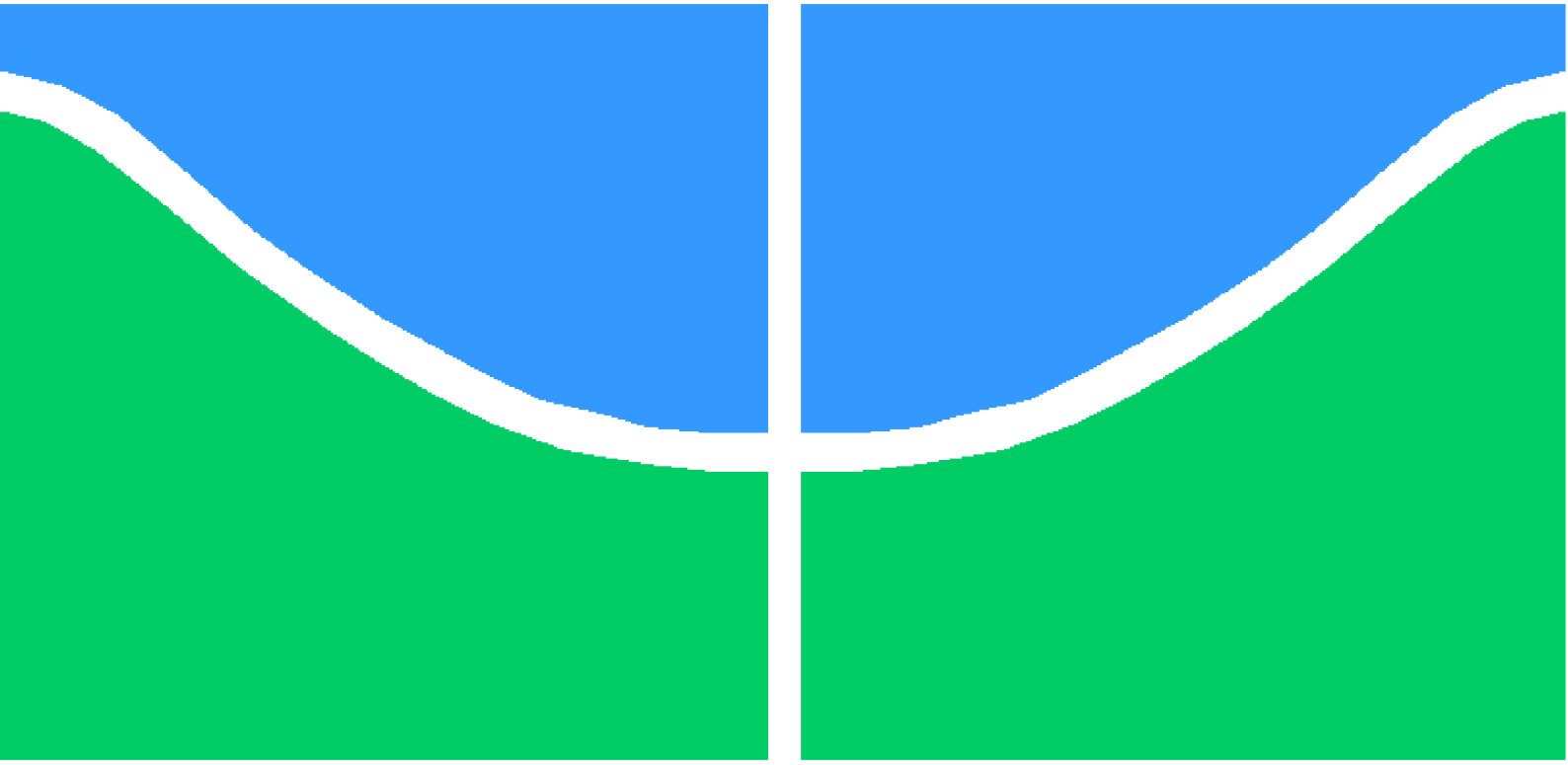

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Engenharia de Software

Estudo comparativo entre o desenvolvimento de aplicativos móveis utilizando abordagem nativa e multiplataforma

Autor: Beatriz Rezener Dourado Matos, João Gabriel de Britto e Silva

Orientador: Prof. Dr. Paulo Roberto Miranda Meirelles

Brasília, DF

2016

Beatriz Rezener Dourado Matos, João Gabriel de Britto e Silva

Estudo comparativo entre o desenvolvimento de aplicativos móveis utilizando abordagem nativa e multiplataforma

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Orientador: Prof. Dr. Paulo Roberto Miranda Meirelles

Brasília, DF

2016

Beatriz Rezener Dourado Matos, João Gabriel de Britto e Silva

Estudo comparativo entre o desenvolvimento de aplicativos móveis utilizando abordagem nativa e multiplataforma/ Beatriz Rezener Dourado Matos, João Gabriel de Britto e Silva. – Brasília, DF, 2016-

101 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dr. Paulo Roberto Miranda Meirelles

Trabalho de Conclusão de Curso – Universidade de Brasília - UnB
Faculdade UnB Gama - FGA , 2016.

1. mobile. 2. multiplataforma. I. Prof. Dr. Paulo Roberto Miranda Meirelles. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. Estudo comparativo entre o desenvolvimento de aplicativos móveis utilizando abordagem nativa e multiplataforma

CDU 02:141:005.6

Beatriz Rezener Dourado Matos, João Gabriel de Britto e Silva

Estudo comparativo entre o desenvolvimento de aplicativos móveis utilizando abordagem nativa e multiplataforma

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Trabalho aprovado. Brasília, DF, 05 de dezembro de 2016:

Prof. Dr. Paulo Roberto Miranda Meirelles
Orientador

Prof. Msc. Renato Coral Sampaio
Convidado 1

Prof. ?? ??
Convidado 2

Brasília, DF
2016

Dedicamos este trabalho às nossas famílias, que nos acompanharam e apoiaram durante nossa vida acadêmica e por serem responsáveis pela base do nosso caráter, servindo sempre como inspirações para nossas vidas.

Agradecimentos

Eu, Beatriz Rezener, agradeço à minha família, em especial aos meus pais, por todo amor, carinho e apoio incondicional. À meu namorado, João Gabriel, por ser tão companheiro e me ajudar a trilhar meu caminho pessoal e profissional. Aos meus amigos Wushuzeiros, por me auxiliarem a manter o foco em minha saúde e por me propiciarem incontáveis momentos de diversão.

Eu, João Gabriel, agradeço a toda minha família que sempre me apoiou e me ajudou a levantar por mais embaixo que eu estivesse, em especial a minha mãe que moveu céus e mares para me fazer ser quem sou. Agradeço a minha namorada Beatriz, co-autora deste trabalho, que me ajudou a trilhar um caminho excelente em minha graduação e em minha vida desde que a conheci, sempre me apoiando e me fazendo ser melhor. Agradeço aos meus amigos de faculdade, pelas risadas, trabalhos, conversas sinceras, brigas e discussões que me fizeram crescer pessoal e profissionalmente. Por fim, agradeço aos Wushuzeiros, por serem tão amigos, companheiros, família e em tão pouco tempo serem uma mudança tão grande na minha vida.

Agradecemos imensamente ao nosso orientador Prof. Dr. Paulo Meirelles, por nos ajudar no momento final da graduação nos apoiando e fazendo o possível para que fizéssemos um bom trabalho. Agradecemos especialmente ao Prof. Dr. Sérgio Freitas, por ter confiado em nós e nos dado a chance de iniciar nossa vida como Engenheiros de Software ainda na faculdade. Agradecemos também a todos os professores, sem exceção, que nos ensinaram, ao longo da nossa graduação, o que significa ser um Engenheiro de Software. Não é possível citar todos, pois foram muitos, mas ficam aqui registrados os nossos sinceros agradecimentos.

"Tudo que havia ao seu redor era o branco da neve,
que reluzia a pureza, a ingenuidade e a ignorância.
Mal sabia o guerreiro que o branco marcava o início
de sua árdua, porém gloriosa jornada.
(Naves, Wesley)"

Resumo

O desenvolvimento de aplicativos móveis intensificou-se nos últimos anos devido ao crescimento acelerado e popularização dos *smartphones*. Cada empresa do ramo móvel possui seu próprio sistema operacional, loja de aplicativos, parcela de mercado e ambientes de desenvolvimento. No entanto, quanto mais sistemas diferentes existem, maior o esforço, custo e tempo para desenvolver um *app* para todas as plataformas existentes. Visando sanar dificuldades, o desenvolvimento multiplataforma tem a premissa de criação de apenas um código que possa abranger várias plataformas. Nesse contexto, o presente trabalho busca definir vantagens e desvantagens da abordagem multiplataforma quando comparada com a abordagem nativa. Por meio da análise de um exemplo de uso, que consistiu na recriação de um aplicativo nativo, implementado originalmente para a plataforma iOS, utilizando o *framework* Ionic, foi possível comparar e comprovar empiricamente os dados obtidos na literatura. Mediante análise exploratória, foram selecionadas funcionalidades para verificar a viabilidade do desenvolvimento dessas em ambiente multiplataforma e para realizar uma comparação com o desenvolvimento das mesmas em um ambiente nativo. Os dados obtidos foram confrontados com as opiniões de especialistas para avaliar a percepção dos mesmos sobre o cenário atual do desenvolvimento multiplataforma. Concluiu-se que cada abordagem tem um momento certo para ser utilizada, não sendo uma melhor que a outra, mas apenas diferentes entre si, e deve-se avaliar cada caso, considerando-se uma série de fatores para escolher qual abordagem utilizar.

Palavras-chaves: desenvolvimento. móvel. multiplataforma. nativo. ionic.

Abstract

Mobile application development has intensified in recent years due to rapid growth and popularization of smartphones. Each company of mobile branch has its own operating system, application store, market share and development environments. However, the more different systems exist, the greater the effort, cost and time to develop an app for all existing platforms. Then came the concept of cross-platform development, with the premise of code only once and span multiple platforms. In this context, this paper seeks to define advantages and disadvantages of the cross-platform approach compared with the native approach. Through an analysis of study, which consisted of the recreation of a native application, originally implemented for iOS platform, using the Ionic framework, it was possible to compare and empirically verify the data obtained in the literature. Through exploratory analysis, a small group of key features were selected in order to verify their development viability on cross-platform enviroment and then compare with their native development. The data obtained were compared with experts opinions to analize their perception of cross-platform development current scenario. It was concluded at the end of this work, that each approach has a certain time to be used, not being better than the other, but only different, and developers must evaluate each case considering a number of factors to choose which approach is better to be used in each context.

Key-words: development. mobile. cross. platform. native. ionic.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

API	Application Programming Interface
CSS	Cascade Style Sheet
CLI	Command-Line Interface
DSL	Dynamic Shared Libraries
GPS	Global Positioning System
HAL	Hardware Abstraction Layer
HTML	HiperText Markup Language
IDE	Integrated Development Enviroment
MVC	Model-View-Controller
OHA	Open Handset Alliance
SDK	Software Development Kit
SO	Sistema Operacional
SPA	Single Page Application

Sumário

1 Introdução

Com o aumento da quantidade de dispositivos móveis no mercado, a demanda por aplicações móveis também aumentou (CEVALLOS, 2014). De início, a única opção que havia era desenvolver aplicações específicas para uma plataforma, utilizando todo o ambiente daquela plataforma, por exemplo, no caso do iOS, a liguagem de programação era o *Objective-C* com a *IDE (Integrated Development Enviroment) Xcode* e o *SDK (Software Development Kit)* do iOS (HEITKöTTER; HANSCHKE; MAJCHRZAK, 2013).

Segundo Prezotto e Boniati (2014), essa forma de desenvolvimento é conhecida como desenvolvimento **nativo** e é aquele no qual um aplicativo é projetado e construído especificamente para uma plataforma. Todas as funcionalidades da plataforma estão disponíveis sem restrição e existem padrões de interface gráfica e experiência de usuário específicos, que ajudam o usuário a entender como aquele aplicativo funciona, já que todos os outros aplicativos daquela plataforma seguem os mesmos padrões (CORRAL; JANES; REMENCIUS, 2012).

No entanto, com o aumento exponencial da demanda por aplicativos, surgiu a necessidade de desenvolvimento rápido para muitas plataformas distintas, o que era caro e difícil, visto que, cada plataforma tinha um ambiente de desenvolvimento completamente diferente das demais, o que demandava da equipe muitos conhecimentos diferentes para poder desenvolver e manter vários aplicativos (PREZOTTO; BONIATI, 2014).

Segundo Heitkötter, Hanschke e Majchrzak (2013), com esse novo panorama do mercado, surgiu a necessidade da criação de apenas um *app* que pudesse ser executado em várias plataformas. Essa forma de desenvolvimento ficou conhecida como **multiplataforma** (*cross-platform*). Consiste na criação de uma página *web*, normalmente utilizando *HTML (HiperText Markup Language)*, *CSS (Cascade Style Sheet)* e *JavaScript*, que pode ser mostrada dentro de uma *web view* embutida em um aplicativo nativo, ou seja, é feito apenas um código, que é executado e mostrado em um *container* dentro de um aplicativo nativo (STARK, 2010; HEITKöTTER; HANSCHKE; MAJCHRZAK, 2013).

Dessa forma, é possível desenvolver um mesmo aplicativo que será executado dentro de várias plataformas diferentes, sem a necessidade de ter uma equipe especialista em cada plataforma. No entanto, não é possível ter acesso total às funcionalidades da plataforma, assim como no desenvolvimento nativo, e também não existe padrão de interface ou de experiência de usuário que sirva para várias plataformas ao mesmo tempo, podendo causar uma sensação ruim na usabilidade e experiência de uso do aplicativo (CORRAL; JANES; REMENCIUS, 2012).

O presente trabalho visa auxiliar desenvolvedores, no momento inicial da criação de

aplicativos para dispositivos móveis, a escolherem a melhor abordagem que se encaixa no contexto em que estão inseridos, considerando as reais vantagens e desvantagens presentes atualmente no desenvolvimento multiplataforma.

1.1 Justificativa

Sabendo que existem diversas plataformas diferentes para dispositivos móveis e que há uma crescente necessidade de mercado de abarcar o maior número de plataformas, o mais rápido, com a melhor qualidade e menor custo possíveis, o presente trabalho visa auxiliar desenvolvedores *mobile* a compreender melhor o mundo do desenvolvimento móvel e tomar a decisão de desenvolver nativamente ou multiplataforma com mais consciência das vantagens e desvantagens de cada abordagem.

1.2 Objetivos

O objetivo deste trabalho é definir, por meio de pesquisa e comprovação empírica, as vantagens e desvantagens de desenvolvimento multiplataforma de aplicações móveis em relação ao desenvolvimento nativo. Além desse objetivo principal, existem ainda outros objetivos secundários que precisam ser atingidos. São eles:

- Pesquisar sobre o desenvolvimento de aplicativos móveis;
- Compreender a arquitetura das duas principais plataformas móveis da atualidade (iOS e Android);
- Desenvolver uma réplica de um aplicativo criado nativamente para uma plataforma, utilizando tecnologias multiplataforma;
- Comparar o desenvolvimento do aplicativo multiplataforma com o aplicativo nativo;
- Comparar funcionalidades chave desenvolvidas nas duas abordagens;
- Confrontar resultados obtidos empiricamente com opiniões de especialistas da área;

Com base nos objetivos traçados para o presente trabalho, a questão problema a ser respondida é:

Quais as vantagens e desvantagens do desenvolvimento multiplataforma de aplicações móveis em relação ao desenvolvimento nativo?

1.3 Organização do Trabalho

Este trabalho foi dividido em seis capítulos. No Capítulo 2, é feito um estudo sobre como é feito hoje e como evoluiu ao longo dos últimos anos, o desenvolvimento de aplicações móveis. Também foi feito um comparativo para avaliar vantagens e desvantagens entre as duas abordagens de desenvolvimento trazidas pela literatura. No Capítulo 3, é apresentada a metodologia adotada para o desenvolvimento do trabalho, o planejamento feito e como será executado, para que o trabalho possa ser replicado e/ou continuado futuramente. No Capítulo 4, é feita uma análise de exemplo de uso, realizado durante o trabalho, sobre o desenvolvimento de aplicativos móveis. No Capítulo 5, é feita a comparação do desenvolvimento de algumas funcionalidades comuns em muitos aplicativos conhecidos, e os dados confrontados com as opiniões dos especialistas a respeito dos dados obtidos. Por fim, no Capítulo 6, apresentam-se a resposta da questão problema, as conclusões do trabalho e uma lista de possíveis limitações e trabalhos futuros.

2 Desenvolvimento de aplicativos móveis

Segundo El-Kassas et al. (2015), o desenvolvimento de aplicativos móveis diferencia-se do desenvolvimento de outros tipos de *software* por possuir particularidades e restrições. Os desenvolvedores devem ter em mente aspectos como as capacidades e especificações dos dispositivo móveis, a mobilidade, o *design* e navegabilidade de interface gráfica, segurança e privacidade do usuário. De acordo com Corral, Janes e Remencius (2012), aplicações móveis são desenvolvidas dinamicamente e lançadas no mercado em pequenos ciclos. Os produtos finais costumam ser de pequeno porte e comercializados a preços baixos. As equipes de desenvolvimento também tendem a ser pequenas. Algumas particularidades a serem consideradas no desenvolvimento *mobile* são listadas a seguir, baseadas em El-Kassas et al. (2015).

- **Limitações de recursos**: As capacidades de processamento e de armazenamento dos dispositivos móveis são limitadas e a conectividade pode ser afetada por ser uma plataforma móvel.

- **Ecossistema heterogêneo**: O desenvolvimento de aplicações móveis encontra-se em um contexto heterogêneo devido aos diferentes sistemas operacionais e à grande quantidade de dispositivos distintos, com variações de poder computacional, configurações de *hardware* e tamanhos de tela. Essas singularidades devem ser consideradas no desenvolvimento de *apps*, o que pode implicar na necessidade de diferentes versões de uma mesma aplicação.

- **Experiência de uso**: é importante que as aplicações sejam simples e que possuam uma interface amigável para atender às expectativas dos usuários, caso contrário, devido ao grande número de aplicações disponíveis, o *app* pode ser descartado e substituído por outros presentes nas lojas de aplicativos.

- **Manutenção**: plataformas *mobile* passam por constantes atualizações que podem afetar *apps* já desenvolvidos, a ponto de torná-los inutilizáveis ou causarem desconforto aos usuários. Para que a aplicação continue a funcionar corretamente, são necessárias frequentes manutenções e atualizações. Atualizações de *apps* desenvolvidos para diferentes plataformas implicam na alteração de cada uma das versões desenvolvidas.

2.1 Desenvolvimento Nativo

Aplicações nativas são desenvolvidas com o uso de ferramentas e linguagens de programação específicas para determinada plataforma, usando o *SDK* e *frameworks* providos por ela. Os *apps* ficam vinculados a esse ambiente, executando apenas nos dispositivos da plataforma alvo (EL-KASSAS et al., 2015). Caso haja a necessidade de implementação para múltiplas plataformas, a aplicação deve ser desenvolvida separadamente para cada uma delas (HEITKOTTER; HANSCHKE; MAJCHRZAK, 2013).

Posto que o *app* é desenvolvido em ambiente próprio, seguem-se os padrões técnicos, de interface e de experiência de usuário determinados pela plataforma, provendo um *look and feel* de aplicação nativa ao usuário. Aplicações nativas tem fácil acesso, por meio de *APIs (Application Programming Interface)* fornecidas pelas plataformas, a recursos dos dispositivos móveis, como sensores, câmera, *GPS (Global Positioning System)*, contatos e *e-mail* (EL-KASSAS et al., 2015).

Nas seções a seguir serão descritas as arquiteturas das plataformas iOS e Android, duas das plataformas mais conhecidas e utilizadas para desenvolvimento nativo de aplicações *mobile* (JOBE, 2013).

2.1.1 iOS

O *SO (Sistema Operacional)* iOS foi lançado em Janeiro de 2007 juntamente com o primeiro *iPhone* e funciona como uma interface entre as aplicações desenvolvidas pelos programadores (*apps*) e o *hardware* dos dispositivos (*iPhone*, *iPad*, *iPod*) (Apple Inc., 2007). Dessa forma, a comunicação com o *hardware* do dispositivo se dá por meio de um conjunto bem definido de interfaces do sistema, o que facilita o desenvolvimento de *apps* que funcionam entre os variados tipos de *hardware* dos dispositivos da Apple (Apple Inc., 2016b).

A arquitetura do iOS é baseada em camadas e, como recomendação, a Apple explica que deve-se preferir o uso de camadas mais elevadas, pois as camadas de possuem abstrações orientadas à objeto de funcionalidades implementadas nas camadas mais baixas. Isso torna o desenvolvimento mais fácil, pois reduz a quantidade de código que deve ser criado e mantém funcionalidades complexas das camadas mais baixas encapsuladas por meio das interfaces. No entanto, não há problema em usar funcionalidades presentes nas camadas mais baixas, se essas não estiverem disponíveis por meio de abstrações nas camadas superiores (Apple Inc., 2014).

A maioria das interfaces disponíveis para uso são disponibilizadas por meio de *frameworks* que podem ser adicionados ao projeto no *Xcode*, contendo *DSLs (Dynamic Shared Libraries)* e os recursos necessários como, imagens, aplicativos auxiliares e arquivos *header*, para o *framework* funcionar corretamente (Apple Inc., 2014).

2.1.1.1 Arquitetura iOS

Conforme dito anteriormente, a arquitetura do iOS é baseada nas camadas listadas, conforme a Figura 1, a seguir.

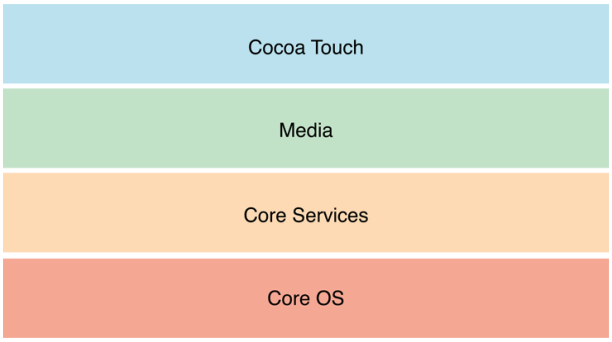

Figura 1 – Arquitetura iOS. Fonte: (Apple Inc., 2014).

As camadas do iOS são explicadas com mais detalhes a seguir.

- A camada *Cocoa Touch* é a camada de mais alto nível onde são fornecidos serviços básicos de interação com o usuário como entrada baseada em toques, notificações *Push* e outras tecnologias necessárias para melhorar a experiência do usuário como multitarefas, Continuidade (*Handoff*) e *AirDrop*, além de *frameworks* de alto nível que permitem acesso às funcionalidades do sistema como *AddressBook* para contatos, *EventKit* para eventos relacionados ao calendário e *MapKit* para mapas (Apple Inc., 2014).

- A camada logo abaixo da *Cocoa Touch* é a camada *Media* que contém tecnologias e *frameworks* necessários para a implementação de experiências multimídia com áudio, vídeo e gráficos (Apple Inc., 2014).

- A próxima camada, logo abaixo da camada *Media*, é a camada *Core Services*. Essa camada está mais próxima do *hardware* e portanto possui acesso a funcionalidades de mais baixo nível como localização, telefonia, *threads* e *SQLite*. Aqui residem dois dos *frameworks* mais importantes do iOS que são o *Foundation* e o *Core Foundation*, ambos relacionados com o gerenciamento de dados e alguns serviços e definem todos os tipos básicos de dados que todos os *apps* usam, como por exemplo, coleções, *strings*, data e hora, *sockets* e *threads* (Apple Inc., 2014).

- A última camada é a camada *Core OS*, na qual as funcionalidades de mais baixo nível são construídas e provavelmente utilizadas por outros *frameworks* em outras camadas. Se a aplicação possui requisitos de segurança ou comunicação com acessórios externos mais complicados, é possível usar as funcionalidades dessa camada (Apple Inc., 2014).

2.1.1.2 Desenvolvimento iOS

O *SDK* iOS permite que os desenvolvedores criem suas aplicações e a testem em emuladores. Contudo, para a utilização de recursos avançados e para a distribuição na *App Store*, loja de aplicativos da Apple (Apple Inc., 2016a), é exigida uma licença do *Apple Developer Program*, com custo de US$99 anuais.

Para a distribuição de aplicativos iOS, além da licença citada, é necessário submeter a aplicação para avaliação e aprovação da Apple antes de ir para a *App Store*. Para agilizar o processo de aprovação, é necessário que o aplicativo esteja de acordo com as diretrizes estabelecidas pela Apple, como as orientações de interface gráfica e as orientações de revisão da *App Store* (Apple Inc., 2016c).

