Fundação Educacional do Município de Assis
Instituto Municipal de Ensino Superior de Assis
Campus "José Santilli Sobrinho"

LEONARDO RODRIGUES DANTAS

UMA ABORDAGEM AO DESENVOLVIMENTO MÓVEL COM FLUTTER E DART

Assis/SP
2020

Fundação Educacional do Município de Assis
Instituto Municipal de Ensino Superior de Assis
Campus "José Santilli Sobrinho"

LEONARDO RODRIGUES DANTAS

UMA ABORDAGEM AO DESENVOLVIMENTO MÓVEL COM FLUTTER E DART

Trabalho de Conclusão de Curso apresentado ao Instituto Municipal de Ensino Superior de Assis, como requisito do Curso de Bacharelado em Ciência da Computação do Instituto Municipal de Ensino Superior de Assis – IMESA e a Fundação Educacional do Município de Assis – FEMA. .

Orientando(a): Leonardo Rodrigues Dantas
Orientador(a): Dr. Almir Rogério Camolesi

Assis/SP
2020

FICHA CATALOGRÁFICA

D192u DANTAS, Leonardo Rodrigues.

Uma abordagem ao desenvolvimento móvel com flutter e dart/ Leonardo Rodrigues Dantas. – Assis, 2020.

74p.

Trabalho de conclusão de curso (Ciência da Computação) - Fundação Educacional do Município de Assis - Fema

Orientador: Dr. Almir Rogério Camolesi

1.Programação móvel. 2.Flutter. 3. Dart

CDD: 005.131
Biblioteca da FEMA

UMA ABORDAGEM AO DESENVOLVIMENTO MÓVEL COM FLUTTER E DART

LEONARDO RODRIGUES DANTAS

Trabalho de Conclusão de Curso apresentado ao Instituto Municipal de Ensino Superior de Assis, como requisito do Curso de Bacharelado em Ciência da Computação do Instituto Municipal de Ensino Superior de Assis – IMESA e a Fundação Educacional do Município de Assis – FEMA.

Orientador: _____
Dr. Almir Rogério Camolesi

Examinador: _____
Dr. Luiz Ricardo Begosso

Assis/SP
2020

DEDICATÓRIA

Dedico este trabalho primeiramente a Deus, por sua infinita graça e misericórdia derramada sobre minha vida a cada dia.

A Maisa, minha namorada, noiva e melhor amiga, por sempre ter estado do meu lado, nunca me deixando desanimar e tirando forças da onde não existiam.

A minha família, pelo apoio incondicional, minha mãe pelas suas orações e sabedoria, meu pai por ser o homem que me inspira e exemplo de batalhador, minha irmã por sempre me encorajar e me fazer saber que sempre existe algo melhor por vir.

Aos meus amigos de sala, que tornaram esses quatro anos mais fáceis.

Ao meu professor e orientador, por me incentivar, desde o início do projeto e por ter acreditado em minha capacidade logo nos primeiros anos.

“Um navio é seguro no porto. Mas não é para
isso que navios foram construídos”

Grace Murray Hoper (1906-1992)

RESUMO

O mercado de aplicativos móveis atrai a atenção de milhares de desenvolvedores, entretanto em um mundo onde tempo é dinheiro, existem muitos obstáculos para que exista a afirmação destes profissionais nesta área, como o fato de cada plataforma possuir a sua própria linguagem. Muitas alternativas já surgiram, como o desenvolvimento híbrido com *Ionic* e a geração de códigos nativos com o *React Native*, entretanto essas soluções sempre trazem consigo algum obstáculo, dependendo para qual finalidade será utilizada, como por exemplo, desempenho, *designer*, produtividade baixa entre outros. Com isto surge o *Flutter*, um *framework* da *Google* que funciona com a linguagem *Dart*, onde com apenas um código conseguimos gerar aplicativos nativos para qualquer plataforma móvel além de *web* e *desktop*. O *Flutter* chegou para ser produtivo, solucionar o problema do desempenho e trabalhar na plataforma como se fosse o próprio código nativo sendo ali compilado. Está pesquisa foi elaborado para explorar e apresentar tudo aquilo que ele tem a oferecer, para isso será desenvolvido um aplicativo que ilustra suas principais características justificando o seu crescimento exponencial no meios de desenvolvedores de aplicações móveis.

Palavras-chave: Programação Nativa; Programação Híbrida; Android; IOS; Mobile.

ABSTRACT

The mobile application market attracts the attention of thousands of developers, but in the world where time is money, there are many obstacles to having a declaration of professional performance in this area, such as the fact that each platform uses its own language. Many alternatives have already emerged, such as hybrid development with Ionic and a generation of native codes with React Native, however, these solutions always bring some obstacle, the application for which it will be used, for example, performance, designer, low priority among others . With this, Flutter, a Google structure that works with the Dart language, appears, where only one code generates native applications for any mobile platform besides the web and, in the future, on the desktop. Flutter arrived to be productive, solve performance problems and work on the platform, as if it were the native code itself being compiled. The research was designed to explore and present everything it has to offer, for that purpose an application will be developed that shows the main characteristics just to justify its exponential growth among mobile application developers.

Keywords: Native Programming; Hybrid Programming; Android; IOS; Mobile.

LISTA DE ILUSTRAÇÕES

Figura 1: Mercado de dispositivos móveis 18

Figura 2: Camadas entre software nativo e hardware 21

Figura 3: Camadas entre Ionic e hardware 28

Figura 4: Camadas entre React Native e hardware 30

Figura 5: Camadas entre Flutter e hardware 31

Figura 6: Arquitetura Flutter 34

Figura 7: Exemplo de variáveis em Dart 36

Figura 8: Exemplos de tipos de lista 36

Figura 9: Exemplos de tipos de Map 37

Figura 10: Funções retornos e concatenações 37

Figura 11: Exemplo de switch 38

Figura 12: Estruturas de repetição 38

Figura 13: Classe com herança 39

Figura 14: Exemplo de stateless e statefull 40

Figura 15: Exemplo de classe principal em Flutter 41

Figura 16: Exemplo de scaffold 41

Figura 17: Resultado da interface principal e scaffold 42

Figura 18: Classe responsável pelo popmenu 43

Figura 19: Visualização do popmenu 43

Figura 20: Classe responsável pelo menu lateral 44

Figura 21: Classe responsável pelo desenho de cada item do menu lateral 44

Figura 22: Visualização do menu lateral 45

Figura 23: Classe responsável pelo formulario 45
Figura 24: Classe responsável pelo desenho de cada input 46
Figura 25: Visualização do formulário sem informações e com informações 47
Figura 26: Comparação entre Flutter e React Native globalmente no Google Trends 49
Figura 27: Comparação entre Flutter e React Native no Brasil com Google Trends 49
Figura 28: Diagrama de classe da aplicação 51
Figura 29: Caso de uso da aplicação 52
Figura 30: Interfaces de login e cadastro 53
Figura 31: Interface inicial 54
Figura 32: Alerta no início de compra 54
Figura 33: Interfaces para cadastro de dados da compra 55
Figura 34: Alerta de informações incompletas 56
Figura 35: Interface de confirmação de dados e de compra bem sucedida 56
Figura 36: Interface inicial com o ponto exato da compra 57
Figura 37: Alerta de não existir cartões ativos 58
Figura 38: Interfaces de histórico de usuário 59
Figura 39: Interfaces de configuração de usuário 59
Figura 40: interfaces de configuração do mapa 60
Figura 41: interfaces de configuração de linguagem 61
Figura 42: interfaces para exclusão de conta 62
Figura 43: Exemplo de classe com Mobx 63
Figura 44: Exemplo de utilização do Mobx 64
Figura 45: Comando para limpar qualquer classe Mobx já existente 64
Figura 46: Comando para gerar novas classes Mobx 64
Figura 47: Exemplo de código gerado pelo Mobx 65

Figura 48: Exemplo de utilização do GetIt 65

Figura 49: Exemplo de instanciação do GetIt 66

Figura 50: Classe com Mobx 66

Figura 51: Exemplo de serviço 67

Figura 52: Classe gerada com Mobx 67

Figura 53: Exemplo de controller 68

Figura 54: Recuperação de instância com GetIt 68

Figura 55: Recuperação do campo email 68

LISTA DE ABREVIATURAS E SIGLAS

HTML	Hypertext Markup Language
CSS	Cascading Style Sheets
SDK	Software Development Kit
UX	User Experience
UI	User Interface
IDE	Integrated Development Environment
API	Application Programming Interface
MVC	Model, View, Controller
WEB	World Wide Web
CLI	Command-Line Interface
XML	Extensible Markup Language
USB	Universal Serial Bus
WI-FI	Wireless Fidelity
APP	Aplicativo
ARM	Advanced RISC Machine

SUMÁRIO

1. INTRODUÇÃO 14
1.1. CONTEXTUALIZAÇÃO 14
1.2. MOTIVAÇÃO 15
1.3. JUSTIFICATIVA 16
1.4. OBJETIVOS 16
1.5. PERSPECTIVA E CONTRIBUIÇÃO 17
1.6. ESTRUTURA DO TRABALHO 17
2. REVISÃO 18
2.1. DISPOSITIVOS MÓVEIS 19
2.2. PROGRAMAÇÃO NATIVA 19
2.1.1. ANDROID 21
2.1.1.1. ANDROID STUDIO 22
2.2.2. IOS 23
2.2.2.1. XCODE 24
2.2. PROGRAMAÇÃO HÍBRIDA 25
2.3.1. PHONEGAP E CORDOVA 26
2.3.2. IONIC 27
2.3.3. REACT NATIVE 28
3. FLUTTER 31
3.1. INTRODUÇÃO 31
3.2. INSTALAÇÃO 33
3.3. ARQUITETURA 33
3.4. DART 34
3.4.1. SINTAXE 35
3.5. WIDGETS 39
3.5.1. EXEMPLOS DE WIDGETS 40
3.5.2. CICLO DE VIDA DE WIDGETS 47
3.6. DART PACKAGES 47
3.7. GERÊNCIA DE ESTADOS 48

3.8. FLUTTER VERSUS REACT NATIVE 48
4. ESTUDO DE CASO 51
4.1. DESENVOLVIMENTO 52
4.2. PRINCIPAIS PACKAGES 62
4.3. VIEWS, CONTROLLERS, SERVICES E MODELS 66
5. CONCLUSÃO 69
REFERÊNCIAS 70

1. INTRODUÇÃO

O mercado de dispositivos móveis está ganhando cada vez mais espaço e atualmente encontra-se em seu auge, segundo dados do *Serasa Experian* somente em 2017 houve movimentação de 82 bilhões de dólares mundialmente, sendo que o Brasil ocupava neste ano o 4° lugar referente ao mercado de aplicativos mundialmente. Muitas alternativas surgem como possibilidade ao desenvolvedor para adentrar neste cenário, todavia o desenvolvimento móvel sempre apresentou dificuldades para aqueles que buscam se deslocar para está área, um dos principais tem sido a questão do *Android* e do *Iphone* possuírem ambientes de desenvolvimento e linguagens diferentes como o *Java* ou *Kotlin* para *Android* e o *Swift* para o *Iphone*.

1.1. CONTEXTUALIZAÇÃO

Com o mercado de dispositivos móveis tão aquecido, desenvolver um aplicativo abrangente para plataformas do mercado transforma-se em uma tarefa que demanda tempo e conhecimento por partes dos realizadores do projeto (Bassoto, 2014). Sendo assim tecnologias alternativas para tal tarefa são vistas com bons olhos, pois cada plataforma possui um ambiente de desenvolvimento próprio, demandando tempo e conhecimentos específicos. Desenvolver um único código que pode ser executado em múltiplas plataformas é vantajoso, porém deve se atentar para o fato de que ao fazer isso não conseguimos explorar o máximo de cada dispositivo, e nem acessar todas as suas funcionalidades naturalmente, além de problemas relacionados ao desempenho, tendo que lidar muitas vezes com a dificuldade de estabelecer um padrão de experiência agradável do usuário (Bassoto, 2014). Portanto diversas alternativas são criadas ou otimizadas para poder lidar com todas estas divergências e conseguir realizar a entrega de um bom produto.

O desenvolvimento híbrido apresenta-se como uma alternativa, onde a maior parte de um aplicativo é desenvolvido utilizando tecnologias como *Html5*, *CSS* e *Javascript* (MENDES;

GARBAZZA; TERRA, 2014). É uma alternativa interessante ao desenvolvimento nativo, porém quando avaliamos o desempenho de aplicativos híbridos e nativos, a diferença pode preocupar a experiência do usuário, e como hoje tudo deve ser focado no usuário e em sua experiência, o desenvolvimento híbrido embora possua um único código para ambas as plataformas tendo alta produtividade, perde muito quando o foco é o desempenho.

Já a abordagem nativa pode ser considerada cara, pois exige mais de uma plataforma e uma mão de obra mais específica, sem contar que tanto o tempo de produção quanto o custo de desenvolvimento podem aumentar dependendo da quantidade de plataformas que o aplicativo pode alcançar (MENDES; GARBAZZA; TERRA, 2014), ainda assim é importante entender que o nativo e o híbrido não se apresentam de forma concorrente, mas deve-se avaliar o momento certo de utilizá-los, e para isso muitos fatores devem ser levados em consideração (MATOS; SILVA, 2016).

Nesse cenário surge o *Flutter*, um *framework* do *Google*, com a proposta da produção de um único código, que tem como objetivo permitir o desenvolvimento de aplicativos de alta performance e com experiência nativa para os usuários (CORAZA, 2018). O Flutter tem diversos pontos a seu favor, tudo nele é feito para ser rápido, o designer é construído para ser bonito, atraindo programadores iniciantes, experientes e até designers, sendo um framework focado na construção de interfaces e criando aplicativos com excelente desempenho (CORAZA, 2018). De início o Flutter pode lembrar o React Native pois os dois possuem propostas semelhantes, entretanto eles se divergem na maneira que resolvem seus problemas, pois o *React* utiliza o conceito de *bridge* enquanto o *Flutter* utiliza determinados processos de compilação (MUNIZ, 2019).

1.2. MOTIVAÇÃO

Apresenta-se como motivação para este trabalho o estudo de uma nova alternativa para o desenvolvimento móvel, pois a necessidade de atender a maioria dos dispositivos

tornou-se inviável desenvolver aplicativos nativos, devido ao seu custo e manutenção (MENDES; GARBAZZA; TERRA, 2014).

1.3. JUSTIFICATIVA

A importância de trazer um tema como este para uma discussão é muito relevante, pois o mercado está em constante mudança assim como as tecnologias que são utilizadas, os usuários se tornaram mais exigentes, tanto em quesito de usabilidade quanto de desempenho, portanto é totalmente considerável apresentar uma alternativa que pode facilitar a vida de desenvolvedores e programadores. Existe também uma necessidade muito grande da comunidade de se manter atualizada e sempre estar disposta a conhecer novas tecnologias. Vale a pena ressaltar que o mercado móvel sempre esteve muito monopolizado, enfrentando consideráveis divergências nas linguagens de programação e nos ambientes de desenvolvimento de cada plataforma (CRUZ; PRETUCELLI, 2016), e mesmo as aplicações híbridas deixam brechas para outros problemas, como desempenho e a falta de definição no padrão de desenvolvimento.

1.4. OBJETIVOS

O objetivo deste trabalho é fazer um estudo sobre as principais alternativas para o desenvolvimento móvel, como o *Cordova*, *Ionic*, *React Native* e as plataformas nativas, em seguida será feito um estudo mais profundo sobre o Flutter de modo que possamos apresentar suas características e principais funcionalidades, além do desenvolvimento de um aplicativo.

1.5. PERSPECTIVA E CONTRIBUIÇÃO

A pesquisa deverá contribuir para a comunidade apresentado uma alternativa mais robusta para o desenvolvimento móvel, demonstrando o seu funcionamento, e quais são as suas principais características. Apesar do framework ainda ser uma novidade, muitas empresas no mundo já o utilizam, como a *Nubank*, *Alibaba* e *Groupon*.

1.6. ESTRUTURA DO TRABALHO

O trabalho será dividido da seguinte maneira

• **Capítulo 1 – Introdução**, neste tópico foi feito a contextualização sobre o tema, apresentando os motivos que levaram a esta pesquisa, justificativas e objetivos.

• **Capítulo 2 – Revisão da Bibliográfica**, será apresentado conceitos relacionados ao desenvolvimento móvel, como mercado, programação nativa e programação híbrida, bem como suas principais ferramentas, sendo uma revisão do estado da arte.

• **Capítulo 3 – Flutter**, neste capítulo será descrito o principal tema deste trabalho, o *Flutter*. Será apresentado profundamente seu funcionamento, sua história, arquitetura, vantagens e desvantagens.

• **Capítulo 4 – Estudo de Caso**, desenvolvimento de um aplicativo fazendo uso do Flutter e de seus principais artefatos, demonstração da utilização de *packages*, utilização de camadas e conexão com o banco.

• **Capítulo 5 – Conclusão**, apresentação das conclusões alcançadas durante a elaboração do trabalho.

• **REFERÊNCIAS**

2. REVISÃO BIBLIOGRÁFICA

Neste capítulo foi realizado um estudo sobre os diferentes modelos e formas utilizadas para o desenvolvimento móvel, explorando conceitos relacionados a programação nativa e híbrida, principais plataformas de desenvolvimento e linguagens, bem como suas semelhanças e diferenças.

