



React Native Desenvolvimento De Aplicativos Mobile Com React - Casa doCodigo

Tópicos Avançados em Programação (Universidade Estadual de Campinas)



Digitalizar para abrir em Studocu

React Native

Desenvolvimento de aplicativos mobile com React



Casa do
Código

This document is available free of charge on

BRUNA ESCUDELARIO



studocu

DIEGO PINHO

DEDICATÓRIA

Dedicamos este livro às nossas famílias: Maria Aparecida de Freitas Escudelario e Irineu Escudelario; Ednilda Cicilini de Pinho, Ilidio Graciano Martins de Pinho e Lucas Martins de Pinho. Graças ao apoio de vocês, conseguimos concluir mais este trabalho. Do fundo dos nossos corações, muito obrigado.

Amamos vocês.

AGRADECIMENTOS

Primeiramente, gostaríamos de agradecer muito às nossas famílias que sempre nos apoiaram, motivaram e educaram a buscarmos sempre sermos pessoas melhores, seja no trabalho, com os amigos e principalmente com o próximo. Também somos muito gratos aos nossos amigos e mentores. A ajuda de todos foi extremamente valiosa para a conclusão deste trabalho. Nosso muito obrigado a cada um de vocês.

Também gostaríamos de agradecer à Editora Casa do Código por nos dar a oportunidade e o espaço para compartilhar este conhecimento de React Native. Em especial, um agradecimento a Vivian Matsui, sempre muito presente e parceira em todo o processo da construção deste livro.

E por fim, e não menos importante, gostaríamos de agradecer a você que está lendo! Muito obrigado pela confiança em nosso trabalho! Nós nos esforçamos muito para trazer um material de qualidade, atual e de acordo com o que o mercado de trabalho está esperando de um profissional que trabalha com esta tecnologia. Ficamos na esperança de que este material ajude você a ser um profissional da tecnologia ainda melhor.

Boa leitura e bons códigos!

AUTORES

Bruna de Freitas Escudelario



Figura 1: Bruna de Freitas Escudelario

Desenvolvedora front-end desde 2016. Fez bacharelado em Ciência da Computação pela Pontifícia Universidade Católica de São Paulo (PUC-SP) e é coautora do livro *Construct 2: Crie seu primeiro jogo multiplataforma*, também publicado pela Editora Casa do Código, e do livro *O Básico da Modelagem 3D com o Blender*, publicado pela Editora Viena.

Sempre gostou muito de ler e estuda diariamente por meio de cursos, artigos e vídeos na internet. Começou a se aventurar no desenvolvimento de jogos há pouco tempo, mas já acumulou experiência suficiente para tocar seu primeiro negócio na internet – junto com o Diego –, a *Time to Play*, uma iniciativa especializada em recursos para a construção de jogos digitais. Hoje atua como desenvolvedora front-end em uma grande empresa de aprendizagem corporativa de São Paulo.

- Site: <https://brunaescudelario.github.io/>

Diego Martins de Pinho



Figura 2: Diego Martins de Pinho

Desenvolvedor de software desde 2013, tem experiência na área da educação e domínio em tecnologias web de front-end e back-end. Dentre as principais linguagens estão o Java e o JavaScript. Fez bacharelado em Ciência da Computação pela Pontifícia Universidade Católica de São Paulo (PUC-SP) e possui MBA em Gerenciamento da Tecnologia da Informação pela Faculdade de Informática e Administração Paulista (FIAP).

Tem uma grande paixão pela educação e gosta muito de ensinar. Escreve artigos sobre tecnologia na internet, faz apresentações em eventos e é entusiasta em programação para jogos, modelagem (2D e 3D) e animação. É coautor do livro *Construct 2: Crie seu primeiro jogo multiplataforma* e autor do livro *ECMAScript 6: Entre de cabeça no futuro do JavaScript*, ambos publicados pela Editora Casa do Código. Hoje atua na área da educação como Professor Especialista de Programação e Jogos na Escola Móvil Integral em São Paulo. Também é responsável pela iniciativa Code Prestige de ensino de programação a distância.

- Site: <https://diegopinho.com.br>

PREFÁCIO

Quando o Diego e a Bruna me convidaram para escrever estas linhas iniciais do novo livro de React Native que estão publicando, fiquei extremamente feliz. Já tive o prazer de trabalhar com ambos em empresas diferentes e o grau de comprometimento e conhecimento dos dois é algo incrível.

Com o Diego, tive a oportunidade de trabalhar no iMasters e foi uma experiência única, principalmente quando viajamos para a Campus Party de Belo Horizonte, onde palestramos para uma turma bem jovem e empolgada. Mas essa é uma história para outro momento... Já com a Bruna, trabalhamos juntos em projetos de aprendizagem corporativa em uma empresa no centro de São Paulo. É impressionante como a dedicação dos dois é parecida.

Já o meu contato com o React e o React Native aconteceu primeiro com o conceito e depois com a ferramenta, algo que não é comum. Desde que trabalho com tecnologia - e isso já faz vários anos - gosto de conhecer primeiro o motivo de usar, para depois usar e foi assim com esta tecnologia.

A ideia por trás do React, tanto a biblioteca, quanto o conceito, é sensacional. O desenvolvimento de aplicações single page consegue deixar tudo mais simples para o usuário, que no fim das contas é quem realmente importa quando estamos desenvolvendo um novo produto ou site.

Desde que o React ganhou vida, primeiro em 2011, dentro do Facebook, depois em 2013 quando foi liberado para a comunidade, o modo como os desenvolvedores passaram a ver suas aplicações

mudou drasticamente, e o mesmo se aplicou ao desenvolvimento mobile com o React Native. É por isso que acredito que o conteúdo deste livro é essencial para programadores de todos os perfis e níveis.

Acredito ser tão importante que, trabalhando nos últimos meses com jovens de baixa renda, tento apresentar, até mesmo para eles que nunca tiveram contato com uma única linha de código na vida, os conceitos básicos desse modelo de programação trabalhado com tanto esforço neste livro. Claro que não é fácil para eles, mas é muito importante que vejam desde cedo o que pode ser feito para melhorar a vida deles como programadores e a dos seus futuros usuários.

Apreendi desde cedo que o conhecimento nunca é demais e as novas tecnologias, linguagens e conceitos surgem para nos ajudar a melhorar o mundo que nos cerca e isso nunca foi tão verdade quanto agora, principalmente para os programadores e desenvolvedores. Tenho absoluta certeza de que o Diego e a Bruna vão mostrar nas próximas páginas como o React e o React Native podem ser utilizados nos mais diversos tipos de projetos, principalmente nos mais complexos.

Este livro não é apenas uma leitura, mas uma coleção de conhecimentos que você vai levar para a sua vida pessoal e profissional. Aproveite cada palavra, cada linha, cada frase, cada página e cada código da forma mais intensa possível.

O mundo é do React e agora chegou a hora de suas aplicações também serem :)

Reinaldo Silotto - Canal TekZoom

INTRODUÇÃO

Um breve resumo sobre o React Native

O React Native é uma plataforma baseada no React que nos possibilita desenvolver aplicativos mobile híbridos, ou seja, que rodam tanto no iOS (Apple) quanto no Android (Google). A tecnologia é toda baseada em JavaScript, assim como a sua base, o React.

O React é a biblioteca JavaScript mais utilizada hoje no mercado de desenvolvimento web. Competindo com os frameworks Angular e Vue.js, o React se consagrou no mercado e se mantém no topo por ser:

1. **Simples de aprender:** o React possui poucos conceitos fundamentais e é fácil sair produzindo aplicações web com ele;
2. **Desenvolvido (e mantido) pelo Facebook:** o time de engenheiros do Facebook já lançou uma série de tecnologias de código aberto além do próprio React. Além de trabalharem constantemente no projeto, eles o utilizam em suas próprias aplicações, o que torna um próprio caso de sucesso da biblioteca;
3. **Adota a especificação de Web Components:** um dos grandes movimentos da web é pelos componentes. O React abraça esse movimento e torna possível o trabalho através de componentes (o que traz uma série de vantagens em relação ao desenvolvimento tradicional);

4. **Adotado pela comunidade e grandes empresas:** a comunidade e o mercado abraçaram a tecnologia, tornando-a popular e mais confiável.

Aproveitando todas essas vantagens do React, nasceu o React Native, cuja proposta é levar todos os aspectos positivos que as pessoas amam no React para o contexto do desenvolvimento mobile. Acreditamos que este livro despertará o interesse de todos que:

1. Usam o JavaScript como linguagem principal de desenvolvimento;
2. Trabalham/conhecem/estudam o React;
3. Se interessam por desenvolvimento de aplicativos mobile.

A quem se destina este livro

A proposta é que o livro seja acessível e prático para pessoas de qualquer nível de entendimento sobre o assunto, de modo que abordaremos a utilização da tecnologia de ponta a ponta (instalação, primeira aplicação, desenvolvimento, otimização etc.). Muitos dos conhecimentos que são pré-requisitos serão retomados, como é o caso dos próprios conceitos fundamentais da biblioteca React (para web).

O que vou aprender neste livro?

Nosso objetivo é que ao final deste livro, você seja capaz de:

1. Entender a terminologia e os fundamentos do React e React Native;
2. Criar aplicações mobile tanto para Android quando iOS que

- realmente resolvam algum problema do mundo real;
3. No processo de desenvolvimento, ser capaz de planejar e desenvolver o código seguindo padrões e recomendações adotadas pelo mercado.

Eu preciso saber JavaScript?

O React Native é uma tecnologia para desenvolvimento mobile totalmente baseada no JavaScript. Isso significa que sim, será necessário um conhecimento prévio da linguagem para conseguir aproveitar com mais profundidade esta produção. Mas antes que você se desespere e desista de continuar a leitura, indicaremos exatamente quais aspectos da linguagem são interessantes que você tenha pelo menos um conhecimento básico para conseguir acompanhar o conteúdo.

1. Sintaxe e estrutura de linguagem;
2. Controle de fluxos e condicionais;
3. Variáveis, funções e objetos.

Além destes pontos estruturantes, é bem importante estar atualizado nas constantes melhorias da linguagem, principalmente nas vindas no ECMAScript 2015 (ES6). Das várias melhorias, podemos destacar:

1. Funções auxiliares de array (ex.: map, filter, reduce);
2. Arrow functions (funções de seta);
3. Desestruturamento de Objetos e Arrays;
4. Classes e Módulos;
5. Const e Let.

O React é muito bem integrado a todas estas funcionalidades e

portanto tornam o nosso desenvolvimento muito mais rápido e o código bem mais limpo e manutenível. Por coincidência (ou não), temos um trabalho prévio publicado pela Casa do Código que pode ajudar bastante no entendimento destas e outras funcionalidades do JavaScript. O livro se chama *ECMAScript 6: Entre de cabeça no futuro do JavaScript* (<https://www.casadocodigo.com.br/products/livro-ecmascript6>) publicado em 2017.

Em resumo, não é necessário ser um especialista na linguagem, mas ter conhecimento prévio dos fundamentos da linguagem é muito bem-vindo.

Eu preciso saber React?

O React é a base de toda tecnologia por trás do React Native. Por mais desejável que seja que você tenha um conhecimento prévio da biblioteca, não é obrigatório.

Como as tecnologias funcionam de forma muito semelhante, por muitas vezes teremos que dar um passo atrás e explicar conceitos importantes do funcionamento do próprio React para entender como montar os nossos aplicativos usando o React Native. Então fique tranquilo, o essencial nós aprenderemos juntos.

Como devo estudar com este livro?

Este livro foi estruturado de modo que os tópicos apresentados se complementem e se tornem gradativamente mais complexos ao decorrer da leitura. Em todos eles, serão apresentados diversos conceitos, sempre apoiados por códigos contextualizados em caso

de usos reais, seguindo as boas práticas adotadas pelo mercado e pela comunidade desenvolvedora. Você notará não somente a evolução dos conceitos, mas também a dos códigos apresentados à medida que a leitura for seguindo.

Como acreditamos que somente com a prática que aprendemos a programar e fixar o conhecimento, no repositório oficial deste livro no GitHub você encontrará uma série de exercícios elaborados pensando na prática dos tópicos apresentados. Junto aos exercícios, você encontrará o gabarito comentado. Ao final do livro, deixamos uma série de recomendações de livros, artigos e cursos para que você aprofunde os estudos.

Consulte este livro sempre que surgirem dúvidas e entre em contato sempre que sentir necessidade. Leia e releia até compreender os conceitos. Cada um tem um ritmo diferente de aprendizado, não acelere o seu e persista até conseguir fazer sozinho.

Em caso de dúvidas, estaremos sempre à disposição. Não deixe de comentar e participar das discussões sobre o livro e os exercícios no site oficial e nos nossos canais de comunicação!

Tanto o site quanto o repositório do livro continuarão sendo atualizados com novos exercícios, artigos, notícias e projetos de exemplo. Vamos aprender juntos!

E o mais importante: nunca deixe de praticar!

- Site oficial: <https://livroreactnative.com.br>
- Repositório com os exercícios feitos ao longo da leitura: <https://github.com/BrunaEscudelar/livro-react-native>

- Repositório de exercícios extras:
<https://github.com/DiegoPinho/react-native>

Sumário

1 História do desenvolvimento do React Native	1
1.1 O que é o React Native?	2
1.2 História	3
1.3 Vantagens do React Native	6
1.4 O que vem por aí	8
2 Instalação e configurações iniciais	11
2.1 Explorando o Expo	12
2.2 Instalando e usando o Expo	14
3 Funcionamento do React Native	27
3.1 Por debaixo dos panos - React	27
3.2 Entendendo o arquivo App.js	34
3.3 Componente funcional	39
4 Criando os primeiros componentes	42
4.1 Criando um componente e importando no App.js	42
4.2 Acessando as propriedades do componente	47
4.3 Propriedades em componentes de classe	50

5 Componentes estilizados (CSS-in-JS)	52
5.1 Aplicando estilos	53
5.2 Utilizando arquivo externo	54
5.3 Estilos internos ao componente	60
5.4 Classes CSS	61
5.5 Separando estilos genéricos - padrão	62
6 O básico de layouts com o Flexbox	66
6.1 Altura e largura (height e width)	66
6.2 Contêineres e elementos flex	69
6.3 Flex Direction	71
6.4 Justify Content	73
6.5 Align Items	75
6.6 Flex-wrap	77
6.7 Flex-grow	80
6.8 Flex-shrink	82
6.9 Flex-basis	84
7 Renderização Condicional	86
7.1 Verificando se o número é par ou ímpar	87
7.2 Renderização condicional com função	91
8 State, eventos e componentes controlados e não controlados	94
8.1 Conhecendo os estados	94
8.2 Usando as informações dos estados	98
8.3 Atualizando o estado (componentes controlados)	100
8.4 Componentes controlados X não controlados	106
9 Requisições AJAX e APIs	109

9.1 Ciclo de vida dos componentes	109
9.2 AJAX	117
10 Navegação	128
10.1 React Navigation	128
10.2 Navegação por Menu Lateral	130
10.3 Navegação por Links	133
10.4 Navegação por abas	137
11 Integração com o banco de dados do Firebase	141
11.1 Configuração	142
11.2 Aplicativo	146
11.3 Integração	147
12 Trabalhando com Hooks	165
12.1 O que são os Hooks?	166
12.2 Hook de Estado (State Hook)	168
12.3 Hook de Reducer (Reducer Hook)	175
13 O futuro do React Native	181
14 Referências	184

HISTÓRIA DO DESENVOLVIMENTO DO REACT NATIVE

Como qualquer tecnologia que usamos hoje em dia — seja web, desktop ou mobile — o React Native também é fruto de um trabalho árduo, constante e muito bem feito por muitas pessoas capacitadas e dedicadas espalhadas pelos quatro cantos do mundo. No entanto, não se engane: não é porque já existe uma versão operacional da tecnologia que o trabalho está acabado, muito pelo contrário. Neste exato momento, muitos desenvolvedores continuam trabalhando para torná-lo mais rápido, eficaz e fácil de usar.

Para que possamos entender como o React Native funciona e o porquê de ele ter sido construído da maneira como é hoje, neste capítulo olharemos para um passado não muito distante e exploraremos um pouco a história da plataforma. Vamos saber mais sobre qual foi a sua origem, quem são os responsáveis pelo seu desenvolvimento e afins. Depois discutiremos brevemente quais suas principais vantagens, funcionalidades e quais as expectativas para o futuro. Com toda essa bagagem histórica, estaremos preparados para entrar de cabeça no desenvolvimento

com o React Native.

1.1 O QUE É O REACT NATIVE?

Se você está lendo este livro, temos certeza de que você já possui ao menos uma ideia do que se trata o React Native. Mas antes de começarmos a falar sobre sua história, vamos entender o que ele é. O React Native pode ser definido como um framework que consiste em uma série de ferramentas que viabilizam a criação de aplicações mobile nativas — para as plataformas iOS e Android — utilizando o que há de mais moderno no desenvolvimento front-end (HTML, CSS e JS). É considerada por muitos como a melhor opção do mercado no que se refere ao desenvolvimento mobile híbrido baseado em JavaScript, estando à frente dos seus concorrentes (ex.: Ionic). A *stack* do React Native é poderosíssima, pois nos permite utilizar ECMAScript 2015+ (ES6+), CSS Grid e Flexbox, JSX, diversos pacotes do npm e muito mais.

O React Native tem duas grandes sacadas: a primeira é que ele se baseia inteiramente no seu idolatrado parente, o React (para web). A segunda é que, ao contrário de outras plataformas semelhantes, o React Native converte todo o código desenvolvido para a linguagem nativa do sistema operacional do seu aparelho. Ou seja, enquanto os outros serviços empacotam as aplicações dentro de uma *webview*, o React Native vai na contramão. Em termos práticos, isso significa que não temos uma camada web como interface do nosso aplicativo, mas sim o próprio aplicativo nativo rodando. Em termos de performance, estar rodando no código nativo da plataforma é bem melhor.

Mas o que tudo isso significa? *Webview*? Código nativo? Não se

preocupe, exploraremos cada um destes pontos no decorrer de todo o livro. Por ora, vamos entender um pouco mais onde tudo isso começou.

1.2 HISTÓRIA

O React Native é um projeto desenvolvido pelos engenheiros do Facebook e teve um início bastante fascinante. Tudo começou no verão de 2013 em um Hackathon interno da empresa. Para quem não está familiarizado com o termo, um Hackathon nada mais é do que uma maratona de programação na qual desenvolvedores de software (e aqui vale mencionar que não é exclusivo para programadores, é muito comum que designers, arquitetos e até mesmo gerente de projetos participem) se reúnem por horas, dias ou até semanas, com os mais diversos objetivos. Em geral a ideia é desenvolver uma solução para um dado problema dentro de um determinado tempo. No entanto, dependendo da meta do respectivo evento, eles podem ir além, como:

- Explorar vulnerabilidades nos produtos;
- Criar novos produtos de software e/ou hardware;
- Desvendar códigos e sistemas legados;
- Discutir novas ideias e propor soluções.

Estes eventos não são exclusivamente fechados às empresas, também podem (e devem!) ser públicos, já que dão visibilidade às empresas organizadoras e soluções rápidas e mais baratas para elas. No caso específico do Facebook, a prática de Hackathons é bem comum, tendo em mente sempre a motivação dos engenheiros, incentivo à criatividade e a inovação de produtos, serviços e tecnologias dentro da empresa. No Brasil já existem alguns eventos

abertos deste tipo promovidos pela própria rede social, mas se você tiver curiosidade de saber um pouco mais sobre o assunto, recomendamos dar uma olhada no filme "A Rede Social" estrelado pelo ator americano Jesse Eisenberg. Além de contar sobre a origem da empresa, o filme fala bastante sobre a sua cultura interna que prevalece até os dias de hoje.

Mas dando continuidade à nossa história, o framework surgiu em um destes eventos e deu no que falar. A partir daí, começou a ser desenvolvido até sua primeira aparição pública em janeiro de 2015, no evento React.js Con. Caso tenha curiosidade e facilidade com a língua inglesa, encorajamos a ver a apresentação completa que está disponível no Youtube (<https://www.youtube.com/watch?v=KVZ-P-ZI6W4/>). Meses depois, no evento F8 — conferência anual realizada pelo Facebook e que ocorre na cidade de São Francisco, Califórnia — anunciaram que o React Native seria um projeto de código aberto e disponível por meio do GitHub (<https://github.com/facebook/react-native/>) — plataforma de hospedagem de código-fonte com controle de versão usando o Git que permite que programadores contribuam em projetos privados e/ou de código aberto de qualquer lugar do mundo. Deste então, a coisa não parou mais.

Em pouco mais de um ano, os números do React Native não paravam de impressionar. Durante este período, 1002 contribuidores commitaram 7971 vezes em 45 branches no meio de 124 releases. Além disso, na época, o repositório se tornou o 14º com mais estrelas no GitHub. Já deu para perceber que o React Native não estava para brincadeira.



Figura 1.1: O número de contribuições para a branch master, excluindo commits de merge

É importante ressaltar que o crescimento precoce do framework também foi emplacado (em partes) pela promessa de duas gigantes da tecnologia - Microsoft e Samsung - de levarem a plataforma para os seus produtos. Com o apoio declarado destas empresas e o suporte da comunidade, o React Native conseguiu superar o desenvolvimento nativo iOS e Android, de acordo com o Google Trends — ferramenta do Google que mostra os mais populares termos buscados em um passado recente.

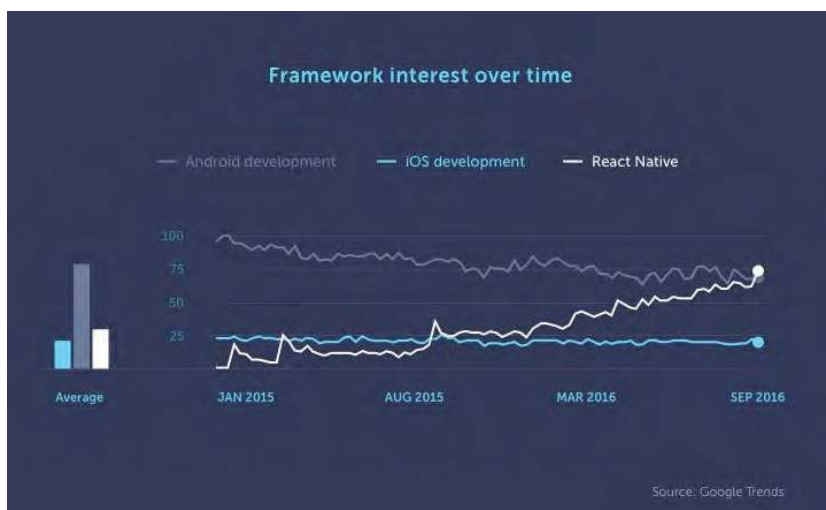


Figura 1.2: Interesse de acordo com o Google Trends

E isso foi só o começo do sucesso. O crescimento continuou no mesmo ritmo e hoje o framework se consolidou como a melhor opção para se desenvolver aplicações mobile híbridas. Só para se ter uma ideia de sua grandeza, grandes empresas como Uber, Tesla, Walmart, Adidas, Baidu, Skype, Salesforce, Pinterest e tantas outras — sem contar as da própria empresa, como os aplicativos do Facebook e Instagram — já possuem aplicativos oficiais criados com a plataforma. Podemos ver uma lista detalhada no site oficial (<https://facebook.github.io/react-native/showcase.html>).

1.3 VANTAGENS DO REACT NATIVE

No decorrer do livro estudaremos várias das facilidades que o React Native nos apresenta para o desenvolvimento de aplicativos, no entanto, é interessante que já tenhamos em mente algumas das

diferenças entre uma aplicação nativa, uma aplicação mobile web e qual o papel do React Native nisso. Comentamos anteriormente que um dos grandes chamativos do React Native é a sua capacidade de transformar o código JavaScript em código nativo das plataformas, ao contrário do que boa parte dos seus concorrentes faz. A vantagem disso é que ganhamos:

1. Carregamento da aplicação mais rápidos;
2. Experiência do usuário fluida;
3. Mais segurança;
4. Melhor integração entre funções nativas do celular como câmera, GPS etc.;
5. Melhor performance em geral.

Pode parecer besteira, mas você sentirá essa diferença na pele. Antes do surgimento do React Native, desenvolver aplicações nativas simultâneas para Android e iOS era bem mais complexo e muito mais caro, afinal, além de o desenvolvedor ter que aprender duas plataformas e linguagens diferentes (Java/Kotlin para o Android e Objective-C/Swift para o iOS), não era possível aproveitar partes do código de uma plataforma para a outra. As empresas tinham que contratar desenvolvedores para cada sistema operacional, tornando o projeto muito mais caro e demorado. Agora, com o React Native, o código pode ser reaproveitado em até 100%, fazendo com que o custo e a duração do desenvolvimento caiam drasticamente (boas notícias para as *startups*).

Funcionalidades do React Native

Aqui não queremos dar muitos *spoilers* e estragar a graça do

livro, mas acreditamos que vale adiantar algumas coisas bacanas da plataforma. Entre as diversas funcionalidades bem interessantes do React Native está o *Hot Reloading*. Esta técnica nos permite fazer com que o programa fique rodando constantemente em segundo plano (em desenvolvimento) e, a cada vez que o código é alterado, ele é interpretado, seu *build* (código final compilado) é feito e as suas alterações são mostradas rapidamente na tela. No desenvolvimento mobile isso é extremamente importante, pois em outros frameworks, a cada vez que o código é modificado a aplicação precisa ser completamente recompilada, o que nos faz perder muito tempo.

Fora isso, também temos a possibilidade de combinar o código JavaScript com outro código nativo em Java, Objective-C ou Swift, se for o caso de você querer utilizar componentes já prontos, ou até otimizar alguns aspectos da aplicação. Pensando em produtos legados, essa é uma grande sacada. Afinal, conseguimos inserir funcionalidades novas aos poucos e fazer a migração gradual das tecnologias (quem nunca sofreu com código legado, não é mesmo?).

Por fim, outra funcionalidade que vale a pena ser mencionada é a possibilidade de depuração da aplicação pelo Dev Tools (ferramentas de desenvolvedor) do Google Chrome, como se fosse uma aplicação web padrão. Para quem já está acostumado a usá-lo no desenvolvimento web se sentirá em casa.

1.4 O QUE VEM POR AÍ

No mundo da tecnologia nada se mantém estático por muito tempo e com o React Native não teria de ser diferente. Para

conseguir acompanhar quais são as últimas novidades e as promessas de funcionalidades futuras, existem três caminhos que acabam convergindo:

- O blog oficial (<https://facebook.github.io/react-native/blog/>);
- O repositório `react-native-releases` (<https://github.com/react-native-community/react-native-releases/>);
- O repositório `discussions-and-proposals` (<https://github.com/react-native-community/discussions-and-proposals/>).

Dentro do blog, os mantenedores nos fornecem notícias quentes sobre o que acontecem nos bastidores do desenvolvimento do framework, principalmente no que diz aos lançamentos fechados. Já para aqueles que querem participar mais ativamente das discussões e propostas de funcionalidades, foram criados os dois repositórios citados. Eles funcionam de forma muito semelhante ao que o TC39 faz com o ECMAScript, ou seja, os usuários podem fazer propostas de funcionalidades - sempre explicando por quê, como e para quê - e estes pedidos são levados a discussão, sendo eventualmente adotados ou rejeitados. É um jeito bem bacana e democrático das pessoas participarem do desenvolvimento do framework.

Conclusão

Neste capítulo demos uma breve olhada no histórico do React Native. Estudamos sobre o seu surgimento, sua proposta e consequentemente sua adoção e sucesso na comunidade e no

mercado de desenvolvimento em todo o mundo. Ao final, vimos também como nos manter atualizado sobre as novidades da tecnologia através dos canais oficiais. No próximo capítulo, começaremos a preparar o nosso ambiente de desenvolvimento para desenvolver os projetos.

INSTALAÇÃO E CONFIGURAÇÕES INICIAIS

Se você estivesse lendo este livro alguns anos atrás — por volta de 2015 ou antes — provavelmente se assustaria com a quantidade de configurações necessárias para montar o ambiente de desenvolvimento nativo para criarmos um aplicativo para smartphones, principalmente para o sistema Android (antes usávamos alguns plugins do Android dentro do Eclipse, a principal IDE para quem usa Java; hoje já existe o Android Studio que facilitou tremendamente a nossa vida). Felizmente, com o passar dos anos, tanto a Apple quanto o Google ouviram o feedback dos seus usuários e tornaram seus ambientes mais acessíveis aos programadores de primeira viagem.

O mesmo vale para o desenvolvimento com o React Native. Nos primórdios da tecnologia era bem complicado, chato e demorado subir todo o ambiente para desenvolver. Não que hoje configurar o ambiente de desenvolvimento tenha se tornado uma alegria, no entanto, já existem alternativas bem mais fáceis. Neste livro vamos explorar uma delas em especial, uma que permite que possamos realizar os testes diretamente no nosso aparelho.

Estamos falando do Expo.

2.1 EXPLORANDO O EXPO

Caso já tenha trabalhado em algum momento com o React para desenvolvimento web, existem grandes chances de que você já tenha ouvido falar do pacote `create-react-app` (<https://github.com/facebook/create-react-app/>). Caso não o conheça e/ou não se lembre exatamente para que ele serve, vamos refrescar sua memória.

O `create-react-app` é uma ferramenta de código aberto — atualmente na sua versão 2.x — criada pelo time de engenheiros do Facebook para facilitar o desenvolvimento de aplicações React. Do mesmo jeito que tínhamos um trabalhão para conseguir subir ambientes Android para desenvolvimento nativo, no React tínhamos o mesmo problema (diríamos que até maior, pois não bastava apenas configurar e encaixar as ferramentas necessárias, era necessário saber como saber usar cada uma delas).

Para nos ajudar com isso, o projeto surgiu com uma premissa muito simples: o usuário, com apenas um comando no terminal, criará toda a estrutura necessária para sair desenvolvendo sua aplicação. Todas as bibliotecas e ferramentas são baixadas, instaladas e configuradas para que possamos nos concentrar no mais importante, as regras de negócio das nossas aplicações.

Mas o que tudo isso significa na prática? Significa que não precisamos nos preocupar (inicialmente) em configurar *bundlers* (ex.: Webpack), *transpilers* (ex.: Babel), utilitários de testes (ex.: Jest) e afins para ter uma aplicação otimizada para produção

(*production ready*). O `create-react-app` já nos dá tudo isso de graça. Literalmente não tem como deixar a vida do desenvolvedor React iniciante mais fácil. Incrível, né?

