UNIVERSIDADE FEDERAL DO CEARÁ
INSTITUTO UNIVERSIDADE VIRTUAL
BACHARELADO EM SISTEMAS E MÍDIAS DIGITAIS

PEDRO FAÇANHA PEIXOTO

MOITA REDONDA: UM ESTUDO SOBRE O DESENVOLVIMENTO CROSS-PLATFORM NO ESTÍMULO DE PRODUÇÃO E COMÉRCIO DE ARTESANATO NO CEARÁ

FORTALEZA – CEARÁ
2020

PEDRO FAÇANHA PEIXOTO

MOITA REDONDA: UM ESTUDO SOBRE O DESENVOLVIMENTO CROSS-PLATFORM NO ESTÍMULO DE PRODUÇÃO E COMÉRCIO DE ARTESANATO NO CEARÁ

Relatório técnico apresentado no curso de Sistemas e Mídias Digitais da Universidade Federal do Ceará, como requisito parcial à obtenção do título de bacharel em Sistemas e Mídias Digitais. Área de concentração: Computação.

Orientador: Prof. Dr. Windson Viana de Carvalho

FORTALEZA – CEARÁ

2020

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Sistema de Bibliotecas
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

P431m Peixoto, Pedro Façanha.
Moita redonda : um estudo sobre o desenvolvimento cross-platform no estímulo de produção e comércio de artesanato no Ceará / Pedro Façanha Peixoto. – 2020.
63 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Instituto UFC Virtual, Curso de Sistemas e Mídias Digitais, Fortaleza, 2020.
Orientação: Prof. Dr. Windson Viana de Carvalho.

1. Cross-platform. 2. React native. 3. JavaScript. 4. Empreendedorismo social. I. Título.

CDD 302.23

PEDRO FAÇANHA PEIXOTO

MOITA REDONDA: UM ESTUDO SOBRE O DESENVOLVIMENTO CROSS-PLATFORM NO ESTÍMULO DE PRODUÇÃO E COMÉRCIO DE ARTESANATO NO CEARÁ

Relatório técnico apresentado no curso de Sistemas e Mídias Digitais da Universidade Federal do Ceará, como requisito parcial à obtenção do título de bacharel em Sistemas e Mídias Digitais. Área de concentração: Computação.

Aprovada em:

BANCA EXAMINADORA

Prof. Dr. Windson Viana de Carvalho (Orientador)
Instituto Universidade Virtual
Universidade Federal do Ceará – UFC

Prof. Dr. José Gilvan Rodrigues Maia
Instituto Universidade Virtual
Universidade Federal do Ceará - UFC

Prof. Dr. Leonardo Oliveira Moreira
Instituto Universidade Virtual
Universidade Federal do Ceará - UFC

A todos que possuem sede pelo conhecimento.

AGRADECIMENTOS

Ao Prof. Dr. Windson Viana de Carvalho, pela sua orientação.

Aos professores participantes da banca examinadora Prof. Dr. José Gilvan Rodrigues Maia e o Prof. Dr. Leonardo Oliveira Moreira, pelo tempo, colaborações e sugestões e por terem contribuído pelo meu interesse na atual carreira que sigo.

Aos amigos formados durante o curso, que conseguiram me ajudar das mais diferentes maneiras, dentro e fora da UFC.

A minha família que permitiu e incentivou minha formação como pessoa e profissional, oferecendo tudo que poderia dentro do seu alcance.

A Amanda Maria Fonseca Correia por ser meu ponto de equilíbrio durante o período pandêmico em que vivemos, me animando em momentos que senti desesperança e dificuldade de continuar.

“Se o conhecimento pode criar problemas, não é através da ignorância que podemos solucioná-los.”

(Asimov, Isaac)

RESUMO

O presente relatório técnico descreve o processo de desenvolvimento do aplicativo Moita Redonda, cujo foco é o suporte de *e-commerce* da comunidade de mesmo nome da cidade de Cascável-CE. O aplicativo *cross-platform* visa a divulgação e promoção do comércio de artesanato de barro criado na comunidade. Ele foi desenvolvido usando o *framework React Native*, na linguagem *JavaScript*. Moita Redonda oferece uma plataforma de loja que conecta os compradores diretamente aos produtores dos itens, por meio de negociações via *WhatsApp*, que é chamado no processo de finalização de compra. Isto garante uma liberdade para negociação na venda e mantém em contato a comunidade de artesãos com seus clientes. Apresenta-se neste relatório técnico o aplicativo Moita Redonda, os conceitos que lideraram sua criação, o processo de implementação, a arquitetura utilizada e os contextos de uso. Também são discutidos aspectos técnicos da escolha da plataforma de desenvolvimento e os passos para uso do *framework React Native*.

Palavras-chave: Cross-Platform. React Native. JavaScript. Empreendedorismo social.

ABSTRACT

The following technical report presents the development process of the Moita Redonda application, whose focus is the e-commerce support of a community in the city of Cascavel, in the state of Ceará, Brazil. The cross-platform application aims at disseminating and promoting the commerce of clay crafts created in the community. It was developed using the React Native framework, in the JavaScript language. Moita Redonda offers a store platform that connects buyers directly to the producers of the items via WhatsApp, which is called during the checkout process. This guarantees freedom of negotiation and keeps the community of artisans in touch with their customers. This technical report presents the Moita Redonda application, the concepts that led its creation, the implementation process, the architecture used and the contexts of use. Also discussed are technical aspects of choosing the development platform and the steps for using the *React Native framework*.

Keywords: Cross-Platform. React Native. JavaScript. Social entrepeneurship.

LISTA DE FIGURAS

Figura 1 – Visita do grupo Varal à comunidade de Moita redonda 14
Figura 2 – Diagrama estrutural do Redux 24
Figura 3 – Fluxo do Design Thinking 31
Figura 4 – Modelo conceitual 33
Figura 5 – Arquitetura do sistema 34
Figura 6 – Protótipo produzido em Adobe XD 36
Figura 7 – Identidade visual 36
Figura 8 – Fluxo de autenticação 38
Figura 9 – Fluxo de loja 39
Figura 10 – Administração de produtos 41
Figura 11 – Pasta raiz 44
Figura 12 – Pasta *src* 45
Figura 13 – Pasta *pages* 49
Figura 14 – Pasta *routes* 51
Figura 15 – Pasta *helpers* 52
Figura 16 – Pasta *components* 54
Figura 17 – Painel do Firestore 59

LISTA DE CÓDIGOS-FONTE

Código-fonte 1 – Exemplo de JSX 20
Código-fonte 2 – Comando para instalar o yarn 42
Código-fonte 3 – Comando para instalar react-native-cli 43
Código-fonte 4 – Comando para inicializar um projeto de React Native 43
Código-fonte 5 – package.json 45
Código-fonte 6 – App.js 47
Código-fonte 7 – AuthContext.js 49
Código-fonte 8 – Navigation.js 49
Código-fonte 9 – routesConstants.js 50
Código-fonte 10 – cartReducer.js 56
Código-fonte 11 – rootReducer 57
Código-fonte 12 – store 57

LISTA DE ABREVIATURAS E SIGLAS

API	*Interface de Programação de Aplicação*
CLI	*Command Line Interface*
CRUD	*Create, Read, Update, Delete*
CSS	*Cascading Stylesheet*
HTML	*Hypertext Markup Language*
JS	*JavaScript*
JSON	*JavaScript Object Notation*
NPM	*Node Package Manager*
URL	*Uniform Resource Locator*
SDK	*Kit de Desenvolvimento de Software*
SMD	*Sistemas e Mídias Digitais*
SQL	*Structured Query Language*
UI	*Interface de usuário*
UX	*Experiência de usuário*

SUMÁRIO

1	**INTRODUÇÃO**	13
1.1	**Problemática**	13
1.2	**Objetivos do trabalho**	16
1.3	**Organização do trabalho**	16
2	**FUNDAMENTAÇÃO TEÓRICA**	18
2.1	**Desenvolvimento Cross-Platform**	18
2.2	***ECMAScript* 6 (ES6) e *JavaScript***	19
2.3	**React**	19
2.3.1	***Componentes***	20
2.3.2	***JSX***	20
2.3.3	***Princípio da imutabilidade***	21
2.3.4	***React Hooks***	21
2.4	***React Native***	22
2.5	***Redux***	23
2.6	***Firebase***	24
2.6.1	***Firebase Auth***	25
2.6.2	***NoSQL***	25
2.6.3	***Firebase Firestore***	26
2.6.4	***Firebase Storage***	27
3	**METODOLOGIA**	28
3.1	***Design Thinking***	28
3.2	**Divisão da equipe**	28
3.3	**Reunião de dados e demandas da comunidade**	29
3.4	**Prototipação**	30
4	**CONCEITUAÇÃO DO APLICATIVO**	32
4.1	**Problema**	32
4.2	**Proposta de solução**	32
4.3	**Descrição do sistema**	33
4.4	**Interface**	34
5	**PRODUTO COMPLETO**	37
5.1	**Dados quantitativos**	37

5.2 **Fluxo de autenticação** 37
5.3 **Loja** 38
5.4 **Administração de produtos** 39
6 **PROCESSO DE DESENVOLVIMENTO** 42
6.1 **Inicializando o projeto** 42
6.2 **Arquivos e estrutura de pastas** 43
6.2.1 ***Pasta src*** 44
6.2.2 ***package.json*** 45
6.2.3 ***App.js*** 46
6.2.4 ***Telas*** 47
6.2.5 ***Navegação*** 48
6.2.6 ***Hooks e Helpers*** 51
6.2.7 ***Componentes*** 53
7 **MANIPULAÇÃO E FONTE DE DADOS** 55
7.1 ***Redux*** 55
7.1.1 ***Implementação no aplicativo*** 55
7.1.1.1 *Actions* 55
7.1.1.2 *Reducers* 56
7.1.1.3 *Store* 57
7.2 ***Firebase*** 58
7.2.1 ***Uso do Firestore no aplicativo de Moita Redonda*** 58
8 **VERSIONAMENTO DE CÓDIGO** 60
9 **CONSIDERAÇÕES FINAIS** 61
REFERÊNCIAS 63

1 INTRODUÇÃO

A arte do barro é um tipo de atividade milenar que há mais de três mil anos está presente entre diversas culturas da civilização humana. Esse tipo de arte, no Brasil, é algo muito representativo para a cultura popular, principalmente, em regiões mais remotas como no interior das regiões Norte e Nordeste (MACHADO, 2003).

Esse tipo de artesanato é uma herança deixada pelos povos indígenas, que utilizavam o barro para produzirem utensílios domésticos como gamelas, tigelas, alguidares, potes, etc. Os materiais eram modelados de acordo com a necessidade de uso e customizados por meio de pinturas com tintas fortes e coloridas inspiradas na natureza.

Hoje em dia, os artesãos de barro são artistas anônimos espalhados pelo norte e pelos sertões e litorais brasileiros. Dentro deste contexto, este trabalho abordará a vila de Moita Redonda, uma comunidade marcada por esse tipo de artesanato. Ela se localiza no interior do estado do Ceará, no município de Cascavel, no litoral leste.

Essa comunidade conta com um grupo de apoio vindo da UFC, o Varal, que auxilia o desenvolvimento social e econômico da região por meio de projetos de *design*. A Figura 1 ilustra uma visita do grupo a Moita Redonda.

Uma dessas soluções é o aplicativo Moita Redonda, que foi desenvolvido em uma parceria entre o grupo Varal e alunos de Sistemas e Mídias Digitais para a disciplina de Projeto Integrado 2, que faz parte da grade curricular do 6º semestre.

1.1 Problemática

A equipe dos alunos de SMD, denominada de *Unicode*, realizou no segundo semestre de 2019 um trabalho de pesquisa e produção de protótipos de design para entregar um modelo de um produto digital. Ele contava com exposição de artesanatos, geração de pedidos, canais de comunicação com a associação dos artesãos de Moita Redonda e um portal de gerenciamento de loja para os trabalhadores. No entanto, a equipe *Unicode* enfrentou um problema comum que dificulta o desenvolvimento do aplicativo de Moita Redonda: a dificuldade de desenvolver uma aplicação para os diversos tipos de *smartphones* existentes, seus diferentes sistemas operacionais e limitações de hardware em certos casos.

Neste sentido, o desenvolvimento de aplicações móveis evolui muito rapidamente. Mesmo assim, os desenvolvedores e engenheiros de software se deparam com diversos desafios

Figura 1 – Visita do grupo Varal à comunidade de Moita redonda

Fonte: acervo da equipe *Unicode*

para criação e execução de projetos, envolvendo custos, tempo, usabilidade e manutenção das aplicações criadas. As inevitáveis mudanças mercadológicas impactam nas metodologias de desenvolvimento e com isso, mudam completamente os ambientes de desenvolvimento estabelecidos. No momento da escrita deste trabalho, as plataformas *mobile* se polarizam nos sistemas operacionais *Android*[1] e *iOS*[2]. Eles seguem princípios opostos no desenvolvimento de aplicações nativas, cada um adotando linguagens e lógicas diferentes em seus processos de desenvolvimento e distribuição.

[1] Android <https://www.android.com/intl/pt-BR_br/>

[2] iOS <https://www.apple.com/br/ios/ios-14/>

Felizmente, existem empresas e grupos *open source* que criaram soluções para minimizar o esforço de duplo desenvolvimento de forma a atender tanto o *Android* quanto o *iOS*. Essas soluções são denominadas de *Cross-platforms*. Surgiram *frameworks* e ambientes que conseguem construir aplicações de maneira eficiente, rápida e de baixo custo que conseguem rodar nas duas plataformas a partir de uma única versão de um código. Contudo, é inegável que diferente dos aplicativos criados nas linguagens nativas, a performance, visual e acesso aos componentes do dispositivo são inferiores e mais complicados nas soluções *Cross-platforms*. Geralmente, a facilidade e agilidade no processo de desenvolvimento compensam esses inconvenientes.

Dentre essas ferramentas que podem auxiliar os desenvolvedores a desenvolver uma aplicação multiplataforma, a escolhida para ser analisada neste trabalho de conclusão de curso será o *framework React Native*[3]. Este *framework* foi criado pelo *Facebook* em 2015, sendo derivado de outro *framework* focado em desenvolvimento web, chamado de *ReactJS*[4]. Esses ambientes tomam como base a linguagem *Javascript*[5] e conseguem tirar proveito de diversas das suas vantagens e facilidades. Hoje em dia, o *React Native* está bem consolidado no mercado e sempre é uma solução utilizada por diversas empresas.

