FACULDADE E ESCOLA TECNICA ALCIDES MAYA

CURSO SUPERIOR DE TECNOLOGIA EM SISTEMAS PARA INTERNET

TECNOLOGIAS PARA O DESENVOLVIMENTO DE APLICAÇÕES MULTIPLATAFORMA:

Um estudo sobre os frameworks React Native e Flutter

ROBSON ROSA DE ALMEIDA
robsonrosadealmeida@gmail.com

Orientador: Prof ME. João Padilha Moreira

Porto Alegre – RS, 2019

ROBSON ROSA DE ALMEIDA

TECNOLOGIAS PARA O DESENVOLVIMENTO DE APLICAÇÕES MULTIPLATAFORMA

Um estudo sobre os frameworks React Native e Flutter

Projeto de Pesquisa para conclusão da disciplina de Projeto II do **Curso superior de Tecnologia Em Sistemas para Internet**

Orientador: Prof. Me. João Padilha Moreira

Porto Alegre - RS

2019

AGRADECIMENTO

Primeiramente gostaria de agradecer à minha mãe Marli Moraes da Rosa e à minha avó Graciolina Moraes por terem lutado sempre, para que eu e meus irmãos tivéssemos uma boa criação. Graças à vocês hoje posso estar aqui escrevendo estas palavras, amo muito vocês duas e serei eternamente grato.

Gostaria de agradecer também à minha dinda Fátima Lima por sempre me ajudar e incentivar a correr atrás dos meus objetivos.

Agradeço também ao quarto irmão que tenho, meu amigo Bruno Barbosa Brasil, que desde os tempos de escola está sempre me aconselhando e incentivando nos momentos nos momentos difíceis e de decisão.

Deixo uma imensa saudação e gratidão ao meu falecido avô, Martinho Cascaes, por ter sido um molde para grande parte da minha formação como pessoa, suas palavras de experiência e sabedoria foram impagáveis. Você foi o maior incentivador no início da minha jornada na área de TI e me deu alguns dos maiores conselhos que pude ter na vida. Muito obrigado!

No mais, agradeço a todos que estiveram do meu lado nos últimos meses durante o desenvolvimento deste trabalho, dando incentivo, conselhos e ajudando como possível. Obrigado!

RESUMO

Disponibilizar um software para múltiplas plataformas pode ser uma tarefa extremamente desafiadora. Ao longo do tempo foram criadas diversas ferramentas que se propõem facilitar a vida de quem almeja alcançar este objetivo. Este trabalho tem o objetivo de apresentar os conceitos envolvidos na construção de aplicações multiplataforma, além de realizar um estudo e comparação entre duas das ferramentas de desenvolvimento multiplataforma mais populares da atualidade, o *framework* React Native e o *framework* Flutter. Utilizando-se de ambas as ferramentas citadas, construiu-se uma mesma aplicação *mobile* objetivando avaliar o desempenho dos *frameworks* com base em 5 diferentes pontos e expor qual das ferramentas oferece maiores benefícios aos seus utilizadores. A experiência realizada com a construção do mesmo protótipo de aplicação com as ferramentas abordadas no trabalho, mostrou que o *framework* obtentor do melhor desempenho foi o Flutter.

Palavras-chave: Multiplataforma, React Native, Flutter, Javascript, Dart.

ABSTRACT

Providing cross-platform software can be an extremely challenging task. Over time various tools have been created that aim to make life easier for those who want to achieve this goal. This paper aims to present the concepts involved in the construction of multiplatform applications, as well as to study and compare two of the most popular multi platform development tools nowadays, the React Native framework and the Flutter framework. Using both tools mentioned, we built the same application to evaluate the performance of frameworks based on 5 different points and expose which of the tools offers greater benefits to its users. Experience with building the same application prototype using both tools discussed in the paper showed that the best performing framework was Flutter.

Keywords: Cross-platform, React Native, Flutter, Javascript, Dart.

LISTA DE FIGURAS

LISTA DE SIGLAS

AOT	Ahead-Of-Time
API	Application Programming Interface
CLI	Command Line Interface
CSS	Cascading Style Sheets
DOM	Document Object Model
ECMA	European Computer Manufacturers Association
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
IDE	Integrated Development Environment
JIT	Just-In-Time
JSX	JavaScript XML
NPM	Node Package Manager
PWA	Progressive Web Apps
SDK	Software Development Kit
UI	User Interface
URL	Uniform Resource Locator

SUMÁRIO

1. INTRODUÇÃO

A constante evolução da tecnologia ocasiona, com uma frequência cada vez maior, o surgimento de novos dispositivos com diferentes interfaces visuais para a interação com os usuários, cada um com suas particularidades e desafios a serem vencidos.

A existência de diferentes plataformas tecnológicas cria grandes oportunidades de mercado para quem souber aproveitá-las, manter contato próximo com o seu usuário final pode aumentar muito as chances deles consumirem seus produtos e serviços, fidelizando-os cada vez mais. Entretanto, disponibilizar seu *software* em diversas plataformas não é uma tarefa simples, muitos são os pontos que devem ser levados em consideração como por exemplo, a tecnologia suportada pela plataforma em questão e o modo de interação que o usuário terá com ela. Dificilmente um *software* desenvolvido para a plataforma X funcionará em uma plataforma Y sem a necessidade de alterações ou até mesmo um novo desenvolvimento a partir do zero. Devido a esses e outros fatores, muitos são os desafios encontrados por nós como desenvolvedores de *software* na hora de produzir aplicações que possam ser executadas em múltiplos dispositivos.

Está incumbida a nós a tarefa de encontrar soluções viáveis para diminuir o empenho de tempo e recursos na hora de desenvolver e atualizar nossas aplicações em múltiplas plataformas. Temos o dever de buscar por novas tecnologias que possibilitem a otimização de nossos esforços e garantam entregas mais ágeis, para que nossos usuários consigam utilizar a versão mais atualizada de nossas aplicações o mais rápido possível.

Nos últimos anos, duas ferramentas de desenvolvimento de aplicações multiplataforma têm ganho cada vez mais atenção da comunidade de desenvolvedores de *software*, o *framework* React Native para dispositivos móveis que é desenvolvido e mantido pela equipe do Facebook, e mais recentemente, o *framework* Flutter criado pela equipe de desenvolvedores do Google.

1.1 MOTIVAÇÃO

Um dos maiores desafios encontrados na área de desenvolvimento de *software* está justamente em otimizar seus esforços para produzir o máximo possível gastando o mínimo de tempo e recursos para tal.

A dificuldade encontrada atualmente para disponibilizar algum software em diferentes plataformas vem principalmente do fato que cada plataforma possui sua própria linguagem de programação utilizada para comunicar-se com suas API's e componentes nativos, o que cria a necessidade de se haver diversas equipes trabalhando na construção de uma mesma aplicação, gerando mais custos e menos produtividade.

Faz-se então crucial a busca por soluções que diminuam o tempo e custo para a produção de aplicações, mesmo que estas necessitem estar disponíveis em diferentes plataformas de diferentes fornecedores, visando assim aumentar a produtividade e as margens de lucro da empresa responsável pelo desenvolvimento do software.

1.2 OBJETIVO GERAL

O presente trabalho objetiva apresentar os conceitos envolvidos na construção de aplicações que possam ser utilizadas em múltiplas plataformas. Através do estudo comparativo, explicação e exemplificação do funcionamento dos *frameworks* React Native e Flutter na construção de aplicações multiplataforma a partir de uma única base de código.

1.3 OBJETIVO ESPECÍFICO

O trabalho almeja realizar a construção de um mesmo protótipo de aplicação utilizando os *frameworks* React Native e Flutter, para que através da experiência obtida com cada uma das ferramentas, seja possível realizar a comparação de ambas de acordo com 5 critérios de avaliação:

- Qual a ferramenta que possui maior praticidade para ser utilizada?
- Qual a ferramenta que possui melhor documentação?
- Qual a ferramenta que tem menor curva de aprendizado inicial?

- A ferramenta que entrega melhor experiência durante o desenvolvimento?
- Qual das ferramentas entregou o melhor resultado final?

1.4 DELIMITAÇÃO

O presente trabalho está delimitado ao estudo da utilização das ferramentas citadas anteriormente, para a criação de um protótipo de aplicação multiplataforma *mobile*, tendo como alvo as plataformas Android e IOS. O processo de desenvolvimento para outras plataformas não será abordado neste trabalho.

Pelo motivo de não se ter uma máquina com sistema operacional macOS, e pela necessidade de se ter a assinatura do “Apple Developer Program” para que seja possível realizar o lançamento da aplicação na loja de aplicativos “App Store”, a aplicação construída não foi submetida para a loja de aplicativos para IOS. Apenas simulações foram realizadas através de emuladores da plataforma, por este motivo as conclusões sobre o resultado final gerado por cada ferramenta foram tomadas com base apenas na versão Android da aplicação.

1.5 METODOLOGIA

Primeiramente realizou-se uma pesquisa bibliográfica em trabalhos relevantes ao tema, a fim de construir a fundamentação teórica sobre os assuntos abordados.

O segunda parte deste trabalho foi desenvolvida através de pesquisa exploratória das ferramentas em foco. Foi realizada a construção de um protótipo de aplicação *mobile* para o sistema chamado PartyU, este sistema tem o objetivo de possibilitar a busca e divulgação de eventos.

A experiência realizada com ambos os *frameworks* serviu como embasamento para as conclusões obtidas sobre as ferramentas no final do trabalho.

Todo desenvolvimento do projeto foi realizado utilizando um ambiente de desenvolvimento composto por uma máquina de desenvolvimento e um *smartphone* de testes, as conclusões sobre a experiência de desenvolvimento foram tomadas baseadas nas configurações destes dois aparelhos, são elas:

- **Máquina de desenvolvimento**:
    - Modelo Acer Aspire E 15;

    - Processador Intel Core i5-6200U 2.3GHz;
    - Memória RAM de 8GB DDR3;
    - Placa gráfica NVIDIA GeForce 920M;
    - Sistema operacional Windows 10 Home x64;
    - IDE Visual Studio Code.
- **Smartphone de testes**:
    - Modelo Samsung J5 Pro;
    - Processador Octa Core 1.6GHz;
    - Memória RAM de 4GB;
    - Sistema operacional Android 9.0.

1.6 ESTRUTURA

Este trabalho está estruturado em 6 capítulos, os capítulos subsequentes estão organizados da seguinte forma:

Capítulo 2: faz a fundamentação teórica do assunto abordado, apresentando seus principais conceitos e salientando sua importância, são citadas algumas das principais abordagens para o desenvolvimento multiplataforma;

Capítulo 3: apresenta os frameworks React Native e Flutter, explicando suas estruturas básicas, as tecnologias envolvidas e seus respectivos modos de funcionamento;

Capítulo 4: apresenta a proposta para a construção de uma aplicação multiplataforma, e demonstra os passos realizados durante o início do projeto, para implementação da primeira funcionalidade da aplicação PartyU, utilizando ambas as tecnologias estudadas neste trabalho;

Capítulo 5: realiza a comparação entre as ferramentas tema do trabalho, expondo seus pontos positivos e negativo;

Capítulo 6: apresenta as conclusões obtidas com o desenvolvimento do trabalho.

2. FUNDAMENTAÇÃO TEÓRICA

Inicialmente, quando se tinha a intenção de produzir uma aplicação que pudesse ser executada em mais de uma plataforma existia apenas um caminho a ser seguido, era necessário realizar o desenvolvimento da mesma aplicação para cada uma das plataformas alvo. Isso fazia com que o suporte para aplicações em múltiplas plataformas fosse extremamente custoso e/ou demorado para as empresas e desenvolvedores de *software*.

Neste cenário, algumas estratégias de desenvolvimento costumam ser adotadas a fim de otimizar os recursos empenhados nessa tarefa. Pode-se alocar diferentes equipes de desenvolvimento para implementar uma versão da aplicação para cada uma das plataformas onde se pretende disponibilizar o *software*, o que permite o suporte a diferentes plataformas em paralelo, mas acaba elevando os custos de desenvolvimento pois são necessários mais desenvolvedores para que o trabalho possa ser concluído. Alternativamente, pode-se dar suporte a cada uma das plataformas sequencialmente, o que mantém o custo de desenvolvimento não muito elevado, porém aumenta significativamente o prazo para que a aplicação fique disponível em todas as plataformas alvo. Ambas as alternativas mostram-se ineficientes nos dias atuais, visto que sempre existe a necessidade de se entregar o *software* em um prazo mínimo e com o menor custo de desenvolvimento possível (EL-KASSAS, Wafaa S. et al. 2017).

Estas dificuldades apresentadas criam a necessidade da existência de soluções mais eficientes na hora de desenvolver aplicações com suporte a múltiplas plataformas, é neste contexto que surgem os *frameworks* de desenvolvimento multiplataforma, os quais possibilitam a implementação de uma única base de código que irá contemplar diversas plataformas ao mesmo tempo.

2.1 O DESENVOLVIMENTO MULTIPLATAFORMA

Desenvolvimento multiplataforma é a habilidade de poder desenvolver *software* que possa ser utilizado em mais de uma plataforma, normalmente sistemas operacionais distintos, independentemente destas plataformas possuírem arquiteturas e API’s nativas totalmente diferentes (CETINER; ABURAS. 2005).

Segundo EL-KASSAS (2017) o principal conceito das soluções multiplataforma é desenvolver determinada aplicação uma única vez e poder utilizá-la em qualquer lugar. Porém, o mesmo reconhece que muitas das soluções multiplataforma existentes ainda estão sob pesquisa e desenvolvimento, e que apesar de algumas já estarem disponíveis para uso comercial, uma solução robusta a ponto de contemplar todas as plataformas necessárias ainda não existe.

Existem diversas abordagens para que se consiga disponibilizar sua aplicação para ser executada em múltiplas plataformas, cada uma dessas abordagens possui pontos a favor e contra. A escolha de qual abordagem será adotada deve ser feita com base em análises dos objetivos de negócio e necessidades que se tem para a aplicação. Para CETINER e ABURAS (2005), independentemente de qual abordagem será escolhida, existe a necessidade de uma análise de negócio como um estágio a parte durante o processo de desenvolvimento de Sistemas da Informação. Uma metodologia de modelagem adequada deve ser aplicada sobre a análise realizada com o objetivo de prover um nível suficiente de abstração dos conceitos específicos que representem o negócio independentemente das dificuldades de *design* ou implementação.

Além da abordagem nativa citada anteriormente, no trabalho desenvolvido por Biørn-Hansen, Grønli e Ghinea (2018) são apontadas cinco principais categorias de abordagens para desenvolvimento multiplataforma possibilitadas por diferentes *frameworks*, as quais são predominantemente citadas nas literaturas relacionadas ao assunto, deixando de fora outras citadas com menor frequência. As categorias levantadas pelos autores são: Híbrida; Interpretada; Compilação multiplataforma; Orientada a modelos e Aplicações web progressivas. As próximas subseções são dedicadas à explicação de cada uma dessas categorias, de modo que seja possível compreender seus fundamentos, benefícios e como elas diferem entre si.

2.1.1 Abordagem de desenvolvimento Híbrida

A abordagem de desenvolvimento Híbrida possibilita a utilização de tecnologias *web* como HTML, CSS e JavaScript para a construção de aplicações multiplataforma. Em plataformas *mobile*, o funcionamento deste tipo de aplicação se dá através da inicialização de uma aplicação nativa que renderiza um

componente de WebView, este funciona de forma similar a um *web browser*, por sua vez a WebView trata de interpretar os arquivos criados utilizando as tecnologias citadas e renderiza-los como se fosse uma aplicação *web*.

