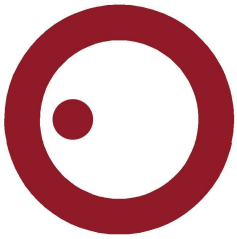


escola superior de tecnologia e gestão
instituto politécnico de leiria

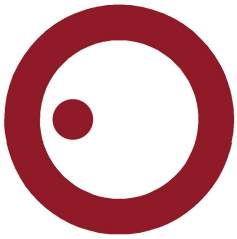
Instituto Politécnico de Leiria
Escola Superior de Tecnologia e Gestão
Departamento de Engenharia Informática
Mestrado em Eng.ª Informática – Computação Móvel

APLICAÇÃO MÓVEL EMPRESARIAL - PORTAL DE GESTÃO FINANCEIRA FACT.PT

ESTUDANTE LUÍS MANUEL MARTINS CAMPOS

Leiria, novembro de 2021

[30 de novembro de 2021 at 23:46 –]



escola superior de tecnologia e gestão
instituto politécnico de leiria

Instituto Politécnico de Leiria
Escola Superior de Tecnologia e Gestão
Departamento de Engenharia Informática
Mestrado em Eng.ª Informática – Computação Móvel

APLICAÇÃO MÓVEL EMPRESARIAL - PORTAL DE GESTÃO FINANCEIRA FACT.PT

ESTUDANTE LUÍS MANUEL MARTINS CAMPOS
Número: 2192735

Projeto realizado sob orientação do Professor Filipe dos Santos Neves (fneves@ipleiria.pt).

Leiria, novembro de 2021

[30 de novembro de 2021 at 23:46 –]

AGRADECIMENTOS

Gostaria primeiramente de agradecer a duas pessoas bastante importantes que permitiram que este relatório se concretizasse. Agradecer ao orientador deste projeto o professor Filipe Neves pela sua enorme disponibilidade e acompanhamento na escrita deste relatório ao longo do ano onde esteve sempre predisposto a ajudar e sempre acreditou em mim em tempos mais difíceis da escrita do relatório, pelas discussões que tivemos que muito me fizerem crescer a nível pessoal, as exigências e pela sua paciência.

Agradecer também ao Hugo Pinto, responsável pela empresa GoFact, em primeiro lugar pela oportunidade que me deu de trabalhar neste magnífica empresa onde adquiri imenso conhecimento em tão pouco tempo com ele, agradecer por todos os desafios que me propôs para eu evoluir a nível técnico e por toda a disponibilidade que apresentou, por sempre se predispor a ajudar seja a nível profissional como a nível pessoal, sempre serei agradecido.

Agradecer também a todos os elementos da empresa GoFact, ao Nuno, Vítor, Isaac, Lucas e Ricardo que me fizerem sempre sentir muito integrado e com um ambiente excecional e que, de uma forma ou outra, me ajudaram a crescer a nível profissional nesta empresa.

Agradecer também a todos os docentes do Instituto Politécnico de Leiria com quem fui convivendo ao longo dos anos e que me ajudaram muito a tornar-me o profissional que sou hoje.

E por último, mas não menos importante, agradecer à minha família, aos meus amigos e, com especial agradecimento, à minha namorada Filipa que sempre estiverem presentes quando necessitei de ajuda ao longo deste percurso que tanto aprendi.

Obrigado.

RESUMO

Este relatório apresenta o trabalho desenvolvido da implementação de uma aplicação móvel para o portal de gestão comercial, Fact.pt, pertencente à empresa GoFact. O Fact.pt é um portal *web-based* com uma solução de faturação integrada, certificada pela Autoridade tributária, que possui um conjunto de soluções para o auxílio dos processos do dia a dia da gestão de um negócio. Surgiu a necessidade de desenvolver uma aplicação móvel que permite a um empresário acompanhar a evolução do seu negócio, visualizando *dashboards* de informação financeira relevante, consultar as suas contas bancárias e respetivos documentos reconciliados, assim como receber notificações em tempo real sobre todos os eventos que estão a acontecer na sua empresa.

Com isto ao longo do relatório serão descritas as metodologias utilizadas que foram fulcrais para desenvolver o projeto, sempre orientadas a uma alta disponibilidade e escalabilidade, que são fatores determinantes para o sucesso nas empresas de tecnologias de informação. É apresentado o processo de recolha das tecnologias, e suas comparações, para serem utilizadas no desenvolvimento do projeto. É também descrito com mais detalhe todas as particularidades da tecnologia escolhida (Flutter), culminando com a descrição do processo integral de desenvolvimento, desde todas as funcionalidades implementadas até aos processos de *build* e *deploy* da aplicação.

O desenvolvimento será avaliado com base nos resultados dos testes unitários implementados ao código da aplicação, e com base nos comentários de agrado feitos pelos utilizadores na loja de aplicações que testaram a aplicação. São ainda descritas as conclusões da aplicação, onde constatam que o objetivo do projeto foi concretizado com sucesso, abrindo portas a um contínuo de novas implementações de elevado interesse para o negócio da empresa.

Palavras-chave: Aplicação móvel, Flutter, CI/CD, GitLab, Android

A B S T R A C T

This report presents the work done to implement a mobile application for the commercial management portal, Fact.pt, belonging to the company GoFact. Fact.pt is a web-based portal with an integrated invoicing solution, certified by the Tax Authority, which has a set of solutions to help the day-to-day processes of running a business. The need arose to develop a mobile application that allows an entrepreneur to follow the evolution of his business, viewing dashboards of relevant financial information, consulting his bank accounts and respective reconciled documents, as well as receiving real-time notifications about all the events that are happening in his company.

With this, throughout the report, the methodologies used will be described that were central to the development of the project, always oriented towards high availability and scalability, which are determining factors for success in information technology companies. The process of collecting the technologies, and their comparisons, to be used in the development of the project is presented. It is also described in more detail all the particularities of the chosen technology (Flutter), culminating with a description of the integral development process, from all implemented functionalities to the application's build and deploy processes.

The development will be evaluated based on the results of the unit tests implemented to the application's code, and based on the comments made by users in the application store who tested the application. The application's conclusions are also described, where they verify that the project's objective was successfully achieved, opening the door to a continuum of new implementations of great interest to the company's business.

Keywords: *Mobile Application, Flutter, CI/CD, GitLab, Android*

ÍNDICE

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS

AAB	*Android App Bundle.*
AOT	*Ahead Of Time.*
API	*Application Programming Interface.*
APK	*Android Package.*
CD	*Continuous Delivery / Continuous Deployment.*
CI	*Continuous Integration.*
CSS	*Cascading Style Sheets.*
HTTP	*Hypertext Transfer Protocol.*
IA	Inteligência Artificial.
IDE	*Integrated Development Environment.*
IVA	Imposto sobre Valor Acrescentado.
JIT	*Just In Time.*
MVC	*Model-View-Controller.*
MVP	*Minimum Viable Product.*
SDK	*Software Development Kit.*
SVN	*Subversion.*
UI	*User Interface.*

1

INTRODUÇÃO

O trabalho desenvolvido e descrito neste relatório foi inserido na opção de Projeto do Mestrado em Engenharia Informática – Computação Móvel. Desta forma, foi realizado um estágio de 9 meses no âmbito dos Estágios ATIVAR.PT do Instituto do Emprego e Formação Profissional (IEFP) na empresa GoFact - Tecnologias de Informação, Lda. Nesta empresa foi proposto o desenvolvimento de um projeto de uma aplicação móvel.

A GoFact é uma empresa de Tecnologias de Informação, sediada em Braga, com experiência no desenvolvimento de soluções digitais integrais *web-based* com alta escalabilidade, onde é feito a adaptação de modelos de negócio ao online através de websites, lojas online, soluções móveis Android/iOS, desenho e especificação de software, aplicações de gestão interna, publicidade e marketing. Aqui é privilegiada na equipa de trabalho uma mentalidade e visão de um empresário no desenvolvimento de aplicações, com o enorme foco na minimização do esforço de adaptação e curva de aprendizagem dos seus clientes ao analisar as necessidades dos seus negócios. O foco principal da empresa GoFact é no desenvolvimento de um produto próprio de marca registada denominado Fact.pt.

O Fact.pt é um portal *web-based*, certificado pela Autoridade Tributária, que permite criar condições para que todos os pequenos negócios e novos empresários possam emitir faturas sem grande esforço e sem perder muito tempo, com custos controlados, e ao mesmo tempo ter uma noção real e simples da situação comercial das suas empresas. De seguida irão ser enumeradas os quatro principais produtos do portal Fact.pt e algumas das suas funcionalidades:

- **Gestão Comercial:**
  - Composição e emissão de faturas, recibos e de outros tipos de documentos onde tudo é guardado em rascunho se o utilizador perder acesso com acesso imediato à lista de clientes e de produtos disponíveis;
  - Todos os documentos possuem um URL único atribuído para uma fácil visualização e download;

  - *Dashboards* para cada produto específico para analisar a progressão das vendas entre outros dados relevantes;
  - *Dashboards* para cada cliente específico para analisar a evolução das vendas entre outros dados relevantes.

- **Gestão Documental:**
  - Possibilidade de converter faturas em papel num formato digital. Isto é possível através da digitalização dos documentos, onde atuará um forte motor de inteligência artificial ficando, assim, estes armazenados na aplicação. Desta maneira, estes documentos podem serem acedidos no telemóvel ou no computador mais tarde;
  - Gestão dos fornecedores;
  - *Dashboard* com painéis gráficos que permitem acompanhar a evolução das despesas da empresa;
  - Entradas: é uma funcionalidade que consiste em que o utilizador tenha um email: <nif>@docs.fact.pt e todos os emails que são envidados para este email são formatados e apresentados numa lista de Entradas com os respetivos anexos/documentos que podem ser lançados para fazerem parte da Gestão Documental.
- **Gestão Bancária:**
  - Possibilidade de ter todas as contas bancárias da empresa e respetivos movimentos numa única plataforma. A configuração com a entidade bancária é apenas necessária fazer uma vez porque existe uma sincronização automática todos os dias para atualizar essa mesma informação;
  - Conciliação avançada entre os movimentos bancários e um documento da Gestão documental com um sistema avançado de sugestões para que estes fiquem interligados;
- **Performance e Tesouraria:**
  - Dashboard com um conjunto de gráficos que permitem entender a performance da empresa através de aspetos relevantes como a comparação despesa/receita ao longo do tempo, *burnrate*, *cash-flow* entre outros aspetos.

O Fact.pt é um portal extremamente completo com um leque de funcionalidades abrangente e com grande potencial de crescimento. Como foi referido anteriormente, o Fact.pt é uma aplicação *web-based* e como é sabido, a utilização de aplicações móveis

tem aumentado consideravelmente ao longo dos anos muito pela sua portabilidade onde é possível ser acedido em todo lado, a qualquer hora apenas num smartphone ou qualquer outro aparelho portátil. Posto isto, surgiu a necessidade de desenvolver uma aplicação móvel para o produto Fact.pt em que o utilizador tivesse uma opção mais simples do portal web, mas que reunisse funcionalidades que permitissem acompanhar os resultados da empresa a nível de vendas, despesas entre outros aspetos. Nunca foi ideia de a aplicação móvel substituir totalmente o que já era feito no portal, mas sim ser uma ferramenta de interação rápida sobre as operações mais frequentes. O pretendido era que o utilizador tivesse uma rápida visualização dos progressos da empresa de maneira simples, maioritariamente através de dashboards gráficos. Tratando-se de uma aplicação móvel foi tirado o partido de utilizar algumas das vantagens do smartphone por exemplo, para receber qualquer tipo de notificações em tempo real, tirar uma foto de forma rápida a um documento que seria mais tarde analisado pelo motor de IA (Inteligência Artificial), etc.

Em relação a trabalho relacionado, depois de uma rápida análise no mercado, foi percebido que não havia nenhuma aplicação com as mesmas funcionalidades e tão completas em relação ao que haveria de ser esta aplicação.

Assim, o presente relatório ficou estruturado da seguinte forma:

- no Capítulo 2 serão descritas as metodologias que permitiram desenvolver o trabalho de forma mais eficiente orientadas sempre a alta disponibilidade e escalabilidade;
- no Capítulo 3 será relatado todo o processo de aprendizagem de novos conhecimentos para a implementação da aplicação e também o processo de desenvolvimento das funcionalidades;
- no Capítulo 4 será descrito a avaliação do desenvolvimento, da avaliação aos testes unitários feitos ao código do projeto e descrever os comentários feitos pelos utilizadores que utilizaram a aplicação;
- no Capítulo 5 será descrito o resultado da aplicação, se o objetivo planeado foi cumprido totalmente, se existem aspetos a melhorar e por fim descrever o trabalho futuro.

2

METODOLOGIAS

2.1 O GIT

Desde o início do desenvolvimento de um projeto, um dos pontos fulcrais é pensar sobre controlo e cooperação do trabalho de desenvolvimento da aplicação em que se está a trabalhar. Posto isto, quando o projeto começa a crescer tanto a nível de linhas de código como a nível de elementos da equipa de programadores no qual estão a trabalhar, é essencial ter uma solução onde a equipa possa estar a trabalhar toda ao mesmo tempo no projeto fazendo as suas alterações sem que haja conflito de tarefas desempenhadas. Um dos grandes problemas que ocorre quando não é usado uma solução de controlo, é quando o elemento X de uma equipa de programadores está a trabalhar no mesmo ficheiro que um elemento Y da mesma equipa e o elemento X faz upload do ficheiro para uma pasta do projeto central onde todos têm acesso. Nesta situação, quando o elemento Y vai para fazer o upload do mesmo ficheiro vai reparar que existem conflitos porque os dois estiveram a trabalhar naquele ficheiro ao mesmo tempo surgindo assim problemas de consistência nos conteúdos do ficheiro. Para resolver este tipo de problemas surgiu o Git.

O Git é um Sistema de Controlo de Versões Distribuído, criado por Linus Torvalds em 2005, o mesmo criador do Linux. Este sistema permite registar as alterações que ocorrem no código fonte de um projeto, permitindo assim que os ficheiros possam ser alterados ao mesmo tempo por mais que um elemento de uma equipa o que evita que haja conflitos e inconsistências resultantes das alterações realizadas. Mas se houver algum tipo de problema, o Git permite voltar a uma versão anterior do código fonte de forma rápida. Outro ponto bastante importante do Git, que difere do seu principal concorrente, SVN, é que permite criar um *snapshot* da pasta central do projeto onde está o código fonte, que no mundo do Git designa-se por branch. Esta funcionalidade é bastante usada quando um dos elementos está a desenvolver uma nova funcionalidade (*feature*) na aplicação, mas não quer ainda que essas alterações entrem no projeto principal enquanto a funcionalidade por si só não estiver com o seu denvolvimento concluído, ou seja, já tenha sido implementada e testada. Ele cria uma *branch* onde trabalha sempre nela, e quando a funcionalidade estiver completa,

esta *branch* pode ser fundida com a linha principal de desenvolvimento para que faça parte do projeto principal.

De forma resumida, todo o sistema Git funciona como um sistema de *branches* (ramificações). A Figura 1 mostra como se representa o *workflow* do Git, em que cada círculo colorido representa uma alteração de código feita, designada por *commit*. Cada *commit* é composto pelas alterações feitas nos vários ficheiros, por exemplo, e por uma descrição sucinta das alterações feitas, além de metadados (data, autor, etc.). Como é possível ver na 1, cada elemento da equipa pode estar a trabalhar em *branches* diferentes ao mesmo tempo como é o caso da *branch* azul e a da *branch* laranja, e enquanto o elemento da *branch* azul está a fazer as suas alterações, o elemento da equipa que está a trabalhar na *branch* laranja não consegue ver essas mesmas alterações pois estão em *branches* diferentes. A *branch* verde, designada por *master*, é a *branch* principal do projeto, que geralmente representa o código fonte que está em produção. O processo da criação de uma *branch*, neste caso da *branch* azul, é representada pela linha preta entre a primeira bola verde e a primeira bola azul, e depois de três *commits* é feito o *merge* (ou convergência) da *branch* azul com a *branch master*. Um repositório em Git representa a pasta do projeto que para ser considerado um repositório tem que ter na sua raiz um ficheiro oculto .git. Outros dois termos bastante importante no sistema Git são o *push* e o *pull*. Para perceber estes dois termos, é importante perceber que cada *commit* feito representa um registo das alterações no repositório local do desenvolvedor, e não no repositório remoto que é acessível por toda a gente. Para o código que está no repositório local ser atualizado no repositório remoto é necessário fazer um *push*, que neste caso irá enviar o código do repositório local para o repositório remoto. O *pull*, por outro lado, representa o processo de atualização do repositório local. É feito um *merge* do repositório remoto com o repositório local, onde geralmente neste processo é necessário resolver alguns conflitos pois um dos ficheiros que foi alterado na *branch*, pode ter sido alterado também noutras *branches* desde o último *pull*.

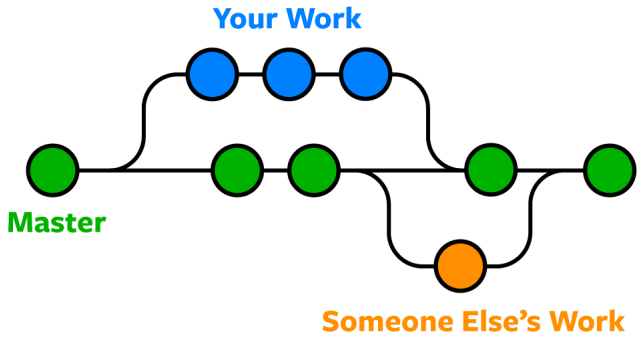


Figura 1: Exemplo de um *workflow* Git

2.2 GIT WORKFLOW

É bastante comum quando se está a trabalhar sozinho num projeto utilizar apenas uma *branch*, geralmente a *branch master* e isto não se pode considerar totalmente errado pois assim é facilmente mantido. O problema disto é quando o número de elementos da equipa que está a trabalhar nesse mesmo projeto cresce, e bastam no mínimo dois elementos para isto tornar-se rapidamente um problema por conta de conflitos, entre outras dificuldades. Para resolver este problema, é importante ter um *workflow* de Git organizado.

Os dois *workflows* mais utilizados são o GitHub *Flow* e o GitFlow. Posteriormente é comum a equipa de programadores adaptar um dos workflows ao seu critério. O GitHub *Flow*, resumidamente, tem apenas uma *branch master* que representa o estado da aplicação em produção, e sempre que é necessário fazer uma nova funcionalidade, cria-se um *branch* com a denominação *feature* e quando estiver completa dá-se então o *merge* para a *branch master*. O GitFlow é um pouco mais complexo e uma adaptação deste é o que está a ser utilizado em todos os projetos da empresa GoFact, por isso, automaticamente foi o *workflow* escolhido para manter este projeto.

O GitFlow foi anunciado pela primeira vez em janeiro de 2010. O GitFlow possui o que são chamadas de *main branches* (*branches* principais): *master* e *develop*. A *branch master* representa o estado da aplicação quando está em produção, e não é permitido realizar commits diretamente nesta *branch* pelos desenvolvedores. Cada *commit* nesta *branch* apresenta uma *release* em produção, e cada *release* contém uma *tag* com o número da versão da aplicação. Uma *tag* em Git é uma referência textual que aponta para um ponto específico na *timeline* do Git, que na maioria das vezes é utilizado para referir a versão de uma aplicação, por exemplo, v1.1.0. A *branch develop* representa o estado do projeto que contém as últimas alterações de desenvolvimento pronta para a próxima *release*. É uma *branch* de integração entre a *branch master* e as *branches features* que irão ser faladas mais à frente. Não deve ser permitido *commits* diretos para esta *branch*. Segundo o *workflow* Git da empresa, só são permitidos *commits* nas *branches master* e *develop* através de *merges/pull requests* e só podem ser aceiteis pelos administradores do repositório (eventualmente passarem testes). Um *merge request* (também conhecido como *pull request*, dependendo da plataforma que está a ser utilizada) é sempre constituído por uma *branch* origem e por uma *branch* destino e consiste num método de verificar todo o código que foi desenvolvido na *branch* origem, e depois de revisto se o *merge*

request for aceite pelo responsável, é feito o *merge* do código da *branch* origem na *branch* destino.

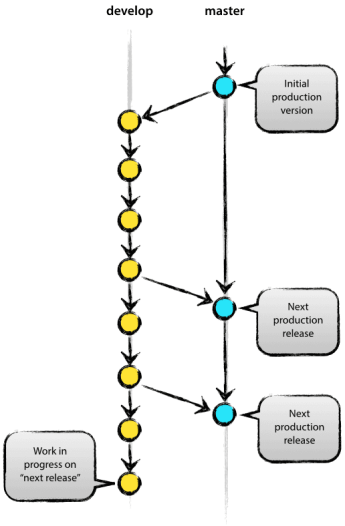


Figura 2: Workflow Git das *branches master* e *develop*

Além das *main branches*, existem *branches* de trabalho: as *supporting branches* (ramificações de suporte), que são *branches* que irão auxiliar no processo de desenvolvimento em paralelo dos elementos da equipa. Ao contrário das *branchs* principais, as branches de suporte têm um tempo de vida útil limitada, pois elas são utilizadas durante o desenvolvimento de código e funcionalidades e quando não são mais necessárias elas são removidas. As *branches* de suporte são constituídas por: *feature branches*, *release branches* e *hotfix branches*. Cada uma destas *branches* tem uma função específica e tem regras específicas para qual deve ser a *branch* origem, e qual deve ser a *branch* destino para ocorrer o *merge*.