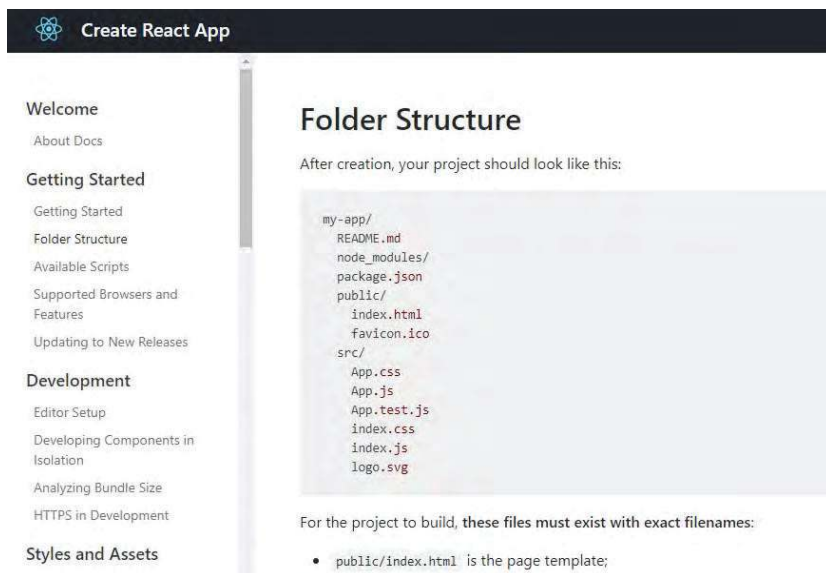


Figura 2.1: Estrutura do projeto React web criado com o `create-react-app`. Fonte: <https://facebook.github.io/create-react-app/docs/folder-structure>

Neste ponto esperamos que você esteja se perguntando: “*Mas o que tudo isso tem a ver com o tal do Expo?*”. E nós respondemos: Tudo! Do mesmo jeito que o `create-react-app` está para o React, o Expo está para o React Native. Em vez de termos que configurar tudo “na unha”, o Expo nos fornece todo o ambiente necessário para desenvolver com tranquilidade nossos aplicativos. Isso será de grande ajuda para darmos o pontapé inicial na tecnologia.

2.2 INSTALANDO E USANDO O EXPO

O Expo — que atualmente se encontra na versão 31.0.0 — é um conjunto de ferramentas de código aberto que nos auxilia no desenvolvimento de aplicações React Native. Ele nos fornece todo o ecossistema necessário para criar, testar e rodar o build da aplicação, tudo de maneira muito simples e fácil. O primeiro passo para utilizá-la é acessar o site oficial do Expo (<https://expo.io>) e baixá-la no seu computador. Uma nota importante é que para isso é necessário ter o Node.js (<https://nodejs.org/en/>) instalado na máquina.

NOTA: Instalação do Node.js

A instalação do Node.js na sua máquina dependerá muito do sistema operacional que você está utilizando. Seja Windows, MacOS ou uma distribuição Linux, o Node.js funcionará perfeitamente. Para saber exatamente como instalar na sua plataforma, siga as instruções do site oficial (<https://nodejs.org/en/download/>).

Uma vez que o tenha instalado, basta executar o seguinte comando no terminal (ou prompt de comando) do seu computador:

```
npm install expo-cli --global
```

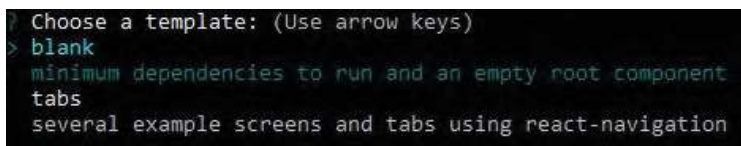
Note que o `expo-cli` é instalado de forma global (por isso usamos o parâmetro `--global`) na sua máquina através do `npm`. Isso torna a ferramenta independente do projeto ou do diretório

em que estivermos trabalhando. Tendo isso em mente, através do terminal entre no diretório em que deseja trabalhar e execute o seguinte comando:

```
expo init nome-do-projeto
```

De maneira análoga ao `create-react-app`, o comando `init` do Expo criará todo o ambiente de desenvolvimento necessário para criar um projeto preparado para produção. Durante esse processo, aparecerão duas perguntas que precisaremos responder através das setas do teclado. São elas:

1. Escolha do template:
2. `blank` : Template vazio com o mínimo de dependências possíveis para rodar o App.
3. `tabs` : Template com várias telas de exemplo e guias usando a navegação do React.



```
Choose a template: (Use arrow keys)
> blank
  minimum dependencies to run and an empty root component
  tabs
  several example screens and tabs using react-navigation
```

Figura 2.2: Escolha do template no Expo

1. Escolha do workflow (fluxo de trabalho):
2. `managed` (padrão): Faz o build do app com JavaScript e APIs Expo.
3. `advanced` (experimental): Faz o build do app com JavaScript, APIs Expo e módulos nativos personalizados.


```
Choose which workflow to use: (Use arrow keys)
> managed (default)
  Build your app with JavaScript with Expo APIs.
  advanced (experimental 🚧)
  Build your app with JavaScript with Expo APIs and custom native modules.
```

Figura 2.3: Escolha do workflow no Expo

Para este primeiro exemplo, utilizaremos o template `tabs` e o workflow `managed`. Essas escolhas significam que criaremos um app com várias telas usando o processo de build padrão do utilitário do Expo. Logo após responder essas questões, precisaremos dar o nome do app que ficará visível na primeira tela.

```
Please enter a few initial configuration values.
Read more: https://docs.expo.io/versions/latest/workflow/configuration » 100% completed
{
  "expo": {
    "name": "teste-app",
    "slug": "app"
  }
}
```

Figura 2.4: Inserindo o nome do aplicativo no Expo

Inicialmente colaremos o nome de `teste-app`. E pronto, agora o processo de criação do projeto será iniciado. O processo em si geralmente demora vários minutos, afinal, são vários e vários pacotes do npm que são baixados para garantir o pleno funcionamento de tudo. Ao final, teremos a seguinte estrutura de projeto:

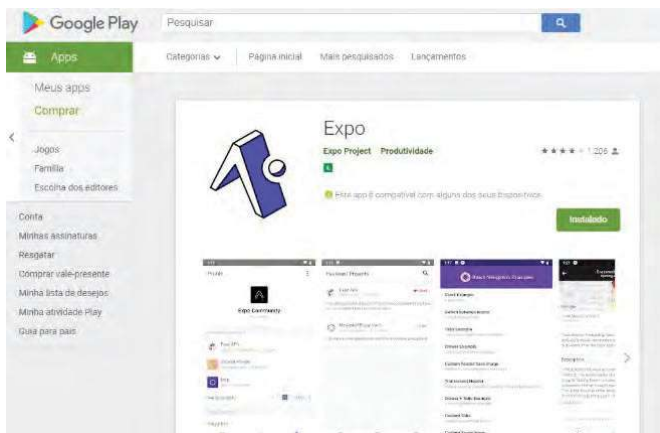


Figura 2.6: Aplicativo do Expo na Google Play. Fonte: <https://play.google.com/store/apps/details?id=host.exp.exponent&referrer=www>

Na App Store da Apple será algo assim:

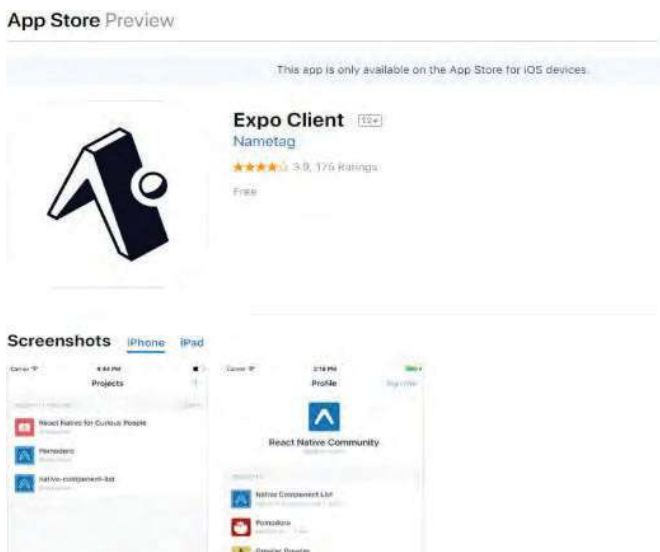


Figura 2.7: Aplicativo do Expo na App Store. Fonte: <https://itunes.apple.com/app/apple-store/id982107779>

Com tudo isso feito, chegou a hora da grande mágica: vamos executar o projeto. Entre no diretório do seu projeto através do terminal e execute o seguinte comando:

```
npm start
```

Neste momento, uma página do Developer Expo Tools se abrirá automaticamente no seu navegador no endereço `http://localhost:19002/`, conforme imagem a seguir:

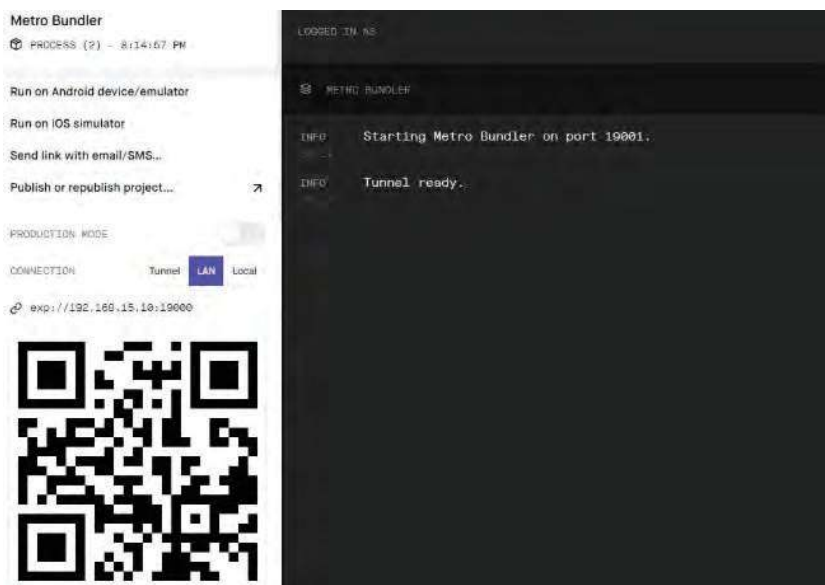


Figura 2.8: Tela de gerenciamento do Expo

Vamos parar um instante para entender todas estas informações desta página. Repare que do lado direito temos o terminal de informações que mostra tudo sobre o processo de build: logs, erros e warnings. Tudo é exibido nessa parte da janela. Do lado esquerdo, temos o menu de opções. Lá há quatro opções

disponíveis:

1. **Run on Android device/emulator:** usaremos esta opção quando quisermos visualizar nossa aplicação diretamente em um emulador do Android ativo na máquina.
2. **Run on iOS simulator:** usaremos esta para quando estivermos tratando de um simulador iOS ativo na máquina.
3. **Send link with email/SMS:** aqui temos a possibilidade de receber um e-mail ou SMS com link direto para a aplicação.
4. **Publish or republish project:** nesta última opção do menu, podemos publicar o nosso aplicativo no site oficial do Expo. Através de um emulador, ele possibilita que os usuários consumam sua aplicação por meio do navegador.

Por enquanto não usaremos nenhuma destas opções, pois faremos uso da funcionalidade de Live no próprio aparelho. Para isso, abra o aplicativo do Expo no seu aparelho e então utilize a câmera para escanear o QR Code gerado no canto inferior esquerdo.

Projects

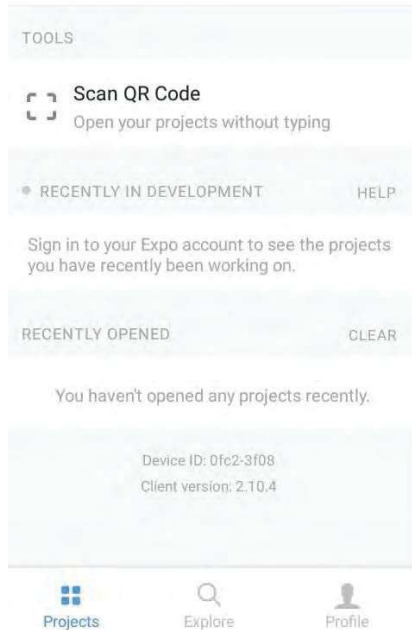


Figura 2.9: Tela do App Expo aberta no celular

NOTA IMPORTANTE: o celular e o computador precisam estar na mesma rede de wi-fi!

Depois que terminar o build, você verá a seguinte tela no seu dispositivo:

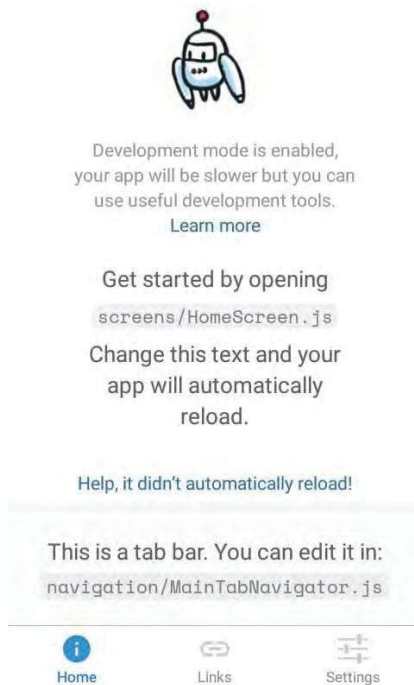


Figura 2.10: Tela do App Expo aberta no celular

Incrível, né? E o mais legal de tudo é que todas as modificações feitas no código são refletidas instantaneamente na tela do aparelho. Para ver a mágica acontecendo, usaremos o editor Visual Studio Code (<https://code.visualstudio.com/>) da Microsoft para mexer no código do projeto. Caso você tenha um outro editor de paixão, como os excelentes Atom (<https://atom.io/>) ou o Sublime (<https://www.sublimetext.com/>), também é possível utilizá-los sem problema algum. Somente recomendamos que verifique se o editor tem suporte visual para arquivos no formato JSX, caso contrário, isso poderá atrapalhar no momento em que você estiver

Abra o projeto no editor e procure pelo arquivo `HomeScreen.js` que está contido na pasta `screens`. Mudaremos o texto **Get Started by opening** para **Testando o reload automático** e salvar o arquivo.

Figura 2.11: Alteração de texto na página Screen

This document is available free of charge on:

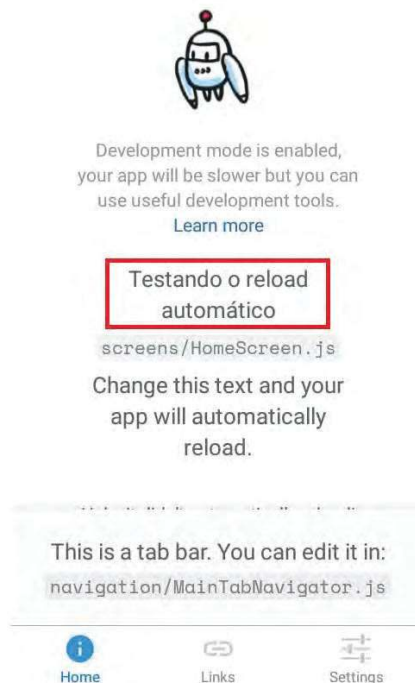


Figura 2.12: Reload Automático do Expo

E parece mágica! Apenas com estes simples passos já temos todo o nosso ambiente configurado para os próximos capítulos. Bem prático, certo? Mas a mágica não termina por aí. Assim como Expo é capaz de compilar e gerar o aplicativo, ele também nos ajuda na identificação de erros no código - e acredite, isso tem um valor inestimável. Para ter um exemplo, experimente alterar o arquivo `HomeScreen.js` incluindo os caracteres `!@#$$%` no final do arquivo. Não há dúvidas de que o JavaScript será incapaz de entender o que é este código, e é por isso mesmo que na tela do seu aparelho uma tela vermelha semelhante a esta vai aparecer:

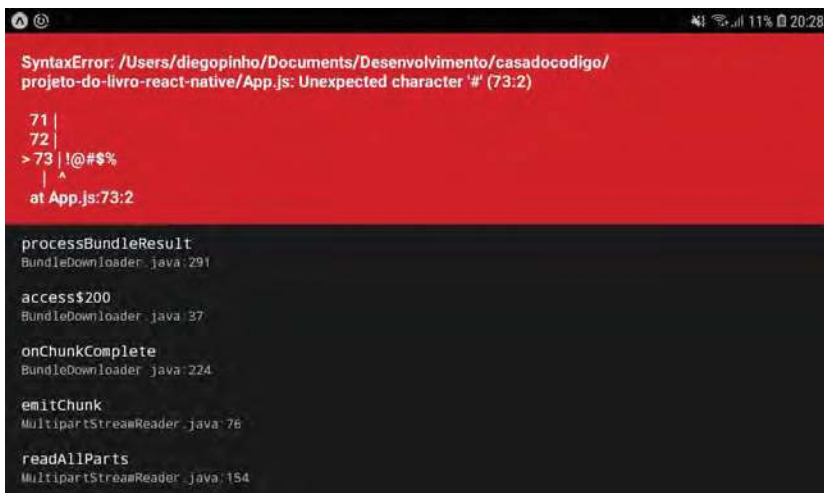


Figura 2.13: Pilha de erros do código apresentados pelo Expo

Veja que o Expo nos apresenta a pilha inteira de erros, apontando qual é o problema, onde foi identificado, e quais foram os métodos executados até que aquele código fosse executado. Assim que o código é alterado e salvo novamente, se o erro não persistir a tela do aplicativo será mostrada novamente no aplicativo.

Estas funcionalidades serão de grande ajuda na nossa jornada no desenvolvimento de aplicativos. Experimente fazer mais algumas alterações e veja como o Expo reage. É importante estarmos confortáveis com o seu ecossistema antes de começarmos.

NOTA: Lags e problemas de conexão com o Expo

Durante o uso do Expo, é possível que você note alguns problemas de lag e conexões entre o seu aplicativo no computador e no aparelho celular. Em alguns momentos o código salvo na máquina não é enviado ao aparelho, dando a sensação de que nada está funcionando. Nestas situações apenas reinicie todo o processo. No terminal finalize o processo e no aparelho feche e abra novamente o aplicativo do Expo. Na maior parte das vezes isso resolverá os problemas.

Para maiores informações, confira a documentação do Expo (<https://docs.expo.io/versions/latest/>)

Conclusão

Neste capítulo passamos rapidamente pelo processo de criação, build e teste de um projeto feito com o React Native por meio do utilitário do Expo. Passamos por uma série de comandos e telas que começarão a fazer parte do nosso dia a dia como desenvolvedores mobile com React Native. No próximo capítulo, vamos explorar as funcionalidades do React Native. Por isso, é importante que você configure todo o seu ambiente antes de seguir em frente. Sabemos que esse processo é o mais chato, mas pense pelo lado positivo: só o faremos uma vez. Feito isso, vamos estar prontos de corpo, mente, alma e computador para seguir adiante com os projetos.

FUNCIONAMENTO DO REACT NATIVE

Neste capítulo começaremos a investigar e entender os tópicos fundamentais sobre o funcionamento do React Native. Isso será extremamente importante para compreender como ele trabalha por debaixo dos panos. Assim, além de saber usar a tecnologia, seremos capazes de reproduzi-la, incrementá-la e, principalmente, entender e nos livrar dos "problemas cabeludos" quando eles aparecerem.

Para fazer tudo isso, tomaremos a mesma estratégia do capítulo anterior e criaremos um novo projeto do zero e entender toda a sua estrutura junto aos conceitos que discutiremos daqui em diante.

3.1 POR DEBAIXO DOS PANOS - REACT

Para sermos capaz de entender e usar o React Native, primeiro temos que entender como funciona o seu padrinho tecnológico, o React para web. Como definido pelos próprios desenvolvedores no site oficial, o React é "uma biblioteca JavaScript declarativa, eficiente e flexível para a criação de interfaces de usuário". Ela surgiu em meados de 2011, no Facebook, e passou a ser utilizada

como solução interna na interface do mural de notícias (feed) da rede social. No ano seguinte, passou a integrar também a rede social Instagram e várias outras ferramentas internas da empresa. Já em 2013, o código foi aberto para a comunidade, o que colaborou para sua grande popularização.

A grande sacada do React é que ele é extremamente rápido e flexível, tanto para uso quanto para aprendizado, pois ele trata tudo como componentes. Se você tem uma boa experiência com desenvolvimento web, deve estar acostumado a ter todo o código HTML de uma página em um único arquivo, assim como os arquivos CSS e JavaScript respectivos. O que o React propõe é totalmente o contrário: separar todo o código em pequenas partes (em arquivos diferentes), que se comportam como componentes reutilizáveis. Parece confuso? Vamos ver um exemplo fictício para entender melhor.

Vamos pegar como exemplo a página principal da nossa editora, a Casa do Código (<https://www.casadocodigo.com.br/>).



Figura 3.1: Página principal do site da Casa do Código

Mesmo sem saber como esta página foi desenvolvida internamente, podemos pegar como base a sua interface e então dividi-la em pedaços que poderiam ser reaproveitáveis. Segue uma sugestão bem simples:

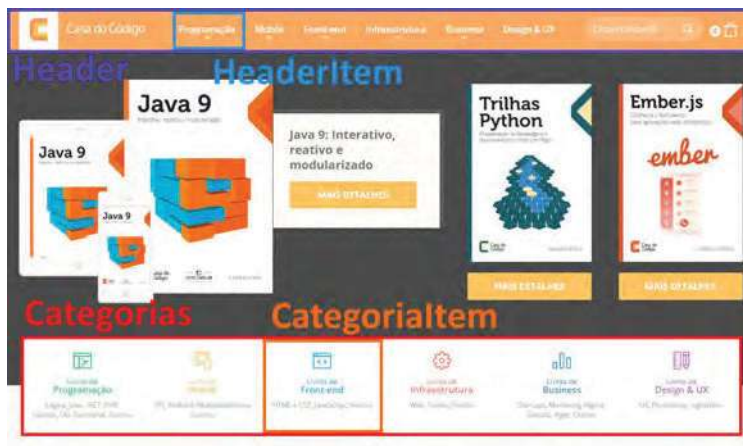


Figura 3.2: Página principal do site da Casa do Código dividida em componentes

Olhe atentamente ao que fizemos. A parte superior separamos em um componente chamado `Header`. Dentro deste componente, temos seis itens muito parecidos, que são as categorias de livros disponíveis. Como eles são muito semelhantes, separamos em outro componente chamado `HeaderItem`. Ou seja, até o momento temos um componente `Header` que é o container de seis componentes do tipo `HeaderItem`. Além disso, também temos o logo, barra de busca e sacola de compras que poderiam ser outros componentes (só não marcamos na imagem para que não ficasse muito confuso).

Logo depois, temos uma divisão muito semelhante. Separamos o contêiner de categorias em um componente chamado

Categorias e dentro chamamos cada categoria de `CategoriaItem`. Até aqui muito simples, certo? Pois bem, a ideia do React é exatamente essa: dividir a tela em componentes reaproveitáveis. É extremamente provável que você tenha pensando em outras divisões para esta página, e isso é realmente esperado. Por isso falamos que o React é uma biblioteca flexível: a divisão dos componentes sempre ficará a cargo do programador responsável.

Legal, né? Agora, é possível que você esteja se perguntando: "mas qual é a melhor forma de separar um layout em componentes?". E é aí que entra a parte complicada (você sabia que ela chegaria)... Não há fórmula para isso. A divisão em componentes sempre dependerá da sua preferência pessoal, de sua equipe e do seu projeto. O que precisamos ter sempre em mente é que o componente deve ser uma unidade independente e reaproveitável. Isso é muito importante. O quanto mais conseguirmos definir estruturas que sigam essa filosofia, melhor o projeto se mantém. E quando estamos falando de componentes web, o mesmo caberá para o React Native. Veremos adiante que a construção do aplicativo seguirá a mesma filosofia. O que temos para o React para web também teremos no React Native.

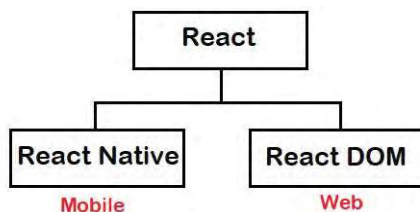


Figura 3.3: Biblioteca React é o centro

Além dos componentes, o React também trabalha com o

conceito de Virtual DOM. Este DOM é uma representação levíssima criada pelo React do DOM (Document Object Model) gerado pela engine do JavaScript no navegador. Toda vez que um componente é renderizado na tela, o React atualiza o Virtual DOM de cada componente já renderizado e procura por mudanças estruturais (ex.: mudança de um texto em um input). E como o Virtual DOM é leve, esse processo é muito rápido. O React então compara o Virtual DOM com uma imagem do DOM feita antes da atualização e descobre o que realmente mudou, atualizando somente no DOM do navegador os componentes que mudaram de estado. Isso torna o processo extremamente rápido e eficaz.

Em resumo, o React computa na sua representação do DOM as mudanças antes de enviá-las para o navegador. Neste processo ganhamos muita rapidez e eficiência, afinal, operações no DOM são custosas (apesar de não parecer).

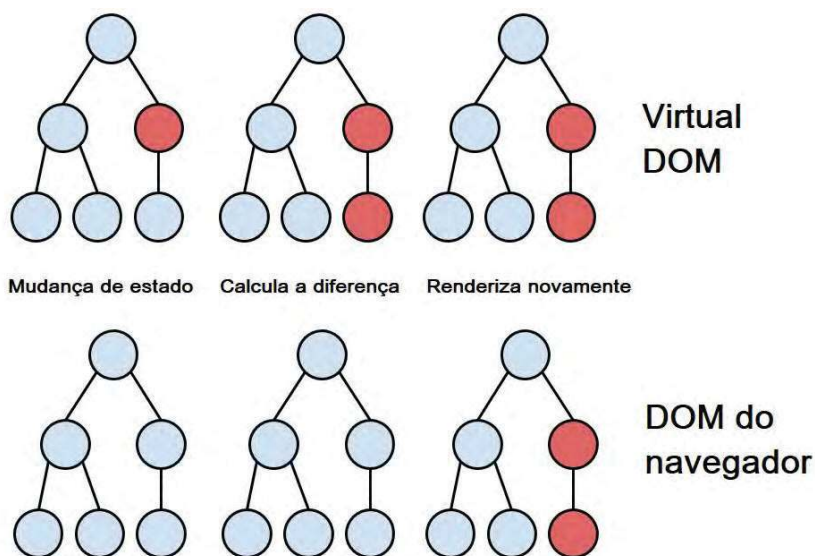


Figura 3.4: Biblioteca React é o centro

Mas ao contrário das aplicações web e o React que utilizam o objeto DOM criado pela engine dos navegadores para representar a estrutura virtual dos elementos renderizados na tela, o React Native realiza chamadas nativas para o dispositivo mobile (Java para Android e Objective-C/Swift pra iOS) a fim de renderizá-los. Esta comunicação entre JavaScript do React Native e o Java/Objective-C/Swift é feita através da chamada Bridge; que como o próprio nome diz, é a ponte que liga os dois lados.

É importante saber que os projetos em React Native rodam diretamente no dispositivo do usuário (nativamente), e para isso, contamos com a seguinte estrutura:

Em dispositivos Android:

- Android Runtime: para o código Java.
- JSCore Virtual Machine: para o código JavaScript.

Em dispositivos iOS:

- Native Runtime: para códigos Objective-C/Swift.
- JSCore Virtual Machine: para o código JavaScript.

Fora isso, é importante saber que existem três processos principais que são executadas em uma aplicação React Native:

1. A primeira é chamada de *Main thread* (processo principal), e está do lado nativo. Ela é responsável por tratar as requisições relacionadas à renderização de elementos na tela e também os toques do usuário na tela do dispositivo;
2. A segunda *thread* é responsável por executar o código JavaScript, que, por sua vez, é responsável por toda a lógica de negócio da aplicação, além de definir a estrutura e as

- funcionalidades da interface gráfica (UI);
3. A terceira *thread* é a chamada *Shadow Queue*, que é responsável pelos cálculos referentes ao layout.

NOTA: É importante mencionar que todo início de comunicação é iniciado pelo lado nativo, já que os eventos de toque na tela são tratados na parte nativa da aplicação.

A interface gráfica tanto do React como do React Native são desenvolvidas em JSX, uma extensão da sintaxe do JavaScript que lembra HTML/XML, porém é uma mistura dos dois. Nesta estrutura, conseguimos escrever HTML, CSS e JavaScript em um único arquivo. É isso mesmo que você leu, essas três tecnologias distintas em um único arquivo. Conseguimos inserir tags dentro de variáveis, chamar funções dentro de tags, colocar regras de CSS em objetos JavaScript, e daí em diante. Parece uma loucura, mas não há nada a temer. Apesar de parecer confuso à primeira vista, a sintaxe do JSX é bem simples e nos dá o poder de criar componentes que contém toda a estrutura necessária para o seu funcionamento.

Um código JSX é muito próximo ao código a seguir:

```
ReactDOM.render(  
  <h1 style={{color: 'red'}}>Estou aprendendo React Native</h1>,  
  document.getElementById('container')  
)
```

Vamos analisar o que está acontecendo neste trecho. O código chama uma função `render` do objeto `ReactDOM` passando dois

parâmetros: um deles é o componente/html que deve ser renderizado, aqui representado pela tag `h1` do HTML5 com a frase "Estou aprendendo React Native"; o segundo parâmetro é uma função clássica da API do JavaScript nos navegadores para buscar o elemento de id `container` do DOM. Além disso, a tag `h1` do primeiro parâmetro possui um estilo CSS para mudar a cor deste texto.

Viu só como conseguimos encaixar o HTML, CSS e JavaScript em uma estrutura só?

Só que o lado nativo (tantos os navegadores quanto os smartphones) não sabem como interpretar esta linguagem JSX. Para resolver isso, o React Native transforma o elemento JSX para uma linguagem de marcação que pode ser entendida pelo lado nativo. Esta linguagem de marcação determinará como o elemento de nossa interface gráfica deverá ser. Então, os componentes visuais serão de fato componentes nativos do dispositivo. (ex.: a tag `<View>` do React Native que veremos logo mais é transformada em `Android.view` no Android e `UIView` no iOS.)

