Instituto Politécnico de Viseu
Escola Superior de Tecnologia e Gestão de Viseu
Departamento de Informática

Relatório de Projeto/Estágio

Projeto aplicação IOS
Paulos-Auto

Orientador ESTGV:

- Francisco Ferreira Francisco

Orientador Softinsa:

- José Carlos Dias

Realizado por:

- Gonçalo Neves - 16812

Viseu, 2020

Instituto Politécnico de Viseu
Escola Superior de Tecnologia e Gestão de Viseu
Departamento de Informática

Relatório de Projeto
Curso de Engenharia Informática

Projeto aplicação IOS
Paulos-Auto

Ano Letivo 2019/2020

Viseu, 2020

ÍNDICE

Índice de Figuras

Índice de tabelas

Lista de Abreviaturas

UC – Unidade Curricular

ESTGV – Escola Secundária de Tecnologia e Gestão de Viseu

IPV – Instituto Politécnico de Viseu

IDE – Integrated Development Environment

JSON – JavaScript Object Notation

API – Application Programming Interface

iOS – iPhone Operating System

IBM – Internation Business Machines

HTTPS – Computer Hypertext Transport Protocol Secure

SMP – Swift Package Manage

1. Resumo

O presente relatório descreve o desenvolvimento de uma aplicação iOS para o cliente PaulosAuto, no âmbito do projeto de estágio realizado na Softinsa. O projeto desenvolvido parte de uma necessidade da PaulosAuto de simplificar e aumentar a eficiência do processo de registo de utilização dos equipamentos alugados aos seus clientes.

A aplicação tem como funcionalidades: a) listagem de equipamentos; b) consulta de faturas pendentes; c) consulta de reporte de problemas submetidos; d) registo de utilização de um equipamento; e) registo de problema associado a um equipamento.

Na conclusão do projeto foram apontadas as dificuldades presentes no desenvolvimento da aplicação bem como o trabalho futuro no melhoramento do produto elaborado. Contudo, e mesmo com as complicações encontradas, a aplicação é considerada um sucesso visto terem sido implementadas todas as funcionalidades esperadas.

2. Introdução

O presente relatório surge no âmbito da Unidade Curricular de Projeto do Curso de Licenciatura em Engenharia Informática e tem como objetivo o desenvolvimento de um produto enquadrado numa realidade empresarial. O processo de escolha do projeto a realizar foi efetuado através da proposta do aluno a um ou vários projetos cuja especificação apresenta-se definida pela empresa responsável, e cujo processo de verificação e aceitação foi desempenhado pelos docentes da UC.

Foi desta forma efetuada uma proposta ao projeto que consiste no desenvolvimento de uma aplicação iOS para o cliente PaulosAuto, projeto pertencente à Softinsa, a qual, ao fim de um processo de seleção através da realização de entrevistas foi aceite.

2.1. Contextualização

A Softinsa (Softinsa, 2020), uma subsidiária da IBM, é especializada em serviços de gestão e desenvolvimento de aplicações e infraestrutura. Com 22 anos de história e experiência no mercado português, conta atualmente com uma equipa de mais de 1000 profissionais, os locais em que se localiza passam por Lisboa, Tomar, Viseu e Fundão. A sua oferta abrangente inclui serviços e soluções de Enterprise Applications, Business Analytics, Application Management Services, Mobility, Smarter Cities, End User Computing, Infrastructures & Managed Services e Site & Facilities.

A PaulosAuto (PaulosAuto, 2020), fundada em 1988 por Germano Costa Paulo, com sede em Viseu, apresenta uma comercialização de toda a gama de máquinas de movimentação de terras, pavimentação e sistemas de energia da marca lider mundial de equipamentos Caterpillar e, no sector industrial, comercializa uma vasta gama de equipamentos de movimentação de cargas representando a Mitsubishi Forklift Trucks nos mercados de Viseu e Guarda. É neste contexto que a PaulosAuto apresenta um serviço de aluguer de equipamento pesado, sendo a utilização destes equipamentos contabilizada em horas.

O projeto a desenvolver parte desta forma de uma necessidade do cliente de simplificar e aumentar a eficiência do processo de registo de utilização dos equipamentos alugados por parte dos seus clientes. Sendo que o processo utilizado até então passa pelo registo manual das horas dos equipamentos de cada cliente. Desta forma, a PaulosAuto pretende a criação de plataformas que assegurem a automatização deste processo, sendo estas:

- Uma aplicação Web;
- Uma aplicação Android;
- Uma aplicação iOS.

Com o desenvolvimento destas plataformas em vista a equipa constituída passa por:

- Um membro responsável pela criação da aplicação Web;
- Um membro responsável pela criação da aplicação Android;
- Um membro responsável pela criação da aplicação iOS;
- Um membro responsável pela criação do backend, fornecendo uma API para as aplicações utilizarem.

2.2. Objetivos e motivação

É estabelecido como objetivo a criação de uma aplicação de qualidade empresarial que represente um meio fácil e intuitivo de registo de utilização de equipamentos sem o inconveniente da realização do mesmo manualmente. Procurou-se adicionalmente expandir a ideia do cliente para novas funcionalidades, pelo que, em concordância com o cliente foram estabelecidas as seguintes funcionalidades principais para a aplicação a desenvolver:

- Listagem de equipamentos;
- Registo de utilização de equipamentos;
- Reporte de eventuais problemas associado a um equipamento;
- Consulta de faturas pendentes;
- Consulta de reportes de problemas efetuados.

2.3. Estrutura do documento

De forma a esclarecer todos os passos desde planeamento ao desenvolvimento do produto foi estabelecida a seguinte estrutura:

- Estado da Arte, onde são descritos aspetos teóricos que se revelaram essenciais na tomada de várias decisões que impactaram o desenvolvimento do projeto, sendo os quais: Tipos de Aplicação, Linguagem de programação, Escolha de IDE e Escolha do gestor de dependências;
- Projeto, onde é descrito em pormenor a realização do projeto desde o planeamento até à aplicação final, abordando temas como: Planeamento e metodologia, Plataformas e softwares utilizados, Contato com o cliente, Prototipagem, API, Ponderações de desenvolvimento, Estrutura e código, Fase de testes e concluindo com a abordagem relativa ao estado final do produto;
- Conclusões e Trabalho Futuro, onde são descritos os elementos a melhorar no produto criado abordando o trabalho futuro no desenvolvimento dessas melhorias, bem como, as conclusões relativas ao projeto desenvolvido e sucesso ou não da implementação do mesmo.

3. Estado da Arte

Neste capítulo encontram-se descritos os aspetos teóricos relevantes na tomada de decisões essenciais ao desenvolvimento do Projeto.

3.1. Tipos de Aplicação

De modo a desenvolver aplicações para dispositivos iOS podem ser utilizadas diferentes tipos de aplicações, entre as quais se destacam as aplicações híbridas e nativas.

3.1.1. Aplicações Híbridas

O desenvolvimento de aplicações híbridas tem nos últimos anos vindo a crescer em popularidade. A base do desenvolvimento híbrido consiste em desenvolver uma única aplicação e manter a compatibilidade em todas as plataformas. Existem diferentes métodos para a realização destas aplicações:

- Usando tecnologia mais relacionadas à web. O código é encapsulado em contentores nativos, carregando a maior parte da informação das páginas conforme o usuário vai navegando. A *framework* [1] que mais se destaca é o ionic que por sua vez é baseada em angular;
- Utilizando uma única linguagem que é compilada para as especificas plataformas. O resultado são aplicações com linguagem nativa que, contudo, não foi criada pelo programador, mas sim convertida. Exemplo de tecnologia com essa abordagem é o Xamarin que gera aplicações para iOS e Android usando C#;
- Um misto dos dois métodos já apresentados, no qual a aplicação é criada em parte por linguagem nativa e outras partes utilizando uma única linguagem que foi traduzida. A grande diferença entre essa e a abordagem anterior, é que o programador tem total autonomia de editar o código gerado, podendo inclusive

[1] Framework- (Wikipedia, 2020) Um framework em desenvolvimento de software, é uma abstração que une códigos comuns entre vários projetos de software provendo uma funcionalidade genérica.

alterar a forma como ele trabalha. Exemplo deste método é a Framework React-Native que utiliza JavaScript.

Este tipo de desenvolvimento apresenta como principais vantagens:

- A rapidez de desenvolvimento, aliado ao desenvolvimento efetuado através de ferramentas que permitem facilitar e acelerar o trabalho do programador;
- Diminuição de custos de desenvolvimento, aliado ao curto tempo de desenvolvimento e ao geral menor custo de contratação de um programador comparativamente ao nativo.

Contudo apresentam também várias desvantagens, das quais se destacam:

- A experiência do utilizador é afetada, o que se deve à menor otimização em comparação ao nativo bem como à limitação de funcionalidades que apenas estão disponíveis em aplicações nativas;
- Alta manutenção, embora as aplicações híbridas possuam uma única linguagem de programação inicial, continua a depender de elementos nativos para poder operar corretamente. Portanto, para manter esta aplicação, sempre que seja necessário alterar o código principal, é necessário alterá-lo três vezes: no código principal, na aplicação Android e na aplicação iOS para garantir que tudo esteja a funcionar perfeitamente em todas as plataformas.

3.1.2. Aplicações Nativas

O desenvolvimento de aplicações nativas representa a nível de vantagens e desvantagens o inverso do desenvolvimento híbrido, desta forma garantem alta performance, acesso imediato a novas funcionalidades, bem como, baixa necessidade de manutenção, contudo existem também desvantagens, é o caso do aumento do tempo de desenvolvimento que, por sua vez, aumenta o custo de desenvolvimento, bem como, o desenvolvimento para apenas uma plataforma ao invés de para várias.

O desenvolvimento de aplicações nativas iOS pode ser efetuado em duas linguagens de programação distintas:

- Objective-C;
- Swift

Sendo a mais predominante no mercado atual o Swift, linguagem criada pela Apple a 2 Junho de 2014 caraterizada pela sua potencialidade e intuitividade no desenvolvimento de aplicações para macOS, iOS, watchOS, tvOS e iPadOS. O Swift consiste na escolha da Apple para substituir o desenvolvimento em Objective-C, padrão utilizado até então, esta linguagem tal como o nome indica tem na sua essência a linguagem C que foi criada em 1972 por Bell Labs e Dennis Ritchie.

O desenvolvimento em Objective-C tem por sua vez algumas vantagens sendo a maior o facto de não haver constante atualização da linguagem (C) tal como acontece em linguagens mais recentes como é o exemplo do Swift, uma vez que estas atualizações não estão presentes, não é por isso necessário atualizar o código para suportar uma nova versão, desta forma aplicações necessitam de uma menor manutenção e a probabilidade de erros inesperados é menor.

3.2. IDE

De forma a desenvolver a aplicação iOS pretendida é necessário recorrer a um IDE sendo o mesmo caraterizado por um ambiente de desenvolvimento integrado, ou seja, uma plataforma que inclui um conjunto de ferramentas necessárias para o desenvolvimento de *software*. A sua principal função é ajudar o programador a editar o código, exemplo disto é a funcionalidade de *code intellisense*[2]. É no IDE que também se encontra o compilador, responsável por executar o programa criado e verificar a existência de erros.

No que conta ao desenvolvimento de software para dispositivos móveis é adicionalmente importante que o IDE suporte a emulação de dispositivos móveis uma vez que o programador poderá não ter um dispositivo compatível com o *software* a

[2] Code intellisense - (Wikipedia, 2020) Code intellisense é um recurso de conclusão de código com reconhecimento de contexto em alguns ambientes de programação que acelera o processo de codificação de aplicativos, reduzindo erros de digitação e outros erros comuns.

desenvolver ou mesmo uma gama alargada de equipamentos de modo a testar o *software*.

De modo a desenvolver aplicações iOS podem ser utilizados diferentes ambientes de desenvolvimento integrado, de entre os quais se destacam:

- XCode;
- AppCode.

O XCode, desenvolvido pela Apple para macOS pretende apresentar uma interface robusta de desenvolvimento de aplicações para macOS, iOS, watchOS, iPadOS e tvOS de uma forma nativa. Não apresenta custos de licença pelo que qualquer utilizador com um equipamento que corra macOS pode desenvolver aplicações de forma gratuita.

O AppCode da JetBrains, empresa especializada no desenvolvimento de IDEs contando de momento com 9 IDEs das mais variadas linguagens de programação, permite à semelhança do XCode o desenvolvimento de aplicações macOS, iOS, watchOS, iPadOS e tvOS de forma nativa, contudo este apresenta várias diferenças em relação ao XCode, tem como principal vantagem a sua disponibilidade para Windows, Linux e macOS ao contrário do XCode que apenas se encontra presente em macOS, por sua vez apresenta várias desvantagens, tais como:

- Não apresenta *Interface Builder*[3], desta forma é necessário que o XCode esteja a correr simultaneamente para o uso desta funcionalidade;
- Não apresenta emulador próprio, utiliza por isso, quando num sistema macOS, o emulador existente do XCode;
- Apresenta custos de licença que podem chegar às centenas de euros por ano.

[3] Interface Builder - (Wikipedia, 2020) O Interface Builder é um aplicativo de desenvolvimento de software que faz parte do Xcode, permitindo aos programadores criar interfaces para aplicativos usando uma interface gráfica.

3.3. Gestor de dependências

Uma boa prática no design de software é a criação de software a partir de módulos menores e de propósito único. Com a ampla adoção de software com código open-source dá-se por vezes a necessidade de reutilizar funcionalidades já existentes, que não se encontram por sua vez implementadas nativamente. Desta forma, para construir uma aplicação poderá ser necessário incluir bibliotecas/dependências externas.

Um gestor de dependências é uma coleção de ferramentas de software que automatiza o processo de instalação, atualização, configuração e remoção de dependências. No desenvolvimento iOS nativo podem ser usados diferentes gestores, é o caso do CocoaPods, Carthage e do recente Swift Package Manager.

3.3.1. CocoaPods

O CocoaPods foi o primeiro gestor de dependência oficial para iOS. É suportado por praticamente todas as dependências externas. A configuração de dependências com o CocoaPods é efetuada da seguinte forma:

- Criação de um Podfile;
- Edição do Podfile de modo a listar as dependências e as suas versões a incorporar no projeto (Figura 1);
- Execução do Podfile na linha de comandos.

```
platform :ios, '8.0'
use_frameworks!

target 'ConquerTheWorld' do
    pod 'XCGLogger', '3.0'
    pod 'SVProgressHUD', '1.1.3'
    pod 'EZAudio', '1.1.1'
end
```

Figura 1- Exemplo Podfile

Após o qual o CocoaPods cria um arquivo *Xcode Workspace* que contém o projeto e todas as dependências, este processo exibe vantagens e desvantagens. Embora o processo de instalação seja razoavelmente simples, o CocoaPods modifica os arquivos do projeto de maneira não transparente. Além disso, o CocoaPods é centralizado, sendo todas as dependências armazenadas em um repositório central. Se esse repositório for desativado por qualquer motivo, o projeto pode deixar de compilar.

3.3.2. Carthage

Carthage é descentralizado e atualmente suporta várias fontes de dependência:

- Repositórios Github;
- Repositórios Git;
- Links binários (HTTPS público).

Existem algumas suposições sobre o processo de configuração e compilação do repositório que ocorrem no Carthage e não no CocoaPods. Um dos princípios do Carthage é a simplicidade, exigindo que o código seja estruturado de maneira previsível.

O processo de instalação de dependências através do Carthage é em grande parte semelhante ao CocoaPods:

- Criação de um Cartfile;
- Edição do Cartfile de modo a listar as dependências a incorporar no projeto;
- Execução do Cartfile na linha de comandos.

Em contraste com o CocoaPods, o Carthage apenas gere dependências, não as integra no projeto, representando esta a maior diferença entre os dois, a forma de incorporação das dependências no projeto. O Cocoapods cria um *workspace* que possui o código fonte de todas as dependências incluídas, esta abordagem tem algumas desvantagens sendo a maior o aumento do tempo de cada *clean build*, uma vez que cada *clean build* necessita de reconstruir todas as dependências. Por outro lado, o Carthage, apresenta-se como uma alternativa muito mais descomplicada, criando as dependências uma única vez, todavia cabe ao programador incluir as

mesmas no projeto, este processo é exemplificado na Figura 2 em que é incluída a dependência RxSwift e RxCocoa.

