UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
CURSO DE BACHARELADO EM ENGENHARIA DA COMPUTAÇÃO

Leonardo Pires dos Santos

DESEMPENHO DE APLICAÇÕES MÓVEIS UTILIZANDO IMPLEMENTAÇÃO NATIVA OU FRAMEWORKS MULTIPLATAFORMAS

RECIFE
2018

UNIVERSIDADE FEDERAL DE PERNAMBUCO

CENTRO DE INFORMÁTICA

CURSO DE BACHARELADO EM ENGENHARIA DA COMPUTAÇÃO

Leonardo Pires dos Santos

DESEMPENHO DE APLICAÇÕES MÓVEIS UTILIZANDO IMPLEMENTAÇÃO NATIVA OU FRAMEWORKS MULTIPLATAFORMAS

Monografia apresentada ao Centro de Informática (CIN) da Universidade Federal de Pernambuco (UFPE), como requisito parcial para conclusão do Curso de Engenharia da Computação, orientada pelo professor Eduardo A. G. Tavares.

RECIFE

2018

UNIVERSIDADE FEDERAL DE PERNAMBUCO

CENTRO DE INFORMÁTICA

CURSO DE BACHARELADO EM ENGENHARIA DA COMPUTAÇÃO

Leonardo Pires dos Santos

DESEMPENHO DE APLICAÇÕES MÓVEIS UTILIZANDO IMPLEMENTAÇÃO NATIVA OU FRAMEWORKS MULTIPLATAFORMAS

Monografia submetida ao corpo docente da Universidade Federal de Pernambuco, defendida e aprovada em 26 de Junho de 2018.

Banca Examinadora:

_____ Orientador

Eduardo Antônio Guimarães Tavares

Doutor

_____ Examinador

Adriano Augusto de Moraes Sarmento

Doutor

Dedico este trabalho de graduação a minha família.

AGRADECIMENTOS

Sou grato pelo apoio da minha família que por todos esses anos me apoiaram bastante (inclusive me forçando a estudar quando eu não queria). Nesses últimos dias, recebi uma grande ajuda deles que sacrificaram muito de seu tempo e talentos para que eu pudesse ter momentos de tranquilidade para realizar o trabalho. A eles eu sou extremamente grato.

Também sou grato pelos meus amigos Victor Casé e Isabela Casé que me ajudaram durante a graduação ao cursarmos muitas disciplinas juntos. Em especial, eles me deram uma grande ajuda na realização dos experimentos deste trabalho e com discussões de possíveis razões para os resultados encontrados.

Sou grato ao conjunto de professores que tive durante a graduação os quais me ajudaram a abrir a mente e ver a tecnologia de um modo mais aprofundado. Em especial, sou grato ao professor Eduardo Tavares que me orientou nesse trabalho e ao professor Adriano Sarmento que marcou alguns pontos positivos de minha graduação.

“Nenhum sucesso na vida compensa o fracasso no lar”.

David O. Mckay

RESUMO

Um obstáculo na área de desenvolvimento de aplicações móveis é a diversidade de plataformas. Isto gera a necessidade de empresas despenderem recursos (de tempo e dinheiro) na implementação de um mesmo software para sistemas operacionais distintos. Duas soluções para este problema surgiram com a criação dos frameworks React Native e Flutter. O primeiro foi desenvolvido pelo Facebook e o segundo pelo Google. Ambos visam utilizar um único código para gerar aplicações Android e iOS. Este trabalho visa fazer um estudo de ambos frameworks e comparar o desempenho e usabilidade de alguns componentes em relação ao desenvolvimento nativo iOS. Para isto, serão implementadas duas aplicações em todas as abordagens (utilizando Swift, React Native e Dart).

Palavras-chave: Desenvolvimento mobile, iOS, React Native, Flutter.

ABSTRACT

A hindrance to the development of mobile applications is the diversity of platforms. This generates a necessity for companies to spend resources (time and money) on the implementation of software on distinct operating systems. Two solutions for this problem emerged with the creation of the React Native and Flutter frameworks. The first was developed by Facebook and the second by Google. Both use one script to generate Android and IOS applications. This research aims to study both frameworks and compare the development and usability of some components related to native IOS development. For this, two applications will be implemented for all approaches (using Swift, React Native and Dart).

Keywords: Mobile development, iOS, React Native, Flutter.

Sumário

1. Introdução

1.1. Motivação

O crescimento das tecnologias (de hardware e software) em dispositivos móveis tem despertado grande interesse de empresas e organizações para implementar suas soluções como aplicativos. Contudo, as plataformas móveis estão se movendo para uma fragmentação ao invés de uma unificação entre elas (Joorabchi, Mesbah, & Kruchten, 2013). Essa diversidade de plataformas tem se tornado um dos aspectos mais desafiadores do desenvolvimento móvel. Cada plataforma tem sua arquitetura, linguagem, benefícios e limitações. Assim, a implementação, manutenção e desenvolvimento de novas funcionalidades em aplicações específicas de cada plataforma (e.g., Android e iOS) demanda tempo e custo para uma empresa.

As principais abordagens no desenvolvimento de um aplicativo são apps nativos, web apps e apps híbridos. O desenvolvimento nativo é o mais comum e utiliza a linguagem própria de cada plataforma (e.g., Java/Kotlin para Android e Objective-C/Swift para iOS). Os web apps são executados em cima de browsers de internet e se beneficiam de frameworks como o web toolkit que possibilita a implementação completa do app usando HTML, CSS e JavaScript (que são elementos familiares a desenvolvedores web). Na abordagem híbrida é possível desenvolver um código mesclando duas ou mais linguagens para o mesmo app (*e.g.*, C/C++ e Java na implementação de um app Android). Estes também podem usar frameworks que são normalmente escritos em uma única linguagem de programação e, após compilados, vão gerar aplicativos que executam em mais de uma plataforma (*e.g.*, uso dos frameworks React Native e Flutter que utilizam respectivamente JavaScript e Dart como principais linguagens de programação).

Cada uma das implementações de apps tem suas vantagens e desvantagens as quais devem ser consideradas antes do desenvolvimento de uma aplicação. As linguagens nativas possuem compatibilidade com todos os componentes de sua plataforma, ou seja, recursos como câmera, Bluetooth, GPS e Wifi são suportados e podem ser manipulados pela linguagem nativa (fato este que nem sempre é alcançado por outras abordagens). Contudo, esta abordagem pode, por exemplo, consumir mais energia em apps de uso intensivo de CPU

em comparação a aplicações escritas em JavaScript (Oliveira, Oliveira, & Castor, 2017). Web apps, por sua vez, normalmente são mais intuitivos de serem implementados por se assemelhar ao desenvolvimento web. Todavia, eles possuem a desvantagem de necessitar de um browser para execução e se limitam ao que é suportado pelo mesmo. As aplicações híbridas muitas vezes também são desenvolvidas utilizando tecnologias web (e.g., HTML5) e, neste caso, se diferem de web apps por usar um browser encapsulado e transparente ao usuário (chamado WebView e UIWebView para Android e iOS respectivamente). Esta abordagem também pode utilizar um compilador que gera código para mais de uma plataforma. Desse modo, ela tem a agilidade no processo de desenvolvimento, manutenção e implementação de novas funcionalidades. Porém, os mesmos podem ter inconsistências no padrão de interface do usuário entre diferentes plataformas, ocupar mais espaço físico (Hansson & Vidhall, 2016) e apresentar menor suporte a componentes específicos de cada sistema operacional.

Diante da fragmentação das plataformas e o surgimento de várias tecnologias que visam agilizar o desenvolvimento móvel, é necessário um levantamento do contexto atual da área tal como uma análise dos frameworks e tendências em evidência.

1.2. Objetivos

Este trabalho tem como objetivo fazer um estudo de duas abordagens multiplataformas e uma nativa. Ademais, é feita uma comparação na implementação e performance das mesmas. O estudo será baseado nos frameworks React Native e Flutter e no desenvolvimento nativo para iOS. O primeiro foi desenvolvido em 2015 pelo Facebook e promete a criação de aplicativos nativos utilizando JavaScript e React (“React Native · A framework for building native apps using React,” n.d.). O segundo foi criado pelo Google e visa a criação de apps com aparência nativa e boa performance de processamento. Por sua vez, a abordagem nativa é suportada pela Apple e tem como linguagem de programação Swift 4. O processo de comparação foi realizado pela implementação de dois aplicativos distintos em cada abordagem (totalizando seis aplicativos). Estes fizeram uma análise do tempo necessário para conclusão de tarefas específicas.