A pesquisa foi focada em sistemas operacionais *Android* e *IOS*, pois eles possuem o domínio de grande parte do mercado como mostra o gráfico a seguir retirado da página *Statcounter*. Segundo estes dados entre novembro de 2018 e novembro de 2019, 75% do mercado era dominado pelo sistema *Android* e 22% por *IOS*, já totalizando aproximadamente mais de 92% de todo o mercado de dispositivos móveis. Na figura 1 temos dados referentes ao mercado móvel.

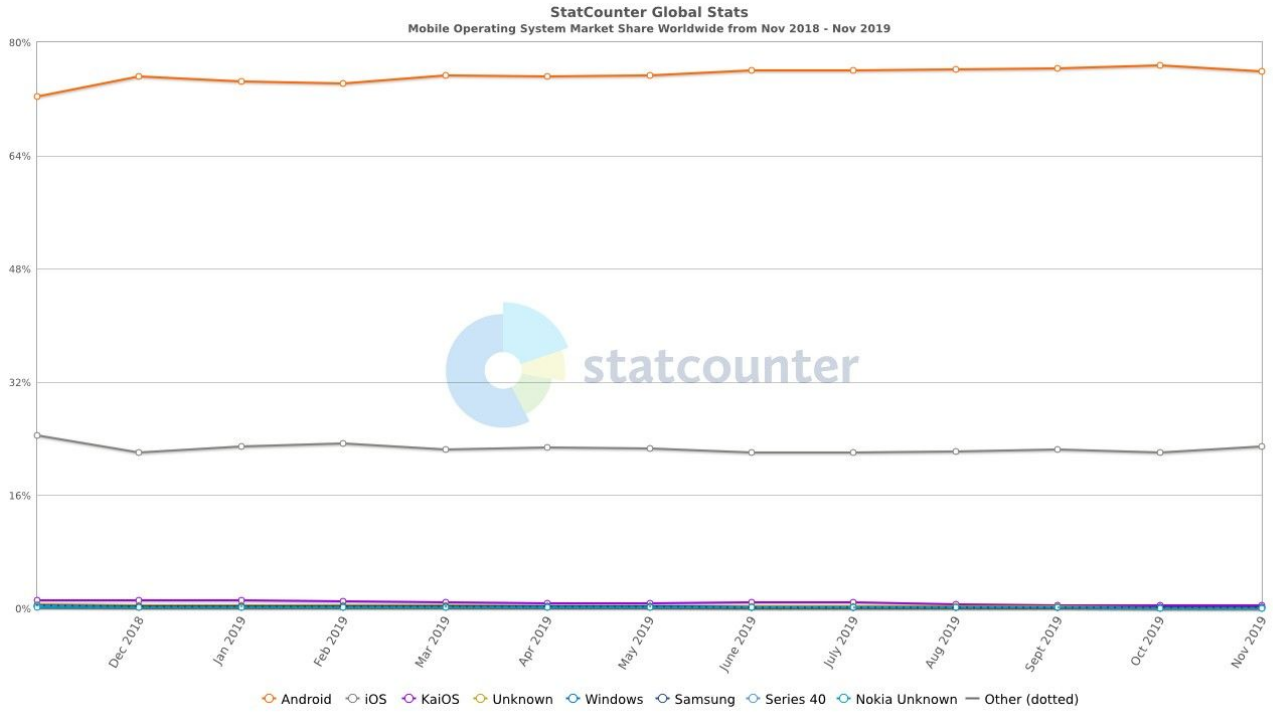


Figura 1: Mercado de dispositivos móveis
Fonte: https://gs.statcounter.com/os-market-share/mobile/worldwide

2.1. DISPOSITIVOS MÓVEIS

Os smartphones estão cada vez mais fazendo parte da rotina dos brasileiros, isto se deve a existência de muitos aplicativos que possuem o propósito de facilitar as suas vidas, como o *Ifood*, *Uber* e *Netflix*, além de aplicativos de pagamento bancário e controle de cartões de créditos, que fazem com que os consumidores evitem filas e tenham controle de tudo na palma de sua mão. De acordo com *Google Consumer Barometer* 40% usam seus celulares todos os dias e 73% não saem de suas casas sem eles, devido a essa grande dependência que criamos já somos o 5° país do mundo que mais utiliza celulares, e considerando dados do primeiro trimestre de 2019 o mercado brasileiro registrou 1,3% no aumento de comércio de aparelhos em pesquisa realizada pela empresa *Gartner*, o maior entre os 5 primeiros países que mais possuem celulares (LIRA, 2019).

Embora o mercado brasileiro seja o 5° maior do mundo, quando um aplicativo não corresponde aos anseios do usuário e apresenta alguma falha por menor que seja, ou seu desempenho possa ser questionável, existe a possibilidade de não ser mais utilizado, sendo assim fatores como a sua arquitetura e modelo de desenvolvimento são fundamentais. Um aplicativo como o *Uber*, tem a necessidade de usar recursos nativos do dispositivo, entretanto este mesmo recurso pode ser obtido utilizando tecnologias híbridas, porém não com o mesmo desempenho, sendo um ponto negativo, contudo quando um aplicativo utilizará poucos recursos nativos, torna-se interessante o uso de tecnologias híbridas, pois além de torná-los multiplataforma com um único código, possuem uma curva de aprendizagem menor e um tempo de desenvolvimento mais ágil. (VALENTE, 2019).

2.2. PROGRAMAÇÃO NATIVA

Programação nativa é o termo que se refere ao desenvolvimento de aplicativos utilizando a plataforma padrão de desenvolvimento para o sistema operacional específico, como android e ios, para isto é feito uso do *SDK* disponibilizado pela empresa do sistema operacional móvel (BASSOTO, 2014, p. 22). Portanto uma aplicação nativa é um software que será executado em uma plataforma específica, onde os arquivos resultantes da

compilação serão instaladas no próprio sistema operacional, como o processamento e o armazenamento de dados (TOLEDO, DEUS, 2012).

Ao fazermos uso de aplicações nativas, o hardware presente no dispositivo pode ser melhor utilizado, como o telefone, câmera, microfone, *bluetooth* e acelerômetro, tornando-se mais útil, fácil e interativo (TOLEDO; DEUS, 2012), possuindo assim acesso a todos os recursos nativos dos dispositivos naturalmente e com um ótimo desempenho não comprometendo a experiência do usuário, pois o processamento é mais rápido assim como a sua resposta. O desenvolvimento nativo também proporciona um padrão de interface mais rico e específico, explorando muito bem tanto a *UX* quanto a *UI*, pois o próprio *Android* e o *IOS* oferecem ao desenvolvedor um guia onde é possível encontrar as melhores práticas que colaboram para que o usuário tenha uma experiência agradável. A forma de distribuí-lo também pode contribuir para que o aplicativo seja mais difundido entre os usuários, pois ele pode ser disponibilizado na loja oficial da plataforma selecionada, porém alguns sistemas operacionais como *IOS* precisam de uma licença de desenvolvimento e o aplicativo precisa ser aprovado para poder enfim ser distribuído na loja oficial, tornando o processo ainda mais demorado e até caro.

A programação nativa embora possua velocidade de processamento, experiência de usuário agradável e acesso a recursos nativos, pode ser considerado custoso devido ao tempo gasto com o desenvolvimento para ambas as plataformas, pois todo aplicativo busca alcançar o maior número de pessoas, porém o desenvolvimento nativo requer tempo, pois a curva de aprendizado de cada plataforma não é pequena e cada linguagem possui as suas particularidades, além do mais desenvolver um mesmo aplicativo nativamente implica na duplicidade dos códigos, gerando mais tempo de programação e manutenção, ainda há de se considerar o fato que cada aplicativo pode se comportar de forma diferente em sua plataforma, exigindo assim além das atualizações periódicas, atualizações específicas, vale ressaltar que embora o desempenho dos dispositivos móveis cresça constantemente, as aplicações nativas são limitadas por restrições na arquitetura dos dispositivos, como poucos recursos de computação disponíveis (Juntunen, 2013). Na figura 2 temos uma visualização das camadas existentes entre o código nativo e o hardware do aparelho.

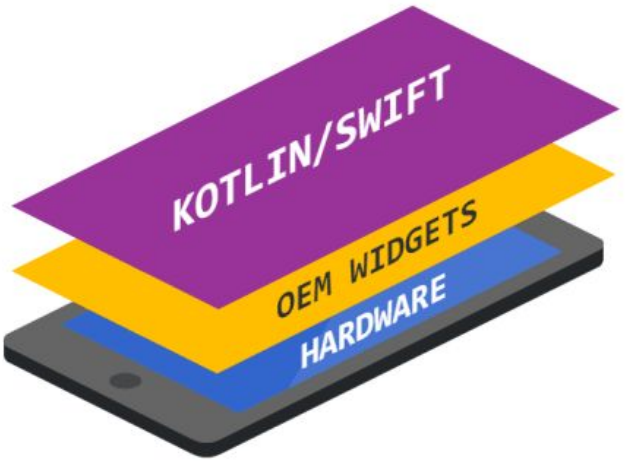


Figura 2: Camadas entre Software Nativo e Hardware
Fonte: https://dev.to/rubensdemelo/flutter-react-native-ionic-and-native-platform-a-visual-guide-2ff8

2.2.1. *ANDROID*

Android é um sistema operacional baseado no *kernel* do Linux criado por uma *startup* em outubro de 2003 e adquirido em agosto de 2015 pela *Google*, embora no início tenha sido criado apenas para utilização em câmeras fotográficas, percebeu-se que poderia ter um enorme potencial no ramo de *smartphones* (Matos; Silva, 2016, p. 32). No início os principais objetivos estava no fato de tentar revolucionar o mercado, possibilitando a criação de aplicações, componentes do sistema e a possibilidades de desenvolvimento moderno em aplicações corporativas (Hubsch, 2012, p. 13-14).

O ano de 2009 foi um marco para o sistema operacional, pois neste ano iniciou-se a arrancada no desenvolvimento de novos aparelhos utilizando o sistema *Android*. Hoje ele ocupa a maior parte do mercado e está presente em mais de um bilhão de dispositivos, um dos principais fatores que cooperam para isto é que ele não se limita a uma determinada marca de aparelho mas pode ser encontrado em dispositivos dos mais diversos fabricantes como *Samsung*, *Sony* e *Motorola*, tornando-o mais acessível. (Matos; Silva, 2016, p. 33).

A cada nova versão o *Google* procura trazer uma aparência mais sofisticada, melhorias de usabilidade, desempenho e otimização da memória apresentando sempre mais rapidez e precisão (Bassoto, 2014, p. 17), uma curiosidade interessante sobre as atualizações do sistema operacional da *Google* é que a cada lançamento, a nova versão é acompanhada por nome de um doce, em ordem alfabética. Para finalizar, segundo Hubsch (2012, p.14) “o *android* é uma plataforma que inclui: sistema operacional, *middleware* e aplicativos. Sua arquitetura é dividida em *kernel*, *runtime*, bibliotecas, *frameworks* e aplicativos”, sendo assim um sistema operacional completo.

2.2.1.1. *ANDROID STUDIO*

Para que seja possível o desenvolvimento de uma aplicação *Android* não existe requisito de utilização de nenhum sistema operacional, entretanto a *Google* recomenda a uso da *IDE Android Studio*, pois é o ambiente oficial para desenvolvimento da *Google* (Matos; Silva, 2016, p. 32). Segundo o site oficial do android studio a *IDE* foi baseada no *Intellij Idea*, o que ocasiona ferramentas e recursos avançados para que o desenvolvedor tenha uma alta produtividade, um emulador rápido, ambiente unificado, sistema de compilação flexível e outras vantagens.

Antigamente *Java* era a única linguagem de programação utilizada no desenvolvimento de aplicações *Android* oficialmente, entretanto a partir do *Android Studio* 3.0 a *Google* adicionou *Kotlin* de forma nativa a *IDE*, a partir deste momento aplicações *Android* poderiam sem desenvolvidas em *Java*, *Kotlin*, ou até mesmo fazendo uso das duas linguagens em um mesmo projeto. *Java* é uma linguagem orientada a objetos e fortemente tipada, onde tudo é representado na forma de um objeto, menos tipos primitivos, é reconhecida por sua portabilidade e versatilidade podendo ser utilizada em qualquer sistema operacional conseguindo atingir uma performance alta.

Já o *Kotlin* é uma linguagem estaticamente tipada, baseada na programação funcional, possuindo uma síntese mais expressiva que *Java*, sendo construída como uma alternativa para o desenvolvimento com *Java*, buscando ser mais precisa, algumas de suas características não são encontradas no *Java*, principalmente o fato de ser

multi-paradigma, onde não é necessário seguir um paradigma específico como orientação a objetos, ela também é mais segura e expressiva, conseguindo fazer mais com menos código.

2.2.2. *IOS*

IOS é o sistema operacional oficial da *Apple*, desenvolvida pela própria e lançado em janeiro de 2007 com o primeiro *Iphone*, foi projetada para o *hardware* de seus aparelhos e possui total integração com os seus recursos nativos como o processador e chips gráficos, embora no início ele tenha sido desenvolvido apenas para o *Iphone* hoje é utilizado em todas as suas plataformas de dispositivos como o *Ipad*, e *Ipod* (Bassoto, 2014, p. 18). O *IOS* funciona como uma interface entre as aplicações desenvolvidas por programadores e o *hardware* (Matos; Silva, 2016, p.28). A comunicação com o *hardware* do dispositivo acontece através de um conjunto bem definido de interfaces do sistema, facilitando o desenvolvimento de aplicativos, que funcionam nos mais variados tipos de hardware dos dispositivos da *Apple* (Matos; Silva (2016, p.28).

Os desenvolvedores que se interessam pelo desenvolvimento *IOS* tem acesso a um conjunto de ferramentas e *APIs* voltados para cada dispositivo *IOS*, auxiliando na criação de aplicativos móveis e jogos, garantindo que a tecnologia possa ser aproveitada ao máximo (Bassoto, 2014, p. 18). Logo no lançamento do *Iphone*, o dispositivo chamou muita atenção pois o sistema operacional *IOS* era baseado no conceito de manipulação direta, onde o usuário podiam ter contato direto com a interface através de toque ou gestos, interagindo diretamente com o dispositivo, ampliando fotos, reduzindo imagens e digitando mensagens, o que era algo totalmente novo (Hubsch, 2012, p. 22).

A sua arquitetura é dividida em camadas, onde segundo a própria *Apple* deve-se utilizar camadas mais elevadas para o desenvolvimento, pois ele se torna mais fácil, reduz a quantidade de código e mantêm as funcionalidades complexas das camadas mais baixas encapsuladas por meio de interfaces. a maioria das interfaces são disponibilizadas através de frameworks (Matos; Silva, 2016, p.30). Através da utilização do *SDK IOS*, desenvolvedores podem criar aplicações e testá-las, porém para utilizar recursos

considerados mais avançados e distribuir o aplicativo na *App Store*, é necessário adquirir uma licença que custa em torno de US$ 99 por ano, ainda assim antes do aplicativo ser publicado na loja oficial, ele precisa estar de acordo com as diretrizes impostas pela *Apple*, por isto é realizado uma avaliação do conteúdo, que pode demorar algum tempo para ser finalizado, se o aplicativo não estiver de acordo será necessário realizar mudanças no código (Matos; Silva, 2016, p.30).

2.2.2.1. *XCODE*

O *XCODE* é uma *IDE* disponibilizada pela *Apple* de forma gratuita que facilita o desenvolvimento. Ela possui algumas ferramentas que são úteis no dia a dia de um desenvolvedor, como o compilador LLVM que além de compilar o código possui uma ferramenta de auto completar códigos, para a parte visual do aplicativo ela possui o interface *Builer* que auxilia na criação de interfaces para o aplicativo, para auxiliar o gerenciamento do projeto e de dispositivos criados para testes existe o *Organizer*, ele ainda possui um simulador onde se pode verificar como o projeto está ficando tanto em *Iphones* quantos *Ipads*. Uma das grandes dificuldades no que se refere ao desenvolvimento *IOS* está no fato de que para isto precisamos como requisito um computador com o sistema operacional *Mac OS*.

Normalmente o padrão utilizado durante o desenvolvimento é o *MVC*, modelo utilizado para separar os *models*, *views* e *controllers*, assim é possível a reutilização do código dos *models* e *controllers* caso exista a necessidade de desenvolver para *Ipads* e *Iphones*, onde a única diferença será nas *views*. Atualmente a linguagem *Swift* é mais amplamente utilizada para o desenvolvimento *IOS*, ela veio para substituir a antiga *Objective C* que era muito próximo da linguagem C, com um conjunto de adições, já o *Swift* tem uma proposta mais moderna, flexível, veloz e ótima para programação funcional.

2.3. PROGRAMAÇÃO HÍBRIDA

O desenvolvimento híbrido é a união entre as tecnologias *web* e a utilização simultânea de recursos considerados nativos. É uma abordagem utilizada para que um aplicativo possa atingir várias plataformas utilizando um único código, assim o custo de desenvolvimento e manutenção se tornam mais baixo e a mão de obra mais fácil de encontrar, o tempo de desenvolvimento também é menor, pois um único código poderá ser replicado para todas as plataformas (Mendes; Garbazza; Terra, 2014, p. 2). Nesta forma de desenvolvimento grande parte do código é escrito com *Html*, *Css* e *Javascript* que são tecnologias web, e o desenvolvimento pode ser feito utilizando *frameworks* que criam uma arquitetura sem necessariamente ter vínculo com alguma plataforma (Bassoto, 2014, p. 24).

