

## **Relatório de Estágio**

Módulo de gestão de comunicação com periféricos, Asseco PST  
Unidade Curricular Projeto/Estágio

Orientador UMa: Prof. Filipe Quintal

E-mail: [filipe.quintal@staff.uma.pt](mailto:filipe.quintal@staff.uma.pt)

Orientador Asseco PST: Carlos Ramos

E-mail: [Carlos.Ramos](mailto:Carlos.Ramos)

Autor/Orientando:

Ricardo Palmeira

Nº Aluno: 2028619

E-mail: [ricardoalexp17@gmail.com](mailto:ricardoalexp17@gmail.com)

## ÍNDICE:

<b>1. CONTEXTO</b>	<b>5</b>
1.1 Proposta de estágio	5
1.2. Asseco PST	6
1.3 Planeamento	6
<b>2. INTRODUÇÃO</b>	<b>7</b>
2.1. Módulo de gestão de comunicação com periféricos	8
<b>3. SOLUÇÃO</b>	<b>9</b>
3.1. Descrição	11
3.2. Ferramentas utilizadas	12
3.2.1. <i>Ambiente de desenvolvimento</i>	12
3.2.2. <i>Spring Boot Framework</i>	13
3.2.3. <i>MySQL Server</i>	14
3.2.4. <i>Spring Data JPA e Hibernate</i>	14
3.2.5. <i>Swagger / OpenAPI</i>	15
3.2.6. <i>WebSocket / StompJS</i>	16
3.2.7. <i>Java Socket</i>	16
3.2.8. <i>Log4j2</i>	16
3.3 Resumo solução/ferramentas	17
<b>4. IMPLEMENTAÇÃO</b>	<b>17</b>
4.1. Requisitos	17
4.1.1. <i>Requisitos ambientais</i>	17
4.1.2 <i>Requisitos funcionais</i>	18
4.1.3. <i>Requisitos de dados</i>	18
4.2. Estratégia	18
4.2.1. <i>Divisão por módulos</i>	19
4.4. Produto final	20
4.4.1. <i>Conceitos importantes</i>	20
4.4.1.1. <i>SpringApplication</i>	20
4.4.1.2. <i>Spring Beans e injeção de dependências</i>	20
4.4.1.3. <i>Acesso ao Spring Context</i>	21
4.4.2. <i>Servidor</i>	21
4.4.3. <i>Comunicação WebSocket</i>	23
4.4.3.1. <i>Servidor</i>	24
4.4.3.2. <i>Lado do Browser</i>	24
4.4.4. <i>Comunicação Socket</i>	26

4.4.4.1. <i>Lado do Servidor</i>	27
4.4.4.2. <i>Lado do Simulador de Dispositivos</i>	27
4.4.5. <i>API REST</i>	28
4.4.6. <i>Sistema de registo de operações (logs)</i>	29
4.4.7. <i>Acesso à base de dados</i>	30
<b>5. DEMONSTRAÇÃO</b>	<b>32</b>
5.1. Inicialização da aplicação	33
5.2. Conexão de clientes e dispositivos	34
5.3. Teste da API REST	35
5.4. Envio de pedidos	36
5.5. Logs	37
<b>6. DISCUSSÃO</b>	<b>39</b>
<b>7. PROGRESSO</b>	<b>39</b>
7.1. Primeira Fase: Criação da base do projeto:	40
7.2. Segunda Fase: Melhoramentos e Adição de Funcionalidades	41
7.3. Terceira Fase: Adições finais, Correção de Erros e Limpeza de Código	41
7.4. Quarta Fase: Fase Final	42
<b>7. CONCLUSAO</b>	<b>43</b>
<b>8. REFERENCIAS</b>	<b>43</b>

## 1. CONTEXTO

Este documento serve o propósito de relatório do estágio realizado na empresa Asseco PST Funchal no âmbito da unidade curricular Projeto/Estágio da licenciatura em Engenharia Informática da Universidade da Madeira, contendo assim a documentação do planeamento, implementação e análise do projeto desenvolvido sobre o contexto do mesmo.

### 1.1 PROPOSTA DE ESTÁGIO

O projeto proposto pela empresa Asseco PST (*Portuguese Speaking Territories*) e aceite por mim foi o desenvolvimento de um módulo gestor de comunicação com periféricos, tendo sido este o ponto focal de todo o estágio permitindo não só consolidar muitos dos conhecimentos absorvidos durante a licenciatura como também aprender outros novos em áreas e ferramentas não exploradas pela mesma.

Desta forma, apresento tal como referido na proposta de estágio submetida:

#### Motivações:

*Desenvolver uma solução em JAVA capaz de gerir a comunicação entre a solução de Frontend e os respectivos periféricos. A principal tarefa a ser executada no contexto do estágio é o desenho da arquitetura da solução bem como a implementação de webservices, websockets e sockets para a comunicação com a solução de Frontend e com os periféricos.*

#### Objetivos:

1. *Permitir a disponibilização dos serviços de comunicação com periféricos por websocket e webservice, disponibilizando a respetiva documentação dos serviços;*
2. *Permitir a comunicação usando os protocolos wss e https;*
3. *Evoluir a solução para facilitar o desenvolvimento de novos eventos in integrações;*
4. *Evoluir tecnologicamente a solução atual e implementar processos de CI/CD.*

### 1.2. ASSECO PST

Como mencionado anteriormente o tema do estágio realizado foi proposto pela empresa Asseco PST Funchal, tendo sido este realizado com a supervisão e orientação do Eng. Carlos Ramos, um dos *developers* integrantes da equipa dos serviços de *middleware* desta.

A Asseco PST é uma empresa de Tecnologias da Informação especializada no desenvolvimento de software bancário integrada na *Asseco Group*, um dos maiores e mais conceituados fornecedores europeus de *software*. Atuando maioritariamente nos mercados de países africanos de língua oficial portuguesa através do desenvolvimento de soluções para o setor financeiro, serviços de Banking Consulting e serviços de IT Infrastructure entre outros. Tendo isto em conta, a Asseco PST é uma empresa de sucesso com ótimas condições de trabalho e valores profissionais exemplares, tornando-a uma excelente fonte de aprendizagem [1].



*Figura 1: Logotipo Asseco na Asseco PST Funchal.*

Finalizando, resta apenas mencionar que após as reuniões presenciais foi-me permitido realizar nas instalações parte do trabalho no tempo restante do horário laboral, o que foi também um ponto importante da aprendizagem de todo este processo, uma vez que pude observar pessoalmente o ambiente de uma empresa em funcionamento.

O estágio na empresa começou no dia 2 de março e terminou dia 10 de junho.

### **1.3 PLANEAMENTO**

O planeamento do estágio foi realizado de forma a cumprir as etapas e requisitos do mesmo, tendo este sido iniciado no dia 2 de março com a primeira reunião de todos os intervenientes (aluno, orientadores e supervisor do projeto do lado da Asseco PST), terminando dia 19 de julho com o término da escrita e entrega deste relatório.

O tempo previsto a ser dedicado à totalidade do estágio foi de 210 horas, tendo a responsabilidade da distribuição do mesmo ao longo das semanas sido atribuída a mim. Este facto permitiu-me realizar uma organização flexível do tempo disponível de forma a conciliar o estágio com as restantes unidades curriculares a que estava inscrito.

Tendo sido delineadas as condições para o desenvolvimento do projeto, foi desenhado um plano como guia geral para seguir, de forma a servir também como auxílio para controlar o tempo dedicado às tarefas a cumprir:

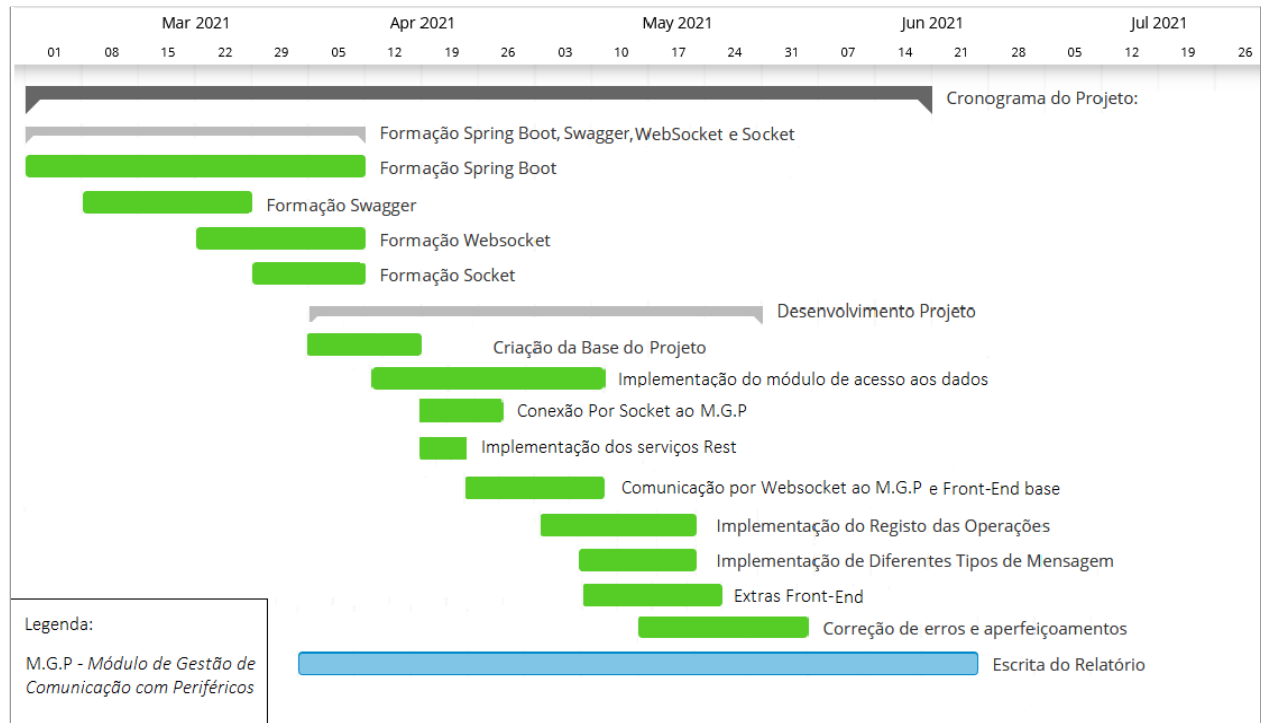


Figura 2: Cronograma do planeamento do estágio.

Resumidamente, foi decidido que iríamos iniciar o processo com um período de formação nas ferramentas principais a utilizar no projeto seguidas da implementação do projeto propriamente dito, começando na sua base e terminando com os aperfeiçoamentos e correção de erros, devendo a escrita do relatório acompanhar o dito processo.

Relativamente à orientação foram efetuadas reuniões com pelo menos um dos orientadores sempre que se justificou, tendo sido estes os principais pontos de controlo do progresso do estágio. As reuniões com o professor orientador Filipe Quintal foram realizadas por vídeo-chamada através da plataforma *Microsoft Teams*, tendo as reuniões com o Eng. Carlos Ramos sido feitas alternando semanalmente entre presencial na empresa e vídeo-chamada também por Teams.

## 2. INTRODUÇÃO

No meio da evolução constante de tudo o que é informática, um dos pilares gerais desta engenharia que não escapa a melhoramentos progressivos são as linguagens de programação. Mais especificamente a linguagem *Java* tem vindo a progredir constantemente sendo a mais recente versão lançada o *Java SE 16* no início de 2020. Ainda assim o *Java 8*, é das versões mais estáveis da linguagem lançadas, e continua a ser o *standard* de produção e desenvolvimento atuais, sendo ainda a mais utilizada [2].

Desta forma, há ainda empresas que ou começaram a utilizar esta versão recentemente ou se encontram na transição de versões mais antigas da linguagem para esta, como é o caso da Asseco, migrando da versão 6 para a 8.

Devido à ausência de suporte para a tecnologia *WebSocket* no *Java 6*, os desenvolvedores da Asseco PST implementaram originalmente uma solução para o módulo gestor de comunicação com periféricos em *Node.js* que por sua vez já possuía suporte para a mesma. No entanto, ao querer uniformizar os projetos em termos de ferramentas utilizadas, surgiu a proposta de desenvolvimento deste módulo em *Java*, sendo assim essa uma das motivações principais para o projeto de estágio em questão.

A fase de formação, que foi uma das fases mais importantes do presente estágio, não será aqui descrita em detalhe e será raramente mencionada, no entanto esta foi a que colocou mais entraves. Durante esta fase pude de forma autónoma (sempre com orientação) pesquisar sobre os temas propostos para a mesma, nomeadamente *Spring Boot*, *Swagger*, *WebSocket*, e *Socket* sendo que todas estas (à exceção de *Socket*) de uma forma ou de outra impuseram barreiras por serem tecnologias com as quais nunca tinha lidado e pouco ou nada sabia. Ainda assim, foi uma das fases fundamentais do estágio tanto para o desenvolvimento do projeto como para a minha aprendizagem e formação pessoal.

## **2.1. MÓDULO DE GESTÃO DE COMUNICAÇÃO COM PERIFÉRICOS**

Tendo introduzido a principal motivação para a migração de *Node.js* para *Java*, vale a pena mencionar que o projeto e as suas funcionalidades não serão diretamente replicadas no projeto a desenvolver. Um dos objetivos principais do desenvolvimento deste, para além da minha aprendizagem em todos os aspectos abordados daqui adiante, é também desenvolver as funcionalidades deste módulo na framework *Spring Boot* e com algumas ferramentas pré-definidas. Posto isto, é importante introduzir o que é pretendido com o módulo de gestão de comunicação com periféricos e as suas funcionalidades.

O objetivo do módulo de gestão de comunicação de periféricos é essencialmente servir como um intermediário entre dispositivos (como impressoras ou scanners) recebendo pedidos de utilizadores que se conectam a este e entregando-os ao dispositivo pretendido, sendo quase completamente agnóstico relativamente ao conteúdo destes pedidos. Alguns destes pedidos resultam também numa resposta que é entregue de volta ao utilizador original caso este esteja conectado, também de forma agnóstica. Devido à sua função, este módulo é um grande exemplo dos conceitos mais importantes do paradigma atual da engenharia de software: a abstração. Este evita a replicação exaustiva do registo e monitorização de um conjunto de dispositivos e dos pedidos processados por estes, podendo estes ser feitos numa só localização. Outro dos aspectos importantes é a diminuição do trabalho na configuração numa rede dos dispositivos mencionados, reunindo-os no mesmo ponto de acesso. Estes fatores simplificam ainda o processo de procura dos periféricos por parte dos utilizadores que os poderão encontrar, novamente, reunidos através do módulo.

