

Universidade Fernando Pessoa

Mestrado em Computação Móvel

Projeto de Aplicações Móveis

João Reis - 37190

Sistema de Informação para um Stand

Porto

abril/2021



Universidade Fernando Pessoa

Praça 9 de Abril, 349

P-4249-004 Porto

Tel. +351-22550.82.70

Fax. +351-22550.82.69

geral@ufp.pt

Imagens

- ❖ Figura 1 - Casos de Uso.
- ❖ Figura 2 - Requisitos Funcionais
- ❖ Figura 3 - Modelação do Projeto
- ❖ Figura 4 - Diagrama de Classes
- ❖ Figura 5 - Login
- ❖ Figura 6 - Main Menu
- ❖ Figura 7 - Car List
- ❖ Figura 8 - Car Details
- ❖ Figura 9 - Parts
- ❖ Figura 10 - Sales
- ❖ Figura 11 - Clients
- ❖ Figura 12 - Partners
- ❖ Figura 13 - Logout
- ❖ Figura 14 - Estrutura de pastas do projeto
- ❖ Figura 15 - Definição da classe SearchPlate
- ❖ Figura 16 - Lógica da classe SearchPlate
- ❖ Figura 17 - Resultado do teste de integração
- ❖ Figura 18 - Conclusão dos Requisitos Funcionais

Índice

Introdução	3
Análise e Design	4
Protótipo	8
Implementação	15
Avaliação	18
Conclusão	19
Anexos	20

I. Introdução

Este projeto consiste na concretização de um sistema de informação para um stand automóvel. Com isto, o grande foco deste projeto passa por informatizar e assistir todo o processo da empresa em questão, de forma a ter um maior controlo do processo de compra, reparação e venda e auxiliar todas as entidades envolvidas. Assim sendo, este projeto tem alguns requisitos funcionais base, descritos no capítulo seguinte deste relatório e mais outros opcionais, dependendo do desempenho e desenvolvimento do presente projeto.

II. Análise e Design

Conhecendo o contexto deste projeto, foi inicialmente necessário a definição de requisitos tanto funcionais quanto não funcionais. Começando pelos requisitos funcionais, os objetivos inicialmente propostos são apresentados como forma de diagrama de casos de uso pela figura 1, e sobre a forma de uma tabela na figura 2.