1.3. Metodologia

A comparação entre a abordagem nativa e as multiplataforma (*i.e.*,React Native e Flutter) foi focada em duas aplicações principais. Para isso foram criados dois aplicativos, cada um contemplando uma das aplicações, em cada uma das abordagens (totalizando seis aplicativos). As aplicações nativa, em React Native e Flutter foram desenvolvidas respectivamente em Swift, JavaScript (junto com React) e Dart. Os aplicativos desenvolvidos foram:

1. Buscador de Números Primos - Esta aplicação teve como objetivo simular o uso intensivo da CPU e verificar o tempo para processar uma ação que não envolva a interface de usuário. Desse modo, foi implementado um app que busca todos os números primos entre 2 e X (onde X é um número escolhido pelo usuário).
2. Lista de Contatos - Esta aplicação tem como objetivo observar o tempo utilizado para construir uma interface. Assim, foi implementada uma lista de contatos com 50 itens. Cada célula contém um título e o telefone.

Os aplicativos serão simulados 30 vezes por abordagem. Por fim, foi feito o cálculo da média de tempo gasto para se realizar as tarefas, o desvio padrão e o intervalo de confiança.

1.4. Estrutura do Trabalho

Inicialmente será feita uma breve discussão dos tipos de desenvolvimento de aplicações móveis (capítulo 2). Especificamente será realizado um estudo do desenvolvimento nativo para a plataforma iOS (seção 2.2). Assim, será apresentada a arquitetura, estrutura de um aplicativo e ciclo de vida do mesmo para este sistema operacional.

Em seguida será analisado os frameworks React Native (seção 3.1) e Flutter (seção 3.2). Em ambas sessões serão apresentados os conceitos gerais dos frameworks, suas arquiteturas e ciclo de vida dos componentes.

Após a discussão teórica das três formas de implementação de aplicativos móveis, será descrito os experimentos (capítulo 4) realizados, os resultados adquiridos e explicações para tais desfechos. Por fim, será apresentada uma conclusão (capítulo 5) para o trabalho e possíveis áreas para estudos futuros.

2. Desenvolvimento de Aplicações Móveis

2.1. Contextualização e Abordagens

O avanço tecnológico tem facilitado a busca por conhecimento e a realização de objetivos através de aplicativos móveis (apps). Estes, por sua vez, são softwares complexos que visam utilizar minimamente os recursos de um smartphone para o cumprimento de uma tarefa específica. Desse modo, no processo de desenvolvimento de tais programas, deve-se levar em consideração fatores como usabilidade do usuário, segurança, portabilidade, eficiência, facilidade de manutenção e gasto de energia. Consequentemente, diversas abordagens para criação de apps têm surgido. As principais abordagens são apps desenvolvidos nativamente, web apps e apps híbridos.

A abordagem nativa utiliza kits de desenvolvimento de software (SDK) específicos em cada sistema operacional e dispõe de vantagens no mesmo. A plataforma iOS utiliza as linguagens de programação Objective-C ou Swift e tem o XCode como principal ferramenta de desenvolvimento. Já os dispositivos android utilizam Java ou Kotlin como linguagem de programação e o Android Studio como principal ferramenta de desenvolvimento. Ademais, o desenvolvimento nativo tem o benefício de explorar por completo, ou seja, utilizar em detalhes todas as funcionalidades de cada componente (seja ele de hardware ou software) da plataforma (Willocx, Vossaert, & Naessens, 2016).

Diferente dos apps nativos, a abordagem de aplicações web (*i.e.*, web apps) utiliza o navegador da internet como abstração do sistema operacional para alcançar a característica de aplicativos multiplataforma. Os web apps funcionam como páginas da web (ver figura 1) e não são distribuídas por lojas de apps específicas de cada plataforma (*e.g.*, Google Play Store e Apple Store). Além disso, eles usam de tecnologias web tais como HTML, CSS e JavaScript (as quais são populares dentre desenvolvedores web), necessitam de internet para navegação entre páginas e possuem acesso limitado ao hardware do dispositivo. Também têm o encargo de ser implementados de forma responsiva para se adequar a tamanho de telas distintas de celulares e devem manter compatibilidade entre browsers.

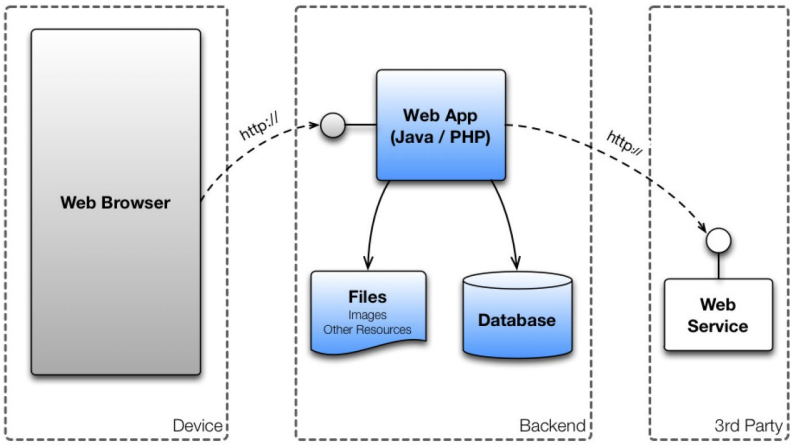

Figura 1: Arquitetura de web apps (Casé, 2015).

Os apps híbridos buscam unir os benefícios de aplicações nativas e web apps. Assim, eles geralmente são referenciados como uma abordagem que é escrita uma única vez e, após compilada, pode ser executada em mais de uma plataforma. Tal como as aplicações nativas, apps implementados por essa abordagem também são distribuídos pelas lojas oficiais de cada plataforma. Os métodos para desenvolvimento híbrido crescem constantemente de modo que é difícil fazer uma definição sólida e única para esse processo. De fato, Oliveira usa o termo híbrido como sendo o uso de uma linguagem de programação não padrão (*i.e.*, que não é java/kotlin nem Objective-C/Swift respectivamente para android e iOS) para uma dada plataforma através de kits de desenvolvimento nativos (NDK) (Oliveira et al., 2017). Já Willocx busca categorizar o desenvolvimento híbrido de acordo com as tecnologias usadas no mesmo (Willocx et al., 2016). Um outro ponto de vista é dado por Hasan que classifica a abordagem híbrida de acordo com o uso ou não de tecnologia web (Hasan & Haque, 2016). Para fins do atual trabalho, pode-se considerar o desenvolvimento híbrido como sendo uma forma de implementação que não usa linguagem nativa da plataforma e que pode gerar aplicações multiplataformas.

Neste capítulo será detalhado o processo de criação de uma aplicação pela abordagem nativa na plataforma iOS. Detalhes sobre o desenvolvimento no sistema operacional android não serão avaliados neste trabalho.

2.2. Desenvolvimento Nativo iOS

Os dispositivos da Apple rodam sobre um sistema operacional próprio chamado iOS. Através deste um desenvolvedor consegue acessar o máximo das funcionalidades e performance do hardware. Desse modo, para desenvolver de modo nativo em tal plataforma é necessário ter um Mac (executando o iOS), o ambiente de desenvolvimento integrado (*i.e.*, IDE) Xcode e o kit de desenvolvimento de software (*i.e.*, SDK) instalado. As linguagens padrão de desenvolvimento são Objective-C e Swift tendo cada uma suas vantagens e desvantagens (Kirichok, n.d.). Ademais, com a ajuda do Xcode, o processo de desenvolvimento pode ser facilitado pois a aplicação pode ser simulada em um emulador nativo ou em um dispositivo físico. Desse modo, o desenvolvedor da aplicação pode entender como a mesma está se comportando em diferentes cenários. Outra vantagem promovida pelo Xcode é o uso do "*Instruments*" que possibilita a captura e verificação de performance (processamento da CPU, memória etc) durante a execução do app.

2.2.1. Arquitetura

A arquitetura do iOS é descrita em quatro principais camadas (ver figura 2): Cocoa Touch, Mídia, Core Services e Core OS. Apesar das camadas estarem uma sobre a outra, uma aplicação pode utilizar as funcionalidades de qualquer camada. Todavia, é importante frisar que quanto mais baixa a camada, maior a complexidade da geração/modificação de código.

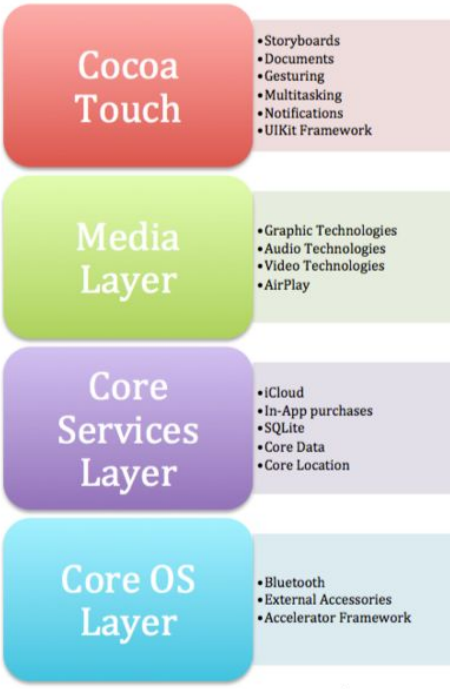

Figura 2: Arquitetura iOS e elementos comuns de cada camada (Gondi, 2015).