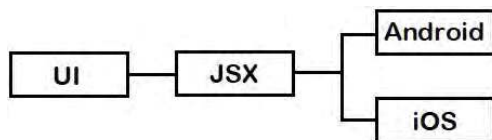


Figura 3.5: JSX é transformado em linguagem nativa

3.2 ENTENDENDO O ARQUIVO APP.JS

Agora que sabemos que o React Native é baseado no React e, por isso, utiliza a ideia de componentização e a sintaxe de JSX para compactar todo HTML, CSS e JavaScript necessário para seu

funcionamento, estamos prontos para entender melhor os arquivos gerados pelo Expo quando um novo projeto é criado. Para começar, vamos relembrar os passos do capítulo anterior e criar um novo projeto. Você lembra como se faz? Não? Tudo bem, vamos dar uma mãozinha. Vá até o diretório no qual você deseja salvar a aplicação e depois execute o comando no seu terminal (trocando a lacuna `<nome_do_projeto>` para o real nome do seu projeto):

```
expo init <nome_do_projeto>
```

Desta vez, escolha o template `blank` e o `workflow managed`.

Depois que tudo for criado, abra o projeto no seu editor de preferência. Por aqui, vamos usar o Visual Studio Code. No terminal, estando dentro da pasta do projeto, execute o comando `expo start` para gerar o QR Code que vamos ler com o aplicativo do Expo que já temos instalado no celular. Agora que temos tudo rodando, vamos parar para entender a estrutura da aplicação que foi gerada:

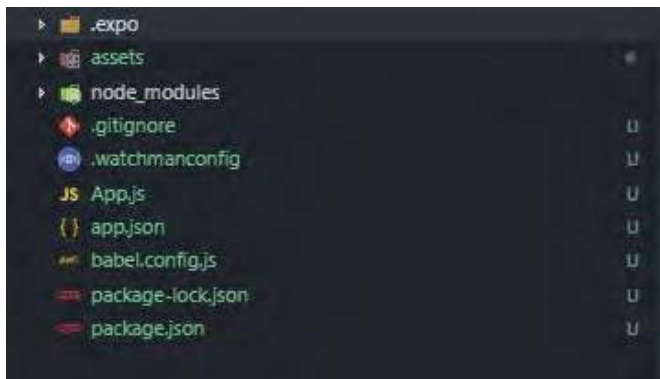


Figura 3.6: Estrutura da aplicação

A primeira pasta é a `.expo` (note que ela é oculta). É nela que estão contidos os arquivos de informação e configuração interna do Expo. Não precisaremos saber detalhes profundos sobre eles, mas caso queira, a documentação do utilitário está impecável (<https://docs.expo.io/versions/latest/>). A segunda é a pasta `/assets`, onde estão contidas as imagens da aplicação. Toda imagem que vamos colocar do nosso app deverá ser colocada nessa pasta. A terceira é o `/node_modules`, onde estão contidas as dependências do projeto. O próximo arquivo é o `.gitignore`. É nele que é informado quais arquivos e pasta devem ser ignorados ao subir o projeto no controle de versionamento do git. (inclusive, note que o Expo já nos fez um favorzão de tirar a pasta `/node_modules` dos commits. Amém!).

Depois dele temos o arquivo `.watchmanconfig`, que, como o próprio nome já sugere, é o arquivo de configuração do Watchman (<https://github.com/facebook/watchman/>). Este é outro projeto de código aberto do Facebook que tem como objetivo fazer o monitoramento de arquivos e torna possível o Live Reload no desenvolvimento.

Logo em seguida temos o `App.js`, que é o arquivo (e componente) ponto de entrada do projeto. Quando o projeto é construído e renderizado, o Expo e o React Native procuram este arquivo como ponto de entrada do aplicativo. Aqui vale lembrar que o React também é caracterizado por ser uma tecnologia de Single Page Application (SPA), ou seja, na prática sempre o que teremos no aplicativo (ou na solução web) é uma única página onde os componentes são renderizados dando a impressão de que o usuário está navegando por páginas diferentes. Logo, é de extrema importância que o React Native saiba por onde começar.

A seguir temos o arquivo `app.json` que contém metadados do aplicativo, tais como: nome, versão, dependências, entre outras configurações do projeto. Depois temos o `babel.config.js`, que é o arquivo de configuração do transpilador Babel, responsável por "traduzir" o JSX para a sintaxe web (HTML, CSS e JS). Isso é necessário pois o JSX, por mais lindo que seja, não é uma tecnologia interpretável pelos navegadores e plataformas. Para que possamos usá-los precisamos de um intermediário que possa fazer essa tradução para nós. É aí que o Babel entra (inclusive para o React web).

Por fim, temos o `package-lock.json` e o `package.json`. Lá já temos os scripts de build para Android e iOS, além do `test`, `start` e por fim o `eject`, que pode ser usado para "ejetar" as configurações de build para a sua aplicação. Isso significa que o Expo sai de cena e expõe todas as configurações que ele faz por debaixo dos panos. Como estamos dando os primeiros passos com o React Native, não nos preocuparemos com isso agora.

Dada a estrutura, vamos clicar no arquivo que dissemos ser o ponto de entrada do aplicativo. Estamos falando do `App.js`:

```
import React from 'react';
import { StyleSheet, Text, View } from 'react-native';

export default class App extends React.Component {
  render() {
    return (
      <View style={styles.container}>
        <Text>Open up App.js to start working on your app!</Text>
      </View>
    );
  }
}

const styles = StyleSheet.create({
```

```

    container: {
      flex: 1,
      backgroundColor: '#fff',
      alignItems: 'center',
      justifyContent: 'center'
    }
  })

```

Aqui há uma série de coisas interessantes que precisamos entender. Na primeira linha importamos o `React` da biblioteca do `React` para dentro do nosso (para que assim possamos usar seus mecanismos de componentização) e na segunda temos a importação de três componentes contidos na biblioteca do `React Native`, que mais tarde serão convertidos em componentes nativos. São eles o `StyleSheet`, `Text` e `View`.

Na linha seguinte, é feita a exportação da classe principal `App`, que é estendida do componente principal do `React`. Esta é a forma de criação de um componente que chamamos de "baseado em classe" (no próximo tópico veremos a outra opção que é a criação de um componente baseado em função). Dentro do corpo da classe `App.js`, temos a função `render`. Guarde o nome desta função pois ela é extremamente importante e os componentes `React` não vivem sem ela: ela que é responsável por retornar o(s) componente(s)/HTML que deverá ser renderizado. Neste código, estamos devolvendo uma tag `<View>` que possui uma tag `<Text>` aninhada. Como veremos adiante, a `<View>` representa praticamente uma `<div>` enquanto o `<Text>` representa um `<p>`.

Fora da classe temos um objeto literal chamado `styles` criado por meio da função `create` do objeto `StyleSheet` importado do `react-native`. Este objeto possui regras CSS para a estilização do componente. Repare que a definição da tag

`<View>` possui um atributo chamado `style` que aponta para este objeto. Tente se acostumar com este tipo de declaração pois a usaremos constantemente no decorrer do livro.

Tendo entendido este código, tente alterar o texto "Open up App.js to start working on your app!" e as regras de CSS declaradas no arquivo `styles`. Brinque, explore, investigue e sempre confira os resultados na tela do seu dispositivo mobile. Depois de se acostumar um pouco mais com essa sintaxe, prossiga para o próximo tópico.

3.3 COMPONENTE FUNCIONAL

No React temos duas formas de criar componentes. Uma é o componente baseado em classe (que vimos no tópico anterior) onde importamos o React e criamos uma classe que estende o `React.Component` e implementa a função `render` responsável por devolver o componente que deverá ser renderizado (descrito por tags HTML). A outra opção que temos é o componente baseado em função (componente funcional). Esse componente é estruturalmente mais simples: basta definirmos uma função que devolve o que deverá ser renderizado. Esta função é como se fosse o `render` do componente de classe:

Por exemplo:

```
export default function() {  
  return <Text>Hello World</Text>  
}
```

O que precisamos saber é que este tipo de componente não sabe administrar estados (que veremos mais adiante). Ela é bem objetiva: basta criar uma função e retornar um trecho de código

JSX, que vai ser renderizado na tela. No decorrer do livro, vamos explorar melhor a utilização de cada um destes tipos, por ora basta que você saiba que ambas coexistem em um único projeto e que podemos optar pela utilização de somente uma delas (geralmente, a escolha é por componentes de classe pois eles possuem mais funcionalidades).

Conclusão

Neste capítulo nós mergulhamos de cabeça no funcionamento do React por debaixo dos panos (tanto para a versão web quanto para mobile). Entendemos a sua estrutura interna (arquitetura) e a forma de criar os componentes por meio do arquivo `App.js`. Depois, vimos como criar componentes funcionais. E para complementar tudo isso, entendemos o papel do Expo em todo este processo.

Durante este capítulo também vimos uma série de funcionalidades do JavaScript que são fruto da especificação ECMA-262 de 2015, também chamada de ECMAScript 2015 (ES6). Dentre elas, vimos:

- `import`
- `export`
- `class`
- `extends`

Neste livro não vamos abordar com detalhes o funcionamento destes mecanismos, mas caso você queira um apoio, recomendamos a leitura do livro citado na introdução: *ECMAScript 6: Entre de cabeça no futuro do JavaScript* (<https://www.casadocodigo.com.br/products/livro-ecmascript6>).

Lá você vai aprender sobre todas as novas normas e regras do JavaScript que vamos usar bastante neste livro.

Na próxima etapa aprenderemos como criar componentes do zero.

CRIANDO OS PRIMEIROS COMPONENTES

No capítulo anterior, vimos com detalhes o funcionamento básico do React para o desenvolvimento web e como ele influencia diretamente a construção dos nossos aplicativos no React Native. Neste capítulo, daremos continuidade a estas explicações focando especialmente em aprender as principais formas da criação de componentes visuais dentro do framework. Partiremos da forma mais simples possível e então conheceremos gradualmente outras formas mais complexas. Para cada uma delas, discutiremos seu funcionamento e em quais as situações cada uma pode ser mais adequada que a outra.

4.1 CRIANDO UM COMPONENTE E IMPORTANDO NO APP.JS

Para começarmos nossos estudos da melhor maneira possível, inicie um novo projeto React Native utilizando o Expo. Depois disso, teremos a estrutura que exploramos passo a passo no capítulo anterior. Com isso tudo em mãos, começaremos criando um novo componente para a nossa aplicação e depois vamos importá-lo no arquivo principal `App.js` (lembrando que este é o

ponto de entrada da nossa aplicação). Pensando na organização geral do nosso projeto, adotaremos uma prática de mercado e criaremos uma pasta na raiz do nosso projeto chamada `componentes`. Dentro dela, criaremos todos os nossos componentes e, sendo necessário no futuro, podemos até mesmo criar outros diretórios internos e ir organizando os componentes. Por ora, vamos criar um novo arquivo chamado `OlaMundo.js` na raiz da pasta.

É neste arquivo que criaremos o nosso novo componente. Para isso, nas duas primeiras linhas faremos a importação da biblioteca do `React` e os componentes `Text` (<https://facebook.github.io/react-native/docs/text.HTML>) e `View` (<https://facebook.github.io/react-native/docs/view.html>) da biblioteca do `react-native`. Lembrando um pouco do que dissemos no capítulo anterior enquanto explicávamos como o `React` funciona, os componentes nativos `Text` e `View` do `React Native` são como as tags `HTML` `<div>` e `<p>` que usamos na construção de aplicações web. Eles funcionam como contêineres de conteúdo e são componentes fundamentais para a construção de interface de usuários. Eles trabalham de forma que conseguimos deixar o conteúdo (textos, imagens, vídeos e afins) organizado, estilizado e interativo dentro das nossas aplicações. Tanto um quanto o outro dão suporte a aninhamento, estilos `CSS` e controle de toque do usuário. Vamos usá-los com bastante frequência.

```
import React from 'react';
import { Text, View } from 'react-native';
```

Agora que importamos tudo o que precisamos, vamos criar o nosso primeiro componente funcional do zero. Caso a memória não esteja fresca, chamamos de componentes funcionais aqueles

que são construídos com base nas funções em vez de nas classes, o que significa que eles devem retornar uma função que nos devolve um conteúdo para devolver na tela. Essa função é análoga à função `render` dos componentes de classe. Por enquanto, como não pretendemos lidar com dados complexos e nem estados, criaremos o componente funcional.

Para este componente, devolveremos apenas um texto dizendo "Olá mundo!". Para isso, colocaremos em ação os componentes nativos do React Native, o `View` e `Text`. Para fins de demonstração, usaremos o componente `View` para agrupar dois componentes `Text`.

```
export default function() {  
  return (  
    <View>  
      <Text>Olá</Text>  
      <Text>Mundo!</Text>  
    </View>  
  )  
}
```

Aqui existem dois detalhes que valem a pena serem mencionados. Primeiramente, repare que usamos o componente `View` para agrupar os componentes `Text`. Isso é importante pois a função deve retornar sempre somente um componente (ou `div/span`). Isso pode parecer óbvio, mas pode ser fruto de muitas frustrações quando você estiver trabalhando. Se tivéssemos tentado fazer algo como o representado a seguir, enfrentaríamos problemas, veja:

```
export default function() {  
  return (  
    // estamos devolvendo dois componentes que são irmãos (estão  
    no mesmo nível)  
    <Text>Olá</Text>  
  )  
}
```

```

    <Text>Mundo!</Text>
  )
}

```

Quando fazemos isso, o React nos devolve o seguinte erro: Adjacent JSX elements must be wrapped in an enclosing tag . Guarde bem esse erro e qual é o seu motivo. Muitas vezes acabaremos desenvolvendo nosso aplicativos e caindo nesse erro por esquecer de "encapsular" os componentes e tags. Mas não há nada temer, basta lembrar de que a função deve retornar apenas um elemento. Achemos que valeria a pena mencionar esse problema por ser extremamente comum no desenvolvimento React (nos agradeça depois).

Mas voltando ao arquivo `App.js` , faremos a importação do componente `OlaMundo` que criamos para esse arquivo. Para conseguir fazer isso, temos que usar o mecanismo de importação/exportação do ECMAScript 2015 (ES6), da mesma maneira que estamos fazendo com os módulos do React. Em resumo, seu funcionamento é bem simples. Como usamos o `export default` apontando para a função no nosso componente, basta que importemos essa função para dentro do `App.js` usando o `import` e apontando para o caminho relativo do componente. No nosso caso, esse caminho é `./componentes/OlaMundo.js` . O topo do nosso código deverá ficar assim:

```

import React from 'react';
import { StyleSheet, Text, View } from 'react-native';
import OlaMundo from './components/OlaMundo';

```

Repare que aqui não temos a necessidade de inserir o sufixo do arquivo JavaScript. A própria engine do JavaScript assume que o arquivo se trata da extensão `.js` por padrão. Se não for o caso, aí

sim se faz necessário colocar sua extensão (como por exemplo, em arquivos de imagem ou json).

Agora que importamos tudo, precisamos chamar o componente no retorno da função `render` dentro da tag `View` (lembre-se, por padrão o Expo cria componentes de classe). Para chamar um componente, basta escrever o nome dele entre tags, como se fosse uma aplicação web.

```
<OlaMundo />
```

Como nosso componente não possui conteúdo interno (como o componente `Text`, por exemplo), podemos fechá-la desta maneira. O último detalhe que vale mencionar aqui é que o nome `OlaMundo` na tag corresponde ao nome inserido no `import`. Se tivéssemos chamado de `ChapeuzinhoVermelho` no `import`, a tag seria `<ChapeuzinhoVermelho>` mesmo que o nome do componente no arquivo seja `OlaMundo.js`. Fique sempre atento a isso e sempre que possível mantenha a consistência para evitar problemas do tipo.

No final de tudo teremos a seguinte estrutura no arquivo:

```
import React from 'react';
import { StyleSheet, Text, View } from 'react-native';
import OlaMundo from '../components/OlaMundo';

export default class App extends React.Component {
  render() {
    return (
      <View style={styles.container}>
        <OlaMundo />
      </View>
    );
  }
}
```

```
const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#fff',
    alignItems: 'center',
    justifyContent: 'center',
  },
});
```

Abra o terminal e entre na pasta do projeto. Feito isso, execute o comando `expo start` (ou `npm start`) para rodar nossa aplicação. Estando tudo certo, o utilitário então abrirá uma tela no seu navegador com o QR Code de acesso para sua aplicação. Com o aplicativo do Expo instalado no seu celular, leia o QR Code e espere até que o build seja completado. Após o build, note que o texto `Olá` aparecerá na tela do seu dispositivo.

Pronto, criamos nosso primeiro componente do zero!

4.2 ACESSANDO AS PROPRIEDADES DO COMPONENTE

Chegou a hora de falar sobre um aspecto importante dos componentes React: as propriedades. As propriedades, como o próprio nome já indica, representam atributos referentes aos componentes. Estas propriedades são informações que são passadas entre os componentes (normalmente pai para filho) e são o principal canal de comunicação entre eles. Como já mostramos anteriormente, seu funcionamento não tem nenhum segredo e é bem fácil.

Vamos colocá-lo em prática no nosso componente `OlaMundo`. No componente funcional que acabamos de criar, retornamos o texto `Olá Mundo` "amarrado" no componente `<Text>`. Vamos

aprender agora como passar o texto como propriedade para esse componente. Para isso, vá ao componente `App` e procure pela linha em que usamos o nosso componente `<OlaMundo />` dentro do componente `View`. Vamos criar um atributo chamado `nome` nesta tag e atribuir o valor "Jonas", conforme mostra o código a seguir:

```
<OlaMundo nome='Jonas' />
```

Veja que esta estrutura é exatamente a mesma que usamos com atributos HTML, a diferença aqui é que estamos usando um atributo inventado - e aqui vale citar um detalhe importante: o nome do atributo é totalmente arbitrário, ou seja, você pode utilizar o nome que achar mais conveniente. Este atributo agora deverá ser usado como uma propriedade dentro do aplicativo `OlaMundo`.

Volte ao arquivo `OlaMundo.js` e remova o texto "Mundo!" da segunda tag `<Text>`. Em vez disso, usaremos o valor passado pelo componente pai como propriedade. Para isso, primeiramente precisamos "habilitar" o nosso componente para que receba estas propriedades.

O jeito mais simples (e utilizado) para componentes funcionais é inserir o parâmetro `props` na função. Desta maneira:

```
export default function(props) {  
  return (  
    <View>  
      <Text>Olá</Text>  
      <Text></Text>  
    </View>  
  )  
}
```

O nome `props` é usado para acessar as propriedades de um

componente. Na prática, o que receberemos aqui é um objeto que possui uma série de chaves e valores, incluindo os valores passados como parâmetros para ele. Como definimos o nome da propriedade como `nome`, vamos usar `props.nome` para poder acessá-la. Se tivéssemos usando outro nome, tal como `texto`, teríamos que usar `props.texto`, e assim por diante - vale lembrar que o nome `props` também é arbitrário, nós o usamos por questão de convenção, mas a escolha no final é sua.

Dentro da tag `<Text>` vamos colocar o valor `props.nome`:

```
export default function(props) {  
  return (  
    <View>  
      <Text>Olá</Text>  
      <Text>{props.nome}</Text>  
    </View>  
  )  
}
```

Para conseguir usar o valor da variável `props.nome` dentro da tag, encapsulamos este valor dentro das chaves, `{}`.

Agora faça o build da aplicação e note que agora a frase que aparece no dispositivo é a que passamos por propriedade: "Olá Jonas!". Legal, né? Experimente mudar o valor do atributo `nome` para outros valores, incluindo uma string vazia. Tente adicionar e remover atributos e veja o resultado.

Nos próximos capítulos estudaremos como usar a estrutura de condicionais (if) dentro dos componentes, de modo que poderemos tratar com mais cuidado os valores que são passados por meio de parâmetros.

4.3 PROPRIEDADES EM COMPONENTES DE CLASSE

Não poderíamos deixar de finalizar este capítulo mostrando como utilizar as propriedades (props) em componentes de classe. O processo não é muito diferente, a diferença fundamental é que dentro de funções gerenciadas pelo React (como o `render`), conseguimos buscar as props utilizando o `this`. No componente `OlaMundo`, teríamos que acessar o `this`, então buscar o objeto `props` e então recuperar a propriedade que estamos buscando. Exatamente desta maneira: `this.props.nome`.

Para validar o funcionamento deste código, vamos transformar o nosso componente funcional em um componente de classe. Para tal, precisamos seguir três passos:

1. Criar uma classe
2. Implementar o método `render`
3. Usar o `this` para acessar o objeto `props`

O resultado desta refatoração deve ser o seguinte:

```
import React from 'react';
import {View, Text} from 'react-native';

class OlaMundo extends React.Component {
  render() {
    return (
      <View>
        <Text>Olá</Text>
        <Text>{this.props.nome}</Text>
      </View>
    )
  }
}
```

```
export default Olamundo;
```

E pronto! Nosso componente está apto a usar propriedades!

Conclusão

Neste capítulo fizemos muitas coisas novas importantíssimas. Criamos nosso primeiro componente funcional do zero inicialmente com um texto "amarrado" ao componente `Text`. Depois, aprendemos sobre o funcionamento de parâmetros no React e então usamos o parâmetro `props` no componente funcional para conseguir usar o valor passado pelo componente pai dentro de sua estrutura. Em seguida, usamos o poder evoluído do JavaScript para criar um componente de classe com suporte a propriedades. Ufa, quanta coisa! Experimente criar componentes e passar propriedades entre eles. É extremamente importante que você fique confortável com isso para que possamos continuar avançando.

No capítulo a seguir, começaremos a entender como estilizar os nossos componentes (sejam eles funcionais ou de classe).

COMPONENTES ESTILIZADOS (CSS-IN-JS)

Nos últimos capítulos estudamos bastante os fundamentos do React. Entendemos o que são componentes, para que eles servem, como podemos criá-los e até como passar informações de um componente para outro utilizando o conceito de propriedades (props).

Entretanto, até o momento, todos os aplicativos que fizemos têm algo em comum: eles não têm estilo nenhum. Seja no sentido literal ou figurado, em nenhum momento paramos para entender como podemos aplicar estilos CSS aos nossos componentes e dar vida ao nossos aplicativos. Chegou o momento de aprendermos como fazer isso.

Neste capítulo, aprenderemos como podemos estilizar os componentes React. Da mesma maneira que usamos o CSS para embelezar o nosso HTML no desenvolvimento web, vamos utilizá-lo para deixar o aplicativo apresentável aos usuários usando a técnica de CSS-in-JS. Para aprender como fazer isso, continuaremos trabalhando com o aplicativo iniciado no capítulo anterior.

5.1 APLICANDO ESTILOS

Até aqui, nosso componente `OlaMundo` exibe corretamente na tela o texto indicado por meio das suas propriedades, no entanto, não aplicamos nenhum estilo a ele. Para ser mais exato, até temos algumas regras CSS sendo aplicadas, mas elas não estão dentro do componente `OlaMundo` e sim dentro do componente `App`. Vamos analisar rapidamente as regras definidas por padrão no componente criado pelo Expo:

```
const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#fff',
    alignItems: 'center',
    justifyContent: 'center',
  },
});
```

Se você já trabalhou com o Flexbox, muito desse código deve ser familiar, como o `alignItems` e o `justifyContent`. Já para quem não teve contato com o Flexbox, talvez tenha pelo menos reconhecido a propriedade `backgroundColor`. Mas tendo reconhecido algo ou não, queremos que você veja que as regras estão sendo definidas dentro de um objeto literal JavaScript. Este é um padrão bastante utilizado para definir as regras de estilo de um componente dentro do React (web e Native). Utilizaremos esse padrão para criar nossas regras.

Nossa missão agora é deixar o componente `OlaMundo` com uma cara mais agradável e apresentável aplicando alguns estilos por meio do CSS. Ao trabalhar com front-end no desenvolvimento web, aprendemos que é uma boa prática separar as folhas de estilo do código da página, ou seja, separar os arquivos de HTML, CSS e

JavaScript. Fazemos isso para fins de organização e para facilitar futuras manutenções, afinal, assim não misturamos as coisas e o código se torna muito mais independente (imagina só em um sistema legado ter que alterar o estilo inline de milhares de componentes; ou então ter que replicar o código JavaScript para cada uma das iterações nos elementos da tela).

Essa prática se tornou muito forte no desenvolvimento web durante muitos anos (e a ainda é, dependendo das tecnologias que você adota), entretanto, quando começamos a falar de componentes, este discurso é diferente. Como estamos lidando com pequenas unidades da página (e não mais sua totalidade), não faz mais sentido tratar sua estrutura, estilo e interatividade como entidades globais e isoladas, mas sim, como uma coisa só. Isso significa que agora temos o HTML, CSS e JS em um único arquivo (nosso adorado JSX).

Para muitos, isso pode parecer esquisito ou mesmo antipadrão de boas práticas. Mas mantenha-se atento, que prometemos que no final do capítulo essa nova estrutura fará pleno sentido. Para que possamos digerir essa transformação da maneira mais gradual possível, trataremos primeiro de como criar estilos compartilhados entre nossos componentes e, em um segundo momento, começaremos a isolá-los dentro do escopo dos componentes.

5.2 UTILIZANDO ARQUIVO EXTERNO

Para começar, criaremos uma pasta dentro do nosso projeto e a nomearemos como `estilos`. Esta pasta será responsável por conter todos os estilos compartilhados entre os nossos componentes. Criada esta pasta, criaremos nela o arquivo

chamado `Estilos.js` . É nele que vamos colocar os estilos do nosso componente (e sim, este arquivo será um arquivo JavaScript).

Para conseguir usar estilos CSS dentro dos arquivos JSX, importaremos o módulo `StyleSheet` do pacote `react-native` . Para isso, basta utilizar o `import` da mesma maneira que fizemos antes:

```
import { StyleSheet } from 'react-native';
```

Esse módulo é bem bacana pois nos traz algumas facilidades. A mais relevante delas é que ele consegue validar as regras CSS quando aplicadas aos componentes. Ou seja, caso alguma regra tenha sido escrita de forma incorreta (como `backgroundcolor` em vez de `backgroundColor`), o erro/warning será apontado de forma transparente no console do Expo. Você notará que é muito fácil cometermos um erro de digitação ao inserir regras de estilo usando a técnica de CSS-in-JS, ainda mais com as particularidades do React Native. Tendo esse módulo em ação, ganharemos um bom tempo no desenvolvimento.

Feita a importação do módulo, exportaremos a criação de uma folha de estilos. Este processo é importante para que o restante do projeto consiga enxergar, acessar e usar os estilos que estamos criando para os componentes. Da mesma maneira que usamos uma única regra do CSS em múltiplos elementos HTML, conseguiremos fazer aqui, e essa é exatamente a ideia neste primeiro passo. Para exportar o CSS, usamos o método `create` do objeto `StyleSheet` . Dentro da chamada desta função, passamos um objeto literal JavaScript vazio (já veremos para que ele serve). Acompanhe o código a seguir:


```
export default StyleSheet.create({  
  })
```

É neste objeto literal vazio que estamos passando como parâmetro na função `create` que criaremos os estilos. A ideia é muito parecida com o que já estamos acostumados no desenvolvimento web, porém, com algumas pequenas diferenças importantes. Uma delas é a forma de nomear as regras de estilo. Por exemplo, para alterarmos o tamanho de uma fonte em CSS, usamos o atributo `font-size`. Na folha de estilos do React Native usamos o `fontSize`, tudo junto escrito no formato *Camel Case* — prática de escrever as palavras compostas ou frases, onde cada palavra é iniciada com maiúsculas e unidas sem espaços (com exceção da primeira). Colocaremos somente o número do tamanho que queremos que a fonte assuma, no caso `18` (não se preocupe, veremos o porquê dessas regras logo em seguida).

Voltando à função `StyleSheet` criada, colocaremos dentro dela um nome de atributo para chamarmos na tag. Isso facilitará bastante a organização do nosso código CSS no JSX. Como estamos lidando com um componente de texto, daremos o nome de `text`.

```
text: {  
}
```

Dentro do objeto, vamos inserir os estilos da maneira como descrevemos anteriormente. Primeiramente, queremos aumentar o tamanho da fonte para a unidade `18`. Para isso, colocaremos:

```
text: {  
  fontSize: 18,  
}
```

Feito isso, vamos fazer mais algumas alterações para ver o efeito do CSS no componente. Primeiramente, aplicaremos o negrito (**bold**), colocaremos uma borda vermelha para destaque e manipularemos o `padding` entre a borda e o próprio texto. Se você já tem experiência no desenvolvimento de sites, já deve ter lembrado das propriedades `font-weight` , `border-width` , `border-color` ... E você não está errado. No entanto, lembre-se do que dissemos algumas linhas atrás: para aplicar estes estilos no React Native, precisamos unir os termos e aplicar o formato *Camel Case*. No fim, teremos um código muito semelhante a este:

```
import { StyleSheet } from 'react-native';

export default StyleSheet.create({
  text: {
    fontSize: 18,
    fontWeight: 'bold',
    borderWidth: 2,
    borderColor: 'red',
    padding: 10,
  }
});
```

Agora que temos a nossa página de estilos, vamos importá-la para o nosso componente por meio do seu caminho relativo:

```
import Estilos from '../estilos/Estilos';
```

Por padrão, todos os componentes criados dentro do React Native aceitam uma propriedade (props) chamada `style` (assim como as tags HTML). Como já é de se imaginar, esta propriedade espera por um objeto JavaScript que descreve o estilo do componente (utilizando a estrutura que usamos).

Como a importamos para dentro do componente, basta conectar as duas pontas. Dentro do componente `Text` do

OlaMundo , vamos inserir o estilo criado com o atributo `text` no objeto e colocá-lo entre chaves dentro da propriedade `style` . Como no exemplo a seguir:

```
<Text style={Estilos.text}>{props.nome}</Text>
```

Repare que estamos pegando o objeto `StyleSheet` e então buscando somente o que atribuímos ao `text` . Muito semelhante ao que fazemos logo em seguida com o objeto `props` e a propriedade `nome` .

Ao salvar tudo e fazer o build da aplicação, veremos que o estilo que colocamos foi aplicado corretamente, conforme imagem:



Olá
Jonas



Figura 5.1: Imagem do dispositivo com a frase com os estilos aplicados

Muito legal, né? Mas essa é somente uma das maneiras de usarmos estilos CSS no React Native, geralmente a mais usada para compartilhar regras de estilos comuns a toda a aplicação. E antes de seguirmos, vamos parar um momento para validar que você entendeu tudo. Vamos propor um desafio: o que precisamos fazer para que a borda vermelha fique em torno de todo o texto e não só do segundo? Assim que você descobrir a resposta, continue a leitura. Caso encontre problemas, recomendamos a página da

documentação oficial (<https://reactjs.org/docs/dom-elements.html#style>).