Após tomarem conhecimento dessa ferramenta, a equipe *Unicode* decidiu adotar o *framework* como principal ambiente de desenvolvimento da versão final da aplicação baseada nos protótipos, por diversos motivos, dentre eles estava a facilidade em replicar elementos visuais de vários *Design Systems*, os diversos pacotes que podem ser instalados para integração com hardware e APIs dos *smartphones* e a agilidade que a linguagem *JavaScript* proporciona para seus desenvolvedores.

Além de todos essas necessidades de construção de *frontend*, o projeto também necessitava de algum suporte para administrar as autenticações, o armazenamento de imagens, a administração de banco de dados, entre outros serviços necessários para garantir segurança e um bom uso para os usuários.

Tendo em vista essa nova necessidade, foi escolhido o serviço *Firebase*[6], da *Google*. Este serviço oferece diversas facilidades aos seus assinantes, tendo plano gratuito, que cobre tudo que qualquer simples aplicação poderia precisar. O pacote de serviços é disponibilizado tanto para web como para *Android* e *iOS* e em combinação com o *React Native*, é possível usar

[3] Core Components and Native Components <https://reactnative.dev/docs/intro-react-native-components>
[4] The History of React.js on a Timeline <https://blog.risingstack.com/the-history-of-react-js-on-a-timeline/>
[5] JavaScript <https://developer.mozilla.org/pt-BR/docs/Web/JavaScript>
[6] Firebase <https://firebase.google.com/>

as três versões simultaneamente, porém com uma única chave de serviço apenas.

O *Firebase* foi utilizado então em três partes principais do aplicativo de Moita Redonda: a parte de autenticação e criação de conta, funcionando para clientes e administradores dos produtos; a etapa de adição e remoção de produtos, sendo possível enviar com fotos e adicionar descrições sobre os autores dos itens de barro e por último a parte de loja, onde os clientes podem ver todas as informações, adicionar itens ao carrinho e obterem acesso a um canal de informação com a associação dos artesãos de Moita Redonda.

1.2 Objetivos do trabalho

Tendo em vista os conceitos apresentados sobre o universo do artesanato de barro, o contexto em que está inserida a comunidade de Moita Redonda e sobre o desenvolvimento *cross-platform*, o **objetivo deste trabalho** é apresentar a construção conceitual do aplicativo de Moita Redonda, na área de *design* e sua arquitetura, e também relatar o processo do desenvolvimento da versão final desse aplicativo.

Os objetivos específicos são de apresentar as características do *React Native*, mostrando casos de uso diferentes a depender da situação apresentada; demonstrar a utilização de bibliotecas comumente utilizadas por desenvolvedores experientes; ilustrar o padrão de *design* utilizado para estruturar o aplicativo de Moita Redonda e demonstrar a integração desse aplicativo em *React Native* com os serviços do *Firebase*.

1.3 Organização do trabalho

O restante deste trabalho está organizado em mais oito capítulos. No Capítulo 2, conceitua-se o termo *cross-platform* e as tecnologias *React* e *React Native*, assim como os serviços do *Firebase*.

No Capítulo 3, a metodologia utilizada para dividir tarefas, desenvolver telas e gerar o código para o aplicativo de Moita Redonda é apresentada e detalhada de maneira breve.

Posteriormente, no Capítulo 4, é abordada a conceituação completa do aplicativo em questão e o Capítulo 5 relata o resultado compilado para as possíveis plataformas de uso, sendo eles ilustrados e analisados neste capítulo.

Já no Capítulo 6, é apresentada as etapas do processo de desenvolvimento e a estruturação de arquivos de uma aplicação em *React Native* em conjunto com *Firebase*, seguido

do Capítulo 7 que demonstra como a aplicação trata os dados de entrada e saída.

No Capítulo 8, é apresentado como foi feito o versionamento do código da aplicação, descrevendo o processo utilizado pelos desenvolvedores.

Concluindo o trabalho, no Capítulo 9, são resumidos os principais resultados do longo do trabalho e também uma série de sugestões de trabalhos futuros são descritas.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo contém o resumo das teorias escolhidas para desenvolver este trabalho. Entre elas existem tecnologias, ferramentas e conceitos que estão descritos de maneira focada para condizer com o contexto geral do trabalho e da aplicação desenvolvida.

2.1 Desenvolvimento Cross-Platform

Segundo El-Kassas et al. (2017), o desenvolvimento de aplicações móveis é um caso especial do desenvolvimento de software pelo fato de seus desenvolvedores terem que considerar aspectos diferentes como: tempo curto de desenvolvimento, capacidades dos dispositivos móveis, mobilidade, tamanho de tela, design e navegação na interface do *app*, segurança e privacidade dos usuários.

Mesmo com todas as restrições e dificuldades, o verdadeiro desafio dessa área é como desenvolver uma aplicação *mobile* uma vez e fazer com que ela consiga rodar em múltiplas plataformas, para economizar tempo, custos e esforços de desenvolvimento.

Os tipos de aplicações *mobile* são: Aplicações Web, Aplicações Nativas e Aplicações Híbridas. As aplicações híbridas combinam as aplicações web e as aplicações nativas. Elas são desenvolvidas utilizando as tecnologias usadas para desenvolver um *Web App* mas é renderizado da mesma maneira de um *App* Nativo. Esse tipo de aplicação consegue acessar as funcionalidades do dispositivo como câmera e sensores e essas funcionalidades são expostas ao aplicativo pela camada de abstração (APIs *Javascript*).

Assim, chegamos nas soluções cross-platform que auxiliam os desenvolvedores a escreverem código apenas uma vez e rodar suas aplicações em diferentes plataformas. Xanthopoulos e Xinogalos (2013) classificam os aplicativos *cross-platform* como: Aplicações Web, Aplicações Híbridas, Aplicações Interpretadas e Aplicações Geradas. Existem diversas ferramentas de desenvolvimento *cross-platform* e o público-alvo dessas ferramentas difere de acordo com as finalidades de cada ferramenta, como auxiliar estudantes a aprender desenvolvimento mobile, médicos a customizar aplicações para seus pacientes, ajudar leigos a desenvolverem aplicações simples e desenvolvedores profissionais a produzirem aplicações mais complexas.

2.2 *ECMAScript* 6 (ES6) e *JavaScript*

ECMAScript[1] é uma linguagem de programação orientada a objeto, originalmente desenvolvida para ser utilizada como uma linguagem de *script*, mas se tornou vastamente utilizada como uma linguagem de programação multi propósitos. Uma linguagem de programação *script* é uma linguagem utilizada para manipular, customizar e automatizar as instalações de um sistema existente, os quais possuem funcionalidades úteis disponibilizadas por meio de uma interface de usuário e a linguagem de *script* é utilizada para expor tal funcionalidade para o controle do programa. Nesse modo o sistema existente tem como função hospedar um sistema de objetos e instalações, que completam as capacidades de uma linguagem *script*.

Designado inicialmente para ser uma linguagem *script* Web, entregando um mecanismo para dar vida às páginas web nos navegadores, *ECMAScript* hoje em dia é utilizado para entregar capacidades diversas e em variados ambientes, escalando para o espectro completo da programação. A especificação Ecma (2015) descreve o seguinte:

> O padrão JavaScript é ECMAScript. Desde 2012, todos os navegadores modernos possuem suporte total ao ECMAScript 5.1. Navegadores mais antigos suportam pelo menos ECMAScript 3. Em 17 de Junho de 2015, a ECMA International publicou a sexta versão do ECMAScript, que é oficialmente chamado de ECMAScript 2015, e foi inicialmente conhecido como ECMAScript 6 ou ES6. Desde então, as especificações do ECMAScript são lançadas anualmente. Essa documentação faz referência à última versão de referência, que atualmente é a ECMAScript 2018. A especificação ECMAScript utiliza terminologia e sintaxe que podem ser desconhecidos para um programador JavaScript. Embora a descrição da linguagem possa ser diferente no ECMAScript, a linguagem em si continua sendo a mesma. JavaScript suporta todas as funcionalidades descritas na especificação ECMAScript.

2.3 React

React é um *framework* da linguagem *JavaScript* desenvolvida pela empresa de redes sociais privada *Facebook*, que em 2013, tomando inspirações como XHP, HTML entre outras linguagens, bibliotecas e *frameworks*, abriu o código na conferência *JS ConfUS*[2]. Com o passar dos anos o *framework*, sua comunidade e produtos criados no ambiente foram se expandindo até alcançar a estabilidade.

Este *framework* é constituído de diversos conceitos complexos, que demandam esforço para serem compreendidos por completo, dentre eles temos o conceito de componentes,

1 ECMAScript® 2015 Language Specification <http://www.ecma-international.org/ecma-262/6.0/>
2 JS ConfUS 2013 <http://2013.jsconf.us/>

que possuem um ciclo de vida único e que também são estruturados por meio de JSX, estes componentes geralmente são estruturados em classes, e mais recentemente tivemos a introdução de componentes funcionais que simplificaram o processo de desenvolvimento.

As subseções seguintes detalharão conceitos comumente citados no universo de desenvolvimento *ReactJS* e presentes na estrutura do trabalho do aplicativo de Moita Redonda.

2.3.1 Componentes

Segundo a documentação do ReactJS[3], Componentes permitem você dividir a UI em partes independentes, reutilizáveis e pensar em cada parte isoladamente. Conceitualmente, componentes são como funções *JavaScript*. Eles aceitam entradas arbitrárias (chamadas "*props*") e retornam elementos *React* que descrevem o que deve aparecer na tela. É possível fazer referência a outros componentes dentro da saída dos componentes criados, sendo possível utilizar a mesma abstração para qualquer nível de detalhe. Uma lista, um botão, um formulário, entre tantos tipos de componentes são normalmente expressos em *ReactJS*.

Cada componente possui *props*, que são parâmetros passados na sua declaração. Esses parâmetros precisam servir apenas como leitura, independente de como foi declarado o componente, seja por classe ou função. Derivado dessa restrição, existe a seguinte regra na documentação do *React*: "Todos os componentes *React* tem que agir como funções puras em relação ao seus *props*".

2.3.2 JSX

No código-fonte 1, a sintaxe de *tags* não é uma *string*, nem *HTML* e sim JSX[4], que é uma extensão de sintaxe para *Javascript* que serve para produzir elementos *React*.

Código-fonte 1 – Exemplo de JSX

```
1 const element = <h1>Hello, world!</h1>;
```

O *React* adota o fato de que a lógica de renderização é inerentemente acoplada com outras lógicas de UI: como eventos são manipulados, como o *state* muda com o tempo e como os dados são preparados para exibição.

[3] Componentes e Props <https://pt-br.reactjs.org/docs/components-and-props.html>
[4] Introduzindo JSX <https://pt-br.reactjs.org/docs/introducing-jsx.html>

É possível escrever qualquer tipo de expressão *Javascript* dentro do *JSX*, como por exemplo um cálculo, uma atribuição de valor, uma desestruturação de objeto ou uma manipulação de lista. Após a compilação, todas se tornam chamadas de funções normais que retornam objetos *Javascript*.

2.3.3 Princípio da imutabilidade

Um conceito que será facilmente encontrado ao programar em *React* é o de imutabilidade. No universo da programação, uma variável pode ser chamada de imutável quando seu valor não pode ser alterado após a declaração.(COPES, 2018)

As Strings são exemplos notáveis de variáveis imutáveis, ao "mudá-las", na realidade está sendo criada uma nova *String* e a atribui ao mesmo nome de variável. O mesmo ocorre com *Arrays* e Objetos: no caso dos *Arrays*, ao adicionar um novo item, um novo *Array* é criado por meio de concatenação do *Array* antigo com o novo item; já no caso dos objetos, eles nunca são atualizados mas copiados antes de serem mudados. Esses casos se aplicam no *React* em várias situações.

Existem as seguintes motivações para que esse conceito tenha esse nível de presença e importância:

- As mutações podem ser centralizadas, melhorando as capacidades de depuração e reduzindo as incidências de erros;
- O código se torna mais simples, com uma aparência mais limpa e fácil de entender. Por não esperar uma função mudar algum valor sem que o desenvolvedor saiba, o código se torna previsível, um ponto positivo. Quando uma função não altera objetos mas retorna um novo objeto, ela é chamada de função pura, tipo consoante às funções do *Framework*;
- O código se torna otimizado, pois *JavaScript* é mais rápido ao trocar uma referência de um objeto antigo por um objeto totalmente novo do que fazer uma mutação em um objeto já existente, garantindo mais performance.

2.3.4 React Hooks

Os *Hooks*[5] foram introduzidos na versão 16.8 do *React*, vindo ao público na *React Conf 2018*, apresentados por Sophie Alpert e Dan Abramov, com uma demonstração de como refatorar uma aplicação os utilizando.

[5] Introdução aos Hooks <https://pt-br.reactjs.org/docs/hooks-intro.html>

Hooks resolvem uma variedade de problemas aparentemente separados em *React* que encontramos ao longo de cinco anos escrevendo e mantendo milhares de componentes. Esteja você aprendendo *React*, usando diariamente, ou até mesmo se prefere outra biblioteca com um modelo de componente parecido, você reconhecerá alguns destes problemas.

No contexto do *Framework*, se torna difícil reutilizar a lógica de estado entre os comportamentos pois *React* não oferece uma forma de vincular um comportamento reutilizável. Com os *Hooks* se torna possível extrair lógica com estado de um componente para que possa ser testada de maneira independente e que seja reutilizada, sem mudar a hierarquia entre os componentes, facilitando compartilhar *Hooks* entre componentes, e no âmbito de comunidade ao criar *Hooks* customizados.

Os *Hooks* também permitem utilizar mais das funcionalidades do *React* sem ter que depender de classes, pois de maneira conceitual os componentes *React* sempre estiveram mais próximos de funções. *Hooks* adotam funções mas ainda mantêm a praticidade de *React*, facilitando a entrada no processo de aprendizado do conceito, sem cobrar técnicas complexas de programação funcional ou reativa.

Um problema que foi detectado e que os *Hooks* conseguem resolver é a complexidade desnecessária de componentes. Eles tornam possível dividir um componente em funções menores, de forma que as partes sejam relacionadas, sem precisar forçar uma divisão baseada nos métodos de ciclo de vida, que adicionam complexidade desnecessária.

2.4 *React Native*

React Native é um *framework* de código aberto para construção de aplicações *mobile*(*Android* e *iOS*), utilizando *React* e as capacidades nativas da plataforma para que se está desenvolvendo. Este *framework* utiliza *JavaScript* para acessar as APIs dos aparelhos e para desenvolver a aparência e comportamento das interfaces gráficas utilizando componentes *React*.