Cocoa Touch é a camada que provê tecnologias para construir a interface do usuário, responder a eventos e gerenciar o comportamento do app (Apple, 2018b). Alguns exemplos de funcionalidades atribuídas por esta camada são: criação de storyboards, uso do auto-layout, notificações locais, serviço de "*push notification*", reconhecedor de gestos e View Controllers padrão.

A camada de mídia disponibiliza os principais frameworks para o uso e controle de multimídias. Assim, essa camada facilita o uso de mídias ao encapsular atividades complexas e disponibilizar as mesmas por chamadas de APIs. Alguns exemplos de uso são: tocar músicas, fazer animações, visualizar vídeos e construir gráficos.

Core Services é a camada que provê serviços básicos que não estejam diretamente ligados com a UI. Alguns exemplos de funcionalidades dessa camada são: integração com mídia social, serviço de armazenamento de arquivos no iCloud, escrita/leitura de arquivos, APIs para uso de concorrência, reconhecimento de voz e armazenamento criptografado de dados sigilosos.

A camada Core OS disponibiliza serviços de baixo nível relacionados ao hardware e rede (Apple, 2018a). Esta camada também oferece serviços de segurança do aplicativo (*e.g.*, utilizando "*App Sandbox*" que protege o app de softwares maliciosos).

Além das camadas descritas anteriormente existe uma quinta camada que possui o kernel iOS e drivers do dispositivo. Ademais, ela dá suporte para sistema de arquivos, extensões do kernel, linguagens de programação e segurança.

2.2.2. Estrutura de aplicativos iOS

Aplicações iOS utilizam o padrão Model-View-Controller (*i.e.*, MVP) que tem o objetivo de desacoplar o modo como a informação é representada internamente, a forma como a mesma é mostrada ao usuário e a conduta do software ao receber dados do usuário. A figura 3 mostra um diagrama relacionando cada uma dessas partes em um ambiente iOS.

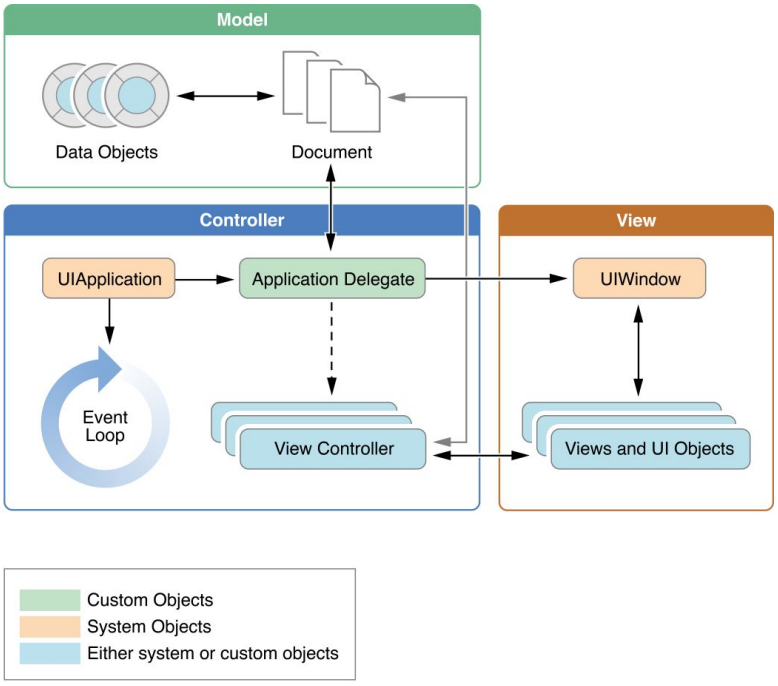

Figura 3: Estrutura de um aplicativo iOS (Apple, 2018c).

O "*Data Model*" (*i.e.*, objeto de modelo de dados) é responsável por lidar com mudanças em arquivos e avisar tais ocorridos ao "*Controller*". Este, por sua vez, é responsável por aceitar inputs do usuário e notificar outros objetos da ação ocorrida. Por fim, "*View*" é o objeto que exibido ao usuário o qual pode responder a ações (*e.g.*, toques na tela). Além disso, todo o sistema e demais objetos têm sua interação facilitada através do objeto "*UIApplication*" (Apple, 2018c).

2.2.3. Ciclo de Vida

A programação nativa para iOS disponibiliza vários eventos que facilitam a identificação e controle de mudanças na view de um app. A figura 4 mostra alguns métodos que são chamados automaticamente pelo View Controller ao identificar uma mudança na view. O método "*viewWillAppear()*", por exemplo, pode ser usado para preparar dados que serão utilizados na construção da view ou que serão apresentados na mesma. Já o método "*viewWillDisappear()*" pode ser usado para salvar mudanças oriundas de atividades em uma tela específica.

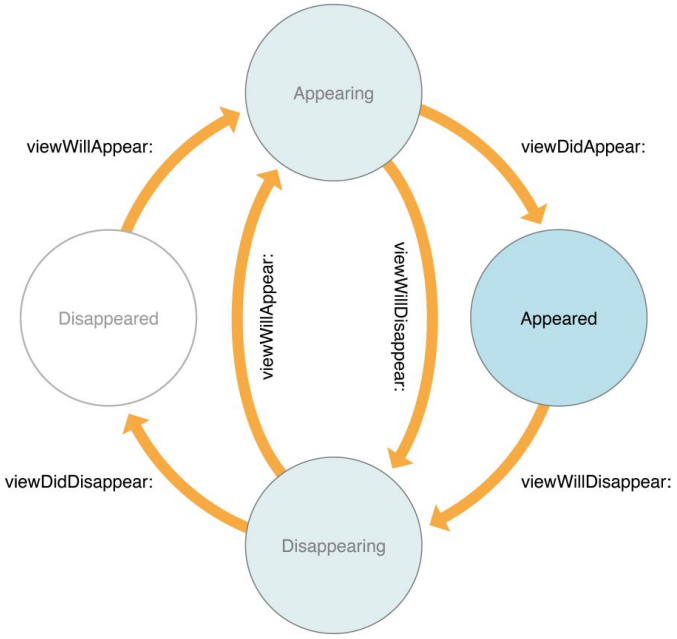

Figura 4: Estados e eventos de transição na programação nativa para iOS (Apple, 2018d).

3. Abordagens Multiplataforma

Como visto no capítulo 2, muitas são as maneiras para se desenvolver uma aplicação. Dentre elas, as abordagens multiplataformas tem se destacado por agilizar o processo de desenvolvimento e diminuir custos para manutenção do app. Assim, kits de desenvolvimento como Corona, Xamarin, Titanium e Unity têm crescido no mercado por oferecerem tal funcionalidade.

Este capítulo visa fazer um estudo de dois frameworks que geram códigos multiplataformas. Os frameworks serão React Native e Flutter.

3.1. React Native

React (ou ReactJS) é uma biblioteca open source escrita em JavaScript para construir interfaces de usuário. Ela foi produzida por Jordan Walke (que era engenheiro de software do Facebook na época) o qual desenvolveu uma extensão sintática do JavaScript chamada JSX (esta foi fortemente influenciada por XHP que é uma extensão de PHP). Além disso, React utiliza um compilador chamado Babel (Tschinder, Ng, Smyth, Sauleau, & Zhu, 2018) o qual dá suporte a JSX. React Native, por sua vez, é um framework criado pelo Facebook para o desenvolvimento de aplicações móveis multiplataforma. Seus autores alegam que é possível fazer apps nativos utilizando JavaScript e React (sendo ambos a base do framework). De fato, isto é possível pois React Native utiliza os mesmos blocos de código de interface de usuário (*i.e.*, UI) que as plataformas Android e iOS (Facebook, 2018d). Ademais, é possível integrar códigos nativos (escritos em Java, Objective-C ou Swift) ao projeto com o framework.

Outro ponto de destaque do framework é a sua capacidade de re-compilar a aplicação instantaneamente e manter seu estado ao modificar qualquer arquivo. Esta funcionalidade é chamada de "Hot Reloading" e tem o objetivo de injetar novas versões dos arquivos enquanto o app está executando (Facebook, 2018c). Para conseguir tal objetivo o framework utiliza o "Hot Module Replacement" (HMR) que observa alterações em arquivos.

