UM ESTUDO SOBRE O DESENVOLVIMENTO MOBILE UTILIZANDO FLUTTER[1]

Felipe Amaro Gradin [2]

Alexandre T. Lazzaretti [3]

RESUMO

Com o crescimento do mercado de dispositivos móveis, a procura por profissionais com conhecimento em desenvolvimento de aplicativos *mobile* vem crescendo continuamente. Este cenário fomenta o estudo sobre as tecnologias de desenvolvimento *mobile*. Os sistemas operacionais de maior utilização no mercado *mobile*, iOS e Android, têm características diferenciadas que, somadas aos diversos dispositivos existentes, produzem uma diversidade de variáveis para os desenvolvedores, gerando custos mais elevados para as empresas que desejam ter seus aplicativos rodando em multiplataforma. Uma variedade de soluções tem sido propostas para mitigar este problema, linguagens de desenvolvimento multiplataforma surgiram da possibilidade de redução de custos. Dentre as diversas tecnologias disponíveis no mercado atual, vem se evidenciando o Flutter. Ele é um *framework* desenvolvido pelo Google na linguagem de programação Dart. Possibilita o desenvolvimento de aplicações a partir da composição de *widgets*. A proposta deste trabalho é apresentar o emprego desta tecnologia em um estudo de caso que consiste no desenvolvimento de uma aplicação *mobile* codificada em Dart utilizando o *framework* Flutter e realizando a persistência dos dados com a plataforma Firebase, a fim de demonstrar suas particularidades e características. Para a aplicação, trata-se de um sistema de pedidos de um restaurante e contará com dois módulos, cliente e restaurante. O módulo cliente terá como recursos o cadastro de usuários bem como a realização de pedidos de produtos e o acompanhamento desses pedidos. O módulo restaurante possibilita ao administrador do restaurante a gerencia das categorias e produtos e o avanço do status dos pedidos. Tal ação envia uma notificação para o celular do cliente informando a alteração do status do pedido. Ao final, pode-se verificar que a ferramenta Flutter dispõe recursos e soluções para diversas necessidades relacionadas ao desenvolvimento de aplicativos para dispositivos móveis, destacamos os diversos *widgets* que abrangem desde o *layout* das telas até a gerencia de estado da aplicação.

Palavras-chave: Multiplataforma, Dart, framework, Firebase.

---

[1] Trabalho de Conclusão de Curso (TCC) apresentado ao Curso de Tecnologia em Sistemas para Internet do Instituto Federal Sul-rio-grandense, Campus Passo Fundo, como requisito parcial para a obtenção do título de Tecnólogo em Sistemas para Internet, na cidade de Passo Fundo, em 2019.

[2] Aluno do curso de Tecnologia em Sistemas para Internet pelo Instituto Federal Sul-rio-grandense, campus Passo Fundo/RS. E-mail: flipepf@gmail.com.

[3] Professor orientador, Instituto Federal Sul-rio-grandense, campus Passo Fundo/RS. E-mail: alexandre.lazzaretti@passofundo.ifsul.edu.br.

1 INTRODUÇÃO

A tecnologia móvel ou *mobile*, em especial o *smartphone* tornou-se parte integrante da vida moderna em todo o mundo. Segundo a International Data Corporation (IDC), em 2017 foram vendidos em torno de 47 milhões de smartphones só no Brasil (IDC, 2017). Já segundo a Fundação Getúlio Vargas (FGV), em sua 29ª pesquisa anual sobre o retrato do mercado de Tecnologia de Informação, demonstra que em 2018 o Brasil possuía mais smartphones ativos que pessoas, sendo 220 milhões de celulares ativos no país contra 207,6 milhões de habitantes, de acordo com os dados do IBGE. A pesquisa também aponta que em torno de 70% dos aparelhos utilizados para conexão com a internet em nosso país são smartphones (FGV, 2018).

De acordo com Danielsson (2016), com o aumento da demanda de mercado para o uso de dispositivos móveis, as empresas têm se mobilizado criando aplicativos para esses dispositivos, a fim de dispor seus serviços para seus clientes de uma forma mais acessível, gerando também o crescimento do mercado de desenvolvimento de aplicações para dispositivos móveis.

A App Annie, empresa especializada em monitorar o mercado de aplicativos, afirmou em um relatório sobre o estado do mercado no final de 2017 (App Annie, 2017), que o número de aplicativos baixados nas duas maiores lojas, iTunes App Store e Google Play, ultrapassou a casa dos 26 bilhões rendendo uma receita total de quase U$ 17 bilhões, o que representa um crescimento de 28% em relação aos números no ano anterior.

Os dados supracitados evidenciam que desenvolvimento de aplicativos móveis é uma área em ascensão no mercado a qual possibilita a criação de soluções para diversas áreas da vida cotidiana das pessoas, com um público-alvo potencial de bilhões de usuários. Tal ascensão torna o estudo sobre desenvolvimento de aplicativos móveis um foco desejável para desenvolvedores de software.

O presente trabalho mostra o uso do *framework* Flutter e sua aplicação no desenvolvimento de um aplicativo multiplataforma, demonstrando que a utilização de um *framework* pode agilizar e viabilizar o desenvolvimento de aplicações multiplataforma. Dentre os objetivos específicos estão: realizar um estudo do framework Flutter; estudar a utilização da plataforma Firebase como *backend* de aplicativos móveis; modelar o aplicativo a ser desenvolvido; desenvolver o aplicativo utilizando Flutter.

O trabalho está organizado em mais três seções: na primeira são apresentadas as principais tecnologias para o desenvolvimento de aplicações móveis, que são: tecnologia nativa

das plataformas e tecnologia multiplataforma que se subdivide em: abordagem web, abordagem híbrida; compilação cruzada e nativa de script. Na segunda seção são apresentadas as características da tecnologia multiplataforma através do *framework* Flutter, aplicada no estudo de caso proposto neste trabalho, por fim na última seção, apresenta-se a aplicação do estudo de caso no desenvolvimento de uma aplicação multiplataforma para dispositivos móveis.

2 REFERENCIAL BIBLIOGRÁFICO

2.1 SISTEMAS OPERACIONAIS PARA DISPOSITIVOS MÓVEIS

Atualmente os principais sistemas operacionais para dispositivos móveis são, o iOS da Apple e o Android da Google. De acordo com relatório da IDC (2018), lançado no final de 2018 acerca da participação das empresas no mercado de *smartphones*, o Android domina o mercado com uma fatia de 85,1%. Seu principal concorrente iOS conta atualmente com uma fatia de 14,9%. Com participação minúscula, outros sistemas representam menos de 0,1% do mercado.