5.3 ESTILOS INTERNOS AO COMPONENTE

Como citamos lá no início do capítulo, quando começamos a pensar em componentes, a estratégia de deixar os arquivos JavaScript, CSS e HTML separados começa a perder o sentido, afinal, um componente é uma estrutura isolada e independente que tem o propósito de ser utilizado em múltiplos lugares no código. Pensando nisso, como podemos aplicar o que fizemos no tópico anterior mas somente dentro do escopo do componente? Muito simples, praticamente já temos a resposta nas nossas mãos, basta alterarmos o jeito de usar.

Primeiramente, vamos voltar para o componente `OlaMundo`. Precisamos importar o módulo `StyleSheet` presente no `react-native` para que possamos criar os estilos. Da mesma forma, também usaremos o método `create` para definir os estilos. Vá até o final do componente que você quer estilizar e então coloque:

```
import Estilos from '../estilos/Estilos';

// [...] restante do código

const estilos = StyleSheet.create({
  text: {
    // Estilos
  },
});
```

Vamos rever o que acabamos de fazer. Em vez de exportar o objeto de estilos (como fizemos anteriormente), criamos uma variável `estilos` que tem a responsabilidade de armazenar estas

regras de estilo. A ideia é que usemos esta variável para administrar os estilos dentro das tags do componente, exatamente da mesma maneira que já fizemos. Tomando como base o mesmo componente que já alteramos, mude o estilo dele para utilizar estilo interno usando o seguinte código:

```
<Text style={estilos.text}>{props.nome}</Text>
```

Recarregue o código no seu aparelho e perceba que o resultado é exatamente o mesmo. A única diferença foi a maneira como lidamos com este estilo: no primeiro exemplo, criamos um arquivo externo que contém essa regra e então a exportamos para o nosso componente; no segundo exemplo, criamos a regra dentro do escopo do componente, deste modo, todas as alterações feitas são limitadas a ele.

5.4 CLASSES CSS

As técnicas de estilos que mostramos anteriormente são técnicas CSS-in-JS, ou seja, usamos o poder que o JSX e o React nos oferece para descrever as regras de estilo utilizando o JavaScript. No entanto, isso não significa que não podemos mais utilizar o formato tradicional de classes CSS nos nossos componentes. Caso faça sentido dentro do seu projeto (como um projeto legado, por exemplo), as duas técnicas podem ser utilizadas juntas. O que precisamos ter em mente é que a cascata CSS continua válida, ou seja, caso utilizemos regras definidas por classes que sejam conflitantes com o CSS inserido no componente, este segundo "vencerá".

Para usarmos classes CSS em nossos componentes, basta usarmos o atributo `className`. Este atributo funciona da mesma

forma que o `class` nas tags HTML. Podemos passar o nome da classe (ou classes) que desejamos usar e o componente vai buscar pelas regras declaradas na classe para aplicá-las em sua visualização. Podemos usar isso, por exemplo, para aplicarmos frameworks em nossos projetos, como o Bootstrap (<https://getbootstrap.com/>).

Por exemplo:

```
<Button to="/" className="btn btn-lg btn-success">Home</Button>
```

5.5 SEPARANDO ESTILOS GENÉRICOS - PADRÃO

O que é bastante comum em toda aplicação - tanto web quando mobile - é que geralmente ela segue um padrão de estilos (seja de cores, espaçamento, fontes etc.). Nesses casos, pode fazer sentido usarmos as duas estratégias que vimos neste capítulo simultaneamente, isolando os estilos que são genéricos na aplicação. Para isso, geralmente é criada uma pasta exclusiva que contém todas estas regras (lembra da nossa pasta `estilos`?). Vamos mostrar a seguir uma possibilidade de estrutura que é adotada em bastante projetos de código aberto.

Dentro dessa pasta especial, podemos organizar os arquivos da seguinte maneira:

- `index.js` : importa os arquivos de estilo e os exporta de forma com que você consiga usar um ou vários arquivos.

```
import cores from './cores';  
import fontes from './fontes';  
import metricas from './metricas';  
import geral from './geral';
```

```
export { cores, fontes, metricas, geral };
```

- `cores.js` : responsável por armazenar as cores utilizadas na aplicação, e aqui vão desde cores para layouts como cores de um componente `TextInput` , textos em geral, botões etc.

```
const cores = {  
  header: '#333333',  
  primario: '#069',  
};
```

```
export default cores;
```

- `fontes.js` : aqui é onde armazenamos os tamanhos de fontes utilizadas no projeto, então, assim como no `cores` , todos os tamanhos de fontes devem possuir algum significado e por isso estão nesse arquivo.

```
const cores = {  
  input: 16,  
  regular: 14,  
  medium: 12,  
  small: 11,  
  tiny: 10,  
};
```

```
export default fontes;
```

- `metricas.js` : margens, paddings, tamanhos configurados pela plataforma (ex.: `StatusBar` , `Border Radius` etc.). Tudo que está ligado diretamente com espaçamento e ocupação de um componente em tela vai nesse arquivo.

```
import { Dimensions, Platform } from 'react-native';
```

```
const { width, height } = Dimensions.get('window'); // Desestruturando
```


ramento (ES6)

```
const metricas = {
  smallMargin: 5,
  baseMargin: 10,
  doubleBaseMargin: 20,
  screenWidth: width < height ? width : height,
  screenHeight: width < height ? height : width,
  tabBarHeight: 54,
  navBarHeight: (Platform.OS === 'ios') ? 64 : 54,
  statusBarHeight: (Platform.OS === 'ios') ? 20 : 0,
  baseRadius: 3,
};
```

```
export default metricas;
```

- `geral.js` : o arquivo geral é o único diferente dos demais. Seu papel não é armazenar variáveis, mas sim armazenar estilos de componentes padrão. Pense que em seu aplicativo você possui um layout de seção que aplica um espaçamento e possui um título em negrito.

```
import metricas from './metricas';
import cores from './cores';
import fontes from './fontes';

const geral = {
  container: {
    flex: 1,
    backgroundColor: cores.background,
  },
  section: {
    margin: metricas.doubleBaseMargin,
  },
  sectionTitle: {
    color: cores.text,
    fontWeight: 'bold',
    fontSize: fontes.regular,
    alignSelf: 'center',
    marginBottom: metricas.doubleBaseMargin,
  },
};
```

```
export default geral;
```

Com isso, em vez de criar um componente chamado `Section`, apenas importamos os estilos do `geral.js` no arquivo de estilos para usar as propriedades `section` e `sectionTitle`. A importação pode ser realizada da seguinte forma:

```
import { StyleSheet } from 'react-native';
import { general } from 'styles';

const styles = StyleSheet.create({
  ...general,
});

export default styles;
```

A partir desse momento, poderemos utilizar todas as propriedades definidas no arquivo `geral.js`.

Conclusão

Neste capítulo vimos como utilizar o CSS para estilizar nossos componentes. Nas próximas páginas aprenderemos como organizar os componentes na tela para que funcionem de forma bacana nos mais diversos tamanhos de telas disponíveis no mercado: seja um iPhone, iPad, Samsung, Xiaomi... Não importa. O conteúdo será organizado proporcionalmente à área disponível. Para concluir este objetivo, veremos como trabalhar com uma especificação muito importante do CSS, o Flexbox.

O BÁSICO DE LAYOUTS COM O FLEXBOX

Assim como usamos o CSS para organizar os elementos do nosso site na tela de forma que atenda a qualquer tipo de tamanho de mídia (quem nunca usou o bom e velho Bootstrap para isso, não é mesmo?), quando utilizamos o React Native, a história não é muito diferente. Claro, a ideia não é usar o Bootstrap no projeto, mas sim, uma especificação que surgiu há alguns anos no CSS para resolver o problema de conteúdo responsivo e quem tem conseguido realizar esta missão com sucesso: estamos falando do Flexbox.

Até o final deste capítulo aprenderemos como utilizar o Flexbox para adaptar os componentes dos aplicativos para os mais diversos tamanhos de tela disponíveis no mercado (e não são poucos!).

6.1 ALTURA E LARGURA (HEIGHT E WIDTH)

Antes de entrar de cabeça nos detalhes do funcionamento do Flexbox, vamos experimentar a forma mais simples de definir o tamanho de um elemento no React Native. Esta forma nada mais é do que simplesmente utilizar as propriedades `height` e `width`

para definir a altura e comprimento, respectivamente. Ambas devem ser inseridas dentro do atributo `style`, como já vimos no capítulo anterior.

Mas algo curioso sobre essa medida é que no React Native as dimensões são `unitless`, ou seja, sem unidade. Isso significa que, na prática, eles representam pixels independentes da densidade. Em termos mais simples, podemos dizer que o seu tamanho sempre será o mesmo independente da dimensão da tela do usuário. Parece estranho? Realmente só o teórico é bem confuso, talvez um exemplo prático nos ajude a entender isso melhor.

Então para validar tudo isso que estamos dizendo, vamos fazer um pequeno experimento. No projeto em que estamos trabalhando, crie um novo componente chamado `DimensoesFixas` (lembrando que para isso precisamos criar um novo arquivo JavaScript no diretório `componentes`). Feito isso, criaremos algumas figuras usando o componente nativo `View` do React Native e compararemos seus tamanhos:

```
import React from 'react';
import { View } from 'react-native';

class DimensoesFixas extends React.Component {
  render() {
    return (
      <View>
        <View style={{width: 50, height: 50, backgroundColor: 'powderblue'}} />
        <View style={{width: 100, height: 100, backgroundColor: 'skyblue'}} />
        <View style={{width: 150, height: 150, backgroundColor: 'steelblue'}} />
      </View>
    );
  }
};
```

```
export default DimensoesFixas;
```

NOTA: desta vez colocamos os estilos dentro da própria tag `View`. Esta prática é chamada de *CSS inline* e é usada somente para colocar estilo em uma tag específica. O `style` é o atributo usado para formatar a tag. Quando mais de uma tag for usar o mesmo estilo, essa prática não é recomendada, pois torna a manutenção muito mais trabalhosa.

Importe este componente dentro do `App` e visualize o seu resultado. Deverá ser muito semelhante ao indicado na figura:

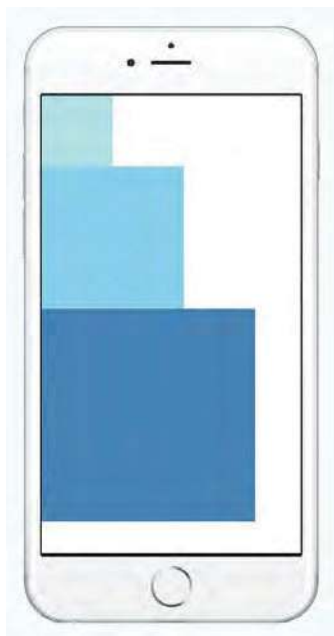


Figura 6.1: Dimensões dos componentes gerenciadas pelas propriedades

Muito fácil, certo? Se possível, experimente abrir esta mesma tela em aparelhos de tamanhos diferentes. Você notará que a proporção do tamanho do componente nunca muda. Mas e se quisermos que ele mude? E se quisermos que ele se adapte à tela? Afinal, estamos tratando de aplicativos mobile e existe um mar de tamanhos de tela que devem ser contemplados. Para este caso, precisamos dar uma olhada no Flexbox.

6.2 CONTÊINERES E ELEMENTOS FLEX

Flexbox é uma especificação CSS3 com a finalidade de organizar os elementos de uma página dentro de contêineres (caixas), mantendo o layout flexível independente das suas dimensões e reorganizando-se de acordo com a necessidade. Como o design dos aplicativos em React Native é feito com Flexbox, precisamos conhecer os seus conceitos básicos para conseguir fazer um bom trabalho.

Primeiramente, para determinar que um elemento é *flex* (aqui entenda como um elemento que respeita as regras definidas pela especificação), não precisamos baixar nenhuma biblioteca ou algo do tipo. Muito pelo contrário. Para começarmos a usar o Flexbox, temos de marcar o elemento com a propriedade de estilo `display: flex`. Com isso, estamos definindo que esse elemento é um *flex-container* e todos os elementos filhos serão *flex-items*. (No contexto web, o funcionamento dependerá do navegador utilizado; por sorte, os grandes navegadores da atualidade já dão suporte a isso).

E o que isso quer dizer? Isso representa para a aplicação que dentro de um elemento *container* todos os elementos filhos vão se

organizar com base no tamanho disponível nele. Se o contêiner ficar maior, os filhos contidos possuem mais espaço; se ficar menor, os filhos se organizam de forma diferente. É a mesma lógica aplicada ao sistema de grid do Bootstrap.

Agora, para entender como itens são distribuídos no `flex-container`, devemos aprender o funcionamento dos eixos no Flexbox. Todo elemento *flex* dentro de um contêiner é sempre orientado por meio de dois eixos: a *main-axis* (eixo principal) e a *cross-axis* (eixo transversal). Por padrão, o eixo principal é orientado horizontalmente; enquanto o eixo secundário, verticalmente.

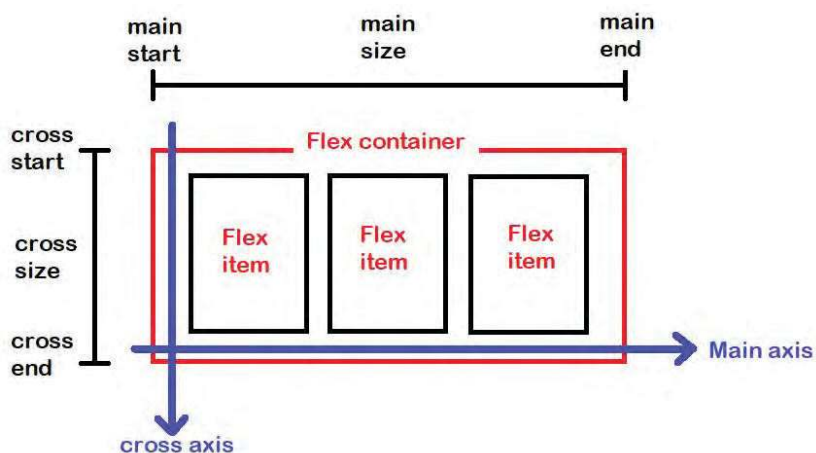


Figura 6.2: Conceitos flexbox

Note que o *main-axis* assim como o *cross-axis*, contém um tamanho, início e fim. Tudo isso para ajudar no alinhamento dos itens dentro do *flex-container*. Todas as orientações dos componentes são feitas com base nestes eixos. Vamos entendê-los melhor.

6.3 FLEX DIRECTION

Quando pensamos no cenário prático de uso dos eixos, os `flex-items` (itens contidos no contêiner) sempre tentarão se organizar inicialmente pelo eixo principal, para só depois optar pelo eixo transversal. No entanto, isso não necessariamente indica que os elementos vão se organizar primeiro na horizontal e depois na vertical, essa ordem pode ser alterada. Tudo depende do valor atribuído à propriedade `flex-direction`.

Se o `flex-direction` receber o valor `row` (linha) ou `row-reverse` (linha reversa), para organização em linhas, o `main-axis` do `_contêiner` será horizontal. Mas se o `flex-direction` receber o valor `column` ou `column-reverse`, para organização em colunas, o `main axis` será vertical. Consequentemente, se definirmos qual é o `main-axis`, o outro será a `cross-axis`.

Para ver toda essa teoria funcionando, volte ao componente `DimensoesFixas` e, dentro no componente `<View>` que contém os demais, insira a definição do tamanho (`height`) e da largura (`width`) para 100%, ou seja, queremos que ele ocupe o tamanho inteiro disponível. Depois insira a propriedade `flexDirection: 'row'` (lembrando que no React Native as propriedades que são divididas por um hífen no CSS da web viram *camelCase* no React Native). E por fim igualar as propriedades `width` e `height` para 50 de todos os filhos.

```
import React from 'react';
import { View } from 'react-native';

class DimensoesFixas extends React.Component {
  render() {
    return (
      <View style={{width: '100%', height: '100%', flexDirection:
```



```

    'row'>>
      <View style={{width: 50, height: 50, backgroundColor: 'powderblue'}} />
      <View style={{width: 50, height: 50, backgroundColor: 'skyblue'}} />
      <View style={{width: 50, height: 50, backgroundColor: 'steelblue'}} />
    </View>
  );
}
};

export default DimensoesFixas;

```

Visualize o resultado, você terá algo parecido com a imagem:



Figura 6.3: Flex Direction aplicado aos nossos componentes

Note a diferença. Por padrão, os componentes são empilhados um em cima do outro, mas a partir do momento em que alteramos

o eixo de orientação usando a propriedade `flexDirection` isso mudou. Podemos voltar ao comportamento anterior alterando a propriedade para o valor `column`. Isso alterará os eixos e os três componentes novamente ficarão empilhados uns nos outros.

6.4 JUSTIFY CONTENT

Para determinar a distribuição dos `flex-items` ao eixo principal, usamos a propriedade `justify-content`. A distribuição pode ser feita de cinco maneiras diferentes:

1. `justify-content: flex-start` : alinha os itens ao início do contêiner;
2. `justify-content: center` : alinha os itens ao centro do contêiner;
3. `justify-content: flex-end` : alinha os itens ao final do contêiner;
4. `justify-content: space-between` : cria um espaçamento igual entre os elementos, mantendo o primeiro grudado no início e o último no final;
5. `justify-content: space-around` : cria um espaçamento entre os elementos. Os espaçamentos do meio são duas vezes maiores que o inicial e final.

No componente `DimensoesFixas` aplicaremos a propriedade `flexDirection: 'column'` e no componente `View`, `justifyContent: 'space-between'`, para testar o resultado.

```
import React from 'react';
import { View } from 'react-native';

class DimensoesFixas extends React.Component {
  render() {
```

```

    return (
      <View style={{width: '100%', height: '100%',
        flexDirection: 'column', justifyContent: 'space-between'}}
    >
      <View style={{width: 50, height: 50, backgroundColor: 'powderblue'}} />
      <View style={{width: 50, height: 50, backgroundColor: 'skyblue'}} />
      <View style={{width: 50, height: 50, backgroundColor: 'steelblue'}} />
    </View>
  );
}
};

```

export default DimensoesFixas;

Rode o código. Teremos algo parecido com a imagem:

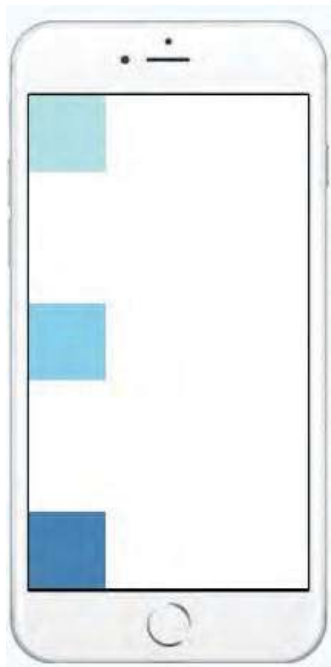


Figura 6.4: Justify Content

Desta vez, os itens estão distribuídos igualmente na direção indicada pela `main-axis`. Teste as outras propriedades do `justify-content` citadas e veja o comportamento na tela.

6.5 ALIGN ITEMS

Para determinar o alinhamento dos `flex-items` ao longo do eixo secundário, usamos a propriedade `align-items`. As possibilidades de alinhamento são muito semelhantes às do item anterior, sendo:

1. `align-items: stretch`: valor padrão, ele que faz com que os flex itens cresçam igualmente;
2. `align-items: flex-start`: alinha os itens ao início;
3. `align-items: flex-end`: alinha os itens ao final;
4. `align-items: center`: alinha os itens ao centro;
5. `align-items: baseline`: alinha os itens de acordo com a linha base da tipografia.

No componente `DimensoesFixas`, vamos aplicar a propriedade `flexDirection: 'column'`, `justifyContent: 'center'` e `alignItems: 'stretch'` para entender essas regras na prática.

```
import React from 'react';
import { View } from 'react-native';

class DimensoesFixas extends React.Component {
  render() {
    return (
      <View style={{width: '100%', height: '100%',
        flexDirection: 'column', justifyContent: 'center', alignI
tems: 'stretch'}}>
        <View style={{width: 50, height: 50, backgroundColor: 'po
wderblue'}} />
      </View>
    );
  }
}
```

```

        <View style={{width: 50, height: 50, backgroundColor: 'skyblue'}} />
        <View style={{width: 50, height: 50, backgroundColor: 'steelblue'}} />
      </View>
    );
  }
};

export default DimensoesFixas;

```

Veja que teremos algo parecido com a imagem a seguir.

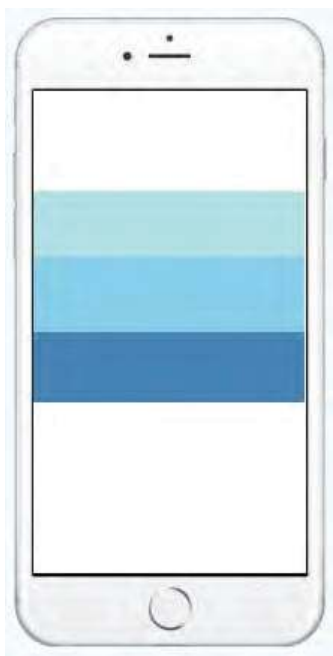


Figura 6.5: Align Items

Assim como fizemos com os itens anteriores, teste todas as propriedades do `align-items` citadas a fim de ver na prática o seu comportamento na tela.

6.6 FLEX-WRAP

Esta é uma propriedade importantíssima que define se os itens devem quebrar ou não a linha. Por padrão eles não a quebram, isso faz com que os `flex-items` sejam compactados além do limite do conteúdo. E quando estamos falando de linha, pode ser tanto a horizontal quanto a vertical, dependendo do valor configurado na propriedade `flex-direction`.

A propriedade pode assumir três valores diferentes:

1. `flex-wrap: nowrap` : valor padrão, não permite a quebra de linha;
2. `flex-wrap: wrap` : quebra a linha assim que um dos `flex` itens não puder mais ser compactado;
3. `flex-wrap: wrap-reverse` : quebra a linha assim que um dos `flex` itens não puder mais ser compactado. A quebra é na direção contrária, ou seja, para a linha acima.

Essa é geralmente uma propriedade que é quase sempre definida como `flex-wrap: wrap`, pois quando um dos `flex` itens atinge o limite do conteúdo, o último item passa para a coluna debaixo e assim por diante. Vamos ver como ela funciona.

Volte ao componente `DimensoesFixas` e altere o código da seguinte maneira:

```
import React from 'react';
import { View } from 'react-native';

class DimensoesFixas extends React.Component {
  render() {
    return (
      <View style={{width: '100%', height: '100%',
        flexDirection: 'column', flexWrap: 'nowrap'}}>
```

```

        <View style={{width: 50, height: 100, backgroundColor: 'p
owderblue'}} />
        <View style={{width: 50, height: 100, backgroundColor: 's
kyblue'}} />
        <View style={{width: 50, height: 100, backgroundColor: 's
teelblue'}} />
        <View style={{width: 50, height: 100, backgroundColor: 'p
owderblue'}} />
        <View style={{width: 50, height: 100, backgroundColor: 's
kyblue'}} />
        <View style={{width: 50, height: 100, backgroundColor: 's
teelblue'}} />
        <View style={{width: 50, height: 100, backgroundColor: 'p
owderblue'}} />
        <View style={{width: 50, height: 100, backgroundColor: 's
kyblue'}} />
        <View style={{width: 50, height: 100, backgroundColor: 's
teelblue'}} />
      </View>
    );
  }
};

export default DimensoesFixas;

```

Teste este componente no seu aparelho. Você notará que o tamanho do `flex-container` é insuficiente, no entanto, os itens não quebram a linha (conforme mostra a imagem). Isso torna o nosso aplicativo bastante problemático, já que o nosso conteúdo é comprometido.



Figura 6.6: Conteúdo ultrapassando o espaço do flex-container

Para resolver isso, altere a propriedade `flexWrap` de `nowrap` para `wrap`. Ao fazer isso, o flex-container vai entender que, quando seus itens ultrapassarem suas limitações, ele deverá reorganizar os flex-itens restantes para ocupar o espaço ao lado. O mesmo efeito é obtido tanto no `column` quanto para `row`.

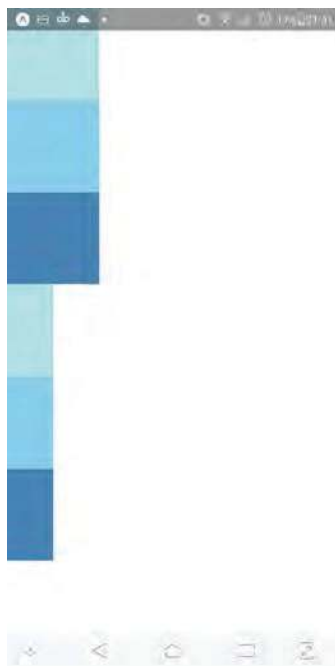


Figura 6.7: Conteúdo sendo reorganizado graças a propriedade flexWrap

6.7 FLEX-GROW

As propriedades que vimos até agora dizem respeito a como os `flex-items` devem ser alocados dentro do `flex-container`, no entanto, existem algumas regras que são específicas aos próprios `flex-items`. Uma destas propriedades é o `flex-grow`. Como estamos dizendo que o Flexbox lida com a variedade do tamanho de telas com flexibilização dos seus itens, nada mais justo do que termos uma propriedade para indicar quanto e como estes itens devem aumentar/diminuir. A habilidade de um `flex-item` de crescer é justamente o que o `flex-grow` faz.

A atribuição desta propriedade é muito simples, basta seguir a

sintaxe: `flex-grow: <número>` . Por definição, o seu valor é zero, assim os `flex-items` ocupam um tamanho máximo relacionado o conteúdo interno deles ou ao `width` definido. Ao definir o valor 1 para todos os `flex-items` , eles tentarão ter a mesma largura e vão ocupar 100% do `flex-container` . Estes valores são proporcionais dentro do `flex-container` . Ou seja, se tivermos uma linha com três itens, onde dois são `flex-grow: 1` e um `flex-grow: 2` , o segundo tentará ocupar 2 vezes mais espaço do que os outros elementos. Em outras palavras, ele crescerá duas vezes mais rápido.

Vamos validar essas regras com o nosso componente. Para isso, vamos remover o tamanho fixo e inserir a propriedade `flex-grow` . Em dois componentes `View` utilizaremos com o valor 1 e no outro com o valor 2.

```
import React from 'react';
import { View } from 'react-native';

class DimensoesFixas extends React.Component {
  render() {
    return (
      <View style={{width: '100%', height: '100%',
        flexDirection: 'column', justifyContent: 'center', alignI
items: 'stretch'}}>
        <View style={{flexGrow: 1, backgroundColor: 'powderblue'}}
      />
        <View style={{flexGrow: 2, backgroundColor: 'skyblue'}} />
        <View style={{flexGrow: 1, backgroundColor: 'steelblue'}}
      />
      </View>
    );
  }
};

export default DimensoesFixas;
```

Acesse novamente o seu aplicativo e repare no resultado. Deverá ser semelhante a isto:

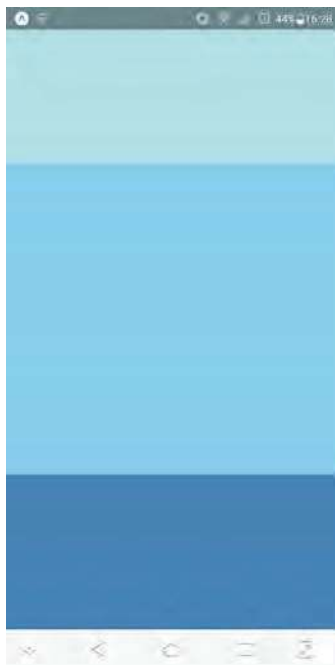


Figura 6.8: Utilização da propriedade flexGrow

A segunda `View` ocupará mais espaço que as demais. Experimente mudar este valor para três, quatro e assim por diante. O tamanho das `View` sempre será proporcional aos valores escolhidos em suas respectivas propriedades `flexGrow`.

6.8 FLEX-SHRINK

Assim como podemos definir a capacidade de crescimento de um `flex-item`, também podemos definir sua capacidade de

redução. Sua sintaxe também é bem simples, sendo: `flex-shrink: <número>`. Por definição, os componentes têm o valor 1, o que permite que seus tamanhos sejam reduzidos para caber dentro do `flex-container`. Se usarmos o valor 0, indicamos ao Flexbox que este item é inflexível, ou seja, ele nunca ficará menor do que indicado em sua `flex-basis` (que veremos a seguir). Ao colocar qualquer outro valor, ele funcionará da mesma forma relativa que o `flex-grow`: um `flex-item` com `flex-shrink: 3` diminuirá 3 vezes mais que um item com 1.

Vamos fazer este teste no nosso componente.

```
import React from 'react';
import { View } from 'react-native';

class DimensoesFixas extends React.Component {
  render() {
    return (
      <View style={{width: '100%', height: '100%',
        flexDirection: 'column', justifyContent: 'center', alignI
tems: 'stretch'}}>
        <View style={{flexShrink: 1, flexBasis: 300, backgroundCo
lor: 'powderblue'}} />
        <View style={{flexShrink: 3, flexBasis: 300, backgroundCo
lor: 'skyblue'}} />
        <View style={{flexShrink: 1, flexBasis: 300, backgroundCo
lor: 'steelblue'}} />
      </View>
    );
  }
};

export default DimensoesFixas;
```

Acesse novamente o seu aplicativo e repare no resultado. Será algo semelhante a isto:



Figura 6.9: Utilização da propriedade `flexShrink`

Ele será inversamente proporcional ao que experimentamos no código anterior. Aqui, a segunda `View` diminui duas vezes mais rápido que as outras, tornando-a sempre menor (a não ser que o `flex-container` tenha espaço suficiente para todos, neste caso todas permanecerão com o mesmo tamanho.)

6.9 FLEX-BASIS

Esta propriedade indica o tamanho inicial do `flex-item` antes da distribuição do espaço restante. O seu valor padrão é `auto`, o que significa que ele faz com que a largura da base seja igual à do item. Se o item não tiver tamanho especificado, o

tamanho será de acordo com o conteúdo. Quando definimos o `flex-grow: 1` e possuímos auto definido, o valor restante para ocupar o contêiner é distribuído ao redor do conteúdo do `flex-item`.

