a configuração da extração de dados através do serviço Cargo;
- **Cargo's Mapper API:** responsável por receber pedidos que definem onde os dados extraídos pelo Cargo são guardados;

- **Frigg's Repository API:** responsável por gerir as instruções ETL;
- **LDAP API:** responsável pela gestão dos acessos, verificando a sessão através das credenciais enviadas no pedido.

Cada serviço tem o seu alvo de extração, entre eles *websites* e repositórios de dados.

4.3 Metodologia de trabalho

Visto que não foi a primeira vez que realizei um estágio na empresa, não foi preciso passar por um processo de adaptação às metodologias de trabalho da DECSIS. A empresa recorre a métodos ágeis para fazer a gestão do trabalho, utilizando a técnica *scrum*.

Todos os dias ocorre uma reunião com todos os elementos da equipa DID, onde cada um indica o que fez no dia anterior, informa acerca do que irá estar a trabalhar no dia seguinte, ou até mesmo para expressar pequenas dúvidas e pedir ajuda ou opinião aos colegas.

Todo o processo de desenvolvimento estava divido em *sprints*. Cada *sprint* era constituído por um conjunto de tarefas escolhidas de modo a que, no final do *sprint*, se conseguissem ver funcionalidades parciais implementadas. Após cada *sprint* era feita uma reunião com o intuito de decidir quais as tarefas que integrariam a fase de desenvolvimento seguinte.

Como já foi referido, o GitLab foi a ferramenta utilizada para gerir todos os projetos. Foi criado um repositório para o projeto principal, responsável por agregar todo o código e documentação do servidor principal e do *front-end* do mesmo, e vários outros repositórios para os subprojetos, para cada um dos *front-end*s necessários. Estes subprojetos não foram criados no projeto principal, mas sim nos projetos dos serviços aos quais se relacionavam.

A seguinte figura pretende eliminar quaisquer dúvidas que existam no que toca à organização, por repositórios, do projeto:

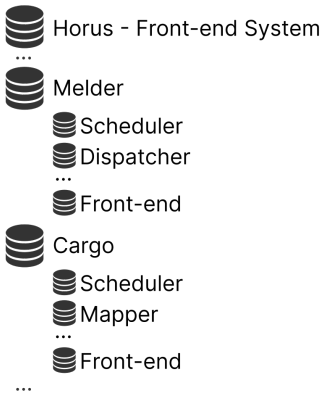

Figura 5: Esquema simplificado da organização, por repositórios, do projeto

O trabalho era planificado gradualmente, ou seja, à medida que eu ia avançando no desenvolvimento do sistema novas tarefas iam surgindo, independentemente do tempo que estas levassem. As tarefas de aprendizagem eram mais demoradas, sendo que demorei cerca de uma semana a perceber como funcionam as tecnologias que foram utilizadas (React.js, Node.js, Figma, etc.). As tarefas de implementação demoravam entre meio dia e dois dias, sendo que, devido à minha pouca experiência com React.js e Webpack.js, algumas delas chegaram a demorar três dias. No final, ainda fiquei com uma semana disponível para começar a elaborar o presente relatório.

Através das credenciais que me foram fornecidas no início do meu estágio, foi possível aceder a um dos canais de comunicação que a empresa usa, o Mattermost [16], um serviço de *chat open source*. Mesmo que, diariamente, os colaboradores da DECSIS não usem este *software*, as minhas dúvidas sempre foram esclarecidas de forma rápida e eficaz.

5 Trabalho desenvolvido

No início do estágio, foram-me explicados os objetivos principais, quer a curto, quer a longo prazo.

Como alguns conceitos e ferramentas eram novas para mim, ficou estipulado que os primeiros dias de estágio seriam utilizados para realizar algumas formações e adquirir alguns conceitos novos:

- React Course
  - Fundamentals
  - Hooks
  - Context API
  - React Router
  - Custom Hooks
- Webpack Course
  - Fundamentals
  - Babel plugin
  - Style loaders
  - Minimal HTML generation
  - React hot refresh
- Redux Course
  - Fundamentals
  - Redux as a store
  - Redux dispatcher
- Axios Course
  - Fundamentals
  - Async/await pattern
  - CRUD functions

Fui aplicando, na prática, os conceitos que fui aprendendo ao longo de todo o processo de aprendizagem e, no final, pude colocar todo o código resultante dessa prática num repositório no GitLab da empresa.

