



INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA (ISEL)

DEPARTAMENTO DE ENGENHARIA ELETRÓNICA E DE  
TELECOMUNICAÇÕES E COMPUTADORES (DEETC)

---

LEIM

LICENCIATURA EM ENGENHARIA INFORMÁTICA E MULTIMÉDIA  
UNIDADE CURRICULAR DE PROJETO

---

## I-Crash



Martim Pinheiro Alves (46286)  
Pedro Jorge (47498)

### *Orientadores*

---

<i>Professor Doutor</i>	Paulo Trigo
<i>Professor</i>	Paulo Vieira
<i>Professora Doutora</i>	Helga Henriques
<i>Professor</i>	José Faria

---

*Setembro, 2023*



# Resumo

Este projeto consiste no desenvolvimento de uma aplicação móvel para agilizar a gestão de *stock* de um carro de emergência.

A realização deste projeto tem em conta o desenvolvimento de um sistema de gestão de base de dados, de um servidor, da aplicação com interface gráfica e integração de um leitor de códigos QR e de matrizes de dados.

Este projeto foi motivado pelas extensas horas de trabalho aplicadas à gestão de *stock* de um carro de emergência, que impedem um(a) enfermeiro(a) de prestar serviços noutras áreas importantes, isto porque, até há data, não existe nenhum sistema/aplicação que permita agilizar o processo de gestão. Este processo é extenso e demorado, porque é necessária a verificação da validade de todos os consumíveis clínicos, atualmente feita a olho, sendo necessário ler um a um, as datas escritas com uma letra muito pequena.

Define-se como consumível clínico, todos os produtos e materiais descartáveis para uso num doente. Falando um pouco do carro de emergência. É uma estrutura móvel que contém um conjunto de dispositivos médicos, medicamentos e outros materiais, indispensáveis para a reanimação cárdio-respiratória. A sua existência, bem como de todo o seu material e a sua organização, constituem ferramentas importantes para o sucesso da abordagem de um doente grave [DGS, 2011].



# Abstract

This project consists in the development of a mobile application to streamline the stock management of an crash cart.

The realization of this project takes into account the development of a database management system, a server, an application with a graphical interface and the integration of a QR code and data matrices reader.

This project was motivated by the long hours of work applied to managing the stock of an crash cart, which prevent a nurse from providing services in other important areas, because, until now, there is no system/application to streamline the management process. This process is extensive and time-consuming, as it is necessary to verify the validity of all clinical consumables, currently done by eye, being necessary to read the dates written in very small letters one by one.

Clinical consumables are defined as all disposable products and materials for use on a patient. Talking a little about the crash cart. It is a mobile structure that contains a set of medical devices, medicines and other materials, essential for cardio-respiratory resuscitation. Its existence, as well as all of its material and organization, constitute important tools for the successful approach of a critically ill patient [DGS, 2011].



# Agradecimentos

Gostaríamos de agradecer aos nossos orientadores, o Professor Doutor Paulo Trigo e o Professor Paulo Vieira, pelo apoio e disponibilidade durante o desenvolvimento do projeto.

Agradecer à Professora Doutora Helga Henriques e ao Professor José Faria pela escolha para realizar a proposta de projeto, e pela orientação e disponibilidade durante o desenvolvimento do mesmo.

Agradecer ao Professor José Faria e ao Hospital de Santa Marta, pela disponibilidade na realização de visitas guiadas no início do desenvolvimento do projeto.

Agradecer às nossas famílias e amigos, que durante todo o desenvolvimento do projeto, mostraram interesse e deram-nos motivação no contínuo desenvolvimento, apesar das dificuldades encontradas.





*Dedicamos o desenvolvimento deste projeto às equipas médicas.  
Esperamos que este projeto venha realmente a agilizar o trabalho. E  
esperamos que este projeto motive outros a serem desenvolvidos para  
melhorar a vida e a qualidade do trabalho realizado.*



# Índice

Resumo	i
Abstract	iii
Agradecimentos	v
Índice	ix
Lista de Tabelas	xi
Lista de Figuras	xiii
<b>1 Introdução</b>	<b>1</b>
<b>2 Trabalho Relacionado</b>	<b>5</b>
<b>3 Modelo Proposto</b>	<b>7</b>
3.1 Requisitos . . . . .	7
3.2 Caracterização geral . . . . .	7
3.2.1 Síntese de objetivos . . . . .	8
3.2.2 Clientes alvo . . . . .	8
3.2.3 Metas a alcançar . . . . .	8
3.3 Caracterização pormenorizada . . . . .	9
3.3.1 Funções do sistema . . . . .	9
3.3.2 Atributos do sistema . . . . .	14
3.3.3 Atributos e funções do sistema . . . . .	16
3.3.4 Casos de utilização . . . . .	16
3.4 Fundamentos . . . . .	19
3.5 Abordagem . . . . .	21

<b>4</b>	<b>Implementação do Modelo</b>	<b>23</b>
4.1	Engenharia de Software (Arquitetura) . . . . .	23
4.1.1	Diagramas de Classes . . . . .	24
4.1.2	Diagramas de Atividades . . . . .	33
4.2	Tecnologias . . . . .	39
4.2.1	Ambiente de desenvolvimento . . . . .	39
4.2.2	<i>Flutter</i> e <i>Dart</i> . . . . .	42
4.2.3	<i>Django</i> , <i>REST framework</i> e <i>Python</i> . . . . .	43
4.2.4	<i>PostgreSQL</i> . . . . .	44
4.2.5	<i>Star UML</i> . . . . .	45
4.3	Desenvolvimento da aplicação gráfica . . . . .	46
4.3.1	Interface gráfica . . . . .	46
4.3.2	Implementação do leitor de códigos . . . . .	48
4.3.3	Desenvolvimento do cliente . . . . .	48
4.4	Sistema de gestão de base de dados . . . . .	51
4.5	Desenvolvimento do Servidor . . . . .	53
4.5.1	Desenvolvimento do modelo de dados . . . . .	54
4.5.2	Desenvolvimento de <i>Serializers</i> . . . . .	57
4.5.3	Desenvolvimento de <i>APIs</i> . . . . .	58
<b>5</b>	<b>Validação e Testes</b>	<b>61</b>
5.1	Testes e resultados com a <i>REST framework</i> . . . . .	61
5.2	Testes e resultados dos registos com a aplicação gráfica . . . . .	64
<b>6</b>	<b>Conclusões e Trabalho Futuro</b>	<b>75</b>
	<b>Bibliografia</b>	<b>77</b>

# Lista de Tabelas

3.1	Funções do Sistema - Aplicação Gráfica . . . . .	10
3.2	Funções do Sistema - Leitor de Códigos . . . . .	11
3.3	Funções do Sistema - Cliente da Aplicação . . . . .	11
3.4	Funções do Sistema - Sistema de Gestão de Base de Dados . .	12
3.5	Funções do Sistema - Servidor . . . . .	13
3.6	Atributos do Sistema - Aplicação Gráfica . . . . .	14
3.7	Atributos do Sistema - Sistema de Gestão de Base de Dados .	15
3.8	Atributos do Sistema - Servidor . . . . .	15
3.9	Relações entre funções e atributos do sistema . . . . .	16



# Lista de Figuras

1.1	Carro de emergência . . . . .	1
1.2	Adaptabilidade da <i>framework Flutter</i> . . . . .	2
1.3	Leitor de códigos . . . . .	2
2.1	Máquina com sistema de auto pagamento e registo . . . . .	5
2.2	Leitor de códigos . . . . .	6
3.1	Casos de utilização de um Enfermeiro(a) . . . . .	17
3.2	Casos de utilização de um Informático(a) - Exemplo para uma entidade . . . . .	18
3.3	Sensor de pressão . . . . .	20
3.4	Sensor <i>RFID</i> . . . . .	20
3.5	Leitor de códigos . . . . .	20
3.6	Cronograma do desenvolvimento do projeto . . . . .	22
4.1	Arquitetura Cliente-Servidor . . . . .	23
4.2	<i>Backend</i> - Diagrama de classes <i>models</i> . . . . .	24
4.3	<i>Backend</i> - Diagrama de classes <i>serializers</i> . . . . .	25
4.4	<i>Backend</i> - Diagrama de classes <i>views</i> . . . . .	26
4.5	<i>Frontend</i> - Diagrama de classes <i>lib</i> . . . . .	27
4.6	<i>Frontend</i> - Diagrama de classes <i>registration</i> . . . . .	28
4.7	<i>Frontend</i> - Diagrama de classes <i>grids</i> . . . . .	29
4.8	<i>Frontend</i> - Diagrama de classes <i>updates</i> . . . . .	30
4.9	<i>Frontend</i> - Diagrama de classes <i>qr_code_reader</i> . . . . .	31
4.10	<i>Frontend</i> - Diagrama de classes <i>request_handler</i> . . . . .	32
4.11	<i>Frontend</i> - Diagrama de atividades registo parte um . . . . .	33
4.12	<i>Frontend</i> - Diagrama de atividades registo parte dois . . . . .	34
4.13	<i>Frontend</i> - Diagrama de atividades registo parte três . . . . .	35

4.14	<i>Frontend</i> - Diagrama de atividades leitor de códigos . . . . .	36
4.15	<i>Frontend</i> - Diagrama de atividades cliente-servidor . . . . .	37
4.16	<i>Frontend</i> - Diagrama de atividades <i>REST framework</i> -servidor . . . . .	38
4.17	Dependências da <i>framework Flutter</i> . . . . .	39
4.18	Instalação do <i>Django</i> e criação do ambiente virtual . . . . .	40
4.19	Ambiente de trabalho - Servidor . . . . .	41
4.20	Ambiente de trabalho - Aplicação gráfica . . . . .	41
4.21	Sistema de pastas da <i>framework Flutter</i> . . . . .	42
4.22	Sistema de pastas da <i>framework Django</i> . . . . .	43
4.23	<i>createInstitution()</i> - Exemplo de uma função assíncrona . . . . .	50
4.24	Modelo entidade associação . . . . .	51
4.25	Modelo <i>CrashCart</i> reestruturado . . . . .	54
4.26	Modelo <i>CrashCart</i> controlo do <i>id_c</i> . . . . .	55
4.27	Comandos usados para migrar o modelo de dados . . . . .	56
4.28	Exemplo do uso <i>unique_together</i> . . . . .	57
4.29	Exemplo de um <i>Serializer</i> . . . . .	57
4.30	Exemplo de um <i>POST</i> . . . . .	59
4.31	Exemplo de um <i>PUT</i> . . . . .	59
4.32	Exemplo de um <i>GET</i> . . . . .	60
4.33	Exemplo de um <i>DELETE</i> . . . . .	60
5.1	Exemplo do método <i>POST</i> . . . . .	62
5.2	Exemplo do método <i>GET</i> . . . . .	62
5.3	Exemplo do método <i>PUT</i> . . . . .	63
5.4	Exemplo do método <i>DELETE</i> . . . . .	63
5.5	Primeira interface . . . . .	64
5.6	Segunda interface . . . . .	65
5.7	Terceira interface . . . . .	65
5.8	Quarta interface . . . . .	65
5.9	Quinta interface . . . . .	66
5.10	Sexta interface . . . . .	66
5.11	Sétima interface . . . . .	66
5.12	Oitava interface . . . . .	67
5.13	Sétima interface preenchida . . . . .	67
5.14	Leitura e validação de uma data inferior a um mês . . . . .	68
5.15	Leitura e validação de uma data superior a um mês . . . . .	69



5.16 Relatório dos <i>slots</i> e do número de leituras . . . . .	70
5.17 Leitura de um código QR de uma gaveta . . . . .	71
5.18 Leitura de um código QR de um carro . . . . .	72
5.19 Nona interface . . . . .	73



# Capítulo 1

## Introdução

O objetivo deste projeto é o desenvolvimento de uma aplicação, para uso principalmente móvel, que permite agilizar o processo de gestão de *stock* de carros de emergência, cf. Figura 1.1. As componentes deste projeto podem estar divididas em quatro partes:

- Desenvolvimento de uma aplicação gráfica
- Integração de um leitor de códigos QR e de matrizes de dados
- Desenvolvimento de um sistema de gestão de base de dados
- Desenvolvimento de um servidor



Figura 1.1: Carro de emergência

No que diz respeito à aplicação gráfica, esta foi desenvolvida usando a *framework Flutter*, desenvolvida pela *Google*. Permite a programação da aplicação apenas uma vez na linguagem *Dart* podendo ser exportada para vários sistemas operativos e para a *Web*, cf. Figura 1.2. Assim a aplicação gráfica sendo para uso principalmente móvel, a *framework Flutter* automaticamente converte o código escrito em *Dart* para *Kotlin* no caso do sistema operativo *Android*, e para *Object C* ou *Swift* no caso do sistema operativo *IOS*. Como permitimos o registo dos conteúdos através da aplicação gráfica, a capacidade de uso da mesma no sistema *Windows* ou na *Web*, por exemplo, facilita este registo.



Figura 1.2: Adaptabilidade da *framework Flutter*

A integração de um leitor de códigos QR e de matrizes de dados, cf. Figura 1.3, tem como objetivo facilitar a leitura dos consumíveis clínicos que as equipas médicas usam no tratamento de um doente grave. Assim, os dados dos vários carros de emergência podem ser acedidos com vários códigos, através de uma simples leitura. O uso do leitor também permite que as equipas médicas mantenham a atenção no tratamento do doente grave podendo realizar uma leitura rápida.



Figura 1.3: Leitor de códigos

O sistema de gestão de base de dados foi desenhado, de modo a ser facilmente expandido e adaptado, para uma grande escala de instituições onde estes carros de emergência são usados, e a sua gestão ainda não está a ser efetuada através de um sistema informático. O sistema de gestão de base de dados usado é o *PostgreSQL*, porque é um sistema relacional compatível com as *frameworks* usadas para desenvolver a aplicação e o servidor, *Flutter* e *Django* respetivamente.

Finalmente o desenvolvimento do servidor foi feito através da *framework Django*, programado em *Python* para ser acessível através da *Web* com *Uniform Resource Locators - urls* definidos para várias operações. A programação é também feita dentro de um ambiente virtual, o que permite no futuro a fácil implementação do servidor numa máquina física ou na *cloud* para ser acedido pela aplicação gráfica por qualquer instituição que a use.



## Capítulo 2

### Trabalho Relacionado

Na procura de uma solução que fosse de fácil uso e eficaz no registo dos conteúdos dos carros de emergência, que não incomodasse as equipas médicas no tratamento de um doente grave e que permitisse agilizar o processo de gestão dos carros de emergência pelas instituições, baseamo-nos no sistema de auto pagamento e registo de compras, cf. Figura 2.1.



Figura 2.1: Máquina com sistema de auto pagamento e registo

Este sistema é um recurso de algumas máquinas de registo de compras em hipermercados, que permite ao consumidor autonomamente o registo das suas compras e o respetivo pagamento. Este sistema é rápido, sendo apenas necessária a leitura dos códigos dos produtos que o consumidor está a comprar. Com base neste sistema podemos utilizar o leitor que é portátil e de fácil registo através de um botão contido no mesmo, cf. Figura 2.2.



Figura 2.2: Leitor de códigos

Numa das visitas ao Hospital de Santa Marta, observámos também o uso de códigos QR para a gestão de limpeza e desinfecção dos carros de emergência. Nesta base destaca-se a necessidade de plastificar os códigos QR, porque o álcool usado na limpeza e desinfecção apaga os códigos, o que os impede de serem lidos.

A consideração de *frameworks* e programas para o desenvolvimento do projeto, foi também inspirada pelas escolhas efetuadas por colegas dos outros projetos propostos pela *ESEL*. Isto porque na existência de duvidas, existe a oportunidade de discussão e ajuda para contribuir para melhores projetos.

Para realizar a grelha e projetar na aplicação uma com o mesmo formato da grelha da gaveta correspondente, inspirá-mo-nos na capacidade do *Excel* para juntar e disjuntar parcelas.



# Capítulo 3

## Modelo Proposto

No presente capítulo será abordada a análise de requisitos. Esta é responsável por coletar dados indispensáveis e necessários para solucionar um problema e alcançar os objetivos, sendo importante na gestão de projetos.

### 3.1 Requisitos

Esta análise é vital para o desenvolvimento do sistema, pois vai determinar o sucesso ou fracasso do projeto. Tem como objetivo tornar o projeto mais coeso e bem estruturado. Serão abordados nesta fase:

- Caracterização geral - síntese, clientes e metas
- Caracterização pormenorizada - funções e atributos do sistema, e casos de utilização
- Fundamentos
- Abordagem

### 3.2 Caracterização geral

Apresenta-se a seguir uma síntese de objetivos correspondentes aos pretendidos com o desenvolvimento do projeto, os clientes para quem desenvolvemos o projeto, e as metas a alcançar para o sucesso na realização e desenvolvimento do projeto.

### 3.2.1 Síntese de objetivos

O projeto consiste no desenvolvimento de um sistema que engloba principalmente quatro partes, previamente referidas na introdução:

- Desenvolvimento de uma aplicação gráfica
- Integração de um leitor de códigos QR e de matrizes de dados
- Desenvolvimento de um sistema de gestão de base de dados
- Desenvolvimento de um servidor

O desenvolvimento da aplicação gráfica permite ao utilizador a interação com o sistema, através de uma interface gráfica, desenhada para fácil compreensão e uso. A integração de um leitor de códigos QR e matrizes de dados permite o rápido registo das leituras, e simples adaptabilidade ao trabalho realizado pelas equipas médicas. O desenvolvimento de um sistema de gestão de base de dados relacional permite uma coesa estrutura dos dados e coesão nas relações entre os mesmos. O desenvolvimento do servidor permite a alocação dos dados num sistema independente que pode ser acessado pela aplicação gráfica, por qualquer instituição registada.

### 3.2.2 Clientes alvo

Os clientes principais são os orientadores da *ESEL*, pois foram os que realizaram a proposta. No entanto o desenvolvimento do projeto tem como publico alvo os alunos da *ESEL* para treino, bem como, no futuro, as equipas médicas das instituições que usam os carros de emergência.

### 3.2.3 Metas a alcançar

As metas estabelecidas são:

- Desenvolvimento de uma aplicação gráfica
- Desenvolvimento de uma interface gráfica de fácil interpretação e leitura
- Teste da aplicação gráfica no sistema *Android*
- Integração de um leitor de códigos QR e matrizes de dados

- Simulação do leitor de códigos, através da câmara do dispositivo móvel
- Desenvolvimento de um cliente para comunicação com o servidor
- Desenvolvimento de um sistema de gestão de base de dados relacional
- Testes ao registo e leitura dos dados guardados no sistema de gestão de base de dados
- Desenvolvimento de um servidor como prova de conceito da proteção dos dados e independência do sistema
- Teste de registo e leitura dos dados através de pedidos ao servidor, por meio do cliente e através da *Web*

### 3.3 Caracterização pormenorizada

Nesta fase de criação de um modelo, são estabelecidas funções do sistema, que representam o que o sistema é suposto fazer, bem como os seus atributos, que representam qualidades não-funcionais. São também definidos os casos de utilização, ou seja, momentos de uso do sistema por parte de um utilizador para realizar uma certa função do sistema.