3.1.1. Arquitetura

Como falado anteriormente, React Native utiliza JavaScript e Rect como principais linguagens de programação. Pelo fato de JavaScript não ser uma linguagem que roda

nativamente em smartphones, é necessário o uso de uma "bridge" (*i.e.*, ponte) que permite a comunicação entre JavaScript e o processador do dispositivo. Desse modo, dois núcleos distintos se destacam na arquitetura do framework: o núcleo nativo e o ambiente de execução do JavaScript. A figura 5 mostra um diagrama da existência e comunicação por essa ponte. É importante ressaltar que que não existe apenas uma ponte para ligar os dois ambientes descritos anteriormente. De fato, um aplicativo pode ter várias instâncias que controlam partes específicas de um sistema operacional. Desse modo, cada controle do React Native será conectado (através de uma ponte) a um componente semelhante no ambiente JavaScript (Heard, 2017b). Contudo, apesar dessa ponte ser determinante para interligar os dois núcleos, ela pode gerar atrasos no processamento de UI (fator esse demonstrado no experimento 2 do capítulo 5 deste trabalho).

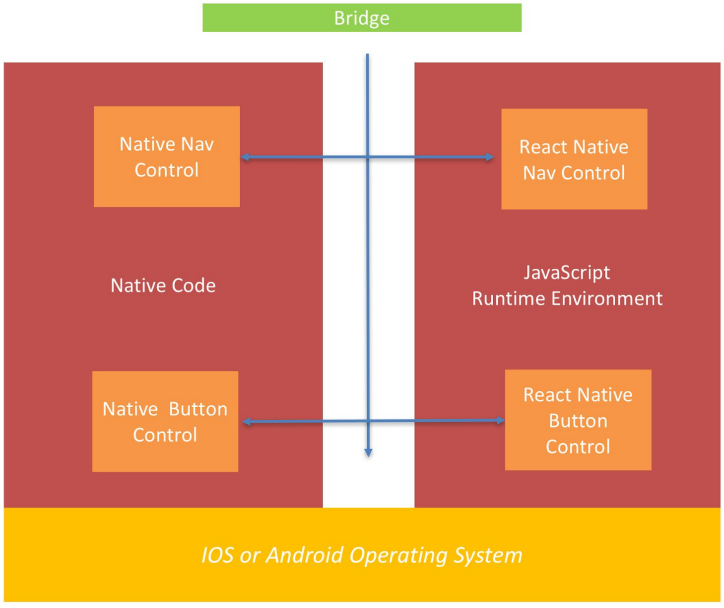

Figura 5: Comunicação entre o núcleo nativo e o ambiente JavaScript no framework React Native. No núcleo esquerdo são encontrados os códigos nativos. Reciprocamente o código em JavaScript e React é encontrado nos menores blocos à direita da figura (Heard, 2017a).

O elemento mais marcante no código que utiliza React Native é o "*Component*" (*i.e.*, Componente). De fato, Componentes permitem o desenvolvedor dividir a UI em pedaços independentes e reusáveis. Os Componentes são como funções em JavaScript recebem

parâmetros (chamados de "*props*") e retornam um elemento informando a aparência da tela (Facebook, 2018a). Dentro de um Componente existem os dados que nunca poderão ser modificado e que foram informados por Componentes pai. Estes recebem o nome de "*props*". Outro tipo de dados, os quais recebem o nome de "*state*" (*i.e.*, estado), são os que poderão sofrer mudanças durante a execução da aplicação. Este último tipo de dado é de extrema importância para o Componente pois ele armazena a atual verdade dos valores do mesmo. Assim, é de extrema importância que o estado seja tratado como se fosse imutável (*i.e.*, o estado nunca seja modificado diretamente como, por exemplo, *this.state = novoValor*). De fato, o estado só deve ser atualizado com o uso do método "*setState()*" que criará um novo estado e definirá o mesmo como estado atual.

3.1.2. Virtual DOM

Tanto o ReactJS quanto o React Native utilizam um conceito familiar no desenvolvimento web que é o "*Document Object Model*" (DOM). Este é responsável por representar todos os elementos apresentados em uma tela por um objeto de hierarquia chamado "*DOM tree*" (*i.e.*, árvore DOM). Desse modo, muitas vezes é menos custoso (em relação a tempo e poder de processamento) atualizar apenas um nó desta árvore do que re-renderizar toda a página. Contudo, à medida que muitos elementos são acrescentados na árvore DOM, o custo para se percorrer todos os nós começa a ser difícil de se manter. Um exemplo desse caso está na técnica de implementação SPA (*i.e.*, "*Single Page Applications"* ou aplicações de página única) a qual páginas web são dinamicamente modificadas sem o usuário ser encaminhado para URLs diferentes. Consequentemente, a árvore DOM pode se tornar bem extensa ao usar tal aplicação.

Diferente do desenvolvimento web, ReactJS aprimora o uso do DOM ao usar um DOM virtual. Este, por sua vez, recebe as modificações em determinados componentes e recalcula a sua nova estrutura. Em seguida, o DOM virtual atualizado é comparado ao DOM do browser e se calcula as mudanças mínimas para que o DOM se iguale ao DOM virtual. Por fim, essas mudanças normalmentes são adicionadas ao DOM de forma assíncrona. Semelhantemente, React Native utiliza uma versão virtual da hierarquia de componentes mostrados na aplicação (Hansson & Vidhall, 2016).

3.1.3. Ciclo de Vida do Componente

A medida que o usuário interage com a aplicação, o estado do Componente é modificado e gera atualizações na DOM. Para melhor identificar tais mudanças, React estabeleceu uma série eventos que são chamados a medida que as alterações vão ocorrendo. Tais eventos formam o ciclo de vida de um Componente em React Native. A figura 6 mostra a ordem de como esses eventos são chamados.

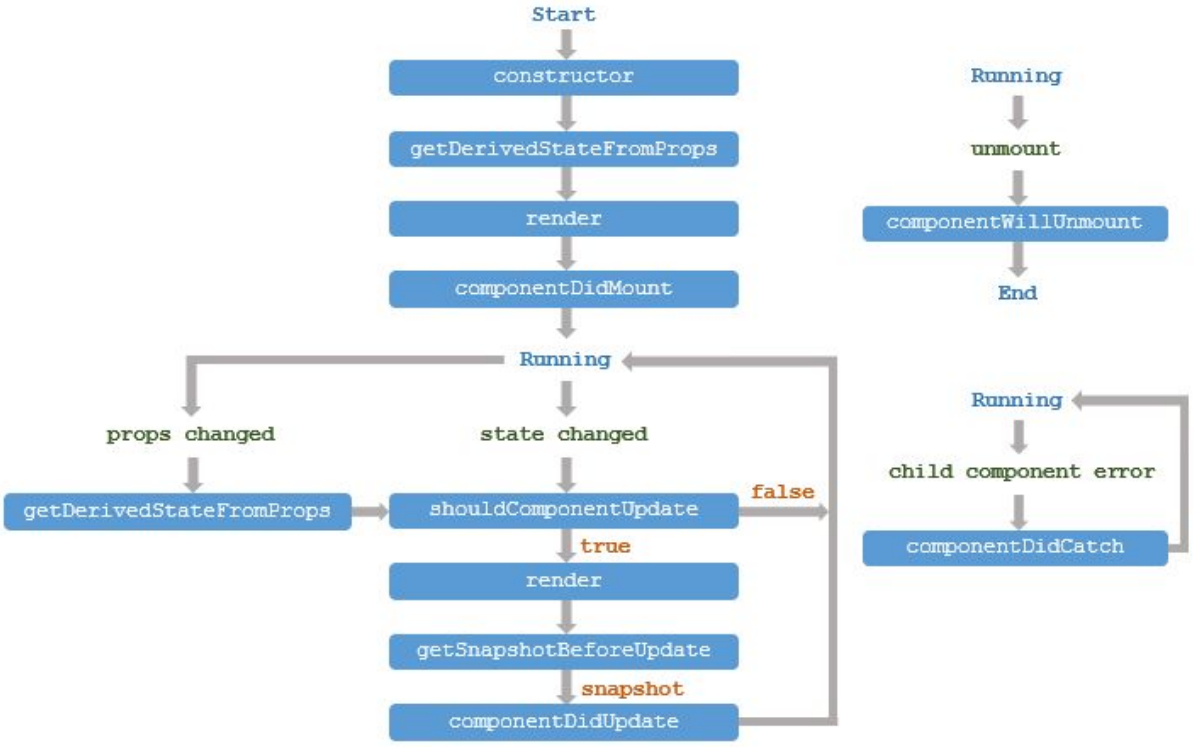

Figura 6: Ciclo de vida do Componente em React (JavaSampleApproach, 2018).