Como citado, no caso do `flex-shrink` quando configurado com o valor 0, o `flex-basis` é quem definirá o tamanho base deste `flex-item`, de modo que ele nunca será menor, mesmo que o contêiner seja menor.

Conclusão

Neste capítulo, aprendemos o que é a tão famosa especificação Flexbox, quais os seus conceitos básicos e como utilizá-la junto com o React Native para a criação de layouts altamente flexíveis. A especificação é bem extensa e existem muitos detalhes que valem a pena ser estudados, mas selecionamos os imprescindíveis para o bom desenvolvimento profissional de um aplicativo. Ao longo do livro, vamos aplicar estas regras para organizar o conteúdo da melhor maneira possível na tela.

Para dar continuidade aos nossos estudos, falaremos de outro aspecto muito importante do React Native: a renderização condicional. Veremos como criar condições para que o nosso aplicativo saiba o que renderizar de acordo com algum evento, condição e/ou ação do usuário.

RENDERIZAÇÃO CONDICIONAL

Quando estamos desenvolvendo uma aplicação web, é comum precisarmos mostrar ou esconder um determinado elemento na tela dada uma certa condição, por exemplo, a interação do usuário com um botão. Uma situação prática bem simples é o detalhamento de um produto em um e-commerce. Imagine que criamos uma página cheia de produtos e que para um, em específico, existe uma descrição mais detalhada. Porém, não queremos que esses detalhes sejam exibidos logo de cara, caso contrário, a interface do nosso programa ficaria uma verdadeira zona. Então, para resolver isso, manipulamos o DOM (*Document Object Model*) por meio do JavaScript para que esse detalhe só apareça caso o item seja clicado (evento de `click`). Isso é bem comum, certo? Temos certeza de que você aí do outro lado sem dúvidas já pensou em uma série de outras situações onde manipulamos a página para que certo conteúdo só apareça de acordo com uma condição - sendo ela um evento causado pelo usuário, uma informação carregada via AJAX, enfim, os exemplos são inúmeros.

Quando estamos trabalhando com o React, esse processo não é muito diferente. Podemos programar nossos componentes para

que eles renderizem conteúdos diferentes de acordo com uma dada condição. Damos a isso o nome de *renderização condicional* e neste capítulo vamos aprender algumas técnicas para aplicá-la.

7.1 VERIFICANDO SE O NÚMERO É PAR OU ÍMPAR

Uma das melhores maneiras de se aprender código é por meio da prática e de exemplos. Para entender os possíveis casos de utilização da renderização no React Native - e note que o que aprendemos neste capítulo também será útil na utilização do React para web - vamos trabalhar com um pequeno exercício que representa algumas das situações onde a técnica de renderização condicional vem a calhar. Neste exercício que vamos propor, queremos o seguinte: vamos passar um número inteiro positivo para um componente e ele vai nos dizer se ele é ímpar ou par. Caso seja par, queremos que ele renderize uma mensagem do tipo: "este número é par" (e vice-versa).

Para começar, dentro da pasta de componentes criaremos o componente de nome `ChecaNumero.js`. Neste arquivo criaremos um componente funcional (você se lembra de qual é a diferença?). Começaremos importando os elementos necessários do React e do React Native necessários (a "velha" receita de bolo para nossos componentes). Aproveitaremos para trazer os componentes nativos de `View`, `Text` e o `StyleSheet`:

```
import React from 'react';
import { View, Text, StyleSheet } from 'react-native';
```

Agora, dentro do componente `View` colocaremos a condição de renderização. Para fazer isso, utilizaremos uma funcionalidade

do JavaScript bastante útil, chamada **operador condicional ternário**

(https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Operators/Operator_Condicional). Apesar do nome confuso e misterioso, seu funcionamento é na verdade bem simples. Esse operador nos permite reduzir uma condição do if-else em apenas uma linha, o que por muitas vezes torna o processo de codificação mais prático e limpo também.

Se você está se sentindo inseguro em relação a este conceito, veja só como é simples: no nosso programa queremos exibir uma mensagem diferente caso o número seja par ou ímpar. Para saber se um número é par ou ímpar, basta que saibamos o resto da sua divisão por 2. Se o resto for igual a zero, significa que o número foi dividido perfeitamente por dois, logo é par. O contrário indica que ele é ímpar. Simples, certo? Pensando na estrutura tradicional de if-else, poderíamos ter algo assim:

```
if(numero % 2 === 0) {  
    return <Text>0 número é par!</Text>  
} else {  
    return <Text>0 número é ímpar!</Text>  
}
```

Com o operador ternário podemos facilmente alcançar isso fazendo o seguinte:

```
numero % 2 === 0 ? <Text>0 número é par!</Text> : <Text>0 número  
é ímpar!</Text>
```

E pronto! Indicamos qual é a condição e o que ele deve fazer caso seja verdadeira ou falsa. Repare que o operador condicional ternário tem uma sintaxe bem simples:

```
condicao ? expr1 : expr2
```

A condição no nosso caso foi: `numero % 2 === 0`. O `expr1`

logo ao lado do símbolo de interrogação indica o que o nosso código deve fazer caso a condição seja verdadeira. Fizemos com que ele "devolva" o componente de texto com a mensagem "o número é par". Os dois pontos (:) funcionam como o `else`. Traduzindo, se a condição não for verdadeira, execute o que está definido em `expr2`.

Para finalizar, basta pensarmos em como incorporar este código a um outro componente, como o `App.js`. Nesta situação, precisamos pensar que o número associado virá como uma propriedade. Sendo uma propriedade, sabemos que ela é acessível através do objeto `props`. Vamos batizar a propriedade que recebe o número como `numero`. Com isso, temos tudo o que precisamos para dar vida ao componente:

```
import React from 'react';
import { View, Text, StyleSheet } from 'react-native';

export default props =>
  <View>
    {
      props.numero % 2 == 0
      ? <Text>0 número é par!</Text>
      : <Text>0 número é ímpar!</Text>
    }
  </View>
```

Para finalizar, vamos usar o `StyleSheet` e sua função `create` para inserir algumas regras de estilo ao componente.

```
import React from 'react';
import { View, Text, StyleSheet } from 'react-native';

<View>
  {
    props.numero % 2 == 0
    ? <Text style={styles.text}>0 número é par!</Text>
    : <Text style={styles.text}>0 número é ímpar!</Text>
```

```

    }
  </View>

  const styles = StyleSheet.create({
    container: {
      flex: 1,
      backgroundColor: '#fff',
      alignItems: 'center',
      justifyContent: 'center',
    },
    text: {
      fontSize: 18,
      fontWeight: 'bold'
    }
  });

```

Agora vamos importar o nosso componente `ChecaNumero.js` para dentro do `App.js` e então chamá-lo dentro da função `render`.

```

import ChecNumero from './componentes/ChecaNumero';

export default class App extends React.Component {
  render() {
    return (
      <View style={styles.container}>
        <ChecaNumero numero={3}/>
      </View>
    );
  }
}

```

Rodando nossa aplicação, veja que foi renderizado o texto "o número é ímpar" na tela do aplicativo como esperado.

O número é ímpar!

Figura 7.1: Renderização do texto Ímpar

Troque o valor três para um número par e veja o texto Par sendo renderizado. Brinque com este código e garanta que você entendeu como fazer isso antes de continuar.

7.2 RENDERIZAÇÃO CONDICIONAL COM FUNÇÃO

Outra possibilidade de fazer uma renderização condicional é abstrai-la dentro de uma função à parte. Tomando como base o componente que criamos, vamos criar uma função chamada

`validaParOuImpar` dentro do componente `checaNumero`. No corpo dessa função, vamos transferir o código que valida se o número é par ou ímpar e devolve um trecho JSX. Feito isso, basta invocar a função entre as chaves para que a validação seja feita. O código ficará assim:

```
import React from 'react';
import { View, Text, StyleSheet } from 'react-native';

export default props =>
  <View style={styles.container}>
    {validaParOuImpar(props.numero)}
  </View>

function validaParOuImpar(numero) {
  return numero % 2 == 0
    ? <Text style={styles.text}>O número é par!</Text>
    : <Text style={styles.text}>O número é ímpar!</Text>
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#fff',
    alignItems: 'center',
    justifyContent: 'center',
  },
  text: {
    fontSize: 18,
    fontWeight: 'bold'
  }
});
```

O resultado final no aplicativo deve ser exatamente o mesmo. Mas, no final, qual a vantagem que ganhamos com isso? A realidade é que desta maneira nós ganhamos organização. Ainda estamos dando nossos primeiros passos dentro da arte da componentização, mas com o uso e aplicação em situações do mercado de trabalho, você notará que mesmo os componentes

mais simples tendem a ter regras mais complexas dentro do seu corpo. Quando usamos `if-else` várias vezes de forma espalhada na função `render`, o código pode se tornar confuso, principalmente para a pessoa que não implementou o componente. Para evitar essa situação, costumamos encapsular uma pequena regra de funcionamento do componente dentro de uma função. Desta maneira ganhamos bastante legibilidade. Além disso, essa técnica de isolar parte do corpo do componente em uma função não é exclusiva para situações onde temos vários `if-else`, mas para qualquer critério que você achar relevante. Mais para a frente, por exemplo, teremos construção de listas, requisições AJAX... Essas e outras serão ótimas oportunidades para aplicar essa técnica.

Conclusão

Neste capítulo, aprendemos o que é a renderização condicional através de dois exercícios pequenos, porém bem práticos. No primeiro, aprendemos como usar o recurso de operador condicional ternário do JavaScript (e aqui vale um parênteses para dizer que este recurso não é exclusivo da linguagem, outras como o próprio Java também a implementam) para indicar ao React Native qual pedaço de código deve ser renderizado de acordo com uma dada condição. No exemplo posterior, vimos como usar funções para fazer um trabalho muito semelhante. A vantagem neste caso é que conseguimos abstrair a condição, de tal modo que, se ela for muito extensa, pode tornar o código mais difícil de ler. Por fim, entendemos que é até mesmo possível unir os dois recursos para criar componentes mais inteligentes.

STATE, EVENTOS E COMPONENTES CONTROLADOS E NÃO CONTROLADOS

Chegou a hora de aprendermos um dos conceitos mais importantes do React junto às propriedades: os estados. Nos capítulos anteriores citamos que entre os componentes funcionais e os de classe, apenas os de classe eram capazes de suportar essa funcionalidade chamada de estados. Mas, afinal de contas, o que eles são? O que eles significam? Será que realmente precisamos deles em nossos componentes ou é apenas uma ferramenta para nos ajudar a escrever componentes? Neste capítulo aprenderemos mais sobre o que são os estados, qual a sua relação com os eventos e como os usamos para criar os famosos "componentes controlados".

8.1 CONHECENDO OS ESTADOS

Os estados — chamados no React de `state` —, assim como as `props` que vimos anteriormente, nada mais são do que dados a

serem usados pelos componentes. Estes dados podem ser uma string, um número, um array ou mesmo um objeto. A única diferença entre os estados e as propriedades é que, no caso do segundo, ela é somente uma informação externa recebida (como um parâmetro de uma função) que deverá ser usada no corpo do componente (ou mesmo ser passada adiante); já os estados são dados privados e completamente controlados pelo próprio componente. Como o próprio nome já indica, ele representa um estado do componente, ou seja, um momento específico no tempo em que ele tem uma informação que poderá ser diferente no futuro.

Confuso? Talvez a definição teórica seja mais complicada do que sua utilização na prática, mas vamos tentar uma abordagem complementar. Imagine que os estados são como "fotos" dos nossos componentes: elas guardam informações sobre o atual momento no ciclo de vida dele. Isso significa que os dados dessa "foto" podem (e provavelmente vão) mudar no futuro, onde teremos um novo estado (ou seja, uma nova foto).

Então, lembre-se desta regrinha de ouro: sempre que um componente for armazenar dados que serão alterados dentro dele durante o seu ciclo de vida, os estados serão usados.

Mas como isso funciona na prática? A interação do usuário com os componentes é um ótimo exemplo de como os estados funcionam. Clicando nos botões, preenchendo formulários, selecionando caixas de seleção etc. Em todos esses eventos, se um usuário tiver que preencher um formulário com entradas de textos, cada campo do formulário manteria seu estado com base na entrada do usuário. Se a entrada do usuário for alterada, então os

estados das entradas de texto serão alterados, causando uma nova renderização do componente e de todos os seus componentes filhos. Isso é o que chamamos de componentes controlados.

Vamos fazer um exemplo prático para entender melhor. Para começar, vamos para o nosso diretório de componentes. Uma vez lá dentro, crie um novo arquivo chamado `Evento.js`. Ele será o nosso guia durante este capítulo. Como sempre, primeiro precisamos fazer as importações necessárias para este novo componente. Além do pacote `react` e do arquivo de estilos, importaremos também os componentes `View`, `Text` e `TextInput` do `react-native`;

```
import React from 'react';
import { View, Text, TextInput, StyleSheet } from 'react-native';
```

Agora partiremos para a parte inédita: vamos definir o nosso primeiro estado. Lembra-se de que dissemos que o estado nada mais é do que uma "foto" dos dados relevantes do componente? Pois bem. Para que os componentes React possam registrar e alterar estes dados com o tempo, precisamos definir qual é o seu estado atual, ou seja, qual o seu ponto de partida. Em outras palavras, temos que "inicializar" o estado do componente.

Como trabalharemos neste exemplo com um componente de `TextInput`, criaremos uma variável chamada `input` que armazenará qual foi o valor inserido nele. Para definirmos isso no estado do componente, basta declararmos a variável `state` e atribuir a ela um objeto literal que possui as informações com as quais o componente será inicializado em seu estado.

Sendo o valor inicial do `input` uma string vazia, vamos atribuí-la ao `state`:

```
import React from 'react';
import { View, Text, TextInput, StyleSheet } from 'react-native';

class Evento extends React.Component {
  state = {
    input: ''
  }
}

export default Evento;
```

Pronto, já temos o nosso primeiro estado definido. Bem direto e objetivo, certo? No entanto, essa informação do jeito que está ainda não é útil. Com o que fizemos, o componente somente tem um objeto `state` que possui o valor de string vazia.

Antes de avançarmos e entendermos como usar esse mecanismo de forma eficaz, vamos inserir alguns estilos no componente:

```
const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#fff',
    alignItems: 'center',
    justifyContent: 'center',
  },
  input: {
    height: 50,
    width: 300,
    fontSize: 30,
    borderWidth: 1,
    borderColor: 'black',
  },
  font30: {
    fontSize: 30
  }
});
```

Agora que já temos estilo, vamos trabalhar um pouco mais neste componente.

8.2 USANDO AS INFORMAÇÕES DOS ESTADOS

Para tornar a nossa declaração do estado um pouco mais eficaz, vamos alterar o componente `Evento`. Abaixo da definição do `state`, implementaremos a função `render`, aquela responsável por retornar qual será o corpo (JSX) do componente a ser mostrado na tela. Nesta função, vamos retornar um componente `Text` que conterá o valor atribuído ao dado `input` do `state`.

Para fazer isso, basta usarmos o `this` para que o componente consiga acessar o seu atributo `state`. Conseguindo fazer isso, poderemos usar seus dados da mesma maneira que fizemos com as `props`: basta acessá-lo por meio da notação de ponto (`.`) ou colchete (`[]`), conforme mostra o código adiante:

```
import React from 'react';
import { View, Text, TextInput, StyleSheet } from 'react-native';

class Evento extends React.Component {
  state = {
    input: ''
  }

  render() {
    return (
      <Text style={styles.font30}>{this.state.input}</Text>
    )
  }
}

export default Evento;
```

Vamos importar este componente para o `App.js` para testá-lo.

```
import React from 'react';
import { StyleSheet, Text, View } from 'react-native';
```

```
import Evento from './componentes/Evento';

export default function App() {
  return (
    <Evento />
  );
}
```

Ao tentar ver o resultado deste componente no aparelho veremos... Nada. E esperamos que você não se assuste com isso, afinal, estamos exibindo um texto cujo valor inicial é uma string vazia. Este resultado já era esperado (ufa!). Experimente agora alterar o valor do estado inicial do componente `Evento`. Coloque, por exemplo, o nome da tecnologia:

```
state = {
  input: 'O React Native é demais!'
}
```

Recompile o projeto e veja: o texto que inserimos no estado do componente é renderizado!



Figura 8.1: O estado do componente sendo utilizado como informação na tela

Bem legal, né? Mas como será que fazemos para que a informação do `state` seja atualizada durante o seu ciclo de vida? Veremos no próximo tópico.

8.3 ATUALIZANDO O ESTADO (COMPONENTES CONTROLADOS)

Agora que já aprendemos como definir um estado e usar seus dados no componente, estamos prontos para entender como atualizar estes dados. Para explicar como fazer isso, daremos continuidade no componente que estávamos construindo. Agora o nosso objetivo é fazer com que o estado seja atualizado toda vez

que um novo dado for inserido no `TextInput` . Para começar, vamos adicionar o `TextInput` logo abaixo da declaração do componente `Text` :

```
render() {  
  return (  
    <View>  
      <Text style={styles.font30}>{this.state.text}</Text>  
      <TextInput></TextInput>  
    </View>  
  )  
}
```

Neste `TextInput` precisamos definir alguns atributos importantes. Primeiramente vamos atribuir o estilo que definimos lá no começo do capítulo:

```
render() {  
  return (  
    <View>  
      <Text style={styles.font30}>{this.state.input}</Text>  
      <TextInput style={styles.input}></TextInput>  
    </View>  
  )  
}
```

Agora, vamos definir o atributo `value` deste componente. Esse atributo é responsável por armazenar o valor do próprio `input` , e é aqui que entra uma outra grande sacada do React. Lembra-se de que dissemos que o React possui algo chamado "componente controlado"? Pois bem, chegou a hora de apresentá-lo formalmente: os componentes controlados são aqueles capazes de administrar o seu próprio valor por meio dos estados (<https://pt-br.reactjs.org/docs/forms.html#controlled-components>). Na prática, isso significa que o componente é autossuficiente, ele não precisa que ninguém indique qual é o seu valor, ele mesmo é capaz de administrá-lo.

Mas como isso funciona na prática? É exatamente o que faremos com o componente `Evento`. Vamos dar a ele a habilidade de administrar o valor do seu próprio input por meio do `state`. Para isso, diremos que o `value` será igual ao `state.input`:

```
render() {  
  return (  
    <View>  
      <Text style={styles.font30}>{this.state.input}</Text>  
      <TextInput  
        style={styles.input}  
        value={this.state.input}>  
      </TextInput>  
    </View>  
  )  
}
```

Com isso que acabamos de fazer, estamos dizendo que o componente sempre terá o valor do seu input como uma `String` vazia. O que nos falta agora é fazer com que este dado seja atualizado. Pensando no caso do input, faz sentido dizer que seu estado deve ser atualizado toda vez que um novo caractere for digitado, afinal, ele indica um novo valor no `TextInput`.

Para lidar com as mudanças que acontecem no `TextInput`, o `React Native` nos oferece um atributo que é uma verdadeira mão na roda: o `onChangeText`. Neste atributo somos capazes de associar uma função que será executada todas as vezes em que houver uma nova alteração no componente. Vamos definir uma função anônima que fará isso. Esta função será responsável por receber o novo valor inserido no input e atualizá-lo no `state`. Para isso, faremos o seguinte:

```
import React from 'react';  
import { View, Text, TextInput, StyleSheet } from 'react-native';
```

```

class Evento extends React.Component {
  state = {
    input: ''
  }

  render() {
    return (
      <View style={styles.container}>
        <Text style={styles.font30}>{this.state.input}</Text>
        <TextInput
          style={styles.input}
          value={this.state.input}
          onChangeText={(input) => this.setState({input})}>
        </TextInput>
      </View>
    )
  }
}

export default Evento;

```

Vamos avaliar este código com atenção. Primeiramente, atualizamos o nosso componente `TextInput` dentro da função `render` com o atributo `onChangeText`. Este atributo, por sua vez, indica qual função deve ser invocada toda vez que uma mudança acontecer no componente. Neste atributo, estamos criando uma função anônima, onde esperamos por um parâmetro - na função chamada de `input` que é fornecida pelo React Native. Este parâmetro é uma `String` que possui o texto atualizado do `input`. No corpo desta função, temos a chamada da função `setState` (<https://pt-br.reactjs.org/docs/react-component.html#setstate>). Esta é a função fornecida pelo React para que possamos atualizar o estado de um componente.

Como diz a documentação do React, o `setState` enfileira mudanças ao `state` do componente e diz ao React que este componente e seus componentes filho precisam ser renderizados

novamente com a atualização do `state`. Em outras palavras, este é o principal método que utilizaremos para atualizar a interface de usuário em resposta a eventos, como toques na tela, alteração nos componentes e resposta a requisições (veremos logo mais como consumir APIs).

A função `setState` espera por parâmetro um objeto que indica quais alterações devem ser feitas. Como estamos lidando apenas com o atributo `input`, indicamos que ele deverá receber o valor associado na variável `input` recebida por parâmetro, ou seja, o que usuário acabou de digitar. Se tivéssemos mais de uma informação no `state`, não seria necessário passar o seu valor, ele é mantido desde que não seja indicado. Uma vez que esta função é chamada, o React se encarrega de atualizar o estado do componente e, quando isso acontece, algo muito importante é feito logo em seguida: a função `render` é executada, tanto no componente em si quanto nos seus componentes filhos. Desta maneira, ele garante que os dados novos do `state` foram propagados para toda a cadeia de documentos que depende dele.

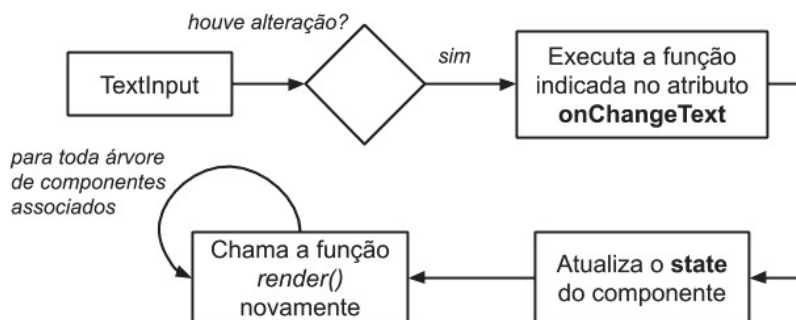


Figura 8.2: Funcionamento da função `setState` do React

A função `setState` possui mais alguns detalhes interessantes que vale a pena saber, mas por ora nos concentraremos em sua utilização mais simples e prática. Feita esta alteração no código, esperamos que, toda vez que um caractere novo for digitado no componente, o componente `Text` acima o reproduza (praticamente instantaneamente).



Figura 8.3: Componente sendo atualizado a cada mudança feita pelo usuário

E é desta maneira que conseguimos lidar com alterações de interface em componentes React. Criamos o estado inicial - que nada mais é que um objeto JavaScript - e então usamos a função `setState` para toda vez que quisermos atualizar esses valores. Uma vez alterado, o componente invoca o seu método `render` para programar todas as mudanças na árvore de componentes que

dependem dele.

8.4 COMPONENTES CONTROLADOS X NÃO CONTROLADOS

O React tem duas abordagens para lidar com entradas de usuários em inputs. Para entender como funcionam os estados no React, nós criamos um exemplo onde usamos um componente controlado. Isso significa que o valor deste componente é controlado pelo React. Ou seja, quando um usuário entra com um dado neste componente, um evento é disparado e este novo valor é armazenado como informação dentro do componente. Esta abordagem é de longe a mais utilizada no nosso dia a dia, no entanto, existe a outra: a dos componentes não controlados.

Os componentes não controlados são exatamente o inverso dos componentes controlados. Sempre que existem novas alterações nos dados, a informação atualizada é refletida na aplicação sem que o React tenha que mover um dedo. Porém, isso também significa que não podemos forçar um componente a ter um determinado valor. Nestas situações, o próprio DOM é responsável por administrá-lo.

Logo de cara este tipo de abordagem pode parecer estranho, afinal, por que iríamos querer que o React ficasse de fora? Não é melhor deixar que ele administre o estado? Na maioria dos casos, sim, mas, em muitas situações, tornando a manipulação para o DOM, conseguimos integrar o React em projetos legados. Vamos ver um exemplo rápido e prático.

Pegue na memória a última vez que você implementou um

formulário em um site e/ou sistema web. É bem provável que você tenha utilizado o JavaScript para interceptar o envio dos dados deste formulário para manipulá-los de alguma forma. Seja para fazer uma validação, um complemento, ou qualquer coisa do tipo. Para extrair o valor dos campos deste formulário, usamos a API do JavaScript. Campo a campo. Quando estamos falando de componentes não controlados, a ideia é exatamente a mesma.

O React é capaz de fazer isso por meio de recursos chamados **Refs**. Estes Refs são referências que nos permite acessar os nós do DOM de forma bem fácil. Como no caso deste livro não lidaremos com os componentes puros do HTML (`div` , `span` etc.) na construção dos nossos aplicativos, mas sim os componentes nativos do React Native, então sempre lidaremos com os componentes controlados usando as APIs disponibilizadas por eles e o `state` . Por isso, não entraremos mais a fundo nesta questão.

Mas caso tenha interessa em saber mais sobre o assunto, recomendamos a documentação oficial (<https://reactjs.org/docs/glossary.html#refs>).

Conclusão

Neste capítulo aprendemos o que são estados e eventos e para isso fizemos um exemplo prático criando um componente controlado com um input de texto e um campo de texto, de forma que, toda vez que foi digitado um caractere no input, ele foi espelhado no campo de texto.

Por fim, vimos a diferença conceitual entre um componente controlado e não controlado e em quais situações eles fazem sentido serem utilizados. No próximo capítulo aprenderemos

como unir o poder das propriedades, estados e dos componentes controlados para criar componentes capazes de consumir conteúdo da internet por meio de requisições AJAX.

REQUISIÇÕES AJAX E APIS

Neste capítulo, abordaremos um tópico que nos aproximará ainda mais do desenvolvimento de aplicativos no mundo real: estamos falando das requisições AJAX. Nos dias de hoje é difícil pensar em um serviço que não utilize um back-end na nuvem, afinal, hoje praticamente o que fazemos está conectado de alguma forma à internet. Seja a música que você escuta no Spotify, a foto que você posta no Instagram ou mesmo o aplicativo de agenda do seu celular; todos eles trabalham com requisições para a internet que fazem as conexões com os serviços na nuvem que estruturam, organizam e armazenam estes dados.

Para entender como trabalhar com requisições na internet, vamos aprender também um pouco mais sobre como funciona o ciclo de vida dos componentes e como podemos interceptá-lo para trazer os dados da internet e mostrá-los na hora em que eles foram carregados.

9.1 CICLO DE VIDA DOS COMPONENTES

Como desenvolvedores de aplicativos móveis, muitas vezes temos que cuidar do ciclo de vida de cada tela/atividade/componente porque em muitas situações

precisamos carregar os dados na tela de acordo com as ações do usuário e das requisições feitas para a internet. Por exemplo, se quisemos implementar um aplicativo de e-commerce onde o usuário será capaz de pesquisar produtos, precisamos saber como fazer os componentes inteligentes o suficiente para saber fazer as requisições, aguardar estes dados chegarem, recebê-los, processá-los e então mostrá-los. Para isso, temos que usar todo o repertório de ferramentas que aprendemos até agora em conjunto com os métodos de ciclo de vida.

Os métodos de ciclo de vida (<https://reactjs.org/docs/glossary.html#lifecycle-methods/>) do React são funcionalidades customizadas que são executadas durante as diferentes fases de um componente. Há métodos disponíveis quando o componente é criado e inserido na tela, quando o componente é atualizado, e quando o componente é desmontado e removido do DOM. Esses métodos são embutidos em todos os componentes, o que significa que estão disponíveis para usarmos a qualquer momento. Por sua vez, os componentes possuem quatro fases no seu ciclo de vida, sendo eles:

1. Montagem (*Mounting*)
2. Atualização (*Updating*)
3. Desmontagem (*Unmounting*)
4. Erros (*Error Handling*)

Cada uma destas fases possui alguns métodos associados que podem ser sobrescritos nas classes dos componentes. Para termos uma visão holística de quais são estes métodos e como eles se relacionam, a seguir temos uma imagem retirada da documentação oficial do React (<http://projects.wojtekmaj.pl/react-lifecycle->

[methods-diagram/](#)). Neste diagrama, temos as três principais fases do ciclo de vida do componente (a parte de erros ficou de fora) e como os métodos navegam entre estas três fases:

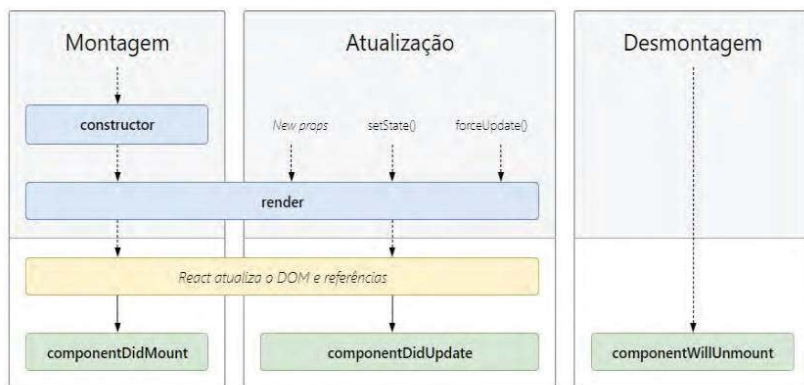


Figura 9.1: Ciclo de vida dos componentes React

Os métodos que estão na parte cinza da imagem estão no que chamamos de "Fase Render": ela é pura e sem efeitos colaterais. Pode ser pausada, abortada ou reiniciada pelo React. As que estão na parte de baixo são as pertencentes a "Fase Commit": podem operar o DOM, executar efeitos colaterais e agendar atualizações. Durante os outros capítulos nós já vimos alguns destes métodos, como o `constructor`, `render` e `setState`. No entanto existem mais alguns que valem a pena conhecermos para dominarmos melhor o uso dos componentes. Vamos entendê-los um pouco mais.

