UNIVERSIDADE FEDERAL DO RIO DE JANEIRO

INSTITUTO DE COMPUTAÇÃO

CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

BRENO COLL DE FREITAS

FLUTTER E REACT NATIVE: uma análise comparativa entre dois frameworks de desenvolvimento mobile multiplataforma

RIO DE JANEIRO

2022

BRENO COLL DE FREITAS

FLUTTER E REACT NATIVE: uma análise comparativa entre dois frameworks de desenvolvimento mobile multiplataforma

Trabalho de conclusão de curso de graduação apresentado ao Instituto de Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação. (justificado sem recuo, início no meio da página).

Orientador: Profa. Silvana Rossetto

RIO DE JANEIRO

2022

CIP - Catalogação na Publicação

F866f

Freitas, Breno Coll de
FLUTTER E REACT NATIVE: uma análise comparativa entre dois frameworks de desenvolvimento mobile multiplataforma / Breno Coll de Freitas. -- Rio de Janeiro, 2022.
68 f.

Orientadora: Silvana Rossetto.
Trabalho de conclusão de curso (graduação) - Universidade Federal do Rio de Janeiro, Instituto de Computação, Bacharel em Ciência da Computação, 2022.

1. Desenvolvimento mobile. 2. Ambientes multiplataforma. 3. Flutter. 4. React Native. I. Rossetto, Silvana, orient. II. Título.

Elaborado pelo Sistema de Geração Automática da UFRJ com os dados fornecidos pelo(a) autor(a), sob a responsabilidade de Miguel Romeu Amorim Neto - CRB-7/6283.

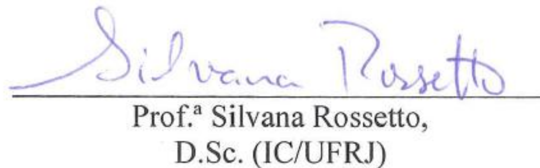
BRENO COLL DE FREITAS

FLUTTER E REACT NATIVE: uma análise comparativa entre dois frameworks de desenvolvimento mobile multiplataforma

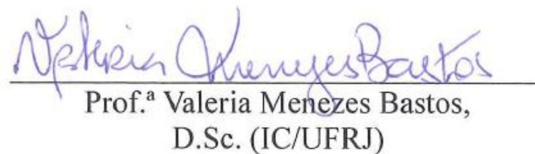
Trabalho de conclusão de curso de graduação apresentado ao Instituto de Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Aprovado em 07 de novembro de 2022.

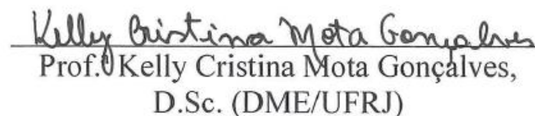
BANCA EXAMINADORA:



Prof.ª Silvana Rossetto,
D.Sc. (IC/UFRJ)



Prof.ª Valeria Menezes Bastos,
D.Sc. (IC/UFRJ)



Prof. Kelly Cristina Mota Gonçalves,
D.Sc. (DME/UFRJ)

AGRADECIMENTOS

Agradeço primeiramente à minha orientadora, professora Silvana Rossetto, por toda a ajuda concedida durante o projeto, desde a escolha do tema aos ajustes finais. Também agradeço pelo apoio durante toda a minha trajetória acadêmica. Agradeço, também, à professora Kelly Cristina, principalmente, pelas aulas de probabilidade e estatística lecionadas, mas também pelos conselhos nos ajustes finais desta monografia. Por fim, agradeço a todos os meus amigos e familiares que estiveram comigo durante todo este trajeto, apoiando e incentivando de diversas formas.

“You may find that the documentation is leaving you stranded. Certain operations aren’t accurately described; misprints lead you into mistakes even when you follow the manual exactly; there’s no section that describes what to do in various error situations, and the error messages you get don’t make any sense; or perhaps you don’t understand some of the terms being used. There is an answer to problems of this type. It may require calling the computer dealer, or even the manufacturer of the computer or software to get an explanation.”

C. Rubin

"Some People Should Be Afraid of Computers"

Personal Computing

vol.7 n.8 pp.55–57,163. ago.1983

RESUMO

O mercado de dispositivos móveis, atualmente dominado pelos sistemas operacionais Android e iOS, está em constante crescimento, tornando essa tecnologia cada vez mais acessível. Cada sistema operacional possui sua própria forma de gerenciar os recursos do dispositivo e de exibir suas funcionalidades para os desenvolvedores de aplicações. Com a evolução deste mercado, criou-se a necessidade de diminuir os custos de desenvolvimento e manutenção de aplicativos móveis. Diferentes ferramentas que permitem o desenvolvimento de aplicativos multiplataformas foram criadas, possibilitando a distribuição de um aplicativo para os dois sistemas operacionais a partir de um único código fonte. Atualmente, Flutter e React Native são os dois frameworks de desenvolvimento multiplataforma mais consolidados no mercado, sendo utilizados por mais de 35% dos desenvolvedores. Neste trabalho, comparamos o desempenho desses dois frameworks por meio de um estudo empírico. Desenvolvemos duas versões, uma para cada framework, de cinco aplicativos diferentes, abrangendo funcionalidades como interação com o usuário, navegação entre telas e rolagem de uma lista. Utilizamos a taxa de *frames* renderizados por segundo como métrica principal para comparar o desempenho dos aplicativos durante um experimento, simulando interações com o usuário. Também efetuamos um experimento adicional comparando o desempenho dos frameworks ao lidar com listas, utilizando a quantidade de espaço em branco apresentado durante uma rolagem como métrica de comparação. Concluímos que, apesar de ambos os frameworks apresentarem resultados similares nos demais aplicativos, o React Native possui certa deficiência ao lidar com listas, apresentando uma alta quantidade de espaço em branco e baixa taxa de *frames* renderizados por segundo quando esse recurso é utilizado.

Palavras-chave: desenvolvimento mobile; ambientes multiplataforma; Flutter; React Native.

ABSTRACT

The mobile market, currently dominated by Android and iOS operational systems, is in constant growth, making this technology increasingly accessible. Each operational system has its own way to manage the device's resources and expose its features to the developer. Due to this market's growth, there is a need to reduce the mobile applications' development and maintenance cost. Different tools that allow for cross-platform development were created, allowing an application to be distributed to both operational systems with a single source code. Currently, Flutter and React Native are the two most consolidated cross-platform development frameworks, being used by more than 35% of developers. In this work, we compared both frameworks' performance through an empirical study. We developed two versions, one for each framework, of five different applications, including features like user interaction, screen navigation and list scrolling. We used the frame rate as the main metric to compare applications' performance during an experiment, simulating an user interaction. We also have performed an additional experiment comparing the frameworks' performance while dealing with lists, using the amount of blank space visible during a scroll as a comparison metric. We concluded that, besides both frameworks presenting similar results for the other apps, React Native has difficulties to deal with lists, presenting a high amount of blank space and a low frame rate when this feature is used.

Keywords: mobile development; cross-platform environment; Flutter; React Native.

LISTA DE ILUSTRAÇÕES

Figura 1 – Porcentagem da utilização de dispositivos por usuários para acessar páginas web no mundo ao longo dos últimos 12 anos 13
Figura 2 – Porcentagem da utilização de frameworks mobile por desenvolvedores no mundo ao longo dos últimos 3 anos 14
Figura 3 – Camadas da arquitetura do Flutter 24
Figura 4 – Árvores criadas durante o processo de renderização no Flutter 29
Figura 5 – Antiga arquitetura do React Native 32
Figura 6 – Nova arquitetura do React Native 34
Figura 7 – Árvores criadas durante o processo de renderização no React Native 36
Figura 8 – Árvores criadas durante o processo de atualização no React Native 37
Figura 9 – Árvores criadas após o processo de atualização no React Native 39
Figura 10 – Árvore de componentes do aplicativo 1 41
Figura 11 – Árvore de componentes do aplicativo 2 42
Figura 12 – Árvore de componentes do aplicativo 3 42
Figura 13 – Árvore de componentes do aplicativo 4 43
Figura 14 – Árvore de componentes do aplicativo 5 44
Figura 15 – Visualização da ferramenta Dart Devtools 49
Figura 16 – Tamanho de um item e da lista em Flutter 52
Figura 17 – Tamanho de um item e da lista em React Native 52
Figura 18 – Porcentagem de *janky frames* por execução do aplicativo 1 55
Figura 19 – Porcentagem de *janky frames* por execução do aplicativo 2 56
Figura 20 – Porcentagem de *janky frames* por execução do aplicativo 3 57
Figura 21 – Porcentagem de *janky frames* por execução do aplicativo 4 58
Figura 22 – Porcentagem de *janky frames* por execução do aplicativo 5 60
Figura 23 – Porcentagem de espaço em branco por frame do aplicativo 5 em Flutter 61
Figura 24 – Porcentagem de espaço em branco por frame do aplicativo 5 em React Native 62

LISTA DE TABELAS

Tabela 1 – Especificidades de desenvolvimento para as plataformas Android e iOS 18

Tabela 2 – Tamanho de um item e da lista nos frameworks 52

Tabela 3 – Quantidade de *janky frames* no aplicativo 1 55

Tabela 4 – Quantidade de *janky frames* no aplicativo 2 56

Tabela 5 – Quantidade de *janky frames* no aplicativo 3 57

Tabela 6 – Quantidade de *janky frames* no aplicativo 4 58

Tabela 7 – Quantidade de *janky frames* no aplicativo 5 59

Tabela 8 – Quantidade de espaço em branco na lista do aplicativo 5 em Flutter 61

Tabela 9 – Quantidade de espaço em branco na lista do aplicativo 5 em React Native 63

LISTA DE CÓDIGOS

Código 1 – Renderização de um Flutter Widget 28
Código 2 – Renderização de um React Element 35
Código 3 – Atualização de um React Element 37

LISTA DE SIGLAS

ADB – Android Debug Bridge

JSI – JavaScript Interface

JNI – Java Native Interface

JSON – JavaScript Object Notation

HTML – HyperText Markup Language

CSS – Cascading Style Sheets

SUMÁRIO

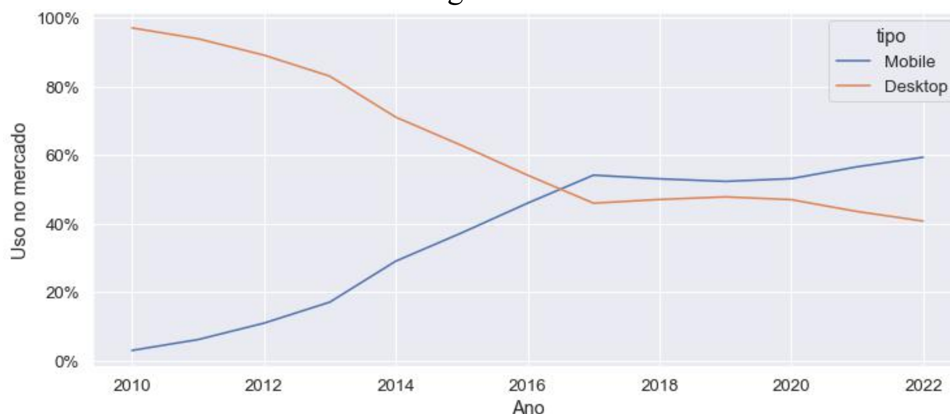
1 INTRODUÇÃO 13
1.1 MOTIVAÇÃO 13
1.2 OBJETIVO 14
1.3 METODOLOGIA 14
1.4 ORGANIZAÇÃO DO TEXTO 15
2 TRABALHOS RELACIONADOS 16
3 DESENVOLVIMENTO MOBILE 18
3.1 DESENVOLVIMENTO NATIVO 18
3.2 DESENVOLVIMENTO MULTIPLATAFORMA 19
3.2.1 Desenvolvimento Web 19
3.2.2 Desenvolvimento Híbrido 20
3.2.3 Desenvolvimento Interpretado 21
3.2.4 Desenvolvimento Cross-Compiled 22
4 FLUTTER 23
4.1 MODELO DE THREADS 23
4.2 ARQUITETURA LÓGICA 24
4.2.1 Framework 25
4.2.2 Engine 26
4.2.3 Embedder 26
4.3 PROCESSO DE RENDERIZAÇÃO 27
4.3.1 Renderização Inicial 28
4.3.2 Renderização de atualizações 29
5 REACT NATIVE 31
5.1 MODELO DE THREADS 31
5.2 ARQUITETURA 32
5.2.1 Antiga Arquitetura 32
5.2.2 Nova Arquitetura 33
5.3 PROCESSO DE RENDERIZAÇÃO 34
5.3.1 Renderização Inicial 35
5.3.2 Renderização de atualizações 37
6 ESTUDOS DE CASO 40
6.1 APLICATIVOS DESENVOLVIDOS 40
6.1.1 Aplicativo 1 - Stopwatch 41
6.1.2 Aplicativo 2 - Multi Stopwatch 41
6.1.3 Aplicativo 3 - Counter 42
6.1.4 Aplicativo 4 - Navigations 43
6.1.5 Aplicativo 5 - List 43

6.2 MINIMIZANDO INTERFERÊNCIAS EXTERNAS 44
6.3 COLETANDO A QUANTIDADE DE JANKY FRAMES 45
6.3.1 React Native 46
6.3.2 Flutter 47
6.4 ANALISANDO ESPAÇO EM BRANCO NAS LISTAS 49
7 RESULTADOS 54
7.1 QUANTIDADE DE JANKY FRAMES 54
7.1.1 Aplicativo 1 - Stopwatch 54
7.1.2 Aplicativo 2 - Multi Stopwatch 55
7.1.3 Aplicativo 3 - Counter 56
7.1.4 Aplicativo 4 - Navigations 58
7.1.5 Aplicativo 5 - List 59
7.2 ESPAÇO EM BRANCO NAS LISTAS 60
7.2.1 Flutter 60
7.2.2 React Native 62
8 CONCLUSÃO 64
8.1 TRABALHOS FUTUROS 65
REFERÊNCIAS 66

1 INTRODUÇÃO

Em constante crescimento desde 2010, dispositivos móveis são cada vez mais utilizados pelos usuários para acessar páginas da Web em todo o mundo. Em 2017, os dispositivos móveis passaram a ser responsáveis pela maior parte dos acessos, deixando os computadores para trás (STATCOUNTER, 2022). Hoje, os dispositivos móveis já representam quase 60% do uso total, como mostra a figura 1.

Figura 1 – Porcentagem da utilização de dispositivos por usuários para acessar páginas web no mundo ao longo dos últimos 12 anos



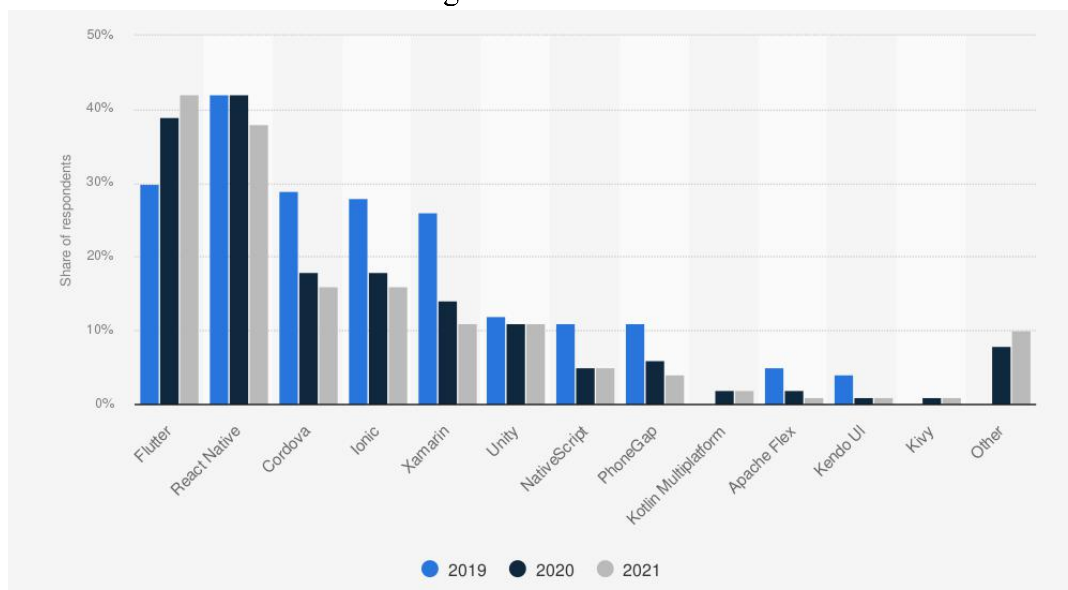
Com os dispositivos móveis sendo utilizados cada vez mais, aumenta-se o mercado de produtos desenvolvidos para este tipo de dispositivo. Ao longo dos últimos anos, várias ferramentas foram desenvolvidas visando minimizar os custos de desenvolvimento de um aplicativo *mobile*. Atualmente, Flutter e React Native são os dois frameworks mais utilizados (JETBRAINS, 2021). A porcentagem de utilização dos demais frameworks pode ser vista na figura 2.

1.1 MOTIVAÇÃO

Devido ao crescimento do Flutter e React Native, é possível encontrar diversos estudos sobre os dois frameworks. Wu (2018) e Jagiello (2019) desenvolveram trabalhos comparando o desempenho dos frameworks através de um estudo empírico. Com a constante atualização desses frameworks, este estudo tem como motivação apresentar uma comparação atualizada. Além disso, algumas questões levantadas pelos referidos autores e não tratadas em seus trabalhos, também serão exploradas aqui, a fim de complementar informações apresentadas em seus respectivos trabalhos.

Os dois frameworks possuem um objetivo semelhante: permitir o desenvolvimento de aplicativos móveis multiplataforma a partir de um único código fonte. Antes de iniciar o desenvolvimento de um aplicativo multiplataforma, o desenvolvedor deve optar por qual framework utilizar. Além de fatores como familiaridade com a linguagem e facilidade de aprendizagem do framework, o seu desempenho é um ponto bastante importante na escolha (SATROM, 2018). Dessa forma, a principal motivação deste trabalho é oferecer um estudo que possa auxiliar na escolha do framework, levantando pontos positivos e negativos de ambos a partir dos resultados obtidos por meio de um estudo empírico.

Figura 2 – Porcentagem da utilização de frameworks mobile por desenvolvedores no mundo ao longo dos últimos 3 anos



Fonte: (STATISTA, 2021)

1.2 OBJETIVO

O objetivo deste trabalho é comparar, em questão de desempenho, os frameworks Flutter e React Native, bastantes consolidados. Para isso, comparamos, de forma empírica, o desempenho de aplicativos similares desenvolvidos a partir dos dois frameworks.

1.3 METODOLOGIA

Para comparar o desempenho dos frameworks, foram desenvolvidas duas versões de cinco aplicativos diferentes, uma em Flutter e outra em React Native. Os aplicativos são simples e possuem foco em apenas uma funcionalidade particular. Assim, foi possível

comparar o desempenho dos frameworks a partir da comparação do desempenho apresentado pelas versões de cada um dos aplicativos.