Segundo Lecheta (2015), o Android é um ambiente de *software* desenvolvido principalmente para dispositivos móveis como *smartphones* e *tablets*, porém pode ser utilizado em outros dispositivos como *notebooks*, TVs (Android TV), relógios (Android Wear), painéis de automóveis (Android Auto), etc. Ao associar sua conta do Google a um dispositivo, o usuário passa a ter acesso aos seus dados nos aplicativos padrões do sistema. Ele conta com o Google Play Store, que é uma loja virtual onde o usuário pode baixar e instalar outros aplicativos desenvolvidos por terceiros.

Conforme Lelli e Bostrand (2016), ao contrário do iOS, vários fabricantes como Samsung, Motorolla, Nexus, LG e outros, implementam o sistema operacional Android em seus dispositivos. Sua história teve início em 2003 com a Android Inc, uma pequena empresa que em 2005 foi comprada pela Google e que desde então tem desenvolvido e mantido o sistema como parte do Android Open Source Project.

Segundo Milani (2012), o iPhone Operating System (iOS), é um sistema operacional desenvolvido pela Apple para uso exclusivo em produtos de *hardware* da empresa. Baseado no Mac OS X, foi revelado em 2007 durante o lançamento do iPhone e posteriormente tornou-se o sistema operacional padrão para os dispositivos móveis da Apple, como o iPad e a Apple TV.

Para utilizar o iOS em um dispositivo Apple é necessário ter um ID Apple. Com essa conta é possível o acesso aos serviços da empresa como o App Store, a loja virtual de aplicativos

exclusivos da Apple, contêm tanto aplicativos padrões quanto aplicativos desenvolvidos por terceiros (LECHETA, 2015).

2.2 DESENVOLVIMENTO DE APLICAÇÕES MÓVEIS

A principais abordagens no desenvolvimento de um aplicativo para dispositivos móveis são: desenvolvimento nativo e desenvolvimento multiplataforma. Cada uma das implementações tem suas vantagens e desvantagens as quais devem ser consideradas antes do desenvolvimento de uma aplicação.

2.2.1 Desenvolvimento Nativo

Segundo Silva e Santos (2014), o desenvolvimento nativo utiliza linguagens de programação específicas de cada plataforma, ou seja, na linguagem que o próprio fabricante do sistema operacional escolheu ou projetou para desenvolver aplicativos. Também existem ambientes de desenvolvimento integrados (IDEs) específicos usados para desenvolver aplicativos para cada plataforma. Essas ferramentas são, geralmente, altamente recomendadas para permitir uma boa depuração, teste e empacotamento dos aplicativos.

Ainda de acordo com os autores acima citados, aplicações nativas podem ser baixadas, instaladas e vendidas em lojas de aplicativos como o Google Play e App Store.

Conforme Lelli e Bostrand (2016), os aplicativos desenvolvidos de forma nativa normalmente geram a melhor experiência de uso, pois são baseados no comportamento e aparência padrão da plataforma de destino. Os autores acrescentam que uma aplicação nativa explora de forma mais eficiente os recursos de hardware resultando geralmente em um desempenho melhor das aplicações desenvolvidas com outras tecnologias.

A desvantagem do desenvolvimento nativo, segundo Carlström (2016), reside no aumento dos custos de desenvolvimento para a empresa que deseja atender Android e iOS. Neste cenário geralmente é preciso contratar um time de desenvolvedores para cada plataforma, tendo como resultado final duas aplicações distintas, com códigos, manutenções e atualizações distintas. O tempo de desenvolvimento é outro fator que tende a aumentar no desenvolvimento nativo.

Em relação ao desenvolvimento nativo para Android, de acordo com Ableson et al. (2012), enquanto os componentes do Android são escritos em C ou C++, as aplicações de usuário são escritas em Java ou Kotlin, que após o final de 2017 também passou a ser considerado como linguagem oficial para o Android.

Segundo Hansson e Vidhall (2016), o ambiente de desenvolvimento de aplicativos Android nativos inclui o Android SDK, bem como o Java JDK. Existem diferentes IDEs que podem ser usados para desenvolver aplicativos para Android, mas o Google recomenda o Android Studio. Para depurar os aplicativos, o Google fornece um emulador que pode simular qualquer dispositivo Android, mas os aplicativos também podem ser instalados e executados em um dispositivo Android físico. Os autores acrescentam que o Google também fornece uma ferramenta para análise e depuração de aplicativos chamada Device Monitor que inclui diversas ferramentas que podem ser usadas, por exemplo, para medir o uso da CPU, uso de memória e tempo de resposta, etc.

Já em relação ao desenvolvimento nativo para iOS, ainda segundo Hansson e Vidhall (2016), existem requisitos que precisam ser preenchidos para poder desenvolver aplicativos para iOS. O ambiente de desenvolvimento inclui um computador Mac com o OS X 10.10 ou posterior e o Apple IDE Xcode instalado. Incluído no Xcode está o SDK do iOS, também necessário para o desenvolvimento. Para poder testar seus aplicativos em um aparelho e disponibilizá-los na App Store é preciso uma conta de *developer* da Apple ativada ao custo de 99 dólares anuais.

As linguagens de desenvolvimento são Objective C ou a nova linguagem da Apple, Swift, lançada em 2014. Uma combinação de ambas as linguagens também pode ser usada se o Swift for projetado para interoperabilidade com o Objective C. Embora a grande maioria dos desenvolvedores para iOS ainda estarem mais familiarizados com o Objetive C, aos poucos o Swift vem crescendo e gerando interesse dos desenvolvedores (CARLSTRÖM, 2016).

Hansson e Vidhall (2016) apontam que, por mais que seja possível testar os aplicativos diretamente em um dispositivo físico, o pacote Xcode inclui um emulador que pode simular um aplicativo em qualquer dispositivo iOS, entretanto, existem algumas restrições em relação aos recursos que podem ser ativados no simulador. Outro recurso que acompanha o Xcode é a ferramenta de análise e teste de desempenho Instruments. Com ela um aplicativo pode ser testado em termos de uso da CPU, uso de memória e desempenho da interface do usuário.