#### 3.3.1 Funções do sistema

Para o desenvolvimento objetivo e prioritário de partes do projeto, dividiram-se as funções do sistema nas seguintes categorias:

- Funções do sistema da aplicação gráfica - Tabela 3.1
- Funções do sistema do leitor de códigos - Tabela 3.2
- Funções do sistema do cliente da aplicação gráfica - Tabela 3.3
- Funções do sistema do sistema de gestão de base de dados - Tabela 3.4
- Funções do sistema do servidor - Tabela 3.5

Funções do sistema da aplicação gráfica		
Requisito	Função	Categoria
Requisito 1.1	Mostrar o menu inicial	Evidente
Requisito 1.2	Mostrar o menu do Leitor de Códigos	Evidente
Requisito 1.3	Mostrar o menu de Registo da Instituição e do número de Carros de Emergência	Evidente
Requisito 1.4	Mostrar a grelha de Carros que pertencem a uma Instituição	Evidente
Requisito 1.5	Mostrar o menu de Registo do número de Gavetas de um Carro de Emergência	Evidente
Requisito 1.6	Mostrar a grelha de Gavetas	Evidente
Requisito 1.7	Mostrar o menu do formato da Gaveta	Evidente
Requisito 1.8	Mostrar a grelha de <i>Slots</i> de uma Gaveta	Evidente
Requisito 1.9	Mostrar o menu de Registo de um Consumível Clínico	Evidente
Requisito 1.10	Executar a transição entre menus	Invisível
Requisito 1.11	Permitir a junção de <i>Slots</i>	Evidente
Requisito 1.12	Permitir a disjunção de <i>Slots</i>	Evidente

Tabela 3.1: Funções do Sistema - Aplicação Gráfica

Funções do sistema do leitor de códigos		
Requisito	Função	Categoria
Requisito 2.1	Efetuar a leitura de códigos QR	Evidente
Requisito 2.2	Efetuar a leitura de matrizes de dados	Evidente
Requisito 2.3	Efetuar a leitura de códigos de barra	Adorno
Requisito 2.4	Efetuar a leitura de um código ao pressionar um botão	Evidente
Requisito 2.5	Comparar datas de validade	Evidente
Requisito 2.6	Contar o número de leituras de um consumível clínico	Invisível
Requisito 2.7	Mostrar uma lista dos consumíveis e quantias a repor	Evidente

Tabela 3.2: Funções do Sistema - Leitor de Códigos

Funções do sistema do cliente da Aplicação		
Requisito	Função	Categoria
Requisito 3.1	Comunicar com o Servidor para Registo de Dados	Invisível
Requisito 3.2	Comunicar com o Servidor para Atualizar os Dados	Invisível
Requisito 3.3	Comunicar com o Servidor para Resgatar os Dados	Invisível
Requisito 3.4	Comunicar com o Servidor para Eliminar os Dados	Adorno
Requisito 3.5	Estabelecer uma Comunicação Persistente	Invisível

Tabela 3.3: Funções do Sistema - Cliente da Aplicação

Funções do sistema do Sistema de Gestão de Base de Dados		
Requisito	Função	Categoria
Requisito 4.1	Auto-geração de <i>IDs</i> únicos	Invisível
Requisito 4.2	Guardar dados de entidades do tipo <i>Institution</i>	Invisível
Requisito 4.3	Guardar dados de entidades do tipo <i>CrashCart</i>	Invisível
Requisito 4.4	Guardar dados de entidades do tipo <i>Drawer</i>	Invisível
Requisito 4.5	Guardar dados de entidades do tipo <i>Slot</i>	Invisível
Requisito 4.6	Guardar dados de atributos de entidades do tipo <i>Institution</i>	Invisível
Requisito 4.7	Guardar dados de atributos de entidades do tipo <i>CrashCart</i>	Invisível
Requisito 4.8	Guardar dados de atributos de entidades do tipo <i>Drawer</i>	Invisível
Requisito 4.9	Guardar dados de atributos de entidades do tipo <i>Slot</i>	Invisível
Requisito 4.10	Estabelecimento de relações entre entidades	Invisível
Requisito 4.11	Eliminação sequencial das entidades e seus respectivos atributos	Invisível
Requisito 4.12	Guardar imagens	Adorno
Requisito 4.13	Exibição dos dados guardados através de <i>queries</i>	Visível

Tabela 3.4: Funções do Sistema - Sistema de Gestão de Base de Dados

Funções do sistema do Servidor		
Requisito	Função	Categoria
Requisito 5.1	Criação do modelo de dados	Evidente
Requisito 5.2	Interfaces de programação para registo de entidades	Evidente
Requisito 5.3	Interfaces de programação para registo de atributos de entidades	Evidente
Requisito 5.4	Interfaces de programação para atualização de atributos de entidades	Evidente
Requisito 5.5	Interfaces de programação para eliminação de entidades	Evidente
Requisito 5.6	Interfaces de programação para obtenção de dados	Evidente
Requisito 5.7	Auto-geração de <i>IDs</i>	Invisível
Requisito 5.8	Auto-geração de <i>strings</i> através dos <i>IDs</i>	Invisível
Requisito 5.9	Auto-geração de nomes para algumas entidades	Invisível
Requisito 5.10	Atualização da base de dados	Invisível
Requisito 5.11	Envio de dados ao cliente	Invisível
Requisito 5.12	Envio de mensagens de sucesso e de erros	Evidente

Tabela 3.5: Funções do Sistema - Servidor

### 3.3.2 Atributos do sistema

A seguir apresentam-se os atributos do sistema, divididos nas seguintes categorias:

- Atributos do sistema da aplicação gráfica - Tabela 3.6
- Atributos do sistema do sistema de gestão de base de dados - Tabela 3.7
- Atributos do sistema do servidor - Tabela 3.8

Atributos do sistema da aplicação gráfica		
Atributo	Detalhe/Restrição de Fronteira	Categoria
Plataforma	<i>Android</i> Outros sistema operativos e <i>Web</i>	Obrigatória Desejável
Interação Homem-Máquina	Controlos <i>Touch</i> Uso do rato e do teclado	Obrigatória Desejável
Acessibilidade	Texto com fonte e letra grande, e legível	Desejável
Facilidade de uso	Navegação entre menus simples	Desejável
Desempenho	Leitura dos códigos rápida Comunicação com o servidor rápida Resposta rápida a ações do utilizador	Obrigatória Obrigatória Desejável

Tabela 3.6: Atributos do Sistema - Aplicação Gráfica



Atributos do sistema do sistema de gestão de base de dados		
Atributo	Detalhe/Restrição de Fronteira	Categoria
Plataforma	<i>Microsoft</i> Outros sistemas operativos	Obrigatória Desejável
Acessibilidade	Texto com fonte e letra grande, e legível Exibição de dados através de tabelas legíveis	Desejável Desejável
Facilidade de uso	Navegação através de comandos	Obrigatória
Desempenho	Comunicação rápida	Obrigatória

Tabela 3.7: Atributos do Sistema - Sistema de Gestão de Base de Dados

Atributos do sistema do Servidor		
Atributo	Detalhe/Restrição de Fronteira	Categoria
Plataforma	<i>Microsoft</i> Outros sistemas operativos	Obrigatória Desejável
Interação Homem-Máquina	Uso da consola de comandos Desenvolvimento de <i>APIs</i>	Obrigatória Desejável
Acessibilidade	Texto com fonte e letra grande, e legível	Desejável
Facilidade de uso	Navegação através de <i>urls</i> Navegação entre menus simples	Obrigatória Desejável
Desempenho	Comunicação rápida	Obrigatória

Tabela 3.8: Atributos do Sistema - Servidor

### 3.3.3 Atributos e funções do sistema

É importante evidenciar todas as relações entre funções e atributos do sistema, descrever atributos relacionados com funções específicas [5]. Estes encontram-se na tabela 3.9:

Atributos do Sistema		Funções do Sistema
Plataforma		R1.1, R1.2, R1.3, R1.4, R1.5, R1.6, R1.7, R1.8, R1.9, R1.11, R1.12, R3.1, R3.2, R3.3, R3.4, R3.5, R4.1, R4.10, R4.11, R4.12, R4.13, R5.2, R5.3, R5.4, R5.5, R5.6
Interação	Homem-Máquina	R1.1, R1.2, R1.3, R1.4, R1.5, R1.6, R1.7, R1.8, R1.9, R2.1, R2.2, R2.3, R2.4, R2.5, R2.7, R5.2, R5.3, R5.4, R5.5, R5.6, R5.11, R5.12
Acessibilidade		R1.10, R2.1, R2.2, R2.3, R2.4, R2.5, R2.7, R4.13
Facilidade de uso		R1.1, R1.2, R1.3, R1.4, R1.5, R1.6, R1.7, R1.8, R1.9, R1.11, R1.12, R2.1, R2.2, R2.3, R2.4, R2.5, R2.6, R2.7, R4.13, R5.2, R5.3, R5.4, R5.5, R5.6, R5.11, R5.12
Desempenho		R1.10, R1.11, R1.12, R2.1, R2.2, R2.3, R2.4, R2.5, R2.6, R2.7, R3.1, R3.2, R3.3, R3.4, R3.5, R4.2, R4.3, R4.4, R4.5, R4.6, R4.7, R4.8, R4.9, R5.11, R5.12

Tabela 3.9: Relações entre funções e atributos do sistema

### 3.3.4 Casos de utilização

Os casos de utilização representam a interação entre o utilizador e o sistema, descrevendo a sequência de eventos que são realizados para completar uma ação [5].

Existem duas formas de identificar os casos de utilização. Uma forma foca-se nos atores, a outra nos eventos. Para o projeto, foi utilizada a forma focada nos atores, pois estes são o foco do projeto. Assim sendo, é necessário identificar [5]:

- Os atores relacionados com o sistema
- Por ator, identificar os processos que ele inicia ou participa

Seguidamente são apresentados os diagramas dos casos de utilização do projeto, cf. Figura 3.1 e Figura 3.2.

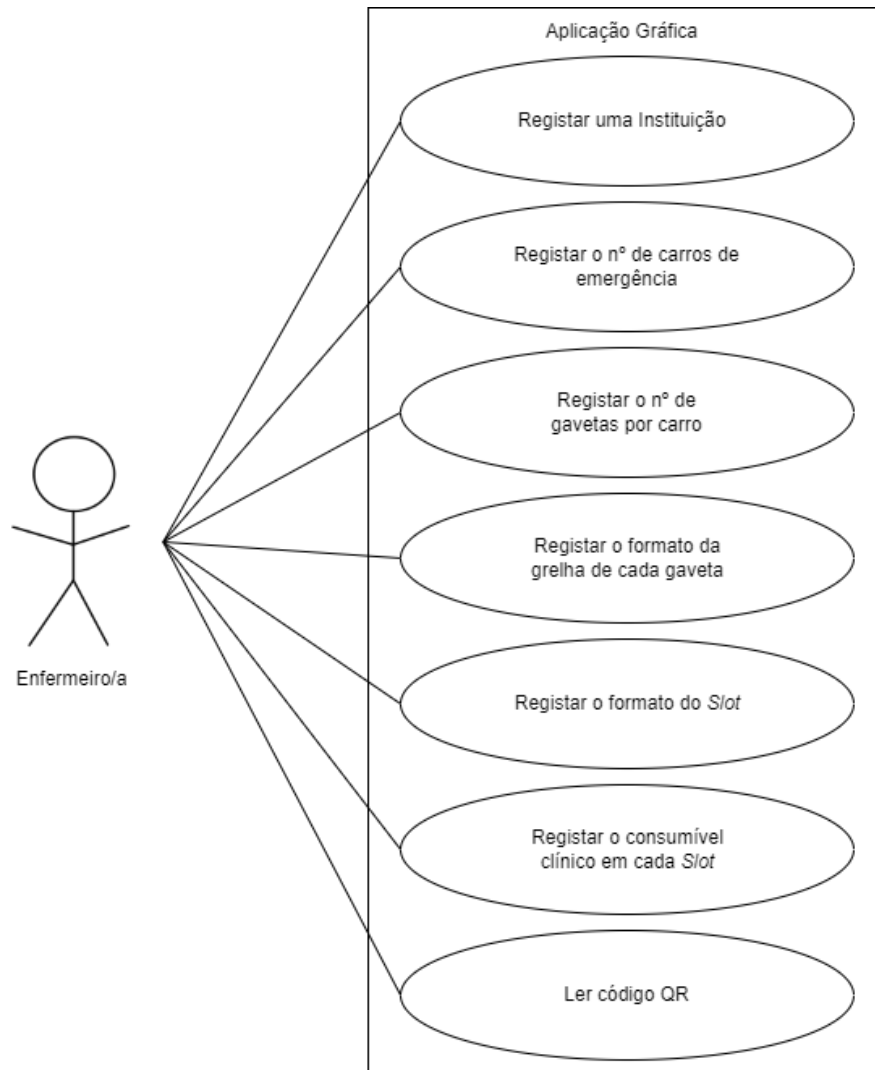


Figura 3.1: Casos de utilização de um Enfermeiro(a)

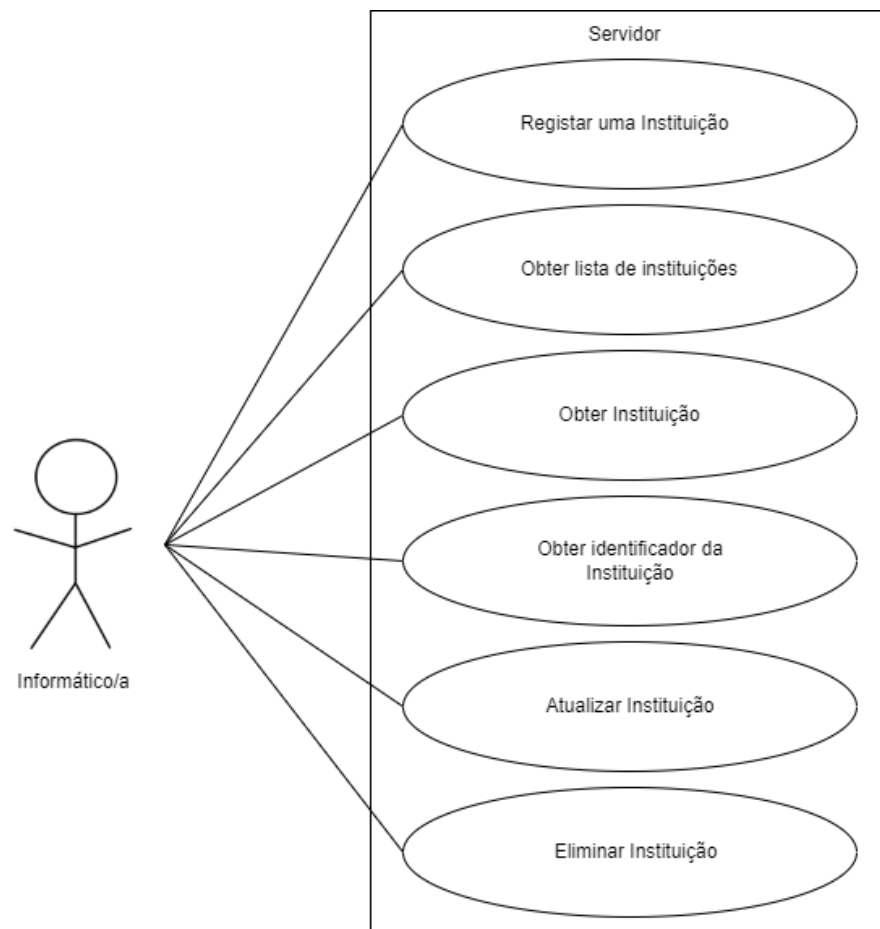


Figura 3.2: Casos de utilização de um Informático(a) - Exemplo para uma entidade

## 3.4 Fundamentos

Nesta secção serão abordados os passos que tornaram este projeto possível. Numa fase inicial foi feita a análise de requisitos, através de reuniões com os clientes e visitas guiadas ao hospital de Santa Marta para ver os carros de emergência num ambiente de uso. Nesta base foi possível determinar dois momentos de gestão dos carros de emergência, observar que existem vários carros, com diferentes tamanhos, diferente número de gavetas e estas em si podem também ter diferentes tamanhos, e várias organizações e composições dos consumíveis clínicos para vários tipos de emergências. Concluiu-se com base nos carros que o sistema deverá possibilitar o registo de carros de emergência com um variado número de gavetas e estas deverão possibilitar uma organização variada. Relativamente aos momentos de gestão tem-se:

- Gestão diária - ocorre durante ou após o uso do carro de emergência para atender a um doente grave
- Gestão mensal - ocorre com a necessidade de verificar as datas de validade dos consumíveis clínicos

Para a gestão mensal, o sistema realiza a leitura de códigos QR. Nesta base foi feito um estudo da composição da informação dos códigos QR. Este estudo foi feito para a união europeia principalmente. No entanto concluiu-se que existem muitas normas de identificação dos produtos farmacêuticos, e seria necessário mais tempo para um maior estudo de todos os tipos de normas, bem como a realização de uma biblioteca dedicada à análise das informações dos códigos para obtenção das datas de validade. O que apresentamos é assim uma prova de conceito do funcionamento para algumas normas, nomeadamente as normas *GS1*, *GTIN-13* e *GTIN-14*.

O passo seguinte foi uma pesquisa dos vários sensores para registo dos consumíveis clínicos e das linguagens de programação para o desenvolvimento do sistema, bem como uma análise do formato do sistema. Relativamente aos sensores, foram considerados:

- Sensores de pressão. cf. Figura 3.3, no entanto as ampolas especificamente têm vários tamanhos e formatos, bem como podem ser de vidro ou plástico o que influencia o peso e obriga a um registo prévio de todos os tipos, bem como equacionar os vários pesos

- Sensores *RFID*, cf. Figura 3.4, são sensores de frequência para ondas rádio, pequenos, mas não o suficiente para certas ampolas, e existiria uma necessidade de adicionar estes a todos os consumíveis clínicos
- Leitor de códigos, cf. Figura 3.5, como cerca de oitenta por cento dos consumíveis clínicos estão identificados por um código de barras, código QR ou uma matriz de dados, podemos utilizar este sensor para identificar e gerir os carros de emergência

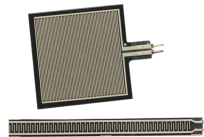


Figura 3.3: Sensor de pressão

Figura 3.4: Sensor *RFID*

Figura 3.5: Leitor de códigos

Para as linguagens de programação e *frameworks* usadas, o *Flutter* foi escolhido de imediato, a conselho de um dos orientadores. Comparado a outras *frameworks*, foi escolhido por ser híbrido, fácil de aprender e programar boas interfaces gráficas. Existiam alternativas como a *framework React Native* e a *framework Ionic* baseadas na linguagem *JavaScript*, no entanto, com base na experiência ainda temos muitas dificuldades na linguagem. A *framework* híbrida é importante porque permite-nos programar uma vez para vários sistemas operativos, nomeadamente *Android* e *IOS*, visto que a maioria dos dispositivos móveis, nomeadamente *tablets* e *smartphones* usam um destes sistemas operativos. Não consideramos *frameworks* não híbridas.

No caso do servidor, após várias tentativas com outras *frameworks* e sem sucesso, optou-se pelo *Django*. Esta *framework* permite o desenvolvimento em *Python* de *Application Programming Interfaces* - *APIs* para a *Web*. A

comunicação é feita através de *urls*, sendo apenas necessário definir os *hosts* e o *port* de comunicação. A programação em *Python* é familiar, sendo lecionada no curso e usada em várias unidades curriculares. Esta *framework* também tem um *Object-Relational Mapping* - *ORM*, o que permite a comunicação com o sistema de gestão de base de dados sem as *queries SQL*, sendo esta feita através de classes e métodos em *Python*.

Finalmente, a escolha do sistema de gestão de base de dados, o *PostgreSQL* deve-se há semelhança com o *MySQL*, este último lecionado no curso não foi escolhido, porque encontra-se bloqueado, e após várias tentativas de desbloquear o *MySQL*, optou-se pelo *PostgreSQL*. Os fatores decisivos para a escolha são a compatibilidade com o servidor e a aplicação gráfica, e o ser relacional, o que ambos os sistemas de gestão de base de dados respeitam.

## 3.5 Abordagem

Para o sucesso no desenvolvimento do projeto, este foi desenvolvido completando objetivos semanais. Certos objetivos demoraram mais de uma semana, como por exemplo, estabelecer conexão ao servidor, outros menos de uma semana, como por exemplo, o desenvolvimento da grande maioria das interfaces gráficas da aplicação gráfica. Na figura 3.6 apresenta-se um cronograma dos acontecimentos para o desenvolvimento do projeto.

Título do projeto: I-Crash						
Fases	Março	Abril	Maio	Junho	Julho	Agosto
Análise de requisitos						
Casos de utilização						
Arquitetura						
Implementação						
Modelo de Dados						
Sistema de gestão de base de dados						
Servidor						
Aplicação gráfica						
Testes						
Relatório final						

Figura 3.6: Cronograma do desenvolvimento do projeto



# Capítulo 4

## Implementação do Modelo

Neste capítulo apresenta-se a arquitetura do sistema e os modelos que permitiram o seu desenvolvimento. Apresenta-se também as tecnologias usadas e as relações entre as mesmas, bem como o desenvolvimento do sistema em si até à sua conclusão.

### 4.1 Engenharia de Software (Arquitetura)

O sistema foi desenhado com base numa arquitetura Cliente-Servidor. Dividiu-se o sistema em duas partes, uma *backend* que consiste no servidor *Django* e o sistema de gestão de base de dados *PostgreSQL*, e uma *frontend* que consiste na aplicação gráfica desenvolvida na *framework Flutter*, que contém em si o leitor de códigos e o cliente para a comunicação. A comunicação é feita através de *urls* para *APIs* desenvolvidas no servidor, onde este realiza as *queries* para escrever e ler da base de dados, cf. Figura 4.1.



Figura 4.1: Arquitetura Cliente-Servidor

### 4.1.1 Diagramas de Classes

Estes diagramas permitem a visualização da estrutura do sistema, dos seus modelos e classes, identificando relações entre classes e partes do sistema. Permitem documentação, análise e desenho para uma modelação orientada a objetos, geração e reestruturação de código.

A seguir apresentam-se alguns diagramas, primeiro da parte *backend* depois da parte *frontend*.