As *branches features* são usadas para desenvolver novas *features* ou *bugfixes*. Geralmente é criada uma *feature branch* para cada *issue* no projeto, e depois de completa essa *issue* é feito então o *merge/pull request* para a *branch develop* para estar pronta para uma nova *release* e de seguida a *branch feature* deve ser eliminada. As regras que a *branch feature* devem respeitar são as seguintes:

- Apenas deve ser criada a partir da *branch develop*;
- Apenas deve dar *merge* para a *branch develop*;
- A convenção do nome da *branch* deve respeitar a seguinte regra: *feature*/nome da *issue*.

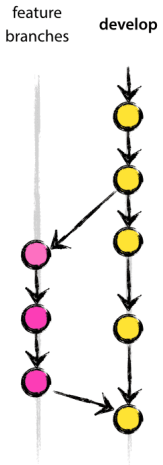


Figura 3: Workflow Git das *branches feature* e *develop*

As *branches releases* auxiliam na preparação de uma nova *release*. É exatamente antes da criação de uma *branch release* que é percebido qual será a próxima versão da aplicação e que irá começar o processo de criação de uma nova *release*. O momento certo para criar uma *branch release* é quando a *branch develop* reflete o estado que é desejável para uma *release*. Segundo o criador original do GitFlow é permitido nesta *branch* efetuar os últimos ajustes e a correção de alguns bugs. Mas segundo o fluxo seguido pela empresa, apenas é permitido um *commit* nesta *branch*, e esse *commit* é responsável por fixar o número de versão da*release* e que ficará descrit no ficheiro CHANGELOG.md. O ficheiro CHANGELOG (escrito em linguagem Markdown) está presente na raiz de todos os projetos e permite fazer um registo de todas as alterações notáveis desenvolvidas em cada versão da aplicação. Na Figura 4, é apresentado um exemplo de uma parte do ficheiro CHANGELOG da aplicação.

```
## [1.2.0] 2021-05-20
### Added
- Scroll to the list of companies in the notifications' company selector
- Verification if the premium is expired
- Support Pin (if the Direct Support addon is active)
- Push Notifications (Firebase Cloud Messaging)

### Fixed
- When the application was still running when the phone was locked
- Bank accounts movements list avatars

### Changed
- All urls to variables in config file

## [1.1.0] 2021-02-10
### Added
- Auto resizing to the font size of all widgets
- App Notifications
```

Figura 4: Exemplo do ficheiro CHANGELOG.md

As regras que a *branch release* devem respeitar são as seguintes:

- Apenas deve ser criada a partir da *branch develop*;
- Apenas deve dar *merge* para *branch develop* e *master*;
- A convenção do nome da *branch* deve respeitar a seguinte regra: *release*/número da versão.

As *branches hotfixes* são bastante semelhantes às *branches releases* pois a sua função é também preparar uma *release*, mas embora uma *branch release* seja planeada, a criação de uma *branch hotfix* surge da necessidade de resolver de imediato um bug ou um erro que ocorreu em produção. O processo consiste em criar uma *branch hotfix* a partir da *branch master* (código fonte que está em produção), fazer os *commits* com a resolução do bug, atualizar o ficheiro CHANGELOG e efetuar o *merge* para *branch master*, e assim que possível deve ser também feito o *merge* para *branch develop* para a resolução deste bug ser incluída na próxima *release*. Este tipo de *branch* é possível verificar pela análise da Figura 5 onde estão representadas as *branches develop e master* com a utilização da *branch hotifix*.

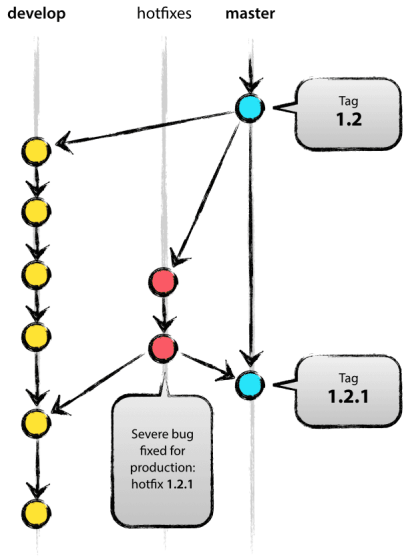


Figura 5: Workflow Git das *branches hotfixes*, *develop* e *master*

2.3 METODOLOGIAS DE DESENVOLVIMENTO DE SOFTWARE

Antes de dar início ao desenvolvimento de um projeto é extremamente importante escolher uma metodologia de desenvolvimento de software para organizar e definir uma filosofia de trabalho. As metodologias de desenvolvimento de software

diferenciam-se em duas grandes classes: Metodologias ágeis (metodologia *Agile*) e metodologias tradicionais. A grande diferença entre estas duas classes é que a tradicional é constituída por etapas rígidas e controladas. Não se apropriam ao tipo de estratégia adotado no projeto aqui apresentado, enquanto uma metodologia de categoria ágil prevê e permite uma flexibilização de etapas, o que é útil à estratégia do projeto que se prevê que venha a necessitar de adaptações.

A metodologia tradicional foi o primeiro tipo de categoria a aparecer, e surgiu da necessidade de as empresas possuírem um tipo de estratégia para se organizarem quando os grandes projetos começaram a ser desenvolvidos. Como foi dito anteriormente, este tipo de estratégias caracteriza-se pela definição de etapas rígidas, muito bem definidas e sequenciais (uma etapa a seguir à outra, sem mudanças inesperadas) e com isto era possível manter qualidade e evitar imprevistos na entrega do projeto. As metodologias tradicionais podem ser chamadas também de *watterfall* (cascata), porque uma etapa só começa quando a etapa anterior é totalmente terminada, e assim sucessivamente sem ter muita atenção às possíveis alterações que possam surgir ao longo do projeto nem podendo voltar atrás, repetindo ou aperfeiçoando uma mesma tarefa.

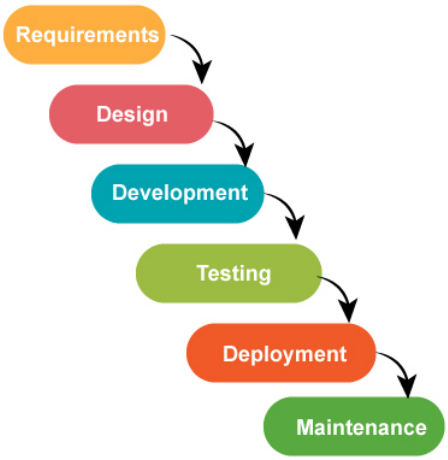


Figura 6: Representação da metodologia em cascata

Como é possível verificar pela Figura 6, esta metodologia é constituída pelas seguintes etapas: análise de requisitos, design, implementação, testes, *deploy* (instalação) e manutenção. Com tudo isto, é possível concluir que as metodologias tradicionais têm como principais vantagens:

- Perceção e previsão por parte do cliente referente aos custos totais que deve investir;

- Boa previsão da data de entrega (pois os prazos são fixos, raramente são mutáveis);
- Garantia da entrega do produto na sua totalidade;
- Maior foco no planeamento do projeto, diminuindo assim os riscos de falhar a entrega do produto.

Por outro lado, estas metodologias apresentam as seguintes desvantagens:

- Falta de adaptabilidade a novas mudanças (não é possível o cliente adicionar novas funcionalidades (*features*) ao longo da implementação do projeto);
- Prazos de entrega geralmente longos;
- Garantia da entrega do produto na sua totalidade;
- Acumulação de erros pois não é possível voltar a etapas anteriores.

Ágil é o nome dado a metodologias de desenvolvimento de software com foco no próprio projeto/produto, metodologias essas pensadas para desenvolver projetos rapidamente e de forma flexível. Surgiu de necessidades apresentadas pelas empresas pois apresentavam dificuldades em cumprir prazos, na integração entre o cliente e a equipa de desenvolvimento ou nos requisitos do produto serem diferentes do pedido feito originalmente. É muito comum uma empresa enfrentar este tipo de desafios na gestão de um projeto.

Esta metodologia surgiu em 2001 desenvolvida por 17 programadores e com isso surgiu o *Agile Manifesto* que é uma declaração de 4 valores e 12 princípios que fundamentam o desenvolvimento ágil de software. Os 4 valores do manifesto dizem que deve-se valorizar mais:

- **Os indivíduos e interações** do que os processos e ferramentas: As ferramentas utilizadas no projeto são obviamente importantes para o seu desenvolvimento, mas é mais importante ter pessoas capazes de trabalharem todas juntas de forma eficiente;
- **O funcionamento do software** do que uma documentação compreensiva: Uma documentação torna-se muito útil para ajudar as pessoas a entender como o software foi criado e como deve ser usado, mas o parte mais importante de um projeto é desenvolver um software eficiente e não uma documentação, pois sem software também não poderia existir uma documentação;
- **A colaboração com o cliente** do que a negociação e o contrato: O contrato é uma parte fulcral para a realização da integração entre o cliente e a empresa,

mas é mais importante estar próximo do cliente para perceber o que ele precisa;

- **A capacidade de resposta a mudanças** mais do que um plano pré-estabelecido: Um plano pré-estabelecido é importante pois pode ajudar se existir um problema durante o desenvolvimento do software, mas não deve ser muito rigoroso para estar disposto a mudanças nas tecnologias utilizadas (pois é um mundo que muda regularmente e às vezes radicalmente).

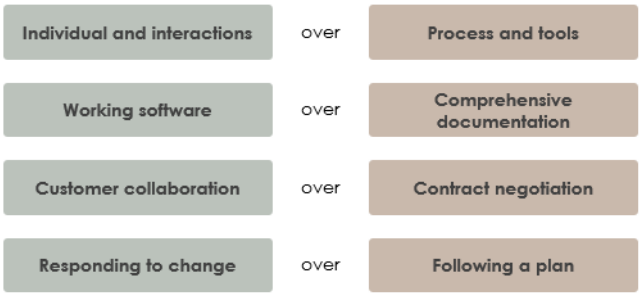


Figura 7: Representação dos valores do *Agile Manifesto*

Resumidamente, segundo a metodologia e a análise da Figura 7, deve-se dar mais prioridade aos aspetos da esquerda (que estão, anteriormente, representados a negrito anteriormente), mas sem retirar todo o valor dos aspetos da direita.

Os 12 princípios do desenvolvimento ágil de software são os seguintes:

1. A maior prioridade é agradar o cliente pelo meio da entrega antecipada e contínua de um software valioso;
2. Aceitar mudanças de requisitos, mesmo no final do desenvolvimento. Os processos ágeis aproveitam a mudança como vantagem competitiva do cliente;
3. Entrega de software funcional com frequência, de algumas semanas a alguns meses, com preferência para a escala de tempo em semanas;
4. As pessoas que entendem do modelo de negócio e os programadores devem trabalhar juntos diariamente ao longo do projeto;
5. Construir projetos em torno de pessoas motivadas. Dar a eles o ambiente e o suporte de que precisam e confiar neles para fazer o trabalho;
6. O método mais eficiente e eficaz de transmitir informações para uma equipa de desenvolvimento é a conversa "cara a cara";
7. A funcionalidade do software é a principal medida de progresso;

8. Os processos ágeis promovem o desenvolvimento sustentável. Os patrocinadores, os programadores e os utilizadores devem ser capazes de manter um ritmo constante indefinidamente;
9. A atenção contínua à excelência técnica e ao bom design aumenta a agilidade;
10. Simplicidade - a arte de maximizar a quantidade de trabalho não realizado - é essencial;
11. As melhores arquiteturas, requisitos e designs surgem de equipas auto-organizadas;
12. Em intervalos regulares, a equipa deve refletir sobre como se tornar mais eficaz e, em seguida, sintonizar e ajustar o seu comportamento.

Muitas das desvantagens das metodologias tradicionais acabam por ser vantagens para uma metodologia ágil, e o contrário também se aplica. Assim, as principais vantagens de uma metodologia Ágil são as seguintes:

- Flexibilidade e adaptabilidade a novas mudanças ao longo do projeto;
- Maior satisfação do cliente, pois ele participa em reuniões e em várias fases do projeto dando um feedback constante;
- Mais colaboração de todos os elementos envolvidos no projeto (cliente, gestor de projeto, equipa de desenvolvimento, etc.) através de reuniões desde o planeamento e desenvolvimento do projeto até à sua implantação no mercado.

Em contrapartida, as desvantagens destas metodologias são as seguintes:

- O projeto tem um custo variável e imprevisível porque os requisitos são definidos ao longo do projeto. Posto isto o cliente não consegue ter uma ideia do valor que irá ter que investir, e desta forma, torna-se importante ele estar a acompanhar a evolução do projeto;
- A entrega do produto é feita por partes o que pode se tornar desvantajoso para o cliente que precisa de um projeto 100% pronto;
- O planeamento do projeto é extenso porque exige várias análises em cada etapa do projeto.

Existem vários tipos de metodologias ágeis que podem ser aplicadas, em que se destacam por exemplo:

- **Scrum**: que tem como caraterística a divisão do projeto em várias fases, chamados *sprints*, que são curtos e rápidos e no seu término é feito sempre uma avaliação;

- **Kanban**: *framework* que diz que se deve montar um quadro com as tarefas que devem ser desenvolvidas, as tarefas que já estão a ser desenvolvidas no momento e as tarefas que já estão finalizadas;
- **Lean**: em que o objetivo é otimizar os processos e diminuir processos que não agregam valor ao projeto, melhorando assim a eficácia e eliminado o desperdício.

A metodologia utilizada para o desenvolvimento deste projeto foi bastante facilitada, pois os restantes projetos da empresa estavam a ser desenvolvidos em cima de uma metodologia ágil, por isso foi este o método de trabalho escolhido para desenvolver a aplicação. A razão da empresa ter escolhido este tipo de metodologia deve-se ao facto da sua flexibilidade e adaptabilidade a novas mudanças ao longo do projeto, que foi um ponto fulcral para a decisão. A forma como foi utilizada este tipo de metodologia foi essencialmente através de duas *frameworks*, Scrum e Kanban. A junção destas duas *frameworks* no processo de trabalho dá se o nome de Scrumban que junta os princípios do Scum e do Kanban numa única metodologia e foi projetada originalmente com o intuito de transitar de Scrum para Kanban. De seguida irá ser descrito de forma mais concreta em que consiste o Scrum e o Kanban e de que forma foi utilizado da implementação deste projeto.

2.4 SCRUM

Scrum é uma *framework* de gestão de projetos que, como foi dito antiormente, faz parte das metodologias ágeis cujo principal objetivo é a resolução de problemas complexos e variados enquanto se produz produtos com a maior qualidade e valor possível no prazo definido. Scrum foi desenvolvido na década de 1990 por Jeff Sutherland e Ken Schwaber e tem sido aprimorado continuamente desde então. O termo Scrum foi inspirado com a formação de início de uma jogada em rugby onde os jogadores estão todos juntos para dar início à jogada. Esta *framework* foi inicialmente planeada para ser usada no desenvolvimento de software, mas com a sua rápida evolução, hoje em dia é utilizado em todos os tipos de áreas. Scrum é fundamentado na teoria que o conhecimento vem da experiência e das tomas de decisão baseadas no que é conhecido, e a implementação desta *framework* no método de trabalho de uma equipa assenta em três pilares importantes:

- **Transparência:** o trabalho deve estar claro e definido para todas as partes envolventes do projeto, e esta transparência dá-se através da comunicação entre todas as partes principalmente durante as reuniões;

- **Inspeção:** o trabalho deve ser inspecionado com frequência para certificar a máxima qualidade na primeira na primeira entrega;
- **Adaptação:** o trabalho deve estar pronto à adaptação da necessidade do negócio. Como foi falado anteriormente, esta capacidade de mudança ao longo do projeto é um ponto bastante forte do Scrum.

A equipa utilizada numa framework Scrum, dada pelo nome Scrum Team, é constituída por três papeis distintos e importantes:

- ***Product Owner***: representa cliente, geralmente representado por uma pessoa, por exemplo de uma empresa que requisitou o produto. É o único responsável por transmitir quais são as funcionalidades, com um grau de prioridade, que devem constar no produto, ou seja, é ele que desenvolve o *Product Backlog* (lista de funcionalidades do produto organizadas com grau de prioridade);
- ***Development Team:*** é a equipa responsável por desenvolver e entregar todas as funcionalidades do projeto. A equipa tem que ser capaz de realizar todas as fases do processo de desenvolvimento, desde o design e a implementação aos testes, documentação e *deploy*.
- ***Scrum Master* (responsável da equipa):** é o responsável por garantir que a equipa de desenvolvimento esteja a trabalhar com base nos princípios e práticas do Scrum. Ele tem também o dever de ajudar a equipa a resolver possíveis problemas que possam prejudicar o desenvolvimento das funcionalidades planeadas.

A integração de todos os papéis deste processo caracteriza-se por uma série de eventos que serão descritos de seguida e desenvolve-se da seguinte forma: o *Product Owner* desenvolve então o *Product Backlog* com o auxílio do *Scum Master* para definir o grau de exigência de cada funcionalidade. Em seguida, o projeto é dividido em *Sprints* que consistem em períodos de tempo (geralmente semanais, 2 a 4 semanas) onde a equipa de desenvolvimento irá desenvolver uma ou mais funcionalidades.

Antes de cada *Sprint*, é feita uma reunião para planear o que será desenvolvido durante a próxima *Sprint*, chamada de *Sprint Planning*, onde é feito também o *Sprint Backlog* que representa a lista de tarefas que são comprometidas a fazer durante o próximo Sprint. A lista do *Sprint Backlog* representa uma extração, feita pela equipa de desenvolvimento, do *Product Backlog* dependendo do grau de prioridade das funcionalidades definidas pelo *Product Owner*. Na conclusão de cada *Sprint*, acontece a *Sprint Review* que consiste numa reunião onde participam os

três papéis presentes no projeto onde será mostrado e avaliado todo o trabalho feito durante a *Sprint* atual. Depois de cada *Sprint*, dá-se também o que é chamado de *Sprint Retrospective* que serve para identificar o que funcionou bem, o que pode ser melhorado e que ações serão tomadas para melhorar.

A cada novo dia de trabalho, o *Scrum Master* e a equipa de desenvolvimento reúnem-se para uma reunião diária, chamada de *Daily Scrum*. Esta reunião tem como principais objetivos: relatar o que foi feito durante o dia anterior, identificar possíveis problemas e o mais importante, o que será feito durante o dia atual. Dado isto, as reuniões devem ser, idealmente, realizadas na parte da manhã para pensar sobre o trabalho a realizar durante o dia. Os problemas identificados na *Daily Scrum* não devem ser resolvidos durante esta, para não perder tempo, preferencialmente devem ser resolvidos pelo *Scrum Master* posteriormente.

2.5 KANBAN

Kanban é uma *framework* que tem como principal preocupação o fluxo e a monitorização das tarefas através da gestão visual de um quadro, como é possível analisar pela Figura 8 que é dividido em três cartões:

- tarefas que estão por desenvolver (*To Do)*;
- tarefas que estão a ser desenvolvidas no momento (*On Going*);
- tarefas que já estão concluídas (*Done*).

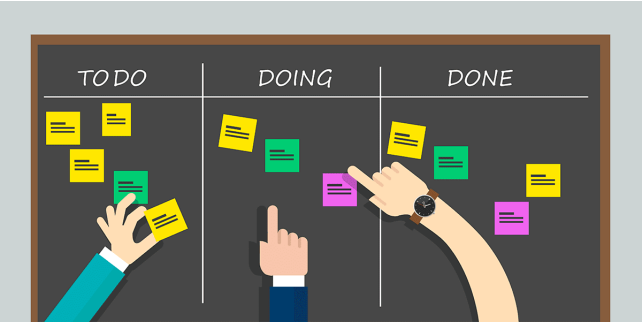


Figura 8: Representação de um quadro Kaban genérico

Para cada tarefa, devem ser definidas as pessoas responsáveis pela implementação, o seu grau de exigência, o prazo para a conclusão, entre outros aspetos. Assim, é possível ter uma melhor visibilidade e organização das funções a serem cumpridas. O Kanban pode ser utilizado de várias formas e como foi referido anteriormente em quadros, pode ser literalmente um quadro de cortiça, por exemplo, dividido em três

partes onde cada tarefa é representada por um *post-it*. Ou pode ser representado também de forma digital, e cada vez há mais plataformas para isso como o Trello, Clubhouse, Jira, Gitlab Issues, GitHub Issues, entre muitos outros, o que torna mais fácil a visualização. A Figura 9 mostra como é representado um quadro Kanban na plataforma GitLab que foi a plataforma utilizada durante o desenvolvimento do projeto.

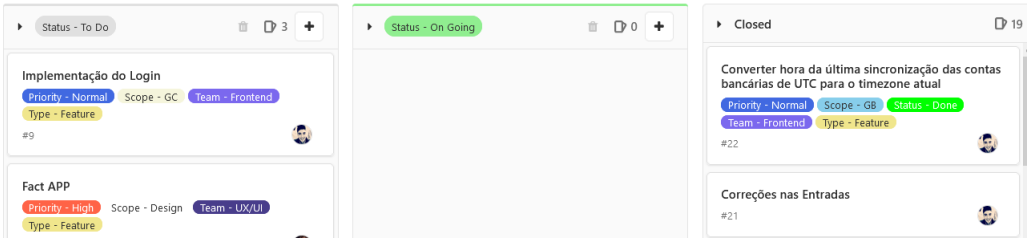


Figura 9: Representação de um quadro Kaban na plataforma GitLab

2.6 COMO FORAM APLICADAS AS METODOLOGIAS

Depois da definição das metodologias, é importante esclarecer de que forma estas foram utilizadas ao longo do desenvolvimento da aplicação. Maioritariamente a metodologia que foi utilizada durante o projeto foi o Scrumban, que junta os conceitos do Scrum e do Kanban.