No desenvolvimento *Android*, os componentes visuais são descritos por meio das linguagens *Java* ou *Kotlin*, e no desenvolvimento *iOS* são utilizadas as linguagens *Swift* ou *Objective-C*. Já com *React Native*, esses componentes são gerados por meio de *JavaScript*, utilizando componentes *React*, apresentando componentes visuais correspondentes em tempo de execução dependendo do ambiente que a aplicação está rodando. Por esse motivo, aplicações desenvolvidas em *React* Native parecem e performam como qualquer outro aplicativo.

2.5 *Redux*

Redux[6] é uma biblioteca de contenção e manutenção de estados para aplicações *JavaScript*, auxilia no desenvolvimento de aplicações que se comportam de maneira consistente, conseguem rodar em ambientes diferentes como cliente, servidor e nativo e tem facilidade para serem testadas. Além disso é compatível com qualquer outra biblioteca, podendo ser utilizada com *React* e possui uma ótima experiência de desenvolvimento. (REDUX, 2019)

Com o aumento da complexidade dos requisitos de aplicações de página simples, a manutenção de estados necessitam ser feitas pelo código mais do que nunca. *Redux* é uma ferramenta valiosa para organização de estado mas sua utilização deve ser mais recomendada nas seguintes situações

- Existe uma quantidade considerável de dados mudando ao longo do tempo;
- É necessária uma única fonte de verdade para seu estado;
- Manter seu estado em apenas um componente pai não é suficiente.

No processo de desenvolvimento com *React*, existem componentes que precisam acessar o mesmo estado mas apresentá-los de maneiras diferentes. Ter dois componentes que compartilham os mesmos dados pode gerar certa confusão ao tentar manter sincronia entre esses dados, que idealmente devem ter a mesma fonte. *Redux* vem para auxiliar esses cenários onde múltiplos componentes querem compartilhar dados mas não estão relacionados, criando um armazenamento que consegue comportar as informações de qualquer lugar na aplicação, esse armazenamento é chamado de *store*, que nada mais é que um objeto de *JavaScript* com alguns métodos, sendo possível conectar os componentes ao *store* e extrair apenas os dados necessários. (ERIKSON, 2017)

Os conceitos principais de *redux* estão presentes nos seguintes tópicos :

- *State*: Um dos mais importantes conceitos de *React* e *Redux* é que sua interface precisa ser criada de acordo com seu estado e, para isso, é necessário modelar o estado de sua aplicação primeiramente. No caso da aplicação de moita redonda, apenas a parte do carrinho da loja utiliza o *Redux* e seus componentes observam o estado correspondente e se comportam de acordo com ele;
- *Actions*: As *Actions* são objetos *Javascript* simples que recebem obrigatoriamente um atributo chamado *type*, que descreve algum evento que ocorreu na aplicação. O armazenamento do *Redux* não se importa com o tipo da *Action* que foi disparado, mas irá

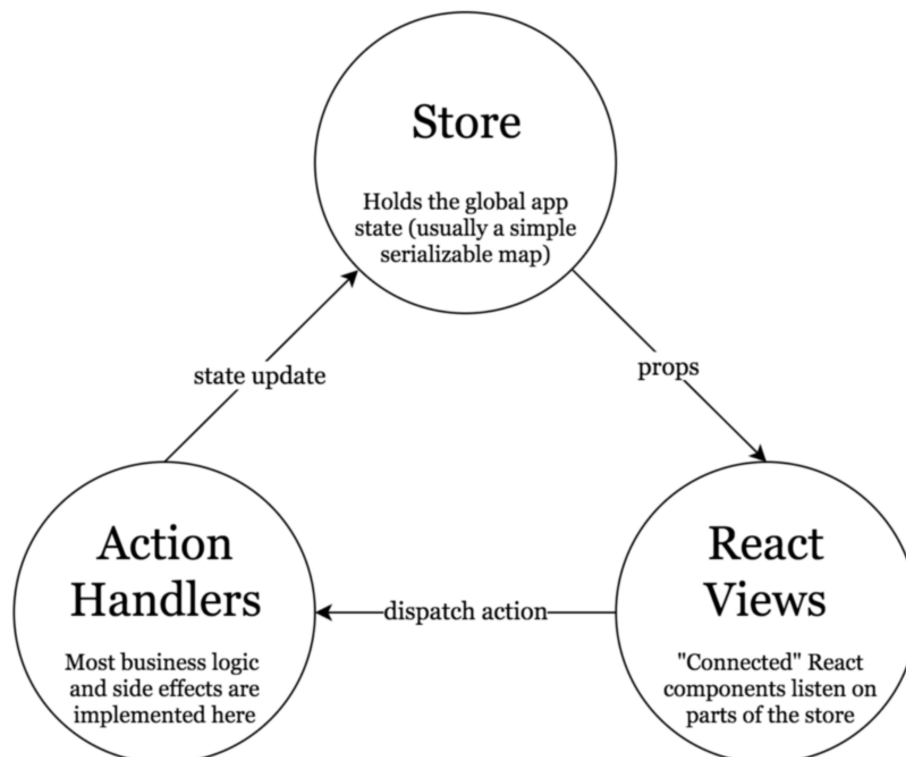
[6] Motivation <https://redux.js.org/understanding/thinking-in-redux/motivation>

olhar o tipo da *Action* para saber se alguma atualização é necessária. Existem casos que a *Action* também recebe um *payload*, que são dados que podem ser utilizados pelos *Reducers*;

- *Reducers*: Os *Reducers* são funções que recebem uma *Action* como parâmetro e o estado atual da aplicação e utilizam estes parâmetros para retornar em um estado novo, passando por estas respectivas mutações trazidas pelas *Actions*. Estas funções podem apenas calcular o novo valor do estado baseados nos parâmetros *Action* e *Reducer* e devem realizar mudanças de maneiras imutáveis, copiando o estado atual e aplicando estas mudanças.

A Figura 2 apresenta um diagrama da estrutura principal do Redux, mostrando seu fluxo de funcionamento.

Figura 2 – Diagrama estrutural do Redux



Fonte: <https://miro.medium.com/max/1000/1*aaczPQXKV-ijwdiDTdjYRA.png>

2.6 *Firebase*

O *Firebase* é uma BaaS(*Backend as a Service*) para aplicações web e *mobile* da *Google*, que foi adquirido em 2014 pela empresa e desde então vem crescendo bastante em

adesão e possibilidades fornecidas aos clientes. Quando se cria um novo projeto de aplicação é necessário levar em consideração diversos fatores como: infraestrutura, mecanismos de autenticação, segurança na autenticação de dados entre outros. Pensando em solucionar os problemas ao tentar alinhar esses fatores ao desenvolvimento das aplicações o *Firebase* foi criado, com as seguintes áreas: *Analytics* (análise estatística), *Develop* (desenvolver), *Grow* (crescer), *Earn* (render) . (ORLANDI, 2018)

Nas seguintes seções estão detalhados os serviços utilizados no desenvolvimento da aplicação de moita redonda, que são o *Firebase Auth, Firebase Firestore e Firebase Cloud Storage*.

2.6.1 *Firebase Auth*

A maioria dos *apps* precisa de um sistema de autenticação de usuários, para que seja possível reconhecer e armazenar a identidade deles. Possuir essa informação permite que o *app* salve os dados do usuário com segurança e que ele tenha a mesma experiência personalizada em todos seus dispositivos que tenham o *app* instalado. O *Firebase Authentication*[7] fornece serviços de *back-end*, SDKs fáceis de usar e bibliotecas de UI prontas para autenticar usuários no seu aplicativo. Ele oferece suporte à autenticação usando senhas, números de telefone, provedores de identidade federados conhecidos, como *Google*, *Facebook* e *Twitter*, entre outros. Para que os usuários possam se conectar aos *Apps*, se tornam necessárias suas credenciais de autenticação, que podem ser um e-mail e uma senha ou um *token* de *OAuth* de um dos provedores de identidade federados. Essas credenciais então são enviadas ao SDK do *Firebase Authentication*, que verificará essas credenciais e retornará uma resposta ao cliente. Por padrão, os usuários autenticados podem ler e gravar dados nos serviços: *Firebase Realtime Database, Firebase Firestore* e *Firebase Cloud Storage*.

2.6.2 *NoSQL*

O termo *NoSQL* é tipicamente utilizado para se referir a qualquer base de dados não-relacional, concordando que bases *NoSQL* são bases que armazenam dados em um formato diferente do tradicional, que são tabelas relacionais. (SCHAEFER, 2020)

Uma confusão comum que acontece é que os bancos de dados *NoSQL* ou bases não-relacionais não conseguem armazenar dados de relacionamento, mas o que realmente acontece

[7] Firebase Authentication <https://firebase.google.com/docs/auth>

é que isso é feito de maneiras diferentes. Quando comparadas a bases de dados *SQL*, muitos usuários acham modelar relacionamentos mais fácil em *NoSQL*, pois as datas relacionadas não precisam ser divididas entre tabelas.

Bancos de dados *NoSQL* emergiram no final dos anos 2000 como consequência do custo de armazenamento ter despencado. A necessidade de criar um modelo de dados complexo e difícil de manter apenas para reduzir duplicação de dados foi diminuída consideravelmente. O custo prioritário para desenvolvimento de software se tornou a contratação de desenvolvedores, então os bancos *NoSQL* otimizam a produtividade destes trabalhadores.

2.6.3 Firebase Firestore

O *Cloud Firestore*[8] é um banco de dados *NoSQL* hospedado em nuvem, flexível e escalonável para desenvolvimento *mobile*, web e servidores. Ele mantém seus dados em sincronia com os aplicativos clientes por meio de listeners em tempo real, além de oferecer suporte offline para que seja possível desenvolver aplicativos responsivos que funcionem sem depender de latência de rede ou conectividade com a internet. Seguindo o modelo de dados *NoSQL* do *Cloud Firestore*, você armazena dados em documentos que contêm mapeamentos de campos para valores. Esses documentos são armazenados em coleções, que são contêineres de documentos que você pode usar para organizar dados e criar consultas. Os documentos são compatíveis com muitos tipos de dados diferentes, desde strings e números simples a objetos complexos e aninhados. Também é possível criar subcoleções dentro dos documentos e criar estruturas de dados hierárquicas que podem ser escalonadas à medida que o banco de dados cresce. O modelo de dados do *Cloud Firestore* é compatível com qualquer estrutura de dados que funcione melhor para seu *app*.

Além disso, as consultas no *Cloud Firestore* são expressivas, eficientes e flexíveis. Crie consultas superficiais para recuperar dados no nível do documento sem precisar recuperar a coleção inteira ou qualquer subcoleção aninhada. Adicione classificação, filtragem e limites às consultas ou cursores para paginar os resultados. Para manter os dados atualizados nos aplicativos sem recuperar todo o banco de dados sempre que ocorrer uma atualização, adicione *listeners* em tempo real. Com eles, você é notificado com um instantâneo de dados em seu *app* sempre que houver mudanças nos dados que seus *apps* cliente estão detectando, recuperando somente as novas alterações. É possível proteger o acesso aos dados do *Cloud Firestore* por meio

[8] Cloud Firestore <https://firebase.google.com/docs/firestore?hl=pt-br>

de regras com *Firebase Authentication* e regras de segurança do *Cloud Firestore*, dependendo do ambiente de desenvolvimento.

2.6.4 Firebase Storage

O Serviço *Cloud Storage* pra *Firebase*[9] é um serviço poderoso, simples e econômico criado para a escala *Google*, sendo possível utilizar a segurança da empresa para fazer *upload* e *download* de arquivos nos aplicativos *Firebase*, independentemente da qualidade da rede utilizada.

Os desenvolvedores realizam o *upload* e o *download* diretamente das aplicações clientes, por meio dos SDKs do *Firebase Cloud Storage*. Quando a qualidade de conexão dos clientes é ruim, é possível tentar executar novamente a operação de onde aconteceu a interrupção, poupando tempo e largura de banda dos usuários.

Os arquivos são armazenados em um repositório do *Google Cloud Storage* e são acessados por meio do *Firebase* e do *Google Cloud*. Isso permite que você tenha a flexibilidade para fazer o *upload* e o download deles a partir de clientes móveis usando os SDKs do *Firebase*. Com o *Google Cloud Platform*, você executa processos no servidor como filtragem de imagens ou transcodificação de vídeo. O escalonamento automático é feito no *Cloud Storage*, o que significa que não é necessário migrar para outro provedor.

Para a identificação dos usuários, os SDKs do *Firebase* para *Cloud Storage* estão completamente integrados ao *Firebase Authentication*. Fornecemos uma linguagem de segurança declarativa que permite definir controles de acesso a arquivos individuais ou a grupos de arquivos. Assim, você pode tornar os arquivos públicos ou particulares conforme sua preferência.

[9] Cloud Storage para Firebase <https://firebase.google.com/docs/storage>

3 METODOLOGIA

Este capítulo do trabalho trata da metodologia utilizada para conseguir os conceitos, as motivações e as informações de base para se dar início ao processo de desenvolvimento da versão final do protótipo. Aqui estão elencados o padrão de desenvolvimento aplicado, a forma em que a equipe se dividiu, o processo de reunião de informações com o público alvo e como foi realizada a prototipação.

3.1 *Design Thinking*

Na disciplina de Projeto 2 do curso de SMD, o padrão aplicado ao desenvolvimento dos projetos das equipes foi o *Design Thinking*, que é um modelo mental iterativo que estabelece processos de design para que seja possível desenvolver uma solução de determinado problema.

Valorizando a experiência e colaboração entre os membros da equipe, considerando possíveis erros como aprendizado e testes, esse modelo incentiva a participação dos integrantes do grupo a colaborar e em consequência resultar em múltiplas visões acerca da resolução do problema em questão.

O *Design Thinking* é composto de múltiplas etapas e ao longo da duração da disciplina elas foram contempladas. Essas etapas não possuem rigidez em sua ordem, sendo possível repetir etapas já passadas anteriormente sempre que for necessário. A Figura 3 apresenta o fluxo completo dessas etapas.

Devido à limitação de escopo deste trabalho, foi dada prioridade à etapa de Implementação, enquanto outras etapas do *Design Thinking* estão presentes de maneira mais breve.

3.2 Divisão da equipe

Durante a disciplina, a equipe analisava o que precisava ser entregue em cada etapa, avaliando as tarefas possíveis de cada etapa e consultando os professores nas reuniões semanais, para que fosse possível tirar dúvidas e retificar o processo de desenvolvimento, para que o retrabalho fosse minimizado e a produtividade maximizada.