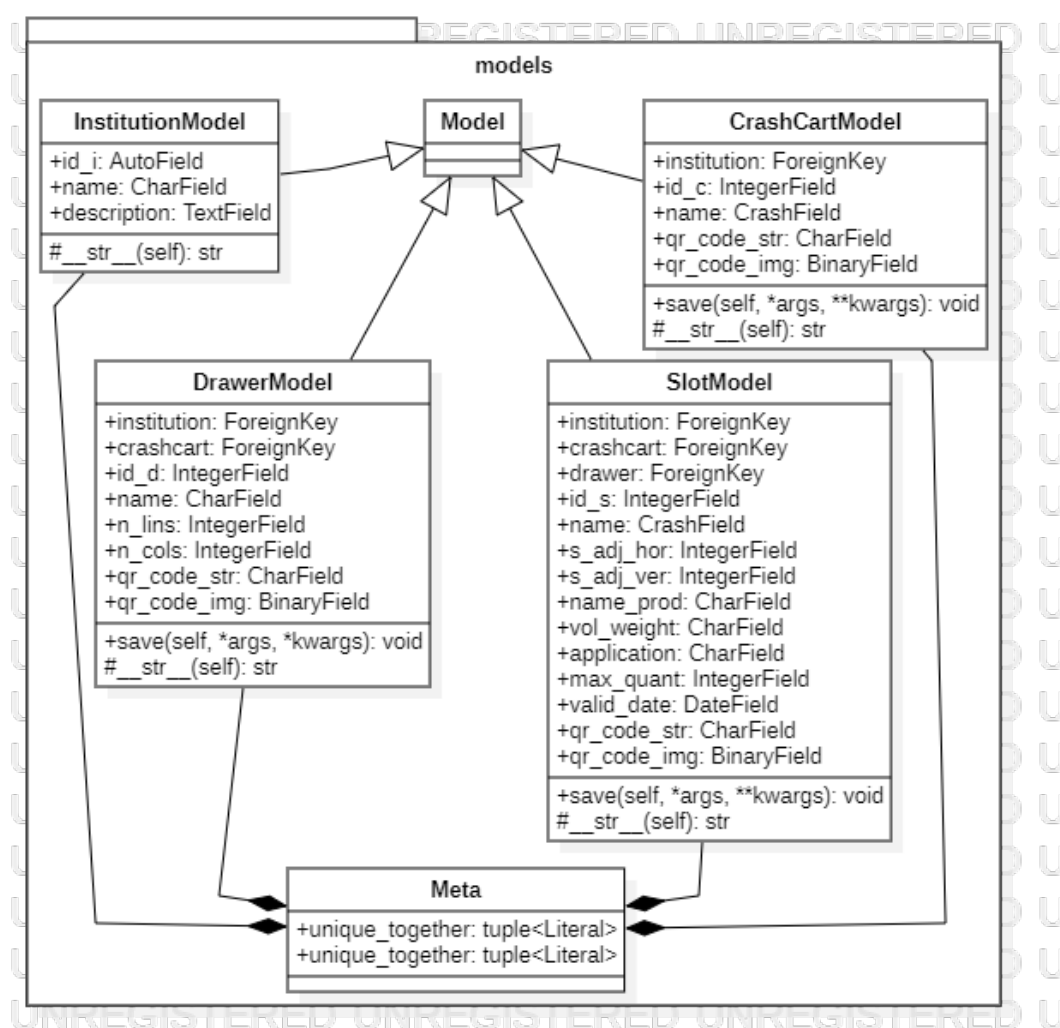
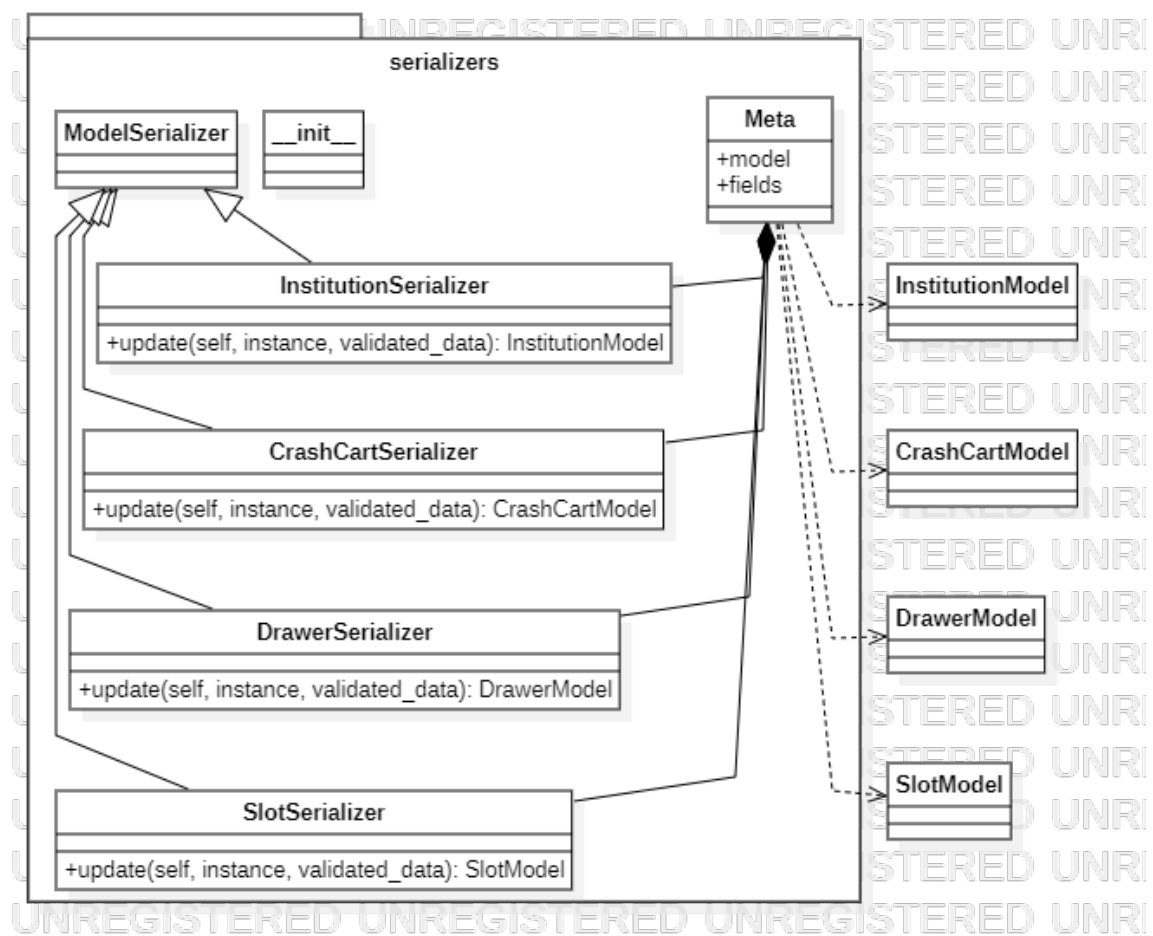
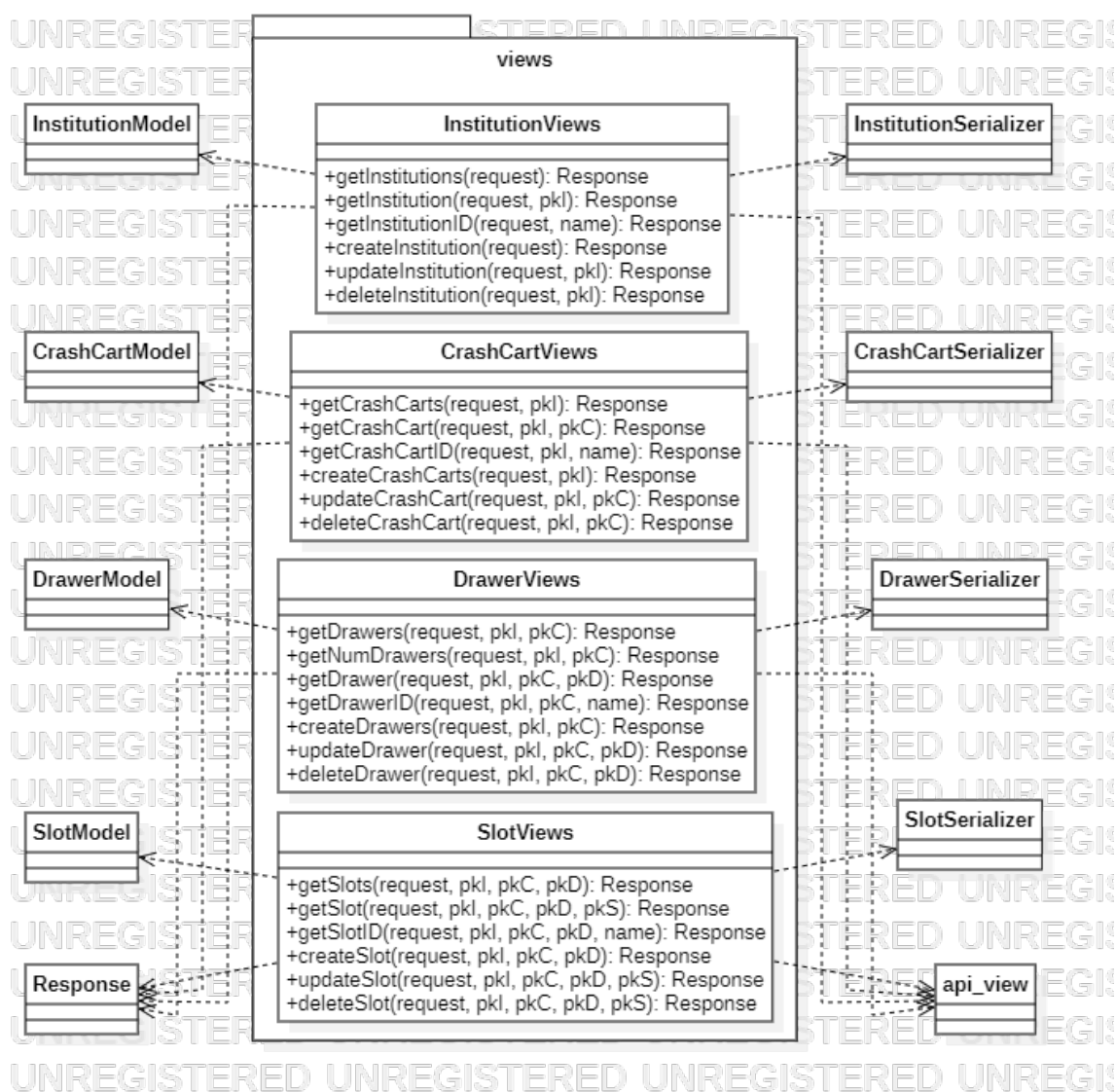
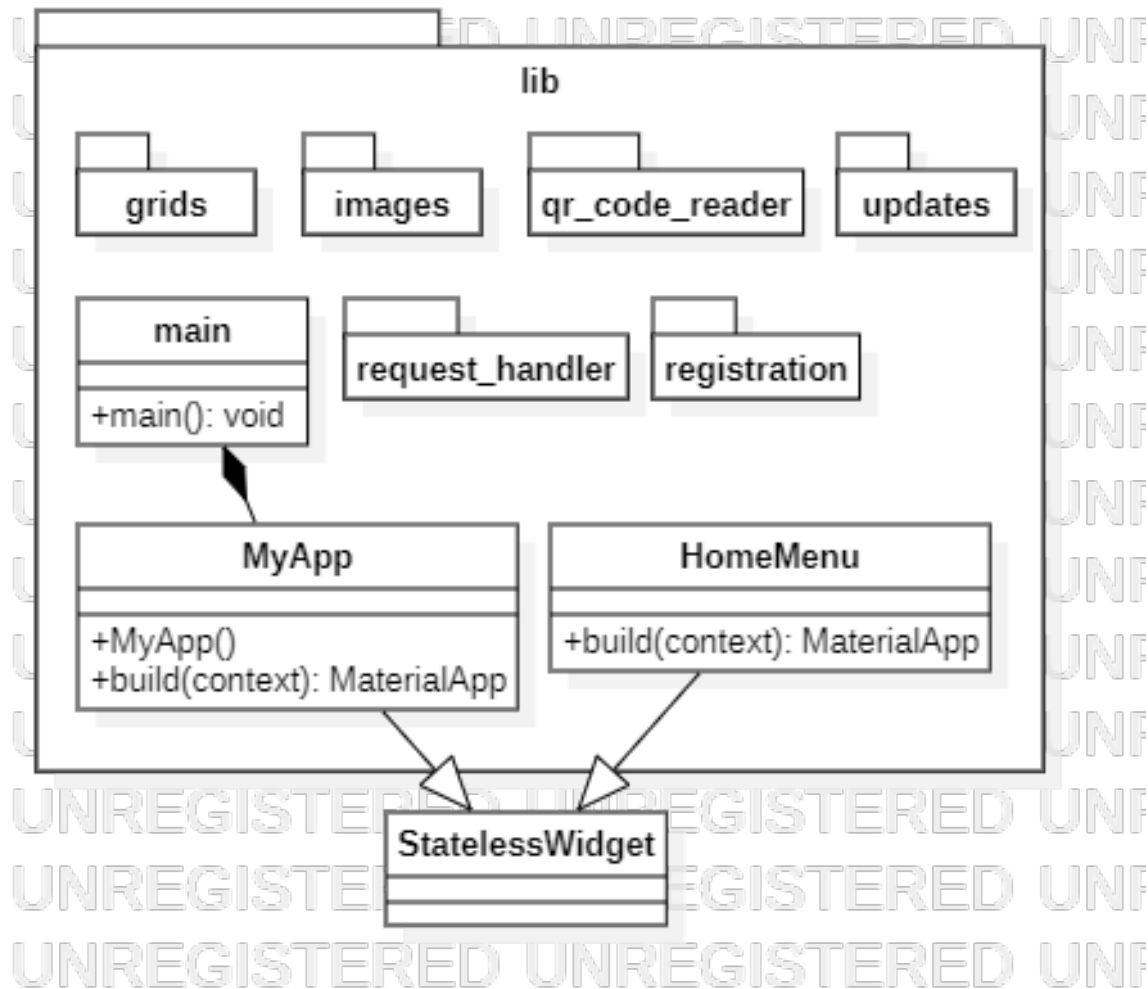
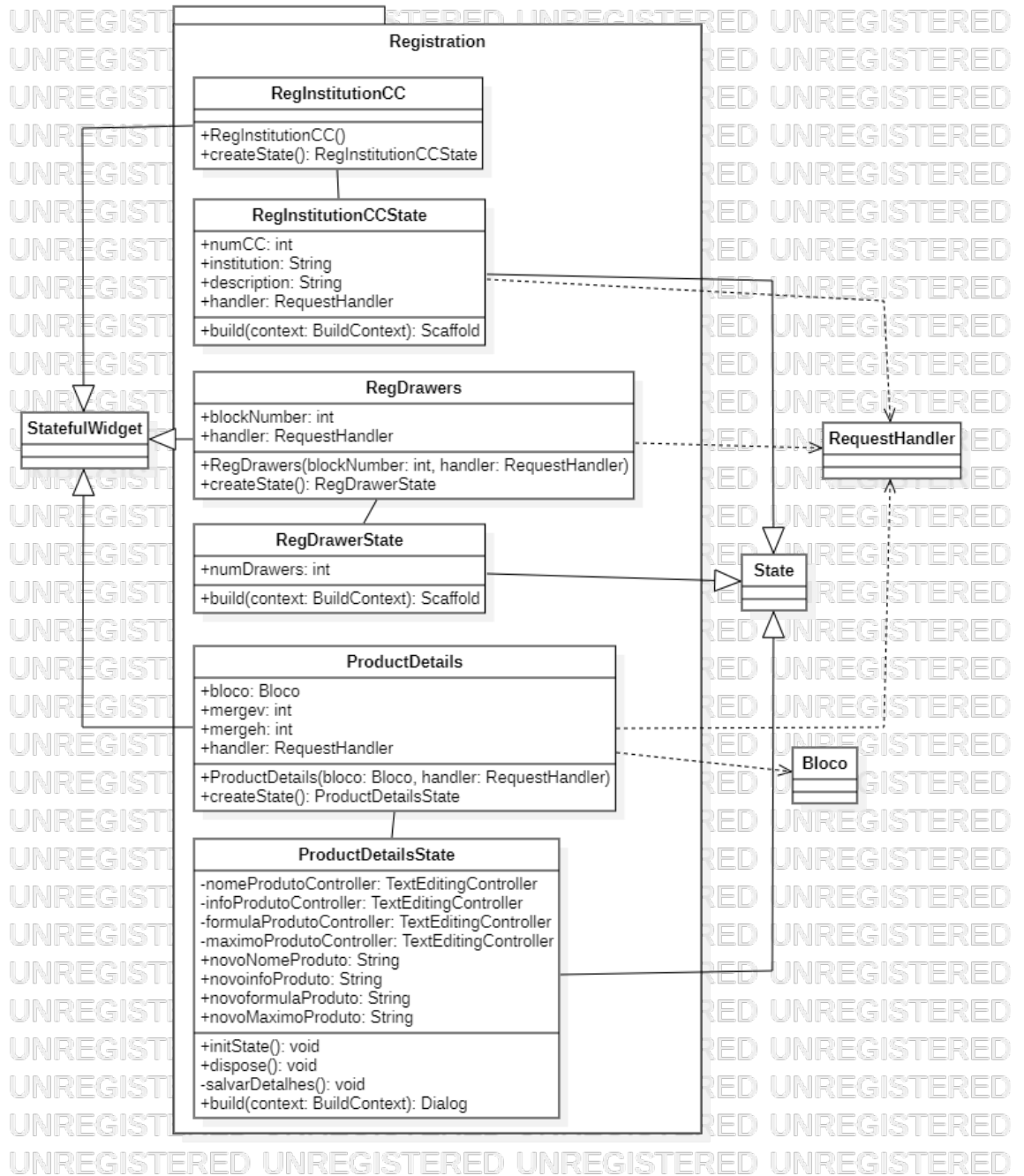


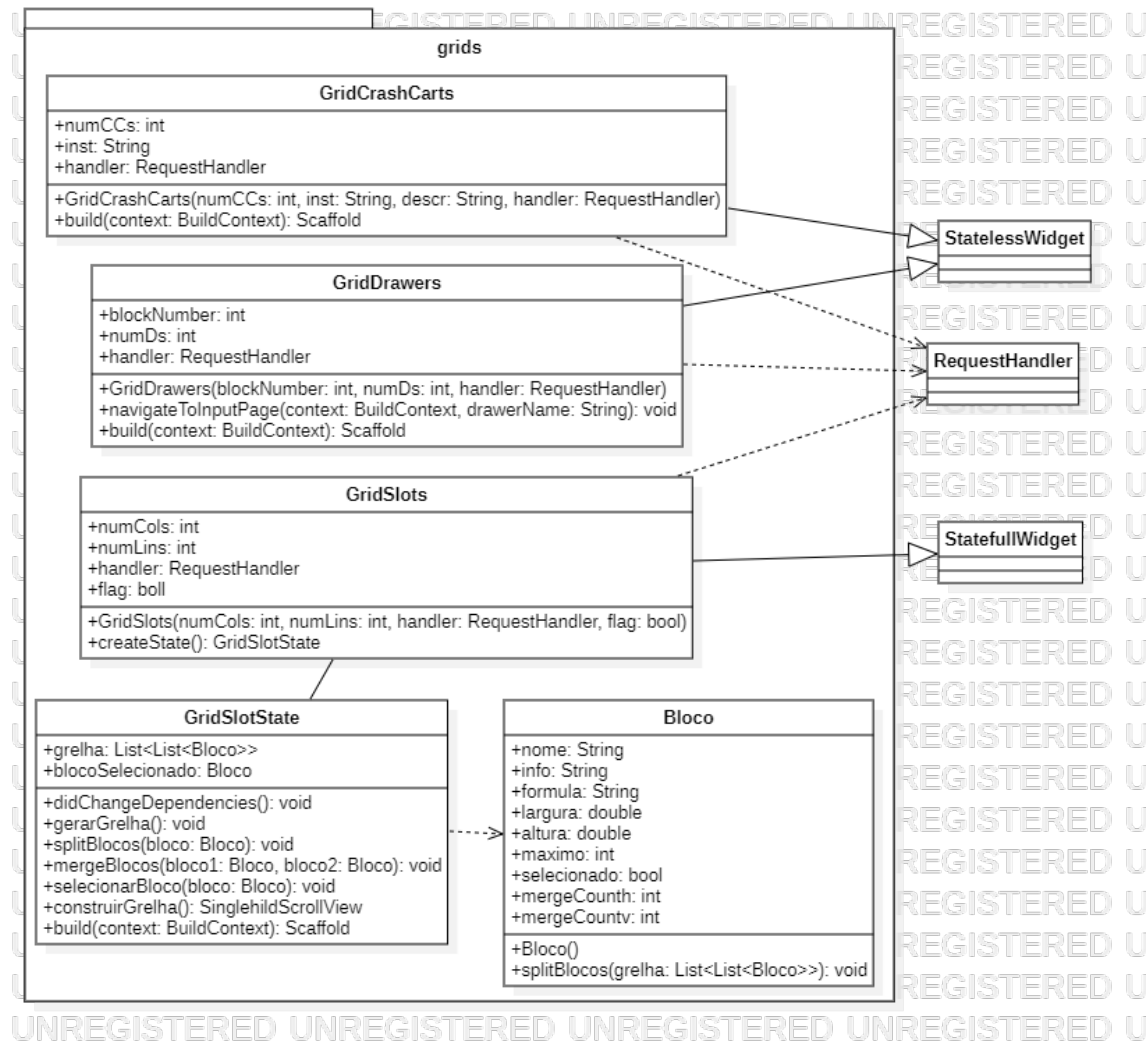
Figura 4.2: *Backend* - Diagrama de classes *models*