Relativamente à metodologia Scrum, esta não foi seguida rigorosamente como foi definida anteriormente, pois há certos eventos que existem na metodologia e não foram executados durante o desenvolvimento. Em relação aos papeis do Scrum, pelo facto da aplicação ser um produto próprio (e não um produto pedido por um cliente externo), neste caso o *Product Owner* e o *Scrum Master* são representados pela mesma pessoa, enquanto a equipa de desenvolvimento é composta por apenas um elemento que é o autor deste relatório.

Antes de dar início ao desenvolvimento da aplicação, foi definido o *Product Backlog* com todas as funcionalidades, anteriormente discutidas pelos responsáveis, que iriam estar presentes na aplicação. Com a lista de todas as funcionalidades que serão defenidas no subcapítulo 3.3.1, foi dado início ao desenvolvimento da aplicação. Uma diferença face à metodologia Scrum original, é que invés de o projeto ser dividido em *Sprints* (onde cada *sprint* pode ser constituída por várias funcionalidades), o projeto foi dividido em funcionalidades (*Issues*). Ou seja, era realizada uma reunião, do género de uma *Sprint Planning*, onde era definida qual funcionalidade deveria ser desenvolvida em seguida.

A forma como o Kanban é utilizado no desenvolvimento do projeto, é através da definição do Product Backlog, e também se for necessário acrescentar mais funcionalidades à aplicação. As funcionalidades são definidas no GitLab através das GitLab Issues. As Gitlab Issues representam uma lista de funcionalidades (Issues), em que cada issue pode ter uma descrição associada, a pessoa responsável pelo seu desenvolvimento, uma data para terminar ou uma lista de *tags* que podem ser altamente customizadas para representar, por exemplo, a que tipo de projeto pertence (*frontend*, *backend*, entre outros) ou em que parte do fluxo Kanban a funcionalidade está localizada (To-do, Doing ou Done).

As GitLab Issues têm duas formas de representação gráfica: através de uma tabela em que cada linha pertence a uma funcionalidade ou através de um quadro que contém os três cartões do Kanban.

2.7 DEVOPS E CI/CD

Antes de enunciar o conceito CI/CD (Continuous Integration/Continuous Delivery), é importante esclarecer do que se trata o DevOps. DevOps resulta da junção das palavras Dev (*Development* – desenvolvimento) e Ops (*Operations* - operações), e este trata-se de uma filosofia no mundo de TI que incentiva a integração, colaboração e transparência contínua entre as equipas de desenvolvimentos de software e as equipas de administração de software com o principal objetivo de quebrar as barreiras e melhorar a comunicação entre ambas as funções. O DevOps permite também melhorar e acelerar o processo do desenvolvimento de aplicações, lançamento do software e de novos recursos para o cliente.

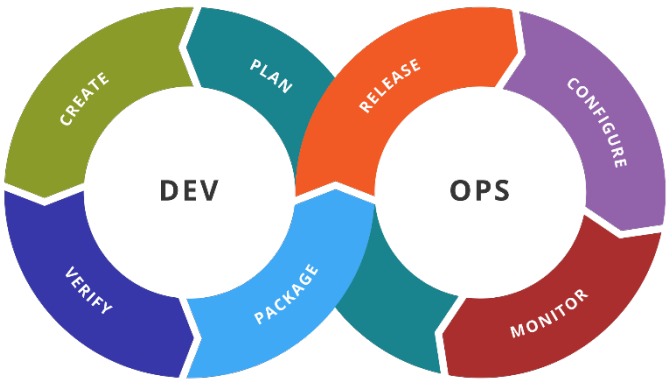


Figura 10: Ilustração das etapas nas ferramentas DevOps

Para ajudar a seguir esta filosofia, os primeiros seguidores de DevOps desenvolveram um conjunto de 7 ferramentas que fazem parte do ciclo de vida de DevOps como é possível ver na Figura 10:

- **Planear:** Esta fase define o modelo e requisitos do negócio;
- **Criação/Código:** Esta fase envolve todo o processo do design e criação do código do software;
- **Verificação/Testes:** Esta fase envolve os testes contínuos (manuais ou automatizados) para garantir a qualidade do código;
- **Verificação/Testes:** Esta fase envolve os testes contínuos (manuais ou automatizados) para garantir a qualidade do código;
- ***Package/Build*:** Esta fase permite compilar todo o código para estar pronto para ser lançado;
- ***Release/Deploy*:** Esta fase inclui todo o processo de gestão, coordenação, agendamento e automatização do lançamento do software para produção;
- **Configuração:** Esta fase faz a gestão do software durante a produção;
- **Monitorização:** Esta fase envolve a identificação e o recolhimento de possíveis problemas de uma versão de software específica em produção.

De seguida, irão ser enunciadas algumas das vantagens de utilizar o DevOps dentro de uma equipa de desenvolvimento:

- Entrega mais rápida e melhor do produto final;
- Resolução mais rápida dos problemas e complexidade reduzida;
- Maior escalabilidade e disponibilidade;
- Maior automatização.

Para aplicar este tipo de filosofia, existem várias práticas que refletem a melhoria e a automatização dos processos. Muitas destas práticas focam numa ou mais fases do ciclo de vida do DevOps, enunciadas acima.

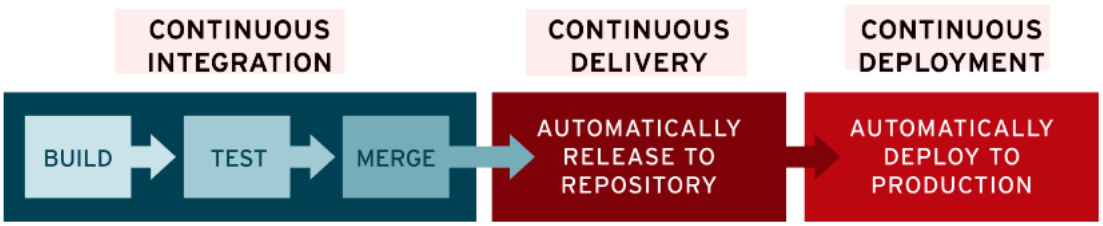


Figura 11: Ilustração das etapas do CI/CD

Durante o desenvolvimento do projeto, foi utilizado uma *stack* de três práticas que tem como nome CI/CD e refere-se às seguintes práticas:

- ***Continuous Integration* (Integração Contínua):** Esta prática é utilizada quando os programadores fundem as mudanças que estavam a trabalhar na *branch* da funcionalidade com a *branch develop*. Sempre que é terminado uma funcionalidade, antes de se fundir com a *branch* principal há um processo de validação das alterações do desenvolvedor que consiste na criação de uma *build* e na execução de testes automatizados. É muito importante realizar esta prática, pois vai evitar possíveis problemas de integração quando é criada uma release com várias funcionalidades feitas por diferentes programadores. Esta prática reúne um conjunto de ferramentas (tais como testes automáticos para validar a qualidade do código, ferramentas para revisão da sintaxe do código, entre outras) para verificar que o novo código desenvolvido está pronto para ir para produção, colocando bastante importância na automatização dos testes sempre que há uma integração entre as branches;
- ***Continuous Delivery* (Entrega Contínua):** A Entrega Contínua segue-se à Integração Contínua. Esta prática é responsável por implementar de forma automática todas as alterações de código num ambiente de testes e/ou produção, após a fase de *build* pertencente à prática Integração Contínua. Nesta prática antes de lançar para um ambiente de testes, são feitos Testes de aceitação, que ao contrário dos testes automatizados da prática anterior que têm como foco o código, estes testes têm como principal objetivo analisar o comportamento da aplicação e avaliar a conformidade com requisitos inicialmente pedidos pelo cliente, avaliando assim se é então aceitável para a entrega. Esta prática retira a complexidade de fazer uma *release* de forma manual aos programadores, porque faz todo este processo de forma automático ficando apenas o processo de colocar o produto em produção como uma ação manual;

- ***Continuous Deployment* (Implantação Contínua):** A prática da Implantação Contínua é bastante semelhante à prática anterior (Entrega Contínua) e tende muitas vezes a confundir pelo simples facto de ambas terem a sigla "CD". A principal diferença para a prática anterior é o facto de esta ser totalmente automática sem qualquer intervenção humana. Esta prática é excelente para que a equipa de programadores apenas tenha o foco na construção do produto/software e não se preocupe com a *release*. Tradicionalmente, as equipas escolhem um dia para lançar uma nova versão para produção onde junta várias implementações de vários programadores. Esta prática tem a particularidade de não ter um dia para o lançamento da aplicação, porque cada implementação de cada programador quando é finalizada é colocada imediatamente em produção, para assim haver um rápido feedback para o cliente.

Para concluir é importante realçar que DevOps foca apenas no processo e na organização, enquanto as práticas CI/CD focam nas ferramentas.

3

DESENVOLVIMENTO

3.1 RECOLHA DAS TECNOLOGIAS UTILIZADAS

O principal objetivo desta fase é a recolha da tecnologia a utilizar, mais propriamente a escolha de uma *framework* que ajude no processo de desenvolvimento da aplicação. Atualmente existem três principais "caminhos" que podemos seguir para desenvolver uma aplicação móvel: aplicações nativas, aplicações cross-platform nativas e aplicações híbridas.

Uma aplicação nativa é desenvolvida especificamente para uma plataforma com a linguagem de programação nativa do próprio sistema. Na atualidade, as duas plataformas que dominam o mercado são o Android pertencente à Google (que usa as linguagens Kotlin ou Java) e iOS pertencente à Apple (que usa as linguagens Swift ou Objetive-C). Tem bastantes vantagens como o facto de ter um melhor desempenho, precisar de menos dependências, ter um acesso direto a todos os recursos do telemóvel (câmara, GPS), estar sempre atualizada em relação a novos recursos e versões e ter um IDE específico de cada plataforma como o Android Studio para Android e o Xcode para iOS, o que facilita bastante o desenvolvimento. É verdade que o desenvolvimento nativo tem imensas vantagens e sem dúvidas que, ainda hoje, é dos tipos de aplicações mais utilizadas para desenvolver uma aplicação móvel. O único problema que este tipo de aplicação apresenta é se uma empresa decidir uma aplicação multiplataforma que consiga ser instalada em vários tipos de sistemas operativos, terá de haver o esforço de se constituir várias equipas (uma para cada tipo de plataforma) o que provoca um tempo de desenvolvimento maior, ou se as equipas estiverem a trabalhar ao mesmo tempo, carece de um investimento maior por parte da empresa. Por último, torna-se mais difícil haver uma padronização da UI da aplicação por serem sistemas diferentes.

As aplicações *cross-platforms* talvez sejam a abordagem mais utilizada para desenvolver uma aplicação atualmente. O desenvolvimento *cross-platform*, como o próprio nome indica, consiste em desenvolver apenas uma vez para várias plataformas, é programado num único código fonte da própria *framework* que por fim é compilado para código nativo da plataforma. Com isto, as desvantagens apresentadas

anteriormente do desenvolvimento nativo acabam por ser vantagens das aplicações *cross-platform*, pois reduz muito o investimento por parte da empresa porque só necessita de apenas uma equipa de desenvolvimento, o que irá acelerar imenso o processo de construção da aplicação para várias plataformas. Não vai haver dificuldade na padronização da UI (*User Interface*) da aplicação pelos vários sistemas porque, como apenas fazemos um *layout*, esse mesmo *layout* irá manter-se pela aplicação nas várias plataformas. Como este tipo de abordagem acaba por ser no fim uma aplicação nativa. O desempenho relativamente às aplicações nativas não irá ser muito discrepante o que torna este tipo de *framework* cada vez mais uma escolha indiscutível no mercado do desenvolvimento móvel. Algumas das *frameworks* para desenvolver uma aplicação multiplataforma são Flutter, React Native, Xamarin, Native Script, etc.

As aplicações híbridas são desenvolvidas utilizando tecnologias para a Web, HTML, CSS (*Cascading Style Sheets*) e o JavaScript. Na sua essência, este tipo de abordagem usa um componente chamado *WebView* (que está presente em todos os sistemas) dos sistemas nativos que funciona como um *browser* integrado dentro da própria aplicação nativa. Se, por exemplo, uma empresa já possuir uma equipa de desenvolvimento Web e não quer investir tempo nem dinheiro em contratar uma equipa especializada no desenvolvimento de aplicações móveis, isto pode se tornar vantajoso na medida em que é possível construir uma aplicação móvel com tecnologias Web. Como as aplicações *cross-platform*, estas também permitem ter apenas uma fonte de código utilizada para várias plataformas. Por outro lado, as aplicações híbridas possuem um desempenho inferior às aplicações nativas, e se for uma aplicação bastante complexa a perda de desempenho na utilização da aplicação pode ser bastante relevante. As opções mais utilizadas para o desenvolvimento de aplicações híbridas são Ionic, Apache Cordova e Capacitor.

A tabela 1 apresenta algumas diferenças entre os três tipos de aplicações relatadas em cima.

Por fim, a decisão do tipo de *framework* a utilizar recaiu sobre utilizar as aplicações *cross-platform*. Como o objetivo era ter uma aplicação para, pelo menos, Android e iOS a decisão foi relativamente fácil por todas as vantagens apresentadas acima. Posteriormente à decisão do tipo de *framework* a utilizar, era necessário perceber as ferramentas mais relevantes que existiam no mercado para o desenvolvimento de aplicações multiplataforma. Depois de algum tempo de pesquisa, foram destacadas 4 *frameworks* principais: Flutter, React Native, Native Script e Xamarin. A tabela 2 mostra, resumidamente, algumas das diferenças entre estas mesmas *frameworks*.

1 Github Stars à data de 4 de março de 2021

Tipo de Aplicação	**Nativo**	**Híbrido**	***Cross-platform***
"Debugging"	Usa ferramentas de "*debugging*" nativas	Usa ferramentas de "*debugging*" nativas e de Web	É dependente da *framework*
Código fonte	Necessita de código individual separado por cada plataforma	Existe um único código fonte com capacidades que são específicas de cada plataforma	Existe um único código fonte com capacidades que são específicas de cada plataforma
Prós	Possui alta performance; Acesso completo aos recursos dos dispositivos e do sistema operativo; UI nativa atualizada junto com o sistema operativo	Suporta vários sistemas operativos; Possui fortes capacidades de personalização; Contém código reutilizável	Elementos da UI do utilizador praticamente iguais aos do nativo; Perto da aparência do Nativo; Código reutilizável
Contras	Suporta uma única plataforma; Não permite reutilização de código	Tem problemas de desempenho; Existe restrição de acesso a recursos do sistema operativo	Tem acesso limitado a recursos
Dependência	É menos dependente de código aberto, bibliotecas e plataformas	É bastante dependente de várias bibliotecas e ferramentas	É bastante dependente de várias bibliotecas e ferramentas
Motor de renderização	Nativo	Browser	Nativo
Ferramentas	XCode, Android Studio	Iconic, Apache Cordova	React Native, Flutter, Xamarin
Custo	Alto	Custo-benefício	Custo-benefício
Tempo de Mercado	Consome tanto tempo como desenvolver uma aplicação de cada vez	Irá poupar muito tempo porque o código é reutilizável	Irá poupar muito tempo porque o código é reutilizável

Tabela 1: Diferenças entre os tipos de aplicações

Framework	**Fluttter**	**React Native**	**Native Script**	**Xamarin**
Linguagem	Dart	JavaScript	JavaScript	C#
Developers	Google	Facebook	Progress Software	Microsoft
Plataformas	iOS, Android, Web, Desktop	iOS, Android, UWP	iOS, Android	iOS, Android, UWP, WPF, macOS, Wasm
GitHub *Stars* [1]	114k	93.7k	19.8k	5.2k
Desempenho	Praticamente igual ao nativo	Próximo do nativo	Moderado	Moderado
"Hot reload"	Sim	Sim	Sim	Não
GUI	Usa Widgets próprios	Usa *controllers* da UI nativa	Usa *controllers* da UI nativa	Usa *controllers* da UI nativa
Reutilização de código	50%-90% do código	90% do código	Mais de 90% do código	96% do código
Preço	*Open Source*	*Open Source*	*Open Source* / Pago também se utilizar os *Sidekick Cloud Services*	*Open Source* / Pago também por causa do IDE Visual Studio
Apps populares	Google Pay, Alibaba, eBay, Stadia	Facebook, Instagram, Airbnb, UberEats, Discord	Dwitch, Regelneef	Olo, Storyo

Tabela 2: Diferenças entre as *frameworks* de aplicações móveis

Depois de uma análise rápida às quatro *frameworks cross-platform*, foram escolhidos o Flutter e o React Native para uma análise mais profunda. Estas duas escolhas deveram-se bastante à popularidade que ambas apresentam e porque as restantes não apresentavam caraterísticas muito diferentes destas duas.

O Flutter é excelente para o desenvolvimento de uma aplicação MVP (Minimum Viable Product) o que era os dos requisitos principais para o desenvolvimento da aplicação em questão. Oferece uma ótima solução para o acesso a recursos nativos como localização, câmara, entre outros. É possível no mesmo projeto Flutter, desenvolver código nativo da plataforma em questão e através de um sistema de comunicação de canais é permitido invocar esse mesmo código nativo em linguagem Dart, o que facilita em muito a utilização de recursos nativos na aplicação. Outro ponto bastante importante é que o desempenho de uma aplicação escrita em Flutter é praticamente idêntico ao desempenho de uma aplicação nativa pois este não necessita de código intermediário para interpretação; ele é construído diretamente no código nativo reduzindo assim bugs de desempenho. Em relação às linguagens de programação, o Dart do Flutter beneficia da compilação *ahead of time* (AOT), onde o código é compilado em tempo de desenvolvimento antes mesmo do tempo de execução, enquanto o JavaScript do React Native utiliza a compilação *just in time* (JIT), onde o código é compilado durante o tempo de execução. O JIT pode gerar problemas de performance em aplicação maiores e produzir *crashes* durante a sua utilização. Isto não acontece em compilações AOT pois o código foi todo compilado antes de executar a aplicação permitindo assim uma maior fluidez, o que é uma vantagem bastante forte do uso do Dart/Flutter.

Em relação às desvantagens mais evidentes, são o facto que, tanto a linguagem de programação Dart como o *framework* Flutter serem recentes o que pode sugerir que não haja uma comunidade tão evoluída como a de *frameworks* mais antigos que pode se tornar evidente na resolução de um erro mais complexo durante o tempo de desenvolvimento da aplicação.

Em relação ao React Native, uma grande vantagem é o facto de apresentar uma das linguagens de programação mais utilizadas do mercado que é o caso do JavaScript, isto que irá diminuir bastante o tempo de aprendizagem da *framework* para quem já possui conhecimentos de JavaScript, mesmo que nunca tenha trabalhado em React. Esta vantagem também se aplica à *framework* destinada ao desenvolvimento web, React onde também é bastante utilizada no mercado. Por conseguinte talvez seja a mais utilizada no desenvolvimento Web, e como apresentam sintaxes e conceitos bastante semelhantes, quem domina React não irá ter muitas dificuldades em se adaptar ao React Native.

Com todas os pontos apresentados, a decisão final seria escolher qual *framework* utilizar durante o desenvolvimento da aplicação. Foi decidido então a utilização do Flutter. A razão pela qual foi escolhido o Flutter, deve-se bastante ao facto do desempenho da aplicação (porque teoricamente seria melhor que React Native pelas várias razões apresentadas em cima), apesar de o React Native apresentar uma comunidade mais forte que Flutter acreditamos que o Flutter mais tempo menos tempo iria chegar aos pontos do seu oponente nesse aspeto, foi está a crescer exponencialmente nos últimos anos apresentado até mais utilizações do que o React Native (como foi possível ver nas Github Stars apresentadas na tabela 2). Como a aplicação também iria necessitar de alguns recursos nativos da plataforma móvel, achámos o Flutter uma melhor alternativa e mais segura do que o React Native.

3.2 FLUTTER

Como foi enunciado no subcapítulo 3.1, a principal ferramenta usada no desenvolvimento da aplicação foi o Flutter. Segundo a documentação oficial, Flutter é um kit de desenvolvimento de UI (*User Interface*) feito pela Google e escrito na linguagem Dart que permite a construção de aplicações compiladas de forma nativa para Android, iOS, Windows, Mac, Linux e Web a partir de um único código fonte.

3.2.1 *"Um pouco de história"*

Voltando um pouco atrás na história, o Flutter surgiu de uma necessidade interna dentro da própria empresa Google que tinha como lema da sua missão "Build a better way to develop for mobile", ou seja, construir uma maneira melhor de desenvolver para aplicações móveis. O principal problema que tinham naquele momento, é que para cada tipo de sistema operativo (Android, iOS, etc.) era necessária uma equipa especializada para desenvolver uma aplicação, o que tornava o desenvolvimento mais lento e mais caro.