O código gerado nos arquivos criados contém a lógica de funcionamento da aplicação, assim como as definições de interface de usuário. O resultado final do desenvolvimento utilizando esta abordagem é um pacote da aplicação contendo os arquivos com definições de lógica e interface, assim como o código necessário para realizar a comunicação entre a WebView e o código nativo da plataforma. Pelo fato da aplicação desenvolvida utilizando tecnologias web ser executada dentro de uma aplicação nativa, o desenvolvedor tem a possibilidade enviar o pacote final gerado para publicação em lojas de aplicativos oficiais de cada plataforma alvo, da mesma forma que faria para submeter aplicações desenvolvidas utilizando a abordagem nativa.

Algumas das vantagens obtidas ao se utilizar este tipo de abordagem vem do fato dela ser baseada em tecnologias padrão da *web*, o que faz com que a aplicação funcione em praticamente qualquer *browser*, deixando pouco complicado o reaproveitamento da base de código criada para ser utilizada também na plataforma *web*. Além disso, a abordagem Híbrida possibilita que desenvolvedores que já possuam conhecimento trabalhando com tecnologias voltadas para a *web* consigam utilizá-lo para desenvolver aplicações mobile para diferentes plataformas ao mesmo tempo. Em contrapartida, por não ser uma aplicação nativa que está sendo apresentada para interação com o usuário, torna-se impossível a utilização de componentes nativos da plataforma para a composição do layout da aplicação, o que pode levar a uma experiência pouco intuitiva para os usuários. Para contornar esse problema, muitos frameworks disponibilizam suas próprias bibliotecas, ou alguma forma de integrar a aplicação com bibliotecas de terceiros que simulam a aparência e o comportamento de componentes nativos de determinadas plataformas, mas que são construídos utilizando HTML, CSS e JavaScript.

2.1.2 Abordagem de desenvolvimento Interpretada

Assim como na abordagem Híbrida, na abordagem de desenvolvimento Interpretada é comum encontrarmos *frameworks* que permitem ao desenvolvedor

construir suas aplicações utilizando a linguagem JavaScript, apesar de que esta não é uma exigência desta abordagem. A fundamental diferença entre a abordagem Híbrida e a Interpretada é que a primeira utiliza um componente de WebView para conseguir executar e renderizar a aplicação desenvolvida, enquanto a segunda faz uso de interpretadores JavaScript presentes nos dispositivos para renderizar componentes de interface nativos e acessar outras API's específicas da plataforma, como câmera, GPS, lista de contatos e etc. Para que seja possível renderizar os componentes nativos, os *frameworks* desta categoria utilizam uma técnica conhecida como *bridging*, que realiza a comunicação entre o código desenvolvido no *framework* e o código nativo da plataforma.

A principal vantagem da utilização dessa abordagem é a possibilidade de renderizar componentes nativos da plataforma onde a aplicação está sendo executada, não havendo assim a necessidade de bibliotecas e código extra para recriar uma experiência de uso nativa para o usuário. Em contraponto, por necessitar que o código desenvolvido em uma outra linguagem seja interpretado e traduzido para ser compreendido pela plataforma alvo, existe uma perda de performance durante a execução da aplicação.

2.1.3 Abordagem de Compilação multiplataforma

A diferença da abordagem de Compilação multiplataforma em relação às duas citadas anteriormente é que esta, não depende de outros artifícios como WebViews ou interpretadores de código no dispositivo em tempo de execução. Os *frameworks* construídos com esta abordagem em mente utilizam uma única linguagem de programação sobre a qual a aplicação é desenvolvida, esta linguagem deve disponibilizar acesso às API's nativas do dispositivo, através de SDK's (Software Development Kit) contidos nos *frameworks*, estes são responsáveis por mapear as funcionalidades desenvolvidas para que possam ser compreendidas pelos SDK's da plataforma em que a aplicação será executada. Ao final do processo de desenvolvimento o *framework* gera um *bytecode* da aplicação a partir do código desenvolvido com a linguagem base, por fim é gerado um executável do *software* para cada uma das plataformas suportadas pelo *framework*.

A grande vantagem obtida ao se utilizar esta abordagem de desenvolvimento multiplataforma é que ao final do processo de desenvolvimento é gerada uma aplicação nativa para cada plataforma alvo, o que garante a aparência e performance igual (ou muito semelhante) se comparada a aplicações desenvolvidas utilizando a abordagem nativa. Um ponto negativo dessa abordagem seria a limitação por parte dos *frameworks* para as plataformas suportadas, em outras palavras, o desenvolvedor só pode gerar aplicações para as plataformas que o *framework* dá suporte, portanto caso surja uma nova plataforma a qual deseja-se dar suporte, o desenvolvedor deverá aguardar até que o *framework* disponha de suporte a tal plataforma.

2.1.4 Abordagem Orientada a modelos

Esta abordagem de desenvolvimento multiplataforma provém do paradigma de desenvolvimento orientado a modelos (Model-Driven Development). Ela consiste na criação de aplicações utilizando linguagens específicas de domínio (Domain-Specific Language) disponibilizadas pelos *frameworks*, esta é uma forma de abstração do problema a ser solucionado pela aplicação em linguagem textual ou gráfica, a qual pode ser compreendida e construída tanto por desenvolvedores quanto por usuários que não possuam conhecimento na área de desenvolvimento de *software*.

A vantagem da utilização dessa abordagem é que ela permite a desenvolvedores e usuários comuns construírem aplicações a partir da linguagem de domínio ao invés de uma linguagem de programação, portanto é possível que, sem conhecimento prévio em programação o usuário consiga gerar uma aplicação para diferentes plataformas a partir de regras de negócio. Assim como na abordagem de compilação multiplataforma, a abordagem orientada a modelos é limitada às plataformas suportadas pelo *framework* que está sendo utilizado para a criação da aplicação.

2.1.5 Abordagem de Aplicações *web* progressivas

As aplicações *web* progressivas tem ganho popularidade entre a comunidade de desenvolvedores nos últimos anos. Basicamente Progressive Web Apps (PWA's) são aplicações *web* com capacidades estendidas, ainda que sua capacidade de utilização dos componentes nativos da plataforma deixe a desejar. Assim como as aplicações *web* típicas, as PWA's são hospedadas e servidas por servidores *web*, devendo ser instaladas por meio de uma URL no *browser* do dispositivo. Pode-se utilizar técnicas similares às utilizadas na abordagem de desenvolvimento híbrida para que se consiga deixar a aplicação com uma interface mais parecida com a de aplicações nativas a partir da estilização dos componentes com HTML e CSS.

O funcionamento de uma aplicação *web* progressiva se dá através da visita a uma URL por meio do *browser* do dispositivo, ao acessar esta URL um *website* é aberto e é feita uma solicitação ao usuário para que seja feito o *download* e a instalação da aplicação na plataforma. Neste *download* estão incluídos os recursos que compõe a aplicação, como por exemplo arquivos HTML, CSS, JavaScript e de imagens, possibilitando assim a sua utilização mesmo sem conexão com a internet. Também é criado um ícone para a aplicação na tela inicial do dispositivo, da mesma forma que ocorre com as aplicações instaladas via lojas de aplicativos.

Da mesma maneira que as aplicações híbridas, o ponto a favor às aplicações progressivas é a possibilidade de construir aplicações que funcionem em diversas plataformas por meio da utilização de tecnologias padrão da *web*. Em oposição, os problemas ao se utilizar esta abordagem provém em grande parte, da limitação das funcionalidades nativas do dispositivo que podem ser utilizadas por estas aplicações, pois estão sendo executadas por um *browser*. Além disso, não existe a possibilidade de submeter este tipo de aplicação para lojas de aplicativos, sendo o endereço *web* do servidor o único meio de instalação da aplicação.

2.2 FRAMEWORK

Segundo o que consta na enciclopédia da revista PC Magazine, um framework de aplicação pode ser definido como um conjunto de rotinas de *software*

que proveem uma fundação da estrutura para o desenvolvimento da aplicação. Geralmente, *frameworks* de aplicações orientadas a objetos são estruturados em forma de bibliotecas de classes.

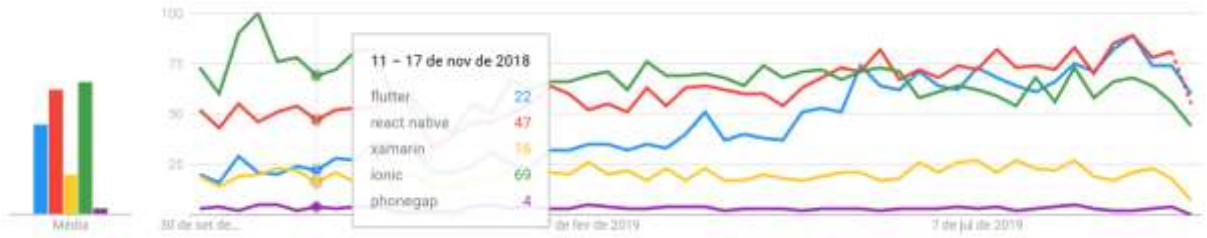
O propósito de um *framework* é eliminar parte do trabalho de desenvolvimento a ser realizado para a criação de aplicações.

Existem *frameworks* construídos para propósitos específicos, como é o caso das ferramentas estudadas neste trabalho.

3. AS FERRAMENTAS ABORDADAS

O capítulo anterior introduziu os conceitos do desenvolvimento multiplataforma e apresentou as cinco principais abordagens que possibilitam a construção deste tipo de aplicação. Atualmente dois *frameworks* vem gerando grande interesse por parte dos desenvolvedores, como pode ser notado na figura 1, ambos foram selecionados para ser objeto de estudo neste trabalho. O React Native é uma ferramenta mais madura, já consolidada e utilizada por grandes empresas no mercado, de modo oposto, o Flutter é um *framework* mais novo que vem ganhando muita popularidade desde seu lançamento. Este capítulo apresentará as ferramentas selecionadas para serem o ponto focal ao longo do desenvolvimento deste trabalho.

Figura 1 - Relevância dos nomes de frameworks multiplataforma em 2019.



Fonte: Google Trends. (2019)

3.1 FRAMEWORK REACT NATIVE

O React Native é um *framework* para desenvolvimento de aplicações nativas baseado na biblioteca de JavaScript React. As próximas subseções apresentam brevemente as tecnologias por ele utilizadas e o seu modo de funcionamento.

3.1.1 JAVASCRIPT

A linguagem de programação JavaScript é uma linguagem de *script* multiplataforma, orientada a objetos. Caracteriza-se por ser pequena e leve, foi projetada para ser facilmente integrada com outros produtos e aplicações, não sendo útil para ser utilizada isolada de outros atores. Quando em um ambiente de hospedagem, o JavaScript pode comunicar-se com objetos do ambiente e exercer controle programático sobre eles. (MDN. 2019)

A linguagem de programação JavaScript foi criada por Brendan Eich em 1995, quando o mesmo foi contratado pela Netscape para criar uma linguagem de programação que seria executada no *browser* desenvolvido pela empresa, o Netscape 2.0. O objetivo da Netscape era criar uma linguagem leve e interpretada que serviria como complemento ao Java que é uma linguagem compilada, rica e complexa. (SEVERANCE. 2012)

O nome oficial da linguagem JavaScript é ECMAScript, pois quem cria os padrões e especificações da linguagem é a associação ECMA (European Computer Manufactures Association) desde 1996. O nome JavaScript já havia sido patenteado pela Sun Microsystems (atualmente Oracle), então foi necessário utilizar uma nova nomenclatura. Como o nome JavaScript já havia se popularizado, os desenvolvedores continuaram a se referir à linguagem pelo antigo nome e passaram a chamar de ECMAScript a versão da especificação desenvolvida pela ECMA (RAUSCHMAYER. 2014). Atualmente a versão da especificação é a ECMAScript 2019.

3.1.2 BIBLIOTECA REACT

O React é uma biblioteca de JavaScript declarativa, eficiente e flexível para criar interfaces com o usuário. Ela permite compor interfaces de usuário complexas a partir de pequenos e isolados códigos chamados “componentes” (DOCUMENTAÇÃO REACT, 2019).

A idéia por trás do React é possibilitar a criação componentes encapsulados que gerenciam o seu próprio estado e através da combinação de diversos componentes poder criar interfaces de usuário complexas. Com isso é possível criar *views* simples para cada estado da aplicação e o React irá tomar conta para atualizar e renderizar de forma otimizada apenas os componentes necessários quando os dados forem modificados.

Basicamente um componente é uma classe JavaScript que estende a classe “React.Component” e implementa um método “render”, este método deve retornar o que deve ser exibido para o usuário através de uma sintaxe chamada JSX, componentes também podem ser criados através de funções, estes são conhecidos como “componentes de função”. O JSX é uma extensão de sintaxe para JavaScript

que serve para descrever como a UI deve ser apresentada, nela é possível ter acesso a lógica de renderização e outros dados do componente. A utilização da sintaxe JSX não é necessária para se utilizar o React, porém é a maneira mais padronizada e a recomendada pelos desenvolvedores da biblioteca.

Figura 2 - Exemplo de um componente simples do React.

Fonte: Documentação React. (2019)

Para renderizar os elementos do JSX que compõem o componente, o React possui o React DOM que é um mapeamento em JavaScript da estrutura do DOM (Document Object Model) chamado de Virtual DOM. Sua função é otimizar a renderização do componente fazendo uma comparação entre os “nodes” existentes no Virtual DOM e no DOM, e realizando a atualização do componente em tela apenas quando e onde for necessário.

Dois conceitos importantes para componentes do React são “props” e “state”, ambos representam as informações que o componente detém, porém possuem importantes diferenças. Props são atributos JSX que podem ser passados para um componente quando este é utilizado dentro da sintaxe do JSX, o valor dessas propriedades é repassado para o componente dentro de um objeto chamado “props” que fica acessível dentro do componente, um componente não deve modificar estes dados. Para que um componente possa manipular seus dados existe o “state”, ele é um objeto privado e totalmente gerenciado pelo componente.

Figura 3 - Exemplo de componente com estado (state) interno.