Normalmente aplicativos híbridos funcionam através de um *browser*, que interpreta a classe e exibe o conteúdo *web*. (Mendes; Garbazza; Terra, 2014, p. 2). Deste modo, assim como os aplicativos nativos, os híbridos podem ser lançados nas lojas virtuais e armazenados no dispositivo, sendo semelhantes para os usuários (Bassoto, 2014, p. 24). A sua maior desvantagem está no fato de que as funcionalidades do dispositivo não são acessadas naturalmente, mas sim através de uma ponte que possibilita conectar-se as funcionalidades nativas e assim executá-las (Mendes; Garbazza; Terra, 2014, p. 3). Apesar de terem um custo menor, aplicativos que exigem maior qualidade gráfica, como jogos, ou que precisam constantemente acessos ao *hardware* do dispositivo, terão um melhor resultado se forem construídos nativamente, agregando ainda a experiência do usuário (Mendes; Garbazza; Terra, 2014, p. 3).

No desenvolvimento híbrido muitas vezes não é fácil estabelecer um padrão no desenvolvimento das interfaces, ocasionando uma experiência não muito agradável aos usuários, portanto tanto a *UX* quanto o *UI* devem ter uma atenção dobrada, e um esforço a mais para que sejam o mais próximo possível ao visual de interfaces nativas. Sendo assim aplicativos híbridos são uma alternativa interessante ao desenvolvimento móvel, pois tem um custo baixo, o desenvolvimento é mais rápido e a manutenção também, entretanto em questão de desempenho eles podem deixar a desejar, tornando-se

necessário uma análise sobre qual a finalidade do aplicativo e a quantidade de recursos nativos que ele precisará. Abordaremos a seguir as principais tecnologias e *frameworks* utilizados para o desenvolvimento híbrido, apresentando um pouco de sua história, origem e como utilizam tecnologias semelhantes para atingir um mesmo objetivo, vale lembrar que o mercado de *frameworks* e bibliotecas que podem ser utilizados para o desenvolvimento híbrido é vasto, onde constantemente são lançadas novas soluções, entretanto algumas tecnologias que já estão a mais tempo no mercado, conquistaram um público grande e tem a sua marca no mercado em soluções corporativas.

2.3.1. *PHONEGAP* E *CORDOVA*

O *Phonegap* é um *framework* criado pela empresa Nitobi e tem por finalidade o desenvolvimento de aplicativos híbridos, posteriormente a empresa do *Phonegap* foi comprada pela *Adobe System inc*. e seu código foi doado para a *Apache Software Foundation*, tendo o seu nome alterado para *Apache Cordova* ou apenas *Cordova*, se tornando uma ferramenta *open source* (Matos; Silva, 2016, p. 33). O *Cordova* além de desenvolver aplicativos híbridos para ambas as plataformas, permite que eles tenham acesso a funcionalidades nativas, como câmeras, sensores e GPS, porém de acordo com Matos e Silva (2016, p. 33) “o *Cordova* apenas consegue fazer um aplicativo criado em *Html* rodar como se fosse nativo em um dispositivo *Android* ou *IOS*, mas não consegue imitar a sua usabilidade e aparência”, sendo a experiência do usuário um fator que deve ser tratado com mais cuidado.

Para que o *Cordova* consiga executar as funcionalidades nativas existem algumas *APIs*, que fornecem o acesso a estes recursos, mesmo quando eles não são acessíveis ou quando os recursos são específicos, existe ainda a possibilidade da criação de *plugins* que podem se comprometer ao fazer a comunicação entre a ferramenta e o componente nativo (Bassoto, 2014, p. 28). O *Cordova* permite que o aplicativo desenvolvido possa ser multiplataforma ou específico para alguma plataforma que é conhecido como desenvolvimento centralizado, segundo Bassoto (2014, p. 28) “o desenvolvimento centralizado é indicado quando o aplicativo necessita de codificação específica em

funcionalidades da plataforma utilizada, tornado-se de baixo nível". Para que seja possível desenvolver multiplataforma é disponibilizada uma ferramenta conhecida como *CLI*, onde conseguimos acessar recursos específicos e instalar *plugins* que nos auxiliarão nesta tarefa (Bassoto, 2014, p. 29).

2.3.2. *IONIC*

O *Ionic* é um *framework* utilizado para desenvolvimento híbrido com tecnologias *web*, como o *Html*, *Css* e o *Javascript*, para isso ele utiliza o *Angular*, um poderoso *framework Javascript*, que foi desenvolvido pela *Google* e é amplamente utilizado, recebendo sempre atualizações constantes (Matos; Silva, 2016, p. 34). Ele foi desenvolvido pela empresa *Drifty co*. em meados de 2013, surgindo como uma alternativa com alta performance e funcionando com padrões e tecnologias *web* (Matos; Silva, 2016, p. 34).

De acordo com Silva e Soto (2017, p. 97) "aplicativos desenvolvidos com *Ionic* são como pequenos sites que funcionam em uma camada embutida do navegador em um aplicativo, o qual possui acesso à camada da plataforma nativa através de bibliotecas fornecidas pelo próprio *Ionic*". Por fazer uso de *Html5* e para que seja executado como uma aplicação próxima ao nativo, acessando recursos e funcionalidades da plataforma, é preciso que ele atue junto com o *Cordova* ou *Phonegap*, que fazem a função de um invólucro nativo (Silva; Soto, 2018, p. 100).

Para o seu desenvolvimento não é necessário nenhum sistema operacional específico, e o ambiente de desenvolvimento pode ser de escolha do desenvolvedor, ao menos que a aplicação seja para *IOS*, neste caso ainda será preciso um *Mac Os*, ou uma emulação do sistema junto com o *XCODE*. Atualmente devido a atualizações constantes da *Google* e ao aprimoramento que o acompanha, outras tecnologias foram adicionadas ao *Ionic*, embora o *Angular* seja o principal *framework* utilizado no seu desenvolvimento, hoje já pode desenvolver com *Javascript* puro, *Vue.js* e até mesmo o próprio *React*. a sua documentação é bem completa e possui uma biblioteca com todos os componentes disponíveis, como customizá-los e ainda como implementá-los independente da plataforma escolhida.

Entre os seus benefícios vale destacar que a curva de aprendizagem é considerada muito baixa, permitindo em pouco tempo construir seus primeiros aplicativos, ainda mais se o desenvolvedor tiver conhecimentos prévios em tecnologias *web* (Silva; Soto, 2018, p. 100). Por se utilizar de tecnologias *web*, também é possível a visualização em tempo real de desenvolvimento e ainda contar com recursos do *Google Chrome* e do *Firefox* (Silva; Soto, 2018, p. 100). Apesar de em um primeiro momento o *Ionic* ser uma alternativa atraente para o desenvolvimento móvel, ele não está isento de desvantagens, pois existem relatos de problemas de performance em sistemas *Androids* antigos, diferença entre os componentes *UI* quando acontece a renderização para a plataforma nativa e problemas de performance. Portanto deve-se analisar muito bem o aplicativo que será desenvolvido ao escolher o *Ionic*, para que assim o usuário tenha uma experiência mais agradável. Na figura 3 temos um exemplo de camadas entre o *IONIC* e o *hardware*.

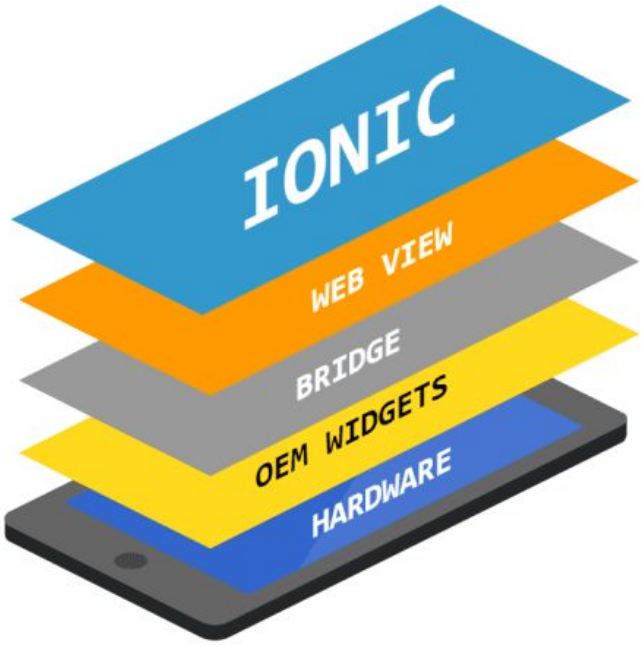


Figura 3: Camadas entre IONIC e Hardware
Fonte: https://dev.to/rubensdemelo/flutter-react-native-ionic-and-native- platform-a-visual-guide-2ff8

2.3.3. *REACT NATIVE*

React Native é um *framework Javascript* criado pelo *Facebook* para desenvolvimento de aplicativos nativos em dispositivos móveis *Android* e *IOS*, é baseado no *React* que é uma

biblioteca Javascript para aplicações *Web*. Assim como o *Ionic* não exige um ambiente de desenvolvimento específico a *IDE* ou o editor pode ser de acordo com a preferência do programador, sendo necessário apenas o uso de algum gerenciador de pacotes *Javascript* como o *NPM* por exemplo, se o aplicativo possuir necessidade de ser desenvolvido para *IOS*, será necessário um *Mac OS* para gerar o executável, e caso venha a ter funções distintas entre as plataformas, é possível deixa tudo isso configurado no código.

O *React* funciona através da criação de componentes modulares reativos por meio de *data binding* sendo utilizado na criação de aplicações de página única. (Cruz; Petrucelli, 2018, p. 5), ele foi criado por engenheiros do *Facebook* para auxiliar na criação de interfaces de usuário complexas, com o propósito de solucionar alguns problemas, entretanto o *React Native* é destinado apenas a plataformas móveis (traça, 2018, p. 22). Seu propósito é possibilitar que desenvolvedores *web*, possam criar aplicativos multiplataformas utilizando basicamente *Javascript*, embora o código possa ser o mesmo isto não implica que ele será um aplicativo híbrido, pois ele se comunica de forma nativa com o sistema operacional.

O seu desenvolvimento consiste na criação de interfaces utilizando uma extensão *Javascript* conhecida como *JSX*, através dela é utilizado sintaxe *XML*, sem seguir alguma semântica ou limitação de *tags*, que possibilita criar interfaces ao lado da lógica de programação (Cruz; Petrucelli, 2018, p. 4). Diferente do *Ionic*, *React Native* não faz uso de uma *webview*, sendo assim o seu desempenho é maior, pois o *JSX* e o *Javascript* trabalham simultaneamente para acessar os recursos nativos do aparelho, o que ocasiona a renderização do aplicativo nativamente, sem utilizar nenhuma *webview* e sim com os recursos próprios do aplicativo (Muller; Soares, 2019, p. 12). Para que isto possa ocorrer é necessário a utilização de um conceito conhecido como *bridge*, considerada uma ponte entre a linguagem e os recursos nativos, realizando uma comunicação eficiente entre o aplicativo e o processador do dispositivo, porém pode gerar atrasos de processamento de *UI* (santos, 2018, p. 20).

O *React Native* possui inúmeras vantagens, por exemplo, testar o aplicativo em tempo real, em um celular por com conexão *usb* ou pela rede *wi-fi*, utilização de um único código para desenvolvimento multiplataforma e até mesmo o reaproveitamento do código quando

já existe uma versão distribuída em formato de site, o fato do *Facebook* estar por trás do *React* aumenta a confiança dos desenvolvedores e do mercado, fazendo com que exista uma comunidade sólida pelo mundo, além de grandes oportunidades de emprego. O *React Native* veio para inovar, sendo veloz, possuindo interfaces bem definidas e um desempenho muito superior comparado com aplicações desenvolvidas com *Ionic*, podendo chegar próximo a experiência nativa. É observado as camadas existentes entre o *React Native* e o *hardware* na figura 4.

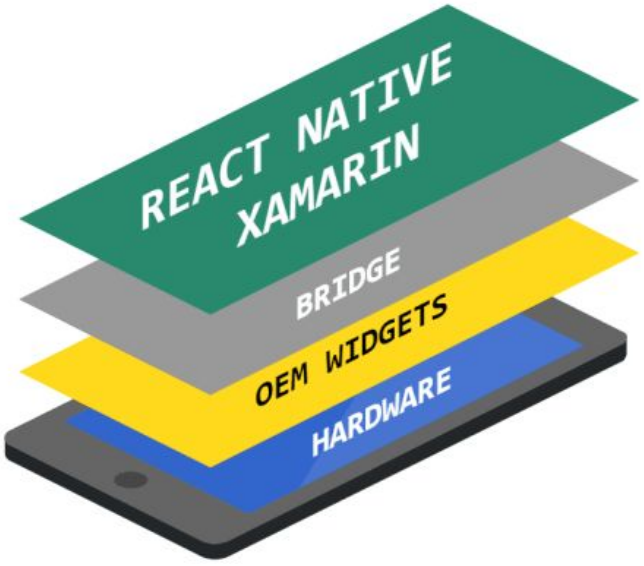


Figura 4: Camadas entre React Native e o Hardware
Fonte: https://dev.to/rubensdemelo/flutter-react-native-ionic-and-native-platform-a-visual-guide-2ff8

3. FLUTTER

Neste capítulo será feito uma apresentação do *Flutter*, começaremos falando um pouco de sua história e finalidade, logo a seguir será apresentado a sua arquitetura e posteriormente a linguagem por trás do *framework*, por fim apresentaremos seus principais conceitos e comandos. A descrição das camadas do *Flutter* com o *hardware* pode ser visualizada na figura 5.

Figura 5: Camadas entre o Flutter e o Hardware
Fonte: https://dev.to/rubensdemelo/flutter-react-native-ionic-and-native-platform-a-visual-guide-2ff8

3.1. INTRODUÇÃO

Flutter é um *framework* desenvolvido pela *Google*, apresentado pela primeira vez em 2015 lançado oficialmente em 2018, que pode ser utilizado para o desenvolvimento nativo de aplicações *Android* e *IOS* (Muller; Soares, 2019, p. 14). Apesar do *Flutter* já ter sido apresentado com uma proposta de inovação genial, a comunidade de desenvolvedores possuía desconfiança, principalmente pelo fato da *Google* ser conhecida por abandonar

seus projetos, entretanto o sucesso do *Flutter* foi imediato, atraindo desenvolvedores e criando comunidades sólidas ao redor do mundo.

O *Flutter* também pode ser definido como um conjunto de ferramentas de *user interface* portáteis tornando o desenvolvimento de interfaces gráficas um de seus pontos fortes, pois é possível com grande facilidade criar interfaces agradáveis e ter acesso a uma extensa biblioteca de *widgets* e animações que vem por padrão no *framework* (Muller; Soares, 2019, p. 14). Assim como o *React Native*, ele possui uma funcionalidade conhecida como *hot reload*, que permite que o aplicativo possa ser recarregado com velocidade, mantendo o estado da última seção e assim testar as mudanças em tempo real (Santos, 2018. p. 23).

Para o desenvolvimento *Android* é necessário a utilização do *Android Studio* ou do *Visual Studio Code*, sendo que o sistema operacional pode ser de acordo com a preferência do desenvolvedor, entretanto para a construção de aplicações *IOS* poderá ser utilizado o *XCODE* junto ao sistema operacional *Mac OS*. Para o desenvolvimento, independente do sistema operacional, a linguagem utilizada é o *Dart*, que foi criada pelo *Google*, para ser rápida, reativa e portável para multiplataformas, além de possuir uma sintaxe clara e consistente (Santos, 2018. p. 23). Embora o foco inicial tenha sido o desenvolvimento móvel, a *Google* tem planos audaciosos para o *framework*, tendo como objetivo a possibilidade de desenvolvimento para *desktop* e *web*, mesmo o *Dart* pode ser utilizado até na criação de *APIs Restful*, demonstrando assim o poder da linguagem, e o comprometimento da *Google* em fazer algo grandioso.

Portanto o *Flutter* é um *framework* poderoso, com o propósito de criar aplicações de forma rápida, interfaces de usuário bonitas, flexíveis e mantendo sempre a performance nativa da aplicação. O *Flutter* também é o *framework* padrão para o desenvolvimento de aplicativos no *Fuchsia OS*, o sistema operacional móvel do *Google*.

3.2. INSTALAÇÃO

O processo de instalação pode ser acompanhado na documentação, e independente do sistema operacional é bem simples, com apenas algumas particularidades entre eles. Para *Windows* e *Linux* será necessário instalar o *Android Studio* embora a sua utilização no desenvolvimento seja opcional. Já no *Mac OS* o *Xcode* é a ferramenta necessária. Independente do sistema operacional, o *Flutter* disponibiliza uma ferramenta conhecida como *Flutter Doctor*, ela tem a tarefa de auxiliar durante a instalação, verificando se tudo está de acordo, se por acaso faltar algo para o funcionamento correto, o próprio *Flutter Doctor* mostra qual a solução. Sendo assim o processo de instalação do *Flutter* tende a ser descomplicado e fácil.

3.3. ARQUITETURA

Segundo Santos (2018, p. 24), "*Flutter* possui uma arquitetura baseada em camadas as quais são bibliotecas responsáveis por funcionalidades específicas em um *app*". As camadas também são construídas e implementadas uma sobre a outra, e é composta pela *engine*, que é considerada uma camada fina de código c/c++ (Corazza, 2018, p. 23). As funcionalidades e camadas são todas disponibilizadas em *Dart*, sendo que o c++ é responsável apenas pela parte de renderização (Muller; Soares, 2018, p. 15).

