

José Rubens Rodrigues

# Programação para **dispositivos móveis**

**Dados Internacionais de Catalogação na Publicação (CIP)**  
**(Simone M. P. Vieira – CRB 8ª/4771)**

---

Rodrigues, José Rubens

Programação para dispositivos móveis / José Rubens Rodrigues. –  
São Paulo: Editora Senac São Paulo, 2022. (Série Universitária)

Bibliografia.

e-ISBN 978-85-396-3183-4 (ePub/2022)

e-ISBN 978-85-396-3184-1 (PDF/2022)

1. Desenvolvimento de sistemas 2. Linguagem de programação  
3. Programação orientada a objetos 4. Software (desenvolvimento)  
I. Título. II. Série.

22-1449t

CDD – 003

005.13

BISAC COM051210

COM051000

---

**Índice para catálogo sistemático**  
**1. Desenvolvimento de sistemas 003**  
**2. Linguagem de programação 005.13**

# PROGRAMAÇÃO PARA DISPOSITIVOS MÓVEIS

José Rubens Rodrigues





## Administração Regional do Senac no Estado de São Paulo

### Presidente do Conselho Regional

Abram Szajman

### Diretor do Departamento Regional

Luiz Francisco de A. Salgado

### Superintendente Universitário e de Desenvolvimento

Luiz Carlos Dourado

## Editora Senac São Paulo

### Conselho Editorial

Luiz Francisco de A. Salgado

Luiz Carlos Dourado

Darcio Sayad Maia

Lucila Mara Sbrana Sciotti

Luís Américo Tousi Botelho

### Gerente/Publisher

Luís Américo Tousi Botelho

### Coordenação Editorial/Prospecção

Dolores Crisci Manzano

Ricardo Diana

### Administrativo

grupoedsadministrativo@sp.senac.br

### Comercial

comercial@editorasenacsp.com.br

### Acompanhamento Pedagógico

Mônica Rodrigues dos Santos

### Designer Educacional

Hágara Rosa da Cunha Araujo

### Revisão Técnica

Gustavo Moreira Calixto

### Preparação e Revisão de Texto

Juliana Ramos Gonçalves

### Projeto Gráfico

Alexandre Lemes da Silva

Emília Correa Abreu

### Capa

Antonio Carlos De Angelis

### Editoração Eletrônica

Sidney Foot Gomes

### Ilustrações

Sidney Foot Gomes

### Imagens

Adobe Stock Photos

### E-book

Rodolfo Santana

Proibida a reprodução sem autorização expressa.  
Todos os direitos desta edição reservados à

Editora Senac São Paulo  
Rua 24 de Maio, 208 – 3º andar  
Centro – CEP 01041-000 – São Paulo – SP  
Caixa Postal 1120 – CEP 01032-970 – São Paulo – SP  
Tel. (11) 2187-4450 – Fax (11) 2187-4486  
E-mail: editora@sp.senac.br  
Home page: <http://www.livrariasenac.com.br>

© Editora Senac São Paulo, 2022

# Sumário

## Capítulo 1

### Orientação a objetos por JavaScript, 7

- 1 A linguagem JavaScript, 8
  - 2 Objetos e propriedades, 10
  - 3 Classes, 13
  - 4 Herança, 16
  - 5 Reescrita de métodos (override), 18
- Considerações finais, 19
- Referências, 20

## Capítulo 2

### Desenvolvimento nativo e híbrido, 21

- 1 A linguagem nativa, 22
  - 2 A linguagem híbrida, 25
  - 3 Linguagem nativa ou híbrida?, 26
  - 4 React Native, 27
  - 5 React Native CLI × Expo, 30
- Considerações finais, 32
- Referências, 33

## Capítulo 3

### Instalando o React Native, 35

- 1 Instalação do React Native com o Expo CLI e o Yarn, 36
  - 2 Hello World, 40
  - 3 Rodando o projeto na web, 41
  - 4 Rodando o projeto no device, 42
  - 5 Rodando o projeto nos emuladores, 43
  - 6 Editor de código, 44
  - 7 Ferramentas de desenvolvimento, 45
- Considerações finais, 48
- Referências, 49

## Capítulo 4

### O Flexbox e estilos, 51

- 1 Estrutura de um componente, 52
  - 2 Conhecendo o JSX, 54
  - 3 Text e aplicação de estilos com o StyleSheet, 55
  - 4 O Flexbox, 58
- Considerações finais, 66
- Referências, 67

## Capítulo 5

### Componentes básicos do React Native, 69

- 1 Utilizando a *props*, 70
  - 2 React Hooks, 72
  - 3 Componentes do React Native, 77
- Considerações finais, 84
- Referências, 84

## Capítulo 6

### Trabalhando com listas e componentes, 85

- 1 *PropTypes*, 86
  - 2 *FlatList* e *SectionList*, 88
  - 3 React-Navigation, 94
  - 4 Consumindo fonte de dados, 99
- Considerações finais, 100
- Referências, 101

## Capítulo 7

### Trabalhando com geocoordenadas, mapas e notificações, 103

- 1 Como buscar as informações das geocoordenadas, 104
  - 2 Trabalhando com mapas, 108
  - 3 Utilizando a câmera, 113
  - 4 Notificações, 116
- Considerações finais, 122

Referências, 123

## **Capítulo 8**

### **Utilizando o React Redux, 125**

- 1** O que é o React Redux?, 126
  - 2** O Redux-Saga, colocando mais força no Redux, 128
  - 3** Exemplo prático de Redux-Saga, 131
- Considerações finais, 143
- Referências, 144

### **Sobre o autor, 147**

# Orientação a objetos por JavaScript

JavaScript é uma linguagem de programação que nos permite desenvolver aplicativos desde as antigas páginas HTML até as novíssimas aplicações front-end e back-end. Neste capítulo, vamos aprender quais são as principais vantagens de utilizar essa linguagem e algumas das suas principais propriedades.

Também vamos relembrar alguns dos principais conceitos de orientação a objetos, como herança e reescrita de métodos, os quais são muito utilizados nas aplicações em todo o mundo.

# 1 A linguagem JavaScript

A linguagem JavaScript é uma das linguagens de programação mais populares do mundo. Inicialmente, era utilizada sobretudo para desenvolvimento web, quando era empregada para aplicar códigos às páginas HTML, a fim de deixá-las mais dinâmicas.

Com o tempo, o JavaScript evoluiu, principalmente com a disponibilidade de bibliotecas como o jQuery. Hoje, é possível escrever códigos para vários tipos de aplicações, como back-end, utilizando o framework Node.js; front-end, com os frameworks ReactJS, AngularJS ou Vue.js; códigos para aplicações mobile, com o framework jQuery Mobile junto com o phonegap; estruturas mais maduras, como o React Native; e até mesmo escrever games, utilizando o framework Unity.



## IMPORTANTE

Esta é uma das grandes vantagens do aprendizado em JavaScript: com a mesma linguagem, mas com pequenas diferenças, é possível escrever códigos para todo esse universo de aplicações. Hoje, são pouquíssimas as linguagens que possuem essa flexibilidade, gerando aplicativos com muita qualidade e robustez.

Além disso, vale a pena destacar outros pontos importantes da linguagem JavaScript:

- **Fácil de aprender e de aplicar:** o JavaScript é uma linguagem de aprendizado muito simples e fácil, cuja sintaxe é semelhante à do Java, do C# e de outros padrões de linguagens.
- **Especificações e padronização:** a ECMA (European Computer Manufacturers Association, em tradução livre, Associação Europeia de Fabricantes de Computador) é uma associação que



padroniza linguagens scriptadas como o JavaScript, cuidando para que as novas versões sejam sempre revistas.