Não cabe a este trabalho detalhar cada um destes eventos. Assim, serão destacados alguns eventos que o autor considera importante para se ter uma visão geral do ciclo de vida do Componente. Uma documentação mais rica pode ser encontrada em (Facebook, 2018b).

O primeiro evento a ser chamado é o *Contructor()*. Este é chamado antes do Componente ser criado (*i.e.*, antes de ser adicionado ao DOM) e é usado para inicializar o estado e/ou vincular eventos a instância do Componente. O evento que marca a finalização da montagem do Componente na tela é o *componentDidMount()*. Ele permite a modificação de elementos adicionados ao DOM. Outro evento importante é o *shouldComponentUpdate()* que

decide se o Componente deve ser re-renderizado ou não. Por fim, o *componentWillUnmount()* marca os últimos momentos antes da finalização do Componente.

3.2. Flutter

Flutter é um framework utilizado para criação de apps de alta performance e fidelidade para as plataformas iOS e Android (Google, 2018e). Ele foi criado pela Google e teve sua versão alfa (v0.0.6) lançada em maio de 2017. Ademais, o framework utiliza Dart como principal linguagem de programação. Esta, por sua vez, também foi criada pela Google e é reconhecida por ter uma sintaxe clara e consistente, ser rápida, ser portável para multiplataformas e ser reativa (Google, 2018a).

Semelhante ao React Native, Flutter tem a funcionalidade "*Hot Reload*" a qual possibilita que, dada qualquer mudança no código, a aplicação seja rapidamente recarregada e mantenha seu estado da última seção. Fator este que agiliza o processo de desenvolvimento e testes do aplicativo. Outro destaque do framework é uso da *Material* que é um sistema de design que pode agilizar o processo de desenvolvimento de aplicações por oferecer componentes reusáveis fáceis de estilizar (Google, 2018h).

3.2.1. Widget

Widget é o bloco fundamental da interface utilizada em Flutter. De fato, elas são responsáveis por dizer a estrutura de um elemento (*e.g.*, um texto, um botão ou uma lista), o estilo (*e.g.,* cor), o aspecto (*e.g.*, margens e paddings) e o modo como devem agir em eventos distintos (*e.g.*, toque na tela). Ademais, os widgets são organizados hierarquicamente de modo que todos os widgets são aninhados em um widget raiz (ver figura 7) (Google, 2018e).

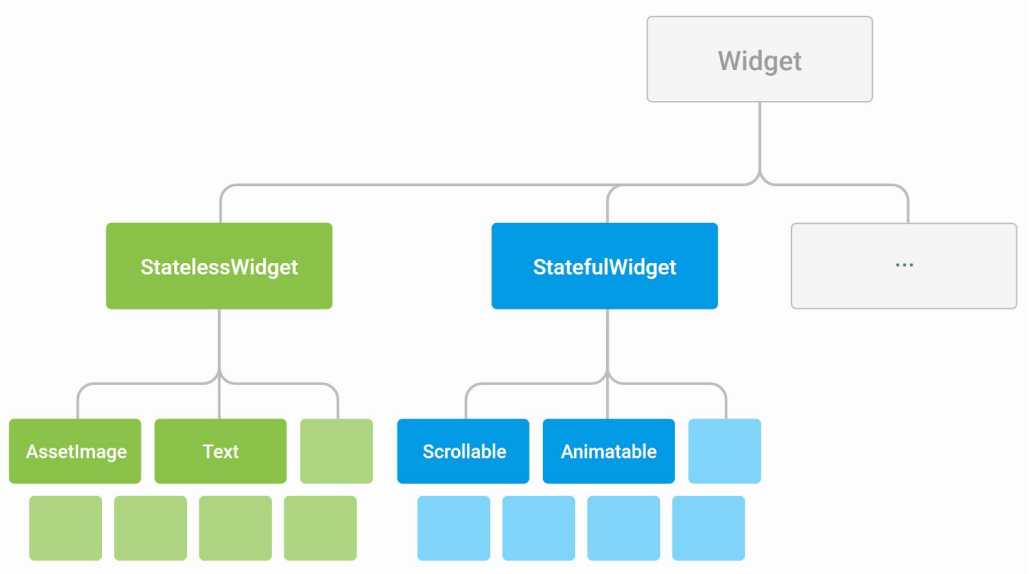

Figura 7: Hierarquia de widgets em Flutter (Google, 2018e).

Como mostrado na figura 7, existem dois tipos de widgets: "*Stateless Widget"* e "*Statefull Widget"*. O primeiro tipo é caracterizado por ser um widget que não muda seu estado. Assim, eles são indicados para criação de interfaces concretas (*i.e.*, que não vão precisar ser modificadas por eventos futuros ou manuseio do usuário). Já o segundo tipo é caracterizado por widgets que alteram o seu estado. Desse modo, eles são normalmente usados para o cumprimento de atividades assíncronas ou para situações que o estado vá mudar em algum ciclo de vida do widget (mudança essa que pode ser oriunda do manuseio do usuário).

3.2.2. Arquitetura

Flutter possui sua arquitetura baseada em camadas (ver figura 8) as quais são bibliotecas responsáveis por funcionalidades específicas em um app. Muitas destas camadas possuem um nome auto-explicativo (*e.g.*, "*Animation*" que é responsável por animações e "*Gesture*" que possui os métodos para lidar com gestos do usuário). Porém, outras não são tão óbvias (*e.g.*, "*Cupertino*" que possui widgets de alta fidelidade ao estilo iOS). Para maiores informações sobre cada biblioteca ver (Google, 2018b).

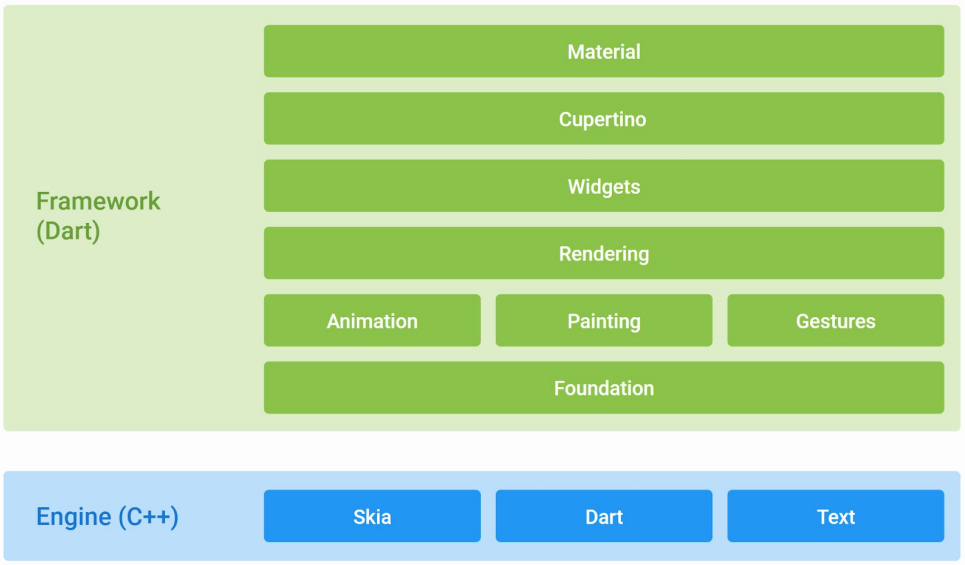

Figure 8: Arquitetura Flutter (Google, 2018e).

A arquitetura do Flutter também foi modulada para aceitar código nativo (seja este em Kotlin, Java, Swift ou Objective-C) através da instalação de pacotes específicos. Isto é feito através do envio e recebimento de mensagens assíncronas entre a parte do app que é está em Flutter (*i.e.*, UI) e a parte nativa (*i.e.*, Android ou iOS). Assim, a UI envia uma mensagem para o "*host*" (*i.e.*,parte nativa do código) através de um canal da plataforma. Por sua vez, o *host* escuta o canal, recebe a mensagem e chama todas as APIs necessárias (utilizando a linguagem nativa) para realizar tal tarefa. Por fim, o *host* envia uma resposta para o "*client*" (*i.e.*, a parte em Flutter) (Google, 2018g). Todo esse processo é mostrado na figura 9.