Após esta fase de aprendizagem, reuni-me com o meu supervisor e foram-me colocadas algumas questões, de modo a que todas as minhas dúvidas fossem esclarecidas, e de modo a que o meu supervisor pudesse entender se eu já estaria suficientemente à vontade com as novas tecnologias para avançar para a próxima fase do trabalho.

A fase seguinte passou por criar um repositório para o projeto no GitLab, configurar o projeto como um projeto Node.js, adicionar-lhe a biblioteca Express.js e criar a API que iria possuir todos os *endpoints* que seriam acedidos pelos *front-ends*, *endpoints* este que reencaminhariam os pedidos para as APIs respetivas. De modo a deixar o sistema o mais seguro possível, criei alguns *endpoints* que serviriam para ações de autenticação e verificação de sessão:

- um *endpoint* que valida as credenciais submetidas, iniciando uma sessão e gerando dois *tokens*, um de acesso, e um de sessão;
- um *endpoint* que valida o *token* de acesso, utilizado para aceder aos vários *endpoints* (definido com um período de vida de 5 minutos);
- um *endpoint* que valida o *token* de sessão, utlizado para gerar novos *tokens* de acesso, se válido (o perído de vida corresponde ao período de vida da sessão;
- um *endpoint* que termina a sessão do utilizador.

Depois, entrei no ciclo de desenvolvimento do sistema, um ciclo composto por dez fases e que se repetiu quatro vezes:

- **1ª. fase: Conhecer o serviço**

  Nesta fase, o objetivo principal era entender qual a função do serviço, quais as componentes que o envolvem, qual era a informação que deveria ser apresentada no seu *front-end* e conhecer a própria API. Este processo foi realizado através de:

  - leitura da *Wiki* das várias componentes do serviço;
  - análise das ligações estabelecidas entre cada um dos componentes;
  - pequenas reuniões com colaboradores da empresa que tivessem conhecimento sobre o serviço.

- **2ª. fase: Criação das *mocks***

  Após entender o serviço, foi necessário começar o processo de criação das *mocks*, um processo que engloba três fases:

  - obter inspiração de imagens, vídeos, ou até mesmo *websites* existentes;
  - criar rascunhos, em papel, de possíveis representações para o serviço pretendido;
  - criar, com a ajuda do Figma, as *mocks* dos vários componentes.

- **3ª. fase: Avaliação das *mocks***

  Depois de terminada a criação das *mocks*, foi necessário que estas fossem avaliadas. Caso não fossem aprovadas, o processo de criação das *mocks* tinha de ser repetido.

- **4ª. fase: Criação e configuração do projeto**

  Nesta fase, foi necessário criar um projeto no GitLab e, após isso, configurar o mesmo como sendo um projeto Node.js que utiliza React.js e Webpack.js.

- **5ª. fase: Criação dos *endpoints* do serviço no servidor principal**

  Nesta fase, todos os endpoints necessários para que o serviço funcionasse a 100% foram implementados. Assim, sempre que fosse preciso fazer um pedido a uma das APIs do serviço, o pedido era feito ao servidor principal, e este reencaminhava-o para a API correta.

  A escolha desta tipologia de servidor, a tipologia *proxy server*, permitiu utilizar o próprio servidor principal como *proxy*, existindo apenas comunicação do *front-end* com apenas um servidor, tornando este um pouco mais seguro.

- **6ª. fase: Adicionar verificação de autorização**

  Após a criação dos *endpoints* foi necessário protegê-los, para que todos eles ficassem seguros e com a sua informação protegida.

- **7ª. fase: Implementação dos componentes**

  Implementar os componentes foi a tarefa seguinte. Inicialmente, os componentes foram implementados de forma isolada. Depois, eram utilizados em conjunto, de modo a criar visualizações completas. Por fim, a comunicação com os *endpoints* do servidor principal era feita, recorrendo à biblioteca Axios.js, apresentada no subcapítulo 5.1.3.

