# React Native

# CRIANDO OS PRIMEIROS COMPONENTES

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 essas 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 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 utilizá-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: criaremos uma pasta na raiz do nosso projeto chamada `components`. 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 primeiro componente do zero. Para isso, nas duas primeiras linhas, faremos a importação da biblioteca do React e dos 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 continuar com 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 exibir 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 OlaMundo() {
  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 sempre deve retornar somente um componente (ou div/span). Isso pode parecer óbvio, mas não agrupar os componentes de `text` pode ser fruto de muitas frustrações quando você estiver trabalhando. Se tentarmos fazer algo como o representado a seguir, enfrentaremos problemas, veja:

```
export default function OlaMundo() {
  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 a sua causa. Muitas vezes, ao desenvolver nossos aplicativos, podemos cair nesse erro por esquecer de "encapsular" os componentes e tags. Mas não há nada a temer, basta lembrar que a função deve retornar apenas um elemento. Achamos 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 importarmos 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 './componentes/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, colocaremos a tag `<OlaMundo />` como retorno da função do `App.js`:

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

Como nosso componente não possui conteúdo interno (como o componente `Text`, por exemplo), podemos fechar a tag 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 `<ChapeuzinhoVermellho>` 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 { StatusBar } from 'expo-status-bar';
import React from 'react';
import { StyleSheet, Text, View } from 'react-native';
import OlaMundo from './componentes/OlaMundo';

export default function App() {
  return (
    <View style={styles.container}>
      <OlaMundo />
      <StatusBar style="auto" />
    </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 `npm start` (ou `expo 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 aparecerá na tela do seu dispositivo.


Figura 4.1: Imagem do dispositivo com a frase Olá Mundo!

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. Essas propriedades são informações passadas entre os componentes (normalmente de pai para filho) e são o principal canal de comunicação entre eles. Seu funcionamento não tem nenhum segredo e é bem fácil.

Vamos colocá-las em prática no componente `OlaMundo`, que acabamos de criar. Da forma como ele está agora, 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 `Leitor`, conforme mostra o código a seguir:

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

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 em vez de usar o texto

"Mundo!" da segunda tag `<Text>`, usaremos o valor passado pelo componente pai (no caso o `App`) como propriedade. Para isso, primeiramente precisamos "habilitar" o nosso componente para que ele receba essas 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. Estes atributos representam nada menos que os atributos que colocamos na tag `<OlaMundo>` no passo anterior.

Como definimos o nome da propriedade como `nome`, vamos usar `props.nome` para poder acessá-la. Se tivéssemos usado 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, Leitor!". Legal, não é? 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` , 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 (ou refatorar o componente atual).
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 para 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, pouco falamos sobre estilo. Seja no sentido literal ou figurado, em nenhum momento paramos de fato para entender como podemos aplicar estilos CSS aos nossos componentes e dar vida aos nossos aplicativos. Chegou o momento de aprendermos a fazer isso.

Neste capítulo, veremos 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 este momento, o componente `OlaMundo` (desenvolvido no capítulo anterior) apenas exibe na tela o texto indicado por meio das suas propriedades. 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 pai (ou contêiner), o `App`. Abra o arquivo `App.js` e 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 a especificação Flexbox do CSS, boa parte desse código deve soar 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, note 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. Manteremos 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 sempre uma boa prática separar as folhas

de estilo do código da página, ou seja, separar os arquivos de HTML dos arquivos CSS (e consequentemente dos arquivos JavaScript também). 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 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 com 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, pois prometemos que ao 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 na raiz do 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 isso 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 *camelCase* — 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 `texto`.

```
texto: {

}
```

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:

```
texto: {
  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` e `border-color` e você não está errado. No entanto, lembre-se do que dissemos algumas linhas atrás: para aplicar esses estilos no React Native, precisamos unir os termos e aplicar o formato *camelCase*. No fim, teremos um código muito semelhante a este:

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

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

Agora que temos a nossa página de estilos, vamos importá-la para o componente `OlaMundo` 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.texto}>{this.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 final, o componente deve estar semelhante a este:

```
import React from 'react';
import {View, Text} from 'react-native';
import Estilos from '../estilos/Estilos';

class OlaMundo extends React.Component {
    render() {
     return (
        <View>
          <Text>Olá</Text>
          <Text style={Estilos.texto}>{this.props.nome}!</Text>
        </View>
      )
    }

}

export default OlaMundo;
```

Caso você tenha optado por contruí-lo ou mantê-lo como componente de função, o funcionamento será exatamente o mesmo, a única diferença será na declaração das propriedades (props). Ao salvar tudo e fazer o build da aplicação, veremos que o estilo que colocamos foi aplicado corretamente, conforme imagem:

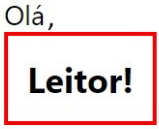


Figura 5.1: Imagem do dispositivo com a frase com os estilos aplicados.

Muito legal, não acha? 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 nos certificarmos 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 da segunda palavra? 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 { StyleSheet } from 'react-native';

// [...] restante do código

const estilos = StyleSheet.create({
  texto: {
    // 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 essas regras de estilo. A ideia é que usemos essa 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.texto}>{this.props.nome}</Text>
```

No final, o código do componente deverá estar assim:

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

class OlaMundo extends React.Component {
  render() {
    return (
```

```
      <View>
        <Text>Olá</Text>
        <Text style={estilos.texto}>{this.props.nome}</Text>
      </View>
    )
  }
}

const estilos = StyleSheet.create({
  texto: {
    fontSize: 18,
    fontWeight: 'bold',
    borderWidth: 2,
    borderColor: 'red',
    padding: 10,
  }
});

export default OlaMundo;
```

**Podemos deixar o estilo no próprio componente (Opcional)**

Recarregue o código no seu aparelho e perceba que o resultado é exatamente o mesmo. A única diferença foi a maneira como lidamos com esse 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 estilo que mostramos anteriormente são técnicas CSS-in-JS, ou seja, usamos o poder que o JSX e o React nos oferecem 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 estilo (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 essas regras (lembra da nossa pasta `estilos/` ?). Vamos mostrar a seguir uma possibilidade de estrutura que é adotada em muitos 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 das 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 o 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'); // Desestrutu
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: metrics.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 deste momento, poderemos utilizar todas as propriedades definidas no arquivo `geral.js`. Essa foi só uma organização fictícia, mas dentro do seu projeto/organização, outras possibilidades são aceitas e bem-vindas.