Na maioria das vezes as camadas possuem nomes autoexplicativos, como *animation* responsável por todo sistema de animação ou *gesture* que seria uma camada própria para lidar com gestos do usuário (Santos, 2018, p. 24). A arquitetura do *Flutter* também é capaz de aceitar a utilização de códigos nativos, independente da linguagem, para isto é necessário a instalação de um pacote específico (Santos, 2018, p. 25).

Na figura 6 pode ser visualizado uma ilustração de sua arquitetura, nele existe camadas divididas em duas categorias, onde a primeira se refere ao *framework* com suas funcionalidades e especificações que ajudam e auxiliam no desenvolvimento, logo depois nos deparamos com a *engine*, está camada construída em c++ possui as informações

sobre a renderização do aplicativo para que possa ser executado na plataforma de forma nativa.

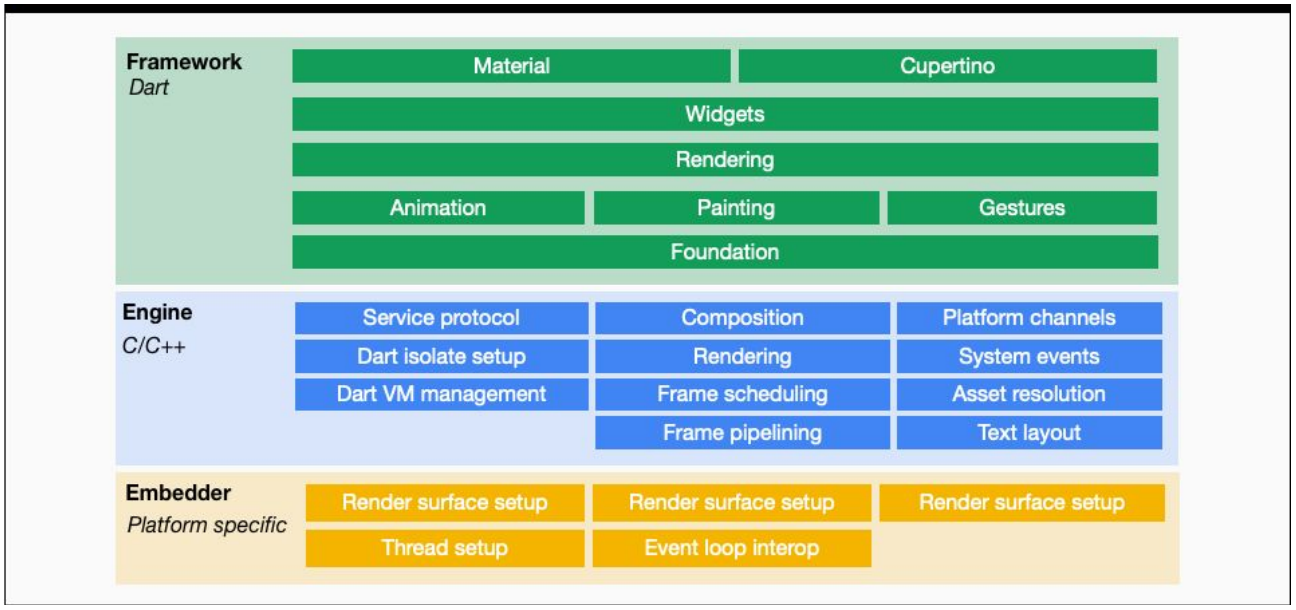


Figura 6: Arquitetura Flutter
Fonte: https://flutter.dev/docs/resources/technical-overview

3.4. *DART*

Dart é uma linguagem de programação criada pelo *Google* em 2011 que tinha como propósito substituir o *Javascript* se tornando a principal linguagem a ser utilizada nos navegadores. Ela é uma linguagem moderna, orientada a objetos, concisa e fortemente tipada (Corazza, 2018, p. 23). Sua sintaxe é clara e consistente, tendo o propósito de ser rápida, portátil, multiplataforma e reativa (Santos, 2018, p. 23). Ela é muito parecida com outras linguagens populares como o *Java*, *C#*, *PHP* e *Javascript,* não trazendo grandes dificuldades para quem conhece estas linguagens.

Pode ser considerada uma linguagem altamente versátil pois pode ser utilizada em desenvolvimento de aplicações móveis nativas, *desktop*, *web*, criação de *scripts* e até mesmo no *back-end*, porém para cada tipo de aplicação a ser desenvolvida será necessário a utilização de um *framework* diferente e que terá o suporte para tal. *Dart* possui como característica a capacidade de poder inferir os tipos, tornando-se opcional

declarar o tipo da variável ao criá-la, visto que a linguagem assumira o primeiro tipo associado a variável.

A linguagem *Dart* possui algumas características interessantes como o *ahead-of- time* que possibilita que o código seja compilado para o *arm* nativo, proporcionando assim um desempenho de uma aplicação nativa, e o *just-in-time* que permite que as alterações sejam vistas em tempo real, assim que o código for modificado[1]. Existem muitos *frameworks* já criados que tem o *Dart* como linguagem, sendo o *Flutter* o mais conhecido e também um dos grandes responsáveis pela disseminação da linguagem.

Para a parte de *web* existem o *Angulardart*, *Material Design Lite*, *Overreact* e o *Vuedart*, já para o *back-end* os mais populares são o *Aqueduct*, *Angel* e o *Jaguar*. A escolha do editor de código para a codificação fica a gosto do desenvolvedor, pois *Dart* possui *plugins* compatíveis com os principais editores do mercado.

3.4.1. SINTAXE

A sintaxe de *Dart* é semelhante a linguagens como *Java*, *Typescript* e C#. A declaração do tipo de variável quando ocorre é sempre antes da própria, às estruturas de repetições são as mesmas e existe a necessidade de ponto e vírgula ao final de cada linha de código. Orientação a objetos é algo estritamente ligado à linguagem, assim como em *Java* e *C#*. *Dart* possui alguns conjuntos de funções muito semelhante ao *Typescript* ou ao próprio *Javascript*. A seguir será demonstrado exemplos da linguagem *Dart* de modo que programadores com experiência em outras linguagens encontraram uma boa base para começar na linguagem.

[1] Disponivel em https://dart.dev/platforms

```dart
String name = 'Leonardo';
bool ativo = true;
double valor = 33.3;
num valor1 = 29;
int valor2 = 50;
Map<String, dynamic> caracteristicas = {};
dynamic dinamico = 55;
dinamico = "Valor dinamico";
List listaUsuarios;
```

Figura 7: Exemplos de variáveis em Dart

Na figura 7 é demonstrado um exemplo de variáveis em Dart que em sua maioria já é conhecidas por programadores, seus tipo principais são, String, bool, double, num e int, existe também o tipo dynamic, uma variável com este tipo pode receber qualquer tipo de valor, muito semelhante ao *var* do *Javascript*.

```dart
List times = ["Flamengo", "Botafogo", "Fluminense", "Vasco"];
List numeros = [1,2,3,4];

List cursos = new List();
cursos.add("BCC");
cursos.add("ADS");

List navegadores = new List(3);
navegadores[0] = "Google Chrome";
navegadores[1] = "Internet Explorer";
navegadores[2] = "Firefox";
```

Figura 8: Exemplos de tipos de listas

A utilização de listas pode ser considerada simples como é observado na figura 8, e é bem semelhante ao uso de vetores. Pode ser criada uma já adicionando diretamente os itens como se fosse um *array*, ou mesmo criar como seu fosse um objeto, instanciado o tipo List(), nesse caso para adicionar os itens, pode-se fazer uso do método *add* ou popular a posição específica de acordo com o desejado.

```
Map materiaBCC = {
  "calculo":6,
  "fisica":7,
  "teoria_da_computacao":9
};
materiaBCC["calculo"] = 10;

materiaBCC
  ..["algoritmo"] = 10
  ..remove("fisica");
```

Figura 9: Exemplos de tipos de Maps

Na figura 9 existem mapas, que são bem simples, essa estrutura de dados precisará sempre ser trabalhada com uma chave e com um valor. Na maioria das vezes são utilizados String para chaves e dynamic para valores, tendo em mente que qualquer valor pode ser atribuído a um mapa. Tipos mapas são utilizados principalmente em conexões e operações com o *Firebase*.

```
void main() {
  calculaValor(5,10);
  String curso = retornaCurso("BCC");
  String dados = retornaNCM("Leonardo", "BCC", "TCC");
  print(curso);
  print(dados);
}

void calculaValor(double valor1, double valor2) {
  double total = valor1 + valor2;
  print(total);
}

String retornaCurso(String curso) {
  return "Hello " + curso;
}

String retornaNCM(String n, String c, String m) {
  return "Dados: $n, $c, $m";
}
```

Figura 10: Funções retornos e concatenações

Na figura 10 pode ser visualizado alguns exemplos de funções que possuem retornos e também como pode se concatenar da forma correta Strings em uma classe. Vale destacar a semelhança existente com as principais linguagens do mercado, e como *Dart* possui uma sintaxe simples e clara.

```
void main() {
  String curso = "BCC";
  switch (curso) {
    case "BCC":
    case "ADS":
      print("Tecnologia");
      break;
    case "DIREITO":
      print("Legislação");
      break;
    case "PUBLICIDADE":
      print("Criatividade");
      break;
    default:
      print("Não encontrado");
  }
}
```

Figura 11: Exemplo de switch

O uso da estrutura *switch* demonstrado na figura 11 não possui complicações, e assim como as funções lembram linguagens como C++, Java e Javascript.

```
void main() {
  List cursos = ["BCC","ADS","SIS"];
  for(var c in cursos) {
    print(c);
  }
  int x = 0;
  while(x < cursos.length) {
    print(cursos);
    x ++;
  }
  do {
    print(cursos);
    x ++;
  } while(x < cursos.length);
}
```

Figura 12: Estruturas de repetição

As estruturas de repetições são simples, claras e existentes na maioria das linguagens modernas. A utilização do *For* da figura 12 é elegante, economiza linhas de códigos e possui uma sintaxe muito simples, onde para cada objeto da lista será criado uma variável do tipo *Var* para ele. O tipo *var* do *Dart* é semelhante ao tipo *var* do *Javascript*.

```
Run | Debug
void main() {
  Professor p = new Professor("Andre", "Anee");
}

class Professor {
  String primeiro;
  String segundo;

  Professor(this.primeiro, this.segundo) {}
}

class Mestrado extends Professor {
  Mestrado(String primeiro, String segundo) :
    super(primeiro, segundo);
}
```

Figura 13: Classe com Herança

A utilização de herança se faz presente através da palavra reservada *extends*, onde a classe terá acesso aos valores e métodos da classe mãe, como na figura 13.

3.5. *WIDGETS*

Widgets são considerados blocos fundamentais da interface, são responsáveis por muitos aspectos do *Flutter*, como a estrutura dos elementos, as cores, estilos e aspectos, além do modo como devem agir e se comportar (Santos, 2018, p. 23). Existem *widgets* específicos para *Android* que são conhecidos como *Material Components* e específicos para *IOS* conhecidos como *Cupertino*, além disso os *widgets* podem ser personalizados, e se ainda for necessário o *Flutter* facilita muita a criação e projeção de novos (Corazza, 2018, p. 23).

O *Flutter* se diferencia muito dos seus concorrentes, pois ele não utiliza os *widgets* fornecidos pelos dispositivos, mas usa seu próprio mecanismo de renderização de alto desempenho para poder assim desenhar o *widget* (Corazza, 2018, p. 23). Os *widgets* podem ser divididos em dois tipos de categoria, que são diretamente ligados ao seu comportamento, as categorias são conhecidas como *stateless widget* e *statefull widget*. *Widgets* do tipo *stateless widget* possuem como característica o fato de não mudarem de estado, eles são usados e indicados para a criação de interfaces concretas, que são aquelas que não mudaram conforme exista a interação do usuário, como um texto fixo na interface ou mesmo uma imagem, um bom exemplo é um container que não sofrerá nenhuma alteração conforme exista a interação do usuário.

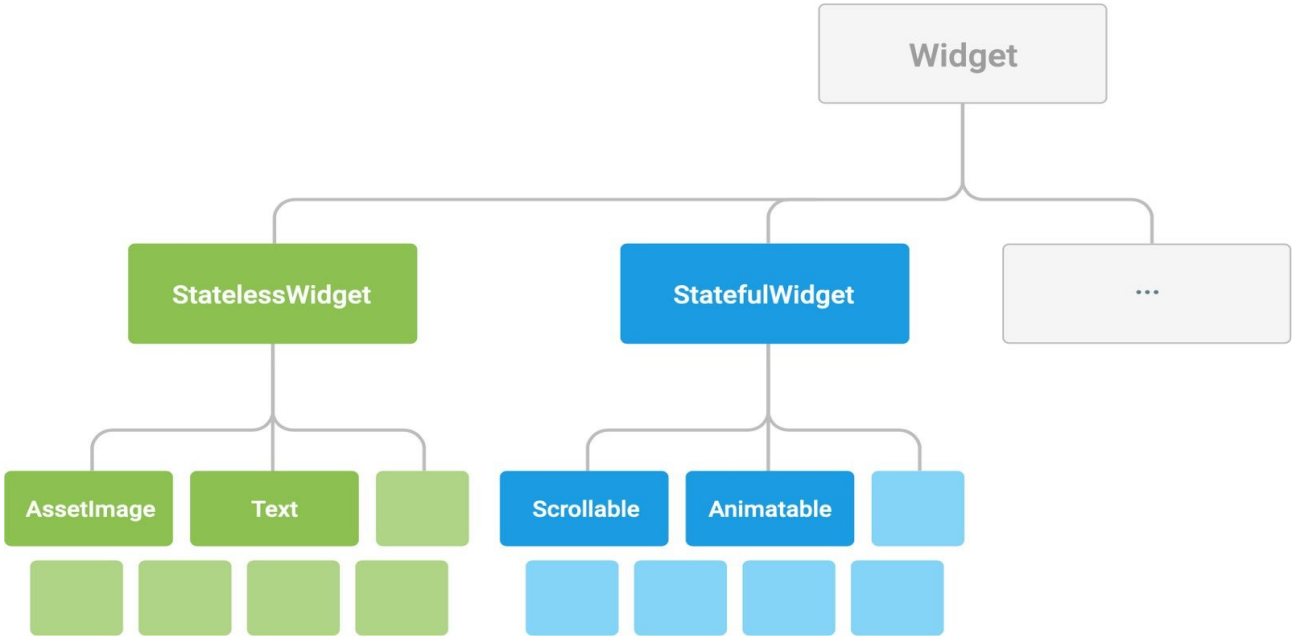


Figura 14: Exemplos de Stateless e Statefull
Fonte: https://proandroiddev.com/flutter-from-zero-to-comfortable-6b1d6b2d20e

Os *widgets* do tipo *statefull* são definidos como aqueles que alteram o seu estado conforme a interação do usuário ou mesmo conforme o andamento do próprio aplicativo, eles são utilizados em grande parte para lidar com atividades assíncronas, ou quando um *widget* passará por alguma mudança oriunda de seu ciclo de vida (Santos, 2018, p. 23). *Widgets* em *Flutter* são extremamente importantes, pois tudo nele é um *widgets*, desde uma caixa de texto, um formulário, uma imagem, um botão ou um *container*, independente do que está sendo escrito ou feito, tudo sempre será um *widget*. Essa diferenças podem ser observadas na figura 14.

3.5.1. EXEMPLOS DE *WIDGETS*

Nesta sessão será demonstrado de forma prática a utilização de *widgets* e o funcionamento do próprio *Flutter*. Para isso será desenhado uma interface simples, mas com funcionalidades interessantes, assim será demonstrado que pode se fazer muito com pouco código.

```
class MyApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {

    return MaterialApp(
      debugShowCheckedModeBanner: false,
      title: 'Flutter Demo',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ), // ThemeData
      home: Home(),
    ); // MaterialApp
  }
}
```

Figura 15: Exemplo de Classe principal no Flutter

De início foi feito na figura 15 a interface responsável pelas configurações iniciais, nela são definidas as cores primárias, secundárias, título do aplicativo e tema. Na propriedade *debugshowcheckedmodebanner* é definido com *true* ou *false* se existirá um *banner* durante a programação informando ser apenas uma versão de desenvolvimento, a propriedade *home*, será a interface inicial. é importante também notar a função *void main* e na sua sintaxe, responsável por inicialmente executar o código.

```
class Home extends StatelessWidget{
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        centerTitle: true,
        title: const Text("Appbar"),
        elevation: 100,
        actions: <Widget>[
          PopMenuWidget()
        ], // <Widget>[]
      ), // AppBar
      drawer: DrawerWidget(),
      body: FormWidget()
    ); // Scaffold
  }

}
```

Figura 16: Exemplo de Scaffold

O código acima é responsável pela interface inicial, ele começa com o *widget* conhecido e que é frequentemente utilizado durante toda a aplicação, o *scaffold*. Nela pode ser

definida uma barra superior com ou sem botões, um menu lateral e o corpo da aplicação. Neste exemplo é definido a barra superior, para isto foi utilizado a propriedade *appbar*. Com o *centertitle* o título da barra será centralizado ou não, a seguir o nome será escrito na barra e depois definido o grau da sua elevação, logo a seguir é definido o *popmenu* e o *drawer*, para estes dois foram escritas classes em arquivos diferentes, referentes a cada *widget*, mantendo assim a organização do código. A figura 17 mostra o resultado inicial, aliados a interface de configuração e a interface com o *scaffold*.