2.2.2 Desenvolvimento Multiplataforma

Tecnologias de desenvolvimento multiplataforma referem-se a qualquer implementação que não faz uso APIs nativas e que podem gerar aplicações que, com um único código, podem ser implementadas em mais de uma plataforma. Nos últimos anos, diferentes técnicas e *frameworks* surgiram para fornecer uma solução para a criação de uma aplicação

multiplataforma (DANIELSSON, 2016). Atualmente podem ser divididas em 4 categorias: aplicações *web*, aplicações híbridas, de compilação cruzada e nativas de script.

Dos benefícios que estas ferramentas compartilham em comum, Silva e Santos (2014) destacam: redução da complexidade; redução de código; redução do tempo de desenvolvimento e custo de manutenção; maior facilidade no desenvolvimento e aumento de participação de mercado. Porém, apesar dos muitos benefícios, o desenvolvimento multiplataforma também tem muitas desvantagens e as mais notáveis são o desempenho e a interface de usuário não usual em comparação com os aplicativos nativos conforme Fayzullaev (2018).

Conforme Litayem et al (2015), o desenvolvimento *web* utiliza as tecnologias comuns que rodam em qualquer navegador: HTML, CSS e JavaScript. Sendo assim, podem funcionar em computadores *desktop* e também em qualquer dispositivo móvel que tenha um navegador.

Tecnologias de desenvolvimento híbrido, procuram unir os benefícios de aplicações nativas e aplicações *web*. Na definição de Litayem (2015), aplicações híbridas são construídas com tecnologias de desenvolvimento *web*, mas são executadas em um contêiner nativo que fornece acesso aos recursos do dispositivo (câmera, GPS, sensores, etc.). Com a ajuda de estruturas, os elementos nativos da interface do usuário podem ser recriados com HTML e CSS. A comunicação entre o aplicativo da *web* e o híbrido geralmente é feita por meio de uma API JavaScript. Ionic, Adobe PhoneGap e Sencha Touch são exemplos de ferramentas de desenvolvimento híbrido.

Na definição de Hansson e Vidhal (2016), tecnologias de compilação cruzada utilizam uma linguagem de programação convencional não nativa para o desenvolvimento de aplicativos que depois são compilados em uma aplicação completamente nativa usando um compilador cruzado ou *crosscompiler*. Xamarin, Titanium, Qt, Kivy e o próprio Flutter são exemplos de *frameworks* de compilação cruzada.

Hansson e Vidhal (2016) definem os aplicativos nativos de script, ou interpretados, como aplicativos que usam um interpretador, que é empacotado com o aplicativo no dispositivo móvel que executa o código para fazer chamadas para APIs nativas. Essa abordagem pode usar qualquer linguagem de script que possa ser interpretada em um dispositivo, mas a maioria das novas estruturas usa o JavaScript como linguagem principal. Exemplos de estruturas de script nativas são, Native Script e React Native.

2.3 FLUTTER

O Google define o Flutter como: “Um kit de ferramentas de interface do usuário para criar aplicativos belos e compilados nativamente para dispositivos móveis, *web* e *desktop* a partir de uma única base de código” (GOOGLE, 2019a). É uma estrutura de desenvolvimento nova, foi apresentado ao público em maio de 2017 e teve sua primeira versão estável lançada em dezembro de 2018.

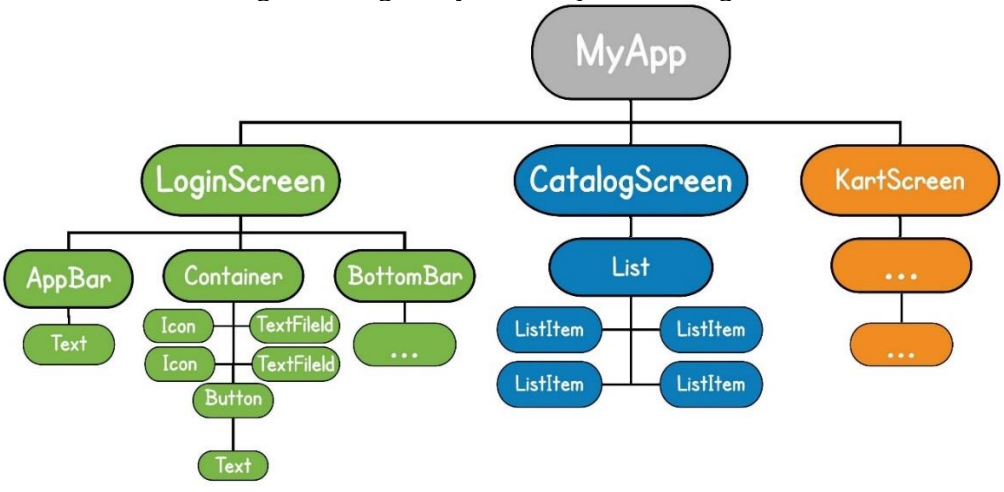
Em sua documentação, O Google destaca que o Flutter inclui uma estrutura moderna de programação reativa, um mecanismo de renderização 2D, *widgets* prontos e ferramentas de desenvolvimento. Esses componentes trabalham juntos para ajudar a projetar, criar, testar e depurar aplicativos (GOOGLE, 2019b).

De acordo com Wu (2018), devido ao Flutter possuir seu próprio mecanismo de renderização, possibilita aos desenvolvedores que optarem pelo Flutter a criação de aplicativos com o desempenho comparável ao desempenho que os aplicativos nativos podem ter.

2.3.1 Widget

O *widget* pode ser considerado o elemento mais importante em uma aplicação Flutter pois tudo no Flutter é considerado um *widget* incluindo o próprio aplicativo. Os *widgets* são os blocos de construção básicos de uma interface de usuário. Eles são uma declaração de tudo o que é desenhado na tela, incluído sua estrutura, aparência, conteúdo, posição e comportamento (GOOGLE, 2019b). Os *widgets* são organizados hierarquicamente de modo que todos os *widgets* são aninhados em um widget raiz conforme ilustrado na Figura 1.

Figura 1: Organização hierárquica de Widgets



Fonte: Do autor, adaptado de https://flutter.dev/docs/resources/technical-overview

Existem dois tipos de *widgets*: s*tateless* w*idget* e s*tateful* w*idget*. De acordo com Dagne (2019) um *stateful widget* vem com um objeto correspondente que representa o estado. São normalmente usados para o cumprimento de atividades assíncronas ou para situações que o estado vá mudar em algum momento no ciclo de vida do widget como por exemplo em uma interação do usuário. Já o *stateless widget* é mais simples pois não muda seu estado. São indicados para criação de interfaces que não serão modificadas por eventos futuros. Esse padrão de *design* permite que o próprio *widget* permaneça imutável, evitando que a estrutura renderize novamente a visualização do *widget* com frequência.