Montagem

Esses métodos são chamados na seguinte ordem quando uma

instância de um componente está sendo criada e inserida no DOM:

1. `constructor`
2. `render()`
3. `componentDidMount()`

O `constructor` de um componente é chamado antes de ser montado. Ao implementar o construtor para uma subclasse `React.Component`, devemos chamar `super(props)` antes de qualquer outra instrução. Caso contrário, `this.props` será indefinido no construtor, o que pode levar a erros. A definição do `constructor` nos componentes não é obrigatória, mas existem duas situações em que ela deve ser usada:

1. Inicializando o estado (`state`), atribuindo um objeto a `this.state`
2. Vinculando métodos manipuladores de eventos a uma instância

O código que fizemos no capítulo anterior é um bom exemplo onde o `constructor` seria bem-vindo. Vamos refatorar aquele código para torná-lo compatível com estas situações:

```
import React from 'react';
import { View, Text, TextInput, StyleSheet } from 'react-native';

class Evento extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      input: ''
    }

    this.alteraInput = this.alteraInput.bind(this);
  }
}
```

```

    alteraInput(input) {
      this.setState({input})
    }

    render() {
      return (
        <View style={styles.container}>
          <Text style={styles.font30}>{this.state.input}</Text>
          <TextInput
            style={styles.input}
            value={this.state.input}
            onChangeText={this.alteraInput}>
          </TextInput>
        </View>
      )
    }
  }
}

export default Evento;

const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#fff',
    alignItems: 'center',
    justifyContent: 'center',
  },
  input: {
    height: 50,
    width: 300,
    fontSize: 30,
    borderWidth: 1,
    borderColor: 'black',
  },
  font30: {
    fontSize: 30
  }
});

```

Note o que fizemos: o método construtor recebe as propriedades e as repassa para a classe pai. Logo em seguida, definimos o nosso state diretamente como um objeto usando o this.state (e esse é o único momento em que podemos atribuir

estados desta maneira. Para as demais alterações, devemos usar o método `setState`). Logo abaixo, temos a definição do método que lida com um evento, no nosso caso o evento de mudança no input. Chamamos esse método de `alteraInput` e usamos o `setState` para atualizar o `input` .

Uma vez que este código é executado, o método `render` é chamado. Ele faz exatamente o que já vimos anteriormente, ele retorna o que deverá ser renderizado na tela. No entanto, logo após isso, temos um método novo, o `componentDidMount` . Este é o primeiro método de ciclo de vida dos componentes que ainda não vimos. Ele é chamado imediatamente depois que o componente é montado e inserido na árvore de componentes. Este é o lugar ideal para fazer requisições para a internet e veremos como fazer isso logo mais.

Atualização

Uma atualização no React pode acontecer por mudanças em seu estado e/ou propriedades. Os métodos listados a seguir lidam com este fluxo quando um componente é novamente renderizado:

1. `setState()`
2. `forceUpdate()`
3. `componentDidUpdate()`

O método `setState` foi o que já estudamos até então. Ele é o responsável por enviar mudanças de estado para o componente. Neste método, passamos apenas aquelas propriedades do estado que devem ser atualizadas. Uma vez que este processo acontece, o React é capaz de identificar as mudanças e propagá-las em toda a cadeia de componentes que estão associados à atualização.

O segundo método no entanto é novo, o `forceUpdate`. Como o próprio nome já nos indica, este método é útil para quando precisamos renderizar novamente o nosso componente, mas ele não depende exclusivamente das propriedades e dos estados. Apesar de termos esta opção, os próprios engenheiros do React não recomendam o seu uso em excesso, sendo mais favorável a utilização da propriedades e estados para isso.

O outro método que faz parte deste processo é o `componentDidUpdate`. De forma análoga ao que vimos no `componentDidMount` no ciclo de montagem, este método é executado imediatamente quando o componente termina de ser atualizado. Este é um bom momento para operar no DOM ou mesmo fazer requisições para a internet, desde que tenhamos o cuidado de comparar as propriedades.

Desmontagem

Como nada nessa vida dura, os componentes também eventualmente morrem. E neste momento também podemos atuar através do método `componentWillUnmount`. Ele é invocado imediatamente antes de um componente ser desmontado e destruído (para ganharmos em otimização). Este método é extremamente útil caso seja necessário fazer qualquer tipo de limpeza, tal como invalidar *timers* e requisições para a internet.

Métodos de ciclo de vida raramente utilizados

Nos tópicos anteriores abordamos os métodos mais utilizados quando estamos falando de ciclo de vida dos componentes. Porém, a verdade é que existem algumas outras APIs que podemos usar

para manipular este ciclo.

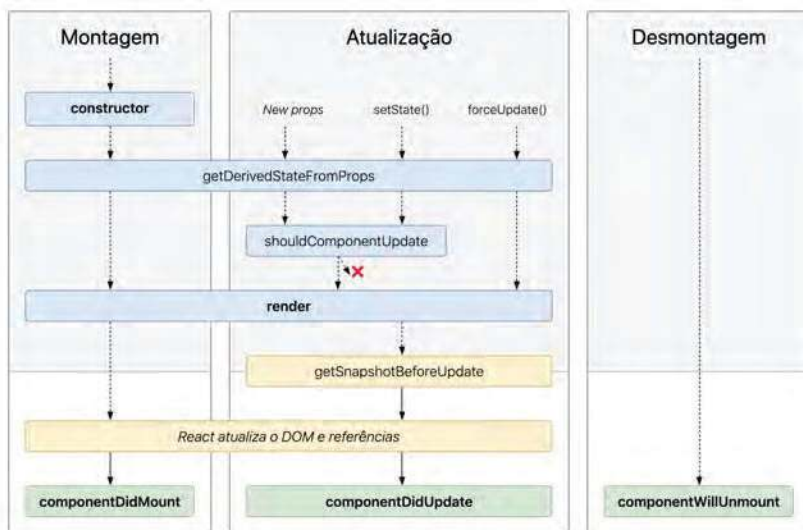


Figura 9.2: Métodos raros de ciclo de vida

Como podemos observar na imagem, estes métodos são:

1. `getDerivedStateFromProps`
2. `shouldComponentUpdate`
3. `getSnapshotBeforeUpdate`

Vamos entender um a um o que eles fazem. O primeiro método da nossa lista, o `getDerivedStateFromProps`, é invocado imediatamente antes de chamar o método de `render`, tanto na montagem inicial quanto nas atualizações subsequentes. Ele deve retornar um objeto para atualizar o estado ou nulo para atualizar nada. Esse método existe para casos de uso extremamente raros, em que o estado depende de alterações nas propriedades ao longo do tempo do ciclo de vida.

O segundo método, o `shouldComponentUpdate`, é utilizado em casos onde queremos que o React saiba se a saída de um componente não é afetada pela alteração atual no estado ou propriedades. Já entendemos que o comportamento padrão é renderizar novamente em cada mudança de estado, mas podem existir casos em que não queremos em que isso aconteça. Este método é chamado logo antes do `render` quando estes novos valores são recebidos pelo componente. Por definição, o retorno desta função é `true`, mas temos a liberdade de implementar regras complexas que mudem isso. A principal razão para o seu uso é puramente por performance, pois com isso conseguimos evitar renderizações "à toa".

Por fim, temos o `getSnapshotBeforeUpdate`. Este método do ciclo de vida de atualização é invocado imediatamente antes de a saída processada mais recentemente ser confirmada para o DOM. Ele permite que nosso componente capture algumas informações do DOM (por exemplo, posição de rolagem) antes que seja potencialmente alterado. Qualquer valor retornado por este ciclo de vida será passado como um parâmetro para `componentDidUpdate`.

9.2 AJAX

Acreditamos que o exemplo mais prático aplicável aos métodos de ciclo de vida dos componentes sejam as requisições para a internet. AJAX é o acrônimo de *Asynchronous JavaScript and XML*, ou seja, JavaScript e XML assíncrono. Usamos o termo assíncrono para definir processos que não são síncronos, isto é, não apresentam sincronia. Em termos computacionais, os processos sem sincronia são aqueles nos quais é impossível prever

de antemão o seu término, por exemplo, uma requisição para a internet — já que esta depende de inúmeras variáveis, desde fatores físicos até virtuais.

O AJAX surgiu no início da web como alternativa para fazermos requisições para uma página e carregar um conteúdo de forma dinâmica sem a necessidade de atualizar toda a página. Para entender o que isso quer dizer, imagine o perfil de uma pessoa que usa o Facebook e/ou Instagram. No momento em que a pessoa abre o aplicativo/site, o serviço dispara uma série de chamadas concorrentes para a sua API afim de carregar as informações do usuário: fotos, amigos, curtidas, reações etc. Como este processo pode levar um tempo — afinal, estamos falando de uma grande quantidade de informação — elas são propositalmente requisitadas simultaneamente e carregadas de acordo com sua chegada no front-end. É por isso que geralmente os dados "leves" como textos e menus são carregados antes de fotos e vídeos. Mas o importante disso tudo é que a experiência de carregamento é pouco desagradável, já que o usuário consegue ter o mínimo de informação disponível na tela logo ao acessá-lo.

Este processo funciona muito bem no mundo web e é facilmente replicável no mundo dos aplicativos híbridos usando o React Native. Com os métodos de ciclo de vida dos componentes conseguimos acionar requisições para APIs e então atualizar o conteúdo do componente de acordo com os dados que ele possui. Para entender como tudo isso funciona na prática, vamos criar um novo componente do zero que terá esta capacidade.

Volte ao nosso projeto e na pasta `componentes/` crie o arquivo `UsuarioGithub.js`. Este componente será responsável

por acessar a API REST aberta do GitHub e renderizar os dados de um usuário que será inserido dentro de um `TextInput` . Veja só que, para cumprir este simples desafio, teremos que usar praticamente todas as ferramentas de React Native que aprendemos até agora.

Começaremos o componente com a velha receita de bolo, importando as bibliotecas necessárias do React e do React Native. Para este caso, teremos que usar os estados e propriedades, então optaremos por um componente de classe. Além disso, a princípio utilizaremos o `Text` , `View` , `TextInput` e `Button` .

```
import React from 'react';
import {Button, Text, View, TextInput, StyleSheet} from 'react-native';

class UsuarioGithub extends React.Component {

  render() {
    return <div></div>
  }

}

export default UsuarioGithub;
```

Pensando que nosso componente terá de lidar com os dados do `TextInput` e da API do GitHub, vamos inicializar o seu estado com dois atributos diferentes: `dados` e `usuario` . Atribuiremos um objeto vazio à variável `dados` , já que ela representa os dados vindos da API e inicialmente não temos nenhum. Por outro lado, vamos atribuir a String "octocat" como inicial para a variável `usuario` . Faremos isso apenas para que o componente carregue a informação do usuário do GitHub antes que o usuário possa escolher qual usuário ele quer investigar.


```

import React from 'react';
import {Button, Text, View, TextInput} from 'react-native';

class UsuarioGithub extends React.Component {

  constructor(props) {
    super(props);
    this.state = {
      dados: {},
      usuario: "octocat"
    }
  }

  render() {
    return <div></div>
  }
}

export default UsuarioGithub;

```

Agora que temos nosso estado inicial estruturado, vamos pensar na estrutura que devolveremos dentro deste componente. Como o objetivo aqui é fazer uma requisição para a web e atualizar o componente de acordo com os dados que vão chegando, vamos manter o foco e deixar o componente visualmente simples. Para isso, vamos estruturá-lo apenas com um espaço para imprimir a resposta da API na tela, um input e um botão que será usado para forçar a ação de busca conforme o que o usuário digitar.

```

render() {
  return (
    <View>
      <Text>{JSON.stringify(this.state.dados)}</Text>
      <View>
        <TextInput />
      </View>
      <View>
        <Button />
      </View>
    </View>
  )
}

```

```
)  
}
```

O componente `Button` (<https://facebook.github.io/react-native/docs/button.html/>) possui algumas propriedades bem legais que podemos usar, entre elas:

1. `onPress` : usado para apontar qual função o botão deve executar quando clicado
2. `title` : título do botão
3. `color` : cor do botão
4. `accessibilityLabel` : Label complementar para acessibilidade

Vamos preencher esses atributos no nosso botão:

```
<Button  
  onPress={this.fetchDados}  
  title="Buscar Dados"  
  accessibilityLabel="Busque os dados do usuário no GitHub"  
>
```

Para o atributo `onPress` passamos uma função chamada `fetchDados` que também ainda não implementamos. Essa função será a grande responsável por fazer a requisição AJAX para a API. Para implementar isso, usaremos a **API fetch** (https://developer.mozilla.org/pt-BR/docs/Web/API/Fetch_API/Using_Fetch). Essa API está disponível na maior parte das engines e nos permite fazer requisições de modo stupidamente simples. Além disso, essa API em particular trabalha com `Promises` em vez de `callbacks`, o que também ajuda muito na hora de organizar o código (lembrando que `Promises` é uma funcionalidade adicionada à linguagem na especificação ES2015).

A seguir, temos como usar esta API para fazer uma requisição:

```
fetchDados() {  
  fetch(`https://api.github.com/users/${this.state.usuario}`)  
    .then(response => response.json())  
    .then(json => this.setState({dados: json}))  
    .catch(err => this.setState({dados: {err}}))  
}
```

Pareceu confuso? Então vamos entender passo a passo este código. Primeiramente, chamamos o `fetch` e passamos a ele o endereço do recurso que queremos acessar. Este recurso não foi inventado, ele é o `endpoint` para a API de usuários disponível pelo GitHub (<https://developer.github.com/v3/users/#get-a-single-user/>). Ao final deste recurso, em vez de passar o nome de um usuário em específico, usamos a interpolação de Strings para buscar o nome do usuário registrado no estado do componente.

O segundo passo foi a atribuição do `.then(response => response.json())`. Este código é disparado assim que temos uma resposta positiva da API. Como o objeto retornado pelo `fetch` não contém somente a resposta da API, mas também uma série de informações adicionais sobre a requisição que foi feita, precisamos usar o método `json` que nos retorna uma outra `Promise`. Na linha de baixo tratamos essa `Promise` que nos dá a resposta do servidor no objeto `json`, que então é atribuído ao `state` com a função `setState`. A última linha é disparada somente em casos de insucesso na requisição.

Muito bem, agora que temos o nosso método de requisição prontos, vamos encaixá-lo nos momentos adequados dentro da aplicação. Já fizemos com que o método seja invocado toda vez que o `Button` for pressionado. Fizemos isso por meio da propriedade `onPress` que a API do React Native nos disponibiliza. Agora

vamos ir além e fazer com que esse método seja invocado pela primeira vez logo que o componente for renderizado na tela. Se consultarmos novamente os métodos de ciclo de vida dos componentes React, constataremos que o método mais adequado para isso é o `componentDidMount` .

Vamos então fazer a chamada do método `fetchDados` dentro do `componentDidMount` .

```
componentDidMount() {  
  this.fetchDados();  
}
```

Pensando na nossa configuração de estado inicial deste componente, assim que ele for renderizado na tela, ele vai disparar uma busca na API do GitHub usando o usuário `octocat` . Quando estes dados forem retornados do servidor, o estado será atualizado e então o método `render` será invocado para renderizar novamente o conteúdo na tela.

O último passo necessário para finalizarmos este componente é torná-lo controlável, afinal, temos o `TextInput` que armazenará a informação de qual foi a entrada do usuário no nosso componente. Para fazer isso, vamos usar dois atributos que vimos no capítulo anterior, o `onTextChange` e o `value` .

```
render() {  
  return (  
    <View style={styles.container}>  
      <Text>{JSON.stringify(this.state.dados)}</Text>  
      <View>  
        <TextInput  
          onChangeText={ usuario => {this.setState({usuario})} }  
          value={this.state.usuario}>  
        </TextInput>  
      </View>  
    <View>
```

```

        <Button
          onPress={this.fetchDados}
          title="Buscar Dados"
          accessibilityLabel="Busque os dados do usuário no GitHub"
        />
      </View>
    </View>
  )
}

```

Agora sim, nosso componente está pronto para rodar em produção. Ao final de todos os passos, ele deve ter esta cara:

```

import React from 'react';
import {Text, View, TextInput, Button} from 'react-native';

class UsuarioGithub extends React.Component {

  constructor(props) {
    super(props);
    this.state = {
      dados: {},
      usuario: 'octocat'
    }

    this.fetchDados = this.fetchDados.bind(this);
  }

  fetchDados() {
    fetch(`https://api.github.com/users/${this.state.usuario}`)
      .then(response => response.json())
      .then(json => this.setState({dados: json}))
      .catch(err => this.setState({dados: {err}}))
  }

  componentDidMount() {
    this.fetchDados();
  }

  render() {
    return (
      <View>
        <Text>{JSON.stringify(this.state.dados)}</Text>

```

```

    <View>
      <TextInput
        onChangeText={ usuario => {this.setState({usuario})}}
        value={this.state.usuario}>
      </TextInput>
    </View>
    <View>
      <Button
        onPress={this.fetchDados}
        title="Buscar Dados"
        accessibilityLabel="Busque os dados do usuário no Git
Hub"
      />
    </View>
  </View>
)
}
}

export default UsuarioGithub;

const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#fff',
    alignItems: 'center',
    justifyContent: 'center',
  }
});

```

Para testá-lo, vamos adicioná-lo ao nosso bom e velho companheiro `App.js`. Altere-o para incluir o componente que acabamos de construir.

```

import React from 'react';
import { StyleSheet, Text, View } from 'react-native';
import UsuarioGithub from './componentes/UsuarioGithub';

export default function App() {
  return (
    <UsuarioGithub />
  )
}

```

```
);
}
```

Ao testar, precisamos validar duas coisas importantíssimas. Primeiro, precisamos verificar se a requisição está sendo feita para o GitHub e os dados iniciais com o usuário `octocat` estão sendo carregados. Em segundo lugar, temos de testar se nosso `TextInput` e `Button` funcionam. Experimente fazer a busca procurando os dados do seu usuário!



Figura 9.3: Requisição AJAX realizada no componente

E pronto! Fizemos nosso primeiro componente que se comunica com a internet! Experimente fazer uma série de buscas com outros usuários, inclusive aqueles que não existem (veja o resultado na tela)!

Conclusão

Neste capítulo, fomos fundo nos métodos de ciclo de vida dos componentes React e vimos como podemos utilizá-los para manipular como e quando as informações são processadas e renderizadas nos componentes. Junto a isso, exploramos as requisições AJAX, que nada mais são do que uma ferramenta disponível no JavaScript para que possamos fazer requisições para a internet e alterar uma página (ou no nosso caso, uma tela) sem a necessidade de carregá-la inteiramente novamente. Com todas as ferramentas que aprendemos até o momento, conseguimos unir os métodos de ciclo de vida, os estados, propriedades, classes, CSS-in-JS para criar um componente capaz de pesquisar na API do GitHub as informações públicas sobre um usuário.

NAVEGAÇÃO

Quando falamos de aplicativos que são usados no mundo real, dificilmente teremos um onde o usuário será capaz de realizar todas as operações em uma única tela. Apesar de o React ser uma tecnologia de aplicação de tela única (*Single Page Application*), ele nos oferece ferramentas bem interessantes para que possamos manipular e trocar o conteúdo da tela, dando a impressão ao usuário de que são telas diferentes, quando na verdade não são.

Neste capítulo vamos explorar como usar essa funcionalidade e incluir navegação e menus laterais em um aplicativo no React Native.

10.1 REACT NAVIGATION

Quando estamos falando de navegação entre telas, existem inúmeras opções muito boas no mercado para o React Native. Neste livro, utilizaremos a mais recomendada pela própria documentação oficial do React Native. Estamos falando da biblioteca React Navigation.

Antes de começarmos, vale ressaltar que no momento em que estamos escrevendo este livro a biblioteca está na versão 3.11.1, então é muito provável que quando você estiver lendo este capítulo

a última versão lançada já seja outra. Isso significa que existe o risco de que os passos que indicaremos estejam em parte desatualizados. Mas antes de entrarmos em desespero é bem importante termos consciência de que os desenvolvedores são bem atenciosos à grande quantidade de usuários da biblioteca, de modo que as mudanças feitas nela dificilmente quebram as versões anteriores - e, se no caso quebrarem, geralmente os ajustes necessários para adequar o código são poucos.

Em todo caso, é sempre aconselhável manter o olho na documentação do projeto. Acompanhe os passos que daremos e, caso encontre algum problema, dê uma olhada na documentação no site oficial (<https://reactnavigation.org/>). Além de estar muito bem escrita, tem uma linguagem bem fácil de entender e existe até mesmo uma tradução para português.

Tendo este ponto esclarecido, vamos pôr as mãos na massa. O primeiro passo para usar a biblioteca será trazer a dependência para dentro do nosso projeto. Para isso, acesse o seu terminal, entre na raiz do projeto e execute o seguinte comando:

```
npm install --save react-navigation
```

Isso instalará a biblioteca e todas as suas dependências - ou melhor dizendo, quase todas. Junto a esta biblioteca temos que trazer uma outra dependência, para a parte que nos ajudará na detecção de gestos na aplicação. Isso será bastante útil para quando formos construir o nosso menu lateral.

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

Muito bem, já temos nossas dependências instaladas. Agora partiremos para a construção dos menus. Dentre os vários tipos de

menus disponíveis, separaremos três que julgamos serem os mais usados no geral: o menu lateral, o menu navegável por meio de botões e links; e o menu por abas.

10.2 NAVEGAÇÃO POR MENU LATERAL

Para nosso primeiro exemplo, utilizaremos o menu chamado de `Drawer Navigator` - que podemos traduzir livremente para algo como "Navegador de gaveta". A tradução para a nossa língua é bem esquisita, mas este tipo é bem comum e utilizado por grande parte dos aplicativos hoje em dia. Este tipo de navegação é o clássico navegador lateral. Ele aparece toda vez que arrastamos a tela da esquerda para a direita.

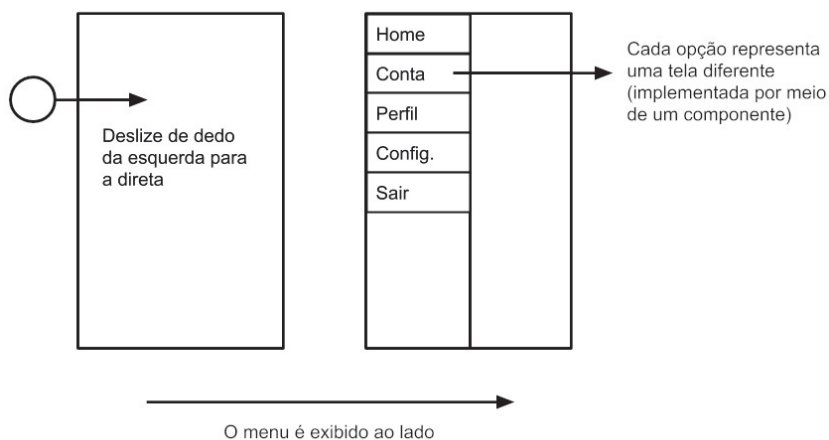


Figura 10.1: Exemplo de uso do menu lateral

Até o momento, estivemos trabalhando com o `App.js` como o ponto de entrada do aplicativo. Como incluiremos um menu, esse se tornará o novo ponto de partida daqui para a frente.

Tecnicamente, isso significa que o `App.js` se tornará o nosso menu e, para tal, vamos usar alguns métodos da biblioteca `react-navigation`. Acesse o arquivo `App.js` e logo no início inclua os imports necessários: o método `createDrawerNavigator` e `createAppContainer`.

```
import {createDrawerNavigator, createAppContainer} from 'react-navigation';
```

O método `createDrawerNavigator` será o responsável por criar e administrar a nossa estrutura de menu. Este método aceita dois objetos literais como parâmetros. No primeiro, nós indicaremos quais são as telas do nosso aplicativo (na ordem em que desejamos que elas sejam colocadas). Aqui aproveitaremos todos os nossos componentes criados até agora. Desde o `OlaMundo` até o `UsuarioGithub`. Vamos criar uma tela para cada um deles — e para os próximos exemplos continuaremos seguindo esta estratégia.

No segundo parâmetro, nós passaremos algumas opções de configuração. A documentação indica que existem mais de 15 configurações possíveis; no nosso exemplo, usaremos somente aquelas correspondentes à configuração das cores: `drawerBackgroundColor`, `overlayColor` e `contentOptions`. Os detalhes de todas as outras podem ser encontrados na documentação (<https://reactnavigation.org/docs/en/drawer-navigator.html>).

```
const DrawerNavigator = createDrawerNavigator({
  {
    DimensoesFixas: DimensoesFixas,
    Evento: Evento,
    OlaMundo: OlaMundo,
    UsuarioGithub: UsuarioGithub,
    ChecaNumero: () => <ChecaNumero numero={3}/>
  }
});
```

```

    },
    {
      hideStatusBar: true,
      drawerBackgroundColor: 'rgba(255,255,255,.9)',
      overlayColor: '#6b52ae',
      contentOptions: {
        activeTintColor: '#fff',
        activeBackgroundColor: '#6b52ae',
      },
    }
  );

```

Aqui há algo bem importante que vale a pena mencionarmos. Notou que o componente `ChecaNumero` está diferente dos demais em sua declaração? Isso acontece porque este componente passa propriedades adiante. Sempre que for este o caso, devemos usar uma função anônima que retorna o componente com seus parâmetros correspondentes. Quando o caso não for este, podemos usar a forma resumida, onde basta passarmos o nome do componente.

Voltando ao exemplo, escolhemos algumas cores, mas fique à vontade para alterá-las ao seu gosto. O mais importante daqui é que possamos construir o nosso menu lateral de maneira funcional. Feito isso, usaremos este objeto criado e vamos passá-lo como parâmetro para o segundo método que importamos do `react-navigation`. Estamos falando do `createAppContainer`.

```
const App = createAppContainer(DrawerNavigator);
```

E pronto! Como já estamos exportando o nosso componente, o Expo já conseguirá fazer o build dele e o menu entrará em ação. Salve o arquivo, suba o Expo no seu computador e confira o resultado (lembrando que é necessário arrastar o dedo da esquerda para a direita, em qualquer tela). Se tudo for feito corretamente, você visualizará o seguinte resultado:

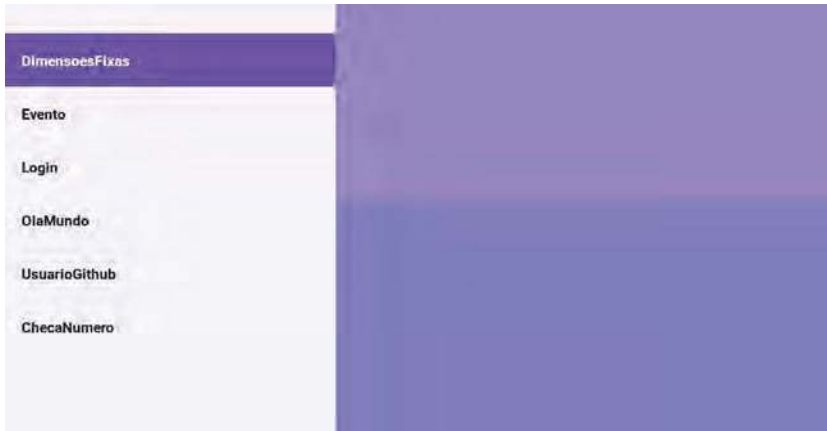


Figura 10.2: Exemplo de uso do menu lateral

Agora que já sabemos como usar este tipo de menu, vamos dar uma olhada em outro tipo, o `Stack Navigator`.

10.3 NAVEGAÇÃO POR LINKS

Este tipo de navegação é ativado por meio de links e botões na aplicação. Sempre que quisermos redirecionar o nosso usuário para outra tela, usaremos este tipo de menu. E então, da mesma maneira que um navegador web "empilha" as páginas em que visitamos, o mesmo será feito aqui. Daí o nome "Stack Navigator", termo que podemos traduzir como "Navegador por Pilha".

Caso você ainda não esteja familiarizado com o termo "Pilha" na Ciência da Computação, ela nada mais é do que uma estrutura de dados do tipo "LIFO" (*Last In, First Out*). Nesta estrutura, todos os novos dados são sempre inseridos no topo. Quando queremos remover um item, tiramos sempre do topo para baixo. É como se fosse uma grande pilha de pratos sujos: toda vez que adicionamos

um novo prato à pilha, ele sempre fica no topo. No entanto, quando vamos lavar, nunca começamos pelo primeiro prato da pilha (até porque tudo cairia no chão), mas sim pelo último que inserimos.

Os navegadores Web funcionam da mesma forma, a cada site que você visita ele é inserido na pilha. Quando clicamos no botão voltar, ele recupera o último site da pilha. E o processo se repete até a pilha terminar. Pensando no aplicativo, faremos o mesmo processo: a cada nova tela visitada ela será adicionada no topo da pilha de telas visitadas. Quando o usuário pressionar o botão voltar no seu aparelho, o React Navigation recuperará da pilha o último item adicionado.

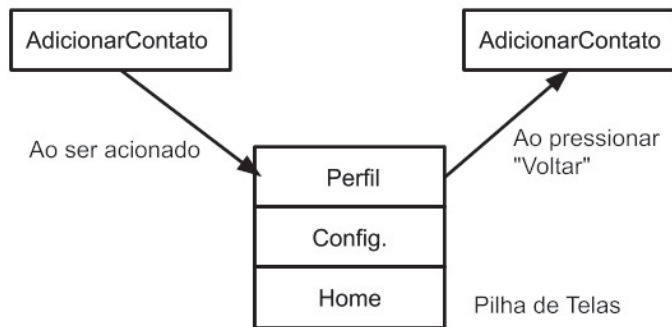


Figura 10.3: Exemplo de uso do menu lateral

Muito bem. Agora que já entendemos o que é uma pilha (como estrutura de dados) e como isso afeta o tipo de menu que criaremos, vamos para a parte prática. A primeira coisa que faremos é criar um novo componente que representará o ponto de entrada da nossa aplicação. Faremos isso por dois motivos.

Primeiro, para que não percamos o menu lateral que já montamos no `App.js`. Em segundo lugar, como este menu é baseado em links e botões, precisamos inserir estes novos componentes na tela. Isso ficará melhor se colocado em um componente à parte, assim, deixamos o `App.js` apenas como administrador das telas.