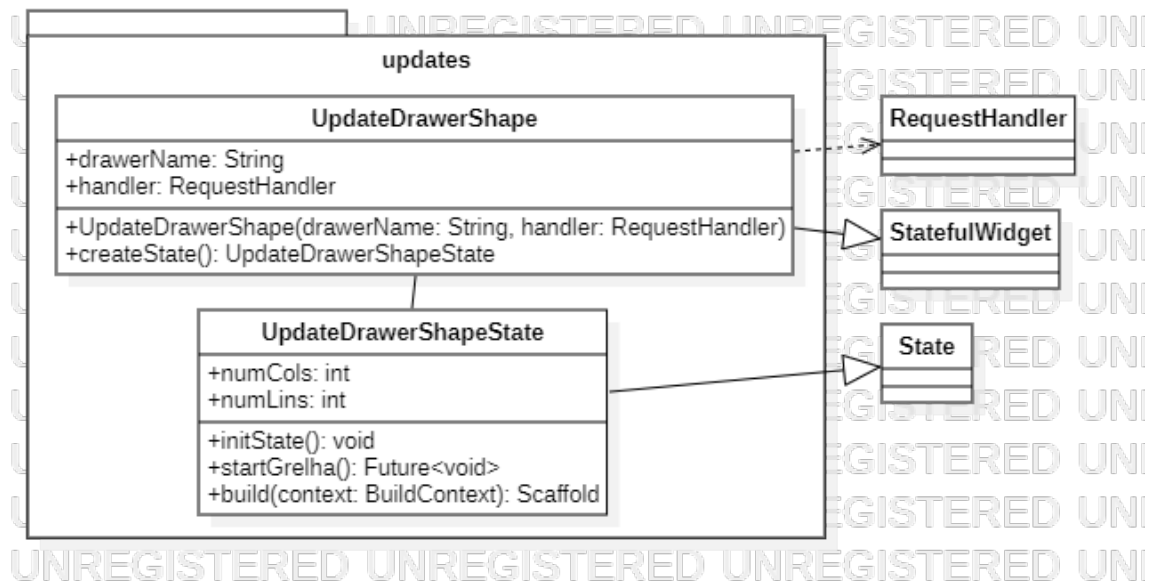
Figura 4.3: *Backend* - Diagrama de classes *serializers*

Figura 4.4: *Backend* - Diagrama de classes *views*

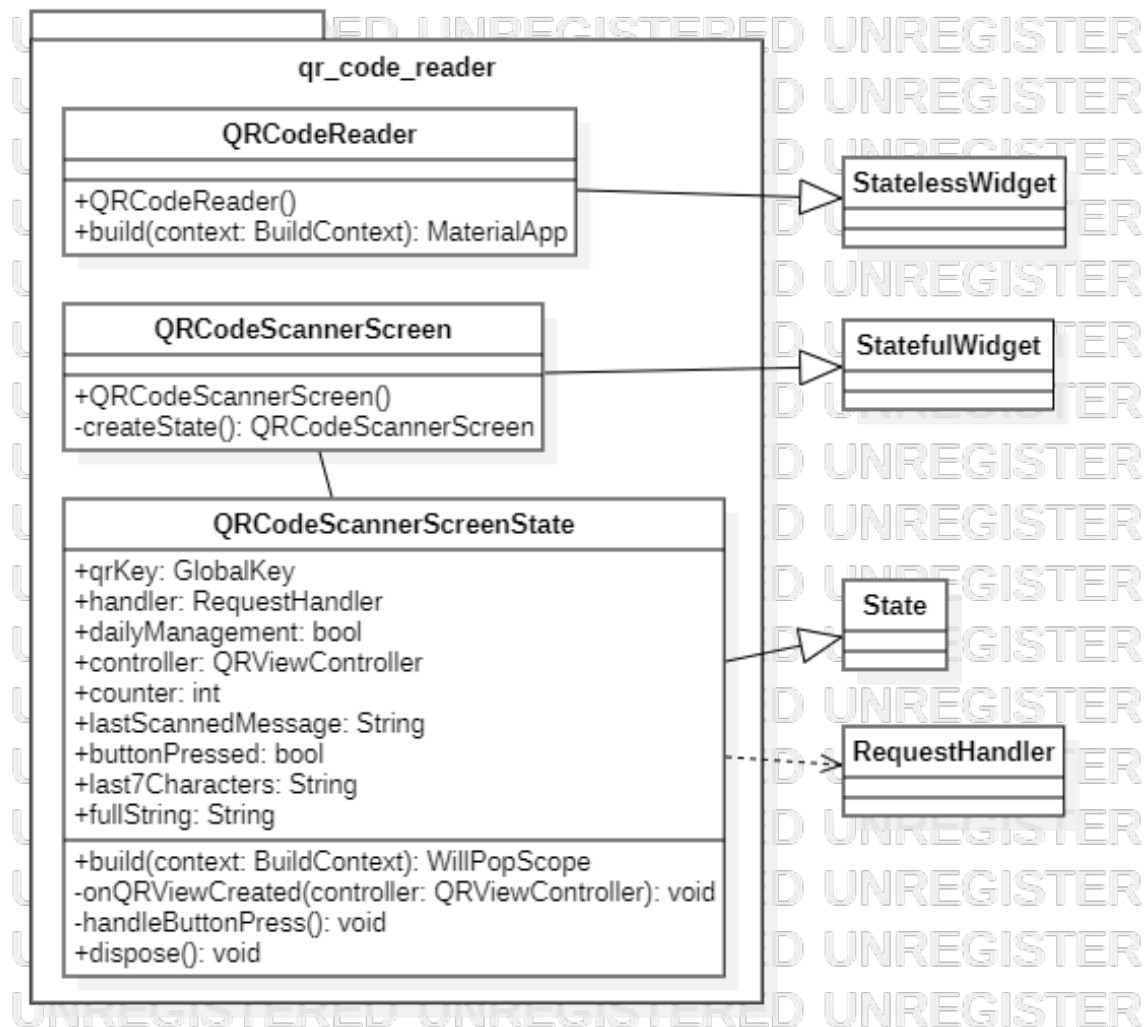
Figura 4.5: *Frontend* - Diagrama de classes *lib*

Figura 4.6: *Frontend* - Diagrama de classes *registration*

Figura 4.7: *Frontend* - Diagrama de classes *grids*

Figura 4.8: *Frontend* - Diagrama de classes *updates*



Figura 4.9: *Frontend* - Diagrama de classes `qr_code_reader`

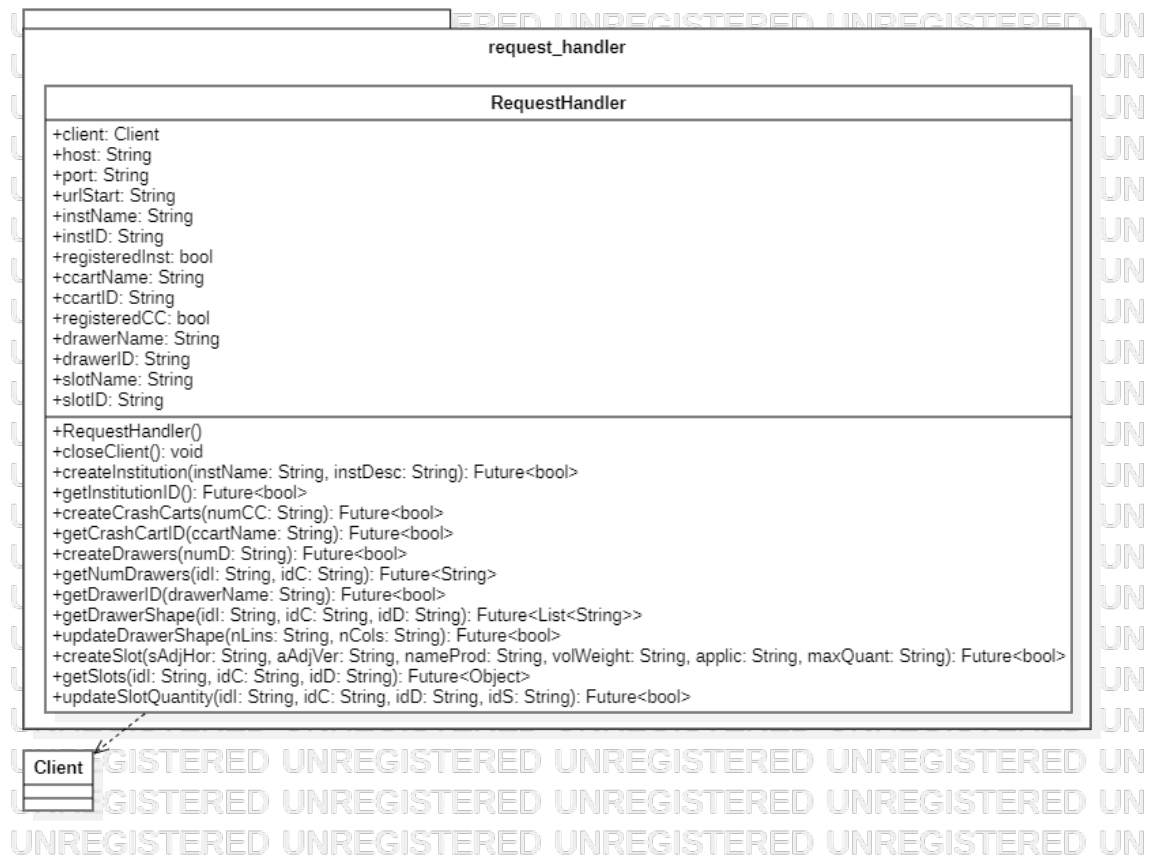


Figura 4.10: *Frontend* - Diagrama de classes *request\_handler*

### 4.1.2 Diagramas de Atividades

Estes diagramas permitem a modelação de processos, estabelecendo a comunicação entre as várias partes dos modelos e do sistema. Permitem também a documentação e desenho de *software*, alocação de recursos e identificar atividades paralelas e sequenciais.

A seguir apresentam-se os diagramas.