Desenvolver aplicativos para a plataforma iOS requer um computador com o *SO* Mac OS e o ambiente de desenvolvimento *Xcode* (HEITKOTTER; HANSCHKE; MAJCHRZAK, 2013). O desenvolvimento é atrelado a duas linguagens de programação, o *Objective-C* e o *Swift*. O *Swift* foi lançado pela Apple como um sucessor do *Objective-C*. É uma linguagem *software* livre e pode ser integrada a códigos *Objective-C* (Apple Inc., 2016d). Pouco tempo após o seu lançamento, tornou-se uma das linguagens de programação mais utilizadas no mundo (REBOUAS et al., 2016).

2.1.2 Android

O *SO* Android foi criado pela *start-up* homônima Android Inc. em outubro de 2003. Em agosto de 2005 foi adquirida pela empresa Google, que lançou em novembro de 2007, juntamente com a *OHA (Open Handset Alliance)*, o sistema Android, *open-source* e baseado no *kernel* do Linux. Uma semana depois, foi liberada a primeira versão do *SDK* para Android. O sistema foi concebido originalmente para câmeras fotográficas, no entanto, foi percebido por seus criadores um mercado maior no ramo da telefonia e desviou-se o foco para *smartphones*, competindo diretamente com Symbian e Windows Mobile (PAPAJORGJI, 2015).

2.1.2.1 Arquitetura Android

A arquitetura Android é baseada nas camadas apresentadas na Figura 2 e são explicadas com mais detalhes a seguir.

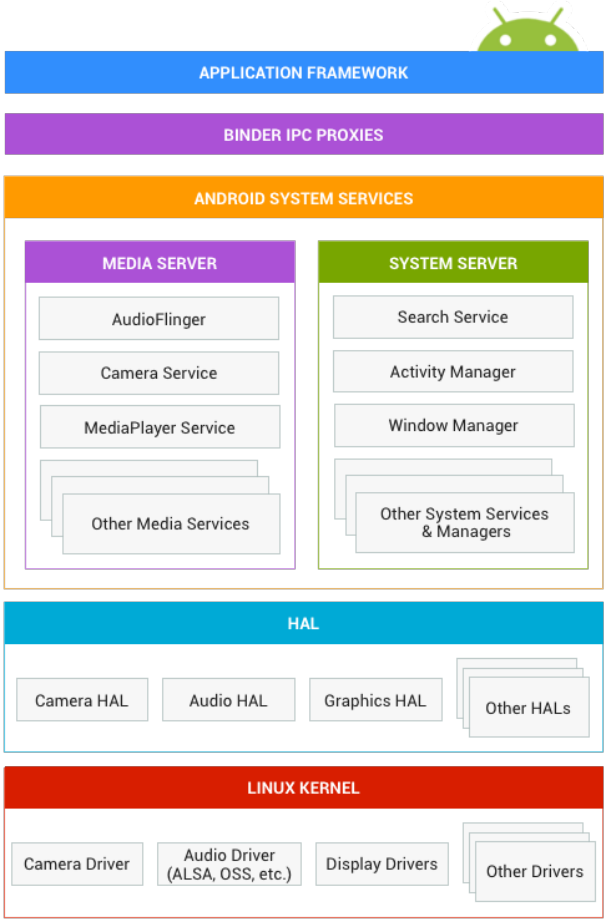

Figura 2 – Arquitetura Android. Fonte: (Android, 2016a).

- A camada mais elevada é a *Application Framework*, onde as aplicações construídas pelos desenvolvedores são instaladas (Android, 2016a).
- A camada logo a abaixo da *Application Framework* é a *Binder IPC Proxies*, onde *IPC* significa *Inter-Process Comunication*. É a camada que faz uma ponte entre as aplicações instaladas na camada superior com a próxima camada, a *System Services*. É uma interface que permite que *APIs* de alto nível interajam com serviços do sistema que residem na camada abaixo dela (Android, 2016a).
- Abaixo da *Binder IPC Proxies* fica a *System Services*, que, por meio de módulos, permite acesso a *hardwares* específicos. Cada serviço presente nessa camada foi desenvolvido para gerenciar um componente específico, como busca e notificações. Os serviços foram divididos em duas categorias, *Media* e *System*, explicadas a seguir (Android, 2016a).
  - *Media Server*: são os serviços responsáveis por gerenciar conteúdos de mídia como gravação e reprodução de áudio e vídeo.
  - *System Server*: são serviços responsáveis por gerenciar os demais tipos de serviço do sistema, como notificações e *windows*.

- Abaixo da camada *System Services* fica a camada *HAL (Hardware Abstraction Layer)* que permite que fornecedores de *hardware* criem interfaces e *drivers* para os *hardwares* que ele oferecem. Com isso, é possível criar novas funcionalidades e implementá-las sem afetar o resto das camadas do sistema (Android, 2016a).

- A camada mais baixa é a *Linux Kernel*, uma versão do *kernel* do Linux com algumas modificações, como um gerenciamento de memória mais avançado e próprio para dispositivos móveis e funcionalidades para dispositivos embarcados (Android, 2016a).

2.1.2.2 Desenvolvimento Android

O desenvolvimento de aplicações Android é atrelado à linguagem de programação *Java*. Não é requerido *SO* específico e a *IDE* indicada pelo Google é o *Android Studio*, oficial para desenvolvimento de aplicativos Android (Android, 2016b).

Para a publicação na loja de aplicativos do Google, *Google Play Store*, é cobrada uma taxa única de US$25. Antes de serem publicados, os aplicativos passam por um processo de revisão para assegurar que não violam as políticas estabelecidas pela loja (MEIER, 2015).

2.2 Desenvolvimento Multiplataforma

Para o desenvolvimento de uma aplicação nativa deve-se desenvolver o mesmo aplicativo para cada plataforma que se deseja alcançar, o que exige tempo, investimento e uma equipe com conhecimento nas variadas tecnologias usadas pelas distintas plataformas alvo. Soluções multiplataforma possibilitam a implementação de uma aplicação que pode ser executada em diferentes plataformas (EL-KASSAS et al., 2015). Conforme Heitkotter, Hanschke e Majchrzak (2013), essas aplicações são comumente categorizadas em *webapps* e aplicações híbridas.

Web apps são aplicações implementadas utilizando tecnologias *web*, levando em conta aspectos intrínsecos aos dispositivos móveis, como tamanho de tela e modo de uso (HEITKOTTER; HANSCHKE; MAJCHRZAK, 2013). Segundo Stark (2010), *web apps* não são instalados no dispositivo e não ficam disponíveis nas lojas de aplicativos, em vez disso, são acessados pelos *browsers* dos dispositivos. Eles não possibilitam o uso de alguns recursos de *hardware* dos dispositivos como sensores e *GPS* (HEITKOTTER; HANSCHKE; MAJCHRZAK, 2013). Conforme Heitkotter, Hanschke e Majchrzak (2013), *web apps* costumam ter a aparência e o comportamento de páginas *web* comuns, divergindo dos elementos de tela padrão providos pelas plataformas *mobile*.

A abordagem híbrida surgiu como uma combinação do uso das tecnologias *web* junto à possibilidade de acesso a funcionalidades nativas (HEITKOTTER; HANSCHKE; MAJCHRZAK, 2013). Em essência, aplicações híbridas são *web app* empacotadas em um aplicativo nativo (STARK, 2010).

Nas próximas subseções serão descritas algumas das tecnologias utilizadas para o desenvolvimento multiplataforma.

2.2.1 PhoneGap e Cordova

PhoneGap é um *framework* para desenvolvimento de aplicativos híbridos, criado pela empresa Nitobi. Após a empresa ser comprada pela Adobe Systems Inc. o PhoneGap teve seu código doado para a Apache Software Foundation para garantir que outras empresas pudessem contribuir (BEZERRA; SCHIMIGUEL, 2016).

Como o PhoneGap é uma marca registrada de propriedade da Adobe, na Apache teve seu nome alterado para Cordova. Tanto o *Apache Cordova* quanto o *Adobe PhoneGap*, são gratuitos e *open sources*, no entanto, o PhoneGap possui um ambiente integrado com serviços da Adobe, como o *PhoneGap Build* (BEZERRA; SCHIMIGUEL, 2016).

O Cordova permite que aplicações não nativas tenham acesso às funcionalidades nativas do dispositivo como sensores, contatos e câmera. No entanto, o Cordova apenas consegue fazer um aplicativo criado em *HTML* rodar como se fosse nativo em um dispositivo com Android e iOS, mas não consegue imitar a usabilidade e aparência dos dispositivos nativos (BEZERRA; SCHIMIGUEL, 2016).

Para preencher essa lacuna, outros *frameworks* foram criados em cima do Cordova como um complemento, provendo bibliotecas *HTML* e *CSS* para a criação de um *front-end* que se aproxime o máximo possível da usabilidade e experiência de usuário de aplicações móveis nativas (BEZERRA; SCHIMIGUEL, 2016).

O Cordova possui muitos *plugins* para poder acessar funcionalidades específicas do aparelho em que está sendo executado. Esses devem ser instalados no projeto do aplicativo via *CLI (Command Line Interface)* (BEZERRA; SCHIMIGUEL, 2016).

A fim de se evitar confusões em relação aos nomes dados ao PhoneGap e ao Cordova, pode-se entender o PhoneGap como uma distribuição do *Apache Cordova*, mantido pela Adobe Systems Inc.

2.2.2 AngularJS

HTML é a principal linguagem de marcação para a criação de páginas *web*. No entanto, os conteúdos exibidos são estáticos e o *HTML* não fornece suporte para conteúdos dinâmicos. AngularJS é um *framework open source*, mantido pelo Google, que permite

estender a sintaxe do *HTML* para poder criar páginas *web* com conteúdos dinâmicos (BEZERRA; SCHIMIGUEL, 2016). Com ele é possível utilizar *tags* não nativas do *HTML* para gerenciar conteúdos que ainda não foram dispostos na tela (GOOGLE, 2016).

O usuário faz uma ação por meio da interface gráfica, o AngularJS interpreta a ação e faz uma requisição para o servidor pedindo apenas o conteúdo necessário, que é diferente do que já está sendo mostrado para o usuário. O servidor retorna o conteúdo específico e o AngularJS apresenta o novo conteúdo inserindo-o corretamente junto com o antigo já mostrado (URSINO; CAPANNA, 2015). O ciclo do AngularJS é esquematizado na Figura 3, a seguir.

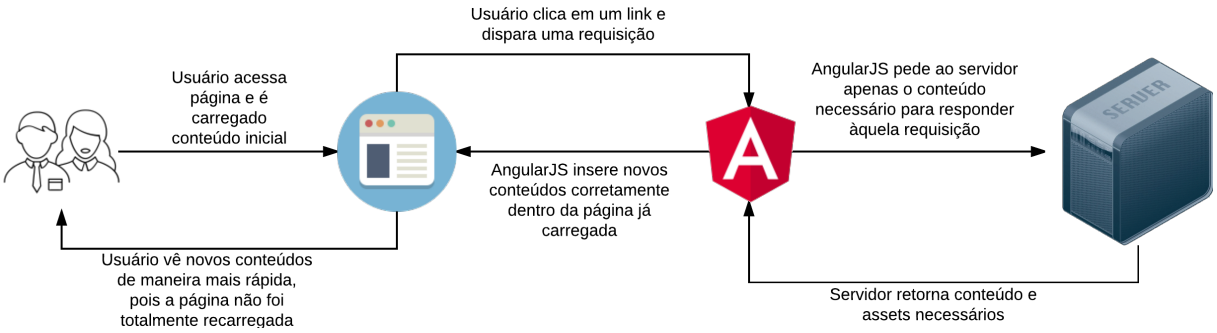

Figura 3 – Ciclo de atualização com AngularJS. Fonte: Baseado em (URSINO; CAPANNA, 2015).

Foi projetado para criação de *SPAs (Single Page Applications)* e com isso faz com que as páginas sejam mais rápidas, visto que, apenas é carregado o conteúdo que precisa ser mostrado, deixando outros conteúdos para serem carregados dinamicamente depois, de acordo com as interações do usuário (RODRíGUEZ; BALDRICH, 2015).

2.2.3 Ionic

Ionic é um *framework software* livre para desenvolvimento de aplicativos híbridos utilizando tecnologias *web* como *HTML*, *CSS* e *JavaScript* otimizadas para dispositivos móveis (DRIFTY, 2016c).

Foi criado pela empresa Drifty Co. em 2013 com base na necessidade que seus clientes apresentavam de criarem *apps*. Foi projetado para ser muito performático e funcionar com padrões e tecnologias *web* modernas (DRIFTY, 2016a).

Criado sobre os *frameworks* Cordova e AngularJS, com apenas um código é possível criar aplicativos para várias plataformas como iOS e Android, por exemplo, dentre outras. O Ionic conta ainda com um conjunto de ferramentas para auxiliar no desenvolvimento dos aplicativos (DRIFTY, 2016c).

2.2.3.1 Arquitetura de um projeto Ionic

Um projeto Ionic pode ser dividido em cinco camadas, listadas e detalhadas a seguir.

- *Views*: Camada de apresentação, onde as informações são mostradas para o usuário. Pode ser comparada como o seu homônimo no padrão *MVC (Model-View-Controller)*. Muitas vezes são chamadas de *templates*, pois é a forma com o AngularJS se refere a elas. São, normalmente, arquivos *HTML* separados, que são chamados e inseridos quando necessário pelo AngularJS. É possível utilizar *data binding* nas Views para poder estabelecer uma conexão com a controladora e compartilhar informações entre as duas camadas (DRIFTY, 2016d).

- *Controllers*: É a camada responsável por controlar o fluxo de dados e lógica da aplicação. Também pode ser comparada com a camada homônima no padrão *MVC*. Ela é responsável por apresentar ao usuário as *Views* e chamar as camadas de dados (*Services/Factories*) para ligar os dados reais da aplicação, por meio de *data binding*, à interface gráfica. As controladoras possuem uma variável chamada *scope*, que possui todas as informações que são necessárias para a criação da *View* (DRIFTY, 2016d).

- *Data(Services/Factories)*: É a camada que se aproxima da camada *model* do padrão *MVC*. Encapsula dados da aplicação e provê esses dados, geralmente, por meio de um *web service*. Essa camada responde às requisições da controladora com os dados a serem utilizados para a criação da *View* e para serem mostrados para o usuário (DRIFTY, 2016d).

- *App Configuration*: Nesta camada, as controladoras são ligadas às suas interfaces por meio de rotas. É possível, também, criar rotas padrão, para o caso de não haver nenhuma rota que esteja sendo identificada, o que poderia fazer o sistema quebrar (DRIFTY, 2016d).

- *Directives*: Essa camada serve para especificar comportamentos específicos em elementos de uma página *HTML*, ou seja, elas são um elemento ou um atributo que podem iniciar um comportamento específico definido pelo programador. É possível, por exemplo, criar uma diretiva para sempre colocar uma imagem padrão caso uma validação retorne um valor falso (DRIFTY, 2016d).

2.2.3.2 Desenvolvimento Ionic

Para o desenvolvimento de aplicações utilizando Ionic, não é necessário nenhum equipamento específico, assim como no desenvolvimento Android. «««< HEAD É necessário apenas um computador com qualquer *SO*, no entanto, para o desenvolvimento do

app iOS, ainda é necessário que se possua um computador com o *SO Mac OS* e com o Xcode (DRIFTY, 2016b). ======= É necessário apenas um computador com qualquer *SO (Sistema Operacional)*, no entanto, para o desenvolvimento do *app* iOS, ainda é necessário que se possua um computador com o *SO MacOS* e com o Xcode (DRIFTY, 2016b). »»»> afe43edf609d1dcc404070dc42f4e5b7ad8a88bb Pode-se utilizar qualquer *IDE* de acordo com as preferências do programador, não há qualquer tipo de recomendação técnica por parte da equipe ou presente na documentação do Ionic. A linguagem utilizada é o *JavaScript*, usando ainda *HTML* e *CSS* para criação da interface gráfica.

2.2.3.3 Ferramentas de Apoio

O Ionic provê, além do *HTML* e *CSS* otimizado para dispositivos móveis, uma série de ferramentas, que compõem um ecossistema de apoio e fornecem velocidade e facilidade no desenvolvimento de aplicativos híbridos. Essas ferramentas são listadas a seguir com uma breve explicação do que cada uma pode fazer (DRIFTY, 2016c).

- **Ionic Framework**: É o *framework front-end* para criação de aplicativos híbridos com tecnologias *web* construído em cima do *Apache Cordova*. Facilita a criação de aplicativos com interface gráfica e interações muito semelhantes aos dispositivos nativos, de modo que a experiência de usuário não seja diferente no uso de uma aplicação nativa e multiplataforma (DRIFTY, 2016c).

- **Ionic Creator**: Com essa ferramenta *on-line* é possível criar protótipos funcionais utilizando *drag and drop* , que podem ser executados em dispositivos ou simuladores, e depois exportar o projeto criado para a inserção de lógicas de negócio. Ela é gratuita para teste, ou seja, é possível criar um protótipo e fazer o *download* do código para continuar o desenvolvimento, no entanto, no momento em que este trabalho foi conduzido, só era possível fazer a pré-visualização em dispositivos reais para usuários pagantes (DRIFTY, 2016e).

- **Ionic Lab**: É uma ferramenta *desktop* para testar o projeto que está sendo criado com o Ionic. Nela é possível executar o projeto em templates prontos de iOS e Android, rodar os simuladores de cada plataforma, gerenciar plugins do Cordova e ver um console com *logs* relativos à execução do projeto. Encapsula o *CLI* do Ionic com uma interface gráfica simples e intuitiva (DRIFTY, 2016f).

- **Ionic Platform**: É a plataforma *on-line* do Ionic que possui serviços de *back-end* em nuvem para análise de dados de uso, usuários, configuração de *Push Notifications*, criação dos pacotes nativos e dar *deploy* nas aplicações criadas sem a necessidade de resubmeter o aplicativo para as respectivas lojas (DRIFTY, 2016g).

- **Ionic Playground**: É uma ferramenta *on-line* para criar e testar códigos *HTML*, *CSS* e *JavaScript* já integrado com a plataforma AngularJS. Nela é possível editar códigos nas três linguagens citadas e ver em tempo real as alterações feitas aparecerem dentro de um *template* de dispositivo móvel, sem a necessidade de salvar, compilar ou executar o projeto e sem ter que configurar o ambiente necessário para o desenvolvimento (DRIFTY, 2016h).

- **Ionic View App**: É um aplicativo disponível para as plataformas iOS e Android onde é possível testar o aplicativo criado no Ionic diretamente em um dispositivo real como se fosse nativo. É possível compartilhar os projetos criados para serem testados por clientes e testadores sem a necessidade de passar pelas lojas de aplicativos de cada plataforma. É semelhante ao serviço *Testflight* da Apple para testar aplicativos, ainda em desenvolvimento, e receber *feedbacks* dos clientes e testadores que o testarem (DRIFTY, 2016i).

2.3 Comparativo (Nativo x Multiplataforma)

Cada plataforma possui seu próprio ambiente de desenvolvimento, arquitetura, ferramentas, mercado de distribuição e público alvo (SHAKSHUKI et al., 2013). E cada abordagem de desenvolvimento possui suas características próprias e inerentes à forma como se dá o projeto, desenvolvimento e distribuição dos *apps* (CORRAL; JANES; REMENCIUS, 2012).

Algo importante que deve ser feito pela equipe de desenvolvimento é decidir para qual plataforma desenvolver, tendo sempre em vista que, ao não criar o aplicativo para uma plataforma, grande parcela do mercado pode estar sendo desconsiderada, o que acaba por reduzir o alcance da aplicação e por consequência os lucros obtidos com o *app* (CORRAL; JANES; REMENCIUS, 2012).

Para o desenvolvimento de várias aplicações para cada plataforma de maneira nativa, é preciso passar por todo o processo de desenvolvimento novamente, pois para cada plataforma há uma arquitetura distinta que deve ser seguida, padrões de código e interface de usuário e *APIs* próprias (HOLZINGER; TREITLER; SLANY, 2012). Dessa forma, cada aplicação deve ser codificada considerando as diferentes arquiteturas, componentes e *guidelines*, customizadas o máximo possível para ser a mesma aplicação em plataformas distintas e por fim, gerados novos executáveis e distribuídos nas diferentes lojas de aplicativos (CORRAL; JANES; REMENCIUS, 2012).

Isso implica em planejar cada *app* com base em uma plataforma distinta, por mais que se trate do mesmo aplicativo, o que acaba por encarecer e tornar mais lento o desenvolvimento (EL-KASSAS et al., 2015). Por exemplo, existem componentes que não existem em todas as arquiteturas (*navigationController* e *tabBarController* presentes na

plataforma iOS, por exemplo) e que devem ser considerados no projeto do *app* para as demais plataformas (SHAKSHUKI et al., 2013).

No entanto, quando se pensa em desenvolvimento multiplataforma, o projeto se torna agnóstico em relação à plataforma, pois o projeto é feito pensando-se apenas em uma arquitetura, um ambiente de desenvolvimento e um conjunto único de ferramentas e não de acordo com as características de cada plataforma. Após o desenvolvimento do núcleo da aplicação, podem ser feitas customizações para adaptar melhor o aplicativo para cada plataforma alvo da distribuição, criando assim um produto final verdadeiramente multiplataforma (CORRAL; JANES; REMENCIUS, 2012).

Na Figura 4 são apresentadas as diferenças no processo de desenvolvimento de um *app* utilizando tecnologias multiplataforma e nativas.

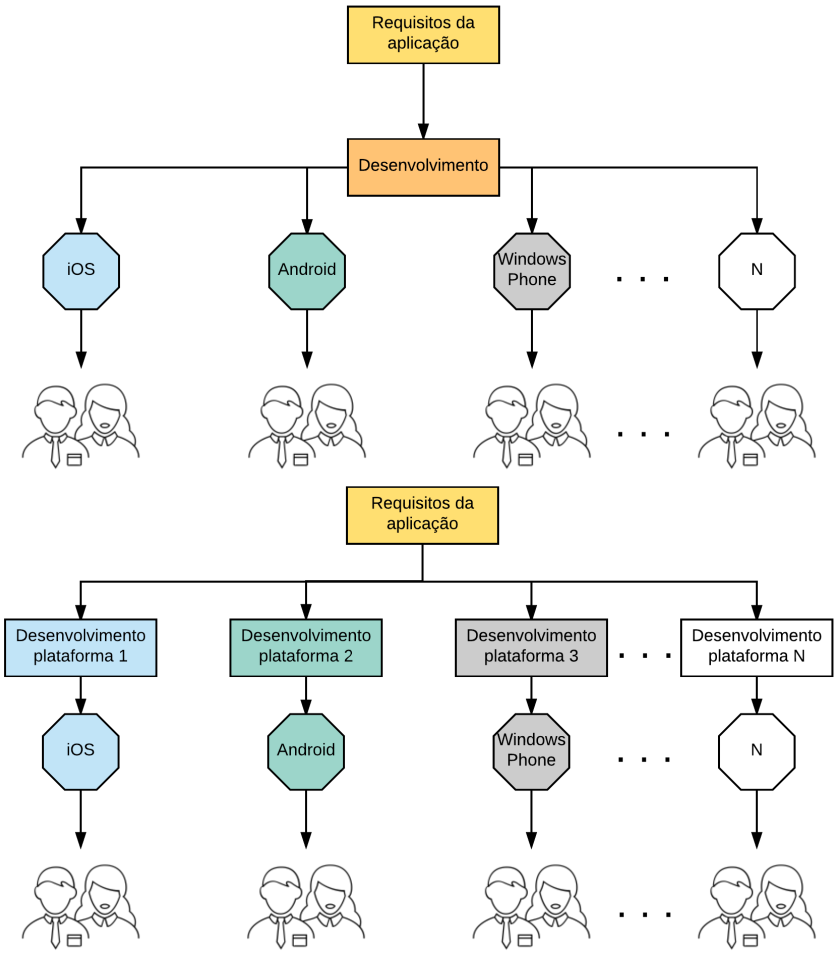

Figura 4 – Desenvolvimento Multiplataforma (acima) e Nativo (abaixo). Fonte: Baseado em (CORRAL; JANES; REMENCIUS, 2012)

A Tabela 1 sumariza as principais vantagens e desvantagens de cada tipo de desenvolvimento para uma melhor compreensão de cada abordagem. As informações contidas na tabela foram reunidas de vários trabalhos encontrados durante a pesquisa realizada.

Esses trabalhos são Corral, Janes e Remencius (2012), Holzinger, Treitler e Slany (2012), Charland e Leroux (2011), BARTH (2014) e El-Kassas et al. (2015).