Abra a pasta `componentes` e crie um novo arquivo JavaScript com o nome `Home.js`. Entre no arquivo e importe os componentes que serão necessários.

```
import React from "react";
import { View, Text, Button } from "react-native";
import { createStackNavigator } from "react-navigation";
```

Agora como conteúdo vamos criar uma tela simples, apenas uma label e uma botão que vai nos direcionar para uma das telas do nosso aplicativo. Vamos começar criando um botão que realoca nosso usuário para o componente `DimensoesFixas`. Como já vimos anteriormente, o componente `Button` do React Native possui alguns atributos bem interessantes que nos ajudam a configurá-lo, como é o caso do `title` — que representa o título do botão — e o `onPress` — que nos permite indicar o que acontece quando clicamos no botão. Vamos usá-los novamente.

Na propriedade `onPress`, nós vamos usar uma propriedade que o React Navigator passa para todas as telas, o `this.props.navigation`. Neste objeto nós encontramos o método `navigate` que é o responsável por indicar a navegação que uma nova tela deve ser carregada ao usuário. Este método aceita um único parâmetro que é exatamente o nome do componente para onde o React Navigator deve redirecionar o usuário.

```
import React from "react";
```



```
import { View, Text, Button } from "react-native";
import { createStackNavigator } from "react-navigation";

class Home extends React.Component {
  render() {
    return (
      <View style={{
        flex: 1,
        alignItems: "center",
        justifyContent: "center"
      }}>
        <Button
          title="Dimensoes Fixas"
          onPress={() =>
            this.props.navigation.navigate('DimensoesFixas')
          }
        />
      </View>
    );
  }
}

export default Home;
```

Com isso, nosso componente já está pronto. Volte ao arquivo App.js para que possamos configurar o menu. Primeiramente, importe o método `createStackNavigator`.

```
import { createStackNavigator, createAppContainer } from 'react-navigation';
```

Agora vamos configurar nossas telas:

```
const AppNavigator = createStackNavigator({
  Home: { screen: Home },
  DimensoesFixas: { screen: DimensoesFixas }
});
```

Por fim, não podemos esquecer de passar este objeto para o método `createAppContainer`.

```
const App = createAppContainer(AppNavigator);
```

Inicie o aplicativo no seu aparelho e confira o resultado. Você notará que no topo do aplicativo será criado um tipo de cabeçalho que ganhará uma seta virada para a esquerda toda vez que uma nova tela for acessada.



Figura 10.4: Exemplo de navegação por botões

10.4 NAVEGAÇÃO POR ABAS

Por fim, vamos ver estudar mais um tipo de navegação bastante utilizado, o menu por abas (<https://reactnavigation.org/docs/en/tab-based-navigation.html/>). Nesse formato, um rodapé é criado e as guias disponíveis são disponibilizadas para o usuário. Toda vez que uma nova aba é pressionada no rodapé, o aplicativo carrega a tela correspondente.

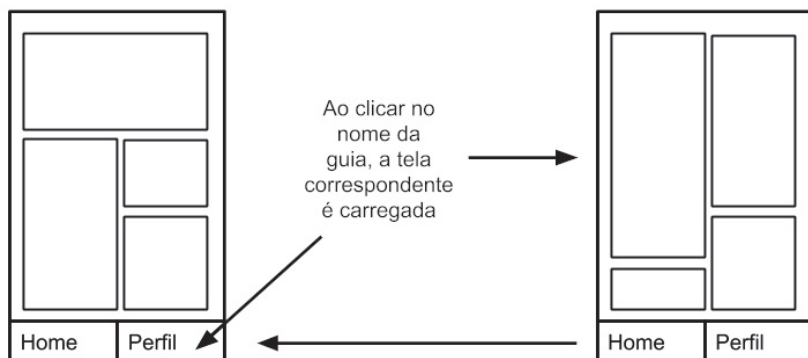


Figura 10.5: Funcionamento da navegação por abas

A configuração deste tipo de navegação é muito semelhante ao que já fizemos nos dois exemplos anteriores. Em primeiro lugar precisamos ir até o arquivo `App.js` e trazer o método `createBottomTabNavigator` do React Navigation.

```
import { createBottomTabNavigator, createAppContainer } from 'react-navigation';
```

Para aproveitar o código que já criamos e, explorar as possibilidades que o React Navigation nos oferece, vamos utilizar o componente `Home` como tela inicial do aplicativo. A grande sacada é que neste componente nós implementamos um outro tipo de navegação, o menu por botões. O que você acha que acontecerá quando misturarmos estes dois tipos? Vamos verificar inserindo o menu por guias no `App.js` e passando este código para o já conhecido `createAppContainer`. E para não nos alongarmos muito no código, vamos colocar apenas duas opções.

```
const TabNavigator = createBottomTabNavigator({  
  Home: Home,
```

```
DimensoesFixas: DimensoesFixas
});
```

Não é necessário utilizar especificamente essas telas, fique à vontade para usar as que você achar mais apropriado. O importante é que no final do dia tudo funcione como esperado. E para que isso aconteça corretamente, vamos passar o objeto `TabNavigator` para o `createAppContainer`.

```
const App = createAppContainer(TabNavigator);
```

E pronto, configuramos o mínimo necessário para que tudo funcione. Agora vamos testá-lo.



Figura 10.6: Exemplo de navegação por abas

Repare que, quando acessamos a tela `DimensoesFixas` e usamos o botão da tela em vez do botão no rodapé, o cabeçalho que tínhamos no exemplo anterior não aparece mais. Isso acontece porque este novo tipo entende o acesso da tela e o incorpora. E isso é muito bom, afinal, não iríamos querer que o usuário tivesse as duas opções ao mesmo tempo, não é?

Conclusão

Neste capítulo, começamos a falar de um aspecto fundamental de qualquer aplicativo: a navegação. Descobrimos que o React Native não possui nenhuma API nativa que nos ajude com isso, entretanto, ele se integra muito bem com uma biblioteca chamada React Navigation. Essa biblioteca, por sua vez, é um projeto de código aberto que nos traz várias possibilidades de menus diferentes, sendo que aqui exploramos três deles: a navegação por menu lateral, por botões e links; e por fim a por abas. Em cada um destes tipos fizemos o passo a passo de como implementá-lo e constatamos que o processo é extremamente simples.

Vale ressaltar que aqui vimos apenas a ponta do iceberg no que se trata de navegação. Recomendamos que você visite os links que nós indicamos em cada um dos métodos e explore todas as opções de configurações disponíveis na biblioteca. Não temos dúvidas de que as configurações disponíveis vão atender à maioria dos casos mais comuns de navegação em aplicativos que usamos em produção.

INTEGRAÇÃO COM O BANCO DE DADOS DO FIREBASE

Ao desenvolver uma aplicação, seja ela mobile, web ou desktop; existe uma série de preocupações que precisamos ter, tais como: infraestrutura, performance, atualizações, mecanismos de login, armazenamento de dados, gerenciamento de erros... E a lista continua. Foi pensando em uma maneira de solucionar todos esses problemas em uma tacada só que surgiu a motivação para a criação do tão conhecido Firebase.

Caso você nunca tenha ouvido falar neste nome, o Firebase é o que chamamos de BaaS (*Backend as a Service*) para aplicações web e mobile, desenvolvido e mantido pelo gigante Google. Sua primeira versão foi lançada há muitos anos, em meados de 2004, e com o passar dos anos a plataforma cresceu e evoluiu muito, tornando-se uma das plataformas favoritas de muitos desenvolvedores no Brasil e no mundo. Sua popularidade se dá ao fato de o serviço oferecido pelo Firebase ser muito bom e bastante vasto.

A verdade é que poderíamos escrever um livro inteiro para

falar somente sobre esta tecnologia, afinal, ela oferece uma gama de serviços que podem ser utilizados dentro da sua aplicação. Estes serviços são separados em quatro grandes categorias, sendo elas: Analytics, Develop, Grow e Earn. Neste capítulo, vamos centralizar em como usar o serviço de Banco de Dados em Tempo Real (*Realtime Database*) do Firebase para armazenar os dados de um aplicativo construído com o React Native.

11.1 CONFIGURAÇÃO

O Banco de Dados em Tempo Real do Firebase permite o armazenamento e sincronismo dos dados entre usuários e dispositivos em tempo real com um banco de dados NoSQL hospedado na nuvem. Os dados atualizados são sincronizados em todos os dispositivos conectados em questão de segundos. Além disso, nossos dados permanecem disponíveis caso o aplicativo fique offline, o que oferece uma ótima experiência do usuário, independentemente da conectividade de rede (já que nos dá a possibilidade de fazer a sincronização de dados quando houver conectividade, mas sem deixar o aplicativo inapto). Mas como tudo o que é bom nesta vida (e na tecnologia), este recurso tem limitações dentro da versão gratuita, entretanto, que serão mais do que suficientes para os nossos experimentos no livro e nos seus primeiros aplicativos do React Native em produção.

O primeiro passo para podermos desfrutar destes serviços é ir até o site oficial do Firebase (<https://firebase.google.com/>). Uma vez lá dentro, precisamos logar com a nossa conta do Google. Caso você ainda não tenha uma, é necessário criá-la gratuitamente. Feito isso, volte a página do Firebase e clique no botão `Ir para o console` localizado logo na parte superior direita do site. Na tela

seguinte, crie um novo projeto com o botão como mostra a imagem a seguir.



Figura 11.1: Criando um projeto no painel do Firebase

Para criar um projeto, precisamos dar um nome para ele. Fique à vontade para escolher o nome que melhor lhe atende, nós optaremos por "Lista de Linguagens". Aceite os termos de uso do Firebase e clique em **continuar**.

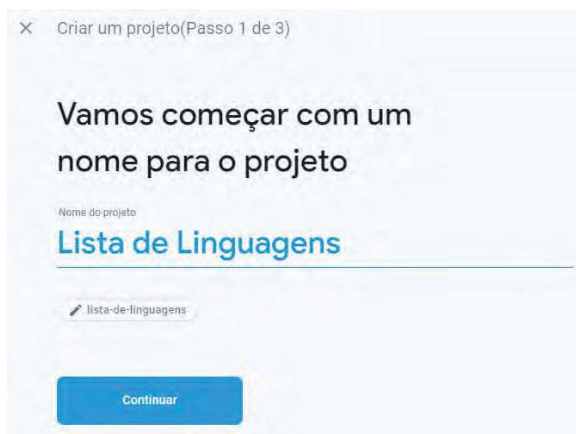


Figura 11.2: Nomeando o projeto no painel do Firebase

O próximo passo é opcional, pois se trata de configurar o Analytics no projeto. Caso você não esteja familiarizado com o serviço, o Analytics é uma solução completa para monitoramento de acesso e uso do seu aplicativo. Ele nos traz informações importantes como: número de acessos, localização, idade, sexo etc. Em termos de conhecer melhor o público do aplicativo, é uma solução impecável.

Mas novamente, como não vamos explorar todos os recursos do Google integrados ao Firebase, vamos clicar na opção `Agora não`. E pronto, nosso projeto será criado dentro do painel do Firebase! A partir daqui já podemos configurar o banco de dados que será conectado à nossa solução no React Native.

Procure na página pela opção `Database`. Ela está localizada do lado esquerdo da tela. Na página que abrir clique no botão `Criar banco de dados`.



Figura 11.3: Criando um banco de dados em tempo real no Firebase

Escolha a opção `Iniciar o modo teste`. Com essa opção, poderemos escrever e ler no banco de dados sem precisar de nenhuma permissão, o próprio Google nos avisa que qualquer um com a referência para este banco conseguirá fazer leitura e escrita

nele. É claro que, pensando em uma aplicação real que vai para produção de um cliente, esta opção deve ser desativada e as permissões devidamente configuradas. Como estamos somente experimentando o Firebase, vamos de modo de teste mesmo.

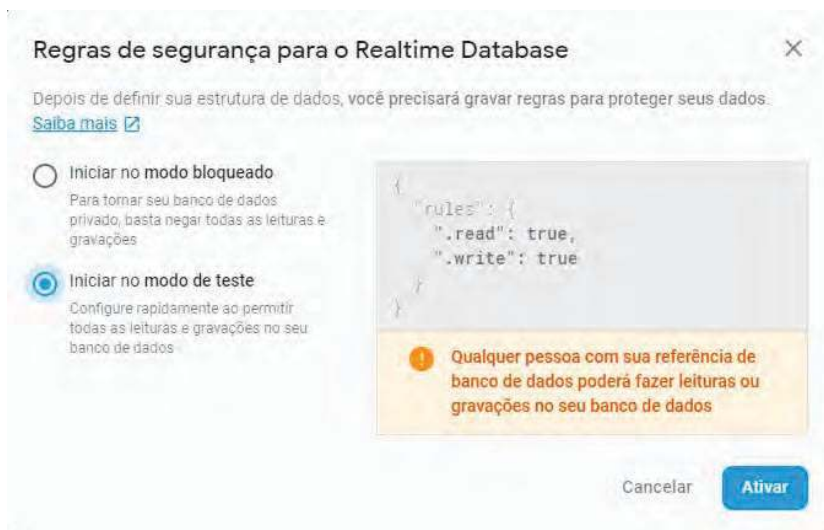


Figura 11.4: Iniciando o modo teste no Firebase

Agora vamos até a página inicial do Firebase e selecionar a opção Adicionar o Firebase ao seu app da web. Dê ao nome ao seu app e copie o código gerado. Vamos utilizá-lo dentro do aplicativo. Precisaremos deste código gerado para conseguir fazer a integração entre o aplicativo e o Firebase. O código que o Firebase vai gerar para você será muito parecido com este:

```
// Your web app's Firebase configuration
var firebaseConfig = {
  apiKey: "",
  authDomain: "",
  databaseURL: "",
  projectId: "",
```

```
storageBucket: "",  
messagingSenderId: "",  
appId: ""  
};  
let app = Firebase.initializeApp(firebaseConfig);  
export const db = app.database();
```

Aqui omitimos os dados da `apiKey` até a `appId`, mas no console do seu Firebase eles devem estar preenchidos. A partir de agora, usaremos estes dados para fazer nossa aplicação se comunicar com o serviço do Google.

11.2 APLICATIVO

Agora que já preparamos o terreno para a nossa aplicação, vale falar um pouco mais sobre qual aplicação construiremos neste capítulo. Já adiantamos que você perceberá que a aplicação não é complexa e seus passos são relativamente simples. E é isso mesmo que queremos passar para você: o Firebase realmente é uma plataforma que nos ajuda muito, desde a instalação até sua utilização.

Os passos que faremos a seguir serão como uma receita de bolo para qualquer operação de leitura e escrita no banco de dados em tempo real do Firebase. Aqui cadastraremos uma lista de linguagens de programação, mas o mesmo servirá para uma lista de pizzas de um restaurante, cadastro de pessoas físicas, carros em uma fábrica e assim por diante. As possibilidades são praticamente infinitas, tudo dependerá da sua necessidade.

Voltando à proposta do nosso aplicativo, ele funcionará da seguinte maneira: na tela inicial vamos criar dois botões, o primeiro levará para uma tela de cadastro enquanto o outro, para

uma tela de listagem. Na tela de cadastro teremos um componente de `TextInput` que receberá a entrada do usuário. Feita essa entrada, colocaremos um componente de `Button` para que a informação seja enviada até o Firebase. Ao terminar, uma caixa aparecerá no aplicativo indicando que tudo ocorreu como esperado.

A tela de listagem será mais simples: apenas buscaremos no Firebase quais são os registros gravados e os exibiremos na tela em formato de lista.

Resumindo, nosso aplicativo deverá seguir este fluxo:

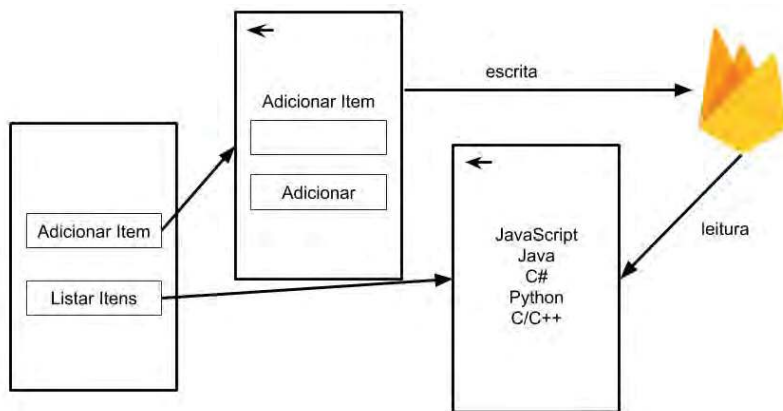


Figura 11.5: Diagrama de fluxo do aplicativo integrado com o Firebase

Vamos colocar a mão na massa.

11.3 INTEGRAÇÃO

Agora que nosso banco de dados está criado, vamos iniciar o nosso aplicativo que vai possibilitar inserir um item através de

componente `TextInput` e depois recuperar todos os itens que foram salvos e listá-los. Usaremos o Firebase para armazenar em tempo real estes dados. Como vimos no diagrama, o aplicativo terá três telas no total. A primeira será onde teremos um botão `Adicionar item` e um botão de `Listar itens`. Clicando no primeiro botão, ele vai para a segunda tela para podermos adicionar o item e, clicando no segundo botão, vai para a terceira tela de listar os itens salvos.

Vamos começar a implementação trazendo a dependência do Firebase ao nosso projeto. Faremos isso através do seguinte comando no terminal: `npm i --save firebase`. Feito isso, volte ao editor de texto onde você está trabalhando com o projeto. Como é necessário armazenar algumas informações do banco de dados criado no Firebase no projeto, vamos criar uma pasta chamada `config` e, dentro dela, colocar um arquivo com o nome de `config.js`. É neste arquivo que colocaremos todo aquele código de integração gerado no Firebase.

```
import Firebase from 'firebase';

var firebaseConfig = {
  apiKey: "AIzaSyAbMFEXYzIMiDbfz05rfc1vvUS5ng3TTcg",
  authDomain: "lista-de-linguagens.firebaseio.com",
  databaseURL: "https://lista-de-linguagens.firebaseio.com",
  projectId: "lista-de-linguagens",
  storageBucket: "",
  messagingSenderId: "916804943247",
  appId: "1:916804943247:web:5f55d97edf11aee8"
};

let app = Firebase.initializeApp(firebaseConfig);
export const db = app.database();
```

Note que, além das configurações que já havíamos visto, trouxemos a dependência do Firebase para esse arquivo e usamos

as funções `initializeApp` para inicializar o banco de dados (por meio das configurações passadas) e então buscamos pelo banco de dados com o método `database` no objeto `app` retornado pelo `initializeApp` . Usaremos este banco de dados em outros arquivos, por isso o exportamos com o nome `db` .

Com as configurações feitas, agora desenvolveremos os componentes de telas para a nossa aplicação. Dentro da pasta `/componentes` , criaremos quatro componentes diferentes:

1. `Inicial.js`
2. `Itens.js`
3. `ListaItens.js`
4. `AdicionaItens.js`

Para sermos capazes de navegar entre as telas, retomaremos o conteúdo do capítulo anterior e criaremos um menu. No arquivo `App.js` , usaremos o método `createStackNavigator` do `React Navigation`.

```
const AppNavigator = createStackNavigator({
  Inicial,
  AdicionaItens,
  ListaItens
},
{
  initialRouteName: 'Inicial'
})

const App = createAppContainer(AppNavigator);
```

Observe que definimos a tela `Inicial` para ser a primeira a ser mostrada no `initialRouteName` . Este atributo diz qual tela deverá ser o ponto de partida. Como esta será a primeira tela visível ao usuário, vamos começar implementando-a. No

componente `Inicial.js` , primeiramente faremos os imports necessários no começo do arquivo. De acordo com o que vimos e o diagrama de fluxos, precisamos da biblioteca `React` e dos componentes `Button` e `View` do `React Native`. Para nos ajudar a aplicar um pouco de estilo, também traremos o `StyleSheet` .

```
import React from 'react';
import { Button, View, StyleSheet } from 'react-native';
```

Com isso, criaremos um componente retornando um componente `View` com os 2 botões que vão levar para as telas de adicionar e listar. Para navegar de uma tela para outra usaremos o atributo `onPress` , que também já exploramos nos capítulos anteriores. O código do nosso componente ficará da seguinte maneira:

```
export default class Inicial extends React.Component {
  render() {
    return (
      <View style={styles.conteudoBtns}>
        <Button
          title="Adicionar item"
          color="green"
          onPress={() =>
            this.props.navigation.navigate('AdicionaItens')
          }
        />
        <Button style={styles.conteudoBtns}
          title="Listar itens"
          color="blue"
          onPress={() =>
            this.props.navigation.navigate('ListaItens')
          }
        />
      </View>
    );
  }
}
```

Note que usamos alguns estilos que ainda não declaramos. Para centralizar os botões na tela vamos colocar um estilo na classe por meio do método `create` do objeto `StyleSheet` do `React Native`.

```
const styles = StyleSheet.create({
  conteudoBtms: {
    flex: 1,
    flexDirection: 'column',
    justifyContent: 'center'
  },
});
```

Agora sim parece que está tudo pronto. Carregue o aplicativo no aparelho usando o Expo que é iniciado com o comando `npm start` no terminal. O resultado deve ser o seguinte:

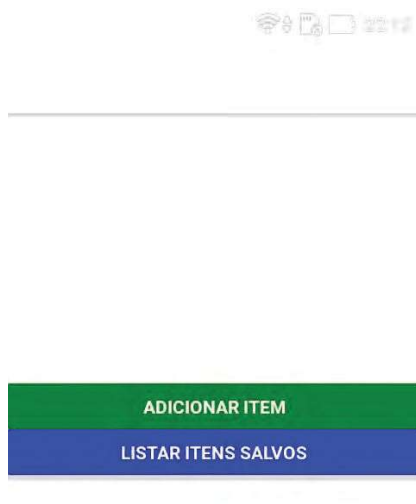


Figura 11.6: Tela inicial do aplicativo usando o Firebase

Parece que já temos nossa tela inicial. Até então não vimos nenhuma grande novidade, somente o arquivo de configuração que criamos para armazenar as informações vindas do Firebase para a aplicação. Partiremos então para a criação do componente da tela `AdicionaItens.js`. Faremos os imports necessários e nesta tela importaremos também o arquivo `config.js` que criamos anteriormente. Além dos componentes de `View`, `Text`, `TextInput` que já vimos, aqui incluiremos duas novidades: o `TouchableHighlight` e o `Alert`. O primeiro componente é o `TouchableHighlight` (<https://facebook.github.io/react->

[native/docs/touchablehighlight/](https://facebook.github.io/react-native/docs/touchablehighlight/)) que nos possibilita criar um contêiner de elementos clicáveis. Uma vez que um elemento está encapsulado neste elemento, ele corresponde ao toque do nosso usuário. O segundo componente é o `Alert` (<https://facebook.github.io/react-native/docs/alert/>), que funciona com a função `alert` dos navegadores. Isso significa que ele mostra uma janela que sobrepõe o conteúdo e informa algo ao usuário. Isso será útil para criarmos nosso alerta de confirmação para quando um item for gravado com sucesso no Firebase.

O componente então começará assim:

```
import React from 'react';

import {
  View,
  Text,
  TouchableHighlight,
  StyleSheet,
  TextInput,
  Alert
} from 'react-native';

import { db } from '../config/config';
```

Definiremos agora uma coleção chamada `itens` no Firebase. É nela que vamos salvar os nossos dados. Como este se trata do nosso primeiro experimento no Firebase, vamos criar uma estrutura simples. O formato do dado salvo será `item: String`. Para fazer isso, vá novamente até o console do Firebase. Entre no banco de dados, procure pela opção `Iniciar coleção` e clique nela. O Firebase então perguntará qual é o caminho da coleção, digite `itens`.

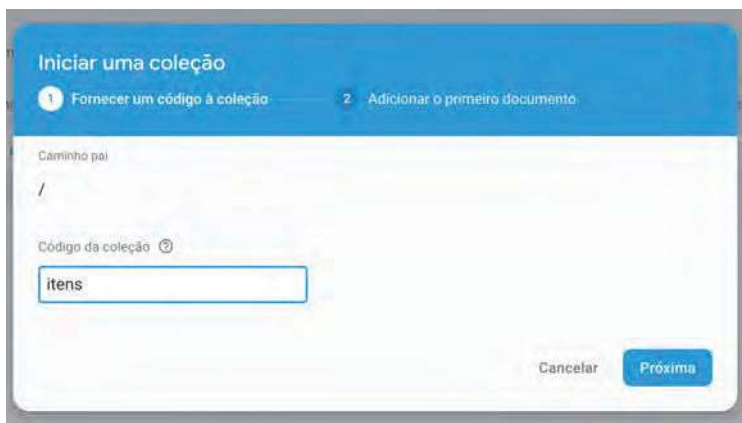


Figura 11.7: Criação de uma nova coleção no banco de dados do Firebase

Ao fazer isso, o Firebase vai perguntar pela estrutura dos documentos que vamos inserir nesta coleção. No espaço indicado como `Campo`, apenas insira `item` e deixe o tipo como `string`. Caso queira inicializar a coleção com algum item em específico, preencha o campo `valor`.

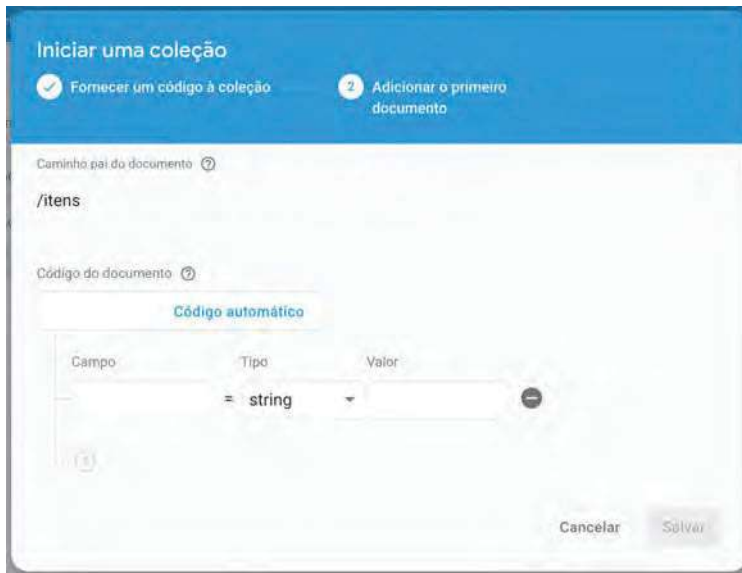


Figura 11.8: Inserindo o primeiro item na coleção do Firebase

Agora que tudo está preparado no back-end, voltaremos ao aplicativo. Continuando o componente `AdicionaItens`, vamos criar uma função que será responsável por gravar os itens que forem gerados na aplicação para o Firebase. Ao final, usaremos o `Alert` para avisar ao usuário de que tudo ocorreu bem. Para tal, utilizaremos o objeto `db` que geramos no `config`:

```
class AdicionaItens extends React.Component {
  state = {
    item: ''
  };

  salvaItem = () => {
    db.ref('/itens').push({
      item: this.state.item
    });

    Alert.alert('Item salvo!');
```

```

    };
}

export default AdicionaItens;

```

Observe que já adiantamos algumas coisas. Iniciamos o componente com o state com item sendo uma String vazia. Este será o local onde o nosso item será gravado. Na função `salvaItem`, esta propriedade do state é enviada para o Firebase. Para o componente ficar completo, precisamos cuidar da parte visual. Para isso, vamos implementar o método `render` que nos retornará um `TextInput` que atualiza o state a cada alteração. Além disso, colocaremos um componente `Button` que quando clicado invocará o método `salvaItem` para acessar o Firebase.

```

render() {
  return (
    <View style={styles.conteudoPrincipal}>
      <Text style={styles.titulo}>
        Adicionar item
      </Text>
      <TextInput
        style={styles.itemInput}
        onChangeText={
          item => { this.setState({item})}
        }
      />
      <TouchableHighlight
        style={styles.btn}
        underlayColor="white"
        onPress={this.salvaItem}
      >
        <Text style={styles.textoBtn}>
          Adicionar
        </Text>
      </TouchableHighlight>
    </View>
  );
}

```

```
}
```

Por fim, vamos inserir alguns estilos em algumas tags para nossa tela ficar mais interessante.

```
const styles = StyleSheet.create({
  conteudoPrincipal: {
    flex: 1,
    padding: 30,
    flexDirection: 'column',
    justifyContent: 'center',
    backgroundColor: 'green'
  },
  titulo: {
    marginBottom: 20,
    fontSize: 25,
    textAlign: 'center'
  },
  itemInput: {
    height: 50,
    padding: 4,
    marginRight: 5,
    fontSize: 23,
    borderWidth: 1,
    borderColor: 'white',
    borderRadius: 8,
    color: 'white'
  },
  textoBtn: {
    fontSize: 18,
    color: '#111',
    alignSelf: 'center'
  },
  btn: {
    height: 45,
    flexDirection: 'row',
    backgroundColor: 'white',
    borderColor: 'white',
    borderWidth: 1,
    borderRadius: 8,
    marginBottom: 10,
    marginTop: 10,
    alignSelf: 'stretch',
    justifyContent: 'center'
  }
});
```

```

    }
  });

```

Agora, antes de criarmos o último componente de tela, precisamos criar um componente que será responsável por exibir cada item salvo no banco de dados. Usaremos a função `map` do Array para fazer isso.

```

import React from 'react';
import { View, Text, StyleSheet } from 'react-native';
import PropTypes from 'prop-types';

class Itens extends React.Component {
  static propTypes = {
    itens: PropTypes.array.isRequired
  };

  render() {
    return (
      <View style={styles.listaItens}>
        {this.props.itens.map(({item}, index) => {
          return (
            <View key={index}>
              <Text style={styles.textItens}>
                {item}
              </Text>
            </View>
          );
        })}
      </View>
    );
  }
}

export default Itens;

const styles = StyleSheet.create({
  listaItens: {
    flex: 1,
    flexDirection: 'column',
    justifyContent: 'space-around'
  },
  textItens: {

```

```

    fontSize: 24,
    fontWeight: 'bold',
    textAlign: 'center'
  }
});

```

Aproveitamos esta oportunidade para apresentar um outro aspecto interessante do React, o PropTypes (<https://reactjs.org/docs/typechecking-with-proptypes.html/>). Na verdade, este recurso estava disponível integralmente no React mas hoje está em um projeto à parte, o prop-types . Esta biblioteca em especial serve para que façamos validação de tipos em componentes React e afins. No nosso componente, vamos utilizá-lo para validar que a propriedade itens que ele receber de um componente deve ser obrigatoriamente um array. Isso nos dá segurança e evita que façamos erros ao passar dados para os componentes.