Inspirado nos trabalhos de Hansson e Vidal (2016), Wu (2018) e Jagiello (2019), a principal métrica utilizada foi a taxa de FPS (*frames* por segundo). Além disso, a fim de aprimorar o estudo feito por Wu sobre o *scroll* nas listas, calculamos a quantidade de itens não renderizados durante uma execução utilizando uma análise automatizada dos frames renderizados. Os aplicativos foram executados de forma automatizada, seguindo um roteiro de interação com o usuário definido para cada aplicativo, e as métricas descritas foram coletadas a partir de *scripts* escritos em *Python*.

Os dois frameworks são capazes de gerar aplicativos para as plataformas Android e iOS. Para executar os aplicativos no iOS, porém, é necessário utilizar um macOS, o que nos impossibilitou de comparar os aplicativos nesta plataforma. Sendo assim, apenas a plataforma Android será levada em consideração neste trabalho. Como a comunicação do framework com o dispositivo é efetuada de forma diferente em cada uma das plataformas, não é possível deduzir quais seriam os resultados obtidos no iOS.

1.4 ORGANIZAÇÃO DO TEXTO

O restante deste texto está organizado da seguinte forma. No capítulo 2, encontram-se os trabalhos relacionados. Nos capítulos 3, 4 e 5, detalha-se a parte teórica do trabalho. O capítulo 3 apresenta as diferentes abordagens de desenvolvimento mobile. Os capítulos 4 e 5 descrevem o funcionamento interno dos dois frameworks que são o assunto principal do trabalho. Destaca-se a forma com que os frameworks se comunicam com o dispositivo, além de qual abordagem eles seguem no desenvolvimento multiplataforma. O capítulo 6 contém informações sobre o desenvolvimento dos aplicativos utilizados para efetuar a comparação entre os frameworks. Descreve-se as métricas utilizadas, assim como a forma com que foram extraídas durante a execução dos aplicativos. No capítulo 7, encontram-se os resultados dos experimentos executados nos aplicativos descritos anteriormente. Por fim, o capítulo 8 apresenta a conclusão deste trabalho.

2 TRABALHOS RELACIONADOS

Neste capítulo elencamos alguns trabalhos relacionados que foram levados em consideração na elaboração deste trabalho.

Hansson e Vidhall (2016) fizeram um estudo sobre o desempenho e usabilidade de um aplicativo desenvolvido utilizando o React Native, na versão 0.20. Como o framework estava em suas versões iniciais, não haviam outros estudos focados em seu desempenho. O estudo foi, então, conduzido pela Attentec, uma empresa sueca de consultoria especializada em desenvolvimento de software e soluções de TI. A principal motivação do trabalho era analisar o desempenho do framework para auxiliar na decisão da empresa de utilizá-lo ou não. O trabalho consistiu em comparar três versões de um aplicativo, duas desenvolvidas de forma nativa — uma para iOS e outra para Android —, e a outra desenvolvida a partir do React Native. O aplicativo funciona como um controlador de dispositivos automatizados em uma residência. Para comparar os aplicativos, Hansson e Vidhall utilizaram como métricas de avaliação a taxa de FPS — taxa de *frames* por segundo, importante para definir a fluidez do aplicativo —, uso de CPU e memória, tempo de resposta e tamanho da aplicação. Eles concluíram que, apesar de outros frameworks de desenvolvimento multiplataforma, como Xamarin e PhoneGap, possuírem estudos que evidenciam seus problemas de desempenho em comparação ao desenvolvimento nativo, o React Native apresentou resultados melhores do que o esperado. Dessa forma, Hansson e Vidhall aprovaram e encorajaram o uso do framework.

Wu (2018) fez um estudo comparando os frameworks Flutter e React Native a fim de estabelecer um estudo completo sobre os frameworks, desde o processo de desenvolvimento até o resultado final apresentado pelos aplicativos. Para comparar o processo de desenvolvimento, como estruturação do aplicativo e detalhes de implementação, Wu desenvolveu, em Flutter, uma versão de um aplicativo existente em React Native. Wu também desenvolveu outros dois aplicativos — cada um com uma versão em Flutter e outra em React Native — para comparar o *scroll* de uma lista e a escrita no disco. As versões utilizadas do React Native são diferentes, variando entre as versões 0.44, 0.51 e 0.52. As versões utilizadas do Flutter não foram especificadas. Enquanto a comparação do processo de desenvolvimento é subjetiva, os outros aplicativos apresentaram resultados objetivos. Wu relatou que o React Native apresentou problemas visuais ao lidar com as listas, além de baixo FPS, mas não utilizou nenhuma métrica para avaliá-lo de forma assertiva. Por outro lado, o React Native

apresentou melhores resultados ao escrever em disco. Finalmente, Wu conclui que os dois frameworks são boas opções para o desenvolvimento mobile, apesar de o Flutter ainda estar em suas versões iniciais.

Jagiello (2019), inspirado no trabalho de Wu, também fez um estudo comparando os frameworks Flutter e React Native. Jagiello tinha como objetivo examinar os frameworks e avaliar se existia uma diferença significativa de desempenho, pois isso motivaria os desenvolvedores a optar pelo framework de melhor desempenho. Ele se propôs a desenvolver aplicativos mais simples, que consistem apenas em contadores que são incrementados automaticamente. Ao todo, foram quatro aplicativos, sendo dois de cada framework, e a única métrica utilizada no trabalho foi a taxa de FPS. Os experimentos foram executados apenas no Android e apontaram uma pequena vantagem no desempenho do React Native. Finalmente, Jagiello conclui que ambos os frameworks atenderam às expectativas e suas diferenças de desempenho não são significativas.

Haider (2021), inspirado no trabalho de Hansson e Vidhall, fez um estudo similar utilizando o framework Flutter. Haider, por sua vez, focou exclusivamente na experiência do usuário. A maior motivação para seu trabalho foi a ausência de trabalhos que tratassem do Flutter pela perspectiva do usuário, pois os desenvolvedores querem que os usuários tenham boas experiências ao utilizar seu produto. Então, Haider desenvolveu um aplicativo relativamente complexo, em que o usuário pode visualizar uma lista com filmes em alta no cinema, buscar e visualizar detalhes de um filme. Para medir o desempenho do aplicativo, Haider não utilizou a taxa de FPS. Em vez disso, ele propôs que pessoas utilizassem seus aplicativos e respondessem um questionário contendo informações sobre a experiência do usuário. De acordo com sua pesquisa, o Flutter não apresenta nenhuma desvantagem perceptível em relação ao desenvolvimento nativo e é um framework viável para o desenvolvimento multiplataforma.

3 DESENVOLVIMENTO MOBILE

Raj e Tolety (2012), Latif, et al. (2016) e Pinto e Coutinho (2018) fizeram estudos sobre diferentes formas de desenvolver um aplicativo *mobile*. Diferentes modos de desenvolvimento foram abordados pelos trabalhos, desde desenvolvimento nativo a diferentes abordagens de desenvolvimento multiplataforma. Neste capítulo, serão apresentados os tipos de desenvolvimento nativo e multiplataforma. O desenvolvimento multiplataforma será subdividido em quatro modos: *web*, híbrido, interpretado e *cross-compiled*.

3.1 DESENVOLVIMENTO NATIVO

Um aplicativo é dito nativo quando ele é desenvolvido especificamente para uma única plataforma, utilizando seu próprio kit de desenvolvimento de software (SDK). Aplicativos desenvolvidos dessa forma ficam *linkados* ao sistema operacional e não podem ser reaproveitados em diferentes ambientes. Os dois maiores sistemas operacionais de dispositivos móveis, atualmente, possuem ferramentas próprias especializadas para o desenvolvimento nativo. Utilizando o SDK nativo, o desenvolvedor possui total acesso a todas as funcionalidades do dispositivo, como câmera, sensores, GPS e outros, sem nenhuma restrição. Além disso, um aplicativo nativo é capaz de aproveitar melhor os recursos, como bateria, memória e CPU, usufruindo ao máximo os recursos do dispositivo (PINTO e COUTINHO, 2018).

Cada plataforma possui suas próprias especificidades para desenvolver um aplicativo. O desenvolvedor deve desenvolver o aplicativo em uma linguagem específica, utilizando o conjunto de ferramentas disponibilizado pela plataforma. O código, então, será compilado para a linguagem de máquina reconhecida pelo sistema em questão. Finalmente, o aplicativo pode ser distribuído nas respectivas lojas virtuais. A tabela 1 indica as especificidades das plataformas Android e iOS.

Tabela 1 – Especificidades de desenvolvimento para as plataformas Android e iOS

PLATAFORMA	FERRAMENTA DE DESENVOLVIMENTO	LINGUAGEM DE PROGRAMAÇÃO	AMBIENTE DE DESENVOLVIMENTO	LOJA DE APLICATIVOS
Android	Android Studio	Kotlin/Java	Windows, Linux, macOS	Play Store
iOS	X-code	Swift/Objective-C	macOS	App Store

3.2 DESENVOLVIMENTO MULTIPLATAFORMA

Com o mercado de mobile crescendo gradativamente nos últimos anos, a demanda de aplicativos para dispositivos móveis aumentou consideravelmente. A fim de evitar a replicação de código e facilitar o desenvolvimento mobile, ao longo dos anos surgiram diferentes formas de desenvolver um aplicativo para várias plataformas sem escrever múltiplos códigos. Competindo com os aplicativos nativos, esses são chamados de aplicativos multiplataforma. Existem diferentes formas de desenvolver aplicativos multiplataformas, que podem ser classificadas quanto ao procedimento de desenvolvimento (RAJ e TOLETY 2012; LATIF et al, 2016; PINTO e COUTINHO, 2018) :

1. Desenvolvimento web;
2. Desenvolvimento híbrido;
3. Desenvolvimento interpretado;
4. Desenvolvimento *cross-compiled*.

Cada um dos quatro tipos será explicado e exemplificado nas seguintes subseções.

3.2.1 Desenvolvimento Web

Atualmente, a Web está desenvolvida o suficiente para suportar diferentes resoluções de tela. Assim, um simples site, desenvolvido com HTML, CSS e JavaScript, é capaz de ser acessado por qualquer dispositivo móvel. Existem, ainda, múltiplas ferramentas para auxiliar no desenvolvimento de um aplicativo Web, como Bootstrap[1], para garantir melhor responsividade e até frameworks mais robustas, como Angular[2].

Nesse caso, o aplicativo roda integralmente dentro do *browser* do dispositivo, sem necessidade de instalação. Portanto, qualquer atualização necessária no aplicativo será feita diretamente no servidor, sem a necessidade de atualizações locais. Como o aplicativo não pode ser instalado, não é possível disponibilizá-lo nas respectivas lojas virtuais dos sistemas em questão. O acesso, porém, é bastante trivial, por meio de uma URL. Devido ao amplo uso da Web no mercado e sua padronização, o código pode ser reutilizado em diferentes plataformas.

Apesar das facilidades descritas acima, o desenvolvimento web também possui grandes desvantagens. Obviamente, o aplicativo requer conexão com a Internet para funcionar

---

[1] Disponível em: getbootstrap.com. Acesso em 22/09/2022.
[2] Disponível em: angular.io Acesso em 22/09/2022.

corretamente. Conexões ruins podem afetar drasticamente seu desempenho. O único componente nativo, nessa abordagem, é o browser instalado no dispositivo. Assim, o desenvolvedor não é capaz de acessar quaisquer funcionalidades nativas do dispositivo, como sensores, câmeras, etc.. Em se tratando de aspectos visuais, os aplicativos web têm dificuldade de se assemelharem aos aplicativos nativos, o que pode causar certo desconforto ao usuário. Por fim, todos os detalhes do desenvolvimento web também devem ser levados em consideração, como suportar diferentes tamanhos de telas e diferentes versões de browsers.

3.2.2 Desenvolvimento Híbrido

O desenvolvimento híbrido é uma combinação entre nativo e web. O aplicativo continua sendo desenvolvido com técnicas web, mas ele não é mais acessado diretamente pelo browser. Agora, um componente nativo é utilizado como um container para acessar o HTML. Em vez de utilizar um browser comum, como Chrome ou Firefox, o usuário deve instalar o aplicativo. Basicamente, o aplicativo é apenas um encapsulador, que possui um visualizador web incorporado. Por exemplo, no caso do Android, o visualizador em questão é o WebView[3], enquanto no iOS é o WKWebView[4].

Diferentemente do desenvolvimento web, os aplicativos híbridos podem ser distribuídos nas lojas virtuais como um aplicativo nativo. Outra vantagem em relação ao aplicativo *web* é a capacidade de acessar funcionalidades nativas do dispositivo, que são expostas via APIs em JavaScript por uma camada de abstração, tanto no iOS quanto no Android.

Apesar de corrigir desvantagens do desenvolvimento web e compartilhar das mesmas vantagens, o desenvolvimento híbrido possui alguns desafios. O desempenho ainda não é o ideal, pois o aplicativo continua sendo executado dentro de um mecanismo de browser. Visualmente, assim como anteriormente, o aplicativo dificilmente será equiparado a um aplicativo nativo. A comunicação com o hardware do dispositivo, que não acontecia antes, é feita via JavaScript e está sujeita a vulnerabilidades (RAJ e TOLETY, 2012).

[3] Disponível em: developer.android.com/reference/android/webkit/WebView. Acesso em 22/09/2022.
[4] Disponível em: developer.apple.com/documentation/webkit/wkwebview. Acesso em 22/09/2022.

Atualmente, a ferramenta multiplataforma mais consolidada que opta pela abordagem híbrida é o framework Ionic[5]. Para usufruir de funcionalidades nativas, como bateria, notificações e armazenamento, Ionic utiliza plugins do Apache Cordova[6].

3.2.3 Desenvolvimento Interpretado

Algumas abordagens de desenvolvimento multiplataforma utilizam, de certa forma, componentes do desenvolvimento nativo. É o caso do desenvolvimento interpretado. A ideia central aqui é, com apenas um código, requisitar e utilizar os componentes nativos de cada plataforma, sem a necessidade de códigos diferentes. O código é escrito em alguma linguagem interpretada, como JavaScript ou TypeScript. As funcionalidades nativas, assim como os elementos de UI nativos, serão expostas por uma camada de abstração. Posteriormente, em tempo de execução, o código será interpretado em diferentes plataformas. Nessa abordagem, o aplicativo não está rodando dentro de um mecanismo web. Os elementos presentes na tela são, de fato, os elementos nativos do dispositivo. Isso é possível pois o interpretador interpreta o código fonte e, durante a execução, providencia os elementos requisitados.

Assim como um aplicativo híbrido, um aplicativo interpretado também pode ser distribuído nas lojas virtuais e instalado normalmente. Evidentemente, sua instalação é necessária. Agora, pela primeira vez, o visual alcançado é totalmente equiparável ao visual de um aplicativo nativo, visto que os elementos nativos de UI são invocados pelo interpretador. As funcionalidades nativas do dispositivo estão disponíveis para serem acessadas via API, assim como no aplicativo híbrido.

Apesar de não rodar em um mecanismo web, problemas de desempenho ainda existem nessa abordagem. Mesmo que os elementos e funcionalidades sejam nativos, o fato de ser uma linguagem interpretada traz problemas de desempenho. A espécie de ponte, feita pelo interpretador em tempo de execução, entre o código fonte e o sistema operacional pode ser bastante problemática.

Além do React Native — que será discutido posteriormente —, outros frameworks também utilizam a abordagem interpretada. Alguns exemplos são o Native Script[7] e

[5] Disponível em: ionicframework.com. Acesso em 22/09/2022.
[6] Disponível em: cordova.apache.org. Acesso em 22/09/2022.
[7] Disponível em: nativescript.org. Acesso em 22/09/2022.

Smartface App Studio[8], poucos utilizados atualmente, ambos tendo JavaScript como principal linguagem.

3.2.4 Desenvolvimento Cross-Compiled

Similar ao desenvolvimento interpretado, o código é escrito somente uma vez e aplicativos diferentes serão gerados para cada plataforma. A diferença principal é que, nessa abordagem, o código é compilado para a plataforma em questão, em vez de ser interpretado. O aplicativo, assim como nas duas abordagens anteriores, será distribuído e instalado como um aplicativo nativo. Porém, em vez de ser interpretado em tempo real, todo o aplicativo é compilado para código de máquina que será executado nativamente no dispositivo.

É evidente o ganho de desempenho ao substituir uma linguagem interpretada por uma compilada. O maior benefício esperado nessa abordagem é a possibilidade de alcançar o mesmo desempenho oferecido por um aplicativo nativo. Por mais que seja possível utilizar os elementos nativos de UI com essa abordagem, alguns frameworks optam por montar a interface de outra forma. A comunicação com as funcionalidades nativas do dispositivo não é feita através de API em JavaScript, o que pode corrigir problemas de vulnerabilidade discutidos anteriormente.

Flutter — que será discutido posteriormente — e Xamarin[9] são frameworks consolidadas que trabalham na abordagem cross-compiled. Entretanto, existem grandes diferenças entre ambas que resultam em diferentes desvantagens. Diferentemente de Flutter, Xamarin permite utilizar os componentes nativos de UI do dispositivo. Assim, é possível se beneficiar do visual nativo oferecido pelas plataformas, com o qual o usuário já está acostumado. Porém, não é trivial usufruir dessa funcionalidade e pode ser custoso lidar com a UI nativa, podendo haver a necessidade de escrever código específico para cada plataforma.

[8] Disponível em: smartface.io. Acesso em 22/09/2022.
[9] Disponível em: https://dotnet.microsoft.com/en-us/apps/xamarin. Acesso em 22/09/2022.

4 FLUTTER

O conjunto de ferramentas Flutter[10] começou a ser desenvolvido pela Google em 2017, com sua primeira versão estável sendo disponibilizada em 2018 (FLUTTER, 2022a). Flutter é um framework de código aberto para desenvolvimento de aplicativos multiplataforma com um único código base.

Para permitir a distribuição do aplicativo em múltiplas plataformas, o código desenvolvido na linguagem Dart[11] é compilado para linguagem de máquina seguindo a abordagem *cross-compiled*. Atualmente, Flutter tem suporte para Android, iOS, Web, Windows, macOS e Linux (FLUTTER, 2022b). O Flutter não utiliza os componentes de UI nativos da plataforma. Em vez disso, ele possui sua própria implementação de componentes. No momento em que esse estudo foi desenvolvido, Flutter estava na versão 3.0.5.

Neste capítulo, serão apresentados detalhes do funcionamento interno do Flutter, como o modelo de threads e arquitetura lógica. Por fim, todo o processo de renderização no dispositivo final será explicado e exemplificado. Apesar de Flutter também possuir suporte para Windows, macOS e Linux, este capítulo focará apenas no Android e iOS, visto que são as únicas plataformas que também são contempladas pelo React Native.

4.1 MODELO DE THREADS

Flutter possui um modelo de, ao todo, quatro threads para executar todo o código no dispositivo (FLUTTER, 2022c). As threads são:

A. Thread da plataforma: a thread principal da plataforma em questão. É representada pela UIKit[12] no iOS e pela MainThread[13] no Android. Os códigos de *plugin* que necessitam se comunicar com a plataforma rodam nesta thread.