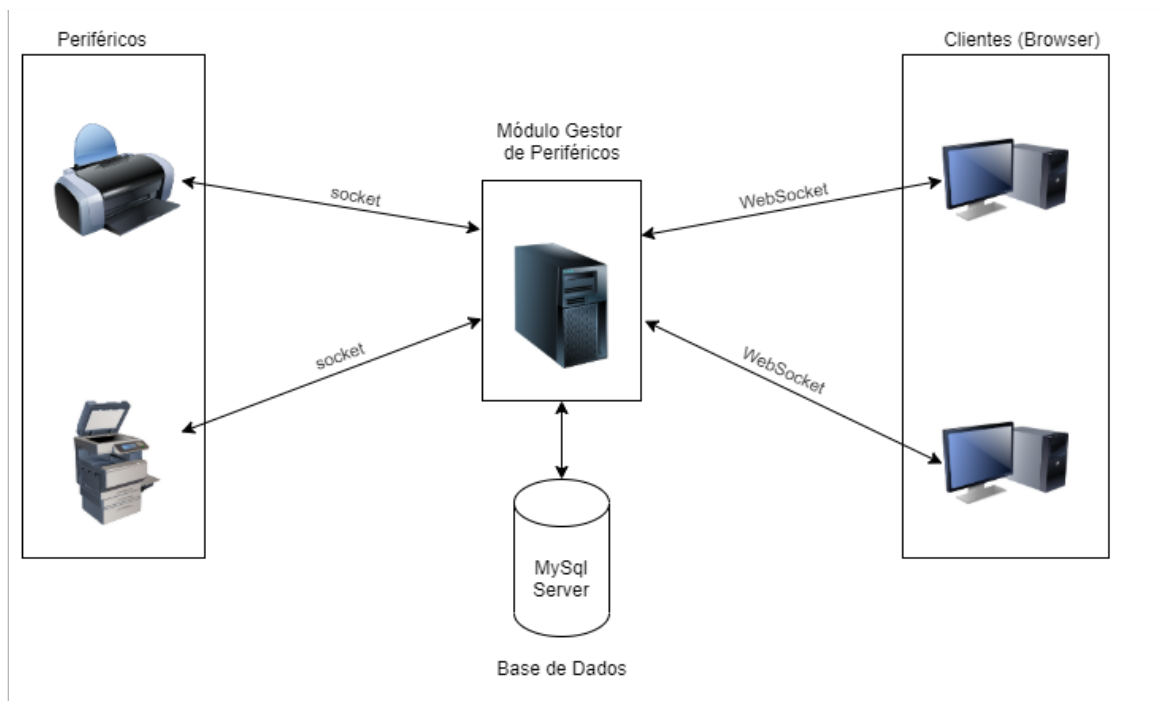
Vale a pena mencionar ainda que um dos pontos que sofreu mais na realização do projeto foi a implementação de processos *CI/CD* [3], uma vez que apenas comecei a compreender melhor tanto as implicações dos princípios *SOLID* como as formas de os aplicar no desenho de software mais tarde neste semestre quando grande parte do projeto já estava implementado.

## **3. SOLUÇÃO**

Um dos primeiros passos relativamente ao desenvolvimento do projeto foi o desenho da solução em si. Um dos pré-requisitos para o sistema a desenvolver foram os protocolos base através dos quais a aplicação iria efectuar os processos de comunicação. Para a comunicação com os utilizadores o

protocolo a usar foi o WebSocket, sendo as conexões do lado do utilizador estabelecidas através de scripting JavaScript, e do lado do servidor através do suporte WebSocket do Spring Boot. Já para a comunicação com os periféricos, o meio indicado foram os Java Sockets, sendo que o servidor fica à escuta numa porta para quando os periféricos se tentarem conectar através desta poder ser estabelecida uma ligação.

Para ilustrar isto foi desenhado um diagrama desta estrutura:



*Figura 3: Diagrama da estrutura do projeto.*

De modo a poder compreender melhor as ações que compõe a comunicação entre os atores do ambiente da aplicação a serem desenvolvidos na solução, foi ainda desenhado um diagrama de casos de uso (*use cases*). Os diagramas de casos de uso são utilizados em análise de sistemas para reunir os requisitos de um sistema tendo em conta as influências internas e externas. Desta forma, este diagrama será posteriormente utilizado para chegar aos requisitos propriamente ditos que irão estabelecer todos os aspectos necessários à realização das ações ilustradas pelo diagrama.

Os use cases representados na figura seguinte surgiram através da análise das comunicações entre todos os intervenientes. Para que seja possível a comunicação ocorrer de forma efetiva tem de haver algum tipo de mecanismo de conexão, para que as mensagens a ser entregues aos seus destinatários possam ser processadas de forma adequada. Assim, para além dos casos de conexão dos atores há também os casos de envio de mensagem entre estes atores. Estas mensagens deverão ser entregues aos seus destinatários independentemente de serem utilizadores ou periféricos mas no caso destes já



não se encontrarem conectados deverá ser executada a ação apropriada. No caso de um dispositivo já não se encontrar conectado o utilizador deverá ser informado que o dispositivo escolhido já não está disponível, e se o utilizador já não estiver conectado quando uma resposta é gerada para si, o pedido deverá ser registado na base de dados como não entregue.

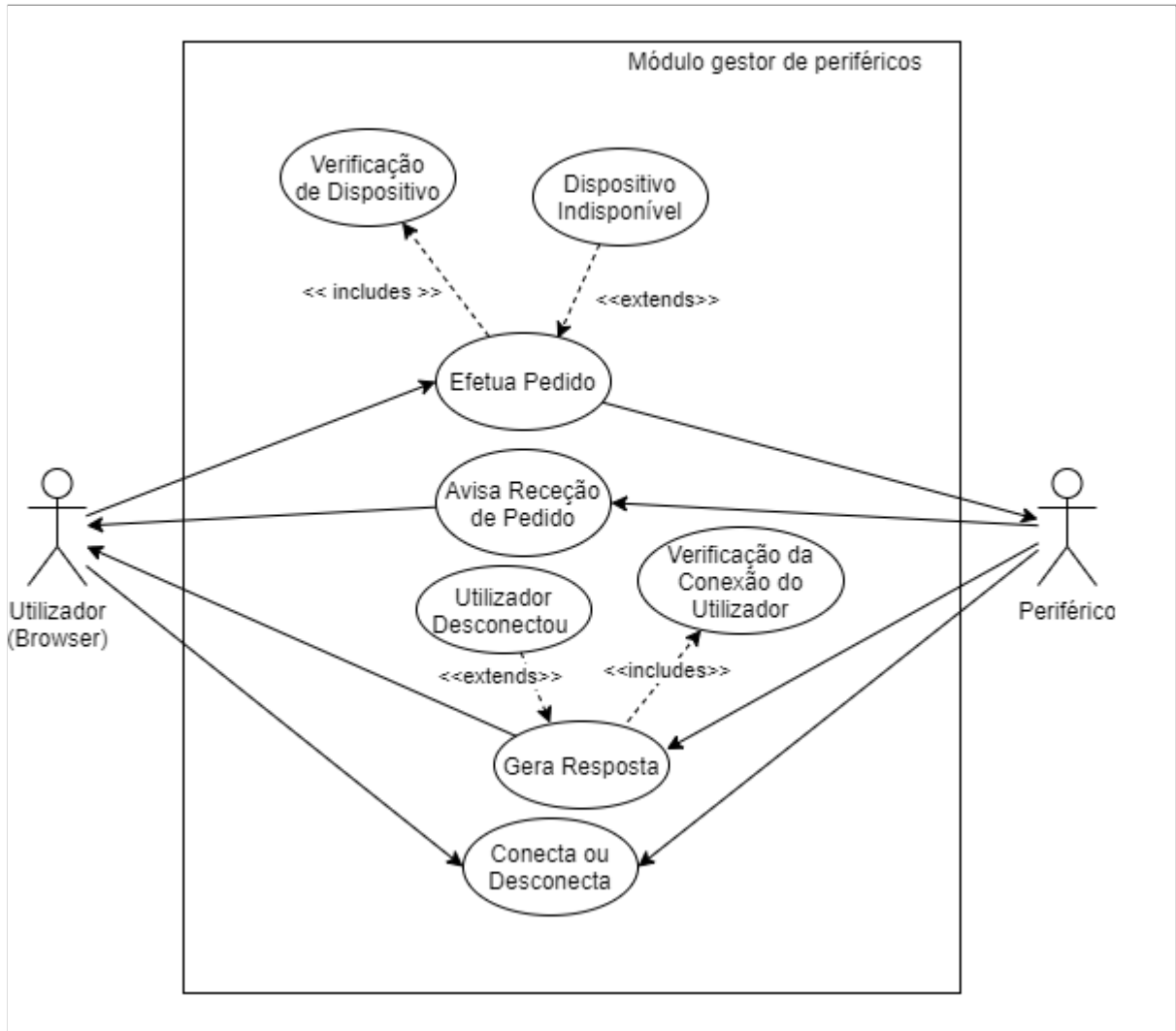


Figura 4: Diagrama de casos de uso desenvolvido.

### 3.1. DESCRIÇÃO

O módulo gestor de comunicação de dispositivos tem como objetivo, como o nome indica, gerir a comunicação de utilizadores com os periféricos conectados ao módulo. A ideia principal da solução a

desenvolver é que os utilizadores se conectem através de um browser, estabelecendo uma ligação contínua através do protocolo WebSocket, e através do cliente *front-end* fazerem pedidos aos dispositivos conectados ao mesmo. Estes dispositivos, por sua vez, comunicam com o módulo através de *sockets* Java sendo através destes e do protocolo de comunicação a desenvolver que ambos comunicam.

Outro dos pontos principais a referir é que a solução será implementada utilizando *Spring Boot*, que é uma framework open source baseada em Java utilizada para criar micro-serviços, e que integra suporte para quase todas as tecnologias a usar para as funcionalidades da aplicação.

Será ainda implementada uma API (*application programming interface*) REST (*Representational State Transfer*) utilizando *Swagger* que terá como objetivo a consulta remota das listas dos utilizadores e dispositivos atualmente conectados.

Todo o armazenamento de dados relativos aos dispositivos e aos pedidos efetuados aos mesmos será feito através de uma base de dados MySQL local, sendo que em memória na aplicação apenas será mantido o registo dos utilizadores que estão conectados.

Finalmente, a aplicação terá ainda a capacidade de efectuar registos de todas as operações relevantes através de um sistema de *logs*, de modo a tornar possível monitorizar o fluxo de execução da mesma.

### 3.2. FERRAMENTAS UTILIZADAS

Para implementar a solução proposta é necessário decidir quais as ferramentas a utilizar para as funcionalidades que esta dispõe. Algumas destas foram deixadas ao meu critério, no entanto outras foram já definidas nos pré-requisitos por parte da Asseco.

As ferramentas já definidas a ser utilizadas são as seguintes:

1. Spring Boot (framework de desenvolvimento em Java);
2. Swagger (ferramentas de desenvolvimento e APIs REST);
3. WebSocket e Java Socket (ferramentas de comunicação entre aplicações);
4. Base de dados SQL;

Tendo em conta a descrição e detalização da solução pretendida, para o seu desenvolvimento irei necessitar de escolher ferramentas para os seguintes pontos:

1. Ambiente de desenvolvimento (IDE e controlo de versões);
2. Host de base de dados a usar;
3. Camada de persistência da base de dados (na aplicação);
4. Protocolo a usar sobre as conexões WebSocket
5. Protocolo a usar sobre as conexões Socket;
6. Ferramenta de registo de operações (logging);

Posto isto serão descritas todas estas ferramentas, tanto as pré-definidas, como as escolhidas e as alternativas para algumas destas.

### 3.2.1. Ambiente de desenvolvimento

Para o ambiente de desenvolvimento há duas escolhas importantes a tomar. A primeira é o IDE a utilizar, uma vez que as funcionalidades disponibilizadas por estas ferramentas variam e podem comprometer todo o processo de desenvolvimento uma vez tomada a decisão. A segunda decisão é o sistema de controlo de versões. Esta é também importante no sentido em que deverá ser uma que o IDE escolhido anteriormente suporte e seja fácil de usar.

Posto isto, as ferramentas escolhidas foram as seguintes:

#### **Escolha de IDE:**

Como já referido anteriormente, foi proposto o desenvolvimento do projeto utilizando a linguagem de programação Java, e após um pequeno período de deliberação decidi que iria fazer toda a implementação do projeto com o *Eclipse IDE*. As duas grandes vantagens disto são a existência do plugin *Spring Initializr* que permite a fácil e rápida criação de projetos Spring Boot, e também o facto de ser o que os desenvolvedores da Asseco utilizam, tornando assim mais fácil a resolução de qualquer problema ou dificuldade que encontre.

Se não tivesse este último fator mencionado em conta, o IDE que utilizaria seria o *IntelliJ IDEA*, uma vez que também possui suporte para Spring Boot Initializr tem um design de interface mais agradável e intuitivo. A única desvantagem relativamente é as diferenças em como os mecanismos de VC (*version control*) são geridos, que não achei tão intuitivos na minha experiência pessoal.

#### **Controlo de versões:**

A necessidade da escolha de um sistema de controlo de versões baseia-se num único aspecto: controlar o desenvolvimento da aplicação de forma a que todas as alterações possam ser controladas, sendo aplicadas ou revertidas dependendo do seu efeito. A certo ponto poderá ser necessário reverter a aplicação para um estado anterior devido a uma alteração cujo arranjo é mais dispendioso do que refazer a mesma. Para além disto é possível ainda testar estas alterações antes de as aplicar à versão oficial, que é uma função extremamente útil.

Posto isto, o sistema imediatamente escolhido foi o *Git*. A motivação principal desta escolha foi o facto de o IDE escolhido, o Eclipse, possuir uma boa suporte para este.

Foi criado um repositório onde foram constantemente guardadas as alterações ao longo de todo o projeto, tanto localmente como através da plataforma *GitHub*.