O Flutter apareceu então pela primeira vez em 2011 numa cimeira de desenvolvedores sobre a linguagem de programação Dart (esta também lançada pela Google em 2011, a aprofundar mais à frente). Esta primeira versão do Flutter foi conhecida com o nome de código "Sky" e era compilada apenas para o sistema operativo Android, e foi uma surpresa na altura porque foi prometido que uma aplicação construída em Sky era capaz de ser renderizarada a 120 frames por segundo, e foi então concretizada essa promessa. Em 2015 durante a Dart Developer Summit, foi

possível ver código Dart a ser executado num dispositivo Android e foi pela primeira vez apresentado com o nome Flutter. A grande estreia do Flutter deu-se no Google IO de 2017, em que foi feita uma sessão de *"live coding"* onde foi desenvolvida uma aplicação integrado com Firebase e com acesso à câmara.

Em setembro de 2018, a Google anunciou durante a keynote do Google Developer em Xangai, o Flutter Release Preview 2, que iria ser o último grande lançamento antes da primeira versão estável do Flutter 1.0. A grande novidade deste release foi o facto de lançarem o estilo de UI de Widgets com nome Cupertino o que permitiu aos desenvolvedores implementar uma aplicação em Flutter com os temas de todos os componentes (botões, texto, input fields, etc.) do iOS e assim ter a possibilidade de aplicação oferecer o mesmo *"feeling"* de uma aplicação iOS. De relembrar, que o Flutter continha o principal tema UI de componentes da Google, Material. Com isto era possível ter a mesma base de código (com mínimas alterações relacionadas com o tema) e ter uma aplicação Android e iOS com os respetivos temas relacionados com cada um dos sistemas operativos. Naquele momento da release, o Flutter já se encontrava no top 50 da utilização de repositórios públicos do GitHub. A Figura 12 apresenta parte desse top 50 onde se pode ver o Flutter em 45º lugar.

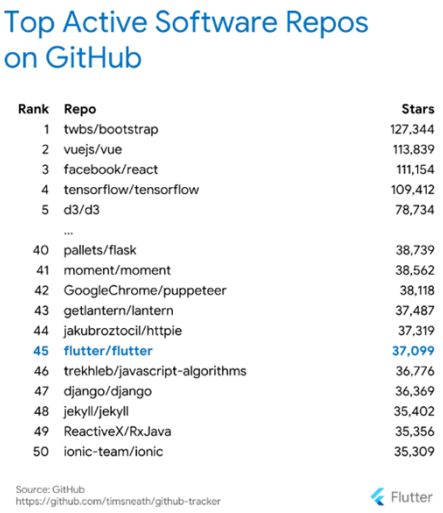


Figura 12: Top 50 dos Repositórios ativos do GitHub em setembro de 2018

Em dezembro de 2018, foi finalmente lançada a primeira versão estável do Flutter. Não trouxe muitas novidades ao nível do desenvolvimento móvel, mas sim, uma grande novidade para o mundo do desenvolvimento Web. Foi anunciado o projeto Hummingbird que mais tarde veio a tornar-se o Flutter para a Web que se tornou estável mais recentemente na versão 2.0 do Flutter. Em março de 2021, foi lançado a segunda versão do Flutter e as principais novidades foram: lançamento da primeira versão estável do Flutter para Web, a adição do *"Sound Null Safety"* à linguagem

Dart que permite que os desenvolvedores evitem ter falhas de nulos, que é um dos erros mais comuns para as aplicações deixarem de responder durante a sua utilização e o lançamento de uma versão beta do Flutter para aplicações Desktop. A versão estável está prevista ser lançada no final do ano de 2021.

3.2.2 *Arquitetura*

Para aprofundar mais o Flutter é importante perceber a sua arquitetura. Esta arquitetura é composta por um sistema de camadas (como é possível verificar pela Figura 13), em que cada camada é composta por várias bibliotecas independentes, onde cada camada depende da sua camada subjacente e nenhuma camada tem acesso privilegiado à camada abaixo. Para o sistema operativo correspondente, as aplicações escritas em Flutter são empacotadas da mesma maneira que uma aplicação desenvolvida nativamente. Existe um *embedder* específico de cada plataforma que fornece o ponto de entrada e coordena com o sistema operativo para ter acesso a serviços como superfícies de renderização, acessibilidade e entrada. Esse embedder é escrito numa linguagem diferente consoante o sistema operativo: neste momento é utilizado Java e C++ para Android, Objective-C / Objective-C ++ para iOS e macOS e C ++ para Windows e Linux.

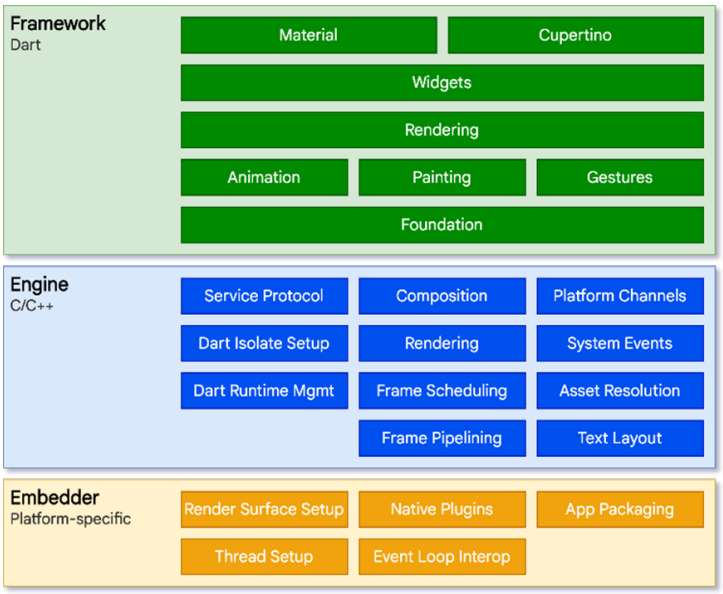


Figura 13: Arquitetura do Flutter

3.2.2.1 *Flutter Engine*

No núcleo do Flutter está o "Flutter Engine" que é escrito maioritariamente em C++. O Engine é um motor gráfico 2D de alto desempenho que gere as dependências de nível inferior como a Dart VM, Skia (biblioteca gráfica de código aberto), o sistema operativo e as APIs específicas de cada plataforma para fornecer uma única API independente da plataforma para a Flutter Framework. O *Engine* é também responsável por renderizar cenas compostas sempre que uma nova *frame* é pintada no ecrã.

3.2.2.2 *Flutter Framework*

A Flutter Framework é uma camada da arquitetura com a qual os desenvolvedores interagem diretamente. É escrita em Dart e é composta pelas seguintes camadas, de baixo para cima, respetivamente:

- Classes básicas necessárias para desenvolver uma aplicação básica em Flutter onde estão incluídas abstrações de serviços para implementar animações, gestos, pintura e blocos de construção (colunas, filas);
- A camada de renderização que fornece uma abstração para lidar com o layout, mais precisamente para construir uma árvore de *widgets* onde podem ser atualizados dinamicamente ao longo da árvore;
- A árvore de *widgets*, onde cada objeto de renderização (*widget*) na camada de renderização corresponde a uma classe na camada de *widgets*. É possível fazer aqui combinações de classes que podem ser reutilizadas em todo o projeto. Esta é a camada onde está presente o modelo de programação;
- As bibliotecas Material (Google) e Cupertino (Apple) que oferecem um conjunto de definições das linguagens de design associadas para implementar uma aplicação com o design apropriado para uma aplicação em Android ou iOS, respetivamente.

A Flutter Framework é relativamente pequena, mas é o necessária para construir uma aplicação bastante robusta em Flutter. Para *features* mais complexas na aplicação é necessário recorrer a *packages* externos construídos pela própria equipa da Google ou pela comunidade para acesso a recursos como a câmara, ou a outros sensores do telemóvel, para consumir uma API ou para fazer animações mais complexas, entre muitos outros *packages* disponíveis no website oficial pub.dev.

3.2.3 *Widgets*

Como é bastante comum dizer no "mundo" do Flutter, "em Flutter tudo é um *widget*", que é o conceito mais importante da *framework*. Um *widget* é um componente da interface do utilizador que afeta e controla a interface da aplicação. Os widgets são construídos com base numa estrutura moderna inspirada pelo React (referida no subcapítulo anterior). No React ou em outras *frameworks* web *frontend*, os *widgets* são semelhantes ao conceito de componente.

Em Flutter, a própria aplicação é um *widget* que contém *widgets* filhos (*subwidgets*). A aplicação é o *widget* de nível superior, e toda a interface de utilizador (UI) da aplicação são *subwidgets*, e cada *subwidgets* pode ter outros *widgets* filhos permitindo assim criar uma UI complexa facilmente. Uma boa comparação para o conceito de *Widget* é se imaginar uma aplicação como uma construção em LEGO, em que toda a construção representa uma aplicação em Flutter e cada peça LEGO representa um *widget*.

Na Figura 14 é possível ver uma aplicação demo fornecida pelo Flutter quando é criado um novo projeto que é constituída por um título na barra da aplicação, uma coluna central com dois textos e um botão flutuante que permite incrementar o número que está presente no segundo texto. Do lado esquerdo da imagem é possível identificar a árvore de *widgets* da aplicação e do lado direito é possível ver a UI da aplicação.

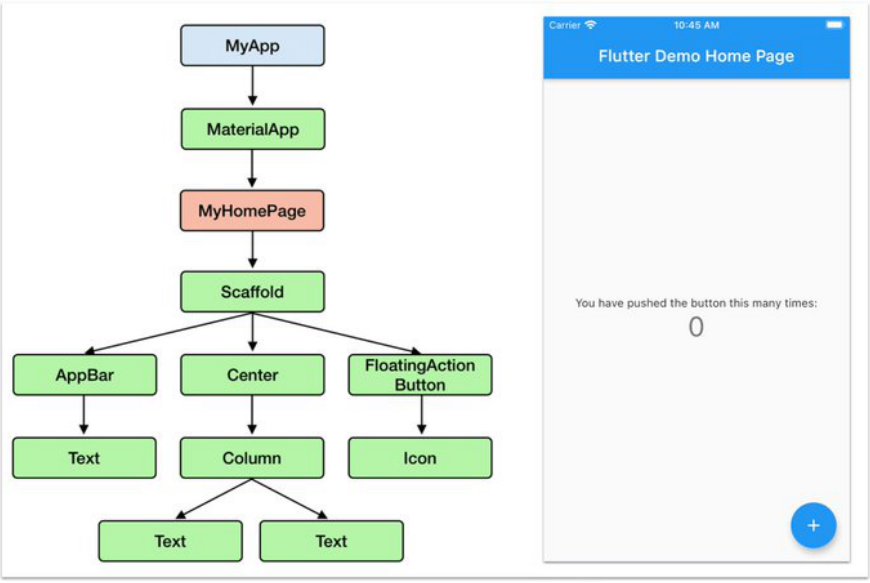


Figura 14: Aplicação *demo* fornecida pelo Flutter

Para cada *Widget*, é possível alterar uma série de propriedades que são responsáveis por personalizar o próprio *widget*. Por exemplo para o *widget* Text acima representado é possível alterar o tipo de letra, o tamanho da fonte, cor, entre muitos outros.

Na Flutter Framework, existem mais de 170 *widgets* diferentes documentados no catálogo online que podem ser inseridos em dois grupos principais:

- Widgets de *layout* que são aqueles que são responsáveis pelo posicionamento dos restantes *widgets* no ecrã (*Column*, *Row*, *Align*, *Stack*, *Padding*, etc.);
- Widgets de UI que são os *widgets* que estão visíveis para o utilizador no ecrã (*Text*, *ElevatedButton*, *Image*, etc.).

3.2.3.1 *Tipos de Widgets*

Existem dois tipos principais de *widgets* no Flutter: StatelessWidget (como o próprio nome indica é um *widget* que não tem estado) e o StatefulWidget (*widget* com estado). Widgets sem estado são *widgets* que não armazenam nenhum estado, ou seja, não armazenam nenhum valor que possa mudar. Para dar um exemplo, o botão flutuante apresentando na Figura 14 é um *widget* sem estado, porque é definido uma vez com o ícone associado e nunca mais muda. Outro exemplo para comprovar este tipo de *widget*, é o primeiro *widget* de Text presente na imagem que contém uma frase que nunca muda de estado. Widgets que contêm estado são *widgets* que contêm valores que podem ser sujeitos a mudança. Isto significa que pode acompanhar as alterações dos valores e atualizar a interface sempre que esse valor é atualizado. O StatefulWidget cria um objeto State que controla as alterações. Quando os valores que estão presentes no objeto State mudam, ele cria um novo *widget* com os valores atualizados. Este tipo de *widget* é ideal para quando é necessário desenvolver um conjunto de *checkbox* ou de *radio buttons*. Sempre que o utilizador clica no *widget* ele muda de estado.

3.2.3.2 *Ciclo de vida dos Widgets*

O ciclo de vida de uma Activity -quando se está em Android nativo ou o ciclo de vida de uma ViewController quando estamos em iOS nativo- é um conceito extremamente importante no mundo do desenvolvimento mobile para entendermos o comportamento da nossa aplicação. Para o Flutter, é importante então perceber o ciclo de vida de um widget. Apesar de apresentarem algumas diferenças, cada um dos *widgets* tem um método *build* (BuildContext context) que descreve a parte da

UI representada por este *widget* e o contexto em Flutter representa a localização do próprio *widget* na árvore de *widgets*.

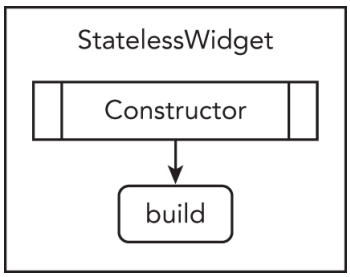


Figura 15: Estrutura de um *StatelessWidget*

O ciclo de vida de um StatelessWidget é bastante simples e, por não ter estado, não exige praticamente nenhum tipo de esforço por parte da Flutter Framework porque depois do *widget* ser construído não sofre nenhuma alteração, já que é imutável. Como é possível ver pela Figura 17, o *widget* primeiramente executa o seu construtor e de seguida executa o método *build*. A Figura 16 representa a estrutura de código Dart necessária para a construção de um StatelessWidget em que neste caso apenas é constituído por um Container vazio.

```dart
class MyWidget extends StatelessWidget {

  @override
  Widget build(BuildContext context) {
    return Container();
  }
}
```

Figura 16: Estrutura básica necessária para implementar um *StatelessWidget*

Pela análise da Figura 16, o ciclo de vida de um StatefulWidget parece, à primeira vista, tão complexo como o ciclo de vida de um StatelessWidget, pois ele também possui o construtor que é chamado primeiramente, sendo de seguida executado o método createState que tem como função criar o estado mutável para o *widget*.

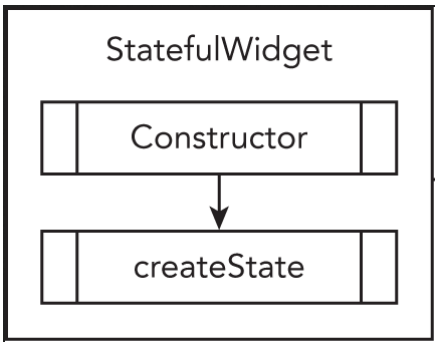


Figura 17: Estrutura de um *StatefulWidget*

Na Figura 18 está representada a estrutura básica necessária para implementar um StatefulWidget. Nela é possível reparar que a estrutura é um pouco diferente da estrutura de um StatelessWidget que irei explicar mais à frente. Em Dart, o uso da sublinha (*underscore*) antes de uma variável ou *widget*, significa que o widget é privado relativamente ao seu âmbito. O método createState irá instanciar o *widget* _MyWidgetState que é um estado do widget MyWidget.

```
class MyWidget extends StatefulWidget {
  @override
  _MyWidgetState createState() => _MyWidgetState();
}

class _MyWidgetState extends State<MyWidget> {
  @override
  Widget build(BuildContext context) {
    return Container();
  }
}
```

Figura 18: Estrutura básica necessária para implementar um *StatefulWidget*

Como o *widget* _MyWidgetState herda de uma classe State, ele irá herdar alguns comportamentos diferentes e é aqui que os tipos de Widgets se diferenciam bastante. De seguida irá ser analisado o ciclo de vida um objeto State que pode ser representado pelo esquema da Figura 19.

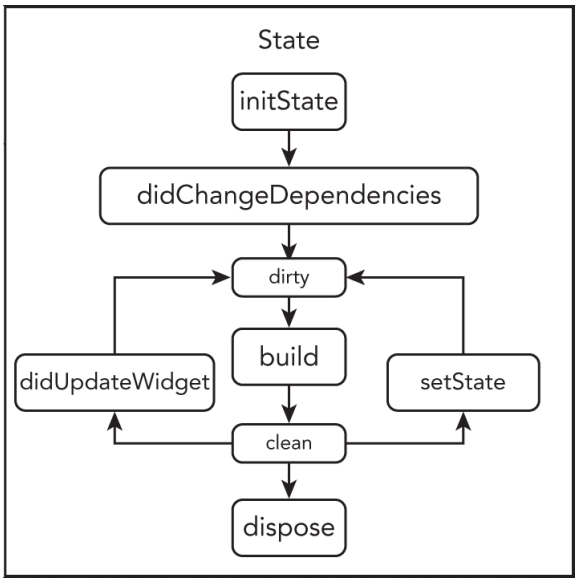


Figura 19: Ciclo de vida de um objeto *State*

Como é possível ver, o esquema é constituído por vários métodos dos quais:

- **initState:** é chamado apenas uma vez quando o objeto é inserido na árvore de widgets;;
- **dispose:** é chamado quando o objeto é removido da árvore de *widgets*;
- **didChangeDependencies:** é chamado sempre que o objeto State tem alterações;
- **build:** é chamado em diferentes situações sempre que o objeto State sofre alterações para reconstruir a UI;
- **setState:** é responsável por notificar a *framework* que o objeto State sofreu alterações para assim então chamar o método *build*;
- **didUpdateWidget:** é chamado quando a configuração do *widget* é alterado.

3.2.4 *Navegação*

A navegação entre ecrãs no Flutter utiliza o conceito de *"stack"*. Ou seja, quando estamos no ecrã 1 e navegamos para o ecrã seguinte, ecrã 2, internamente, estamos a colocar este por cima do ecrã 1. Quando queremos voltar ao ecrã 1, como o ecrã está no topo da "stack", basta remover o ecrã 2. Para ser possível manipular esta *"stack"* de ecrãs, é utilizado o *widget* Navigator, onde este é constituído por dois métodos principais, o push e o pop. Como os próprios nomes indicam, o método

push permite "empurrar" um novo ecrã para o topo da "stack", enquanto o método pop permite remover o ecrã que está no topo da *"stack"*. Existe também o método pushReplacement que permite substituir um ecrã por um outro na *"stack"*, por exemplo: se temos o ecrã 1, 2 e 3 na nossa aplicação sendo que o ecrã 3 está por cima do ecrã 2, e este está por cima do ecrã 1, quando é chamado o método pushReplacement do ecrã 2 para o ecrã 3, posteriormente quando removemos o ecrã 3 da *"stack"* a aplicação irá navegar para o ecrã 1, e não para o ecrã 2, pois o ecrã 2 foi substituído pelo ecrã 3. O método representado na Figura 20 mostra a sintaxe do método *push*, em que recebe dois parâmetros: o contexto do primeiro ecrã (para a *framework* saber onde está posicionado este *widget* na árvore de *widgets*) e o segundo parâmetro para indicar que *widget* cremos sobrepor a este.

```
Navigator.push(context, MaterialPageRoute(builder: (_) => Screen2(controller, context)));
```

Figura 20: Estrutura de código básica para implementar o método *push* do Navigator

O método representado na Figura 21 mostra a sintaxe do método *pop*, que como é possível ver é bastante simples, apenas é necessário um parâmetro para indicar o contexto do nosso *widget*, para mais uma vez, a *framework* saber onde este *widget* está posicionado na árvore de *widgets*.

```
Navigator.pop(context);
```

Figura 21: Estrutura de código básica para implementar o método *pop* do Navigator

O método *pushReplacement* do *widget* Navigator tem exatamente a mesma sintaxe que o método *push*.

3.2.4.1 *Estrutura de ficheiros*

De seguida, irá ser analisada a estrutura de ficheiros de uma aplicação em Flutter aquando é criada e como foi estruturada ao longo do projeto para ser possível ter uma aplicação escalável. O Flutter não fornece nenhum tipo de estrutura de ficheiros para o projeto (tirando algumas exceções que têm de ser respeitadas que irão ser faladas de seguida), deixando assim ao critério do desenvolvedor criar a estrutura que achar melhor. A seguinte imagem mostra a estrutura de ficheiros inicial quando é criado um projeto:

A **pasta android** contém todos as pastas e ficheiros necessários para executar uma aplicação Android, esses que são gerados automaticamente durante a criação

do projeto Flutter e que permanecem praticamente intocáveis a qualquer tipo de alteração ao longo do desenvolvimento de todo o projeto. As subpastas principais da pasta android são a pasta res, os ficheiros AndroidManifest.xml e build.gradle. A pasta res contém recursos não programáveis necessários para a aplicação, como ícones, imagens e fontes, o ficheiro AndroidManifest.xml contém informações necessárias para o SDK da aplicação, como a indicação das permissões que a aplicação irá precisar (escrever/ler no armazenamento, dados biométricos, etc.), o nome da aplicação, o ícone que aparece no ambiente de trabalho, entre outras configurações. E o ficheiro build.gradle apresenta configurações relacionadas com o *build* da aplicação onde se destacam a versão mínima do Android para instalar a aplicação, e a versão do Android onde a aplicação irá ser compilada, configurações de assinatura da aplicação para fazer *deploy* no PlayStore, entre muitas outras.