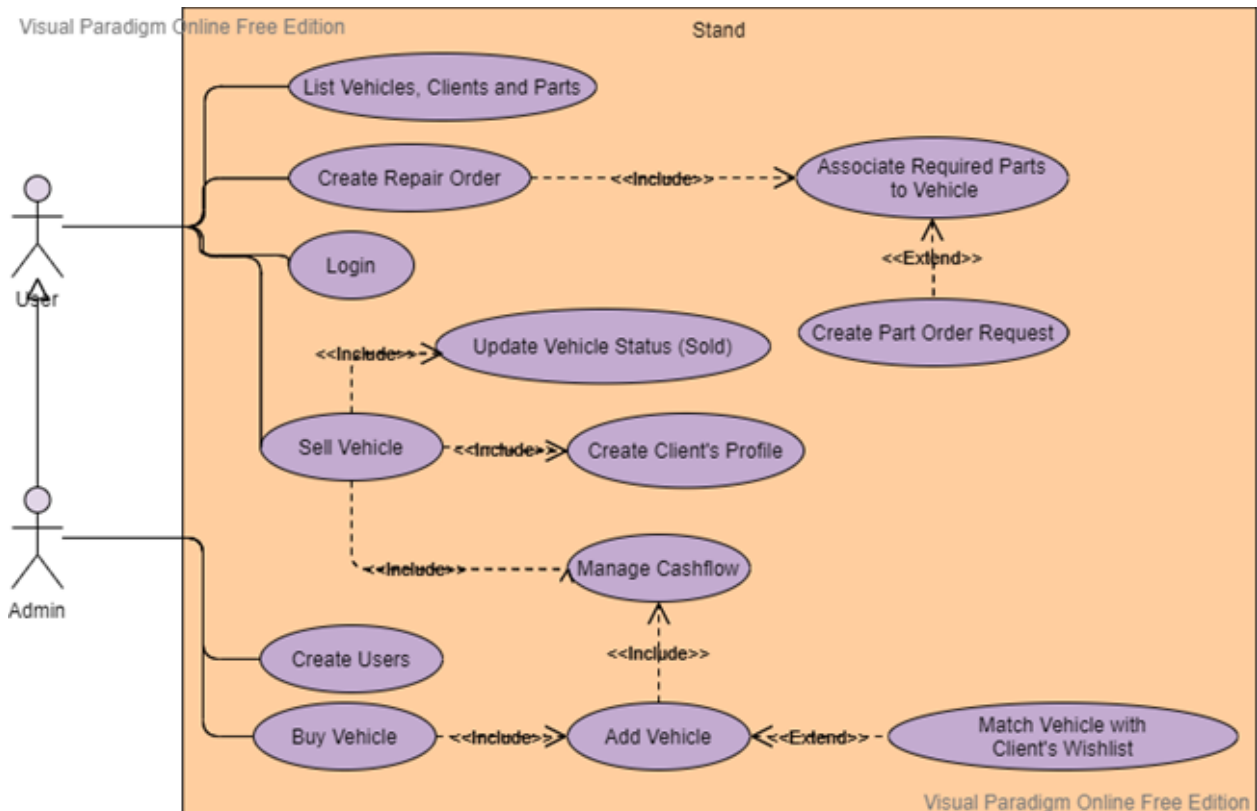


Figura 1 - Casos de Uso

Requisitos Funcionais
1 - Login
2 - Listar veículo
3 - Filtrar veículo
4 - Listar vendas
5 - Filtrar vendas
6 - Listar clientes

7 - Listar peças
8 - Listar parceiros
9 - Criar ordem de reparação
10 - Vender Veículo
11 - Criar users (apenas admin)
12 - Comprar veículo

Figura 2 - Requisitos Funcionais

Para além dos requisitos funcionais, existem outros não funcionais, entre os quais:

- O sistema deve facultar segurança ao utilizador;
- O sistema deve ser rápido;
- O sistema deve apresentar uma interface simples e intuitiva.

Com os requisitos definidos, foi necessário definir a modelação do projeto. O modelo escolhido consiste numa aplicação móvel flutter que estabelece uma conexão a um servidor Java SpringBoot para obter todos os dados necessários, tendo este servidor uma ligação a uma base de dados MongoDB. Este modelo é apresentado pela figura 3.

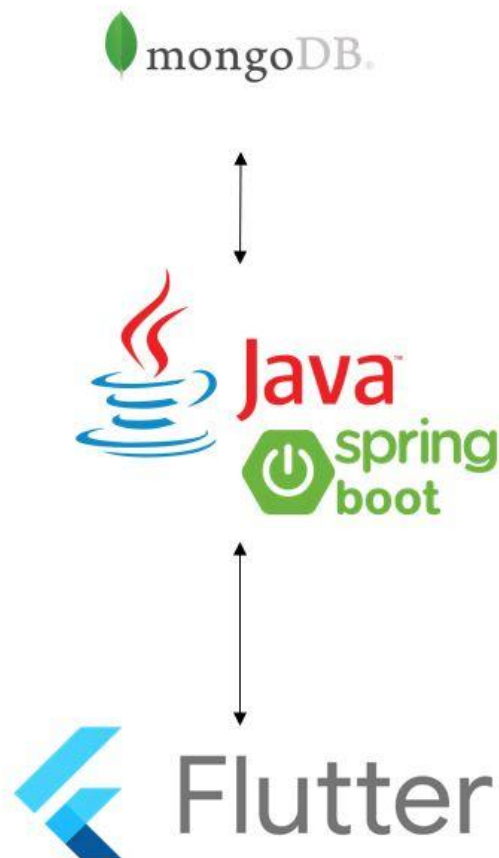


Figura 3 - Modelação do Projeto

De seguida, procedeu-se à modelação das classes necessárias ao projeto. O projeto apresenta um diagrama de classes relativamente simples, com cerca de 5 classes distintas (Entity, Sale, Car, User e Part) que se relacionam consoante o diagrama de classes apresentado na figura 4.