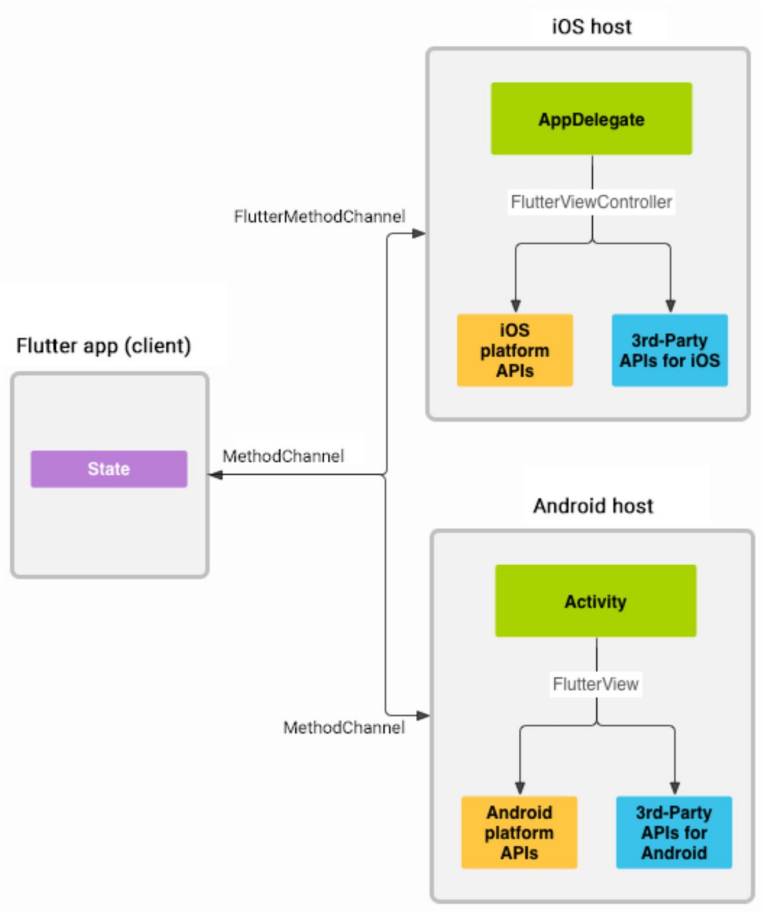

Figura 9: Arquitetura do framework Flutter para dar suporte a códigos nativas (Google, 2018c).

3.2.3. Ciclo de Vida

O ciclo de vida de um widget é composto por poucos estados. Um dos primeiros estados a serem chamados é o "*initState*". Este é fortemente utilizado para inicializar *listeners* e executar tarefas que precisam ser realizadas apenas uma vez (*e.g.*, configurar animações ou ajustar dados que serão apresentados na tela decorrente). No método "*build*" o widget é delineado. Assim, este estado conterá toda a hierarquia de widgets que formará a tela. Por fim, quando um estado vai ser deixado de ser utilizado, o método "*dispose*" é chamado e pode ser utilizado para remover os *listeners* adicionados no *initState*. A sequência dos estados é descrita na figura 10.

Outro ponto importante do ciclo de vida de um widget é o método "*setState*". Semelhante ao "*setState()*" de React, este é responsável por atualizar o estado do widget. Uma vez que o estado é mudado pela chamada deste método, o widget vai passar pelo processo de renderização novamente. Porém, desta vez, de acordo com o novo estado da aplicação.

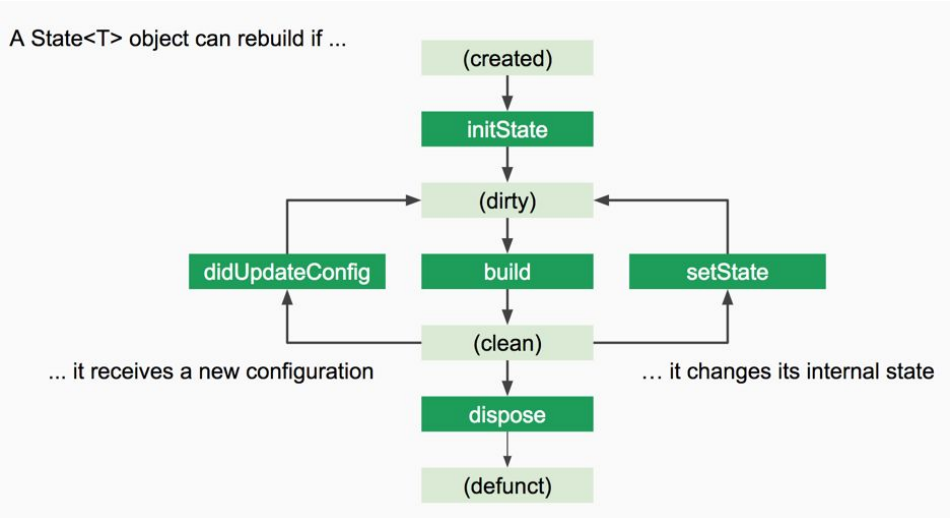

Figura 10: Ciclo de vida de widgets do framework Flutter (Google, 2018f).

4. Estudo de Casos

Com o intuito de comparar a performance da abordagem nativa em relação ao desenvolvimento utilizando os frameworks React Native ou Flutter, foram desenvolvidas duas aplicações principais. O primeiro aplicativo (o qual é contemplado na seção 4.1) tem como objetivo simular o estressamento do processador do dispositivo móvel utilizando operações que não envolvam a UI (*i.e.*, que não façam modificações na interface do usuário). A forma aqui utilizada para alcançar esse propósito foi buscando os números primos em um intervalo dado. Já o segundo aplicativo (explicado na seção 4.2) tem o objetivo de verificar o tempo gasto para se construir a interface de uma tela por completa (*i.e.*, verificar o tempo utilizado em operações puramente de UI).

Cada uma das aplicações descritas anteriormente foi desenvolvida de três modos diferentes: (a) utilizando Swift 4.1 em uma abordagem nativa para iOS, (b) utilizando o framework React Native e (c) utilizando o framework Flutter. Desse modo, foram implementados no total seis aplicativos distintos (*i.e.*, três contemplando um experimento e três para outro). Além disso, é importante ressaltar que o foco dos experimentos não é de ressaltar qual a melhor abordagem nem comparar a interface criada por cada uma. Os experimentos aqui descritos visam mostrar cenários em que uma abordagem pode ser mais vantajosa ou não de se utilizar do que outra abordagem aqui descrita.

4.1. Experimento 1: Cálculo de Números Primos

4.1.1. Implementação

Este aplicativo tem uma única tela com um input (*i.e.*, TextField) e um botão para calcular os números primos entre 2 e N (sendo N o valor dado como entrada). Desse modo, o usuário digita um número e aperta o botão “calcular”. Ao ativar o botão, um contador de tempo é inicializado. Em seguida, um algoritmo busca os números primos entre 2 - N. Ao encontrar todos os primos, o contador finaliza o tempo e mostra no log da aplicação o tempo percorrido. Vale-se frisar que o foco do experimento não é utilizar o algoritmo mais eficiente para se calcular os números primos. Assim, o algoritmo utilizado vai percorrer, para cada valor n (sendo n um valor entre 2 - N), um loop de 2 até raiz quadrada de n e verificar se existem algum número que divide n. Em caso positivo, o loop é finalizado, n não é

considerado primo e um novo n' é verificado. Todo esse processo vai ocorrer até o valor N digitado pelo usuário ser analisado. Para maior detalhe de como o algoritmo funciona, ver figura 11.

```
1  FindPrimeNumbers(n) {
2      let primeNumbers = [];
3      let isPrime = true;
4
5      for (let i=2; i <=n; i++) {
6          isPrime = true;
7          for (let j = 2; j < Math.ceil(Math.sqrt(i)); j++) {
8              if (i % j === 0) {
9                  isPrime = false;
10                 break;
11             }
12         }
13
14         if (isPrime) {
15             primeNumbers.push(i);
16         }
17     }
18 }
```

Figura 11: Algoritmo de busca de números primos.

4.1.2. Resultados

Para esse experimento foi utilizada a entrada 99999. Assim, o experimento foi executado 30 vezes tanto em um simulador de iOS (utilizando o XCode) quanto em um iPhone 6 (atualizado para a versão iOS 11.4). Após cada execução, foi acrescentado em uma tabela o tempo gasto para realizar os cálculos. Dessa tabela foram criados os gráficos da figura 12 com os resultados da simulação.