- **Atualizações:** além das padronizações, a ECMA também cuida para que o JavaScript esteja sempre atualizado e livre de bugs, de modo a deixá-lo cada vez mais fácil de utilizar.
- **Mercado:** o mercado de trabalho está utilizando cada vez mais o JavaScript, principalmente por sua flexibilidade full-stack. Segundo o relatório Octoverse, distribuído pelo Github (GITHUB, s. d.), o JavaScript é a linguagem de programação mais utilizada em repositórios da empresa. Isso mostra uma comunidade bem ativa, com muito material de aprendizado e que se comunica em sites ou fóruns para a correção de erros.

Apesar de, à primeira vista, o Java e o JavaScript serem semelhantes, existem grandes diferenças entre eles, como detalha David Flanagan (2013, p. 1):

Na verdade, o nome "Javascript" é um pouco enganoso. A não ser pela semelhança sintática superficial, Javascript é completamente diferente da linguagem de programação Java. E Javascript já deixou para trás suas raízes como linguagem de script há muito tempo, tornando-se uma linguagem de uso geral robusta e eficiente. A versão mais recente da linguagem [...] define novos recursos para desenvolvimento de software em grande escala.

Como dito, a ECMA praticamente lança novas atualizações da padronização do JavaScript a cada ano, o que torna o código cada vez mais simples e menos verboso. Atualmente, estamos na versão ES2020, mas é preciso destacar que, a partir da versão ES6 de 2015, ocorreram várias atualizações em pontos importantes na declaração de classes, nas operadores spread e nas operações assíncronas.



## NA PRÁTICA

Por conta dessas constantes atualizações, caso você procure algum exemplo ou dúvida na internet durante nossa jornada, é importante entender se o código está na versão mais nova. Como as atualizações a partir da versão ES6 são relativamente recentes, ainda existem muitas informações na internet que estão desatualizadas. Por exemplo, quando falarmos de classes, se você encontrar uma declaração de classes que utiliza a palavra-chave `Prototype` (algo bem comum), entenda que esse código é anterior ao ES6. Utilizar esses códigos não acarretará nenhum problema, pois o JavaScript continua compatível com versões anteriores, porém o ideal é já utilizar os padrões de código mais atuais.

Com todas as vantagens apresentadas, vamos aprender e utilizar a linguagem JavaScript na nossa jornada no aprendizado do React Native.

## 2 Objetos e propriedades

Um dos conceitos básicos de programação no JavaScript é a declaração de objetos. Os objetos são um tipo de dado não primitivo, que pode conter tanto propriedades como métodos. As propriedades, por sua vez, podem tanto conter objetos primitivos (um inteiro ou uma string, por exemplo) como outros objetos.

Para declarar um objeto, podemos utilizar a maneira linear de declaração. Por exemplo:

```
const object = {prop1: 1, prop2: "Valor 2"};
console.log(object);

console.log("Valor da propriedade 1" + object.prop1);
console.log("Valor da propriedade 1" + object.prop2);

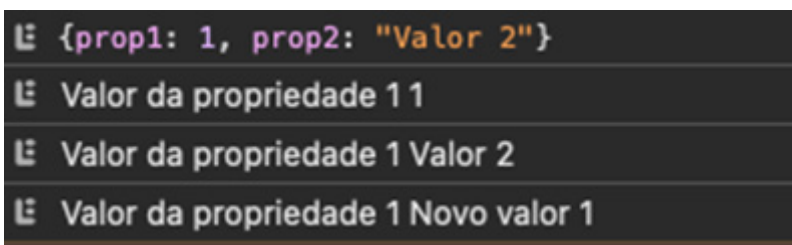
object.prop1 = "Novo valor 1";
console.log("Valor da propriedade 1" + object.prop1);
```

Na primeira linha, fazemos a declaração do objeto. Perceba que, primeiramente, utilizamos a palavra-chave *const* para declarar a variável *object*. Utilizamos as chaves para declarar o início e o fim de um objeto, e, dentro das chaves, separadas por vírgulas, ficam as propriedades. No nosso exemplo, o objeto possui as propriedades *prop1* e *prop2*, cujos valores são, respectivamente, o inteiro "1" e a string "Valor 2".

Nas três linhas seguintes, colocamos no console o valor do objeto inteiro (que mostrará todas as propriedades do objeto). E, nas outras linhas, mostramos como acessar os valores desse objeto, sempre utilizando o padrão nome da variável + ponto + nome da propriedade (no caso apresentado, *object.prop1*).

Em seguida, nosso exemplo mostra como adicionar um valor ao nosso objeto mesmo depois de declarado, o que é muito simples: colocamos o padrão nome da variável + ponto + nome da propriedade, e assim atribuímos o novo valor. Na figura 1, temos o resultado no console do navegador com as informações de propriedades.

Figura 1 – Console do navegador com as informações das propriedades



```
{prop1: 1, prop2: "Valor 2"}
Valor da propriedade 1 1
Valor da propriedade 1 Valor 2
Valor da propriedade 1 Novo valor 1
```



## IMPORTANTE

Você deve ter percebido que declaramos a variável *object* com a palavra-chave *const*. Esse novo conceito, implementado no ES6, permite que possamos declarar variáveis cujos valores não podem ser alterados, como *const*, e variáveis cujo valor pode ser alterado, como *let*. A antiga palavra-chave *var*, apesar de ainda funcionar, deve ser evitada.

Outro ponto importante é que também conseguimos colocar métodos nessa declaração linear. Por exemplo:

```
const object = { prop1: 1, metodo1: () => {return "Valor M1"}  
};  
let info = object.metodo1();  
console.log("Valor do método 1 =" + info);
```

Na primeira linha, fazemos a declaração do objeto com uma propriedade *prop1* e com um método chamado *metodo1*. Esse método é bem simples e somente retorna uma string com o valor "Valor M1". Na linha seguinte, chamamos o método usando o padrão objeto + ponto + nome do método – no caso anterior, *object.metodo1()*. No console, resulta a linha exibida na figura 2.

**Figura 2 – Console do navegador exibindo a informação retornada no método da classe**



Se analisarmos outras linguagens, como Java ou C#, um objeto necessariamente precisa de uma classe, mas isso não se aplica ao JavaScript, no qual temos a liberdade de colocar qualquer propriedade. É claro que isso também causa um ônus: sempre que você precisar utilizar uma propriedade, talvez seja necessário conferir se ela existe no objeto atual e se possui um valor.



### **IMPORTANTE**

O JavaScript possui dois tipos de valores “nulo”: o *null*, como qualquer outra linguagem, e o *undefined*, que significa que a variável não possui um valor definido. Portanto, ao verificar se a propriedade possui um valor, lembre-se de checar essas duas definições ou utilizar uma biblioteca como o *lodash*, que já possui um método para essa verificação.

Enquanto estamos trabalhando somente com propriedades, não há muito problema em utilizar essa forma linear de objeto. Porém, se nosso objeto contiver métodos, é altamente recomendável utilizar a declaração de classe construtora com um arquivo separado para nossa classe. Dessa maneira, o código ficará bem mais organizado.

## 3 Classes

Como a grande maioria das linguagens de programação, o JavaScript permite a utilização de todas as vantagens da orientação a objetos. E a primeira e mais básica função de orientação a objetos é a criação de classes.

Em versões prévias ao ES5, a construção de classes era bem mais complexa, porém, nas novas versões, ela ficou bem parecida com a sintaxe das linguagens mais comuns. Como no objeto literal, a classe também terá suas propriedades e métodos, mas incluirá alguns outros pontos, como o construtor.