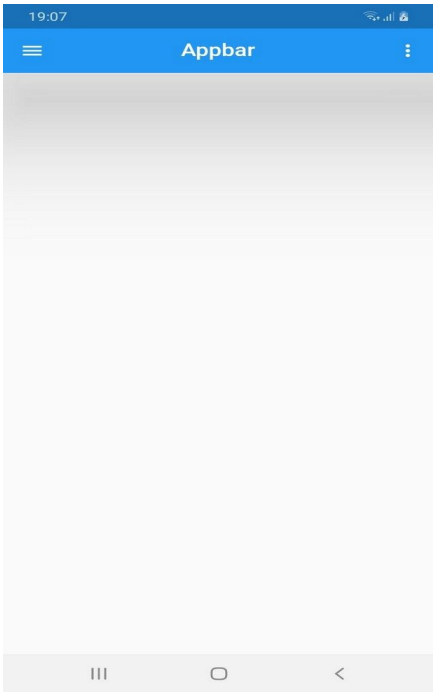


Figura 17: Resultado da interface principal e scaffold

O código da figura 18 se refere ao *popmenubutton*, com ele será criado um menu lateral que se sobrepõe a interface inicial, primeiramente será necessário criar um enum com as alternativas do menu, assim código ficará mais organizado, a seguir será usado a propriedade *icon* para definir um ícone no *popmenu*.

```
enum OPCOES {SETTINGS, EXIT}
class PopMenuWidget  extends StatelessWidget{
  @override
  Widget build(BuildContext context) {
    return PopupMenuButton<OPCOES>(
      icon: Icon(Icons.more_vert),
      onSelected: null,
      itemBuilder: (context) => <PopupMenuEntry<OPCOES>> [
        const PopupMenuItem(child: Text("Opções")),
        const PopupMenuItem(value: OPCOES.EXIT, child: Text("Sair"),)
      ], // <PopupMenuEntry<OPCOES>>[]
    ); // PopupMenuButton
  }
}
```

Figura 18: Classe responsável pelo PopMenu

Após isso é escrito o código para a função que realmente faz o menu funcionar, a propriedade *itembuilder*, nele é configurado o enum que contêm as opções do menu, com seu *value*, e seu *child*, onde pode ser adicionado um texto que será referência para as opções deste menu, para definir uma ação ao selecionar a opção é necessário programar a propriedade *onselect*. Na figura 19 é demonstrado o resultado final do *popmenu*.

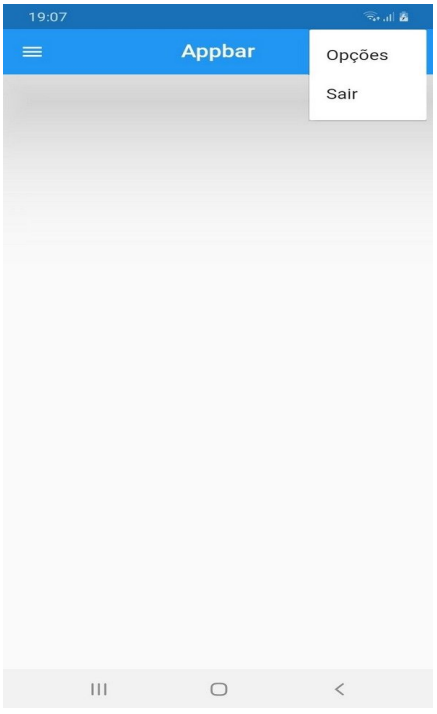


Figura 19: Visualização do PopMenu

Na figura 20 será feito a criação de um menu lateral, muito popular e utilizado em diferentes aplicações. Neste código retornaremos um *drawer* responsável pela criação

deste menu e dentro dele existirá um *child* que conterá uma *listview*, nesta propriedade é inserido cada item do menu lateral, entretanto neste caso será criado uma classe específica para desenhar a opção na interface, assim quando é necessário inserir um novo elemento no menu chamamos a classe *drawertile* e passamos em seu construtor as suas propriedades, como seu ícone e nome.

```
class DrawerWidget extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Drawer(
      child: ListView(padding: EdgeInsets.only(top: 75), children: <Widget>[
        DrawerTile(Icons.home, "Home"),
        DrawerTile(Icons.account_circle, "Perfil"),
        DrawerTile(Icons.build, "Ferramentas"),
      ], // <Widget>[]
      )); // ListView // Drawer
  }
}
```

Figura 20: Classe responsável pelo menu lateral

A classe *drawertile*(figura 21) retorna um *container* e dentro deste *container* existe uma *row*, onde será inserido um ícone, um espaço utilizado o widget *sizedbox* e um texto, assim todos os itens do menu possuirão o mesmo tamanho de ícone e texto, mantendo o código mais limpo e organizado.

```
class DrawerTile extends StatelessWidget{

  final IconData icon;
  final String txt;
  DrawerTile( this.icon, this.txt);

  @override
  Widget build(BuildContext context) {
    return Container(
      padding: EdgeInsets.only(left: 32), height: 60,
      child: Row(
        children: <Widget>[
          Icon(icon, size: 32, color: Colors.blue,),
          SizedBox(width: 32),
          Text(txt,style: TextStyle(fontSize: 16, color: Colors.blue),)
        ], // <Widget>[]
      ), // Row
    ); // Container
  }
}
```

Figura 21: Classe responsável pelo desenho de cada item do menu lateral

Na figura 22 é apresentado o resultado do menu lateral.

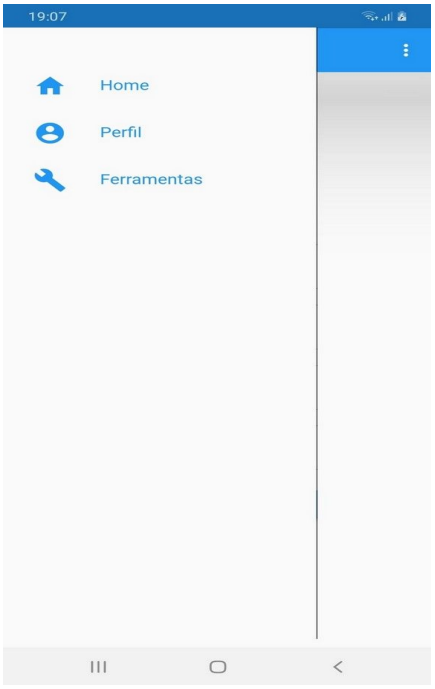


Figura 22: Visualização do menu lateral

A seguir será feito a parte final da interface, o formulário de cadastro. nesta interface equivalerá o exemplo anterior, onde para cada componente do formulário é chamada uma classe para inseri-lo na interface, mantendo o código limpo.

```
class FormWidget extends StatelessWidget{
  @override
  Widget build(BuildContext context) {
    return Container(
      padding: const EdgeInsets.only(bottom: 20, left: 30, right: 30),
      child: ListView(
        children: <Widget>[
          Container(padding: EdgeInsets.only(top: 40, bottom: 30),
            child: Icon(Icons.face, color: Colors.blue, size: 150),
          ), // Container
          InputWidget(obs: false, campo: "Nome", icon:Icons.supervised_user_circle),
          InputWidget(obs: false, campo: "Email", icon:Icons.mail),
          InputWidget(obs: true, campo: "Senha", icon:Icons.lock),
          InputWidget(obs: false, campo: "Endereço", icon:Icons.confirmation_number),
          RaisedButton(color: Colors.blue, onPressed: (){},
            child: Text("Entrar", style: TextStyle(color: Colors.white),),
            shape: RoundedRectangleBorder(borderRadius: new BorderRadius.circular(30.0)),
          )],), // RaisedButton // <Widget>[] // ListView
    ); // Container
  }
```

Figura 23: Classe responsável pelo formulário

Na figura 23, para que possa desenhar o formulário será retornado um *container*, ele conterá propriedades de tamanhos específicos para que assim possa ficar centralizado na interface e logo em seguida retornará uma *listview*, primeiramente será inserido o ícone,

para isto fará uso de mais um *container*, logo em seguida será chamado a classe *inputwidget* e em seu construtor poderá ser informado o nome, ícone e se o campo será obscuro ou não. Por fim é inserido um *raisedbutton*, este *widget* é responsável por inserir um botão na interface, sendo que também pode ser definido as suas características e ainda chamar uma função para ser executada quando for pressionado.

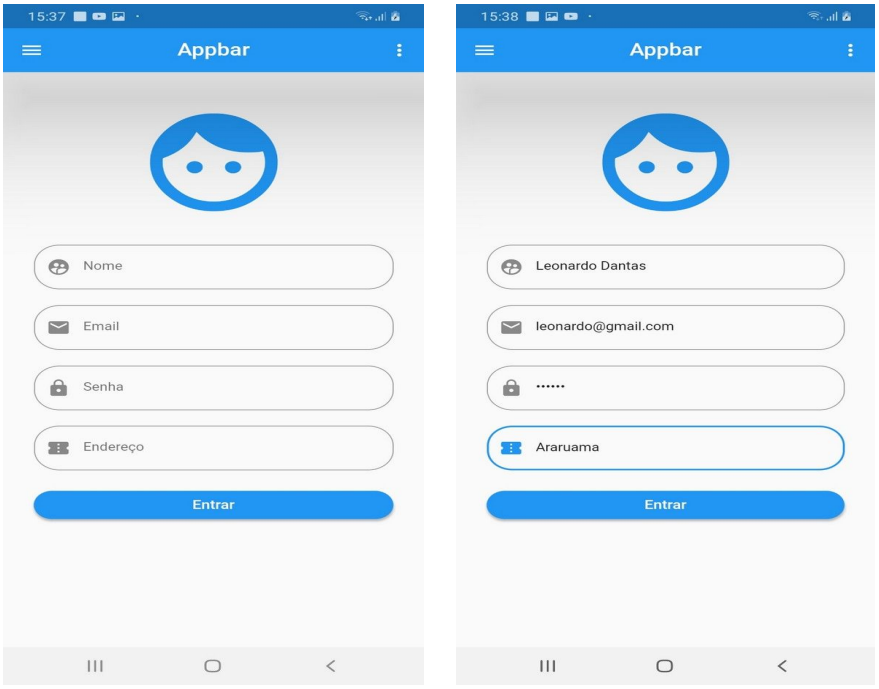
```
class InputWidget extends StatelessWidget {

  final bool obs;
  final String campo;
  final IconData icon;
  InputWidget({this.obs, this.campo, this.icon}){}

  @override
  Widget build(BuildContext context) {
    return Padding(
      padding: const EdgeInsets.only(bottom: 20),
      child: TextFormField(
        obscureText: obs,
        style: TextStyle(fontSize: 13),
        decoration: InputDecoration(
          prefixIcon: Icon(icon),
          hintText: campo,
          border: OutlineInputBorder(borderRadius: BorderRadius.circular(25))
        ),),); // InputDecoration // TextFormField // Padding
  }
}
```

Figura 24: Classe responsável pelo desenho de cada campo input

Na figura 24 é mostrado o código da classe *inputwidget*, onde é retornado um padding, e dentro dele existe uma *textformfield*, onde é definido se o texto digitado será do tipo senha, o ícone e o texto que será apresentado na interface. A seguir na figura 25 é demonstrado uma figura representando o código acima e logo depois uma figura com os campos devidamente preenchidos.



a) Formulário vazio **b)** Formulário preenchido
Figura 25: Visualização do formulário sem informações e com informações

3.5.2. CICLO DE VIDA DE *WIDGETS*

O ciclo de vida de um *widget* é composto por pouco estados. o primeiro sempre a ser chamado é o *initstate*, que tem como propósito dar início a tarefas que precisam ser executadas apenas uma vez, inicializando os *listeners*. Logo depois vem o *build*, ele possui a hierarquia que será desenhada na interface, por fim é chamado o método *dispose* responsável por remover os *listeners* inicializados no método *initstate*. Um método importante e ligado diretamente aos *widgets* é o *setstate*, através dele pode ser mudado o estado do *widget*, fazendo que ele seja renderizado novamente, mas desta vez com um novo estado da aplicação (Santos, 2018, p. 26).

3.6. *DART PACKAGES*

O *Flutter* permite a utilização de pacotes que visam facilitar a vida do desenvolvedor, estes pacotes podem ser desenvolvidos por qualquer pessoa e ser publicado no site

pub.dev. Por exemplo, se por acaso o aplicativo necessitar durante seu desenvolvimento se comunicar com o banco de dados *Firebase*, então pode ser utilizado algum package já criado por outro usuário, tornando o desenvolvimento mais produtivo, para que o package seja instalado basta adicionar o seu nome no arquivo pubspec.yaml. O próprio *Flutter* irá se encarregar de fazer todos os ajustes necessários. Caso exista alguma dúvida sobre a utilização do *package*, é possível consultar a sua documentação no próprio site pub.dev. Existem diversos packages *desenvolvidos*, embora o exemplo empregado seja o de conexão com um banco de dados, eles podem ser relacionados a diversos assuntos como *designer*, melhorias de experiência de usuário, conexão com diversos bancos de dados e até mesmo a integração com algumas *APIs*.

3.7 GERÊNCIA DE ESTADOS

Gerência de estado é um termo que se refere a atualização de toda a interface quando existe alguma modificação em um widget. embora esse processo possa passar despercebido por ser estritamente rápido, o *hardware* do dispositivo é limitado, podendo ocasionar lentidão ou queda de desempenho em alguns momentos. Entretanto o *Flutter* oferece algumas alternativas para a gerência destes estados, entre elas as mais conhecidas e utilizadas estão o *Bloc* e o *Mobx*. Além de solucionar o problema do desempenho e da reconstrução de interfaces ao serem atualizadas, esses *packages* também permitem que exista uma organização maior em relação a parte de *designer* e a parte lógica da aplicação.

3.8 FLUTTER VERSUS REACT NATIVE

Para muitos *Flutter* e *React Native* protagonizam a maior disputa no que se diz respeito ao desenvolvimento móvel para ambas as plataformas. Embora o *Flutter* tenha um menor tempo de vida, ele já conquistou grande parte da comunidade e ganhou notoriedade no

mercado, abaixo na figura 26 segue uma comparação realizada no Google Trends comparando os dois termos de pesquisa nos últimos dois anos no mundo todo.

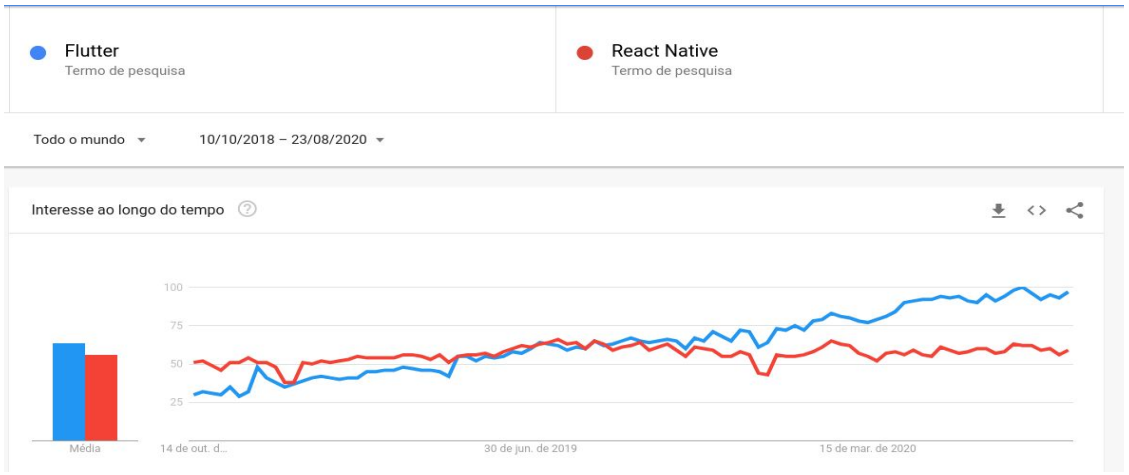


Figura 26: Comparação entre Flutter e React Native globalmente no Google Trends

É observado que *React Native* esteve a frente por um bom tempo, entretanto o flutter vem em uma linha crescente nos últimos meses, enquanto o *React* deu uma caída, para muitos essa caída se deu por conta do crescimento do *Flutter*. A seguir na figura 27 pode ser visto um gráfico com a comparação dos termos de pesquisa no Brasil nos últimos dois anos e novamente os dados se repetem em relação ao mundo, onde embora nos últimos dois anos o *React* tenha mantido a hegemonia o *Flutter* teve um crescimento considerável, onde bateu de frente e assumiu o domínio nos últimos meses.