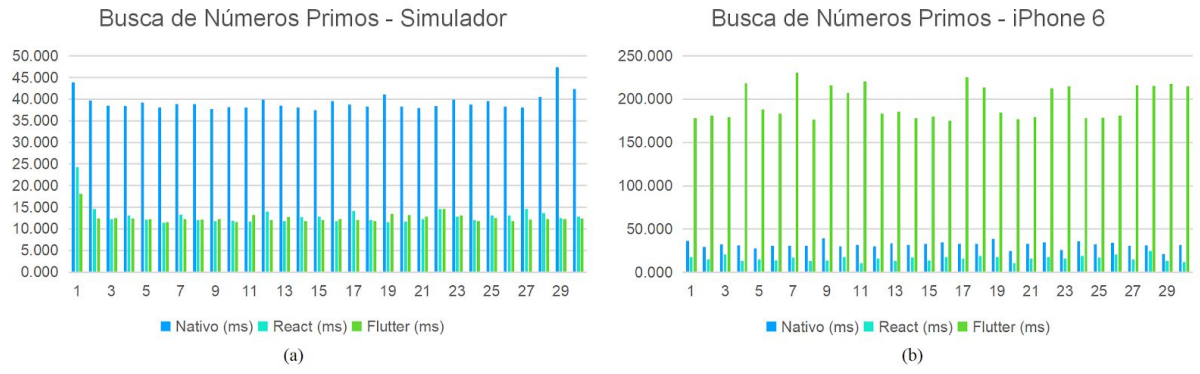

Figura 12: Tempo para encontrar os números primos no intervalo 2 - N utilizando Swift, React Native e Flutter. (a) Resultados obtidos por simulador. (b) Resultados obtidos em iPhone 6.

Na execução do experimento no simulador (figura 12a) pode-se observar que o tempo de execução do código utilizando React Native e Flutter foram bem similares. De fato, o tempo médio para calcular todos os números primos com React Native e Flutter foi respectivamente de 13,093 $\pm$ 2,32 ms e 12,592 $\pm$ 1,209 ms. Ademais, para um fator de confiança de 95%, o uso respectivos dos dois frameworks teve um intervalo de confiança de 12,227ms - 13,974ms e 12,141ms - 13,051ms. Já na abordagem nativa obteve-se a média, desvio padrão e intervalo de confiança (95%) respectivamente de 39,371ms, 2,079ms e 38,595ms - 40,161ms. Com essa execução observa-se que Swift tende a ser uma linguagem mais devagar do que JavaScript em relação a atividades que não envolvam UI. Contudo, o mesmo não é necessariamente correto em relação a Swift e Dart. De fato, ao observar a realização do experimento no iPhone 6 (figura 12b), nota-se que Dart (o qual tinha demonstrado o melhor desempenho no simulador) tornou-se o código menos performático com uma média de execução de 196,312 $\pm$ 19,061 ms. Já as abordagens nativa e utilizando o framework React Native obtiveram a média de execução de 31,782 $\pm$ 3,7 ms e 16,1 $\pm$ 3,155 ms (valores esses semelhantes ao encontrado pelo simulador). De acordo com a documentação oficial do Flutter, essa diferença de performance nos dois ambientes acontece porque algumas operações são mais rápidas em simuladores do que em dispositivos reais enquanto outras são mais devagar. Assim, o desempenho da aplicação executando em modo debug ou em simuladores não é um indicador do comportamento da mesma em um dispositivo real em modo de produção (Google, 2018d).

Tabela 1: Resultados do cálculo de números primos entre 2 - 99999 no Simulador e iPhone 6.

	Simulador			iPhone 6		
	Nativo(ms)	**React(ms)**	**Flutter(ms)**	**Nativo (ms)**	**React (ms)**	**Flutter (ms)**
Média	39.371	13.093	12.592	31.782	16.100	196.312
Desvio Padrão	2.079	2.320	1.209	3.700	3.155	19.061
Intervalo de Confiança: Inferior (95%)	38.582	12.212	12.133	30.377	14.902	189.074
Intervalo de Confiança: Superior (95%)	40.161	13.974	13.051	33.187	17.298	203.550

Ao utilizar o "*Instruments*" do Xcode, pode-se observar que o código de React Native gerava várias threads para executar o cálculo dos números primos. Já a abordagem nativa praticamente só utilizou a main thread.

Os demais resultados da execução do experimento 1 estão descritos na tabela 1.

4.2. Experimento 2: Lista de Contatos

4.2.1. Implementação

Esta aplicação terá uma única tela com uma lista de contatos informando nome e telefone. A lista será construída lendo um array estático (com 50 registros) o qual foi previamente produzido utilizando uma API open source que gera usuários aleatórios (Armstrong & Hunt, 2018). Ao abrir a aplicação um contador vai ser inicializado. Este vai ser finalizado quando a lista de usuários for montada por completo na tela. Por fim, a duração desta operação vai ser informada no log da aplicação.

Vale-se frisar que esse experimento tem o foco de observar o tempo utilizado para fazer operações cotidianas puramente de UI. Assim, não houveram preocupações de se gerar interfaces idênticas. Além disso, foram utilizados, em cada implementação, os componentes mais apropriados para renderizar a lista de forma ótima, ou seja, mostrar apenas os itens que estarão visíveis na tela (evitando que a aplicação utilize tempo extra para construir partes da UI a qual nem vão estar sendo visualizada). Desse modo, foram utilizados o TableView, FlatList e ListView.builder respectivamente para as implementações iOS nativo (Swift), React Native e Flutter.

Para facilitar a identificação da renderização da lista, foi utilizado o plugin "*after_layout: 1.0.2*" (Lightfoot, 2018) na implementação com Flutter. Este acrescenta o método "*afterFirstLayout()*" que é chamado após o widget ser apresentado na tela.

4.2.2. Resultados

O experimento foi executado 30 vezes por cada abordagem em um iPhone 6 (atualizado para a versão 11.4). Este teste não foi realizado no simulador pois o mesmo não mostrou exatidão no experimento anterior nem é aconselhado de ser usado como meio de medir performance (como explicado na seção 4.1.2). Por fim, a duração para criar a lista de

contatos de cada execução foi acrescentada em uma tabela a qual gerou os gráficos da figura 13.

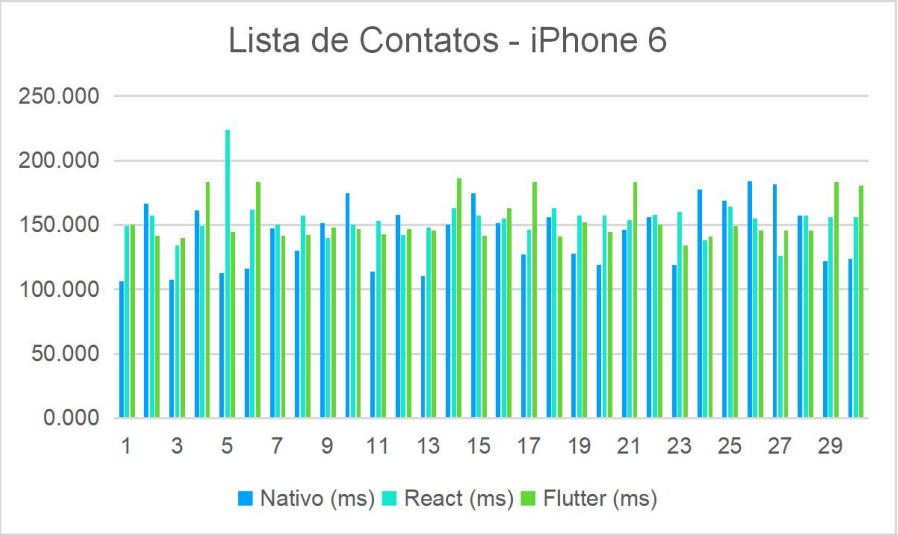

Figura 13: Tempo para renderizar uma lista de contatos na tela de um iPhone 6.

Diferente do experimento 1 com o cálculo dos números primos, observou-se que os três casos em estudo se assemelham quanto a processamento exclusivamente na UI. De fato, as médias das abordagens nativa, com React Native e Flutter foram respectivamente 143.187 ms, 154.567 ms e 154.169 ms. Isto também comprova que os elementos TableView, FlatList e ListView.build atuam de mesmo modo na construção de uma lista. Em especial, React Native surpreende com esse resultado pois conceitualmente ele pode aparentar ser mais devagar pelo uso de pontes para a comunicação entre o núcleo nativo e o ambiente JavaScript. Contudo, ele se mostrou equivalente no manuseio da interface.

Os demais cálculos realizados neste experimento estão descritos na tabela 2.

Tabela 2: Resultado de cálculos no tempo para renderizar lista de contatos.

	Nativo (ms)	**React (ms)**	**Flutter (ms)**
Média	143.187	154.567	154.169
Desvio Padrão	24.708	15.861	16.945
Intervalo de Confiança (95%): Limite Inferior	133.805	148.544	147.734
Intervalo de Confiança (95%): Limite	152.570	160.590	160.604

Superior			

5. Conclusão

5.1. Conclusões

O objetivo deste trabalho foi fazer um estudo e comparação do desenvolvimento de aplicações para iOS utilizando uma de três abordagens distintas: nativo (com Swift), React Native e Flutter. Assim, para cada uma das abordagens, descreveu-se sua arquitetura, ciclo de vida de componentes e principais características. Em seguida, dois experimentos entre as implementações foram realizados com o intuito de comparar as performances das mesmas em situações distintas.