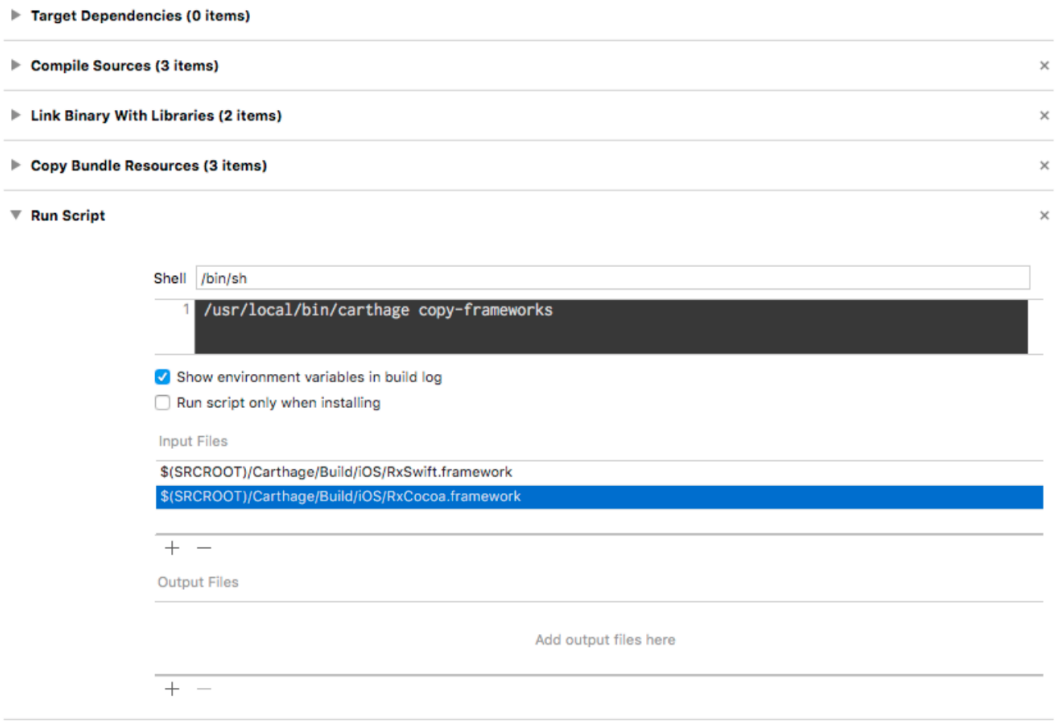


Figura 2- Script de especificação de path para Frameworks

3.3.3. Swift Package Manager

Com o novo Swift Package Manager, é possível substituir as soluções não nativas utilizadas até ao momento por uma profundamente integrada. O Swift Package Manager apresenta-se disponível desde o Swift 3.0, contudo inicialmente encontrava-se apenas disponível para projetos Swift do lado do servidor ou da linha de comandos. Desde o lançamento do Swift 5 e do Xcode 11, o SPM tornou-se uma alternativa real ao CocoaPods e Carthage.

Ao contrário da instalação de dependências até ao momento, com o SPM este processo é efetuado através de uma interface gráfica acessível e intuitiva. Os passos para a instalação de uma dependência com SPM passam por:

- Em “File” seguido de “Swift Packages” selecionar a opção “Add Package Dependency” (Figura 3);

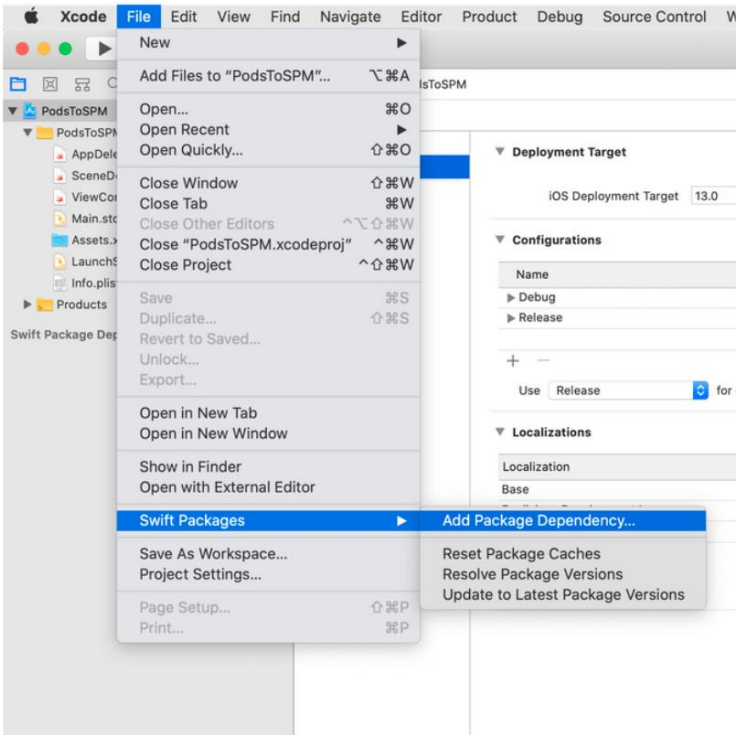


Figura 3- Opção Adicionar Dependência SPM

- Especificar o repositório git que contém a dependência (Figura 4);

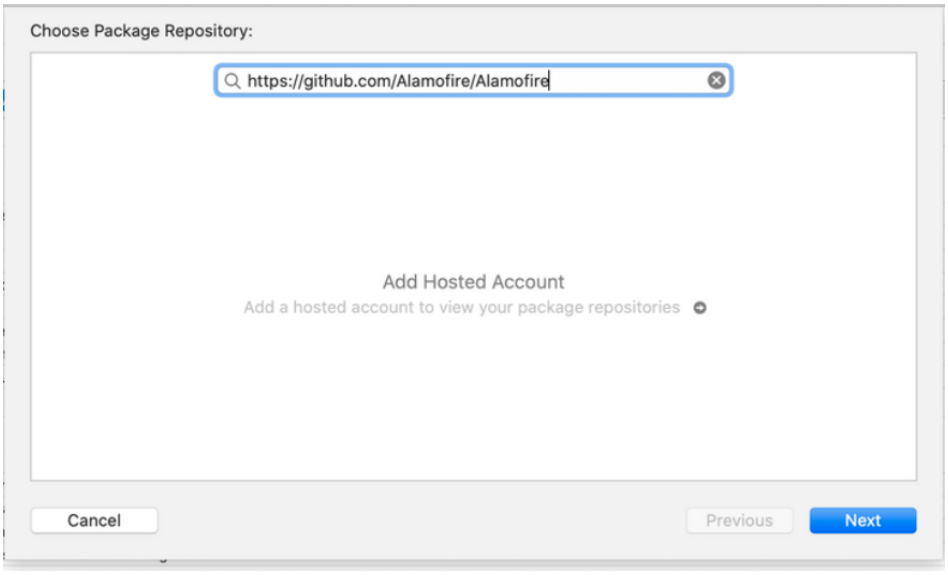


Figura 4- Adicionar Repositório da Dependência com SMP

- Especificar a Versão pretendida (Figura 5);

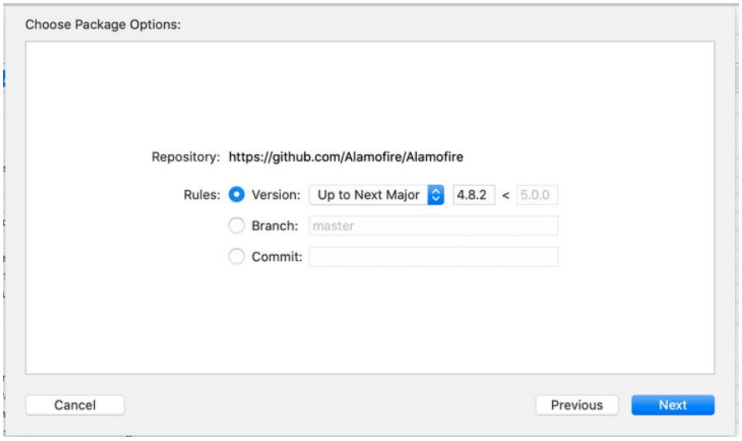


Figura 5- Seleção da versão da Dependência com SMP

- Por fim, o Xcode realiza o processo de integração necessário, como incorporar a dependência ao projeto e configuração do compilador, como é possível observar pela Figura 6 a dependência foi adicionada ao projeto com sucesso.

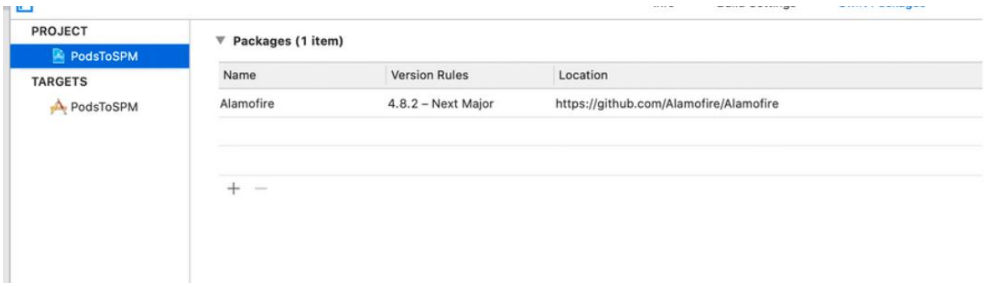


Figura 6- Dependência adicionada com SMP

4. Projeto

Neste capítulo é descrito em pormenor a realização do projeto desde o planeamento até à aplicação final.

4.1. Planeamento e Metodologia

De modo a construir uma aplicação de encontro com o nível de exigência estabelecido pelo cliente foi necessário ponderar numa fase inicial todas as etapas de realização do projeto antes do início do mesmo, desta forma foram percorridos vários passos de forma a garantir que a preparação quer a nível de competências técnicas como de decisões práticas estabelecidas com o cliente eram tomadas.

Num momento inicial foram adquiridas competências teóricas no que diz respeito aos tipos de aplicação existentes bem como a que IDE recorrer, passando pelo gestor de dependências externas, tendo desta forma sido efetuadas decisões sobre que rumo tomar para o desenvolvimento da aplicação.

Após as decisões técnicas efetuadas foi iniciado o processo de aprendizagem quer da sintaxe da linguagem de programação a utilizar, quer do IDE elegido, para tal foi recorrido à documentação oficial da Apple bem como a desafios práticos propostos pelo orientador, sendo que qualquer dúvida era esclarecida com o mesmo. Esta fase decorreu durante o período de 17 de Fevereiro a 2 de Março.

No dia 3 de Março foi efetuada uma reunião com o cliente de modo a compreender de uma forma mais sucinta a ideia do mesmo quanto à aplicação em questão, quer em termos de design, bem como de funcionalidade.

Após a primeira reunião com o cliente foram analisadas por todos os elementos envolvidos no projeto PaulosAuto as necessidades estéticas e funcionais do projeto de modo a dar inicio à construção de protótipos de baixa fidelidade, após os ajustes nos protótipos de baixa fidelidade efetuados foi tomada a metodologia da realização de protótipos de alta fidelidade individuais, um para aplicação iOS, android e web de

modo a permitir reunir as melhores ideias num único protótipo o qual procuraria dar a conhecer ao cliente uma amostra da estética e funcionalidade da aplicação final, estando esta recetível a alterações consoante o feedback recebido.

No desenvolvimento desta aplicação optou-se pela metodologia ágil Kanban. (heflo, 2020) Proveniente de técnicas de gestão de indústrias japonesas dos anos 60, o Kanban começou como uma técnica de gestão de filas para controle de inventário, desenvolvida pela Toyota. Nos dias de hoje Kanban é uma metodologia ágil de gestão de projetos, baseado num quadro Kanban, onde devem constar diversas colunas dependendo da complexidade do projeto e onde deve existir uma organização da esquerda para a direita indicando as fases pela qual as tarefas, indicadas em cartões, devem passar. O objetivo do Kanban não é substituir a comunicação e a colaboração, mas sim aumentar a rapidez de desenvolvimento ao mesmo tempo que permite controlar o estado do desenvolvimento do projeto. De forma a produzir os cartões caraterizados por representar as várias tarefas a desempenhar foi realizada a análise de tarefas do projeto, tendo sido estabelecidas as seguintes tarefas como principais:

- Implementação de listagem de equipamentos;
- Implementação de informação de equipamento;
- Implementação do registo de problemas;
- Implementação do reporte de problema;
- Implementação do NetworkManager;
- Implementação do Log-in;
- Implementação do Perfil;
- Implementação do histórico de equipamento;
- Implementação da listagem de problemas;
- Implementação da listagem de faturas;
- Fase de testes.

Com as necessidades funcionais da aplicação estabelecidas foi criado um diagrama de Gantt (ANEXO A) espectável, o qual representa as metas no desenvolvimento da aplicação, sendo a partir do mesmo possível observar se o desenvolvimento se encontra ao ritmo esperado.

A segunda reunião com o cliente encontrava-se marcada para dia 17 de Março, contudo devido ao surto de Covid-19 e à consequente limitação social dos contactos com o cliente todos os contactos futuros passaram a ser efetuados através de correio eletrónico.

A partir deste momento e em regime de teletrabalho foram efetuadas reuniões diárias com os elementos envolvidos no projeto PaulosAuto de forma a:

- Discutir as funcionalidades a realizar durante o presente dia;
- Esclarecer dúvidas técnicas ou funcionais;
- Garantir um desenvolvimento paralelo entre as aplicações android e iOS de modo a assegurar uma aplicação final com o mesmo layout e funcionalidade.

Sempre que necessário e para maximizar a eficiência dos contactos com o cliente foram aglomeradas duvidas de todos os envolvidos com o projeto e enviado um email com as mesmas.

4.2. Plataformas e Softwares Utilizados

Foram utilizadas plataformas e *softwares* de modo a garantir o desenvolvimento da aplicação, bem como a comunicação com os elementos envolvidos no projeto, dado o estado de teletrabalho que preencheu em grande parte o tempo de realização do projeto.

De forma a assegurar a comunicação foi utilizado o Microsoft Teams da Softinsa, quer para a realização de videochamadas como para a partilha de ficheiros. Em integração com o Teams apresenta-se a plataforma Trello, desenvolvido pela Atlassian, cuja utilização apresentou-se fundamental na organização e partilha do estado das tarefas de cada pessoa, encontrando-se estas organizadas por: em realização, finalizadas, em revisão ou por começar, como é possível observar pela Figura 7.

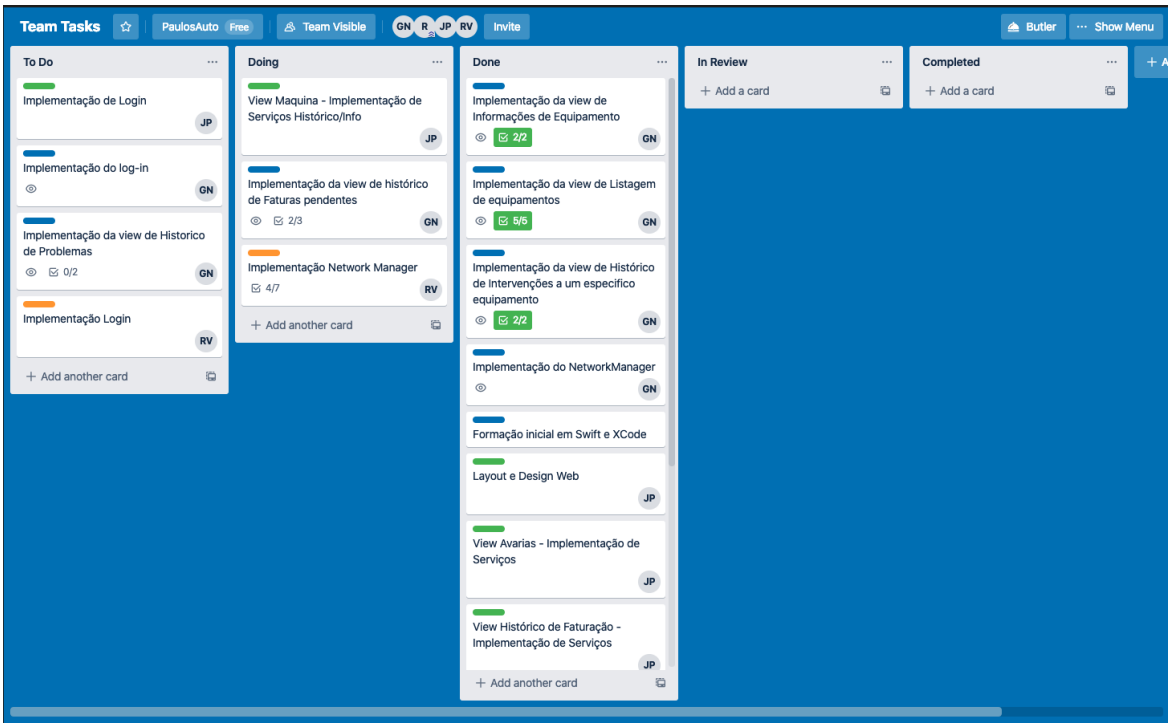


Figura 7- Trello Projeto PaulosAuto

No que conta ao desenvolvimento da aplicação, para além do software já mencionado, nomeadamente XCode foi adicionalmente utilizado o Adobe XD, que é uma plataforma criada pela Adobe caraterizada por ser poderosa, colaborativa e fácil de usar na criação de designs e que foi empregue na criação dos protótipos de alta fidelidade.