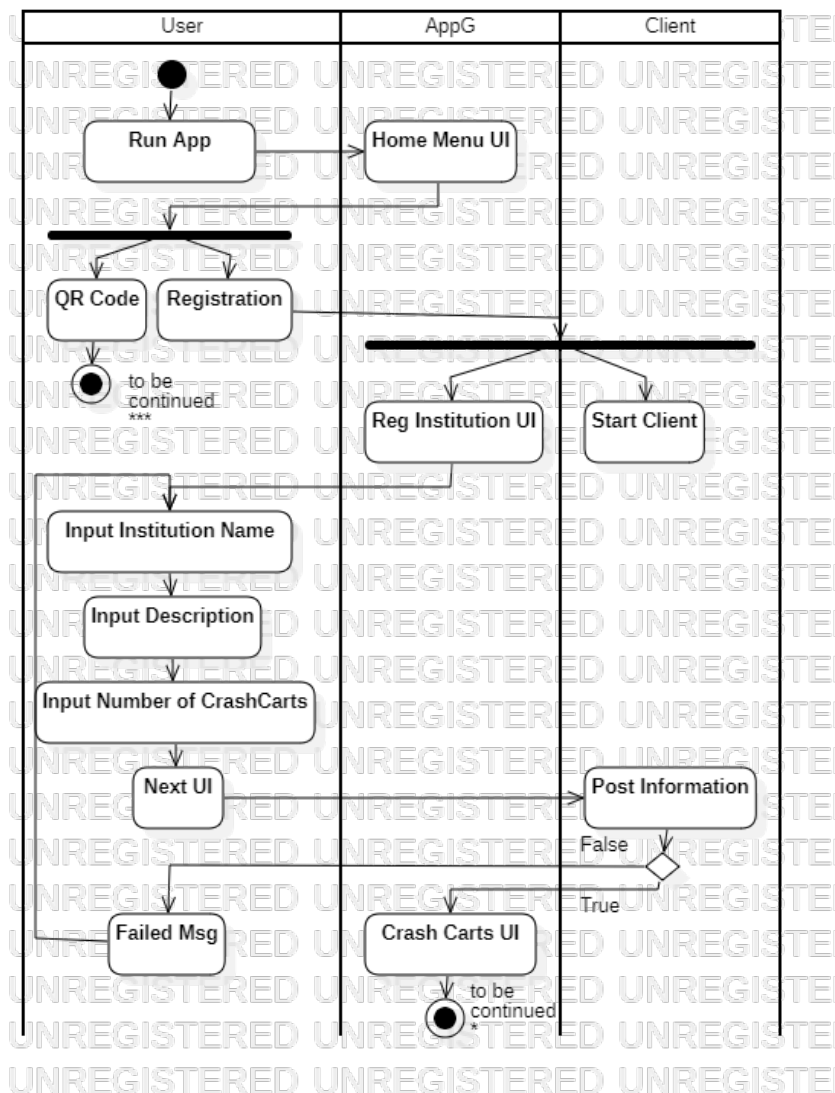


Figura 4.11: *Frontend* - Diagrama de atividades registo parte um

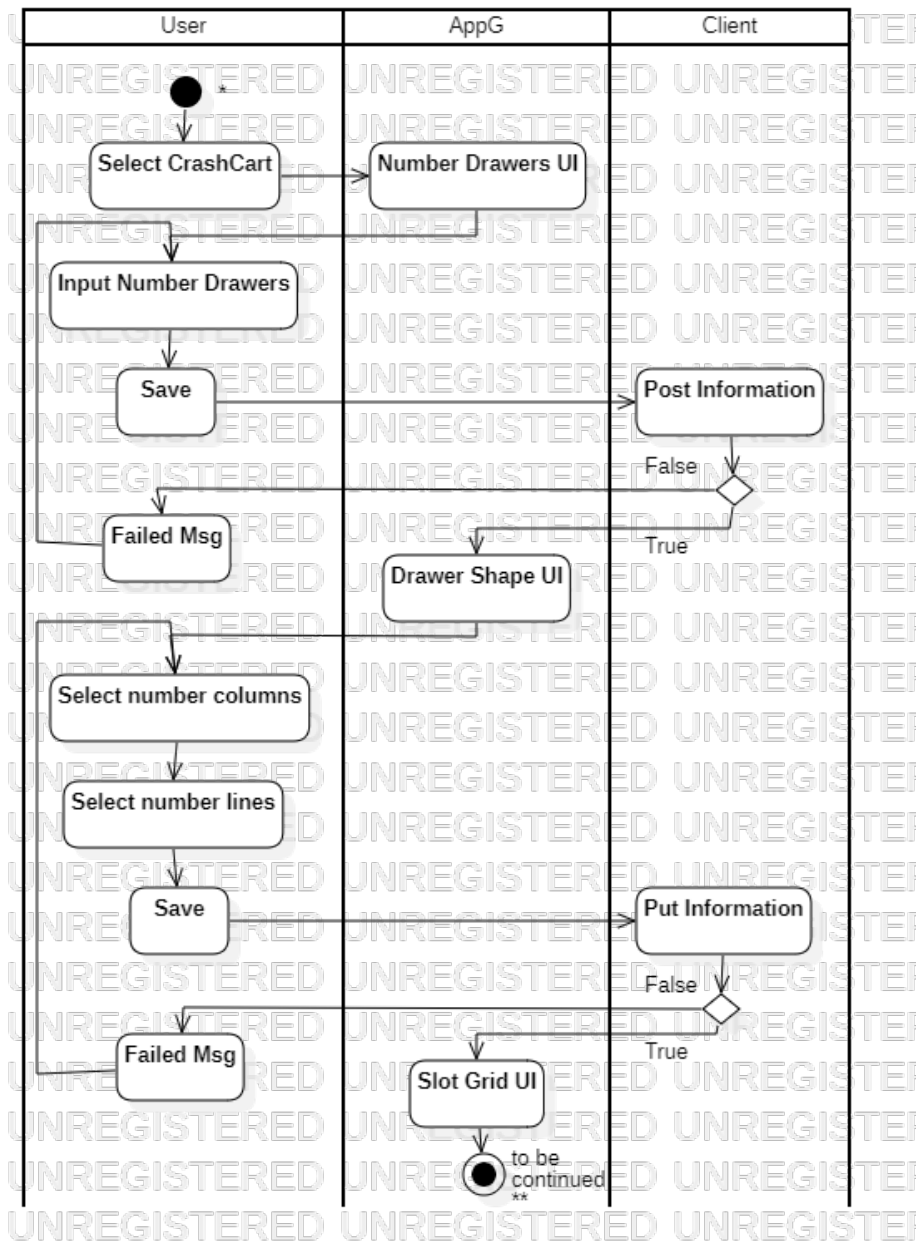


Figura 4.12: *Frontend* - Diagrama de atividades registro parte dois

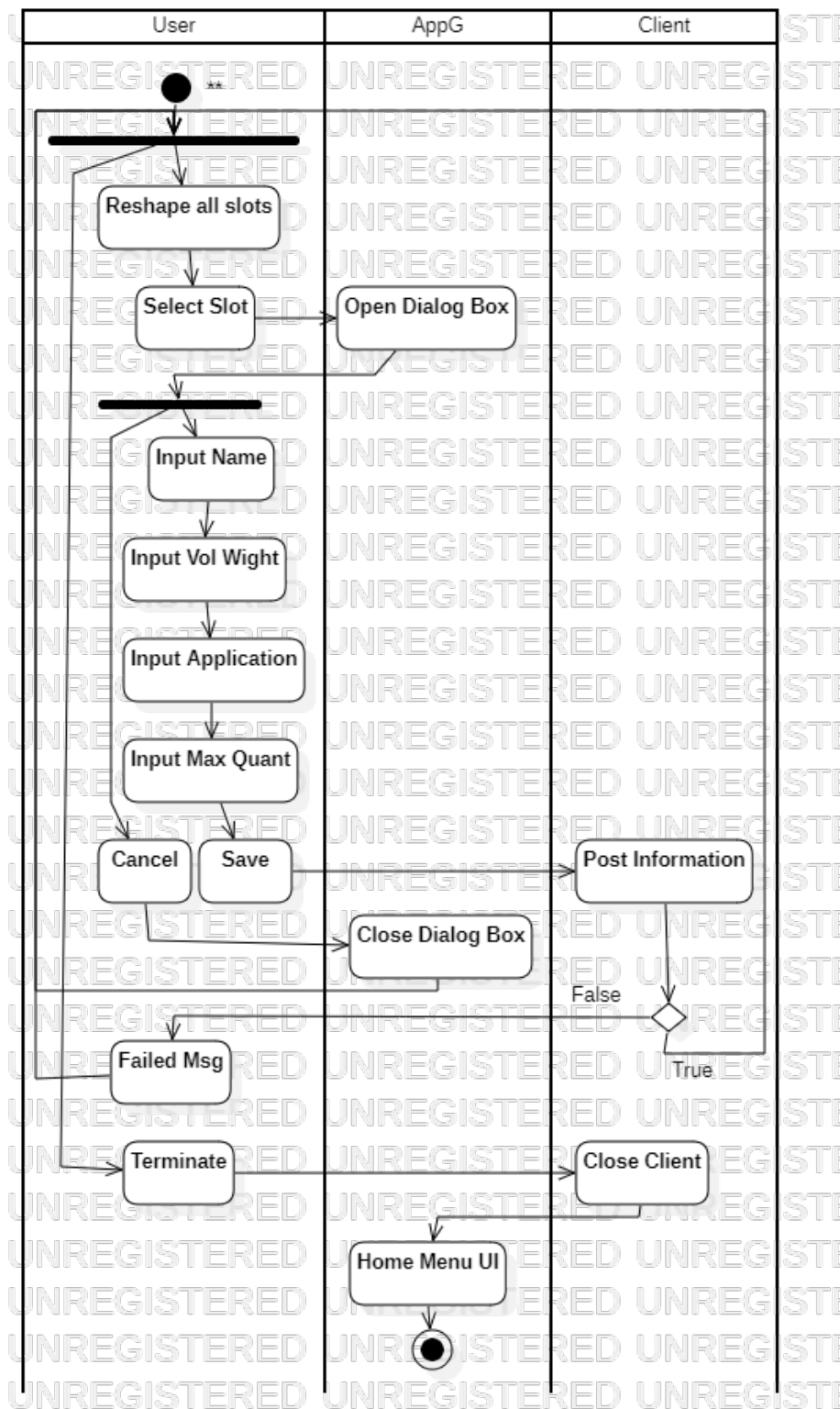


Figura 4.13: *Frontend* - Diagrama de atividades registro parte três

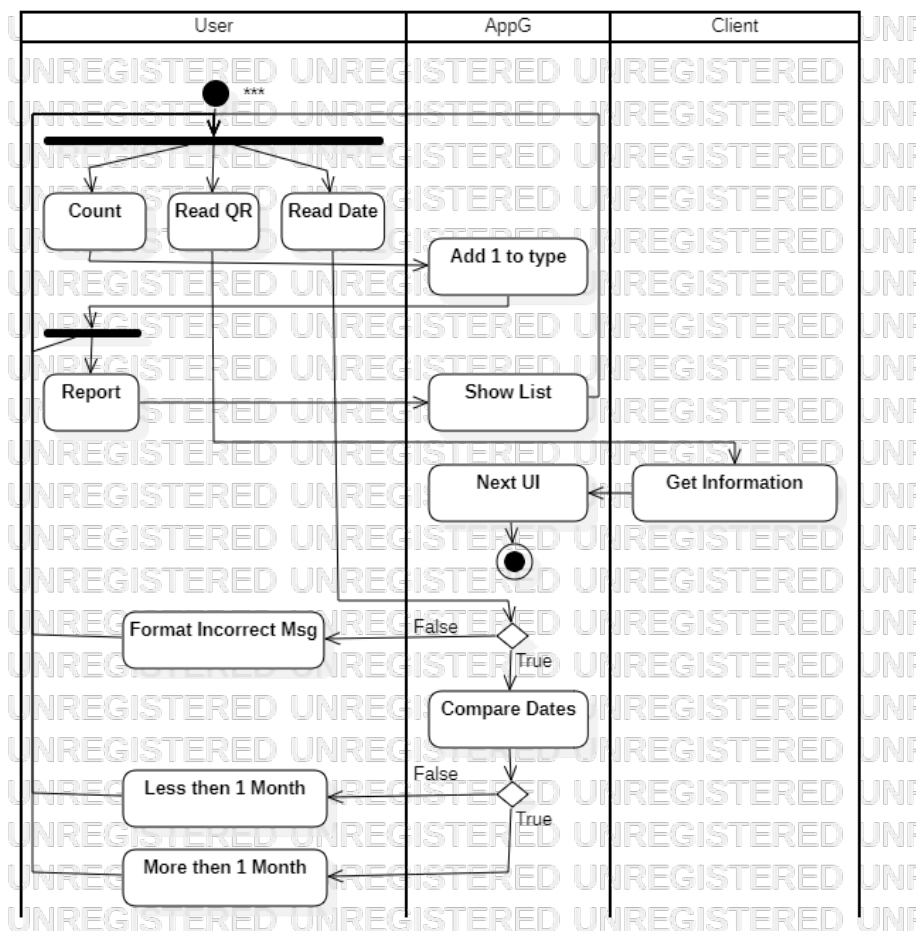
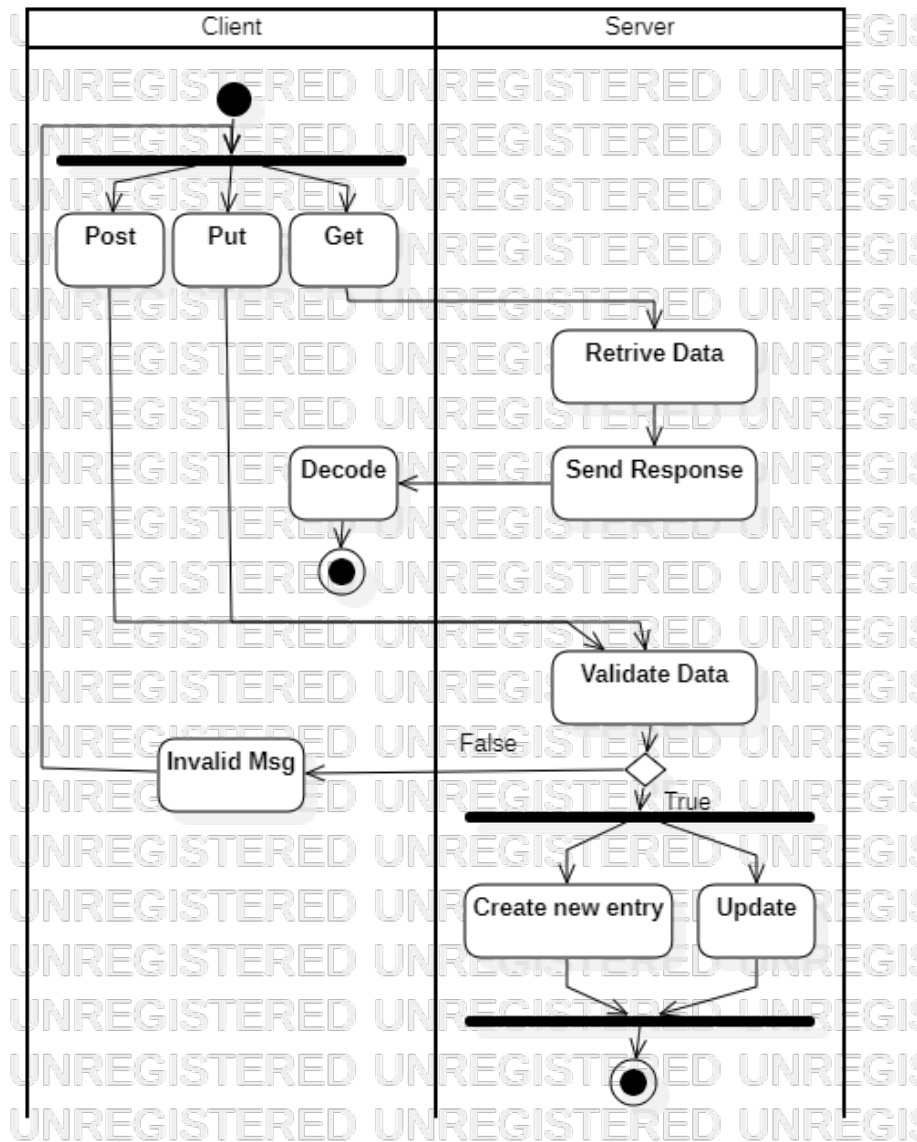


Figura 4.14: *Frontend* - Diagrama de atividades leitor de códigos

Figura 4.15: *Frontend* - Diagrama de atividades cliente-servidor

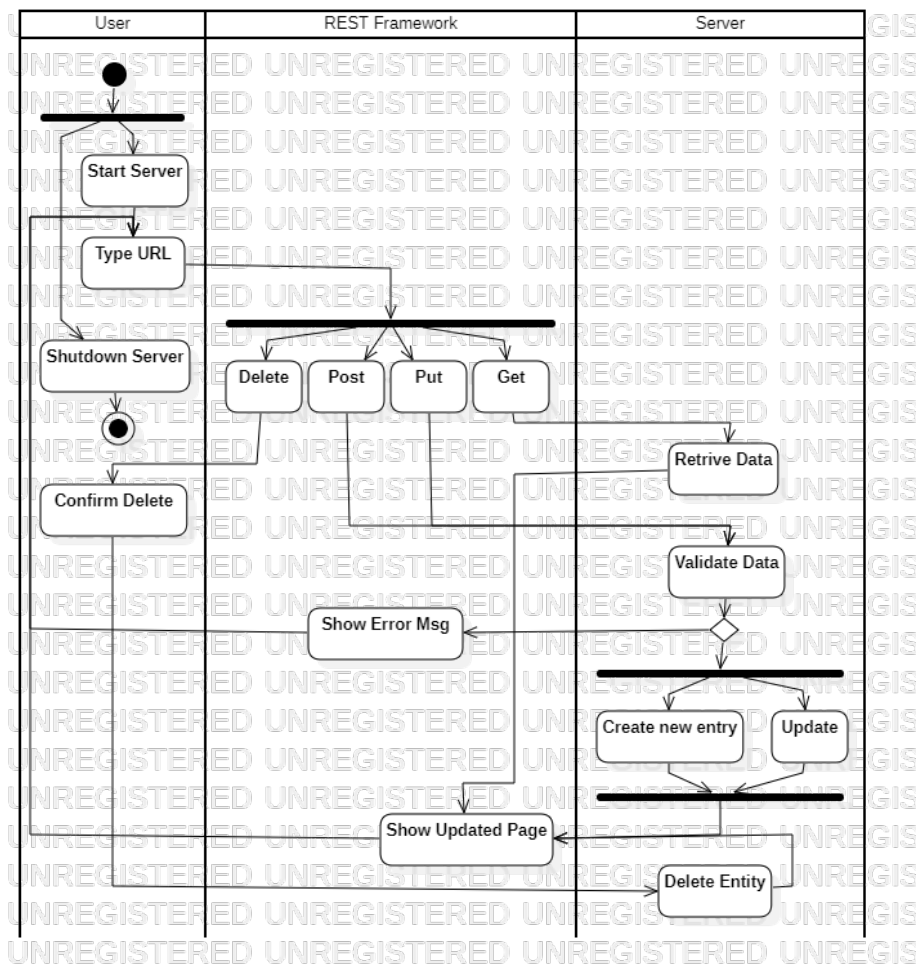


Figura 4.16: *Frontend* - Diagrama de atividades *REST framework*-servidor



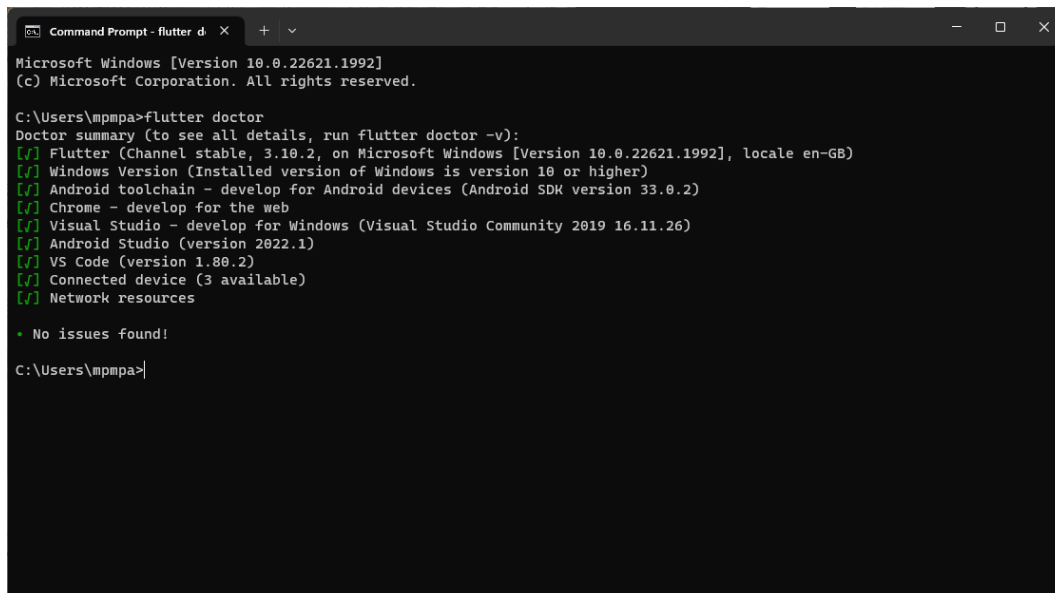
## 4.2 Tecnologias

Nesta secção fala-se das tecnologias usadas e como estas funcionam, a sua instalação e a criação de um ambiente de desenvolvimento.

### 4.2.1 Ambiente de desenvolvimento

Antes de iniciar o desenvolvimento do sistema, foi feita a instalação dos programas necessários para o projeto. O *Visual Studio*, *Visual Studio Code* e o *Python 3*, tal como os clientes *Web*, *Google Chrome* e *Microsoft Edge* já estavam instalados. O *Visual Studio* é uma das dependências da *framework Flutter*, o *Visual Studio Code* é o editor de código escolhido para o desenvolvimento, o *Python* é a linguagem de programação do servidor e os clientes *Web* são para testes da aplicação.

Para a *frontend* do sistema, a instalação da *framework Flutter* fez-se seguindo o site [9], o que nos obrigou, por causa das dependências da *framework*, cf. Figura 4.17 a instalação do *Android Studio*. No *Android Studio* foi instalado o mais recente *system development kit* - *sdk*, que corresponde à última versão do *Android* e criado um emulador correspondente a um *tablet Google* para testes no sistema *Android*.



```
Microsoft Windows [Version 10.0.22621.1992]
(c) Microsoft Corporation. All rights reserved.

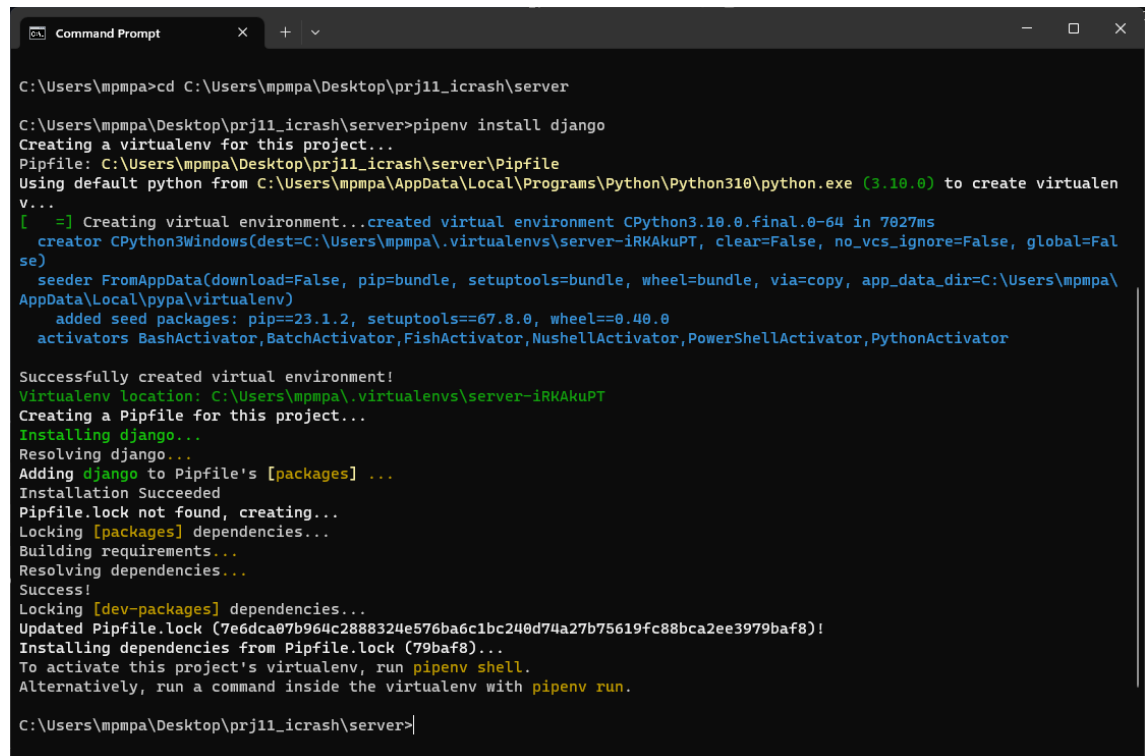
C:\Users\mpmpa>flutter doctor
Doctor summary (to see all details, run flutter doctor -v):
[✓] Flutter (Channel stable, 3.10.2, on Microsoft Windows [Version 10.0.22621.1992], locale en-GB)
[✓] Windows Version (Installed version of Windows is version 10 or higher)
[✓] Android toolchain - develop for Android devices (Android SDK version 33.0.2)
[✓] Chrome - develop for the web
[✓] Visual Studio - develop for Windows (Visual Studio Community 2019 16.11.26)
[✓] Android Studio (version 2022.1)
[✓] VS Code (version 1.80.2)
[✓] Connected device (3 available)
[✓] Network resources

• No issues found!

C:\Users\mpmpa>
```

Figura 4.17: Dependências da *framework Flutter*

Para a *backend* do sistema, o sistema de bases de dados *PostgreSQL*, foi instalado seguindo o site [10]. A *framework Django* foi instalada através de um comando na consola de comandos, cf. Figura 4.18. A instalação da *framework* foi feita com base nos seguintes vídeos de tutoriais no *Youtube* [11], [12].



```
C:\Users\mpmpa>cd C:\Users\mpmpa\Desktop\prj11_icrash\server

C:\Users\mpmpa\Desktop\prj11_icrash\server>pipenv install django
Creating a virtualenv for this project...
Pipfile: C:\Users\mpmpa\Desktop\prj11_icrash\server\Pipfile
Using default python from C:\Users\mpmpa\AppData\Local\Programs\Python\Python310\python.exe (3.10.0) to create virtualenv
v...
[ ] Creating virtual environment...created virtual environment CPython3.10.0.final.0-64 in 7027ms
creator CPython3Windows(dest=C:\Users\mpmpa\.virtualenvs\server-IRKakuPT, clear=False, no_vcs_ignore=False, global=False)
seeder FromAppData(download=False, pip=bundle, setuptools=bundle, wheel=bundle, via=copy, app_data_dir=C:\Users\mpmpa\AppData\Local\pypa\virtualenv)
added seed packages: pip==23.1.2, setuptools==67.8.0, wheel==0.40.0
activators BashActivator,BatchActivator,FishActivator,NushellActivator,PowerShellActivator,PythonActivator

Successfully created virtual environment!
Virtualenv location: C:\Users\mpmpa\.virtualenvs\server-IRKakuPT
Creating a Pipfile for this project...
Installing django...
Resolving django...
Adding django to Pipfile's [packages] ...
Installation Succeeded
Pipfile.lock not found, creating...
Locking [packages] dependencies...
Building requirements...
Resolving dependencies...
Success!
Locking [dev-packages] dependencies...
Updated Pipfile.lock (7e6dca07b964c2888324e576ba6c1bc240d74a27b75619fc88bca2ee3979baf8)!
Installing dependencies from Pipfile.lock (79baf8)...
To activate this project's virtualenv, run pipenv shell.
Alternatively, run a command inside the virtualenv with pipenv run.

C:\Users\mpmpa\Desktop\prj11_icrash\server>
```

Figura 4.18: Instalação do *Django* e criação do ambiente virtual

Para desenvolver o sistema, usou-se o *Visual Studio Code*, onde foram instaladas todas as extensões necessárias, tendo acesso ao sistema todo num só editor de código. As seguintes imagens, cf. Figura 4.19 e Figura 4.20, exibem o ambiente de desenvolvimento do sistema de um dos computadores do grupo.

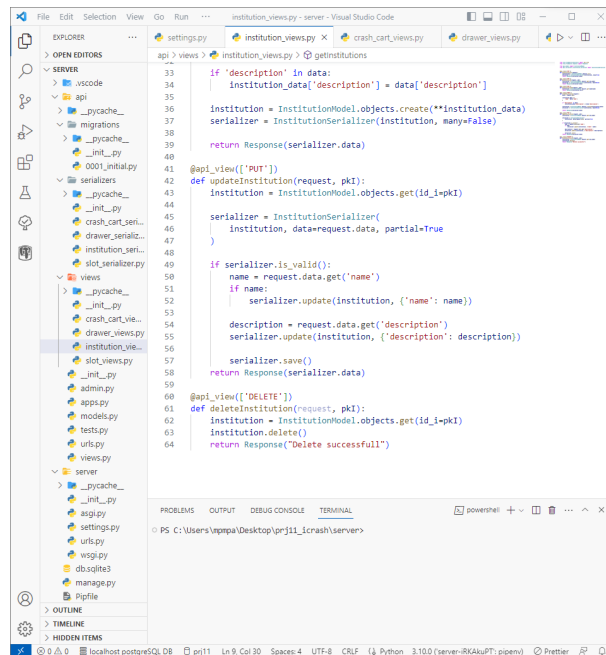


Figura 4.19: Ambiente de trabalho - Servidor

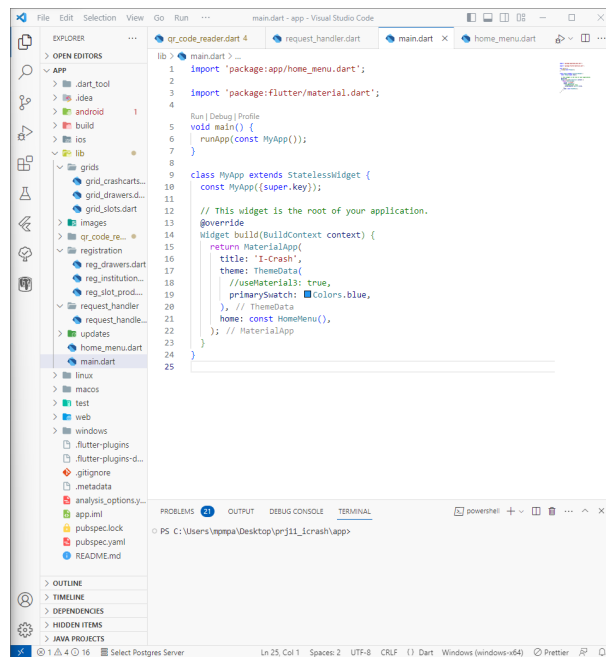


Figura 4.20: Ambiente de trabalho - Aplicação gráfica

### 4.2.2 *Flutter e Dart*

Como já foi falado, o *Flutter* é uma framework híbrida, assim sendo, a criação de uma aplicação *Flutter* automaticamente cria o sistema de pastas, cf. Figura 4.21. A criação dos ficheiros *.dart* para desenvolvimento do projeto e a organização da arquitetura é toda feita dentro da pasta *lib*. O ficheiro *main.dart* é também automático, e é neste que ocorre a execução do sistema *frontend* desenvolvido. As restantes pastas automaticamente criadas e restantes ficheiros permitem a instalação de bibliotecas e a transformação do código para outras linguagens de programação consoante o sistema operativo para onde o sistema é exportado.