## 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.

# 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 de prática e de exemplos. Para entender os possíveis casos de utilização da renderização no React Native - e note que o que aprendermos 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 o número 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 (a "velha" receita de bolo para nossos componentes). Aproveitaremos para trazer os componentes nativos de `View`, `Text` e `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/Operador_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 esse 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>O número é par!</Text>
} else {
  return <Text>O número é ímpar!</Text>
}
```

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

```
numero % 2 === 0 ? <Text >O número é par!</Text> : <Text>O 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 "devolvesse" 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 esse 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 => {
  return (
    <View>
      {
        props.numero % 2 == 0
        ? <Text>O número é par!</Text>
        : <Text>O 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';

export default props => {
```

```
  return (
    <View style={estilos.container}>
      {
        props.numero % 2 == 0
        ? <Text style={estilos.texto}>O número é par!</Text>
        : <Text style={estilos.texto}>O número é ímpar!</Text>
      }
    </View>
  )
}

const estilos = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#fff',
    alignItems: 'center',
    justifyContent: 'center',

  },
  texto: {
    fontSize: 18,
    fontWeight: 'bold',
   borderWidth: 2,
   borderColor: 'red', padding: 10
  }
});
```

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

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

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

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


Figura 7.1: Renderização do texto ímpar.

Troque o valor três para um número par e veja o texto `Par` sendo renderizado. Brinque com esse 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 abstraí-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 => {
  return (
    <View style={estilos.container}>
      {validaParOuImpar(props.numero)}
    </View>
  )
}

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

const estilos = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#fff',
    alignItems: 'center',
    justifyContent: 'center',
  },
  texto: {
    fontSize: 18,
    fontWeight: 'bold',
    border: '2px solid red',
    padding: 10
  }
});
```

O resultado final no aplicativo deve ser exatamente o mesmo. Mas, afinal, qual é a vantagem disso? 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 à frente, por exemplo, teremos construção de listas e requisições AJAX, que 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 é a relação que eles têm com os eventos e como utilizá-los para criar os famosos "componentes controlados".

## 8.1 CONHECENDO OS ESTADOS

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

serem usados pelos componentes. Esses dados podem ser strings, números, arrays ou mesmo objetos. A única diferença entre os estados e as propriedades é que as propriedades são somente informações externas recebidas, 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. Ao clicar em botões, preencher formulários, selecionar caixas de seleção etc., 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 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 esses 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 estilos = 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`. Ao fazer isso, poderemos usar os dados do componente da mesma maneira que fizemos com as `props`: basta acessá-los 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 (
      <View style={estilos.container}>
        <TextInput
          style={estilos.input}
          value={this.state.input}
          onChangeText={text => this.setState({ input: text })}
        />
      </View>
    )
  }
}
```

Vamos importar este componente para o `App.js` e 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 isso não te assuste, afinal, estamos exibindo um texto cujo valor inicial é uma string vazia, então esse 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.

# 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 tudo 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 esses 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-los 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 quisermos 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 esses 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 dessas fases possui alguns métodos associados que podem ser sobrescritos nas classes dos componentes. Para termos uma visão holística de quais são esses 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 essas três fases:

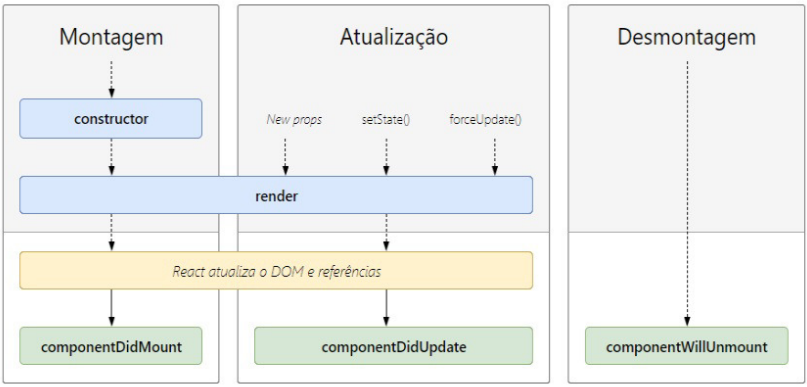


Figura 9.1: Ciclo de vida dos componentes React.

Os métodos que estão na parte cinza da imagem estão no que chamamos de "Fase Render". Eles são puros, sem efeitos colaterais, podem ser pausados, abortados ou reiniciados pelo React. Os métodos que estão na parte inferior branca são os 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 vale a pena conhecermos para dominarmos melhor o uso dos componentes. Vamos entendê-los um pouco mais.

## Montagem

Os métodos associados à fase de montagem 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. Inicializar o estado (state) de um componente, atribuindo um objeto que representa o estado inicial ao `this.state`.
2. Vincular 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>
    )
  }
}

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
  }
});

export default Evento;
```

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 adiante, 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. Esse método é 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 em suas propriedades. Os métodos listados a seguir lidam com esse 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 das propriedades e estados para isso.