	Vantagens	Desvantagens
Nativo	• Explora todas as capacidades do dispositivos • Maior performance • Oferece experiência nativa de usabilidade para o usuário • Integração próxima com o sistema e outros aplicativos	• Necessário um aplicativo por plataforma • Mais caro e mais difícil de manter, por ter que manter vários *apps* diferentes • Requer domínio de vários ambientes e linguagens para cada plataforma
Multiplataformas	• Só é preciso dominar uma linguagem e um ambiente • Desenvolve apenas um código e pode distribuir em várias lojas de *apps*, o que faz com que o alcance do *app* seja maior • Reduz custo, tempo e esforço de desenvolvimento e manutenção	• Não possui acesso a todas as funcionalidades do dispositivo • Menor performance comparada ao nativo, linguagens interpretadas e compiladas • Não possui a mesma usabilidade e experiência de uso das aplicações nativas • Não possui as últimas atualizações lançadas pelo *SO*, pois para cada atualização do *SO*, é preciso ser feita uma atualização no multiplataforma

Tabela 1 – Vantagens e desvantagens das abordagens de desenvolvimento nativo e multiplataforma

3 Metodologia

Neste capítulo são descritas as pesquisas feitas e os procedimentos realizados a fim de atingir os objetivos do trabalho, para detalhar como se deu o desenvolvimento do mesmo e, dessa forma, fornecer insumos para a sua correta reprodução ou continuação futuramente por outros pesquisadores.

Este trabalho baseia-se na comparação entre duas abordagens de desenvolvimento móvel, são elas, nativa e multiplataforma. Com isso, o objetivo principal é responder a seguinte questão problema:

Quais as vantagens e desvantagens do desenvolvimento multiplataforma de aplicações móveis em relação ao desenvolvimento nativo?

A partir da questão feita, as duas hipóteses formuladas acerca do tema, são apresentadas e explicadas a seguir.

- **H1**: As ferramentas multiplataforma estão evoluídas ao ponto de apresentarem mais vantagens do que desvantagens.
  - As ferramentas para desenvolvimento multiplataforma apresentam, atualmente, benefícios suficientes para serem consideradas uma opção viável ou talvez melhor que a abordagem nativa a depender da situação que envolve a aplicação a ser construída.
- **H2**: Ao longo do tempo, as possíveis desvantagens do desenvolvimento multiplataforma serão sanadas ou mitigadas.
  - Observando a evolução das ferramentas multiplataforma ao longo dos últimos anos, pode-se perceber que não apresentam mais os gargalos citados na literatura.

Para responder a questão problema será feita uma pesquisa sobre aplicações móveis e sobre as abordagens de desenvolvimento **nativa** e **multiplataforma**. Para isso, serão utilizados artigos e materiais *on-line* para definição de como é esse desenvolvimento e, com isso, será feita uma comparação entre os dois modos.

Após o estudo e a definição de como se dá esse desenvolvimento específico de *software*, será feita uma pesquisa de plataformas para desenvolvimento multiplataforma e então será selecionada uma ferramenta.

Encerrada a fase de pesquisas, será realizado um estudo de uso, que consistirá em replicar um aplicativo nativo para plataforma iOS utilizando a ferramenta selecionada, para comprovar empiricamente as diferenças pesquisadas e descritas no Capítulo 2 deste trabalho.

No decorrer da realização do estudo, que será descrito mais adiante no Capítulo 4, serão anotadas as diferenças na implementação das mesmas funcionalidades no ambiente nativo e no multiplataforma.

Após a conclusão do estudo de uso, será feita uma análise exploratória, descrita no Capítulo 5, a fim de melhorar o embasamento para a tomada de decisão entre o desenvolvimento nativo e multiplataforma. Para isso serão reunidas e comparadas em cada abordagem algumas funcionalidades comuns em aplicativos móveis muito conhecidos como Facebook, Snapchat, Uber e Google Maps, e que não serão exploradas no estudo de uso.

Para encerrar a análise dos dados será realizado um questionário com algumas perguntas objetivas a respeito das funcionalidades avaliadas a fim de confrontar os resultados obtidos com as opiniões de especialistas em desenvolvimento móvel atuantes no mercado e com isso avaliar se os profissionais possuem conhecimento do cenário atual do desenvolvimento móvel multiplataforma. Os resultados serão consolidados ao final do Capítulo 5.

Com isso, ambas as abordagens poderão ser comparadas de maneira empírica com embasamento teórico oriundo das pesquisas feitas anteriormente e espera-se, ao final deste trabalho, poder auxiliar desenvolvedores a escolher a abordagem de desenvolvimento mais adequada para cada caso.

3.1 Trabalhos relacionados

Durante a pesquisa conduzida, foram encontradas pesquisas relacionadas com o tema abordado por este trabalho, como Prezotto e Boniati (2014) que implementou um aplicativo utilizando a ferramenta multiplataforma *Apache Cordova*. A conclusão da pesquisa foi que a abordagem multiplataforma não é melhor que a nativa, mas que há situações onde cada abordagem se encaixa melhor.

Bezerra e Schimiguel (2016) também fez a implementação de um aplicativo utilizando uma ferramenta multiplataforma, nesse caso, o PhoneGap. O autor concluiu que há a necessidade de avaliar os requisitos da aplicação a ser criada para se tomar a melhor decisão, sobre qual abordagem optar. Cabe ressaltar, no entanto, que ambos os trabalhos não realizaram o desenvolvimento nativo para uma comparação mais precisa entre as duas formas de desenvolvimento tratadas neste trabalho.

Charland e Leroux (2011) faz um estudo comparativo teórico sobre desenvolvimento nativo e multiplataforma, no entanto, não faz uma comparação prática entre as duas abordagens. A conclusão que o autor chega é que o desenvolvimento híbrido é a solução mais provável no embate *nativo vs multiplataforma*, no entanto, há de se avaliar as necessidades da aplicação e do negócio.

Em Corral, Janes e Remencius (2012), é feito um levantamento sobre potenciais vantagens e desvantagens da abordagem multiplataforma sob a perspectiva de três *stakeholders* distintos: Usuário, Desenvolvedor e Provedor de Plataformas. O autor chegou à conclusão que os usuários são os principais guias do mercado e as suas preferências é que devem definir o futuro do desenvolvimento móvel. No entanto, não foi feito nenhum estudo prático comparando as duas abordagens alvo deste trabalho.

Embora a pesquisa tenha revelado que está sendo discutida qual a melhor abordagem para o desenvolvimento móvel, os estudo feitos, em sua maioria, são de três a cinco anos atrás, fazendo-se necessárias novas avaliações do desenvolvimento multiplataforma, devido à evolução das ferramentas e *frameworks* neste meio tempo. Não foram encontrados trabalhos recentes com a proposta de desenvolvimento de um mesmo *app* e comparação de funcionalidades utilizando as duas abordagens, para se fazer uma análise mais embasada das características de cada abordagem, suas vantagens e desvantagens.

3.2 Planejamento do Exemplo de Uso

Nas subseções a seguir, são apresentados os critérios usados pelos autores para a seleção do projeto a ser recriado e da ferramenta a ser utilizada para o estudo de uso, assim como um planejamento do desenvolvimento do projeto.

3.2.1 Seleção do projeto

Para a realização do exemplo de uso, foi escolhido o aplicativo Mini Farma, criado nativamente para a plataforma iOS. A escolha se deu por alguns fatores, listados a seguir:

- É de propriedade de um dos autores deste trabalho, o que facilita na obtenção e no entendimento do código;
- Explora funcionalidades nativas do dispositivo, como localização, que serão explicitadas mais adiante;
- É um aplicativo pequeno e simples, o que o torna ideal para ser refeito dentro do tempo de execução do trabalho;

3.2.2 Seleção das plataformas

Existem, atualmente, muitas ferramentas e *frameworks* para o desenvolvimento de aplicações móveis multiplataforma. Dentre elas, a escolha do Ionic se deu por alguns fatores, listados a seguir.

- É um *framework* sob uma licença de *software* livre;
- Possui um ambiente de apoio gratuito e robusto, com muitas ferramentas e tutoriais disponíveis;
- Utiliza AngularJS e Cordova que são dois *frameworks* estáveis, muito conhecidos e utilizados e mantidos por duas grandes empresas;
- Possui muitos *plugins* disponíveis e uma comunidade muito ativa para resolução de problemas;
- Consegue criar aparência e usabilidade próximas às das plataformas nativas;
- Possui uma arquitetura projetada para otimizar a performance nos dispositivos móveis;

3.2.3 Planejamento do desenvolvimento

Após definidos o projeto e a ferramenta que serão utilizadas, o próximo passo é planejar como será feito o desenvolvimento do *app* multiplataforma. A replicação do projeto passou por algumas etapas, citadas e detalhadas a seguir.

- Configuração do *framework* Ionic e suas dependências nas máquinas dos desenvolvedores;
- Definição e implantação de ambiente de desenvolvimento igual para ambos os desenvolvedores;
- Familiarização com *framework* Ionic e seu ambiente por meio de criação de projetos teste e tutoriais;
- Familiarização com AngularJS e sua arquitetura;
- Separação do aplicativo Mini Farma nas suas principais funcionalidades a serem recriadas;
- Relato de observações acerca das diferenças, dificuldades, facilidades, vantagens e desvantagens.

3.3 Planejamento da Análise Exploratória

A execução da comparação de funcionalidades se deu conforme a lista a seguir.

- Seleção de funcionalidades comuns em grandes aplicativos;
- Contextualização e explicação da funcionalidade por meio de textos;
- Pesquisar possibilidade de desenvolvimento da funcionalidade em Ionic;
- Caso seja possível, minerar ou desenvolver códigos para a funcionalidade no Ionic;
- Minerar ou desenvolver códigos nativos para a mesma funcionalidade;
- Avaliação das funcionalidades quanto à possibilidade e complexidade de desenvolvimento em cada abordagem;
- Criação e aplicação de questionário com base nos resultados obtidos das funcionalidades;
- Avaliar entendimento dos profissionais sobre o cenário atual de desenvolvimento móvel multiplataforma.

4 Exemplo de Uso

Com o intuito de comparar o desenvolvimento de aplicações móveis utilizando a abordagem nativa e multiplataforma, foi realizado um estudo preliminar. Para isso, um aplicativo feito em iOS nativo foi recriado utilizando o *framework* Ionic. A seguir, o aplicativo escolhido é detalhado em termos de funcionalidades e recursos utilizados.

4.1 Descrição do projeto selecionado

Mini Farma é um aplicativo criado inicialmente para a plataforma iOS que serve para controle dos medicamentos que as pessoas possuem em casa em suas "farmacinhas" particulares.

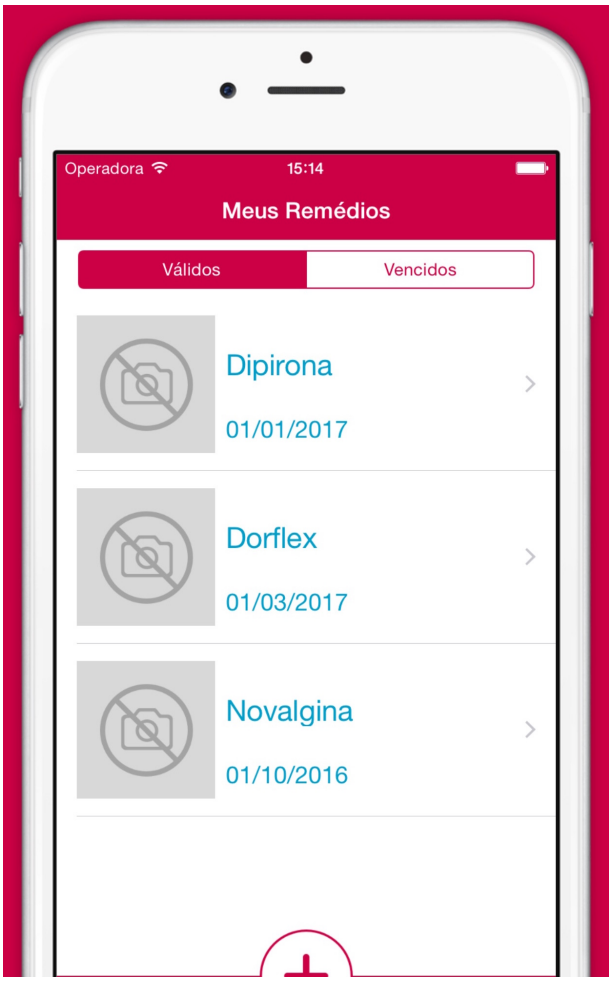

Figura 5 – Tela inicial do aplicativo Mini Farma.

O aplicativo utiliza recursos nativos do sistema, os quais são descritos a seguir.

- **Localização geográfica**: Define a posição geográfica do dispositivo que o usuário está utilizando para salvar a localização da farmácia onde ele comprou um medicamento para, caso seja necessário, explicar para alguém onde a farmácia fica. Também é possível com base nessa localização da farmácia e do usuário, traçar uma rota para levá-lo diretamente à farmácia em questão;

- **Notificações locais**: Diferente das notificações *Push*, as notificações locais são criadas e agendadas na central de notificações do dispositivo e o sistema se encarrega de entregá-las corretamente de acordo com parâmetros definidos pelo aplicativo. No caso do Mini Farma, as notificações são usadas para lembrar o usuário, na data e hora corretas, de que o mesmo deve tomar seus medicamentos;

- **Ligação**: As notificações podem conter ações que executam um determinado bloco de código dentro do aplicativo. No Mini Farma, uma das notificações possíveis é a de aviso de pouca quantidade ou remédio esgotado, no caso do medicamento ter acabado. Quando essa notificação é enviada, pode ser feita uma ligação diretamente pela ação da notificação para o número da farmácia cadastrado no aplicativo. Dessa forma, o usuário pode solicitar uma nova quantidade de medicamento diretamente com a farmácia;

- **Câmera**: Para facilitar a identificação dos medicamentos, é possível tirar uma foto com a câmera do dispositivo para cada remédio cadastrado. Além de uma foto para o medicamento em si, é possível tirar uma foto da receita dele, caso haja;

- **Rolo de câmera**: Se o usuário já tiver uma foto que o ajude a identificar o seu medicamento ou da receita do mesmo salva no rolo de câmera, é possível escolher a foto sem precisar tirar uma nova.

Com exceção da Ligação, todos os outros recursos nativos do sistema precisam necessariamente serem autorizados pelo usuário para funcionar. Caso o usuário não autorize, o aplicativo fica com funcionalidades reduzidas.

Como banco de dados foi usado o SQLite, por meio do *framework* externo e *open-source* *FMDB*[1], disponível no Github;

A arquitetura do sistema foi criada com base no padrão de projeto *MVC*, mostrado na Figura 6, possuindo uma camada *DAO (Data Access Object)* para comunicação com o banco de dados. No entanto, o *MVC* tradicional difere em alguns aspectos do *MVC* usado no iOS.

No iOS, o *MVC* trás uma controladora ligada à classe de *view*, que é chamado de *ViewController*, responsável por criar a ponte entre a interação do usuário pela *view* com

[1] <https://github.com/ccgus/fmdb>

as classes modelos. Dessa forma, as *ViewControllers* têm a responsabilidade de repassar as entradas do usuário para a *model* e controlar a *view* para apresentar os resultados que a modelo retornar.

Diferente do *MVC* tradicional, no qual a controladora apenas repassa as informações que estão entrando na fronteira da aplicação, as *ViewControllers* têm, além dessa responsabilidade, também a responsabilidade de instanciar e gerenciar a *view* no tocante à organização dos elementos e apresentação das informações.

Outra observação sobre as diferenças entre os dois *MVCs*, é que no *MVC* tradicional pode haver uma controladora apenas para todas as modelos ou uma controladora por modelo, mas nenhuma ligada diretamente a uma *view*, no entanto, no iOS há uma *ViewController* por *view*, podendo haver ainda uma controladora relacionada com cada modelo conforme o padrão tradicional do *MVC*.

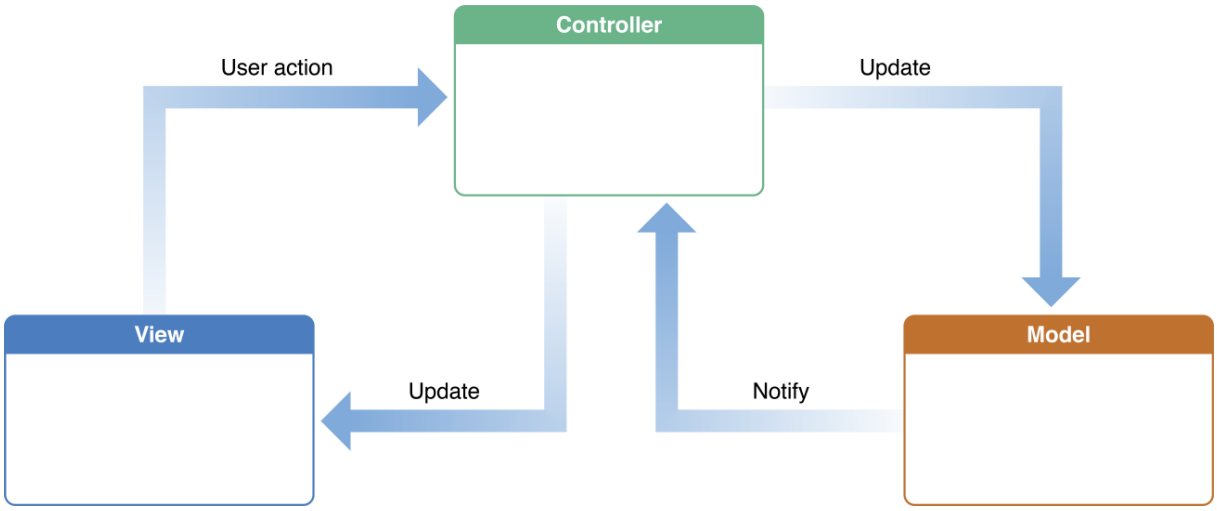

Figura 6 – Padrão *Model-View-Controller*. Fonte: Apple Documentation.

4.1.1 Ambiente de desenvolvimento

Nesta seção é apresentado todo o ambiente de desenvolvimento do *app* Mini Farma quando foi feito originalmente para a plataforma iOS. O projeto foi executado contando com uma equipe de dois integrantes com conhecimentos intermediários na plataforma iOS. Os detalhes técnicos são listados a seguir.

- **Máquinas**: Dois MacBook Pro Retina 13", processador *Intel Core i5*, 8 GB de *RAM*;
- **Sistema Operacional das máquinas**: *Mac OS X Yosemite* v10.10.5;
- **Sistema Operacional alvo do aplicativo**: iOS 8;

- ***IDE***: *Xcode* v6.4;
- **Linguagem de Programação**: *Swift* v1.3;
- **Banco de dados**: SQLite v3.8.10.2;
- **Ambiente de suporte**: *Plugin SQLite Manager* para *Mozilla Firefox*, para criação do banco de dados;

4.2 Desenvolvimento multiplataforma do projeto

A partir do aplicativo feito em iOS, foi construída uma versão utilizando o Ionic para as plataformas Android e iOS, disponível em um repositório aberto no Github[2]. O desenvolvimento foi dividido nas seguintes funcionalidades:

- **Listagem de Remédios e Alertas**;
- **Cadastro de Remédios**: sendo necessário o acesso a câmera fotográfica e galeria de fotos do dispositivo;
- **Cadastro de Farmácias**: sendo necessário o acesso à localização do dispositivo e ligação de voz;
- **Cadastro de Alertas**: sendo necessário o acesso à central de notificações locais do dispositivo;

Para a criação dessa versão do *app*, a equipe foi alterada, mas permaneceu com dois integrantes, que são os autores deste trabalho. No entanto, os integrantes não possuíam qualquer conhecimento prévio nos *frameworks* Ionic, AngularJS e Cordova e apenas possuíam um conhecimento básico em *HTML*, *CSS* e *JavaScript*. Os detalhes técnicos são listados a seguir.

- **Máquinas**: Dois MacBook Pro Retina 13", processador *Intel Core i5*, 8 GB de *RAM*;
- **Sistema Operacional das máquinas**: *Mac OS X El Capitan* v10.11.5;
- **Sistema Operacional alvo do aplicativo**: iOS 9.3 e Android 6.0.0;
- ***IDE***: *WebStorm* v2016.1.1;
- ***Frameworks***:

[2] <https://github.com/crossdev-tcc/ionic-apps>

  - **Ionic**: *Copenhagen* v1.2.4[3];
  - **AngularJS**: *foam-acceleration* v1.4.3[3];
  - **Cordova**: v6.0.0;

- **Linguagem de Programação**: *JavaScript* v1.7.
  - *HTML* 5 e *CSS* 3 foram usados paralelamente para a criação da interface gráfica do aplicativo;

- **Banco de dados**: SQLite v3.8.10.2;

- **Ambiente de suporte**: Google Chrome, para *debug* do aplicativo;

- **Simuladores**:
  - **iOS**: iPhone 6S Plus
  - **Android**: Nexus 7

4.2.1 Relato de desenvolvimento

É apresentado, a seguir, o relato das experiências obtidas durante o desenvolvimento com Ionic, dividido nas funcionalidades já descritas na Subseção 3.2.3 deste trabalho, bem como feita a comparação entre o desenvolvimento nativo para iOS e o multiplataforma Ionic. Há, ainda, um tópico de observações gerais pertinentes à implementação do *app* como um todo.

- **Listagem de Remédios e Alertas**;
  - Para a tela inicial do *app*, a lista de remédios e alertas, foi preciso realizar uma pequena alteração em relação ao original, para considerar, agora, dois sistemas operacionais diferentes (iOS e Android). Foi preciso criar duas *views* para a mesma tela, pois o componente de abas (conhecido como *tabs*), é diferente no iOS e no Android. Com isso, quando foi feito pensando-se no iOS do Ionic, ele não correspondia muito bem ao padrão de interface do Android, por isso foi preciso criar duas telas separadas e fazer uma verificação se o dispositivo é iOS ou Android. Após feita a verificação de dispositivo, o próprio Ionic se encarrega de definir a aparência do componente para o padrão do sistema no qual o aplicativo está sendo executado.

[3] Durante a execução deste trabalho, já haviam sido disponibilizadas as versões 2 do Ionic e do AngularJS, no entanto, os autores optaram por não utilizá-las por ainda se tratarem de versões *beta* e portanto instáveis, o que poderia impactar negativamente no desenvolvimento do projeto.

– Para a criação da lista de remédios e alertas que o usuário possui cadastrados, o componente usado no iOS é chamado de *UITableView*. No Ionic, o componente similar é o conjunto das diretivas *ion-list* e *ion-item* do Ionic com a diretiva *ng-repeat* do AngularJS. É muito simples de ser implementada, possuindo documentação ampla e exemplos de uso nos *sites* oficiais dos *frameworks*.

– Para a realização das filtragens dos remédios por data de validade, e dos alertas por ativos e inativos, bastou usar o componente *filter* padrão do AngularJS. Dessa forma, foi possível mostrar apenas as informações que deveriam ser apresentadas. Novamente, possuindo muitos exemplos e ajuda na documentação do *framework*.

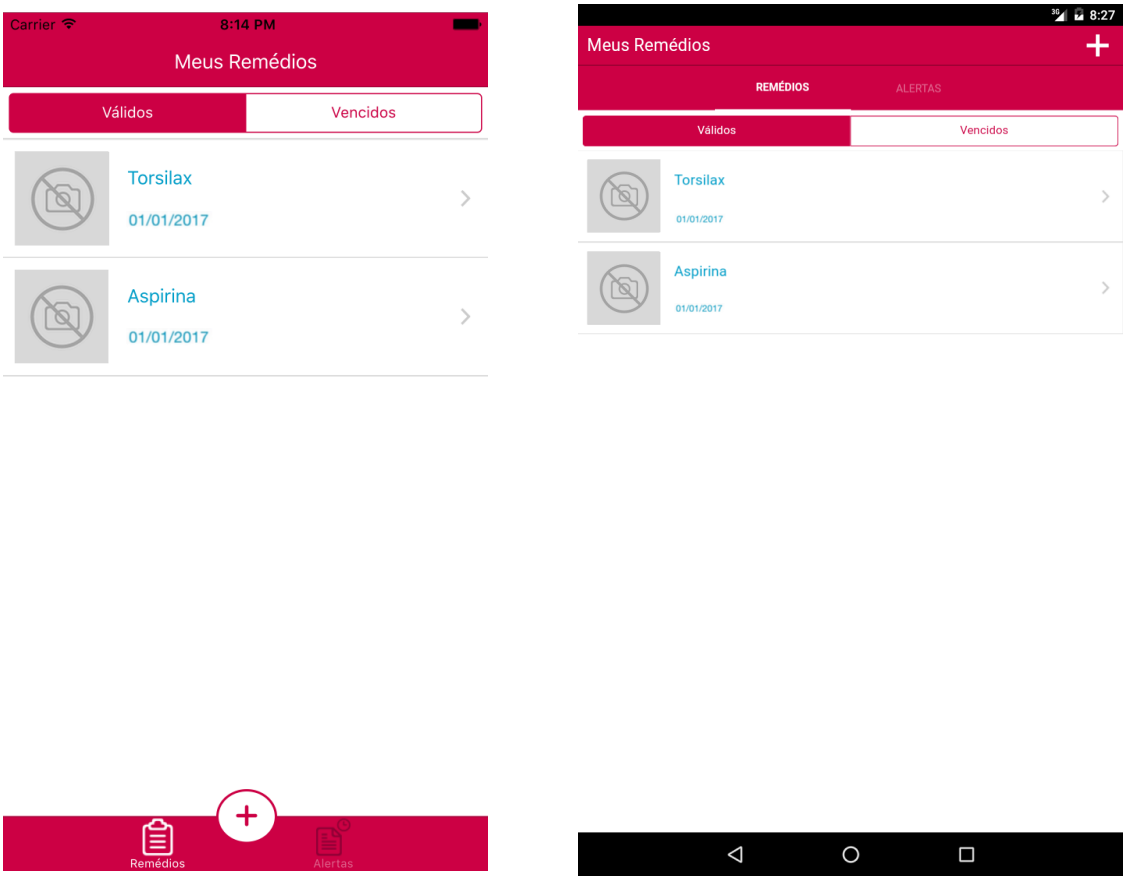

Figura 7 – Telas da lista de remédios (Ionic iOS à esquerda *versus* Ionic Android à direita).