- **8ª. fase: Implementação de testes unitários**

  A criação de testes unitários, não só aos componentes de *front-end* desenvolvidos, mas também aos *endpoints* que iam sendo criado, foi uma tarefa que proporcionou a minimização do número de erros e a redução do código do programa.

- **9ª. fase: Realização de testes funcionais**

  Esta fase não foi desenvolvida por mim, mas sim pela equipa de validação e testagem da DECSIS. Aqui, foram realizados os testes funcionais necessários para validar o funcionamento do *front-end* criado. Estes testes têm duas vertentes, a vertente automatizada e a vertente manual.

- **10ª. fase: Elaboração da documentação**

  Por fim, mas não menos importante, era necessário deixar o código comentado e, para além disso, documentar a *Wiki* do subprojeto.

No fim de cada ciclo, era sempre realizada uma reunião com o meu orientador, o Danilo Andrade, e com o chefe de equipa, o Paulo Caeiro, para validação do trabalho desenvolvido.

Após a implementação de todo o sistema, e já tendo a documentação do mesmo terminada, foi-me pedido que apresentasse, resumidamente, a toda a equipa DID, o trabalho desenvolvido, sendo este recebido com *feedback* muito positivo.

5.1 Descrição detalhada

5.1.1 LDAP - Lightweight Directory Access Protocol

Como método de autenticação para o sistema foi utilizado o LDAP. O LDAP é um protocolo que permite gerir diretorias, ou seja, aceder a bases de dados de utilizadores de uma rede através do protocolo TCP/IP. Geralmente, as bases de dados são relativas a utilizadores, mas também podem ser utilizadas para outros objetivos, como por exemplo, gerir o material de uma empresa.

Este protocolo foi desenvolvido em 1993 pela Universidade de Michigan, nos Estados Unidos. Definindo o método de acesso aos dados no servidor a nível do cliente e não a maneira como os dados estão armazenados, o protocolo LDAP está na versão 3 e foi padronizado pelo IETF (*Internet Engineering Task Force*) [17].

5.1.2 Configuração do projeto

Como foi referido, o projeto foi criado utilizando Node.js, um ambiente de execução JavaScript *server-side* desenvolvido em 2009. Basicamente, o Node.js é um ambiente que permite criar aplicações em JavaScript, de modo a que estas sejam executadas como uma aplicação *standalone* numa máquina, sem depender de um *browser* para a execução do mesmo [18]. Os principais motivos da sua utilização são:

- a alta capacidade de escala;
- a arquitetura;
- a flexibilidade;
- o baixo custo.

Todos estes motivos tornam o Node.js uma boa escolha para a implementação de microsserviços e componentes da arquitetura *serverless*.

Apesar de ser relativamente recente, o Node.js já é utilizado por grandes empresas no mercado da tecnologia, como a Netflix, o LinkedIn e a Uber.

Depois da criação do projeto, foi necessário configurar o React.js, uma biblioteca JavaScript para a criação de UI (*user interfaces*). Esta biblioteca foi criada em 2011 pela equipa de desenvolvimento do Facebook, com o objetivo de otimizar a atualização e a sincronização de atividades simultâneas no feed de notícias na rede social, entre eles *chat*, *status*, lista de contactos, e outros. Esta foi a ferramenta-chave utilizada para a criação dos *front-ends* do sistema.

Como passo final da configuração, temos o *setup* do Webpack.js, um *bundler* (empacotador) de módulos estáticos para aplicações JavaScript modernas. Ao processar a aplicação, o Webpack.js gera um gráfico que mapeia cada módulo e as suas dependências e gera um ou mais pacotes [19].

Através do ficheiro de configuração de Webpack.js é possível definir:

- a *proxy*;
- o *loader* de estilos para SCSS;
- o *loader* de ficheiros para imagens, vídeos, etc.;
- os *plugins* necessários, como por exemplo o Babel.js, um *transpiler* de JavaScript.