O outro método que faz parte desse 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. Agora vamos conhecer os métodos menos usados.

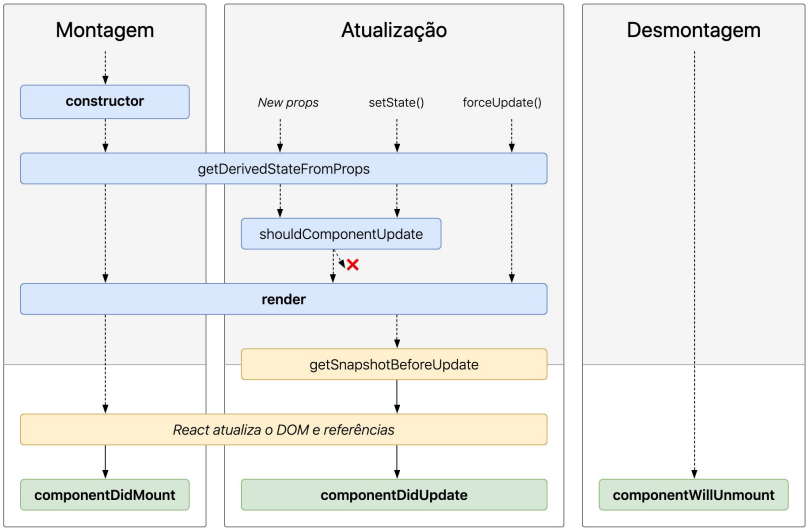


Figura 9.2: Métodos raros de ciclo de vida.

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 que isso aconteça. Este método é chamado logo antes do `render` quando estes novos valores são recebidos pelo componente. Por definição, o retorno dessa 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 o Instagram. No momento em que a pessoa abre o aplicativo/site, o serviço dispara uma série de chamadas concorrentes para a sua API a fim de carregar as informações do usuário: fotos, amigos, curtidas, reações etc. Como esse 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 crie o arquivo `UsuarioGithub.js` na pasta `componentes/`. Este componente será responsável por acessar alguns endpoints da API REST do GitHub e renderizar os dados de um usuário que será dado pelo usuário 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-na
tive';

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` : usada 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) do JavaScript. Essa API está disponível na maior parte das engines e nos permite fazer requisições de modo estupidamente 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 seguinte 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 pronto, 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 esses 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 GitHu
b"
        />
      </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 style={estilos.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 Git
Hub"
          />
        </View>
      </View>
    )
  }

}

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

export default UsuarioGithub;
```

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 se 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! E só para não deixarmos tanta

informação na tela, que tal selecionarmos somente algumas delas para aparecer na tela? Para este caso, vamos pegar somente as informações de nome, número de repositórios e seguidores.

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

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() {
    const {name, public_repos, followers} = this.state.dados;
    return (
      <View style={estilos.container}>
        <Text style={estilos.font30}>Dados do Usuário</Text>
        <Text>Nome: {name}</Text>
        <Text>Repositórios: {public_repos}</Text>
        <Text>Seguidores: {followers}</Text>
        <View>
          <TextInput
            style={estilos.input}
            onChangeText={ usuario => {this.setState({usuario})}
```

Criar versão B e mudar só o abaixo.

```
}
            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>
    )
  }

}

const estilos = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#fff',
    alignItems: 'center',
    justifyContent: 'center',
  },
    input: {
      height: 50,
      width: 300,
      fontSize: 30,
      borderWidth: 1,
      borderColor: 'black',
      margin: 10,
      padding: 5
  },
  font30: {
    fontSize: 30
  }
});

export default UsuarioGithub;
```

Aproveitando que estamos tratanto de situações mais próximas do mundo real, o que acontece se o usuário digitar o nome de um

usuário que não existe? Da forma que nosso componente está agora, simplesmente as informações não apareceriam. Mas que tal fazermos algo mais interessante usando o que aprendemos sobre redenrização condicional? Podemos checar o `state` do componente e se ele estiver vazio, podemos mostrar uma mensagem diferente, algo como "Este usuário não existe".

Para fazer esta alteração, vamos novamente utilizar tudo o que aprendemos até o momento. Pedimos para que você observe com atenção a função `formaDadosDoUsuario` e veja como abstraímos estrategicamente essa pequena lógica para não "poluir" a função `render`. Além disso, aproveitamos essa situação para usar os fragmentos do React. Esses fragmentos nada mais são do que uma forma de encapsular as tags usando uma tag vazia ( `<>` e `</>` ).

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

class UsuarioGithub extends React.Component {

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

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

Criar versão C e mudar só o abaixo.

```
  formaDadosDoUsuario() {
    if (this.state.dados.name !== undefined) {
      const { name, public_repos, followers } = this.state.dados;
      return (
        <>
          <Text style={estilos.font30}>Dados do Usuário</Text>
          <Text>Nome: {name}</Text>
          <Text>Repositórios: {public_repos}</Text>
          <Text>Seguidores: {followers}</Text>
        </>
      )
    } else {
      return (
        <Text style={estilos.font30}>
          Este usuário não existe!
        </Text>
      )
    }
  }

  componentDidMount() {
    this.fetchDados();
  }

  render() {
    return (
      <View style={estilos.container}>
        {this.formaDadosDoUsuario()}
        <View>
          <TextInput
            style={estilos.input}
            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>
      )
    }
}

const estilos = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#fff',
    alignItems: 'center',
    justifyContent: 'center',
  },
  input: {
    height: 50,
    width: 300,
    fontSize: 30,
    borderWidth: 1,
    borderColor: 'black',
    margin: 10,
    padding: 5
  },
  font30: {
    fontSize: 30
  }
});

export default UsuarioGithub;
```