- **Cadastro de Remédios**;

  – No cadastro de remédios há o uso da câmera do dispositivo para tirar uma foto do medicamento e para registrar a prescrição médica. Opcionalmente, pode-se selecionar as fotos a partir da biblioteca de imagens do dispositivo. Com o uso do *plugin* Cordova de acesso à câmera, essa foi uma tarefa simples e que funcionou corretamente em ambas as plataformas, Android e iOS.

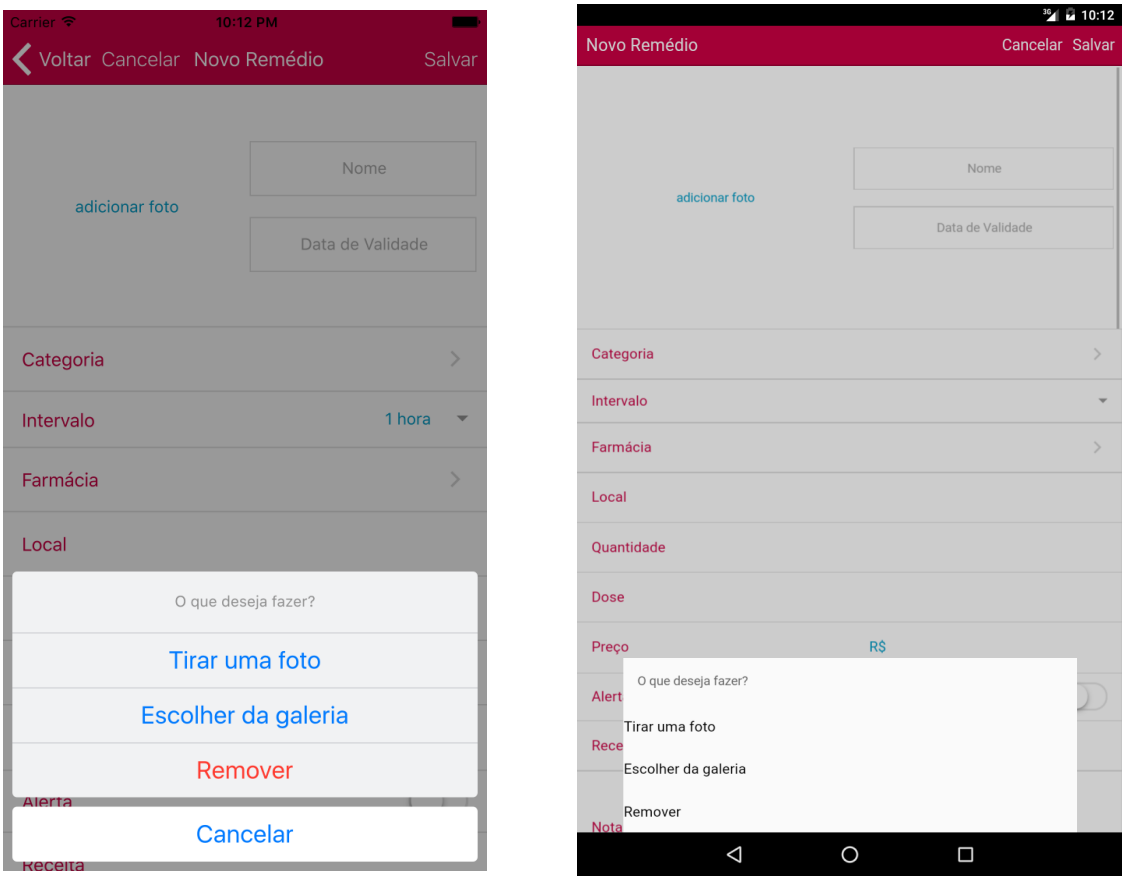

Figura 8 – Telas de cadastro de foto de remédio (Ionic iOS à esquerda *versus* Ionic Android à direita).

– Para que o usuário opte entre tirar uma nova foto e escolher da galeria do dispositivo, foi utilizado um componente conhecido no iOS como *UIActionSheet*. Com o uso de um único código, o Ionic possibilitou a geração de um componente exatamente igual ao nativo do iOS e de um correspondente no Android. Na Figura 8 é possível observar as diferenças de interface, as quais foram adequadas pelo Ionic para cada plataforma.

– Outro componente utilizado e propriamente adaptado pelo Ionic para as plataformas foi o denominado no iOS como *UIPickerView*. Ele foi utilizado para a escolha de intervalo de uso do medicamento e é apresentado na Figura 9.

– No formulário de cadastro há a opção de selecionar uma categoria para o medicamento, para isso, o usuário é direcionado a uma página que contém a lista de categorias disponíveis, seleciona alguma delas ou adiciona uma nova e é redirecionado à página de cadastro, com o campo de categoria preenchido com a escolha feita. Para isso é necessário o envio de dados de uma tela para a antecessora. No aplicativo nativo, essa funcionalidade foi implementada com o uso dos chamados *Protocols and Delegates*. Já no aplicativo multiplataforma, não foi encontrada uma correspondência aos protocolos do iOS, foi, então,

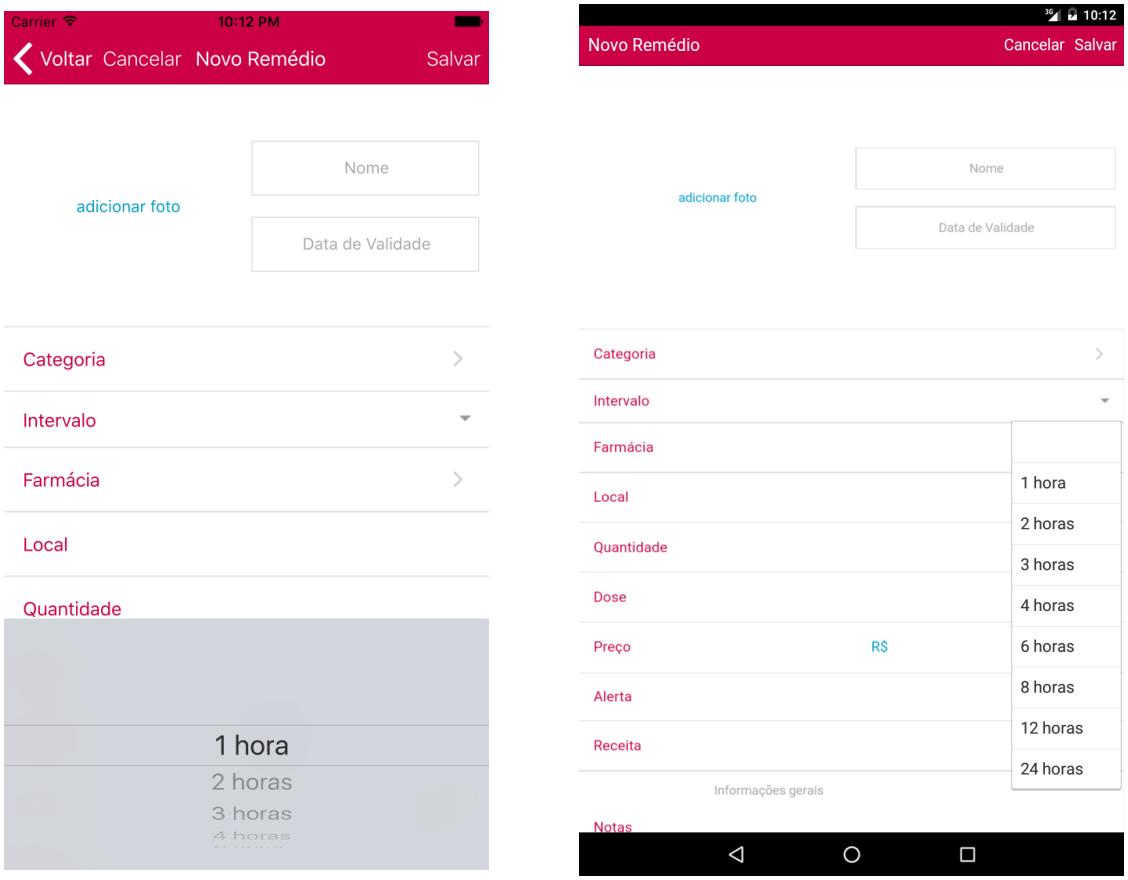

Figura 9 – Telas de cadastro de intervalo de remédio (Ionic iOS à esquerda *versus* Ionic Android à direita).

utilizada uma *factory* para armazenar temporariamente o valor da categoria. Observou-se que o nível de dificuldade de ambas as abordagens é semelhante, não apresentando grande dificuldade de implementação.

– Para a criação de uma nova categoria de remédios, foi utilizada uma interação por meio de uma caixa de diálogo, também conhecida como alerta. Para isso, utilizou-se os chamados *popups* do Ionic. Apesar de facilmente implementados e personalizáveis, esses componentes não se assemelham às caixas de diálogo nativas do iOS, *UIAlertView* e do Android, *AlertDialog*. O alerta criado é apresentado na Figura 10. Posteriormente, foi verificado que há no Cordova uma *API* para caixas de diálogo que, diferente da disponibilizada pelo Ionic, gera uma interface adequada aos padrões de ambas as plataformas. Um exemplo de alerta gerado pelo Cordova e idêntico à uma das plataformas, neste caso o iOS, é apresentado na Figura 11.

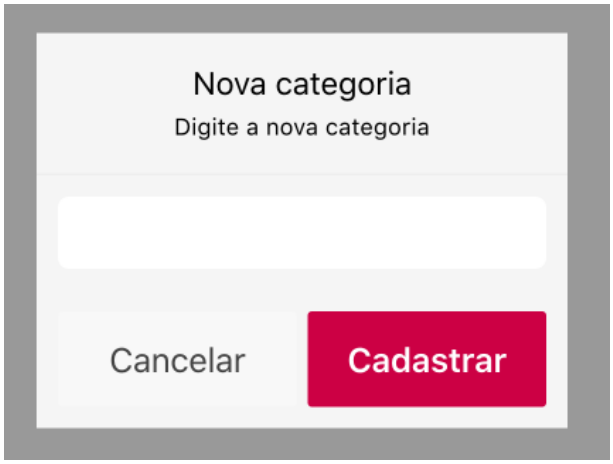

Figura 10 – Tela de alerta - Ionic.

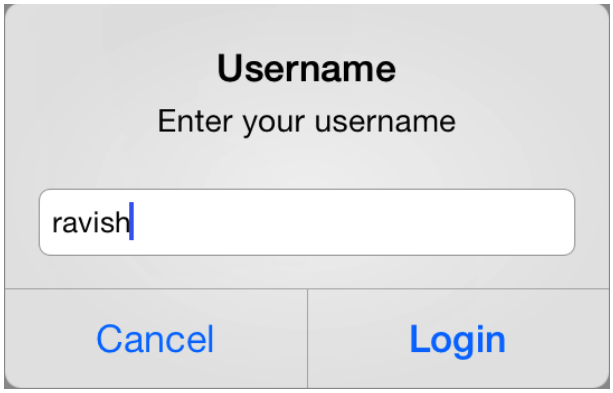

Figura 11 – Alerta Cordova - iOS nativo. Fonte: (FRAMEWORK, 2016).

– Para a seleção do local de armazenamento do remédio foi utilizado um componente inexistente no iOS, que é uma caixa de seleção. O resultado não agradou muito por ter ficado com aspecto de componente *web* e por não ter lembrado nenhum dos componentes nativos do iOS. Já para a plataforma Android o resultado foi satisfatório. O componente é apresentado na Figura 12.

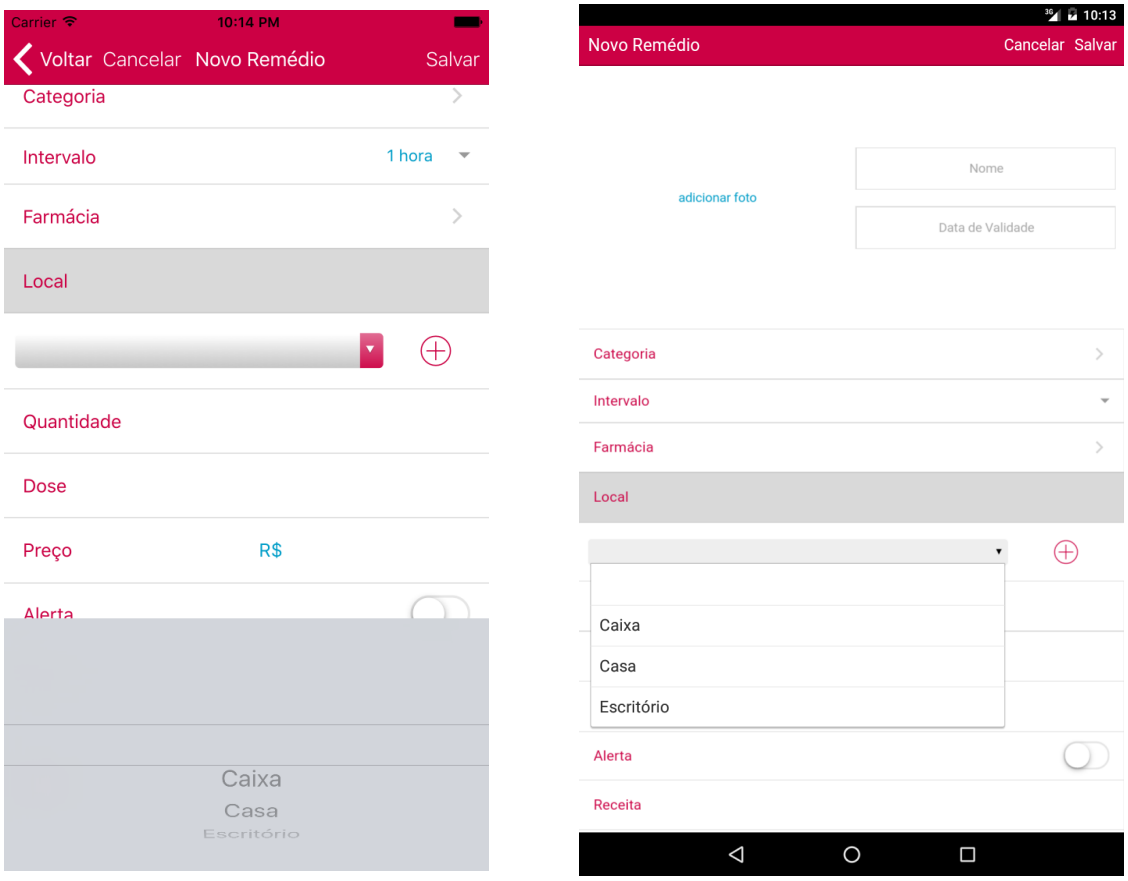

Figura 12 – Tela de cadastro de local do remédio (Ionic iOS à esquerda *versus* Ionic Android à direita).

– Em relação ao banco de dados, utilizou-se a *API* Cordova SQLite. Como o banco de dados construído nativamente também foi feito utilizando o SQLite, foi possível aproveitar os *scripts* de definição e manipulação de dados. Não houveram complicações no uso da *API* e facilmente foram encontrados exemplos de uso na internet que auxiliaram na estruturação da camada de comunicação do aplicativo com o banco de dados.

– De forma geral, o resultado da tela de cadastro de remédios foi satisfatório e a interface construída ficou semelhante à tela do aplicativo nativo, apresentado na Figura 13. A tela construída com Ionic é apresentada nas Figuras 8 e 12.

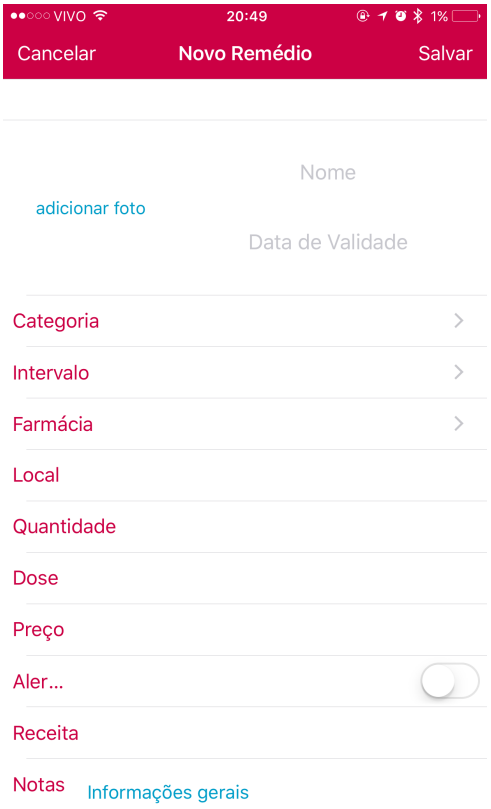

Figura 13 – Tela de cadastro de remédio - iOS nativo.

- **Cadastro de Farmácias**;

  – Para cadastrar uma nova farmácia, o usuário deve marcar no mapa onde a farmácia se encontra. Para a utilização do mapa no Ionic, foi utilizada a *API Maps* do Google, diferentemente da *API MapKit* do iOS. Não houveram grandes dificuldades na criação do mapa e renderização do mesmo na tela. A dificuldade de implementação é similar a do iOS nativo. Apenas realizando pesquisas e estudos na documentação da *API* do Google, já foi possível a elaboração do que precisava ser feito, no caso, mostrar o mapa e fornecer a opção de mover o pino para um local específico e capturar as coordenadas daquele ponto. No caso, foi necessário utilizar a diretiva *map* para renderização do mapa na tela e definir o *callback* de reposicionamento do cursor e captura da localização. Na Figura 14 é possível ver as diferenças entre o mapa do *MapKit* do iOS e da *API Maps* do Google.

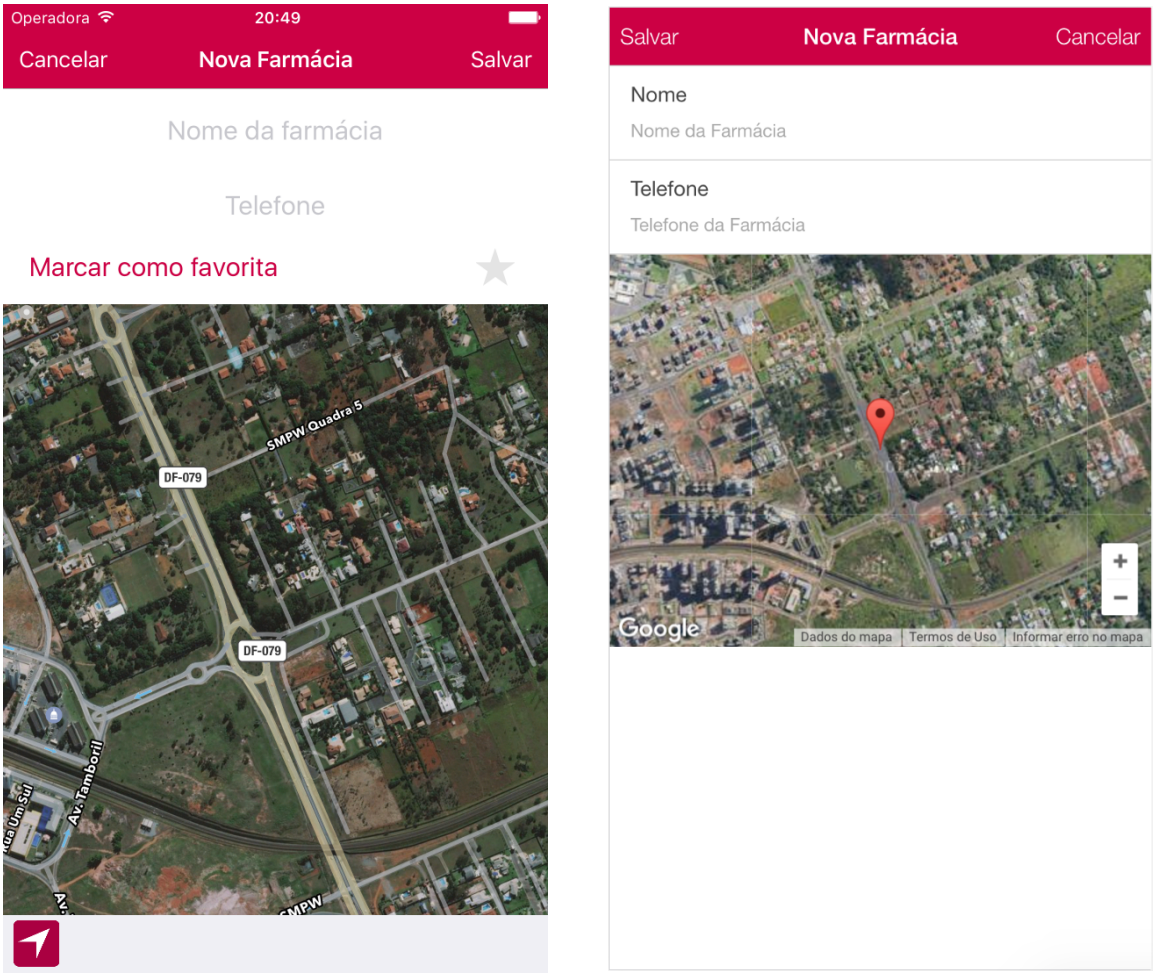

Figura 14 – Tela de adicionar farmácia (iOS nativo à esquerda *versus* Ionic iOS à direita).

- **Cadastro de Alertas**;

  – Para o cadastro de um novo alerta, é preciso selecionar data e hora que o usuário será alertado. Para isso, o Ionic 1 não possui o componente adequado. Com isso, foi necessário procurar e utilizar um *framework* terceiro. Os *frameworks* escolhidos foram o *ionic-datepicker*[4] e *ionic-timepicker*[5], respectivamente para seleção de data e hora. Ambos foram desenvolvidos pelo mesmo usuário no Github e instalados no projeto do aplicativo via *CLI*. Nas próprias documentações dos *frameworks* existem exemplos de uso suficientes para guiar a implementação das funcionalidades. Vale ressaltar que o componente *DatePicker* do iOS permite a seleção de data e hora ao mesmo tempo, enquanto nos *frameworks* utilizados, a seleção de data e hora é separada em dois componentes distintos, gerando uma pequena alteração na interface gráfica da tela de cadastro de alertas quando comparada com a do aplicativo original. As diferenças

[4] <https://github.com/rajeshwarpatlolla/ionic-datepicker>
[5] <https://github.com/rajeshwarpatlolla/ionic-timepicker>

entre o original e Ionic podem ser vistas na Figura 15. O mesmo *framework* para seleção de data também foi utilizado na tela de cadastro de remédios para definir a data de validade do mesmo.

– Para a criação dos alertas é preciso cadastrar notificações locais na central de notificações do dispositivo. Realizar essa tarefa teve o mesmo nível de dificuldade da implementação no iOS nativo. Apenas foi instalado um *plugin* do Cordova para gerenciar a central de notificações e agendar notificações com as datas e horas corretas para serem entregues.