5.1.3 Criação da API do servidor principal

Tal como referido, o servidor foi criado utilizando bibliotecas de JavaScript. Para que o servidor pudesse receber e enviar pedidos foram utilizadas duas bibliotecas:

- Express.js, uma *framework* rápida utilizada para o desenvolvimento de aplicações de JavaScript com Node.js [20]. No sistema, foi utilizada para criar o servidor e receber os pedidos feitos ao mesmo;
- Axios.js, um cliente HTTP baseado em *promises* para Node.js. É um cliente isomórfico, ou seja, pode ser executado no navegador e no Node.js com a mesma base de código [21]. Para o estágio, foi utilizado para reencaminhar os pedidos que chegam ao servidor para as APIs respetivas.

5.1.4 Testes unitários

Sendo que todo o sistema seria implementado utilizando bibliotecas de JavaScript, a *framework* Jest.js foi utilizada para implementar e executar os testes unitários. Através de um pequeno ficheiro de configuração, foi possível que os testes realizados não se aplicassem apenas a funções de JavaScript (figura 6), mas também aos componentes de React.js (figura 7).

```
import {isNotEmptyJsonObject} from "../src/utils/json";

test(`JSON object passed as argument can't be an array or an empty object`, () => {
    const emptyObject = {};
    const notEmptyObject = {nome: 'abc'};
    const emptyArray = [];
    const notEmptyArray = ['one', 'two', 'three'];
    expect(isNotEmptyJsonObject(emptyObject)).toBe(false);
    expect(isNotEmptyJsonObject(notEmptyObject)).toBe(true);
    expect(isNotEmptyJsonObject(emptyArray)).toBe(false);
    expect(isNotEmptyJsonObject(notEmptyArray)).toBe(false);
})
```

Figura 6: Exemplo de teste a uma função JavaScript.

```
import React from "react";
import ReactDOM from "react-dom";
import Button from '../src/components/buttons/Button';

test('Button render successfully', () => {
    const div = document.createElement('div');
    ReactDOM.render(<Button/>, div);
    ReactDOM.unmountComponentAtNode(div);
});
```

Figura 7: Exemplo de teste a um componente de React.js.

5.1.5 Estrutura do sistema

O diagrama abaixo representa a estrutura do sistema criado.

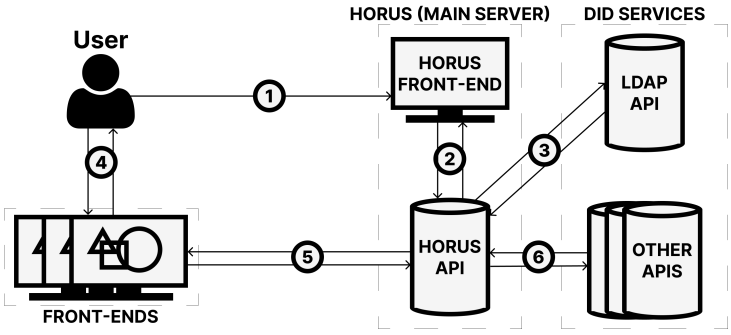

Figura 8: Representação da estrutura do sistema e das conexões estabelecidas entre si.

Como pode ser observado, o sistema pode ser dividido em três setores:

- **Servidor principal**

  Como referido antes, foi utilizada a biblioteca Express.js para a criação do servidor principal. Através dela, foi possível, não só definir os *endpoints* da API, mas também exportar o *front-end* do mesmo.

Na conexão 1, o utilizador acede ao *front-end* exportado pelo servidor principal e tenta autenticar-se. A conexão seguinte, a conexão 2, faz um pedido, enviando as credenciais do utilizador para o servidor principal, e este, através da conexão 3, valida as credenciais submetidas, enviando um pedido à API do LDAP.

- ***Front-ends* dos serviços**

  Foram criados *front-ends* para quatro serviços da DID. Para que um utilizador aceda a um dos *front-ends* implementados (conexão 4), é necessário que esteja autenticado no *front-end* do Horus (servidor principal). Qualquer ação realizada pelo utilizador em qualquer um dos *front-ends* desencadeia um pedido que é feito à API do Horus, como se pode verificar na conexão 5.

- **APIs dos serviços**

  Após receber um pedido, a API do Horus reencaminha-o para a API correta (conexão 6).