É possível declarar a classe dentro de outras classes, porém o ideal é que seu objeto tenha seu próprio arquivo para melhor organização. Por exemplo:

```
export default class TestClass {  
  
  prop1 = 0;  
  #prop2 = 200;  
  
  constructor() {  
    console.log("Constructor");  
  }  
  
  doSomething(param) {  
    return "Doing" + this.#prop2 + " - " + param;  
  }  
  
}
```

Na classe apresentada, usamos as palavras-chaves *export default* para permitir que a classe seja exportada no JavaScript e seja importada em outros arquivos ou classes. Logo depois, utilizamos a palavra-chave *class*, que determina a declaração de uma classe, e, logo na sequência, o nome que utilizaremos para a classe – no nosso caso, *TestClass*. Dentro dessa classe, primeiro, temos as declarações das propriedades. A propriedade *prop1* é declarada como pública, sem utilizar o sinal *#*, e a propriedade *prop2* é declarada como privada, utilizando o sinal *#*. Vale destacar que não podemos utilizar nem o *let* nem o *const* nas declarações dessas propriedades.



## IMPORTANTE

Os métodos e as propriedades declarados como públicos podem ser acessados ou chamados fora da classe. Porém, métodos e propriedades declarados como privados somente podem ser acessados ou chamados internamente na classe. Em ambos os casos, para acessar ou chamar um método, é necessário utilizar a palavra-chave *this*.

O conceito de método privado ainda é bem novo no JavaScript. Ele foi implementado na versão do ES2019/ES10, mas a versão do Node 15 ainda não permite esse recurso. Por isso, por enquanto, ainda é melhor não utilizar essa função.

Após a declaração de propriedades, vamos declarar os métodos. Logo no início, temos o construtor, que é onde devem ser executadas todas as tarefas necessárias no momento que a classe é construída e iniciada. Logo após, criamos nosso primeiro método, que vai retornar uma string juntando ao parâmetro original.

Com a classe já pronta, vamos chamá-la em algum dos nossos arquivos:

```
import TestClass from './TestClass';

const testClass = new TestClass();
console.log("Prop 1 = " + testClass.prop1);
console.log("Prop 2 = " + testClass.prop2);
//console.log("Prop 2 = " + testClass.#prop2);

const info = testClass.doSomething("P1");
console.log(info);
```

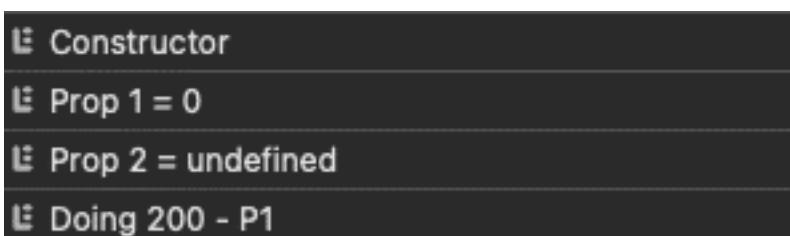
Na primeira linha do nosso arquivo, vamos utilizar a palavra-chave *import* para importar nossa classe no arquivo atual; para isso, fornecemos o nome da classe e qual o caminho para acessar seu arquivo.

O nome da classe declarado na linha *import* não precisa necessariamente ter o mesmo nome da classe que você criou. Esse caso se aplica, principalmente, quando há duas classes com o mesmo nome sendo importadas no mesmo arquivo. Porém, é claro, para uma melhor organização e compreensão do código, esse ponto deve ser tratado como exceção, e, se for possível, o ideal é manter o mesmo nome em todos os lugares.

Logo depois, utilize o operador *new*, que vai construir essa classe para acessarmos as propriedades e os métodos. Nas linhas abaixo, pegamos os valores das propriedades *prop1* e *prop2* e do método *doSomething*, passando como parâmetro a string "P1".

Dessa forma, teremos o resultado no console apresentado na figura 3.

**Figura 3 – Console exibindo as informações das propriedades da classe**



Podemos notar que recebemos o valor da *prop1* e o método *doSomething* sem problemas; porém, a variável *prop2* veio como *undefined*. Isso acontece porque essa propriedade está declarada como privada, lembra? Mesmo se tentarmos utilizar o caractere *#* na chamada da propriedade, não será possível acessá-la, pois aparecerá um erro no código.

## 4 Herança

Em orientação a objetos, a herança é o mecanismo que, ao criar uma classe, permite que esta utilize as propriedades e os métodos de uma classe base, evitando que se escrevam códigos duplicados e que se testem novamente códigos já certificados. Essa definição pode ser bem confusa no começo, mas vamos utilizar exemplos de códigos para deixá-la mais clara. Para começar, temos uma classe base chamada *Animal*, que possui as propriedades *peso* e *altura* e os métodos *comer* e *tomarAgua*:

```
export default class Animal {  
  
  peso = 50;  
  altura = 120;  
  
  comer() {  
    return "Comer";  
  }  
  
  tomarAgua() {  
    return "Tomar Agua";  
  }  
}
```

Vamos supor que essa classe já tenha sido testada e que agora precisemos criar uma classe chamada *Girafa*. Ela não precisa refazer os métodos *comer* e *tomarAgua*, porque eles já estão prontos na classe *Animal*. Nesse caso, vamos herdar esses métodos da classe *Animal* e acrescentar um novo método chamado *andar*:



```
import Animal from './Animal';

export default class Girafa extends Animal {

  andar(){
    return "Andar utilizando 4 patas";
  }
}
```

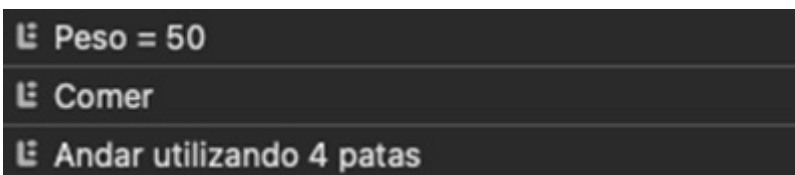
No JavaScript, para herdar a classe *Animal*, primeiro é preciso importá-la para dentro do arquivo e depois utilizar a palavra-chave *extends*. No final, vamos acessar a classe *Girafa* no nosso código, chamando a propriedade *peso* e os métodos *andar* e *comer*:

```
import Girafa from './Girafa';

const girafa = new Girafa();
console.log("Peso = " + girafa.peso);
console.log(girafa.comer());
console.log(girafa.andar());
```

No console, os resultados são apresentados conforme a figura 4.

**Figura 4 – Console exibindo as informações retornadas dos métodos da classe**



```
↳ Peso = 50
↳ Comer
↳ Andar utilizando 4 patas
```



## NA PRÁTICA

Sei que o exemplo *Animal* pode ser muito distante do nosso dia a dia. Então como podemos, na prática, usar a herança?

Bom, um exemplo bem básico de utilização desse conceito foi quando eu criei uma estrutura de log que já mostrava a data, o horário, o nome do arquivo e a informação de log. Assim, em vez de repetir esse método em todas as classes, ou construí-lo e chamar um objeto em todas as outras classes, pude herdar a classe *Log* e chamar o método. Desse modo, economizei bastante código, e, quando tinha a necessidade de mudar alguma coisa, era fácil mexer nele.

Podemos usar a mesma lógica, por exemplo, para classes que fazem a conexão com servidores. Nesse caso, toda a configuração pode ser herdada da classe pai, e a classe filho somente precisa se preocupar em realizar a conexão e chamar os retornos.