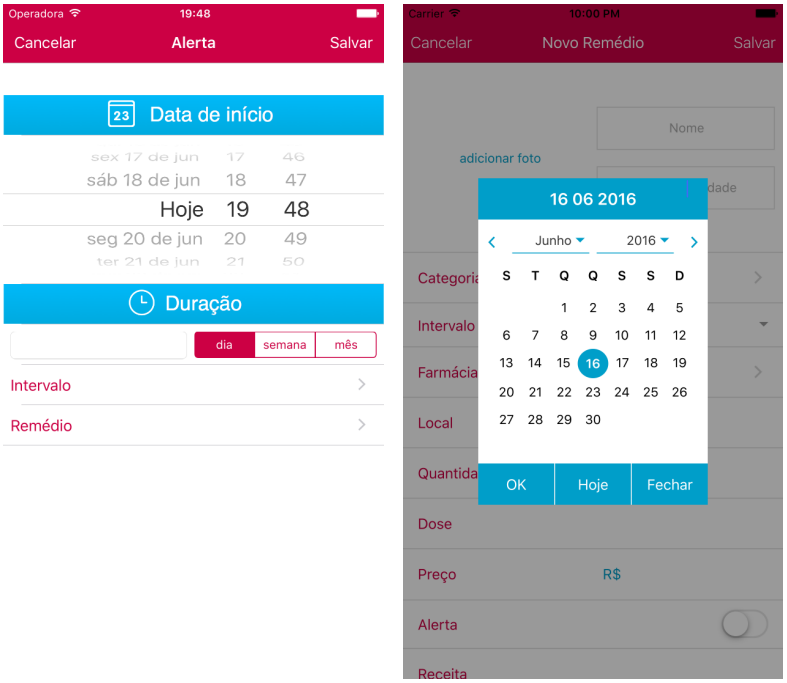

Figura 15 – Seleção de data e hora dos alertas (iOS nativo à esquerda *versus* Ionic iOS à direita).

- **Observações gerais**;

  – Houve um problema no carregamento da lista inicial de remédios. Por ficar na primeira tela do aplicativo, acontecia de o acesso ao banco de dados para obtenção da lista ocorrer antes do total carregamento do Ionic, o que acarretava em falha na execução do aplicativo. Esse problema não existiu na criação do *app* original e para contorná-lo foram feitas pesquisas na comunidade para saber como resolver. Umas das soluções sugeridas era causar um *delay* proposital no aplicativo para que desse tempo do Ionic carregar completamente, mas foi descartada para não impactar negativamente na performance do *app*. A solução utilizada foi fazer uma verificação a cada transação realizada no banco de dados para que só sejam efetuadas caso o Ionic esteja pronto para ser usado e apresentado.

- O Ionic apresenta uma útil funcionalidade que permite o *debug* dos aplicativos pelo terminal, porém, a partir da versão 9 no iOS não é possível realizar essa ação, enquanto que no Android continua funcionando adequadamente. Para corrigir o problema, usuários do Ionic recomendam a modificação de uma configuração no arquivo *.plist* do projeto iOS, para desabilitar a opção *App Transport Security* e permitir requisições que não sejam *HTTPS*. No entanto, ao modificar essa opção há o risco do aplicativo ser recusado no momento da sua publicação, portanto é necessário lembrar de reabilitá-la antes de submeter para a loja de aplicativos. Caso opte-se por não realizar essa alteração, para debugar é preciso um computador com *Xcode* ou é possível também utilizar os navegadores como *Google Chrome* ou *Safari* no modo desenvolvedor para debugar utilizando o console *JavaScript* dos navegadores.
- Alguns erros do *JavaScript* não são acusados pelo terminal do Ionic, dificultando a tarefa de *debug* da aplicação. Com isso, em um momento de dificuldade em achar um problema que estava acontecendo, algumas pesquisas foram realizadas na comunidade *on-line* e em um fórum de discussão sobre Ionic foi dada uma dica sobre o uso de alertas do *JavaScript* para mostrar erros para o desenvolvedor e com isso facilitar na tarefa de debugar o aplicativo.
- A diretiva *ng-class* do AngularJS não funciona em conjunto com a diretiva *ion-nav* do Ionic. No entanto, isso não foi achado nas documentações, nem do Ionic e nem do AngularJS. Novamente, recorremos à comunidade e foi dito que esses componentes não funcionam corretamente juntos.
- Por mais que os autores não possuíssem conhecimentos avançados em *HTML*, *CSS* e *JavaScript*, vale ressaltar que o conhecimento inicial nessas tecnologias ajudou muito durante o desenvolvimento do *app*.
- O Cordova e o Ionic possuem documentações bem completas, apresentando exemplos de uso de códigos. A comunidade é ativa e disponibiliza *APIs* adicionais e exemplos de códigos em repositórios online.

Apesar das desvantagens citadas na Tabela 1, não houveram limitações quanto ao uso dos recursos necessários para o projeto ou problemas de performance que impactassem nos requisitos do aplicativo. Foi possível construir apenas um código e gerar dois executáveis de plataformas diferentes, o que poderia acarretar em redução de custos e tempo de desenvolvimento. Além disso o aplicativo desenvolvido apresentou aparência e usabilidade semelhantes ao nativo, adaptando-se a cada plataforma alvo seguindo seus padrões específicos de interface de usuário.

5 Análise Exploratória

Foram selecionadas algumas funcionalidades muito comuns em aplicativos para serem comparadas entre iOS, Android e Ionic, com o objetivo de melhorar o embasamento para a tomada de decisão entre o desenvolvimento nativo e multiplataforma. Importante ressaltar que para o aplicativo Ionic utilizar os recursos do celular, tanto *hardware* quanto *software*, é sempre necessária a utilização de *plugins* do Cordova, ou seja, sempre que um novo recurso for desenvolvido para um dispositivo, há um tempo de espera para a criação do *plugin* necessário para utilizar o novo recurso, e se o *plugin* for descontinuado, eventualmente poderá quebrar a aplicação e prejudicar a experiência do usuário. A seguir são apresentadas as funcionalidades e, em alguns casos, trechos de códigos, minerados de repositórios públicos e criados pelos autores, comentados para facilitar o entendimento das funcionalidades e diferenças entre cada plataforma e abordagem de desenvolvimento.

5.1 Funcionalidades analisadas

As funcionalidades alvo da análise e comparação são listadas a seguir e serão explicadas e comparadas nas Subseções 5.1.1 à 5.1.11.

- Login com Facebook;
- Consumo de Web Services;
- Detecção de força do toque;
- Leitor Biométrico;
- Extração de metadados de arquivos;
- Envio de e-mail e SMS;
- Widgets;
- Assistentes Pessoais;
- Smartwatches;
- Câmeras customizadas;
- Detecção facial;

5.1.1 Login com Facebook

Independente da plataforma, sempre deve haver um passo anterior ao desenvolvimento desta funcionalidade que é o registro do aplicativo a ser desenvolvido na plataforma para desenvolvedores do Facebook[1]. O registro do aplicativo deve ser feito para cada plataforma que se deseja atingir, por exemplo, iOS e Android, utilizando para isso, o nome do pacote do aplicativo em cada plataforma.

Com esta estapa concluída, pode-se iniciar o desenvolvimento desta funcionalidade utilizando para isso o *SDK* do Facebook para iOS e Android. No caso do Ionic, existem alguns *plugins* que podem ser utilizados, como por exemplo o *$cordovaOauth*[2], disponível dentro do próprio ngCordova, que permite a conexão com vários provedores de serviços, tais como Google, Github, Facebook e Linkedin, por exemplo.

Além desse, existem outros meios de se conectar aos provedores, por exemplo, utilizando o Firebase[3] do Google. Desta forma, é possível acessar os dados públicos do usuário como nome, e-mail e foto de perfil para realizar um cadastro e *login* no aplicativo. Em todas as três plataformas citadas (iOS, Android e Ionic), a complexidade é similar, havendo apenas diferenças inerentes a cada linguagem e plataforma.

Vale ressaltar, que o *login* com redes sociais, nada mais é do que a obtenção dos dados do usuário de uma plataforma terceira por meio do protocolo *OAuth*, ou seja, não são criadas sessões a partir das plataformas terceiras, apenas os dados do usuário na outra plataforma são acessados para que não haja a necessidade do usuário informá-los novamente. A partir dos dados obtidos, o aplicativo se encarrega de realizar o cadastro e *login* do usuário de acordo com as regras de negócio do aplicativo. As Figuras 16 e 17 apresentam dois trechos de código no iOS e no Ionic para *login* com o Facebook. No iOS foi utilizado o *SDK* nativo do Facebook, já no Ionic, foi utilizado o *Firebase*.

[1] <https://developers.facebook.com>
[2] <http://ngcordova.com/docs/plugins/oauth/>
[3] <https://www.firebase.com>

```
// Instanciando gerenciador de login do Facebook e
// definindo como a janela de login do Facebook será exibida
let fbLoginManager = FBSDKLoginManager()
fbLoginManager.loginBehavior = FBSDKLoginBehavior.Web
// Requisitando e-mail e perfil público dos dados do usuário no Facebook
fbLoginManager.logInWithReadPermissions(["email","public_profile"],
                                        fromViewController: self) { (result, error) in
    // Caso não haja erro nesta requisição
    if (error == nil) {
        // Caso o token de acesso do Facebook tenha sido obtido com sucesso
        if FBSDKAccessToken.currentAccessToken() != nil {
            print("Logged in with facebook!\n \(result)")
            self.facebookCurrentToken = FBSDKAccessToken.currentAccessToken().tokenString
            // Requisitando o e-mail do usuário
            // para checar se o usuário já fez login anteriormente
            let request = FBSDKGraphRequest(graphPath:"me", parameters: ["fields":"email"]);

            request.startWithCompletionHandler {(
                connection : FBSDKGraphRequestConnection!,
                result : AnyObject!,
                error : NSError!) -> Void in
                if error == nil {
                    let r = result as! [String: String]
                    // Checando se usuário já fez login
                    User.sharedInstance.checkUserEmail(
                        r["email"]!,
                        completionHandler: { (response, user) in
                        // Caso o usuário já tenha feito login antes
                        if response == "registered" {
                            // Faz login na API criada
                            self.facebookLogin("\(user?.addresses_id)")
                        }else{
                            // Caso contrário, é necessário cadastrar o usuário na API
                            // com as informações obtidas do Facebook
                            let sb = UIStoryboard(name: "Main", bundle: nil)
                            let vc =
                                sb.instantiateViewControllerWithIdentifier("CEPViewController")
                                as! CEPViewController
                            vc.cameFrom = "SocialLogin"
                            self.presentViewController(vc, animated: true, completion: nil)
                        }
                    })
                } else {
                    print("Error getting information \(error)");
                }
            }
        }
    }
}
```

Figura 16 – Requisição de dados do Facebook no iOS

```
.controller('loginCtrl', function($scope, $ionicModal, $state, $ionicHistory, Profile, $http) {
    // Definindo URL da API no Firebase
    $scope.ref = new Firebase("https://partiuapp.firebaseio.com");
    // Definindo função de login com Facebook e callback para tratamento da resposta
    $scope.fbLogin = function () {
        // A string "facebook" indica qual o provedor se deseja conectar, no caso, Facebook
        // Se houvesse a necessidade de conexão com Twitter, por exemplo,
        // bastaria alterar a string para "twitter" e definir um callback diferente
        $scope.ref.authWithOAuthPopup("facebook", $scope.loginCallback);
    };

    $scope.loginCallback = function(error, authData) {
        if (error) {
            console.log("Login Failed!", error);
        } else {
            // Caso a requisição tenha dado certo, são obtidos os dados do Facebook do usuário
            // e então armazenados na variável data
            var data = authData.facebook;
            // Criando um perfil, a partir dos dados obtidos
            Profile.setUser(data.displayName, data.email, data.accessToken,
            data.cachedUserProfile.gender, data.profileImageURL, data.id,
            data.cachedUserProfile.link);
            // Criando um dicionário com os dados do usuário para enviar parar a API salvar
            // e então serem salvos no banco de dados
            var requestData = {"user": {"name": data.displayName, "email": data.email,
            "facebook_id": data.id,"photo_url": data.profileImageURL, "token": data.accessToken,
            "gender": data.cachedUserProfile.gender, "link_profile": data.cachedUserProfile.link }
            };
            // Enviando um requisição POST para um endpoint na API com o dicionário dos dados do usuário
            $http.post(AppSettings.baseApiUrl + '/api/users', requestData)
            .success(function(response) {
                // Após salvar os dados do usuário corretamente na API,
                // atualiza-se a informação do ID do Facebook no service Profile
                $http.get(AppSettings.baseApiUrl + '/api/get_user_id', {params:{"facebook_id": data.id}})
                .then(function(response) {
                    Profile.updateBackendId(response.data);
                }, function(error) {
                    // Caso haja algum erro, trata-se aqui
                });
            });
            // Redirecionando para tela inicial do aplicativo
            $state.go('menu.home');
        }
    };
})
```

Figura 17 – Requisição de dados do Facebook no Ionic. Fonte: Baseado em Github[4]

Vale ressaltar que o *plugin* do Cordova é genérico para vários provedores, ou seja, com o mesmo código é possível requisitar os dados em vários serviços distintos. Já nos *SDKs* nativos, isso não ocorre. Caso haja a necessidade de obtenção de dados de vários provedores, como por exemplo, no caso de *login* com Facebook e Google, na abordagem nativa é preciso utilizar a *SDK* do Facebook para iOS e Android e a *SDK* do Google para iOS e Android. O mesmo não ocorre no multiplataforma, pois com o mesmo *plugin* é possível fazer a conexão com vários serviços apenas alterando uma *string*, o que pode tornar o desenvolvimento mais simples que na abordagem nativa.

[4] <https://github.com/fga-gpp-mds/2016.1-Partiu_frontend>

5.1.2 Consumo de Web Services

Como o poder computacional dos dispositivos móveis, embora elevado, ainda não é comparável ao dos servidores, muitos aplicativos utilizam um modelo cliente/servidor, no qual as operações mais custosas computacionamente são processadas no lado servidor e não no cliente. Além disso, a persistência dos dados da aplicação dificilmente é feita apenas localmente nos dispositivos para evitar perdas de dados no caso de roubos, extravios ou danos ao aparelho, facilitar a migração para outros dispositivos e o uso em outras plataformas. Com isso, é necessário que os aplicativos estejam preparados para se comunicarem com *Web Services*. Para realizar essa comunicação, cada plataforma oferece alguns meios nativos como classes e *frameworks*, no entanto, existem *frameworks* de terceiros que são muito utilizados e conhecidos na comunidade de desenvolvedores tanto para comunicar com os *Web Services*, como para lidar com JSON e XML usados na comunicação entre aplicativo/servidor.

No Android, existe o *Volley*[5], da própria Google para facilitar a envio e recebimento dos dados. Assim como no iOS, define-se o método da requisição, os parâmetros a serem enviados e dentro de um *Listener* receber a resposta do servidor e tratar de acordo com as necessidades do aplicativo. Vale ressaltar que o *Volley*, no momento da realização deste trabalho, não é nativamente preparado para enviar dados, apenas receber, de maneira que deve-se criar uma classe customizada de requisição que adiciona os parâmetros a serem enviados na requisição e faz o tratamento da resposta, para só então ser criada uma requisição que permite o envio de dados. No exemplo da Figura 18, a classe criada chama-se *CustomJSONObjectRequest* e sua implementação não se encontra na imagem.

Para realizar essa atividade no Ionic, o AngularJS dispõe do *plugin ngResource* muito conhecido e utilizado para interação com *web services*. Ele trás o objeto *$resource* que recebe uma *URL* e parâmetros da requisição de maneira muito similar à abordagem nativa, apenas distinguindo em termos de linguagem e ambiente de programação. Assim como nas abordagens nativas, também é possível receber o retorno da requisição e tratá-lo dentro de uma *$promise*, cujo modo de uso se assemelha ao *completion* do iOS e ao *Listener* do Android. A Figura 19 apresenta um trecho de código implementando o consumo de um *web service* privado no Ionic.

No iOS é possível realizar essa comunicação utilizando algumas classes nativas do sistema como por exemplo, *NSURL*, *NSData* e *NSJSONSerialization*, informando uma *URL* na qual o *app* irá se conectar para receber ou enviar algum dado para o servidor, no entanto, toda essa comunicação e tratamento das respostas deverá ser feita manualmente. Para facilitar, existem alguns *frameworks* terceiros como *AlamoFire*[6] e *SwiftyJSON*[7] que

[5] <https://developer.android.com/training/volley/index.html>
[6] <https://github.com/Alamofire/Alamofire>
[7] <https://github.com/SwiftyJSON/SwiftyJSON>

abstraem essa comunição e tratamento dos dados em JSON, respectivamente, tornando o código mais simples e limpo. Ao utilizá-lo, o programador apenas indica a *URL*, o método da requisição (*POST, GET, PUT, DELETE, etc*) e os parâmetros que deseja enviar ao servidor. Após isso, basta receber a resposta na forma de *JSON* dentro de um *completion* do *Alamofire*, e utilizar os dados de acordo com as necessidades do aplicativo. Um trecho de código exemplificando o uso do *AlamoFire* e *SwiftyJSON* é apresentado na Figura 20.

```
private void addJoin() {
    /**
        Iniciando uma requisição do Volley para adicionar um novo produto
        tipo de requisição: POST
        url: string de uma URL que indica o endpoint da API
        dicionário de parâmetros da requisição em um método chamado getParams()
    */
    CustomJSONObjectRequest jsonObjectRequest = new CustomJSONObjectRequest(Request.Method.POST,
                                                url,
                                                getParams(),
                                                new Response.Listener<JSONObject>() {
        @Override
        public void onResponse(JSONObject response) {
            try {
                // Fazendo um parser da resposta para JSON
                JSONObject myProduct = response.getJSONObject("product");
                productId = myProduct.getInt("id");
            } catch (Exception e) {
                e.printStackTrace();
                System.out.print(response);
            }
        }
    },new Response.ErrorListener() {
        // Tratamento de erro, caso ocorra
        @Override
        public void onErrorResponse(VolleyError error) {
            new MaterialDialog.Builder(NewJoinFragment.this)
            .title("Desculpe, houve um erro ao adicionar seu produto.")
            .positiveText("OK")
            .negativeText("Tentar novamente")
            .show();
        }
    });
    // Tag usada para cancelar a requisição caso seja necessário
    jsonObjectRequest.setTag("newObject");
    // Adicionando a requisição criada a uma fila de requisições a serem executadas
    rq.add(jsonObjectRequest);
}
// Método usado para retirar dados da view e mapeá-los em um objeto Map com
// as respectivas chaves
protected Map<String,String> getParams (){
    Map<String,String> params = new HashMap<String, String>();
    params.put("name",titleField.getText().toString());
    params.put("description",descriptionField.getText().toString());
    params.put("price", priceField.getText().toString());
    params.put("category_id",String.valueOf(category_id));
    params.put("subcategory_id",String.valueOf(subcategory_id));

    return params;
}
```

Figura 18 – Requisição utilizando Volley no Android

```
// Criação de um factory de usuário para consumo de uma API
.factory('UserAPI', function($resource) {
    // Definindo URL a ser consultada, assim como o parâmetro que
    // poderá ser enviado junto à requisição, no caso, o ID do usuário
    return $resource(AppSettings.baseApiUrl + "/api/users/:userId", null, {
        // Definindo método para buscar todos os usuários, e informando que retorno será um array
        'query': {method:'GET', isArray: true},
        // Definindo método para buscar apenas um usuário pelo ID
        'get': {method:'GET'},
        // Definindo método para salvar um usuário no banco de dados
        'save': {method:'POST'},
        // Definindo método para excluir um usuário
        'remove': {method:'DELETE'},
        // Definindo método para atualizar um usuário
        'update': {method:'PUT'}
    });
})

// Para utilizar
// Chamando factory UserAPI
// Executando o método query
UserAPI.query().$promise.then(function(response) {
    // Tratando a resposta dentro de uma promise
    $scope.users = response;
});
```

Figura 19 – Requisição utilizando o ngResource no Ionic. Baseado em Github[8]

```
func checkUserEmail (email: String, completionHandler: (response : String, user: User?) -> ()) {

    /**
        Iniciando uma requisição do Alamofire para encontrar um usuário por e-mail, passando
        os parâmetros:
        tipo de requição: POST
        url: string de uma URL concatenada com o endpoint da API
        dicionário de parâmetros da requisição
        no caso o email a ser buscado identificado pela chave "email"
    */
    Alamofire.request(.POST, baseURL + "/api/find-by-email",
        parameters: ["email" : email], headers: nil).responseJSON { response in
        // Recebendo a resposta do servidor e fazendo um parser para o
        // formato JSON utilizando a classe JSON do framework SwiftyJSON
        let data = JSON(response.result.value!)
        // Validando os dados do usuário.
        // Retornam-se dois valores, uma chave de controle (response) e um objeto (user).
        if data["user"] != nil {
            // Caso a requisição tenha encontrado algum usuário,
            // cria-se um objeto User a partir dos dados do JSON.
            let user: User = self.decodeJSON(data)
            completionHandler(response: "registered", user: user)
        }else{
            // Caso a requisição não tenha encontrado um usuário, retorna-se um objeto nulo
            completionHandler(response: "not registered", user: nil)
        }
    }
}
```

Figura 20 – Requisição utilizando Alamofire e SwiftyJSON no iOS

[8] <https://github.com/fga-gpp-mds/2016.1-Partiu_frontend>

5.1.3 Detecção de força do toque

Em 2015, alguns dispositivos móveis foram comercializados com uma tecnologia de detecção de pressão nas telas sensíveis ao toque. Primeiramente pela companhia chinesa Huawei e depois pela Apple, trata-se de um recurso capaz de identificar a quantidade de força que o usuário exerceu em um toque, o que abre o leque de possibilidades para novas funcionalidades nos aplicativos desenvolvidos. No iOS, é chamado de *3DTouch* e por enquanto está disponível nos modelos de iPhone 6S ou superior. No Apple Watch e no novo MacBook, assim como no Mate S, da Huawei, é chamado apenas de *Force Touch*, nome original da tecnologia. Vale ressaltar que o Android já possui a capacidade de ler a área de toque por meio de *software* apenas, no entanto, o *Force Touch* não é apenas a leitura de área de toque, mas sim da pressão do toque, o que envolve *hardware* e *software*.

As Figuras 21 e 22 apresentam trechos de código no iOS e no Ionic para utilização do *3DTouch* criando *Quick Actions*, uma funcionalidade do iOS que permite que o usuário aperte com mais força o ícone do aplicativo para ter algumas opções antes de entrar no aplicativo.

```
func createShortcutItemsWithIcons() {

    // Criando um ícone com uma imagem customizada
    let icon = UIApplicationShortcutIcon.init(templateImageName: "iCon2")

    // Criando uma Quick Action dinâmica com o ícone customizado
    let item = UIMutableApplicationShortcutItem.init(type: "dynamicQuickAction",
                                                     localizedTitle: "Minha ação",
                                                     localizedSubtitle: "Executar a minha ação",
                                                     icon: icon,
                                                     userInfo: nil)

    // Colocando todas as ações, nesse caso somente uma, em um vetor de ações
    let items = [item] as Array

    // Adicionando as novas ações juntamente com as ações estáticas criadas na .plist
    let existingItems: Array = UIApplication.shared.shortcutItems! as Array
    let updatedItems: Array = existingItems + items
    UIApplication.shared.shortcutItems = updatedItems
}

func application(_ application: UIApplication,
                 performActionFor shortcutItem: UIApplicationShortcutItem,
                 completionHandler: @escaping (Bool) -> Void) {
    if shortcutItem.type == "dynamicQuickAction" {
        //faz algo
    }
}
```

Figura 21 – Criando uma Quick Action com o 3DTouch no iOS. Fonte: Baseado em Github[9]

[9] <https://github.com/versluis/3D-Touch>

```
function QuickActionService($rootScope, $q) {
    // Verificar a disponibilidade do 3DTouch no dispositivo
    function check3DTouchAvailability() {
        return $q(function(resolve, reject) {
            if (window.ThreeDeeTouch) {
                window.ThreeDeeTouch.isAvailable(function (available) {
                    resolve(available);
                });
            } else {
                reject();
            }
        });
    };

    // Para configurar as QuickActions, basta chamar esta
    // função configure() do factory 'QuickActionService'
    function configure() {
        check3DTouchAvailability().then(function(available) {
            // Configurar as ações que serão mostradas na tela inicial
            // Por exemplo, uma ação de compartilhar o aplicativo em alguma rede social
            window.ThreeDeeTouch.configureQuickActions([
                {
                    // Identificador da ação
                    type: 'share',
                    // Título da ação
                    title: 'Share',
                    // Subtítulo
                    subtitle: 'Share like you care',
                    // Ícone nativo do sistema que aparece ao lado do título
                    iconType: 'Share'
                },
                {
                    type: 'favorite',
                    title: 'Show favorites',
                    // Imagem customizada que aparece ao lado do título
                    iconTemplate: 'HeartTemplate'
                }
            ]);

            // Definindo callback para executar uma ação de
            // acordo com a opção selecionada, nesse caso a
            // ação apenas redireciona para uma página específica do aplicativo
            window.ThreeDeeTouch.onHomeIconPressed = function(payload) {
                if (payload.type == 'share') {
                    document.location = 'share.html';
                } else if (payload.type == 'favorite') {
                    document.location = 'favorite.html';
                }
            };
        })
    };
}
```

Figura 22 – Criando uma Quick Action com o 3DTouch no Ionic. Fonte: Baseado em Github[10]

Em ambos os casos a implementação é simples apenas devendo-se considerar as diferenças entre plataformas e linguagens, no entanto, no caso do Ionic é necessária a utilização do *plugin cordova-plugin-3dtouch*[11].