  Depois de processar o pedido, a API envia a resposta ao Horus, e este reencaminha-a para o *front-end* que enviou o pedido inicial, disponibilizando assim toda a informação necessária e requerida pelo utilizador.

6 Avaliação crítica

A realização do estágio foi uma experiência que me enriqueceu, não só a nível profissional, permitindo-me consolidar alguns dos conceitos obtidos no decorrer da licenciatura, mas também a nível pessoal, através da aprendizagem de novas competências relativas ao trabalho de equipa e ao *web development*. O decorrer do estágio também me fez entender quais as vertentes da programação que eu mais gosto, apercebendo-me efetivamente que a área que eu mais gosto é o *front-end*.

Durante as primeiras semanas de estágio, surgiram alguns problemas relacionados às ferramentas utilizadas:

- o React.js utiliza estados (objetos internos do React.js utilizados para armazenar dados ou informações sobre um componente) para fazer a gestão do que deve ser renderizado; o estado de um componente pode mudar ao longo do tempo, sendo que quando muda, o componente é renderizado novamente; este conceito foi difícil de entender, sendo que acabava por cometer alguns erros, como por exemplo, tentar ver a alteração do valor de uma variável sem que o componente fosse renderizado novamente;
- devido à complexidade elevada na utilização do Webpack.js, passei algum tempo a tentar encontrar soluções para pequenos erros que iam surgindo, como a utilização de imagens, exportação de estilos, entre outros.

Após uma análise final da versão atual do sistema, é possível afirmar que, em certos aspetos, o mesmo possui alguns problemas de segurança, de carga e, até mesmo, de responsividade dos *front-end*s.

- **Problemas de segurança**

  Sendo um sistema utilizado internamente na DID, e que só pode ser acedido através da rede da mesma, não houve necessidade de implementar mecanismos de segurança para além da autenticação por LDAP.

  Caso fosse necessário expor o sistema para o exterior, ou seja, permitir que fosse possível aceder ao sistema a partir de fora da rede, algumas alterações seriam necessárias para manter a segurança, não só dos dados que passam pelos canais de comunicação do sistema, mas também de toda a infraestrutura da DID:

  - encriptar qualquer dado que circula nas comunicações internas do sistema (credenciais de utilizador, dados de processos, etc.);
  - adicionar *Two Factor Authentication* ao método de autenticação do sistema;
  - minimizar o código HTML, JavaScript e CSS gerado pelo Webpack.js, de modo a impedir que os endereços alvo dos pedidos sejam mostrados no *source code*.

- **Problemas de carga**

  Como foi referido, o servidor foi implementado num ambiente Node.js, em JavaScript, com recurso à biblioteca Express.js. Por si só, o Node.js permite que o impacto da carga seja reduzido, comparativamente a outros ambientes. O problema está que o servidor, por albergar mais que um serviço, vai estar a processar n vezes mais pedidos que um servidor normal, configurado para apenas um serviço (n corresponde ao número de serviços utilizados no servidor).

  Existem algumas alterações que podem ser feitas para controlar a carga excessiva e prevenir possíveis oscilações de desempenho do servidor, como por exemplo:

  - dividir o servidor principal em $n+1$ servidores (um para a gestão da segurança do sistema, conectado à API do LDAP, e os restantes para suportar individualmente cada serviço);
  - diminuir, ao máximo, todo o processamento de dados realizado no servidor.

- **Problemas de responsividade dos *front-end*s**

  Relativamente aos *front-end*s implementados, nenhum deles ficou com uma versão para ecrãs reduzidos, como telemóveis ou *tablets*. Isto aconteceu pois o sistema foi desenvolvido para ser utilizado em ambiente interno, nos computadores de todos os colaboradores da DID.

Este é um problema comum a todos os *front-ends* e pode ser resolvido aplicando uma técnica lecionada numa das unidades curriculares da licenciatura, Tecnologias Web:

  - a utilização de *media queries* nos vários ficheiros de estilização permitiria definir várias formas de apresentação para o mesmo código HTML, tendo em conta, não só o tamanho da janela do utilizador, mas também as dimensões dos vários elementos utilizados, tanto em largura, como em altura.