Os integrantes se dividiram em vários cargos, porém todos se intercalavam entre as atividades, tendo participação ativa em todas as partes do projeto. Essa decisão foi tomada para que não houvesse sobrecarga de trabalho e que a influência de múltiplos integrantes agregasse na

qualidade das entregas.

Portanto, após ter esses objetivos definidos, a equipe se dividiu da seguinte maneira:

- Bruno Cidade: UX Designer e UX *Analyst*
- Mateus Targino: UX *Designer* e *Social Media*.
- Matheus Brandão: Gestão do Projeto.
- Pedro Façanha: Desenvolvimento *frontend* e versionamento de código.
- Rafael Piazza: Desenvolvimento *backend*.

3.3 Reunião de dados e demandas da comunidade

Para conseguir os dados necessários a fim de realizar o desenvolvimento do projeto em questão , foi utilizado um método descritivo, que parte de um amplo grau de generalização e toma como principal necessidade utilizar o princípio da naturalidade implicando a ocorrência natural dos fatos, sem influências, utilizando também observações tanto qualitativas quanto quantitativas para conseguir o levantamento das informações sobre os objetos de estudo de maneira minuciosa e detalhada. Para realizar esse levantamento, foi importante conseguir as seguintes informações sobre os dispositivos(*smartphones*) presentes na comunidade: o modelo, sistema operacional, a data de compra, se o dispositivo é de 2ª mão e a frequência de uso diário.

Essas informações foram reunidas por meio de entrevistas, que é um método de coleta de dados qualitativo que permite ao pesquisador um contato direto com o objeto de estudo, é possível planejar a entrevista para conseguir respostas controladas ou respostas mais livres, depende da intenção do pesquisador e dos dados que deseja obter. Essas entrevistas foram realizadas com os proprietários dos *smartphones* e devido a necessidade de saber como é a experiência de uso entre os habitantes da região.

Durante as entrevistas também foram perguntados sobre as informações técnicas dos dispositivos presentes em Moita Redonda, tendo como principal intenção descobrir de maneira estatística e percentual as características em comum entre os dispositivos utilizados pelos moradores. Todas essas informações serviram para direcionar o desenvolvimento e facilitaram a triagem das dependências presentes na aplicação, dos seus padrões de design e decisões de UI.

3.4 Prototipação

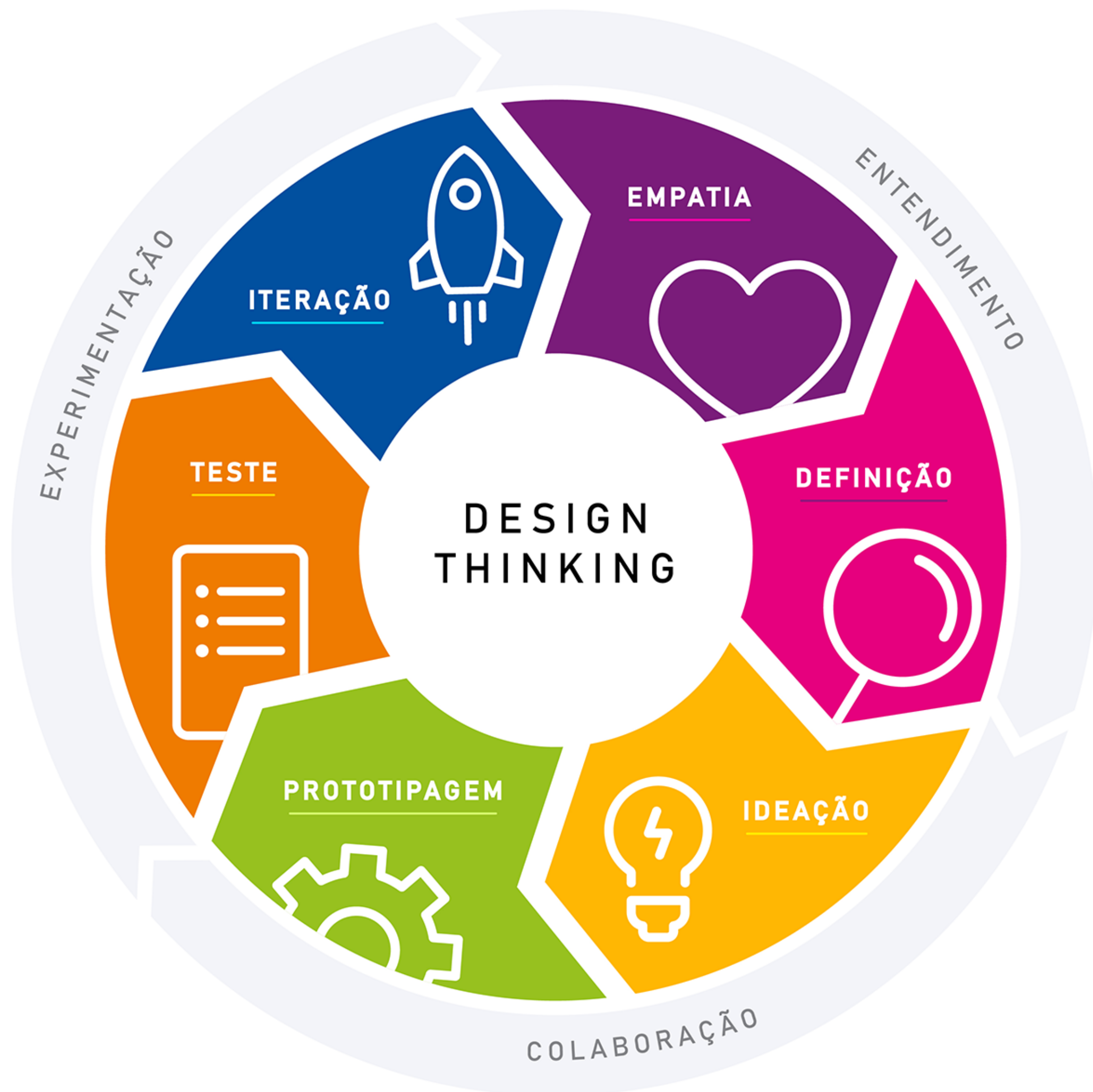
Após a reunião de dados e informações sobre o aplicativo que a equipe deveria desenvolver, foi dado o início aos processos de prototipação. Primeiramente, foram criados diagramas e modelos para que fosse possível obter um mapa mental das interfaces e características da aplicação.

Então, passando por revisões e comentários dos professores, os protótipos foram evoluindo, passando inicialmente por um protótipo de baixa fidelidade feito em papel, para apenas gerar as ideias para aqueles que viriam posteriormente, como o de média fidelidade, que já detalhava de maneira mais próxima ao que seria realmente desenvolvido, com elementos de UI mais complexos e estruturas coerentes.

Por fim, o protótipo de alta fidelidade foi apresentado e entregue, sendo bastante aprovado pelos professores e guiando o plano de desenvolvimento que seria tomado. Este protótipo contava com toda a navegação da aplicação, os estilos finais dos componentes das telas, as interações planejadas e parte do conteúdo final, abrindo uma oportunidade para o início deste trabalho que corresponde à versão funcional da loja. Esses protótipos foram foram desenvolvidos no software *Adobe XD*[1], que oferece diversas ferramentas para prototipação de tipos diferentes de interface.

[1] Adobe XD | Ferramenta de colaboração de design de UI/UX <https://www.adobe.com/br/products/xd.html>

Figura 3 – Fluxo do Design Thinking



Fonte: <http://labcom.com.br/blog/design-thinking-novos-desafios-novas-solucoes>

4 CONCEITUAÇÃO DO APLICATIVO

Este capítulo apresenta os conceitos principais que fundamentam a elaboração do aplicativo, ilustrando modelos e interfaces que motivam o processo de desenvolvimento, além de descrever profundamente o problema e a solução proposta do contexto geral.

4.1 Problema

Reiterando o que havia sido descrito na introdução deste trabalho, um dos problemas que a aplicação se propõe em resolver é o prejuízo econômico e subvalorização do trabalho dos artesãos pertencentes à comunidade de Moita Redonda causados pelos atravessadores de mercadoria que visitam a vila, revendendo os produtos com preços abusivos e não dando o crédito nos trabalhos aos artesãos que dedicam tempo e esforço diariamente na produção dessas peças de barro.

Outro problema identificado foi o isolamento geográfico da cadeia produtiva e da comercialização dos produtos, ou seja, mesmo com o apoio do grupo Varal, os artesãos ainda sofrem para fazer a divulgação e para que o público consiga acessar à comunidade, os produtores e os itens produzidos, limitando a receita e o lucro da comunidade.

Após analisar estes problemas que afetam a produção e comercialização de produtos de bairro em Moita Redonda, a equipe *Unicode* decidiu, analisando as habilidades dos membros, desenvolver um sistema de loja para a comunidade.

4.2 Proposta de solução

O aplicativo da loja de Moita Redonda serve como uma ferramenta tanto para os compradores interessados em artesanato de barro, quanto para os próprios artesãos, que podem administrar a loja de maneira autossuficiente, sem precisar do auxílio dos desenvolvedores da aplicação para realizar manutenção de produtos. Os compradores podem realizar encomendas, visto que não é costume manter estoque entre os artesãos, e fecham a compra entrando em contato diretamente com algum artesão membro da associação.

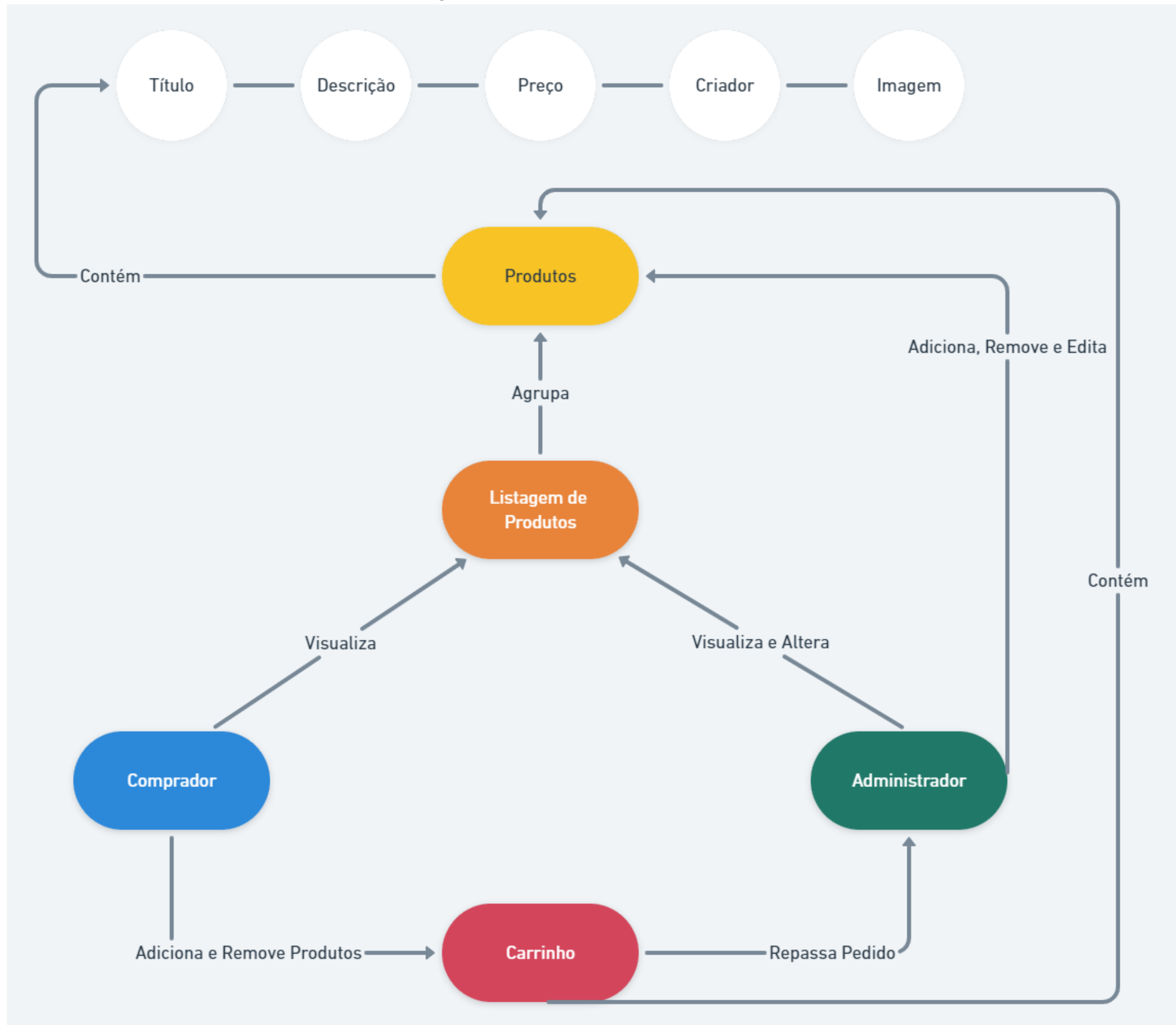
Este aplicativo deve fornecer múltiplas informações sobre os produtos, como uma descrição breve sobre a utilização da peça, seu processo produtivo, o preço, o artesão que produziu e uma imagem apresentando uma amostra do item. O foco da aplicação se dá pela automatização dos processos de vendas, divulgação dos processos e aproximação dos compradores com os

artesãos, no momento do fechamento da compra.

4.3 Descrição do sistema

Na Figura 4 apresenta-se o modelo conceitual do aplicativo de Moita Redonda.

Figura 4 – Modelo conceitual



Fonte: elaborado pelo autor

O modelo conceitual possibilita relacionar as principais funcionalidades e agentes do aplicativo:

Os compradores podem:

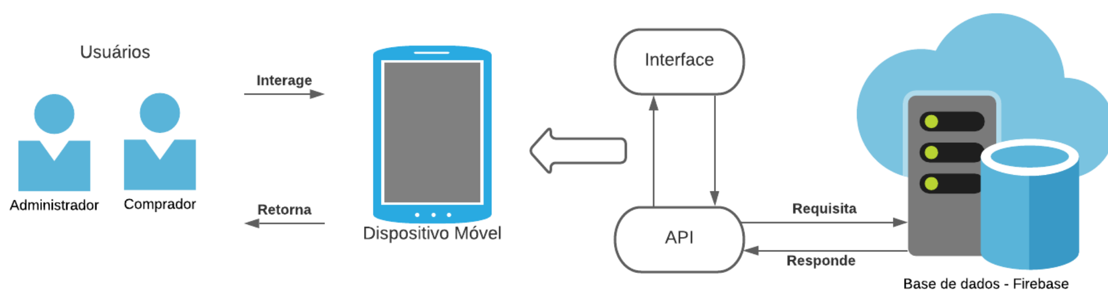
1. Visualizar produtos da loja
2. Adicionar e remover produtos do carrinho
3. Finalizar pedido

Já os Administradores podem:

1. Visualizar e alterar a listagem dos produtos
2. Adicionar remover e editar um produto singular

Além disso, foi elaborado um modelo de Arquitetura do Sistema, disposto na Figura 5, que permite enxergar a forma como as tecnologias usadas se relacionam e se comunicam. Devido ao tempo limitado da disciplina de projeto II, a equipe *Unicode* decidiu adotar uma arquitetura menos complexa, para que fosse possível otimizar o tempo de certas tarefas.