De modo a partilhar e estruturar o código criado foi utilizado o padrão de organização presente na empresa, ou seja, o GitLab da Softinsa, caraterizado por ser um gestor de repositórios Git, no qual foi criado um repositório para o projeto. Sendo que para um acesso com maior funcionalidade e rapidez ao mesmo foi usado o SourceTree, uma GUI para Git que oferece uma representação visual dos repositórios bem como acesso a várias funcionalidades, tais como:

- Criação e clonagem de repositórios;
- *Commit*[4], *push*, *pull* and *merge*;
- Detecção e resolução de conflitos;
- Acesso ao histórico de *commits*.

[4] Commit - (Wikipedia, 2020) Um commit adiciona as alterações mais recentes de parte do código-fonte ao repositório.

No auxílio à implementação da ligação à REST API foi utilizado o Postman, tendo este um importante papel no teste de API's, sendo possível verificar quer o input necessário como o output esperado da mesma.

De modo a hospedar a API foi recorrido ao Docker Toolbox, este software apresenta-se como um shell pré-configurado para um ambiente de linha de comandos Docker incluindo para tal um conjunto de softwares, sendo estes:

- Docker *Machine*;
- Docker *Engine*;
- Docker *Compose*;
- Kitematic, um GUI para docker;
- Oracle VirtualBox.

O Docker (opensource, 2020) é uma ferramenta projetada para facilitar a criação, implantação e execução de aplicações usando contentores, sendo um contentor caraterizado por um pacote de todas as ferramentas necessárias para correr uma dada aplicação, ferramentas estas como bibliotecas e outras dependências. Desta forma, esse container poderá ser executado em qualquer outra máquina Linux, independentemente das configurações personalizadas que a máquina possa ter e que possam diferir da máquina usada para escrever e testar o código. O Docker Toolbox difere do Docker nativo uma vez que os containers são executados numa máquina virtual Linux, deste modo é instalado o Oracle VirtualBox, o qual possibilita a criação da mesma.

De forma a auxiliar a criação de modelos que suportam os dados recebidos e enviados para a API foi recorrido ao SwiftyJSONAccelerator. Esta aplicação gera a partir do JSON[5] recebido um modelo em Swift que pretende suportar o JSON pretendido, como é possível verificar pela Figura 8.

[5] JSON - (Wikipedia, 2020) JSON, um acrônimo de JavaScript Object Notation, é um formato compacto, de padrão aberto independente, de troca de dados simples e rápida entre sistemas

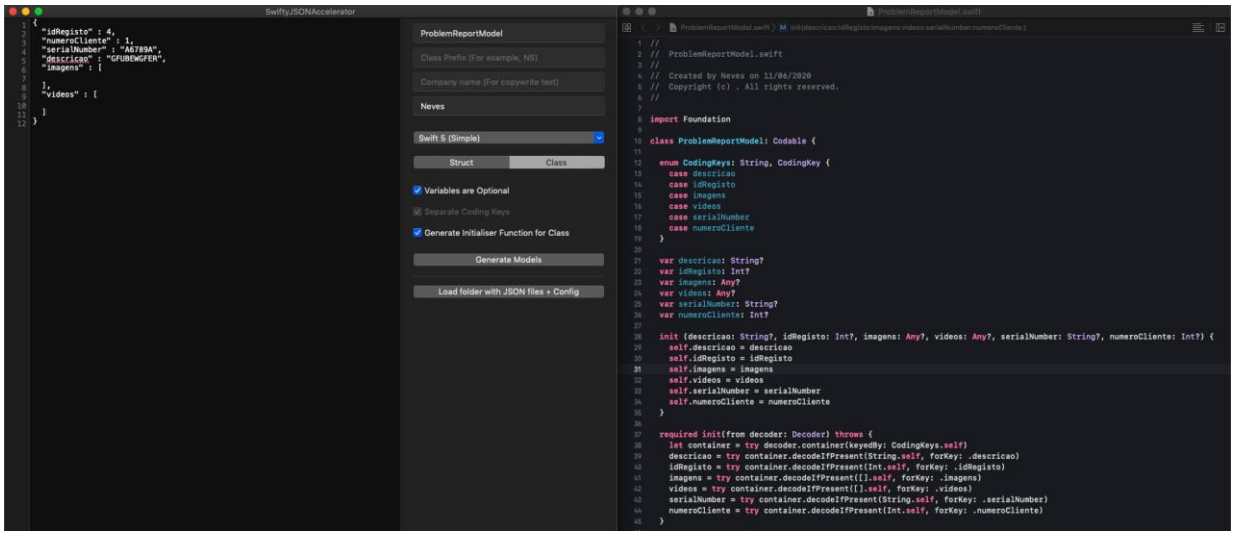


Figura 8- Modelo gerado a partir do SwiftyJSONAccelerator

4.3. Contacto com o Cliente

No dia 3 de Março foi efetuada a primeira reunião com o cliente pelo que foram discutidos os requisitos funcionais da aplicação em questão, bem como os elementos gráficos pretendidos.

Os requisitos funcionais mencionados pelo cliente passam por:

- Dar a possibilidade de listar todos os equipamentos de um dado cliente, mostrando informação relativa às horas de utilização, histórico de reparações, entre outras funcionalidades a definir;
- Fornecer uma forma fácil e intuitiva de recolher o número de horas de utilização de cada equipamento por parte dos seus clientes;
- Fornecer uma forma de comunicação por parte dos clientes aquando de um eventual problema com o equipamento que possuam;
- Fornecer uma listagem das faturas relativas a um dado cliente de modo a que o mesmo seja alertado caso existam faturas pendentes.

Adicionalmente foi dada liberdade para a implementação de novas funcionalidades que tornem a aplicação mais útil e intuitiva para os clientes. No que conta aos elementos gráficos foi mais uma vez dada liberdade de implementação pelo que foi apenas sugerido a utilização das cores relativas à PaulosAuto e à CAT (empresa

parceira da PaulosAuto) pelo que as cores base da aplicação passam pelo vermelho, amarelo caterpillar, cinzento, preto e branco.

No dia 24 de Março foi efetuado um contacto via email onde foram esclarecidas várias questões relativas à funcionalidade e elementos gráficos da aplicação, adicionalmente foi apresentado o trabalho desenvolvido até ao momento, tendo para isso sido enviados os protótipos de alta fidelidade. Neste email foram efetuadas questões acerca de vários aspetos da aplicação, entre os quais:

- A preferência da existência de dois tipos de vista na listagem de equipamentos, entre os quais em lista e mosaico, ou apenas uma delas;
- Consideração da necessidade de *upload* de várias imagens/videos na submissão de um reporte de problema ou apenas de um só;
- Necessidade de um modo noturno;
- Que alterações realizar tendo em conta os protótipos enviados.

Pelo que o feedback recebido assenta nas principais considerações:

- Necessidade de filtragem por equipamentos com contrato ativo, a qual por sugestão do cliente passaria a apresentar-se como um toggle com o nome SMP, o qual quando ativo apresenta apenas os equipamentos com contrato ativo;
- Um maior aproveitamento de espaço no menu de filtragem, podendo o mesmo ocupar verticalmente a totalidade do ecrã;
- Existir apenas um histórico de intervenções ao invés de um histórico com três tipos diferentes de acontecimentos: reporte de problema, intervenção e registo de utilização;
- Inexistência da informação detalhada de cada equipamento na base de dados
- A listagem de equipamentos utilizando uma só vista;
- A necessidade de upload de várias imagens/vídeos de modo a facilitar a identificação de um dado problema;
- Ajustes linguísticos de modo a facilitar a compreensão do utilizador;
- Obrigatoriedade de inserção do número de horas no reporte de problemas;
- A não necessidade de um modo noturno.

4.4. Prototipagem

Neste capítulo serão apresentadas e esclarecidas as decisões de funcionalidade e design tomadas na realização da prototipagem de alta fidelidade.

4.4.1. Autenticação

De forma a autenticar o cliente estabeleceu-se como necessária uma interface simples e intuitiva que pretende iniciar sessão através do email e da palavra-passe do cliente, esta *interface* não apresenta inicio de sessão por outros meios uma vez que não é um requisito por parte do cliente (Figura 9).

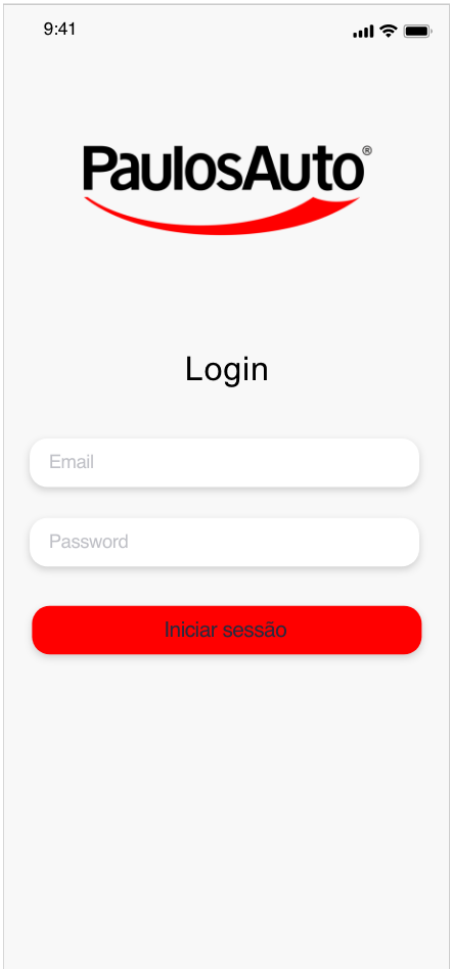


Figura 9- Prototipagem - Autenticação

4.4.2. Listagem de Equipamentos

De modo a listar os equipamentos associados ao cliente foram idealizados dois tipos de visualização, é o caso da vista em lista presente na Figura 11 e da vista em mosaico presente na Figura 10. Em ambas as vistas é possível observar o equipamento, o seu modelo, utilização em horas e o número de série correspondente.

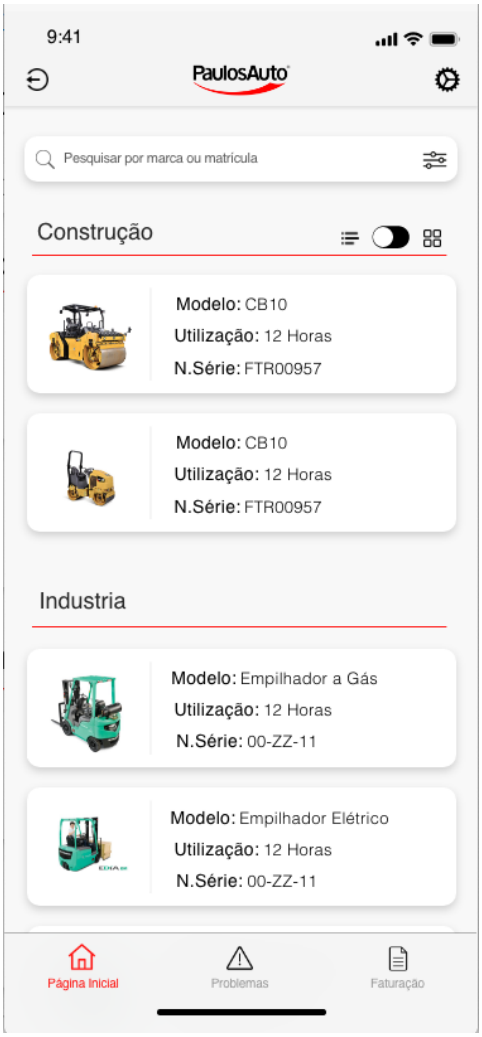


Figura 11- Prototipagem - Listagem de equipamentos em Lista

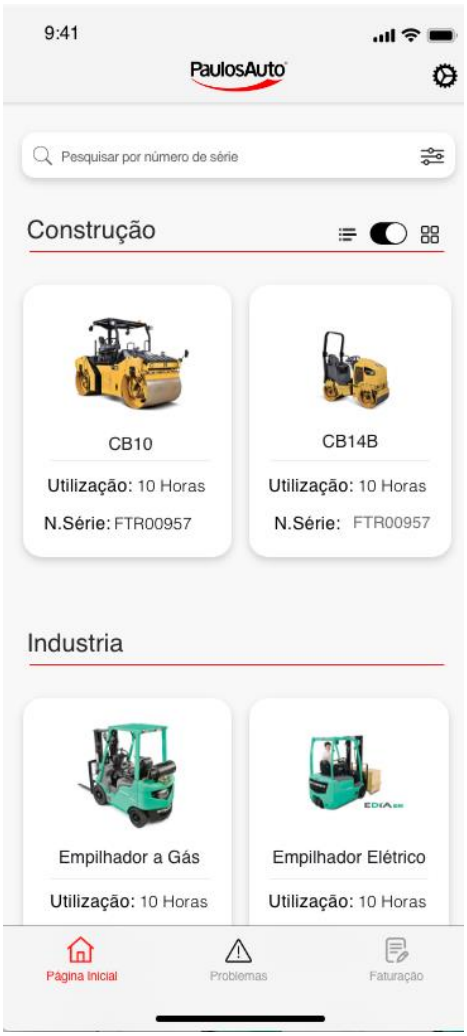


Figura 10- Prototipagem - Listagem de equipamentos em Mosaico

Foi adicionalmente idealizado um menu de filtragem de modo a ordenar e categorizar os equipamentos, apresentando-se o menu de ordenação na Figura 13 e o menu de categorias na Figura 12.

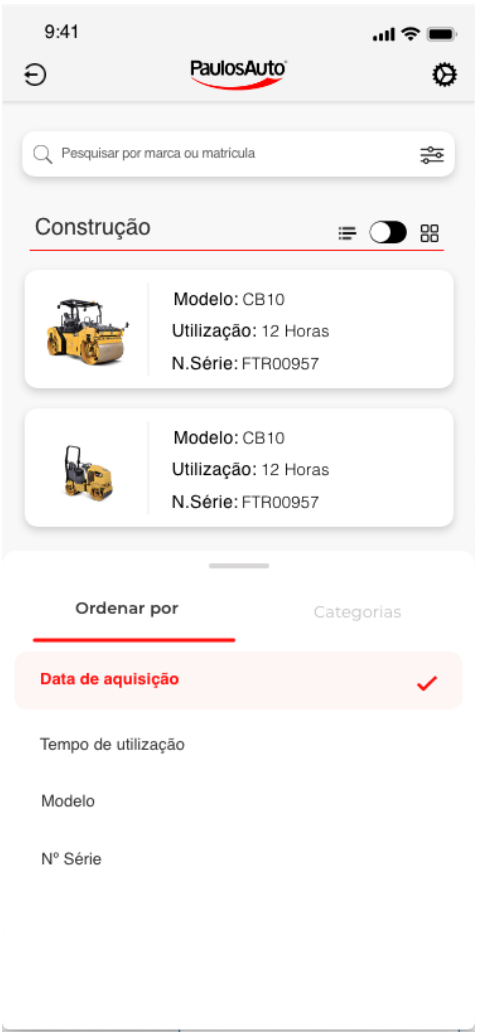


Figura 13- Prototipagem - Menu de Filtragem - Ordenar por

Figura 12- Prototipagem - Menu de Filtragem- Categorias

4.4.1. Informações de Equipamento

A vista da Figura 14 foi idealizada de modo a permitir ao utilizador obter mais informações sobre o equipamento selecionado, apresentando-se nesta adicionalmente dois botões, é o caso do registar utilização e reportar problema que serão demonstrados e explicados de seguida.

Figura 14- Prototipagem - Informações de Equipamento

4.4.2. Histórico de Equipamento

De modo a permitir ao utilizador visualizar os acontecimentos ligados a um dado equipamento foi criada uma interface (Figura 15) que contempla um histórico do equipamento, este histórico é constituído por reportes de problemas submetidos, intervenções e registos de utilização.

Figura 15- Prototipagem - Histórico de Equipamento

4.4.1. Registo de Utilização de Equipamento

De forma a permitir o registo de horas de um equipamento foi criado um *layout* (Figura 16) que permite inserir o número de horas atuais. De modo a evitar erros de registo é mostrado ao utilizador o número de série do equipamento e o último registo de horas.