### 3.2.2. Spring Boot Framework

Uma das ferramentas centrais e mais importantes do projeto foi o *Spring Boot*. Esta ferramenta é uma framework open-source baseada em Java que permite a criação de aplicações Spring. Para além disto, o Spring Boot tem ainda a grande vantagem de facilitar imenso a integração de bibliotecas de terceiros. Quase todas as tecnologias integradas neste projeto beneficiaram deste aspeto, reduzindo a quantidade de tempo gasta em configurações e escrita de ficheiros XML. Para alguns dos módulos desenvolvidos como componentes da solução apenas foi necessária uma classe de configuração ou um ficheiro XML curto, e outros não precisaram de nenhum destes. Um bom exemplo disto é um dos "starters" (dependências próprias da Spring Framework que desempenham estas configurações a que me estou a referir), o Spring Web. Ao adicionar este como dependência do projeto, o Spring Boot inicializa uma instância embebida do Apache Tomcat (servidor web) sem qualquer outro tipo de indicação necessária [4].

Outra funcionalidade importante relativamente ao seu modo de funcionamento - que complementa o exemplo do servidor Tomcat referido anteriormente - é a configuração automática da “canalização” dos componentes. Tal como foi referido, faz parte dos requisitos o desenvolvimento de uma API REST, e para isto é necessário definir os controladores REST. Devido à gestão automática destes controladores por parte do Spring Boot, apenas é necessário adicionar uma anotação “@RestController” à classe que queremos que aja como controlador e esta é automaticamente integrada pelo servidor.

Finalmente, há ainda uma outra grande vantagem desta framework, que é o facto de usar como uma das suas funcionalidades fundamentais a Injecção de Dependências. A utilização deste princípio permite atingir um baixo acoplamento dos componentes delegando a responsabilidade de os gerir para o *container* Spring, que por sua vez injeta as dependências onde estas são necessárias [5]. Relativamente a este último conceito irei entrar mais em detalhe na secção em que descrevo a implementação.

### 3.2.3. MySQL Server

Uma vez que foi decidido utilizar uma base de dados para o registo dos pedidos e dispositivos, é necessário perceber que base de dados e que tipo de host utilizar.

Na apresentação inicial do projeto por parte da Asseco, foi-me referido que se fosse necessário utilizar uma base de dados seria para utilizar uma que fosse baseada em SQL. Desta forma, uma vez que já tinha uma mínima experiência com MySQL Server pareceu-me adequado utilizar esta tecnologia, uma vez que é bastante simples de monitorizar através do MySQL workbench.

Uma excelente alternativa ao MySQL Server mas não com SQL, seria o MongoDB. As razões principais para isto é a simplicidade do esquema de base de dados final que possui apenas duas tabelas. A outra razão é existir o MongoDB Atlas, que é uma plataforma de hosting de bases de dados em cloud gratuita (com limitações de armazenamento entre outras, claro) que permitiria iniciar a aplicação em qualquer host sem qualquer reconfiguração da base de dados na aplicação (por mais simples que seja) necessitando no entanto uma conexão à internet.

### 3.2.4. Spring Data JPA e Hibernate

Uma ferramenta importante a escolher que condiciona o acesso e tratamento dos dados na aplicação é aquela que será utilizada na camada de persistência de dados da aplicação. Esta escolha condicionará não só a forma de conexão à própria base de dados como a representação, e acesso aos dados da mesma na aplicação.

Como referido anteriormente na descrição do Spring Boot, este possui dependências próprias que automaticamente configuram parte de componentes e de bibliotecas de terceiros. Um exemplo destes é o *starter* Spring Data JPA, que é uma camada de abstração sobre a JPA (*Java Persistence API*) através da qual é possível conectar a uma base de dados utilizando o driver adequado, e realizar operações sobre a mesma. Uma das implementações da JPA mais utilizadas e relevantes é o Hibernate, integrada no Spring Data JPA, e que será utilizada por mim para realizar todas as operações necessárias sobre a base de dados criada.

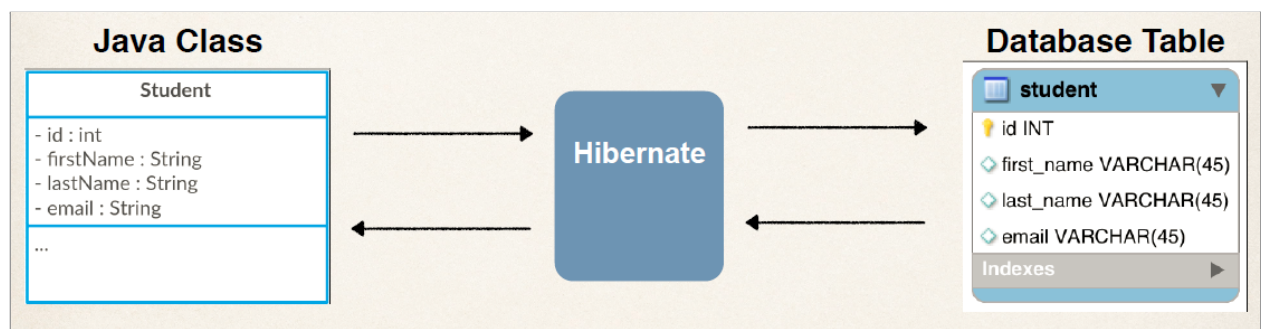


Figura 5: Hibernate como mapeador ORM.

### 3.2.5. Swagger / OpenAPI

Uma funcionalidade importante a implementar na aplicação é a API REST. Estes componentes integrados em aplicações, websites, entre outros, servem não só para obter informações relativamente ao estado de execução ou dados armazenados nos mesmos, como também permite que através destas informações estes possam ser facilmente monitorizados. Esta é um dos pontos de aprendizagem do estágio, sendo que é uma das ferramentas habituais a ser integradas nos projetos da Asseco.

Relativamente à implementação desta API REST no projeto, a ferramenta a utilizar por indicação da Asseco é o *Swagger*. Sendo um conjunto de ferramentas desenhadas para auxiliar o desenho, implementação, documentação e monitorização de APIs da especificação OpenAPI, o Swagger é uma das ferramentas mais conhecidas e utilizadas do tipo.

Duas das características mais pertinentes deste *tool/set* são a existência de plugins que permitem a geração de código de interfaces da API prontos a ser implementados a partir de um ficheiro “.yaml” ou “.json” que respeite a especificação, e a integração simples da ferramenta Swagger UI que permite visualizar uma documentação das APIs presentes no projeto gerada automaticamente.

Para além das ferramentas mencionadas nesta secção, para o desenho da API e exportação do ficheiro yaml utilizei a aplicação StopLight, e a aplicação Postman para testar as implementações.

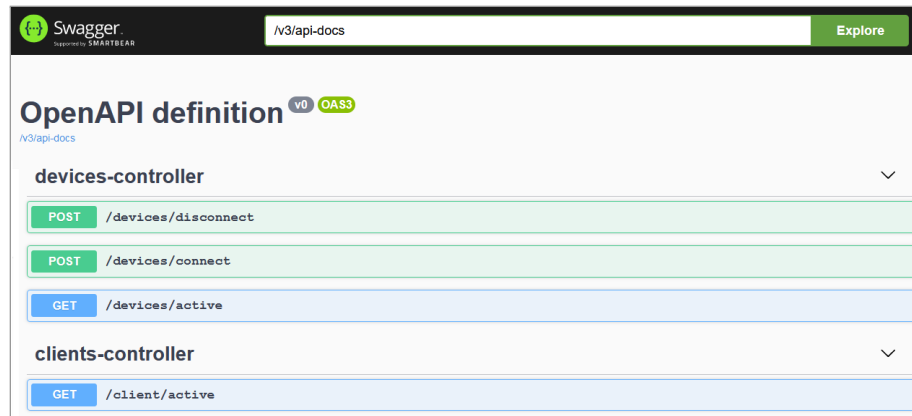


Figura 6: SwaggerUI da API da aplicação.

### 3.2.6. WebSocket / StompJS

O WebSocket é um protocolo que permite uma comunicação bidireccional entre um servidor e um cliente. O início de uma conexão por WebSocket começa com um pedido normal HTTP por parte de um cliente sendo este respondido pelo servidor. Durante esta troca, o cliente pede ao servidor para este abrir uma ligação WebSocket, e caso este aceite os dois passam a utilizar a ligação TCP/IP estabelecida para o pedido HTTP como uma conexão WebSocket. Após estabelecida uma conexão, passa a poder haver envio de dados através de um protocolo de troca de mensagens.

Juntamente com os WebSockets será utilizado o STOMP (*Simple Text Orientated Messaging Protocol*) para realizar todas as trocas de informação entre o browser e o servidor necessárias ao funcionamento da aplicação.

### 3.2.7. Java Socket

Para a comunicação por Socket é necessário escolher que tecnologia utilizar para estabelecer a conexão e que protocolo de comunicação em si utilizar nas mensagens a transportar pelo canal.

Um socket é um “end-point” de uma comunicação bidireccional entre duas aplicações numa rede. Será através das classes Java *Socket* e *ServerSocket* que será implementada a ligação entre a aplicação e os dispositivos que se queiram conectar à mesma, através de um protocolo de comunicação simples.

### 3.2.8. Log4j2

Para os mecanismos de registo de operações e do fluxo de execução da aplicação, é benéfico tanto em termos de tempo gasto no desenvolvimento como da eficiência da aplicação utilizar uma ferramenta já desenvolvida para as funcionalidades de logging. Desta forma, após uma análise breve das ferramentas disponíveis e comparar reviews de outros utilizadores, decidi que a ferramenta a utilizar seria o Log4j2.

O Log4j2 é uma ferramenta utilitária de *logging* baseada em Java que, embora não seja propriamente recente é ainda bastante utilizada devido à sua simplicidade tanto em termos de

configuração como de implementação, tendo sido essas as razões pelas quais foi a ferramenta escolhida.

### **3.3 RESUMO SOLUÇÃO/FERRAMENTAS**

Tendo em conta o trabalho revisto, será então resolvida uma solução em Java no Eclipse IDE, utilizando o controlo de versões Git e sobre a framework Spring Boot.

O seu objetivo principal será a comunicação entre periféricos e utilizadores, sendo que entre o servidor e a aplicação será utilizado WebSocket com o protocolo de mensagens STOMP, e entre os periféricos e a aplicação será utilizado Java Socket com um protocolo de mensagens criado por mim.

Os pedidos dos utilizadores e os dispositivos e o seu estado serão armazenados numa base de dados MySQL local utilizando MySQL Server, sendo que a ferramenta de persistência de dados a utilizar na aplicação para comunicar com a base de dados é o Hibernate, integrado no Spring Data JPA.

Os dados dos dispositivos e utilizadores conectados poderão ser acedidos através de uma API REST que será desenvolvida através da ferramenta Swagger e da aplicação StopLight, sendo os testes feitos através da aplicação Postman.

Por fim, todas as operações relevantes ao fluxo da aplicação serão registadas através de um sistema de registo de operações através da ferramenta Log4j2.

## **4. IMPLEMENTAÇÃO**

Para a implementação da solução para o projeto proposto foram seguidos os diagramas anteriormente apresentados e as ferramentas descritas após os mesmos. Esta fase na sua essência foi a fase em que pude pôr em prática todos os conhecimentos absorvidos durante a fase de formação atendendo aos requisitos reunidos em antemão e que serão apresentados nesta secção do relatório.

### **4.1. REQUISITOS**

Os requisitos são uma parte fundamental do desenvolvimento de uma aplicação, uma vez que não só estabelecem o acordo entre a equipa desenvolvedora relativamente ao que esta é suposto fazer como também permitirá à equipa definir de uma forma clara o planeamento do processo de desenvolvimento.

Todos os requisitos reunidos foram fruto das conversas, explicações e indicações obtidas por parte do Eng. Carlos Ramos nas reuniões em que discutimos os objetivos da aplicação, sendo que alguns deles foram diretamente explicitados, e outros foram redigidos por mim como consequência da análise dos objetivos gerais, preenchendo as lacunas ao observar o que era necessário para garantir as funcionalidades da aplicação. Para além das discussões referidas anteriormente, o outro factor importante para obter os requisitos não referidos diretamente foram os use cases desenvolvidos. Realizar os diagramas permitiu-me obter um *insight* e uma visualização mais clara do problema, sendo que com essa fase do projeto consegui compreender melhor a importância deste tipo de planeamento estudado em várias unidades curriculares ao longo da licenciatura.

Posto isto, apresento então os requisitos obtidos:

#### **4.1.1. Requisitos ambientais**

Os requisitos ambientais são os requisitos relacionados com o ambiente envolvente da aplicação e com as suas interações com o que a rodeia.

1. O sistema deverá possuir protocolos de comunicação WebSocket para que utilizadores se possam conectar e interagir com este.
2. O sistema deverá possuir protocolos de comunicação Socket para que os periféricos se possam conectar e interagir com este.
3. O sistema deverá ter a capacidade de se conectar a uma base de dados local.
4. O sistema deverá possuir uma API REST para monitorização.

#### *4.1.2 Requisitos funcionais*

Estes requisitos são os requisitos reunidos e acordados relativamente às funcionalidades que o sistema deverá ter.

1. O sistema deverá ter a capacidade de aceitar e registar conexões de utilizadores.
2. O sistema deverá ter a capacidade de aceitar e registar conexões de dispositivos.
3. O sistema deverá prevenir o envio e registo de pedidos a dispositivos não conectados.
4. O sistema deverá incorporar um sistema de logs que descreva e registe de forma clara e explícita o fluxo de execução deste.
5. O sistema deverá ter um protótipo funcional de front-end desenvolvido de modo a permitir simular pedidos de utilizadores.
6. O sistema deverá ter um simulador de dispositivos de modo a ser possível testar a comunicação com estes.

#### *4.1.3. Requisitos de dados*

Estes requisitos são os relacionados com os dados que a aplicação irá tratar, manter e transmitir.

1. O sistema deverá manter um registo dos dispositivos conectados.
2. O sistema deverá manter um registo dos utilizadores conectados.
3. O sistema deverá enviar todos os dados relacionados com pedidos de forma agnóstica.

### **4.2. ESTRATÉGIA**

O pretendido no início da implementação da solução e primeiro grande objetivo foi a criação da base deste, de forma a que se conseguisse um protótipo funcional do conceito base da aplicação: o envio de um pedido por parte de um utilizador através da aplicação obtendo este de seguida uma resposta gerada pelo dispositivo para o qual dirigiu o pedido. Esta funcionalidade é transversal a vários componentes do sistema projetado, e portanto algumas fases do desenvolvimento destes componentes - separados em módulos distintos - foram feitas simultaneamente.