Figura 5 – Arquitetura do sistema



Fonte: elaborado pelo autor

Esta arquitetura é composta por 3 principais níveis:

- Usuários: Podendo ser ou comprador ou administrador, os usuários interagem com o segundo nível da arquitetura, recebendo respostas da aplicação rodando no dispositivo móvel.
- Dispositivo Móvel: Segundo nível da arquitetura, parte intermediária que liga a interface acessível aos usuários com o *backend* da aplicação, apresentando os resultados das interações vindas dos usuários e das requisições à API do servidor do *Firebase*.
- Base de dados - *Firebase*: A parte que armazena, altera, remove e recebe dados, que realiza todas as operações do lado do servidor. Se comunica com os dispositivos por meio de uma API, que aplica mudanças nos dados exibidos, alterando a interface.

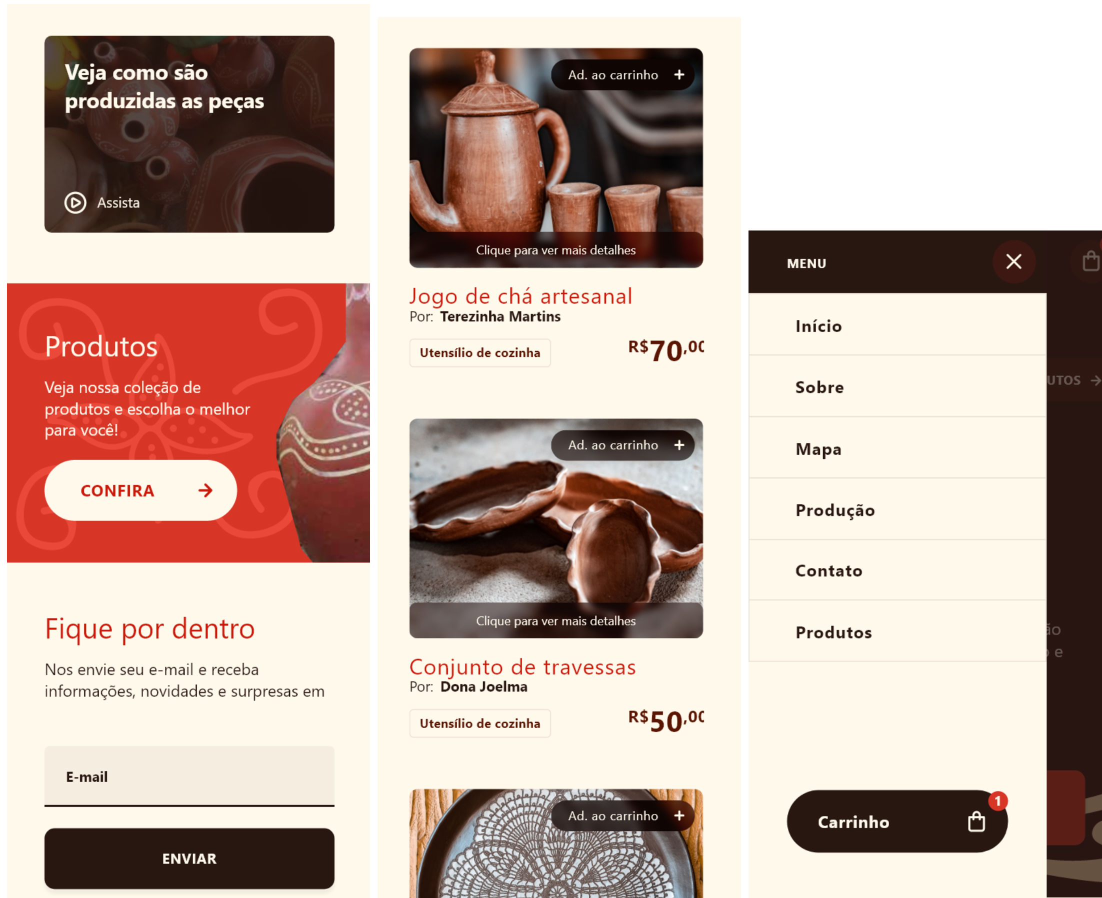
4.4 Interface

Com base nas informações estabelecidas nas seções anteriores, os designers da equipe *Unicode* produziram protótipos de alta fidelidade no software *Adobe XD*, criando as telas que compõem a interface do Aplicativo de Moita Redonda.

A Figura 6 possui imagens do protótipo apresentado na etapa final da disciplina de Projeto Integrado II. Estas telas representavam: a parte inicial do aplicativo, que conta com seções informativas e um *banner* de acesso aos produtos, os produtos, que possuem como principal funcionalidade adicionar um produto ao carrinho e um menu lateral que concedia acesso a outras seções da aplicação. No entanto, devido às limitações de tempo e conhecimento técnico da equipe, apenas foi implementada a seção da loja, que está presente neste protótipo, e a parte de administração de produtos, que não necessitava de prototipação, após uma avaliação da equipe.

Já a Figura 7 corresponde a identidade visual, apresentando detalhes, as fontes, cores e ícones utilizados tanto no protótipo, quanto nos componentes da aplicação. A criação dessa identidade auxiliou para guiar as motivações e facilitou no processo de desenvolvimento e alterações nos componentes.

Figura 6 – Protótipo produzido em Adobe XD



Fonte: acervo da equipe Unicode

Figura 7 – Identidade visual

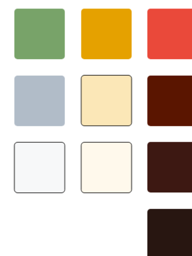
Heading 1: Young Serif - 38

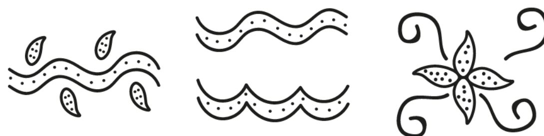
Heading 2: Casper Bold - 28

Heading 3: Casper Bold - 21

Heading 4: Casper Bold - 16

Paragraph: Casper Regular - 16. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Maecenas ac est lacus. Mauris varius dignissim libero, auctor rutrum nunc commodo et. Integer iaculis sit amet ipsum nec elementum. Morbi ullamcorper ut enim vitae ullamcorper. Donec





Versão Rezuzida:
Simbolo

Fonte: acervo da equipe Unicode

5 PRODUTO COMPLETO

Este capítulo tratará dos resultados obtidos após o processo de desenvolvimento ter encerrado seu ciclo. Estão dispostos aqui os fluxos possíveis de interação dentro da aplicação, apresentando as interfaces, no emulador de dispositivo disponível ao instalar o *Android Studio* e em um dispositivo pessoal.

Como foi antes mencionado neste trabalho, devido a arquitetura limitada pela duração da disciplina, apenas foi possível capturar os resultados em dispositivos *Android*(emulador e *smartphone*), sendo impossível extrair resultados da aplicação rodando no sistema operacional *iOS*, mas é possível perceber isso não prejudicou a demonstração das interações da aplicação.

5.1 Dados quantitativos

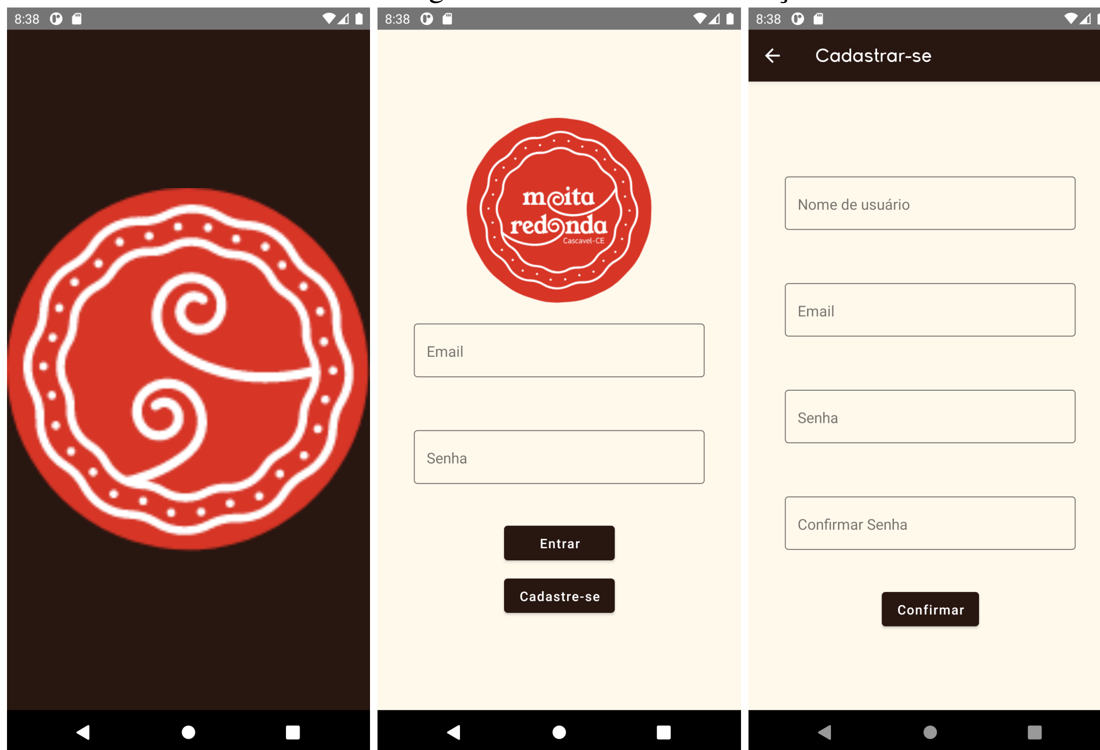
Após sua finalização, o aplicativo Moita Redonda é composto por: 8 componentes autorais, dando destaque para os componentes de lista de produtos da loja, *StoreProductsList*; 7 telas, todas concluídas seguindo os protótipos e utilizando os componentes criados. Considerando apenas os arquivos da própria aplicação, excluindo pacotes instalados e arquivos pertencentes ao *Git*, o projeto possui 157 arquivos que ocupam 1,04 MB (megabytes).

5.2 Fluxo de autenticação

Este fluxo, ilustrado na Figura 8 corresponde à seção anterior às páginas principais da aplicação, no caso para usuários, a loja, e para administradores, a administração de produtos. A primeira tela acessível a qualquer usuário é a de *Login*, que corresponde a um formulário que os usuários inserem e-mail e senha para se autenticarem, direcionando a loja se quem for autenticado é usuário padrão ou a administração de produtos caso o usuário for um administrador.

Também é possível cadastrar-se como usuário comum, ao tocar no botão “Cadastre-se”, que direciona a um formulário que pede inserção de nome de usuário, e-mail, senha e confirmação de senha. Todos formulários deste fluxo estão com medidas de segurança fornecidas pela biblioteca *Yup*, que previne erros de preenchimento pelos usuários.

Figura 8 – Fluxo de autenticação



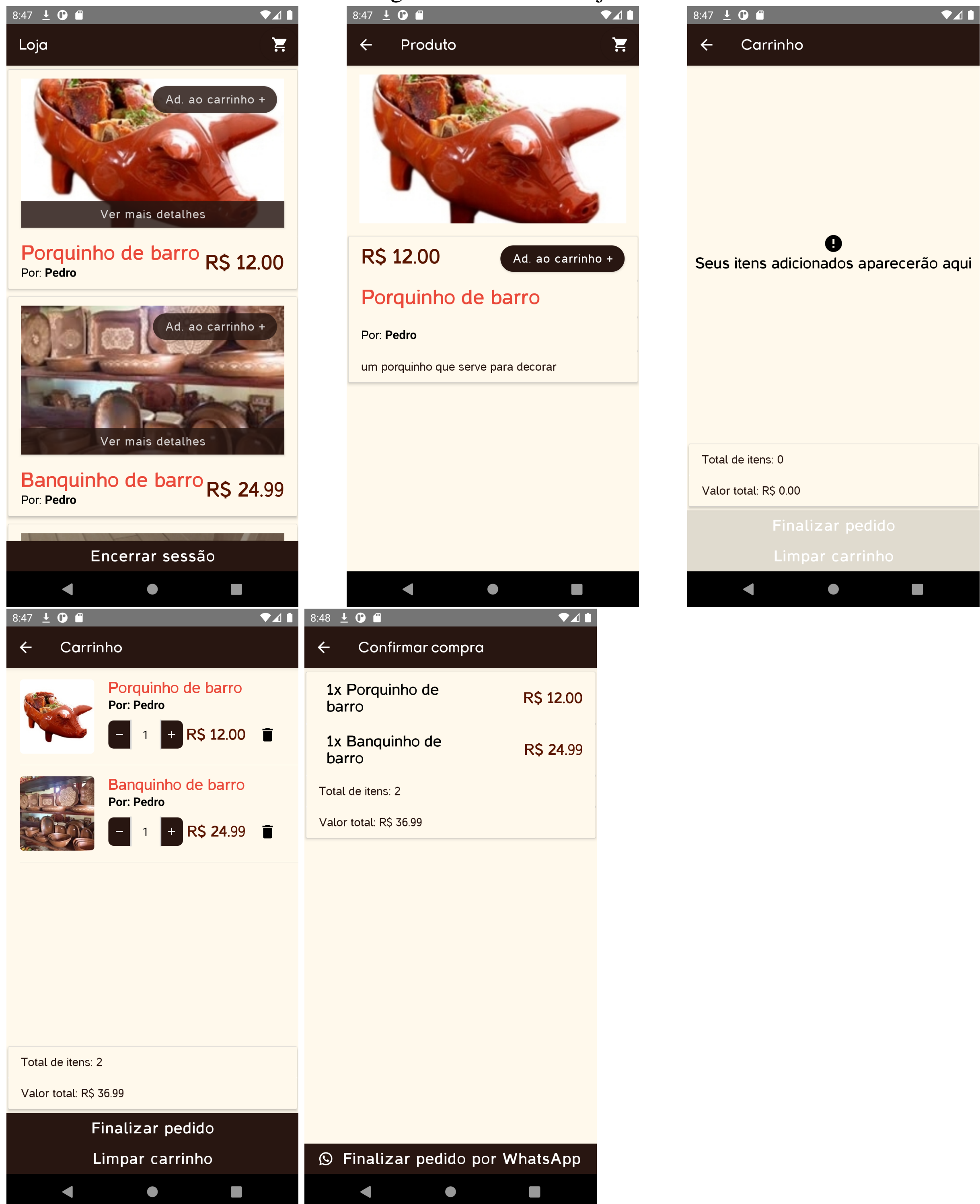
Fonte: elaborado pelo autor

5.3 Loja

A interação com a loja, apresentada na Figura 9, se inicia por um *login* bem sucedido vindo da página inicial, dando acesso aos usuários à listagem principal de produtos, sendo também possível acessar o carrinho. A listagem de produtos permite acessar os detalhes de um item específico, sendo possível também adicioná-lo ao carrinho tanto por essa lista quanto pela página do produto.