2.3.2 Dart

O Flutter e todas as aplicações desenvolvidas com ele são escritas em Dart. É uma linguagem de programação desenvolvida e mantida pelo Google. De sintaxe similar ao Java, procura ser uma mescla das boas práticas das linguagens existentes (WU, 2018).

Um dos recursos do Flutter que é fornecido pelo Dart é o recurso do *hot reload* que permite aos desenvolvedores ver mudanças em simuladores e emuladores sem reexecutar o aplicativo. Esse é um resultado do Dart VM (*Virtual Machine*) e seus diferentes modos de operação para as compilações de depuração (JIT *Just-in-Time*) e liberação (AOT - *Ahead of Time*) (DAGNE, 2019).

2.4 FIREBASE

Firebase é uma plataforma de desenvolvimento *mobile* e *web* adquirida pela Google em 2014. Com foco em ser um *back-end* completo e de fácil usabilidade, disponibiliza diversos serviços diferentes que auxiliam no desenvolvimento e gerenciamento de aplicativos (FIREBASE, 2019).

Dentre as diversos serviços disponibilizados pelo *Firebase*, para este trabalho foram utilizados: Firebase Authentication - serviço de autenticação de usuário, Firebase Storage - armazenamento de arquivos, Firebase Cloud Firestore - um banco de dados NoSQL baseado em nuvem aonde os dados são armazenados em documentos JSON[4] e todos os clientes conectados compartilham uma instância, recebendo automaticamente atualizações com os dados mais recentes, e por fim o Firebase Cloud Messaging - uma solução de mensagens entre plataformas que permite o envio de notificações sem custo (FIREBASE, 2019).

[4] JavaScript Object Notation, é um formato compacto, de padrão aberto independente, de troca de dados entre sistemas

3 RESULTADOS

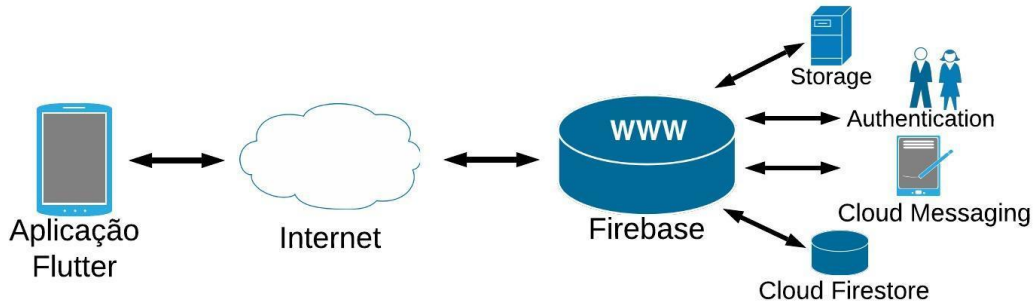
A aplicação desenvolvida foi baseada nas necessidades de um sistema de pedidos de um restaurante de comida japonesa. Através do aplicativo os clientes do restaurante podem ter acesso a uma listagem de produtos comercializados organizados em categorias. Somente após a criação de uma conta dentro da aplicação os clientes podem adicionar ou remover os produtos ao pedido. Após definir quais produtos e em quais quantidades quer, o cliente finaliza o pedido e a partir disso poderá acompanhar o andamento do mesmo.

No módulo restaurante é possível manter o cadastro dos produtos e categorias além da listagem de todos os pedidos realizados pelos clientes. A partir desta listagem é realizado o controle do status do pedido. Cada alteração efetuada no status do pedido gera uma notificação para o cliente.

3.1 ARQUITETURA DA APLICAÇÃO

A Figura 2 apresenta uma ilustração sobre a arquitetura da aplicação durante sua execução, onde as requisições de dados são realizadas através da internet com protocolo padrão *web* ao Firebase. No Firebase, são utilizados quatro serviços que a plataforma proporciona: o Cloud Firestore para armazenar os registros, o Storage para armazenar as imagens, o Authentication que controla toda a parte de criação, validação e autenticação dos usuários da aplicação e por fim o Firebase Cloud Messaging para o envio de notificações.

Figura 2: Comportamento da aplicação

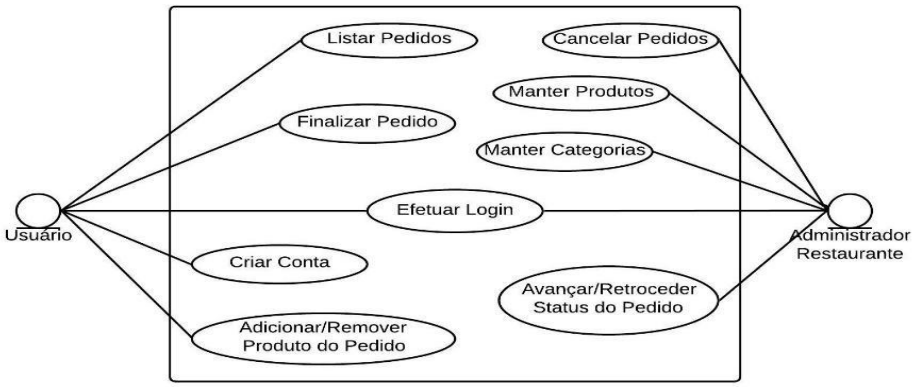


Fonte: Do autor.

3.2 ESTUDO DE CASO

A Figura 3 ilustra o diagrama de caso de uso que tem por objetivo demonstrar a interação do sistema com o usuário. Permite uma visualização resumida das ações que ocorrerão no sistema e dos atores envolvidos em cada caso de uso.

Figura 3: Diagrama de caso de uso

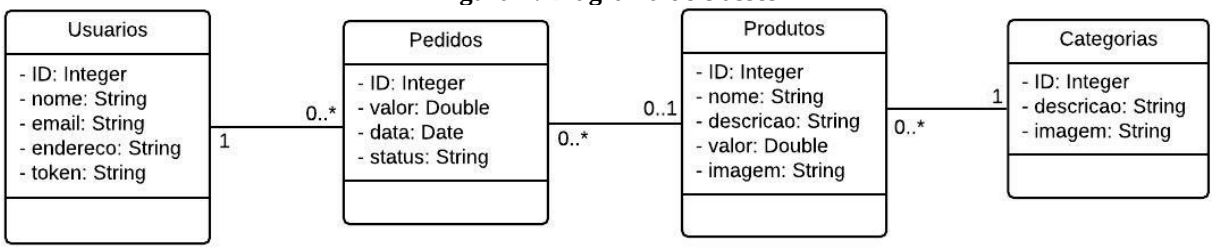


Fonte: Do autor.