B. Thread de UI: a thread responsável por executar o código desenvolvido pelo programador. Todo o código em Dart, tanto do aplicativo quanto do próprio framework, é executado nessa thread. Ao renderizar uma tela, esta thread constrói uma árvore com comandos de pinturas — agnósticos à plataforma —, que deverá ser renderizada no dispositivo pela thread de Raster.

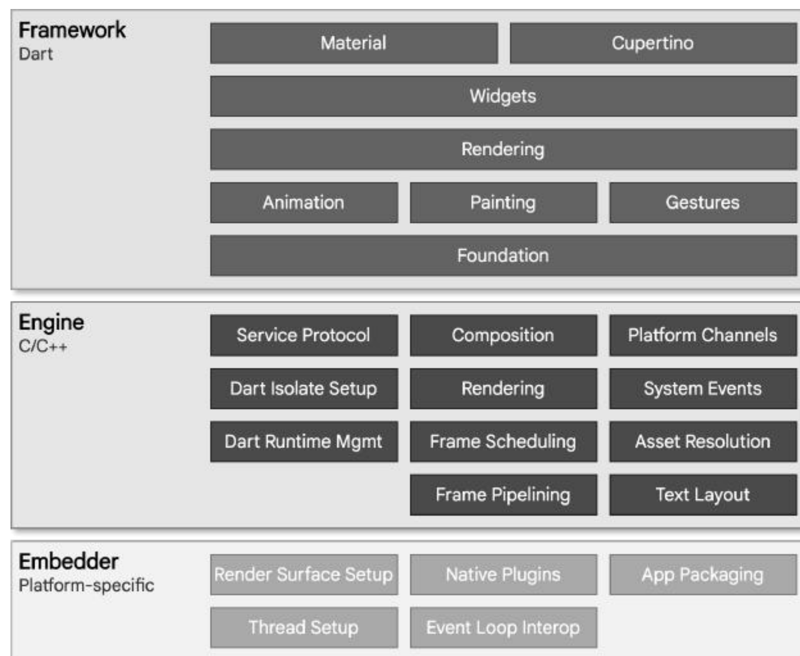
[10] Disponível em: flutter.dev/. Acesso em 2022/09/2022.
[11] Disponível em: dart.dev/. Acesso em 22/09/2022.
[12] Disponível em: developer.apple.com/documentation/uikit. Acesso em 05/08/2022.
[13] Disponível em: developer.android.com/reference/android/support/annotation/MainThread. Acesso em 05/08/2022.

C. Thread de Raster: a thread que se comunica com a GPU para completar a renderização. Esta thread não pode ser acessada diretamente pelo programador. A árvore construída pela thread de UI é renderizada por esta thread, com o uso do motor gráfico Skia[14].
D. Thread de entrada e saída: responsável por executar operações mais custosas, como entrada e saída.

4.2 ARQUITETURA LÓGICA

O conjunto de ferramentas Flutter é constituído de três camadas, como mostra a figura 3. Para a maioria dos programadores, a única camada acessada é a camada Framework. A próxima camada na arquitetura é a Engine, responsável por transformar o código escrito em Dart pelo desenvolvedor em código de máquina a ser lido pelo sistema operacional. Por fim, a camada Embedder se comunica diretamente com o sistema, encapsulando todo o aplicativo em Flutter e renderizando-o na tela do dispositivo .

Figura 3 – Camadas da arquitetura do Flutter



Fonte: (FLUTTER, 2022d)

[14] Disponível em: skia.org. Acesso em 05/06/2022.

4.2.1 Framework

Framework é a última camada na arquitetura do conjunto de ferramentas. É com ela que os desenvolvedores, geralmente, interagem. Todos os seus componentes são escritos em Dart, mesma linguagem que os desenvolvedores utilizam ao escrever um aplicativo em Flutter.

O *Framework* é composto por pacotes, que podem ser importados pelo desenvolvedor ao escrever um aplicativo Flutter. Como ilustrado na figura 3, os pacotes são organizados internamente em uma hierarquia de camadas, mas também podem ser acessados diretamente pelo código da aplicação.

Na primeira camada encontra-se o pacote Foundation[15], que possui classes e funções de baixo nível que serão utilizadas em outras camadas. Na camada acima, utilizando o pacote Foundation, encontram-se outros três pacotes. *Animation*[16], *Painting*[17], *Gestures*[18], possuindo implementações responsáveis por prover funcionalidades de animação, pinturas e gestos do usuário, respectivamente. O pacote de *Rendering*[19], na camada mais acima, provê uma abstração para lidar com layout.

Na próxima camada encontra-se o pacote de *Widgets*[20]. Cada objeto na camada anterior (Rendering) possui um correspondente nesta camada. Os *Widgets* que não possuem características visuais específicas de uma plataforma são definidos aqui. Por exemplo, os *Widgets Row* e *Column*, que apenas organizam seus filhos em uma visualização horizontal e vertical, respectivamente, não são específicos de nenhuma plataforma.

Por fim, na camada mais alta, encontram-se os pacotes Material[21] e Cupertino[22]. Esses pacotes são os mais importantes e, geralmente, os mais utilizados por desenvolvedores comuns. Eles utilizam as primitivas definidas nas camadas abaixo para implementar os elementos de UI seguindo as diretivas do Material Design[23] — proposto pela própria Google para o Android — e Human Interface[24] — proposto pela Apple para dispositivos iOS —, respectivamente.

---

[15] Disponível em: api.flutter.dev/flutter/foundation/foundation-library.html. Acesso em 05/06/2022.
[16] Disponível em: api.flutter.dev/flutter/animation/animation-library.html. Acesso em 05/06/2022.
[17] Disponível em: api.flutter.dev/flutter/painting/painting-library.html. Acesso em 05/06/2022.
[18] Disponível em: api.flutter.dev/flutter/gestures/gestures-library.html. Acesso em 05/06/2022.
[19] Disponível em: api.flutter.dev/flutter/rendering/rendering-library.html. Acesso em 05/06/2022.
[20] Disponível em: api.flutter.dev/flutter/widgets/widgets-library.html. Acesso em 05/06/2022.
[21] Disponível em: api.flutter.dev/flutter/material/material-library.html. Acesso em 05/06/2022.
[22] Disponível em: api.flutter.dev/flutter/cupertino/cupertino-library.html. Acesso em 05/06/2022.
[23] Disponível em: material.io/design. Acesso em 05/06/2022.
[24] Disponível em: developer.apple.com/design/human-interface-guidelines. Acesso em 05/06/2022.

4.2.2 Engine

O núcleo do Flutter, e a próxima camada na arquitetura, é a *Engine*. Ela é majoritariamente escrita em C++ e possui as primitivas necessárias para o funcionamento de um aplicativo Flutter. A *Engine* também possui uma cópia embutida do Skia, um motor gráfico desenvolvido em C/C++ que se comunica com a GPU/CPU do dispositivo para efetuar o processo de renderização na tela. O Skia também é o motor gráfico de outros diferentes produtos, como Google Chrome, Chrome OS e o próprio Android. A *Engine* provê a implementação de baixo nível do núcleo do Flutter, como layout de texto, entrada e saída de arquivos, acesso à Internet, suporte a acessibilidade e um conjunto de ferramentas de execução e compilação de Dart.

A camada *Framework* acessa a *Engine* a partir do pacote **dart:ui**[25], que encapsula o código desenvolvido em C++ em classes escritas em Dart. Esse pacote, então, expõe, para a camada *Framework*, as primitivas de baixo nível.

4.2.3 Embedder

Por fim, na camada mais baixa, efetuando a conexão direta com o dispositivo, encontra-se o *Embedder*. O *Embedder* é uma aplicação totalmente nativa rodando no dispositivo. Ele se comunica com a camada *Engine* através de uma API agnóstica ao sistema operacional. O *Embedder* é responsável por inicializar a *Engine*, lidar com o ciclo de vida do aplicativo e dispositivos de entrada — teclado, mouse, toques na tela —, gerenciamento de threads e comunicação com o sistema operacional. Atualmente, o Flutter possui *embedders* para Android, iOS, Windows, macOS e Linux.

Ao ser inicializado, o *Embedder* é responsável por criar uma visualização em que a *Engine* do Flutter seja capaz de desenhar. No caso do Android, o Flutter fica contido dentro de uma *Activity*. Dentro da *Activity*, o aplicativo em Flutter é encapsulado por uma *FlutterView*, que renderiza todo o conteúdo do aplicativo. Diferente de um aplicativo nativo, em que uma *Activity* é composta por diversos outros elementos — como *Text, Image, Button* —, nesse caso o *FlutterView* é o único elemento reconhecido pelo Android. Todos os elementos do aplicativo não são elementos do Android, são somente itens que estão sendo desenhados sem o conhecimento do sistema.

---

[25] Disponível em: github.com/flutter/engine/tree/main/lib/ui. Acesso em 05/06/2022.

4.3 PROCESSO DE RENDERIZAÇÃO

O código escrito em Dart precisa ser renderizado na tela do dispositivo. Inicialmente, ao entrar em uma tela pela primeira vez, o conteúdo representado pelo código em questão deve ser renderizado desde o princípio na tela. Chamaremos esse processo de renderização inicial. Após o conteúdo ser renderizado pela primeira vez, algumas alterações podem ser engatilhadas a partir do toque do usuário ou da resposta de uma requisição *web*, por exemplo. Nesse caso, como o conteúdo já foi renderizado, não é necessário repetir todo o processo. Apenas as atualizações necessárias serão efetuadas para garantir que o conteúdo esteja atualizado. Chamaremos esse processo de renderização de atualizações. Os dois processos serão explicados posteriormente.

A composição de Widgets é um conceito bastante importante para compreender este processo. De forma geral, Widgets são compostos por outros Widgets de propósito único. Por exemplo, o *Container*[26] — Widget bastante utilizado para encapsular outro Widget, oferecendo funcionalidades de pintura, posicionamento e dimensionamento —, é composto por Widgets mais simples que, combinados entre si, são capazes de oferecer as funcionalidades propostas por ele. Finalmente, o aplicativo como um todo pode ser compreendido como uma estrutura hierárquica de Widgets, chamada de Widget Tree. Essa Widget Tree, obviamente, precisa de um nó raiz — geralmente *MaterialApp*[27] ou *CupertinoApp*[28], que será um Widget pai de vários outros Widgets.

Widgets são imutáveis. Eles podem ser *StatelessWidgets*[29] ou *StatefulWidgets*[30]. Intuitivamente, *StatelessWidgets* são Widgets que não possuem estado e são, de fato, imutáveis e não sofrem nenhuma alteração desde sua criação até o fim de seu ciclo de vida. Por outro lado, de forma menos intuitiva, *StatefulWidgets* também são imutáveis e não sofrem alterações. Porém, *StatefulWidgets* estão atrelados a um estado — um objeto da classe State[31]. O estado de um *StatefulWidget* é mutável e pode sofrer alterações ao longo do seu ciclo de vida, alterando o visual do Widget.

Os Widgets são imutáveis, então toda vez que um Widget precisa ser modificado, uma nova cópia é criada, em vez de modificar a instância existente. Dessa forma, eles não são capazes de guardar informações sobre relacionamentos anteriores (FLUTTER, 2022e). Por

[26] Disponível em: api.flutter.dev/flutter/widgets/Container-class.html. Acesso em 05/06/2022.
[27] Disponível em: api.flutter.dev/flutter/material/MaterialApp-class.html. Acesso em 05/06/2022.
[28] Disponível em: api.flutter.dev/flutter/cupertino/CupertinoApp-class.html. Acesso em 05/06/2022.
[29] Disponível em: api.flutter.dev/flutter/widgets/StatelessWidget-class.html. Acesso em 05/06/2022.
[30] Disponível em: api.flutter.dev/flutter/widgets/StatefulWidget-class.html. Acesso em 05/06/2022.
[31] Disponível em: api.flutter.dev/flutter/widgets/State-class.html. Acesso em 07/08/2022.

esse motivo, a *Element Tree* e *Render Tree* devem ser introduzidas. A *Element Tree* é basicamente uma representação, mutável, auxiliar da *Widget Tree*. Ela contém as informações sobre o estado dos Widgets, além de suas relações de parentesco. Ainda mais importante: a *Element Tree* é responsável por fazer a conexão entre a *Widget Tree* e a *Render Tree,* que de fato será renderizada no dispositivo. Uma *Render Tree* é composta por *RenderObjects*[32], objetos que contêm toda a lógica necessária para renderizar um Widget. *RenderObjects* são responsáveis por lidar com layout, pintura e teste de colisão e são muito mais custosos para serem instanciados do que um Widget. Cada *Element* na *Element Tree* está associado a um *Widget* na *Widget Tree* e a um *RenderObject* na *Render Tree*.

4.3.1 Renderização Inicial

Para ilustrar o processo de renderização inicial, vamos tomar como exemplo a renderização do *Widget* representado pelo código 1:

Código 1 – Renderização de um Flutter Widget

```
class MyComponent extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Container(
      color: Colors.red,
      child: Text("Hello, world!"),
    );
  }
}
```

O *Widget* representado pelo código 1 é um simples texto encapsulado por um bloco vermelho. O *Widget MyComponent*, então, é uma árvore composta por outros dois *Widgets*: *Container* — seu filho direto — e *Text* — filho do *Container*.

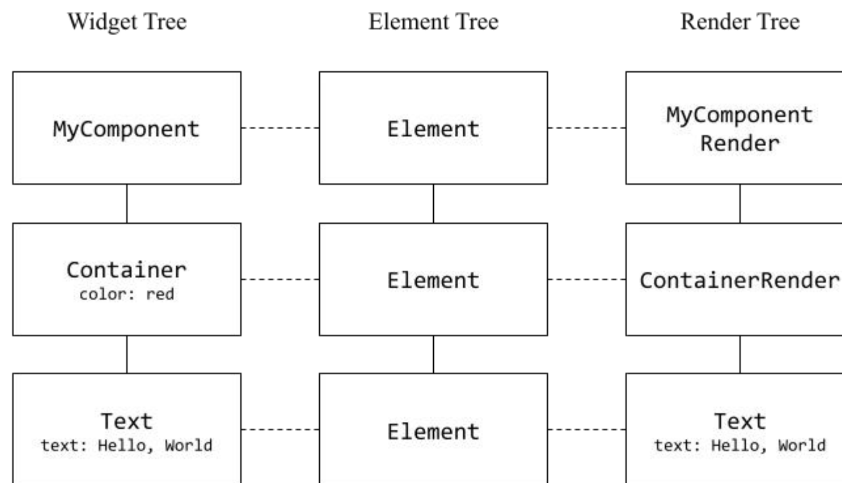
A partir dessa *Widget Tree*, será criada, então, a *Element Tree*. O Flutter vai percorrer toda a *Widget Tree* chamando o método *createElement()* em cada um dos Widgets. Para cada Widget, esse método irá criar o Element correspondente, criando, assim, a *Element Tree*. Assim como a *Widget Tree*, ela também terá três nós.

Finalmente, de forma similar, o Flutter irá criar um *RenderObject* para cada *Element Tree* a partir do método *createRenderObject()*. Agora, teremos uma *Render Tree*, com um nó para cada nó da *Element Tree*. Cada Element da *Element Tree* guardará uma referência de seu

[32] Disponível em: api.flutter.dev/flutter/rendering/RenderObject-class.html. Acesso em 07/08/2022.

correspondente na *Widget Tree* e *Render Tree*. O resultado final das três árvores pode ser visualizado na figura 4.

Figura 4 – Árvores criadas durante o processo de renderização no Flutter



Na primeira renderização, a árvore precisa ser criada do zero. Como dito anteriormente, os *RenderObjects* são consideravelmente mais pesados do que os *Widgets* e os *Elements*, pois possuem a lógica necessária para a renderização na tela. Por esse motivo, tem-se a necessidade de construir três diferentes árvores. A *Widget Tree* poderá ser modificada frequentemente, pois a modificação de um único *Widget* provoca a construção de uma nova árvore, pois ela é imutável. Toda vez que uma *Widget Tree* é modificada — a partir de uma atualização de estado causada pelo toque do usuário, por exemplo —, o Flutter utiliza a *Element Tree* para comparar os novos Widgets com os *RenderObjects* já existentes. Quando não há a necessidade de atualização, o *RenderObject* pode ser reaproveitado, evitando reconstruções desnecessárias.

A *Render Tree* é, finalmente, renderizada na tela com o uso do motor gráfico Skia. A renderização é executada na *Thread de Raster*, enquanto a criação das árvores é feita na *Thread de UI*.

4.3.2 Renderização de atualizações

Como Widgets são imutáveis, a *Widget Tree* pode acabar sendo reconstruída múltiplas vezes. No código 1, caso alterássemos a cor do Container, toda a *Widget Tree* seria reconstruída. Em termos de desempenho, não há problema nenhum nisso, pois Widgets são

extremamente "baratos" de se construir. Entretanto, reconstruir toda a *Render Tree* seria extremamente problemático, visto que *RenderObjects* são mais custosos que Widgets. Nesse momento, justifica-se a utilização de três árvores.

Ao alterar a cor do Container, por exemplo, uma nova *Widget Tree* será criada, com um novo Widget contendo a nova cor. Para cada *Element* presente na *Element Tree*, será feita uma comparação entre o novo Widget e o *RenderObjec*t da *Render Tree* anterior. O Element relacionado ao Container deve verificar que uma atualização é necessária no respectivo *ContainerRender*. Como o tipo do Widget não foi alterado — o Container apenas teve sua cor modificada, mas continua sendo um Container —, o *ContainerRender* pode ser aproveitado, sem a necessidade da criação de um novo *RenderObject*. Dessa forma, será feita apenas uma mudança de propriedade no *ContainerRender,* já renderizado na tela, para alterar sua cor, enquanto todos os outros *RenderObject* são reaproveitados. Por mais que toda a *Widget Tree* seja reconstruída múltiplas vezes, a *Render Tree* é atualizada somente quando necessário.

5 REACT NATIVE

O React Native[33] é um framework desenvolvido pela empresa Facebook — hoje chamada de Meta Platforms. Sua primeira versão estável foi disponibilizada em 2015 (REACT NATIVE, 2022a). O propósito do framework é permitir a utilização da biblioteca ReactJS[34], já consolidada no mercado web, para desenvolver aplicativos nativos para Android e iOS.

O React Native segue a abordagem de desenvolvimento interpretado para executar os aplicativos em múltiplas plataformas. Diferente do Flutter, o React Native possui suporte apenas para dispositivos móveis, atualmente Android e iOS (REACT NATIVE, 2022b). O código escrito em JavaScript deve ser interpretado para que o React Native possa se comunicar com o dispositivo e utilizar seus componentes e módulos nativos.

No momento em que este estudo foi desenvolvido, React Native estava na versão 0.69. Porém, durante o desenvolvimento, alguns problemas de compatibilidade foram encontrados ao utilizar uma biblioteca necessária para o desenvolvimento de alguns aplicativos. Dessa forma, fixamos todos os aplicativos para utilizarem a versão mais recente que não apresentasse problemas. Assim, a versão escolhida foi a 0.68.2.