Para além destes módulos foi ainda desenvolvida uma simples aplicação Java que se conecta ao sistema através de um socket para simular um dispositivo, podendo vários dispositivos de tipos diferentes ser simulados correndo várias instâncias da aplicação em simultâneo.

Tendo em conta esta necessidade de trabalhar em vários módulos simultaneamente, para não implementar porções de código demasiado complexas de cada vez tanto em relativamente aos componentes em si como ao acoplamento entre estes, tomei a abordagem de desenvolvimento iterativo. Esta era a abordagem já proposta pelo planeamento do estágio descrito pelo cronograma apresentado no início deste relatório, começando pela base do projeto, e adicionando funcionalidades e pormenores incrementalmente. Assim, testando cada iteração do projeto à medida que estas eram desenvolvidas, os



módulos foram adicionados, melhorados e aprofundados de forma a que cada iteração seguinte tivesse uma base estável e funcional.

Posto isto, apresento assim os módulos desenvolvidos:

#### 4.2.1. Divisão por módulos

De forma a alcançar um nível de abstração adequado e separar as responsabilidades e funcionalidades, decidiu-se que seriam desenvolvidos os seguintes módulos:

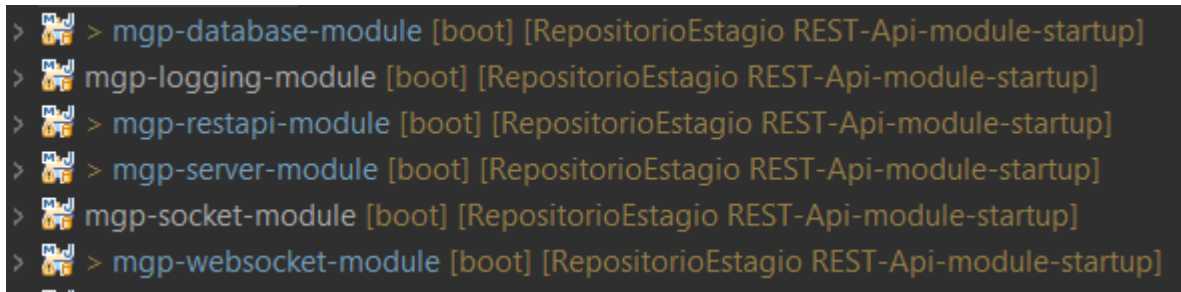


Figura 7: Divisão em módulos do projeto.

Passo assim a explicar a função destes:

1. Módulo Acesso aos Dados (mgp-database-module)

O propósito principal deste módulo é, como o nome indica, gerir o acesso aos dados. Através do driver para MySQL e do Spring Data JPA este utiliza o Hibernate para efectuar todas as operações que envolvem persistência de dados ou obtenção dos mesmos da base de dados. Será este que conterà os serviços que permitirão os restantes módulos interagirem com a base de dados.

2. Módulo de Registos (mgp-logging-module)

O único objetivo deste módulo é tratar do registo das operações da aplicação em logs. Será este módulo que possuirá os serviços que permitirão os outros módulos registar o fluxo de execução da aplicação.

3. Módulo API REST (mgp-restapi-module)

Como mencionado anteriormente, o módulo de gestão da comunicação com periféricos terá uma API REST. O objetivo deste módulo é conter essa API juntamente com o ficheiro yaml com a sua descrição, as interfaces geradas automaticamente através do Swagger Codegen e também as suas implementações.

4. Módulo Servidor (mgp-server-module)

O módulo servidor será um simples módulo que terá o propósito de inicializar o contexto Spring Boot, unificar todos os módulos, gerir a aplicação de consola do programa e também conter um protótipo de front-end simples cujo objetivo é simular utilizadores a usufruir das funcionalidades da aplicação.

5. Módulo Socket (*mcp-socket-module*)

Será aqui que estará tudo o que é relacionado com a comunicação por socket com os dispositivos. Através de um protocolo presente neste módulo será possível os dispositivos conectarem, receberem pedidos e enviar respostas de volta para a aplicação, sejam estas notificações ou respostas a pedidos concretos.

6. Módulo WebSocket (*mcp-websocket-module*)

Por fim temos o módulo WebSocket. O objetivo principal deste módulo é controlar e permitir a comunicação da aplicação com os utilizadores. Utilizando a tecnologia WebSocket o browser poderá manter-se conectado à aplicação de forma a receber informação do servidor relativamente aos dispositivos que estão conectados, o estado dos seus pedidos e as suas respostas, e também ajudar a que o dispositivo registe os utilizadores conectados.

#### 4.4. PRODUTO FINAL

Nesta secção irei apresentar a implementação dos módulos referidos anteriormente. A implementação destes foi feita tendo em conta os conceitos discutidos anteriormente de forma a produzir as funcionalidades e cumprir os requisitos definidos neste relatório, que refletem o objetivo da aplicação tanto em termos de produto final como de apropriação e adaptação futuras.

Como também já discutido nas secções precedentes, a implementação foi efectuada através de módulos que isolam os componentes de forma a criar abstrações entre os mesmos. Isto foi feito também através de interfaces que visam estabelecer os contratos entre os módulos e aumentar a abstração pretendida[6].

##### 4.4.1. Conceitos importantes

Antes de começar a entrar em detalhes relativamente à implementação em si do projeto, há alguns conceitos que é importante serem explicados. Estes conceitos são relacionados com o Spring Boot em si e com a forma como este funciona, e sem um breve esclarecimento poderão ser facilmente mal compreendidos.

##### 4.4.1.1. SpringApplication

Relativamente aos aspectos de funcionamento e montagem da framework Spring Boot, SpringApplication é a classe central e mais importante. É através do método "run" desta que a aplicação junta todos os componentes implementados/importados e os junta no mesmo ambiente de execução.

Após a inicialização desta classe, para além de esta inicializar os componentes, trata ainda de realizar todas as configurações automáticas ou implementadas de forma a que estes funcionem corretamente, e ainda para que estes possam sem problemas ser utilizados através dos mecanismos de injeção de dependências.

##### 4.4.1.2. Spring Beans e injeção de dependências

Os *Beans* da Spring são objetos que constituem o "backbone" das aplicações Spring. Estes são os objetos que, sendo inicializados como singletons podem ser injetados nos lugares apropriados.

Exemplos destes "beans" são, por exemplo, os controladores REST definidos na API rest, serviços como desenvolvido para as classes de acesso aos dados, etc:

(Imagem dos tipos de anotações)

Assim, tendo declarado os beans de forma apropriada, basta declarar dentro de um componente (que pode ser também outro bean) o tipo estático deste com uma anotação "@Autowired" que o Spring Boot injetará nesse mesmo sítio a tal instância criada.

Um dos pontos principais e mais importantes deste aspecto é o facto de todas as instâncias destes objectos são criadas na inicialização da aplicação por parte do Spring Boot, sendo que qualquer tentativa de criação de uma instância dessas classes resultará em null pointers nas referências estáticas às quais se esperaria um bean injetado.

No entanto, apesar da limitação referida anteriormente, há uma forma de aceder às instâncias criadas pelo Spring Boot em classes e situações em que não há outra solução senão criar uma instância em tempo de execução e em que se necessita de utilizar um bean nessa instância. Será esta abordagem descrita no ponto seguinte.

#### *4.4.1.3. Acesso ao Spring Context*

O Spring Context é como o nome indica, o contexto Spring da aplicação. É neste "local" que todas as configurações, componentes e instâncias inicializadas pelo Spring Boot se encontram. Um destes componentes é o Spring Container.

O Spring Container IoC (Inversion of Control - inversão de controlo) é o local onde todos os beans (instâncias singletons de controladores) mencionados anteriormente se encontra, e o Spring recomenda a que estes não sejam acedidos de outra forma a não ser pela injeção de dependências gerida por este. No entanto, o Spring disponibilizou uma forma de aceder a estas instâncias caso necessário, e foi através deste modo que no projeto acedi aos beans Spring nas classes de leitura e escrita do socket, que precisam de ser instanciadas uma quantidade de vezes imprevisível.

#### *4.4.2. Servidor*

O módulo servidor foi o módulo implementado da forma mais simples. Este reúne os módulos todos através do POM do projeto (Maven). Através da capacidade de busca e integração de componentes do Spring Boot, bem como de autoconfiguração dos mesmos, a SpringApplication utiliza estes para criar os seus beans de forma a que possam ser injectados nos locais apropriados.

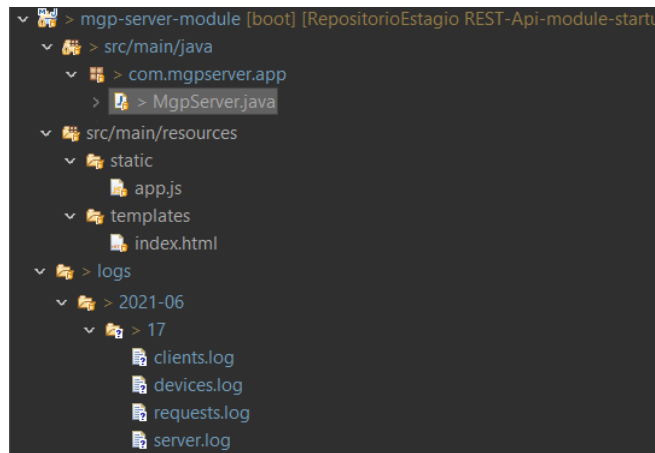


Figura 8: Estrutura de ficheiros do módulo servidor.

A classe principal é a MgpServer, que é a classe “main”. É através desta que ocorre o processo Spring Boot descrito atrás, com o auxílio de anotações que indicam algumas configurações e também uma que auxilia a aplicação a encontrar os componentes das dependências. Adicionalmente é também nesta classe que podemos encontrar a inicialização do ServerSocket, classe que age como listener para os sockets e será explicada mais à frente.

```
@SpringBootApplication
@EnableJpaRepositories("com.mgpdatabase.repository")
@EntityScan("com.mgpdatabase.entities")
@EnableScheduling
@ComponentScan(basePackages= {
    "com.mgplogging",
    "com.mgpdatabase.service",
    "com.mgpwebsocket.api",
    "com.mgpsocket.util",
    "com.mgpserver.config",
    "com.mgpserver.controller",
    "com.mgpserver.app.frontend",
})
public class MgpServer {

    private static boolean finish = false;
    private static final int SOCKET_PORT = 5000;

    public static void main(String[] args) {
        SpringApplication.run(MgpServer.class, args);
    }
}
```

Figura 9: Anotações da classe main MgpServer.

Para além disto, a outra funcionalidade desta classe é providenciar a interface de consola, de forma a que possam ser inseridos comandos manualmente na aplicação.

O único comando inserido foi o de paragem da aplicação, sendo que não foi necessário mais nenhum, e quando este é usado a aplicação faz uma paragem em segurança desligando o SocketServer e fazendo uma limpeza básica em alguns dados da base de dados.

```

while(!finish) {
    try {
        userInput = consoleReader.readLine();
        if(userInput.equals("server stop")) {

            // Terminar o listener do socket
            socketListener.stopListening();

            // Limpeza do estado dos devices da base de dados
            SpringContext.getBean(DeviceService.class).cleanDeviceRepository();
            finish = true;
        }
    } catch (IOException e) {
        e.printStackTrace();
        System.out.println("An error occured.");
    }
}

```

Figura 10: Controlo de input da consola com o comando de paragem do programa implementado.

Para além destas funcionalidades, é também neste módulo que são gerados os logs como pode ser observado na figura da sua estrutura.

Por fim, por convenção é aqui que estão guardados os ficheiros referentes à parte front-end da aplicação (um ficheiro html para a estrutura e um JavaScript para os scripts que apoiam as funcionalidades), no entanto estes serão discutidos mais adiante na secção da comunicação.

#### 4.4.3. Comunicação WebSocket

Esta secção serve para discutir a forma com que a comunicação por WebSocket é efectuada na aplicação. Como referido anteriormente esta é feita através da dependência STOMP que permite expor os endpoints que representam os canais de comunicação tanto do lado do servidor como do lado do browser, bem como decidir a estrutura do conteúdo das mensagens em si.

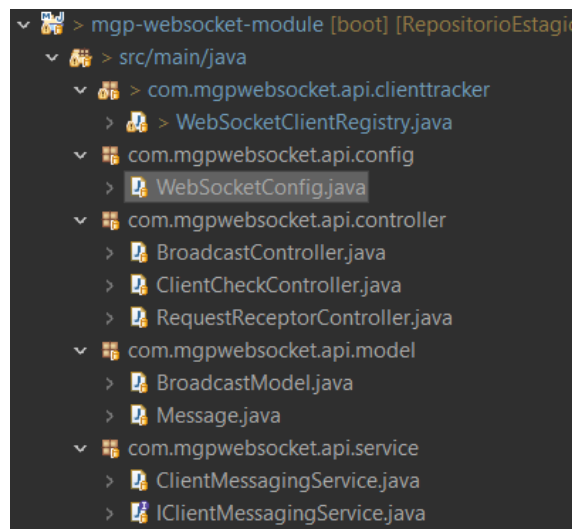


Figura 11: Estrutura de ficheiros do módulo WebSocket.

```
@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {

    @Override
    public void registerStompEndpoints(StompEndpointRegistry registry) {
        registry.addEndpoint("/session").withSockJS();
    }

    @Override
    public void configureMessageBroker(MessageBrokerRegistry registry) {
        registry.setApplicationDestinationPrefixes("/app").enableSimpleBroker("/topic");
    }
}
```

Figura 12: Classe de configuração da comunicação WebSocket.

Através da classe de configuração, é definido o canal principal do WebSocket como um endpoint ("/session"), através do qual os clientes poderão estabelecer uma conexão. Para além disto é também definido um prefixo para os canais a partir dos quais os clientes podem enviar mensagens ao servidor ("/app") e outro ("/topic") a que os clientes podem escutar para eventuais mensagens do servidor.

#### 4.4.3.1. Servidor

Do lado do servidor, para além das configurações e funcionalidades primárias de escutar mensagens dos clientes há ainda outras três responsabilidades atribuídas:

1. Receber pedidos através dos canais específicos de cada cliente, tratar e registar estes pedidos, e oferecer ferramentas para os outros módulos poderem notificar estes utilizadores tanto o progresso destes pedidos como eventuais respostas;
2. Fazer broadcast dos dispositivos conectados por socket para todos os clientes conectados por browser regularmente;
3. Manter o registo de todos os clientes conectados.

O primeiro ponto é efectuado através de canais específicos a cada cliente identificados por uma string única a cada um incluídos na estrutura dos pedidos, podendo os módulos enviar mensagem através do serviço desenvolvido no módulo WebSocket (ClientMessagingService).