---

## 5 Reescrita de métodos (override)

Voltando um pouco ao conceito da herança, podemos, em uma nova classe, utilizar os métodos que já foram escritos na classe base. Mas e se for preciso alterar um pouco ou até totalmente esse método? Essa alteração é permitida por meio da reescrita de métodos, ou *override*. Então, vamos pensar no nosso exemplo anterior: a girafa, por exemplo, é um animal herbívoro. E se precisássemos adicionar essa “funcionalidade” ao objeto girafa? Por exemplo:

```
export default class Girafa extends Animal {  
  
  comer() {  
    return super.comer() + "somente folhas e vegetais";  
  }  
  
  andar(){  
    return "Andar utilizando 4 patas";  
  }  
}
```

Portanto, no exemplo, criamos um novo método chamado *comer*, que sobrescreverá o método *comer* da classe *Animal*. Se não precisássemos reutilizar nada na classe pai, era só desenvolver o que fosse necessário; mas, no nosso caso, vamos aproveitar o método *comer* da classe pai e juntá-lo com a nossa informação “somente folhas e vegetais”. Para isso, utilizamos a palavra-chave *super*, que vai buscar qualquer informação da classe pai, independentemente de o método atual estar ou não sendo sobrescrito.

Ao final, o resultado do nosso código é apresentado na figura 5.

Figura 5 – Console exibindo as informações dos métodos sobrescritos

```
🐞 Peso = 50
🐞 Comer somente folhas e vegetais
🐞 Andar utilizando 4 patas
```



### PARA SABER MAIS

Para entender mais sobre orientação a objetos, recomendamos a leitura do livro *Java: como programar*, dos autores Paul e Harvey Deitel, que está na 10ª edição e aprofunda ainda mais esses conceitos, com exemplos e dicas de implementação utilizando a linguagem Java.

## Considerações finais

Já temos na ponta da língua os conceitos sobre a linguagem JavaScript, bem como seus pontos fortes. Também já aprendemos a aplicar os conceitos de orientação a objetos de forma prática.

Vale reforçar que os conceitos de herança e reescrita de método são muito aplicados no dia a dia e ajudam a construir códigos mais limpos

e robustos. Por isso, antes de desenvolver um novo código, o planejamento da estrutura é realmente muito importante, se possível, aplicado à UML (unified modeling language, ou linguagem unificada de modelagem), para evitar a duplicação de códigos ou o não aproveitamento de um código já desenvolvido.

## Referências

FLANAGAN, David. **JavaScript**: o guia definitivo. Porto Alegre: Bookman, 2013.

GITHUB. The state of Octaverse. **GitHub**, [s. d.]. Disponível em: <https://octoverse.github.com>. Acesso em: 18 maio 2021.

# Desenvolvimento nativo e híbrido

Os celulares têm uma história relativamente recente, com início nos anos 1990 no Brasil e no mundo. Eles mudaram muito desde o início, quando serviam apenas para realizar ligações e armazenar contatos. Hoje, os celulares praticamente reúnem várias funções e gadgets, como câmera fotográfica, carteira, relógio, alarme e tantos outros.

Houve também grandes revoluções no desenvolvimento de aplicativos para celular. No início, havia grandes preocupações com armazenamento de imagens, gestão de recursos, processamento e memória; hoje, o celular é um ótimo computador em nossas mãos. Por isso,

vamos relembrar toda a história do desenvolvimento, chegando aos dias atuais e explicando os conceitos e as vantagens e desvantagens que existem entre os códigos nativos e híbridos, e por que você deve optar por escolher um deles.

Também vamos conhecer a história do início do desenvolvimento React e sua relação com o Facebook, e, claro, explicar os motivos de escolher essa plataforma de desenvolvimento, analisando suas vantagens. Por fim, vamos abordar a primeira dúvida que passa pela nossa cabeça ao optarmos pelo React Native: escolher entre o React Native CLI ou o Expo?

## 1 A linguagem nativa

Como falamos, os primeiros celulares eram muito simples, e praticamente só realizavam ligações. Quem não se lembra do bom e velho Motorola V3, colorido e febre de vendas? Segundo Harada (2016), os dois celulares que conseguiram revolucionar esse paradigma foram o Nokia 6110 e o Nokia 5110. Eles não foram os primeiros a vir com um jogo embarcado (o primeiro foi o Hagenuk MT-2000), mas, como possuíam um preço acessível e uma boa duração de bateria, foi a partir deles que essa nova funcionalidade se tornou um marco.

**Figura 1 – A evolução dos celulares, passando pelo modelo Nokia 5110**



Alguns anos depois, por volta de 2002, começaram a surgir os primeiros celulares que permitiam baixar aplicativos e jogos. Os primeiros foram o Nokia 3410 e o Siemens M50, que tinham a capacidade de baixar somente um aplicativo, mas que começaram a abrir o mercado para o desenvolvimento de aplicativos. Com isso, surgiram as duas primeiras “linguagens” nativas para o desenvolvimento de celulares, o Brew e o J2ME.

O Brew foi lançado pela Qualcomm, empresa que criou a tecnologia CDMA. Não era bem uma linguagem, mas sim uma biblioteca com base na qual era possível desenvolver aplicativos utilizando C ou C++ e acessar as features que estavam começando a ser disponibilizadas nos celulares, como a conexão com a internet, o GPS (na verdade, triangulação de antenas) e as primeiras câmeras.

O J2ME, por sua vez, era uma versão muito simplificada da Java. Com ele, era possível desenvolver aplicativos e jogos e acessar as features dos celulares. Porém, somente estava acessível nos celulares com a tecnologia GSM.

Por isso, essas plataformas estavam bem separadas por países. O mercado europeu adotava o GSM, que permitia somente aplicativos J2ME, enquanto nos EUA as operadoras utilizavam o CDMA, com o BREW. Se considerarmos o mercado brasileiro, nossos primeiros celulares foram os CDMA, que não tinham chip e possuíam Brew. Depois de alguns anos, o GSM começou a dominar o mercado, com as operadoras migrando para essa tecnologia e trazendo os aplicativos com a linguagem J2ME.

Ainda assim, o mercado era muito restrito, e somente as operadoras de telefonia móvel é que possuíam as lojas nas quais se podia baixar os aplicativos e jogos. Com isso, as operadoras faziam o filtro do que elas queriam ou não que fosse disponibilizado, e assim somente grandes empresas tinham acesso à publicação.

Mas aí veio a outra revolução do mercado, o iPhone. Além de um upgrade na tecnologia, nas features e, principalmente, no sistema touch, que era muito bem desenvolvido, a versão 3G do iPhone tinha a sua própria loja, na qual qualquer desenvolvedor poderia pagar US\$ 99 e lançar o seu próprio aplicativo (desde que estivesse dentro das políticas da Apple, é claro). Utilizando a linguagem Objective-C, era muito mais fácil e rápido lançar seu próprio aplicativo, e muitos desenvolvedores que se adaptaram aos novos recursos e possibilidades dos iPhones começaram a ganhar muito dinheiro com as vendas nas lojas.

Logo após o iPhone, em 2008, conforme descrito por Strain (2015), foi a vez de o Google lançar seu sistema operacional, o Android, e, claro, sua loja de aplicativos. Como era um sistema operacional open source, ou seja, de código aberto, o Android caiu no gosto das fabricantes de celulares e logo ficou disponível na maior parte do mercado de celulares, rivalizando com a Apple e com algumas fabricantes que ainda tentavam emplacar seus sistemas operacionais e lojas, como a Samsung, mas que depois acabaram desistindo.

Segundo Lecheta (2015, p. 25):

Tanto as empresas como os desenvolvedores buscam uma plataforma moderna e ágil para desenvolver aplicativos. Os fabricantes (LG, Motorola, Samsung, HTC, Intel, Sony etc.) precisam de uma plataforma robusta e rica em funcionalidade para lançar no mercado os seus produtos. É aqui onde o Android se encaixa, pois ele é perfeito para todos os casos.

No Android, também era possível submeter jogos e aplicativos de forma bem simples e fácil, utilizando o Java para Android e acessando as features dos celulares por meio do SDK (software development kit, ou kit de desenvolvimento de software).

Alguns anos depois, a Apple lançou sua nova linguagem, chamada Swift, muito mais simples e fácil de programar que a linguagem Objective-C. O Google também lançou sua nova linguagem, chamada



Kotlin, devido à sua briga na justiça com a Oracle, que havia comprado a linguagem Java.