Neste capítulo, serão apresentados detalhes do funcionamento interno do React Native, como o modelo de threads e arquitetura. Por fim, todo o processo de renderização no dispositivo final será explicado e exemplificado.

5.1 MODELO DE THREADS

Ao todo, o renderizador do React Native utiliza três threads para renderizar o aplicativo no dispositivo (REACT NATIVE, 2022c). As threads são:

A. Thread de UI: a única thread que pode lidar diretamente com os componentes nativos nas plataformas finais. É responsável por renderizar os componentes na tela do dispositivo, assim como efetuar todas as atualizações necessárias.
B. Thread de JavaScript: thread responsável por executar o código desenvolvido pelo programador. A criação e interpretação da árvore de elementos ocorre nesta thread.
C. Thread de background: thread dedicada para cálculos de layout. A árvore de elementos tem suas informações de layout calculadas nesta thread.

[33] Disponível em: reactnative.dev. Acesso em 07/10/2022.
[34] Disponível em: reactjs.org. Acesso em 07/10/2022.

5.2 ARQUITETURA

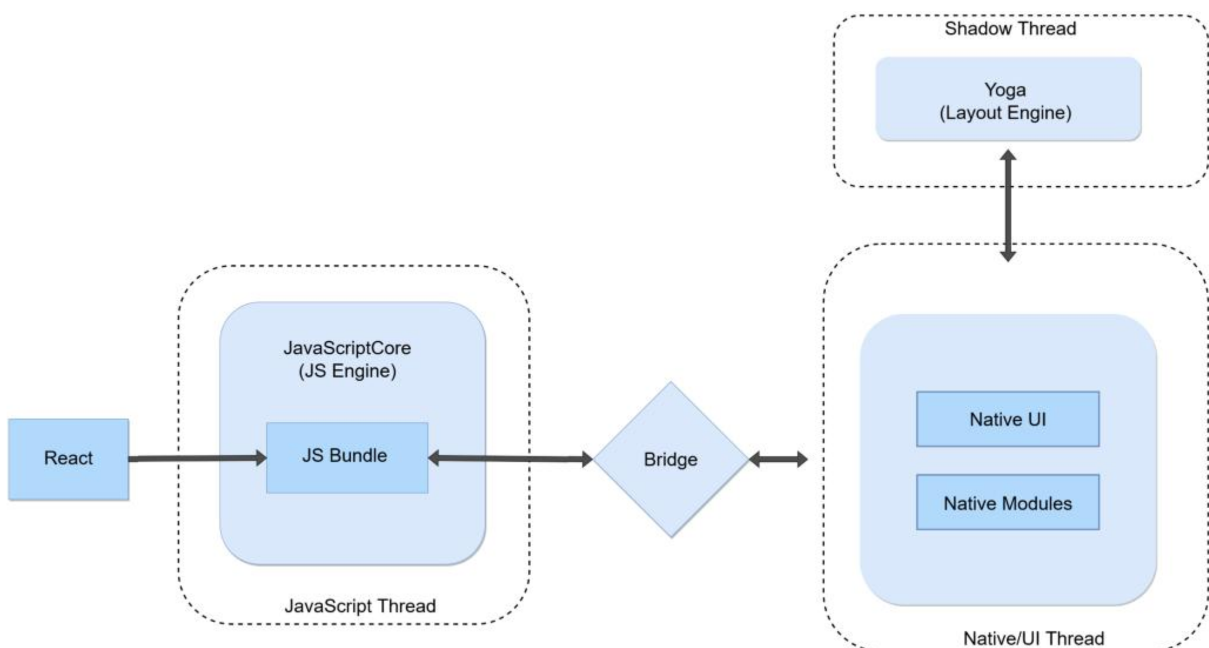
Durante o desenvolvimento deste estudo, a nova arquitetura com o novo renderizador ainda estava em fase de lançamento e o lançamento da documentação oficial estava em andamento. Os estudos feitos por Hansson e Vidhall (2016), Wu (2018) e Jagiello (2019) foram desenvolvidos com a antiga arquitetura e não devem ser considerados como referências sobre o funcionamento interno do framework.

Atualmente, a arquitetura do React Native conta com um novo renderizador, chamado de Fabric. O Fabric teve seu desenvolvimento iniciado em 2018 e foi implantado no React Native em 2021 (REACT NATIVE, 2022d).

5.2.1 Antiga Arquitetura

O principal componente da antiga arquitetura é a *Bridge*, uma ponte entre o código desenvolvido em JavaScript e a parte nativa do dispositivo, como pode ser visto na figura 5. Toda a comunicação do React Native com o dispositivo é feita através dessa ponte. A ponte é capaz de enviar apenas conteúdo serializado como JSON, nos dois sentidos. Além disso, a comunicação é sempre assíncrona.

Figura 5 – Antiga arquitetura do React Native



Fonte: (PATIL, 2022)

Os módulos nativos, como por exemplo *bluetooth,* geolocalização e armazenamento interno, precisam ser inicializados no momento de abertura do aplicativo. Para acessar esses módulos, a comunicação com o dispositivo deve ser feita, obviamente, a partir da ponte. A thread de JavaScript deve preparar a mensagem para a thread nativa, que será serializada como JSON para ser enviada pela ponte. Após a recepção da mensagem, ela precisa ser decodificada para que, enfim, o código nativo seja executado pela thread nativa.

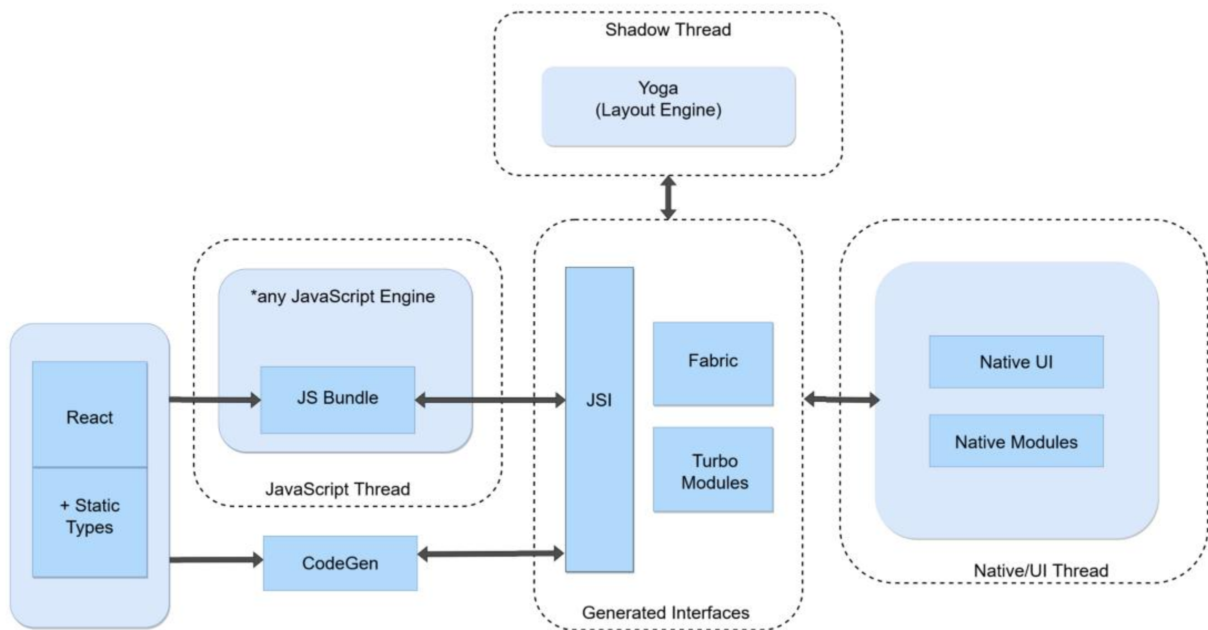
5.2.2 Nova Arquitetura

Na nova arquitetura, apresentada na figura 6, um componente bastante importante é o novo renderizador *Fabric*, desenvolvido em C++. Ele é o responsável pela interação com o dispositivo para renderizar o conteúdo do aplicativo. Essa comunicação é feita com o Android a partir do JNI (*Java Native Interface*), uma API para escrever métodos nativos em Java. A comunicação entre o JavaScript e o *Fabric* é feita, de forma síncrona, via JSI (*JavaScript Interface*). Dessa forma, os métodos nativos são expostos para o JavaScript a partir de objetos em C++, que podem ser referenciados diretamente pelo JavaScript.

Diferentemente da arquitetura antiga, agora os módulos não precisam mais ser inicializados ao abrir o aplicativo. O JavaScript possui uma referência a esses módulos a partir da interface *Turbo Modules*. Dessa forma, o tempo de início do aplicativo pode ser reduzido, além de permitir uma comunicação mais eficiente com os módulos nativos.

Finalmente, outro componente importante é a ferramenta chamada *CodeGen*. Como o JavaScript é uma linguagem dinamicamente tipada, enquanto o C++ possui tipagem estática, é necessário garantir uma comunicação suave entre as duas linguagens. Para isso, o React Native utiliza o *CodeGen* para definir interfaces utilizadas pelo *Turbo Modules* e o *Fabric*.

Figura 6 – Nova arquitetura do React Native



Fonte: (PATIL, 2022)

5.3 PROCESSO DE RENDERIZAÇÃO

Assim como no Flutter, o código, dessa vez escrito em JavaScript, precisa ser renderizado na tela do dispositivo. Da mesma forma, o processo de renderização pode ser dividido em renderização inicial — a renderização que ocorre pela primeira vez ao entrar em uma tela — e renderização de atualizações — renderização das mudanças causadas pelo toque do usuário, por exemplo. Os dois processos serão explicados posteriormente.

De forma similar ao Flutter, a composição de elementos no React Native também é um conceito bastante importante. Os componentes visuais, aqui chamados de *React Elements*, são compostos de outros *React Elements*. Cada *React Element* pode ser reduzido em uma composição de *React Host Components* — elementos que possuem uma implementação nas respectivas plataformas, como por exemplo *View*, *Button* e *Text*.

Como o React Native segue uma abordagem interpretada, os *React Elements*, objetos em JavaScript, devem ser interpretados para que sejam renderizados na tela. Eles deverão ser transformados em *React Shadow Nodes*, o correspondente em C++ dos elementos em JavaScript. Assim, o Fabric é capaz de renderizá-los na tela do dispositivo a partir do JNI.

5.3.1 Renderização Inicial

Para ilustrar o processo de renderização inicial, vamos tomar como exemplo a renderização do *React Element* representado pelo código 2:

Código 2 – Renderização de um React Element

```
class MyComponent extends React.Component {
  render() {
    return (
      <View style={{backgroundColor: 'red'}}>
        <Text>
          Hello, World!
        </Text>
      </View>
    )
  }
}
```

O elemento representado pelo código 2 é um simples texto encapsulado por um bloco vermelho. Primeiramente, o React precisa reduzir, recursivamente, esse *React Element* em *React Host Components*. O exemplo acima é bastante simples e essa redução será aplicada somente no elemento *MyComponent*, visto que todos os seus filhos já são *React Host Components*. Ao final da redução, a *React Element Tree* estará formada, com os nós *View* e *Text* sendo filhos de um nó raiz. Importante notar que o *MyComponent* não está nessa árvore, pois ele foi reduzido em outros dois elementos.

Durante o processo de redução, conforme cada *React Host Component* vai sendo invocado, o renderizador também cria, de forma assíncrona, um *React Shadow Node*. No código 2, um *ViewShadowNode* é criado a partir do elemento *View*, e um *TextShadowNode*, a partir do elemento *Text*. Importante notar, novamente, que não existe um *React Shadow Node* que representa diretamente o elemento *MyComponent*.

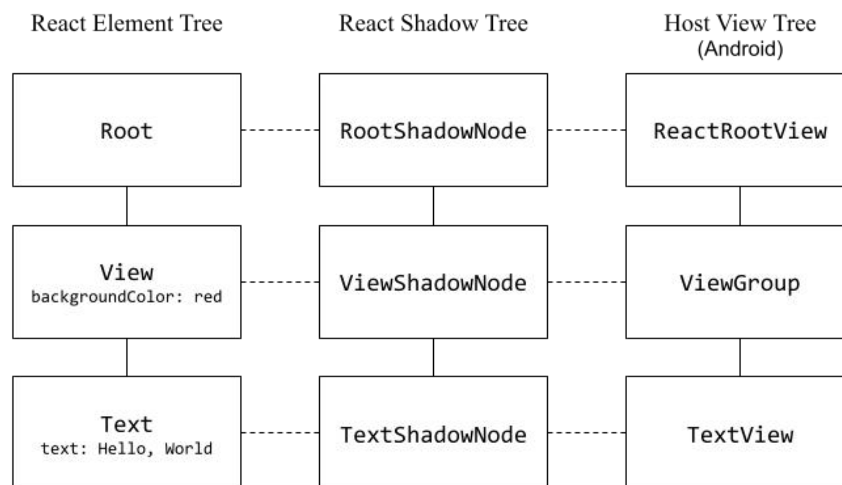
Sempre que uma relação de pai e filho é criada entre dois *React Element Nodes*, a mesma relação é criada entre seus *React Shadow Nodes* correspondentes. Assim, a *React Shadow Tree* é montada.

Na próxima etapa, o mecanismo de layout utilizado pelo *React Native*, chamado de *Yoga*, calcula a posição e tamanho de cada *React Shadow Node*. O cálculo é feito utilizando as propriedades de estilo do *React Element*, definidas em JavaScript, em conjunto com as restrições de layout do nó raiz da *React Shadow Tree*, que determina a quantidade de espaço disponível para o nó em questão. Importante ressaltar que, apesar da maioria dos cálculos serem executados em C++, o cálculo de alguns componentes depende da plataforma. Nesse

caso, o *Yoga* precisa invocar uma função definida na plataforma para realizar o cálculo. Após ter suas informações de layout já calculadas, a *React Shadow Tree* é promovida como “a próxima árvore” a ser montada. Isso indica que a árvore já possui todas as informações necessárias para ser renderizada e representa o estado mais recente da *React Element Tree*. A árvore será, então, renderizada no próximo ciclo da thread de UI.

Finalmente, a *React Shadow Tree* deve ser transformada em uma *Host View Tree*, composta por *Host Views* — os elementos nativos das respectivas plataformas. Então, o renderizador cria um *Host View* correspondente para cada *React Shadow Node* e o renderiza na tela. Para o exemplo apresentado pelo código 2, no caso do Android é criado uma instância de *android.view.ViewGroup* para o elemento *View* e outra de *android.widgets.TextView* para *Text*. Analogamente, no iOS, uma *UIView* é criada, enquanto o texto é escrito na tela com *NSLayoutManager*. Cada *Host View* é configurado com as propriedades do *React Shadow Node*, e seu tamanho e posição são definidos com a informação calculada na etapa anterior. A figura 7 ilustra o resultado final das três árvores criadas durante todo o processo de renderização.

Figura 7 – Árvores criadas durante o processo de renderização no React Native



Ao final da renderização, a React Shadow Tree é promovida a “última árvore renderizada”. Dessa forma, nas atualizações posteriores, o renderizador será capaz de efetuar um *diff* entre esta e a próxima árvore para computar quais são as atualizações necessárias.

5.3.2 Renderização de atualizações

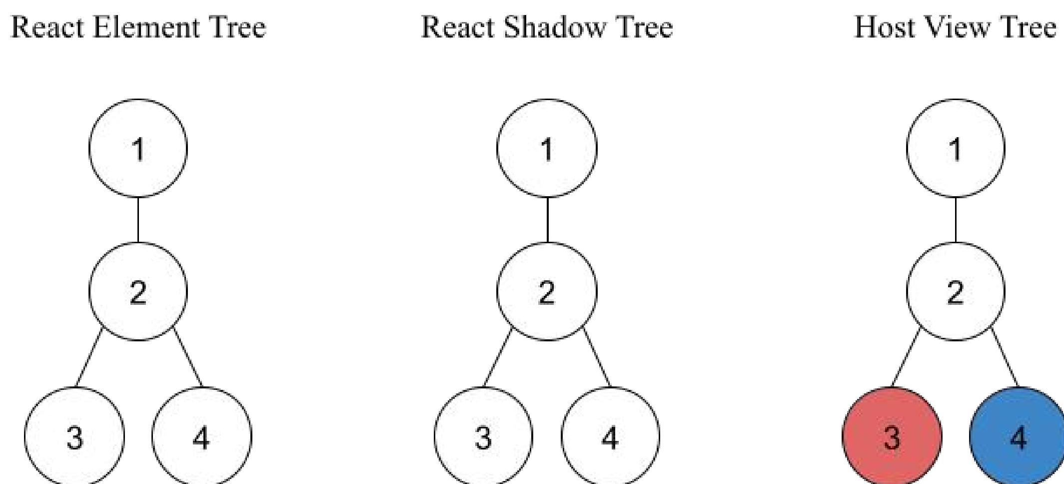
Assim como no Flutter, o processo de renderização do React Native consiste em criar três árvores. Porém, diferentemente do Flutter, o React Native não possui uma árvore auxiliar para efetuar comparações e definir o que deve ser reaproveitado. Tanto a *React Element Tree* quanto a *React Shadow Tree* são imutáveis. Dessa forma, para evitar reconstruir árvores inteiras a toda atualização, o React Native faz o uso de compartilhamento estrutural. Para ilustrar como funciona esse processo de atualização, vamos tomar como exemplo a renderização do elemento representado pelo código 3:

Código 3 – Atualização de um *React Element*

```
class MyComponent extends React.Component {
  render() {
    return (
      <View>
        <View style={{ backgroundColor: 'red', height: 20, width: 20 }} />
        <View style={{ backgroundColor: 'blue', height: 20, width: 20 }} />
      </View>
    )
  }
}
```

O elemento representado pelo código 3 é apenas um bloco que encapsula outros dois blocos, um vermelho e outro azul. Aplicando o que foi apresentado na seção anterior, teremos as seguintes árvores, como mostra a figura 8:

Figura 8 – Árvores criadas durante o processo de atualização no React Native



O nó representado pelo número 1 corresponde ao nó raiz, enquanto os demais nós correspondem a cada uma das Views presentes no código 3. Os nós 3 e 4 representam,

respectivamente, as Views de cor vermelho e azul. Ao atualizar a cor da View representada pelo nó 3 de vermelho para amarelo, por exemplo, o React Native deve atualizar a cor do elemento já renderizado na tela.

Primeiramente, ao executar uma alteração, todos os nós afetados devem ser clonados e modificados. Para minimizar a sobrecarga causada pela imutabilidade, o renderizador faz o uso de compartilhamento estrutural. Assim, os nós que não sofrerem alterações serão compartilhados entre mais de uma árvore.