Experimente fazer uma série de buscas com outros usuários, inclusive aqueles que não existem e veja os resultados 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 em que 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 `5.9.4`, 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 @react-navigation/native
```

Em seguida, utilize o comando abaixo para que o Expo organize tudo:

```
npm install react-native-screens @react-native-community/masked-view
```

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". Este tipo de navegação é o clássico navegador lateral. Ele aparece toda vez que arrastamos a tela da esquerda para a direita.

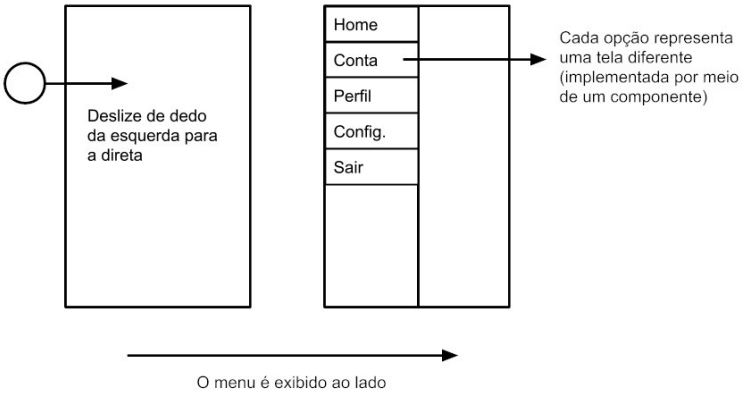


Figura 10.1: Diagrama do menu lateral.

Para poder utilizá-los, vamos precisar importá-lo para o projeto:

```
npm install --save @react-navigation/drawer
```

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:

```
import { createDrawerNavigator } from '@react-navigation/drawer';
import { NavigationContainer } from '@react-navigation/native';
```

O método `createDrawerNavigator` será o responsável por criar toda a estrutura do menu. Os detalhes de todas as outras podem ser encontrados na documentação (https://reactnavigation.org/docs/en/drawer-navigator.html). O segundo objeto que importamos é um componente que servirá como contêiner para os menus que quisermos utilizar na aplicação. Dito isso, o próximo passo será instanciar o menu e armazená-lo em uma variável. Para seguir o padrão sugerido pela documentação, chamaremos essa variável de `Drawer`.

```
const Drawer = createAppContainer();
```

Agora já podemos utilizar esta estrutura para criar o menu. Como dissemos, tudo deverá ficar encapsulado dentro do componente `<NavigationContainer>`. O menu em si será construído através do componente `<Drawer.Navigator>` e para cada item dentro deste menu, usaremos o componente `<Drawer.Screen>`. Vamos aproveitar os componentes `Evento` e `UsuarioGihub`, que construímos nos capítulos anteriores.

O componente deverá ficar assim:

```
import React from 'react';
import Evento from './componentes/Evento';
import UsuarioGithub from './componentes/UsuarioGithub';
import { createDrawerNavigator } from '@react-navigation/drawer';
import { NavigationContainer } from '@react-navigation/native';

const Drawer = createDrawerNavigator();
```

```
export default function App() {
  return (
    <NavigationContainer>
      <Drawer.Navigator>
        <Drawer.Screen name="Evento" component={Evento} />
        <Drawer.Screen name="Github" component={UsuarioGithub} />
      </Drawer.Navigator>
    </NavigationContainer>
  );
}
```

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.

Não se esqueça de que não será possível testar esse menu na visualização web! 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 esse tipo de menu. E então, da mesma maneira que um navegador web "empilha" as páginas 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*). Nessa 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.

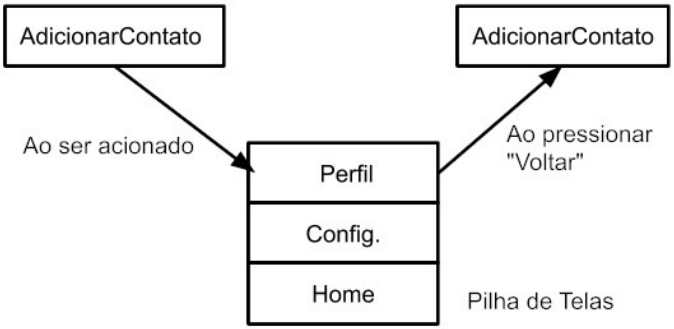


Figura 10.3: Exemplo de pilha de telas.

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. O primeiro passo será importar esse tipo de menu para o projeto:

```
npm install --save @react-navigation/stack
```

Feito isso, entre novamente no `App.js` e certifique-se de que os imports necessários estão lá. Desta vez, além do `NavigationContainer`, usaremos o método `createStackNavigator` da biblioteca e os componentes `Button` e `View` do React Native.

```
import React from 'react';
import { Button, View, StyleSheet } from 'react-native';
import { NavigationContainer } from '@react-navigation/native';
import { createStackNavigator } from '@react-navigation/stack';
```

Muito bem. Agora precisamos improvisar e aproveitar essa oportunidade para mostrar como criar múltiplas telas dentro de um único arquivo. Como esse menu depende de botões e links que

redirecionam para telas diferentes e, até o momento, não construímos nenhum componente desse tipo, criaremos um componente de função com dois botões: um que leva ao componente de `Evento` e outro para o componente de `UsuarioGithub`. Para que isso funcione adequadamente, faremos com que este componente receba o objeto `navigation` como `props` para que então usemos a função `navigate`. Em tempo de execução, o React Navigation fornecerá esse objeto para o componente.