Depois de toda essa história, o que as antigas linguagens Brew (C/C++), J2ME, Objective C, Java (Android) e as mais novas, Swift e Kotlin, têm em comum?

Todas elas trabalham nativamente na camada de aplicação dos celulares. Porém, apresentam um grande problema: quando trabalhamos com os dois grandes sistemas operacionais do mercado – o iOS, da Apple, e o Android, da Google –, precisamos desenvolver um mesmo aplicativo para cada uma das linguagens específicas de cada sistema, tendo um esforço duplicado.

## 2 A linguagem híbrida

Para tentar resolver esse problema de duplicidade de programação, começaram a ser desenvolvidas linguagens e bibliotecas cujo principal objetivo era criar um único código que pudesse ser rodado nos dois sistemas operacionais.

Uma das primeiras tentativas de biblioteca foi o jQuery Mobile. Segundo Silva (2013, p. 20),

jQuery Mobile é um framework baseado em interfaces criadas com marcação HTML5, otimizado para interação por toque e que se destina à criação de sites e aplicações para serem acessadas por smartphones, tablets e desktop.

Logo depois vieram o Sencha e o Ionic, que adotam uma linha similar. Em todas essas linguagens, é utilizada a biblioteca Phonegap para que os recursos dos celulares, como GPS ou câmera, possam ser acessados.

Todos eles funcionam bem, principalmente para aplicativos mais simples. Porém, a interface e a performance do aplicativo nunca ficam

tão boas como quando usada a aplicação nativa, pois todas essas linguagens trabalham na camada de navegador do celular, enquanto a aplicação nativa trabalha na camada de aplicação. Com isso, a diferença entre as duas linguagens é gritante. Por esse motivo, essas linguagens ficaram conhecidas como *hybrid web*.

Porém, algum tempo depois, foram desenvolvidas linguagens híbridas que são compiladas para o sistema operacional nativo e rodam diretamente na camada de aplicativo. São as chamadas *hybrid native*, como o React Native e o Flutter. Dessa maneira, você consegue ter uma performance muito próxima à do aplicativo nativo, mas desenvolvendo em uma única linguagem, o que torna os custos relativos a desenvolvimento, testes e tempo de lançamento muito menores.

### 3 Linguagem nativa ou híbrida?

Mas qual dos dois caminhos escolher, a linguagem híbrida ou a linguagem nativa? Lendo o parágrafo anterior, parece que a resposta é óbvia: a linguagem híbrida. Mas, como quase sempre nesse tipo de pergunta, a resposta certa é: depende.

Na grande maioria dos aplicativos, a performance das linguagens híbridas e nativas serão muito semelhantes entre si. Porém, se você precisa desenvolver um aplicativo que necessita de uma performance muito robusta, com um alto poder de processamento, o caminho é adotar a linguagem nativa. No caso dos aplicativos com grandes cálculos ou objetos 3D renderizados, o uso da linguagem nativa fará diferença.

Também devemos utilizar a linguagem nativa quando o código precisar rodar em vários tipos de linguagens. Imagine que você tenha de desenvolver uma biblioteca a ser inserida dentro do aplicativo de um cliente ou parceiro. Nesse caso, você acha que teria de desenvolver para React Native, Flutter, Ionic e Sencha? Não, o melhor caminho seria

desenvolver em Swift e Kotlin e criar apenas uma bridge (um código de ponte) entre o framework e o código nativo.

Porém, salvo os dois casos apresentados (e na imensa maioria dos casos), a linguagem híbrida que roda na camada de aplicativo, a hybrid native, será a melhor solução.



### PARA SABER MAIS

Caso você queira aprender sobre a linguagem nativa Android, um bom livro sobre o assunto é *Google Android: aprenda a criar aplicações para dispositivos móveis com o Android SDK*, de Ricardo R. Lecheta, que apresenta muitas dicas para desenvolver na plataforma Android.

## 4 React Native

O React Native é um framework open source desenvolvido e mantido pela equipe do Facebook. Ele permite o desenvolvimento de aplicativos mobile tanto para Android quanto para iOS, utilizando apenas a linguagem JavaScript ou TypeScript.

Mas por que o Facebook criou e mantém uma plataforma de desenvolvimento? Vamos voltar um pouco na história para entender isso. Segundo o perfil Shoutem (2016), da plataforma Medium, no início dos anos 2010 o Facebook estava crescendo, aumentando cada vez mais suas features e seu time de engenheiros. Com isso, o código estava ficando impraticável, principalmente com a quantidade de updates e a rerrenderização de pequenas mudanças na view. Por isso, o time de engenheiros do Facebook resolveu criar uma biblioteca com um sistema mais eficiente e com uma melhor experiência de usuário, e a chamaram de FaxJS.

Em 2012, o Instagram foi adquirido pelo Facebook, e, quando os engenheiros do Instagram viram essa biblioteca, quiseram aplicá-la no

desenvolvimento, mas ela estava muito enraizada na stack de desenvolvimento do Facebook. Portanto, eles decidiram desacoplá-la do código e transformá-la em código aberto. A partir daí, nasce o ReactJS.

Também em 2012, o Facebook decidiu se tornar uma empresa mobile. O primeiro passo foi utilizar HTML5 para renderizar o aplicativo, criando um aplicativo nativo que rodaria o código em uma WebView (o navegador acoplado no código nativo). Mas o resultado foi tão desastroso que o próprio Zuckerberg disse, em um evento do TechCrunch em 2013, que continuar com o HTML5 foi a pior decisão que ele havia tomado.

Foi então que os engenheiros tentaram começar a migrar para o código nativo, mas, além do código duplicado e difícil de ser gerenciado, esbarraram no problema das lojas, que muitas vezes demoram para testar o aplicativo e lançá-lo. Além disso, a maioria dos engenheiros era muito voltada para desenvolvimento front-end para web, o que tornaria a mudança muito custosa e demorada.

Por isso, eles tentaram uma maneira de, utilizando a biblioteca ReactJS, fazer com que um código JavaScript pudesse gerar elementos da interface gráfica nativa e rodá-la diretamente na camada de aplicação do celular. E eles conseguiram! Esse poderia ser um caminho para acabar com o problema de performance dos aplicativos híbridos.

Um ano depois, foi organizado um hackaton interno no Facebook e eles tiveram a certeza de que o React poderia se tornar um framework para desenvolvimento mobile. E, em 2015, foi lançado o React Native como código aberto, disponibilizado no GitHub. A primeira versão estaria disponível apenas para iOS, mas logo depois foi disponibilizada para Android, Windows e Tizen.

É interessante conhecer a história porque ficam bem claras as vantagens de utilizar o React Native. Vamos a elas:

- A primeira e mais óbvia vantagem é ter um único código gerando aplicativos para iOS, Android, Windows e Tizen, com performance muito similar à de um aplicativo nativo.
- O código é muito similar ao desenvolvimento web, fazendo com que desenvolvedores web também possam criar facilmente aplicativos móveis.
- Como surgiram da mesma biblioteca, desenvolver versões do seu aplicativo nativo para a web (e vice-versa) fica muito fácil. Se a camada de controle de tela e o modelo de negócio forem desenvolvidos com um bom padrão de projetos, pode-se aproveitá-los para as duas aplicações e somente mudar a camada visual.
- As linguagens JavaScript ou TypeScript possuem bibliotecas/frameworks que permitem desenvolver aplicações tanto em front-end (por exemplo, o React) quanto em back-end (por exemplo, utilizando o NodeJS). A grande vantagem desse ponto é que, tendo uma única linguagem para back-end e front-end, o time de desenvolvedores pode ser mais bem aproveitado, tornando-se full-stack com menos esforço e migrando de back-end para front-end conforme a necessidade do projeto.