No exemplo do código 3, ao modificar a cor do nó 3, o renderizador criará uma nova árvore com as seguintes operações:

1. `No3’ ← ClonaNo(No3, {backgroundColor: ‘yellow’})`
2. `No2’ ← ClonaNo(No2)`
3. `AdicionaFilho(No2’, No3’)`
4. `AdicionaFilho(No2’, No4)`
5. `No1’ ← ClonaNo(No1)`
6. `AdicionaFilho(No1’, No2’)`

O primeiro passo é alterar o nó que teve o estado modificado. Ou seja, o Nó 3, que teve seu *backgroundColor* alterado de vermelho para amarelo. O segundo passo é clonar o Nó 2, que está imediatamente acima do Nó 3. O terceiro e quarto passo são adicionar os filhos do Nó 2’. Os filhos serão o Nó 3’, que é o novo clone do Nó 3, e o Nó 4, que não foi alterado. O quinto passo é clonar o Nó 1, o próximo nó no caminho. Por fim, basta adicionar o Nó 2’ como filho do Nó 1.

Assim como na renderização inicial, também é necessário efetuar as etapas de cálculo de layout e promoção da nova árvore, marcando-a como a próxima a ser montada. Além disso, agora é necessário fazer um *diff* entre a árvore anterior e a próxima árvore a ser montada que acabou de ser promovida.

O cálculo de layout ocorre de forma bastante similar ao cálculo na renderização inicial. Importante notar que esse cálculo pode ocasionar a clonagem de nós que estavam sendo compartilhados até então. Isso pode ocorrer quando o pai de um elemento tem seu tamanho alterado e, consequentemente, o elemento tem seu tamanho ou posição afetado. Nesse caso, os resultados do cálculo de layout desse nó precisam ser alterados. Como os nós são imutáveis, ele deve ser clonado e não será mais compartilhado entre as árvores.

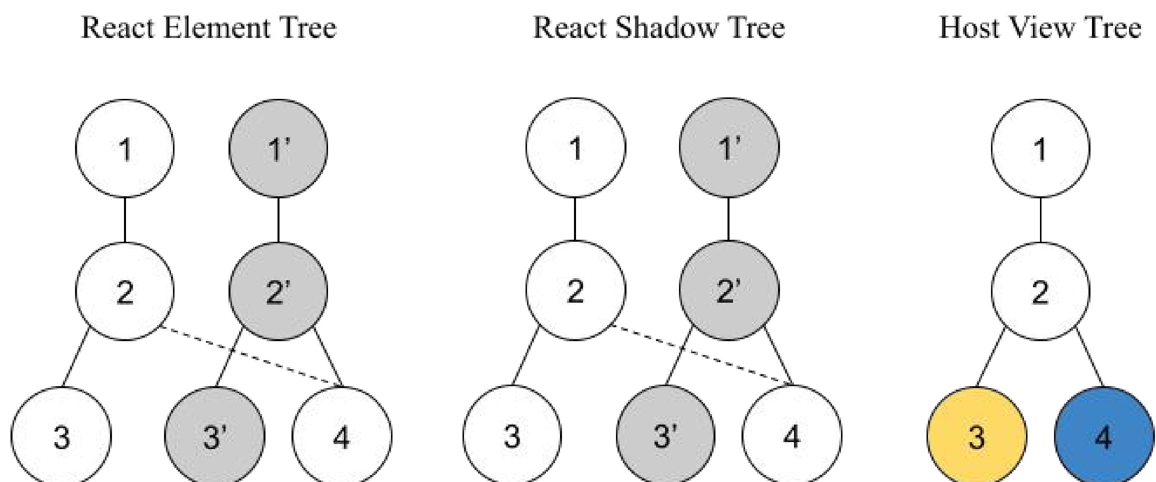
Após finalizar a montagem da *React Shadow Tree,* é necessário realizar o *diff* entre a última árvore montada e a próxima. O resultado deve ser uma lista de operações atômicas de

mutação para serem executadas nas *Host Views*. No exemplo citado acima, o resultado dessa etapa seria uma operação para atualizar a cor de fundo do Nó 3 para amarelo. Interessante notar que essa última etapa não é necessária na renderização inicial, pois não existe árvore anterior e todos os elementos da primeira árvore a ser montada devem ser criados do zero.

Finalmente, são aplicadas as operações da lista resultante do *diff* na fase anterior. No exemplo acima, em vez de criar uma *Host View Tree*, como acontece na renderização inicial, ela só será atualizada. Dependendo da mudança do estado, novas *Host Views* podem ser criadas, e antigas podem ser deletadas ou atualizadas. Por fim, a *React Shadow Tree* é, como na renderização inicial, promovida a "última árvore renderizada".

A visualização das três árvores após todo o processo de renderização de atualizações pode ser vista na figura 9. Importante verificar o uso do compartilhamento estrutural com o nó 4.

Figura 9 – Árvores criadas após o processo de atualização no React Native



6 ESTUDOS DE CASO

Neste capítulo encontram-se as informações sobre os aplicativos desenvolvidos durante o estudo deste trabalho. As duas métricas utilizadas — quantidade de *janky frames* e quantidade de espaço em branco visível nas rolagens de uma lista — para a comparação do desempenho, assim como os *scripts* desenvolvidos para automatizar os experimentos, também serão abordadas neste capítulo.

Todo o código escrito durante este trabalho está disponível no repositório do *github*[35]. Os aplicativos estão na pasta raiz do projeto, enquanto os *scripts* encontram-se organizados na pasta ***scripts***.

6.1 APLICATIVOS DESENVOLVIDOS

Para comparar o desempenho de ambos os frameworks, tomamos como exemplo, principalmente, a abordagem seguida por Jagiello (2019). Em seu estudo, Jagiello desenvolveu aplicativos similares utilizando os dois frameworks, Flutter e React Native, e comparou o desempenho apresentado por ambos. O primeiro aplicativo possui uma tela com um contador que tem seu valor incrementado repetidamente em uma frequência de tempo determinada. O segundo aplicativo é apenas uma expansão do primeiro, com seis contadores simultâneos. Ao final de seu estudo, Jagiello concluiu que os experimentos efetuados geraram resultados que foram contra à sua hipótese inicial — de que Flutter apresentaria melhores resultados. Por fim, o autor relata que seus aplicativos eram muito simples, e aplicativos mais complexos deveriam ter sido utilizados para garantir uma melhor confiabilidade no estudo.

Para este trabalho, desenvolvemos aplicativos simples, cada um com funcionalidades específicas, como resposta ao toque do usuário e interação com listas e navegação entre telas. Os aplicativos propostos por Jagiello também foram reproduzidos.

Os aplicativos desenvolvidos em Flutter não utilizam nenhum pacote ou biblioteca de terceiros. Por outro lado, os aplicativos desenvolvidos em React Native utilizam as bibliotecas *react-native-paper* e *react-native-vector-icons* para possibilitar o uso de elementos de *material UI* — elementos de interface seguindo as diretivas do *Material Design* da Google. Também é utilizada a biblioteca *react-navigation* quando o aplicativo possui navegação entre

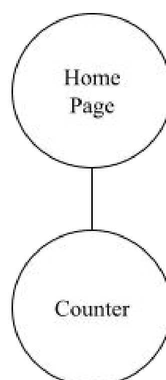
[35] Disponível em: github.com/collbreno/flutter-vs-react-native. Acesso em 13/10/2022.

telas. O uso de bibliotecas de terceiros é necessário pois o React Native não possui essas funcionalidades (*material UI* e navegação entre telas) implementadas em seu código base.

6.1.1 Aplicativo 1 - *Stopwatch*

O primeiro aplicativo é uma cópia aproximada do primeiro aplicativo desenvolvido por Jagiello (2019). O aplicativo consiste em apenas uma tela que possui um contador centralizado. Esse contador é incrementado, automaticamente, como um cronômetro, em uma unidade a cada 16 milissegundos. O objetivo é testar uma simples atualização recorrente. O tempo escolhido para a frequência de atualização deve-se ao fato de 16 milissegundos ser o menor valor possível que não ultrapassa a marca de 60 frames por segundo, visto que essa é a taxa de atualização da tela dos dispositivos. Uma simples ilustração da árvore de componentes do aplicativo pode ser vista na figura 10.

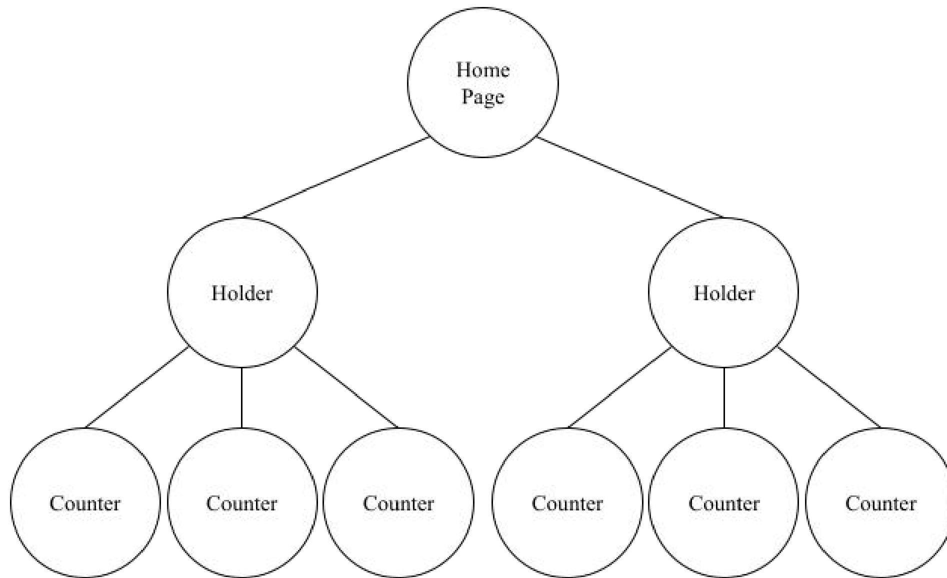
Figura 10 – Árvore de componentes do aplicativo 1



6.1.2 Aplicativo 2 - *Multi Stopwatch*

O segundo aplicativo é, novamente, uma cópia do aplicativo desenvolvido por Jagiello (2019). Dessa vez, Jagiello utilizou seis contadores simultâneos, como mostra a figura 11. Três contadores são encapsulados por um Holder e têm seus valores atualizados em uma frequência de 16 milissegundos. Outros três contadores são encapsulados por outro Holder, com uma frequência de 32 milissegundos. Isso significa que, em 50% das vezes, apenas três contadores serão atualizados, enquanto os outros três permanecerão inalterados. O objetivo é verificar se o framework está gerenciando as atualizações de estado de forma eficiente, evitando problemas de desempenho.

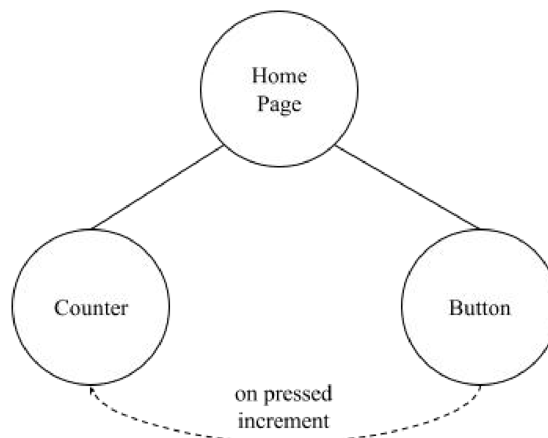
Figura 11 – Árvore de componentes do aplicativo 2



6.1.3 Aplicativo 3 - *Counter*

O terceiro aplicativo, primeiro a lidar com toques do usuário, é composto por um texto centralizado na tela e um botão localizado no canto inferior. O botão, ao ser pressionado, dispara uma mudança de estado no aplicativo, que reflete na incrementação do valor apresentado no texto centralizado. A árvore de componentes, assim como as ações emitidas pelo usuário, pode ser visualizada na figura 12. Esse é o aplicativo padrão gerado pelo Flutter ao criar um projeto pela primeira vez. A versão em React Native é, então, uma cópia da versão em Flutter.

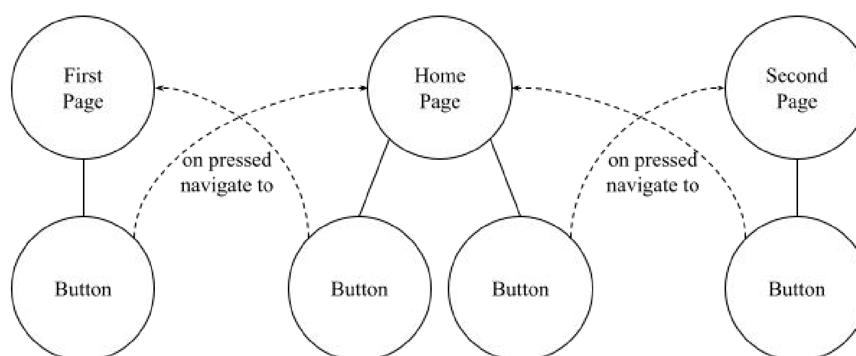
Figura 12 – Árvore de componentes do aplicativo 3



6.1.4 Aplicativo 4 - *Navigations*

O quarto aplicativo é composto por três telas simples. A tela inicial possui dois botões. Cada botão, ao ser clicado, navega para uma tela distinta. As duas outras telas são semelhantes e possuem apenas um botão, que retorna para a tela inicial. Apesar de ambos os frameworks possuírem opções para customizar as animações ao navegar entre telas, optamos por utilizar as animações padrões. Dessa forma, os dois aplicativos possuem exatamente a mesma animação. As árvores de componentes de cada uma das telas, assim como as ações executadas a partir do toque nos botões, podem ser visualizadas na figura 13. O objetivo desse aplicativo é verificar como os frameworks lidam com a navegação entre diferentes telas.

Figura 13 – Árvore de componentes do aplicativo 4

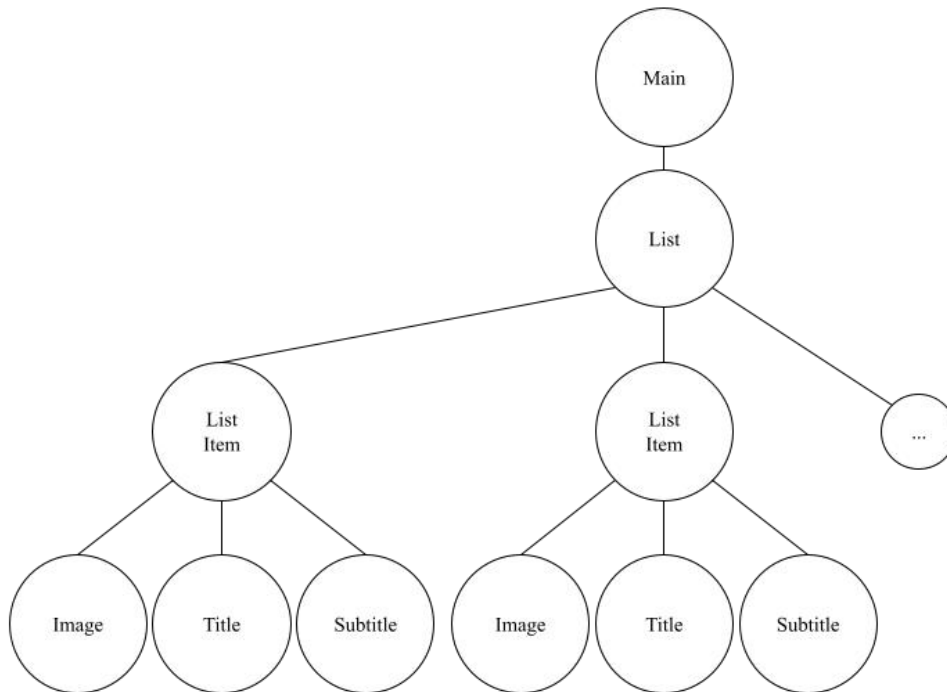


6.1.5 Aplicativo 5 - *List*

O quinto aplicativo é responsável por comparar o desempenho dos frameworks ao lidar com grandes listas. Ele possui apenas uma tela, que contém uma única lista, como mostra a figura 14. Ao todo, a lista possui mil itens — mil nomes de animais e ícones gerados pelo site *Mockaroo*[36], um gerador de dados aleatórios. Cada item da lista é renderizado na tela somente com um título, um subtítulo e uma imagem. Os itens da lista são simples e não possuem grande complexidade, pois o objetivo deste aplicativo é avaliar o desempenho da tarefa de atualização da lista na tela.

[36] Disponível em: mockaroo.com/. Acesso em 16/06/2022.

Figura 14 – Árvore de componentes do aplicativo 5



6.2 MINIMIZANDO INTERFERÊNCIAS EXTERNAS

Para a realização dos experimentos, é importante minimizar ao máximo possíveis interferências externas para garantir que os resultados não sejam enviesados. Alguns aplicativos requerem interações durante suas execuções. No caso dos aplicativos que requerem toques em botões, como os aplicativos 3 e 5, toques mais longos, pressionando o botão por um maior período, poderiam resultar em animações de maiores durações, com mais frames a serem renderizados. No aplicativo 4, o *scroll* também pode variar bastante dependendo da velocidade e duração do deslizamento do dedo. Deslizamentos mais rápidos poderiam resultar em uma maior quantidade de itens da lista sendo renderizados. Portanto, para evitar possíveis discrepâncias nos experimentos devido às imperfeições das interações do usuário, todos os comandos são simulados via ADB[37] — *Android Debug Bridge,* uma ferramenta desenvolvida pela Google para facilitar a comunicação com o dispositivo Android. Com essa ferramenta, é possível simular toques e deslizamentos na tela de forma uniforme. Os toques são simulados a partir do comando `adb shell input tap`, enquanto os

[37] Disponível em: developer.android.com/studio/command-line/adb. Acesso em 10/10/2022.

deslizamentos são simulados a partir do comando `adb shell input swipe`. Dessa forma, pode-se garantir que os toques serão sempre uniformes, resultando em animações de tempos consistentes.

A fim de garantir alta confiabilidade nos experimentos, os aplicativos devem ser executados mais de uma vez. Após avaliar os resultados obtidos com poucas execuções, incrementamos o número de execuções gradativamente até encontrar resultados que apresentassem poucos *outliers*. Optamos, então, por executar cada aplicativo 50 vezes.

Para isso, automatizamos todo o processo fazendo uso de um *script* na linguagem Python. Cada um dos cinco aplicativos possui um JSON de configuração, que contém, por exemplo, quais são os comandos que devem ser executados durante o experimento — como os toques e deslizamentos. Então, para cada aplicativo, o script deve ler o arquivo de configuração, iniciar o aplicativo, executar todos os comandos necessários uma quantidade determinada de vezes e, por fim, salvar as informações de frames por segundo em um arquivo que será analisado posteriormente.

6.3 COLETANDO A QUANTIDADE DE JANKY FRAMES

A quantidade de frames por segundo de uma animação é importante para definir sua fluidez e suavidade. Flutter e React Native têm como objetivo prover um desempenho de 60 frames por segundo em seus aplicativos (FLUTTER, 2022c; REACT NATIVE, 2022e). Então, assim como Wu (2018) e Jagiello (2019), também utilizamos a taxa de frames como uma métrica de desempenho. Para alcançar a taxa de 60 frames por segundo, cada frame deve ser construído e renderizado em até 16,6 milissegundos. Todo frame que ultrapassa esse limite de tempo é chamado de *janky frame*. Então, ao invés de medir efetivamente a taxa de frames por segundo, iremos calcular a quantidade de *janky frames* presentes em uma execução.