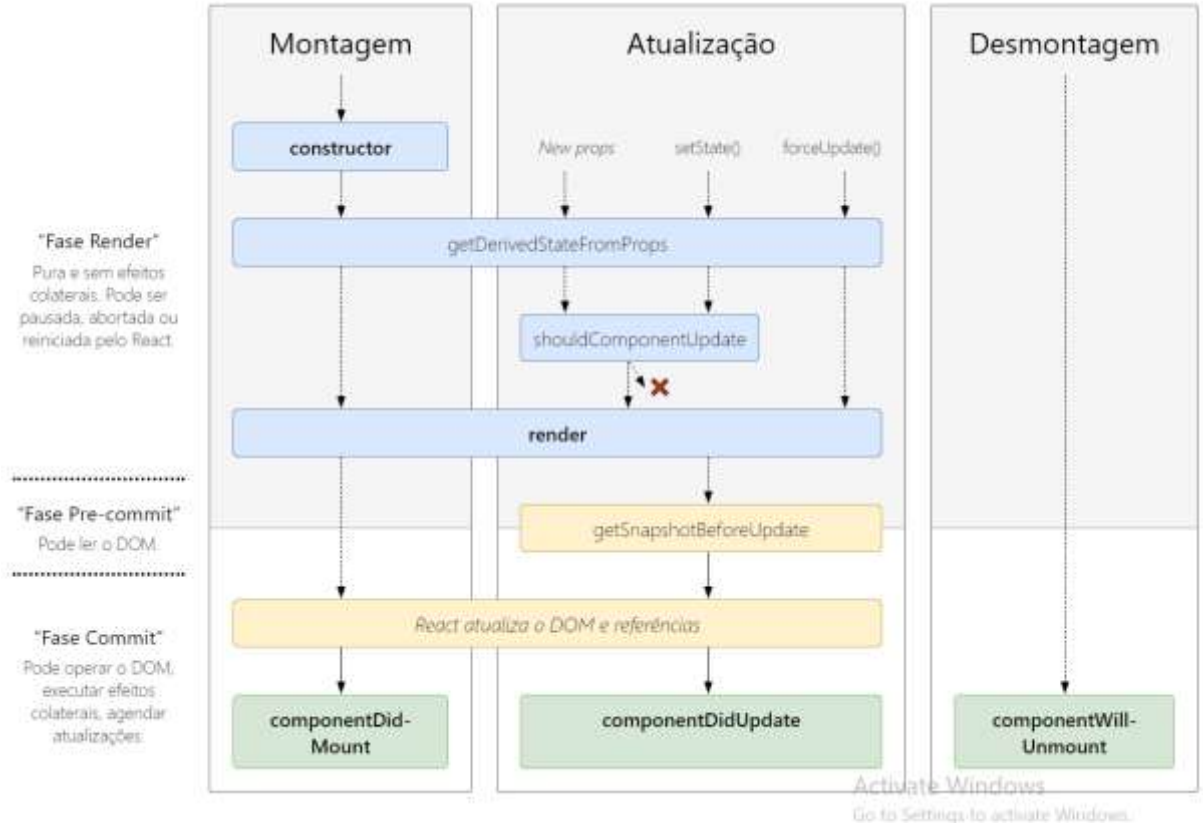
```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

Fonte: Documentação React. (2019)

Cada componente do React possui um ciclo de vida, o ciclo de vida de um componente são métodos que podem ser sobrescritos para executar um trecho de código em determinados momentos do ciclo. A figura 4 representa o diagrama do ciclo de vida dos componentes do React. Os métodos de ciclo de vida utilizados mais comumente são:

- **constructor()**: o construtor é chamado antes que o componente seja montado;
- **render()**: o método render é o único método obrigatório em componentes de classe, normalmente seu valor de retorno é um elemento do React construído com a sintaxe JSX como “ ” ou “<CustomComponent />”, mas também pode retornar outros tipos de valores do React;
- **componentDidMount()**: o método “componentDidMount” é executado imediatamente após o componente ser montado e inserido no DOM;
- **componentDidUpdate()**: é invocado após uma atualização ocorrer, ele não é executado na primeira renderização do componente;
- **componentWillUnmount()**: este método é invocado antes do componente ser desmontado e destruído.

Figura 4 - Métodos de ciclo de vida dos componentes React.



Fonte: Documentação React. (2019)

3.1.3 O REACT NATIVE

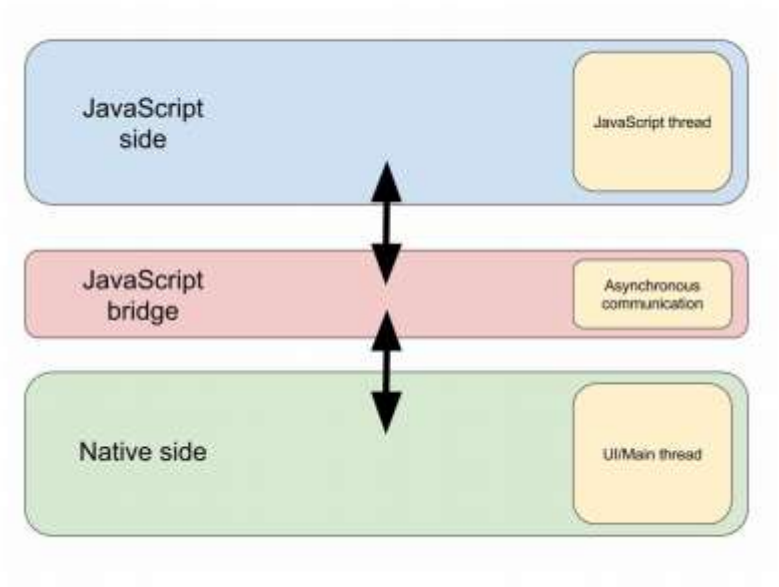
O React Native é um framework para a criação de aplicações multiplataforma que utiliza a abordagem de desenvolvimento interpretada, descrita na seção 2.1.2. Utiliza-se JavaScript para desenvolver a base de código da aplicação, este código é interpretado pela *engine* de JavaScript do dispositivo que renderiza a aplicação utilizando API's nativas da plataforma.

O React Native utiliza o React para a construção das interfaces de usuário, então todos os aspectos da biblioteca vistos anteriormente são válidos neste contexto, a grande diferença é que ao utilizar o React Native, o *framework* irá disponibilizar um conjunto de componentes como "View", "Text" e "Image" que quando utilizados, serão mapeados diretamente para componentes de construção de interface de usuário nativos da plataforma.

A comunicação do React Native com as API's nativas da plataforma é feita através de uma ponte (bridge) JavaScript que utiliza o JavaScriptCore, a mesma

engine de JavaScript do projeto open source WebKit, como seu interpretador. A figura 5 apresenta uma demonstração visual de como funciona essa comunicação.

Figura 5 - Comunicação do React Native com API's nativas da plataforma.



Fonte: HANSSON; VIDHALL. (2016)

A ponte JavaScript recebe chamadas de código em JavaScript e cuida para transformar os tipos de dados para se adequarem aos tipos nativos da plataforma, essas chamadas são processadas e repassadas, caso a mesma não possa ser enviada no momento, uma pilha de mensagens é criada para que possa ser processada posteriormente. Os componentes de interface de usuário nativos não sofrem interferência durante esse processo, pois a parte JavaScript não roda na *thread* principal do dispositivo e sim em uma *thread* assíncrona. (HANSSON; VIDHALL. 2016)

3.2 FRAMEWORK FLUTTER

O Flutter é um SDK para desenvolvimento de aplicações multiplataforma a partir de uma única base de código. Seu objetivo é possibilitar que desenvolvedores construam aplicações de alta performance e que proporcionem uma experiência de usuário nativa em diferentes plataformas. O Flutter utiliza a linguagem programação Dart para o desenvolvimento do código das aplicações. (DOCUMENTAÇÃO

FLUTTER. 2019) As próximas subseções abordam as tecnologias utilizadas pelo Flutter e o seu modo de funcionamento.

3.2.1 A LINGUAGEM DE PROGRAMAÇÃO DART

No dia 10 de Outubro de 2011 o Google anunciou publicamente o Dart, uma nova linguagem de programação projetada por Lars Bak e Kasper Lund com o objetivo de possibilitar o desenvolvimento para *web* de forma estruturada. (BLOG CHROMIUM, 2011; BRACHA, 2016)

O Dart é uma linguagem baseada em classes, com herança única, puramente orientada a objetos e opcionalmente tipada. Os programas em Dart são organizados de maneira modular, em unidades chamadas *libraries*, um programa utiliza uma ou mais *library* para compor a sua base de código. Os objetivos do projeto da linguagem são:

- Criar uma linguagem estruturada e flexível;
- Fazer a linguagem ser familiar e natural para os programadores, além de ser fácil de aprender;
- Garantir a entrega de alta performance em todos os navegadores modernos e em diversos ambientes e plataformas, como dispositivos *handheld*, *smartphones* e servidores.

O código escrito em Dart pode ser executado de duas maneiras diferentes, pode-se utilizar uma máquina virtual nativa ou utilizar um compilador de JavaScript para traduzir o código escrito em Dart e criar uma versão do mesmo em JavaScript. A linguagem é *open source* e vem com um conjunto básico de *libraries*, além de ferramentas para compilar e executar a linguagem. (L. BAK, 2011) Para poder executar suas aplicações em diferentes plataformas o Dart possui uma tecnologia de compilação flexível, ele possui o Dart Native que possibilita criar programas para dispositivos móveis, *desktop* e outros, além do Dart Web que traz ferramentas necessárias para o desenvolvimento de aplicações que serão executadas na *web*.

- **Dart Native**: conta com a Dart VM que possui compilação just-in-time (JIT), conta também com um compilador ahead-of-time (AOT) que transforma o código escrito com a linguagem Dart em código de máquina;

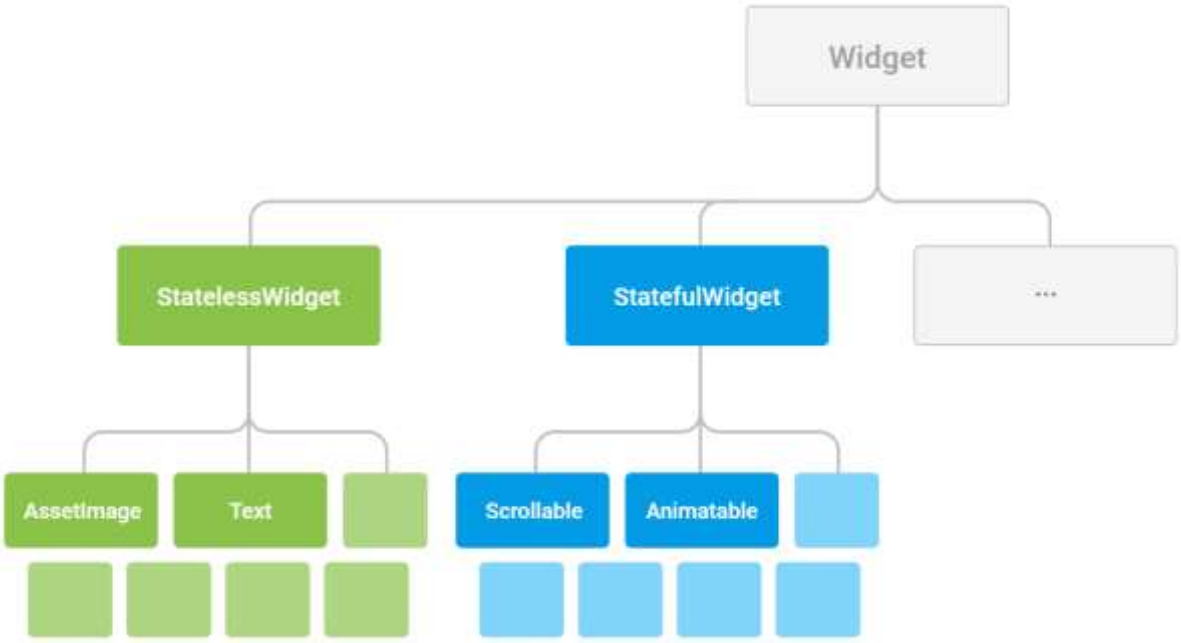
- **Dart Web**: inclui o “dartdevc” um compilador de tempo de desenvolvimento e o “dart2js” um compilador para JavaScript de produção.

3.2.2 FLUTTER

O Flutter é o SDK open source do Google que possibilita a criação de aplicações que possam ser executadas em múltiplas plataformas a partir de uma única base de código. A primeira versão estável do Flutter (v1.0) foi disponibilizada no dia 4 de Dezembro de 2018, atualmente o Flutter está em sua versão 1.9. O objetivo do *framework* é possibilitar que desenvolvedores possam criar aplicações de alta performance, com uma experiência nativa em diferentes plataformas.

O fluxo de desenvolvimento do Flutter é baseado na criação de *widgets*, existem *widgets* para definir todos tipos de elementos estruturais e de estilização. *Widgets* podem ser: elementos de estrutura como botões e menus; elementos de estilo como fontes e cores; elementos de *layout* como margens, espaçamentos e etc. O Flutter disponibiliza *widgets* já estilizadas com o *design* específico tanto para a plataforma Android (Material Components) quanto para a plataforma iOS (Cupertino). Utilizando as *widgets* disponibilizadas pelo *framework*, o desenvolvedor pode compor e criar suas próprias *widgets* customizadas de acordo com suas necessidades. Os *widgets* são organizados em uma hierarquia baseada na composição onde cada *widget* herda propriedades de seu superior, esta hierarquia é semelhante ao DOM. (CORAZZA; 2018. DOCUMENTAÇÃO FLUTTER; 2019.) A Figura 6 demonstra um exemplo da hierarquia de *widgets* do Flutter.

Figura 6 - Exemplo da estrutura hierárquica de widgets no Flutter.



Fonte: Documentação Flutter. (2019)

As *widgets* do Flutter podem ser divididas em duas categorias: StatelessWidgets e StatefulWidgets. StatelessWidgets são *widgets* que não possuem nenhum tipo de mudança de estado, assim que criadas permanecem do mesmo modo durante todo o seu período de vida. As StatefulWidgets são *widgets* que possuem qualquer tipo de alteração em seu estado, seja essa gerada por alguma interação do usuário ou qualquer outro fator, por exemplo, um contador que incrementa seu valor a medida que o usuário toca a tela do dispositivo seria uma StatefulWidget.

Como uma demonstração um pouco mais visual da metodologia de construção de interfaces através da composição de *widgets* no Flutter, podemos tomar como exemplo o menu da figura 7.

Figura 7 - Menu de exemplo.

Fonte: Documentação Flutter. (2019)

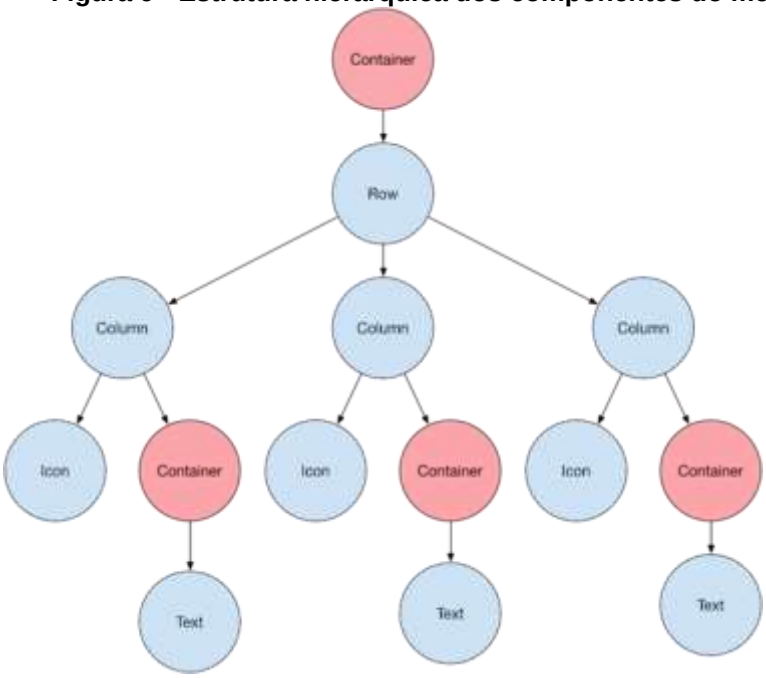
Ao tentar recriar a estrutura desse menu utilizando o Flutter devemos primeiramente dividir todos os elementos do *layout* (texto, imagens, espaçamentos, etc) em unidades isoladas, que representarão as *widgets* da aplicação.

Figura 8 - Separação dos elementos do menu.

Fonte: Documentação Flutter. (2019)

Podemos observar a existência de um elemento contêiner que possui uma cor de fundo clara, dentro dele estão três colunas dispostas em uma linha horizontal, cada coluna destas possui como seus elementos filhos uma *widget* de imagem e uma *widget* com texto. A figura 9 demonstra a representação da hierarquia destes componente em estrutura de árvore.

Figura 9 - Estrutura hierárquica dos componentes do menu.

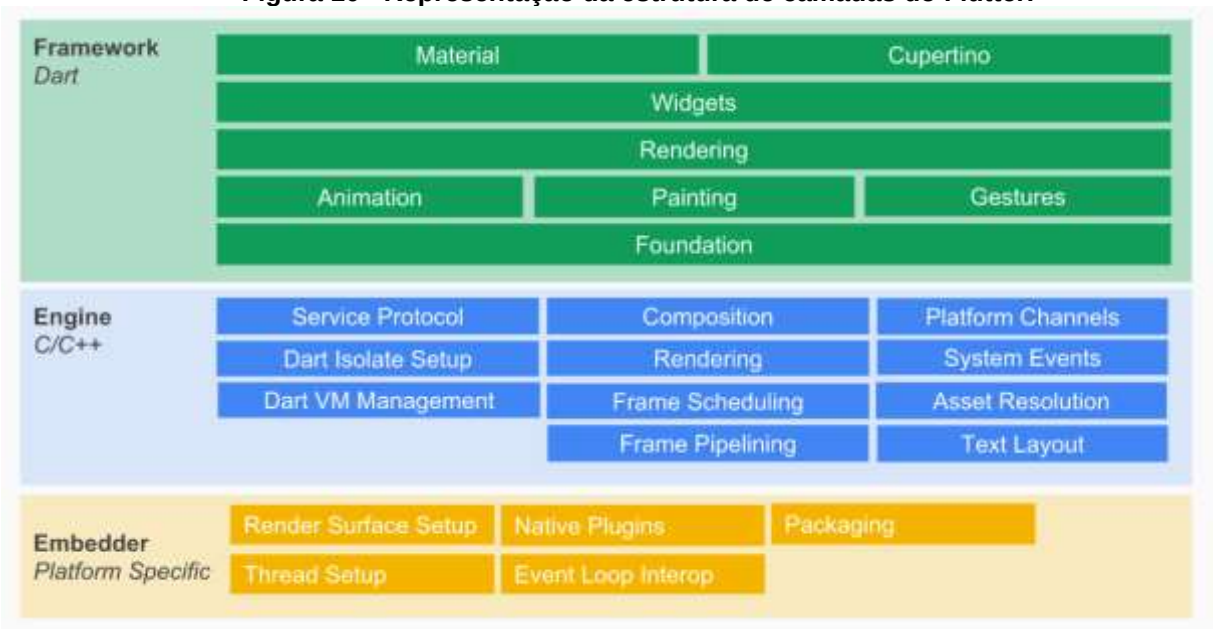


Fonte: Documentação Flutter. (2019)