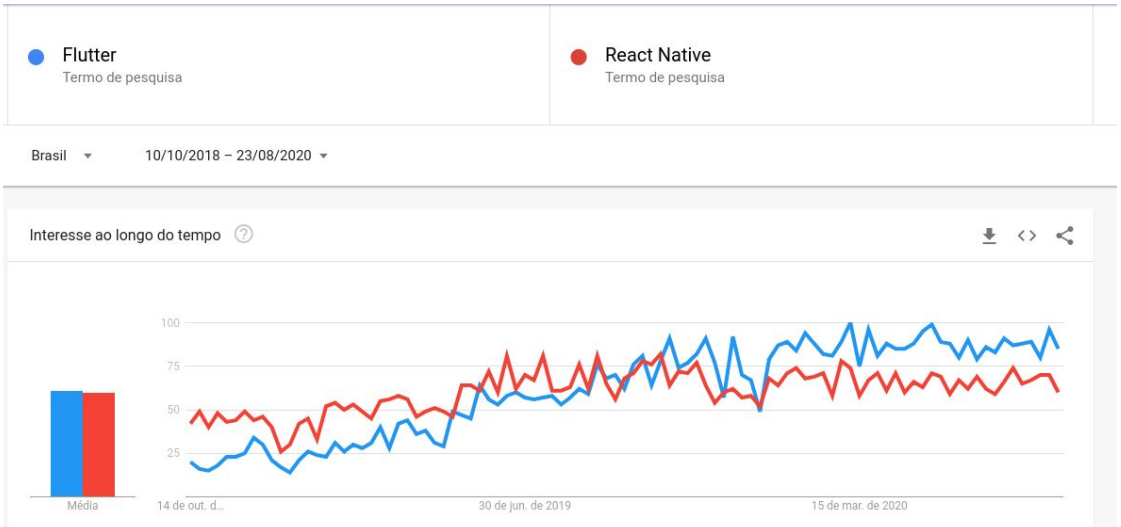


Figura 27: Comparação entre Flutter e React Native no Brasil com Google Trends

Quanto a comparação entre as duas ferramentas, todas as suas diferenças e semelhanças já foram descritas até o momento neste presente estudo.

4. ESTUDO DE CASO

Afim de demonstrar e verificar a eficiência, capacidade e produtividade do *Flutter* foi desenvolvido um aplicativo para compra de cartões de estacionamento, semelhante ao Zona Azul. O desenvolvimento foi realizado utilizando a *IDE Visual Studio Code*, com o banco de dados não relacional *Firebase* e *plugins* como o *Google Maps* e o *Geolocator*, dentre os seus concorrentes, nem sempre a integração com essas ferramentas não encontram problema, ou mesmo quando acontece precisa de um intermediário entre a aplicação e a *API*, como visto ao tentar utilizar o *Google Maps* durante o desenvolvimento de um aplicativo *Ionic*, na seguinte figura 28 é visualizado o diagrama de classe do banco de dados.

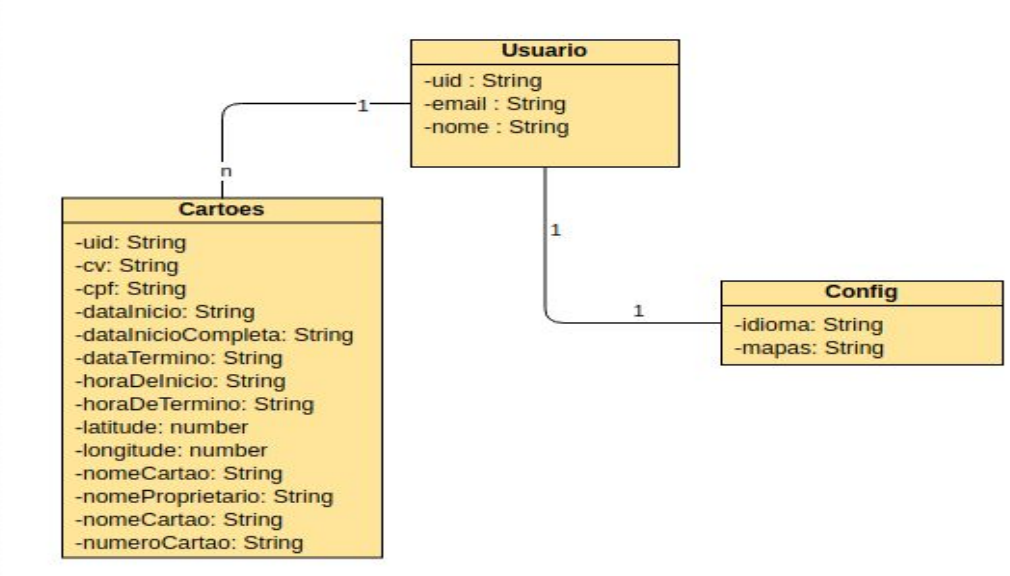


Figura 28: Diagrama de classe da aplicação

O código possuirá 4 camadas principais, onde, *views* será referente a parte visual do aplicativo, *controllers* relacionado às regras de negócio e a gerência de estado do aplicativo, *service* pertencente a parte de conexão com o banco de dados e *models* referente às classes utilizadas, sendo assim conseguiremos avaliar se a estrutura da aplicação irá se comportar de forma madura e a um padrão tão utilizado. Se o *Flutter* se

mostrar competente em todos os objetivos que serão buscados, então concluiremos que ele está apto e maduro o suficiente para o mercado, na figura 29 é ilustrado uma visão geral das ações do aplicativo com um diagrama de caso de uso.

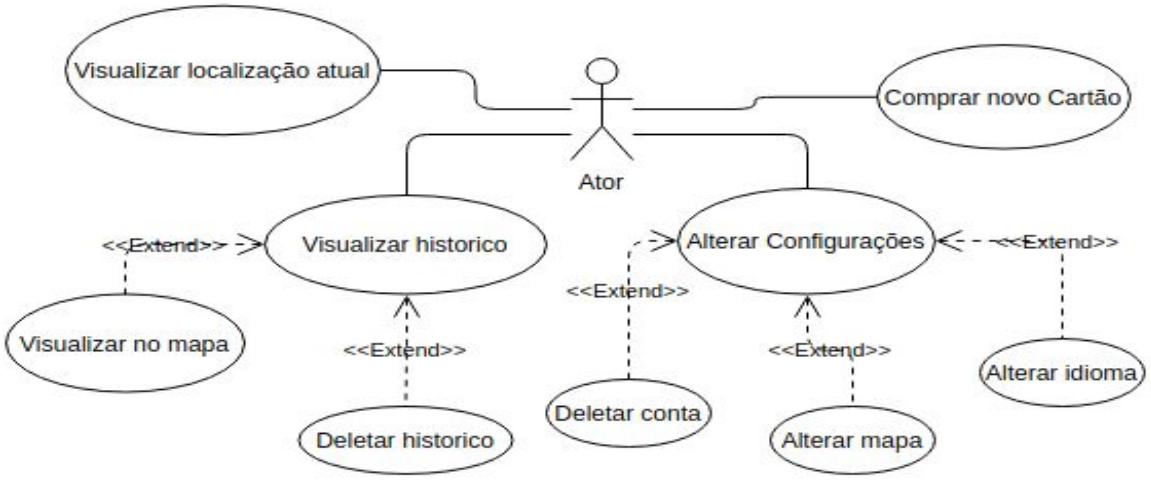
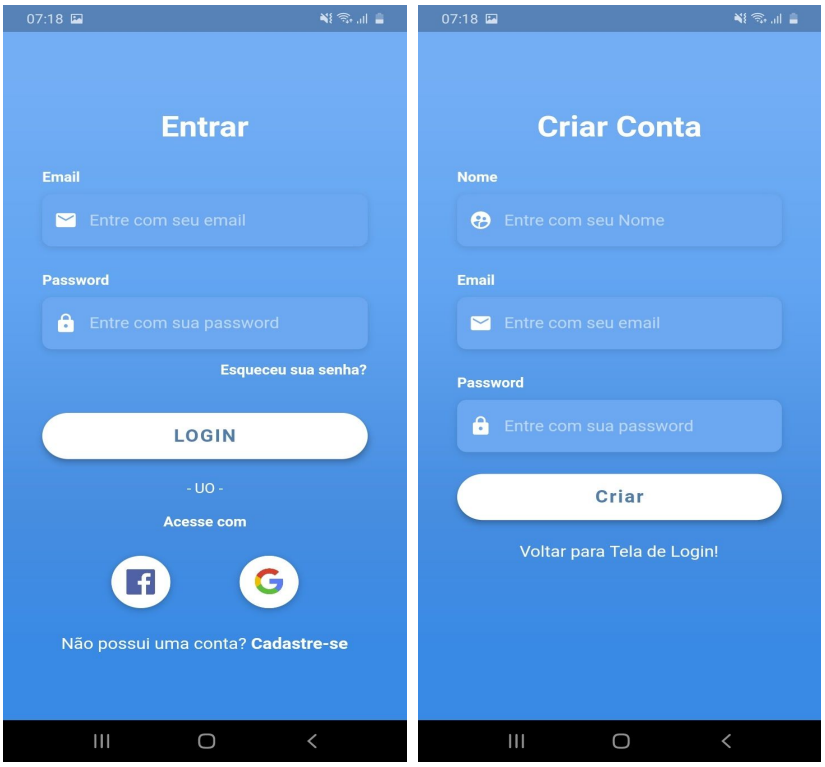


Figura 29: Caso de uso da aplicação

4.1. DESENVOLVIMENTO

Na figura 30 é apresentada a interface de *login* que também possui a opção de cadastro para novos usuários, utilizando um nome, email e uma senha. A parte visual destas interfaces foram desenvolvidas utilizando apenas código puro sem nenhum *package* específico que auxiliasse durante esta tarefa, porém o aplicativo faz uso de acesso automático, ou seja, é verificado se existe algum usuário autenticado no dispositivo, se existir ele será redirecionado para a interface inicial, tudo isso é realizado com um serviço desenvolvido no *service* que utiliza o *package FireAuth* para verificar na base de dados do *Firebase* se existe algum usuário autenticado, caso exista o *service* se comunica com o *controller* que neste exato momento emite através de um outro plugins chamada *Mobx* um alerta para a interface que ela deve alterar o seu estado.



a) Interfaces de login **b)** Interface de de cadastro

Figura 30: Interfaces de login e cadastro

Para realizar o acesso será necessário informar o e-mail e a senha, ao acessar será mostrado uma interface com o *Google Maps* indicando a localização atual do usuário. A partir deste momento o usuário terá acesso a um botão flutuante em seu lado direito. Este menu lateral será composto por opções como compra de cartões, histórico do usuário, localização atual, configurações e sair. A interface possui a utilização de *package* como o *Google Maps* para a renderização do mapa, *Geolocator* para recuperar a localização atual e *Fab Circular Menu*, que cria de forma simples um *Fab* botão de menu altamente configurado, tudo isso pode ser visto na figura 31.

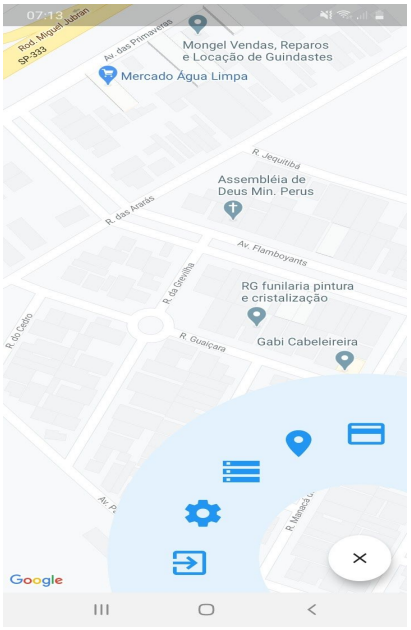


Figura 31: interface inicial

Para a compra de um novo cartão o usuário precisará acessar a primeira opção do menu, ao acessar esta opção primeiramente uma alerta aparecerá na interface informando que ao comprar um novo cartão o mesmo será automaticamente ativado (figura 32), este alerta foi desenvolvido com o uso do *package Awesome Dialog*, é personalizável e pode ser utilizado para informar, alertar ou enviar uma mensagem de sucesso ao usuário.

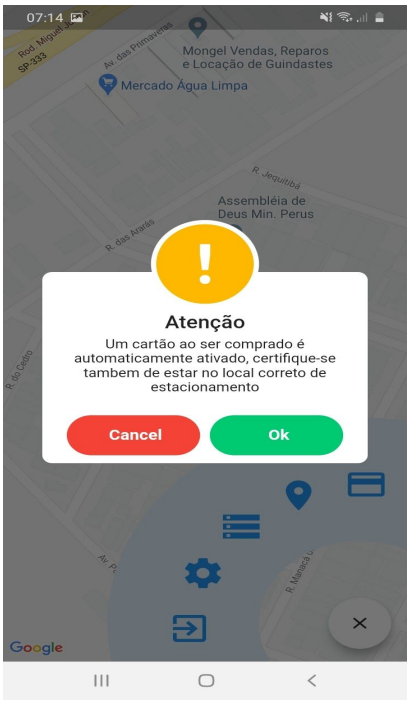
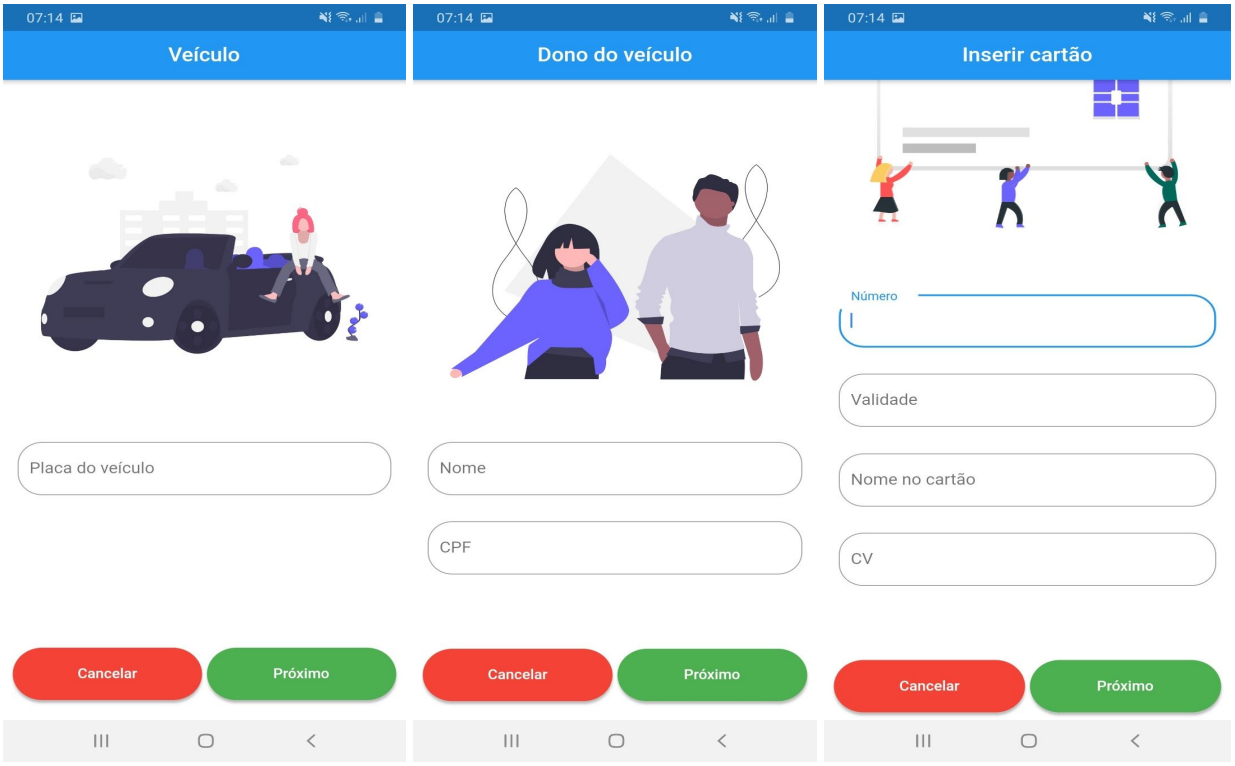


Figura 32: Alerta para início de compra

A seguir três novas interfaces serão exibidas para o processo de compra de um novo cartão, o usuário precisará preencher todas as informações, e por fim confirmar a compra, o processo pode ser visto na figura 33, para a parte visual das interfaces de compra não foi utilizado nenhum *package*.



a) Cadastro de placa do veículo **b)** Cadastro de nome e CPF **c)** Cadastro de cartão

Figura 33: interfaces para cadastro de dados da compra

Caso alguma informação não esteja preenchida de forma correta, uma alerta utilizado o Awesome Dialog será disparado na interface, e o processo voltará ao inicio porem com mensagens de erro em cada *input* para que tudo possa ser feito corretamente, todo esse processo pode ser observado na figura 34.