O segundo é através de um canal comum a todos os clientes, enviando o servidor uma lista dos dispositivos atualmente conectados regularmente.

Por fim, o terceiro é realizado através de outro canal comum a todos os clientes, que estes utilizam para regularmente comunicar a sua string de identificação para indicar que ainda estão conectados. O servidor verifica regularmente este registo e se um cliente não comunicar há um determinado tempo limite, é removido do registo.

```

public class BroadcastModel {

    private int id;
    private List<Device> deviceList;

    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public List<Device> getDeviceList() {
        return deviceList;
    }
    public void setDeviceList(List<Device> deviceList) {
        this.deviceList = deviceList;
    }
}

public class Message {

    private String message;
    private String origin;
    private String deviceName;
    private String eventType;

    public String getEventType() {}
    public void setEventType(String eventType) {}
    public String getMessage() {}
    public void setMessage(String message) {}
    public String getOrigin() {}
    public void setOrigin(String origin) {}
    public String getDeviceName() {}
    public void setDeviceName(String deviceName) {}
    public String toString() {}
    public ClientRequest convertToClientRequest(int id) {}
}

```

Figura 13: Modelos para as mensagens de broadcast (esquerda) e de pedidos de utilizador (direita).

#### 4.4.3.2. Lado do Browser

Do lado do browser, ocorrem operações inversas às do servidor. Este está responsável por comunicar ao servidor regularmente para indicar que está conectado, e para além disto subscreve ainda ao canal de broadcast (para ser informado dos dispositivos conectados) disponibilizando esta informação ao utilizador.

Todas as informações que o browser recebe atualizam a página automaticamente através do JavaScript implementado, que utilizando JQuery dinamicamente altera a página para produzir o efeito pretendido. Isto deve-se principalmente ao facto de a receção das mensagens por WebSocket serem tratadas como eventos, podendo logo ser apresentadas.

Foram implementados quatro tipos de mensagens, associadas a quatro tipos de pedidos: print (impressão), scan (digitalização), fingerprint scan (digitalização de impressão digital) e check scan (digitalização de cheques).

```

else if(!currentEvent.localeCompare("print")){
    if($("#copyammount").val() === "" || $("#sheet
        error = true;
    }
    else{
        msgstring = JSON.stringify({
            copyAmmount: $("#copyammount").val(),
            sheetSize: $("#sheetsize").val(),
            margin: $("#margin").val(),
            sheetPages: $("#sheetpages").val(),
            data: $("#inputmessage").val()
        });
    }
}

else if(!currentEvent.localeCompare("scan")){
    if($("#quality").val() === "" || $("#sheets
        error = true;
    }
    else{
        msgstring = JSON.stringify({
            quality: $("#quality").val(),
            sheetSize: $("#sheetsize").val(),
            margin: $("#margin").val(),
            type: $("#type").val(),
        });
    }
}

```

Figura 14: Estrutura dos parâmetros dos pedidos impressão (esquerda) e digitalização (direita).

```

else if(!currentEvent.localeCompare("fingerprintscan")){
    if($("#fingerid").val() === "" || $("#capturemode").val() === ""){
        error = true;
    }
    else{
        msgstring = JSON.stringify({
            fingerId: $("#fingerid").val(),
            captureMode: $("#capturemode").val(),
            margin: $("#margin").val(),
        });
    }
}

else if(!currentEvent.localeCompare("checkscan")){
    if($("#imagecolor").val() === "" || $("#imagesize").val() === ""){
        error = true;
    }
    else{
        msgstring = JSON.stringify({
            imageColor: $("#imagecolor").val(),
            imageFormat: $("#imageformat").val(),
            minSize: $("#minsize").val(),
            maxSize: $("#maxsize").val(),
        });
    }
}

```

Figura 15: Estrutura dos parâmetros dos pedidos digitalização de impressão digital (esquerda) e digitalização de cheques (direita).

O utilizador selecciona o tipo de pedido que quer fazer, e o formulário para cada tipo de pedido altera-se dependendo desta escolha, de forma a que sejam apresentados os campos necessários à submissão do pedido no formato correto, como os formatos da figura anterior.

<div>Print File</div> <div> <b>Printer ID:</b> <input type="text"/> </div> <div> <b>Number of copies:</b> <input type="text"/> <small>Nr. of copies</small> </div> <div> <b>Sheet Size:</b> <input type="text"/> </div> <div> <b>Margin:</b> <input type="text"/> </div> <div> <b>Pages per sheet:</b> <input type="text"/> </div> <div> <b>Data:</b> <input type="text"/> <small>Your message here</small> </div>	<div>Scan File</div> <div> <b>Scanner ID:</b> <input type="text"/> </div> <div> <b>Quality:</b> <input type="text"/> </div> <div> <b>Sheet Size:</b> <input type="text"/> </div> <div> <b>Margin:</b> <input type="text"/> </div> <div> <b>File type:</b> <input type="text"/> </div>
<div>Fingerprint Scan</div> <div> <b>Scanner ID:</b> <input type="text"/> </div> <div> <b>Finger Id:</b> <input type="text"/> </div> <div> <b>Capture Mode:</b> <input type="text"/> </div> <div> <b>Margin:</b> <input type="text"/> </div>	<div>Check Scan</div> <div> <b>Scanner ID:</b> <input type="text"/> </div> <div> <b>Image Format:</b> <input type="text"/> </div> <div> <b>Image Color:</b> <input type="text"/> </div> <div> <b>Minimum Size:</b> <input type="text"/> <small>Min. Size</small> </div> <div> <b>Maximum Size:</b> <input type="text"/> <small>Max. Size</small> </div>

Figura 16: Botões de escolha de tipo de pedido e respectivos formulários.



#### 4.4.4. Comunicação Socket

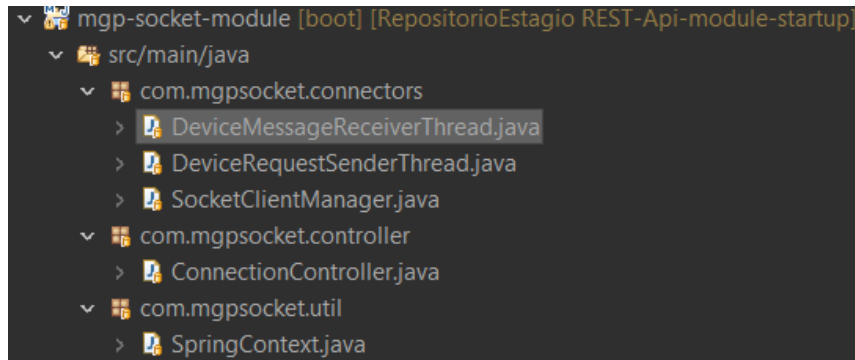


Figura 17: Estrutura de ficheiros do módulo de comunicação por socket.

A comunicação por Socket, como o nome indica, é realizada através de sockets Java, através dos quais os dispositivos estabelecem uma conexão com o servidor. Em vez de utilizar um protocolo de mensagens como na comunicação WebSocket, como as mensagens trocadas pelo servidor e os dispositivos seguem um formato muito mais simples que e não necessitam de mecanismos de gestão de mensagens por canais (cada dispositivo tem um socket específico atribuído e não há broadcasts), foi desenvolvido um protocolo bastante simples de comunicação pelo formato de strings. Estas strings primeiramente têm a sua informação separada por uma string (os três caracteres “:-:”) e o último parâmetro é por sua vez analisado como JSON. A primeira separação inicialmente era com dois pontos “:”, no entanto teve de ser alterada para a string arbitrária indicada. Isto deve-se ao facto do formato JSON e de algumas porções de texto incluírem os dois pontos ou outros caracteres o que levava ao corte indevido das strings, tendo isto sido corrigido desta forma.

##### 4.4.4.1. Lado do Servidor

Do lado do servidor, este está constantemente à espera de conexões por parte de dispositivos. Assim que um dispositivo tenta conectar-se, o servidor abre uma instância Socket na porta indicada, abre uma thread para lidar com o processo de conexão oficial do dispositivo exclusiva a este e aguarda a comunicação da identificação do dispositivo. Assim que o dispositivo comunica a sua identificação (nome e tipo de dispositivo) este é ou registado e definido como conectado ou simplesmente definido como conectado. Ao ser conectado o protocolo de comunicação sofre um upgrade, mudando de estado. Esta mudança de estado implica duas situações:

1. A thread que estava à escuta das mensagens do dispositivo para oficializar a conexão passa a ser a thread que regularmente verifica a base de dados para ver se existe algum pedido destinado ao dispositivo conectado;
2. A primeira thread cria outra thread que inclui uma referência ao socket, e começa a verificar regularmente se o dispositivo associado enviou alguma mensagem pelo socket ou não.

Assim, estas duas threads gerem tanto o envio como receção de mensagens por socket para os dispositivos.

A thread que envia mensagens para o dispositivo estará daí em diante responsável por apenas uma coisa: verificar a base de dados, e enviar os pedidos que encontra destinados ao seu dispositivo para o mesmo.

Já a thread que verifica o socket para eventuais mensagens do dispositivo tem a função de analisar as mensagens do mesmo, que podem ser de dois tipos: ou notificam a receção de um pedido

#### 4.4.4.2. Lado do Simulador de Dispositivos

O simulador de dispositivos tem duas responsabilidades. Simular o processo de conexão dos dispositivos realizando uma conexão consistente com o protocolo de comunicação descrito secção do lado do servidor, e receber os pedidos. Após receber um pedido, notifica o servidor da receção e processa-o, devolvendo uma resposta automática gerada consoante o tipo de pedido. Esta resposta automática é processada através da conversão da string JSON que contém o tipo de pedido e os parâmetros do mesmo, sendo esta devolvida por socket para ser entregue ao utilizador que originou o pedido.

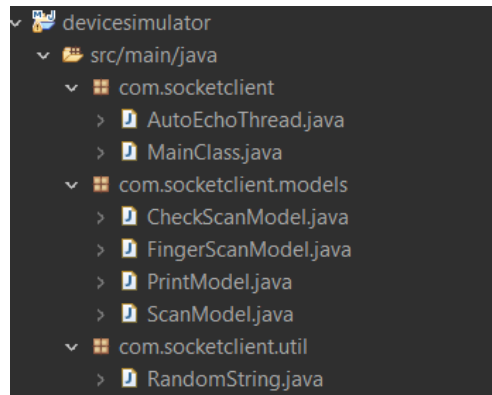


Figura 18: Estrutura de ficheiros do simulador de dispositivos.

#### 4.4.5. API REST

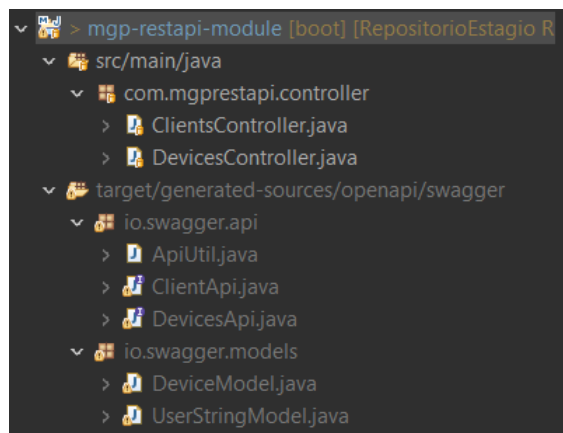


Figura 19: Estrutura de ficheiros do módulo da API REST.

O módulo da API REST tem o objetivo de disponibilizar os controladores necessários para o Spring Boot integrar os end-points da API como componentes funcionais.

Uma das particularidades deste módulo, é a existência do Swagger Codegen. Através da especificação da API desejada a partir de um ficheiro .yaml o plugin gera interfaces que estabelecem os contratos e as anotações que qualificam os seus métodos como end-points REST para o Swagger detectar e documentar. Ao implementar estas interfaces e assinalando a classe que as implementa com a anotação “@RestController” o Spring Boot passa a reconhecê-las e os end-points das mesmas passam a estar acessíveis.

```

@Api(value = "client", description = "the client API")
public interface ClientApi {

    @ApiOperation(value = "Get All Active Users", nickname = "getAllclients")
    @ApiResponse(value = {
        @ApiResponse(code = 200, message = "OK", response = UserStringModel)
    })
    @GetMapping(
        value = "/client/active",
        produces = { "application/json" }
    )
    default ResponseEntity<List<UserStringModel>> getAllclients() {
        return new ResponseEntity<>(HttpStatus.NOT_IMPLEMENTED);
    }
}

@RestController
public class ClientsController implements ClientApi{

    @Override
    public ResponseEntity<List<UserStringModel>> getAllclients() {

        List<UserStringModel> list = new ArrayList<UserStringModel>();
        List<String> clients = clientRegistry.getClientList();

        for(int i = 0; i<clients.size();i++) {
            UserStringModel user = new UserStringModel();
            user.setId(clients.get(i));
            list.add(user);
        }
        return ResponseEntity.ok(list);
    }
}

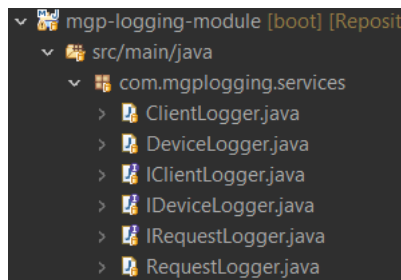
```

*Figura 20: Interface gerada automaticamente pelo Swagger para os pedidos relacionados com clientes e respetiva implementação.*

Neste módulo foram implementados dois end-points relevantes ao objetivo final, e outros dois para efeitos de simulação. Os de efeito de simulação servem apenas para ativar e desativar um dispositivo na base de dados para testes de front-end. Já os dois relevantes aos propósitos finais de monitorização servem para obter as listas dos utilizadores e dos dispositivos conectados.

#### 4.4.6. Sistema de registo de operações (logs)

O sistema de logs foi um dos módulos que levou mais tempo a desenvolver. Não pela sua complexidade mas pelo tempo que levou a ser aperfeiçoado para tornar os registos o mais informativos e relevantes possível. Este foi um dos aspectos mais reforçados pela orientação do Eng. Carlos Ramos, e a importância deste tipo de funcionalidades é um dos conhecimentos que terei para sempre em conta no meu progresso profissional.