E agora por último vamos criar o componente de tela ListaItens.js . Além dos imports necessários do React, vamos importar nesta tela o componente Itens.js que criamos por último e o config.js para fazer a conexão com o banco de dados no Firebase.

```

import React, { Component } from 'react';
import { View, Text, StyleSheet } from 'react-native';
import ItensComponente from './ItensComponente';
import { db } from '../config/config';

```

Depois vamos criar um array chamado itens para guardar a lista de itens que forem adicionados na tela AdicionaItens.js . E chamaremos a função componentDidMount() após o componente for montado, como já aprendemos anteriormente.

```

import React from 'react';
import { View, Text, StyleSheet } from 'react-native';

```



```

import Itens from './Itens';

import { db } from '../config/config';

let itensRef = db.ref('/itens');

class ListaItens extends React.Component {
  state = {
    itens: []
  };

  componentDidMount() {
    itensRef.on('value', snapshot => {
      let data = snapshot.val();
      let itens = Object.values(data);
      this.setState({ itens });
    });
  }

  render() {
    return (
      <View style={styles.conteudoPrincipal}>
        { this.state.itens.length > 0
          ? <Itens itens={this.state.itens} />
          : <Text>Não há itens salvos</Text>
        }
      </View>
    );
  }
}

export default ListaItens;

```

Então após o componente ser carregado na tela todos os itens salvos no array `itens` serão mostrados na tela. Caso o array estiver vazio, a mensagem `Não há itens salvos` será mostrada na tela. Por fim, vamos colocar alguns estilos neste componente para mudar a cor de background e centralizar todo seu conteúdo.

```

const styles = StyleSheet.create({
  conteudoPrincipal: {
    flex: 1,

```

```

        justifyContent: 'center',
        backgroundColor: 'blue',
    }
  });

```

Tudo pronto para testar! Na primeira tela do aplicativo clique no botão Adicionar item .



Figura 11.9: Tela de adição de item

Digite alguns itens no `TextInput` e clique no botão Adicionar .



Figura 11.10: Informação de item salvo

Note que ao clicar sobre o botão um pop-up aparece na tela informando que o item foi salvo no banco de dados com sucesso. Depois de salvar, vamos voltar para a página inicial do app e clicar no botão `Lista Itens salvos`.

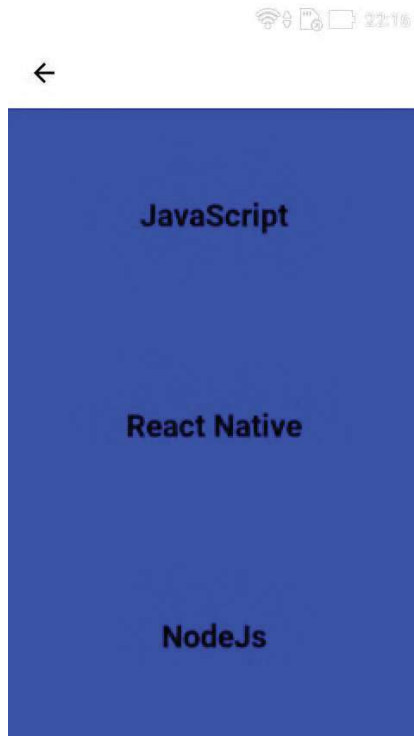


Figura 11.11: Tela de listagem de Itens

Nesta tela são mostrados todos os itens que foram digitados pelo `TextInput` e salvos no banco de dados. Para vocês não pensarem que isso é mágica, vamos voltar à página do Firebase e ir até o banco de dados chamado `lista-de-linguagens` que criamos e observar que informações temos lá.



Figura 11.12: Tela do Firebase mostrando os itens salvos

Note que os itens digitados estão salvos direitinho dentro do banco de dados. É simples assim.

Conclusão

Neste capítulo, nós vimos como integrar um aplicativo criado no React Native com um dos poderosos recursos oferecidos pelo serviço do Firebase. Apesar de o aplicativo ser relativamente simples, contemplamos todo o percurso desde o registro da conta, a criação do aplicativo na plataforma, as opções necessárias e as informações que precisamos transferir para o aplicativo no React Native para que isso funcione. Como dissemos, o Firebase é um recurso que está há muitos anos no mercado e tem soluções completas para qualquer tipo de aplicação. Seja o projeto de pequeno, médio ou grande porte, temos bastante convicção de que a plataforma poderá ser muito útil nos seus projetos no mercado de trabalho.

TRABALHANDO COM HOOKS

Quando pensamos na natureza da tecnologia em si, podemos considerar que a sua parte mais interessante também pode ser considerada sua parte mais frágil. Estamos falando da sua velocidade de transformação. O lado positivo da transformação rápida e contínua é que ela nos fornece uma constante evolução, como um sistema retroalimentado por feedback. Para as falhas, nós encontramos soluções e, para o que não funciona tão bem, encontramos maneiras de fazer melhor. O lado negativo é que isso exige de nós (programadores) também uma constante atualização. Precisamos estar atentos às novidades e depreciações para sempre manter o código atualizado. E isso não é diferente com o React e o React Native.

Durante o ano de 2019 o React sofreu uma grande transformação na versão 16. Muitas melhorias foram feitas na biblioteca e novas funções entraram em cena. De modo geral, foi uma versão que trouxe mudanças significativas. Uma destas mudanças chegou na versão 16.8 e está relacionada ao gerenciamento de estados de componentes dentro da aplicação. Até o momento vimos que os estados são informações que por lei armazenamos apenas em componentes de classe.

Não mais. O React 16.8 trouxe o que chamamos de Hooks (ganchos) e eles mudaram um pouco a maneira como as coisas funcionam no React, seja para web ou mobile. Neste capítulo, vamos entender o que são os Hooks e como eles nos ajudam a administrar os estados dos componentes na nossa aplicação. Nós não abordaremos todos, mas veremos o funcionamento de dois deles que julgamos ser os mais importantes: `useState` e `useReducer`.

12.1 O QUE SÃO OS HOOKS?

Para sermos capazes de entender o impacto dos Hooks no desenvolvimento das aplicações com React Native, primeiro precisamos entender o que eles são e qual foi o motivo para o qual eles foram criados (qual problema eles resolvem?). De maneira bem sucinta, podemos dizer que os Hooks são uma adição ao React que nos permite usar o `state` e outros recursos sem a necessidade de classes, ou seja, apenas usando componentes funcionais (<https://pt-br.reactjs.org/docs/hooks-intro.html>).

De acordo com a documentação oficial, os Hooks foram desenvolvidos por três motivos:

1. **É difícil reutilizar lógica com estado entre componentes:** o React não oferece uma forma prática de "vincular" comportamentos reutilizáveis em um componente. Com os Hooks, conseguimos extrair uma lógica com estado de um componente de uma maneira que possa ser testada independentemente e reutilizada. Além disso, os Hooks nos permitem reutilizar lógica com estado sem mudar a hierarquia de componentes, o que facilita o

compartilhamento de Hooks com vários outros componentes.

2. **Componentes complexos se tornam difíceis de entender:** com o tempo nossos componentes podem ficar complexos e seus métodos (principalmente os que gerenciam o ciclo de vida) acabam ficando sobrecarregados, o que torna o componente pouco manutenível. Com os Hooks, conseguimos dividir um componente em funções menores baseadas em pedaços que são relacionados em vez de forçar uma divisão baseada nos métodos de ciclo de vida.
3. **Classes confundem tanto pessoas quanto máquinas:** os desenvolvedores do React perceberam que, além de deixar o reúso e a organização do código mais difícil, as classes podem ser uma barreira no aprendizado do React. Isso porque, assim como já vimos no decorrer do livro, precisamos entender como funciona o `this`, lembrar de fazer o `bind` em métodos que atuam em eventos (para usar os métodos do React) e assim por diante. Além disso, classes têm se mostrado um grande desafio para ferramentas dos dias de hoje. Por exemplo, classes não minificam muito bem e elas fazem com que *hot reloading* funcione de forma inconsistente e não confiável. Para resolver esses problemas, os Hooks foram criados para permitir que usemos mais das funcionalidades de React sem classes.

O mais legal desta funcionalidade, no entanto, é que ela foi inteiramente desenvolvida de modo que todo o nosso conhecimento prévio do React (ou seja, tudo que vimos até então) continue inteiramente válido. Os Hooks são uma funcionalidade

opcional do React e de maneira alguma substituem as classes. Quer usar componentes funcionais? Ótimo. Quer usar componentes de classe? Ótimo também. A única coisa que não devemos fazer é sair reimplementando todos os componentes de classe para componentes funcionais com Hooks, nem tentar usar os Hooks em componentes de classe.

Em resumo, tudo o que apresentaremos neste capítulo é uma funcionalidade opcional do React para componentes funcionais e que pode ser integrada gradualmente nos projetos React Native. Não somos obrigados a utilizá-la, mas para nos mantermos atualizados com as práticas do mercado, é essencial que ao menos conheçamos seu funcionamento. Nosso objetivo neste capítulo é fazer com que você entenda como e quando usá-los.

12.2 HOOK DE ESTADO (STATE HOOK)

Como dissemos anteriormente, os estados sempre foram uma funcionalidade vinculada somente aos componentes de classe pois precisamos dos métodos de `state` e do construtor para inicializar e posteriormente manipular estas informações dentro do componente. No entanto, com os Hooks ganhamos a possibilidade de trabalhar com os estados também dentro dos componentes de função.

Para entender seu uso, implementaremos mais um componente que terá um objetivo relativamente simples: ele será um contador que acrescenta uma unidade na contagem ao apertamos o botão "incrementar" e que fará o contrário quando apertarmos o botão "decrementar". A ideia deste pequeno componente/aplicativo está representada no diagrama a seguir:

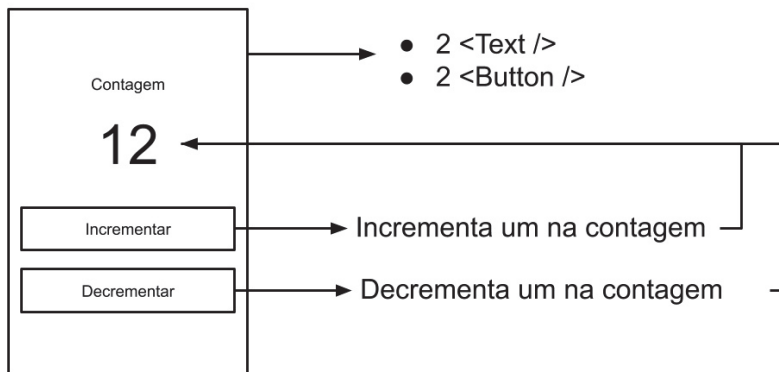


Figura 12.1: Aplicativo de contagem com dois botões

Primeiramente, vá até a pasta `componentes/` do nosso projeto e crie um novo arquivo JavaScript com o nome `Contador.js`. Concluído este passo, vá até o arquivo `App.js` e importe o componente recém-criado. Procure a linha onde está sendo feito o menu de navegação e inclua o acesso ao `Contador`. Faremos isso logo de cara para não nos esquecermos de colocar este experimento com fácil acesso no nosso aparelho. Volte ao arquivo `Contador.js` e vamos fazer a mágica dos Hooks acontecer.

Logo na primeira linha do arquivo já precisamos fazer algo diferente. Para conseguir usar os Hooks do React, temos de importá-lo para dentro do componente. Para isso, faça a seguinte declaração:

```
import React, {useState} from 'react';
import {Button, View, Text} from 'react-native';
```

Note com atenção que o `useState` é o nosso primeiro Hook. É com esta função que conseguiremos manipular o estado dentro

do componente de função. Tendo isso, podemos elaborar o esqueleto do nosso contador usando os componentes de `<Button>` , `<Text>` e `<View>` da API do React Native.

```
const Contador = () => {
  return (
    <View>
      <Button
        title="Incrementar"
        onPress={() => {
          // aumente o valor
        }}
      />
      <Button
        title="Decrementar"
        onPress={() => {
          // diminua o valor
        }}
      />
      <Text>Valor atual: 0</Text>
    </View>
  )
}

export default Contador;
```

Com o esqueleto da aplicação preparado, precisamos refletir como será o fluxo de funcionamento deste componente. Tudo começará com o valor zero, atualmente sendo mostrado na tela dentro do componente `</Text>` . Ao pressionarmos algum dos botões, precisamos incrementar ou decrementar este número e fazer com que o componente renderize seu conteúdo novamente para que o valor atual seja atualizado. Neste ponto, você já deve ter percebido que precisamos usar os estados, afinal, sem eles teríamos que pensar em estratégias mais complexas para conseguir armazenar o valor total (`localStorage` , banco de dados etc.).

Para conseguir usar os estados nesta situação, vamos chamar o

Hook `useState` (<https://reactjs.org/docs/hooks-state.html/>). Em uma linha antes da declaração do JSX de saída, implemente o seguinte código:

```
const [contador, setContador] = useState(0);
```

Uau! Muita coisa aconteceu nesta única linha de código. Para garantir que entendemos tudo o que ela está fazendo por debaixo dos panos, vamos analisá-la passo a passo. Repare que primeiramente a função `setState` é chamada com o valor zero como parâmetro no lado direito da atribuição. O que este código está fazendo é basicamente a inicialização do nosso estado. Nós não havíamos definido que o valor inicial do estado seria zero? Pois bem, é dessa maneira que indicamos esta informação ao React.

Do lado esquerdo da atribuição temos duas variáveis sendo extraídas do `useState`: a primeira é o `contador` e a segunda é o `setContador`. O primeiro parâmetro é o nome do estado que estamos atualizando. Como estamos fazendo uma contagem, achamos que seria prudente chamar o estado desta maneira (mas que fique claro que este nome é totalmente arbitrário). O segundo parâmetro em si se trata de uma função. Usaremos esta função todas as vezes em que for necessário atualizar o valor do estado. Você se lembra de que falamos que nunca alteramos o valor do estado diretamente? De que, em vez disso, precisamos usar - quando estamos trabalhando com classes - o método `this.setState({})`? Com os Hooks, essa regra continua valendo, a única real diferença é que neste exemplo nós passaremos por parâmetro o valor que desejamos atribuir ao estado.

Para que tudo isso faça mais sentido, dê uma olhada em como fica a implementação deste componente usando a variável `contador` e o método `setContador`.

```
import React, {useState} from 'react';
import {Button, View, Text} from 'react-native';

const Contador = () => {
  const [contador, setContador] = useState(0);

  return (
    <View>
      <Button
        title="Incrementar"
        onPress={() =>
          setContador(contador + 1)
        }
      />
      <Button
        title="Decrementar"
        onPress={() =>
          setContador(contador - 1)
        }
      />
      <Text>Valor atual: {contador}</Text>
    </View>
  )
}

export default Contador;
```

Repare bem nos detalhes. Nos botões, onde antes havíamos colocado um comentário, usamos a função `setContador` para atribuir um novo valor ao estado do componente, que neste caso específico foi definido como `contador`. Além disso, no componente `<Text>` conseguimos usar a variável para atualizar o valor na tela. Lembre-se do processo de funcionamento dos estados: toda vez que o valor do estado do componente for alterado, ele renderiza novamente todo o seu conteúdo. Na prática,

isso quer dizer que toda vez que o método `setContador` for invocado o valor do estado será alterado e o componente renderizado novamente.

Mas antes de sairmos testando, vamos colocar um pouco de estilo neste componente. Para esta missão podemos fazer uso do objeto `StyleSheet` do React Native para facilitar o nosso trabalho. Siga a nossa recomendação ou use a sua criatividade e dotes artísticos para estilizar o componente.



Figura 12.2: Aplicativo de contagem usando os Hooks do React

Caso você opte por seguir nossa recomendação, use o código a seguir:

```
const styles = StyleSheet.create({
  conteudo: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center'
  },
  contador: {
    fontSize: 32
  }
});
```

Com as regras declaradas, basta inseri-las nos componentes usando o atributo `style` :

```
const Contador = () => {
  const [contador, setContador] = useState(0);

  return (
    <View style={styles.conteudo}>
      <Text>Contagem</Text>
      <Text style={styles.contador}>{contador}</Text>
      <Button
        title="Incrementar"
        onPress={() => setContador(contador + 1)}
      />
      <Button
        title="Decrementar"
        onPress={() => setContador(contador - 1)}
      />
    </View>
  )
}
```

Experimente o resultado final deste componente. Faça alguns testes e valide se ele realmente incrementa/decrementa o contador em uma unidade. Se tudo der certo, puxe na memória fluxos de funcionamento de aplicativos onde poderíamos fazer uso desta funcionalidade. Depois disso, vá além e tente refatorar os

componentes que já construímos usando classes e estados para componentes de função usando Hooks. Pratique o uso deste Hook e só então prossiga para o próximo tópico.

12.3 HOOK DE REDUCER (REDUCER HOOK)

Se existe uma ferida aberta em todos os desenvolvedores de React (para web) ela tem nome: **Redux**. É muito provável que você já tenha ouvido falar deste termo em algum contexto, mesmo sem saber o que ele significa. O Redux (<https://redux.js.org/>) é uma biblioteca que foi criada com o intuito de nos ajudar a administrar os estados dentro de uma aplicação. Pense nos componentes que fizemos até agora e você verá que todos eles têm algo em comum: administram poucas informações. Quando levamos os estados para uma aplicação complexa, a situação é totalmente diferente. Em vez de lidar com o estado em apenas um componente, precisamos lidar com inúmeros estados de forma compatível com as ações do usuário na aplicação, ou seja, a ação em um componente precisa se comunicar com vários outros componentes diferentes para que todos eles reflitam um estado global da aplicação. E fazer esta "comunicação" entre estados pode se tornar uma grande dor de cabeça.

Foi então que o Redux entrou como uma solução prática, porém pouco trivial. Para programadores React de primeira viagem, tentar usar o Redux como gerenciador de estados em uma aplicação pode ser uma missão bastante complicada, pois o próprio traz consigo várias terminologias e protocolos diferentes. Para nós particularmente foi um parto entender estes conceitos (e até hoje surgem dúvidas quando os usamos).

Com o React 16.8 nós temos à disposição o Hook `useReducer` (<https://reactjs.org/docs/hooks-reference.html#usereducer>), que nada mais é do que o funcionamento da biblioteca Redux como alternativa ao Hook `useState`. Para entender como tudo isso funciona, vamos retomar o componente `Contador` desenvolvido no tópico anterior e vamos refatorá-lo para usar este novo Hook. Caso você queira manter o `Contador.js` intocável para referência, basta criar uma cópia e trabalhar nela. A escolha é sua, combinado?

O primeiro passo será o mais fácil e o único óbvio a partir daqui. Na primeira linha onde importamos o `useState`, substitua por `useReducer`. Agora, na primeira linha dentro da função `Contador`, substitua a linha que usa o método `useState` com este código:

```
const [state, dispatch] = useReducer(reducer, { contador: 0 });
```

E logo de cara já estamos lidando com os três conceitos que são a base do fluxo de funcionamento do controle de estados no Redux. Estamos falando dos conceitos de `state`, `dispatch` e `reducer`. O primeiro lugar para onde vamos olhar é o primeiro argumento do método `useReducer`, o parâmetro `reducer`. Ao contrário do que pode parecer, o `reducer` não é um objeto literal, mas sim uma função. Esta função será invocada toda vez que quisermos atualizar um estado do componente. Como somos nós que sabemos o que o componente deve fazer quando precisar ser atualizado, temos que implementar essa função `reducer` que será utilizada no `useReducer`. Por questões de legibilidade, criaremos essa função acima do `Contador`.

A função `reducer` deve receber dois parâmetros clássicos: o

state e a action . Como você já deve estar imaginando, o state nada mais é do que o próprio estado do componente que estamos manipulando. Mas e a action ? Ela se trata do objeto que nos dirá qual tipo de alteração deve ser feita no estado. Por definição da comunidade, este objeto tem dois atributos: type e payload . O primeiro define qual o tipo de operação que está sendo feita, o segundo, os dados que vão afetar esta operação.

Parece complicado? Imaginamos que sim. Então vamos tentar visualizar desta maneira:

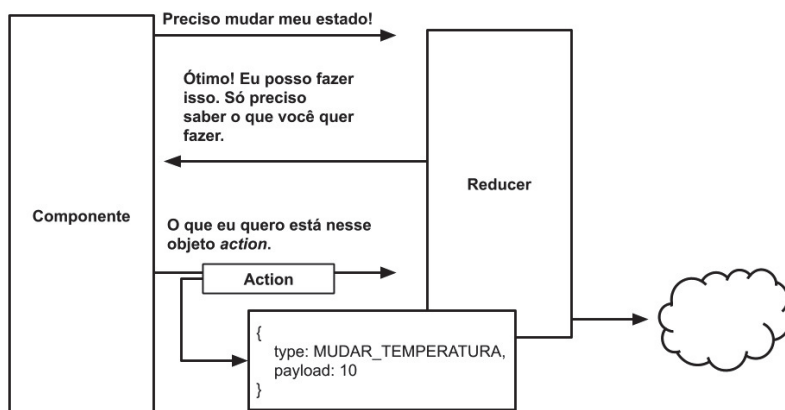


Figura 12.3: "Conversa" entre o componente o reducer

Note que o reducer funciona como o centro de todo conhecimento. Ele é uma única função que vai saber solucionar qualquer tipo de problema envolvendo os estados, sua única condição é saber qual é o tipo de alteração que deve ser feita e qual é o valor desta operação. Ou seja, em vez de distribuirmos a responsabilidade da manutenção do estado entre várias funções e locais diferentes, concentramos tudo no reducer . E como ele

será o responsável por lidar com os vários tipos de alterações de estados, precisamos da `action` para nos dizer o que fazer.

Vamos dar uma olhada em como isso ficaria no caso do componente `Contador`. Considerando que existem dois tipos de alterações que podemos fazer (incrementar ou decrementar) e que estamos sempre aumentando ou diminuindo apenas uma unidade, podemos fazer o seguinte:

```
const reducer = (state, action) => {  
  // state === { contador: 0}  
  // action === { type: string, payload: 1 }  
  switch(action.type) {  
    case 'incrementar':  
      return {...state, contador: state.contador + action.payload  
    };  
    case 'decrementar':  
      return {...state, contador: state.contador - action.payload  
    };  
    default: return state;  
  }  
}
```

Há alguns detalhes nesse código sobre os quais vale a pena refletirmos um pouco mais. Em primeiro lugar, a escolha da estrutura de `switch/case` em vez de `if/else` foi opcional. Geralmente, usamos esta estrutura quando lidamos com reducers, mas a escolha é totalmente sua. O segundo ponto é que o código atende a três casos diferentes: quando há incremento, decremento e quando não há nenhum deles (default). Além disso, todos os casos devolvem o objeto `state`. Isso nos diz que obrigatoriamente a função precisa retornar um `state`. No último caso, apenas devolvemos o `state` que veio por parâmetro de forma intacta. No entanto, para os dois outros temos uma operação que pode parecer esquisita à primeira vista:

```
case 'incrementar':
  return {...state, contador: state.contador + action.payload };
```

O que este código está fazendo? Aqui estamos usando uma artimanha do ES6 para copiar todos os elementos do `state` em um novo objeto e então alterá-lo logo em seguida. Fazemos isso porque nunca devemos alterar o `state` diretamente. Vamos repetir: nunca devemos alterar o objeto do `state` diretamente, em vez disso, criamos uma cópia e então alteramos. Esta é uma prática muito bem difundida e as chances de você encontrar um código parecido com este são bem altas.

Agora já entendemos o que é um `reducer` e uma `action`. Mas e esse tal do `dispatch`? Ele na verdade é o aspecto mais fácil do trio, o `dispatch` é a função que usaremos para chamar um `reducer`. Neste `dispatch`, definimos a `action` que será enviada ao `reducer`. Pensando no exemplo do componente `Contador`, queremos que o estado seja atualizado em duas ocasiões, quando o botão de incrementar ou decrementar foi clicado. Vamos implementar isso.

```
const Contador = () => {
  const [state, dispatch] = useReducer(reducer, { contador: 0 });
  return (
    <View style={styles.conteudo}>
      <Text>Contagem</Text>
      <Text style={styles.contador}>{state.contador}</Text>
      <Button
        title="Incrementar"
        onPress={() => dispatch({type: 'incrementar', payload: 1})}
      />
      <Button
        title="Decrementar"
        onPress={() => dispatch({type: 'decrementar', payload: 1})}
      />
    </View>
  );
};
```

```
    </View>  
  )  
}
```

E pronto! Repare que neste caso nem precisaríamos enviar a informação de `payload`, afinal, eles serão iguais em ambos os casos. Mas como faz parte do padrão, acabamos inserindo no exemplo também. Aliás, os nomes usados também são os adotados pelo mercado. Isso significa que recomendamos que você os utilize para se apropriar da técnica de forma mais próxima à utilizada pelo mercado, no entanto, não quer dizer que você não tem a liberdade de alterar esses nomes. Você tem. Por ventura ao usarmos outros nomes os conceitos podem fazer mais sentido, no entanto, tente manter em mente os nomes adotados pela comunidade.

Conclusão

Neste capítulo, estudamos uma das funcionalidades que foram mais aguardadas no React, os Hooks. Eles nos permitem lidar com os estados em componentes de função, coisa que era simplesmente impossível antes. No entanto, os engenheiros responsáveis pela biblioteca deixaram bem claro que o uso dos estados nos componentes de classe continuará firme e forte. Existem tipos diferentes de Hooks, cada um com uma responsabilidade diferente. Neste livro abordamos dois deles: o `useState` e o `useReducer`. O primeiro funciona como a função `setState` dos componentes de classe enquanto o segundo nos traz a ideia consagrada pelo Redux, o uso de um único lugar para atualizar as diferentes informações de estados dos componentes. Para isso, aprendemos o que é um `reducer`, `action` e `dispatch`.

O FUTURO DO REACT NATIVE

O React Native nasceu de uma experiência bem-sucedida em um Hackathon interno do Facebook em meados de 2013 e desde então se tornou o queridinho dos desenvolvedores e desenvolvedoras mobile ao redor do mundo por sua proposta de desenvolvimento híbrido performático usando JavaScript. Grande parte do seu sucesso se deve à biblioteca React usada em escala mundial para o desenvolvimento de aplicações web. Por trabalhar com a ideia de componentes e potencializar o uso do JavaScript para a criação de componentes em JavaScript integrados ao HTML (visualização) e CSS (estilo), o React se popularizou com grande velocidade, alcançando os seus rivais com uma velocidade inacreditável. Anos depois do seu lançamento, a biblioteca continua mais forte do que nunca.

Durante o nosso percurso neste livro, vimos muita coisa. Saímos da instalação e configuração das dependências do React Native até o desenvolvimento de aplicações que consomem serviços na internet. No meio desse percurso, inevitavelmente tivemos que parar e estudar o funcionamento da biblioteca React, afinal, todos os conceitos trabalhados nela também são usados no React Native. A partir deste ponto vimos o que é o JSX, o que é um

componente, quais são as principais dependências, o que são propriedades e estados, estilos por meio do Flexbox e CSS, navegação de telas, quais são as principais funções do ciclo de vida de um componente, como passar informações de um componente para outro, consumir serviços da internet e atualizar os componentes com as respostas... Até chegarmos à última grande novidade do React, os Hooks. A quantidade de informação apresentada e o conhecimento adquirido não foi pouco. Você deve se sentir orgulhoso!

Realmente esperamos que este livro tenha contribuído de maneira significativa com o seu aprendizado do React Native. Nós nos esforçamos para trazer a informação de maneira divertida e descontraída, sem nos distanciar das informações essenciais e da maneira como o mercado vai cobrar essas informações. Temos esperança de que todo este conhecimento será um grande diferencial profissional no competitivo mercado de desenvolvimento de software que temos não só aqui no Brasil, mas no mundo todo.

Ficou alguma dúvida? Encontrou algo estranho? Precisa de uma mão para entender alguns conceitos? Não tenha receio de entrar em contato nos canais oficiais deste livro. Acreditamos que, somente ajudando uns aos outros, a comunidade de pessoas programadoras brasileiras vai ser fortalecer e se superar.

No entanto, antes de nos despedirmos, tememos ter uma notícia que pode parecer desanimadora, mas não é: o aprendizado desde livro sobre essa tecnologia é só o começo. Com o engajamento da comunidade de desenvolvedores, o envolvimento de grandes empresas e a constante evolução da plataforma e da

própria biblioteca do React Native, temos certeza de que muitas novidades aparecerão no decorrer dos anos e, se você quiser acompanhar tudo isso, terá que continuar a ler, estudar e principalmente, fazer muito código.

Mas afinal, se não fosse para evoluir constantemente, qual seria a graça da tecnologia?

Por fim, agradecemos a sua atenção, dedicação e confiança. Desejamos o melhor para você, na vida pessoal e profissional. Muito obrigado!

Não deixe de acompanhar o site oficial para novidades:

<https://livroreactnative.com.br>

REFERÊNCIAS

Aqui, você encontrará dicas de outras publicações, artigos, cursos, sites e vídeos para auxiliar nos seus estudos e que foram usados para a elaboração deste livro.

Livros

- EISENMAN, Bonnie. *Learning React Native: Building Native Mobile Apps with JavaScript* - 2th Edition. O'Reilly Media, 2017.
- DABIT, Nader. *React Native in Action*. Manning Publications, 2019.

Artigos

- <https://medium.com/react-native-development/a-brief-history-of-react-native-aae11f4ca39>
- <https://tableless.com.br/react-native-construa-aplicacoes-moveis-nativas-com-javascript/>
- <https://blog.rocketseat.com.br/como-organizar-estilos-no-react-native/>
- <https://origamid.com/projetos/flexbox-guia-completo/>
- <https://medium.com/@jamesmarino/getting-started-with-react-native-and-firebase-ab1f396db549>

Links

- React Native - <https://facebook.github.io/react-native/>
- Expo - <https://expo.io/>
- JSX - <https://jsx.github.io/>
- React - <https://facebook.github.io/react/>
- Babel - <https://babeljs.io/>
- JSON - <https://json.org/>
- React Navigation - <https://reactnavigation.org/>

Cursos

- The Complete React Native + Hooks Course (2019 Edition)
- <https://www.udemy.com/course/the-complete-react-native-and-redux-course/>
- React Native Fundamentals -
<https://egghead.io/courses/react-native-fundamentals>
- Simplify React Apps with React Hooks -
<https://egghead.io/courses/simplify-react-apps-with-react-hooks>
- React Native: Desenvolva APPs Nativas para Android e iOS
- <https://www.udemy.com/course/curso-react-native/>