Figura 16- Prototipagem - Registo de Utilização de Equipamento

4.4.2. Reporte de Problema com um Equipamento

De modo a registar eventuais problemas ligados à utilização dos equipamentos foi criada uma *interface* (Figura 17) que permite adicionar uma descrição bem como provas visuais ao reporte.

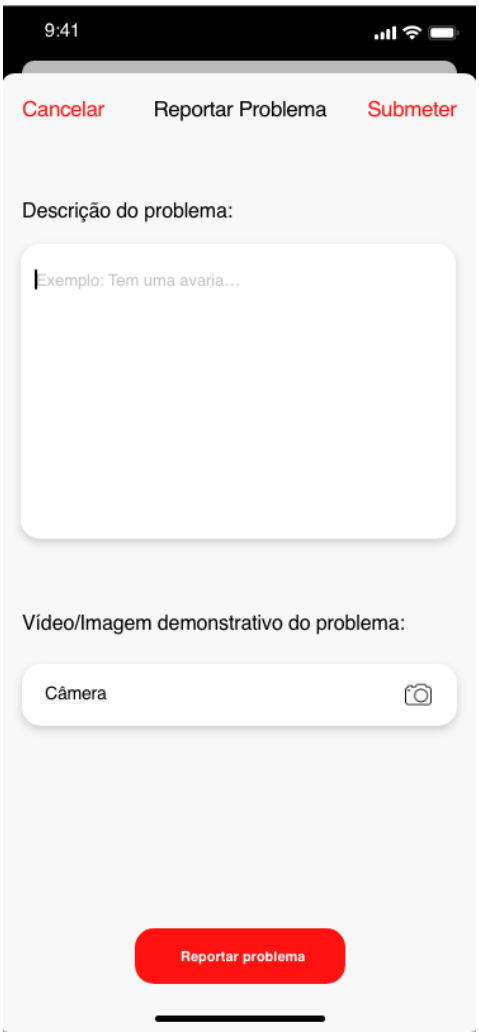


Figura 17- Prototipagem - Reporte de Problema com Equipamento

4.4.3. Listagem de Problemas Submetidos

De forma a listar os reportes de problemas efetuados foi criada a vista visível na Figura 18, segundo a qual é possível observar o progresso de envio do mesmo, organizado por data.

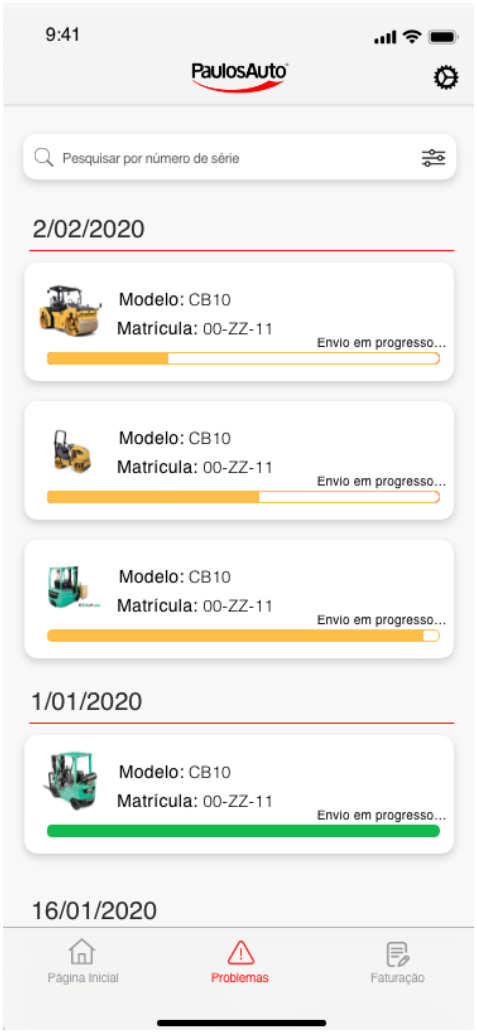


Figura 18- Prototipagem - Listagem de Problemas Submetidos

4.4.4. Listagem de Faturas Pendentes

Outra funcionalidade pretendida pelo cliente estabelece-se na consulta de Faturas com pagamento pendente, desta forma foi criada a vista presente na Figura 19 que pretende esclarecer o cliente sobre dados relevantes da Fatura como é o caso do valor, valor liquidado, data emissão da fatura e data de vencimento da mesma. Existe adicionalmente um botão de modo a descarregar a fatura.

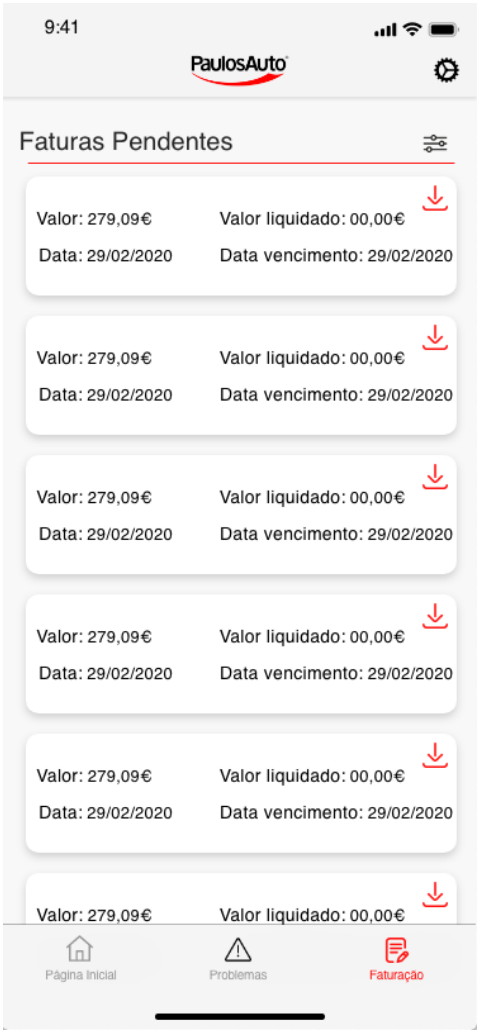


Figura 19- Prototipagem - Listagem de Faturas Pendentes

4.4.5. Página Perfil

Adicionalmente foi concebida uma página de perfil presente na Figura 20 onde é possível visualizar o nome do utilizador e com as opções de ativação ou não de notificações, modo noturno e possibilidade de terminar sessão.

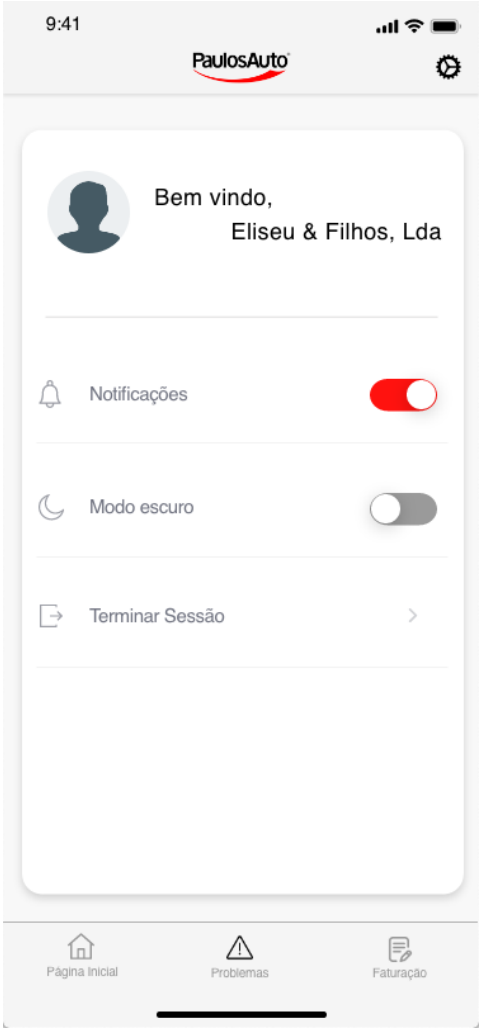


Figura 20- Prototipagem - Página de Perfil

4.5. API

Uma API é uma *interface* de computação que define interações entre vários intermediários de *software*. Esta define os tipos de chamadas que podem ser efetuadas, como fazê-las, apresentando os formatos de dados que devem ser usados bem como as convenções a seguir.

A API utilizada denomina-se de REST API (sitepoint, 2020). O termo significa *Representational State Transfer* e representa um conjunto de regras de funcionamento de uma API. Embora uma REST API possa ser usada em praticamente qualquer protocolo, geralmente recorre ao HTTP. Como os dados não estão vinculados a métodos e recursos, o REST pode lidar com vários tipos de chamadas e retornar diferentes formatos de dados. Esta liberdade e flexibilidade inerentes ao design da API REST permitem criar uma API que atenda às suas necessidades e, ao mesmo tempo, atenda às necessidades de clientes muito diversos. O REST não é restrito ao XML, podendo retornar XML, JSON, YAML ou qualquer outro formato, dependendo do que o cliente solicitar. Sendo que na API utilizada o tipo de dados retornado é o JSON.

Na arquitetura REST, os clientes enviam *requests* para retornar ou modificar recursos e os servidores enviam respostas para essas *requests*. Sendo que cada *request* é constituída por:

- *Endpoint*, corresponde ao url, o qual segue a estrutura *root-endpoint*/?;
- *Method*, o que corresponde ao tipo de request que se efetua à API, podendo ser de 5 tipos diferentes: *GET*, *POST*, *PUT*, *PATCH*, *DELETE*;
- *Headers*, os quais são utilizados para fornecer informações ao cliente e ao servidor. Pode ser usado para vários propósitos, como autenticação e fornecimento de informações sobre o conteúdo do corpo.
- *Body*, contém dados que se pretende enviar ao servidor. Esta opção é usada apenas com request do tipo *POST*, *PUT*, *PATCH* ou *DELETE*.

As respostas da API contêm códigos de resposta para alertar o cliente sobre o tipo de resposta do servidor a um request, sendo os códigos de resposta presentes nesta API:

- 200, correspondente ao código de resposta de sucesso;
- 401, o *request* não foi sucedido uma vez não possuir credenciais de autenticação válidas;
- 403, o *request* não foi sucedido por o acesso ao recurso solicitado se encontrar proibido por algum motivo, não sendo este diretamente ligado à falta de credenciais de autenticação válidas como é o caso no código 401;
- 404, ocorre quando não é encontrada informação relativa ao *request* efetuado;
- 413, ocorre quando os dados de um request passam o limite estabelecido pela API;
- 500, indica que o servidor encontrou uma condição inesperada que o impediu de responder ao *request*.

Os *endpoints* utilizados na API apresentam-se na Tabela 1, de salientar que:

- Não existem métodos *DELETE* uma vez que não existe qualquer funcionalidade no produto desenvolvido que permita realizar operações de remoção de dados;
- Todas as respostas da API são efetuadas em JSON;
- Todos os requests à API são efetuados em JSON ao contrário das correspondentes ao *endpoint* “Reporte de Problema com Equipamento” que é efetuado em multipart-data, usado no *upload* de ficheiros, sendo que multi-part significa que os dados do formulário se encontram divididos em várias partes.

Tabela 1- Endpoints utilizados na API

Nome	Endpoint	Método
Listar Equipamentos	/PaulosAutoAPI/clientes/equipamentos_v2/{clientNumber}	GET
Listar Historico Equipamento	/PaulosAutoAPI/equipamentos/intervencoes/{serialNumber}	GET
Listar Faturas Pendentes	/PaulosAutoAPI/clientes/faturas/{id}	GET
Listar Problemas submetidos	/PaulosAutoAPI/equipamentos/avarias/{id}	GET

Validação de Token	/Autenticacao/checkall	GET
Reporte de Problema com Equipamento	/PaulosAutoAPI/equipamentos/avarias	POST
Registo de Utilizacao de Equipamento	/PaulosAutoAPI/equipamentos/utilizacao	POST
Autenticacao Utilizador	/Autenticacao/login	POST

4.6. Ponderações de Desenvolvimento

Neste capítulo são descritas e esclarecidas as ponderações no desenvolvimento da aplicação, passando pela demonstração das questões que surgiram durante o período de desenvolvimento, bem como os caminhos tomados para a sua resposta.

Numa primeira fase foram efetuadas decisões técnicas, nomeadamente:

- Por que tipo de aplicação optar;
- Que linguagem de programação utilizar;
- A que IDE recorrer;
- Em caso de necessidade de instalação de dependências externas que gestor de dependências utilizar;
- Que dispositivos suportar;
- Que versões de iOS suportar.

Considerando as vantagens e desvantagens apresentadas no Capítulo 3.1, dos dois tipos de aplicações é optado por o desenvolvimento de uma aplicação nativa, uma vez que esta permitir explorar todas as funcionalidades do sistema operativo, bem como garantir boa performance e menor necessidade de manutenção. Dentro das aplicações nativas a linguagem de programação escolhida é o Swift visto representar o standard atual no desenvolvimento de aplicações iOS nativas.

De modo a eleger o IDE utilizado para o desenvolvimento da aplicação, foram analisadas as vantagens e desvantagens mencionados no capítulo 3.2, desta forma o

IDE escolhido foi o XCode visto apresentar as melhores funcionalidades sem um custo adicional, o que não se verifica no IDE AppCode.

A metodologia de desenvolvimento da aplicação adotada estipula que apenas seja recorrido a dependências externas quando for, de um modo geral, bastante facilitador em comparação com a implementação da mesma funcionalidade nativamente. Com a atualização do Swift para a versão 5 a programação nativa tornou-se de um modo geral bastante mais simples e rápida. Dependências externas utilizadas de uma forma padrão até então, como é o caso do Alamofire que tem como funcionalidade gerir pedidos HTTPS, passam a não ser necessárias dadas as melhorias na implementação da mesma funcionalidade nativamente. Desta forma no desenvolvimento da aplicação pretende-se que seja utilizado o menor número de dependências externas possível sem por isso comprometer a implementação de qualquer funcionalidade. Contudo, caso seja necessária a instalação das mesmas, e considerando os pontos apresentados no Capítulo 3.3 é escolhida a instalação através de Swift Package Manager, dado que:

- O processo de instalação é de um modo geral mais simples e intuitivo em relação aos restantes gestores de dependências;
- Apresenta suporte nativo.

Porém uma das principais desvantagens do SMP é o reduzido suporte de dependências em relação aos restantes gestores, apresentando apenas versões de instalação SMP as dependências mais utilizadas e/ou mais recentes. Desta forma e dada a eventual necessidade de instalação de dependências sem suporte SMP é necessário existir um gestor de dependências alternativo pelo que é optado pelo Carthage ao invés do CocoaPods pelas principais razões:

- Tempo de compilação bastante inferior;
- Não existe uma modificação não transparente de ficheiros do projeto.

No suporte de dispositivos foi optado pelo suporte de apenas iPhone uma vez que não ia de encontra as necessidades do cliente o suporte de iPad. No que conta à versão de iOS suportada, através de discussão com a equipa envolvida no projeto foi optado pelo suporte até 2 versões anteriores à atual, ou seja, o suporte da aplicação é assegurado desde o iOS 11.

4.7. Estrutura e Código

Neste capítulo é descrita a estrutura da aplicação desenvolvida, passando pela demonstração das principais funcionalidades implementadas e aspetos relevantes da fase de desenvolvimento.