*Figura 21: Estrutura de ficheiros do módulo de registo de operações.*

Para este módulo foram desenvolvidos três loggers diferentes, um para as operações relativas a dispositivos, outro para as operações relativas a pedidos de utilizadores e um terceiro para as operações relativas à conexão de utilizadores.

Exemplos do resultado destes loggers a funcionar serão incluídos na secção de demonstração.

#### 4.5.7. Acesso à base de dados

Relativamente ao módulo de acesso à base de dados, foram implementados dois serviços distintos: um para operações relacionadas com a tabela de dispositivos e outro para operações relacionadas com a tabela de pedidos.

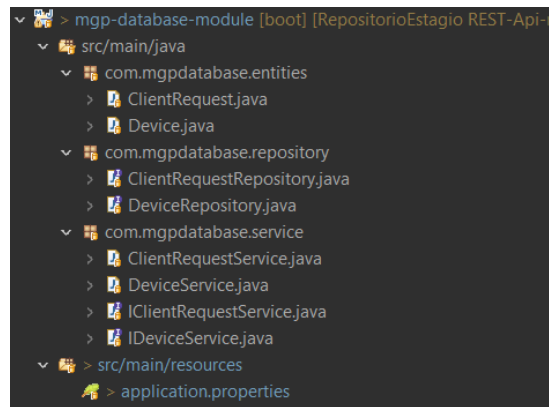


Figura 22: Estrutura de ficheiros do módulo de acesso à base de dados.

Um dos pontos importantes deste módulo, é a conexão com a base de dados. Ao incluir o ficheiro de configuração "application.properties", o Spring Boot detecta os parâmetros de conexão e tenta realizar a mesma. Caso falhe, a aplicação falha também a sua inicialização.

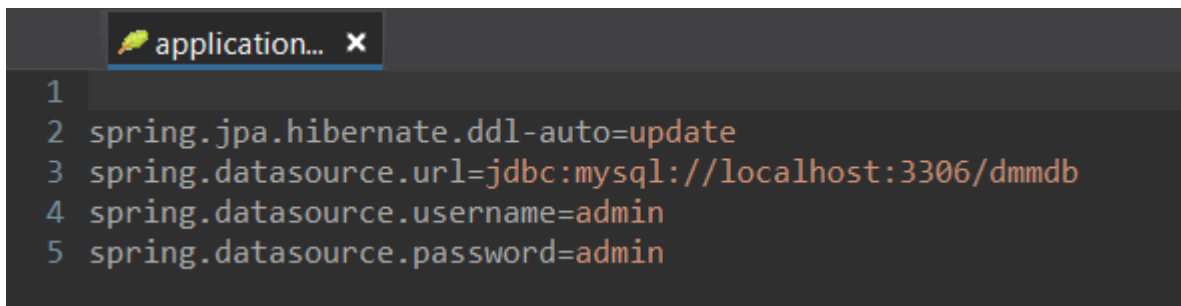


Figura 23: Ficheiro de configuração do acesso à base e dados.

Tendo a conexão à base de dados sido efectuada com sucesso, o Hibernate acede à mesma e cria as tabelas associadas às entidades criadas como classes caso estas não existam ainda no *schema* da mesma. Os modelos criados para as entidades são os demonstrados nas figuras seguintes:

<pre> @Entity @Table(name = "request") public class ClientRequest {      @Id     @GeneratedValue(strategy=GenerationType.AUTO)     private int id;      @Column     private String origin;      @Column     private String eventType;      @Column     private int deviceId;      @Column     private String requestData;      @Column     private String state;      @Column     private String response; </pre>	<pre> @Entity @Table public class Device {      @Id     @GeneratedValue(strategy=GenerationType.AUTO)     private int id;      @Column     private String deviceName;      @Column     private String state;      @Column     private String type; </pre>
---	---

Figura 24: Modelos criados para representar as entidades da base de dados.

Depois de serem definidas as entidades, para poderem ser feitas queries às tabelas respectivas foi necessário criar as “classes repositório”. Estas permitem fazer queries à base de dados através de métodos. A figura seguinte, do repositório da classe “Device” apresenta alguns destes métodos:

```

@Repository
public interface DeviceRepository extends JpaRepository<Device,Integer>{

    List<Device> findAll();

    List<Device> findByState(String state);

    Optional<Device> findByDeviceName(String deviceName);

    Optional<Device> findByIdAndState(int id, String state);

    Optional<Device> findByDeviceNameAndState(String deviceName, String state);

    @Transactional
    @Modifying
    @Query("update Device d set d.state = ?1 where d.id = ?2")
    int setDeviceStateById(String state, Integer id);

```

Figura 25: Classe repositório da entidade “device”.

O método “findByState” realiza uma query que retorna todos os tuplos da tabela cuja coluna “state” corresponde ao parâmetro de entrada usado no método. De forma análoga, o método “findByIdAndState” realiza uma query que retorna todos os tuplos da tabela cujas colunas “state” e “id” correspondem a ambos os parâmetros de entrada do método, e por aí em diante. Adicionalmente, como

podemos ver no último método da classe, é possível realizar uma query SQL explicitamente através da forma demonstrada neste.

Foi também criada uma classe repositório semelhante para a entidade “ClientRequest”, e para ambas as entidades foi criada uma interface que estabelecia os métodos necessários a implementar e utilizar nos serviços a desenvolver. Abaixo está apresentada uma destas interfaces.

```
public interface IDeviceService {  
  
    List<Device> getActiveDevices();  
  
    boolean isDeviceActiveByDeviceName(String deviceName);  
  
    Optional<Device> checkIfDeviceExists(String deviceName);  
  
    Optional<Device> registerNewDevice(Device newDevice);  
  
    boolean setDeviceState(int id, String state);  
  
    boolean isDeviceActiveById(int id);  
  
    void cleanDeviceRepository();  
}
```

*Figura 26: Interface IDeviceService.*

## 5. DEMONSTRAÇÃO

Tendo apresentado a solução e os aspectos mais importantes da sua implementação, realizarei agora uma demonstração que visa apresentar a aplicação em funcionamento, passo a passo com todos os módulos e aspectos desta a intervir.

Os passos gerais e procedimento que irei tomar será o seguinte:

1. Inicializar a aplicação;
2. Conectar dois utilizadores através do browser;
3. Conectar três dispositivos de tipos diferentes;
4. Mostrar a API REST a funcionar;
5. Enviar um pedido de cada tipo;
6. Mostrar as mensagens geradas em ambos os extremos;
7. Desconectar todos os intervenientes;
8. Mostrar os logs gerados;

Desta forma espero que seja claro o funcionamento da aplicação e todos os seus detalhes.

Note-se ainda que apenas irei mostrar partes relevantes dos ecrãs devido ao tamanho destes aumentar demasiado a extensão deste relatório e o espaço desperdiçado das páginas do mesmo.

### 5.1. INICIALIZAÇÃO DA APLICAÇÃO

Para inicializar a aplicação, é bastante simples. Primeiro assegura-se que a base de dados à qual nos estamos a conectar está ligada e que os parâmetros de conexão estão corretos. Segundo, assegura-se que as portas para as quais o web server da aplicação e a conexão por socket estão

```

=====
:: Spring Boot ::                (v2.4.4)

05:02:23:022 26-06-2021 [main] c.m.a.MgpServer
05:02:23:059 26-06-2021 [main] c.m.a.MgpServer
05:02:24:262 26-06-2021 [main] o.s.d.r.c.RepositoryConfigurationDelegate
05:02:24:358 26-06-2021 [main] o.s.d.r.c.RepositoryConfigurationDelegate
05:02:25:704 26-06-2021 [main] o.s.b.w.e.t.TomcatWebServer
05:02:25:725 26-06-2021 [main] o.a.c.h.Http1NioProtocol
05:02:25:728 26-06-2021 [main] o.a.c.c.StandardService
05:02:25:729 26-06-2021 [main] o.a.c.c.StandardEngine
05:02:25:979 26-06-2021 [main] o.a.c.c.c.[./]
05:02:25:980 26-06-2021 [main] o.s.b.w.s.c.ServletWebServerApplicationContext
05:02:26:250 26-06-2021 [main] o.h.j.i.u.LogHelper
05:02:26:365 26-06-2021 [main] o.h.Version
05:02:26:714 26-06-2021 [main] o.h.a.c.Version
05:02:26:923 26-06-2021 [main] c.z.h.HikariDataSource
05:02:27:770 26-06-2021 [main] c.z.h.HikariDataSource
05:02:27:851 26-06-2021 [main] o.h.d.Dialect
05:02:29:363 26-06-2021 [main] o.h.e.t.j.p.i.JtaPlatformInitiator
05:02:29:378 26-06-2021 [main] o.s.o.j.LocalContainerEntityManagerFactoryBean
05:02:29:380 26-06-2021 [main] o.s.s.c.ThreadPoolTaskExecutor
05:02:29:382 26-06-2021 [main] o.s.s.c.ThreadPoolTaskExecutor
05:02:29:384 26-06-2021 [main] o.s.s.c.ThreadPoolTaskExecutor
05:02:29:386 26-06-2021 [main] o.s.s.c.ThreadPoolTaskScheduler
05:02:29:388 26-06-2021 [main] o.s.b.a.o.j.JpaBaseConfiguration$JpaWebConfiguration
05:02:29:390 26-06-2021 [main] o.s.b.a.w.s.WelcomePageHandlerMapping
05:02:32:045 26-06-2021 [main] o.a.c.h.Http1NioProtocol
05:02:32:114 26-06-2021 [main] o.s.b.w.e.t.TomcatWebServer
05:02:32:115 26-06-2021 [main] o.s.m.s.b.SimpleBrokerMessageHandler
05:02:32:115 26-06-2021 [main] o.s.m.s.b.SimpleBrokerMessageHandler
05:02:32:116 26-06-2021 [main] o.s.m.s.b.SimpleBrokerMessageHandler
05:02:32:130 26-06-2021 [main] c.m.a.MgpServer
[Socket Listener] - Initialization Successful

- Starting MgpServer using Java 11.0.10 on ricardoalex with PID
- No active profile set, falling back to default profiles: default
- Bootstrapping Spring Data JPA repositories in DEFAULT mode.
- Finished Spring Data repository scanning in 85 ms. Found 2 JPA
  Tomcat initialized with port(s): 8080 (http)
- Initializing ProtocolHandler ["http-nio-8080"]
- Starting service [Tomcat]
- Starting Servlet engine: [Apache Tomcat/9.0.44]
- Initializing Spring embedded WebApplicationContext
- Root WebApplicationContext: initialization completed in 2764 ms
- HH0000204: Processing PersistenceUnitInfo [name: default]
- HH0000412: Hibernate ORM core version 5.4.29.Final
- HCANN000001: Hibernate Commons Annotations {5.1.2.Final}
- HikariPool-1 - Starting...
- HikariPool-1 - Start completed.
- HH0000400: Using dialect: org.hibernate.dialect.MySQL8Dialect
- HH0000490: Using JtaPlatform implementation: [org.hibernate.eng
- Initializing JPA EntityManagerFactory for persistence unit 'defa
- Initializing ExecutorService 'clientInboundChannelExecutor'
- Initializing ExecutorService 'clientOutboundChannelExecutor'
- Initializing ExecutorService 'brokerChannelExecutor'
- Initializing ExecutorService 'messageBrokerTaskScheduler'
- spring.jpa.open-in-view is enabled by default. Therefore, dat
  Adding welcome page template: index
- Starting ProtocolHandler ["http-nio-8080"]
- Tomcat started on port(s): 8080 (http) with context path ''
- Starting...
- BrokerAvailabilityEvent[available=true, SimpleBrokerMessageHand
  Started.
- Started MgpServer in 9.89 seconds (JVM running for 12.738)

//--- SERVER SUCCESSFULLY STARTED ---//

```

A ausência de erros nesta fase indica que a aplicação inicializou com sucesso.

A conexão de clientes é outro processo bastante simples. Basta abrir o browser, e abrir dois separadores diferentes na página da aplicação. No meu caso, como estou a desenvolver a aplicação localmente posso fazer isto através do endereço "localhost:8080".



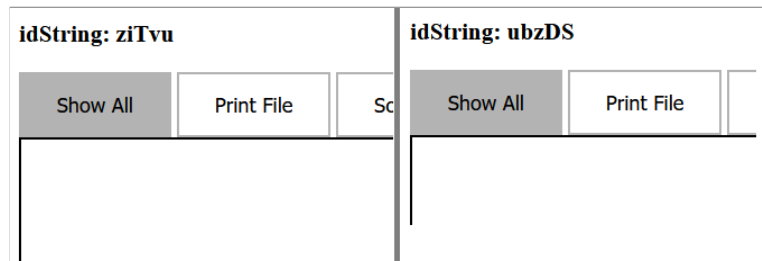


Figura 28: Strings de identificação dos clientes conectados.

De seguida, irei inicializar dois dispositivos através do simulador de dispositivos.

Este simulador funciona da seguinte forma: ao ser iniciada a aplicação esta conecta-se automaticamente ao servidor por socket através da porta pré-definida, e depois tenta estabelecer uma ligação oficial através da comunicação do seu nome e do seu tipo ao servidor.

Assim, irei inicializar três dispositivos diferentes.

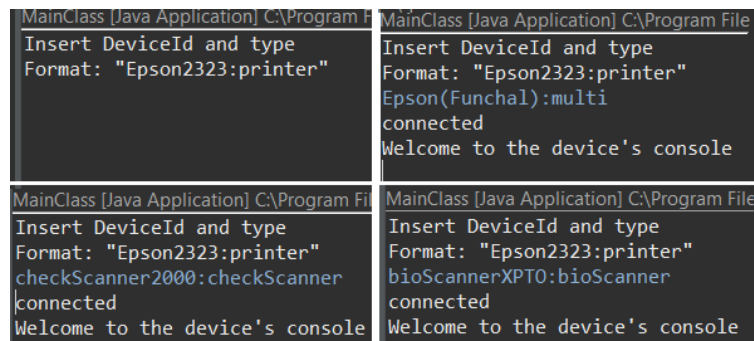


Figura 29: Consola à espera de input (cima à esquerda), simulador da multifunções “Epson(Funchal)” (cima à direita), simulador do leitor de cheques “checkScanner2000” (baixo à esquerda) e simulador do leitor de impressões digitais “bioScannerXPT0” (baixo à direita)

Após conectar estes três dispositivos, já podemos observar o broadcast da lista de dispositivos conectados no browser, em ambos os utilizadores.