3.3 MODELAGEM DOS DADOS

A Figura 4 ilustra a modelagem lógica dos dados. Utilizou-se este tipo de diagrama para descrever as informações necessárias a persistência dos dados na aplicação.

Figura 4: Diagrama de classes



Fonte: Do autor.

A modelagem dos dados representada acima pode ser traduzida como um documento JSON, que é o formato de dados armazenados no Firebase Cloud Firestore, um exemplo é ilustrado na Figura 5.

Figura 5: Exemplo de documento JSON


```
"produtos": { "idCategoria": { "imagem": "https:/firebasestorage.googleapis.com/%2F5%20.png", "nome": "Pratos Quentes",
                               "items": { "61rfZMsd": { "descricao": "Frango c/ molho", "nome": "Frango Teriaki", "valor": 18.99,
                                                        "imagem": "https:/firebasestorage.googleapis.com/%12F%20.png" }
                                        }
                             }
            },
"usuários": { "KslRoLPZ": { "nome": "fulano", "email": "fulano@email.com", "endereço": "Rua 20 de setembro 460/402 - Centro",
                            "pedidos": {"idPedido1": "idPedido1", "idPedido2": "idPedido2" }
                          }
            },
"pedidos": { "X7OJs2HL": { "idUsuario": "KslRoLPZ", "desconto": 0, "valorEntrega": 5.00, "valorProdutos": 23.98, "valorTotal": 28.98,
                           "0": { "idProduto": "61rfZMsd", "descrição": "Frango ao molho fugi", "nome": "Frango Teriaki",
                                  "quantidade": 2, "categoria": "Pratos Quentes", "valor": 18.99  },
                           "1": { "idProduto": "DkRuLYrC", "descrição": "Refrigerante", "nome": "Coca Cola", "quantidade": 1,
                                  "categoria": "Bebidas", "valor": 4.99 }

                         }
            }
```


Fonte: Do autor.

3.4 INTERFACES DA APLICAÇÃO

Nesta seção, são ilustradas as interfaces da aplicação desenvolvida. Inicialmente não é preciso estar logado para visualizar os produtos organizados por categorias. A Figura 6 ilustra a sequência de telas de navegação disponibilizadas ao usuário a partir no menu principal (Figura 6, tela A) até a visualização dos detalhes de um produto (Figura 6, tela D) seguindo as seleções: produtos (Figura 6, tela A), pratos quentes (Figura 6, tela B) e *yaksoba* vegetariano (Figura 6, tela C).

Figura 6: Sequência de telas do menu até os detalhes de um produto.

Fonte: Do autor

Ao selecionar o botão “Adicionar ao Carrinho” o aplicativo redireciona o usuário a tela de *login* (Figura 7, tela E), nela ele pode se logar através de uma conta existente ou criar uma nova conta (Figura 7, tela F). Após efetuado *login*, o usuário é redirecionado a tela do carrinho

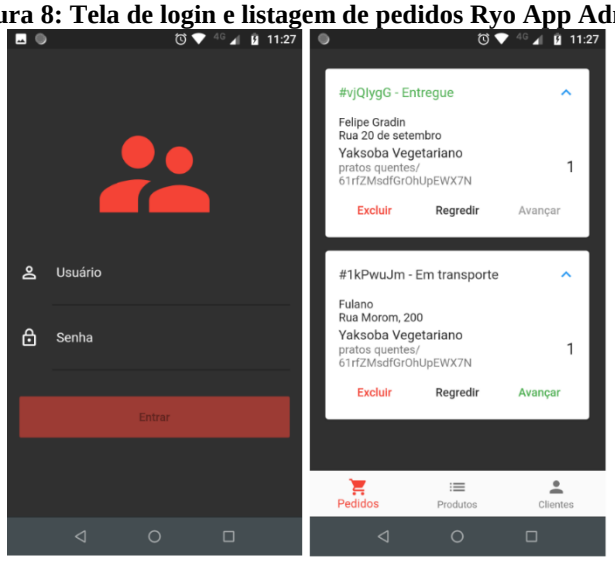
(Figura 7, tela G), nesta tela é possível ajustar a quantidade desejada do produto ou removê-lo do carrinho. Ao finalizar o pedido o usuário é redirecionado a tela de histórico de pedidos (Figura 7, tela H), nesta tela é possível visualizar o histórico de pedidos realizados e acompanhar o status desses pedidos.

Figura 7: Sequência de telas para realizar e acompanhar um pedido

Fonte: Do autor

No módulo restaurante, após efetuar *login* o usuário tem acesso as informações dos pedidos realizados pelos clientes, a partir desta lista, o usuário pode excluir um pedido além de avançar ou regredir seu status. A Figura 8, ilustra as telas do módulo restaurante.

Figura 8: Tela de login e listagem de pedidos Ryo App Admin



Fonte: Do autor

3.5 PRINCIPAIS ASPECTOS DO DESENVOLVIMENTO

Tutoriais sobre como instalar, configurar e iniciar um projeto Flutter podem ser encontrados com facilidade na internet assim como conteúdos contendo os procedimentos iniciais para comunicar um aplicativo Flutter com os serviços fornecidos pelo Firebase[5].

Para o desenvolvimento da aplicação foi utilizado o Android Studio como IDE de programação. Os testes para a plataforma iOS foram executados somente com o uso de emuladores, uma vez que, para gerar e testar o aplicativo para a plataforma, se faz necessário ter um equipamento da Apple.

Um aspecto do desenvolvimento no Flutter foi o uso de uma série de APIS do Firebase. Para utilizar essas ferramentas, é preciso adicionar as dependências no arquivo de nome *pubspec.yaml*. O Quadro 1, mostra o trecho do código em que estas dependências são adicionadas. Nesta seção são apresentados os exemplos de uso de duas delas: o Cloud Firestore (Quadro 1, linha 5) e após o Firebase Cloud Messaging (Quadro 1, linha7)

Quadro 1: Trecho de código do arquivo pubspec.yaml

```
1  dependencies:
2    flutter:
3      sdk: flutter
4    cupertino_icons: ^0.1.2
5    cloud_firestore: ^0.12.9
6    firebase_auth: ^0.11.1+12
7    firebase_messaging: 5.1.4
```

Fonte: Do autor

O conteúdo das linhas 5 a 8 foram adicionadas ao código já criado na construção do projeto. O código que antecede ":" (dois pontos) acrescenta ao projeto a dependência desejada e o que vem depois indica a versão que será utilizada.