Na página de carrinho, o usuário terá acesso a outra relação de produtos, desta vez com as opções de alterar quantidade dos itens, remover um item em específico, limpar o carrinho e finalizar pedido, caso não existam produtos adicionados ao carrinho, a seguinte mensagem é exibida no centro da tela: "Seus itens adicionados aparecerão aqui". Ao escolher finalizar o pedido, é exibida uma página que apresenta o resumo dos itens escolhidos, o valor total e o número de itens, então se o pedido estiver correto, o usuário pode prosseguir apertando o botão "Finalizar pedido por *WhatsApp*", que abre o aplicativo *WhatsApp*, se estiver instalado. A ideia é permitir ao cliente entrar em contato com a associação de artesãos de Moita Redonda para combinar pagamento e envio, mandando uma mensagem montada com a relação de itens escolhidos no pedido e seu valor total. Este fluxo inteiro está representado na Figura 9.

Figura 9 – Fluxo de loja



Fonte: elaborado pelo autor

5.4 Administração de produtos

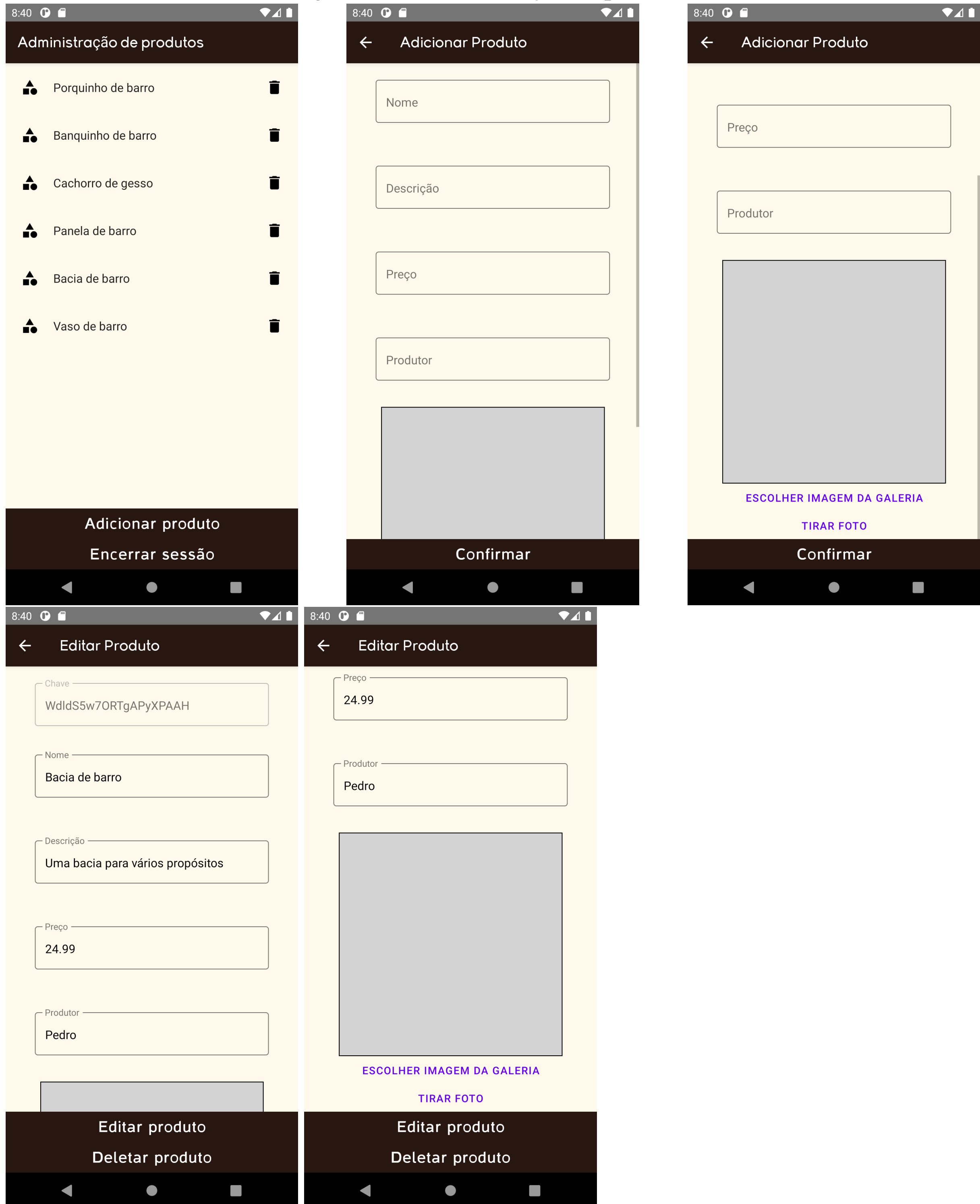
Esta interação apresentada na Figura 10 é iniciada por um *login* bem sucedido, resultante da verificação da variável de ambiente que armazena o e-mail de administrador e

senha, direcionando o usuário com cargo de administrador para um painel que apresenta a lista dos produtos disponíveis na loja, um botão para adicionar um novo produto,que leva o usuário para o formulário de adição, outro que permite encerrar sessão, que leva o usuário para a página inicial de *login*. A listagem permite ao usuário acessar o formulário de edição e excluir um item por meio de um botão com ícone de lixeira.

O formulário de adição de produtos também está assegurado pelas restrições do *Yup*, para que os administradores não cometam erros no preenchimento. Esta tela possui um mecanismo de envio de imagens, que o usuário pode escolher tanto uma imagem do armazenamento do dispositivo, quanto da própria câmera.

Já o formulário de edição de produtos segue o mesmo modelo do de adição, recebendo as medidas de segurança e os mesmos campos, no entanto alguns já chegam preenchidos, dependendo do item selecionado. É possível excluir o produto nesta tela, além de realizar uma substituição na imagem enviada.

Figura 10 – Administração de produtos



Fonte: elaborado pelo autor

6 PROCESSO DE DESENVOLVIMENTO

Este capítulo relata o processo de desenvolvimento do aplicativo de Moita Redonda feito com o *framework* React Native. São abordadas as várias etapas do processo, como as configurações iniciais, organização de pastas, padrões utilizados, versionamento de código, dificuldades no processo de desenvolvimento e a versão final do aplicativo.

6.1 Inicializando o projeto

Tomando como base a documentação oficial do *React Native*[1], há duas formas principais de se inicializar um projeto do *framework*: por meio de *React Native CLI* e pelo pacote *Expo CLI*. O *Expo CLI* não foi escolhido para este projeto por adicionar dependências desnecessárias, aumentando o tamanho do projeto. Mesmo adicionando diversas comodidades e melhorando a produtividade no processo de desenvolvimento, esse tamanho adicional não era interessante para o ambiente em que a aplicação iria rodar, por se tratar de celulares de menor desempenho.

Para se ter um ambiente que rode diretamente o *React Native CLI* , era necessário fazer uma série de preparativos. Inicialmente, foi preciso instalar a IDE *Android Studio*, que traz consigo o kit de desenvolvimento para aplicações *Android*, a plataforma para utilizar esse kit e o dispositivo virtual *Android*, que serve para rodar o aplicativo em ambiente de desenvolvimento. Também foi preciso instalar os pacotes de Node e o kit desenvolvimento de *Java*(*Java SE Development Kit* - JDK).

Após a instalação dessas dependências, foi necessário configurar variáveis de ambiente no *Windows*. Mais precisamente, foi adicionada a variável ANDROID-HOME, apontando para o diretório que está instalado o kit de desenvolvimento *Android*. Por fim, ao adicionar o Node, ele traz consigo o NPM[2], uma linha de comando para manutenção e instalação de pacotes mas por questões de preferência e comodidade, o desenvolvedor optou pela alternativa *yarn*[3], que garante mais performance e organização, foi instalada executando o comando presente no código fonte. 2

Código-fonte 2 – Comando para instalar o yarn

[1] Setting up the development environment <https://reactnative.dev/docs/environment-setup>
[2] npm <https://www.npmjs.com/>
[3] Yarn <https://yarnpkg.com/>

```
1 npm install --global yarn
```

Então, para instalar o *React Native CLI* foi executado o comando do código fonte 3.

Código-fonte 3 – Comando para instalar react-native-cli

```
1 yarn add global react-native-cli
```

Com isso, se torna acessível de maneira global o pacote *React Native CLI*, e então para inicializar o projeto do aplicativo Moita Redonda, foi executado o comando presente no código-fonte 4.

Código-fonte 4 – Comando para inicializar um projeto de React Native

```
1 yarn react-native init MoitaRedonda
```

O processo de geração de arquivos e download de dependências é iniciado e então a estrutura básica do projeto *React Native* é criada.

6.2 Arquivos e estrutura de pastas

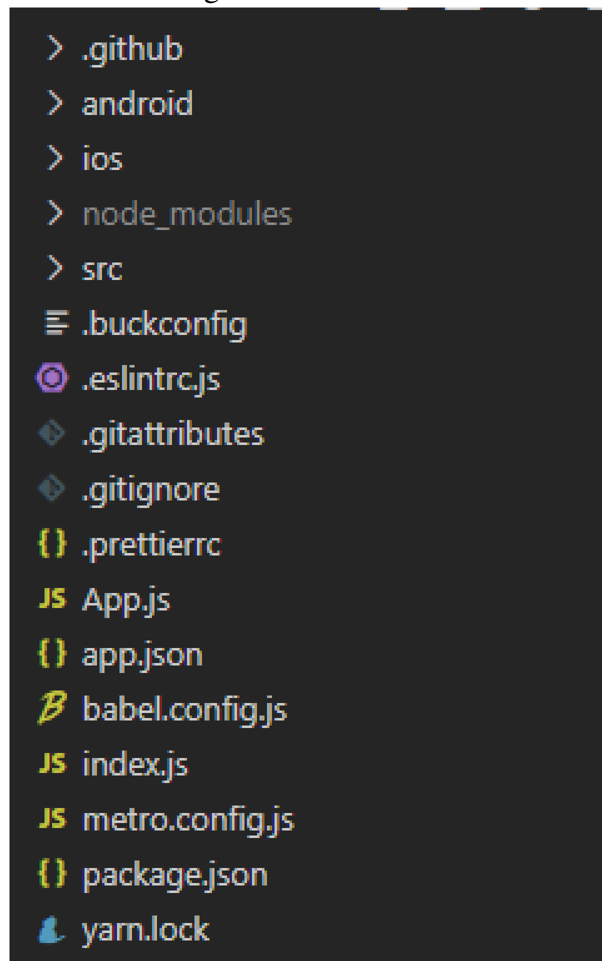
Ao rodar os comandos, um projeto com a estrutura de pastas e arquivos similar ao da Figura 11 é gerado. No diretório raiz estão arquivos de configuração de diversas bibliotecas como *Babel*[4], *Prettier*[5] e *ESLint*[6], além de arquivos para gerenciar pacotes e versionamento de código(*package.json, yarn.lock, .gitignore*).

O diretório *.github* serve para configurações do versionamento na plataforma *GitHub*, que é onde o repositório do projeto se encontra. No caso, essa pasta apenas armazena um modelo de *Pull Request*. Já o diretório *android*, contém a estrutura da versão *Android* do projeto, contendo todos os arquivos, classes e métodos para compilar para dispositivos desse sistema operacional. No diretório *ios*, está a estrutura para compilar e renderizar a aplicação nas versões *iOS*. Geralmente, no processo de desenvolvimento, o conteúdo dessas pastas não são muito alterados, pois a estrutura já está bem definida, a não ser com a inclusão de um pacote que demande alterações nesses arquivos.

---

[4] Babel - The compiler for next generation JavaScript <https://babeljs.io/>
[5] Prettier - Opinionated Code Formatter <https://prettier.io/>
[6] Pluggable JavaScript linter <https://eslint.org/>

Figura 11 – Pasta raiz



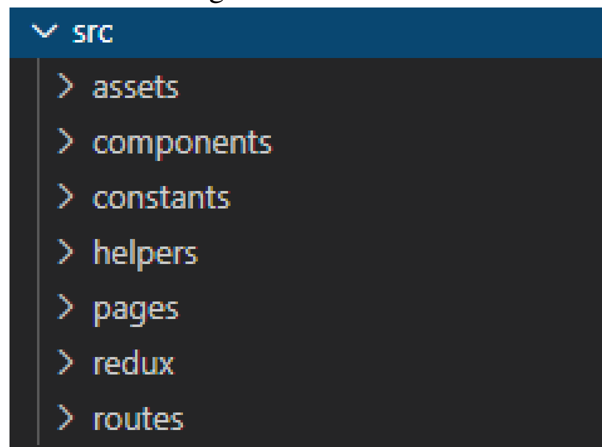
Fonte: elaborado pelo autor

6.2.1 *Pasta src*

A pasta src apresentada na Figura 12 é a pasta principal do projeto. Nela está contido todo o código principal de *JavaScript/React Native* . A pasta *assets* contém imagens que são comumente utilizadas no projeto, para que não seja necessário realizar importações de imagens na web com frequência. O diretório *components* armazena todos componentes *React* que serão reutilizados, ou aqueles que possuem potencial para serem reutilizados, mas aparecem apenas em uma página. Na *constants*, estão todas variáveis que não podem ser alteradas e serão utilizadas em mais de algum trecho de código, por exemplo, as variáveis locais de administrador para autenticação, que verificam se o usuário corresponde ao administrador.