- **Problemas no *deployment* dos *front-end***

  No processo de *deploy* de todos os *front-ends* implementados, é necessário ter em conta que, após gerar a Docker *image* do projeto, é impossível alterar o seu conteúdo. Isto acontece, pois os ficheiros resultantes da *build* são estáticos.

  Este problema impossibilita o *front-end* de ser alterado após a *build*, ou seja, não é possível definir variáveis de ambiente utilizadas diretamente pelo React.js. A solução para este problema passaria por definir as variáveis de ambiente antes da *build*, o que não é uma solução ótima, tendo sido a única encontrada até ao momento.

Glossário

API (ou *Application Programming Interface*) é um conjunto de normas que possibilita a comunicação entre plataformas, através de uma série de padrões e protocolos [22]. 1, 2, 4, 7, 8, 9, 10, 12, 13, 14

arquitetura *serverless* é uma arquitetura orientada a eventos, sem a necessidade de utilização de servidores [23]. 11

ASP (ou *Active Server Pages*) é o primeiro mecanismo de *script* do lado da Microsoft que permitiu gerar páginas *web* dinamicamente [24]. 4

back-end é a parte de um programa ou aplicação que permite que este funcione e que não pode ser acedido por um utilizador [25][26]. 1, 4, 7

container é uma unidade padrão de software que empacota o código e todas as suas dependências para que o programa seja executado rapidamente e de forma confiável [27]. 6

deploy significa enviar alterações ou atualizaçõs de um ambiente para outro [28]. 1, 6, 7, 15

DevOps conjunto ferramentas, processos e metodologias para equilibrar as necessidades ao longo do ciclo de desenvolvimento de *software*, desde a codificação e *deployment*, até à manuntenção e atualização do mesmo [29]. 1

Docker *image* é um modelo *read-only* que contém um conjunto de instruçõoes para criar um *container* que pode ser executado. Fornece uma maneira conveninente de empacotar programas e ambientes de servidor pré-configurados [30]. 6, 15

DOM (ou *Document Object Model*) é a representação de dados dos objetos que compõem a estrutura e o conteúdo de um documento na *web* [31]. 5

ETL tipo de integração de dados em três etapas - extração, transformação e carregamento [32]. 8

front-end é a parte de um website com a qual o utilizador interaje diretamente. Inclui tudo o que esteja relacionado com a experiência de utilização do mesmo (estilos, imagens, cores, butões, etc.) [25][26]. 1, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15

git é um sistema *open source* de controlo de versões, projetado para lidar com tudo, desde projetos pequenos a projetos muito grandes, com velocidade e eficiência [33]. 6

IDE (ou *Integrated Development Environment*) é um *software* para construção de programas que combina ferramentas comuns de desenvolvimento numa única interface gráfica [34]. 6, 7

IETF (ou *Internet Engineering Task Force*) é um grupo internacional responsável pelo desenvolvimento e promoção dos padrões utilizados na Internet [**?**]. 11

Jest.js é uma biblioteca para a criação, estruturação e execução de testes em JavaScript [35]. 12

Kubernets (ou k8s) é um sistema *open source* para automatizar o *deployment*, o escalamento e a gestão de aplicações e programas containerizados [36]. 7

microsserviço é uma abordagem arquitetónica e organizacional do desenvolvimento de software no qual este é constituído por pequenos serviços independentes que se comunicam utilizando APIs [37]. 11

mock (ou *mockup*) consiste na representação de um determinado projeto, que pode ser feita em escala ou em tamanho real. Simula o tamanho, formato, perspetiva, e outros aspetos de um produto [38]. 6, 10

Overlay é um *plugin* de conversão de componentes criados no Figma para código reutilizável em React.js/Vue.js/HTML [39]. 6