O exemplo do uso do Cloud Firestore pode ser verificado no código da tela que abre quando o usuário seleciona uma categoria e é redirecionado para a tela que exibe os produtos da categoria selecionada. A tela em questão é ilustrada na Figura 7 (tela C) e seu código está presente no arquivo *produtosPage.dart*.

Primeiramente deve-se importar o plug-in do Cloud Firestore como é demonstrado no Quadro 2.

[5] Mais informações em: https://firebase.google.com/docs/flutter/setup?hl=pt

Quadro 2: Trecho de código do arquivo produtosPage.dart

```
1  import 'package:cloud_firestore/cloud_firestore.dart';
```

Fonte: Do autor

Ainda no arquivo *produtosPage.dart*, o código demostrado no Quadro 3 é o responsável por carregar na tela os produtos da categoria selecionada.

Tudo "acontece" dentro de um *scaffold* (Quadro 3, linha 2) que nada mais é do que um *widget* que implementa a estrutura básica da tela que neste caso é composta pelo cabeçalho (Quadro 3, linha 3, *appBar*) e conteúdo principal da tela (Quadro 3, linha 7, *body*).

Quadro 3: Trecho de código do arquivo produtosPage.dart

```
1  Widget build(BuildContext context) {
2   return Scaffold(
3     appBar: AppBar(
4     title: Text(snapshot.data["nome"]),
5     ...
6     ),
7     body: FutureBuilder<QuerySnapshot>(
8      future: Firestore.instance.collection("produtos").document
             (snapshot.documentID).collection("itens").getDocuments(),
9      builder: (BuildContext context, DocumentSnapshot snapshot){
10       if(!snapshot.hasData)
11         return Center(child: CircularProgressIndicator(),);
12       Else
13         return GridView.Builder (
14           padding: EdgeInsets.all(4.0),
15           itemCount: snapshot.data.documents.length,
16           itemBuilder: (context, index){
17              ProdutoData produto = ProdutoData.fromDocument
                    (snapshot.produto.documents[index]);
18              return ItemProduto(produto);
19           }
20         );
21       }
22     )
23   );
24 }
```

Fonte: Do autor

No *body*, utilizou-se um *FutureBuilder* (Quadro 3, linha 7) que será do tipo *QuerySnapshot*. O *FutureBuilder* trata-se de um *widget* que se constrói baseado no último instantâneo (*snapshot*) da interação de seu método *future* (Quadro 3, linha 8). Já um *QuerySnapshot* contém os resultados de uma consulta. Pode conter zero ou mais objetos do tipo *DocumentSnapshot* que por sua vez contém dados lidos de um documento no banco de dados do Cloud Firestore.

A requisição assíncrona do método *future* se dá na coleção produtos, no documento de ID igual a propriedade *documentID* do snapshot passado por parâmetro na tela anterior (representa a categoria de produtos selecionada) e neste documento, busca todos os documentos dentro da coleção "itens".

O método *builder* (Quadro 3, linha 9) recebe em um de seus parâmetros um *DocumentSnapshot*, resultado da requisição do método *future*, e retorna um *widget* de acordo com o valor deste *DocumentSnapshot*. Se estiver vazio (sem dados), retorna um *widget* que exibe uma animação circular indicando que está carregando os dados (Quadro 3, linha 11), quando conter algum dado, retorna um *GridView.Builder* (Quadro 3, linha 13), trata-se de uma *widget* apropriado para criação de grids dinâmicos que possuem um número grande de itens.

No *GridView.Builder* dois parâmetros são necessários. O *itemCount* (Quadro 3, linha 15) que define quantas vezes a função em *itemBuilder* (Quadro 3, linha 16) será chamada. A função em *itemBuilder* (Quadro 3, linhas 17 a 18) define um objeto de nome produto que converte os dados do *DocumentSnapshot* em dados da classe de nome ProdutoData e por fim retorna um *widget* chamado ItemProduto passando por parâmetro o objeto produto. Este *widget* foi criado com o objetivo de exibir um componente com as informações do produto. O resultado descrito nesta seção pode ser visualizado na Figura 7 (tela C).

Outro serviço disponibilizado pelo Firebase que foi utilizado no desenvolvimento da aplicação foi o FCM. Ele permite utilizar o recurso de p*ush notification*[6].

Na aplicação do lado cliente, que recebe as notificações, objetivando maior organização foi criado um arquivo de nome *firebaseNotification.dart*. Este arquivo encontra-se na pasta lib do projeto e seu conteúdo é demonstrado no Quadro 4.

Quadro 4: Trecho de código do arquivo firebaseNotification.dart

```
1   import 'dart:io';
2   import 'package:firebase_messaging/firebase_messaging.dart';
3   class FirebaseNotifications {
4    final FirebaseMessaging _firebaseMessaging = new FirebaseMessaging();
5    var Token = "";
6    FirebaseNotifications();
7    void iniciarFirebaseListeners() {
8      _firebaseMessaging.configure(
9        onMessage: (Map<String, dynamic> message) async { },
10       onResume: (Map<String, dynamic> message) async { },
11       onLaunch: (Map<String, dynamic> message) async { }, );
12     _firebaseMessaging.requestNotificationPermissions(
13       const IosNotificationSettings(sound: true, badge: true, alert: true));
14     _firebaseMessaging.getToken().then((token) {
15       Token = token.toString();});
16   }
17   String pegaToken(){return Token;}
18  }
```

Fonte: Do autor

[6] Trata-se de uma mensagem de alerta enviada automaticamente ao usuário por aplicativos de *smartphone*, *tablet* ou navegador. Essas notificações aparecem diretamente na tela do dispositivo acompanhadas de toques sonoros.

Inicialmente importa-se o *plugin* do FCM (Quadro 4, linha 2) e define-se a classe *FirebaseNotifications* (linha 4). O serviço é então inicializado na variável *_firebaseMessaging* (Quadro 4, linha 4).