Devido às diferenças entre os frameworks, as informações de frames por segundo precisam ser recuperadas de formas diferentes. Para facilitar as análises posteriores, as informações coletadas dos aplicativos de ambos os frameworks serão salvas de forma uniformizada. As informações de frames por segundo serão salvas em um arquivo *json* contendo um *map*, em que cada *key* representa o tempo, em milissegundos, levado para a renderização do *frame* e o *value* representa a quantidade de *frames* que levaram esse tempo para serem renderizados. Dessa forma, será possível analisar todas as execuções de forma bastante simples e contar a quantidade de *janky frames* em cada uma delas. Além disso,

também seria possível fazer outras análises em cima destes resultados, visto que estamos salvando o tempo de renderização de todos os *frames*.

6.3.1 React Native

No React Native é possível utilizar uma ferramenta nativa do Android para obter informações sobre o tempo de renderização dos frames de um aplicativo. O *dumpsys*, uma ferramenta oferecida pelo ADB, executa diretamente no dispositivo Android e provê informações diversas sobre os serviços do sistema. Ao utilizar essa ferramenta com o comando *gfxinfo*, são retornadas informações de desempenho relacionadas aos frames renderizados durante um determinado período de tempo. A partir da saída desse comando, é possível visualizar, além de outras informações pouco relevantes para este estudo, uma distribuição de frequência pontual do tempo de renderização dos *frames*.

Para executar o comando em questão e obter sua saída em forma de texto a partir do *script* em Python utilizado para automatizar os experimentos, basta fazer uma chamada a um simples comando da biblioteca `subprocess`. Então, para coletar as informações necessárias, o *script* em Python efetua uma chamada a essa biblioteca com o comando `adb shell dumpsys gfxinfo <PACKAGE_NAME>` e salva a distribuição de frequência em forma de JSON, para ser analisado posteriormente. Antes de cada medição, é necessário executar o comando `adb shell dumpsys gfxinfo <PACKAGE_NAME> reset` para limpar os dados armazenados anteriormente e não contabilizar frames já contabilizados por outras medições.

Dentre outras informações que não são relevantes para este estudo, o comando citado acima apresenta uma saída semelhante a seguinte:

```
HISTOGRAM:  5ms=1  6ms=3  7ms=13  8ms=6  9ms=10  10ms=4  11ms=0  12ms=1
13ms=0 14ms=0 15ms=0 16ms=0 17ms=0 18ms=0 19ms=0 20ms=0 21ms=0 22ms=0
23ms=0 24ms=0 25ms=0 26ms=0 27ms=0 28ms=0 29ms=0 30ms=0 31ms=0 32ms=0
34ms=0 36ms=0 38ms=0 40ms=0 42ms=1 44ms=0 46ms=0 48ms=0 53ms=0 57ms=0
61ms=0 65ms=0 69ms=0 73ms=0 ...
```

A seção definida como *HISTOGRAM* contém a distribuição de frequência pontual do tempo de renderização do *frame*, que é justamente a informação que queremos salvar. O *script* deve, então, apenas parsear essas informações e salvá-las de maneira uniformizada.

6.3.2 Flutter

O Flutter, como citado no capítulo 4, evita a camada nativa do dispositivo e utiliza seu próprio motor gráfico para renderizar o aplicativo na tela. Dessa forma, infelizmente, a ferramenta citada acima não é capaz de coletar, corretamente, informações sobre os frames renderizados por um aplicativo desenvolvido em Flutter. Ao executar o comando, apenas um único frame aparece na saída do comando, pois o Android não tem conhecimento do que está sendo renderizado. Durante toda a execução do aplicativo, nenhum outro frame é contabilizado, apenas o frame inicial.

Para obter as informações de tempo dos frames renderizados em um aplicativo Flutter, precisamos, então, utilizar outra ferramenta. Sendo assim, optamos por utilizar um *profiler* próprio do framework, chamado *Dart Devtools.* Essa ferramenta é executada diretamente no navegador *web* do computador e fornece várias informações de *debugging* e *profiling* do aplicativo em Flutter. A partir da execução do aplicativo com o comando `flutter run`, é disponibilizada a url com o caminho para o *Dart Devtools* vinculado ao aplicativo.

Dessa vez, não é possível coletar as informações a partir de uma simples chamada do terminal. Para automatizar os experimentos com o *script* em Python, é necessário utilizar o *Selenium*[38] — um conjunto de ferramentas para testar aplicações *web* pelo *browser* de forma automatizada. Para coletar as informações desejadas, então, o *script* cria uma instância do navegador, navega até a url do *Dart Devtools* e simula cliques nos botões. No início do experimento é efetuado um clique no botão de resetar as informações sobre os frames renderizados. Ao final do experimento, é efetuado um clique no botão de exportar um JSON com todas as informações disponibilizadas pela ferramenta. Por fim, o *script* lê o conteúdo desse JSON, calcula a distribuição de frequência do tempo de renderização dos *frames* e a salva, em outro JSON.

Infelizmente, não há uma forma documentada de exportar, via código, os dados da ferramenta. Dessa forma, os experimentos automatizados tendem a ser mais demorados, pois necessitam simular a interação com um navegador web.

Dentre outras informações que não são relevantes para este estudo, o comando citado acima apresenta uma saída semelhante a seguinte:

```
"performance": {
  "selectedFrameId": null,
  "flutterFrames": [
```

[38] Disponível em: selenium.dev/. Acesso em 11/10/2022.

```
{
  "number": 2,
  "startTime": 46674851478,
  "elapsed": 84454,
  "build": 322,
  "raster": 33264,
  "vsyncOverhead": 37038
},
{
  "number": 3,
  "startTime": 46698608258,
  "elapsed": 49139,
  "build": 6233,
  "raster": 34671,
  "vsyncOverhead": 1171
},
...
```


A lista *flutterFrames*, dentro do objeto de *performance*, contém informações de tempo de renderização de todos os *frames*, sequencialmente, renderizados durante a execução do experimento. Entretanto, a ferramenta utilizada para coletar os dados dos aplicativos desenvolvidos em React Native não guarda informações sobre a ordem dos frames. Como o intuito deste estudo é comparar o desempenho de dois *frameworks*, não faz sentido efetuar análises sobre uma métrica que está presente em apenas um *framework*. Sendo assim, apenas a distribuição de frequência do tempo de renderização dos *frames* será utilizada. Diferentemente da ferramenta anterior, neste caso o tempo de duração de um frame não está claramente definido. Então, é necessário definir uma abordagem para efetuar o cálculo do tempo de duração total da renderização de um frame.

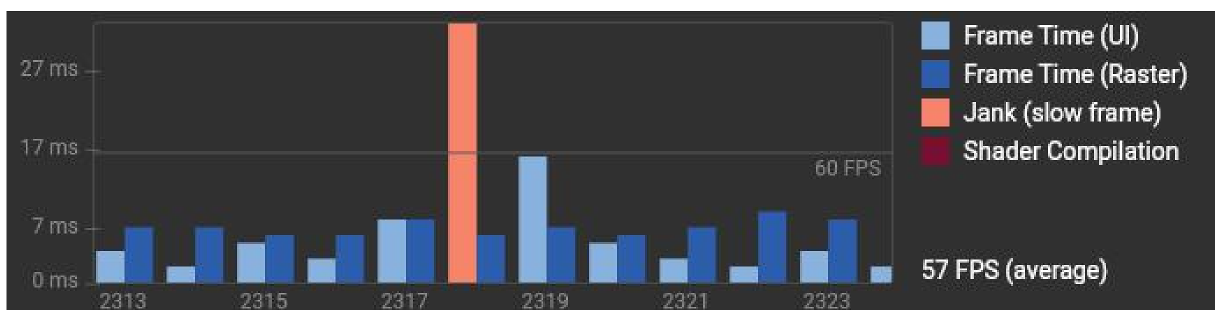
Primeiramente, é importante entender o conteúdo do JSON. Os campos *build* e *raster* representam a duração, em microssegundos, do trabalho efetuado, respectivamente, nas threads de *UI* e *Raster*. Como visto na seção 4.1, um *frame* é construído pela thread de UI e renderizado na tela pela thread de *raster*. Existem quatro abordagens possíveis para determinar se um *frame* foi renderizado no tempo esperado (FLUTTER, 2020). São elas:

A. frame UI time + frame Raster time < 16.66ms
B. max(frame UI time, frame Raster time) < 16.66 ms
C. max(frame UI time, frame Raster time) < 8.33 ms
D. end frame Raster time - start frame UI time < 16.66 ms

No momento em que o documento com essas abordagens foi escrito, o *Dart Devtools* seguia a abordagem A. Porém, atualmente, a abordagem seguida é outra, pois, como podemos visualizar na figura 15, o frame 2319 não foi marcado como *jank*, apesar de sua soma entre UI e Raster ultrapassar o limite de 17 milissegundos. Como existem duas threads diferentes para construção e renderização, cada *thread* tem um tempo máximo de 16 milissegundos para efetuar sua tarefa (FLUTTER, 2022f).

Tomando como base a atual implementação e encorajados pela documentação oficial do Flutter, optamos por seguir a abordagem B. Dessa forma, o tempo total de um frame será o tempo máximo entre os tempos definidos nos campos *build* e *raster* presentes no JSON.

Figura 15 – Visualização da ferramenta Dart Devtools



6.4 ANALISANDO ESPAÇO EM BRANCO NAS LISTAS

O React Native possui um problema de desempenho conhecido ao lidar com listas. Quando uma *VirtualizedList* — componente responsável por renderizar itens de uma lista sob demanda — não consegue renderizar seus itens de forma rápida o suficiente durante um *scroll*, o usuário pode se deparar com espaços em brancos nos lugares dos itens faltantes (REACT NATIVE, 2022f).

Nos resultados apresentados por Wu (2018, p. 21), os espaços em branco na lista são citados como um problema recorrente. Wu, porém, não se propõe a utilizar uma métrica para estimar a quantidade de espaço em branco durante o experimento e apenas apresenta uma captura de tela em que podemos visualizar a lista toda em branco.

Ao executar os experimentos, podemos notar que, no aplicativo 5, esse problema é bastante recorrente. Obviamente, espaços em branco durante a navegação pela lista podem afetar negativamente a experiência do usuário. Para comparar a quantidade de itens não

renderizados durante a execução dos experimentos em ambos os frameworks, desenvolvemos alguns *scripts* em Python para gravar a tela e analisar cada um dos frames do vídeo.

Para iniciar, finalizar e extrair a gravação do celular para o computador, utilizamos o ADB. Inicialmente, a gravação é iniciada com o comando `adb shell screenrecord /sdcard/video.mp4` e, ao final do experimento, encerrada com `adb shell pkill -2 screenrecord`. Isso irá criar o vídeo no dispositivo no diretório indicado. Para extrair o vídeo para o computador, utilizamos o comando `adb pull /sdcard/video.mp4 <file_name>`, indicando o diretório, no computador, em que queremos salvar o vídeo.

O próximo passo, então, é extrair cada frame desse vídeo para que sejam analisados posteriormente. Para isso, utilizamos a ferramenta *ffmpeg*[39]. Importante ressaltar que os frames renderizados pelo *framework* não possuem uma equivalência de 1 para 1 com os frames extraídos pelo vídeo. Enquanto o *framework* se propõe a rodar o aplicativo com uma taxa de 60 frames por segundo, o vídeo é gravado em uma taxa aproximada de 30 frames por segundo. Dessa forma, alguns frames são perdidos. Entretanto, isso não é um problema, pois o objetivo é estimar a quantidade de espaço em branco durante a execução do experimento, e não analisar especificamente cada um dos frames renderizados. Os frames são, então, extraídos do vídeo com o comando `ffmpeg -i {video} {output_path}/frame%03d.png`. Com esse comando, os frames serão salvos no formato de imagem *png*, identificados sequencialmente com três algarismos no fim do nome do arquivo.

Após extrair os frames do vídeo gravado durante a execução dos experimentos, é necessário um último tratamento antes de iniciar as análises. Não é possível garantir qual o momento exato em que a gravação é iniciada. Também não é possível saber qual o momento exato, de forma automatizada, em que o vídeo deve ser interrompido, pois a lista continua rolando por um tempo após o fim da interação do usuário. Então, os primeiros frames do vídeo, assim como os últimos, são frames repetidos e representam momentos que não fazem parte da execução do experimento. Dessa forma, eles servem apenas para aumentar o número total de frames, reduzindo a importância dos demais, e não devem ser contabilizados para análise. A fim de remover os frames repetidos do início e final do vídeo, executamos mais um script em Python, com o uso da biblioteca de análise de imagens *opencv-python*[40]. O script é responsável por comparar o primeiro frame com os seus subsequentes e deletar todos que sejam semelhantes. A comparação é encerrada ao encontrar um frame que não apresente

[39] Disponível em: ffmpeg.org. Acesso em: 19/08/2022.
[40] Disponível em: pypi.org/project/opencv-python. Acesso em: 28/07/2022.

semelhança com o primeiro. Analogamente, os frames repetidos do final do vídeo também são deletados.

Para analisar o espaço em branco na tela, algumas precauções devem ser tomadas. Pixels brancos de textos, imagens ou ícones da barra superior do Android podem influenciar no resultado final. É importante, então, garantir que não há nenhum texto, imagem ou ícone branco em nenhum componente do aplicativo, como também em nenhum item da lista. O texto da *toolbar*, assim como a cor de fundo dos itens da lista, eram originalmente brancos e passaram a ser azul claro. A barra superior do Android, que era preta com ícones brancos, passou a ser totalmente preta. Por fim, com um script em Python e, novamente, o auxílio da biblioteca *opencv-python*, diminuímos o brilho de todas as imagens presentes na lista, eliminando todos os pixels brancos. Dessa forma, podemos garantir que todo pixel branco presente nos frames analisados, representam um item faltante da lista.

Como o objetivo é estimar a quantidade de itens faltantes a partir da porcentagem de espaço em branco na tela, precisamos calcular o tamanho que cada item ocupa na lista, assim como o tamanho que a lista inteira ocupa na tela. Para isso, simulamos um aplicativo com todo o seu conteúdo em branco e outro com um item de lista faltante. Finalmente, para calcular a porcentagem de espaço em branco, desenvolvemos um script em Python, baseado em um código disponível publicamente[41], fazendo uso novamente da biblioteca *opencv-python*. Importante ressaltar que a cor de fundo do aplicativo não é exatamente branco, ou seja, o valor exato do pixel não é `#FFF`. Primeiramente, é necessário converter todos os pixels próximos a branco, com uma pequena margem de erro, para pixels totalmente brancos. Enquanto isso, o resto da imagem se torna totalmente preta. Dessa forma, agora basta contabilizar a quantidade de pixels brancos na imagem gerada. Os resultados do algoritmo aplicado nos aplicativos citados acima, para calcular o tamanho de um item e da lista inteira, nos *frameworks* Flutter e React Native, podem ser vistos nas figuras 16 e 17, respectivamente. As porcentagens calculadas pelo algoritmo podem ser vistas na tabela 2.

[41] Disponível em: stackoverflow.com/a/66757536/11313788. Acesso em: 22/07/2022.

Figura 16 – Tamanho de um item e da lista em Flutter

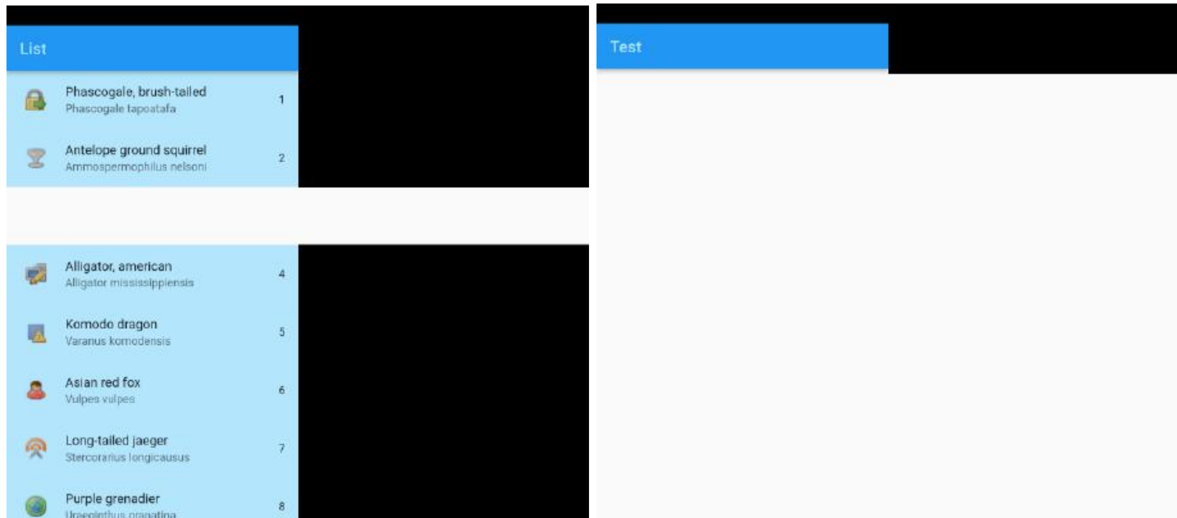


Figura 17 – Tamanho de um item e da lista em React Native

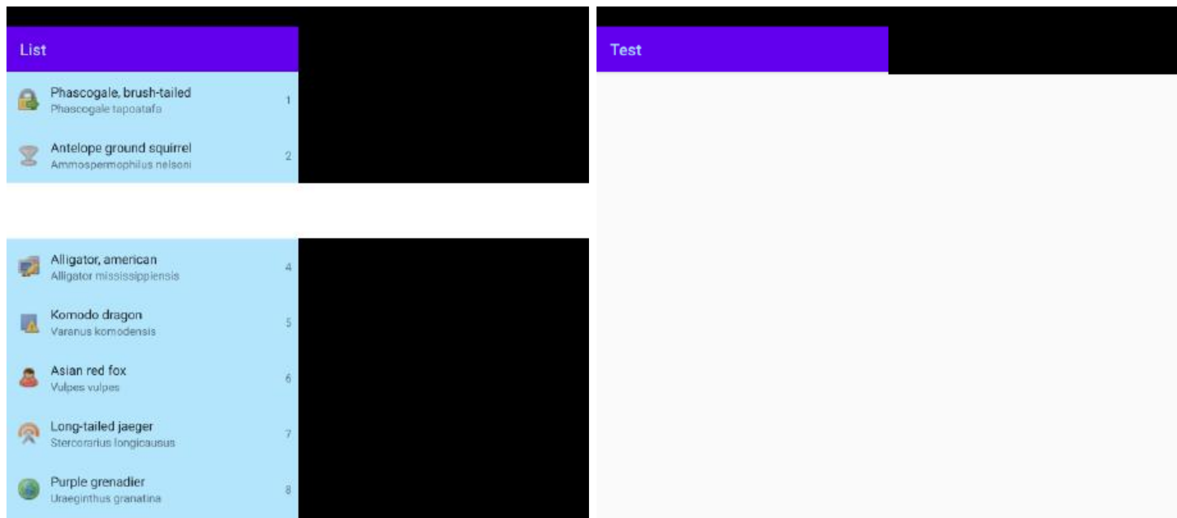


Tabela 2 – Tamanho de um item e da lista nos frameworks

FRAMEWORK	TAMANHO DE UM ITEM EM RELAÇÃO À TELA	TAMANHO DA LISTA EM RELAÇÃO À TELA
Flutter	11,11%	86,46%
React Native	10,59%	86,98%

Um item de lista no aplicativo em Flutter representa 11,11% de todo o frame, enquanto no aplicativo em React Native representa 10,59%. A pequena diferença entre os frameworks se deve ao fato de suas diferentes implementações no *design* de um item de lista. Enquanto isso, a lista toda ocupa 86,46% do frame em Flutter e 86,98% em React Native. Dessa vez, a diferença de porcentagem acontece devido às diferentes implementações de *design* da componente *toolbar* nos *frameworks*. A *toolbar* do Flutter possui um sombreado um pouco

maior, o que escurece um pouco os pixels localizados abaixo. Dessa forma, o algoritmo não é capaz de contabilizar esses pixels como brancos, gerando uma pequena diferença. Novamente, as diferenças são mínimas e não impactam no resultado final.