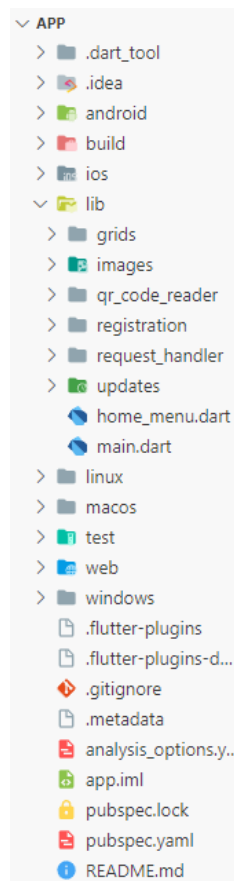


Figura 4.21: Sistema de pastas da *framework Flutter*

A linguagem *Dart*, é orientada a objetos, denominados *widgets* que podem ser *stateful* ou *stateless*, caso sejam atualizados (mudança de estado

do *widget*) em termos de apresentação gráfica ou não respetivamente. Estes *widgets* podem ser graficamente apresentados de formas diferentes consoante o tipo de *widget*, por exemplo, botões, contentores ou caixas de texto para *input* de dados. Suporta comunicação assíncrona, usada na comunicação com o servidor. Tem um *garbage collector* libertando memória automaticamente. Permite a compilação do código antes de tempo e no momento de execução, o que permite recarregar mais rápido, após alterações e correção de erros.

### 4.2.3 *Django, REST framework e Python*

Com a instalação da *framework Django*, é automaticamente criado o sistema de pastas, cf. Figura 4.22, com exceção da pasta *api*, a qual é dada a escolha do nome. Esta *framework* permite a existência de uma interface de administração para controlo dos dados guardados. Permite o mapeamento do modelo de dados para evitar as *queries*, fazendo o mapeamento de *urls* e *views*. Estas últimas são codificadas através da *REST framework*.

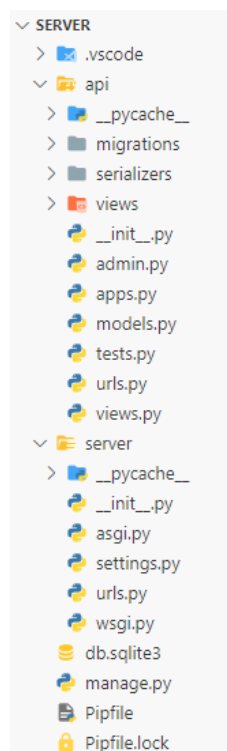


Figura 4.22: Sistema de pastas da *framework Django*

A *REST framework* foi então usada para a criação de *views* e *serializers*. Neste contexto *views* são funções em *Python* que lidam com solicitações *Hypertext Transfer Protocol - HTTP* e retornam respostas *HTTP*, sendo que através da *REST framework* temos disponíveis páginas *Web* para utilizar estas funções através de uma interface gráfica já construída. Os *serializers* permitem a conversão de tipos de dados complexos, como por exemplo, objetos *Python* para *JavaScript Object Notation - JSON* ou *eXtensible Markup Language - XML* e vice-versa.

A linguagem *Python* é de alto nível, para uso geral e orientada a objetos, sendo fácil de ler, escrever e interpretar, usada muito para a análise de grandes conjuntos de dados.

#### 4.2.4 *PostgreSQL*

O *PostgreSQL* é um programa que permite o desenvolvimento e gestão de um sistema de gestão de base de dados. É através deste programa que os dados são guardados e acedidos através de *queries*. No entanto apenas damos uso há capacidade de guardar e relacionar os dados, visto que todos os pedidos são realizados através do servidor, bem como o desenvolvimento da estrutura dos dados.

A linguagem *Structured Query Language - SQL* é uma linguagem de programação desenhada para a gestão e manipulação de bases de dados relacionais. Permite a criação, modificação e *querie* da base de dados para guardar, atualizar, eliminar e obter dados.

### 4.2.5 *Star UML*

O *Star UML* é usado para a elaboração de modelos na linguagem *Unified Modeling Language - UML*. Este programa permite a elaboração de todos os diagramas apresentados na arquitetura, necessários para o bom funcionamento do sistema.

*UML* é uma linguagem de alto nível, porque permite o desenho, visualização, especificação e documentação de sistemas de *software* com um alto nível de abstração, onde a linguagem de programação não é considerada, facilitando a comunicação e compreensão da estrutura e comportamento destes sistemas, através de um conjunto de diagramas.

## 4.3 Desenvolvimento da aplicação gráfica

Falamos a seguir do desenvolvimento da aplicação gráfica, repartida em três partes:

- Interface gráfica
- Implementação do leitor de códigos
- Desenvolvimento do cliente

### 4.3.1 Interface gráfica

Para a aplicação ser orientada ao utilizador foram criadas oito interfaces gráficas, sendo que cada uma dessas interfaces tem como objetivo, dar ao utilizador opções de escolha, que vão mudar o aspeto das interfaces que as seguem.

- Primeira interface: Esta interface tem como objetivo ser a capa da aplicação, apresentando os dois logótipos das faculdades (do lado esquerdo o logótipo da ESEL - Escola Superior de Enfermagem de Lisboa, e à direita o logótipo do ISEL - Instituto Superior de Engenharia de Lisboa), por baixo destes logótipos estão presentes dois botões, um botão de registo, redirecionando o utilizador para o registo da instituição, carros de emergência e o seu conteúdo, e um botão de acesso ao leitor de códigos. A seleção de um dos botões permite iniciar a comunicação com o servidor.
- Segunda interface: Esta interface permite o registo de uma instituição, através do seu nome com a opção de uma pequena descrição relacionada com a mesma. Permite também o registo do número de carros, (para esta opção foi implementada com o mínimo de um carro e um máximo de cem carros). Esta interface contém também dois botões, um presente no topo esquerdo da interface, que permite ao utilizador voltar para a primeira interface (capa da aplicação) e um outro botão, que permite ao utilizador passar para a próxima e terceira interface do registo se o mesmo for bem sucedido.



- Terceira interface: Esta interface exhibe ao utilizador uma grelha com os carros existentes num determinado instituto, no topo desta interface está presente o nome da instituição e um botão que permite voltar para a página anterior. A grelha de carros, cada bloco representa um carro, cada bloco contém o número do carro, no formato *CrashCart n*, onde  $n$  é o número do carro. O utilizador ao tocar num bloco será redirecionado para a seguinte e quarta interface.
- Quarta interface: Esta interface permite ao utilizador escolher o número de gavetas que um carro contém, registando o número de gavetas. Também está presente no topo da interface um botão que permite ao utilizador voltar para a interface anterior. A interface contém a identificação do carro (*CrashCart n*) indicando ao utilizador, em que carro está a ser realizada a criação de gavetas e um espaço para o utilizador introduzir o número de gavetas que o carro contém. A interface contém um outro botão que permite passar para seguinte e quinta interface, se o registo das gavetas for bem sucedido.
- Quinta interface: Esta interface apresenta o número do carro no qual o utilizador está a trabalhar e o número de gavetas que foram criadas a partir da quarta interface. Esta interface contém também um botão que permite ao utilizador voltar há interface anterior, cada uma das gavetas presentes num carro é representada com um botão que ao ser premido, redireciona o utilizador para a próxima e sexta interface.
- Sexta interface: Esta interface tem como objetivo escolher o número de linhas e colunas da grelha que representa uma gaveta, sendo que para escolher o número de linhas e colunas foram definidos dois menus *dropdown*, onde existe um conjunto de números de um a doze, estes correspondentes respetivamente ao mínimo e máximo número de linhas e colunas. A interface exhibe também um botão onde é atualizada a forma da gaveta (número de linhas e colunas) e feita a transição para a próxima e sétima interface.
- Sétima interface: Esta interface representa uma gaveta de um carro de emergência através de uma grelha de *slots*, onde estão presentes as informações dos consumíveis clínicos. Para concretizar esta grelha

é feito o cálculo do número de *slots* existentes na gaveta a partir do número de linhas e colunas da mesma, depois é realizado o cálculo do tamanho que cada *slot* terá, a partir do tamanho da tela, finalmente estes (*slots*) são guardados como posições de uma lista. Esta interface tem também a opção de fazer *merge* de dois ou mais *slots* e *split* de *slots* previamente *merged*, possibilitando a criação de *slots* com vários tamanhos e formatos. Não é possível a separação de um *slot* que não tenha sido previamente *merged*. Existe também um botão terminar que retorna o utilizador à primeira interface e fecha a comunicação com o servidor. Esta interface, na seleção de um dos *slots* exibe uma *dialog box*, onde é permitido o preenchimento da informação do consumível clínico. A *dialog box* exibe dois botões um para cancelar o outro para guardar a informação e envio da mesma para o servidor.

- Oitava interface é explicada na implementação do leitor de códigos.

### 4.3.2 Implementação do leitor de códigos

Esta interface tem como objetivo realizar a leitura de código QR e redirecionar o utilizador para a página correspondente ao conteúdo lido. Também permite, através do mesmo botão realizar a contagem de consumíveis clínicos que foram retirados de um certo *slot* e ao pressionar o botão de relatório final, é apresentado ao utilizador uma lista numa *dialog box* com nomes e números de produtos que foram retirados naquela emergência. É possível também ler uma data de validade com o formato ano-mês-dia ao pressionar o botão Verificar Data, este irá comparar com a data atual e verificar se tem mais ou menos um mês de validade, se tiver mais de um mês irá mostrar ao utilizador uma mensagem de que está dentro da validade, se tiver menos irá mostrar que está fora da validade. Caso seja feita a leitura de um formato errado, irá apresentar uma mensagem a dizer ao utilizador que está a ler um texto com o formato errado.

### 4.3.3 Desenvolvimento do cliente

Foi importado o *package http* e atualizado o ficheiro *pubspec.yaml* nas dependências do *Flutter*, sendo a partir deste que o cliente persistente é desenvolvido, bem como o conjunto de funções assíncronas para solicitação

e respostas *HTTP*. Este encontra-se no *package request\_handler* no ficheiro *.dart* com o mesmo nome.

O nome da instância, *RequestHandler*, deve-se ao objetivo desta classe. Esta classe exerce um conjunto de funções que realizam solicitações *HTTP* para o servidor, aguardando respostas *HTTP*. São usados três tipos de solicitações:

- *POST* neste contexto através de *client.post*, usado para o envio de novos dados ao servidor
- *PUT* neste contexto através de *client.put*, usado para o envio de dados para atualização
- *GET* neste contexto através de *client.get*, usado para pedir dados ao servidor

O cliente para ser persistente, é definido através da seguinte linha de código *final Client = http.Client();*, tudo o resto é tratado pelo *package http*. É persistente para que as mensagens sejam mais rápidas, para que o cliente guarde informação da conexão no caso da necessidade de repetição de solicitações, conservando recursos em conexões fracas e sendo escalável na existência de muito tráfego de dados, estabelecendo também uma conexão persistente em caso de falhas de rede.

As funções são assíncronas através do uso da palavra chave *async* associado ao uso das palavras chave *Future* e *await*. Isto deve-se há necessidade de que a interface continue a execução e possa ser atualizada futuramente, durante e após um pedido do cliente ao servidor. Por exemplo, cf. Figura 4.23 a função *createInstitution()*, é chamada de forma assíncrona, isto porque, a condição *if* espera o retorno da função, seja este *true* ou *false*. Este valor só será atualizado futuramente, ou seja, após o *post()* feito pelo cliente, e durante a execução desta função e deste pedido do cliente, a execução da interface gráfica não deve ser interrompida, evitando erros que possam bloquear o funcionamento da aplicação.

```
48 Future<bool> createInstitution(String instName, String instDesc) async {  
49     if (registeredInst) return registeredInst;  
50  
51     String url = '$urlStart/institution/create/';  
52     final data = {'name': instName, 'description': instDesc};  
53  
54     final response = await client.post(Uri.parse(url), body: data);  
55     if (response.statusCode == 200) {  
56         this.instName = instName;  
57         registeredInst = true;  
58         return true;  
59     }  
60     return false;  
61 }
```

Figura 4.23: *createInstitution()* - Exemplo de uma função assíncrona

## 4.4 Sistema de gestão de base de dados

O sistema de gestão de base de dados foi inicialmente desenhado, usando o aplicativo online [13] com base no modelo entidade associação, cf. Figura 4.24.

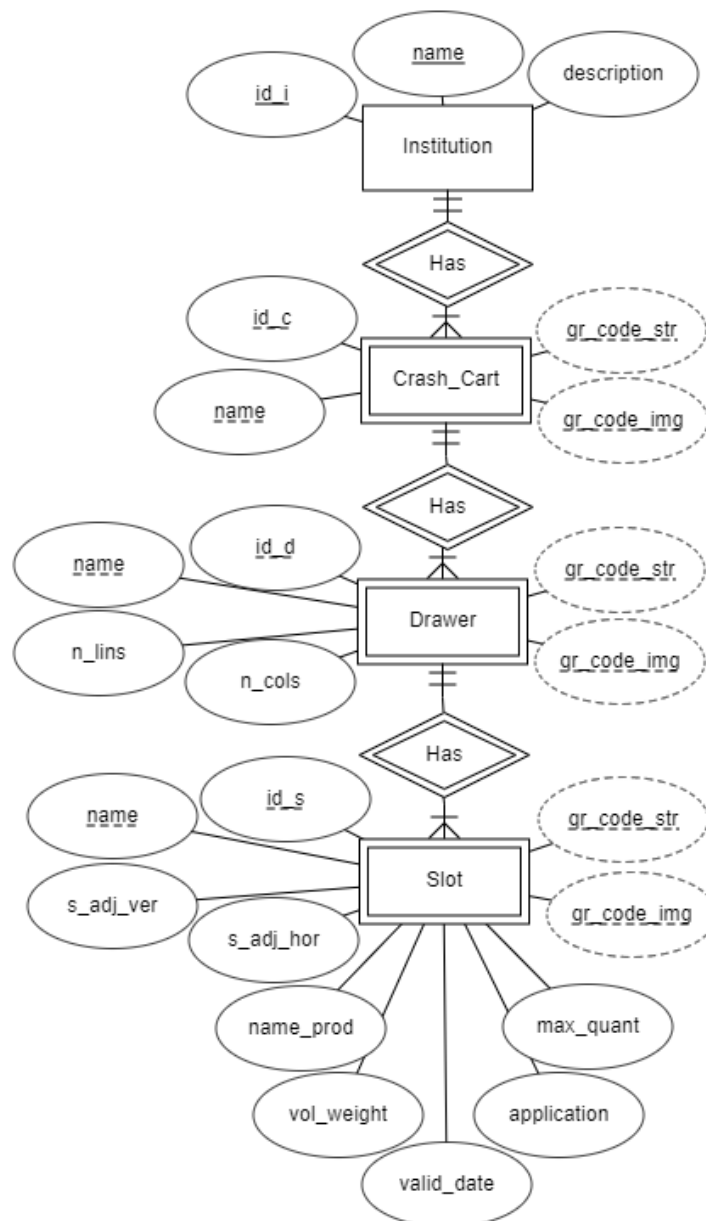


Figura 4.24: Modelo entidade associação

Este modelo encontra-se normalizado, e estabelece uma sequência de relações e entidades fracas de outras, representadas pelo duplo losango e duplo retângulo, ou seja, por exemplo, a existência de uma entidade *Crash\_Cart* só fez sentido se esta pertencer a uma entidade *Institution*. O mesmo acontece para as entidades *Drawer* e *Slot*. Observa-se que, por exemplo, uma entidade *Crash\_Cart* pertence a uma só entidade *Institution* obrigatoriamente, relação representada através do duplo traço. No sentido contrário, uma entidade *Institution* tem obrigatoriamente várias entidades *Crash\_Cart*, relação representada através do traço e do "pé de galo". No modelo, os atributos sublinhados com um traço são únicos, os sublinhados com um tracejado são parcialmente únicos, por causa da relação. Os atributos rodeados a tracejado são calculados através de outros atributos, por exemplo, os vários atributos *qr\_code\_str* são definidos através dos vários identificadores.

Em seguida apresenta-se o modelo entidade relação:

```

Institution (
    id_i ,
    name,
    description
)
Primary key: id_i
Candidate keys: name

CarshCart (
    id_i ,
    id_c ,
    name,
    qr_code_str ,
    qr_code_img
)
Primary key: (id_i , id_c)
Foreign keys: id_i
Candidate keys: qr_code_str , qr_code_img

Drawer (
    id_i ,
    id_c ,
    id_d ,
    name,
    n_lins ,
    n_cols ,

```

```

        qr_code_str ,
        qr_code_img
    )
Primary key: (id_i , id_c , id_d)
Foreign keys: (id_i , id_c)
Candidate keys: qr_code_str , qr_code_img

Slot(
    id_i ,
    id_c ,
    id_d ,
    id_s ,
    name ,
    s_adj_hor ,
    s_adj_ver ,
    name_prod ,
    vol_weight ,
    application ,
    max_quant ,
    valid_date ,
    qr_code_str ,
    qr_code_img
)
Primary key: (id_i , id_c , id_d , id_s)
Foreign keys: (id_i , id_c , id_d)
Candidate keys: qr_code_str , qr_code_img

```

Apesar deste modelo entidade associação e deste modelo entidade relação, o uso da *framework Django* para desenvolver o servidor obrigou à reestruturação do modelo de dados, a qual será apresentada e explicada no próximo subcapítulo.

## 4.5 Desenvolvimento do Servidor

O desenvolvimento do servidor foi realizado como prova de conceito de uma arquitetura cliente-servidor, bem como, com o objetivo de prova de um sistema, onde os dados estão seguros e não são perdidos, caso um dispositivo se estrague, ou seja, os dados não são guardados localmente.

A escolha da *framework Django*, não sendo a primeira, mas a que funciona, após as tentativas com outras, obrigou a uma reestruturação do modelo de dados apresentada a seguir.

### 4.5.1 Desenvolvimento do modelo de dados

A *framework Django* contendo um *ORM*, obriga à existência de algumas restrições, a que interessa, é a incapacidade do mapeamento de chaves primárias com duas ou mais colunas, ou seja, não podemos ter relações e entidades fracas, pois a chave primária dessas entidades, neste caso, *Crash\_Cart*, *Drawer* e *Slot* contem as chaves estrangeiras, chaves primárias das entidades fortes.

Com a *framework* e estas incapacidades no mapeamento, é aconselhado o desenvolvimento e modelação através da *framework*, e o mesmo foi feito. Para resolver o problema das chaves primárias, mantiveram-se os identificadores modelados (*id\_c*, *id\_d*, *id\_s*), e acrescentou-se novos identificadores que funcionam como chaves primárias. Assim não existem as relações e entidades fracas, mas são simuladas através dos identificadores modelados, ignorando as chaves primárias. A seguir, cf. Figura 4.25, apresenta-se a modelação de uma entidade com esta reestruturação. Observando a figura, a chave primária é gerada automaticamente e neste caso *id\_c* é inteiro e o seu incremento é controlado pelo seguinte código (semelhante para os restantes modelos), cf. Figura 4.26. A entidade *Institution* é a única, que não foi reestruturada.

```

38 class CrashCartModel(models.Model):
39     """
40     Model of the CrashCart table in the database.
41
42     Args:
43         institution (ForeignKey): The ID of the Institution. Foreign key
44         relation to the Instituiton model.
45         id_c (IntegerField): Identifies the CrashCart.
46         name (CharField): The name of the CrashCart.
47         qr_code_str (CharField): The QR code string of the CrashCart.
48         qr_code_img (BinaryField): The binary representation of the QR code
49         image of the CrashCart.
50     """
51     institution = models.ForeignKey('InstitutionModel', on_delete=models.CASCADE)
52     id_c = models.IntegerField()
53     name = models.CharField(max_length=20)
54     qr_code_str = models.CharField(max_length=20, unique=True)
55     qr_code_img = models.BinaryField(unique=True, null=True)

```

Figura 4.25: Modelo *CrashCart* reestruturado



```
57     def save(self, *args, **kwargs):
58         """
59         To control the increment of the id_c of a CrashCart depending on the
60         Institution. This way there can be multiple CrashCarts with the same
61         id_c, but different id_i's.
62         """
63         if not self.pk: # Only executed on creation, not on updates
64             existing_crash_carts = CrashCartModel.objects.filter(
65                 institution=self.institution
66             )
67             if existing_crash_carts.exists():
68                 last_crash_cart = existing_crash_carts.order_by('-id_c').first()
69                 self.id_c = last_crash_cart.id_c + 1
70             else:
71                 self.id_c = 1
72         super().save(*args, **kwargs)
```

Figura 4.26: Modelo *CrashCart* controle do *id\_c*

Com a modelação feita, é necessário fazer as migrações e executá-las para o sistema de gestão de base de dados *PostgreSQL*. Foi modificado o ficheiro *settings.py* para que o motor do sistema de gestão de base de dados fosse o *PostgreSQL* e executados os comandos, cf. Figura 4.27. Apenas um dos ficheiros, o ficheiro *0001\_initial.py* é que contém o modelo de dados, os restantes são automaticamente gerados pela *framework Django*.

```
PS C:\Users\mpmpa\Desktop\prj11_icrash\server> python manage.py makemigrations
Migrations for 'api':
  api\migrations\0001_initial.py
    - Create model CrashCartModel
    - Create model DrawerModel
    - Create model InstitutionModel
    - Add field institution to crashcartmodel
    - Create model SlotModel
    - Alter unique_together for crashcartmodel (1 constraint(s))
PS C:\Users\mpmpa\Desktop\prj11_icrash\server> python manage.py migrate
Operations to perform:
  Apply all migrations: admin, api, auth, contenttypes, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying admin.0003_logentry_add_action_flag_choices... OK
  Applying api.0001_initial... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying auth.0008_alter_user_username_max_length... OK
  Applying auth.0009_alter_user_last_name_max_length... OK
  Applying auth.0010_alter_group_name_max_length... OK
  Applying auth.0011_update_proxy_permissions... OK
  Applying auth.0012_alter_user_first_name_max_length... OK
  Applying sessions.0001_initial... OK
```

Figura 4.27: Comandos usados para migrar o modelo de dados

A geração automática do ficheiro *0001\_initial.py* contém alguns erros que foram editados antes de executar o segundo comando no terminal, cf. Figura 4.27. Estes são, a geração alfabética das entidades, que faz com que seja necessária a adição da referência *institution* após a criação do modelo *CrashCartModel*, e a adição de uma única restrição, quando mais são necessárias.