4.7.1. Organização de Projeto

Um dos principais pontos no desenvolvimento de um projeto é a organização do mesmo. Para a realização desta aplicação foram utilizadas metodologias de organização segundo as boas práticas de desenvolvimento, pelo que, através dos conhecimentos passados pelo orientador foi possível desenvolver um projeto com a estrutura e o código organizados. Este tópico apresenta uma grande importância uma vez que um projeto é desenvolvido por vários elementos onde princípios de organização corretos são críticos na eficiência e sucesso da implementação do mesmo. Destacam-se por isso vários aspetos principais na organização do projeto:

- Organização do projeto em diferentes grupos, como é possível observar na Figura 22 e Figura 21 todos os ficheiros pertencem a um grupo especifico de modo a garantir uma fácil leitura da estrutura do projeto e um rápido acesso ao ficheiro pretendido;

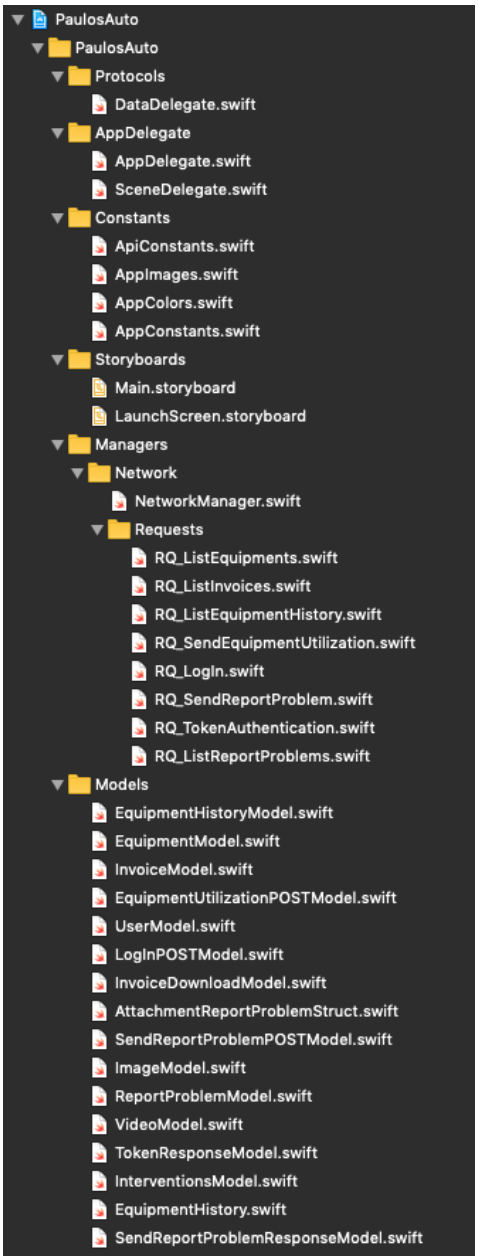


Figura 22- Estrutura de Projeto - Parte 1

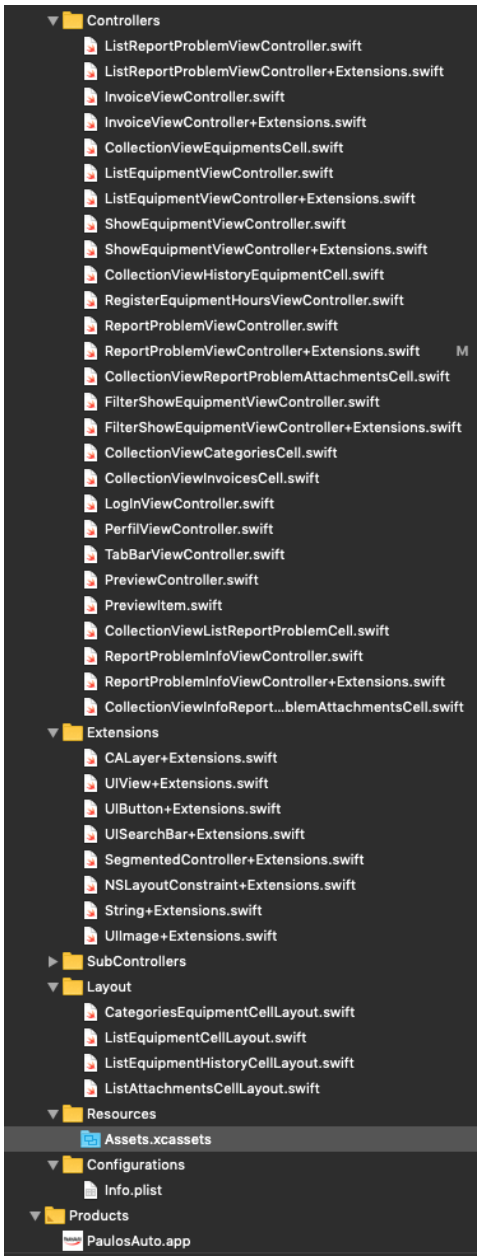


Figura 21- Estrutura de Projeto - Parte 2

- Utilização de Marks de modo a organizar os diferentes métodos existentes num dado ficheiro por categorias, podendo os Marks ser de diferentes tipos: *Outlets*, *Constants*, *Properties*, *Override inherited functions*, *Private*, *Public*, Objc *functions* e *Actions*. Adicionalmente à organização dentro dos métodos, a utilização de Marks possibilita adicionalmente a visualização desta organização na pré-visualização de um ficheiro, exemplificado na Figura 23;

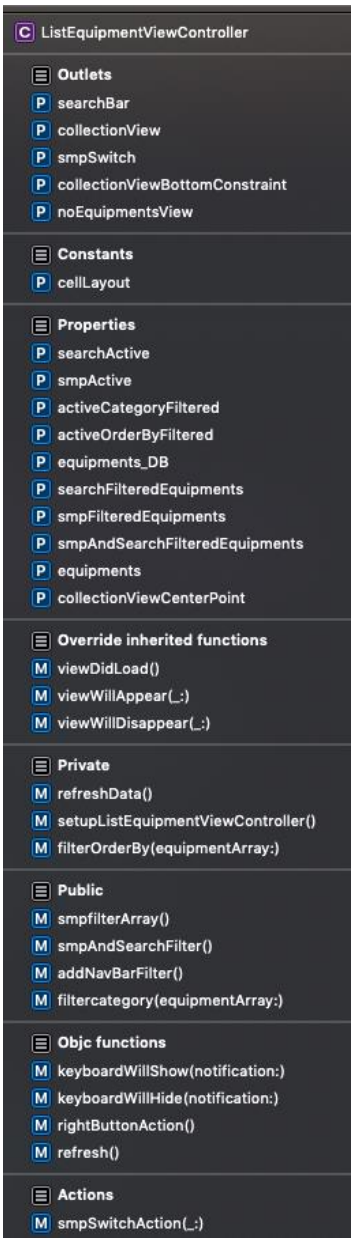
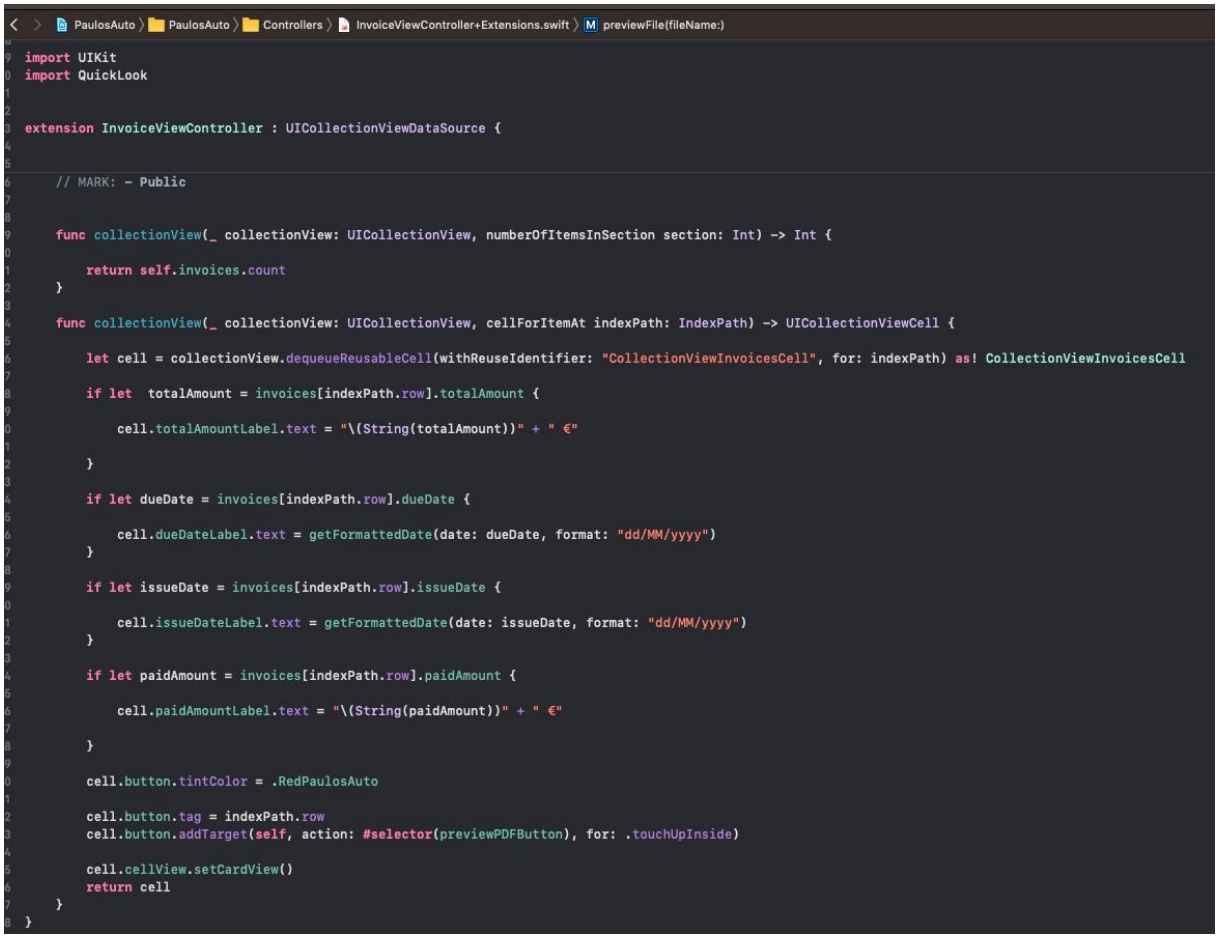


Figura 23- Utilização de notação MARK

- Criação de Extensions de ViewControllers com a nomenclatura "ViewControllerName+Extensions", esta prática é tomada quando um ViewController necessita de implementar métodos delegate[6] ou datasource[7], dá-se o exemplo da criação de um file Extensions de modo a contemplar os métodos delegate e dataSource de uma collectionView (Figura 24).

[6] Delegate - (Wikipedia, 2020) delegate refere-se à avaliação de um membro (propriedade ou método) de um objeto (o receptor) no contexto de outro objeto original (o remetente).

[7] Datasource - (Apple, 2020) Um protocolo que fornece aviso prévio dos requisitos de dados para, por exemplo uma tableview, permitindo iniciar operações de dados potencialmente demoradas mais cedo.




```swift
import UIKit
import QuickLook


extension InvoiceViewController : UICollectionViewDataSource {


    // MARK: - Public


    func collectionView(_ collectionView: UICollectionView, numberOfItemsInSection section: Int) -> Int {

        return self.invoices.count
    }

    func collectionView(_ collectionView: UICollectionView, cellForItemAt indexPath: IndexPath) -> UICollectionViewCell {

        let cell = collectionView.dequeueReusableCell(withReuseIdentifier: "CollectionViewInvoicesCell", for: indexPath) as! CollectionViewInvoicesCell

        if let  totalAmount = invoices[indexPath.row].totalAmount {

            cell.totalAmountLabel.text = "\(String(totalAmount))" + " €"

        }

        if let dueDate = invoices[indexPath.row].dueDate {

            cell.dueDateLabel.text = getFormattedDate(date: dueDate, format: "dd/MM/yyyy")
        }

        if let issueDate = invoices[indexPath.row].issueDate {

            cell.issueDateLabel.text = getFormattedDate(date: issueDate, format: "dd/MM/yyyy")
        }

        if let paidAmount = invoices[indexPath.row].paidAmount {

            cell.paidAmountLabel.text = "\(String(paidAmount))" + " €"

        }

        cell.button.tintColor = .RedPaulosAuto

        cell.button.tag = indexPath.row
        cell.button.addTarget(self, action: #selector(previewPDFButton), for: .touchUpInside)

        cell.cellView.setCardView()
        return cell
    }
}
```

Figura 24- InvoiceViewController+Extensions

- Criação de cores como *assets* [8], o que possibilita uma alteração rápida de uma dada cor em toda a aplicação ao invés do processo de substituição da mesma em todos os componentes que a utilizem.

[8] Asset - (Wikipedia, 2020) Um asset é qualquer dado, dispositivo ou outro componente do ambiente que suporte atividades relacionadas a informações.

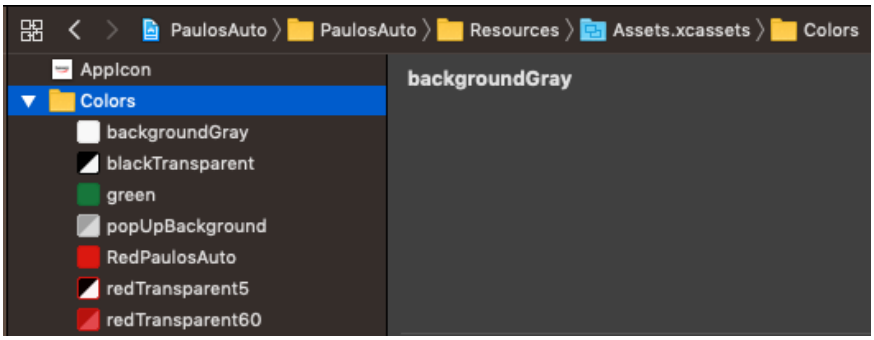


Figura 25- Criação de cores como assets

Utilização de ficheiros de constantes, os quais apresentam constantes que são utilizadas ao longo da aplicação e que ao se encontrarem num único local podem ser facilmente inspecionadas e alteradas. Na aplicação em questão foram utilizados quatro ficheiros de constantes, sendo os mesmos: ApiConstants, AppImages, AppColors, AppConstants. Dá-se o exemplo do ficheiro ApiConstants, presente na Figura 26.

```swift
struct ApiConstants {

    static let baseAPI = URL(string: "http://192.168.1.53:5080/")
    static let listEquipmentURL = baseAPI?.appendingPathComponent("PaulosAutoAPI/clientes/equipamentos_v2/")
    static let listEquipmentHistoryURL = baseAPI?.appendingPathComponent("PaulosAutoAPI/equipamentos/intervencoes/")
    static let listInvoicesURL = baseAPI?.appendingPathComponent("PaulosAutoAPI/clientes/faturas/")
    static let listReportProblemURL = baseAPI?.appendingPathComponent("PaulosAutoAPI/equipamentos/avarias/")
    static let sendEquipmentUtilizationURL =
        baseAPI?.appendingPathComponent("PaulosAutoAPI/equipamentos/utilizacao")
    static let loginURL = baseAPI?.appendingPathComponent("Autenticacao/login")
    static let testTokenURL = baseAPI?.appendingPathComponent("Autenticacao/checkAll")
    static let problemReportURL = baseAPI?.appendingPathComponent("PaulosAutoAPI/equipamentos/avarias")
    static let dateFormatter1 : DateFormatter = {

        let dateFormatter = DateFormatter()
        dateFormatter.dateFormat = "yyyy-MM-dd'T'HH:mm:ss"
        return dateFormatter
    }()
    static let dateFormatter2 : DateFormatter = {

        let dateFormatter = DateFormatter()
        dateFormatter.dateFormat = "yyyy-MM-dd'T'HH:mm:ss.SSS"
        return dateFormatter
    }()
    static let boundary = "------------------------\(UUID().uuidString)"
}

enum APPError: Error {

    case networkError(Error)
    case dataNotFound
    case jsonParsingError(Error)
    case invalidStatusCode(Int)
    case requestEntityTooLarge
    case unauthorized
    case forbidden
    case InternalError
}

enum Result<T,Error> {

    case success(T)
    case failure(APPError)
}
```

Figura 26- Ficheiro ApiConstants

4.7.2. Reutilização de código

Outro ponto principal no desenvolvimento de um projeto é o reaproveitamento de funcionalidades já criadas, evitando desta forma o desenvolvimento repetitivo de funcionalidades em diferentes ViewControllers. Para garantir um ponto de contacto entre todas as ViewControllers foi criado um SubController designado de “ViewController”, a qual apresenta os métodos cuja utilização é generalizada durante a aplicação. Assim, todas as ViewControllers dão *extend* ao SubController com nomenclatura ViewController, como é possível observar no exemplo presente na Figura 27.

```
class InvoiceViewController: ViewController {
```

Figura 27- Exemplo de Extend ao SubController "ViewController"

O SubController dá *extend* desta forma ao UIViewController (Figura 28) pelo que qualquer ViewController passa a ter acesso quer aos métodos presentes no SubController bem como da classe nativa UIViewController.