Sabendo a porcentagem da tela que um item da lista ocupa e a porcentagem de toda a lista, é trivial estimar, a partir da porcentagem de branco presente em um frame, quantos itens estão faltando. Além disso, a porcentagem de branco em relação a tela toda pode parecer menos prejudicial à experiência do usuário do que realmente é. Por exemplo, supondo que um frame contenha exatamente 43,49% de pixels brancos e a lista inteira corresponda a 86,98% do frame, como é o caso do aplicativo em React Native, então, na verdade, 50% da lista está em branco.

7 RESULTADOS

Durante os experimentos, todos os aplicativos foram executados em um dispositivo físico com o sistema operacional Android. É importante não utilizar emuladores para executar os experimentos, pois eles utilizam o hardware do computador e podem apresentar resultados destoantes com a realidade. O dispositivo utilizado foi um Xiaomi Mi A1[42] com 4GB de RAM e um processador *Snapdragon* 625 com oito núcleos.

Para garantir uma alta confiabilidade nos resultados, os experimentos foram executados várias vezes. Cada aplicativo foi executado 50 vezes. Durante os experimentos, não temos total controle sobre o sistema operacional e não é possível garantir que não há outros processos em execução utilizando recursos do dispositivo, afetando o desempenho do aplicativo. Dessa forma, para minimizar possíveis interferências, os experimentos foram executados em diferentes dias. Para minimizar processos em segundo plano, o modo avião foi ativado e as notificações foram desativadas.

Os resultados brutos obtidos, utilizados para fazer as análises apresentadas posteriormente, também podem ser encontrados no repositório deste trabalho.

7.1 QUANTIDADE DE JANKY FRAMES

Nesta seção iremos discutir os resultados obtidos a partir da medição da quantidade de *janky frames* encontrados durante a execução dos experimentos. Apresentaremos a média, a mediana e o desvio padrão da quantidade de *janky frames* (em porcentagem) encontrados nas 50 execuções de cada um dos 5 aplicativos. É importante medir a quantidade média de *janky frames* para analisar o desempenho do aplicativo de modo geral. Enquanto isso, o desvio padrão é uma boa métrica para verificar instabilidades.

7.1.1 Aplicativo 1 - *Stopwatch*

O aplicativo 1 apresenta apenas um contador, incrementado automaticamente. Em uma execução limpa de *janky frames* o contador é incrementado 60 vezes em um segundo. A presença *janky frames* ocasiona uma demora na atualização do contador, fazendo com que ele seja incrementado menos vezes, demorando mais para chegar em determinado número.

[42] Disponível em: gsmarena.com/xiaomi_mi_a1_(mi_5x)-8776.php. Acesso em: 04/08/2022.

Interessante ressaltar que esse contador não é de fato um *stopwatch*, cuja integridade não pode depender de *janky frames*.

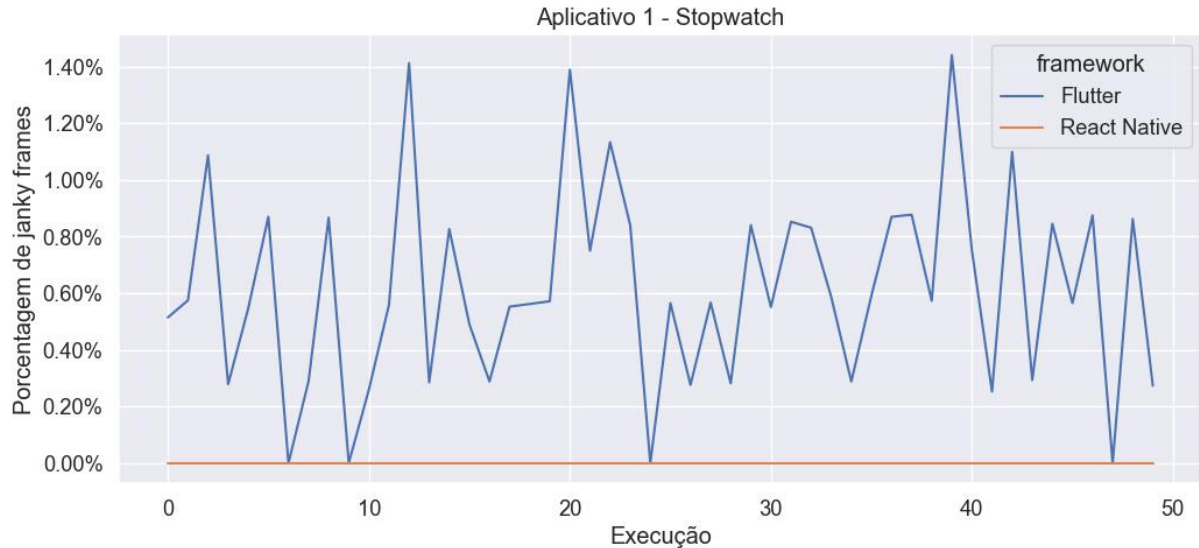
Como podemos ver na tabela 3, o React Native não apresentou nenhum *janky frame* em nenhuma das 50 execuções. O Flutter apresentou alguns *janky frames*, porém com média e mediana extremamente baixas e certa consistência, sem grandes picos.

Tabela 3 – Quantidade de *janky frames* no aplicativo 1

FRAMEWORK	MÉDIA DE JANKY FRAMES (EM %)	MEDIANA DE JANKY FRAMES (EM %)	DESVIO PADRÃO (EM %)
Flutter	0,61	0,57	0,35
React Native	0	0	0

Analisando cada uma das execuções, como pode ser visto no gráfico apresentado pela figura 18, é possível confirmar que a porcentagem de *janky frames* variou pouco entre as execuções no Flutter. Algumas execuções chegaram a apresentar nenhum *janky frame*, enquanto outras alcançaram um pico de aproximadamente 1,4%.

Figura 18 – Porcentagem de *janky frames* por execução do aplicativo 1



7.1.2 Aplicativo 2 - *Multi Stopwatch*

O aplicativo 2 também é composto apenas por contadores. Dessa vez, são apresentados seis contadores simultâneos. Assim como no aplicativo 1, a presença de *janky frames* impactará na quantidade de vezes que os contadores serão incrementados.

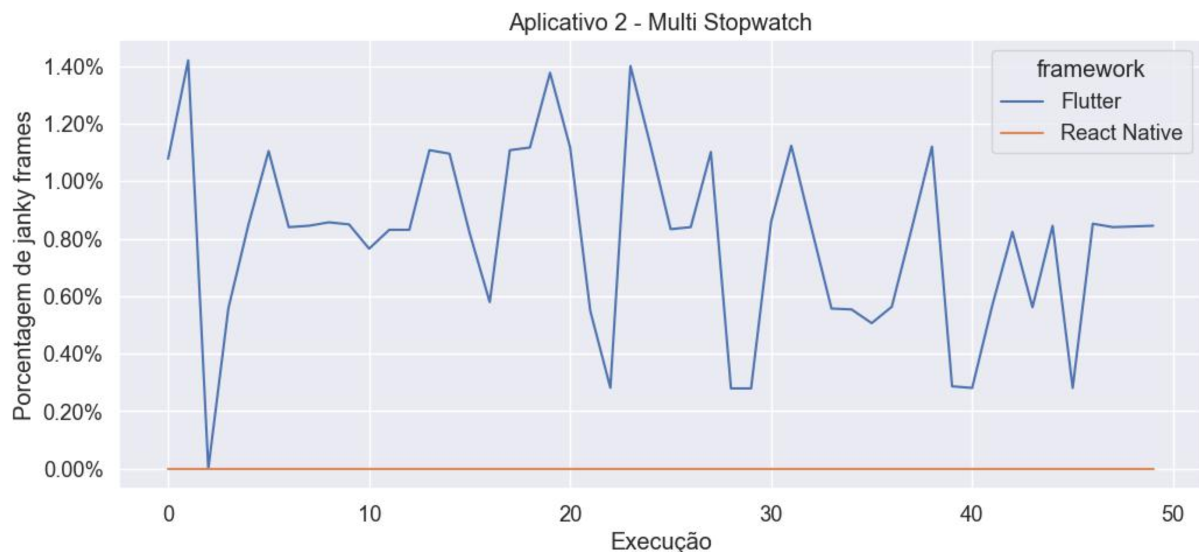
Novamente, como podemos ver na tabela 4, o React Native não apresentou *janky frames*. Enquanto isso, o Flutter apresentou uma média um pouco pior que o aplicativo anterior. Já era esperado que os resultados fossem iguais ou piores, pois a complexidade do aplicativo aumentou.

Tabela 4 – Quantidade de *janky frames* no aplicativo 2

FRAMEWORK	MÉDIA DE JANKY FRAMES (EM %)	MEDIANA DE JANKY FRAMES (EM %)	DESVIO PADRÃO (EM %)
Flutter	0,8	0,84	0,32
React Native	0	0	0

Analisando o gráfico que representa o Flutter na figura 19, podemos notar algumas semelhanças com o gráfico do aplicativo 1. Apesar da média e mediana serem levemente maiores, os picos continuam acontecendo por volta de 1,4%. Dessa vez, apenas uma execução apresentou nenhum *janky frame*.

Figura 19 – Porcentagem de *janky frames* por execução do aplicativo 2



7.1.3 Aplicativo 3 - *Counter*

O aplicativo 3 é o primeiro aplicativo a lidar com interação do usuário. Ao tocar no botão, o contador é incrementado em uma unidade. Importante ressaltar que a incrementação do número não é a única atualização efetuada a partir do toque no botão. O botão, ao ser tocado, emite uma animação para que o usuário receba um *feedback* de que o botão foi, de fato, tocado. A presença de *janky frames* pode causar animações mais lentas ou travadas,

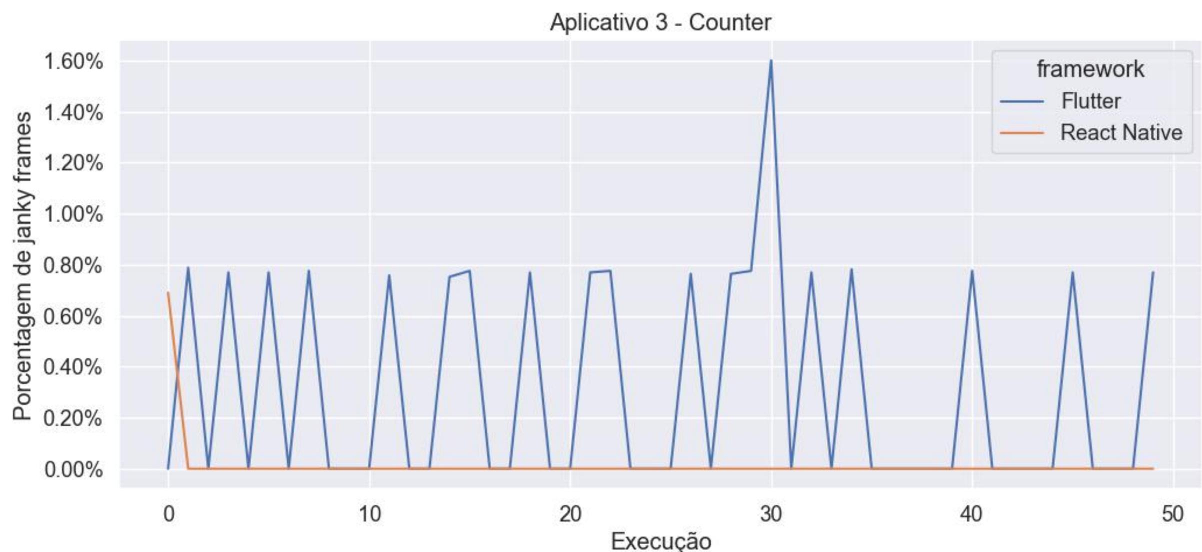
impactando na sua suavidez. Por ser um aplicativo bastante simples, espera-se que a quantidade de *janky frames* seja extremamente baixa, a ponto de não afetar negativamente a experiência do usuário.

Dessa vez, como mostra a tabela 5, o React Native saiu do zero e apresentou alguns *janky frames*, porém com uma média extremamente baixa — próxima de zero — e mediana igual a zero. O Flutter apresentou resultados melhores que os dois primeiros aplicativos, com uma mediana também igual a zero, mas ainda obteve resultado pior que o React Native, com média mais alta e mais instabilidade.

Tabela 5 – Quantidade de *janky frames* no aplicativo 3

FRAMEWORK	MÉDIA DE JANKY FRAMES (EM %)	MEDIANA DE JANKY FRAMES (EM %)	DESVIO PADRÃO (EM %)
Flutter	0,31	0	0,41
React Native	0,01	0	0,09

Figura 20 – Porcentagem de *janky frames* por execução do aplicativo 3



Apesar de o React Native possuir uma média maior que zero, analisando o gráfico da figura 20, podemos perceber que apenas uma execução apresentou *janky frames*. Por outro lado, o Flutter apresentou uma porcentagem aproximada de 0,8% de *janky frames* em aproximadamente metade das execuções. Enquanto isso, a outra metade apresentou 0%, com apenas uma exceção, apresentando cerca de 1,6%. A mediana igual a zero significa que mais da metade das execuções apresentou 0%. Analisando os resultados mais profundamente, é possível ver que o Flutter renderizou uma média de 129,5 frames por execução, com um

desvio padrão de apenas 1,5. Dessa forma, as porcentagens 0,8% e 1,6% representam, respectivamente, apenas 1 e 2 frames do total.

7.1.4 Aplicativo 4 - *Navigations*

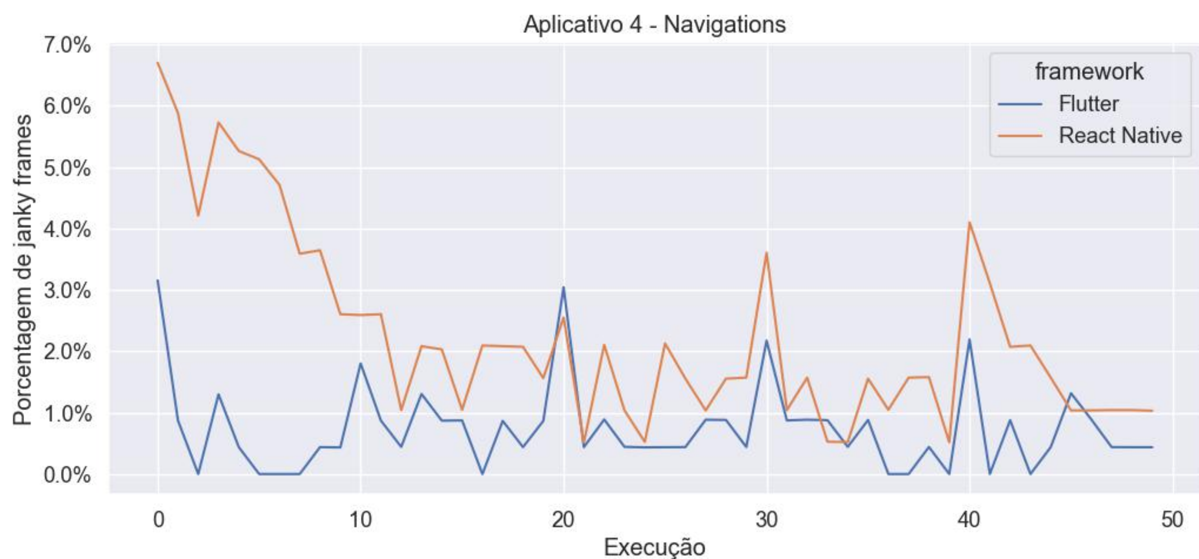
O aplicativo 4 é consideravelmente mais complexo que os anteriores. Além de lidar com toques do usuário, este aplicativo também é responsável por efetuar navegação entre telas. Então, além da animação do toque do botão em si, dessa vez também ocorrem animações de transição de telas. Assim como no aplicativo anterior, a presença de *janky frames* pode causar problemas nas animações, impactando na experiência do usuário. Por ser um aplicativo mais complexo, apresentando múltiplas animações, é natural que a quantidade de *janky frames* seja maior que no aplicativo anterior.

Com os resultados da tabela 6, podemos ver que o React Native teve mais dificuldade para lidar com o aumento de complexidade e apresentou números piores que o Flutter.

Tabela 6 – Quantidade de *janky frames* no aplicativo 4

FRAMEWORK	MÉDIA DE JANKY FRAMES (EM %)	MEDIANA DE JANKY FRAMES (EM %)	DESVIO PADRÃO (EM %)
Flutter	0,74	0,44	0,7
React Native	2,26	1,8	1,55

Figura 21 – Porcentagem de *janky frames* por execução do aplicativo 4



A partir do gráfico representado na figura 21, podemos confirmar que, na maioria das execuções, o Flutter obteve resultados melhores. Ambos frameworks apresentaram alguns picos, porém o React Native chegou a ultrapassar 4% algumas vezes, enquanto o pico do Flutter foi por volta de 3%. Além disso, o Flutter foi responsável pelas execuções com menores *janky frames*, chegando a apresentar nenhum *janky frame* em algumas execuções, o que, dessa vez, não ocorreu no React Native.

7.1.5 Aplicativo 5 - *List*

O aplicativo 5 lida com a interação do usuário com uma lista. Ao rolar uma lista, os itens presentes na tela devem ser movidos para cima ou para baixo, dando lugar a novos itens. A presença de *janky frames* pode, novamente, causar problemas nas animações de rolagem da lista. Uma alta quantidade de *janky frames* pode impactar negativamente a experiência do usuário, pois, dependendo da velocidade da rolagem, o usuário pode ter a impressão de que a lista está “travando”.