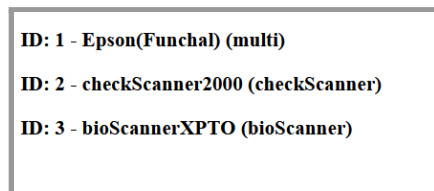


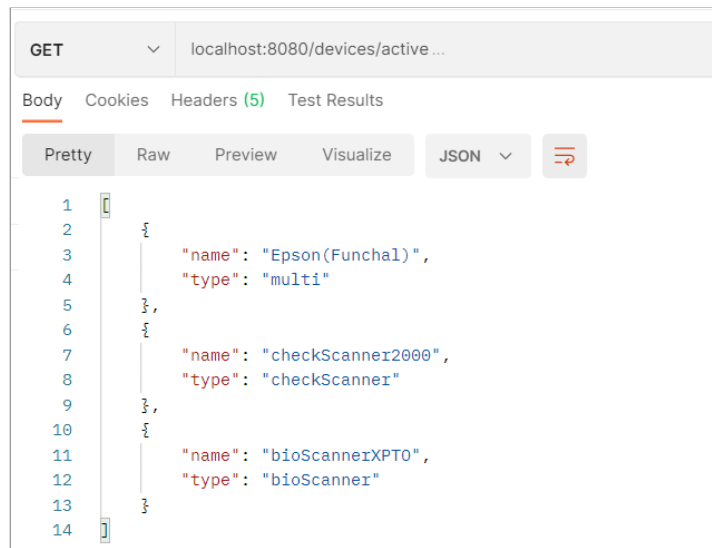
Figura 30: Resultado do broadcast dos dispositivos conectados, do lado do cliente.

De seguida, iremos testar a API REST, e verificar se os atores conectados aparecem nos pedidos REST efectuados.

### 5.3. TESTE DA API REST

Para verificar a API REST em funcionamento recorrerei à aplicação Postman, através da qual irei realizar os pedidos REST pretendidos.

Primeiro, realizarei um pedido para obter todos os dispositivos conectados:



*Figura 31: Pedido de obtenção da lista dos dispositivos conectados por API REST.*

Verificando que os dispositivos coincidem com os inicializados na etapa anterior, concluímos que esta funcionalidade está a funcionar corretamente e cumpre a sua função.

De seguida irei testar o outro pedido, para obter os utilizadores conectados:

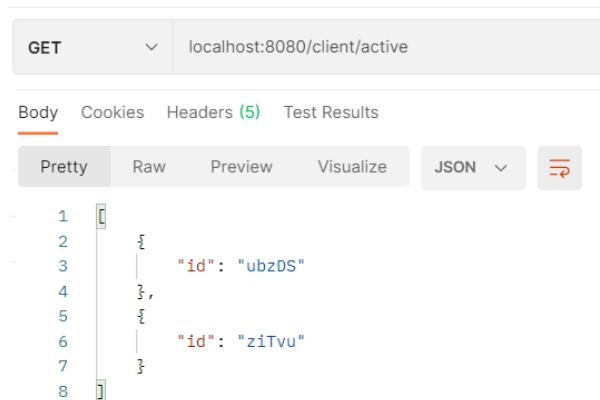


Figura 32: Pedido de obtenção da lista dos utilizadores conectados por API REST.

Tal como o pedido anterior, este devolveu os resultados pretendidos, uma vez que estas strings de conexão são exatamente as mesmas das geradas aleatoriamente para os utilizadores. Podemos concluir que esta funcionalidade também está a funcionar como o pretendido.

#### 5.4 ENVIO DE PEDIDOS

De seguida, iremos demonstrar a funcionalidade principal da aplicação, o envio de pedidos e retorno de mensagens de notificação e respostas. Para isto irei enviar dois pedidos diferentes a partir de um dos utilizadores e apresentar os outputs apresentados em ambos os lados:

A partir do utilizador “ubzDS” vou então submeter um pedido de impressão e um pedido de digitalização:

<p><b>Printer ID:</b> Epson(Funchal) ▾</p> <p><b>Number of copies:</b> 2</p> <p><b>Sheet Size:</b> A4 ▾</p> <p><b>Margin:</b> Normal ▾</p> <p><b>Pages per sheet:</b> 6 ▾</p> <p><b>Data:</b> Ficheiro1</p> <p><b>Submit</b></p>	<p><b>Scanner ID:</b> Epson(Funchal) ▾</p> <p><b>Quality:</b> High ▾</p> <p><b>Sheet Size:</b> A5 ▾</p> <p><b>Margin:</b> Extended ▾</p> <p><b>File type:</b> Document ▾</p> <p><b>Submit</b></p>
--	---

Figura 33: Parâmetros seleccionados nos pedidos da demonstração.

Ao submeter estes pedidos, estas foram as notificações do lado do simulador:

```
Epson(Funchal):multi
connected
Welcome to the device's console
Request received - id:4, type: print Data: {"copyAmmount":"2","sheetSize":"A4","margin":"Normal","sheetPages":"6","data":"Ficheiro1"}
Responded to request - id: 4, Response: Printed 2 copies of Ficheiro1 with 6 sheets per page.
Request received - id:5, type: scan Data: {"quality":"High","sheetSize":"A5","margin":"Extended","type":"Document"}
Responded to request - id: 5, Response: File j9IRJuIuat was scanned with 'High' quality. DocType: 'Document', Size: 'A5'
```

*Figura 34: Mensagens apresentadas no simulador do dispositivo "Epson(Funchal)".*

Como se pode observar, os parâmetros dos pedidos coincidem com os parâmetros inseridos do lado do utilizador. De seguida apresentarei as notificações que o utilizador recebeu para estes pedidos:

**Request to device 'Epson(Funchal)' registered (request id: 4)**

**Request with id 4 has been sent to device id Epson(Funchal)**

**Request id 4 complete. Response: Printed 2 copies of Ficheiro1 with 6 sheets per page.**

**Request to device 'Epson(Funchal)' registered (request id: 5)**

**Request with id 5 has been sent to device id Epson(Funchal)**

**Request id 5 complete. Response: File j9IRJuIuat was scanned with 'High' quality. DocType: 'Document', Size: 'A5'**

*Figura 35: Mensagens apresentadas no browser do utilizador que realizou os pedidos.*

Como podemos também observar, tanto as notificações apresentadas ao utilizador coincidem com os eventos que aconteceram do lado do dispositivo, como as respostas geradas por este são baseadas nos parâmetros do pedido submetidos. Sendo assim, podemos concluir que a comunicação está a funcionar corretamente.

## 5.5. Logs

Por fim, sobra apenas demonstrar os registos que aconteceram como consequência destas interações com a aplicação. Para isto, irei desconectar os utilizadores e os dispositivos terminando as aplicações de simulação e fechando os separadores do browser.

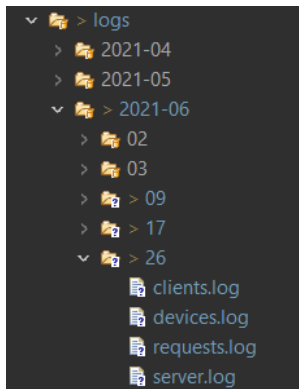


Figura 36: Estrutura de armazenamento dos logs.

Acedendo ao diretório dos logs e abrindo os ficheiros, poderei observar os registos efectuados pela aplicação. Sendo assim, veremos primeiro o ficheiro relativamente aos clientes, o “clients.log”:

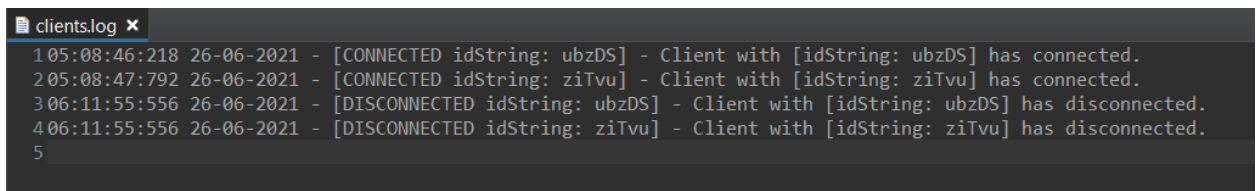


Figura 37: Ficheiro “clients.log” depois dos pedidos da demonstração.

Podemos observar na figura que os logs relativamente aos utilizadores foram realizados e registados com sucesso, tanto relativamente à conexão como à desconexão.

Passamos então ao ficheiro de registos relativamente aos dispositivos, o “devices.log”:

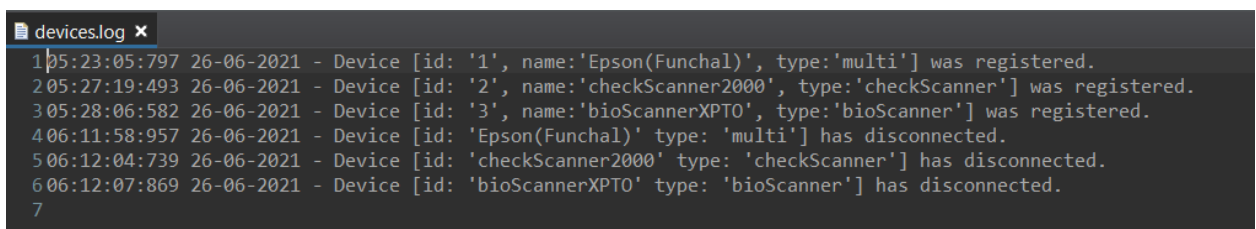
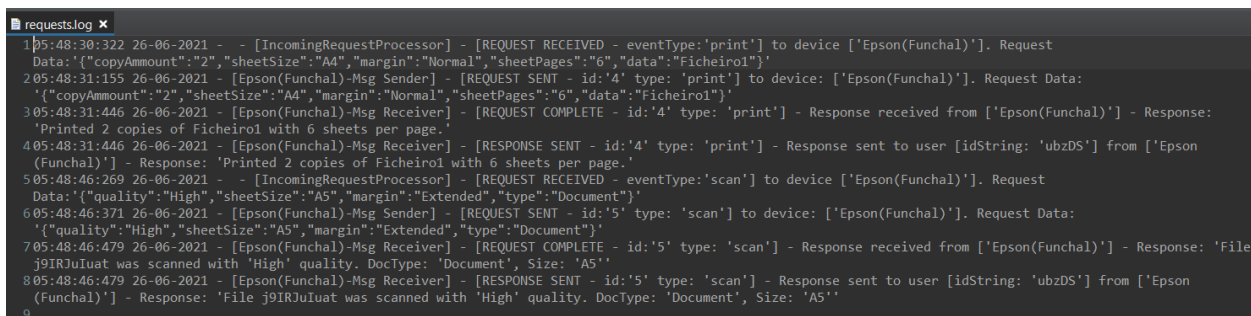


Figura 38: Ficheiro “devices.log” depois da conexão e desconexão dos dispositivos da demonstração.

Uma vez que os dados da base de dados da tabela de dispositivos foram limpos antes da demonstração, o log indica que estes foram registados, e assim conectados. Quando isto ocorre não há mensagem de conexão, uma vez que o registo ocorre automaticamente na primeira conexão de um dispositivo, assumindo-se assim que esse registo também se trata de uma conexão.

Para além disto, também foram registadas com sucesso as mensagens de desconexão, pelo que dá para concluir que estes mecanismos também estão a funcionar corretamente.

Por último irei então mostrar os registos relacionados com os pedidos efectuados, no ficheiro "requests.log":



```
requests.log x
15:48:30:322 26-06-2021 - [IncomingRequestProcessor] - [REQUEST RECEIVED - eventType:'print'] to device ['Epson(Funchal)']. Request
Data: '{"copyAmount": "2", "sheetSize": "A4", "margin": "Normal", "sheetPages": "6", "data": "Ficheiro1"}'
205:48:31:155 26-06-2021 - [Epson(Funchal)-Msg Sender] - [REQUEST SENT - id:'4' type: 'print'] to device: ['Epson(Funchal)']. Request Data:
'{"copyAmount": "2", "sheetSize": "A4", "margin": "Normal", "sheetPages": "6", "data": "Ficheiro1"}'
305:48:31:446 26-06-2021 - [Epson(Funchal)-Msg Receiver] - [REQUEST COMPLETE - id:'4' type: 'print'] - Response received from ['Epson(Funchal)'] - Response:
'Printed 2 copies of Ficheiro1 with 6 sheets per page.'
405:48:31:446 26-06-2021 - [Epson(Funchal)-Msg Receiver] - [RESPONSE SENT - id:'4' type: 'print'] - Response sent to user [idString: 'ubzDS'] from ['Epson
(Funchal)'] - Response: 'Printed 2 copies of Ficheiro1 with 6 sheets per page.'
505:48:46:269 26-06-2021 - [IncomingRequestProcessor] - [REQUEST RECEIVED - eventType:'scan'] to device ['Epson(Funchal)']. Request
Data: '{"quality": "High", "sheetSize": "A5", "margin": "Extended", "type": "Document"}'
605:48:46:371 26-06-2021 - [Epson(Funchal)-Msg Sender] - [REQUEST SENT - id:'5' type: 'scan'] to device: ['Epson(Funchal)']. Request Data:
'{"quality": "High", "sheetSize": "A5", "margin": "Extended", "type": "Document"}'
705:48:46:479 26-06-2021 - [Epson(Funchal)-Msg Receiver] - [REQUEST COMPLETE - id:'5' type: 'scan'] - Response received from ['Epson(Funchal)'] - Response: 'File
j9IRJuIuat was scanned with 'High' quality. DocType: 'Document', Size: 'A5'
805:48:46:479 26-06-2021 - [Epson(Funchal)-Msg Receiver] - [RESPONSE SENT - id:'5' type: 'scan'] - Response sent to user [idString: 'ubzDS'] from ['Epson
(Funchal)'] - Response: 'File j9IRJuIuat was scanned with 'High' quality. DocType: 'Document', Size: 'A5'
9
```

Figura 39: Ficheiro "requests.log" após a realização dos pedidos da demonstração.

Ao observar os registos com atenção podemos verificar que todas as etapas pretendidas do processo de registo e tratamento de um pedido foram realizadas e registadas no ficheiro. Primeiro o pedido foi recebido ( "REQUEST RECEIVED" ), tendo sido enviado para o dispositivo de seguida ( "REQUEST SENT" ). Assim que a resposta ao pedido foi recebida pelo servidor, este registou a resposta na base de dados e fez o registo deste facto ( "RESPONSE COMPLETE " ). Depois disto, verificou que o utilizador ainda se encontrava conectado e enviou o pedido de volta, registando também este evento ( "RESPONSE SENT" ).