O diretório *helpers* contém trechos de código de *JavaScript* puro, que possui certa frequência de repetição e que tem boa utilidade dentro da lógica total. A pasta *pages* contém todas as páginas/*views* do aplicativo, condensando os componentes e apresentando os contextos possíveis dentro do produto. *Redux* contém a administração de estados globais com o uso da

Figura 12 – Pasta *src*



Fonte: elaborado pelo autor

biblioteca homônima, contendo *reducers*, *actions* e *stores*, exportando para o resto da aplicação. Por último, o diretório *routes* contém a arquitetura de navegação da aplicação, configuração das páginas e regras de navegação com autenticação.

6.2.2 package.json

O arquivo *package.json* em projetos *JavaScript* possui os seguintes propósitos(NPMJS, 2018):

- Listar as dependências que o projeto utiliza e suas versões;
- Tornar a *Build* do seu projeto reproduzível, facilitando o compartilhamento com outros desenvolvedores;
- Listar os comandos possíveis(vide a chave *scripts*).

Neste arquivo, referenciado no código fonte 5, estão listados os pacotes utilizados para desenvolver e executar a aplicação por meio do gerenciador *yarn*, que ao rodar um simples comando *yarn install*, é iniciada a instalação de todos pacotes listados na pasta .node-modules. Já para executar a aplicação localmente, era necessário executar os comandos *yarn android* ou *yarn ios*, que disparavam um servidor Node e a instalação do aplicativo.

Código-fonte 5 – package.json

```
1  {
2   "main": "index.js",
3   "scripts": {
4     "android": "react-native run-android",
```

```
5        ...
6        "start": "react-native start",
7      },
8      "dependencies": {
9        "@react-native-firebase/app": "^10.4.0",
10       "@react-native-firebase/auth": "^10.3.1",
11       "@react-native-firebase/firestore": "^10.3.1",
12       "@react-native-firebase/storage": "^10.4.0",
13       ...
14       "formik": "^2.2.6",
15       "native-base": "^2.15.0",
16       "prop-types": "^15.7.2",
17       "react": "16.13.1",
18       "react-native": "~0.63.3",
19       ...
20       "react-redux": "^7.2.2",
21       "redux": "^4.0.5",
22       "styled-components": "^5.2.1",
23       "uuid": "^8.3.2",
24       "yup": "^0.32.8"
25     },
26     "devDependencies": {
27       "@babel/core": "~7.9.0",
28       "eslint": "^7.15.0",
29      ...
30       "prettier": "^2.2.1",
31     },
```

6.2.3 App.js

O código fonte 6, que corresponde ao arquivo App.js é o primeiro elemento a ser carregado na execução do aplicativo de Moita Redonda. Ele define o componente pai de todos

os outros existentes na aplicação. No início do trecho de código da imagem são feitas as importações de *Providers*, de pacotes de UI e do *Redux*, que permitem que os componentes acessem os métodos, estilos e dados incluídos nessas bibliotecas. Enfim, a função *App*, quando chamada retorna a aplicação envolvida pelos *Providers* e pelo componente *Navigation*, que leva à estrutura da navegação da aplicação, agregando as telas e a navegação chaveada por autenticação.

Código-fonte 6 – App.js

```
import React from 'react';
import { Provider as PaperProvider } from 'react-native-
    paper';
import { Provider } from 'react-redux';
import { Root } from 'native-base';
import Navigation from './src/routes/Navigation';

import store from './src/redux/store';

export default function App() {
  return (
    <Provider store={store}>
      <PaperProvider>
        <Root>
          <Navigation />
        </Root>
      </PaperProvider>
    </Provider>
  );
}
```

6.2.4 Telas

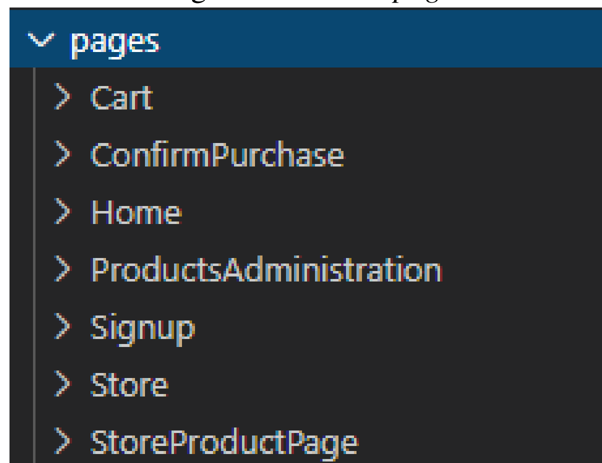
As telas do aplicativo de Moita Redonda estão agrupadas no diretório *pages*, como na Figura 13, e dentro desse diretório estão divididas por subdiretórios da seguinte maneira:

- *Cart*: Esta página é o carrinho da loja de produtos de barro. É possível visualizar os itens adicionados ao carrinho, o valor individual de cada um, o número de itens e o valor total da compra, aumentar ou diminuir a quantidade dos itens, limpar o carrinho e finalizar a compra. Todas essas operações e armazenamento são feitas alterando um estado global por meio do *Redux*.
- *Confirm Purchase*: Esta página apresenta um resumo da compra total, dessa vez exibindo apenas o número de itens, seus valores individuais e quantidades e o preço total da compra. Se o usuário deseja finalizar a compra, é possível utilizar um *deep linking* com o aplicativo de mensagens *WhatsApp Messenger* onde ele entra em contato com a associação dos artesãos de barro de Moita Redonda, combinando a entrega e o pagamento dos seus itens.
- *Home*: A página inicial da aplicação quando o usuário não está autenticado. Daqui é possível fazer login caso o usuário já possua conta ou então é possível ser direcionado para a página *Signup*.
- *Products Administration*: A página principal na *Stack* de navegação autenticada se o usuário for um administrador. A partir daqui, o administrador pode acessar os formulários de cadastro e edição de produtos, onde é possível inserir ou alterar as informações como nome, descrição, preço, autor e imagem dos produtos. Também é possível remover algum produto da loja.
- *Signup*: Esta página serve para cadastrar um novo usuário por meio de um formulário que possui os campos: *username*, e-mail, senha e confirmação de senha. Com sucesso na autenticação o usuário é direcionado à página da loja, conhecida como *Store*.
- *Store*: Página em que os usuários conseguem ter acesso aos itens cadastrados pelos administradores. É possível adicionar produtos ao carrinho, encerrar sessão, visualizar o carrinho e ter acesso às páginas de produto individuais(*Store Product Page*).
- *Store Product Page*: A página em que são mostrados os detalhes dos produtos. A partir dessa página o comprador pode adicionar produtos ao carrinho, visualizar todas as informações cadastradas pelos administradores e ter acesso ao próprio carrinho.

6.2.5 Navegação

A pasta *routes*, ilustrada na Figura 14 contém os arquivos que tratam da navegação e das rotas da aplicação, onde o principal arquivo é o *Navigation.js*, que implementa a lógica inteira, inclusive as rotas acessíveis apenas quando o usuário está autenticado. Esta lógica

Figura 13 – Pasta *pages*



Fonte: elaborado pelo autor

está implementada utilizando o *hook useContext*. Ela simplifica a *Context API*[7] do *React* para utilização mais rápida, mostrando dois contextos em que o usuário está autenticado ou não. Isto é feito observando o estado da API do *Firebase Auth* e mudando o contexto se o usuário está autenticado. O contexto de autenticação é criado como no código fonte 7.

Código-fonte 7 – AuthContext.js

```
import { createContext } from 'react';

export const AuthContext = createContext(null);
}
```

Utilizando esse contexto mutável, o componente *Navigation.js*, representado no código fonte 8 engloba duas navegações. Uma no contexto autenticado e outra não, tendo cada uma delas suas propriedades e uma lógica aplicada por um *hook* de *useEffect*. Este observa se o contexto de autenticação foi alterado e assim alterar qual navegação é exibida, usando um operador ternário para fazer essa troca.

Código-fonte 8 – Navigation.js

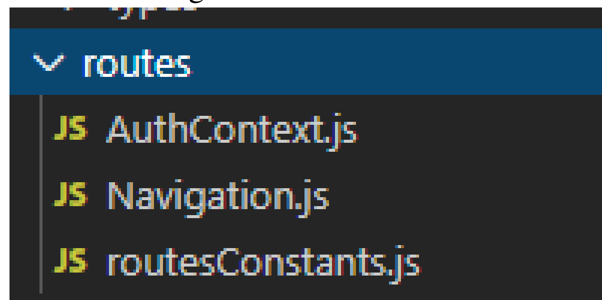
```
const Navigation = () => {
  const { Screen, Navigator } = createStackNavigator();
  const [user, setUser] = useState(null);
```

[7] Context <https://pt-br.reactjs.org/docs/context.html>

```
4    const [initializing, setInitializing] = useState(true);
5
6    const watchUserChanges = (result) => {
7      setUser(result);
8      if (initializing) {
9        setInitializing(false);
10     }
11   };
12
13   const { Provider } = AuthContext;
14
15 ...
16
17   return (
18     <NavigationContainer>
19       {user ? (
20         <Provider value={user}>
21           <SignedInStack />
22         </Provider>
23       ) : (
24         <SignedOutStack />
25       )}
26     </NavigationContainer>
27   );
28 };
29
30 ...
```

Por último, existe o arquivo *routesConstants.js*, descrito no código-fonte 9 ,que serve para formalizar as rotas de maneira constante. Esse arquivo é importado em diversos trechos de código do resto da aplicação e para facilitar o processo de desenvolvimento, essas variáveis são criadas para utilização repetitiva.

Figura 14 – Pasta *routes*



Fonte: elaborado pelo autor

Código-fonte 9 – routesConstants.js

```
...
const routesEnum = {
  home,
  signup,
  store,
  storeProduct,
  productsAdmin,
  addProductForm,
  editProductForm,
  cart,
  confirmPurchase,
};
...
```

6.2.6 *Hooks e Helpers*

A pasta *helpers*, apresentada na Figura 15, contém trechos de códigos que são repetidos com frequência e não possuem a necessidade de estarem descritos mais de uma vez no código dos componentes. Nesta seção, são incluídos também os seguintes *hooks* customizados:

- *useCart*: O *hook* que é importado para ter acesso a funções do *redux* que manipulam o estado global do carrinho da loja. Ele contém funções descentralizadas para adicionar, remover, limpar, carregar os itens e finalizar compra por meio de *WhatsApp*[8].

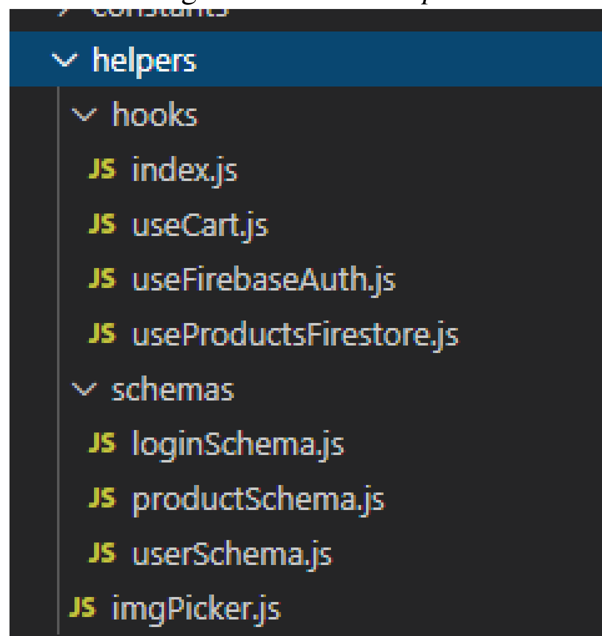
[8] WhatsApp <https://www.whatsapp.com/>

- *useFirebaseAuth*: Este *hook* importa as funções da biblioteca *Firebase Auth*, que serve para prover autenticação aos usuários. Ele contém funções para cadastrar, iniciar e encerrar sessão dos usuários e representar quando uma solicitação está carregando.
- *useProductsFirestore*: O último *hook* importa as funções da biblioteca *Firebase Firestore*, que serve para armazenamento e manutenção de dados na forma de um banco *NoSQL*, mais especificamente para o registro de produtos. É possível cadastrar, editar, deletar e fazer *upload* de uma imagem além de também representar quando uma solicitação está carregando.

Além dos *hooks*, também estão armazenados os *schemas*, que são objetos da biblioteca *Yup*[9], que consegue aplicar regras aos campos de formulários. Esses objetos são utilizados em combinação com a biblioteca *Formik*[10], que adiciona utilitários aos formulários dentro de *React*, facilitando o desenvolvimento e garantindo mais segurança nesses itens.

Por último, o arquivo *imgPicker.js* possui trechos de código para agilizar as chamadas da biblioteca *react-native-image-crop-picker*[11], que serve para acessar a câmera ou a galeria dos dispositivos, para a inserção de imagens dos produtos. Por ter trechos de código repetitivos e longos, eles foram isolados dos componentes onde eram chamados.