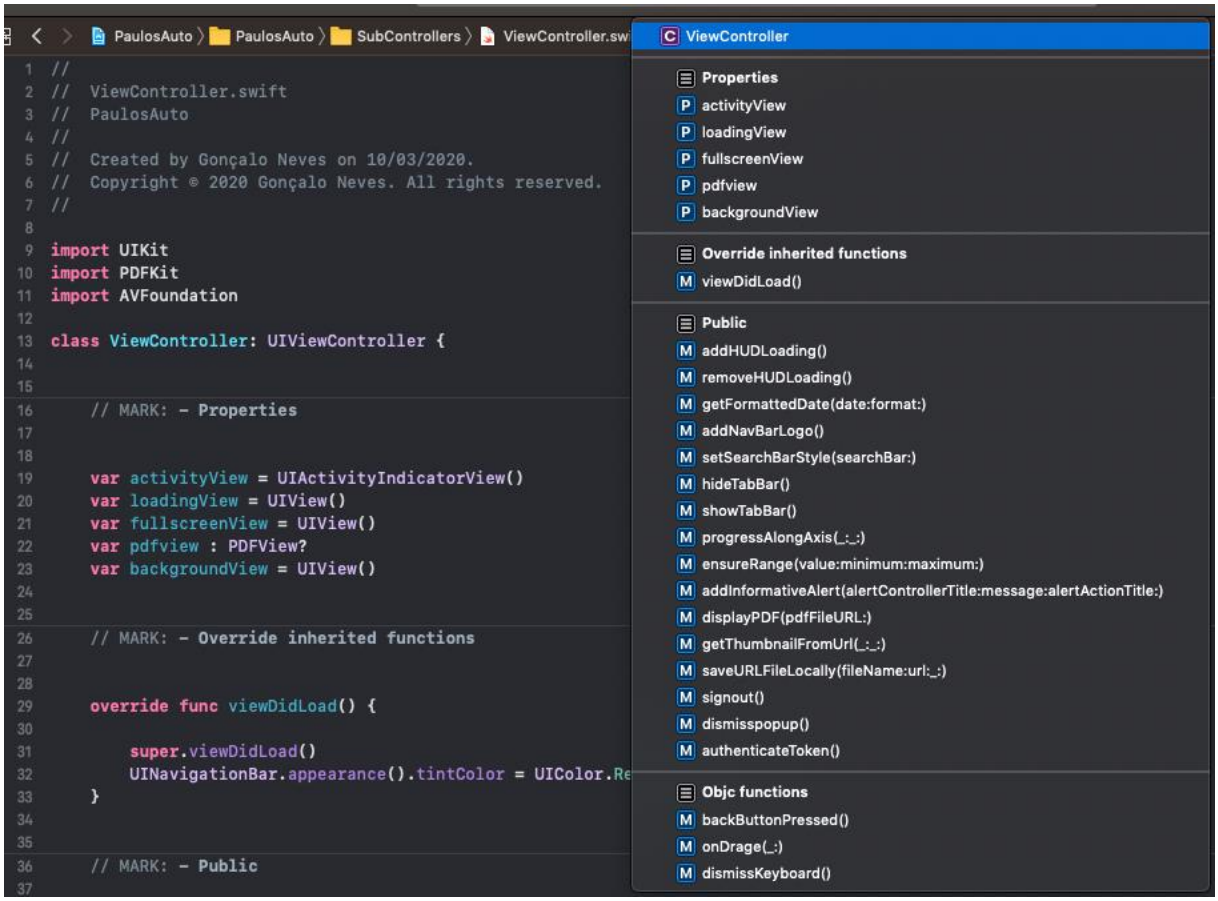


Figura 28 - SubController "ViewController"

4.7.3. Ligação à API

De modo a estabelecer a comunicação entre a aplicação e a API foi implementado um ponto único de contacto entre a aplicação e a API. É o caso da classe NetworkManager, a qual através de um método genérico fetchAPIData efetua *requests* através dos parâmetros recebidos:

- Url;
- Tipo de *request*;
- Parâmetros;
- Parâmetros *multipart*;
- Tipo de conteúdo;
- Token de acesso;
- Objeto modelo que se pretende ser retornado do método.

O método em questão realiza o decode[9] do json recebido da API para o tipo de dados (modelo) pretendido. Em caso de erro ou de código de resposta diferente de 200 é retornada uma variável de erro a qual dependendo do código de resposta pode ser constituída por diferentes mensagens de erro. Como é possível observar na Figura 29 a resposta de erro pode ser:

- Erro na conversão do json para o modelo pretendido;
- Acesso não autorizado;
- Acesso proibido;
- Informação não encontrada;
- Tamanho de dados enviados ultrapassa o limite estabelecido pela API;
- Erro interno.

[9] Decode - (Wikipedia, 2020) Decode é o processo de conversão de código em texto sem formatação ou em qualquer formato útil para processos subsequentes.

```
                switch httpResponse.statusCode {

                case 200:
                    DispatchQueue.main.async {

                        do {

                            let response = try JSONDecoder().decode(T.self, from: data)
                            completion(.success(response))
                        }
                        catch {
                            completion(.failure(APPError.jsonParsingError(error)))
                        }
                    }

                case 401:
                    completion(.failure(APPError.unauthorized))

                case 403:
                    completion(.failure(APPError.forbidden))

                case 404:
                    completion(.failure(APPError.dataNotFound))

                case 413:
                    completion(.failure(APPError.requestEntityTooLarge))

                case 500:
                    completion(.failure(APPError.InternalError))

                default:

                    completion(.failure(APPError.invalidStatusCode(httpResponse.statusCode)))
                }
            }
        }
        dataTask.resume()
    }
```

Figura 29- Resposta do método FetchAPIData

De modo a efetuar as chamadas ao NetworkManager foi criada uma classe *Request* com o nome RQ_{Nome do metodo} para cada um dos 8 tipos de pedidos diferentes à API já demonstrados (Figura 30).

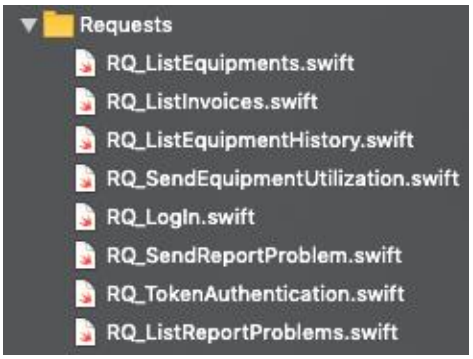


Figura 30- Classes Request

Desta forma, cada classe *Request* apresenta um método repos que prepara a chamada ao NetworkManager, a qual recebe a informação necessária para o processo, como é o caso do número de cliente, número de série de um equipamento, entre outros. Na Figura 31 é possível observar a *Request* Class para listagem de equipamentos (RQ_ListEquipments).

```
class RQ_ListEquipments {

    // MARK: - Public

    func repos(username: Int, _ completion: @escaping ([EquipmentModel]?, Error?) -> Void ) {

        let request = URLRequest(url: (ApiConstants.listEquipmentURL?.appendingPathComponent(String(username)))!)
        let url = request.url


        NetworkManager.fetchAPIData(url: url!) { (result: Result<[EquipmentModel], Error>) in
            switch result {

            case .success(let data):
                completion(data,nil)


            case .failure(let error):
                completion(nil,error)


            }
        }
    }
}
```

Figura 31- Método RQ_ListEquipments

Concluindo, o processo de ligação à API pode ser compreendido pelo fluxo presente na Figura 32.

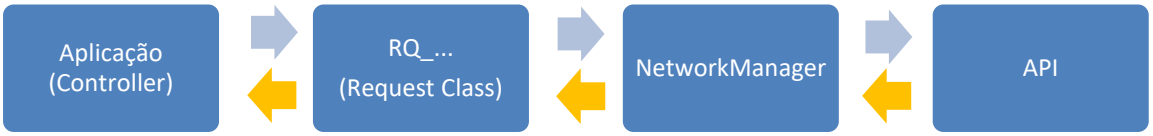


Figura 32- Fluxo de ligação à API

4.7.4. Feedback ao Utilizador

Outro dos pontos que foi tido em conta na conceção da aplicação foi a transmissão de informação ao utilizador de modo a esclarecer o mesmo de qualquer alteração que tenha efetuado. Este ponto apresenta-se como alertas ao utilizador sempre que se considere necessário, para tal e visto ser utilizado de uma forma generalizada pela aplicação foi criado um método (Figura 33) que recebendo o título, a mensagem e a ação mostra um alerta com a informação pretendida.

```
func addInformativeAlert(alertControllerTitle: String, message: String,alertActionTitle: String) {

    let alert = UIAlertController(title: alertControllerTitle, message: message, preferredStyle: UIAlertController.Style.alert)

    alert.addAction(UIAlertAction(title: alertActionTitle, style: UIAlertAction.Style.default, handler: nil))

    self.view.addSubview((alert.view)!)
    DispatchQueue.main.async {

        self.present(alert, animated: true, completion: nil)
    }
}
```

Figura 33- Método addInformativeAlert

De forma a exemplificar um alerta é dado como exemplo o alerta mostrado quando o processo de autenticação é efetuado sem sucesso devido a inexistência de internet, presente na Figura 34.

Figura 34- Exemplo de alerta

4.8. Fase de testes

Uma das fases mais importantes do desenvolvimento de uma aplicação é a fase de testes. A metodologia de testes utilizada para aumentar a eficácia de desenvolvimento e aumentar a qualidade do código produzido passou pelo teste de uma funcionalidade após a implementação da mesma, desta forma foi possível solucionar vários problemas encontrados.

Contudo, após o desenvolvimento de todas as funcionalidades e como se pode comprovar pelo Diagrama de Gantt presente no ANEXO B foi dedicada uma semana

para o teste e resolução de problemas encontrados. Para a realização destes testes foram repetidas todas as funcionalidades da aplicação tendo sido testadas mais exaustivamente as funcionalidades de registo de utilização e reporte de problema visto representarem as funcionalidades mais complexas e cuja implementação poderia originar mais problemas. Estes testes foram analisados recorrendo a 3 dispositivos diferentes, um emulador iPhone 11 pro max a correr iOS 13, pretendendo representar o equipamento maior e com versão iOS mais recente que a apple disponibiliza, um emulador iPhone 5s a correr iOS 11, representando no lado oposto, ou seja, o equipamento mais pequeno com a versão iOS mais antiga suportada e por fim um iPhone 5s físico a correr iOS 12, de modo a testar as funcionalidades que não são possíveis ser testadas recorrendo a um emulador, como é o caso da captura de provas visuais utilizando a câmara do dispositivo, ao contrário de um emulador android, um emulador iPhone não simula a aplicação da câmara.

Esta metodologia de teste do produto desenvolvido permitiu encontrar e solucionar um total de 32 problemas registados como *commits* no repositório da aplicação, foram adicionalmente detetados diversos problemas de menor relevância que por isso a sua resolução não se encontra registada como *commit*.

4.9. Aplicação Final

O desenvolvimento da aplicação focou-se na representação o mais semelhante possível dos protótipos produzidos e presentes no Capítulo 4.4, contudo perante a resposta do cliente aos protótipos foram efetuadas alterações que surgiram efeito na aplicação final. Adicionalmente, perante problemas técnicos ou maior facilidade de desenvolvimento recorrendo a funcionalidades já implementadas nativamente foram realizados ajustes o que leva a aplicação final a se diferenciar em relação aos protótipos elaborados. Neste capítulo será feita a demonstração de cada vista no estado final, enumerando as alterações perante a prototipagem e a razão das mesmas.

4.9.1. Autenticação

No que conta à vista que permite a autenticação do cliente não existem alterações em relação à prototipagem, apresentando-se uma vista que possibilita a autenticação com a inserção do email e password. Esta vista encontra-se presente na Figura 35.

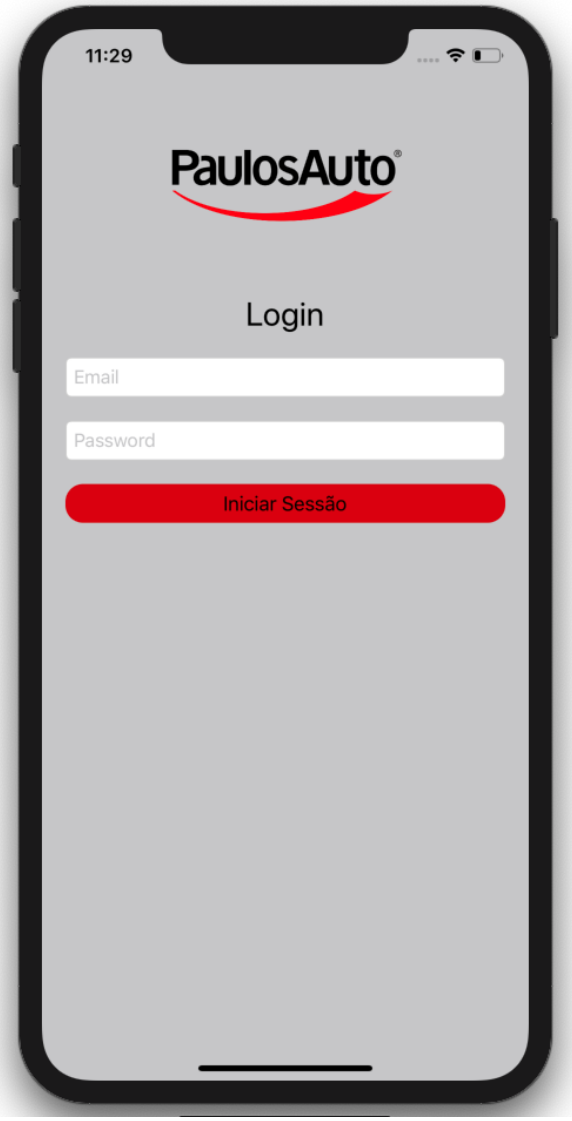


Figura 35- Aplicação Final – Autenticação

4.9.2. Listagem de Equipamentos

No que conta à listagem de equipamentos, e por decisão do cliente foi apenas utilizado um tipo de vista. Optou-se desta forma por uma vista híbrida entre lista e mosaico, como é possível observar na Figura 36. Adicionalmente foi adicionado um toggle com o nome SMP, o qual quando ativo apresenta apenas os equipamentos com contrato ativo.

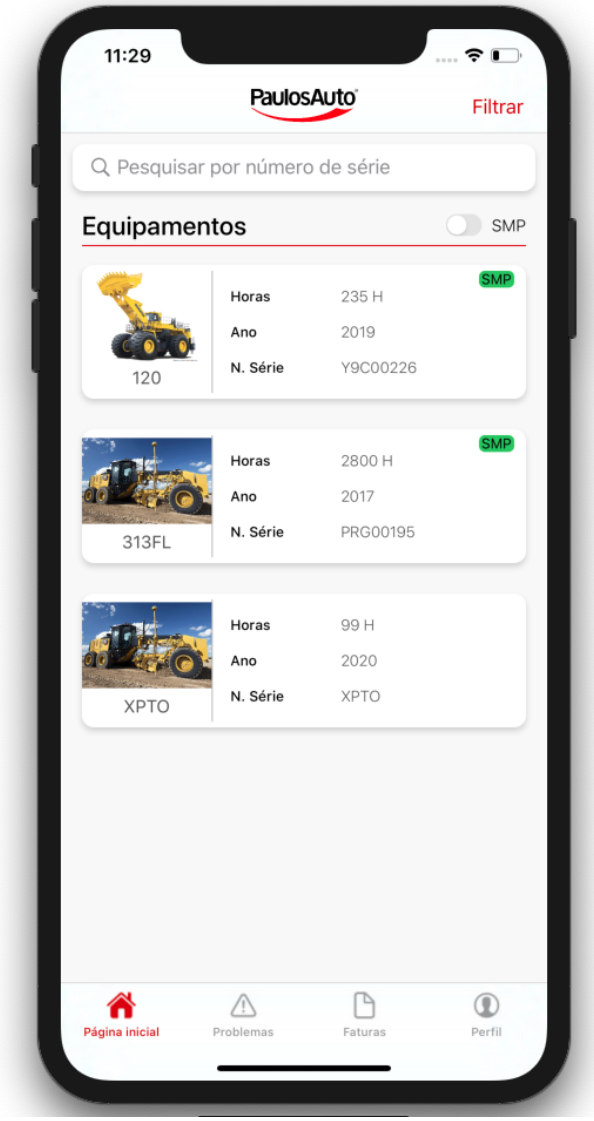


Figura 36- Aplicação Final - Listagem de Equipamentos

No menu de filtragem foi efetuado, por decisão do cliente, um maior aproveitamento de espaço, desta forma foi realizada a decisão de quando existirem até 4 opções no menu de filtragem o mesmo ocupa 50% do ecrã, com mais de 4 opções passa a ocupar 70% do ecrã e quando existirem mais de 7 opções ocupa a totalidade do ecrã. A vista corresponde ao menu de filtragem é desta forma apresentada na Figura 37 e Figura 38.