Na sequência na função iniciarFirebaseListenner (Quadro 4, linhas 7 a 16) primeiramente são configurados os eventos *onMessage*, *onResume* e *onLaunch* (Quadro 4, linhas 8 a 11). Estes eventos podem ser tratados de acordo com a necessidade de cada aplicação[7]. Em seguida a função *requestNotificationPermitions* para solicitar a permissão de exibição da notificação no celular (Quadro 4, linhas 12 e 13) e na sequência a função *getToken* que armazena o *token* exclusivo do celular em uma variável (Quadro 4, linhas 14 e 15). Esta informação representa o dispositivo/celular e geralmente é salvo no banco de dados para que seja possível mandar notificações para um usuário especifico e não para todos os usuários cadastrados no aplicativo. Em nosso caso esta informação é retornada através da função pegaToken (Quadro 4, linha 17).

O arquivo *usuarioModel.dart* (Quadro 5) que se localiza na pasta *models* do projeto contém dentre outros códigos, a função responsável pelo login do usuário. A atualização do *token* na base de dados do usuário se dará no momento que o usuário efetuar *login*.

Quadro 5: Trecho de código do arquivo usuarioModel.dart

```
1   import 'package:cloud_firestore/cloud_firestore.dart';
2   ...
3   import '../firebaseNotification.dart';
4   ...
5   class UsuarioModel extends Model {
6      ...
7      final _firebaseNotifications = new FirebaseNotifications();
8
9      @override
10     void initState() {
11        super.initState();
12        _firebaseNotifications.iniciarFirebaseListeners();
13     }
14     ...
```

Fonte: Do autor

Inicialmente são realizadas as importações necessárias (Quadro 5, linha 1 e 2) e após o arquivo *firebaseNotification.dart* é instanciado (Quadro 5, linha 3). Em seguida cria-se uma instância de FirebaseNotification (Quadro 5, linha 7) para adiante chamar seu método iniciarFirebaseListeners() (Quadro 5, linha 12),

Na função atualizaToken() exibida no Quadro 6 encontra-se o código que atualiza o token na base de dados do usuário.

[7] Mais informações em: https://pub.dev/packages/firebase_messaging

Quadro 6: Trecho de código do arquivo usuarioModel.dart

```
1  void _atualizaToken() async {
2     final token = _firebaseNotifications.PegaToken();
3     Map<String, dynamic> mapa = {"token": token};
4     await Firestore.instance.collection("usuarios").document
          (firebaseUser.uid).updateData(mapa);
5  }
```

Fonte: Do autor

Primeiro define-se uma variável que armazenará o retorno da função PegaToken() (Quadro 6, linha 2), Em seguida cria-se um *map* (Quadro 6, linha 3) que armazena o *token* em formato JSON. Por fim, utilizando o recurso do Cloud Firestore atualizamos a base de dados do usuário correspondente passando o valor do *map* (Quadro 6, linha 4).

Para o envio das notificações utilizou-se uma requisição POST[8], e para este fim foi utilizado o *package* http[9]. No arquivo *pubsbec.yaml* agora do aplicativo do lado restaurante adiciona-se a dependência para utilizá-lo (Quadro 7).

Quadro 7: Trecho de código do arquivo pubsbec.yaml

```
1    http: ^0.12.0+4
```

Fonte: Do autor

No arquivo que a requisição será enviada, importa-se o plug-in do http como é demonstrado no Quadro 8.

Quadro 8: Trecho de código do arquivo itemListaPedidos.dart

```
1  import 'package:http/http.dart' as http;
```

Fonte: Do autor

O JSON exibido no Quadro 9 é o código da notificação que é enviada quando o usuário administrador do restaurante avança o status do pedido de um cliente de “em preparação” para “em transporte”.

Notification (Quadro 9, linhas 1 a 3) define o título e o texto a ser exibido na aba de notificações do celular, *priority* determina a prioridade da notificação, *data* (Quadro 9, linhas 6 a 10) é o dado que se deseja enviar (nada foi alterado em relação ao sugerido na documentação) e por fim o *to* (Quadro 9, linha 11) define quem receberá a notificação, em nosso caso especificou-se o *token* que está armazenado na base de dados do usuário que fez o pedido.

[8] É um dos muitos métodos de requisição suportados pelo protocolo HTTP usado na web.
[9] Mais informações em: https://pub.dartlang.org/packages/http.

Quadro 9: Trecho de código do arquivo itemListaPedidos.dart

```
 1  String DATA = "{"notification": {
 2                    "body": "Quase lá, seu pedido saiu para entrega!!!",
 3                    "title": "Pedido: "+pedido.documentID+" " },
 4                  "priority": "high",
 5                  "data": {
 6                    "click_action": "FLUTTER_NOTIFICATION_CLICK",
 7                    "id": "1",
 8                    "status": "done" },
 9                  "to": "+_usuario["token"]+"
10                }";
11  http.post("https://fcm.googleapis.com/fcm/send",
12             body: DATA,
13             headers: {"Content-Type": "application/json",
14                       "Authorization": "key="+fcmKey }
15           );
```

Fonte: Do autor

Nos parâmetros da requisição POST (Quadro 9, linhas 11 a 15) envia-se o JSON definido anteriormente no campo *body* (Quadro 9, linha 12) e no campo *header* destaca-se o conteúdo de *Authorization* (Quadro 9, linha 14) que é chave do servidor do FCM. Esta chave deve ser coletada nas configurações do projeto Cloud Messaging[10].

4 CONSIDERAÇÕES FINAIS

A popularização dos dispositivos móveis nos dias atuais possibilitou o surgimento da demanda por diversas aplicações que facilitam a vida das pessoas oferecendo ferramentas práticas que podem ser usadas nas tarefas do dia a dia.

No decorrer deste trabalho foram abordados as características e o funcionamento do Flutter e como essa ferramenta pode ser utilizada para o desenvolvimento de aplicações multiplataforma. Também foram destacadas o uso da plataforma Firebase de *backend*. Os conhecimentos tratados foram colocados em prática através de um estudo de caso que consistia em um aplicativo direcionado as necessidades de um restaurante. A partir disto foi possível enumerar os pontos positivos e negativos observados durante o processo.

4.1 ASPECTOS POSITIVOS

Destaca-se entre os pontos positivos que o Flutter, embora seja uma ferramenta nova, possui ampla documentação e dispõe de uma variedade de *widgets* prontos para diversas funcionalidades, este aspecto além de facilitar a construção da interface da aplicação, também permitiu a integração com o Firebase de uma forma bastante simples. A linguagem utilizada

[10] Mais informações em: https://firebase.google.com/docs/cloud-messaging/concept-options?hl=pt-br

também merece destaque, pois certos aspectos do Dart funcionam muito bem neste contexto como o recurso de *hot reload* que proporciona mais agilidade ao desenvolvimento.