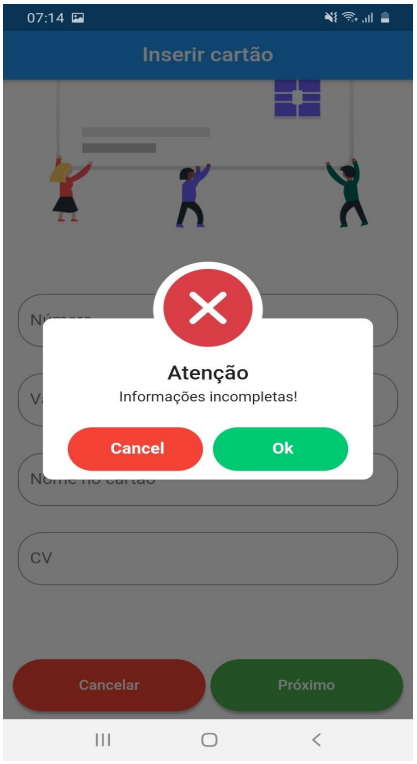
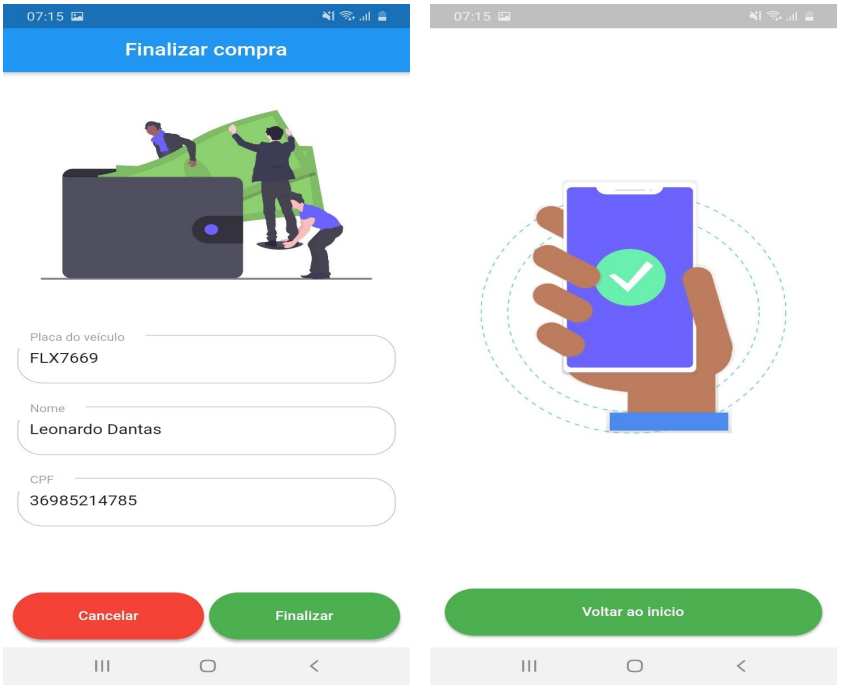


Figura 34: Alerta de informações incompletas

Se tudo estiver correto, desta vez será visualizado as informações para que assim o usuário possa confirmar e dar continuidade, para as duas interfaces foram desenvolvidas sem a utilização de nenhum *package* (figura 35).



a) Interface com dados obtidos **b)** Interface de sucesso

Figura 35: Interface de confirmação de dados e de compra bem sucedida

Caso tudo ocorra da forma correta, a interface inicial voltará a ser exibida, porém além de puxar a localização atual do usuário, ela também contará com uma marcação sinalizando onde o cartão foi comprado, como na figura 36. Para isto foi utilizado os *package* do *Google Maps* e *Geolocator*, de forma simples, no momento da compra as informações de compra do usuário são armazenadas no banco de dados, e toda vez que a interface inicial e reconstruída é feita uma verificação para saber se existe algum cartão ativo, vale lembrar que cada cartão possui apenas a duração de 1 hora.

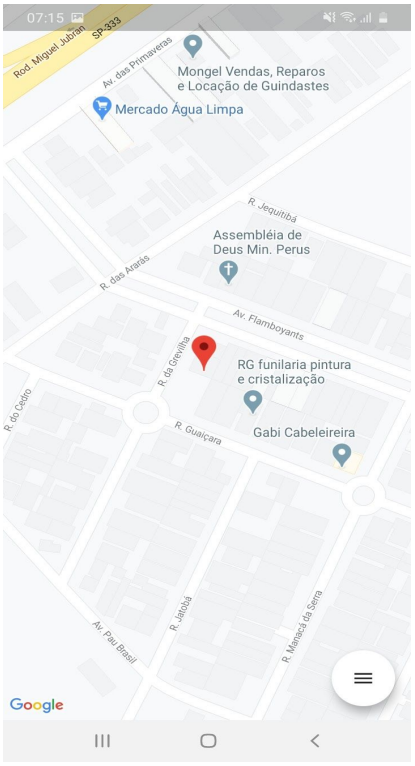


Figura 36: interface inicial com o ponto exato da compra

Caso o usuário queira visualizar se possui alguma marcação, basta acessar a segunda opção do menu e será redirecionado para ela, caso não possua nenhuma um alerta será exibido na interface com o *package Awesome Dialog*, como pode ser observado na figura 37.

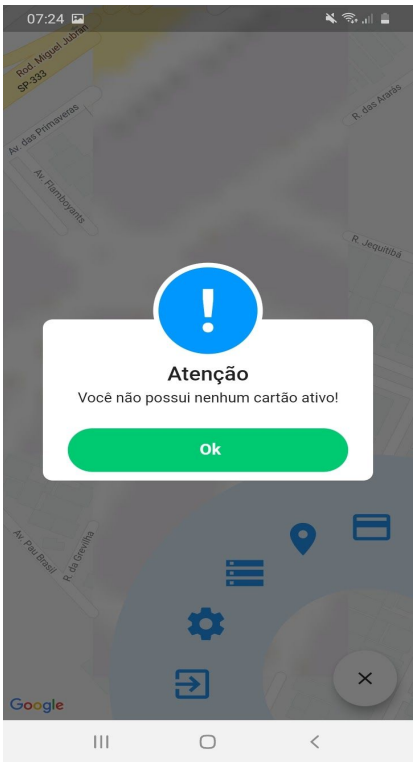
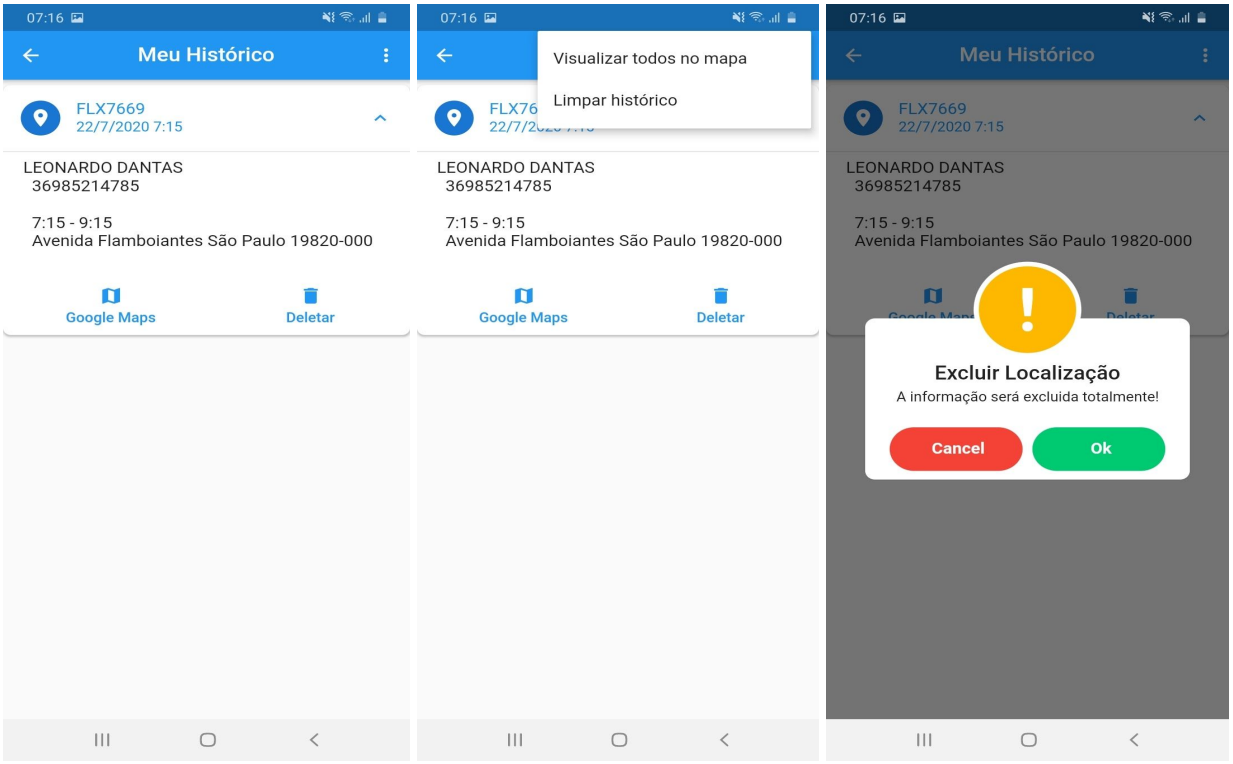


Figura 37: Alerta de não existir cartões ativos

O próximo item do menu se refere ao histórico do usuário atual, ao acessá-lo ele poderá visualizar informações sobre todas as localizações ao qual ele já comprou algum cartão, acessar essas informações e visualizá-las no *Google Maps*, apagar, ou mesmo visualizar todas as localizações de uma única vez, caso queira, ainda existe a opção para apagar todo o histórico. As informações no menu histórico são exibidas utilizando o *package Expansion Tile Card*, para a confirmação de ações importantes do usuário, foi utilizado o *package Awesome Dialog*, o processo é ilustrado na figura 38.



a) Interface de histórico **b)** Visualização menu PopUp **c)** Visualização Alert

Figura 38: Interface de histórico do usuário

Logo depois na figura 39 é visto as configurações do usuário, nesta parte ele poderá escolher quais são as preferências e até mesmo excluir a conta. Todas as interfaces foram codificadas sem a utilização de *package*, que continuaram sendo utilizados apenas para mostrar mensagens importantes para o usuário com o *Awersome Dialog*.

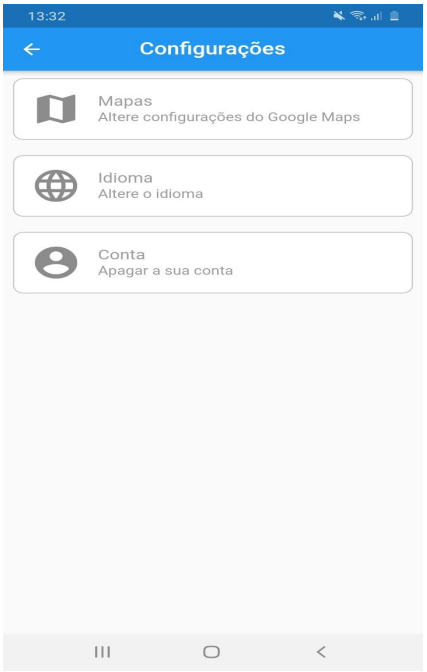
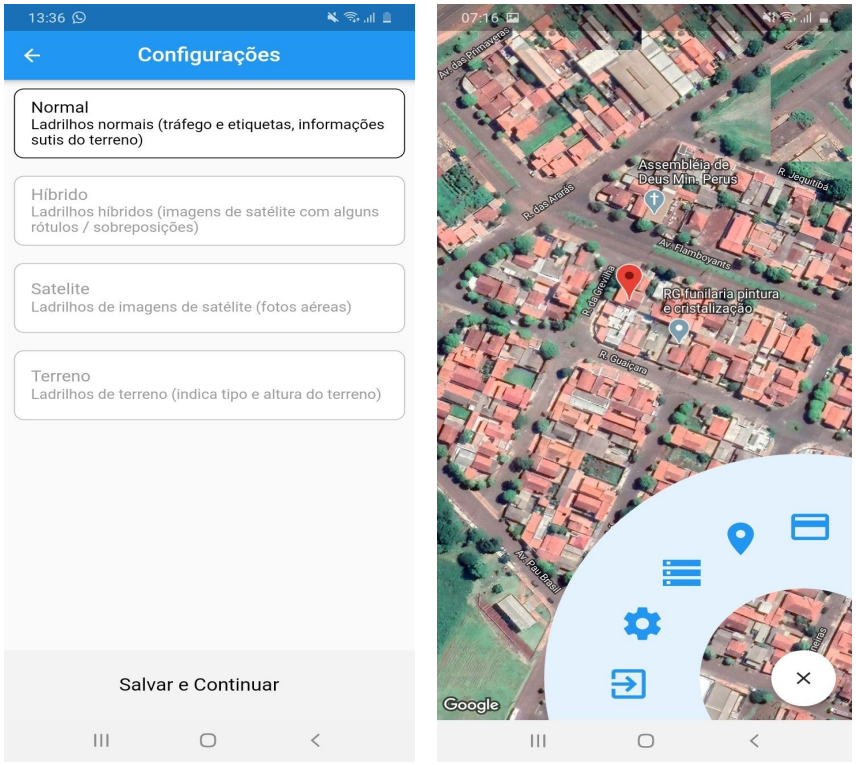


Figura 39: interfaces de configurações do usuário

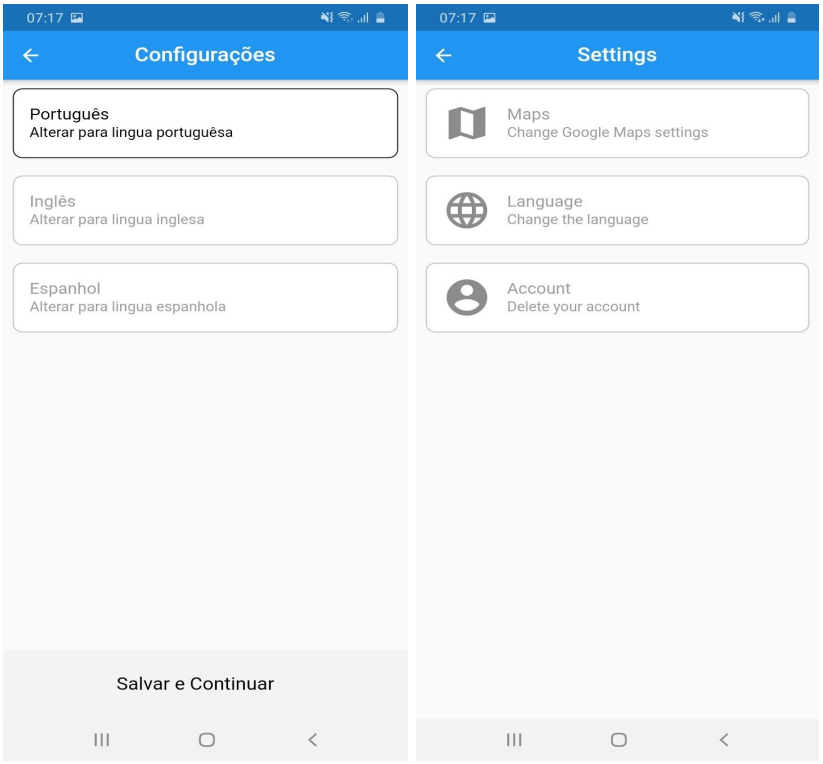
A primeiro opção se diz respeito à possibilidade de escolher qual o melhor tipo de mapa a ser visualizado, as opções disponíveis são normal, híbrido, satélite e terreno. Para alterar, basta selecionar a opção desejada e logo em seguida sobre o botão salvar e continuar. Ao escolher a opção Híbrido o mapa será alterado para o seguinte tipo.



a) Opções do mapa **b)** Demonstração de mapa Híbrido

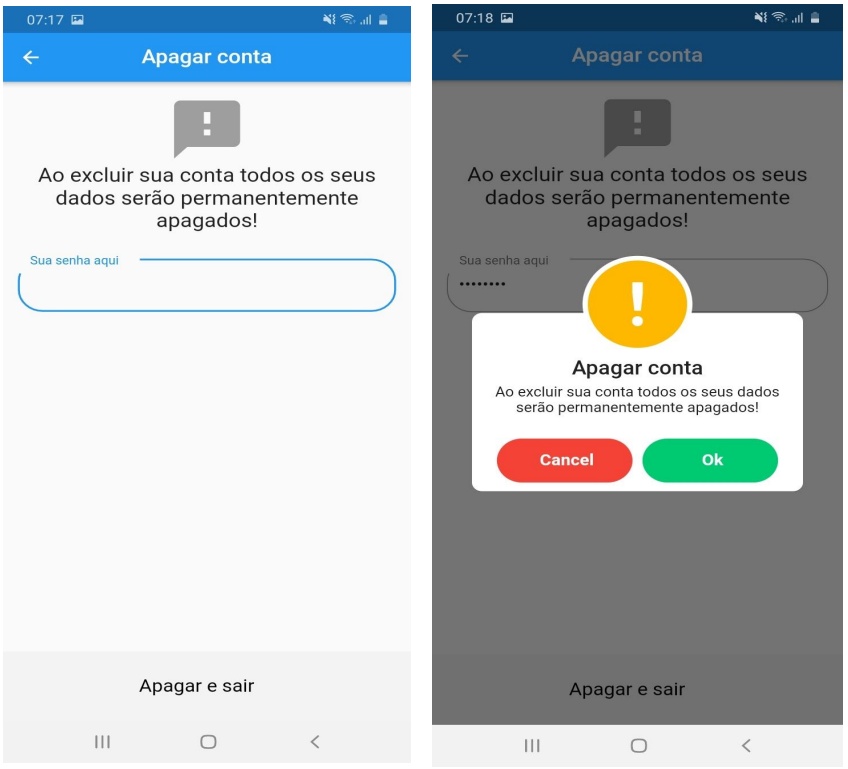
Figura 40: Interface de configurações do mapa

Outra opção disponível é a escolha do idioma utilizado no aplicativo, embora a opção padrão seja português do Brasil, o usuário pode escolher entre, espanhol ou inglês. Ao alterar o idioma e clicar sobre o botão salvar, as informações são gravadas e o idioma é trocado instantaneamente, como na figura 41.



a) Opções de linguagem **b)**Demonstração de interface com outra língua
Figura 41: Interface de configurações de linguagem