PHP é uma linguagem de *script* utilizada para o desevolvimento web [40]. 4

***pipelines* de CI/CD** (ou *pipelines* de integração e entrega contínuas) consistem numa série de etapas a serem realizadas para a disponibilização de novas versões de um programa [41]. 6

promise é um objeto usado para processamento assíncrono. Representa um valor que pode estar disponível "agora", no "futuro" ou "nunca" [42]. 12

protocolo TCP/IP representa um conjunto de protocolos que permitem que diversos equipamentos que constituem uma rede comuniquem entre si. Este protocolo é estruturado por camadas, na qual cada camada utiliza e presta serviços às camadas adjacentes [43]. 11

proxy é um servidor intermediário que separa os utilizadores finais do *websites* em que navegam [44]. 10, 11

single-threaded processo cujas instruções são executadas sequencialmente [45]. 4

SCSS é a segunda sintaxe de SASS (*Syntactically Awesome Stylesheet*). Esta sintaxe utiliza chavetas em vez de identação. É similar a um ficheiro .css, mas com funcionalidades adicionais, tais como variáveis ou regras dentro de regras (*nested rules*) [46]. 6

scrum metodologa de desenvolvimento baseada na criação de pequenos ciclos de atividade (os *sprints*, com a finalidade de desenvolver tarefas em equipa [47]. 8

tag é um referência mutável para imagens do Docker. Facilitam a extração e execução de imagens e permitem que os autores lançem automaticamente atualizações [48]. 6, 7

transpiler (ou *source-to-source compiler*) é um programa que traduz o código fonte de uma linguagem para outra com o mesmo nível de abstração, o que é diferente de um compilador, cujo *output* possui um nível menor que o *input* [49]. 11

VPN (ou *Virtual Private Network*) descreve a oportunidade de estabelecer uma conexão de rede protegida ao usar redes públicas [50]. 6

Referências

[1] Decsis - quem somos. https://www.decsis.eu/pt/a-decsis/quem-somos/solucoes-ti/. [Online; acessed 25-May-2022].

[2] Decsis - grupo decsis. https://www.decsis.eu/pt/a-decsis/grupo-decsis/a-nossa-historia/. [Online; acessed 25-May-2022].

[3] Node.js introduction. https://www.w3schools.com/nodejs/nodejs_intro.asp. [Online; acessed 25-May-2022].

[4] What is react. https://www.w3schools.com/whatis/whatis_react.asp. [Online; acessed 25-May-2022].

[5] The benefits of reactjs and reasons to choose it for your project. https://www.peerbits.com/blog/reasons-to-choose-reactjs-for-your-web-development-project.html. [Online; acessed 25-May-2022].

[6] What is the difference between .js, .tsx and .jsx in react? https://stackoverflow.com/questions/64343698/what-is-the-difference-between-js-tsx-and-jsx-in-react. [Online; acessed 6-June-2022].

[7] Docker overview — docker documentation. https://docs.docker.com/get-started/overview/. [Online; acessed 26-May-2022].

[8] What is docker? https://www.redhat.com/en/topics/containers/what-is-docker. [Online; acessed 26-May-2022].

[9] What is figma? (and how to use figma for beginners). https://www.theme-junkie.com/what-is-figma/. [Online; acessed 26-May-2022].

[10] Figma(software). https://en.wikipedia.org/wiki/Figma_(software). [Online; acessed 26-May-2022].

[11] Gitlab. https://pt.wikipedia.org/wiki/GitLab. [Online; acessed 26-May-2022].

[12] Webstorm. https://www.jetbrains.com/webstorm/. [Online; acessed 26-May-2022].

[13] What is phpstorm. https://monovm.com/blog/what-is-phpstorm/. [Online; acessed 26-May-2022].

[14] Postman tutorial. https://www.javatpoint.com/postman, note="[Online; acessed 27-May-2022]".

[15] What is rancher? https://www.openlogic.com/blog/what-is-rancher. [Online; acessed 27-May-2022].

[16] Mattermost. https://en.wikipedia.org/wiki/Mattermost. [Online; acessed 27-May-2022].

[17] Lightweight directory access protocol. https://en.wikipedia.org/wiki/Lightweight_Directory_Access_Protocol. [Online; acessed 27-May-2022].

[18] What is node.js and why you should use it. https://kinsta.com/knowledgebase/what-is-node-js/. [Online; acessed 27-May-2022].

[19] What is webpack. https://survivejs.com/webpack/what-is-webpack/. [Online; acessed 26-May-2022].