Além dessas vantagens, devemos ressaltar que o React Native possui uma comunidade muito ativa na internet. No GitHub, ele possui hoje mais de 94 mil estrelas e 20 mil forks. Também vale reforçar que ele tem código aberto, ou seja, se alguém encontrar um bug, pode dar um pull no código, corrigir o erro e subir uma atualização que será analisada e, se estiver correta, adicionada ao branch principal. Isso faz com que as novas versões estejam com menos bugs e sejam mais otimizadas.

Outro ponto bem importante é a facilidade de buscar ajuda para possíveis problemas, bem como tutoriais na internet. Na pesquisa de 2020 do Stack Overflow, o React Native é a plataforma de desenvolvimento mobile mais utilizada pelos desenvolvedores, o que contribui muito para buscar essa ajuda.

Por esses motivos, grandes empresas também adotaram o React Native como sua stack para desenvolvimento mobile. São exemplos de aplicativos desenvolvidos com esse framework: Facebook, Instagram, Skype, Salesforce, Pinterest, CBS Sport, Call of Duty, Walmart, Uber Eats, Discord, Tesla e Wix.



### **PARA SABER MAIS**

Para começar a entender melhor o React Native, um bom lugar para buscar informações é o próprio site do desenvolvedor. Lá você encontrará a documentação da biblioteca, os componentes e as APIs (interface de programação de aplicações, do inglês *application programming interface*), além dos contatos das comunidades oficiais de desenvolvedores no Twitter e no Discord, por exemplo, onde é possível aprender bastante.

## **5 React Native CLI x Expo**

Vamos iniciar o desenvolvimento em React Native. Ao abrir a página de setup no site de desenvolvimento do aplicativo, já visualizamos uma aba na qual temos de escolher entre dois caminhos possíveis: Expo CLI ou React Native CLI. Qual é a diferença entre eles?

O Expo CLI é uma série de ferramentas construídas para o React Native que tornam o desenvolvimento muito mais fácil. Em questão de minutos, você configura o ambiente de desenvolvimento e consegue gerar um aplicativo de Hello World. Além disso, existem ferramentas como a Snack, com a qual é possível codificar e compilar diretamente na web, sem precisar configurar o ambiente.

Já o React Native CLI é a maneira tradicional de desenvolver aplicativos com React Native. Ele necessita da instalação dos ambientes de desenvolvimento para aplicativos nativos, como o Android Studio e/ou xCode. Se você já possui esses ambientes instalados, consegue rodar

aplicações em minutos. Caso contrário, a instalação e a preparação do ambiente pode levar algumas horas.

Mas, na prática, o que se deve considerar para escolher entre os dois caminhos? Para esclarecer essa escolha, vamos explicar as vantagens e desvantagens do Expo CLI. As principais vantagens são:

- início do desenvolvimento muito mais rápido (sem necessidade de instalar tantas ferramentas);
- alterações de código over-the-air (para alterar o aplicativo, não é necessário submeter as alterações às lojas); e
- módulos gerenciados e testados pelo pessoal do Expo.

Agora, as principais desvantagens do Expo CLI são:

- não é possível utilizar código nativo ou alguma biblioteca que utilize código nativo;
- nem todas as APIs do Android e do iOS estão disponíveis, como as APIs de bluetooth, WebRTC e in-app purchase;
- o SDK não suporta todos os tipos de execução em background;
- o aplicativo acaba ficando mais pesado;
- não é possível utilizar bibliotecas terceiras para envio de push notification; nesse caso, você deve utilizar a própria ferramenta da Expo, com o Firebase e o APNS;
- a versão mínima para o Android é a 5, e para o iOS é a 10; e
- não é possível lançar aplicativos para crianças até 13 anos.

O Expo CLI é uma boa ferramenta quando você está desenvolvendo aplicativos mais simples, porém, para aplicativos mais robustos, com maiores APIs, sugere-se utilizar o React Native CLI.

Em nossa jornada, para facilitar o aprendizado, vamos optar pelo Expo. Como os códigos são iguais, e caso você já possua um ambiente de desenvolvimento iOS ou Android, nada impede que você siga o caminho React Native CLI.



## NA PRÁTICA

Saiba que é possível iniciar o desenvolvimento em Expo CLI para aproveitar as vantagens de começar rapidamente e, se for o caso, migrar posteriormente para o React Native CLI. No entanto, essa não é uma tarefa muito simples, então vale a pena pensar bem antes de escolher um caminho.

## Considerações finais

A ideia de explicar toda a história por trás dessas linguagens de desenvolvimento é para mostrar como elas têm caminhado desde o início até hoje. Assim, quando nos aprofundamos na história do React Native, conseguimos entender por que o Facebook criou essa plataforma e a transformou em código aberto. E o fato de ela ter se tornado um código aberto nos traz segurança, pois a empresa a utiliza hoje no seu desenvolvimento e dificilmente realizará uma troca brusca.

Também é bem interessante conhecer quais dificuldades o Facebook enfrentou quando tentou escolher entre as linguagens nativas ou híbridas. Isso é bem importante, porque pode chegar um determinado momento de sua carreira em que você precisará optar por uma linguagem, e você deve estar preparado para analisar os pontos fortes e fracos de cada uma delas, em vez de simplesmente optar pela que está na moda.

Isso sem contar casos como a escolha entre o React Native CLI ou o Expo, a qual tem algumas consequências no início ou no final do projeto.



Por isso, sempre que for optar por uma linguagem, é bom refletir e entender suas vantagens e desvantagens e se questionar: será que essa é a linguagem que mais se adéqua à minha necessidade do momento?

## Referências

HARADA, Eduardo. Da cobrinha ao realismo: como os jogos de celular evoluíram com o tempo. **TecMundo**, 11 mar. 2016. Disponível em: <https://www.tecmundo.com.br/video-game-e-jogos/102175-cobrinha-realismo-jogos-celular-evoluiram-tempo.htm>. Acesso em: 27 mar. 2021.

LECHETA, Ricardo R. **Google Android**: aprenda a criar aplicações para dispositivos móveis com o Android SDK. São Paulo: Novatec Editora, 2015.

SHOUTEM. A brief history of React Native. **React Native Development**, 3 out. 2016. Disponível em: <https://medium.com/react-native-development/a-brief-history-of-react-native-aae11f4ca39>. Acesso em: 27 mar. 2021.

SILVA, Maurício Samy. **jQuery Mobile**: desenvolva aplicações web para dispositivos móveis com HTML5, CSS3, AJAX, jQuery e jQuery UI. São Paulo: Novatec Editora, 2013.

STRAIN, Martin. 1983 to today: a history of mobile apps. **The Guardian**, 13 fev. 2015. Disponível em: <https://www.theguardian.com/media-network/2015/feb/13/history-mobile-apps-future-interactive-timeline>. Acesso em: 27 mar. 2021.

# Instalando o React Native

Um dos grandes desafios que temos ao aprender uma nova linguagem é a montagem do ambiente, principalmente no mobile, onde, para conseguir gerar o aplicativo, muitas vezes é necessário montar um ambiente para Android e outro para iOS. Mas, com o React Native e o Expo, você vai perceber o quanto isso é fácil.

Para isso, será necessário instalar algumas bibliotecas e, depois, rodar algumas linhas de comando. E pronto! Você já conseguirá rodar o seu primeiro aplicativo na web e nos seus celulares.

Por fim, vamos conhecer algumas ferramentas para desenvolver em React Native, entre elas, a que consideramos a mais importante: o Fast Refresh. Essa ferramenta, ao salvar o arquivo, permitirá visualizar as alterações quase instantaneamente, sem que você tenha de esperar os intermináveis segundos (e muitas vezes minutos) para a aplicação terminar de compilar e rodar novamente.