O *framework* Flutter utiliza a linguagem de programação Dart e foi projetado utilizando uma abordagem de estrutura de camadas, onde cada camada foi construída sobre a camada anterior. A motivação do uso desta abordagem é possibilitar que os desenvolvedores sejam mais produtivos, podendo criar mais funcionalidades necessitando escrever menos linhas de código. As camadas

utilizadas com maior frequência são as camadas mais superiores da estrutura, onde encontram-se as bibliotecas do Material Design (Android) e do Cupertino (IOS). As camadas inferiores da estrutura são utilizadas com pouca frequência, porém disponibilizam acesso e manipulação de objetos de mais baixo nível provenientes do motor de renderização do *framework*. Ao organizar e combinar objetos de diferentes camadas é possível compor e criar novas *widgets* totalmente customizadas pelo desenvolvedor, obviamente isso irá necessitar de um maior esforço durante o desenvolvimento.

Figura 10 - Representação da estrutura de camadas do Flutter.



Fonte: Documentação Flutter. (2019)

4. CONSTRUINDO UMA APLICAÇÃO

No capítulo anterior apresentamos duas das principais tecnologias que existentes no momento para a criação de aplicações multiplataforma, este capítulo tem o objetivo de demonstrar como os *frameworks* abordados neste trabalho podem ser utilizados para construir uma aplicação multiplataforma. Através de uma abordagem prática e explicativa das duas ferramentas, deseja-se implementar a primeira funcionalidade da aplicação proposta e gerar o mesmo (ou o mais similar possível) resultado utilizando ambas as ferramenta. Para a realização desta comparação serão avaliados os 5 pontos levantados na seção “Delimitação e metodologia” deste trabalho.

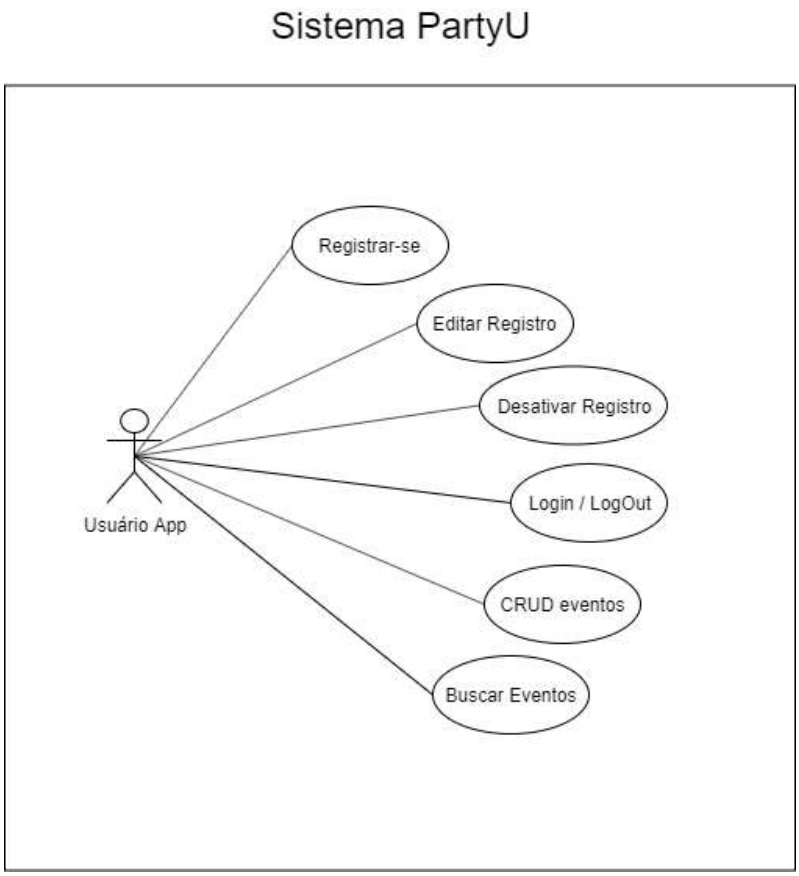
4.1 PROPOSTA DE APLICAÇÃO MULTIPLATAFORMA

O nome da aplicação que criaremos é PartyU, o objetivo do sistema é a busca e divulgação de eventos. O protótipo será simples e consistirá em uma aplicação *mobile* que se comunica com uma API RESTful para executar ações no sistema. A figura 11 mostra o diagrama UML de caso de uso das funcionalidades para o sistema, a aplicação deverá possuir um conjunto de nove funcionalidades, as quais serão descritas a seguir como requisitos funcionais. Os requisitos funcionais da aplicação PartyU são:

- **RF01** - A aplicação deverá permitir a listagem e busca por eventos para todos os usuários;
- **RF02** - A aplicação deverá permitir a criação de eventos apenas por usuários autenticados;
- **RF03** - A aplicação deverá permitir a edição de um evento apenas pelo usuário que o criou;
- **RF04** - A aplicação deverá permitir a deleção de um evento apenas pelo usuário que o criou;
- **RF05** - A aplicação deverá permitir ao usuário desativar o seu próprio cadastro;
- **RF06** - A aplicação deverá permitir a criação de um registro com e-mail, senha e outras informações do usuário;

- **RF07** - A aplicação deverá permitir a edição de um registro apenas pelo usuário que o criou;
- **RF08** - A aplicação deverá permitir que usuários com cadastro ativo façam *login* na aplicação;
- **RF09** - A aplicação deverá permitir que usuários logados façam *logout* da aplicação.

Figura 11 - Diagrama de caso de uso para a aplicação PartyU.



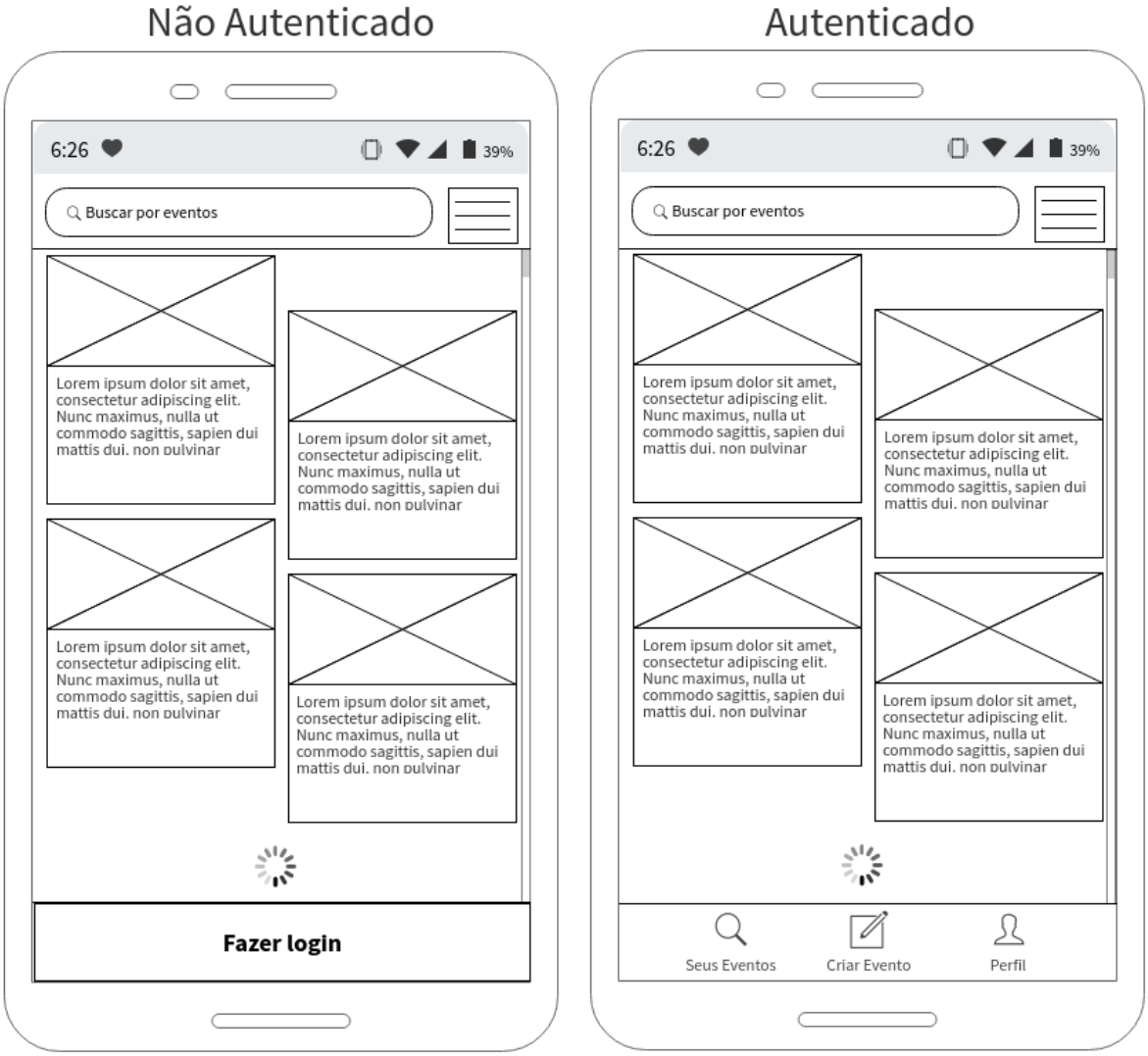
Fonte: Autor. (2019)

As aplicações desenvolvidas devem basear-se nos *wireframes* construídos que podem ser vistos em anexo. Ao todo serão sete modelos de tela diferentes nos quais as funcionalidades apresentadas estarão dispostas. Os telas projetadas para a aplicação são:

- Tela de listagem de eventos;
- Tela de detalhes do evento;
- Tela de login;
- Tela de criação de cadastro;

- Tela de criação de eventos;
- Tela de perfil do usuário;
- Tela de edição do perfil o usuário.

Figura 12 - Tela de listagem de eventos.



Fonte: Autor. (2019)

Figura 13 - Tela de detalhes do evento.

Fonte: Autor. (2019)

Figura 14 - Telas de login e criação de cadastro.

Fonte: Autor. (2019)

Figura 15 - Telas de perfil e edição de perfil.

Fonte: Autor. (2019)

4.2 CONFIGURAÇÃO INICIAL DAS FERRAMENTAS

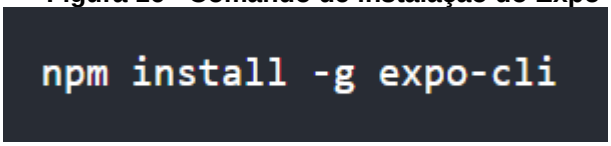
Primeiramente iremos realizar o download e a instalação das ferramentas que serão utilizadas, para que se possa avaliar qual delas possui maior praticidade de utilização. A máquina de trabalho utilizada para a criação da aplicação proposta conta com o sistema operacional Windows 10 e arquitetura x64.

4.2.1 INSTALANDO O REACT NATIVE

O React Native roda em Node.js, portanto é necessário fazer a instalação da plataforma e seu do gerenciador de pacotes o NPM. Depois de instaladas estas ferramentas, a maneira mais simples de se começar a trabalhar com o *framework* é através de um conjunto de ferramentas construídas para o React Native chamado

de Expo CLI, pode-se fazer a instalação desta CLI pela linha de comando através do NPM como mostrados a seguir:

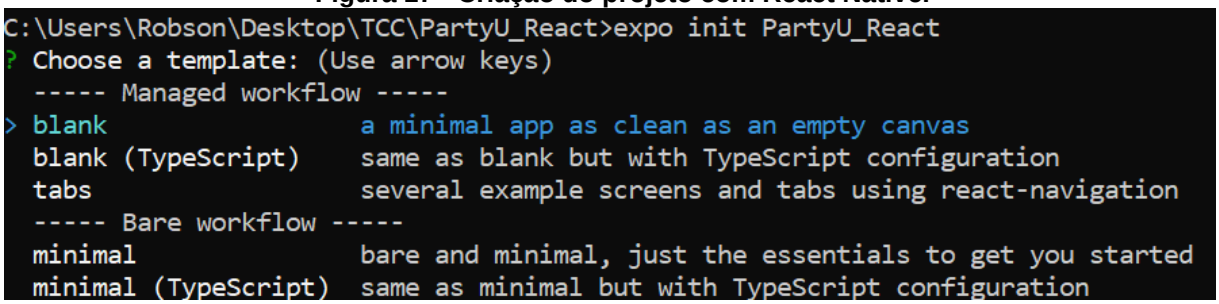
Figura 16 - Comando de instalação do Expo CLI.



Fonte: Autor. (2019)

Após o término da instalação do Expo CLI, o comando “expo” ficará disponível na linha de comando do sistema. É possível utilizar o comando “expo” com a opção “init”, seguido de um nome sem espaços para que a ferramenta crie automaticamente um novo projeto. Serão apresentadas algumas opções de templates para novas aplicações, como mostra a figura à seguir:

Figura 17 - Criação do projeto com React Native.



Fonte: Autor. (2019)

Depois de escolhidas as opções desejadas, serão instalados diversos módulos necessários para o funcionamento da aplicação e serão criados os arquivos iniciais do projeto. A figura 18 mostra o conteúdo do arquivo “App.js”, este é o arquivo principal da aplicação, ele serve como ponto de início do projeto e a partir dele construímos a aplicação.

Figura 18 - Conteúdo do arquivo “App.js”.

Fonte: Autor. (2019)

Ao final de todo o processo de criação, basta que pela ferramenta de linha de comando você vá até a pasta criada e utilize o comando “npm start” para executar a aplicação. Isso fará com que o Expo inicialize um servidor de desenvolvimento na sua máquina e abra uma aba de navegador com a dashboard de monitoramento da ferramenta. Na *dashboard* apresentada existem diversas opções para executar a aplicação, neste trabalho utilizaremos para testes um dispositivo físico com Android 9. Para testes em aparelhos físicos é possível fazer o *download* do app Expo para IOS ou Android e utilizá-lo para ler o código QR apresentado na *dashboard*, se o aparelho estiver conectado na mesma rede de internet que o servidor de desenvolvimento, o app do Expo irá inicializar o *download* dos arquivos necessários para a execução da nova aplicação no dispositivo, além de inicializar o mecanismo de reload automático. Ao final de tudo a aplicação é carregada e executada no *smartphone*.

Figura 19 - Primeiro carregamento da aplicação.

Fonte: Autor. (2019)

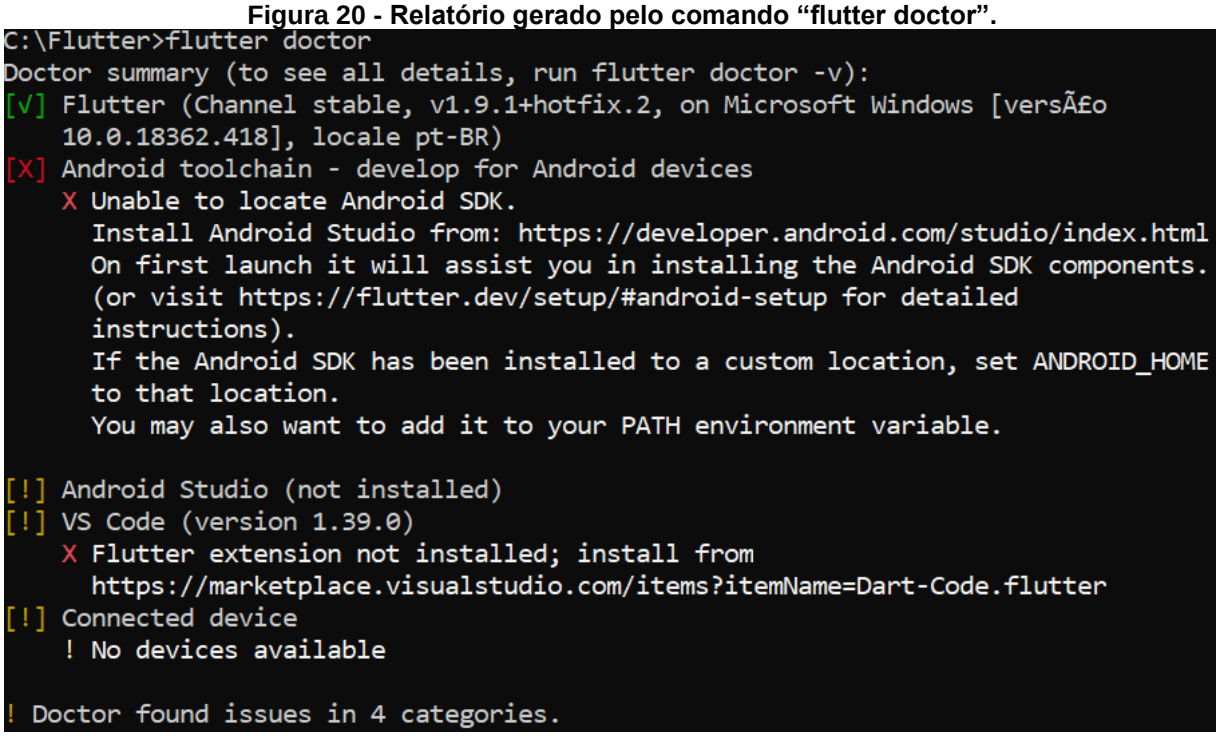
4.2.1 INSTALANDO O FLUTTER

Para realizar a instalação do *framework* Flutter devemos primeiramente garantir que a máquina de desenvolvimento cumpra com os requisitos mínimos listados no guia de instalação da ferramenta. Depois de instaladas as dependências necessárias, deve-se baixar os arquivos do Flutter e extraí-los em uma pasta que não necessite de permissão do administrador da máquina, como por exemplo “C:\Flutter” no caso do sistema operacional Windows. Se desejado, pode-se adicionar o comando “flutter” em uma variável de ambiente da máquina para que ele fique disponível na ferramenta de linha de comando do sistema.

Depois de executar os passos anteriores, deve-se utiliza o comando “flutter doctor” para que seja feita uma checagem da instalação, ao final é apresentado um relatório de outros *softwares* que devem ser instalados ou tarefas a serem

realizadas para o correto funcionamento da ferramenta, este relatório é demonstrado na figura à seguir:

Figura 20 - Relatório gerado pelo comando “flutter doctor”.



Fonte: Autor. (2019)

Para o desenvolvimento de aplicações para a plataforma Android, o Flutter depende de uma instalação completa da IDE Android Studio para contemplar diversas dependências da plataforma, entretanto, pode-se utilizar outros editores para escrever o código da aplicação como por exemplo o Visual Studio Code em conjunto com sua extensão para Flutter, esta configuração será a utilizada neste projeto.

Para poder executar e testar é necessário ter um aparelho com a versão do Android 4.1 ou superior, também é necessário habilitar a configuração “Opções do desenvolvedor” no dispositivo e instalar o driver USB do Google no Windows.

Após serem realizadas todas as configurações, pode-se criar um novo projeto do Flutter à partir da ferramenta de linha de comando com o comando “flutter create” seguido do nome do projeto, como no exemplo:

Figura 21 - Criação e execução da aplicação Flutter.