## 6. DISCUSSÃO

Chegando ao fim do desenvolvimento do projeto é necessário perceber se a solução desenvolvida satisfaz os objetivos ou não. Para isto, é necessário ter a noção que inicialmente não existia um objetivo final bem definido relativamente à complexidade da aplicação. Desde o início que do lado da Asseco sempre me foi dito que era suposto aprender bem os conceitos e compreender as ferramentas, e a complexidade da versão final seria uma consequência da rapidez da minha habilidade de aprendizagem e de aplicar os conhecimentos. Assim, discutirei a qualidade e da solução e o quanto está se adequa ao problema apresentado tendo isto em conta.

Relativamente à adaptação da solução para Java 8 utilizando as tecnologias indicadas pela Asseco, considera-se que o objetivo foi cumprido. Através da versatilidade do Spring Boot e o quanto este facilitou o desenvolvimento e integração de ferramentas foi possível produzir uma aplicação que efetivamente atinge o objetivo de gerir a comunicação entre dispositivos e utilizadores. Não só gere a comunicação, mas através do formato JSON e dos mecanismos que o transmitem é permitido que com um mínimo de informação adicional (origem, destino e tipo de evento) este transmita qualquer tipo de informação sendo agnóstico ao que está a transportar.

Para além da capacidade de transmitir bem estas informações, é ainda bastante fácil por esta razão adicionar tipos de mensagens. Os utilizadores ao estar conectados por websocket poderão enviar virtualmente qualquer coisa em formato JSON desde que está informação seja encapsulada da forma

correta no protocolo de mensagens, ficando à responsabilidade dos dispositivos de interpretar estas mensagens (o meio vale para o sentido inverso de comunicação).

Um dos maiores problemas da aplicação, no entanto, é o código não estar desenhado em si da forma mais correta, não tendo sido utilizado nenhum padrão de desenho para além dos singletons do próprio Spring Boot. Implementar padrões de desenho teria melhorado bastante a capacidade de expansão de funcionalidades da aplicação em si e também teria ajudado a diminuir bastante o acoplamento entre componentes e módulos.

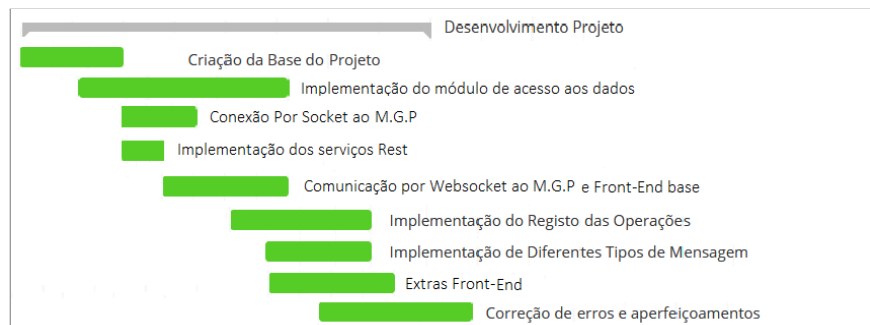
Outra das questões que não está no formato adequado são as queries à base de dados, que tem imenso espaço para melhoramento. Por exemplo, ao realizar inserções é realizada outra pesquisa para obter o ID da entidade inserida, mas seria muito mais eficiente reduzir isto para uma query que devolve imediatamente este valor. Este e outros melhoramentos em queries são alterações que melhorariam imenso o desempenho da aplicação.

Por fim, resta apenas mencionar outra das falhas principais de implementação que foi o tratamento de erros no módulo de comunicação por socket, uma vez que tem vários tipos de exceções no mesmo bloco de código relacionadas com várias causas diferentes, e originalmente estas foram tratadas de forma a permitir que a aplicação avançasse sem problemas, e não de forma a estes erros poderem ser analisados e compreendidos para a própria aplicação ser melhorada nestes pontos.

No entanto, com todos estes pontos pude perceber que situações me passaram ao lado inicialmente, de forma a que possa ter atenção a estas e outras que consegui resolver em projetos futuros.

## 7. PROGRESSO

A implementação foi dividida em vários passos como poderemos ver na figura seguinte, excerto do cronograma apresentado anteriormente:



*Figura 40: Excerto do cronograma apresentado no início do relatório, referente ao desenvolvimento do projeto.*

Não é possível observar uma divisão lógica de fases muito clara através da figura, no entanto houveram quatro fases principais:

### 7.1. PRIMEIRA FASE: CRIAÇÃO DA BASE DO PROJETO:

Esta fase estendeu-se um pouco mais do que o indicado na previsão do tempo dedicado às tarefas no plano inicial. A razão principal deveu-se a problemas iniciais com a minha compreensão

relativamente a questões de gestão de módulos e dependências, tanto relativamente ao Maven como ao Spring Boot. Antes deste projeto nunca me tinha sido necessário utilizar ferramentas do tipo nem implementar projetos com uma estrutura deste género, tendo sido isto um obstáculo. No entanto, consegui compreender estes mecanismos o suficiente a permitir-me avançar no projeto, tendo sido o obstáculo transposto.

Durante esta fase, foi onde ocorreu a maior parte do desenvolvimento do módulo de acesso à base de dados. A razão para isto foi o facto de o objetivo desta primeira fase do trabalho o necessitar, uma vez que sem os registos de dispositivos e de pedidos teria sido impossível efectuar a comunicação base que se pretendia.

Para além de ter sido um ponto fulcral do desenvolvimento da base de dados, também foi neste ponto que ocorreu o desenvolvimento principal da comunicação por WebSocket com utilizadores, e por Socket com dispositivo.

O produto final desta fase de desenvolvimento foi portanto:

1. Acesso à base de dados funcional;
2. Front-End simples que simula um utilizador;
3. Aplicação simples que simula um dispositivo;
4. Protótipo dos canais WebSocket funcional;
5. Protótipo da comunicação Socket funcional;
6. Comunicação Utilizador - Aplicação - Dispositivo rudimentar funcional, com ida e volta.

Posto isto, a base do projeto estava completa, embora com muitos pontos com espaço para melhoramento. A este ponto a comunicação embora funcional dependia demasiado de sequências pré-determinadas em que ou a aplicação ou os dispositivos ficavam à espera de feedback. As operações de leitura e escrita do socket em especial contribuíam para este factor uma vez que um dispositivo lia e escrevia no socket através da mesma thread, sendo portanto necessário alternar sequencialmente entre um ou outro.

## **7.2. SEGUNDA FASE: MELHORAMENTOS E ADIÇÃO DE FUNCIONALIDADES**

Na segunda fase da implementação do projeto, foi onde ocorreu o maior número de mudanças no mesmo. Um dos aspectos principais foi o desenvolvimento dos mecanismos de monitorização da aplicação. Outro foi o grande melhoramento do protocolo de comunicação por socket com uma solução *multithreading*, em que os sockets passam a ter não só uma *thread* de comunicação sobre o seu socket mas sim duas, uma que efectua leitura sobre o socket e outra para escrita. De forma semelhante, a aplicação de simulação foi alterada também para funcionar como descrito. Assim, ambos os lados do canal socket podem enviar mensagens de forma assíncrona sem terem de esperar uma resposta para ler a próxima, bem como ler respostas e processá-las sem a atenção imediata do dispositivo.

Assim, as mudanças principais desta fase da aplicação foram as seguintes:

1. Protótipo do registo de operações funcional.
2. API REST funcional e capaz de ser expandida.



3. Comunicação por socket e simulador de dispositivos bastante melhorados.
4. Melhoramento da interface de utilizador.
5. Reformulamento de alguns blocos de código para versões melhoradas.

No final desta fase a aplicação já tinha ganho forma e funcionava sem problemas, no entanto ainda havia alguns pormenores a melhorar. O principal e que foi maior alvo de atenção por parte do Eng. Carlos Ramos foi os mecanismos de registo. Isto devido à importância que estes mecanismos têm em aplicações como esta, em que uma boa implementação de registo de operações facilita imenso monitorização do fluxo da aplicação e também a identificação de problemas ou *bugs*.

### **7.3. TERCEIRA FASE: ADIÇÕES FINAIS, CORREÇÃO DE ERROS E LIMPEZA DE CÓDIGO**

Durante o desenvolvimento desta fase do trabalho já começava a aproximar-se o fim do semestre e com este o tempo disponível para implementação. Desta forma, foi aqui que foi desenvolvida a última expansão de funcionalidades da aplicação: a adição de diferentes tipos de mensagem.

Relacionado com a adição de diferentes tipos de mensagens, esteve uma das partes mais trabalhosas do projeto: a limpeza do código e correção de erros. Uma vez que adicionar estes tipos implicava uma expansão da aplicação transversal a quase todos os módulos, isto permitiu-me observar que erros precisavam de correção e que funcionalidades precisavam de alterações. Desta forma, durante esta fase ocorreu toda uma limpeza do código do projeto em geral, deste renomeamento de variáveis, refactoring de nomes de classes e a separação dos módulos de WebSocket e de Socket que até aqui se encontravam juntos.

Assim, estes foram os pontos chave a reter desta fase:

1. Adição de diferentes tipos de mensagens:
  - a. Alterar funcionalidades de outros módulos que não o permitiam;
  - b. Implementação e integração dos tipos de mensagem;
2. Re-estruturação do front-end que passou do envio de uma simples mensagem de texto para quatro tipos de mensagem estruturada, cada um com o seu conjunto de parâmetros.
3. Re-estruturação do mecanismo de resposta do simulador de forma a gerar uma resposta automática consoante os parâmetros recebidos e o tipo de mensagem;
4. Conclusão do desenvolvimento do sistema de registo de operações;
5. Conclusão do desenvolvimento de funcionalidades dos módulos.
6. Limpeza geral do código e refactorings;
7. Correção de erros

Posto isto, no final desta fase o projeto já se encontrava muito próximo da sua forma final.

#### **7.4. QUARTA FASE: FASE FINAL**

Ao chegar a esta fase, estamos já no fim do semestre. O tempo é já escasso para poder aplicar grandes esforços a nível de implementações e alterações, e portanto esta fase foi apenas uma fase de análise e da implementação de uma última funcionalidade.

Nesta fase ocorreu também, presencialmente na Asseco PST Funchal uma sessão de demonstração em que pude apresentar ao líder dos serviços Middleware, o Eng. Fábio Santos, a aplicação em funcionamento. Foi ele que, por sua vez, pediu que implementasse a funcionalidade de controlo de utilizadores em memória, de forma a que fosse possível consultar através da API da aplicação que utilizadores se encontram conectados num dado momento.

Os pontos principais desta fase final são portanto:

1. Desenvolvimento do registo temporário de utilizadores conectados;
2. Correções finais do código.
3. Limpeza final do código.
4. Arranque do desenvolvimento do relatório.

#### **7. CONCLUSAO**

Chegando ao fim do estágio e do desenvolvimento do projeto, considero que foi uma excelente experiência de aprendizagem e que o saldo geral é bastante positivo. Ainda que a qualidade de certas partes do código não seja a mais desejável, a aplicação desenvolvida desempenha a sua função pretendida e implementa algumas funcionalidades interessantes que inicialmente achei que seriam muito mais complicadas do que realmente foram, sendo bastante satisfatório ultrapassar essas barreiras de aprendizagem e crescer como pessoa e profissional, consolidando os conhecimentos aprendidos tanto durante este estágio como na licenciatura.

A primeira fase de todo o processo, a fase de formação, foi a teve mais ocasiões das referidas no parágrafo anterior. Isto porque, como referido anteriormente neste relatório, nunca tinha trabalhado com nenhum das ferramentas e frameworks propostas, havendo muitos momentos de stress até conseguir compreender as mesmas e perceber que afinal não eram tão difíceis, e que é até mesmo mais simples utilizar as mesmas do que implementar as suas funções. Este facto foi uma das coisas mais importantes que aprendi e experienciei. Ao procurar ferramentas já implementadas adequadas à situação que se tem em mãos poupamos trabalho e tempo que iríamos gastar desnecessariamente.

Na parte do desenvolvimento da solução foi semelhante, no entanto não houve tantos impasses complexos para as minhas capacidades, devido às fase de formação. O que foi mais difícil compreender foram as questões de dependências e organização por módulos, tendo gasto algum tempo para perceber como o Maven gere isto de forma a que um projeto que engloba vários módulos corra em conjunto com estes. Tendo compreendido isto foi-me possível acelerar bastante o processo de implementação da solução. Todas as ferramentas utilizadas estão bem aproveitadas à exceção do socket pelas razões apresentadas na discussão, sendo que isto seria a parte a melhorar, em conjunto com algumas queries do módulo de acesso à base de dados.

As funcionalidades da aplicação são satisfatórias, no entanto há algumas a ser implementadas para esta aplicação atingir uma versão final adequada ao problema real para a qual está destinada. Primeiro seria interessante incluir um mecanismo de recuperação de logins de forma a que um utilizador não

perdesse as respostas dos pedidos que faz se perder a ligação ainda que apenas momentaneamente. Para além desta, deveria ser introduzida uma capacidade de encriptação de conteúdo.

Por fim, resta agradecer aos orientadores envolvidos por toda a orientação providenciada, que me permitiu compreender bem os problemas que enfrentei e superar muitas das barreiras com as quais lidei, e perceber sempre que estava atrasado ou a a i na direção errada, tendo sido um dos aspectos essenciais e centrais de todo o estágio.

## 8. REFERENCIAS

- [1] «Asseco PST- member of Asseco Group», *Asseco PST*. <https://www.pst.asseco.com> (acedido Jun. 16, 2021).
- [2] Framework Training UK, «Why is Java 8 more popular than Java 14?», *Framework Training*. <https://www.frameworktraining.co.uk/blog/why-is-java-8-more-popular-than-java-14/> (acedido Jun. 16, 2021).
- [3] «What is CI/CD?», <https://www.redhat.com/en/topics/devops/what-is-ci-cd> (acedido Jun. 16, 2021).
- [4] «Deploying Spring Boot Applications». <https://spring.io/blog/2014/03/07/deploying-spring-boot-applications> (acedido Jun. 16, 2021).
- [5] baeldung, «Spring Dependency Injection | Baeldung», Jan. 29, 2019. <https://www.baeldung.com/spring-dependency-injection> (acedido Jun. 16, 2021).
- [6] «Module and Software Components in Software Engineering». <https://www.includehelp.com/basics/module-and-software-components-in-software-engineering.aspx> (acedido Jun. 18, 2021).