Figura 15 – Pasta *helpers*



Fonte: elaborado pelo autor

[9] Yup <https://github.com/jquense/yup>
[10] Overview <https://formik.org/docs/overview>
[11] react-native-image-crop-picker <https://github.com/ivpusic/react-native-image-crop-picker>

6.2.7 Componentes

A Figura 16 representa a pasta *components*, que contém os componentes que são invocados com bastante frequência na aplicação, ou que possuem código e lógica grandes o suficiente para terem a necessidade de serem desacoplados de outros componentes e páginas. Estes componentes possuem lógica isolada e todos possuem verificação de propriedades por meio da biblioteca *prop-types*[12], para que não existam erros nas invocações durante o processo de desenvolvimento. A estilização desses componentes, assim como das páginas, é feita por meio da biblioteca *styled-components*[13], que utiliza uma sintaxe própria conhecida como *css-in-js* que atribui as classes e lógicas do CSS popular da web para componentes nativos. Os principais componentes e suas utilidades são:

- *CartListItem*: Componente simples de item de lista de carrinho. Possui à esquerda uma imagem do produto, ao centro o nome e seu preço. É possível interagir neste componente ao alterar sua quantidade por meio de outro componente *CartQuantityInput* e deletando o produto da lista, por meio de um botão com um ícone de lixeira.
- *CartQuantityInput*: Componente simples que mostra uma quantidade e possui botões para aumentar ou diminuir, em uma unidade, os itens do contexto que está inserido.
- *CartStatusCard*: Este componente observa o estado global do carrinho de compras, listando de maneira dinâmica o número de itens no carrinho e o valor total da compra. É utilizado nas páginas de fechar pedido e carrinho.
- *ErrorMessageComponent*: Componente de mensagens de erro padrão, por propriedades recebe uma mensagem que é exibida junta de um ícone de alerta quando determinado carregamento falha ou alguma lista se encontra vazia.
- *LargeButton*: Botão customizado oriundo da biblioteca *react-native-paper*. Por ser utilizado em várias páginas no rodapé do aplicativo, se torna melhor utilizá-lo de maneira mais isolada, passando como propriedade seu título, efeito e cor. Seus estilos estão pré definidos para garantir uma padronização na aplicação.
- *ProductsListItem*: Item que representa um produto na lista na seção de administração. É possível visualizar um sumário com imagem, preço e nome do produto, além de acessar o formulário de edição e excluir diretamente o produto.
- *StoreListItem*: Item que representa um produto na lista na seção da loja. É possível

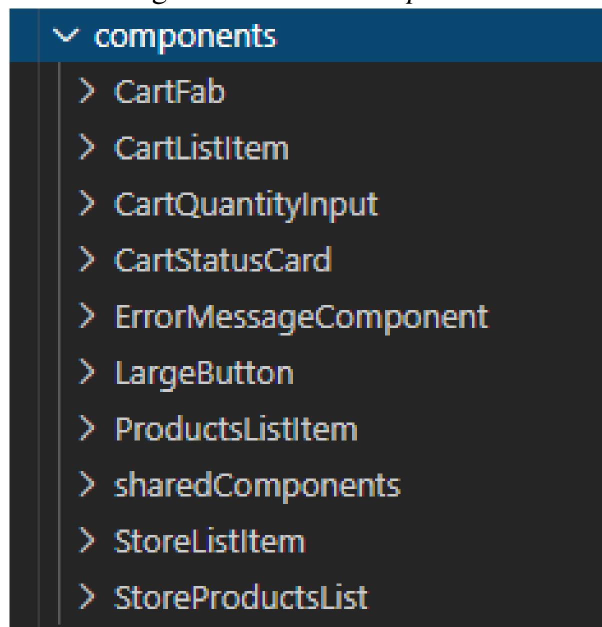
[12] prop-types <https://github.com/facebook/prop-types>
[13] styled-components <https://styled-components.com/>

visualizar um sumário com imagem, preço e nome do produto, além de acessar a página do produto e adicionar diretamente o produto ao carrinho.

- *StoreProductsList*: Este componente consiste de uma lista que itera os itens *StoreListItem*, passando as propriedades os itens provenientes do *Firestore*, cuja lógica para realizar a busca de dados está implementada neste componente. Se existir algum erro na listagem, o componente *ErrorMessageComponent* é invocado.

A pasta *sharedComponents* possui componentes simples que servem apenas para estilização e organização dos elementos nas telas, como *grids, wrappers e containers* personalizados por meio da biblioteca *styled-components*.

Figura 16 – Pasta *components*



Fonte: elaborado pelo autor

7 MANIPULAÇÃO E FONTE DE DADOS

Este capítulo trata das principais técnicas utilizadas para a manutenção dos dados dentro da aplicação, descrevendo como ela é realizada de maneira local entre os componentes, em um contexto mais interno e também descrevendo as interações com o banco de dados selecionado para integrar a arquitetura, em um contexto mais externo.

7.1 *Redux*

Como biblioteca principal para manutenção de estados globais em contexto local da aplicação, esta seção trata de como está estruturada a configuração para que seja possível utilizar o *Redux* com sucesso em toda extensão do aplicativo.

7.1.1 Implementação no aplicativo

Como citado, o *Redux* foi implementado apenas na parte de carrinho da loja, todas as partes do código referentes a essa biblioteca servem para observar o estado do carrinho e estão resumidas no *hook useCart*, que foi criado utilizando os outros *hooks* customizados *useDispatch* e *useSelector* que servem respectivamente para enviar alguma ação(nome obrigatório e *payload* opcional) para o *store* da aplicação e recuperar o objeto do carrinho.

7.1.1.1 Actions

- *addToCart(item)*: Serve para adicionar um item ao carrinho, atualizando o *array* de itens do *store*.
- *changeItemQuantity(item,quantity)*: Serve para procurar um item no *store* e alterar a sua quantidade de acordo com a passada no parâmetro.
- *clearCart()*: Serve para remover todos os itens do carrinho ou seja, limpar o *array* dos itens do *store*. Função sem parâmetros, passando apenas seu tipo.
- *removeFromCart(item)*: Serve para procurar um item do carrinho e gerar um novo *array* sem o item do parâmetro passado.

7.1.1.2 Reducers

Existe apenas um *reducer* na aplicação, que é o do carrinho, que está representado no código fonte 10. Ele é inicializado com um *array* vazio, representando o *state* e a depender da *Action* recebida, um retorno diferente é feito e o *store* é atualizado de acordo. É possível observar no código que existem mecanismos de busca e atualização de *arrays* trazidos por meio da linguagem *JavaScript*, facilitando o processo de desenvolvimento e garantindo segurança nas operações.

Código-fonte 10 – cartReducer.js

```
const cartReducer = (state = initialState, action) => {
  switch (action.type) {
    case ADD_TO_CART:
      return {
        ...state,
        ...
      };
    case CHANGE_ITEM_QUANTITY:
      return {
        ...state,
        cart: state.cart.map((item) =>
          item.key === action.payload.item.key
            ? { ...item, quantity: action.payload.quantity }
            : item
        ),
      };
    ...
    default:
      return state;
  }
}
```

7.1.1.3 Store

Para formar o estado central da aplicação, é necessário combinar todos os *reducers*, então formando um objeto acessível a toda aplicação, isto é realizado com o método *combineReducers()*, que recebe um conjunto de *reducers*. Então é formado o objeto *rootReducer*, descrito no Código-fonte 11, que no arquivo *index* da pasta *store*, descrito no Código-fonte 12, é passado como parâmetro para a função *createStore*, que por sua vez é exportada e passada para o *provider* no componente *App.js*, que recebe um *provider* que envolve toda a aplicação, implementando todas as funcionalidades do *Redux*.

Código-fonte 11 – rootReducer

```
import { combineReducers } from 'redux';
import cartReducer from '../reducers/cart';

const rootReducer = combineReducers({
  cartReducer,
});

export default rootReducer;

}
```

Código-fonte 12 – store

```
import { createStore } from 'redux';
import rootReducer from './rootReducer';

const store = createStore(rootReducer);

export default store;
```

7.2 *Firebase*

O *Firebase* foi utilizado para que seja possível persistir os dados dos produtos em nuvem, por meio do serviço *Firestore*, que é um banco de dados *NoSQL* baseado em documentos *JSON*.

7.2.1 Uso do Firestore no aplicativo de Moita Redonda

O *Firestore* usa apenas uma coleção, apresentada no painel da Figura 17, a dos produtos que é modelada da seguinte maneira:

- *name*: O nome do produto adicionado, campo obrigatório;
- *description*: A descrição do produto adicionado, campo obrigatório;
- *price*: O preço em reais do produto, obrigatoriamente ponto flutuante, campo obrigatório;
- *owner*: O criador do produto adicionado, campo obrigatório;
- *downloadUrl*: A url gerada ao enviar a imagem do produto, apenas se o administrador selecionar uma imagem para ser enviada ao *Firebase Storage*;
- *imgFileName*: O nome do arquivo gerado ao enviar a imagem do produto, apenas se o administrador selecionar uma imagem para ser enviada ao *Firebase Storage*;
- *createdAt*: Horário de criação, gerado por um método próprio da *lib* que conecta o *React Native* com o *Firestore*; e
- *updatedAt*: Horário de atualização, gerado por um método chamado tanto no momento de criação e no de atualização próprio da *lib* que conecta o *React Native* com o *Firestore*.

Como foi mencionado anteriormente, os métodos CRUD estão centralizados no *hook useProductsFirestore*. Este *hook* implementa métodos oriundos da biblioteca *React Native Firebase*, que apresenta em sua documentação exemplos recomendações de usos, por exemplo, o processo para recuperar todos itens da loja se dá por um *useEffect* que atualiza sempre que uma alteração é feita na coleção '*Products*', rodando novamente o método de recuperação e atualizando no *frontend* em tempo real.

Figura 17 – Painel do Firestore

Fonte: elaborado pelo autor

8 VERSIONAMENTO DE CÓDIGO

Para realizar um versionamento do código, visualizar histórico, ter acesso a versões anteriores e acessar de maneira remota o ambiente de desenvolvimento do projeto, foi decidido hospedar um repositório no site *Github*, que utiliza o sistema de controle de versão de arquivos *Git*, comumente utilizado no universo da programação.

O fluxo de trabalho dentro do *GitHub* consistia das seguintes etapas:

- Escolha da *Feature*: O trabalho se iniciava quando era escolhida uma parte da aplicação para produzir, com isso era criado um *branch* com o seguinte nome: *Feature* - Nome da *feature*;
- Resolução da *Feature*: Então, após criar o *branch*, o desenvolvedor realizava as mudanças necessárias e realizava os *commits* e então subia para o repositório remoto;
- Abertura de *Pull Request*: O desenvolvedor acessava o repositório no *GitHub* e escolhia a *branch* criada para iniciar uma nova *Pull Request*, adicionando uma descrição sobre o que foi alterado e iniciando a revisão das alterações;
- Finalização: Quando a revisão era finalizada, se não houvessem mudanças necessárias no código, o desenvolvedor poderia selecionar fundir os *branches*, por meio da opção *merge*.

Este fluxo de trabalho foi selecionado pois garante mais agilidade, sem demandar muito tempo para revisão e garantindo qualidade e segurança ao código. Além disso, o *GitHub* fornecia várias comodidades, como visualizar *Pull Requests* antigas e a criação de *Readmes* para apresentação e documentação.

9 CONSIDERAÇÕES FINAIS

Este trabalho demonstrou o processo de conceituação e desenvolvimento do aplicativo, buscando focar em quais tecnologias auxiliaram no desenvolvimento em *React Native*, como era realizado o gerenciamento dos dados dentro da arquitetura selecionada e de que maneira a equipe interagia para realizar um bom versionamento de código. Foi mostrado como iniciar a programação de um aplicativo em *React Native* por meio de *react-native-cli*, reunindo as principais características e boas práticas desse desenvolvimento.

De modo geral, foi concluído que o processo de reescrita da aplicação foi facilitado pela experiência de desenvolvimento confortável que o *framework* oferece, além das ferramentas disponíveis que facilitam o processo de tradução de interfaces para código. O processo de construção do *backend* também foi facilitado graças aos serviços disponibilizados pelo *Firebase*, que pouparam bastante tempo ao evitar que os desenvolvedores criem um sistema de banco de dados, autenticação e envio e armazenamento de arquivos próprio.

Sobre o padrão de design escolhido, durante o desenvolvimento foi possível perceber que as bibliotecas utilizadas como *Redux* e a utilização de diversos *hooks* personalizados permitiram a escolha do padrão *High Order Components*[1], que são conjuntos de funções que recebem um componente existente e retorna um outro componente, envolvendo-o e atribuindo-lhe novas funcionalidade, assim reutilizando a lógica do componente.

Por fim, apenas foi possível conseguir gerar e capturar o funcionamento da aplicação em dispositivos *Android*, pois a equipe não possuía acesso a dispositivos *iOS*. Acredita-se que o resultado seria similar, devido às bibliotecas utilizadas. O código da aplicação está disponibilizado no *GitHub* pessoal do autor[2], com instruções de como instalar corretamente e rodar uma versão de desenvolvimento. Também foi produzida uma demonstração em vídeo[3], que está disponível para acesso no canal do *YouTube* do autor. O vídeo apresenta as principais interações disponíveis na versão desenvolvida neste trabalho.

Como trabalhos futuros, pretende-se:

- Realizar a implementação das seções informativas presentes nos protótipos, que ficaram ausentes nessa versão por limitações de tempo e força de trabalho;
- Desenvolver testes unitários e de integração entre os componentes da aplicação para garantir segurança em futuras manutenções e versões do aplicativo; e

---

[1] Componentes de Ordem Superior <https://pt-br.reactjs.org/docs/higher-order-components.html>
[2] Repositório da aplicação <https://github.com/PedroFAC/MoitaRedonda-RN>
[3] Demo - Moita Redonda <https://youtu.be/YThhVFYoj8Q>

- Pesquisar meios de inserir um pagamento direto à associação de artesãos de Moita Redonda, também incluindo uma seção de administração de pedidos.

REFERÊNCIAS

COPES, F. **React Concept: Immutability**. 2018. Disponível em: <https://flaviocopes.com/react-immutability/>. Acesso em: 2021-03-24.

ECMA, E. **262: Ecmascript language specification**. 2015. Disponível em: <https://262.ecma-international.org/6.0/>. Acesso em: 2021-04-21.

EL-KASSAS, W. S.; ABDULLAH, B. A.; YOUSEF, A. H.; WAHBA, A. M. Taxonomy of cross-platform mobile applications development approaches. **Ain Shams Engineering Journal**, v. 8, n. 2, p. 163–190, 2017. ISSN 2090-4479. Disponível em: <https://www.sciencedirect.com/science/article/pii/S2090447915001276>.

ERIKSON, M. **Fullstack React: Redux and Why it's Good For You**. 2017. Disponível em: <https://www.newline.co/fullstack-react/articles/redux-with-mark-erikson/>. Acesso em: 2021-03-24.

MACHADO, R. C. V. **Artesanato do Barro**. 2003. Disponível em: <http://basilio.fundaj.gov.br/pesquisaescolar/index.php?option=com_content&view=article&id=350&Itemid=180>. Acesso em: 2021-04-21.

NPMJS. **Creating a package.json file | npm Docs**. 2018. Disponível em: <https://docs.npmjs.com/creating-a-package-json-file/>. Acesso em: 2021-03-24.

ORLANDI, C. **Firebase: serviços, vantagens, quando utilizar e integrações**. 2018. Disponível em: <https://blog.rocketseat.com.br/firebase/>. Acesso em: 2021-03-24.

REDUX. **Getting Started with Redux | Redux**. 2019. Disponível em: <https://redux.js.org/introduction/getting-started>. Acesso em: 2021-03-24.

SCHAEFER, L. **What is NoSQL? NoSQL Databases Explained**. 2020. Disponível em: <https://www.mongodb.com/nosql-explained>. Acesso em: 2021-03-24.

XANTHOPOULOS, S.; XINOGALOS, S. A comparative analysis of cross-platform development approaches for mobile applications. In: . [S.l.: s.n.], 2013.