```
C:\Users\Robson\Desktop\TCC\PartyU_Flutter>flutter create partyu_flutter
Creating project partyu_flutter...
  partyu_flutter\.gitignore (created)
  partyu_flutter\.idea\libraries\Dart_SDK.xml (created)

C:\Users\Robson\Desktop\TCC\PartyU_Flutter>cd partyu_flutter

C:\Users\Robson\Desktop\TCC\PartyU_Flutter\partyu_flutter>flutter run
Launching lib\main.dart on SM J530G in debug mode...
Initializing gradle...                                          69,7s
Resolving dependencies...
  273,8s (!)
Running Gradle task 'assembleDebug'...
Running Gradle task 'assembleDebug'... Done                     109,8s
Built build\app\outputs\apk\debug\app-debug.apk.
Installing build\app\outputs\apk\app.apk...                      14,7s
```

Fonte: Autor. (2019)

Todos os arquivos de uma aplicação de demonstração serão criados. A figura 22 mostra o arquivo principal da aplicação que é criado, o “main.dart”, assim como no “App.js” do React Native, é a partir dele que construiremos nossa aplicação.

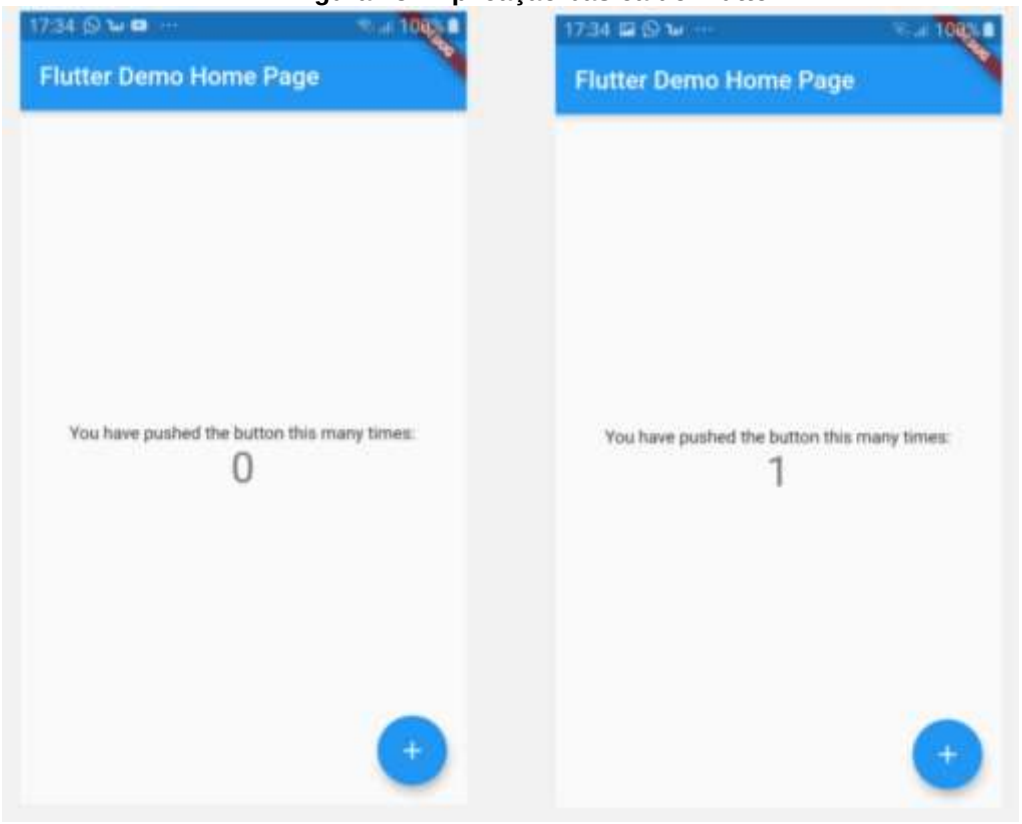
Figura 22 - Arquivo “main.dart”.

```
lib > main.dart > ...
1   import 'package:flutter/material.dart';
2
3   void main() => runApp(MyApp());
4
5   class MyApp extends StatelessWidget {
6     @override
7     Widget build(BuildContext context) {
8       return MaterialApp(
9         title: 'Flutter Demo',
10        theme: ThemeData(
11          primarySwatch: Colors.blue,
12        ), // ThemeData
13        home: MyHomePage(title: 'Flutter Demo Home Page'),
14      ); // MaterialApp
15    }
16  }
17
18 > class MyHomePage extends StatefulWidget {···
26
27 > class _MyHomePageState extends State<MyHomePage> {···
```

Fonte: Autor. (2019)

Após o término do processo de criação da aplicação base, pode-se ir até o diretório da aplicação criada e executar o comando “flutter run” para que a ferramenta rode a aplicação no dispositivo, este processo demora alguns minutos na primeira vez em que é executado mas ele cria um mecanismo de reload que possibilita a sua atualização em milésimos de segundo. A figura 23 demonstra a aplicação base gerada após a realização de todo o processo.

Figura 23 - Aplicação básica do Flutter.



Fonte: Autor. (2019)

4.3 RF01 - LISTAGEM DE EVENTOS

A funcionalidade que criaremos diz que devemos exibir eventos para todos os usuários que utilizarem a aplicação, ela será utilizada principalmente para a tela inicial da aplicação que de acordo com os *wireframes* deve exibir itens de informações dos eventos. Vamos partir das aplicações base que as ferramentas criaram e adicionar os trechos de código necessários para construir as funcionalidades, modificando os arquivos de acordo com o necessário.

4.3.1 RF01 CONSTRUÍDO NO REACT

Para começar a construção do primeiro requisito funcional da aplicação utilizando o *framework* React Native, o arquivo “App.js”, localizado na pasta raiz do projeto criado no processo de instalação, será modificado como demonstrado na figura 24.

Figura 24 - Novo conteúdo do arquivo “App.js”.

Fonte: Autor. (2019)

Na primeira linha do novo código está sendo utilizada a declaração “import”, introduzida ao JavaScript pela especificação ECMAScript 6 de 2015, com ela importamos o código da biblioteca React e disponibilizamos acesso direto a sua classe “Component”.

Na sequência estamos importando o componente que irá fazer o controle da navegação entre as páginas da aplicação. Por padrão o React Native não disponibiliza um controlador de navegação, entretanto é possível utilizar plugins criados pela comunidade open source do *framework* com esta finalidade, estes são instalados através do NPM. Para a construção deste projeto, optou-se pela utilização do plugin “React Navigation”, visto que este é indicado atualmente na documentação oficial da ferramenta.

Para instalar o React Navigation iremos executar os comandos demonstrados na figura 25, estando dentro da ferramenta de linha de comando do computador e na pasta do projeto. Issa fará com que o plugin e suas dependências sejam instalados no projeto.

Figura 25 - Comandos para instalação do React Navigation.

```
npm install --save react-navigation
npm install --save react-native-gesture-handler
npm install --save react-navigation-stack
```

Fonte: Autor. (2019)

Após a instalação ser concluída, criaremos o componente “AppContainer”, o nome de seu arquivo vai ser “index.js” e ele ficará dentro do diretório “./src/router”, isso possibilita que a sua importação seja feita como na linha 2 de “App.js”, sem escrever o nome do arquivo.

O conteúdo do arquivo criado é apresentado na próxima figura, nele importamos a biblioteca React, duas funções responsáveis por criar o gerenciador de navegação, além de outros dois componentes customizados.

Figura 26 - O componente “AppContainer”.

src > router > JS index.js > ...