4.2 ASPECTOS NEGATIVOS

Porém, por se tratar de uma tecnologia relativamente recente, é comum deparar-se com erros inesperados tanto durante o desenvolvimento, quanto durante a execução, muitos deles causados por atualizações de versão do *framework*. Muitas vezes esses erros podem ser complexos para resolver e podem exigir um nível de conhecimento mais profundo seja do Flutter como das bibliotecas e tecnologias utilizadas em conjunto. Apesar disso, ainda é uma tecnologia que traz mais vantagens do que desvantagens quando o assunto é o desenvolvimento de aplicações multiplataforma.

4.3 TRABALHOS FUTUROS

Como trabalho futuro podem ser citadas as funcionalidades a serem acrescentadas na aplicação, tais como: sistema de pagamento; acesso ao recurso de GPS do dispositivo, para definir o local de entrega; chat em tempo real, proporcionando a comunicação entre os clientes e o restaurante, por exemplo.

ABSTRACT

With the growth of the mobile device market, the demand for a professional with knowledge in mobile application development has been growing continuously. This scenario encourages the study of mobile development technologies. The operating systems most used in the mobile market, iOS and Android, have different characteristics that, added to the various existing devices, a diversity of variables for developers, generating higher costs for companies that wish to have their applications running on a multiplatform. A variety of solutions have been proposed to mitigate this problem, multiplatform development languages have emerged from the possibility of cost savings. Among the various technologies available in the current market, the Flutter has become evident. It is a framework developed by Google in the Dart programming language. Enables the development of applications based on the composition of widgets. The purpose of this work is to present the use of this technology in a case study that consists of the development of a mobile application coded in Dart using the Flutter framework and performing data persistence with a Firebase platform, in order to demonstrate its particularities and characteristics. For the application, it is a restaurant ordering system and will have two modules, customer and restaurant. The client module will have the resources to register users as well as place product orders and follow up on these orders. The restaurant module allows the restaurant administrator to manage the categories and products and advance the status of orders. Such action sends a notification to the customer's cell phone informing them of the change in the order status. At the end, it can be seen that the Flutter tool has resources and solutions for

various needs related to the development of applications for mobile devices, we highlight the various widgets that range from the layout of the screens to an application state management.

Keywords: Multiplatform, Dart, framewok, Firebase.

REFERÊNCIAS

ABLESON, W. Frank et al. Android em Ação (ISBN 978-85-352-4841-8). 3ª Edição. São Paulo: Campus, 2012.

App Annie, 2017. Disponível em: <https://www.appannie.com/en/insights/market-data/global-app-downloads-consumer-spend-hit-q3-2017-recap/>. Acesso em: 29 mar. 2019.

CARLSTRÖM, Oscar A. F. Evaluation Targeting React Native in Comparison to Native Mobile Development. Department of Design Sciences Faculty of Engineering LTH, Lund University. Suécia. 2016. Disponível em: <https://lup.lub.lu.se/student-papers/search/publication/8886469>. Acesso em 10 abr. 2019.

DAGNE, Lukas. Flutter for cross-platform App and SDK development. 2019. Dipsonível em: <https://www.theseus.fi/handle/10024/172866> Acesso em: 15 ago. 2019.

DANIELSSON, William. React Native application development – A comparison between native Android and React Native. 2016. Disponível em: <http://www.diva-portal.org/smash/record.jsf?pid=diva2%3A998793&dswid=7892 >. Acesso em 18 abr. 2019.

FAYZULLAEV, Jakhongir. Native-like Cross-Platform Mobile Development Multi-OS Engine & Kotlin Native vs Flutter. 2018. Disponível em: <https://www.theseus.fi/handle/10024/148975>. Acesso em: 31 jul. 2019.

FGV. 29ª Pesquisa Anual do Uso de TI. 2018. Disponível em: <https://eaesp.fgv.br/ensinoeconhecimento/ centros/cia/pesquisa>. Acesso em: 25 abr. 2019.

FIREBASE, Documentation. 2019. Disponível em: <https://firebase.google.com/docs/>. Acesso em: 20 set. 2019.

GOOGLE. Flutter. 2019a. Disponível em: <https://flutter.dev/>. Acesso em: 20 set. 2019.

_____. Flutter Technical Overview. 2019b. Disponível em: <https://flutter.dev/docs/resources/technical-overview>. Acesso em: 20 set. 2019.

HANSSON, Niclas; VIDHALL, Tomas. Effects on performance and usability for cross-platform application development using React Native. 2016. Disponível em: <http://www.diva-portal.org/smash/record.jsf?pid=diva2%3A946127&dswid=2151>. Acesso em: 17 abr. 2019.

IDC. Smartphone Market Share. 2018. Disponível em: <https://www.idc.com/promo/smartphone-market-share/os>. Acesso em: 31 mar. 2019.

LECHETA, Ricardo R. Google Android: Aprenda a criar aplicações para dispositivos móveis com o Android SDK (ISBN: 978-85-7522-468-7). 5ª Edição. São Paulo: Novatec, 2015.

LELLI, Andreas; BOSTRAND,Viktor. Evaluating Application Scenarios with React Native A Comparative Study between Native and React Native Development. 2016. Disponível em: <http://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1071001&dswid=-8355 >. Acesso em: 15 abr. 2019.

LITAYEM, Nabil; et al. Review of Cross-Platforms for Mobile Learning Application Development. International Journal of Advanced Computer Science and Applications (IJACSA), Vol. 6, nº. 1, 2015. Disponível em: <https://www.researchgate.net/publication/271642395_Review_of_Cross-Platforms_for_Mobile_Learning_Application_Development>. Acesso em 18 abr. 2019.

MILANI, Andre. Programando Iphone e Ipad Aprenda a Construir Aplicativos para iOS. (ISBN: 978-85-7522-394-9). São Paulo: Novatec, 2012.

SILVA, Marcelo M. da; SANTOS, Marilde T. P. Os Paradigmas de Desenvolvimento de Aplicativos para Aparelhos Celulares. Tecnologias, Infraestrutura e Software. São Carlos: v. 3, n. 2, p. 162-170, mai-ago 2014.

WU, Wenhao. React Native vs Flutter, cross-platform mobile application frameworks. 2018. Disponível em: <https://www.theseus.fi/handle/10024/146232>. Acesso em: 20 jul. 2019.