A **pasta ios** contém todos as pastas e ficheiros necessários para executar uma aplicação iOS, e tal como a pasta android, estes são gerados automaticamente durante a criação do projeto Flutter. O ficheiro info.plist é semelhante ao ficheiro AndroidManifest.xml em Android. Ele armazena informações necessárias pelo SDK da aplicação.

A **pasta lib** é a pasta mais importante de todo o projeto Flutter, pois é nela onde irá ser escrito a maioria do código Dart ao longo do desenvolvimento. Uma das regras que é preciso manter para respeitar, como foi referido anteriormente, é o facto de o ficheiro main.dart (que é o ficheiro de ponto de entrada do projeto) deve ser mantido na raiz da pasta lib.

A **pasta test** contém todos os ficheiros usados para desenvolver testes para a aplicação. Os ficheiros de teste devem terminar com "test.dart" que é a convenção usada pelo executor de testes quando vai procurar por ficheiros de testes.

O **ficheiro pubspec.yaml** também é um ficheiro bastante importante num projeto Flutter, é nele onde indicamos o nome, versão, descrição e dependências/pacotes. É neste ficheiro também que indicamos o caminho para arquivos estáticos tal como imagens ou fonte de letras para o Flutter puder executar, porque sem a indicação destes caminhos, não é possível os usar no código. Como é um ficheiro yaml, é importante respeitar a indentação de 2 espaços em branco.

O **ficheiro pubspec.lock** é um ficheiro que é gerado automaticamente quando é feito o *build* do projeto. Ele lista a versão de cada dependência utilizada no projeto (que são indicadas no pubspec.yaml), e a versão das dependências que dependem dessa mesma dependência indica no ficheiro.

Estes foram os ficheiros e pastas de maior relevância para um projeto desenvolvido em Flutter. Mas como é possível ver na Figura X existem mais ficheiros, sendo alguns já são conhecidos pois são bastante utilizados na maioria dos projetos de outras linguagens de programação, como é o caso do ficheiro .gitignor; sendo que este é constituído por ficheiros e pastas que o desenvolvedor quer ignorar quando o projeto é carregado para plataformas de controlo de versões. Já, O ficheiro README.md é um ficheiro em escrito em Markdown (linguagem simples de marcação), que é bastante usado para descrever a aplicação em repositórios como GitHub, GitLab, entre outros. Depois, Os ficheiros .metadata e .packages são ficheiros próprios do Flutter que devem ser incluídos no ficheiro .gitignore e não devem sofrer alterações por parte do desenvolvedor. O ficheiro .metadata contém os metadados necessários pela ferramenta Flutter para rastrear propriedades do projeto Flutter. Por fim, o ficheiro .packages contém o caminho para cada dependência usado no projeto. Antes de dar início ao desenvolvimento da aplicação, foi perdido algum tempo e dada importância ao modo como os ficheiros iriam estar estruturados, principalmente na pasta lib, porque como foi referido anteriormente, o Flutter não fornece uma estrutura de ficheiros específica. Manter uma boa estrutura de pastas é uma excelente prática de programação pois irá ajudar a prevenir alguns dos seguintes problemas:

- Incapacidade de encontrar um ficheiro específico (vista, modelo, serviço, controlador, etc.);
- Escrever blocos de código repetidos ao longo do projeto que poderiam ser escritos apenas uma vez para reaproveitar;
- Mistura do código relacionado com as vistas e código relacionado com os serviços de acesso à API, por exemplo;
- Confusão ao desenvolver a aplicação numa equipa com alguns elementos.

Porque apesar da aplicação estar a ser desenvolvida apenas por um elemento, é extremamente importante pensar que a aplicação tem que ser escalável tanto a nível de código, como a nível de elementos da equipa que nela irão trabalhar, a estrutura de pastas e ficheiros é importante refletir no processo inicial do desenvolvimento da aplicação. Posto isso, como em todos os projetos da empresa estavam a utilizar o padrão de arquitetura de software MVC (que irei falar mais à frente), ficou decidido que este iria ser o padrão utilizado no desenvolvimento da aplicação. Este padrão que tem como principal objetivo a separação das responsabilidades do nosso código, mas que influencia bastante na estrutura de pastas de um projeto.

De seguida irá ser enunciado a forma como ficou organizada a estrutura de pastas e ficheiros do projeto. Na raiz do projeto foram acrescentadas as seguintes pastas:

- Pasta **assets** – contém todas as imagens (png e svg), dados *mock*;
- Pasta **fonts** - contém as fontes de letra utilizadas na aplicação;
- Pasta **lang** – contém os ficheiros de traduções em formato json pois a aplicação permite a internacionalização. A aplicação permite ter como idiomas o português e o inglês, então a pasta lang tem os ficheiros pt.json e en.json;
- Ficheiro **.env** - contém variáveis de ambiente utilizadas por todo o projeto (como variáveis para serem utilizadas nos testes, urls e portas de *hosts* utilizados para acessar a API, etc);
- Ficheiro **.gitlab-ci.yml** - responsável por configurar algumas instruções específicas utilizados pelo GitLab CI/CD. Este tema irá ser aprofundado mais à frente quando for relatado o processo de *build* e *deploy* da aplicação.

Na raiz da pasta lib (a pasta que contém a maioria do código da aplicação) ficou organizada com as seguintes pastas:

- Pasta **models** - contém todos as classes de modelo que segundo o MVC, consistem na parte lógica da aplicação, que é responsável pelo acesso e manipulação dos dados;
- Pasta **views** - contém maioritariamente *widgets* que representam parte gráfica da aplicação que será apresentada ao utilizador;
- Pasta **controllers** - contém os controladores que, segundo o MVC, é responsável por ligar os modelos às vistas, fazendo assim com que os dados armazenados nos modelos possam ser representados nas vistas. Neste projeto, o trabalho dos controladores irá centrar-se muito em manipular a resposta fornecida pela API, e passar essa informação, de forma estruturada pelos modelos, para as vistas;
- Pasta **services** - contém ficheiros constituídos por métodos responsáveis por acessar os *endpoints* da API que devolvem a resposta;
- Pasta **theme** - contém os ficheiros de temas da aplicação, como cores, fonte de letras, entre outros. Nesta pasta contém ficheiros para definir as cores para o tema claro e o tema escuro da aplicação;
- Pasta **utils** - contém todos os ficheiros com Helpers (como por exemplo funções para mostrar um dialog, ou para formatar números que são usadas várias vezes por todo o projeto), funções para escrever e ler informações de armazenamento seguro no telemóvel (Keychain em iOS, KeyStore em Android), ou numa base de dados SQLite local, entre muitos outros métodos...

3.3 REQUISITOS FUNCIONAIS E NÃO FUNCIONAIS

De seguida, serão enunciados os requisitos funcionais, que definem o que é que a aplicação deve fazer, ou seja, acabam por ser todas as funcionalidades planeadas para serem parte da aplicação e, os requisitos não funcionais que definem como é que a aplicação deve funcionar, ou seja, são características e/ou limites obrigatórios relacionados com desempenho, disponibilidade, manutenção, entre outros aspetos que a aplicação deve respeitar enquanto está a ser utilizada.

3.3.1 *Requisitos funcionais / Funcionalidades*

Como foi explicado anteriormente, o objetivo seria ter uma aplicação móvel com algumas das funcionalidades que o próprio portal Fact já tem, mas claro, adaptadas a uma realidade mobile com todas a utilidade que estes tipos de equipamento possuem, e com isto foi sugerido as seguintes funcionalidades para serem desenvolvidas durante todo o tempo do projeto:

- Autenticação com as credenciais Fact em que é necessário um email e uma password, em que se houver sucesso no login é pedido um código PIN e, opcionalmente, a impressão digital para o utilizador não ter necessidade de preencher as credenciais de autenticação ao voltar a entrar na aplicação;
- Uma página de Definições onde apresenta dados do utilizador, lista de empresas, o ecrã que o utilizador pretende ser aberto quando a aplicação é aberta, idioma da aplicação, alteração do código PIN e se deve ser utilizado os dados biométricos para iniciar sessão na aplicação;
- Secção com os dados do suporte técnico do Fact;
- Possibilidade de o utilizador escolher qual empresa deseja visualizar todos os dados;
- Dashboard de Vendas com um conjunto de gráficos apropriados;
- Dashboard de Depesas com um conjunto de gráficos apropriados;
- Dashboard de Performance um conjunto de gráficos apropriados;
- Dashboard da Posição Integrada e um conjunto de gráficos apropriados;
- Lista das contas bancárias da empresa selecionada e respetivos movimentos;

- Em cada movimento de cada conta bancária exibir uma pré-visualização do documento com ele reconciliado;
- Lista de Entradas e detalhe dos repetivos anexos/documentos com a possibilidade de estes poderem ser lançados;
- Edição e classificação dos documentos recebidos nas entradas;
- Notificações *push* e respetiva lista.

Com todas as funcionalidades definidas, estas serão aprofundadas a um nível mais técnico no subcapítulo 3.4 onde será explicado de que forma foram todas estas funcionalidades desenvolvidas.

3.3.2 *Requisitos não funcionais*

Tão importante como a lista dos requisitos funcionais, está a lista de requisitos não funcionais que acabam por ser fatores externos ao desenvolvimento da aplicação mas que se tornam essenciais para não haver falhas na utilização da aplicação. Posto isto, os requisitos não funcionais da aplicação foram definidos da seguinte forma:

- A aplicação deve funcionar com o máximo desempenho sem haver quebras durante a utilização da mesma;
- A aplicação deve prever os casos de falta de conectividade, apresentado feedback ajustado ao utilizador;
- A aplicação deve ter uma alta disponibilidade, espero estar 99% do tempo disponível para ser utilizada com o tempo restante utilizado para manutenções;
- A aplicação deve guardar alguma informação internamente para algumas das funcionalidades possam ser acedidas sem conexão à internet, como por exemplo os dados das *dashboards* podem ser visualizados com os últimos dados disponíveis ou a visualização de documentos;
- A aplicação deve ser compatível com a maioria dos smartphones dos utilizadores Android. A versão mínima Android para a aplicação funcionar deve ser o Android 4.4.
- A segurança dos dados deve estar salvaguardada e apenas ser acedida pelo utilizador responsável por aqueles dados, ou seja, deve haver um forte sistema de autenticação.

- O utilizador deve ter sempre feedback do que a aplicação está a fazer. Por exemplo, sempre que forem carregados novos dados para uma lista a aplicação deve apresentar um *loading*;
- A apresentação deve ser sempre cuidada seja qual for a resolução do ecrã do dispositivo do utilizador;
- A aplicação deve possuir idiomas em que a maioria dos utilizadores seja capaz de utilizar.

3.4 DESENVOLVIMENTO DA APLICAÇÃO

Depois de escolher o Flutter como principal tecnologia a trabalhar e de aprofundar alguns conhecimentos sobre a própria *framework*, era hora de desenvolver a aplicação móvel, o principal objetivo deste projeto. De seguida, irão ser aprofundadas todas as funcionalidades descritas no subcapítulo 3.3.1 e de que forma foram desenvolvidas.

3.4.1 *Menus de navegação*

Primeiramente foi desenvolvido o "esqueleto"da aplicação que neste caso representa os dois principais widgets de navegação: o BottomNavigationBar e o DrawerNavigator.

Na Figura 22 representa o widget BottomNavigationBar que consiste no menu que está localizado na parte inferior e que está sempre vísivel em qualquer parte da aplicação. Segundo o *Material Design* da Google (que é um sistema de design desenvolvido pela Google para ajudar as equipas de desenvolvimento a construir as aplicações onde apresenta imensos *guidelines* relacionados com o layout), um BottomNavigationBar deve conter entre 3 a 5 itens diferentes porque mais que isso já devem ser representados em menus laterais como o DrawerNaviagator que será abordado de seguida. Posto isto, os os menus representados no BottomNavigationBar devem ser os principais menus da aplicação e, supostamente, os que irão ser mais utilizados.

Em termos visuais, os menus são sempre representados por um ícone e um título associado em que quando o menu está ativo deve apresentar a cor verde, e quando o menu está inativo deve conter a cor cinzenta. Outro aspeto que pode ser utilizado neste tipo de menus são os *badges* que estão representados de cor vermelha no canto superior do menu. Estes *badges* são utilizados quando há nova informação a ser consultada. No contexto desta aplicação, quando chega um novo email à lista de

Entradas, ou existe uma nova notificação por ler é, então, apresentado um *badge* vermelho junto ao menu correspondente. É bastante comum também apresentar dentro do *badge* o número de "nova informação"que existe, ou seja, por exemplo o número de notificações que o utilizador tem por ler. Neste caso, foi optado por não utilizar este tipo de abordagem pois acrescentava algum ruído visual à aplicação.

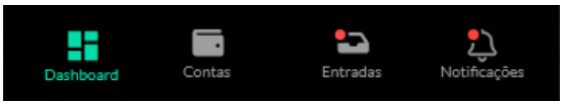


Figura 22: Representação do menu inferior da aplicação

O widget DrawerNavigator é um menu que fica localizado na parte lateral esquerda, e ao contrário do BottomNavigatorBar não está sempre visível. Este tem que ser aberto através de *Hamburger button* (que tem este nome pois o ícone do botão apresenta semelhanças a um hambúrguer) que fica localizado no canto superior esquerdo. O menu lateral da aplicação pode ser observado na Figura 23. Neste menu é representado o avatar, o nome e o email do utilizador autenticado, um botão para carregar um documento para a Gestão Documental (esta funcionalidade não foi desenvolvida durante o trabalho deste projeto), um menu para a lista das empresas do utilizador, uma página de definições, uma página de Suporte e um botão para terminar sessão.

Por fim, houve particular cuidado em assegurar os escalamentos automáticos de forma a que a apresentação ficasse sempre bem cuidada qualquer quer seja a resolução de ecrã do dispositivo do utilizador. Foi também incluído neste processo uma reformatação para colmatar os casos de aplicação correr em tablets ou dispositivos de média/grande dimensão.

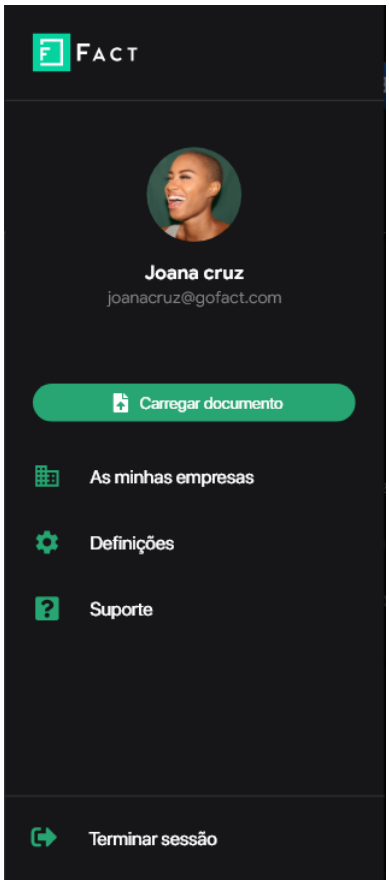


Figura 23: Representação do menu lateral esquerdo da aplicação

3.4.2 *i18n*

Como a aplicação iria ter a opção de possuir diferentes idiomas, este "ambiente"teria que ser preparado o mais cedo possível porque quantas mais funcionalidades são feitas maior o número de traduções que serão de ser feitas, e então esta funcionalidade foi das primeiras a ser implementada. O nome dado a este processo de preparar a aplicação para suportar diferentes idiomas é dado pelo nome de *i18n*. *i18n* é uma abreviação para a palavra *internationalization* (internacionalização), e é conhecida por esta a abreviação porque 18 é o número de letras entre a letra "i"e a letra "n". Para isto foi necessário adicionar o *package* "intl"que contém código preparado para a internacionalização da aplicação. Para cada idioma diferente, foi criado um ficheiro json com o nome de código do país seguindo a norma ISO 639 (norma que sugere códigos para os nomes de todas as línguas), em que cada ficheiro continha todas as traduções em formato JSON, ou seja, todos os ficheiros de traduções possuiam as

mesmas *keys* e em que o valor era a tradução. Estes ficheiros de traduções podem ser analisados através da Figura 24

```
"support.title": "Suporte",
"support.contacts": "Contactos",
"support.company": "Empresa",
"support.address": "Morada",
"support.phone": "Telefone",
"support.website": "Website",
"support.email": "Email",
"support.phoneSupport": "Suporte telefónico",
```

```
"support.title": "Support",
"support.contacts": "Contacts",
"support.company": "Company",
"support.address": "Address",
"support.phone": "Phone",
"support.website": "Website",
"support.email": "Email",
"support.phoneSupport": "Phone support",
```

Figura 24: Ficheiros de traduções dos idiomas Português e Inglês

Foi criada uma classe, como é possível verificar pela Figura para carregar todas as traduções disponíveis no ficheiro de traduções e uma função chamada *translate* que retorna a tradução através da *key*, função essa que será utilizada através de todo o projeto.

```
class AppLocalizations {
  Locale locale;
  bool isTest;

  AppLocalizations(this.locale, {this.isTest = false});

  static AppLocalizations? of(BuildContext context) {
    return Localizations.of<AppLocalizations>(context, AppLocalizations);
  }

  static const LocalizationsDelegate<AppLocalizations> delegate = AppLocalizationsDelegate();

  Map<String, String>? _localizedStrings;

  Future<bool> load() async {
    String jsonString = await rootBundle.loadString('lang/${locale.languageCode}.json');
    Map<String, dynamic> jsonMap = jsonDecode(jsonString);

    _localizedStrings = jsonMap.map((key, value) {
      return MapEntry(key, value.toString());
    });

    return true;
  }

  Future<AppLocalizations> loadTest(Locale locale) async {
    return AppLocalizations(locale);
  }

  String translate(String key) => _localizedStrings![key] ?? "";
}
```

Figura 25: Classe para carregar as traduções dos idiomas

A partir da implentação do *i18n* era importante que sempre que fosse acrescentada uma nova palavra ou frase na aplicação esta iria, obrigatoriamente, fazer parte do ficheiro de traduções de todas os idiomas que estão disponíveis no projeto.

3.4.3 *Comunicação com a API*

Para a maioria das próximas funcionalidades a implementar, era importante configurar a comunicação com a API do Fact para ler, criar ou atualizar dados. Para esta implementação foi utilizado um *package* com o nome *dio* que é um poderoso cliente HTTP (*Hypertext Transfer Protocol*) que suporta *Interceptors* (que tem a função de intercetar e manipular pedidos HTTP), confgiurações globais, cancelamento de pedidos, *download* de ficheiros, etc. Foi desenvolvido um novo *Interceptor* para que em todos os pedidos fosse aplicados os seguintes *headers*:

- **Cookie**: que contém o token de autenticação recebido quando é feito o login na aplicação para a API (*Application Programming Interface*) saber qual é o utilizador autenticado. Este aspeto será aprofundado no subcapítulo 3.4.4.
- Header que contém o ID da empresa seleccionada para que quando forem pedidos novas informações sejam sempre para uma empresa específica.
- Header para que o tipo dos recursos recebidos sejam do MIME Type: *application/json*.

Como se pode ver pela Figura 26, foi implementada uma nova classe que acaba por ser uma abstração do cliente "Dio"para que sempre que se chama esta nova classe seja aplicado o *baseUrl* da API e o interceptor anteriormente referido. Os *endpoints* da API foram implementados numa aquitetura RESTful. Aqui é importante referir que foi utilizada o *package* flutter_dotenv para utilizar o ficheiro *.env* como variáveis globais que podem ser acedidadas por todo o projeto, em que estas variáveis são declaradas com chave: valor. A variável *baseUrl* foi declarada no ficheiro *.env* com a chave "FACT_API_HOST"em que o valor era o *endepoint* da API do Fact. A partir daqui, foi sempre utilizado este cliente *DioClient* para fazer os pedidos à API com qualquer método HTTP seja GET, POST ou DELETE.

```
class DioClient {
  final Dio dio;

  DioClient(this.dio) {
    dio.options.baseUrl = "${env['FACT_API_HOST']}";
    dio.interceptors.add(APIInterceptor());
  }
}
```

Figura 26: Classe para criar um cliente Dio *custom*

3.4.4 *Autenticação*

Posteriormente, foi desenvolvido uma funcionalidade bastante importante para a maioria dos *softwares*, a autenticação do utilizador na aplicação. A autenticação foi dividida em três tipos diferentes:

- *Login* com as credenciais (email e *password*) através da API;
- Preenchimento de um código PIN com 4 números e respetiva confirmação depois do Login ser efetuado com sucesso;
- Página para definir se o utilizador deseja ou não iniciar sessão na aplicação através da impressão digital, onde apenas contém dois botões: "Usar a impressão digital"(para utilizar a impressão digital) e "Agora não"(para não utilizar a impressão digital, mas existe sempre a possibilidade de alterar esta definição mais tarde nas Definições da aplicação).

A primeira página que o utilizador vê sempre que inicia a aplicação é a *Splash Screen* que está representada na imagem da esquerda na Figura 27. Uma *Splash Screen* é a página de transição enquanto a aplicação está a ser carregada. Na imagem da direita está representada a página de Login. Quando as credenciais estão incorretas depois do utilizador carregar no botão Login os *inputs* de email/*password* ficam com a borda em vermelha e o telemóvel emite uma vibração através do *package* vibration. Quando o login é efetuado com sucesso, os dados do utilizador, retornados do login, são armazenados numa base de dados SQLite do telemóvel (este tema será descrito com detalhe no subcapítulo 3.4.5). Aliás isto foi uma das vantagens da escolha do Flutter, uma vez que já incluía métodos abstraídos para interação com o *hardware* de reconhecimento de impressão digital.