O primeiro teste foi realizado com o foco verificar o desempenho usando apenas operações que não envolvessem UI (*i.e.*, que não modificasse os componentes descritos na tela). Assim, React Native demonstrou ser mais performático por calcular os números primos em menor intervalo de tempo em um dispositivo (sendo a abordagem nativa a segunda mais rápida com quase o dobro de tempo gasto pelo React). Já o framework Flutter surpreendeu com dois resultados contrastantes. O primeiro resultado, encontrado executando a aplicação em um simulador, foi o de melhor tempo de processamento em relação às três abordagens. Contudo, quando o mesmo experimento foi realizado em um dispositivo móvel, Flutter obteve o pior resultado por uma diferença maior que 10x em relação ao React Native. Desse modo, após busca cuidadosa na documentação do framework, observou-se que o mesmo pode ser mais ou menos performático em operações específicas dependendo de onde ele estiver executando. Por isso, é aconselhado que os testes com Flutter sempre sejam realizados no dispositivo com menor capacidade de processamento que vai dar suporte a aplicação.

O segundo teste foi realizado com o foco de analisar situações contrárias ao primeiro experimento (*i.e.*, operações que envolvam primordialmente a mudança na interface do usuário). Assim, apesar dos resultados terem sido bem próximos entre as três implementações, a abordagem nativa demonstrou ser mais rápida em manusear a UI.

Com esses experimentos pode-se constatar que o desenvolvimento multiplataforma oferecido pelos frameworks React Native e Flutter podem ser de grande ajuda em empresas que almejam um desenvolvimento mais rápido de seus produtos. Contudo, ainda é necessário

analisar quais as principais funcionalidades do app para se optar qual abordagem usar. De fato, caso o aplicativo utilize pedaços muito específicos do hardware de uma plataforma, é provável que a melhor abordagem seja a nativa. Já se o app precisar alcançar um público maior (*i.e.*, que utiliza ambas plataformas Android e iOS), React Native é uma ótima escolha pois o mesmo já está no mercado a alguns anos e possui um excelente suporte da comunidade de desenvolvedores. Contudo, se a aplicação não precisar utilizar muitas funcionalidades específicas e precisar dar suporte a mais de uma plataforma, Flutter é uma boa opção.

5.2. Trabalhos Futuros

Em estudos futuros pode-se investigar as razões para a performance no simulador e no dispositivo serem tão diferentes ao utilizar Flutter. Desse modo, poderão ser apresentados mais cenários que mostram essa desigualdade e meios para evitá-los. Outro ponto que pode ser explorado é a análise e comparação de outros componentes não estudados aqui de cada abordagem. Também, pode-se fazer o mesmo estudo aqui descrito porém com o foco na plataforma Android. Ademais, aplicações mais complexas (*e.g.,* que utilizem banco de dados, façam requisições http e tenham concorrência) podem ser estudadas para comparar componentes e suas performances na mesma.

6. Referências

Apple. (2018a). Mac Technology Overview - Core OS Layer. Visualizado em 20 de junho, 2018, em https://developer.apple.com/library/archive/documentation/MacOSX/Conceptual/OSX_Technology_Overview/CoreOSLayer/CoreOSLayer.html#//apple_ref/doc/uid/TP40001067-CH9-SW1

Apple. (2018b). Mac Technology Overview - Mac Architecture. Visualizado em 20 de junho, 2018,em https://developer.apple.com/library/archive/documentation/MacOSX/Conceptual/OSX_Technology_Overview/About/About.html#//apple_ref/doc/uid/TP40001067-CH204-TPXREF101

Apple. (2018c). The App Life Cycle. Visualizado em 20 de junho, 2018, em https://developer.apple.com/library/archive/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/TheAppLifeCycle/TheAppLifeCycle.html#//apple_ref/doc/uid/TP40007072-CH2-SW1

Apple. (2018d). UIViewController Lifecycle. Apple. Visualizado em https://developer.apple.com/documentation/uikit/uiviewcontroller

Armstrong, K., & Hunt, A. J. (2018). Random User Generator. Visualizado em 20 de junho, 2018, em https://randomuser.me/

Casé, V. C. (2015). *EM DIREÇÃO A UMA ABORDAGEM PARA O DESENVOLVIMENTO DE APLICATIVOS MÓVEIS MULTIPLATAFORMA* (Bacharelado). (V. C. Garcia, Ed.). Universidade Federal de Pernambuco.

Facebook. (2018a). Components and Props. Visualizado em 16 de junho, 2018, em https://reactjs.org/docs/components-and-props.html

Facebook. (2018b). React Component Lifecycle. Visualizado em 16 de junho, 2018, em https://reactjs.org/docs/react-component.html

Facebook. (2018c). React Native - Hot Reloading. Visualizado em 16 de junho, 2018, em http://facebook.github.io/react-native/blog/2016/03/24/introducing-hot-reloading.html

Facebook. (2018d). React Native Official Webpage. Visualizado em 16 de junho, 2018, em https://facebook.github.io/react-native/

Gondi, T. (2015, January 14). iOS Architecture and Common Elements. Visualizado em https://tilakgondi.wordpress.com/2015/01/14/ios-architecture/

Google. (2018a). Dart. Visualizado em 22 de junho, 2018, em, from https://www.dartlang.org/

Google. (2018b). Flutter API Documentation. Visualizado em 22 de junho, 2018, em https://docs.flutter.io/

Google. (2018c). Flutter Architectural overview - platform channels. Google. Visualizado em https://flutter.io/platform-channels/#architecture

Google. (2018d). Flutter - Performance in a Physical Device. Visualizado em 19 de junho, 2018, em https://flutter.io/ui-performance/#connect-to-a-physical-device

Google. (2018e). Flutter Technical Overview. Visualizado em 22 de junho, 2018, em https://flutter.io/technical-overview/

Google. (2018f). Flutter Widget Lifecycle. Google. Visualizado em https://docs.google.com/presentation/d/1cw7A4HbvM_Abv320rVgPVGiUP2msVs7tfGbkgdrTy0I/edit#slide=id.g70d668005_2_22

Google. (2018g). Flutter - Writing custom platform-specific code with platform channels. Visualizado em 22 de junho, 2018, em https://flutter.io/platform-channels/#architecture

Google. (2018h). Material Design. Visualizado em 22 de junho, 2018, em https://material.io/

Hansson, N., & Vidhall, T. (2016, June 16). *Effects on performance and usability for cross-platform application development using React Native*. (A. Froberg, Ed.). Linköpings Universitet. Visualizado em http://www.diva-portal.org/smash/get/diva2:946127/FULLTEXT01.pdf

Hasan, M. M., & Haque, M. A. (2016). *Mobile Application Development Approaches: Recommendation for E-commerce Enterprises* (Bachelor). Laurea University of Applied Sciences. Visualizado em https://www.theseus.fi/handle/10024/120281

Heard, P. (2017a). React Native Architecture. Visualizado em

https://www.logicroom.co/react-native-architecture-explained/

Heard, P. (2017b, June 13). React Native Architecture : Explained! Visualizado em 16 de junho, 2018, em https://www.logicroom.co/react-native-architecture-explained/

JavaSampleApproach. (2018, April 3). React Component Lifecycle. JavaSampleApproach. Visualizado em http://javasampleapproach.com/frontend/react/react-component-lifecycle-methods-from-v16-3-react-lifecycle-example

Joorabchi, M. E., Mesbah, A., & Kruchten, P. (2013). Real Challenges in Mobile App Development. Em *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*. https://doi.org/10.1109/esem.2013.9

Kirichok, S. (n.d.). Is Swift Faster Than Objective-C? Visualizado em 20 de junho, 2018, em https://yalantis.com/blog/is-swift-faster-than-objective-c/

Lightfoot, S. (2018). After Layout. Visualizado em 20 de junho, 2018, em https://pub.dartlang.org/packages/after_layout

Oliveira, W., Oliveira, R., & Castor, F. (2017). A Study on the Energy Consumption of Android App Development Approaches. Em *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. https://doi.org/10.1109/msr.2017.66

React Native · A framework for building native apps using React. (n.d.). Visualizado em 6 de junho, 2018, em https://facebook.github.io/react-native/index.html

Tschinder, D., Ng, B., Smyth, L., Sauleau, S., & Zhu, H. (2018). Babel. Visualizado em 18 de junho, 2018, em http://babeljs.io/

Willocx, M., Vossaert, J., & Naessens, V. (2016). Comparing performance parameters of mobile app development strategies. Em *Proceedings of the International Workshop on Mobile Software Engineering and Systems - MOBILESoft '16*. https://doi.org/10.1145/2897073.2897092