```
function TelaInicial({ navigation }) {
  return (
    <View style={estilos.container}>
      <Button
        title="Evento"
        onPress={() => navigation.navigate('Evento')}
      />
      <Button
        title="Usuário GitHub"
        onPress={() => navigation.navigate('Github')}
      />
    </View>
  );
}
```

Com o componente que servirá de tela inicial pronto, usaremos a função `createStackNavigator` para criar a estrutura de menu. Novamente seguindo o padrão, chamaremos a variável que armazenará essa estrutura de `Stack`.

```
const Stack = createStackNavigator();
```

Por fim, seguiremos uma estrutura muito semelhante a do exemplo anterior.

```
export default function App() {
  return (
    <NavigationContainer>
```

```
      <Stack.Navigator>
        <Stack.Screen name="Home" component={TelaInicial} />
        <Stack.Screen name="Evento" component={Evento} />
        <Stack.Screen name="Github" component={UsuarioGithub} />
        </Stack.Navigator>
    </NavigationContainer>
  );
}
```

No final, o arquivo deverá estar assim:

```
import React from 'react';
import { Button, View, StyleSheet } from 'react-native';
import Evento from './componentes/Evento';
import UsuarioGithub from './componentes/UsuarioGithub';
import { createStackNavigator } from '@react-navigation/stack';
import { NavigationContainer } from '@react-navigation/native';

function TelaInicial({ navigation }) {
  return (
    <View style={estilos.container}>
      <Button
        title="Evento"
        onPress={() => navigation.navigate('Evento')}
      />
      <Button
        style={estilos.botao}
        title="Usuário GitHub"
        onPress={() => navigation.navigate('Github')}
      />
    </View>
  );
}

const Stack = createStackNavigator();

export default function App() {
  return (
    <NavigationContainer>
      <Stack.Navigator>
        <Stack.Screen name="Home" component={TelaInicial} />
        <Stack.Screen name="Evento" component={Evento} />
        <Stack.Screen name="Github" component={UsuarioGithub} />
        </Stack.Navigator>
```

```
    </NavigationContainer>
  );
}

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

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.

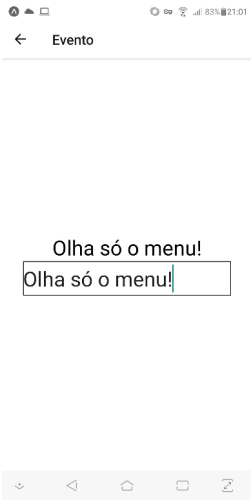


Figura 10.4: Exemplo de navegação por botões.

## 10.4 NAVEGAÇÃO POR ABAS

Por fim, vamos 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.

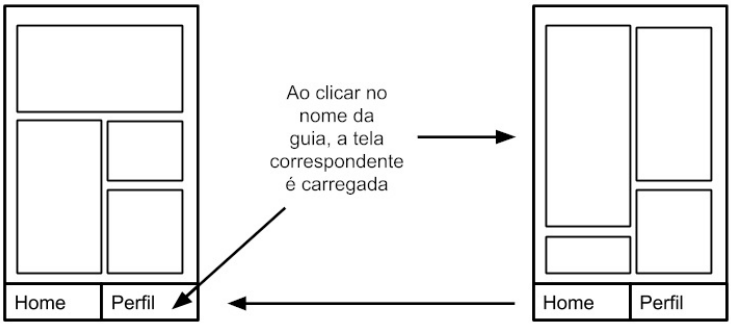


Figura 10.5: Funcionamento da navegação por abas.

A configuração desse tipo de navegação é muito semelhante ao que já fizemos nos dois exemplos anteriores. Primeiramente, precisamos importar esse tipo de menu para dentro do projeto:

```
npm install @react-navigation/bottom-tabs
```

Com esse passo garantido, o método `createBottomTabNavigator` precisa ser importado para dentro do componente junto ao componente `NavigationContainer`, que usamos nos exemplos anteriores.

```
import { NavigationContainer } from '@react-navigation/native';
import { createBottomTabNavigator } from '@react-navigation/botto
m-tabs';
```

Acreditamos que agora você já deve imaginar os próximos passos. É isso mesmo, logo em seguida armazenamos em uma

variável a estrutura criada no método `createBottomTabNavigator`. Seguindo o padrão recomendado da documentação, chamaremos essa variável de `Tab`.

```
const Tab = createBottomTabNavigator();
```

Agora basta configurar quais telas você quer mostrar no seu aplicativo. Como se trata de um menu que ficará fixo na parte inferior do conteúdo, não recomendamos colocar mais do que duas ou três telas.

E pronto, configuramos o mínimo necessário para que tudo funcione. Agora vamos testá-lo. Para garantir que tudo funcione como esperado, confira o seu código com o mostrado a seguir:

```
import React from 'react';
import Evento from './componentes/Evento';
import UsuarioGithub from './componentes/UsuarioGithub';
import { createBottomTabNavigator } from '@react-navigation/botto
m-tabs';
import { NavigationContainer } from '@react-navigation/native';

const Tab = createBottomTabNavigator();

export default function App() {
  return (
    <NavigationContainer>
      <Tab.Navigator>
        <Tab.Screen name="Evento" component={Evento} />
        <Tab.Screen name="Github" component={UsuarioGithub} />
      </Tab.Navigator>
    </NavigationContainer>
  );
}
```


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 esse 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 navegação por abas. Em cada um desses 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 essa tecnologia, afinal, ela oferece uma gama de serviços que podem ser utilizados dentro da sua aplicação. Esses serviços são separados em quatro grandes categorias, sendo elas: Analytics, Develop, Grow e Earn. Neste capítulo, vamos focar 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 serão mais do que suficientes para os nossos experimentos no livro e para os seus primeiros aplicativos do React Native em produção.