Figura 38- Aplicação Final - Menu de Filtragem - Ordenar por

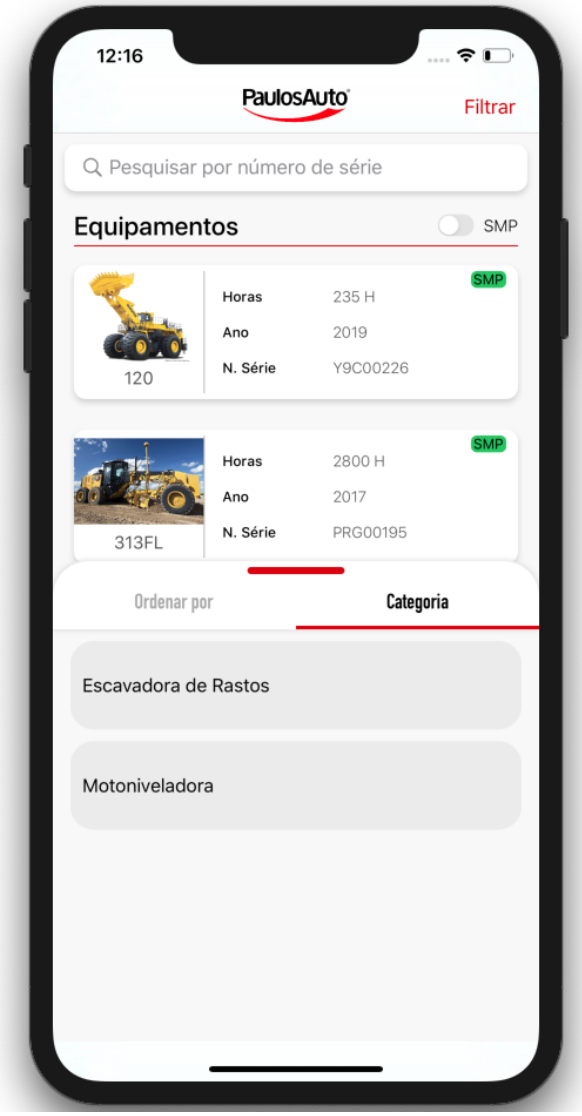


Figura 37- Aplicação Final - Menu de Filtragem- Categorias

4.9.3. Informações de Equipamento

Na página de Informações de um equipamento e devido à não existência na base de dados de informação relativa às caraterísticas de cada equipamento, foi alterada esta informação para contemplar a informação do contrato associado ao mesmo, adotando a estrutura visível na Figura 39.

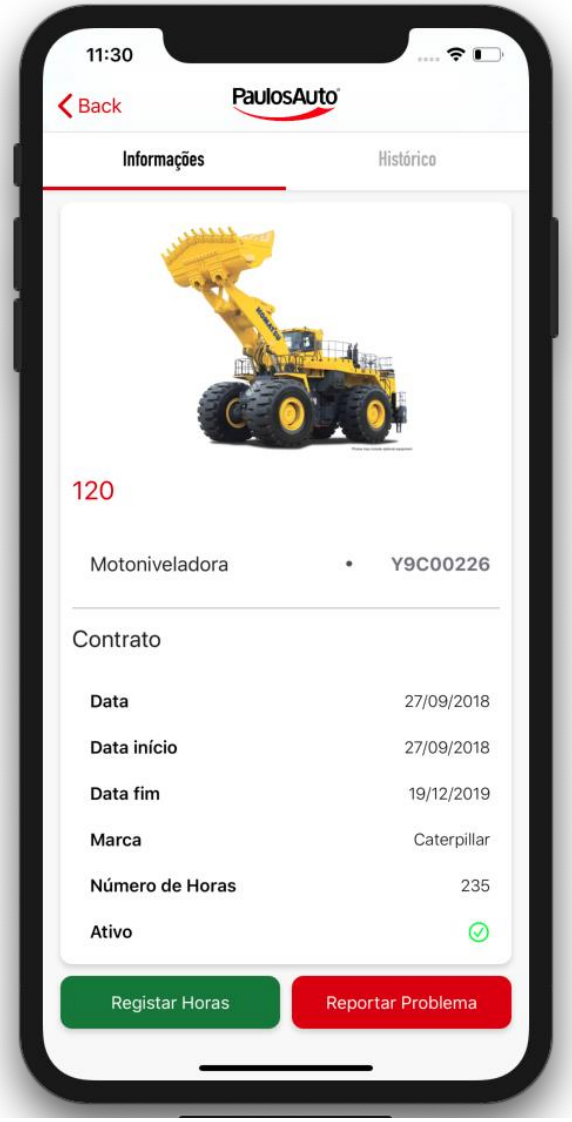


Figura 39- Aplicação Final - Informações de Equipamento

4.9.4. Histórico de Equipamento

No histórico individual de equipamento a pedido do cliente passa existir apenas um histórico de intervenções ao invés de um histórico com três tipos diferentes de acontecimentos: reporte de problema, intervenção e registo de utilização. Visível na Figura 40.

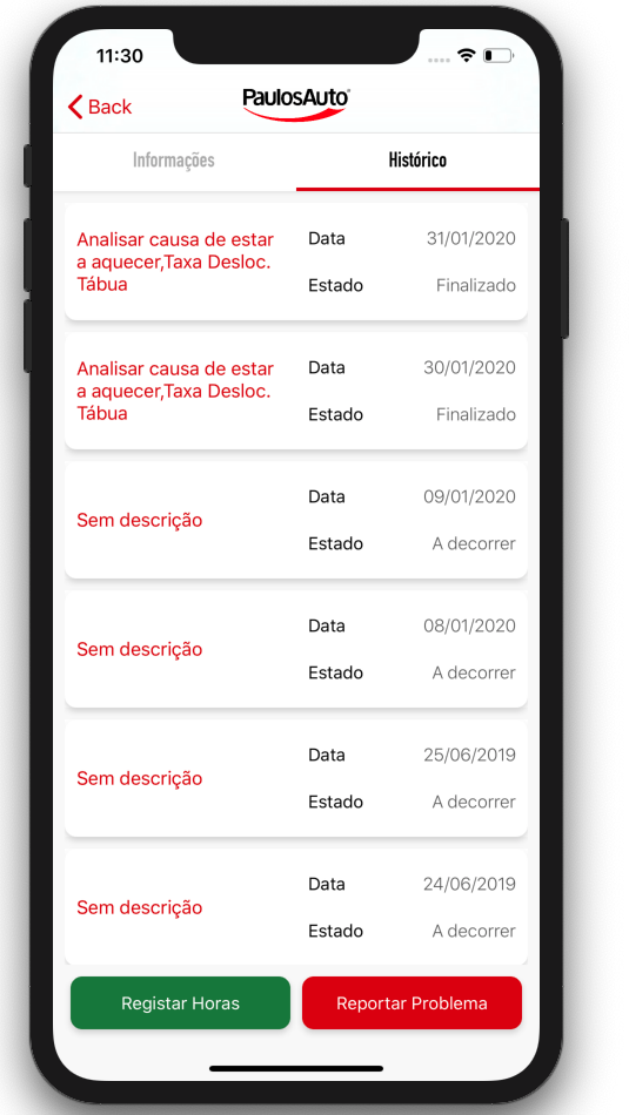


Figura 40- Aplicação Final - Histórico de Equipamento

4.9.5. Registo de Utilização de Equipamento

A vista de registo de utilização de equipamento é em toda semelhante ao protótipo concebido, apresentando-se como única diferença a funcionalidade criada no botão de submissão, o qual apenas se apresenta ativo quando o valor do número de horas indicadas se apresentar mais elevado que o último registo. Desta forma na Figura 41 é possível observar o botão em estado desativado pelo que apresenta menor opacidade.

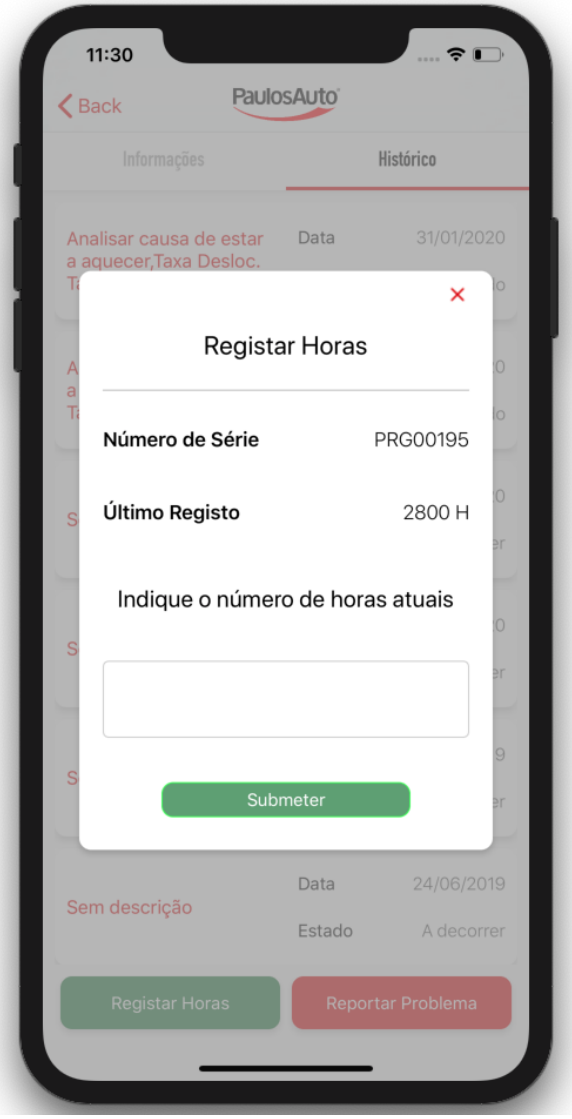


Figura 41- Aplicação Final - Registo de Utilização de Equipamento

4.9.6. Reporte de Problema com um Equipamento

Na vista de reporte de um problema passa a existir a informação do modelo do equipamento, bem como do número de série, de forma a diminuir o número de erros que possam advir da submissão de um reporte ao equipamento errado. Adicionalmente, é necessária a introdução do número de horas do equipamento aquando da deteção do problema. Os anexos introduzidos apresentam-se numa caixa com a pré-visualização dos mesmos, no caso das imagens aparecem as mesmas, no caso do vídeo é capturado o primeiro frame como imagem. De salientar que existe adicionalmente a funcionalidade de carregar num dos anexos para os visualizar

(Figura 42), quer se trate de imagens ou videos, bem como a possibilidade de excluir anexos. Esta vista pode ser observada na Figura 43.

Figura 43- Aplicação Final - Reporte de Problema com Equipamento

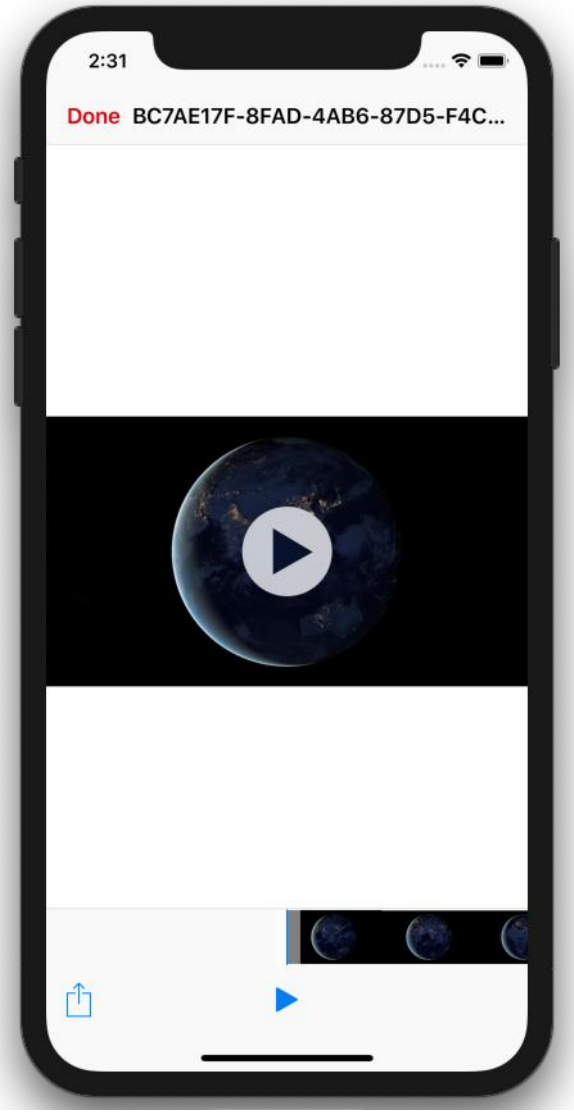


Figura 42- Pré-visualização de anexos

4.9.7. Listagem de Problemas Submetidos

Na listagem de reportes de problemas deixa de existir a barra de progresso de envio, isto deve-se ao facto de que esta funcionalidade não pode ser implementada recorrendo aos elementos já existentes nativamente, teria por isso de ter sido criada de raiz. Avaliando o tempo necessário para implementação desta funcionalidade com

a importância da mesma chegou-se em conjunto com os elementos envolventes no projeto à conclusão de que a implementação da mesma não era viável. Ainda nesta vista (Figura 44) foram realizadas mais mudanças estéticas, de forma a diferenciar a vista da listagem de equipamentos foi retirada a imagem do equipamento. Adicionalmente, de modo a fornecer mais informação ao utilizador passou a ser possível abrir o reporte e observar toda a informação introduzida aquando da criação do mesmo (Figura 45), podendo também abrir os anexos associados à semelhança da funcionalidade existente na vista de reporte de problemas.

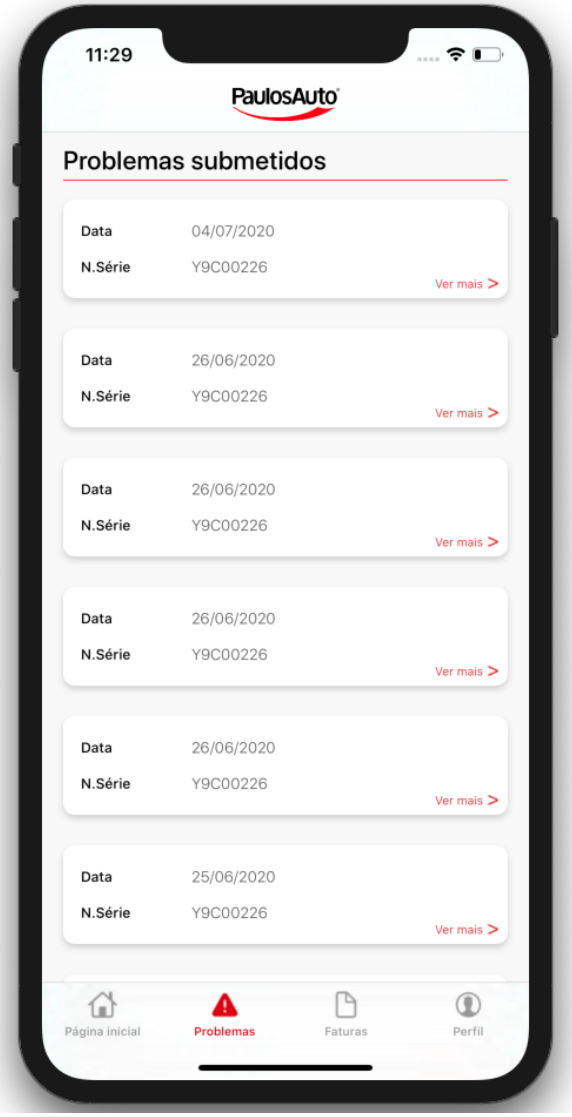


Figura 44- Aplicação Final - Listagem de Problemas Submetidos

Figura 45- Aplicação Final - Problema Submetido

4.9.8. Listagem de Faturas Pendentes

A vista relativa à listagem de faturas pendentes (Figura 47) apresenta a informação de cada “célula” na vertical, ao invés da estrutura prototipada com 2 elementos lado a lado, esta alteração foi efetuada de forma a tornar a interface da aplicação mais uniforme visto que este tipo de disposição de informação é utilizada no resto da aplicação. Adicionalmente passa a existir a possibilidade de não só visualizar as faturas em pdf como descarregar as mesmas (Figura 46).