Para corrigir estes erros, refere-se a criação do modelo *InstitutionModel* antes do modelo *CrashCartModel* e foram adicionadas manualmente as restantes restrições.

A adição da linha *unique\_together* para os identificadores, é a que permite a simulação das entidades e relações fracas, cf. Figura 4.28.

```
83 | class Meta:
84 |     """
85 |     Define the context where name and id_c are unique.
86 |     """
87 |     unique_together = (('institution', 'name'))
88 |     unique_together = (('institution', 'id_c'))
```

Figura 4.28: Exemplo do uso *unique\_together*

## 4.5.2 Desenvolvimento de *Serializers*

Os *Serializers* são simples, a sua implementação requer uma função *update* para atualizar os valor dos atributos dos vários modelos acima referidos, bem como uma sub classe *Meta*, onde se define o modelo e os atributos do mesmo. Em seguida tem-se exemplo de uma classe *Serializer* para um dos modelos desenvolvidos, cf. Figura 4.29.

```
6 | class InstitutionSerializer(ModelSerializer):
7 |     """
8 |     Converts the type of data, to create, update and delete from the system
9 |     database an Institution.
10 |     """
11 |     class Meta:
12 |         """
13 |         Define the model type, and the fields.
14 |         """
15 |         model = InstitutionModel
16 |         fields = '__all__'
17 |
18 |     def update(self, instance, validated_data):
19 |         """
20 |         Updates the instance with the validated_data received.
21 |
22 |         Args:
23 |             instance (InstitutionModel): The instance.
24 |             validated_data (dict): A dictionary with the data.
25 |
26 |         Returns:
27 |             InstitutionModel: The model updated.
28 |         """
29 |         instance.name = validated_data.get('name', instance.name)
30 |         instance.description = validated_data.get(
31 |             'description', instance.description
32 |         )
33 |         instance.save()
34 |         return instance
```

Figura 4.29: Exemplo de um *Serializer*

Como já foi dito, os *Serializers* fazem a conversão de objetos complexos, em dados *JSON* e/ou *XML*, neste sentido, apesar de apenas existir uma função *update*, a sub classe *Meta*, através da seguinte linha de código: *fields = '\_\_all\_\_'*, permite a criação de todos os atributos, e a solicitação dos mesmos, quando o utilizador realiza um pedido.

A função *update* para os vários modelos exhibe sempre uma estrutura semelhante, recebe uma instância, correspondente ao objeto do modelo e os dados validados pelo *Serializer*. No corpo, realiza a atualização dos atributos, consoante o modelo, sendo que no fim, através da linha de código: *instance.save()*, a instância é salva e depois retornada.

### 4.5.3 Desenvolvimento de *APIs*

As *APIs* foram desenvolvidas, consoante a necessidade dos pedidos efetuados pelo cliente, ou realizados através da *REST framework*, como por exemplo, a eliminação de registos, algo que o cliente não faz. Sendo assim temos os seguintes tipos de solicitações:

- *POST* através de *client.post* ou *REST framework*, usado para o envio de novos dados ao servidor
- *PUT* através de *client.put* ou *REST framework*, usado para o envio de dados para atualização
- *GET* através de *client.get* ou *REST framework*, usado para ir buscar dados ao servidor
- *DELETE* através da *REST framework*, usado para eliminar entidades da base de dados

No caso de uma solicitação do tipo *POST*, consoante o modelo, é feita uma configuração dos dados, e no fim através do próprio modelo e da função *create()*, é criado o modelo para registo na base de dados. A função *create()* recebe um dicionário com os dados do modelo para a sua criação. É feito uso do duplo asterisco para indicar que o dicionário pode não conter todos os dados do modelo, sendo que uns serão definidos para *Null*. No fim é feita a serialização dos dados através do respetivo *Serializer* e retornada uma resposta com os dados, cf. Figura 5.1.

```
25 @api_view(['POST'])
26 def createInstitution(request):
27     data = request.data
28
29     institution_data = {
30         'name': data['name']
31     }
32
33     if 'description' in data:
34         institution_data['description'] = data['description']
35
36     institution = InstitutionModel.objects.create(**institution_data)
37     serializer = InstitutionSerializer(institution, many=False)
38
39     return Response(serializer.data)
```

Figura 4.30: Exemplo de um *POST*

No caso de uma solicitação do tipo *PUT*, consoante o modelo e os dados que se pretendem atualizar, é obtido o objeto do modelo, e feita a serialização dos dados da solicitação, sempre de forma parcial, permitindo a atualização de vários dados através de uma só função. A seguir é feita uma validação dos dados, e se estes forem válidos é chamada a função *update()* do respetivo *Serializer*. No fim os dados são gravados através da função *save()* do *Serializer* e retornados numa resposta. Caso os dados sejam inválidos, não é feita a atualização, cf. Figura 5.3.

```
41 @api_view(['PUT'])
42 def updateInstitution(request, pkI):
43     institution = InstitutionModel.objects.get(id_i=pkI)
44
45     serializer = InstitutionSerializer(
46         institution, data=request.data, partial=True
47     )
48
49     if serializer.is_valid():
50         name = request.data.get('name')
51         if name:
52             serializer.update(institution, {'name': name})
53
54         description = request.data.get('description')
55         serializer.update(institution, {'description': description})
56
57         serializer.save()
58     return Response(serializer.data)
```

Figura 4.31: Exemplo de um *PUT*

No caso de uma solicitação do tipo *GET*, foram desenvolvidas para vários objetivos e necessidades, várias funções. A estrutura no entanto é repetida. Inicialmente obtém-se o ou os objetos do modelo, sendo possível a filtração dos objetos pretendidos através das chaves primárias das entidades. É possível a utilização de uma ou mais chaves primárias de várias entidades para obter um ou mais objetos de um certo modelo. Por exemplo, no pedido dos *crash carts*, cf. Figura 5.2, é feita a filtragem através da chave primária *pkI* da entidade *Institution*, porque para uma certa instituição, só faz sentido o acesso aos seus carros. A seguir há obtenção do ou dos objetos, caso seja necessário é feita a serialização dos dados guardados, de modo a retornar uma resposta com os mesmos. Não é feita a serialização, quando se obtém o identificador das entidades.

```
8  @api_view(['GET'])
9  def getCrashCarts(request, pkI):
10     crash_carts = CrashCartModel.objects.filter(institution_id=pkI)
11     serializer = CrashCartSerializer(crash_carts, many=True)
12     return Response(serializer.data)
```

Figura 4.32: Exemplo de um *GET*

No caso de uma solicitação do tipo *DELETE*, através de uma filtração dos objetos pelas chaves primárias, obtém-se o objeto, recorrendo há função *delete()* para eliminar a entidade da base de dados. No final é retornada uma mensagem a confirmar o sucesso da eliminação, cf. Figura 5.4.

```
60  @api_view(['DELETE'])
61  def deleteInstitution(request, pkI):
62     institution = InstitutionModel.objects.get(id_i=pkI)
63     institution.delete()
64     return Response("Delete successfull")
```

Figura 4.33: Exemplo de um *DELETE*

# Capítulo 5

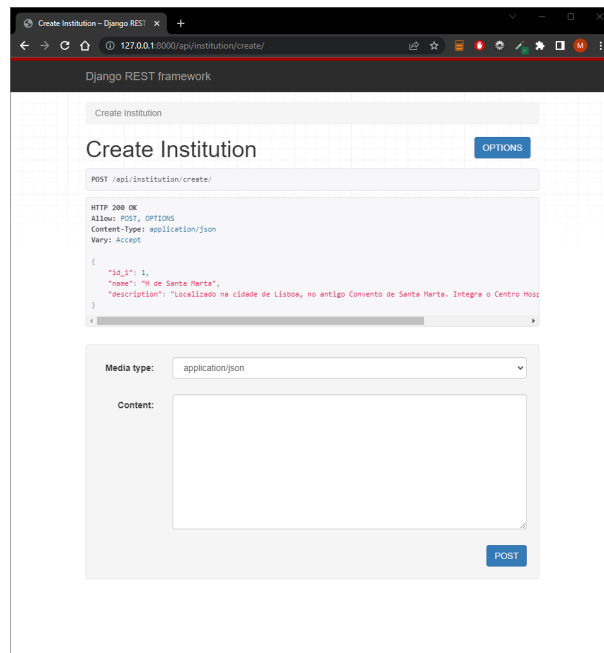
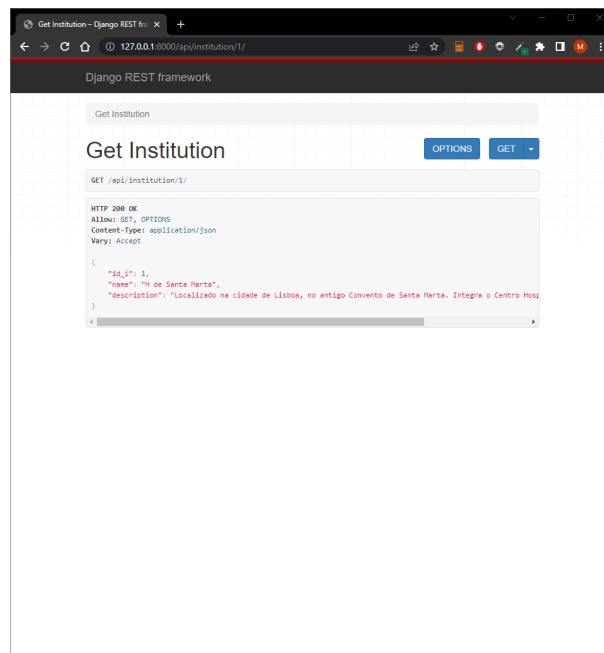
## Validação e Testes

Neste capítulo apresenta-se um conjunto de testes e resultados do sistema. Primeiro testes de solicitações e respostas ao servidor através da *REST framework*, em seguida, testes de registo através da aplicação gráfica desenvolvida com a *framework Flutter* e finalmente testes de leitura dos códigos QR, que envolvem solicitações ao servidor para obter os dados. Não foram efetuados testes pelos clientes, nem validados os resultados devido a escassez no tempo.

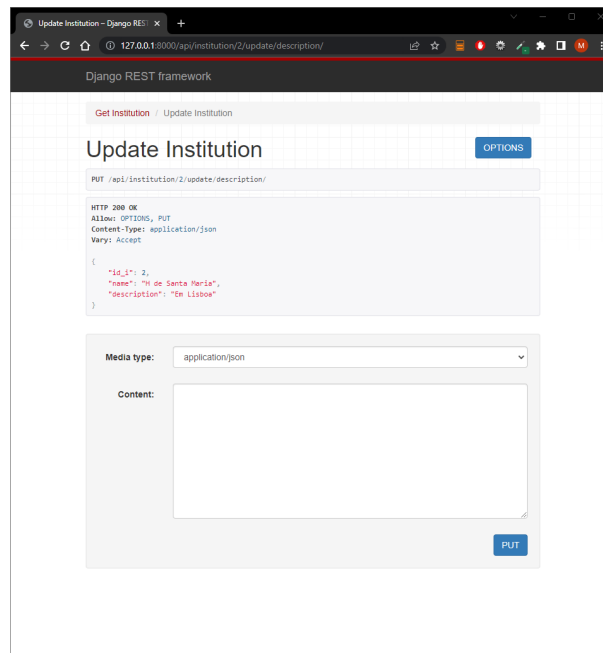
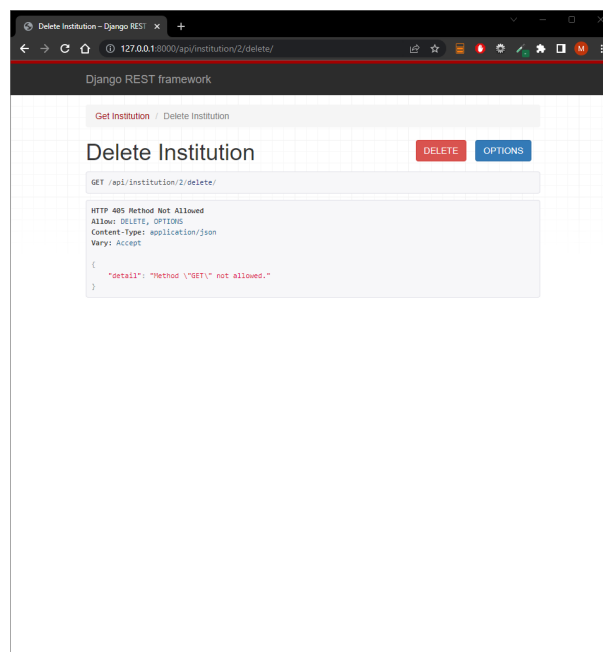
### 5.1 Testes e resultados com a *REST framework*

Antes de efetuar testes com a aplicação gráfica, foram realizados um conjunto de testes a partir da *REST framework* para solicitação e resposta de dados guardados no sistema de gestão de base de dados. Foram feitos testes aos vários métodos - *POST*, *GET*, *PUT* e *DELETE*. A seguir apresenta-se um exemplo dos testes, um para cada método.

A primeira figura, cf. Figura 5.1, apresenta o registo de uma instituição com nome e descrição. A segunda figura, cf. Figura 5.2 apresenta a obtenção dos dados dessa instituição. A terceira figura, cf. Figura 5.3, apresenta a atualização da descrição de uma outra instituição, previamente registada sem a mesma. A quarta e última figura, cf. Figura 5.4 apresenta a eliminação de uma instituição registada.

Figura 5.1: Exemplo do método *POST*Figura 5.2: Exemplo do método *GET*



Figura 5.3: Exemplo do método *PUT*Figura 5.4: Exemplo do método *DELETE*

## 5.2 Testes e resultados dos registos com a aplicação gráfica

Para realizar os testes de registo no servidor a partir da aplicação utilizou-se um telemóvel *Android* e o modo de aplicação *Windows*, no modo de aplicação *Windows* não foram obtidos problemas a executar a aplicação e ao fazer os registos. Não foi possível realizar os testes de leitura com códigos QR, devido há falta de um leitor de códigos. No sistema *Android* não foram obtidos problemas até chegar ao registo do conteúdo de cada *slot*, onde apesar de mostrar a grelha com o tamanho certo e de enviar as informações dos *slots* para o servidor, não ficavam gravadas nos *slots*. Na leitura utilizando o sistema *Android* foi possível visualizar tudo exceto quando se realizava a leitura de um código QR correspondente a uma grelha onde aparecia a informação dos *slots*, mas a secção correspondente à fórmula não era apresentada, cf. Figura 5.17.

Nas imagens em baixo apresentadas, é possível ver os resultados obtidos, cf. Figura 5.5 a Figura 5.13, estão expostos os resultados dos testes de registo através de um sistema *Windows*, cf. Figura 5.5 até à Figura 5.12 onde são apresentadas as interfaces da nossa aplicação, exceto a interface correspondente ao leitor de códigos QR, cf. Figura 5.19. Por fim, cf. Figura 5.14 até à Figura 5.19 estão os testes realizados com o sistema *Android*.

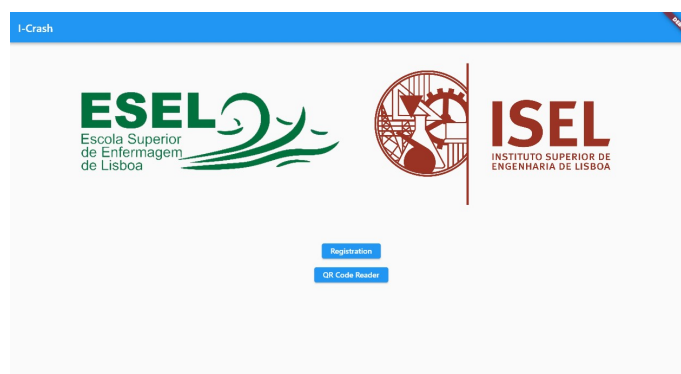
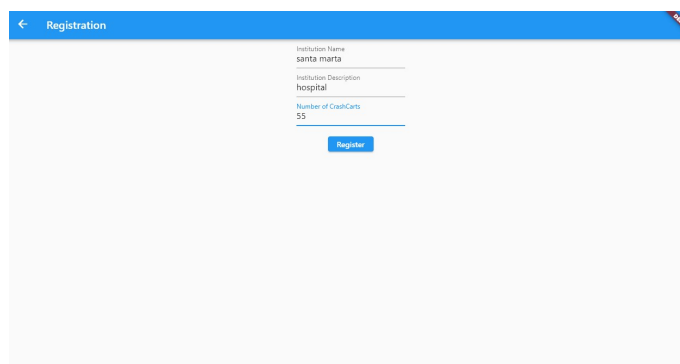


Figura 5.5: Primeira interface



The registration form interface features a blue header with a back arrow and the title 'Registration'. The form fields are as follows:

Field	Value
Institution Name	santa maria
Institution Description	hospital
Number of CrashCarts	55

A blue 'Register' button is positioned below the form fields.

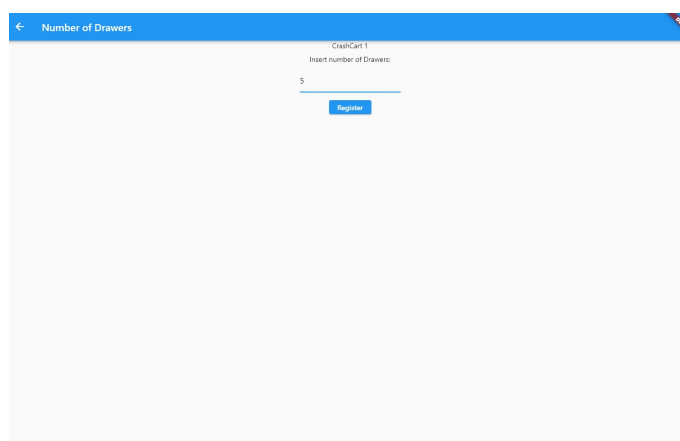
Figura 5.6: Segunda interface



The interface displays a grid of 55 blue buttons, each labeled 'CrashCart' followed by a number from 1 to 55. The buttons are arranged in a 6x10 grid, with the last row containing only 5 buttons (CrashCart 51 to 55).

CrashCart 1	CrashCart 2	CrashCart 3	CrashCart 4	CrashCart 5	CrashCart 6	CrashCart 7	CrashCart 8	CrashCart 9	CrashCart 10
CrashCart 11	CrashCart 12	CrashCart 13	CrashCart 14	CrashCart 15	CrashCart 16	CrashCart 17	CrashCart 18	CrashCart 19	CrashCart 20
CrashCart 21	CrashCart 22	CrashCart 23	CrashCart 24	CrashCart 25	CrashCart 26	CrashCart 27	CrashCart 28	CrashCart 29	CrashCart 30
CrashCart 31	CrashCart 32	CrashCart 33	CrashCart 34	CrashCart 35	CrashCart 36	CrashCart 37	CrashCart 38	CrashCart 39	CrashCart 40
CrashCart 41	CrashCart 42	CrashCart 43	CrashCart 44	CrashCart 45	CrashCart 46	CrashCart 47	CrashCart 48	CrashCart 49	CrashCart 50
CrashCart 51	CrashCart 52	CrashCart 53	CrashCart 54	CrashCart 55					

Figura 5.7: Terceira interface



The 'Number of Drawers' form interface has a blue header with a back arrow and the title 'Number of Drawers'. It includes the following fields:

Field	Value
CrashCart 1	
Insert number of Drawers	5

A blue 'Register' button is located at the bottom of the form.