```
1  import React from 'react';
2  import { createAppContainer } from 'react-navigation';
3  import { createStackNavigator } from 'react-navigation-stack';
4
5  import AppHeader from '../components/AppHeader'
6  import Homepage from './routes/Homepage'
7
8  const AppNavigator = createStackNavigator({
9    Homepage: {screen: Homepage,},
10 }, {
11   initialRouteName: 'Homepage',
12   defaultNavigationOptions: ({navigation}) => ({
13     headerTintColor: '#FFF',
14     headerStyle: {
15       backgroundColor: '#2196f3',
16     },
17     headerTitle: () => <AppHeader navigation={navigation}/>,
18   }),
19 });
20
21 const AppContainer = createAppContainer(AppNavigator);
22
23 export default AppContainer;
```

Fonte: Autor. (2019)

Criamos uma variável do tipo “const” com o nome de “AppNavigator”, no JavaScript, este tipo de variável não pode sofrer uma reatribuição de valores. Ela recebe como valor o resultado da função importada “createStackNavigator”, esta

função recebe dois objetos literais como parâmetros e cria um mecanismo de empilhamento das telas por onde o usuário passou.

Seu primeiro parâmetro é um objeto contendo referências para os componentes de tela da aplicação, a tela inicial terá o nome “Homepage” e seu valor é um objeto onde a propriedade “screen” (tela) é uma referência ao componente importado na linha 6.

O segundo parâmetro de “createStackNavigator” é um objeto onde pode-se passar configurações do navegador da aplicação, estamos passando as seguintes instruções:

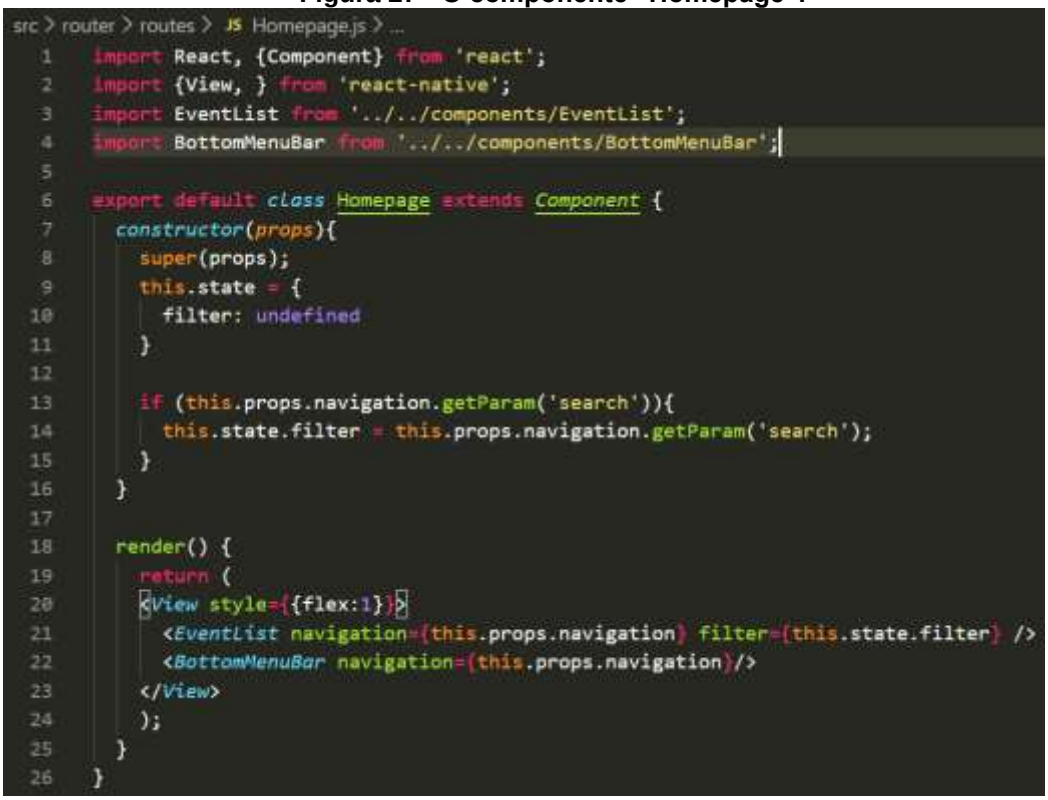
- **initialRouteName**: indica qual o componente inicial da aplicação;
- **defaultNavigationOptions**: esta propriedade pode receber uma função que retorna um objeto com as definições para as configurações de navegação, aqui utilizamos uma “arrow function” do ES6 que está recebendo um parâmetro e fazendo uma atribuição por desestruturação, ou “destructuring assignment” outra funcionalidade nova da especificação, para obter o valor do atributo “navigation” do objeto que será passado como parâmetro da função;
    - **headerTintColor**: define a coloração dos itens que compõem o cabeçalho de navegação;
    - **headerStyle**: é um objeto com definição de estilos para serem aplicados no elemento container do cabeçalho de navegação, definindo cor de fundo, espaçamento e etc;
    - **headerTitle**: essa propriedade permite a utilização de um componente customizado para ser o cabeçalho da aplicação, ela recebe uma função que retorna o componente desejado, neste caso está sendo retornado o componente "AppHeader", que foi importado na linha 5, passamos o parâmetro “navigation” como atributo do componente para que se possa controlar a navegação de dentro dele.

Depois de definidas as configurações do gerenciador da navegação, a função “createStackNavigator” terá seu resultado final armazenado na constante “AppNavigator”, utilizaremos este valor logo na sequência para criar outra constante chamada “AppContainer”, seu valor será o resultado da função

“crateAppContainer”, para quem passamos “AppNavigator”. Containers são responsáveis por gerenciar o estado do aplicativo e vincular seu navegador de nível superior ao ambiente do aplicativo. Na última linha do arquivo é definida como a exportação padrão o componente armazenado na constante “AppContainer”, que agora servirá como componente raiz de nossa aplicação no arquivo “App.js”.

Para dar prosseguimento ao desenvolvimento da funcionalidade, será criado um arquivo chamado “Homepage.js”, ele ficará dentro do diretório “src/router/routes/”, esta pasta irá conter os arquivos para as telas da aplicação. A figura 27 mostra o conteúdo do arquivo criado.

Figura 27 - O componente “Homepage”.



Fonte: Autor. (2019)

Neste arquivo definimos como exportação padrão a classe que estamos criando para “Homepage”, ela estenderá da classe “Component” da biblioteca React, ambos importados na linha 1. Na linha 7 definimos um construtor para a classe, ele recebe como parâmetro as propriedades passadas para o componente no momento de sua construção, na sequência chamamos o método construtor da classe “Component” também passando “props”, através do comando “super(props)”.

Na linha 9 do arquivo utilizamos “this.state”, este é um objeto representacional que armazena o estado do componente, quando utilizado no construtor podemos colocar valores no estado inicial do componente. Aqui estamos colocando apenas uma informação como estado do componente “Homepage”, o valor de “filter” irá indicar se existe alguma busca ativa em nossa aplicação e vai criar uma filtragem nos eventos exibidos, seu valor inicial padrão será “undefined”.

A construção do componente “AppContainer”, que foi realizada na etapa anterior, permite que todos os componentes que foram definidos como telas no primeiro parâmetro passado para “createStackNavigator”, tenham acesso a uma propriedade “navigation”. Essa propriedade será uma referência para o gerenciador de navegação criado anteriormente.

Na linha 13 estamos verificando o gerenciador de navegação recebeu um parâmetro de nome “search” no momento em que a navegação para a tela “Homepage” ocorreu, fazemos isto através da utilização da função “getParam” disponibilizada pelo gerenciador criado. Caso o resultado da verificação seja positivo atribuímos o valor desse parâmetro para nossa variável de estado “filter”.

Na linha 18 do arquivo estamos definindo a função render de nosso componente, seu retorno é uma estrutura JSX composta por três elementos, são eles:

- **View**: este é um componente é disponibilizado pelo React Native que serve para ser utilizado como container para outros componentes, em sua propriedade “style” estamos passando um objeto de estilização inline onde dizemos que sua propriedade “flex” tem valor 1;
- **EventList**: este componente representa a listagem de eventos da nossa aplicação, para ele passamos duas propriedades “navigation” e “filter”, o primeiro é uma referência para o objeto de navegação, o segundo é o valor que deve ser aplicado como filtro na lista;
- **BottomMenuBar**: este é o componente customizado que ficará na parte inferior da aplicação, quando o usuário não estiver autenticado ele irá exibir um botão para que seja possível que se efetue login, caso ele esteja autenticado são exibidas outras opções de navegação.

Agora veremos como foi realizada a construção do arquivo do componente de listagem dos eventos “EventList.js”. A class “EventList” possui diversos métodos

que realizam a construção de partes do componente, aqui veremos apenas os pontos que possuem maior relevância para serem mostrados, os métodos desta classe que veremos são:

- **componentDidMount**: este é um dos métodos do ciclo de vida dos componentes do React;
- **_fetchEvents**: este método comunica-se com a API e recupera informações;
- **_getEventList**: cria a lista de eventos;
- **_getEventItem**: cria a estrutura de cada item da lista.

Figura 28 - O componente “EventList”.

Fonte: Autor. (2019)

Primeiramente vamos abordar dois pontos fora os métodos citados, primeiramente a constante de nome “styles”, vista na linha 135. Esta constante está recebendo o resultado de “Stylesheet.create”, este método é disponibilizados pelo React Native e permite a criação de uma abstração de folha CSS a partir de um conjunto de objetos de configurações de estilo, cada propriedade do objeto passado

é mapeado para um dos objetos de estilo e pode ser utilizado dentro das marcações JSX, através da sintaxe vista nas ilustrações que serão apresentadas.

O segundo ponto que precisamos ver é o estado inicial de nosso componente, pois ele será modificado durante sua construção. No construtor do componente definimos o estado inicial de nosso componente com duas propriedades, “items”, que recebe um array inicialmente vazio, e “error”, que possui o valor “false”. Na linha 17 estamos verificando se o componente recebeu uma propriedade chamada “events”, e em caso positivo atribuímos seu valor para a propriedade “items” do estado do componente.

Figura 29 - Método construtor da classe “EventList”.

```
10    constructor(props){
11      super(props);
12      this.state = {
13        items: [],
14        error: false,
15      };
16
17      if(this.props.events){
18        this.state.items = this.props.events;
19      }
20    }
```

Fonte: Autor. (2019)

O primeiro método da classe “EventList” que veremos é o “componentDidMount”, este método é executado após a primeira renderização do componente, portanto será executado depois do método “render”, onde estamos verificando se possuímos eventos para mostrar e de acordo com a resposta mostrar um ícone de carregamento. Ao ser renderizado pela primeira vez o componente não terá itens de eventos em seu estado, e então irá mostrar um “ActivityIndicator”, indicando que um carregamento está sendo realizado. Em “componentDidMont” verificamos se temos itens passados via “props”, e caso não tenhamos executa o método “_fetchEvents”.

Figura 30 - O método de vida “componentDidMont”.

Fonte: Autor. (2019)

O método “_fetchEvents” utiliza a chamada de “this.setState” para fazer uma atualização do estado do componente, este método irá realizar uma operação assíncrona portanto, diremos que inicialmente não há nenhum erro com o componente. A função “setState” está recebendo como parâmetro uma outra função, que retorna um objeto contendo as propriedades de “state” que devem ser atualizadas, dentro dessa função temos acesso ao estado atual do componente através do parâmetro “state”.

Figura 31 - Mudança de estado no método “_fetchEvents”.

Fonte: Autor. (2019)

Na sequência utilizamos uma classe utilitária que criamos para lidar com requisições HTTP assíncronas, o método “PartyUAPI.getEvents” retorna um objeto “Promise” do JavaScript, que quando é finalizada com sucesso verifica se é

necessário filtrar os eventos e retorna uma lista de eventos atualizada. Quando um erro ocorre na operação, ele é capturado pelo método “catch” e a propriedades “error” do estado do componente é atualizada para o valor “true”.

Em “_getEventList” iniciamos a construção da listagem de eventos, neste método criamos uma lista de elementos utilizando o componente “FlatList”, ele cria uma lista de elementos dispostos um abaixo do outro e cria uma barra de *scroll* automaticamente. Em seu atributo “data” passamos os dados da lista que criaremos, a propriedade “keyExtractor” recebe uma função que cria uma “chave” para cada elemento da lista, ela é importante para questões de performance. Por último, a propriedade “renderItem” está recebendo uma função que será executada para a construção de cada um dos elementos da listagem, nele verificamos se devemos criar um ou dois eventos no item da lista, e retornamos um novo item de lista chamando a função “_getRow”, esta função apenas realiza uma verificação se deve renderizar um ou dois itens de evento, chamando “_getEventItem” de acordo.

Figura 32 - O método “_getEventItem”.

```
_getEventItem(item) {
  return (
    <View style={styles.eventItem}>
      <View style={styles.eventItemImage}>
        <Image
          style={{flex: 1, width: undefined, height: undefined}}
          source={require('../../assets/event_image.jpg')} />
      </View>
      <View style={styles.info}>
        <View style={{paddingHorizontal: 5, paddingVertical: 10,}}>
          <Text style={styles.eventName}>{item.name}</Text>
          <Text style={styles.description}>{
            item.description.length > 70
              ? item.description.substr(0, 60) + '...'
              : item.description }</Text>
          <Text style={{fontWeight: '900'}}>{
            formatDateBR(item.date_occurrence)
          }</Text>
        </View>
        <TouchableOpacity
          style={styles.eventButton}
          onPress={() => {
            this.props.navigation.push('EventDetails', {itemData: item});
          } }>
          <Text style={{color:"#fff", fontWeight: 'bold'}}>Detalhes</Text>
        </TouchableOpacity >
      </View>
    </View>
  );
}
```

Fonte: Autor. (2019)

Dentro de “_getEventItem” utilizamos uma combinação de elementos “View” e “Text” estilizados para exibir informações do evento, dois componentes novos são importantes neste método, o componente “Image” e o “TouchableOpacity”. O primeiro serve para a exibição de recursos gráficos, para isso devemos apenas utilizar o componente dentro do JSX e passar como atributo “source”, o caminho do arquivo que desejamos utilizando a função “require” do Node.JS. O segundo componente cria um elemento que executa uma função de “callback” quando tocado pelo usuário., para isso, passamos como valor do atributo “onPress” que irá utilizar o método “push” do objeto gerenciador da navegação da aplicação, direcionando o usuário para a tela com detalhes do evento selecionado.

Assim concluímos a criação da funcionalidade de listagem de eventos utilizando o *framework* React Native.

4.3.2 RF01 CONSTRUÍDO NO FLUTTER

Para inicializar a funcionalidade no Flutter, o código existente no arquivo “main.dart” será substituído pelo seguinte:

Figura 33 - Novo código do arquivo “main.dart”.