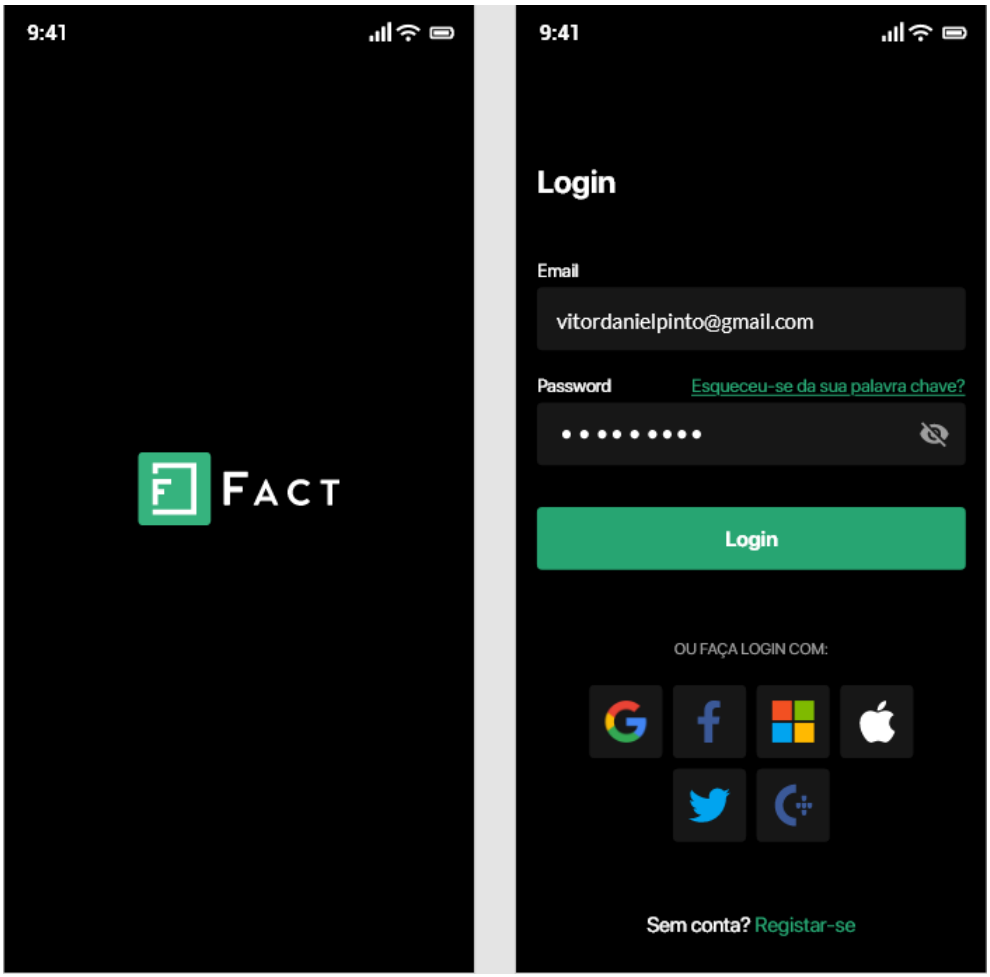


Figura 27: Página de Login da aplicação

De seguida, é apresentada a página para preencher o código PIN que contém 4 algarismos e respetiva página para confirmação do código PIN. Depois do código PIN ser preenchido em ambas as páginas o código PIN é guardado na *secure storage* do telemóvel (este tema será descrito com detalhe no subcapítulo 3.4.5).

Por fim, é apresentado a página para definir a informação sobre a impressão digital. A informação escrita acaba por ser apenas um *boolean* onde é armazenada junto dos dados do utilizador na base de dados SQLite;

A partir daqui, sempre que a aplicação é iniciada, se a informação da impressão digital estiver ativa, é pedido para colocar a impressão digital, se não, é pedido o código PIN onde o utilizador tem três hipóteses para o colocar. Se houver 3 falhas no preenchimento do PIN é feito automaticamente o logout da aplicação. Se o utilizador tiver a impressão digital ativa, também poderá utilizar a abordagem do código PIN se tiver preferência.

3.4.5 *Armazenamento dos dados*

No desenvolvimento desta aplicação foram utilizados dois tipos de abordagens para armazenar dados na memória interna do telemóvel: *Flutter Secure Storage* (através do *package* flutter_secure_storage) e base de dados SQLite (através do *package* sqflite).

O *Flutter Secure Storage* tem várias abordagens de armazenamento consoante o sistema operativo do telemóvel em questão. No caso do Android, é utilizado o Sistema *Keystore* em que são armazenadas chaves criptográficas num *container* para dificultar o acesso a este tipo de informação. Neste tipo de armazenamento foram guardados informações bastante importantes como o *token* e o código PIN. Foi criada uma classe para lidar com os dados armazenados na *Flutter Secure Storage*, como se pode ver pela Figura 28: o primeiro método escreve o valor do token, o segundo método regista o código PIN e o terceiro método elimina os dois valores da *storage*.

```dart
class AuthSession {
  FlutterSecureStorage storage = FlutterSecureStorage();

  /*
  * Starts a new session.
  *
  * Will drop existing session.
  */
  Future<void> start(String token) async {
    end();

    await storage.write(key: "token", value: token);
  }

  Future<void> registerPINCode(String pin) async {
    await storage.write(key: "pin", value: pin);
  }

  /*
  * Ends session.
  */
  Future<void> end() async {
    await storage.delete(key: "token");
    await storage.delete(key: "pin");
  }
```

Figura 28: Classe para lidar com a *Flutter Secure Storage*

O *package sqflite* é uma biblioteca que proporciona métodos abstraídos para lidar com um motor de SQL SQLite. Ao contrário de outras base de dados, esta é bastante leve, ótima para armazenar dados num telemóvel. Na Figura 29 mostra alguns métodos necessários para a configuração da base de dados SQLite. É necessário o método *_initDatabase* para abrir a base de dados, e o método *_onCreate* que contém o código para criar as tabelas necessárias para guardar os dados. Neste caso foi criada a tabela "Users".

```
class DatabaseHelper {
  static final _databaseName = "Fact.db";
  static final _databaseVersion = 1;

  DatabaseHelper._privateConstructor();
  static final DatabaseHelper instance = DatabaseHelper._privateConstructor();

  Database? _database;

  Future<Database> get database async {
    if (_database == null) return _database!;
    _database = await _initDatabase();
    return _database!;
  }

  _initDatabase() async {
    Directory documentsDirectory = await getApplicationDocumentsDirectory();
    String path = p.join(documentsDirectory.path, _databaseName);

    return await openDatabase(path, version: _databaseVersion, onCreate: _onCreate);
  }

  Future _onCreate(Database db, int version) async {
    await db.execute('''
              CREATE TABLE ${User.tableName} (
                ${User.columnId} INTEGER PRIMARY KEY,
                ${User.columnEmail} TEXT NOT NULL,
                ${User.columnName} TEXT NOT NULL,
                ${User.columnTin} TEXT,
                ${User.columnAvatar} TEXT NOT NULL,
                ${User.columnFingerPrint} INTEGER NOT NULL,
                ${User.columnLoginAttempts} INTEGER NOT NULL
              )
              ''');
  }
```

Figura 29: Classe para lidar com a SQLite

3.4.6 *Definições*

A página de Definições é aberta através do menu presente no menu lateral esquerdo da aplicação (*DrawerNavigator* e é constituída por vários menus que têm funções

bastante simples como estão representadas através da Figura 30. Esta página contém os seguintes menus:

- **Os meus dados**: apenas contém os dados do utilizador (nome completo, morada, telemóvel, email, etc).
- **As minhas empresas**: contém a lista de empresas do utilizador autenticado, onde também é possível definir qual a empresa aparece por defeito ao abrir a aplicação. Esta página também poderá ser inicializada através do menu da lista de empresas que está presente no *DrawerNavigator*;
- **Tema**: permite escolher se a aplicação contém o tema claro, escuro ou o tema por defeito do sistema;
- **Ecrã de abertura**: permite definir qual dos menus presentes no *BottomNavigationBar* será aberto quando a aplicação é iniciada;
- **Idioma**: permite alterar o idioma da aplicação, onde neste momento apenas estão disponíveis as línguas Português e Inglês.
- **Alterar código PIN**: possibilidade de alterar o código PIN, onde é necessário preencher primeiro o código PIN atual e depois preencher o novo e respetiva confirmação;
- **Iniciar sessão com dados biométricos**: possibilidade de alterar se o utilizar deseja iniciar sessão com a impressão digital ou não;
- Lista dos termos e condições do serviço do Fact.

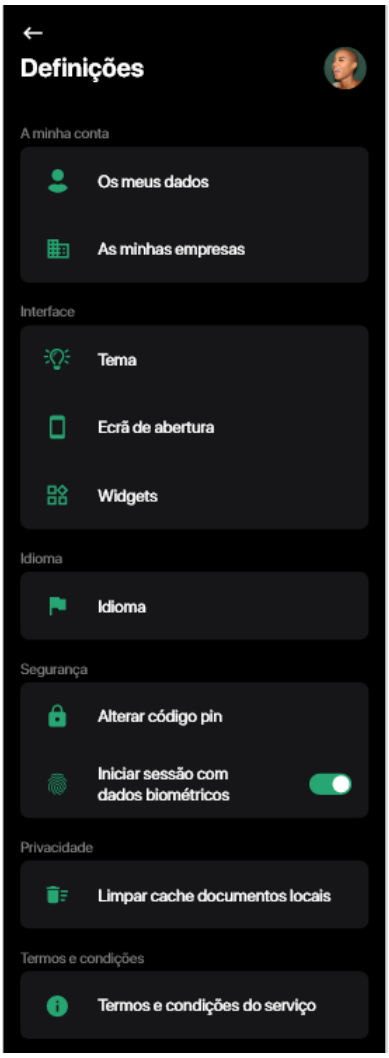


Figura 30: Classe para lidar com a SQLite

3.4.7 *Dashboards*

As *dashboards* são das funcionalidades mais importantes e a que tem o maior número de vistas de toda a aplicação pois há quatro páginas de *dashboards*: Vendas, Despesas, Performance e Posição Integrada. Em cada uma destas *dashboards* há um vasto conjunto de gráficos em que o principal *package* utilizado para fornecer todo o tipo de gráficos foi o *syncfusion_flutter_charts*. Aqui o trabalho foi bastante simples mas demoroso pois havia um grande número de gráficos para implmentar. Para cada *widget* é chamado um novo *endpoint* da API. As Figuras 31, 32, 33 e 34 representam 3 exemplos dos gráficos implementados de cada uma das diferentes *Dashboards*.

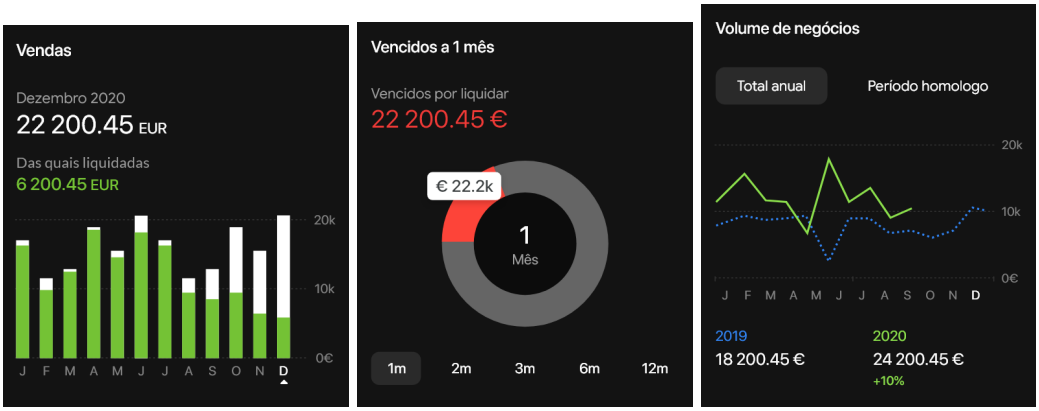


Figura 31: Exemplos de gráficos da *Dashboard* de Vendas

Figura 32: Exemplos de gráficos da *Dashboard* de Depesas

Figura 33: Exemplos de gráficos da *Dashboard* de Performance

Figura 34: Exemplos de gráficos da *Dashboard* da Posição Integrada

3.4.8 *Gestão de estado*

Um aspeto muito importante a ter em conta hoje em dia no desenvolvimento de aplicações modernas (sejam aplicações Web ou aplicações móveis) é a gestão de estado. Este tema é útil por exemplo quando é necessário passar dados entre *widgets*. Foram utilizadas duas abordagens na gestão do estado no desenvolvimento desta aplicação:

- setState;
- MobX;

3.4.8.1 *setState*

É uma função que recebe uma outra função por parâmetro e que já vem por omissão no Flutter sendo apenas utilizada StatefulWidgets. Quando esta é chamada. *framework* é notificada que o objeto *State* mudou e então atualiza o conteúdo da UI.

Um caso bastante simples em que este tipo de abordagem foi utilizada foi, por exemplo, nas Definições quando é pretendido modificar a informação relativa a "Iniciar sessão com dados biométricos". Existe uma variável booleana que gere se o *switch* está ativo ou não, e sempre que é clicado no *switch* para atualizar a informação, é chamado o setState para mudar o valor dessa variável e então atualizar a UI para o círculo branco do *switch* estar deslocado para a esquerda ou para a direita. Na Figura ?? está representado a definição do *switch* descrito anteriormente,

e como pode ser visualizado na função *onToggle* (que é chamada sempre que se clica no *switch*), a variável é atualizada dentro do setState com o novo valor e é também atualizado a informação na base de dados na tabela "Users"como foi referido no Subcapítulo 3.4.4.

```
FlutterSwitch(
    value: isFingerPrint,
    width: 47,
    height: 22,
    activeColor: Color(0XFF27A671),
    padding: 1.25,
    toggleSize: 20,
    onToggle: (s) async {
      setState(() {
        isFingerPrint = s;
      });
      await _database.updateFingerPrintUser();
    }) // FlutterSwitch
```

Figura 35: Widget do *switch* para atualizar a informação relativa à impressão digital

3.4.8.2 *MobX*

MobX é uma biblioteca do *package mobx* que, como referido anteriormente, de gestão de estado que simplifica a conexão dos dados reativos a novas mudanças da aplicação com a UI. No coração do MobX existem três conceitos imporantes: *Observables, Actions* e *Reactions*. Os *observables* guardam o estado de um determinado objeto ou variável que é reactivo a novas mudanças. Os *observables* no MobX por *default* são definidos da seguinte forma:

```
final counter = Observable(0);
```

Mas graças ao *package mobx_codegen* que também usado junto do *package mobx* que permite adicionar anotações para o código ficar mais legível, os *observables* podem ser defenidos da seguinte forma:

```
@observable
int value = 0;
```

As *Actions* são as responsáveis por modificar os valores dos *observables* que geralmente são funções. Estas podem ser representadas, por exemplo, da seguinte forma:

```
@observable
```

```
int value = 0;

@action
void increment() {
    value++;
}
```

Aqui é possível observar que é utilizado a anotação "@action"quando é implementada uma nova *Action*.

As *Reactions* são observadores que estão sempre reactivas ao sistema e são notificadas quando um *observable* muda de estado. Junto destes *packages* foi utilizado também o *package flutter_mobx* em que propõe um *widget* chamdo *Observer* que fica à escuta das mudanças dos *observables* e automaticamente faz a reconstrução da UI quando existe uma nova mudança. Este *widget* foi bastante utilizado quando eram feitos pedidos à API, e enquanto não chegavam os dados (ou seja a variável que continha os dados estava a *null*), era apresentado um *loading* para o utilizador como forma de feedback, e mal o pedido fosse concluído a variável era então atualizada, mudando o seu estado e por conseguinte o *widget* era reconstruído.

3.4.9 *Notificações push*

Uma *feature* bastante neste projeto foram as notificações *push*. Uma notificação *push* é uma notificação que aparece em tempo real no dispositivo mesmo que o utilizador não tem solicitado. São muito utilizado, no contexto da aplicação, por exemplo quando é criado um novo documento e todos os utilizadores recebem uma notificação dessa mesma informação, quando são alteradas permissões dos utilizadores da empresa, entre muitos outros aspetos.

Este tipo de notificações, foi desenvolvida através do serviço Firebase Cloud Messaging da Google.

3.5 PROCESSOS DE *BUILD* E *DEPLOY*

De seguida, irá ser dado ênfase aos processos de *build* (compilação) e de *deploy* (implantação/instalação) da aplicação móvel. Fica aqui como nota que durante todo o relatório irão ser referidos os processos com a denominação em inglês por estas serem mais usadas no mundo das Tecnologias de Informação. Estas duas fases são

imensamente importantes porque é através delas que iremos ver o resultado final de todo o desenvolvimento e, então testar e utilizar o produto.

De forma sucinta, a fase de *build* consiste no processo de compilação de todos os recursos (que consistem em arquivos adicionais ao código e conteúdo estático usado pelo código) e código fonte do projeto e empacotar tudo (*packaging*) num APK (*Android Package*) e/ou AAB (*Android App Bundle*) que este último irá ser usado para a próxima fase. A fase de *deploy* representa a etapa do ciclo de software que consiste na passagem do software desenvolvido para produção, e no caso das aplicações móveis, representa quando a aplicação fica disponível na loja de aplicações.

3.5.1 *Build*

Antes de dar início ao processo de *build* da aplicação, era fulcral possuir um ambiente de desenvolvimento apropriado ao tipo de sistema operativo. Como se trata de uma aplicação *cross-platform*, é possível realizar o *build* para vários sistemas operativos, como o Android e/ou iOS. No decorrer do projeto houve apenas foco no sistema Android para acelarar o MVP e muito também por ser o sistema com o maior número de utilizadores. Inicialmente, foi implementado um *script* em Shell em que o objetivo seria com uma só execução do *script* fosse possível configurar uma máquina virtualizada com todo o ambiente necessário para fazer o *build* e *deploy* do projeto. A decisão para implementar este *script* baseou-se no facto de quando fosse necessário trocar para uma máquina diferente ou adotar um *deployment* em Docker, o processo de todos as *libraries* necessárias ser feito com maior agilidade possível. Um dos propósitos de se adotar a implementação em máquina virtual, advém da empresa já ter uma estrutura de máquinas virtuais a correm como *Runners* do sistema GitLab. Então, a implementação do *script*, ou seja, a configuração de uma máquina com o ambiente de desenvolvimento necessário para o projeto consistiu nos seguintes passos:

- Instalação do Android SDK (*Software Development Kit*) que consiste num conjunto de ferramentas para desenvolver aplicações para plataforma Android que inclui bibliotecas necessárias, um *debugger*, um *emulator*, entre outras ferramentas para ser possível também fazer o build da aplicação;
- Instalação do *sdkmanager* que é um comando que permite instalar e atualizar pacotes;

- Instalação do *package* Android SDK Platform API 29 (neste caso para fazer *build* para versão 10 do Android) através do comando *sdkmanager*;
- Instalação do Flutter SDK para ser possível compilar o código fonte para aplicações nativas, neste caso seria útil para compilar para uma aplicação Android;
- Instalação da *library fastlane* para automatizar o processo de colocar a aplicação na loja de aplicações (será detalhada mais à frente);
- Instalação da aplicação GitLab Runner e registo de um novo Runner para ter automatização de fluxos do GitLab. Esta aplicação irá ser mais detalhada de seguida no subcapítulo que fala sobre a automatização dos processos de *build* e *deploy*.

3.5.2 *Deploy*

Depois de o *build* estar concluído é hora agora de disponibilizar a aplicação em produção, ou seja, colocar a aplicação na Google Play. Como foi possível ver anteriormente, o resultado do *build* do projeto consiste em dois ficheiros com formato .aab e apk. O ficheiro com formato .aab (denominado de *Android App Bundle*) é um formato de publicação e é este que irá ser usado para fazer o upload na loja de aplicações, ao contrário do ficheiro com formato .apk que é um formato de *packaging* que eventualmente será instalado diretamente no telemóvel do utilizador em questão. A Google usa os *App Bundle* para gerar e disponibilizar os APKs otimizados para a configuração do telemóvel de cada utilizador para ser feito apenas o download do código e dos recursos que são necessários para a execução da aplicação no dispositivo, como é possível observar pela Figura 36. Até à data de agosto de 2021 ainda era possível fazer o upload de APKs ou AABs na Google Play, mas a partir daí apenas é permitido o upload dos ficheiros de formato .aab.