Figura 5.8: Quarta interface

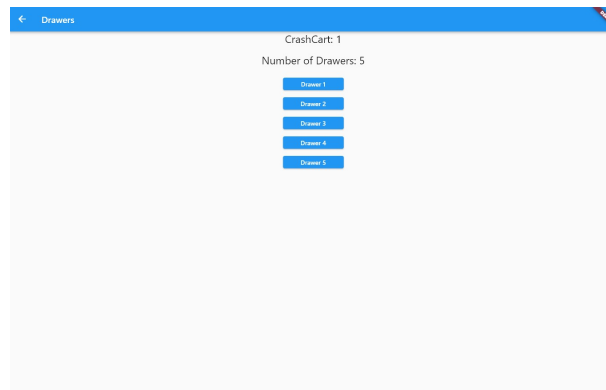


Figura 5.9: Quinta interface

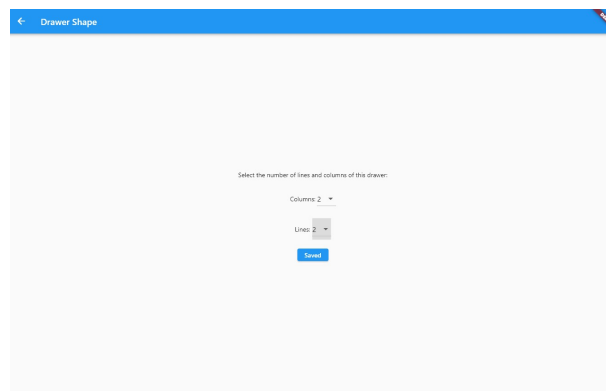


Figura 5.10: Sexta interface

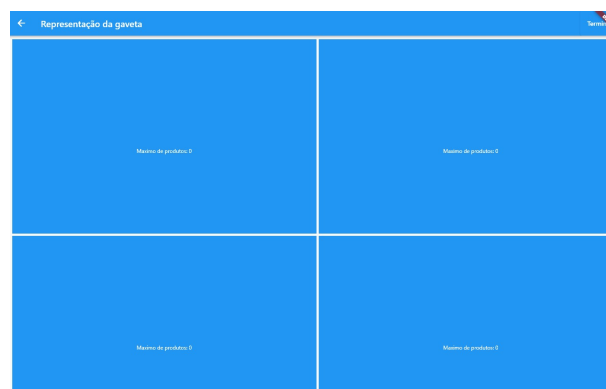


Figura 5.11: Sétima interface

The screenshot shows a form titled 'Detalhes do Produto'. It contains four input fields: 'Nome do Produto' with the value 'benuron', 'Volume/peço' with the value '200mg', 'Forma de aplicação' with the value 'comprimido', and 'Máximo do Produto' with the value '06'. At the bottom right of the form are two buttons: 'Cancelar' and 'Salvar'.

Figura 5.12: Oitava interface

The screenshot shows a screen titled 'Representação da gaveta'. It displays three product cards arranged in a grid. The top row has two cards: 'benuron 200mg comprimido Máximo do produto: 6' and 'brufen 200mg comprimido Máximo do produto: 7'. The bottom row has one card: 'strygal 500mg pastilha Máximo do produto: 6'. The background is blue.

Figura 5.13: Sétima interface preenchida

Na Figura 5.13 é possível observar a sétima interface preenchida com informações dadas a partir da oitava interface, também é possível observar que foi realizado um *merge* horizontal nos dois últimos blocos, todas estas informações são guardadas no sistema de gestão de base de dados a partir do servidor.

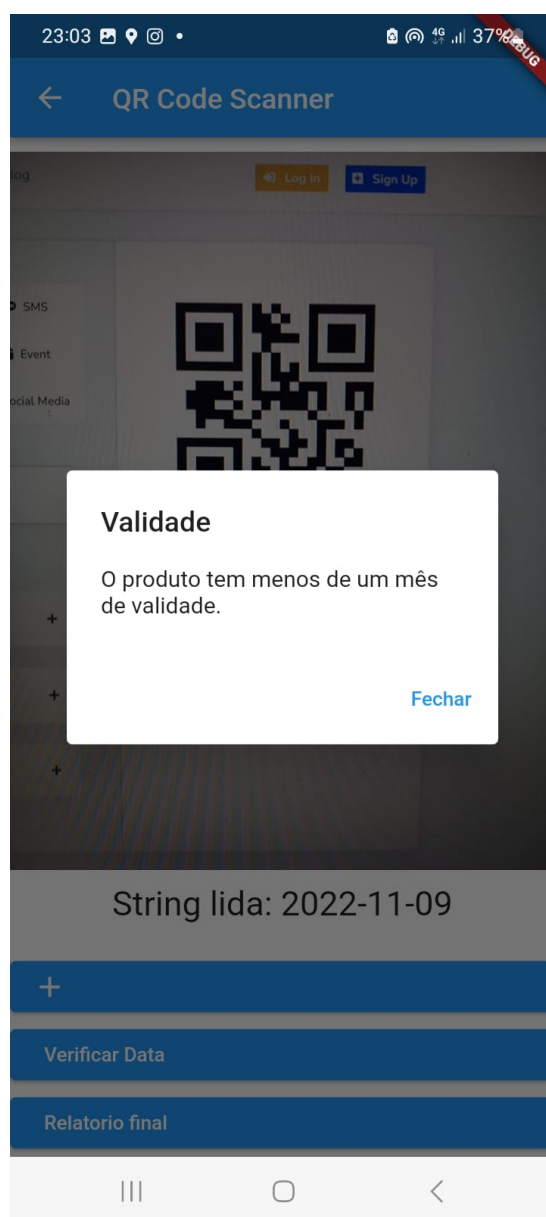


Figura 5.14: Leitura e validação de uma data inferior a um mês

Na figura 5.14 é possível observar a leitura de um código QR a partir da nossa aplicação. Ao ser pressionado o botão "Verificar Data", e ao ler a *string* "2022-11-09", é feita uma comparação com a data "atual", data do dia "2023-09-06", devolvendo a mensagem que o consumível clínico tem menos de um mês de validade.

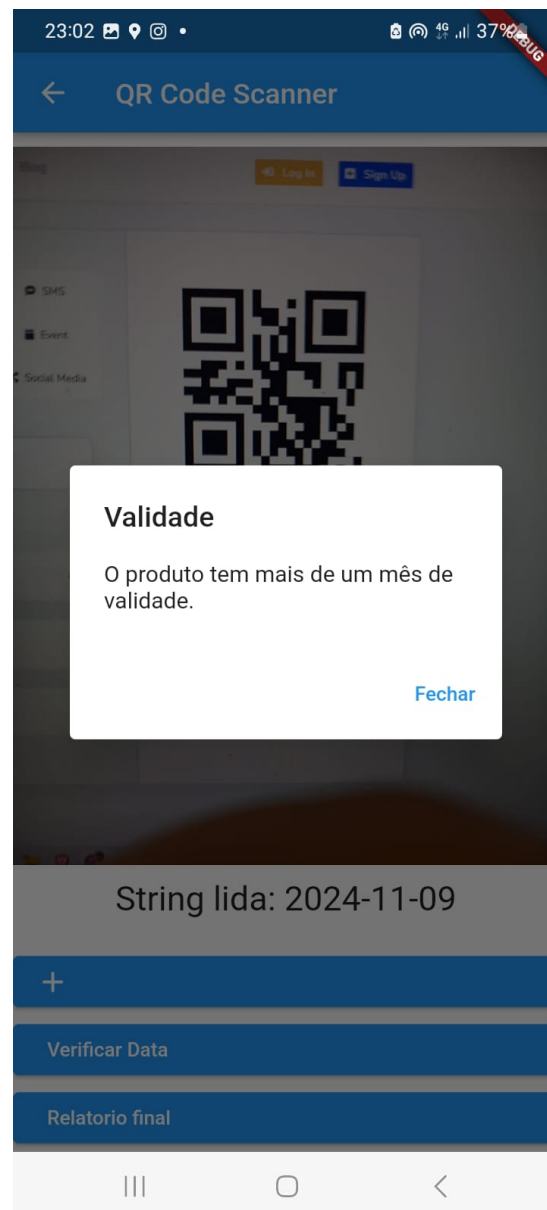


Figura 5.15: Leitura e validação de uma data superior a um mês

Na figura 5.15 é possível observar a leitura de um código QR a partir da nossa aplicação. Ao ser pressionado o botão "Verificar Data", e ao ler a *string* "2024-11-09", é feita uma comparação com a data "atual", data do dia "2023-09-06", devolvendo a mensagem que o consumível clínico tem mais de um mês de validade.

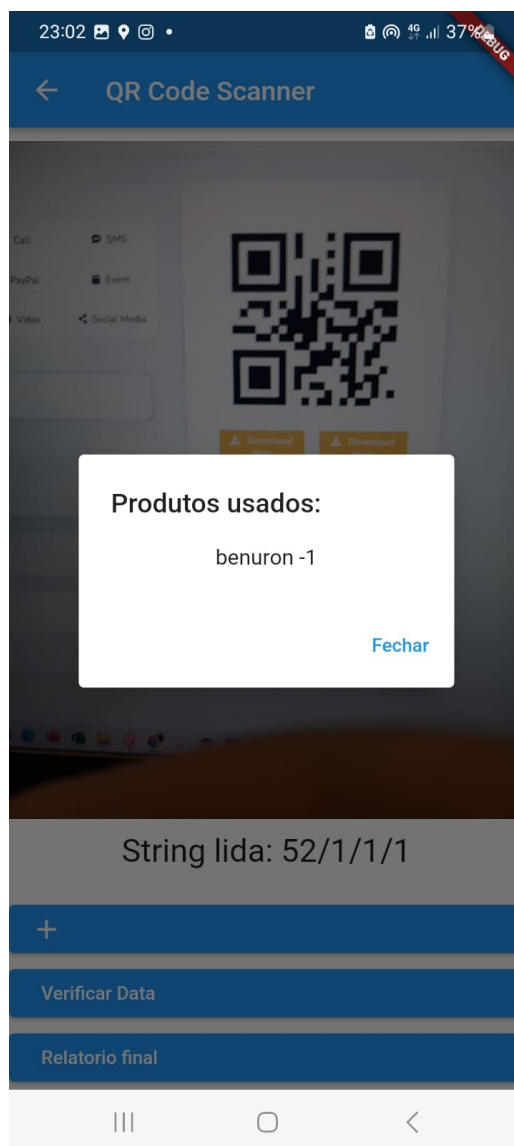


Figura 5.16: Relatório dos *slots* e do número de leituras

Na Figura 5.16 é possível observar a leitura de um código QR a partir da nossa aplicação. Ao ser pressionado o botão "+" e ler a *string* "52/1/1/1", que corresponde ao primeiro *slot* da primeira gaveta do primeiro carro da instituição número cinquenta e dois, e depois ao pressionar o botão "Relatório final", aparece a mensagem que contém o nome do consumível clínico que está presente no *slot* lido, e à sua direita o número de consumíveis clínicos usados.



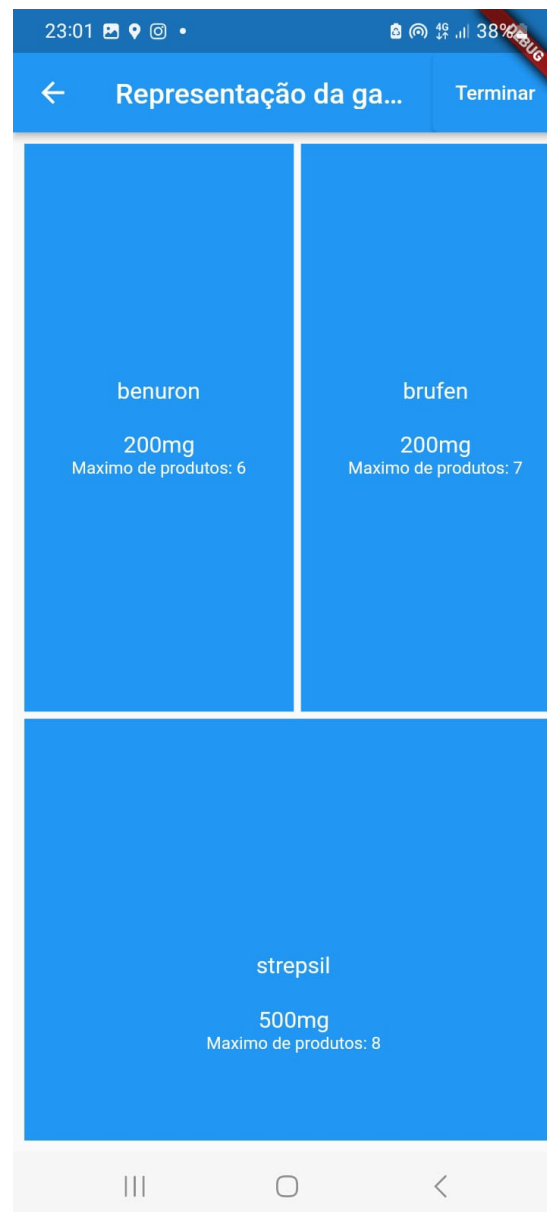


Figura 5.17: Leitura de um código QR de uma gaveta

Na Figura 5.17 é possível observar o resultado da leitura do código QR "52/1/1", que corresponde há primeira gaveta do primeiro carro da instituição número cinquenta e dois. Está aqui presente a representação da gaveta. Tal como foi falado em cima, também é possível observar o *bug* encontrado ao executar a aplicação no *Android*.

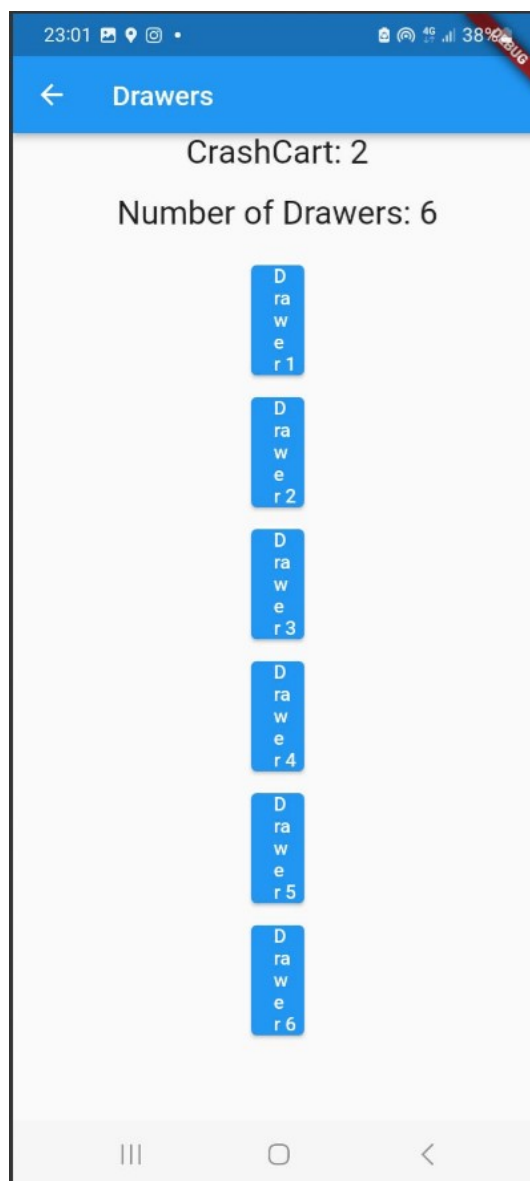


Figura 5.18: Leitura de um código QR de um carro

Na Figura 5.18 é possível observar o resultado da leitura do código QR "52/2", que corresponde ao primeiro carro da instituição número cinquenta e dois.

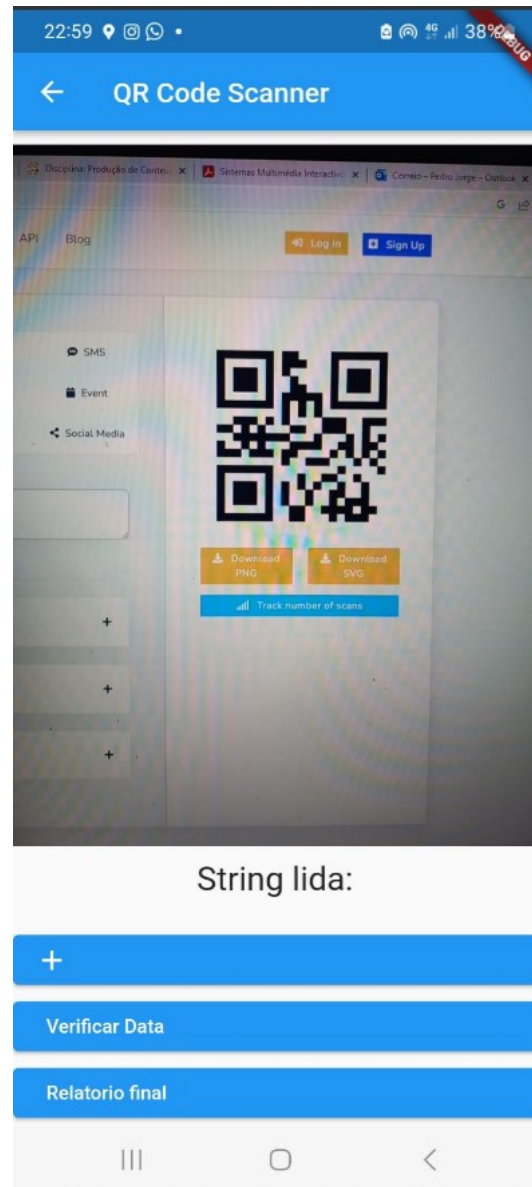


Figura 5.19: Nona interface



## Capítulo 6

# Conclusões e Trabalho Futuro

Com este projeto, ganhamos conhecimento ao aprender duas novas *frameworks*, o *Django* e o *Flutter*. No caso do *Django*, a linguagem de programação é o *Python*, a qual estamos habituados devido ao uso em várias unidades curriculares, no entanto, aperfeiçoamos o conhecimento com o uso de *wrappers*. No caso do *Flutter*, aprendemos uma linguagem nova, o *Dart*, muito útil pelo facto de permitir o desenvolvimento de código uma vez com a capacidade de exportação e conversão, através do *Flutter*, para outras linguagens de programação, consoante o sistema operativo. A capacidade de desenvolver um sistema híbrido remove muito tempo na conversão e escrita de código para múltiplos sistemas operativos. A aprendizagem de sistemas híbridos conclui-se assim muito útil para o nosso futuro. Não trabalhamos muito com o sistema de gestão de base de dados *PostgreSQL*, visto que quase tudo foi realizado através do servidor, mas permitiu a aplicação de conhecimentos no desenvolvimento de um modelo de dados. Também aprendemos a lidar com clientes que não tenham conhecimento em informática, a partir de múltiplas reuniões onde fomos tentando arranjar e apresentar soluções aos problemas que nos foram colocados.

Apesar de termos uma aplicação funcional, podem ser encontrados alguns *bugs* como o *merge* vertical que visualmente não está desejável ou como o *bug* aprestado na figura 5.17, estes *bugs* não foram corrigidos devido ao facto de termos obtido muitos contratempos, como por exemplo, na criação do servidor que demorou mais do que o esperado, pois tínhamos muito pouco conhecimento desta área o que acabou por causar grandes atrasos no nosso projeto.

Como trabalho futuro propomos a melhoria do aspeto da grelha, correção de *bugs* e também a criação de um modo em que se utilize o leitor de códigos e o sistema de gestão de base de dados, para fazer a leitura de datas de validade, guardando a informação da leitura para emitir um relatório. Com a leitura de datas de validade propomos também o estudo e desenvolvimento de uma biblioteca que descodifique as várias normas de escrita de códigos exigidas pela união europeia no que consta a produtos farmacêuticos.

Outra melhoria é a adição de um administrador ao servidor e exportá-lo para a *cloud*, assim o acesso não está restrito a uma *Local Area Network* - *LAN*, permitindo que o sistema seja acedido pelo país todo e com um maior desenvolvimento, controlo e proteção de dados, eventualmente, por vários países pelo mundo.

Outra melhoria proposta é a independência da câmara do dispositivo móvel, através do uso de um leitor de códigos físico. Esta melhoria, permite também que se possa ler códigos de barra através do sensor infravermelho do leitor de códigos, algo que as câmaras dos dispositivos móveis ainda não têm. Também permite que a leitura seja mais rápida.

# Bibliografia

- [1] [Rafael Santos, 2015] Instituto Superior de Engenharia de Lisboa (ISEL)  
- Dissertation and Thesis LaTeX Template Overleaf
- [2] [Viver Melhor] Carro de emergência, <https://vivermelhor.pt/pt/mobiliario-clinico-equipamento-hospitalar-acessorios-mobiliario-clinico/p/467-carro-de-emergencia-em-inox-e-tampo-em-abs-c-4-rodas.html?gclid=Cj0KCQjw2qKmBhCfARIsAFy8buIsJhlM4lCnHj9RnerbkIVRwDrIpW67vfKqEBGNw2JDh0wcB>
- [3] [Copigés] Leitor de códigos, <https://copiges.com/produtos-copiges/captura-dados/leitores-codigo-barras/zebra-ls2208/>
- [4] [Ambitur, 2019] Caixas de autopagamento, <https://www.ambitur.pt/grupo-sonae-avanca-com-lojas-continente-bom-dia-500-a-pensar-nos-turistas/>
- [5] [Paulo Trigo, Helder Bastos, 2023] Transparentes, Moodle
- [6] [JGraph] *Drawio*, <https://app.diagrams.net/>
- [7] [Mouser] Sensor de pressão, <https://pt.mouser.com/new/dfrobot/dfrobot-rp-thin-film-sensors/>
- [8] [Competitive Card Solutions] Sensor *RFID*, <https://ccs.com.ph/products/ccs-uhf-slrec-uhf-rfid-sticker-label-840-960-mhz-frequency>
- [9] [Flutter] Documentação de instalação, <https://docs.flutter.dev/get-started/install/windows>
- [10] [PostgreSQL] Documentação de instalação, <https://www.postgresql.org/download/windows/>

- [11] [Programming with Mosh, 2021] <https://www.youtube.com/watch?v=rHux0gMZ3Eg>
- [12] [Dennis Ivy, 2021] <https://www.youtube.com/watch?v=VnztChBw70g>
- [13] [ERDPlus, 2015] <https://erdplus.com/>