[20] Express - node.js web application framework. https://expressjs.com/. [Online; acessed 27-May-2022].

[21] Getting started — axios documentation. https://axios-http.com/docs/intro. [Online; acessed 27-May-2022].

[22] Api. https://en.wikipedia.org/wiki/API. [Online; acessed 24-May-2022].

[23] Serverless architecture: What is and how it works. https://www.datadoghq.com/knowledge-center/serverless-architecture/. [Online; acessed 24-May-2022].

[24] What are active server pages (asp)? - definition from techopedia. https://www.techopedia.com/definition/23088/active-server-pages-asp. [Online; acessed 8-June-2022].

[25] Backend. https://techterms.com/definition/backend. [Online; acessed 23-May-2022].

[26] Frontend vs backend. https://www.geeksforgeeks.org/frontend-vs-backend/. [Online; acessed 23-May-2022].

[27] What is container? https://www.docker.com/resources/what-container/. [Online; acessed 28-May-2022].

[28] Software deployment. https://en.wikipedia.org/wiki/Software_deployment. [Online; acessed 26-May-2022].

[29] Devops. https://en.wikipedia.org/wiki/DevOps. [Online; acessed 30-May-2022].

[30] A beginner's guide to understanding and building docker images. https://jfrog.com/knowledge-base/a-beginners-guide-to-understanding-and-building-docker-images/. [Online; acessed 28-May-2022].

[31] Introdução ao dom. https://developer.mozilla.org/pt-BR/docs/Web/API/Document_Object_Model/Introduction. [Online; acessed 27-May-2022].

[32] Etl: o que é e qual a sua importância. https://www.sas.com/pt_br/insights/data-management/o-que-e-etl.html. [Online; acessed 29-May-2022].

[33] Git. https://git-scm.com/. [Online; acessed 27-May-2022].

[34] What is an ide? https://www.codecademy.com/article/what-is-an-ide. [Online; acessed 29-May-2022].

[35] Jest tutorial for beginners: Getting started with javascript testing. https://www.valentinog.com/blog/jest/. [Online; acessed 6-June-2022].

[36] Kubernets. https://kubernetes.io/. [Online; acessed 29-May-2022].

[37] What are microservices. https://smartbear.com/solutions/microservices/. [Online; acessed 28-May-2022].

[38] What are microservices. https://cliquestudios.com/mockups/. [Online; acessed 30-May-2022].

[39] Overlay — design production ready code components. https://www.overlay-tech.com/. [Online; acessed 31-May-2022].

[40] Php: Hypertext preprocessor. https://www.php.net/. [Online; acessed 5-June-2022].

[41] O que são pipelines de ci/cd? https://www.redhat.com/pt-br/topics/devops/what-cicd-pipeline. [Online; acessed 31-May-2022].

[42] Promise - javascript. https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Promise. [Online; acessed 29-May-2022].

[43] Tcp/ip proocols. https://www.ibm.com/docs/en/aix/7.2?topic=protocol-tcpip-protocols. [Online; acessed 27-May-2022].

[44] What is a proxy server and how does it work? https://www.varonis.com/blog/what-is-a-proxy-server. [Online; acessed 28-May-2022].

[45] Single-threaded and multi-threaded processes. https://www.tutorialspoint.com/single-threaded-and-multi-threaded-processes. [Online; acessed 31-May-2022].

[46] What is the difference between css and scss? https://www.geeksforgeeks.org/what-is-the-difference-between-css-and-scss/. [Online; acessed 30-May-2022].

[47] Metodologia scrum: o que é, métodos ágeis e guia prático. https://blog.contaazul.com/metodologia-scrum. [Online; acessed 1-June-2022].

[48] The fundamentals of docker image tags. https://blog.atomist.com/docker-image-tags/. [Online; acessed 27-May-2022].

[49] What is a transpiler? https://blog.bam.tech/developer-news/what-is-a-transpiler. [Online; acessed 27-May-2022].

[50] What is vpn? how it works, types of vpn. https://www.kaspersky.com/resource-center/definitions/what-is-a-vpn. [Online; acessed 31-May-2022].