Figura 36: Relação entre os formatos AAB e APK

Para garantir a segurança, antes de ser feito o *deploy* da aplicação na Google Play, o Android exige que todos os APKs (que irão ser instalados no telemóvel do utilizador) sejam assinados digitalmente com um certificado. A assinatura da aplicação permite que os programadores identifiquem quem é o utilizador da aplicação e atualizem a sua aplicação sem criar permissões complicadas. É obrigatório que as aplicações sejam assinadas pelos programadores, porque sem isso a aplicação será rejeitada no momento de upload na Google Play. Este é importante também, porque mesmo que alguém tenha acesso às credencias da Google Play, seria impossível lançar uma atualização maligna sem também ter acesso à chave privada de upload. Com o lançamento do novo formato AAB, é apenas necessário assinar este com uma chave de upload antes de fazer o upload e de seguida a Assinatura de aplicações do Google Pay trata de assinar cada um dos APKs. A Assinatura de aplicações do Google Play utiliza dois tipos de chaves: a chave de assinatura da aplicação e a chave de upload. A chave de upload deve ser gerada pelo programador (onde irá ser explicado de seguida como é feito) e é usada para assinar as aplicações quando for feito o upload na Google Play. O Google usa o certificado de upload para verificar a identidade da chave de upload e assina os APKs com a chave de assinatura da aplicação, como é possível verificar pela imagem seguinte. Esta chave de assinatura da aplicação é uma cópia da chave de upload que a Google guarda na primeira vez que aplicação é colocada na PlayStore, e sempre que é feito uma nova atualização na aplicação é verificada se a chave de upload usada pelo programador é igual à chave de assinatura da aplicação, que está na posse da Google, e se estas forem iguais então o novo *package* é aceite e é então feita a distribuição pela Google Play. É possível verificar a relação entre estes dois tipos de chaves pela observação da Figura 37.

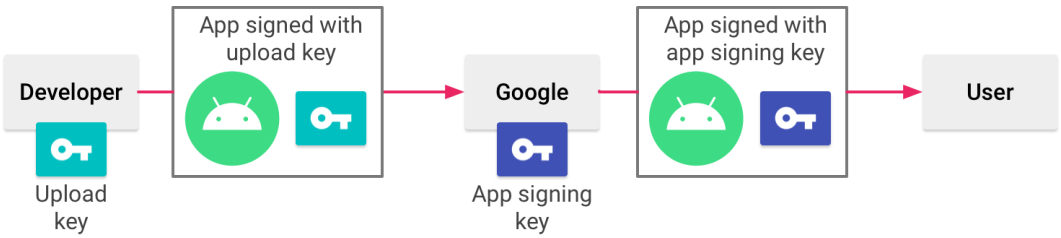


Figura 37: Interligação entre a chave de *upload* e a chave de assinatura

A chave de upload deve ser uma *Keystore* Java que são arquivos binários que servem como chave privada. Esta chave deve ser gerada através do comando *keytool*:

"keytool -genkey -v -keystore <key store file path> -keyalg RSA -keysize 2048 -validity 10000 alias <alias>"

Este comando aceita como parâmeros o local onde será guardada a chave, o algoritmo de encriptação, o tamanho, a validade (em dias), uma alias e por último é pedida uma password que é usada para aceder a esta chave privada. Com a chave criada, é necessário criar um ficheiro de configurações na pasta android com o nome de key.properties. Este ficheiro contém apenas 4 linhas com a atribuição de 4 variáveis que irão ser usadas pelo Android para fazer a assinatura da aplicação. As 4 variáveis e as suas atribuições de valor são os seguintes:

- **storePassword:** password que foi inserida aquando da criação da *keystore*;
- **keyStore:** password que foi inserida aquando da criação da *keystore*;
- **keyAlias:** alias preenchido no comando da criação da *keystore*;
- **storeFile:** localização da *keystore*.

A configuração deste ficheiro tem que ser feita a cada *build* nova, posto isto esta configuração foi colocada no *script* de automatização do processo de *build* que será detalhado mais à frente.

O passo final do processo de deploy de uma aplicação móvel consiste no upload do *package* resultante da build na loja de aplicações, neste caso na Google Play. O processo de fazer um novo upload sempre que é feita uma release torna-se bastante dispendioso em termos de tempo e desnecessário pois é sempre o mesmo fluxo. Então surgiu a necessidade de ter uma ferramenta que fizesse este processo de forma automática e depois de uma pequena pesquisa foi encontrado o *fastlane*. O *fastlane* é um conjunto de ferramentas *open source* para automatizar os lançamentos e implantações de uma aplicação móvel. Posto isto foi necessário correr o comando *"fastlane init"* que criou dois ficheiros importantes numa pasta chamada *fastlane* dentro da pasta android: Appfile e Fastfile. O ficheiro Appfile apenas é constituído por duas configurações: o nome do *package* (packagename) da aplicação em questão, nome esse que está presente no ficheiro AndroidManifest.xml e uma variável *jsonkey* que é o caminho para um ficheiro em JSON que contém uma chave privada das credenciais da Google Play. Este ficheiro é necessário para fazer a comunicação entre a Google Play e a ferramenta do *fastlane*, e este ficheiro pode ser criado através do Google Cloud Platform. O ficheiro Fastfile é o ficheiro principal que é usado para configurar o *fastlane* que consiste num conjunto de *lanes* em que cada *lane* representa um conjunto de comandos que podem ser executados através do comando *"fastlane"* seguido do nome da *lane*. Por exemplo, como podemos ver pela Figura 38, foi criado uma lane que apenas continha um comando disponibilizado pelo *fastlane* com o nome *uploadtoplaystore* cujo objetivo era fazer o upload de um ficheiro .aab

para a Google Play que foram passados parâmetros como a versão da aplicação, o caminho para o *Android App Bundle* e o tipo de publicação que é pretendida no Google Play: *production* (a aplicação é disponibilizada para todos os utilizadores Android), *internal* (a aplicação é disponibilizada na Google Play Store Internal onde são adicionados um conjunto de emails e apenas os utilizadores com esse email irá ter acesso à aplicação), entre outras opções.

```
desc "Deploy (Upload to play store)"
lane :deploy_android do |options|
  upload_to_play_store(
    track: options[:production] ? 'production' : options[:internal] ? 'internal' : options[:alpha] ? 'alpha' : 'beta',
    version_code: flutter_version()["version_code"],
    aab: '../build/app/outputs/bundle/release/app-release.aab',
    skip_upload_images: true,
    skip_upload_screenshots: true,
  )
```

Figura 38: Estrutura de código de uma *lane* para dar *upload* da app na PlayStore

Com os processos bem definidos, era importante perceber de que forma estes poderiam ser automatizados para não ser necessário fazer estes fluxos manualmente sempre que fosse feita uma nova *release*, e é precisamente esse tema que será falado de seguida.

Um outro aspeto relevante a referir, é que aplicação é primeiramente disponibilizada num canal de testes interno para os utilizadores autorizados com *beta testers* e só depois, de testada, é que é disponibilizada no canal principal da Google Play Store.

3.6 AUTOMATIZAÇÃO DOS PROCESSOS

Um dos grandes desafios e requisito inicial proposto para desenvolvimento do projeto era a automatização do processo de *build* e *deploy* da aplicação. Como o projeto tratava-se do desenvolvimento de uma aplicação móvel, os processos de *build* e *deploy* são um pouco complexos, e iriam exigir imenso trabalho sempre que fosse necessário fazer uma nova *release*. A proposta inicial para tornar estes dois processos os mais automáticos possíveis foi de integrá-los com o processo de GitFlow (explicado anteriormente no Capítulo 2) em conjunto com as ferramentas disponibilizadas pelo GitLab para automaçõa de processos em eventos Git específicos. Foram idealizados vários cenários do processo de GitFlow onde seria implementado este tipo de automatização, e um dos casos mais importantes seria que quando ocorresse uma nova *release* no projeto iria despoletar automaticamente os dois processos de *build* e *deploy* sem qualquer ação do desenvolvedor.

3.6.1 *GitLab CI/CD*

Como referido anteriormente, foi proposto fazer uma integração dos processos de *build* e *deploy* com o GitFlow, e ao referir o GitFlow automaticamente é referido o GitLab que foi a ferramenta utilizada para fazer a gestão do repositório do projeto. Neste ponto era importante perceber de que forma poderia ser obtida este tipo de integração, e a solução foi através do GitLab CI/CD. Esta ferramenta é incorporada no GitLab para o desenvolvimento de software por meio das metodologias CI/CD - Continuous Integration/Continuous Delivery (Integração contínua e Entrega contínua). Segundo a documentação do GitLab, para usar o GitLab CI/CD é necessário a aplicação estar hospedada num repositório Git e um ficheiro com o nome .gitlab-ci.yml à raiz do projeto, com o especial cuidado do ficheiro ser obrigatório ter este nome. O ficheiro contém *scripts* que irão ser corridos em sequência ou em paralelo e algumas configurações, como a localização onde será dado o *deploy* da aplicação, se os *scripts* devem ser executados automaticamente ou manualmente, entre muitas outras configurações. A documentação de toda a configuração e sintaxe do ficheiro .gitlab-ci.yml encontra-se no endereço https://docs.gitlab.com/ee/ci/yaml/index.html. Os *scripts* são agrupados num conjunto de *jobs*, e estes fazem parte de uma única pipeline. Mal este ficheiro é adicionado ao repositório, é detetado pelo GitLab e uma aplicação com o nome de GitLab Runner corre os *scripts*.

O GitLab Runner é uma ferramenta que trabalha juntamente com o GitLab CI/CD para correr os *jobs* da pipeline. Esta ferramenta pode ser uma máquina virtual, uma máquina física, um *container* Docker, ou mesmo um Shell *script*, entre outros sistemas, e como a instância do GitLab comunica com o *Runner* através de uma API, é necessário que a máquina onde está localizado o Runner tenha uma ligação à Internet uma vez que esta que faz o *deploy* da *build* para produção. No caso do projeto, o *Runner* foi instalado numa máquina física com o sistema operativo Ubuntu 16.04. O *Runner* instalado tem a principal função de processar os *jobs* da pipeline que está definida num script presente dentro do projeto. De seguida irão ser explicados alguns conceitos que são bastante importantes para entender o GitLab CI/CD (dois dos quais já foram enunciados anteriormente, *"job"* e "pipeline"). Todo o motor do GitLab permite a monitorização dos *jobs* do *Runner* facilitando os processos de *debug* durante a execução dos automatismos de *build* e *deploy*;

As *pipelines* são o componente de nível superior da integração, entrega e implantação contínua. As *pipelines* são constituídas por *jobs*, que definem o que correr e pelos *stages* que definem quando estes *jobs* devem correr. Como fui enunciado anteriormente, as *pipelines*, na maioria das vezes são executadas automaticamente

sem nenhuma intervenção humana (e este é um dos propósitos de usar este tipo de metodologias como o CI/CD), mas muitas vezes é necessário interagir manualmente com a pipeline, diretamente na lista de *pipelines* do GitLab, para por exemplo reiniciá-la quando for necessário, geralmente quando há uma falha durante a sua execução.

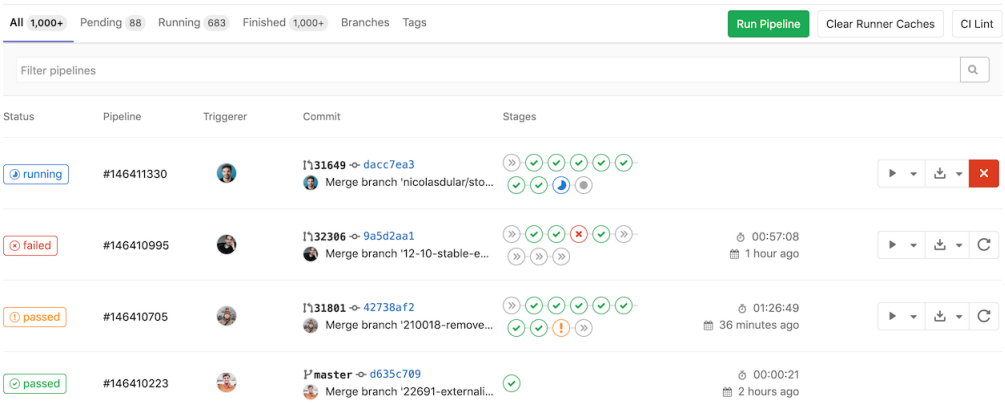


Figura 39: Exemplo de uma lista de *pipelines* do GitLab

A Figura 39 representa um exemplo de uma lista de *pipelines* da plataforma do GitLab e como é possível verificar, uma pipeline apresenta os seguintes dados: o status, um ID único, o *triggerer* que representa a pessoa que desencadeou a pipeline, o *commit* que desencadeou o pipeline, a lista de *stages* e um conjunto de ações como pausar, reiniciar, cancelar ou download de recursos (artifacts) resultantes da pipeline. A pipeline apresenta os seguintes diferentes status (estados): *created, waitingforresource, preparing, pending, running, success, failed, canceled, skipped, manual, scheduled.*

```
android:build:
  stage: build
  tags:
    - android
  before_script:
    - FACT_APP_BUILD_FOLDER=$(pwd)
    - cp .env.example .env
    - sed -i "s|^FACT_NODE_HOST=|FACT_NODE_HOST=${FACT_NODE_HOST}|g" .env
    - sed -i "s|^FACT_NODE_PORT=|FACT_NODE_PORT=${FACT_NODE_PORT}|g" .env
    - sed -i '/^json_key_file/d' ${FACT_APP_BUILD_FOLDER}/android/fastlane/Appfile
    - echo "json_key_file(\"${GOOGLE_PLAY_JSON_PATH}\")" >> ${FACT_APP_BUILD_FOLDER}/android/fastlane/Appfile
  rules:
    # Ignore merge request from release/* to develop
    - if: '$CI_PIPELINE_SOURCE == "merge_request_event"
      && $CI_MERGE_REQUEST_SOURCE_BRANCH_NAME =~ /^release\/.*/
      && $CI_MERGE_REQUEST_TARGET_BRANCH_NAME == "develop"'
      when: never
    # Merge request from anything (feature,fix...) to develop
    - if: '$CI_PIPELINE_SOURCE == "merge_request_event" && $CI_MERGE_REQUEST_TARGET_BRANCH_NAME == "develop"'
  script:
    # Flutter local configuration
    - touch ${FACT_APP_BUILD_FOLDER}/android/local.properties
    - echo flutter.sdk=$FLUTTER_PATH > ${FACT_APP_BUILD_FOLDER}/android/local.properties
    - echo sdk.dir=$ANDROID_SDK_PATH >> ${FACT_APP_BUILD_FOLDER}/android/local.properties
    - echo flutter.buildMode=release >> ${FACT_APP_BUILD_FOLDER}/android/local.properties

    # App Versioning
    - ${FACT_APP_BUILD_FOLDER}/deploy_tools/flutter_versioning.sh $CI_MERGE_REQUEST_TARGET_BRANCH_NAME ${FACT_APP_BUILD_FOLDER}/pubspec.yaml
```

Figura 40: Parte da definição de um *job* para fazer o *build* da aplicação (a)

```
  # # Build app
  - bundle exec fastlane build_android
  - ./gradlew --stop
artifacts:
  expire_in: 1 week
  paths:
    - build/app/outputs/apk/release/app-release.apk
    - build/app/outputs/bundle/release/app-release.aab
```

Figura 41: Parte da definição de um *job* para fazer o *build* da aplicação (b)

Como foi referido anteriormente, o conjunto de *jobs* resultam numa pipeline. Posto isto, estes são os elementos mais importantes do ficheiro *.gitlab-ci.yml*, com a principal função de definir o que deve ser executado durante a execução de uma pipeline. Os *jobs* devem conter pelo menos a cláusula *script* (esta cláusula especifica os comandos em linguagem de *script* Shell que devem ser executados pelo *Runner* na máquina onde está hospedado o projeto), podem ser definidos com um conjunto de restrições/regras que declaram em que ocasiões deve este ser executado e não há um limite de jobs na implementação de um ficheiro *.gitlab-ci.yml.*

Por exemplo, as Figuras 40 e 41 representam parte da definição de um *job* utilizado neste projeto cuja função é fazer o build do projeto resultante num ficheiro .apk. Como é possível verificar o *job* em questão contém as cláusulas: *stages, tags, beforescript, rules, script e artifacts*. Na documentação do GitLab existem uma enorme quantidade de diferentes cláusulas que podem ser utilizadas para a construção do ficheiro *.gitlab-ci.yml*, mas de seguida irão ser explicadas, de forma sucinta, as cláusulas que estão representadas na imagem. Antes de adicionar uma cláusula *stage* ao *job*, é necessário acrescentar a cláusula *stages* no início do ficheiro onde é referida a lista de todas as *stages* que irão ser utilizadas durante o *script*, e a ordem pela qual as *stages* estão definidas na lista é importante pois é por essa mesma ordem que os jobs irão ser executados. Os jobs que estão presentas na mesma *stage*, correm em paralelo e os *jobs* da próxima *stage* só é executada depois de todos *jobs* da *stage* anterior sejam completos com sucesso.

Resumidamente, uma *stage* é um conjunto de *jobs*, e a indicação da cláusula de mesmo nome no *job* (como é possível verificar pela imagem), indica em qual *stage* deve o *job* ser executada. Se for não referido nenhum *stage*, por *default* é escolhido o nome *test* para a *stage*. Na maioria das vezes é comum utilizar os seguintes nomes para *stages: build, test e deploy.*

A cláusula *tags* é usada quando é pretendido especificar um *Runner* que tem que estar obrigatoriamente na lista de todos os *Runners* disponíveis associados ao projeto, isto porque quando é registado um *Runner* é possível associar uma *tag*.

Na cláusula *beforescript* é possível definir um conjunto de comandos para serem executados antes do *script* principal do *job*.

A cláusula *rules* permite implementar um conjunto de regras para incluir ou excluir os *jobs* de uma pipeline. Esta cláusula aceita os seguintes tipos de regra: *if* (se a expressão for verdadeira, adiciona o *job* à pipeline), *changes* (adiciona o *job* se o ficheiro que se passa por parâmetro tenha sofrido alterações), *when* (que recebe vários tipos de input como o *onsuccess*: quando é pretendido que o *job* corra apenas quando os *jobs* anteriores tenham terminado com sucesso, ou o *never* para não correr o *job*).

Segundo a Figura 40 e pela segunda condição *if* o *job "androidbuild"* irá ser executado sempre que houver um *merge request* em que a *branch* destino é a *branch develop*. Mas há uma exceção presenta na primeira condição que nos diz quando a *branch* origem é uma *branch release* este *job* não deve ser executado. De seguida vem a cláusula *script* que é obrigatória e que representa que especifica a lista de comandos que devem ser executados pelo *Runner* durante o *job*. Por fim, a cláusula *artifacts* define uma lista de ficheiros ou diretórios que irão ser anexados ao *job* quando este termina com sucesso, falha ou com qualquer um dos estados.

No exemplo, foram adicionados dois ficheiros resultantes da *build* da aplicação (o ficheiro .apk e .aab) com a particularidade de colocar a subcláusula *expires-in* que nos diz quando é que estes ficheiros devem ser expirados e apagados. Como foi enunciado anteriormente, estes *artifacts* podem ser descarregados através da lista de *pipelines*.

De seguida irão ser enunciados os cenários do processo de GitFlow idealizados para fazer a automatização e que *stages* e *jobs* irão ser executados automaticamente mal esse processo é inicializado:

- Quando existe um *merge request* em que a *branch* destino é a *branch develop* são sempre executadas a *stage test* (responsável por testar todo o código, testes de funcionalidades com apoio de mock data, entre outros) e a *stage build* (responsável por fazer toda a configuração necessária de uma *build* em Android e gerar o ficheiro .apk e .aab utilizados para, respetivamente, instalar a aplicação no smartphone e fazer o *deploy* na loja de aplicações). Na grande maioria das vezes as *branchs* origem são a *branch feature* e a *branch release*;

- Quando existe um *commit* na *branch master* quer dizer que tudo já foi previamente testado e compilado então está na hora de ser executada a *stage deploy* que o seu trabalho é utilizar o ficheiro na *build* para colocar na loja de aplicações utilizando o comando *fastlane* para fazer este fluxo automaticamente.

Posto isto, foi conseguido um resultado extraordinário porque apenas é necessário fazer, manualmente, um *merge request* das *features* para o *develop* e depois do *develop* para o *master* e tudo o resto é feito automaticamente desde que o projeto é compilado até ao ponto de a aplicação ser disponibilizada na loja de aplicações. Para concluir, esta fase de automatizar os processos de *build* e *deploy* foi bastante exigente de concretizar a nível técnico pois foi apreendido muito conhecimento novo através de todos os conceitos que envolvem o GitLab CI/CD que é uma ferramenta extremamente completa, da implementação de todos os *scripts* exigidos para execução dos *jobs* e aqui foram conseguidos conhecimentos mais complexos a nível dos comandos de Shell *script* necessários para construção dos *scripts*. Mas posto tudo isto, foi das fases de todo o desenvolvimento do projeto que mais "gozo" deu de implementar devido ao facto de o resultado final ter ficado exatamente como foi planeado e assim ter poupado imenso tempo de cada vez que é feita uma nova *release* da aplicação móvel.

4

AVALIAÇÃO

No presente Capítulo será realizada uma parte fulcral de todo o trabalho desenvolvido que consiste na avaliação dos resultados obtidos. No subcapítulo 4.1 serão descritos os comentários elaborados pelos utilizadores na loja de aplicações e respetivas pontuações, e no subcapítulo 4.2 será descrito o desenvolvimento e resultados de dois tipos de testes relativos ao código da aplicação.

4.1 COMENTÁRIOS DOS UTILIZADORES

Como este projeto resultou numa aplicação de cariz pública (que podia ser instalada porque qualquer pessoa com acesso à Google Play Store, e mais propriamente, só poderia ser usufruída por uma pessoa que tivesse credenciais do Fact.pt), um dos aspetos mais importantes a reter são as opiniões dos utilizadores que utilizaram a aplicação. Pois um dos grandes objetivos de todas as empresas é sempre ter o intuito de agradar os clientes e com isto tentar atrair um maior número de utilizadores para a aplicação.