## 1 Instalação do React Native com o Expo CLI e o Yarn

O primeiro passo para iniciarmos nossa jornada no React Native é instalar todas as bibliotecas necessárias à criação e à execução de um projeto. Aqui, vamos focar no Expo CLI, que é uma estrutura mais simples e fácil de ser executada.

Dessa maneira, precisaremos de algumas ferramentas para realizar o desenvolvimento:

- **Node.js:** sim, para executar o React Native (assim como o ReactJS) é necessária a instalação do Node.js. Apesar de ser conhecido como o nome do framework back-end, o Node.js é, na verdade, um ambiente de execução JavaScript. Isso quer dizer que ele permite criar aplicações JavaScript para rodar diretamente na sua máquina, sem depender de um navegador. Ele nos ajudará a compilar nossos arquivos e transformá-los em um aplicativo mobile.
- **Yarn:** o Yarn é um controlador de pacotes para JavaScript. Ele é responsável por instalar os pacotes necessários para rodar o aplicativo, como o pacote expo-cli ou as bibliotecas responsáveis pela utilização da câmera, do GPS e do recebimento de notificações.
- **Watchman (somente máquinas Mac):** o Watchman é uma ferramenta que monitora e grava arquivos quando eles são alterados. É utilizada para recriar o aplicativo assim que salvamos o arquivo.

- **Expo CLI:** utilizando o Yarn, vamos instalar o pacote Expo CLI, que permite que utilizemos todas as ferramentas que ajudam no desenvolvimento do aplicativo React Native.



## IMPORTANTE

Você deve estar se perguntando por que instalar o Yarn em vez de usar o npm, que já vem na instalação do Node.js. Antigamente, o Yarn tinha uma feature essencial: o lock que garantia que diversas máquinas estivessem com a mesma versão da biblioteca. Porém, a versão 5 do npm também veio com essa feature, e hoje os dois gerenciadores são muito parecidos, mas o Yarn ainda apresenta uma performance um pouco melhor e uma nova feature muito importante, que possibilita ver as licenças de cada biblioteca instalada. Como o Expo CLI recomenda a instalação do Yarn (ou o utiliza como default, caso já esteja instalado), vamos adotá-lo em nossa jornada.

Vamos começar com a instalação do Node 14. Para isso, acesse o site do desenvolvedor e clique no ícone do seu sistema operacional.

Figura 1 – Download do Node.js

node

JS

HOME | ABOUT | DOWNLOADS | DOCS | GET INVOLVED | SECURITY | CERTIFICATION | NEWS

Downloads

Latest LTS Version: 14.16.0 (includes npm 6.14.11)

Download the Node.js source code or a pre-built installer for your platform, and start developing today.

LTS

Recommended For Most Users

Windows Installer

node-v14.16.0-x86.msi

macOS Installer

node-v14.16.0.pkg

Source Code

node-v14.16.0.tar.gz

Windows Installer (.msi)

Windows Binary (.zip)

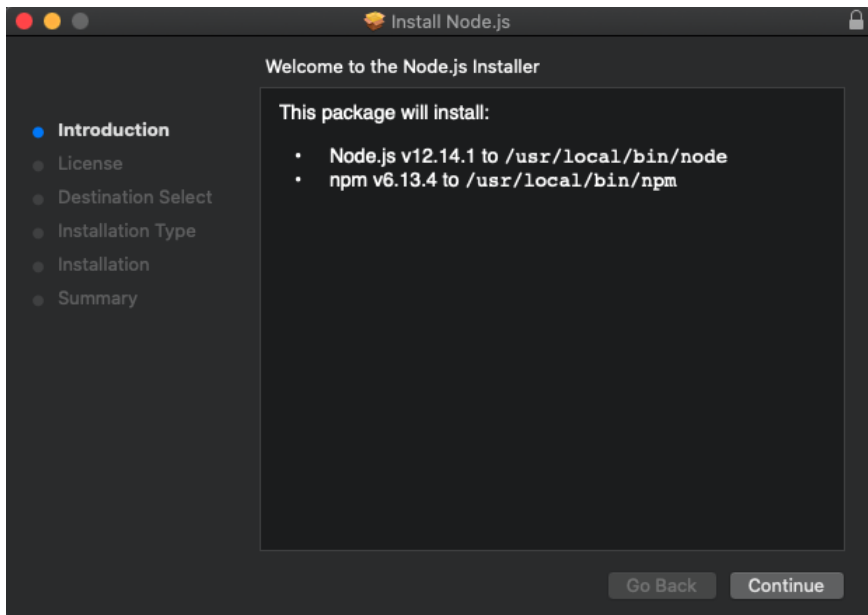
macOS Installer (.pkg)

macOS Binary (.tar.gz)

32-bit	64-bit
32-bit	64-bit
64-bit	
64-bit	

Após baixar o arquivo, execute-o e siga o passo a passo da instalação.

Figura 2 – Instalando o Node.js



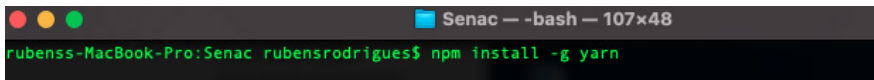
Após a instalação, vamos precisar do prompt de comando. Se você estiver usando o Windows, pressione *Window + x* e depois clique em *Executar*. No Mac, pressione *Command + espaço* e digite *Terminal*.

Figura 3 – Abrindo o terminal



Segundo o site do Yarn (YARN, s. d.), para realizar a instalação do Yarn, precisamos digitar o seguinte comando no terminal, ou prompt de comando aberto: *npm install -g yarn*.

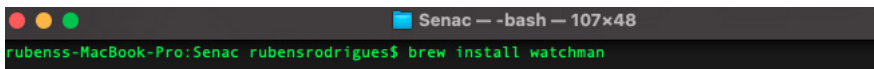
Figura 4 – Instalando o Yarn



```
Senac — -bash — 107x48
rubenss-MacBook-Pro:Senac rubensrodrigues$ npm install -g yarn
```

Se você estiver utilizando o macOS, precisa instalar o Watchman. Para isso, de acordo com o site do Watchman (s. d.), se você tiver o HomeBrew instalado, deve utilizar o comando *brew install watchman*; ou, se tiver o MacPorts instalado, deve utilizar o comando *sudo port install watchman*.

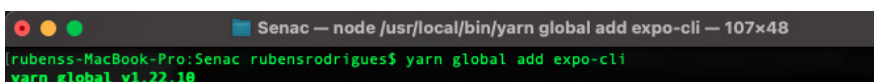
Figura 5 – Instalando o Watchman



```
Senac — -bash — 107x48
rubenss-MacBook-Pro:Senac rubensrodrigues$ brew install watchman
```

Por fim, vamos instalar o pacote Expo CLI. Consultando o site de desenvolvimento do Expo (s. d.), sabemos que devemos utilizar o comando *yarn global add expo-cli*.

Figura 6 – Instalando o Expo CLI



```
Senac — node /usr/local/bin/yarn global add expo-cli — 107x48
rubenss-MacBook-Pro:Senac rubensrodrigues$ yarn global add expo-cli
yarn global v1.22.10
```

Pronto, estamos com todo o ambiente pronto para ser executado.



## NA PRÁTICA

Você percebeu que, praticamente, vamos precisar montar o mesmo ambiente para o back-end Node.js? É por isso que é fácil os desenvolvedores front-end e back-end migrarem entre ambos os contextos. Além do ambiente pronto para as duas plataformas, a linguagem será a mesma.

## 2 Hello World

Para criar um novo projeto no Expo CLI, é preciso acessar o terminal, ou prompt de comando, escolher a pasta onde o projeto será criado e executar a seguinte linha: `expo init MyProject` (lembrando que *MyProject* é o nome do projeto que você deseja criar).