A última opção do menu de configurações se diz respeito a deleção da conta. Ao acessar esta parte, uma nova interface será aberta, informando sobre a situação e que o usuário perderá todos os seus dados, caso ainda deseje ele deverá informar uma senha e logo depois confirmar no alerta que aparecerá na interface. Após isso o usuário será deletado, e o aplicativo voltará para a interface de *login*, parte do processo pode ser visualizado na figura 42.



a) Interface para inserção da senha **b)** Alert para confirmação da ação
Figura 42: Interfaces para exclusão da conta

4.2. PRINCIPAIS *PACKAGES*.

Como já visto a utilização de *package* no *front-end* do aplicativo é mínima, a sua utilização se fez altamente forte e presente no coração da aplicação. Para isto foram utilizados três *package* que ajudaram a ter um maior controle sobre tudo o que acontece no aplicativo, a ter um reaproveitamento de código maior, produtividade e desempenho acelerados. Tudo isso foi possível graças ao *Mobx*, *Build Runner* e *GetIt*, que atuando em conjunto se auto completam. O primeiro ponto a se destacar aqui é a gerência de estado do aplicativo, isto nada mais é que o fato do sistema operacional renderizar a interface cada vez que uma mudança é detectada, como por exemplo um *input* sendo preenchido pelo usuário, a cada letra digitada a interface é refeita. Em uma operação simples isto pode passar de forma imperceptível, porém se tivermos uma interface que faz a utilização de mais recursos, isso pode acabar trazendo uma péssima experiência para o usuário. Sendo assim os três primeiros *packages* citados aqui são essenciais para controlar esse fluxo e permitir que

apenas a parte que exista interação seja renderizada. O primeiro deles *Mobx* é considerada por muitos uma biblioteca completa para o gerenciamento do estado da aplicação, fazendo uma ligação firme e reativa com os dados e a interface do usuário. Dentro do *Mobx* existem três conceitos principais, *Observables*, *Actions* e *Computed*, de maneira simples Observables está fortemente relacionado a variáveis ou campos que serão modificadas em determinada interface, ou seja, em uma interface apenas o conteúdo da *Observable* será renderizado, *Actions* está relacionada e ligada a uma função capaz de alterar o valor da *Observable*, e *Computed* refere-se a alguma variável e retorna o seu estado ou valor de forma reativa na interface. A seguir na figura 43 é visualizado um pequeno exemplo de classe com os conceitos de Mobx.

```
  @observable
  Set<Marker> _markers = {};
  @action
  setMarket(Set<Marker> marker) => _markers = marker;
  @computed
  Set<Marker> get getLocations {
    return _markers;
  }

  @observable
  Set<Marker> marcacaoUsuario;
  @action
  setMarcacaoUsuarioAtual(Set<Marker> m) => marcacaoUsuario = m;
  @computed
  Set<Marker> get getMarcacaoUsuario {
    return marcacaoUsuario;
  }
}
```

Figura 43: Exemplo de classe com MobX

O próximo *package* de extrema importância é o *Build Runner*, um gerador de código que pode auxiliar o desenvolvimento com *Mobx*. Como visto na figura anterior, é demonstrado um pequeno exemplo de classe utilizando os principais conceitos do *Mobx*, entretanto nem tudo é tão simples assim, pois apesar de estarmos usando as anotações corretas, este código não terá funcionalidade nenhuma. Acontece que para o devido funcionamento, outro código que é utilizado, e é esse o papel do *Build Runner*, gerar o código que vai ser devidamente utilizado pela aplicação. A classe que utilizar o *Mobx* deve ser iniciada como na figura 44.

```
import 'package:mobx/mobx.dart';

part 'user.g.dart';

class UserModel = UserModelBase with _$UserModel;

abstract class UserModelBase with Store {
```

Figura 44: Exemplo de utilização do MobX

Para que o processo tenha continuidade, precisa-se executar o seguinte comando, no terminal onde está localizado o projeto.

```
flutter packages pub run build_runner clean
```

Figura 45:E Comando para limpar qualquer classe MobX já existente

O comando da figura 45 será responsável por limpar qualquer código que já tenha sido gerado no projeto, impedindo código duplicados e que venha causar problemas mais tarde. Logo após, é executado a seguinte linha no terminal da pasta do projeto (figura 46).

```
flutter packages pub run build_runner build
```

Figura 46: E Comando para gerar novas classe MobX

Esse comando será responsável por gerar um novo código, que será encarregado de fazer com que as variáveis e funções sejam relativas, embora o programador fique responsável apenas pela parte mais simples, como na classe acima, ele não precisa fazer nenhuma alteração no código que foi gerado, mas deve executar os dois comandos de terminal visto anteriormente quando fizer alguma modificação na classe. A seguir na figura 47, é demonstrado como o código gerado foi totalmente diferente daquele que vimos no início.

```
final _$emailAtom = Atom(name: 'UserModelBase.email');

@override
String get email {
  _$emailAtom.reportRead();
  return super.email;
}

@override
set email(String value) {
  _$emailAtom.reportWrite(value, super.email, () {
    super.email = value;
  });
}
```

Figura 47: Exemplo de código gerado pelo MobX

Por último vem o *GetIt*, este *package* auxilia o programador a ter um maior controle de sua aplicação. O conceito é muito simples, ele permite instanciar o objeto logo ao iniciarmos o aplicativo, e durante a sua execução usar sempre a mesmo instância. Isso de fato é vantajoso pois permite guardar valores em variáveis específicas, ou mesmo utilizar a mesma função várias vezes sem precisar de uma nova instância, é claro que esse recurso deve ser utilizado com planejamento, pois pode ocorrer de sobrecarregar a aplicação em dado momento. Um bom exemplo de utilização é em uma classe que guarda as informações sobre o usuário do aplicativo, como seu email e seu nome, utilizando-o durante todo o aplicativo sem a necessidade de acessar o banco cada vez que precisamos desta informação. A seguir na figura 48 é demonstrado um pequeno exemplo de como é a utilização do *GetIt*.

```
Run | Debug
void main(){

  GetIt getIt = GetIt.I;
  getIt.registerSingleton<UserController>(UserController());
  runApp(MyApp());

}
```

Figura 48: Exemplo de utilização do GetIt

A instanciação acontece durante a inicialização do aplicativo. Para fazer uso dela em qualquer parte do aplicativo basta fazer como na figura 49.

```
final userController = GetIt.I<UserController>();
```

Figura 49: Exemplo utilização de classe pelo GetIt

4.3. *VIEWS*, *CONTROLLERS*, *SERVICES* E *MODELS*.

A seguir demonstraremos como acontece no código a funcionalidade de *login* do usuário, com isto conseguiremos ter uma visão geral de cada etapa, desde a digitação do usuário até o acesso ao banco *Firebase*. Para isso vamos focar em apenas um campo, o campo email. O primeiro passo a se fazer é a criação de uma classe, com o campo email e com as devidas funções do *Mobx* que poderão ser utilizadas durante esse procedimento (figura 50).

```
@observable
String email = "";
setEmail(String novoEmail) => email = novoEmail;
@computed
String get getEmail {
  return email;
}
```

Figura 50: Classe com MobX

O próximo passo é a criação de uma função no devido *service* que será a responsável por acessar o banco é realizar o *login*. Nesse passo também precisa-se utilizar um tipo de variável chamada *FirebaseAuth*, ela é nativa do *package Firebase Auth* e possui um conjunto de funções que fazem a ligação do usuário com o banco de dados, nesse caso a comunicação com o banco ocorrerá através da função *signInWithEmailAndPassword* que espera um email e uma senha, para assim validar o acesso do usuário, este código pode ser visualizado na figura 51.

```
class UserService {

  FirebaseAuth auth;
  login(String email, String pass) async {
    try {
      auth = FirebaseAuth.instance;
      await auth.signInWithEmailAndPassword(email: email, password: pass);
    } catch (e) {
      print(e);
    }
  }
```

Figura 51: Exemplo de serviço

Após o *Build Runner* gerar o código com o *Mobx,* será reproduzido um arquivo com a seguinte (figura 52) codificação representado a variável email da classe.

```
final _$emailAtom = Atom(name: 'UserModelBase.email');

@override
String get email {
  _$emailAtom.reportRead();
  return super.email;
}

@override
set email(String value) {
  _$emailAtom.reportWrite(value, super.email, () {
    super.email = value;
  });
}
```

Figura 52: Classe gerada com o MobX

Logo a seguir será codificado a função no *controller*. Está parte é muito importate, pois o *controller* que fará a ligação automaticamente com a interface do usuário e a lógica do negócio. A primeira coisa a ser feita é a instanciação do *model* do usuário, e do *service* referente ao mesmo, como na figura 53.

```
UserModel userModel = UserModel();
UserService userService = new UserService();

@action
login() async {
  try {
    await userService.login(userModel.email, userModel.senha);
  } catch (e) {
    print(e);
  }
}
```

Figura 53: Exemplo de controller

A parte lógica já está pronta, agora será recuperado a sua instância com o *GetIt* e assim executar a função login. A recuperação é feita conforme a figura 54.

```
final userController = GetIt.I<UserController>();
```

Figura 54: Recuperação de instância com GetIt

O campo email do *login* é um *widget* chamado *TextField*, esse tipo de *input* contém uma chamada de função conhecida como *onChanged*, onde a cada alteração no campo um evento é disparado, passando como parâmetro o valor digitado, o código pode ser visualizado na figura 55. Nesse campo é usado a instanciação do *UserController* que automaticamente já faz uma chamada para o *UserModel* instanciado dentro dele e que por fim chama a função *setEmail*, responsável por alterar o conteudo da variavel email dentro do *UserModel*.

```
child: TextField(
  keyboardType: TextInputType.emailAddress,
  onChanged: userController.userModel.setEmail
),
```

Figura 55: Recuperação do campo email

A execução da função de *login* se passara ao clicar sobre o botão *Login* do formulário, a função será executada dentro do *RaisedButton* que nada mais é que um botão que contém um evento chamada *onPressed*, este evento será executado ao clicar sobre o botão, com isso chamará a função login criada dentro do *UserController*.

5. CONCLUSÃO

Para este trabalho foi proposto um estudo minucioso sobre o desenvolvimento móvel e as principais ferramentas utilizadas no mercado, apresentando assim suas vantagens e desvantagens. Após isso apresentamos o *Flutter* como uma nova alternativa para um mercado que traz certa incertezas, e que está em constante crescimento. Por fim foi proposto e desenvolvido um aplicativo buscando explorar pontos específicos no *framework* o colocando a prova em pontos que podem trazer certas instabilidade quando utilizadas em tecnologias semelhantes, como o *Google Maps*, ou mesmo a ferramenta de geolocalização nativa do dispositivo. *Flutter* se mostrou uma ferramenta altamente produtiva para o desenvolvimento móvel, possui uma comunidade rica e que está em crescimento exponencial principalmente no Brasil. O ambiente de desenvolvimento é leve, pois apenas o *Visual Studio Code* é necessário durante a codificação, sendo que o próprio celular pode ser utilizado para a compilação do código em tempo real, com um *hot reload* poderoso observando as mudanças em tempo de codificação. Podemos dizer que o *framework* em si é totalmente fácil de aprender, e a linguagem *Dart* é semelhante as principais linguagens do mercado. A integração com os principais *packages* do mercado foi simples, pois possui tutoriais muito bem elaborados e grandes fóruns que auxiliam durante o processo. Por fim, demonstrou ter um desempenho favorável, e um tempo de desenvolvimento que surpreende. A tendência é que venha a dominar o mercado de desenvolvimento móvel tanto para *Android* como para *IOS*, portanto Flutter é uma excelente opção para o desenvolvimento pois com ele o programador terá alta produtividade, grande suporte da comunidade, um único código e um aplicativo com o desempenho que nada tem a perder quando comparado com aplicativos totalmente nativos.

REFERÊNCIAS

ANDRADE, Kleber. **Introdução a Linguagem de Programação Dart**.Medium. Disponível em<https://medium.com/flutter-comunidade-br/introdu%C3%A7%C3%A3o-a-linguagem-de-programa%C3%A7%C3%A3o-dart-b098e4e2a41e >. Acesso em: 18 jan. 2020.

BASSOTO, Helene. **Desenvolvimento de um protótipo de aplicativo móvel utilizando ferramenta multiplataforma**. 2014. 82. Trabalho de Conclusão de Curso – Computação – Universidade de Caxias do Sul, Rio Grande Do Sul, Caxias do Sul, 2014.

CISSOTO, Leandro. **Sete motivos para trocar Objective-C por Swift**. MacMagazine. Disponível em <https://macmagazine.uol.com.br/post/2015/05/22/7-motivos-para-trocar-objective-c-por-swift/>. Acesso em: 14 dez. 2019.

CORAZZA, Paulo Victor. **Um aplicativo multiplataforma desenvolvido com Flutter e NoSql para o cálculo de probabilidade de apendicite**. 2018. 64. Trabalho de Conclusão de Curso – Instituto de Informática – Universidade Federal do Rio Grande do Sul, Rio Grande Do Sul, Porto Alegre, 2018.

COSTA, Alves Suelhy. **Estudo comparativo das abordagens de desenvolvimento de aplicações para dispositivos móveis**. 2014. 72. Trabalho de Conclusão de Curso – Computação – Universidade Federal do Ceará, Ceará, Quixadá, 2014.

HUBSCH, Eduardo. **Uma Abordagem Comparativa do Desenvolvimento de Aplicações para dispositivos móveis**. 2012. 44p. Monografia – São Paulo (Faculdade Tecnologia de São Paulo), São Paulo, São Paulo, 2012.

JUNTUNEN, Antero; JALONEN, Eetu; LUUKKAINEN, Sakari. HTML5 **in Mobile Devices – Drivers and Restraints**. 46th Hawaii International Conference on System Sciences, 2013.

LIRA, Vinicio. **Batalha dos smartphones: aumento das vendas no Brasil, liderança da Samsung e invasão chinesa**. Promobit. Disponível em <https://www.promobit.com.br/blog/batalha-dos-smartphones-aumento-das-vendas-no-brasil-lideranca-da-samsung-e-invasao-chinesa/ >. Acesso em: 12 dez. 2019.

MADUREIRA, Daniel. **Aplicativo Nativo, Web App ou Aplicativo Híbrido**. Disponível em <https://usemobile.com.br/aplicativo-nativo-web-hibrido/>. Acesso em: 13 dez. 2019.

MANCINI, Cristiane. **O duvidoso extenso mercado de smartphones no Brasil. Negócios em Desenvolvimento**. Disponível em<http://www.negociosemmovimento.com.br/economia/o-duvidoso-extenso-mercado-de-smartphones-no-brasil/ >. Acesso em: 12 dez. 2019.

MATOS, Beatriz Rezener Dourado; SILVA, João Gabriel de Brito. **Estudo comparativo entre o desenvolvimento de aplicativos móveis utilizando abordagem nativa e multiplataforma**. 2016. 103. Trabalho de Conclusão de Curso – Universidade de Brasília, Brasília, Distrito Federal, 2016.

MENDES, Mariana Ribeiro; GARBAZZA, Itagildo Edmar; TERRA, Daniela Costa. **Desenvolvimento híbrido versus desenvolvimento nativo de aplicativos móveis**. In: VII Semana de Ciência e Tecnologia do IFMG campus Bambuí VII Jornada Científica e I Mostra de Extensão, 6, 2014. Bambuí, Brasil. Resumos. Bambuí: Semana de Ciência e Tecnologia do IFMG, 2014. Res. 6.

MULLER, Gabriel da Rosa; SOARES, Inali Wisniewski. **Estudo comparativo sobre ferramentas de desenvolvimento multiplataforma para aplicações móveis**. 2019. p.25. Trabalho de Conclusão de Curso – Ciência da Computação – Unicentro, Paraná, Guarapuava, 2019.

MUNIZ, Beto. **Flutter para desenvolvedores Javascript e/ou React Native**. Beto Muniz. Disponível em <https://betomuniz.com/blog/flutter-para-desenvolvedores-javascript-e-ou-react-native/ >. Acesso em: 16 out. 2019.

PIRES, Leonardo dos Santos. **Desempenho de aplicações móveis utilizando implementação nativa ou frameworks multiplataformas**. 2018. 38. Monografia – Engenharia da Computação – Universidade Federal de Pernambuco, Recife, Recife, 2018.

SANTANA, Fabiano. **Flutter: Porque você deveria apostar nesta tecnologia**. Medium. Disponível em <https://medium.com/tableless/flutter-porque-voc%C3%AA-deveria-apostar-nesta-tecnologia-94a510fffd18 >. Acesso em: 16 out. 2019.

SILVA, Ewerthon Patricio Araujo; SOTTO, Eder Carlos Salazar. **A utilização do Ionic Framework no desenvolvimento de aplicações híbridas em arquitetura orientada a serviços**. Interface Tecnológica, 15, 1, jun. , 2018, p.97 – p.108.

TRAÇA, Tiago Ribeiro. **Ferramenta de software para auxiliar a promoção de saúde em ambientes universitários**. 2018. p.64. Trabalho de Conclusão de Curso – Coordenadoria do Curso de Engenharia de Software – Universidade Tecnológica Federal do Paraná, Paraná, Dois Vizinhos, 2018.

VALENTE, Jonas. **Brasil é 5º país em ranking de uso diário de celulares no mundo**. Agência Brasil. Disponível em<http://agenciabrasil.ebc.com.br/geral/noticia/2019-01/brasil-foi-5o-pais-em-ranking-de-uso-diario-de-celulares-no-mundo >. Acesso em: 12 dez. 2019.