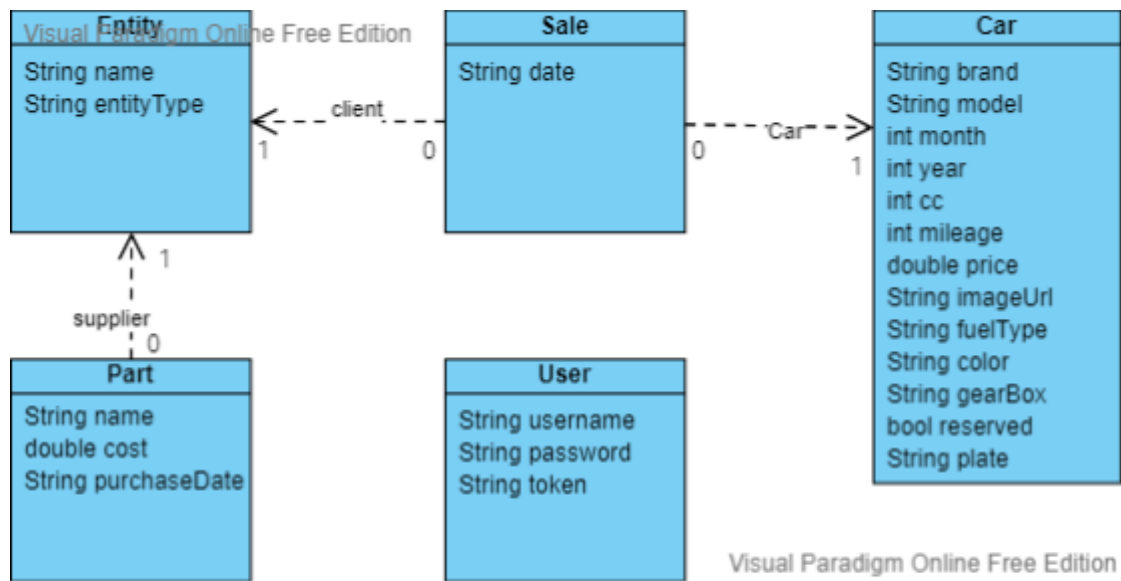


Figura 4 - Diagrama de Classes

III. Protótipo

Podemos observar pelas figuras 5 a 13 alguns wireframes de algumas das funcionalidades da aplicação, não estando apresentado todas, uma vez que, por uma questão de constrangimento de tempo, não foi possível implementar todas as funcionalidades desejadas no projeto. Com isto, estão apresentadas as wireframes para os seguintes ecrãs:

- Login;
- Main Menu;
- Car List;
- Car Details;
- Parts;
- Sales;
- Clients;
- Partners;
- Logout.