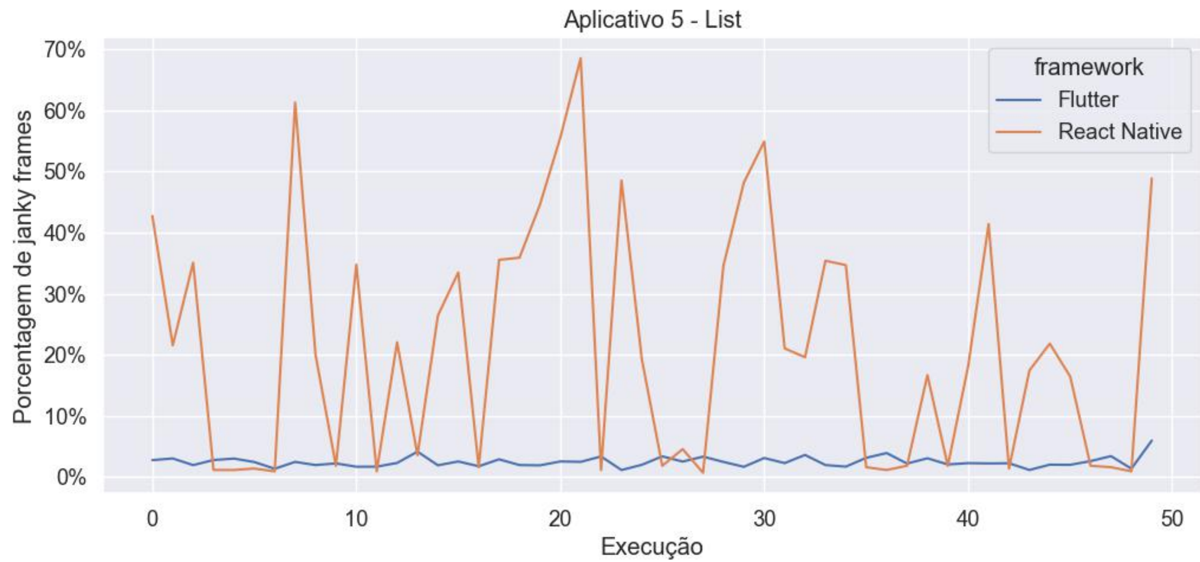
Como podemos ver na tabela 7, o último aplicativo é o que possui maiores problemas de desempenho. Apesar de ambos os frameworks apresentarem seus piores resultados nesse aplicativo, o React Native alcançou uma média drasticamente alta de 21,34%. Além disso, o desvio padrão também é extremamente alto, mostrando que o aplicativo apresentou boas e péssimas execuções. O Flutter foi mais consistente, apresentando um desvio padrão menor que 1% e uma média relativamente baixa.

Tabela 7 – Quantidade de *janky frames* no aplicativo 5

FRAMEWORK	MÉDIA DE JANKY FRAMES (EM %)	MEDIANA DE JANKY FRAMES (EM %)	DESVIO PADRÃO (EM %)
Flutter	2,47	2,28	0,85
React Native	21,34	19,41	19,74

Como mostra a figura 22, o React Native oscilou bastante, chegando a apresentar uma porcentagem perto de zero em alguns momentos e alcançando quase 70% em outros. Apesar de poucas execuções possuírem uma porcentagem tão elevada, podemos notar que muitas apresentam uma porcentagem entre 20% e 40%, o que é extremamente alto e, certamente, impacta na experiência do usuário. O Flutter, além de apresentar maior regularidade, não ultrapassa 10% em nenhuma execução, alcançando um desempenho melhor nessa situação.

Figura 22 – Porcentagem de *janky frames* por execução do aplicativo 5



7.2 ESPAÇO EM BRANCO NAS LISTAS

Nesta seção iremos discutir os resultados obtidos a partir da medição da quantidade de espaço em branco encontrado no aplicativo 5 (*List*) durante a execução dos experimentos. O experimento consiste em 6 rolagens de tela, como se o usuário estivesse rolando a tela de cima para baixo seguidamente. A primeira rolagem é a maior de todas e antecede outras 5 rolagens de tamanhos iguais. Todas as 6 rolagens são executadas uma após a outra. Apesar de todas as interações serem simuladas via ADB, alguns atrasos de milissegundos podem ocorrer entre a execução de dois comandos. Dessa forma, não é possível garantir que todas as execuções terão, no geral, exatamente a mesma rolagem de tela.

Foram analisadas as gravações de telas de 50 execuções. O gráfico representado pelas figuras 23 e 24 mostram a porcentagem de espaço em branco da lista de cada um dos frames analisados da gravação. A linha mais escura representa a quantidade média de espaço em branco no frame de número **x** nas 50 execuções, enquanto a região mais clara delimita o intervalo de 95% de confiança.

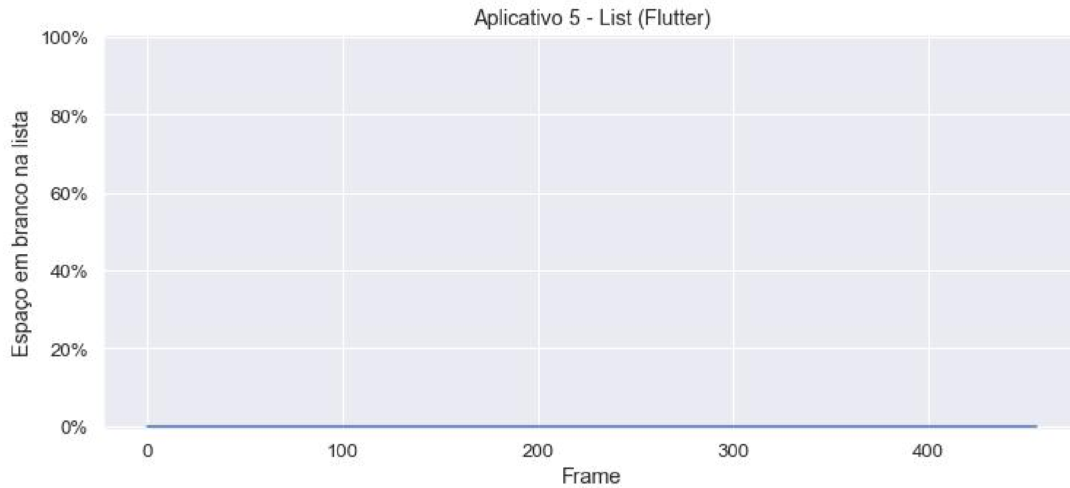
7.2.1 Flutter

Diferentemente do React Native, que possui uma seção em sua documentação oficial exclusivamente sobre os problemas de desempenhos presentes em grandes listas, o Flutter não

apresentou nenhum espaço em branco durante a execução dos experimentos. Os itens são, assim como no React Native, criados sob demanda, então apenas os itens visíveis são de fato construídos para serem renderizados na tela (FLUTTER, 2022g).

O gráfico representado pela figura 23 mostra a porcentagem de espaço em branco na lista, em média, em cada um dos frames renderizados.

Figura 23 – Porcentagem de espaço em branco por frame no aplicativo 5 em Flutter



Na tabela 8, podemos visualizar, em média, quantos frames (em porcentagem) de uma execução possuem a lista com uma porcentagem de branco maior que o valor determinado.

Tabela 8 – Quantidade de espaço em branco na lista do aplicativo 5 em Flutter

PORCENTAGEM DE BRANCO (MAIOR QUE)	MÉDIA DE FRAMES (EM %)	DESVIO PADRÃO (EM %)
0	0	0
50	0	0
90	0	0
99	0	0

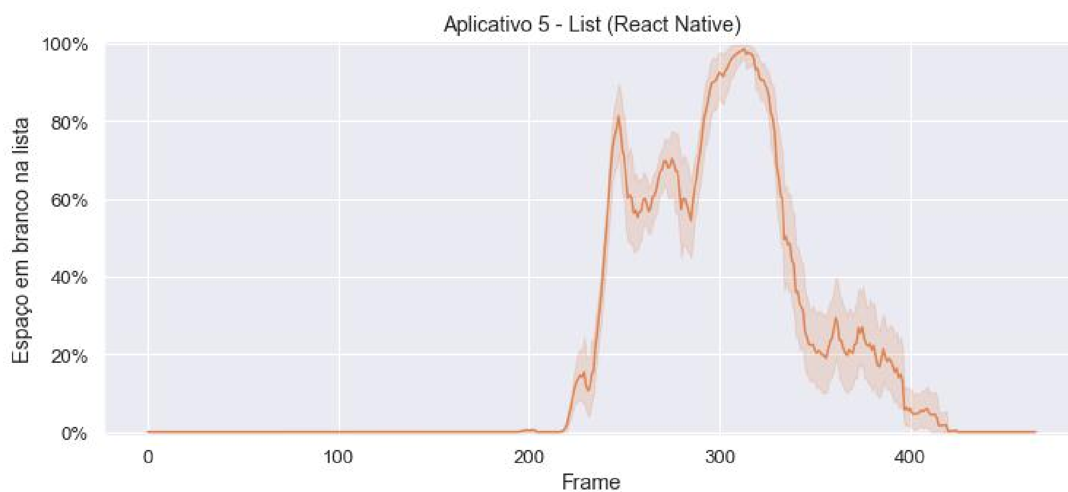
Apesar da criação de itens sob demanda, o Flutter consegue gerenciar muito bem a criação de novos itens a serem renderizados enquanto o usuário realiza a rolagem na tela. Durante a execução dos experimentos, todos os itens foram renderizados corretamente, sem nenhum tipo de atraso, não apresentando espaços em branco.

7.2.2 React Native

Como esperado, o React Native apresentou problemas recorrentes de renderização de itens. Muitas vezes, vários itens não foram renderizados a tempo, causando a aparição de espaços em branco na lista durante a rolagem de tela.

Podemos perceber, analisando o gráfico da figura 24, que o resultado é bastante consistente. Os espaços em branco começam a aparecer, de forma sutil, por volta do frame 200, no fim da terceira rolagem. Durante a quarta rolagem, por volta do frame 240, o espaço em branco aumenta consideravelmente, extremamente rápido, alcançando cerca de 90% da lista. Durante a quinta e a sexta rolagem, podemos observar outros dois picos, por volta dos frames 270 e 310, respectivamente. Nesse segundo pico, a lista chega a ficar quase completamente branca, com nenhum item sendo renderizado. A lista continua com bastante espaço em branco até o fim da sexta rolagem, quando a interação com o usuário acaba e nenhum novo item precisa ser renderizado. Nesse momento, conforme a rolagem vai chegando ao fim, o espaço em branco vai diminuindo, até que nenhum item esteja faltando na tela.

Figura 24 – Porcentagem de espaço em branco por frame no aplicativo 5



Na tabela 9, podemos visualizar, em média, quantos frames (em porcentagem) de uma execução possuem a lista com uma porcentagem de branco maior que o valor determinado.

Tabela 9 – Quantidade de espaço em branco na lista do aplicativo 5 (React Native)

PORCENTAGEM DE BRANCO (MAIOR QUE)	MÉDIA DE FRAMES (EM %)	DESVIO PADRÃO (EM %)
0	28,13	5,19
50	20,52	4,42
90	13,22	4,15
99	11,87	4,03

Em média, 28,13% dos frames apresentam pelo menos um item faltante na lista. Mais de 20% apresentam mais da metade da lista em branco. Outro fato que chama bastante atenção é que, em média, 11,87% — com um desvio padrão de apenas 4,03%— dos frames possuem mais de 99% da lista em branco. Isso significa que mais de um décimo dos frames apresenta a lista praticamente toda em branco, uma quantidade extremamente alta.

8 CONCLUSÃO

O objetivo deste trabalho foi realizar uma análise comparativa do desempenho dos aplicativos desenvolvidos, usando os frameworks Flutter e React Native. De forma similar a Wu (2018) e Jagiello (2019), desenvolvemos aplicativos iguais com os dois frameworks e utilizamos a quantidade de *janky frames* como métrica principal para compará-los. Além disso, efetuamos um experimento extra para analisar o desempenho dos frameworks ao lidar com grandes listas, utilizando como métrica a quantidade de espaço em branco visível na tela durante uma rolagem.

Por mais que Wu e Jagiello tenham encontrado resultados divergentes — Jagiello obteve resultados melhores no React Native, ao contrário de Wu —, os dois trabalhos confirmam que Flutter e React Native apresentam resultados bastante parecidos. A divergência no resultado final pode ser explicada pela diferença da complexidade dos aplicativos.

Os resultados apresentados confirmam que, majoritariamente, os dois frameworks possuem desempenho similar. Os aplicativos 1, 2 e 3 apresentaram resultados levemente favoráveis ao React Native. Enquanto o React Native obteve um ótimo desempenho nesses casos, sem nenhum *janky frame*, o Flutter não se distanciou muito, apresentando taxas de *janky frame* sempre abaixo de 2%. Por outro lado, o aplicativo 4 favoreceu, com uma diferença sútil, o Flutter. Os dois frameworks apresentaram uma queda considerável, mas, dessa vez, o React Native apresentou uma média de *janky frames* bem maior que o Flutter, assim como um um desvio padrão mais alto.

A contribuição principal deste estudo foi a análise comparativa sobre como os frameworks lidam com listas. Enquanto o Flutter apresentou bons resultados, o React Native confirmou a premissa de não lidar bem com esse recurso. Com uma taxa média bem elevada de *janky frames* — além de um desvio padrão significativo — e uma quantidade alta de espaço em branco durante o *scroll* da lista, o React Native deixou bastante a desejar.

Por fim, salvo o aplicativo 5 (*List*), nenhuma grande diferença significativa foi encontrada em relação ao desempenho dos frameworks.

O código de todos os aplicativos desenvolvidos, assim como os *scripts* utilizados para a execução dos experimentos está disponível no repositório deste trabalho. Assim, os aplicativos podem ser executados em outros dispositivos facilmente, e seus códigos podem ser modificados, seja para corrigir problemas de desempenho ou pequenas atualizações.

8.1 TRABALHOS FUTUROS

Uma das limitações do trabalho foi a impossibilidade de utilizar a mesma ferramenta para captar a taxa de FPS dos aplicativos. Seria interessante utilizar a mesma ferramenta para ambos os frameworks a fim de minimizar possíveis diferenças decorrentes da ferramenta utilizada.

Outro ponto importante que pode ser revisado é o funcionamento da lista no React Native. Talvez seja possível alterar alguns parâmetros do componente *FlatList* a fim de obter melhores resultados. Modificações nos componentes utilizados, como por exemplo passar a utilizar uma lista desenvolvida pela comunidade, também pode render experimentos interessantes.

Como a arquitetura do React Native está em processo de mudança, o código pode ser constantemente atualizado para estar de acordo com a última versão do framework. Dessa forma, problemas graves de desempenho podem acabar sendo corrigidos, assim como pequenas melhorias podem surgir.

Para aumentar a abrangência dos experimentos, outras métricas podem ser utilizadas. O uso de CPU e memória, tempo de resposta e tamanho da aplicação, assim como Hansson e Vidhall (2016) utilizaram, são métricas interessantes para avaliar como os frameworks gerenciam os recursos disponíveis. Também é extremamente importante executar os experimentos em diferentes dispositivos para garantir maior abrangência.

Finalmente, como, devido a falta de equipamentos, o foco deste trabalho foi somente o sistema operacional Android, uma análise complementar seria repetir os experimentos na plataforma iOS. Nesse caso, as ferramentas do Android utilizadas — como o ADB, utilizado para capturar informações sobre a taxa de FPS —, precisariam ser substituídas.

REFERÊNCIAS

RAJ, C. P. R.; TOLETY, S. B. **A study on approaches to build cross-platform mobile applications and criteria to select appropriate approach.** 2012 Annual IEEE India Conference (INDICON), 2012.

LATIF, M.; LAKHRISSI, Y.; NFAOUI, E. H.; ES-SBAI, N. **Cross platform approach for mobile application development: A survey.** 2016 International Conference on Information Technology for Organizations Development (IT4OD), 2016.

PINTO, C. M; COUTINHO, C. **From Native to Cross-platform Hybrid Development.** 2018 International Conference on Intelligent Systems (IS), 2018.

HAIDER, A. **Evaluation of cross-platform technology Flutter from the user's perspective.** Dissertation, 2021.

JAGIELLO, J. **Performance comparison between React Native and Flutter.** Dissertation, 2019.

WU, W. **React Native vs Flutter, cross-platform mobile application frameworks.** Dissertation, 2018.

HANSSON, N.; VIDHALL, T. **Effects on performance and usability for cross-platform application development using React Native**. Dissertation, 2016.

SCHWEIGER, F. **The layer cake.** 2018. Disponível em: medium.com/flutter-community/the-layer-cake-widgets-elements-renderobjects-7644c314240 1

RASHEVSKAYA, A. **React Native Re-Architecture — What to Expect from The Popular Cross-Platform Framework?** 2020. Disponível em: litslink.com/blog/new-react-native-architecture

PATIL, A. **Deep dive into React Native's New Architecture.** 2022. Disponível em: medium.com/coox-tech/deep-dive-into-react-natives-new-architecture-fb67ae615ccd

SATROM, B. **Choosing the Right JavaScript Framework for Your Next Web Application.** Prog./Kendo UI. 2018.

STATCOUNTER. **Platform market share worldwide.** 2022. Disponível em gs.statcounter.com/platform-market-share.

JETBRAINS. **Miscellaneous Tech - The State of Developer Ecosystem in 2021 Infographic | JetBrains: Developer Tools for Professionals and Teams.** 2021. Disponível em: jetbrains.com/lp/devecosystem-2021/miscellaneous.

STATISTA. **Cross-platform mobile frameworks used by developers worldwide 2019-2021.** 2022. Disponível em: statista.com/statistics/869224/worldwide-software-developer-working-hours/.

FLUTTER. **Flutter SDK releases.** 2022a. Disponível em: docs.flutter.dev/development/tools/sdk/releases.

FLUTTER. **Beautiful apps for every screen.** 2022b. Disponível em flutter.dev/multi-platform.

FLUTTER. **Flutter performance profiling.** 2022c. Disponível em docs.flutter.dev/perf/ui-performance.

FLUTTER. **Flutter architectural overview.** 2022d. Disponível em docs.flutter.dev/resources/architectural-overview.

FLUTTER. **Inside Flutter.** 2022e. Disponível em: docs.flutter.dev/resources/inside-flutter.

FLUTTER. **Performance best practices.** 2022f. Disponível em: docs.flutter.dev/perf/best-practices.

FLUTTER. **ListView Class.** 2022g. Disponível em: api.flutter.dev/flutter/widgets/ListView-class.html.

FLUTTER. **Calculating Flutter frame rate.** 2020. Disponível em: flutter.dev/go/calculating-flutter-frame-rate.

REACT NATIVE. **React Native versions.** 2022a. Disponível em: archive.reactnative.dev/version.

REACT NATIVE. **Learn once, write everywhere.** 2022b. Disponível em: reactnative.dev.

REACT NATIVE. **Threading Model.** 2022c. Disponível em: reactnative.dev/architecture/threading-model.

REACT NATIVE. **Architecture Overview.** 2022d. Disponível em: reactnative.dev/architecture/overview.

REACT NATIVE. **Performance Overview.** 2022e. Disponível em: reactnative.dev/docs/performance.

REACT NATIVE. **Optimizing Flatlist Configuration.** 2022f. Disponível em: reactnative.dev/docs/optimizing-flatlist-configuration.