O primeiro passo para podermos desfrutar desses 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, o que pode ser feito gratuitamente. Feito isso, volte para 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.

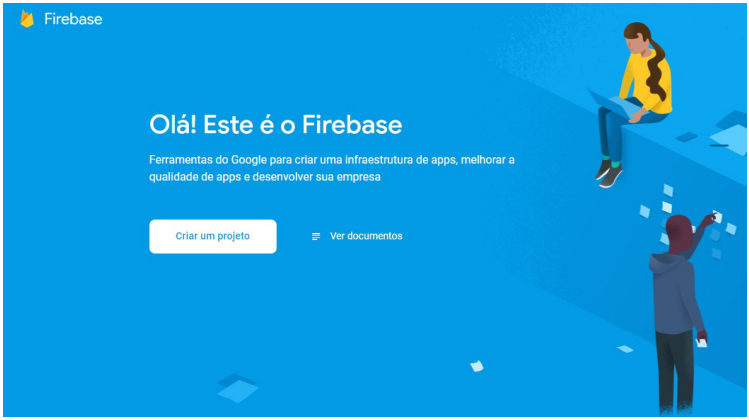


Figura 11.1: Criando um projeto no painel do Firebase.

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`.

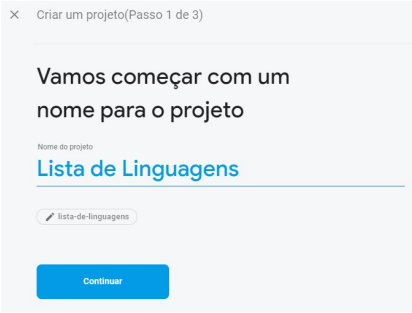


Figura 11.2: Nomeando o projeto no painel do Firebase.

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

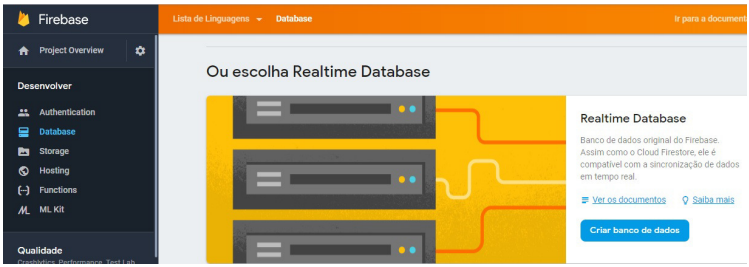


Figura 11.3: Criando um banco de dados em tempo real no Firebase.

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 a produção de um cliente, essa 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 volte até a página `Visão geral do Projeto` usando o menu lateral e selecione a opção `Adicionar o Firebase ao seu app da Web` . Dê um 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:

```
  <!-- The core Firebase JS SDK is always required and must be li
sted first -->
  <script src="https://www.gstatic.com/firebasejs/8.7.1/firebase-
app.js"></script>
  // Your web app's Firebase configuration
  var firebaseConfig = {
    apiKey: "AIzaSyCt8NmoKXdPKMYht8V_gs0AVqMNvZ2KVag111",
    authDomain: "lista-de-linguagens-750971.firebaseapp.com",
    databaseURL: "https://lista-de-linguagens-750971-default-rtdb
.firebaseio.com",
```

```
    projectId: "lista-de-linguagens-750971",
    storageBucket: "lista-de-linguagens-750971.appspot.com",
    messagingSenderId: "5276139692026",
    appId: "1:527613969206:web:6ec65a82543b1f08274a997"
  };
  // Initialize Firebase
  firebase.initializeApp(firebaseConfig);
```

A partir de agora, usaremos esses 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 a 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:

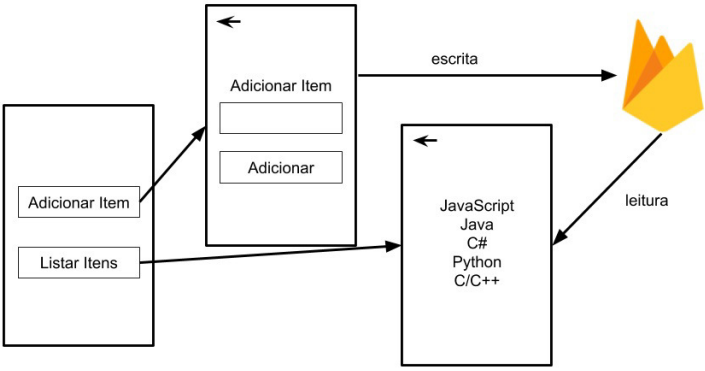


Figura 11.5: Diagrama de fluxo do aplicativo integrado com o Firebase.

Vamos colocar a mão na massa. Você pode utilizar o mesmo projeto que estava utilizando até agora, mas recomendamos criar um novo.

## 11.3 INTEGRAÇÃO

Agora que nosso banco de dados está criado, vamos iniciar o nosso aplicativo, que vai possibilitar inserir um item através do componente `TextInput` e depois recuperar todos os itens que

foram salvos e listá-los. Usaremos o Firebase para armazenar em tempo real esses 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 nos levará para a segunda tela para podermos adicionar o item e, clicando no segundo botão, nos levará 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: "AIzaSyCt8NmoKXdPKMYht8V_gs0AVqMNvZ2KVag111",
  authDomain: "lista-de-linguagens-750971.firebaseapp.com",
  databaseURL: "https://lista-de-linguagens-750971-default-rtdb.f
irebaseio.com",
  projectId: "lista-de-linguagens-750971",
  storageBucket: "lista-de-linguagens-750971.appspot.com",
  messagingSenderId: "5276139692026",
  appId: "1:527613969206:web:6ec65a82543b1f08274a997"
};

const 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, usamos a

função `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 três componentes diferentes:

1. Itens.js
2. ListarItens.js
3. AdicionaItens.js

Usaremos esses componentes muito em breve. Por ora, vamos apenas deixar o "terreno preparado". Para evitar problemas em tempo de execução, esses arquivos não podem estar vazios. Por isso, em cada um deles, coloque o seguinte código.

```
import React from 'react';

export default () => {
  <></>
}
```

Com as telas prontas para serem preenchidas, podemos estruturar o menu. Para sermos capazes de navegar entre as telas, retomaremos o conteúdo do capítulo anterior. No arquivo `App.js`, usaremos o método `createStackNavigator` do React Navigation. Para organizar essas telas, usaremos a mesma estratégia e criaremos um componente interno chamado `Inicial`, que servirá como contêiner para as telas de adicionar e listar.

```
import React from 'react';
import { StyleSheet, Text, View, Button } from 'react-native';
import { NavigationContainer } from '@react-navigation/native';
import { createStackNavigator } from '@react-navigation/stack';

import AdicionaItens from './componentes/AdicionaItens';
import ListarItens from './componentes/ListarItens';

function Inicial({ navigation }) {
  return (
    <View style={estilos.container}>
      <View style={estilos.botao}>
        <Button
          title="Adicionar"
          onPress={() => navigation.navigate('Adicionar')}
        />
      </View>
      <View style={estilos.botao}>
        <Button
          title="Listar"
          onPress={() => navigation.navigate('Listar')}
        />
      </View>
    </View>
  );
}

const Stack = createStackNavigator();

export default function App() {
  return (
    <NavigationContainer>
      <Stack.Navigator>
        <Stack.Screen name="Inicial" component={Inicial} />
        <Stack.Screen name="Adicionar" component={AdicionaItens}
/>
        <Stack.Screen name="Listar" component={ListarItens} />
        </Stack.Navigator>
    </NavigationContainer>
  );
}

const estilos = StyleSheet.create({
  container: {
    flex: 1,
```

```
    alignItems: 'center',
    justifyContent: 'center',
  },
  botao: {
    margin: 10
  }
});
```

Apesar do código relativamente extenso, até aqui não temos novidades. Carregue o aplicativo no aparelho usando o Expo, que é iniciado com o comando `npm start` no terminal. O resultado deve ser o seguinte:

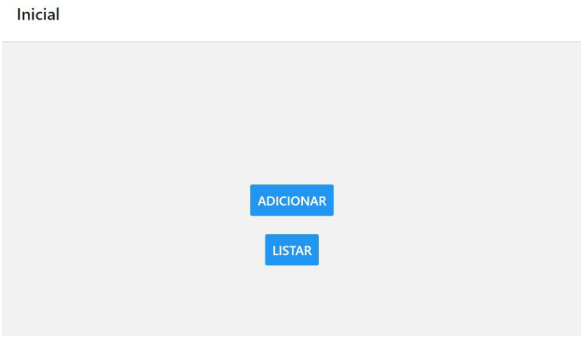


Figura 11.6: Tela inicial do aplicativo usando o Firebase.

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 `TouchableHightlight` (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 componente, 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 (lembrando que este alerta não funciona na versão web).

O componente então começará assim:

```
import React from 'react';

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

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

Continuando o componente `AdicionaItens`, vamos criar uma função que será responsável por gravar os itens que forem gerados na aplicação no Firebase. Ao final, usaremos o `Alert` para avisar ao usuário 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. Criamos um estado chamado `item`, que será o responsável por armazenar aquilo que o usuário escreverá na tela. Como no início da aplicação o usuário ainda não escreveu nada, esse estado precisa começar como sendo vazio. Fazemos isso iniciando ele como uma string vazia. Quando ela for preenchida pelo usuário, a função `salvaItem` enviará este estado lá 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, por sua vez, 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={estilos.conteudoPrincipal}>
      <Text style={estilos.titulo}>
        Adicionar item
      </Text>
      <TextInput
        style={estilos.itemInput}
        onChangeText={
          item => { this.setState({item})}
        }
      />
      <TouchableHighlight
        style={estilos.btn}
        underlayColor="white"
        onPress={this.salvaItem}
```

```
      >
        <Text style={estilos.textoBtn}>
          Adicionar
        </Text>
      </TouchableHighlight>
    </View>
  );
}
```

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

```
const estilos = StyleSheet.create({
  container: {
    flex: 1,
    padding: 30,
    flexDirection: 'column',
    justifyContent: 'center',
  },
  titulo: {
    marginBottom: 20,
    fontSize: 25,
    textAlign: 'center'
  },
  itemInput: {
    height: 50,
    padding: 4,
    marginRight: 5,
    fontSize: 23,
    borderWidth: 1,
    borderColor: 'white',
    borderRadius: 8,
    color: 'black',
    backgroundColor: 'white',
  },
  textoBtn: {
    fontSize: 18,
    alignSelf: 'center',
    color: 'white',
    fontWeight: 'bold'
  },
  btn: {
    height: 45,
    flexDirection: 'row',
```

```
    backgroundColor: '#2196F3',
    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. Chamaremos este componente de `Itens`.

```
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 essa oportunidade para apresentar um outro aspecto interessante do React, o PropTypes (https://reactjs.org/docs/typechecking-with-proptypes.html/). Na verdade, esse 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 a validação de tipos em componentes React e afins. No nosso componente, vamos utilizá-lo para determinar que a propriedade `itens`, que ele receber de um componente, deve ser obrigatoriamente um array. Isso nos dá segurança e evita que cometamos erros ao passar dados para os componentes.