Figura 5 - Login

Figura 6 - Main Menu

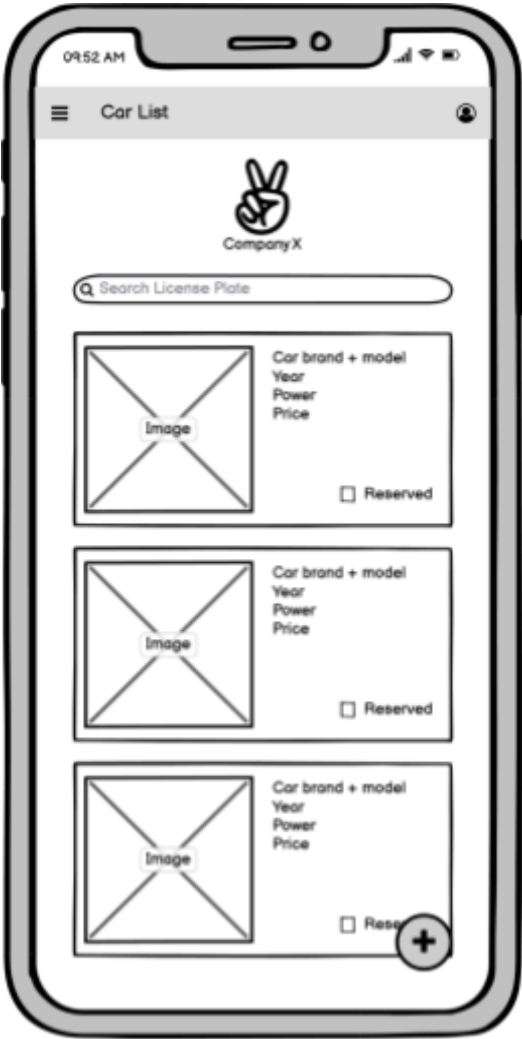


Figura 7 - Car List

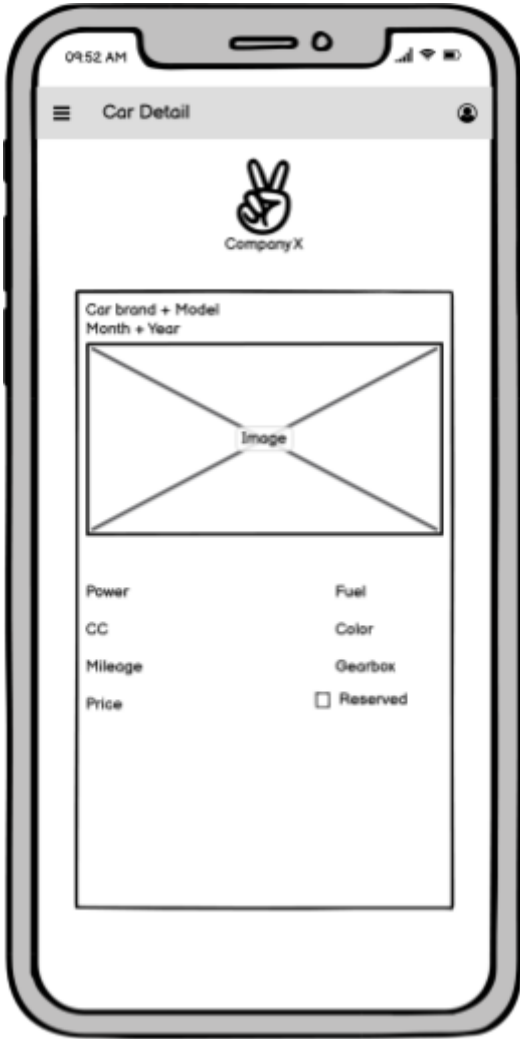


Figura 8 - Car Details

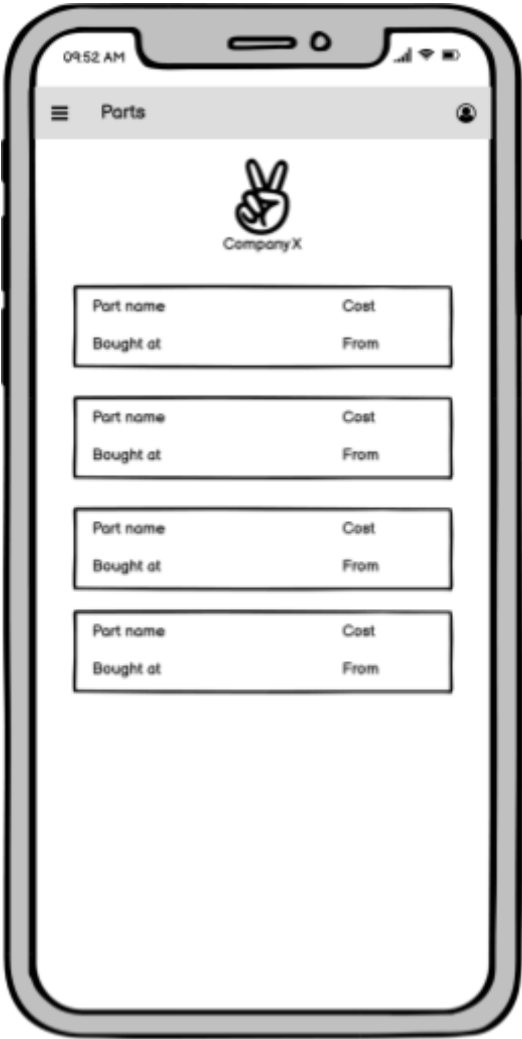


Figura 9 - Parts

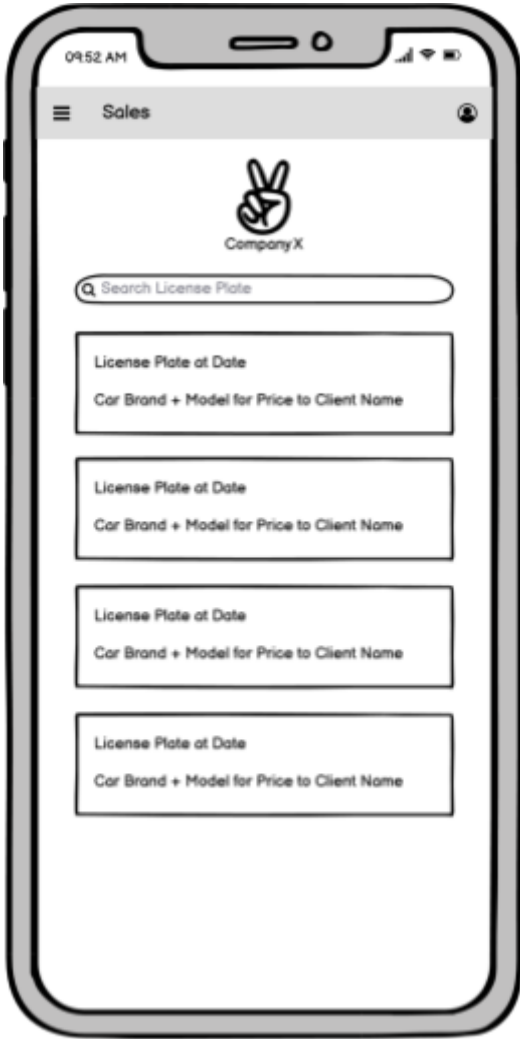


Figura 10 - Sales

09:52 AM

≡ Clients


Company X

Mr(s). Client

Mr(s). Client

Mr(s). Client


Mr(s). Client

Mr(s). Client

Figura 11 - Clients

09:52 AM

≡ Partners


Company X

Partner Name
Partner Type

Partner Name
Partner Type

Partner Name
Partner Type

Partner Name
Partner Type

Partner Name
Partner Type

Figura 12 - Partners

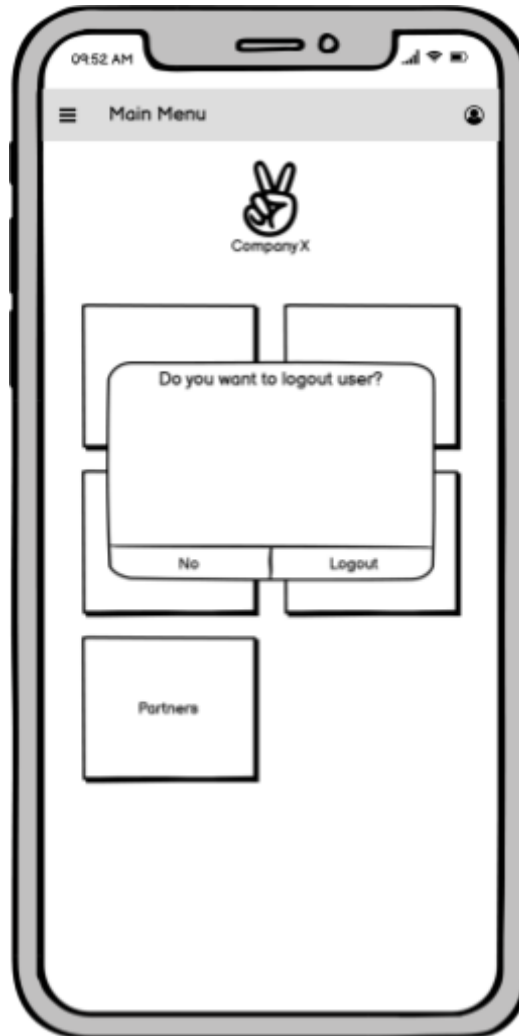


Figura 13 - Logout

É importante notar que, o Widget Drawer presente nas Wireframes acima, não se encontra no projeto. É também importante salientar como é possível atingir cada uma frames:

- Login - Frame raiz
- Main Menu - Após o login do utilizador ou através do botão back de um dos seus menus filhos;
- Car List - Pressionar o menu “Cars” ou através do botão back de Car Details;
- Car Details - Pressionar um carro no menu “Cars”;
- Parts - Pressionar o menu “Parts”;
- Sales - Pressionar o menu “Sales/History”;
- Clients - Pressionar o menu “Clients”;

- Partners - Pressionar o menu “Partners”;
- Logout - Pressionar o botão no canto superior direito em qualquer uma das frames exceto no login

IV. Implementação

O projeto é composto por uma estrutura de pastas bem definida. Na root do projeto existe uma package lib onde todo o código de implementação pode ser encontrado. Esta package, por sua vez, é dividida entre as packages components (onde todos os componentes criados estão guardados), models (onde estão definidos os modelos das classes conforme o diagrama de classes) e pages (que contém uma lista de packages dividida por ecrãs com o seu código correspondente). Dentro da package models pode-se ainda encontrar uma package generated na qual se encontram alguns ficheiros autogerados explicados abaixo neste relatório. Voltando à root do projeto, temos ainda as packages test_driver (necessária para a implementação de testes na aplicação) e integration_test (onde se encontram os testes de integração).

```
/lib
  /components
  /models
    /generated
  /pages
    /cars
    /clients
    /login
    /partners
    /parts
    /sales
/test_driver
/integration_test
```

Figura 14 - Estrutura de pastas do projeto

Conforme referido anteriormente, temos a package generated, package esta que é responsável pela geração automática de métodos to e from Json para cada um dos modelos definidos. Estes métodos são guardados em ficheiros “.g.dart”. Esta geração é possível graças à biblioteca “JsonSerializable”, e permite ao autor deste projeto remover algum boilerplate nos métodos anteriormente referidos.