5.1.4 Leitor Biométrico

A preocupação por segurança dos dados e privacidade se intensificou, pois os dispositivos móveis agora possuem muitas informações pessoais como contatos, fotos, *e-mails*, localização e até mesmo senhas, no entanto, por serem móveis são mais fáceis de serem extraviados, furtados ou invadidos. Com isso, cada dispositivo possui seus próprios meios de autenticação e segurança, sendo a forma mais comum de segurança senhas alfanuméricas. Com o avanço da tecnologia presente nos dispositivos móveis, uma forma que se tornou comum para autenticação de usuário é a leitura biométrica, ou seja, reconhecimento de impressões digitais. Muitos *smartphones* já possuem esse recurso adicionando ao celular uma camada extra de segurança e praticidade.

Além do celular e dos dados pessoais do usuário, é possível que os dados de um aplicativo em específico sejam protegidos pelo mesmo recurso, ou seja, caso o aparelho seja encontrado desbloqueado, ainda não será possível ter acesso à todos os dados disponíveis dentro dele. Alguns aplicativos que se utilizam desse recurso são, por exemplo, bancos, diários, chaveiros eletrônicos e protetores de fotos. Nos dispositivos da Apple o recurso foi introduzido no iPhone 5 e foi chamado de *Touch ID*, nos dispositivos Android, os aparelhos mais avançados de cada marca possuem o sensor biométrico.

Como mencionando anteriormente, é possível proteger os dados de uma aplicação com o leitor biométrico. Para isso, o iOS dispõe do *framework LocalAuthentication*, que permite que a aplicação solicite a leitura da digital, caso o aparelho possua o recurso, e compare com as impressões digitais cadastradas no dispositivo. Caso o dispositivo não possua o recurso, será requerida uma senha alfanumérica comum para desbloquear os dados do aplicativo. No Android, de maneira similar ao iOS, a partir da versão 6.0, foi introduzida uma nova *API*, a *Fingerprint Authentication*[12], que permite utilizar o sensor biométrico dentro dos aplicativos. No Android, é preciso efetuar alguns passos, listados a seguir, antes de poder utilizar a autenticação biométrica.

- **Permissão**: É preciso adicionar uma permissão especial no arquivo *Manifest* para que seja possível utilizar o sensor;
  - *<uses-permission android:name="android.permission.USE_FINGERPRINT"/>*

[10] <https://github.com/ashteya/ionic-tutorial-quickactions>
[11] <https://github.com/EddyVerbruggen/cordova-plugin-3dtouch>
[12] <https://developer.android.com/about/versions/marshmallow/android-6.0.html>

- ***Container* de chaves**: É preciso acessar o *container* de chaves do Android, para poder armazenar uma chave que será criada para a aplicação;

  - *KeyStore*: onde o sistema guarda todas as chaves de criptografia;

- **Gerar uma chave**: Tendo acesso ao *KeyStore*, é preciso gerar uma chave para a aplicação e então armazená-la no *KeyStore*;

As Figuras 23 e 24 apresentam trechos de códigos que implementam a verificação biométrica em iOS e Android.

```
func authenticateUser() {
    // Iniciando contexto de autenticação local
    let context = LAContext()
    // Variável para armazenamento de possíveis erros
    var error: NSError?
    // Verificando se o dispositivo permite leitura biométrica
    if context.canEvaluatePolicy(LAPolicy.deviceOwnerAuthenticationWithBiometrics, error: &error) {
        // Se permitir, emite alerta informando o porquê
        context.evaluatePolicy(LAPolicy.deviceOwnerAuthenticationWithBiometrics,
                               localizedReason: "Autenticação necessária para acessar aplicativo.",
                               reply: { (success: Bool, evalPolicyError: NSError?) in
            if success {
                // Se a autenticação foi bem sucedida, executa alguma ação
            } else {
                switch evalPolicyError!.code {
                case LAError.systemCancel.rawValue:
                    // O sistema cancelou a autenticação pelo Touch ID
                    // Porque outra aplicação foi iniciada, por exemplo
                    break
                case LAError.userCancel.rawValue:
                    // Usuário cancelou a autenticação pelo Touch ID
                    break
                case LAError.userFallback.rawValue:
                    // Usuário decidiu não utilizar Touch ID, nesse caso,
                    // Solicitar outro método de autenticação, por exemplo, senha
                    break
                default:
                    // Autenticação falhou, nesse caso, solicitar outro método de autenticação
                    // Por exemplo senha
                    break
                }
            }
        } as! (Bool, Error?) -> Void)
    }
    else{
        // Nesse caso, o dispositivo não permite autenticação pelo Touch ID
        // Solicitar outro método de autenticação, por exemplo, senha
    }
}
```

Figura 23 – Implementação do Touch ID no iOS. Fonte: Baseado em AppCoda[13]

[13] <https://www.appcoda.com/touch-id-api-ios8/>

```java
// Método que inicia a leitura biométrica
public void startListening(FingerprintManager.CryptoObject cryptoObject) {
    // Se a leitura não estiver disponível encerra o método
    if (!isFingerprintAuthAvailable()) {
        return;
    }
    // Instancia um objeto que responde aos eventos de cancelamento da leitura biométrica
    mCancellationSignal = new CancellationSignal();
    // Realiza a leitura
    mFingerprintManager.authenticate(cryptoObject, mCancellationSignal, 0, this, null);
}
// Métodos de callback da leitura
@Override
public void onAuthenticationSucceeded(FingerprintManager.AuthenticationResult result) {
    // Autenticação bem sucedida
    // O usuário foi reconhecido
}

@Override
public void onAuthenticationFailed() {
    // Autenticação falhou
    // Digital não foi reconhecida
}

@Override
public void onAuthenticationHelp(int helpMsgId, CharSequence helpString) {
    // Autenticação falhou, mas o sistema pode indicar como resolver
}


@Override
public void onAuthenticationError(int errMsgId, CharSequence errString) {
    // Autenticação falhou por algum outro motivo que o sistema não pode resolver
}
```

Figura 24 – Implementação da leitura biométrica no Android. Fonte: Baseado em Google[14]

No Ionic também é possível utilizar a leitura biométrica nos aplicativos. No entanto, para utilizar esse recurso, é necessária a utilização de dois *plugins* separados, um para iOS e um para Android, pois em cada plataforma a autenticação se dá de maneira específica e ainda não há um *plugin* para ambas as plataformas. As Figuras 25 e 26 apresentam dois trechos de códigos no Ionic implementando a leitura biométrica. Foram usados os seguintes *plugins*, para iOS e Android, respectivamente:

- **iOS**: *cordova-plugin-touchid*[15];
- **Android**: *cordova-plugin-android-fingerprint-auth*[16];

[14] <https://developer.android.com/samples/FingerprintDialog/index.html>
[15] <https://github.com/leecrossley/cordova-plugin-touchid>
[16] <https://github.com/mjwheatley/cordova-plugin-android-fingerprint-auth>

```
.controller("ExampleController", function($scope, $ionicPlatform, $cordovaTouchID) {
    // Verifica se o dispositivo suporta Touch ID
    $cordovaTouchID.checkSupport().then(function() {
        // Se sim, emite alerta com mensagem solicitando o Touch ID
        $cordovaTouchID.authenticate("Mensagem").then(function() {
            // Se a verificação da biometria foi bem sucedida
            // Faz algo
        }, function(error) {
            // Se a verificação da biometria falhou
            // Faz algo
        });
    }, function(error) {
        // Se não há o suporte para Touch ID, solicita algum outro
        // método de autenticação, por exemplo, senha
    });
})
```

Figura 25 – Implementação da leitura biométrica para iOS no Ionic. Fonte: Baseado em Github[17]

```
// Verificando se a verificação biométrica está disponível
// Passando como parâmetros da função duas funções de callback para sucesso e falha
FingerprintAuth.isAvailable(isAvailableSuccess, isAvailableError);

function isAvailableSuccess(result) {
    if (result.isAvailable) {
        // Se a verificação biométrica é possível, inicia-se a leitura
        FingerprintAuth.show({
            // Alias para a chave criptográfica do KeyStore
            clientId: "myAppName",
            // Chave criada pelo desenvolvedor única para a aplicação
            // Usada para criptografar a resposta da verificação biométrica
            clientSecret: "chave_muito_secreta"
        }, successCallback, errorCallback);
        // Novamente, passa-se métodos de callback para sucesso e falha na
        // leitura da biometria
    }
}

function isAvailableError(message) {
    // Caso a verificação biométrica não esteja disponível no dispositivo
}

function successCallback(result) {
    // Verificação biométrica bem sucedida
}

function errorCallback(error) {
    // Verificação biométrica falhou
}
```

Figura 26 – Implementação da leitura biométrica para Android no Ionic. Fonte: Baseado em Github[18]

É possível criar aplicações que utilizem desse recurso nas duas abordagens, no entanto, de maneira geral, a implementação dessa funcionalidade é mais simples quando se trata do iOS, tanto nativo quanto multiplataforma, já que no Android são necessários

[17] <https://github.com/leecrossley/cordova-plugin-touchid>
[18] <https://github.com/mjwheatley/cordova-plugin-android-fingerprint-auth>

alguns passos anteriores ao desenvolvimento. Em relação ao desenvolvimento no Android, o multiplataforma é mais simples que o nativo, em relação à complexidade e quantidade de código a ser desenvolvido.

5.1.5 Extração de metadados de arquivos

Em alguns casos, pode ser necessário a extração de metadados de algum arquivo específico, como uma imagem ou um áudio. No caso de uma imagem, podem ser retiradas informações como local, hora de criação e até mesmo dados climáticos no momento da captura da foto. No caso de um arquivo de áudio, é possível extrair o título do áudio, o autor ou artista, tempo de duração e álbum. Esses metadados podem ser usados para melhorar a experiência de uso do aplicativo fornecendo sugestões e funcionalidades mais próximas das necessidades reais do usuário, ou até mesmo para fazer filtros e buscas mais inteligentes nos dados que o aplicativo manuseia.

Para extração de metadados no iOS existem classes nativas como a *AVPlayerItem* e *ALAssetsLibrary* para extração de metadados, respectivamente, de audio e imagens. No Android existe a classe *MediaMetadataRetriever*, que lida com a extração de tipos variados de mídia. Em ambos os casos, iOS e Android, a implementação é simples, pois é preciso apenas seguir os métodos das classes utilizando constantes do sistema para indicar qual dado se deseja obter. Por exemplo, se o objetivo é obter a informação de título, no iOS, basta chamar pela *string* "title". Já no Android, para o mesmo objetivo, basta chamar pela *string* "METADATA_KEY_TITLE".

As Figuras 27 e 28 apresentam trechos de códigos implementando a extração de metadados no iOS e no Android.

```
//Intanciando um AVPlayerItem a partir do caminho onde o arquivo se encontra
let playerItem = AVPlayerItem(url: audioPath as! URL)
let metadataList = playerItem.asset.commonMetadata

//Percorrendo lista de metadados do arquivo
for item in metadataList {
    if let stringValue = item.value as? String {

        //Obtendo título do arquivo através da chave "title"
        if item.commonKey == "title" {
            print("Titulo: \(stringValue)")
        }
        //Obtendo artista do audio  através da chave "artist"
        if item.commonKey == "artist" {
            print("Artista: \(stringValue)")
        }
        //Caso haja a necessidade de outras informações, basta procurar por outras chaves
        //Como por exemplo "albumName"
    }
}
```

Figura 27 – Extração de metadados de um arquivo áudio no iOS

```
public String getTitle(String mediaPath) {
    // Recebe-se o caminho pelo parâmetro do método
    String titleMedia = null;

    try {
        // Definindo o caminho para o arquivo de mídia no objeto mmr, onde,
        // mmr é um objeto da classe MediaMetadataRetriever
        this.mmr.setDataSource(context, Uri.parse(Uri.encode(mediaPath)));
    } catch (Exception e) {
        Log.e("AudioExtractor-getTitle", e.getLocalizedMessage() +" - " +e.getCause());
    }
    // Extraindo metadado  de título do arquivo de mídia
    titleMedia = mmr.extractMetadata(MediaMetadataRetriever.METADATA_KEY_TITLE);

    return titleMedia;
}
```

Figura 28 – Extração de metadados de um arquivo áudio no Android. Baseado em Github[19]

A extração de dados no Ionic pode ser realizada utilizando um *plugin* do cordova chamado *File*, utilizado para obter acesso aos arquivos e diretórios dos dispositivos. Na documentação do *ngCordova* é apresentada a estrutura de pacotes do iOS e do Android, para auxiliar o desenvolvedor a acessar as pastas desejadas. É apresentada, também, uma lista de possíveis erros que podem ocorrer durante o uso do *plugin*, como arquivo não encontrado, caminho inexistente, erro de *encoding*, entre outros. Com o *plugin File* é possível obter as seguintes propriedades:

- ***name***: nome do arquivo seguido de sua extenção
- ***localURL***: a url do arquivo, seguido de seu nome e extenção
- ***type***: *Internet media type* ou *MIME type*, padrão usado na Internet para indicar um tipo de dado
- ***lastModifiedDate***: data e hora da última modificação do arquivo
- ***size***: tamanho do arquivo em bytes

Um exemplo de dados obtidos é apresentado na Figura 29. O código usado para obtenção destes dados é apresentado na Figura 30. Para obtenção da duração de uma música ou um vídeo, é necessário o uso de outro *plugin* chamado *Media*. Não foram encontrados *plugins* para obtenção de dados tão detalhados quanto os disponíveis no iOS e Android, como informações do autor e álbum de uma música ou a localização geográfica do local em que uma foto foi tirada.

[19] <https://github.com/adorilson/MMUnB>

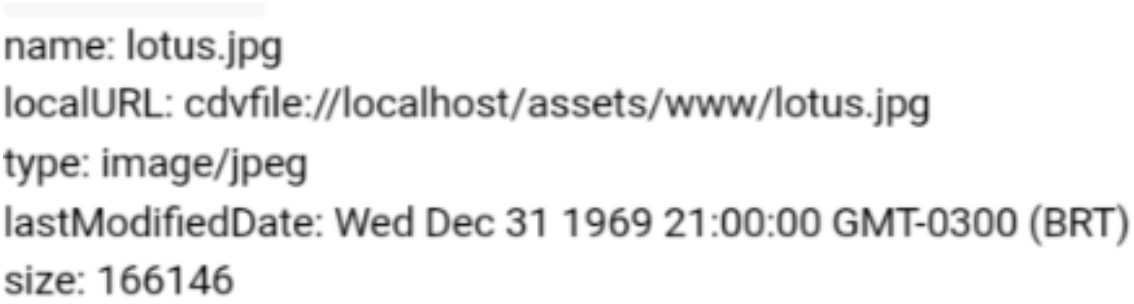

Figura 29 – Apresentação dos metadados de uma imagem - Ionic

```
app.controller('HomeCtrl', function($scope) {

    $scope.getFileInfo = function() {
        // Tentando obter uma imagem denominada "lotus", que encontra-se na pasta do projeto.
        // Caso encontre o arquivo, chama-se a função onSuccess, se não, a função onFail é chamada.
        window.resolveLocalFileSystemURL(cordova.file.applicationDirectory
        + "www/lotus.jpg", $scope.onSuccess, $scope.onFail);
    };

    $scope.onFail = function(e) {
        console.log("FileSystem Error");
    };

    // Obtenção de metadados do arquivo encontrado
    $scope.onSuccess = function (fileEntry) {
        fileEntry.file(function(file) {
            var fileInfo = "";
            fileInfo += "name: " + file.name + "<br>";
            fileInfo += "localURL: " + file.localURL + "<br>";
            fileInfo += "type: " + file.type + "<br>";
            fileInfo += "lastModifiedDate: " + (new Date(file.lastModifiedDate)) + "<br>";
            fileInfo += "size: " + file.size + "<br>";

            document.querySelector("#status").innerHTML = fileInfo;
            console.dir(file);
        });
    }

});
```

Figura 30 – Extração de metadados de mídia no Ionic. Baseado em Github[20]

5.1.6 Envio de e-mail e SMS

Cada plataforma possui seus próprios *frameworks* para incluir dentro do aplicativo a ser desenvolvido o mecanismo de envio de *e-mails* ou mensagens de texto (*SMS*), muito utilizados para compartilhar e divulgar informações do aplicativo ou até mesmo o próprio

[20] <https://github.com/cfjedimaster/Cordova-Examples/tree/master/getfiledata>

aplicativo e entrar em contato com a equipe de desenvolvimento do mesmo. Para utilizar desses recursos no Ionic, é possível utilizar o *plugin cordova-plugin-email-composer*[21] para ter acesso à interface padrão do sistema no qual o aplicativo está sendo executado para envio de *e-mail*. De maneira similar, para o envio de *SMS*, é possível utilizar o *plugin cordova-sms-plugin*[22]. Tanto na abordagem nativa, quanto na multiplataforma o desenvolvimento é simples diferenciando apenas na linguagem e no ambiente. As Figuras 31 e 32 apresentam dois trechos de código, no iOS e no Ionic, para o envio de *e-mail*.

```
// Verificando se o dispositivo pode enviar e-mail
if MFMailComposeViewController.canSendMail() {
    let mail = MFMailComposeViewController()
    mail.mailComposeDelegate = self
    // Definindo um array de destinatários
    mail.setToRecipients(["somostodos@wushuzeiros.com"])
    // Definindo o assunto
    mail.setSubject("Assunto do e-mail")
    // Definindo uma mensagem padrão
    mail.setMessageBody("Corpo padrão do e-mail", isHTML: true)
    // Chamando interface padrão do iOS para envio de e-mail com as informações definidas
    present(mail, animated: true)
} else {
    // Caso o dispositivo não possa enviar e-mails
}
```

Figura 31 – Envio de *e-mail* no iOS

```
// Esperando o dispositivo dizer que está pronto para utilizar os plugins de e-mail
document.addEventListener('deviceready', function () {
    // Verificando se o dispositivo pode enviar e-mails
    cordova.plugins.email.isAvailable(
        function (isAvailable) {
            cordova.plugins.email.open({
                // Definindo destinatário
                to:      'max@mustermann.de',
                // Definindo outro destinatário em cópia
                cc:      'erika@mustermann.de',
                // Definindo outros destinatário em cópia oculta
                bcc:     ['john@doe.com', 'jane@doe.com'],
                // Definindo o assunto do e-mail
                subject: 'Bem vindos',
                // Definindo uma mensagem padrão do e-mail
                body:    'Como vai você? Estou te esperando! :)'
            });
        }
    );
}, false);
```

Figura 32 – Envio de *e-mail* no Ionic. Baseado em Github[23]

[21] <https://github.com/katzer/cordova-plugin-email-composer>
[22] <https://github.com/cordova-sms/cordova-sms-plugin>

5.1.7 *Widgets*

Widgets são atalhos que facilitam o acesso a algum programa. Vários dispositivos possuem esse recurso, como computadores, camêras fotográficas, *smartphones* e *tablets*. Eles ajudam o usuário a realizar ações mais rapidamente e sem entrar necessariamente em um programa ou aplicativo, sendo alguns dos mais comuns e práticos, o de agenda de eventos e informações climáticas.

Cada plataforma possui suas próprias maneiras de implementá-los, sendo no iOS apenas disponíveis na central de notificações do dispositivo e no Android sendo disponíveis tanto na tela inicial do aparelho quanto na central de notificações. A Figura 33 mostra exemplos de *widgets* no iOS e Android respectivamente.

Figura 33 – *Widgets* no iOS (à esquerda) x *Widgets* no Android (à direita).

Não é possível criar *widgets* com Ionic e Cordova, pois utilizam *HTML, CSS* e

[23] <https://github.com/katzer/cordova-plugin-email-composer>

JavaScript embutidas em uma *web view*, e fora dos aplicativos não há um *browser* para renderizar o aplicativo feito no Ionic. O que poderia ser feito é criar o aplicativo multiplataforma e depois alterar os projetos nativamente em cada plataforma para adicionar o *widget* que se deseja. Dessa forma, cada plataforma iria possuir códigos nativos para os respectivos *widgets* além do código do aplicativo multiplataforma em si. Caso o *widget* seja uma parte fundamental da aplicação, pode não ser uma boa ideia o desenvolvimento multiplataforma.

5.1.8 Assistentes Pessoais

Os dispositivos móveis têm se tornado cada vez mais inteligentes e úteis para seus usuários. Funcionam como computadores, agendas, calendários, máquinas fotográficas, etc. Para atenderem ainda mais às necessidades de seus usuários, os dispositivos possuem em sua maioria inteligências artificiais, na forma de assistentes pessoais, criadas para auxiliar no uso dos *smartphones* e *smartwatches*. Na Apple, a assistente é conhecida como Siri, já no Android existem alguns aplicativos de terceiros que tentam criar uma assistente pessoal. O mais conhecido é Google Now, da própria Google, embora nesse caso ele não seja uma assitente pessoal, pois não possui traços de personalidade, ele apenas reconhece voz e responde a perguntas com muita eficiência.

Com a Siri é possível controlar o dispositivo, criando eventos, alarmes, lembretes, abrindo aplicativos e controlando eletrodomésticos, por exemplo, e fazer buscas variadas na *internet*, com o Google Now apenas essa última opção está disponível. Durante a execução deste trabalho, foi lançado o primeiro celular inteiramente feito pela Google chamado de Pixel, que conta com o Google Assistant, uma evolução do Google Now para se tornar uma assistente pessoal. Com ele será possível também controlar o dispositivo assim como as outras assistentes de outras plataformas fazem.

Durante a execução deste trabalho, a Apple lançou juntamente com a nova versão do sistema operacional, iOS 10, a *API* da Siri para que aplicativos de terceiros possam interagir com a assistente pessoal tornando a experiência de uso dos aplicativos ainda mais intuitiva e prática. Ainda não há *plugins* do Cordova que permitam o uso desse recurso, ou seja, caso haja a necessidade de utilizar a Siri, o multiplataforma ainda não é um opção viável. Vale ressaltar que existem alguns *plugins* para utilizar o reconhecimento de voz e transcrever em texto[24] o que foi dito, no entanto, isso não é o mesmo que interagir com a Siri. O Google Assistant ainda não está disponível em outros dispositivos além do Pixel da Google e não existem *SDK* nativo ou *plugins* de terceiros para utilizá-lo. Com isso, por enquanto, ainda não é possível desenvolver aplicativos multiplataforma integrados com as assistentes pessoais mais utilizadas.

[24] <https://github.com/api-ai/api-ai-cordova>

5.1.9 *Smartwatches*

Além de *smartphones* e *tablets*, estão começando a ficar populares os dispositivos *vestíveis* como relógios inteligentes. Isso implica em mais plataformas para dominar e mais possibilidades de aplicativos e funcionalidades para desenvolver. Um dos *smartwatches* mais populares, atualmente, é o Apple Watch. Lançado com um sistema operacional próprio derivado do iOS, o WatchOS, por enquanto, funciona como uma extensão dos aplicativos criados para iPhone, ou seja, adiciona funcionalidades de rápida interação ao relógio para que não seja necessário interagir com o celular a todo momento.

Caso haja a necessidade de criar um aplicativo que suporte o Apple Watch, pode-se optar por desenvolver multiplataforma também, pois já existem *plugins* [25] do Cordova para realizar a integração entre o aplicativo multiplataforma e a extensão para Apple Watch. No entanto, o *plugin* citado só funciona enquanto o aplicativo no iPhone está rodando. Quando, por algum motivo, o aplicativo deixa de rodar o *plugin* não consegue mais se comunicar com o relógio podendo prejudicar a experiência de uso do aplicativo. Com isso, pode não ser uma boa ideia utilizar funcionalidades que precisam de atualização em tempo real, por exemplo, que recebe dados de um servidor, pois se o aplicativo no iPhone não estiver rodando, o relógio não conseguirá se comunicar e pegar os dados mais recentes, fazendo com que apresente dados desatualizados, prejudicando a experiência de uso.

Vale ressaltar que o aplicativo no Apple Watch deve ser feito nativamente, diretamente no *Xcode*. O *plugin* apenas fornece um meio para que o *app* multiplataforma do iPhone possa transmitir e receber dados do *app* do relógio, mas não é possível criar a interface e controladoras do relógio via multiplataforma, pois o Apple Watch não possui um navegador (*WebView*) para interpretar o *JavaScript* assim como o iPhone. Então, deve ser feito o aplicativo utilizando Cordova e Ionic, por exemplo, para o iPhone, e depois criar o *app* no *Xcode* para Apple Watch e então utilizar um *plugin* para realizar a comunicação entre os dois aplicativos.