E agora por último vamos criar o componente de tela `ListarItens.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 foram adicionados na tela `AdicionaItens.js` e

chamaremos a função `componentDidMount()` depois que 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();
      if(data) {
        let itens = Object.values(data);
        this.setState({ itens });
      }
    });
  }

  render() {
    return (
      <View style={estilos.container}>
        { 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 esteja

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 o seu conteúdo.

```
const estilos = StyleSheet.create({
  conteudoPrincipal: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center'
  }
});
```

Estamos com tudo pronto para testar! Na primeira tela do aplicativo, clique no botão `Adicionar item`. Digite alguns itens no `TextInput` e clique no botão `Adicionar`.

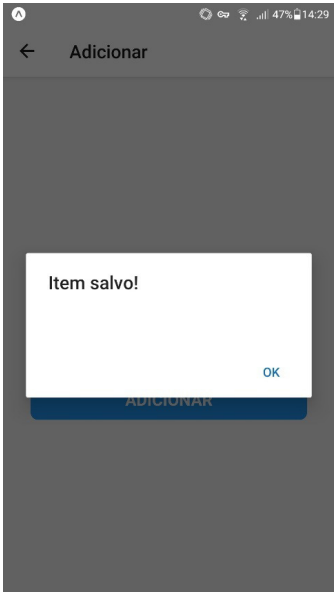


Figura 11.7: Informação de item salvo.

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 `Listar Itens salvos` .


Figura 11.8: Tela de listagem de Itens.

Nessa 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 quais informações temos lá.


Figura 11.9: 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 um 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 a 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 também exige de nós (programadores) 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 dessas 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 serem 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 por qual motivo eles foram criados (qual é o problema que 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, as classes têm se mostrado como 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 dessa funcionalidade, no entanto, é que ela foi inteiramente desenvolvida de modo que todo o nosso conhecimento prévio do React (ou seja, tudo o 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-la.

## 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 essas 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 apertarmos 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:

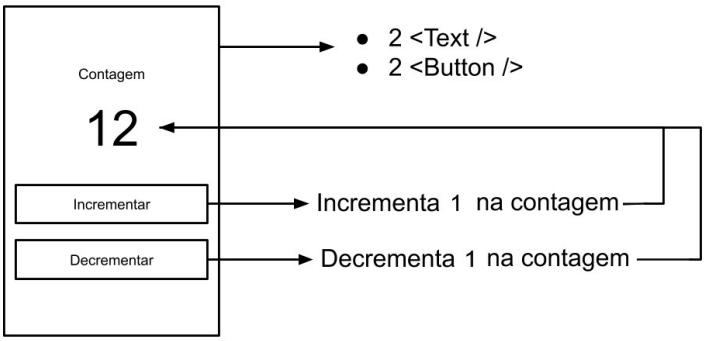


Figura 12.1: Aplicativo de contagem com dois botões.

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 esse 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 a função `useState` é o nosso primeiro

Hook. É com essa 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 desse 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 nessa única linha de código. Para garantir que entendemos tudo o que ela está fazendo internamente, 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 esse 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 essa 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 dessa maneira (mas que fique claro que esse nome é totalmente arbitrário). O segundo parâmetro em si se trata de uma função. Usaremos essa 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 desse 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 nesse componente. Para essa 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 desse 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 dessa 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 desse 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 essa "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 e programadoras React de primeira viagem, tentar usar o Redux como gerenciador de estados em uma aplicação pode ser uma missão bastante complicada, pois ele traz consigo várias terminologias e protocolos diferentes. Para nós particularmente foi um parto entender esses 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

`setState`. Para entender como tudo isso funciona, vamos retomar o componente `Contador`, desenvolvido no tópico anterior, e vamos refatorá-lo para usar esse 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-o por `useReducer`. Agora, na primeira linha dentro da função `Contador`, substitua a linha que usa o método `useState` por este código:

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

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 o tipo de operação que está sendo feita, o segundo, os dados que vão afetar essa operação.

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

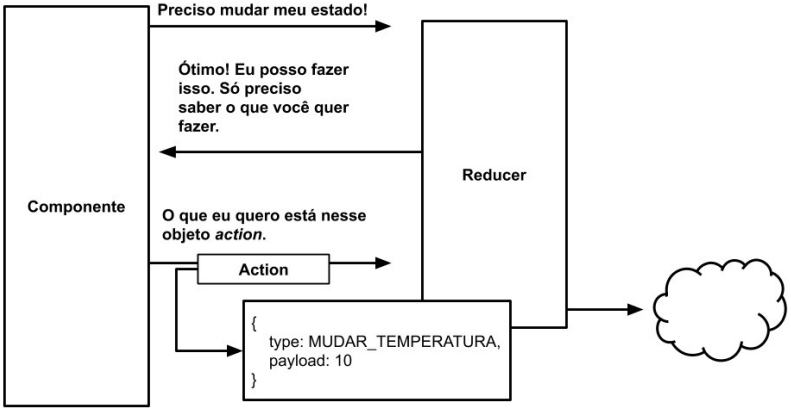


Figura 12.3: "Conversa" entre o componente e o reducer.

Note que o `reducer` funciona como o centro de todo o 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 essa estrutura quando lidamos com funções `reducer`, mas a escolha é totalmente sua. O segundo ponto é que o código atende a três casos diferentes: quando há incremento, quando há 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 a alteramos. Esta é uma prática muito bem difundida e as chances de você encontrar um código parecido com este são bem altas, já que é o método seguro de alterar o `state`.

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`. Nesse `dispatch`, definimos a `action` que será enviada ao `reducer`. Pensando no exemplo do componente `Contador`, queremos que o estado seja atualizado quando o botão de incrementar ou decrementar for 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>
  )
}
```

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 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`, uma `action` e um `dispatch`.