Quanto à implementação do código da aplicação, o autor teve uma especial atenção à criação e reutilização de componentes customizáveis, para que o código se tornasse mais curto, limpo, fácil

de ler e de manter. Um exemplo deste cuidado é com o componente `searchPlate`, que se trata de um filtro com uma `textbox`, onde uma matrícula pode ser inserida para que a lista do componente pai possa ser filtrada. No exemplo abaixo verifica-se que, para que este filtro funcionasse tanto para listas de carros como de vendas, foi utilizado o tipo `dynamic`, que permite ao componente receber dados de qualquer tipo. Uma vez recebendo os dados, o código é capaz de reconhecer o tratamento adequado para os mesmos, consoante a origem desses dados.

```
class SearchPlate extends StatefulWidget {  
  
  final String _filter;  
  final List<dynamic> _list;  
  final Function (List<dynamic>) _callback;  
  
  SearchPlate(this._filter, this._list, this._callback);  
  
  @override  
  _SearchPlateState createState() => _SearchPlateState();  
  
}
```

Figura 15 - Definição da classe `SearchPlate`

```
class _SearchPlateState extends State<SearchPlate> {  
  @override  
  Widget build(BuildContext context) {  
    return Padding(  
      padding: EdgeInsets.fromLTRB(16.0, 0.0, 16.0, 16.0),  
      child: TextField(  
        key: Key("filter"),  
        onChanged: (value) => widget._filter == "car" ? _runFilterCar(value) : _runFilterSale(value),  
        decoration: const InputDecoration(  
          labelText: 'Search license plate',  
          suffixIcon: Icon(Icons.search)), // InputDecoration  
        ), // TextField  
      ); // Padding  
    )  
  }  
  
  void _runFilterCar(String keyword) {  
    List<Car> results = [];  
    results = keyword.isEmpty  
      ? widget._list  
      : widget._list  
        .where((c) => c.plate.toLowerCase().contains(keyword.toLowerCase()))  
        .toList();  
  
    widget._callback(results);  
  }  
  
  void _runFilterSale(String keyword) {  
    List<Sale> results = [];  
    results = keyword.isEmpty  
      ? widget._list  
      : widget._list  
        .where((s) => s.car.plate.toLowerCase().contains(keyword.toLowerCase()))  
        .toList();  
  
    widget._callback(results);  
  }  
}
```

Figura 16 - Lógica da classe SearchPlate

V. Avaliação

Quanto à testagem da aplicação, apenas um teste de integração foi criado por motivos de constrangimento de tempo. Isto levou o autor a focar-se na testagem de um dos principais requisitos da aplicação, a pesquisa filtrada de carros. Este teste é encontrado no ficheiro `app_test.dart` dentro da pasta `integration_test`. Aqui, todo o procedimento desde o login da aplicação, até à listagem e filtragem de carros é testada, terminando o teste com o logout do utilizador. A execução do teste foi bem sucedida.



Figura 17 - Resultado do teste de integração

VI. Conclusão

Chegando ao final deste projeto, alguns dos objetivos iniciais propostos não foram alcançados infelizmente. Para além da conclusão destes objetivos, algumas possíveis melhorias detetadas pelo autor deste relatório são a cobertura total da aplicação com testes unitários e de integração, assim como a inclusão de algum tipo de persistência na aplicação. Abaixo pode ser observado todos os objetivos propostos, assim como a sua conclusão.

Requisitos Funcionais	Obtido
1 - Login	Sim
2 - Listar veículo	Sim
3 - Filtrar veículo	Sim
4 - Listar vendas	Sim
5 - Filtrar vendas	Sim
6 - Listar clientes	Sim
7 - Listar peças	Sim
8 - Listar parceiros	Sim
9 - Criar ordem de reparação	Não
10 - Vender Veículo	Não
11 - Criar users (apenas admin)	Não
12 - Comprar veículo	Não

Figura 18 - Conclusão dos Requisitos Funcionais

VII. Anexos

Github - <https://github.com/cebada/stand>