```
lib > main.dart > ...
1  import 'package:flutter/material.dart';
2  import 'package:partyu_flutter/src/widgets/Homepage.dart';
3
4  void main() => runApp(MyApp());
5
6  class MyApp extends StatelessWidget {
7    @override
8    Widget build(BuildContext context) {
9      return MaterialApp(
10       title: 'PartyU Flutter',
11       home: Homepage(),
12     ); // MaterialApp
13   }
14 }
```

Fonte: Autor. (2019)

Aqui estamos executando as seguinte operações:

- **Linha 1**: Faz a importação da biblioteca responsável por *widgets* do Material Design;
- **Linha 2**: Faz a importação da *widget* customizada que criaremos para a página inicial de nossa aplicação;
- **Linha 4**: A função “main” utiliza uma notação de “flecha” para executar a aplicação através do método “runApp” em apenas uma linha;
- **Linha 6**: Definimos a classe raiz da aplicação, a classe “MyApp” vai estender a classe “StateLessWidget”, portanto ela será uma *widget* também;
- **Linha 7**: Dentro da classe “MyApp” fazemos a sobrescrita do método “build” herdado de “StateLessWidget”, este método deve retornar uma *widget* que descreve como ela deve ser exibida.

Ao ser iniciada a aplicação, automaticamente o método “build” da classe “MyApp” será executado, dando início ao programa. O retorno do método “build” é uma *widget* disponibilizada pelo biblioteca “material.dart” do Flutter, a “MaterialApp” cria um mecanismo de navegação e possibilita a criação dos elementos básicos de uma aplicação que utiliza o “Material Design”, recebe como atributo o título da aplicação e o atributo “home” que deve receber uma *widget* com a rota inicial de nossa aplicação, a *widget* customizada “Homepage” será exibida ao início da aplicação.

A estrutura do Flutter incentiva a composição e reutilização de *widgets* para a construção de *layouts*. Seguindo este ideal, a “Homepage” irá importar outras três *widgets* customizadas, “BottomMenuBar”, “EventList” e “PartyUAppBar”, além da biblioteca de elementos “material”. A classe “PartyUAPI.dart” é uma classe de métodos utilitários para fazer requisições HTTP para a API RESTful do sistema.

Figura 34 - Importações do arquivo “Homepage.dart”.

Fonte: Autor. (2019)

Como pode ser notado na figura 34, a classe “Homepage” é uma stateless *widget*, seu construtor na linha 11 define um parâmetro nomeado “searchValue” a ser recebido, que será associado a variável definida como propriedade na linha 9. Esta forma de definir parâmetros é uma sintaxe da linguagem Dart que possibilita a definição de parâmetros opcionais para um método, podendo este possuir opcionalmente tipagem.

Figura 35 - A classe “Homepage”.

Fonte: Autor. (2019)

A sobrescrita do método “build” na linha 14 irá retornar a construção de uma *widget* “Scaffold”, esta é uma *widget* da biblioteca “material” que cria a estrutura básica de um *layout* baseado no Material Design, provendo algumas API’s para facilitar o trabalho do desenvolvedor, pode-se por exemplo, exibir uma mensagem informativa para o usuário através de uma "snackbar". Essa *widget* está recebendo dois atributos, “appBar” e “body”, eles representam o conteúdo da barra superior e o corpo da página, respectivamente.

Para criar uma barra superior customizada e reutilizável, construiremos uma *widget* chamada “PartyUAppBar”, ela será colocada como elemento filho (child) de “PreferredSize”, esta é uma *widget* que indica um tamanho preferido para seu

conteúdo sem restringi-lo. No Flutter, criamos uma propriedade de tamanho utilizando a classe “Size”, podemos utilizar diferentes métodos para criar configurações de altura e largura, utilizamos o método “fromHeight” definindo uma altura de 56 pixels, sem definir largura.

O corpo da página inicial da aplicação é composto da listagem de eventos e por uma barra de navegação inferior, que deve ser visível apenas se o usuário estiver autenticado no aplicativo.

Para conseguir reproduzir a disposição desses elementos como no *layout* projetado, vamos criar uma estrutura com outros elementos de *layout* disponibilizados pelo *framework*, são eles:

- **SingleChildScrollView**: cria um container que pode conter uma barra de rolagem caso seu conteúdo seja maior do que o espaço disponível;
- **Container**: é uma *widget* de conveniência que pode receber atributos de cor, *padding*, tamanho, entre outros;
- **Column**: recebe um array de *widgets* e os dispõe verticalmente no espaço disponível;
- **Row**: recebe um array de *widgets* e os dispõe horizontalmente no espaço disponível;
- **Expanded**: expande seu conteúdo para ocupar todo o espaço disponível.

A *widget* “Expanded” recebe como filho a *widget* customizada “EventList”, esta pode receber dois parâmetros nomeados, “searchValue” e “eventsFuture”. O primeiro será o valor recebido no construtor da classe “Homepage”, o segundo é uma “Future” disponibilizada pela classe “PartyUAPI”, Futures são o mecanismo de programação assíncrona da linguagem Dart, elas funcionam de maneira semelhante às Promises do JavaScript, podendo-se inclusive utilizá-las em conjuntos com os comandos “async” e “await” ou “then” e “catchError”.

Figura 36 - Exemplo de uma “Future”.

```
static Future<List> getEvents() async {
  final response = await _makeGet('events', false);
  if (response.statusCode == 200){
    var usersList = (jsonDecode(response.body) as List)
      .map((data) => Event.fromJson(data))
      .toList();

    return usersList;
  }
}
```

Fonte: Autor. (2019)

Para construirmos efetivamente a *widget* que lista eventos, criaremos um arquivo chamado “EventList.dart”, nele constam duas classes, “EventList” que estende de “StatefulWidget”, e “EventListState” que é uma subclasse de “State” e irá armazenar o estados da *widget* “EventList”.

Figura 37 - Importações do arquivo “EventList.dart”.

```
lib > src > widgets > EventList.dart > ...