Como foi referido no subcapítulo 3.5.2, esta aplicação foi disponibilizada na loja de aplicações do Android denominada Google Play Store. Esta loja de aplicações contém um sistema de críticas e classificações que qualquer utilizador, com as credenciais do Google, tem a possibilidade de avaliar a aplicação e fazer uma apreciação pública que estará disponível para todos os utilizadores visualizarem. No processo de classificar a aplicação, o utilizador pode preencher uma classificação que é representada por cinco estrelas onde deve preencher no mínimo uma estrela e no máximo cinco estrelas. Sempre que for iniciada uma nova crítica por parte de um utilizador, o preenchimento da classificação é obrigatória. Além da classificação, é possível ser escrito um texto que normalmente representa a descrição da experiência que o utilizador teve com a aplicação.

Desde o lançamento da primeira versão da aplicação, que foi feito no dia 4 de fevereiro de 2021, até à data de 30 de novembro de 2021 a aplicação obteve oito classificações de utilizadores diferentes em que três deles decidiram descrever a sua

experiência, e as restantes 5 pessoas optaram por apenas preencher a classificação. De realçar, que todas as classificações foram avaliadas com cinco estrelas resultando assim, numa média de cinco na classificação geral da aplicação. A Figura 42 apresenta os comentários escritos e respetivas avaliações dos três utilizadores.

A página de aplicação pode ser visualizada através do Apêndice B.

Figura 42: Comentários realizados pelos utilizadores na Google Play Store

Para concluir, é relevante referir que até à data de 30 de novembro de 2021, 133 utilizadores diferentes já descarregaram a aplicação da loja, e que neste momento estão 103 utilizadores ativos.

É com enorme orgulho que estes resultados são apresentados, pois representa que os utilizadores ficaram satisfeitos com o resultado da aplicação.

4.2 TESTES

Um outro aspeto importante referente à avaliação dos resultados, era o desenvolvimento de testes para avaliar o código escrito ou as interfaces da aplicação. Os testes são a melhor forma de garantir o funcionamento da aplicação enquanto são adicionadas novas funcionalidades ou alteradas funcionalidades existentes, ou seja, é poupado o trabalho de verificar que todas as funcionalidades estão a funcionar corretamente depois de uma nova funcionalidade ser adicionada. No decorrer do projeto, foram utilizados dois tipos de testes: testes unitários e testes de *widgets* que

serão descritos individualmente de seguida, onde ambos necessitavam do *package* *flutter_test*.

4.2.1 *Testes unitários*

Os testes unitários são utéis para verificar o comportamento de uma determinada função, método ou classe. No projeto da aplicação, foram implantadas muitas funções *Helpers* (são funções que são chamadas por todo o projeto e, então, o código pode ser reutilizado para apenas ser definido numa única função), e era importante testar cada uma dessas funções. Na Figura 43 mostra um exemplo de um teste unitário feito uma função (bastante utilizada nos gráficos por exemplo) que convertia o número do mês para o mês escrito por extenso, ou seja, quando a função recebesse o número 1, era esperado que retornasse a palavra "Janeiro".

```
Future<Null> convertNumberToMonthName(WidgetTester tester) async {
  String monthName = Helpers.convertNumberToMonthName(1);

  expect(monthName, 'Janeiro');
}
```

Figura 43: Exemplo de um teste unitário a uma função *Helper*

4.2.2 *Testes de widgets*

Os testes de *widgets*, como o nome indica, são utilizados para testar os *widgets* na interface onde fornece funcionalidades para interagir com os *widgets* durante os testes. Na Figura 44 é apresentado um exemplo de um teste situado na página de Login da aplicação onde é preenchido os campos de email e *password* com "dados teste"armazenados nas variáveis de ambiente (no ficheiro *.env*), e depois de clicar no botão Login é esperar que apareça o texto *"Set a PIN code"* que introduz a página do preenchimento do código PIN.

```dart
Future<Null> login(WidgetTester tester) async {
  await tester.pumpAndSettle();

  final emailTextField = find.byKey(Key("emailTxtFormField"));
  await tester.enterText(emailTextField, env['TEST_EMAIL']);

  final passwordTextField = find.byKey(Key("passwordTxtFormField"));
  await tester.enterText(passwordTextField, env['TEST_PASSWORD']);

  final loginButton = find.byKey(Key("loginBtn"));
  await tester.tap(loginButton);
  await tester.pump();

  expect(find.text("Set a PIN code"), findsOneWidget);
```

Figura 44: Exemplo de um teste de *widget* na página de Login da aplicação

Como foi referido no subcapítulo X, estes testes são automaticamente chamados sempre que é realiza uma nova *feature* ou preparada uma nova *release*.

5

CONCLUSÕES E TRABALHO FUTURO

No subcapítulo 5.1 serão apresentas as principais conclusões do projeto desenvolvido, concluir se o objetivo anteriormente definido foi totalmente cumprido como planeado e no subcapítulo 5.2 serão apresentados alguns aspetos a melhorar e o que poderá ser implementado para o futuro.

5.1 CONCLUSÕES

Como conclusões finais, é importante concluir que o objetivo proposto no início do projeto foi totalmente concluído juntamente com as funcionalidades planeadas. Todo o desenvolvimento foi implementado com particular cuidado nas boas práticas de desenvolvimento, especificação, arquitetura e programação. Percebe-se o sentido de todo o esforço tido em todas as fases de implementação quando, já com a aplicação em produção, sempre que surge uma nova funcionalidade ou correção dá origem a desenvolvimento organizado e isolado, com uma boa articulação de trabalho em equipa, promovendo os melhores padrões de controlo de qualidade pelas fases de revisão e *deployment* para *beta testers*, culminado com a agilidade e rapidez com que à custa de todas as automações se faz chega qualquer atualização ao utilizador final. Existe de facto um esforço acrescido no início para fazer as coisas bem, mas rapidamente se percebe como todo este esforço é capitalizado e diluído em todas as funcionalidades, correções e desenvolvimentos posteriores.

De salientar que também se percebe, estas boas práticas são transversais à uma grande parte dos projetos de Engeharia de software e programação, pelo que este percurso foi uma grande mais valia para a minha qualificação de competências profissionais.

Mesmo sendo um primeiro contacto profissional, a verdade é que este projeto foi concretizado entre os colaboradores da empresa com máxima disponibilidade para me orientar, assistir e estudar em conjunto todos os desafios que me foram apresentados sempre com o máximo respeito profissional e bom ambiente de trabalho no geral.

5.2 TRABALHO FUTURO

Dado o potencial de novas funcionalidades que a empresa tem por objetivo implementar, ficou assegurado uma boa relação profissional com a empresa com vista à implementação de novas funcionalidades do projeto.

6

BIBLIOGRAFIA

[1] https://www.fact.pt/, acedido a 2021/11/30

[2] https://flutter.dev, acedido a 2021/11/30

[3] https://dart.dev, acedido a 2021/11/30

[4] https://www.youtube.com/watch?v=ok9zKc8S5Jg, acedido a 2021/03/20

[5] https://www.netsolutions.com/insights/cross-platform-app-frameworks-in-2019/, acedido a 2021/03/20

[6] https://appinventiv.com/blog/cross-platform-app-frameworks/, acedido a 2021/03/21

[7] https://imaginedone.com.br/blog/web-e-mobile/aplicativo-nativo-hibrido-pwa/, acedido a 2021/03/21

[8] https://dzone.com/articles/native-vs-hybrid-vs-cross-platform-how-and-what-to, acedido a 2021/03/21

[9] https://www.konstantinfo.com/blog/react-native-vs-flutter-vs-ionic-vs-nativescript-vs-pwa/, acedido a 2021/03/21

[10] https://medium.com/@ecavalcanti/flutter-uma-breve-introdução-d9071fcb8474, acedido a 2021/04/21

[11] https://javascript.plainenglish.io/flutter-vs-react-native-which-is-the-best-choice-for-2021-e695e79c6707, acedido a 2021/06/21

[12] https://www.sitepoint.com/premium/books/beginning-flutter/read/1/, acedido a 2021/06/12

[13] https://www.flutterparainiciantes.com.br/basico/ciclo-de-vida, acedido a 2021/06/12

[14] https://www.section.io/engineering-education/flutter-folder-organization/, acedido a 2021/06/12

[15] https://medium.com/flutter-community/flutter-scalable-folder-files-structure-8f860faafebd, acedido a 2021/08/21

[16] https://firebase.google.com/docs/cloud-messaging, acedido a 2021/11/27

[17] https://source.android.com/setup/build/building, acedido a 2021/11/27

[18] https://developer.android.com/studio/publish, acedido a 2021/11/027

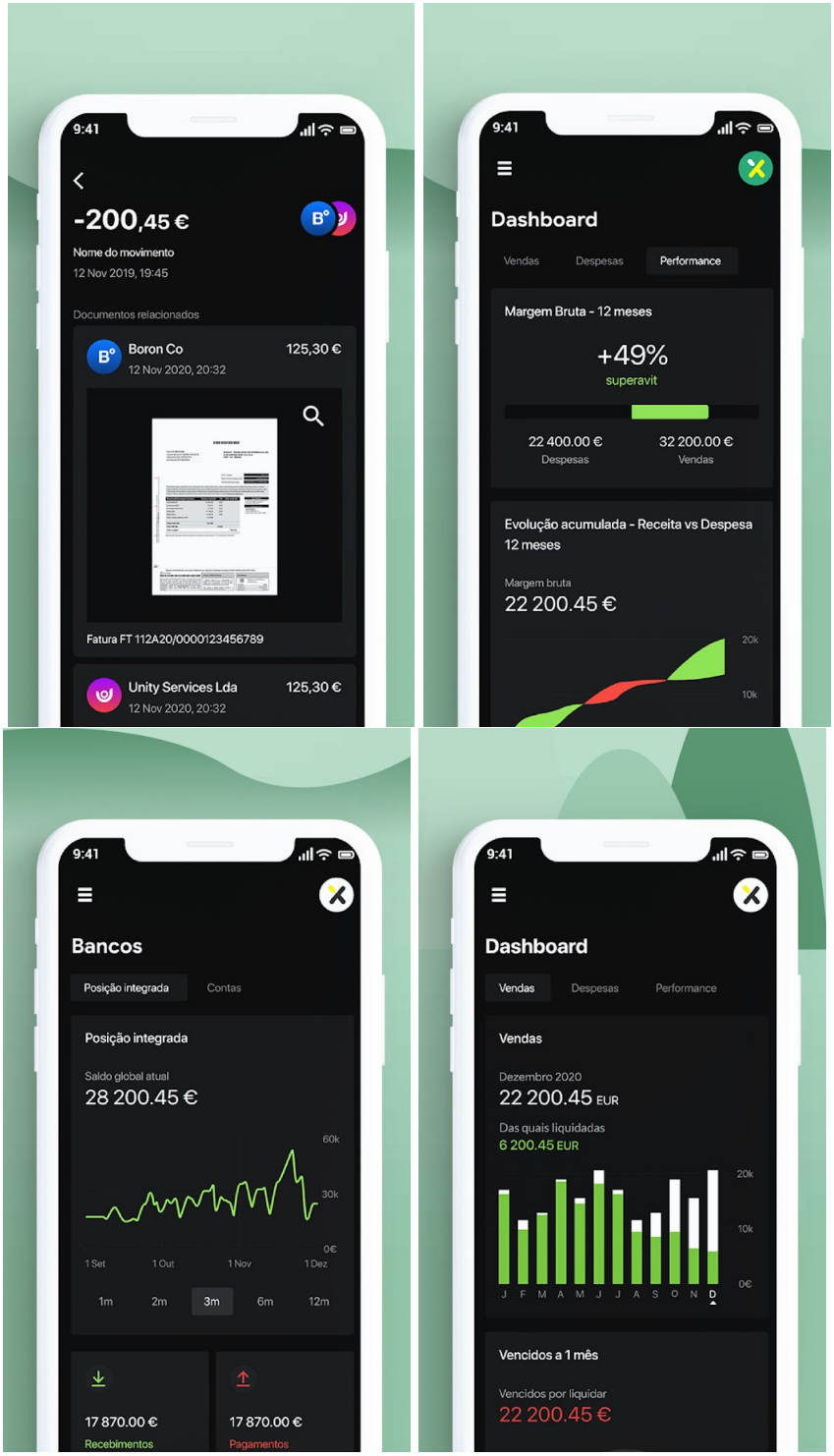
[19] https://developer.android.com/studio/publish/app-signing, acedido a 2021/11/27

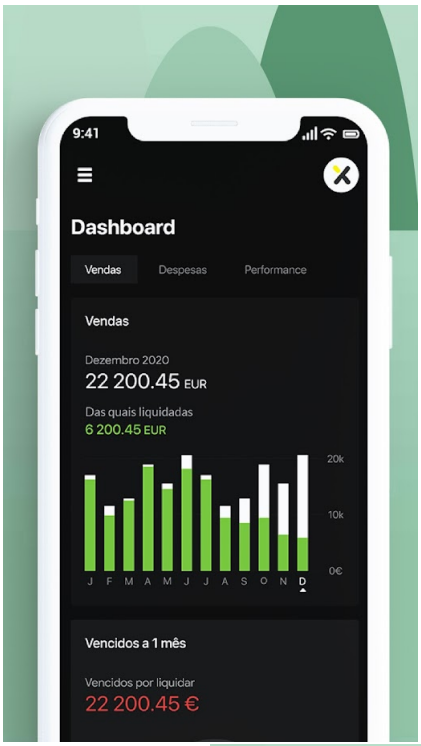
[20] https://play.google.com/store/apps/details?id=pt.fact.fact_app, acedido a 2021/11/30

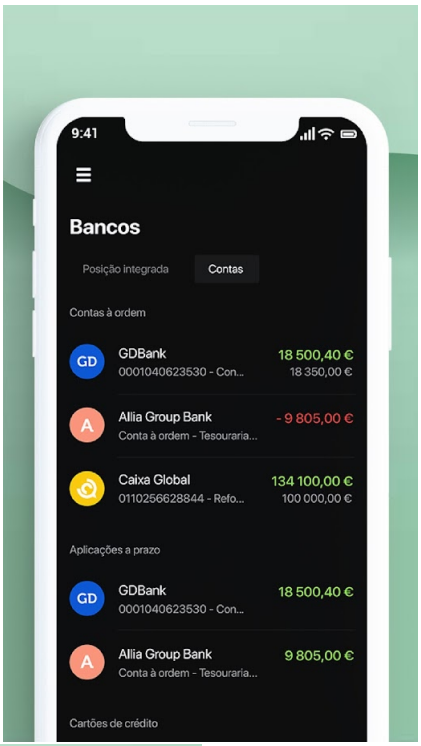
APÊNDICES

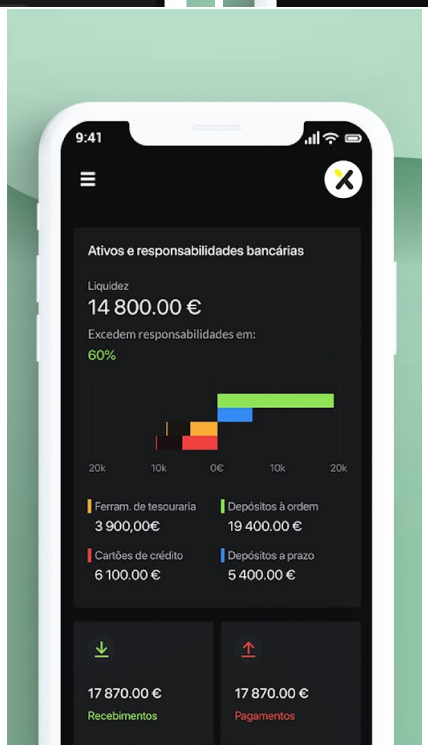
A

SCREENSHOTS DA APLICAÇÃO



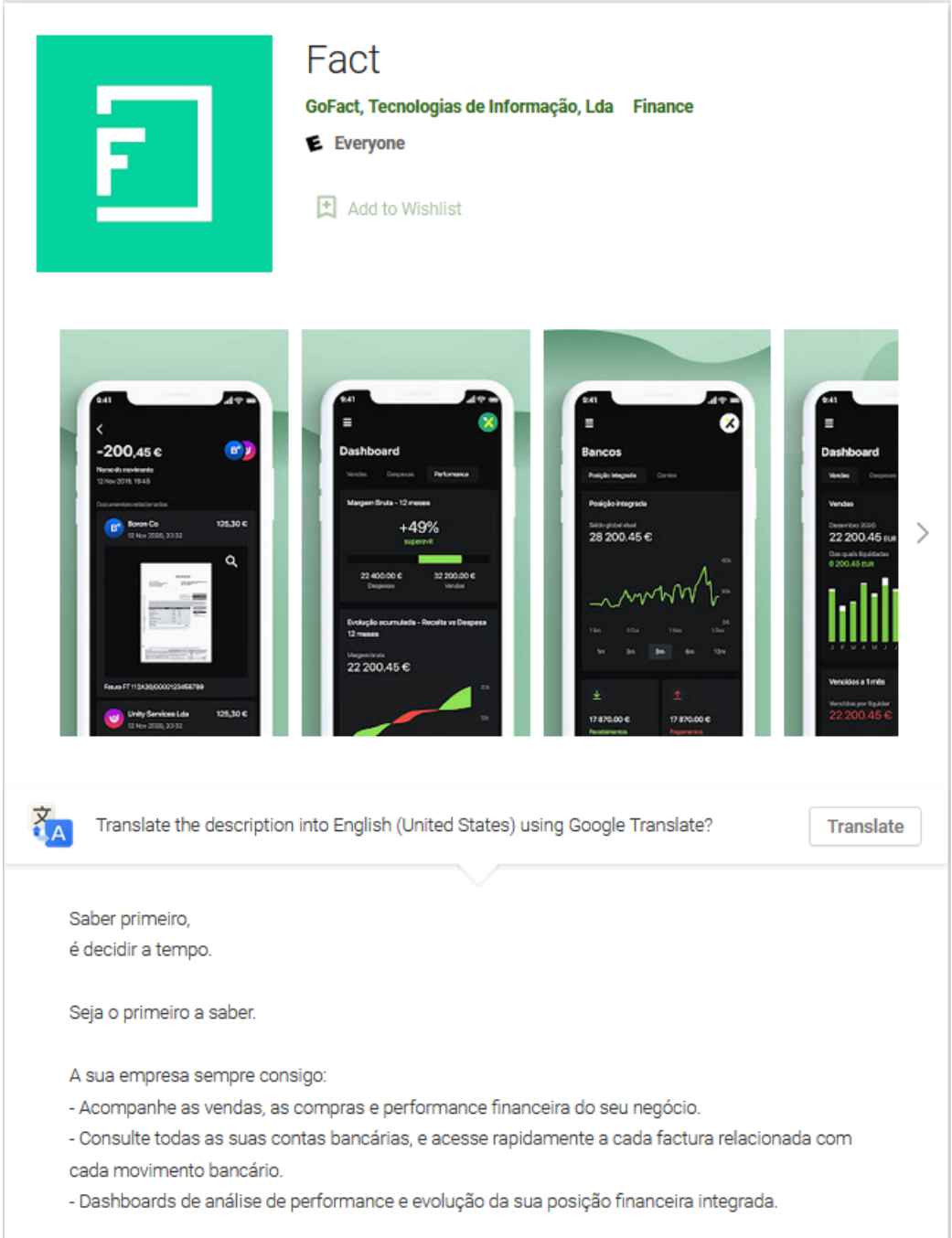

9:41
Dashboard
Vendas
Despesas
Performance
Vendas
Dezembro 2020
22 200.45 EUR
Das quais liquidadas
6 200.45 EUR
20k
10k
0€
J F M A M J J A S O N D
Vencidos a 1 mês
Vencidos por liquidar
22 200.45 €


9:41
Bancos
Posição integrada
Contas
Contas à ordem
GD
GDBank
0001040623530 - Con...
18 500,40 €
18 350,00 €
A
Allia Group Bank
Conta à ordem - Tesouraria...
- 9 805,00 €
Caixa Global
0110256628844 - Refo...
134 100,00 €
100 000,00 €
Aplicações a prazo
GD
GDBank
0001040623530 - Con...
18 500,40 €
A
Allia Group Bank
Conta à ordem - Tesouraria...
9 805,00 €
Cartões de crédito


9:41
Ativos e responsabilidades bancárias
Liquidez
14 800.00 €
Excedem responsabilidades em:
60%
20k
10k
0€
10k
20k
Ferram. de tesouraria
3 900,00€
Depósitos à ordem
19 400.00 €
Cartões de crédito
6 100.00 €
Depósitos a prazo
5 400.00 €
17 870.00 €
Recebimentos
17 870.00 €
Pagamentos

B

PÁGINA DA APLICAÇÃO NA GOOGLE PLAY STORE



D E C L A R A Ç Ã O

Declaro, sob compromisso de honra, que o trabalho apresentado neste projeto, com o título *"Aplicação móvel empresarial - Portal de gestão financeira FACT.PT"*, é original e foi realizado por Estudante Luís Manuel Martins Campos (2192735) sob orientação de Professor Filipe dos Santos Neves (fneves@ipleiria.pt).

Leiria, novembro de 2021

Assinado por : **Luís Manuel Martins Campos**
Num. de Identificação: 13507304
Data: 2021.11.30 23:48:18+00'00'

CHAVE MÓVEL

Estudante Luís Manuel Martins Campos