Se planeja-se que o aplicativo utilize muitos recursos do Apple Watch, como frequencímetro, pedômetro e acelerômetro, como aplicativos voltados para a área da saúde, pode ser melhor desenvolver nativamente, pois o foco não estará sendo em desenvolver o aplicativo para várias plataformas, mas sim para um dispositivo específico. Com isso, pode-se aproveitar melhor todos os recursos do relógio e todos os benefícios da integração do mesmo com o iPhone.

Caso o aplicativo já exista, ainda que seja multiplataforma, é possível desenvolver o aplicativo para o relógio, no entanto, a comunicação entre o iPhone e o relógio não será perfeita no multiplataforma como é no nativo, o que implica em limitar as funcionalidades

[25] <https://github.com/leecrossley/cordova-plugin-apple-watch>

do aplicativo no relógio para não prejudicar a experiência do usuário, podendo até mesmo fazer com que não valha a pena investir tempo e esforço na criação do aplicativo para relógio. Cabe ressaltar que também já existem *plugins*[26] do Cordova para Android Wear, os *smartwatches* que rodam Android.

5.1.10 Câmeras customizadas

Um dos recursos mais utilizados nos *smartphones* é a câmera fotográfica. Muitos usuários definem o aparelho a ser adquirido com base na capacidade da camêra, com isso muitos aplicativos aproveitam para oferecer mais funcionalidades que os aplicativos de camêras nativos dos dispositivos. Funcionalidades como filtros de imagens, edição de vídeos, câmera lenta, integração com redes sociais e, em alguns casos, os aplicativos são uma rede social em si. Caso o aplicativo tenha a necessidade de camêras customizadas é possível desenvolvê-lo multiplataforma utilizando alguns *plugins* para o Cordova, por exemplo o *cordova-plugin-camera-preview*[27].

Tanto o iOS quanto o Android fornecem suporte para a criação de camêras customizadas de acordo com as necessidades dos aplicativos. No iOS é possível utilizar o *framework AVFoundation* para identificar as camêras presentes no dispositivo e utilizá-las de acordo com as regras de negócio do aplicativo. É possível criar interfaces completamente customizadas com quaisquer funcionalidades que a equipe de desenvolvimento seja capaz de implementar. Assim como no iOS, no Android também é possível criar camêras customizadas utilizando o *framework android.hardware.camera2*.

5.1.11 Detecção facial

A detecção facial é uma tecnologia usada em uma grande variedade de aplicações que identificam rostos humanos em imagens. Detectar rostos humanos e extrair traços faciais é uma importante funcionalidade com diversas possibilidades de aplicações, tais como reconhecimento facial para autenticação de usuário e videoconferências.

A plataforma iOS possui um processador de imagens nativo denominado *CIDetector*, capaz de identificar faces em imagens e vídeos. O *SDK* do Android possui uma *API* para detecção facial, a *Face Detection*, capaz de detectar faces em imagens.

É possível realizar o processo de detecção facial como o uso de bibliotecas não nativas, como o OpenCV, uma biblioteca livre e multiplataforma amplamente utilizada para processamento de imagens. Com ela é possível não apenas a detecção, como também o reconhecimento facial.

[26] <https://github.com/tgardner/cordova-androidwear>
[27] <https://github.com/cordova-plugin-camera-preview/cordova-plugin-camera-preview>

Outro exemplo de tecnologia multiplataforma para detecção facial é o Beyond Reality Face NXT, um *SDK* que oferece suporte a aplicações iOS nativas e a aplicações multiplataformas desenvolvidas com o uso de *actionscript*.

O Mobile Vision, da Google, é uma *API* que possibilita a detecção facial em fotos, vídeos e *streams* ao vivo, tanto em plataformas Android, como em iOS.

O Cordova não apresenta bibliotecas próprias para a realização de detecção facial, sendo necessário o uso de bibliotecas externas. O Scanbot é um exemplo de *SDK* que pode ser utilizado junto ao Cordova e possibilita detecção de documentos e de faces em tempo real, porém não é totalmente compatível com iOS e não é uma ferramenta gratuita.

5.2 Consolidação dos resultados

As funcionalidades **Login com Facebook**, **Consumo de Web Services**, **Envio de e-mail e SMS** e uso do **Leitor Biométrico** podem ser desenvolvidas tanto na abordagem nativa quanto na multiplataforma, sem ressalvas, não causando influência na escolha entre abordagens.

Para a realização de **extração de metadados de arquivos de mídia**, é possível utilizar ferramentas multiplataforma, porém há restrições quanto à obtenção dos dados, visto que alguns deles puderam ser obtidos apenas no desenvolvimento nativo.

A criação de **Widgets** ainda não é possível com o uso de ferramentas multiplataforma baseadas em soluções *web*, como o Cordova.

Para o uso de **assistentes pessoais**, como a Siri e o Google Assistant, é necessário o uso de tecnologias nativas.

Não é possível criar aplicativos para **SmartWatches**, como o Apple Watch, utilizando o Cordova. O que pode-se fazer é criar o aplicativo de celular, utilizando tecnologias multiplataforma, o aplicativo do relógio, feito nativamente, e a comunicação do aplicativo com o relógio usando tecnologias multiplataforma, porém esta comunicação terá restrições, por exemplo o aplicativo do relógio só conseguirá trocar informações se o aplicativo do celular estiver aberto, o que não é necessário na solução totalmente nativa.

É possível desenvolver aplicativos multiplataforma com **câmeras customizadas**, porém quanto maior a customização, mais difícil o desenvolvimento, tanto para nativo quanto para multiplataforma, cabendo analisar a *expertise* da equipe para saber qual abordagem escolher.

Para realização de **detecção facial** em fotos, vídeos e em tempo real, é possível utilizar ferramentas multiplataformas, porém muitas das *APIs* disponibilizadas para isso não são gratuitas ou apresentam algum tipo de restrição, como funcionamento em apenas uma das plataformas, Android ou iOS. Já no desenvolvimento nativo, as próprias

plataformas disponibilizam *APIs* que possibilitam detecções faciais.

5.3 Opinião dos especialistas

Com o propósito de confrontar a opinião dos especialistas da área de desenvolvimento móvel com os resultados obtidos empiricamente, foi elaborada uma série de afirmações baseadas nas informações obtidas na primeira etapa da análise exploratória. Além das afirmações de caráter técnico, foram elaboradas outras duas para avaliar a percepção geral dos especialistas em relação ao desenvolvimento multiplataforma. Considerando as funcionalidades analisadas, foi abordada a viabilidade de desenvolvimento das mesmas no ambiente multiplataforma. Os especialistas poderiam expressar as suas opiniões de acordo com as opções listadas a seguir.

- **Concordo**: funcionalidade em questão pode ser desenvolvida no ambiente multiplataforma, sem ressalvas;
- **Concordo, mas há limitações quando comparado ao desenvolvimento nativo**: funcionalidade em questão pode ser desenvolvida no ambiente multiplataforma, com ressalvas;
- **Discordo**: funcionalidade em questão não pode ser desenvolvida no ambiente multiplataforma;
- **Não sei opinar**: não sabe se é possível desenvolver a funcionalidade no ambiente multiplataforma;
- **Outros**: resposta aberta, caso o profissional queira dar uma opinião mais detalhada a respeito da funcionalidade;

O questionário, bem como os gráficos de análise dos resultados encontram-se no Apêndice A.

5.3.1 Análise dos dados coletados

Avaliando as respostas dadas ao questionário, observou-se que a percepção dos especialistas para as funcionalidades de **Login com Facebook**, **Consumo de *Web Service*** e **envio de *e-mail* e SMS** coincidiu com os resultados obtidos na análise exploratória. Tais funcionalidades já estão muito difundidas entre os desenvolvedores, além disso, possuem baixo grau de complexidade de implementação no ambiente nativo. Além da experiência própria, esses podem ter sido os principais motivos para que tenham sido analisadas como passíveis de serem implementadas no ambiente multiplataforma.

Para as funcionalidades de **Leitor Biométrico**, ***Widgets*** e **Assistentes Pessoais** a maior parte dos especialistas não soube opinar. Por serem funcionalidades não essenciais, isto é, funcionalidades que se ausentes, não impedem o funcionamento do aplicativo, pode ser que nem todos os especialistas conheçam ou tenham tido experiência desenvolvendo alguma delas, dificultando o julgamento.

Em relação ao desenvolvimento de aplicativos para ***Smartwatches***, a maioria não soube opinar. Por ser uma tecnologia recente no mercado e não popularizada no país, é possível que os especialistas não tenham experiência e nem interesse neste nicho de aplicações. Para os que opinaram, cerca de 70% acha que não é possível a criação de aplicativos para os mesmos utilizando tecnologias multiplataforma, coincidindo com os resultados obtidos na análise exploratória. Quanto à comunicação entre o aplicativo para ***Smartwatch*** e o aplicativo no celular, apenas 5% dos especialistas responderam de acordo com os dados da análise exploratória, ou seja, concordam que há restrições na comunicação entre ambos os aplicativos. Isso reforça a falta de conhecimento e interesse dos especialistas neste tipo de tecnologia.

No que diz respeito à funcionalidade de **Detecção Facial**, metade dos especialistas foram equivocados em suas respostas, discordando ou concordando totalmente com a possibilidade do uso deste recurso no ambiente multiplataforma. Na realidade, a implementação deste recurso é possível, porém com limitações, de acordo com os dados obtidos na análise exploratória.

Quanto à possibilidade de criação de **Câmeras Customizadas**, mais da metade dos especialistas não sabem que esta funcionalidade pode ser implementada sem limitações no ambiente multiplataforma, conforme os dados obtidos na análise exploratória.

Considerando a necessidade de **extração de metadados de arquivos de mídia**, apenas 25% dos especialistas estão de acordo com o que foi obtido na análise exploratória, isto é, é possível a extração de metadados, porém com limitações nos dados obtidos.

Foi questionado se a abordagem multiplataforma pode ser utilizada em qualquer cenário de desenvolvimento móvel, onde 75% dos especialistas discordaram. Os demais concordaram com a afirmação, no entanto, acreditam que poderão ocorrer algum tipo de restrição no decorrer do desenvolvimento do aplicativo. Também foi questionado se com a evolução das ferramentas multiplataforma, elas não apresentam mais limitações quando comparadas com as nativas, onde 80% dos especialistas discordaram.

Foi percebido que muitos especialistas não souberam opinar ou estavam equivocados sobre muitas das questões anteriores e mesmo assim nenhum acredita que o desenvolvimento multiplataforma pode ser usado em qualquer cenário sem restrição. Além disso, a grande maioria acredita que as ferramentas são limitadas quando comparadas às nativas. De fato, o multiplataforma não é a melhor opção para todos os cenários e ainda apresenta

limitações quando comparado à abordagem nativa, porém os especialistas compactuam com essas ideias mesmo sem conhecer as reais capacidades do ambiente multiplataforma, indicando um possível prejulgamento da abordagem.

6 Conclusão

Na primeira etapa deste trabalho foi realizado um levantamento na literatura das características do desenvolvimento móvel nativo e multiplataforma para compreender as diferenças entre ambos. Ao final do estudo teórico, foi realizado um estudo de uso, onde um aplicativo nativo para iOS foi replicado utilizando o Ionic, ferramenta multiplataforma difundida no mercado para desenvolvimento multiplataforma, a fim de obter uma comprovação das diferenças encontradas na literatura, descritas no Capítulo 2.

Com o estudo de uso finalizado, chegou-se à conclusão de que não se pode negar a hipótese **H1**, que sugere que as ferramentas multiplataforma estão evoluídas ao ponto de apresentarem mais vantagens do que desvantagens. Todas as funcionalidades do aplicativo escolhido, que foram planejadas para serem desenvolvidas no ambiente multiplataforma, puderam ser desenvolvidas, não havendo quaisquer limitações quanto ao uso dos recursos nativos do dispositivo necessários para o projeto selecionado. O *app* multiplataforma se assemelhou muito ao nativo em relação a aparência e usabilidade, o que confirma a ideia de que as ferramentas multiplataforma estão cada vez mais se aproximando das nativas. Também foi possível perceber que as ferramentas para desenvolvimento multiplataforma evoluíram muito desde sua criação, o que as tornam, hoje, uma opção que deve ser considerada no momento da criação de um novo *app*, avaliando cada caso.

Na segunda etapa deste trabalho foi realizada uma análise exploratória onde algumas funcionalidades foram selecionadas a fim de avaliar a possibilidade e complexidade de desenvolvimento no ambiente multiplataforma quando comparado ao nativo. Para isso foram selecionadas funcionalidades encontradas em aplicativos populares, analisada a viabilidade da criação dessas funcionalidades no ambiente multiplataforma e, para aquelas que foram viáveis, foram comparados trechos de códigos que as implementassem em iOS, Android e Ionic.

Com base no estudo de uso e na análise exploratória, pode-se negar a hipótese **H2**, que afirma que, ao longo do tempo, as possíveis desvantagens do desenvolvimento multiplataforma serão sanadas ou mitigadas. Foi possível perceber, ao longo da execução do trabalho, que os gargalos antes vistos para a abordagem multiplataforma, não condizem mais com a realidade, no entanto, sempre haverá um *gap* entre as tecnologias nativas e as multiplataformas, pois sempre será necessário que seja implementado uma forma de utilizar os novos recursos de cada plataforma trazidos pelas próprias *SDKs*. Além disso, para aplicações que utilizem *smartwatches*, *widgets* ou assistentes pessoais, não é possível a utilização de tecnologias multiplataforma.

A partir das análises das hipóteses levantadas, é possível responder à questão

problema deste trabalho, sendo ela:

Quais as vantagens e desvantagens do desenvolvimento multiplataforma de aplicações móveis em relação ao desenvolvimento nativo?

O desenvolvimento multiplataforma apresenta como vantagem a criação de um aplicativo com a implementação de apenas um código e a sua distribuição em várias lojas de aplicativos. Diversas outras vantagens podem ser derivadas, como a redução de tempo de desenvolvimento, custo e esforço, pois requer apenas o domínio de um ambiente de desenvolvimento e possui apenas um código para ser mantido e evoluído. Além disso, é possível imitar a aparência e a usabilidade dos aplicativos nativos. Conforme o estudo de uso realizado, foi possível perceber que a diferença de performance entre os dois aplicativos, nativo e multiplataforma, é imperceptível ao usuário final, o que nos mostra que para o nicho de aplicativos abordados neste trabalho, as diferenças de performance citadas na literatura não se aplicam mais.

Embora as ferramentas multiplataformas estejam em constante evolução, sempre haverá um *gap* em relação às nativas, o que faz com que as aplicações desenvolvidas utilizando tecnologias multiplataforma não estejam ao par com o que há de mais novo em cada plataforma (iOS, Android). Por exemplo, com o lançamento do iOS 10, foi liberada a *API SiriKit* para integrar aplicativos de terceiros com a assistente pessoal da Apple e ainda não há como realizar esta integração por meio de tecnologias multiplataforma. Além disso, existem alguns casos específicos que não podem ser desenvolvidos com multiplataforma, como *widgets* e aplicativos para *smartwatches*, o que limita as possibilidades do multiplataforma quando comparado ao nativo.

O desenvolvimento móvel requer uma análise aprofundada de uma série de fatores, como mercado, público e tecnologias para decidir qual abordagem é a mais adequada para cada situação. É importante ressaltar que a abordagem nativa não é melhor que a multiplataforma ou vice-versa, sendo apenas distintas e deve-se avaliar qual utilizar caso a caso. Para cada situação existem fatores que devem ser avaliados de uma maneira conjunta e os selecionados como mais importantes pelos autores são apresentados na Figura 34 e explicados a seguir.

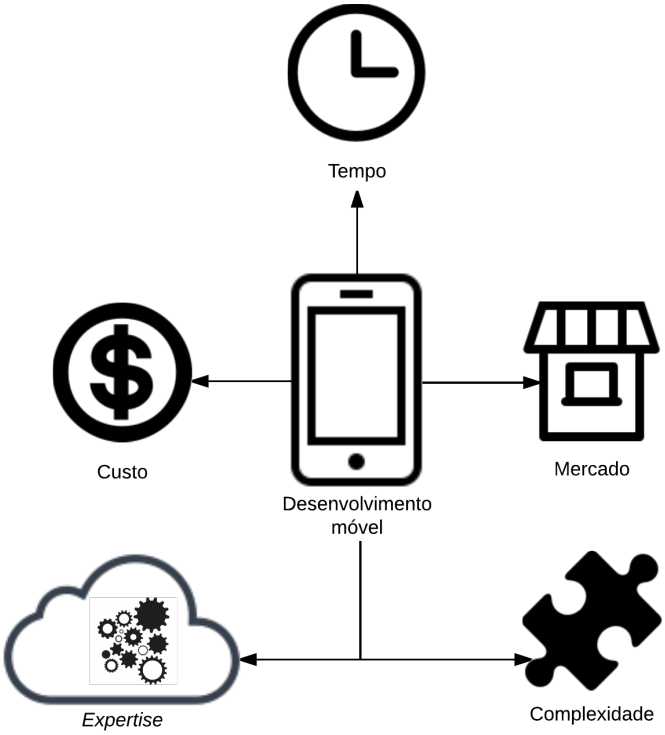

Figura 34 – Fatores a serem avaliados no desenvolvimento móvel.

- **Tipo e complexidade da aplicação**: cada aplicação possui requisitos diferentes e próprios que originam necessidades e dificuldades inerentes daquele aplicativo. Com isso, deve-se avaliar, com base nos requisitos da aplicação, qual abordagem suporta melhor o *app*;
- ***Expertise* da equipe**: cada equipe possui um conjunto único de habilidades e conhecimentos. No momento da escolha de uma abordagem, esses conhecimentos devem ser levados em consideração, visto que é a equipe de desenvolvimento que irá conceber o produto final. Se a equipe possui mais conhecimentos em uma abordagem do que em outra, isso pode ser um indicativo de qual abordagem escolher;
- **Nicho de mercado**: Cada plataforma móvel (iOS, Android, *Windows Phone*, etc), domina uma parcela do mercado e possui um grupo de usuários com características, opiniões, necessidades e gostos próprios atrelados à plataforma que usam. Com isso, no momento de criar um *app* deve-se pensar para quem é esse aplicativo. Se ele for concebido para suprir uma demanda de um grupo específico, talvez não haja a necessidade de criá-lo para várias plataformas;
- **Prazo de desenvolvimento**: Quanto mais plataformas para atender, maior é o tempo necessário para desenvolver a solução. Se o prazo do projeto for apertado para desenvolvimento de mais de uma solução nativa, há de considerar o desenvolvimento multiplataforma, visto que apenas será codificada uma solução que poderá atender várias plataformas diferentes;

- **Capital disponível para investimento**: Desenvolver para plataformas nativas exige ambiente, infraestrutura e conhecimentos diferentes para cada plataforma. Dessa forma, quanto mais plataformas se quer abarcar, mais custoso o projeto será. Uma solução multiplataforma pode ser mais viável economicamente dependendo da situação;

6.1 Limitações do trabalho

Uma limitação deste trabalho é o fato da comparação entre abordagens ter sido feita considerando uma tecnologia multiplataforma baseada em componentes *web*. Existem outros tipos de tecnologia multiplataforma que podem não se encaixar nas conclusões obtidas neste trabalho por não ficarem restritas à execução em *web views*, por exemplo, o Xamarin.

Outra limitação é o tipo de aplicativo explorado neste trabalho. Foram avaliados e desenvolvidos aplicativos de sistemas de informação móveis que acessam e mantém informações em servidores. Outros tipos de aplicativos como jogos e aplicações seguras, como bancos eletrônicos e mensageiros, possuem requisitos de performance e privacidade que podem impactar na avaliação das abordagens.

6.2 Trabalhos Futuros

Como trabalho futuro pode-se avaliar empiricamente ferramentas para desenvolvimento multiplataforma não baseadas em tecnologias *web*, por meio do desenvolvimento de aplicativos e da comparação destes com *apps* desenvolvidos nativamente.

Outro trabalho interessante seria realizar uma análise comparativa do desenvolvimento multiplataforma e nativo de aplicativos de nichos distintos aos aqui abordados, como jogos ou aplicações que possuam requisitos críticos de performance e/ou segurança.

Referências

Android. *Android Interfaces and Architecture | Android Open Source Project*. 2016. Disponível em: <http://source.android.com/devices/index.html>. Citado 2 vezes nas páginas 31 e 32.

Android. *Meet Android Studio | Android Studio*. 2016. Disponível em: <https://developer.android.com/studio/intro/index.html>. Citado na página 32.

Apple Inc. News, *Apple - Hot News*. 2007. Disponível em: <http://web.archive.org/web/20071020040652/http://www.apple.com/hotnews>. Citado na página 28.

Apple Inc. *About the iOS Technologies*. 2014. Disponível em: <https://developer.apple.com/library/ios/documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/Introduction/Introduction.html>. Citado 2 vezes nas páginas 28 e 29.

Apple Inc. *Apple Developer Program - Apple Developer*. 2016. Disponível em: <https://developer.apple.com/programs/>. Citado na página 30.

Apple Inc. *Developing for iOS 9 - Apple Developer*. 2016. Disponível em: <https://developer.apple.com/ios/>. Citado na página 28.

Apple Inc. *Submitting Your App to the Store*. 2016. Disponível em: <https://developer.apple.com/library/ios/documentation/IDEs/Conceptual/AppDistributionGuide/SubmittingYourApp/SubmittingYourApp.html>. Citado na página 30.

Apple Inc. *Swift - Apple (BR)*. 2016. Disponível em: <http://www.apple.com/br/swift/>. Citado na página 30.

BARTH, N. *ANÁLISE COMPARATIVA DE FERRAMENTAS DE DESENVOLVIMENTO DE APLICATIVOS MÓVEIS MULTIPLATAFORMA*. UNIVERSIDADE REGIONAL DE BLUMENAU, 2014. Disponível em: <http://dsc.inf.furb.br/arquivos/tccs/monografias/2014_1_nikson-barth_monografia.pdf>. Citado na página 38.

BEZERRA, P. T.; SCHIMIGUEL, J. *Desenvolvimento de aplicaçõeses mobile cross-platform utilizando phonegap*. 2016. Disponível em: <http://eumed.net/cursecon/ecolat/br/16/phonegap.html>. Citado 2 vezes nas páginas 33 e 42.

CEVALLOS, E. A. *Case Study on Mobile Applications UX: Effect of the Usage of a Cross Platform Development Framework*. Tese (Master Thesis) — UNIVERSIDAD POLITÉCNICA DE MADRID, Madrid, jun. 2014. Disponível em: <http://oa.upm.es/30422/1/EMSE-2014-05_Esteban_Angulo-1.pdf>. Citado na página 23.

CHARLAND, A.; LEROUX, B. Mobile Application Development: Web vs. Native. *Commun. ACM*, v. 54, n. 5, p. 49–53, 2011. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/1941487.1941504>. Citado 2 vezes nas páginas 38 e 43.

CORRAL, L.; JANES, A.; REMENCIUS, T. Potential Advantages and Disadvantages of Multiplatform Development Frameworks - A Vision on Mobile Environments. *Procedia*

Computer Science, v. 10, p. 1202–1207, jan. 2012. ISSN 1877-0509. Disponível em: <http://www.sciencedirect.com/science/article/pii/S1877050912005303>. Citado 5 vezes nas páginas 23, 27, 37, 38 e 43.

DRIFTY. *About Ionic - Ionic Components*. 2016. Disponível em: <http://ionicframework.com/docs/overview/#about>. Citado na página 34.

DRIFTY. *Installing Ionic and its Dependencies - Ionic Framework*. 2016. Disponível em: <http://ionicframework.com/docs/guide/installation.html>. Citado na página 35.

DRIFTY. *Ionic: Advanced HTML5 Hybrid Mobile App Framework*. 2016. Disponível em: <http://ionicframework.com/>. Citado 2 vezes nas páginas 34 e 36.

DRIFTY. *Ionic Concepts - App Structure - Ionic Framework*. 2016. Disponível em: <http://ionicframework.com/docs/concepts/structure.html>. Citado na página 35.

DRIFTY. *Ionic Creator*. 2016. Disponível em: <http://ionic.io/products/creator>. Citado na página 36.

DRIFTY. *Ionic Lab*. 2016. Disponível em: <http://lab.ionic.io/>. Citado na página 36.

DRIFTY. *Ionic Platform*. 2016. Disponível em: <http://ionic.io/platform>. Citado na página 36.

DRIFTY. *Ionic Play*. 2016. Disponível em: <http://play.ionic.io>. Citado na página 36.

DRIFTY. *The Ionic View App | Share your apps with the world*. 2016. Disponível em: <http://view.ionic.io/>. Citado na página 37.

EL-KASSAS, W. S. et al. Taxonomy of Cross-Platform Mobile Applications Development Approaches. *Ain Shams Engineering Journal*, 2015. ISSN 2090-4479. Disponível em: <http://www.sciencedirect.com/science/article/pii/S2090447915001276>. Citado 5 vezes nas páginas 27, 28, 32, 37 e 38.