1   import 'package:flutter/material.dart';
2   import 'package:partyu_flutter/src/Utils/StringHelper.dart';
3   import 'package:partyu_flutter/src/models/Event.dart';
4   import 'EventDetails.dart';
5
6 > class EventList extends StatefulWidget {···
15
16 > class EventListState extends State<EventList> {···
308
```

Fonte: Autor. (2019)

Neste arquivo importaremos as classes “material.dart”, “StringHelper.dart” que é uma classe com funções úteis para trabalhar com dados do tipo string, “Event.dart” que é a classe modelo de eventos, “EventDetails.dart” que é a tela de exibição de detalhes do evento.

A classe “EventList” possui dois atributos inicializados por seu construtor e apenas um método, “createState” cria uma instância de “EventListState” passando as propriedades “searchValue” e “eventsFuture” como parâmetros.

A classe “EventListState” vai conter diversos métodos construídos para a criação da *widget* de listagem de eventos, inclusive o método “build”. Dentre estes

métodos revisaremos apenas aqueles que trazem novos conceitos sobre a ferramenta ou pontos importantes a serem destacados, são eles:

- **_buildEventList**: utiliza a “Future” de eventos para construir a listagem ou mostrar outros resultados de acordo com o estado do processamento assíncrono;
- **_buildList**: recebe como parâmetro os dados da lista de eventos, executa o método “_verifyActiveSearch” para verificar se existe alguma busca ativa e em caso positivo filtra os elementos da lista;
- **_buildEventListItem**: cria os pares de eventos que são exibidos em cada linha da listagem;
- **_getItemImage**: cria a foto ilustrativa apresentada em cada evento;
- **_getEventAction**: cria o botão que quando pressionado direciona o usuário para a página de detalhes do evento em questão.

Figura 38 - A classe “EventList”.

Fonte: Autor. (2019)

O método “_buildEventList” inicia a construção da lista de eventos da nossa aplicação, seu valor de retorno é uma “Widget”. A *widget* retornada por nosso método é uma “FutureBuilder”, elas são *widgets* que constroem-se baseadas nos estados de uma “Future”, a qual passamos como atributo e uma função anônima que realizará a lógica de construção das possíveis *widgets* que retornaremos. A função anônima está tendo acesso a dois parâmetros, “context” e “snapshot”, o primeiro contém uma referência para o escopo superior, o segundo armazena informações sobre o estado da “Future”. De acordo com o conteúdo de “snapshot” nossa função irá retornar uma *widget* diferente, se a propriedade “hasData” for um booleano verdadeiro, construiremos a lista de eventos com os dados de “snapshot” chamando o método “_buildList”. Se o teste anterior não for bem sucedido verificamos se existe um erro na variável “snapshot”, se houver mostramos uma mensagem de erro, caso nenhuma das condições sejam satisfeitas significa que a “Future” ainda não está completa, portanto mostraremos um indicador de carregamento.

Figura 39 - O método “_buildEventList”.

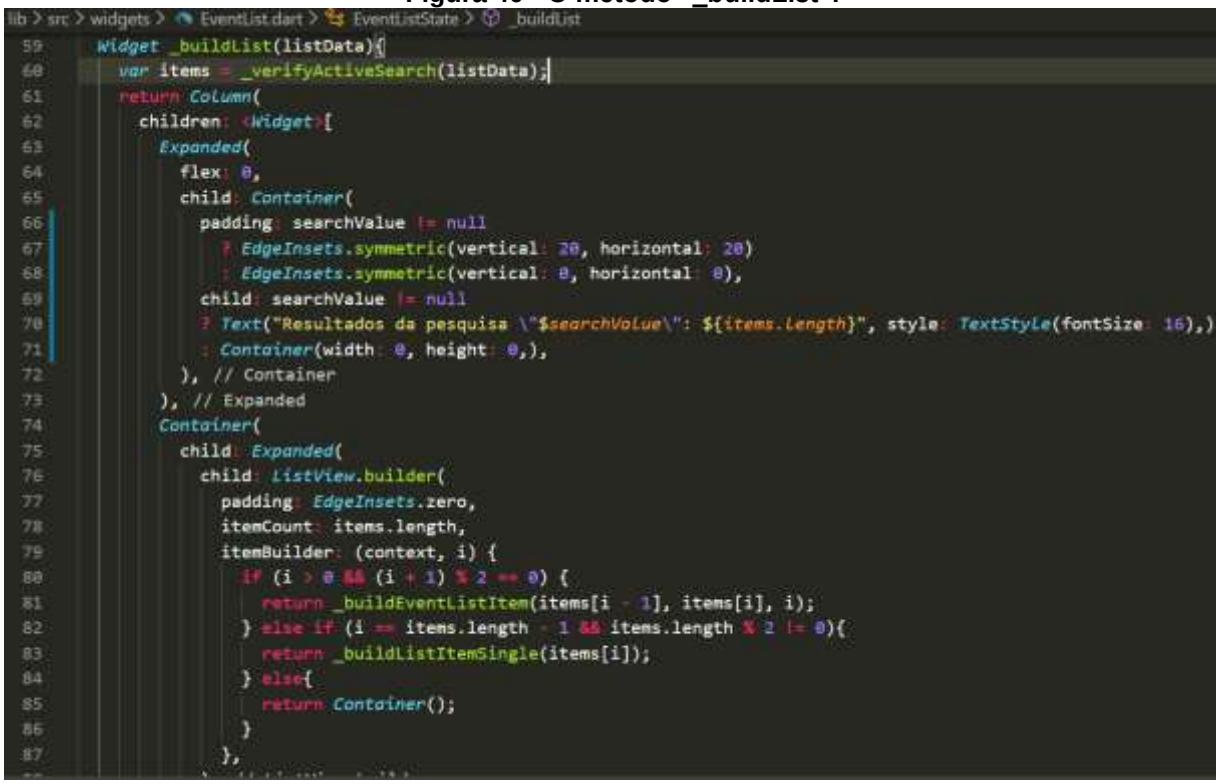
Fonte: Autor. (2019)

Existem novas *widgets* e alguns pontos que podem ser explicados neste trecho:

- **SizedBox**: essa *widget* define um tamanho específico para o seu elemento filho;
- **MediaQuery**: esta classe permite o acesso a informações sobre o dispositivo, além de acesso a algumas preferências de *layout* do usuário, aqui a estamos utilizando para recuperar o tamanho vertical do *display* do dispositivo e utilizá-lo para calcular a altura das *widgets*;
- **Text**: essa *widget* permite a exibição e a estilização de textos;
- **MainAxisAlignment**: esta classe fornece valores para a propriedade de mesmo nome, seu objetivo é definir como os elementos filhos da *widget* serão distribuídos pelo espaço disponível no eixo principal, em uma *widget* “Row” o eixo principal é o horizontal, em uma “Column” é o vertical;
- **CrossAxisAlignment**: esta classe fornece valores para a propriedade de mesmo nome, seu objetivo é definir como os elementos filhos da *widget* serão distribuídos pelo espaço disponível no eixo secundário, em uma *widget* “Row” o eixo secundário é o vertical, em uma “Column” é o horizontal;
- **CircularProgressIncicator**: essa *widget* cria um indicador de progresso circular que gira indicando que está havendo um carregamento.

Agora iremos abordar alguns pontos do *framework* que aparecem dentro da função “_buildList”, o primeiro deles é a utilização de “EdgeInsets.symetric” como valor para o atributo “padding”, esta classe do Flutter permite a criação de deslocamentos nas quatro direções, pode ser utilizada como valor para o atributo “margin” também.

Figura 40 - O método “_buildList”.



Fonte: Autor. (2019)

O outro elemento novo que o método que estamos analisando possui é a *widget* “ListView”, ela possibilita a criação de uma lista composta por outras *widgets*, que são dispostas linearmente uma embaixo da outra. Ela cria e gerencia o *scroll* automaticamente de acordo com o tamanho de seus elementos, seu método “builder” recebe os parâmetros que serão utilizados na sua construção, aqui usamos “itemCount” para alertar previamente qual o tamanho da lista que iremos criar, e a propriedade “itemBuilder” que recebe uma função anônima que irá realizar a lógica de construção dos elementos. Nessa função recebemos o contexto do superior e o índice atual da lista, utilizamos o índice para colocar dois elementos customizados dentro de cada item da lista, esses elementos vão exibir as informações dos eventos e serão construídos pelo método “_buildEventListItem”, o qual veremos a seguir.

Figura 41 - O método "_buildEventListItem".

Fonte: Autor. (2019)

No método "_buildEventListItem" recebemos três parâmetros, dois objetos contendo as informações dos eventos que serão renderizados e o *index* do elemento atual do item da lista. Utilizamos uma "ListTile" para exibir os itens da lista, e em seu atributo "title" criamos uma estrutura utilizando um conjunto de *widgets* para exibir dois eventos lado a lado. Para estilizar os elementos com informações dos eventos foram criados alguns métodos na classe "EventList", destes apenas "_getItemImage" e "_getEventAction" trazem novos elementos importantes de serem citados.

O método "_getItemImage" retorna a imagem do evento, utilizando o atributo "decoration" na *widget* "Container", podemos criar uma "BoxDecoration" para criar estilos customizados para a *widget*, como por exemplo bordas, sombras e imagens.

Figura 42 - O método "_getItemImage".

Fonte: Autor. (2019)

Criamos uma "DecorationImage" e a passamos como atributo "image", essa classe realiza a criação de uma imagem para ser utilizada especificamente dentro de uma "BoxDecoration", por sua vez, "DecorationImage" está recebendo dois atributos, "image" e "fit". O primeiro atributo cria um "AssetImage", passando como parâmetro o caminho do arquivo em relação a pasta raiz do projeto, para que o *framework* possa renderizar a imagem. A classe "BoxFit", passada como atributo "fit", é mais um enum que o Dart disponibiliza, ele vai definir como a imagem será exibida dentro de seu elemento container, utilizando o seu valor "cover" fará com que a imagem seja esticada até que encoste nas duas margens de de seu container.

Estes foram os pontos relevantes de serem citados dentro do método "_getItemImage", para finalizar veremos os alguns tópicos abordados dentro do método "_getEventAction".

Figura 43 - O método “_getEventAction”.

Fonte: Autor. (2019)

O método “_getEventAction” tem o objetivo de criar o botão que será exibido na parte inferior de cada item da lista de eventos, quando o usuário pressioná-lo, ele será direcionado para uma tela com os detalhes do evento.

Criamos um container com cor de fundo azul, utilizando a classe enum “Colors” e seu valor “blue”, este enum possui valores que representam as cores da paleta de cores do Material Design. Dentro desse container, colocamos uma widget "FlatButton", ela possibilita a criação de uma superfície que reage ao ser tocada. Como parâmetros estamos dando para "onPressed" uma função que será executada sempre que houver um toque no botão, como "child" estamos passando uma *widget* que será exibida como texto do botão.

Dentro da função anônima passada em "onPressed" estamos utilizando o método "push" da classe "Navigator", esta classe possibilita a criação de uma pilha de *widgets* representando as telas pelas quais o usuário passou, o método que estamos utilizando vai adicionar uma nova *widget* no topo da pilha, ele recebe dois parâmetros, o primeiro é o contexto de onde estamos invocando o método, o segundo cria uma transição nativa da plataforma entre as telas através de uma rota para onde o usuário será levado.

Utilizamos a classe "MaterialPageRoute" para que seja realizada uma transição nativa entre às telas, ela será diferente dependendo de em qual plataforma

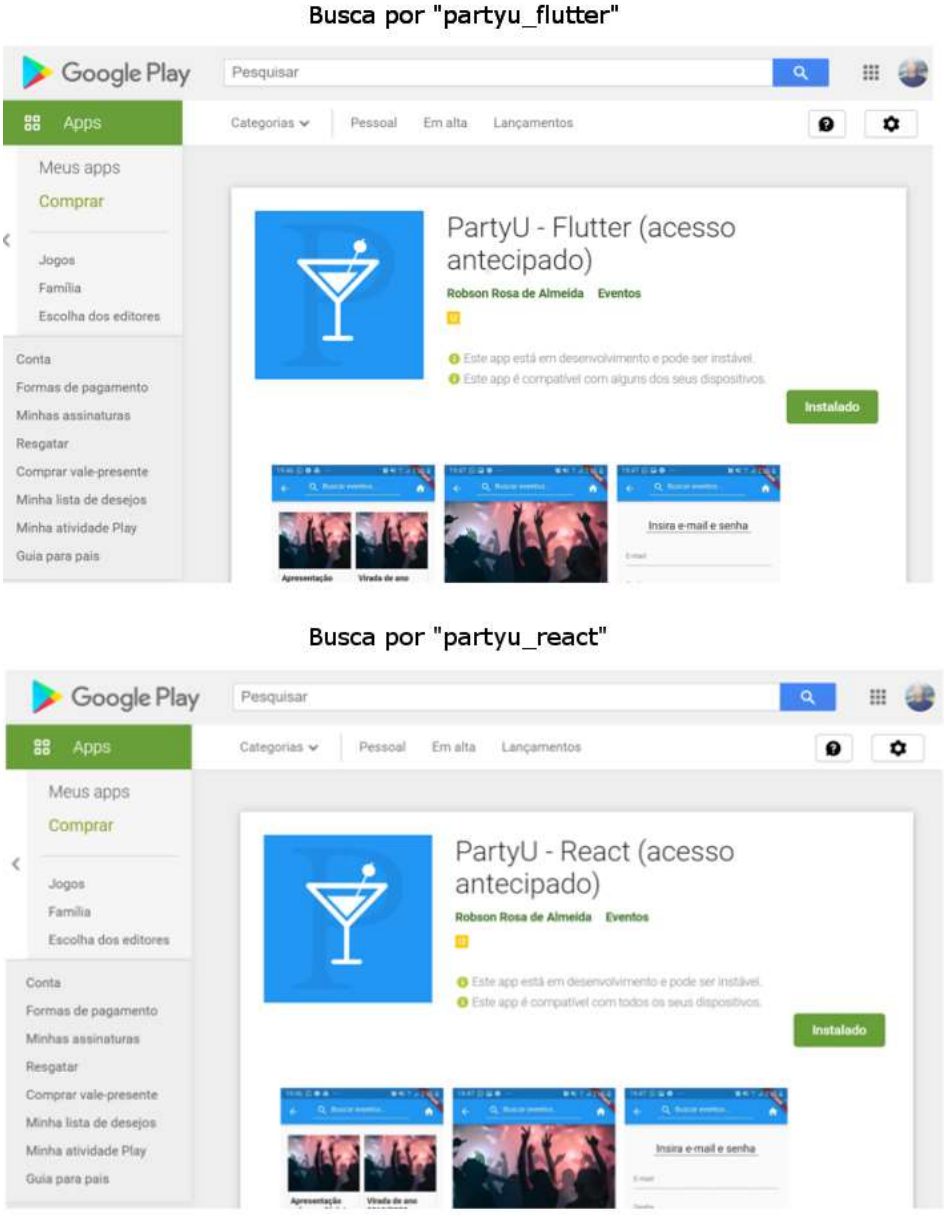
a aplicação está sendo executada. Para que o efeito possa ser realizado é necessário passarmos como parâmetro uma função "builder", seu valor de retorno deve ser a *widget* para onde o usuário será levado.

Assim concluímos a criação da funcionalidade de listagem de eventos utilizando o *framework* Flutter.

5. COMPARAÇÃO ENTRE AS FERRAMENTAS

Após a realização do desenvolvimento de todas as funcionalidades do protótipo, foi realizada a preparação e submissão das aplicações construídas com cada *framework* para a loja de aplicativos Play Store, como versão beta. Ambas podem ser encontradas para *download* realizando uma pesquisa pelos termos “partyu_react” ou “partyu_flutter”, como mostrado na Figura 44.

Figura 44 - Aplicações publicadas na loja de aplicativos Play Store.



Fonte: Google. (2019)

As próximas subseções irão descrever como foi o desempenho das ferramentas nos pontos de avaliação levantados, ao final é apresentada uma figura ilustrativa com o resultado final das comparações realizadas.

5.1 QUAL FERRAMENTA POSSUI MAIOR PRATICIDADE PARA SER UTILIZADA?

De acordo com o experimento realizado no ambiente de desenvolvimento deste trabalho, a ferramenta que detém maior praticidade de utilização é o framework React Native. Como foi mostrado na Seção 4.2.1, a necessidade de configuração inicial da ferramenta é mínima, mostrando-se mais simples e rápida de ser realizada durante a comparação.

O processo de instalação e criação de uma nova aplicação com a ferramenta Flutter mostrou-se mais complicado, pois necessita de uma maior atenção e tempo para configurar todas as suas dependências. Além do fato de que a ferramenta depende da instalação de uma terceira ferramenta consideravelmente grande, o Android Studio.

5.2 QUAL A FERRAMENTA QUE POSSUI MELHOR DOCUMENTAÇÃO?

As documentações das ferramentas abordadas possuem disparidade em relação à quantidade e qualidade de informação oficial disponível.

A documentação do *framework* React Native se mostrou mais enxuta e amigável à novos usuários, com textos simplificados e a possibilidade de modificar e testar exemplos *online* no site da documentação.

O *framework* Flutter possui uma documentação muito rica de informação, além de detalhar as API's e componentes tanto de sua camada superior quanto da inferior, existem diversos vídeos explicativos da ferramenta, disponíveis no site da documentação e em seu canal oficial na rede social YouTube.

Apesar do React Native possuir muita informação proveniente da comunidade *open source*, para esta comparação serão considerados apenas conteúdos de documentos oficiais. Portanto, a documentação que possui melhores conteúdos oficiais e consequentemente obtém melhor performance neste quesito da comparação é o Flutter.

5.3 QUAL FERRAMENTA QUE TEM MENOR CURVA DE APRENDIZADO INICIAL?

Com o resultado obtido com a experiência realizada, pode-se presumir que a ferramenta que apresentará menor curva de aprendizado para iniciantes é o Flutter.

Por utilizar a linguagem Dart, a ferramenta dispõe de muita flexibilidade no desenvolvimento do código, deixando a ferramenta acessível para quem tem experiência com linguagens fortemente tipadas ou dinamicamente tipadas, sendo necessário apenas uma noção básica de orientação a objetos. O sistema de composição de *widgets* da ferramenta é bastante intuitivo, além disso, o *framework* também possui várias API's que funcionam de forma similar às de outras linguagens. Sem experiência prévia com o *framework* ou com a linguagem Dart, foi possível realizar a construção do protótipo em 6 dias de trabalho.

O *framework* React Native, que é baseado na biblioteca React, possui uma sintaxe e metodologia de construção de aplicações bastante diferente em relação ao JavaScript puro, o que pode tornar a ferramenta inicialmente confusa até mesmo para desenvolvedores experientes com JavaScript. A construção do protótipo da aplicação com a ferramenta React Native levou 7 dias de trabalho, sendo que já existia algum conhecimento prévio na linguagem JavaScript e no *framework* React Native.

5.4 QUAL A FERRAMENTA QUE ENTREGA A MELHOR EXPERIÊNCIA DURANTE O DESENVOLVIMENTO?

Para julgar qual ferramenta entregou a melhor experiência durante o período de desenvolvimento foram avaliados 3 pontos:

- Qual das ferramentas apresentou as maiores facilidades e necessitou do menor número de dependências extras?
- Qual ferramenta apresenta a sintaxe mais agradável?
- Qual ferramenta apresentou o melhor ambiente de emulação

Durante o desenvolvimento do projeto a ferramenta Flutter foi a que entregou o maior número de facilidades, foi necessário instalar menos extensões desenvolvidas por terceiros do que em relação ao framework React Native.

Ao todo, foi necessário instalar 2 dependências extras no projeto com Flutter:

- O pacote “http” que facilita a manipulação de requisições HTTP;
- O pacote “flutter_secure_storage” que possibilita o armazenamento de dados no dispositivo de forma segura.

O projeto realizado com o *framework* React Native necessitou da instalação de quatro dependências extras, além das suas subdependências:

- “React Navigation” que realiza o controle da navegação na aplicação;
- “SecureStore” para armazenar dados no dispositivo de maneira segura;
- “React Native DateTimePicker” que cria componentes para seleção de data e hora;
- “React-Native-SnackBar-Component” que cria componentes que podem exibir mensagem para os usuários;

O *framework* Flutter obteve melhor desempenho neste quesito comparativo, por já incluir o mecanismo de navegação, além dos componentes para seleção de datas e exibição de mensagens em sua biblioteca padrão, não havendo assim necessidade de utilizar código desenvolvido por terceiros para a criação destas funcionalidades.

Quanto a sintaxe utilizada para o desenvolvimento, o Flutter mostrou-se mais difícil de manter o código organizado. A metodologia de criação de *widget* pode fazer com que componentes complexos necessitem de várias linhas de código para serem construídos, o que pode deixar a estrutura do componente confusa, caso não seja feita uma separação adequada. Por sua vez, o React Native possui uma estrutura de criação de componentes que facilita mais a organização, muito devido à sintaxe do JSX. É possível, por exemplo, isolar as configurações de estilo e marcação dos componentes em arquivos separados, tornando os arquivos mais enxutos, melhorando a organização.

O último ponto da avaliação de qual ferramenta entrega a melhor experiência de desenvolvimento foi qual das ferramentas possui o melhor emulador para testes da aplicação. Durante o desenvolvimento, a ferramenta React Native apresentou

diversas instabilidade na emulação da aplicação através do aplicativo “Expo”, constantes falhas fatais apresentadas pela ferramenta faziam necessária a reinicialização do servidor de desenvolvimento com uma frequência muito maior do que em relação ao Flutter, diminuindo a produtividade do programador. Isto fez o *framework* Flutter obter o melhor desempenho final neste quesito comparativo.

5.5 QUAL DAS FERRAMENTAS ENTREGOU O MELHOR RESULTADO FINAL?

Foi possível atingir o resultado visual e funcional almejado para o protótipo com ambas as ferramentas. Não foi possível constatar grandes diferenças visuais ou de performance entre as duas versões finais do protótipo.

A grande diferença percebida ficou por parte do tamanho do arquivo final de instalação gerado por cada uma das ferramentas. O tamanho do arquivo de instalação da aplicação desenvolvida com React Native na loja de aplicativos "Play Store" varia de 23,9 a 25,2 MB, enquanto a aplicação desenvolvida com o Flutter gerou um "App Bundle" que varia de 6,83 a 7,14 MB de acordo com o dispositivo em que é instalado. Para a plataforma mobile esta diferença é bastante significativa, portanto como não houveram outras discrepâncias, esse foi o ponto decisivo neste quesito da comparação.

Figura 45 - O resultado final das comparações realizadas no trabalho.

	A ferramenta que possui maior praticidade para ser utilizada?	A ferramenta que possui melhor documentação?	A ferramenta que tem menor curva de aprendizado inicial?	A ferramenta que entrega melhor experiência durante o desenvolvimento?	A ferramentas entregou o melhor resultado final?
					
					

Fonte: Autor. (2019)

6 . CONCLUSÃO

Ao longo do desenvolvimento deste trabalho acadêmico, estudamos os conceitos que envolvem a construção de aplicações multiplataforma, a fim de explicar quais metodologias de trabalho podem ser adotadas para que se consiga disponibilizar uma mesma aplicação em diferentes plataformas, sem a necessidade de duplicação do código fonte.

Um mesmo protótipo de aplicações foi construído com duas das ferramentas de desenvolvimento multiplataforma mais populares da atualidade, o *framework* React Native e o *framework* Flutter, com o objetivo de realizar uma comparação entre as duas ferramentas e avaliar qual entrega maiores vantagens ao desenvolvedor durante o estágio de construção da aplicação.

A conclusão que pudemos obter de acordo com as avaliações realizadas é que, no contexto deste projeto, a ferramenta que melhor auxiliou o desenvolvedor durante o período de desenvolvimento foi o *framework* Flutter, devido principalmente à consistência de emulação da aplicação e os diversos componentes já construídos que a ferramenta dispõe para seu utilizador.

Por limitações da pesquisa, o projeto não pode ser submetido para a loja de aplicativos para aparelhos com sistema operacional IOS, portanto as conclusões obtidas sobre a versão final da aplicação teve como embasamento apenas a versão para aparelhos Android.

A partir deste trabalho podem ser realizadas novas pesquisas focadas em outras plataformas além da *mobile*, e com maiores testes de performance e usabilidade das aplicações construídas com estes dois *frameworks*.

REFERÊNCIAS

BIØRN-HANSEN, Andreas; GRØNLI, Tor-morten; GHINEA, Gheorghita. A Survey and **Taxonomy of Core Concepts and Research Challenges in Cross-Platform Mobile Development**. Acm Comput. Surv., [s. L.], v. 51, n. 5, p.1-34, 2018.

BRACHA, Gilad, **The Dart programming language**, Boston: Addison-Wesley, 2016.

CORAZZA, Paulo Victor. **Um aplicativo multiplataforma desenvolvido com flutter e NoSQL para o cálculo da probabilidade de apendicite**. 2018. lume.ufrgs.br, https://lume.ufrgs.br/handle/10183/190147.

CETINER, Gültekin; ABURAS, H., **Development of a Cross-Platform Artificial Neural Network Component for Intelligent Systems**, Journal of King Abdulaziz University-Engineering Sciences, v. 16, n. 2, p. 97–113, 2005.

DOCUMENTAÇÃO **Dart**. 2019. Disponível em: <https://dart.dev>. Acesso em: 15 set. 2019.

DOCUMENTAÇÃO **Flutter**. 2019. Disponível em: <https://flutter.dev/docs>. Acesso em: 01 jul. 2019.

DOCUMENTAÇÃO **React**. 2019. Disponível em: <https://reactjs.org>. Acesso em: 14 set. 2019.

DOCUMENTAÇÃO **React Native**. 2019. Disponível em: <https://facebook.github.io/react-native>. Acesso em: 14 set. 2019.

EL-KASSAS, Wafaa S. et al, **Taxonomy of Cross-Platform Mobile Applications Development Approaches**, Ain Shams Engineering Journal, v. 8, n. 2, p. 163–190, 2017.

HANSSON, N.; VIDHALL, T. **Effects on performance and usability for cross-platform application development using React Native**. Linköping, Switzerland: Linköping University, 16 jun. 2016.

Introdução ao JavaScript, MDN Web Docs, disponível em: <https://developer.mozilla.org/pt-PT/docs/Web/JavaScript/Guia/Introdu%C3%A7%C3%A3o_ao_JavaScript>, acesso em: 8 set. 2019.

PC Magazine Encyclopedia. **application framework Definition from PC Magazine Encyclopedia**, https://www.pcmag.com/encyclopedia. Acessado 8 de dezembro de 2019.

RAUSCHMAYER, Axel, **Speaking JavaScript: an in-depth guide for programmers**, First edition. Sebastopol, CA: O'Reilly Media, Inc, 2014.

SEVERANCE, Charles, **JavaScript: Designing a Language in 10 Days**, Computer, v. 45, n. 2, p. 7–8, 2012.