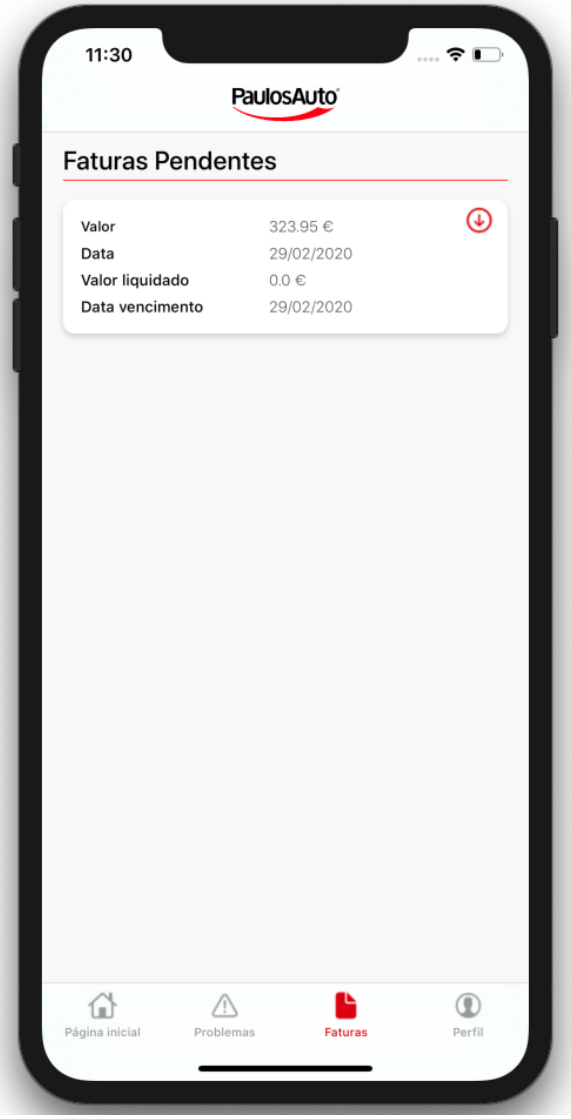


Figura 47- Aplicação Final - Listagem de Faturas Pendentes

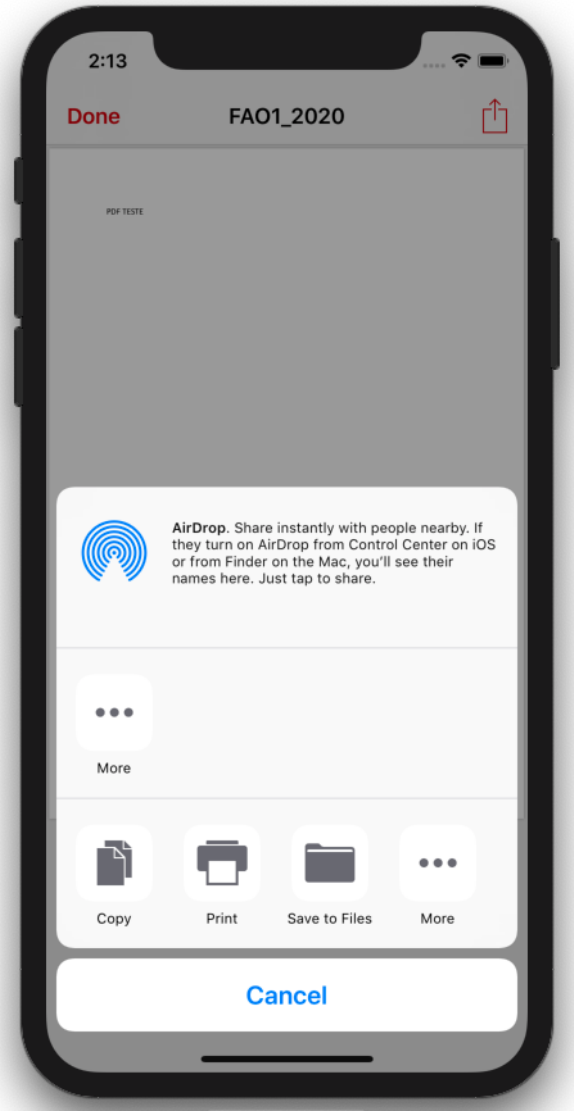


Figura 46- Aplicação Final - Pré-visualização de Faturas em PDF

4.9.9. Página Perfil

No perfil do utilizador, devido à não necessidade de existência de modo noturno a opção para essa funcionalidade desaparece, bem como a opção de notificações, uma vez que, apresentando-se como requisito opcional não existiu tempo para a sua implementação. Esta vista encontra-se representada na Figura 48.

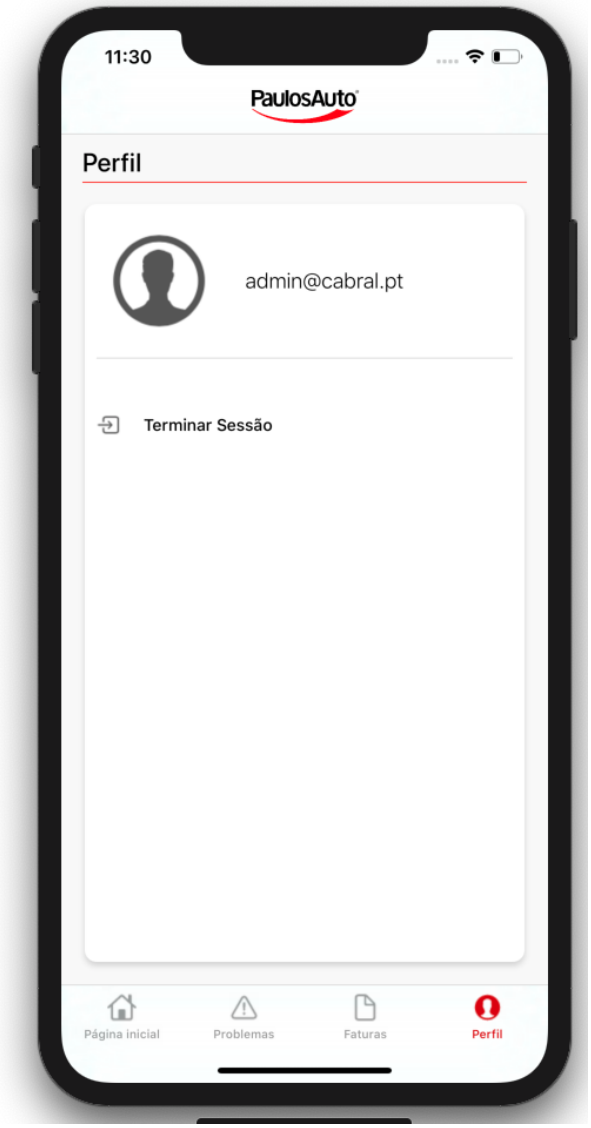


Figura 48- Aplicação Final - Página de Perfil

5. Conclusões e Trabalho Futuro

Com o fim do desenvolvimento da aplicação foi possível comparar o diagrama de Gantt real (ANEXO B) com o diagrama de Gantt esperado na fase de planeamento (ANEXO A). É possível desta forma visualizar várias diferenças entre os dois, os quais se conclui serem provocadas pelos seguintes pontos:

- Complicações relacionadas com a API, o que decorreu devido à desistência no período probatório do elemento do *back-end*. Pelo que, o desenvolvimento da mesma foi realizado pela empresa, tendo como se pode comprovar no diagrama de gantt real, existido duas API diferentes. A primeira representa uma API *mock*[10] de modo a permitir a funcionalidade necessária para o normal desenvolvimento da aplicação, enquanto se dava o desenvolvimento da API final a qual teve efeito no dia 1 de junho;
- Complexidade não prevista na implementação de algumas funcionalidades, como é o caso do NetworkManager e do *upload* de imagens e vídeos por multipart;
- Desenvolvimento sem recorrer a dependências externas. Todas as funcionalidades foram implementadas nativamente, sem recorrer, portanto, a código desenvolvido e mantido por terceiros;
- Desenvolvimento da aplicação utilizando as funcionalidades oferecidas na última versão do swift (5), a qual, dada a sua curta existência ainda não se encontrava de uma forma total utilizada pela empresa, o que por vezes diminuiu a eficiência da resposta a questões efetuadas;
- Trabalho Remoto provocado pelo isolamento social devido à pandemia de Covid-19, o que diminuiu a rapidez de comunicação entre os elementos pertencentes ao projeto PaulosAuto.

Estas razões levaram à entrega do projeto em época de recurso ao invés da entrega prevista em época normal. Contudo, e mesmo com as complicações encontradas o

[10] Mock - (Wikipedia, 2020) Na programação orientada a objetos, mocks são objetos simulados que imitam o comportamento de objetos reais de maneira controlada.

desenvolvimento da aplicação é considerado um sucesso visto terem sido implementadas todas as funcionalidades esperadas, bem como algumas funcionalidades extras, como é o caso da visualização de anexos. Todavia, tal como em todas as aplicações produzidas existe trabalho futuro, assentando-se o mesmo no desenvolvimento das principais funcionalidades:

- Implementação de modo offline;
- Implementação de notificações;
- Permissões de acesso a funcionalidades atendendo ao tipo de utilizador.

Queria agradecer o apoio prestado durante a realização do projeto ao orientador ESTGV Francisco Ferreira Francisco, ao orientador Softinsa José Carlos Dias, aos estagiários da Softinsa Rafael Pinto e Gabriel Silva, bem como a todos os colegas constituintes do projeto PaulosAuto, nomeadamente Rui Vide e João Diogo.

6. Referências

Apple. (07 de 07 de 2020). *datasource*. Obtido de developer.apple: https://developer.apple.com/documentation/uikit/uitableview/1614955-datasource

heflo. (10 de 06 de 2020). *Não confunda mais: Agile, Scrum e Kanban*. Obtido de heflo: https://www.heflo.com/pt-br/agil/agile-scrum-e-kanban/

PaulosAuto. (10 de 06 de 2020). *PaulosAuto*. Obtido de Linkedin: https://www.linkedin.com/company/paulosauto-lda./?originalSubdomain=pt

Softinsa. (10 de 06 de 2020). *Softinsa*. Obtido de Linkedin: https://www.linkedin.com/company/softinsa/about/

Wikipedia. (07 de 07 de 2020). *Asset(computer security)*. Obtido de Wikipedia: https://en.wikipedia.org/wiki/Asset_(computer_security)

Wikipedia. (07 de 07 de 2020). *Commit*. Obtido de Wikipedia: https://en.wikipedia.org/wiki/Commit_(version_control)

Wikipedia. (07 de 07 de 2020). *Decoding*. Obtido de Wikipedia: https://en.wikipedia.org/wiki/Decoding

Wikipedia. (07 de 07 de 2020). *Delegation(object-oriented programming)*. Obtido de Wikipedia: https://en.wikipedia.org/wiki/Delegation_(object-oriented_programming)

Wikipedia. (07 de 07 de 2020). *Framework*. Obtido de Wikipedia: https://pt.wikipedia.org/wiki/Framework

Wikipedia. (07 de 07 de 2020). *Intelligent code completion*. Obtido de Wikipedia: https://en.wikipedia.org/wiki/Intelligent_code_completion

Wikipedia. (07 de 07 de 2020). *Interface Builder*. Obtido de Wikipedia: https://en.wikipedia.org/wiki/Interface_Builder

Wikipedia. (07 de 07 de 2020). *JSON*. Obtido de Wikipedia: https://pt.wikipedia.org/wiki/JSON

Docker. (20 de 06 de 2020). *DockerToolbox*. Obtido de docs.docker: https://www.docs.docker.com/toolbox/overview/

Wikipedia. (07 de 07 de 2020). *Mock Object*. Obtido de wikipedia: https://en.wikipedia.org/wiki/Mock_object

7. Bibliografia

Apple. (25 de 04 de 2020). *Xcode*. Obtido de developer.apple: https://developer.apple.com/documentation/xcode/

Apple. (07 de 07 de 2020). *Swift*. Obtido de developer.apple: https://www.developer.apple.com/swift/

codementor. (10 de 05 de 2020). *Swift Package Manager vs CocoaPods vs Carthage for All Platforms*. Obtido de codementor: https://www.codementor.io/blog/swift-package-manager-5f85eqvygj

codewithchris. (01 de 05 de 2020). *Swift Tutorial: How To Master The Fundamentals (2020)*. Obtido de codewithchris: https://www.codewithchris.com/swift-tutorial-complete/

JetBrains. (25 de 04 de 2020). *Quick start guide - AppCode*. Obtido de jetbrains: https://www.jetbrains.com/help/objc/appcode-quick-start-guide.html

medium. (30 de 04 de 2020). *6 aspectos essenciais para decidir entre aplicações mobile híbridas e nativas*. Obtido de medium: https://medium.com/@wfelix/6-aspectos-essenciais-para-decidir-entre-aplicações-mobile-h%C3%ADbridas-e-nativas-51bce0dace68

medium. (30 de 04 de 2020). *Creating a Mobile App in 2020: Native vs. Cross-Platform Development*. Obtido de medium: https://medium.com/simbirsoft/creating-a-mobile-app-in-2020-native-vs-cross-platform-development-d90f25cef188

opensource. (20 de 06 de 2020). *What is Docker?* Obtido de opensource: https://www.opensource.com/resources/what-docker

Quental, C. (2016). *Normas e Orientações versão 16.* Viseu: Carlos Quental.

raygun. (30 de 05 de 2020). *Understanding native app development - what you need to know in 2019*. Obtido de raygun: https://raygun.com/blog/native-app-development/

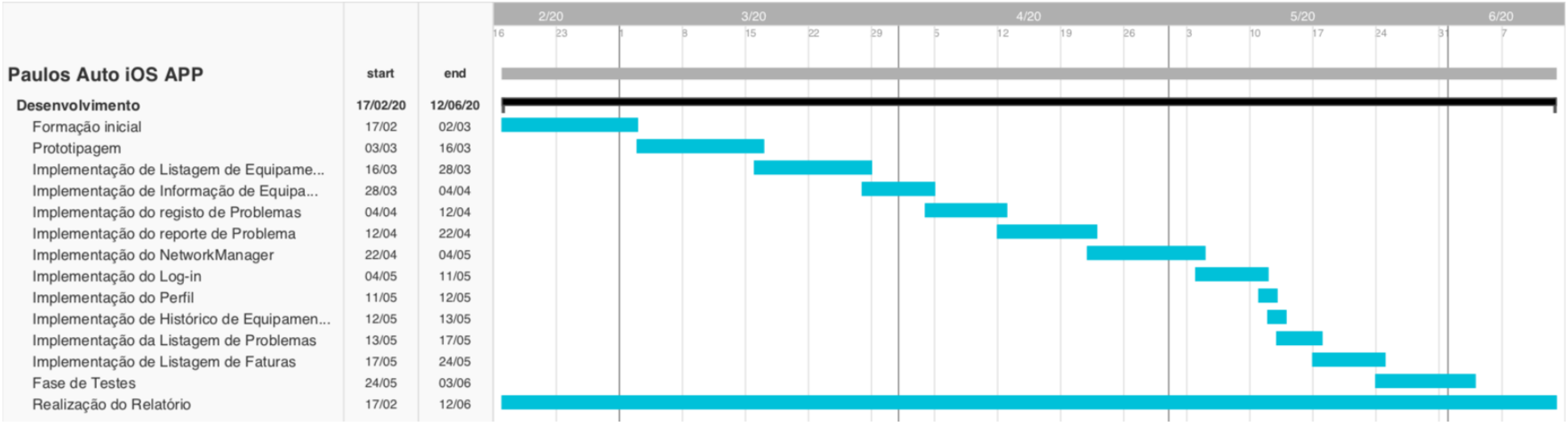
raywenderlich. (10 de 05 de 2020). *Carthage Tutorial: Getting Started*. Obtido de raywenderlich: https://www.raywenderlich.com/416-carthage-tutorial-getting-started

raywenderlich. (10 de 05 de 2020). *CocoaPods Tutorial for Swift: Getting Started*. Obtido de raywenderlich: https://www.raywenderlich.com/7076593-cocoapods-tutorial-for-swift-getting-started

Wikipedia. (15 de 06 de 2020). *Trello*. Obtido de wikipedia: https://www.en.wikipedia.org/wiki/Trello

ANEXO A

No presente anexo é possível observar o diagrama de gantt criado na fase de planeamento, o mesmo pretende representar o ritmo de desenvolvimento esperado.



Paulos Auto iOS APP	start	end
Desenvolvimento	**17/02/20**	**12/06/20**
Formação inicial	17/02	02/03
Prototipagem	03/03	16/03
Implementação de Listagem de Equipame...	16/03	28/03
Implementação de Informação de Equipa...	28/03	04/04
Implementação do registo de Problemas	04/04	12/04
Implementação do reporte de Problema	12/04	22/04
Implementação do NetworkManager	22/04	04/05
Implementação do Log-in	04/05	11/05
Implementação do Perfil	11/05	12/05
Implementação de Histórico de Equipamen...	12/05	13/05
Implementação da Listagem de Problemas	13/05	17/05
Implementação de Listagem de Faturas	17/05	24/05
Fase de Testes	24/05	03/06
Realização do Relatório	17/02	12/06

ANEXO B

No presente anexo é possível observar o diagrama de gantt obtido no desenvolvimento da aplicação, podendo a partir do mesmo observar os períodos de realização das várias tarefas.