FRAMEWORK, I. *ngCordova - Document and Examples*. 2016. Disponível em: <http://ngcordova.com/docs/plugins/dialogs/>. Citado na página 55.

GOOGLE. *AngularJS - Superheroic JavaScript MVW Framework*. 2016. Disponível em: <https://angularjs.org/>. Citado na página 33.

HEITKOTTER, H.; HANSCHKE, S.; MAJCHRZAK, T. A. Evaluating cross-platform development approaches for mobile applications. In: *Web information systems and technologies*. Springer, 2013. p. 120–138. Disponível em: <http://link.springer.com/chapter/10.1007/978-3-642-36608-6_8>. Citado 3 vezes nas páginas 28, 30 e 32.

HEITKöTTER, H.; HANSCHKE, S.; MAJCHRZAK, T. A. Comparing Cross-platform Development Approaches for Mobile Applications. In: *Web information systems and technologies*. Springer, 2013. p. 120–138. Disponível em: <http://link.springer.com/chapter/10.1007/978-3-642-36608-6_8>. Citado na página 23.

HOLZINGER, A.; TREITLER, P.; SLANY, W. Making Apps Useable on Multiple Different Mobile Platforms: On Interoperability for Business Application Development on Smartphones. In: QUIRCHMAYR, G. et al. (Ed.). *Multidisciplinary Research and Practice for Information Systems*. Springer Berlin Heidelberg, 2012, (Lecture Notes in Computer Science, 7465). p. 176–189. ISBN 978-3-642-32497-0 978-3-642-32498-7. DOI: 10.1007/978-3-642-32498-7_14. Disponível em: <http://link.springer.com/chapter/10.1007/978-3-642-32498-7_14>. Citado 2 vezes nas páginas 37 e 38.

JOBE, W. Native Apps Vs. Mobile Web Apps. *International Journal of Interactive Mobile Technologies (iJIM)*, v. 7, n. 4, p. 27, out. 2013. ISSN 1865-7923. Disponível em: <http://online-journals.org/i-jim/article/view/3226>. Citado na página 28.

MEIER, R. *Creating Better User Experiences on Google Play | Android Developers Blog*. 2015. Disponível em: <http://android-developers.blogspot.com.br/2015/03/creating-better-user-experiences-on.html>. Citado na página 32.

PAPAJORGJI, P. *Automated Enterprise Systems for Maximizing Business Performance*. 1 edition. ed. Hershey, PA: IGI Global, 2015. ISBN 978-1-4666-8841-4. Citado na página 30.

PREZOTTO, E.; BONIATI, B. Estudo de Frameworks Multiplataforma Para Desenvolvimento de Aplicações Mobile Híbridas. 2014. Citado 2 vezes nas páginas 23 e 42.

REBOUAS, M. et al. An Empirical Study on the Usage of the Swift Programming Language. In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. [S.l.: s.n.], 2016. v. 1, p. 634–638. Citado na página 30.

RODRíGUEZ, A. M.; BALDRICH, R. Diseño e implementación de una aplicación multidispositivo en un entorno HTML5. 2015. OCLC: 878531227. Citado na página 34.

SHAKSHUKI, E. M. et al. Component based Framework to Create Mobile Cross-platform Applications. *Procedia Computer Science*, v. 19, p. 1004–1011, jan. 2013. ISSN 1877-0509. Disponível em: <http://www.sciencedirect.com/science/article/pii/S1877050913007485>. Citado na página 37.

STARK, J. *Building iPhone Apps with HTML, CSS, and JavaScript*. O'Reilly Media, 2010. ISBN 978-0-596-80578-4. Disponível em: <http://shop.oreilly.com/product/9780596805791.do>. Citado 2 vezes nas páginas 23 e 32.

URSINO, D.; CAPANNA, C. A. AngularJS - un framework di frontiera per la realizzazione di siti Web. 2015. Disponível em: <http://www.barbiana20.unirc.it/wp-content/uploads/2015/07/Tesi-Cristiano-finale-2.pdf>. Citado na página 34.

Apêndices

APÊNDICE A – Questionário

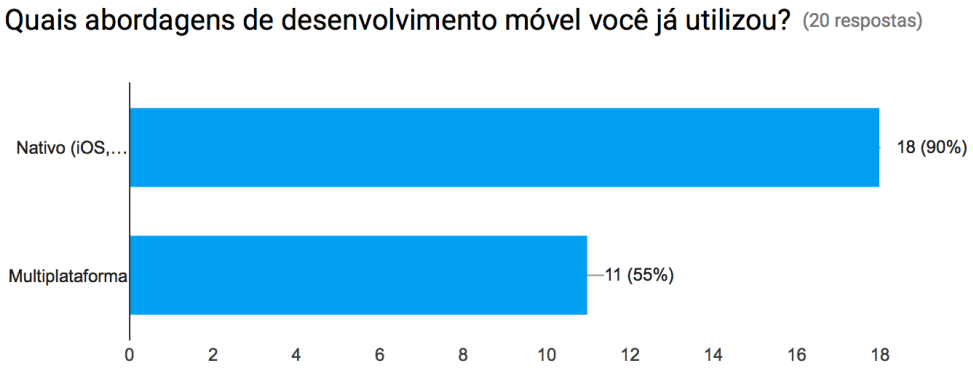

Figura 35 – Questão 1

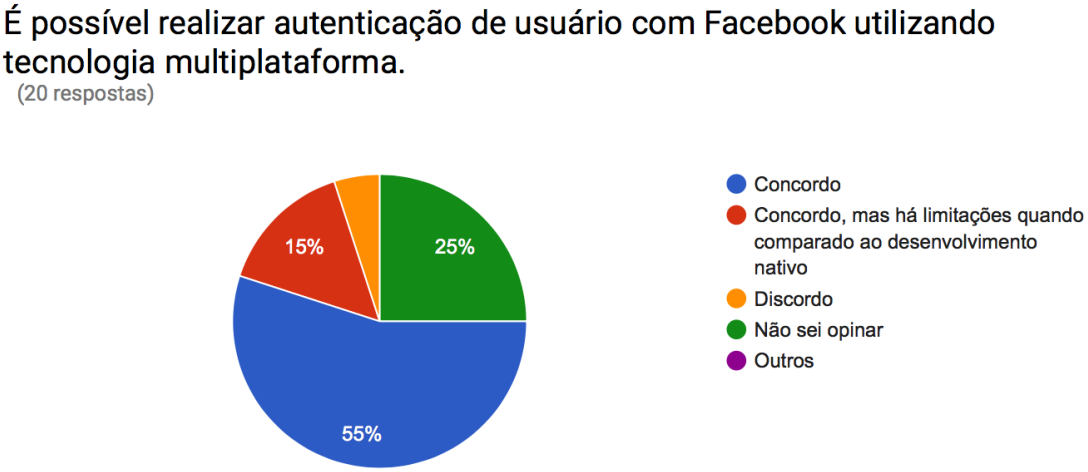

Figura 36 – Questão 2

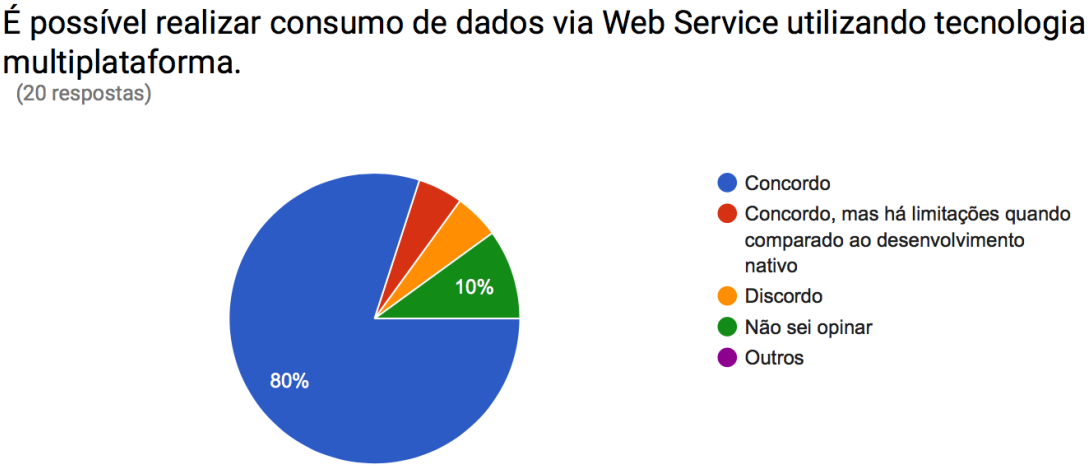

Figura 37 – Questão 3

A utilização do leitor biométrico para fins de autenticação de usuário é possível em um desenvolvimento multiplataforma.
(20 respostas)

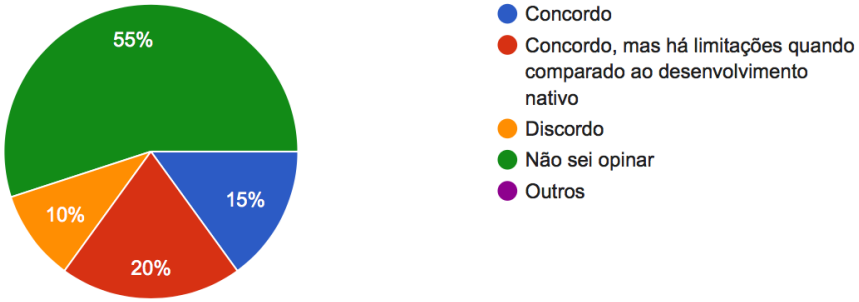

Figura 38 – Questão 4

Considerando a necessidade de extrair metadados de arquivos de mídia, é possível extraí-los utilizando tecnologias multiplataforma.
(20 respostas)

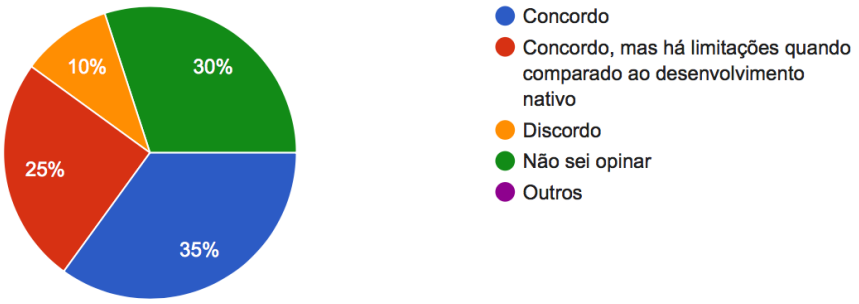

Figura 39 – Questão 5

É possível enviar e-mail e SMS diretamente pelo aplicativo se este foi desenvolvido em multiplataforma.
(20 respostas)

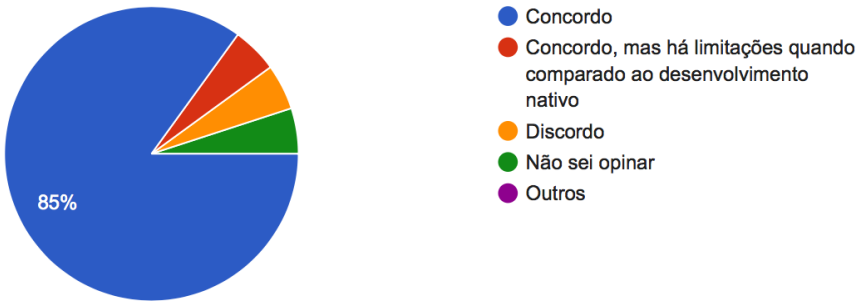

Figura 40 – Questão 6

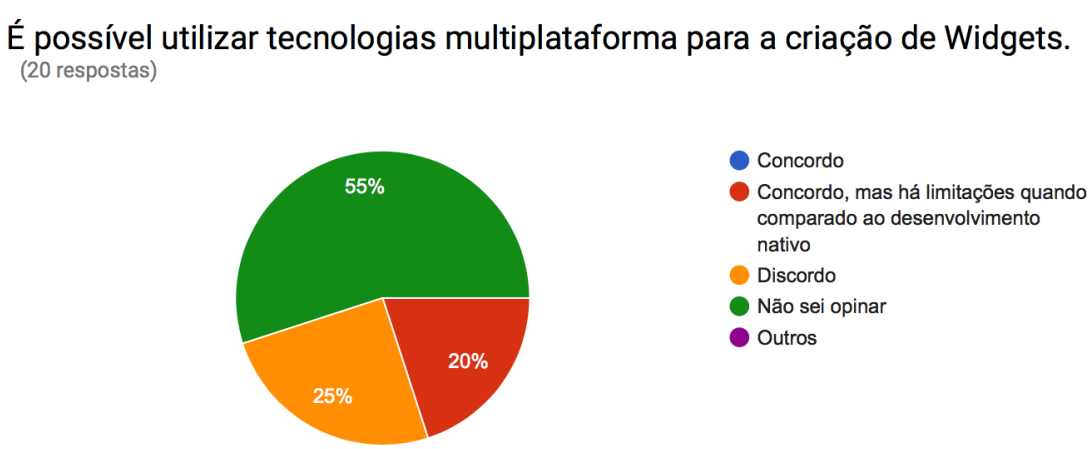

Figura 41 – Questão 7

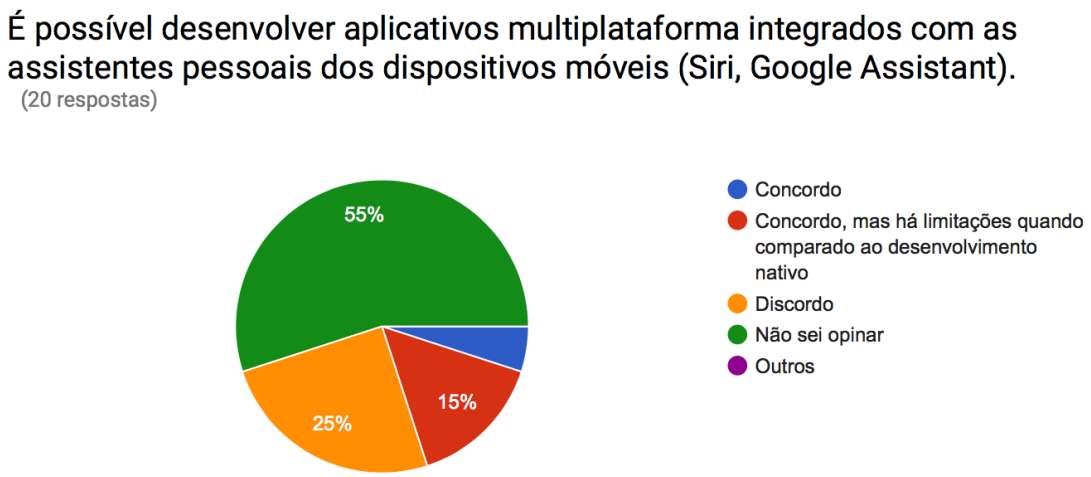

Figura 42 – Questão 8

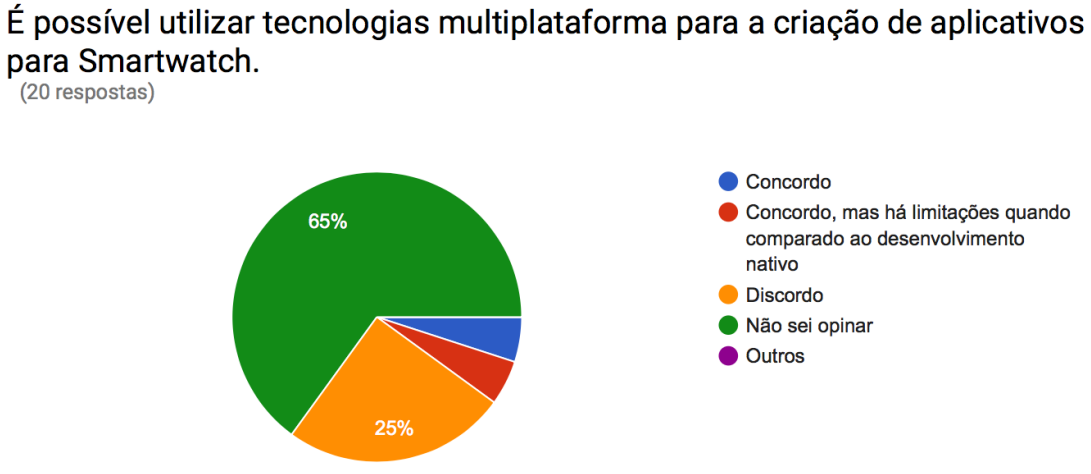

Figura 43 – Questão 9

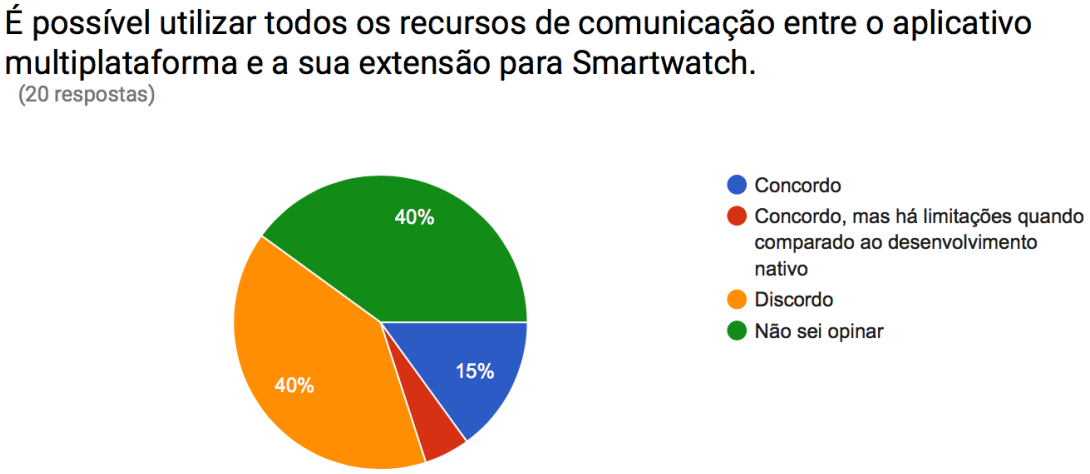

Figura 44 – Questão 10

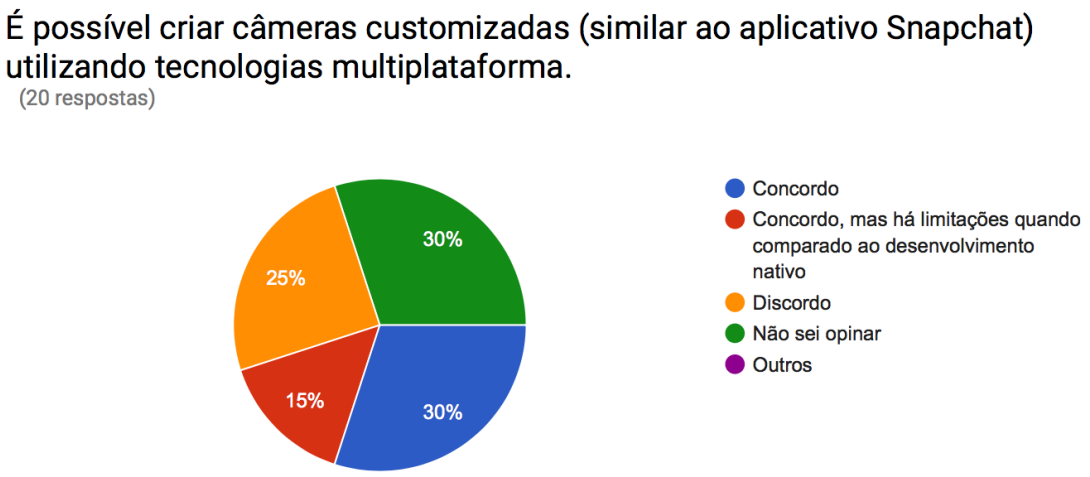

Figura 45 – Questão 11

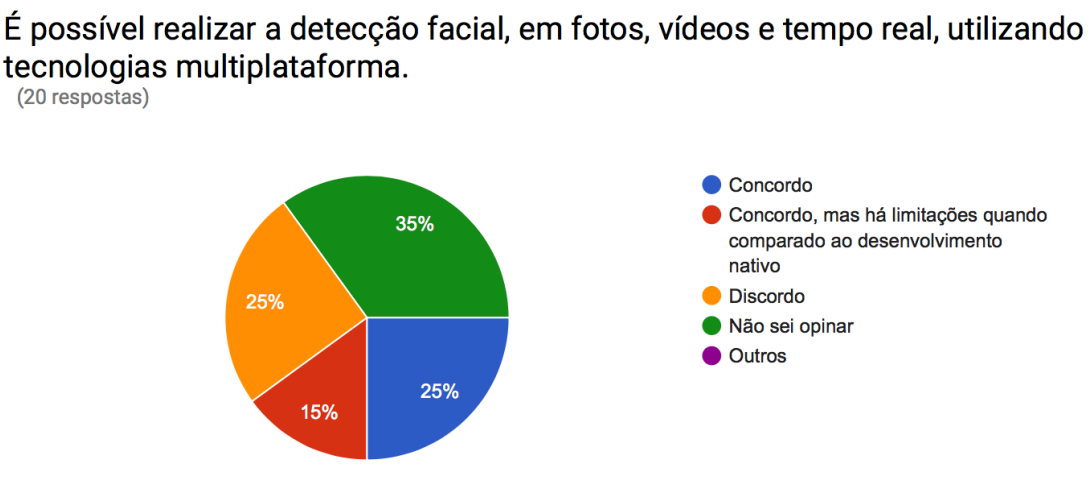

Figura 46 – Questão 12

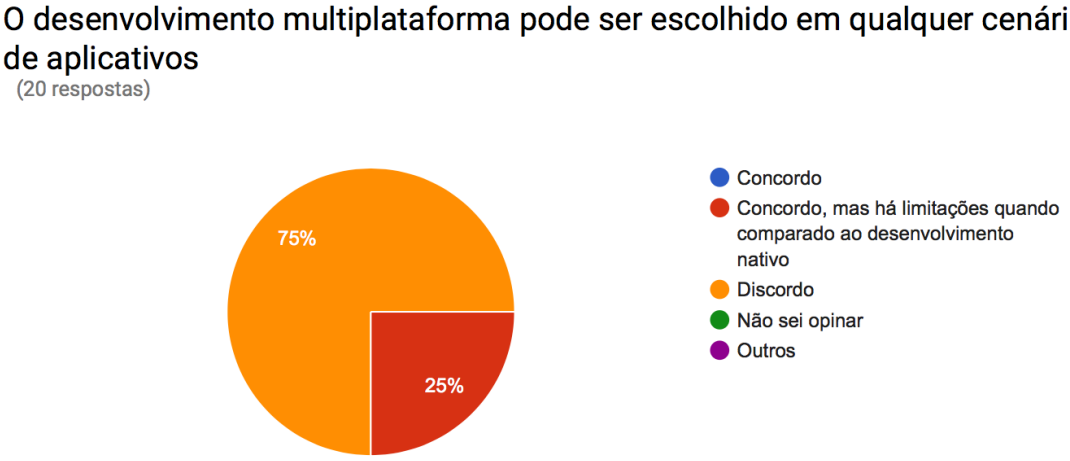

Figura 47 – Questão 13

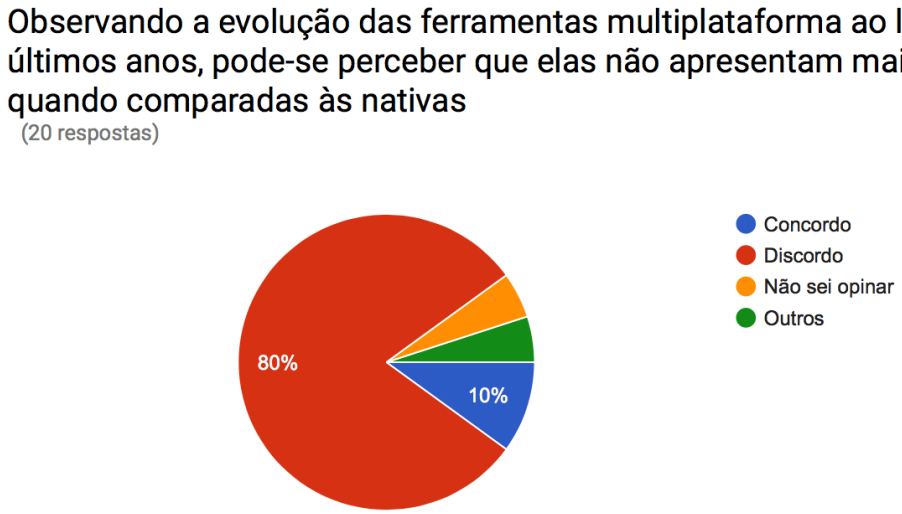

Figura 48 – Questão 14