Ao ser executado, o pacote nos dá algumas opções de inicialização:

- **Manage workflow:** essa é a opção mais vantajosa na utilização do Expo CLI, pois realiza o gerenciamento de toda a complexidade de criar aplicativos, tirando essa dor de cabeça do desenvolvedor. Porém, a ela também se aplicam as limitações apresentadas no capítulo anterior. Possui três opções:
  - *blank*: projeto em branco com o menor número de bibliotecas possível e que utiliza a linguagem JavaScript;
  - *blank (TypeScript)*: é o mesmo projeto do caso anterior, porém tem a possibilidade de rodar a linguagem TypeScript; e
  - *tabs (TypeScript)*: vem com um projeto padrão com algumas telas de exemplo usando o react-navigation e o TypeScript.
- **Bare workflow:** nesta opção, o desenvolvedor tem o controle completo da aplicação e, inclusive, não fica preso às limitações do Expo CLI. Porém, é necessário trabalhar e desenvolver utilizando o xCode e o Android Studio. Possui duas opções:
  - *minimal*: projeto em branco com o menor número de bibliotecas possível e que utiliza a linguagem JavaScript; e
  - *minimal (TypeScript)*: é o mesmo projeto do caso anterior, porém tem a possibilidade de rodar a linguagem Typescript.

Acompanhe na figura 7.

Figura 7 – Criando o projeto com o Expo

```
Senac — node /usr/local/bin/expo init MyProject — 80x24
rubenss-MacBook-Pro:Senac rubensrodrigues$ expo init MyProject

There is a new version of expo-cli available (4.3.4).
You are currently using expo-cli 4.3.2
Install expo-cli globally using the package manager of your choice;
for example: 'npm install -g expo-cli' to get the latest version

? Choose a template: > - Use arrow-keys. Return to submit.
  ----- Managed workflow -----
  > blank a minimal app as clean as an empty canvas
    blank (TypeScript) same as blank but with TypeScript configuration
    tabs (TypeScript) several example screens and tabs using react-navigatio
n and TypeScript
  ----- Bare workflow -----
  ed minimal bare and minimal, just the essentials to get you start
    minimal (TypeScript) same as minimal but with TypeScript configuration
```

Como vamos criar apenas um Hello World, vamos escolher *managed workflow* com a opção *blank*. Feito isso, após alguns segundos, o projeto será criado.

### 3 Rodando o projeto na web

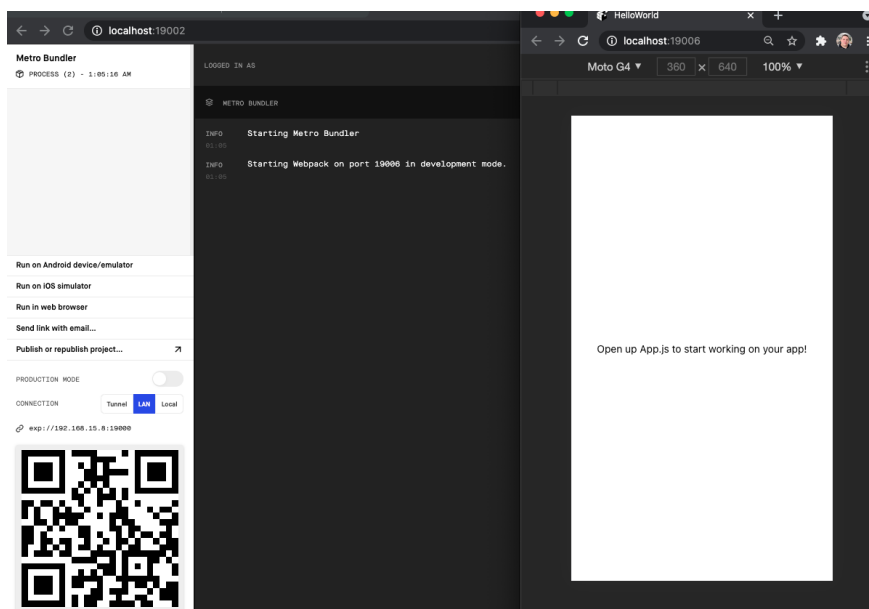
Após criar o projeto, vamos rodá-lo na nossa máquina. Para isso, se você prestar atenção ao final do texto que será exibido após o comando de criação do projeto, poderá executar os seguintes comandos: abrir somente a página de controle (*yarn start*), abrir diretamente o aplicativo Android no emulador ou no device (*yarn android*), abrir diretamente o aplicativo iOS no emulador ou no device (*yarn ios*) e abrir o aplicativo direto na web (*yarn web*).

Como é possível perceber, uma das grandes vantagens de utilizar o Expo CLI é que você pode testar seu aplicativo direto no seu navegador. Sim, a execução é bem diferente do que no emulador ou no device, mas essa pode ser uma bela ajuda para ajustes rápidos.



Então, vamos experimentar! Basta executar o comando *yarn web* para rodar diretamente, ou executar *yarn start* e, na página de controle, clicar em *Run in web browser*, conforme a figura 8.

Figura 8 – Criando o projeto com o Expo



## 4 Rodando o projeto no device

O Expo CLI também permite que você rode e faça a depuração, ou debug, do seu aplicativo diretamente no device sem instalar nenhum SDK (software development kit, ou kit de desenvolvimento de software). E isso é bem interessante, porque permite que você desenvolva um aplicativo para iPhone ou iPad sem a necessidade de possuir uma máquina Apple – o que é um requisito quando você está desenvolvendo em código nativo ou React Native CLI.

Para rodar seu aplicativo, é importante que sua máquina e seu device estejam na mesma rede wi-fi. No celular, baixe o aplicativo Expo Go na loja e, após a instalação, basta apontar a câmera do celular para o QR code da tela de controle do Expo CLI, e pronto! O seu aplicativo já vai começar a rodar diretamente no seu celular.

## 5 Rodando o projeto nos emuladores

Caso você não possua o device específico para o qual queira desenvolver (como um iPhone 12, por exemplo), outro caminho é rodar sua aplicação diretamente nos emuladores.

Porém, para rodar os emuladores, infelizmente é preciso instalar os SDKs dos códigos nativos. Como explicado por Lecheta (2015, p. 42), “o Android SDK é o software utilizado para desenvolver aplicações no Android que tem um emulador para simular o dispositivo, ferramentas utilitárias e uma API completa”. Essa referência funciona tanto para Android quanto para iOS, e vale ressaltar que, para rodar o emulador do iOS, é necessária uma máquina com o macOS.

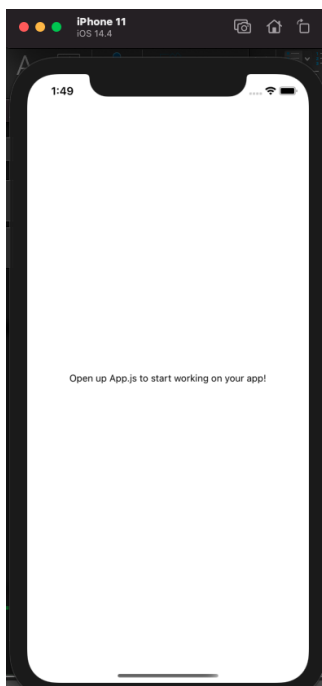


### PARA SABER MAIS

Não vamos nos aprofundar na instalação dos SDKs, que pode ser bem complexa. Para isso, na página da Expo na internet há um passo a passo de como instalar as ferramentas necessárias para rodar o emulador, tanto Android como Apple/iOS.

Após a instalação concluída, basta clicar nos botões *Run on Android device / simulator* ou *Run on iOS simulator* para rodar sua aplicação.

Figura 9 – Rodando seu aplicativo



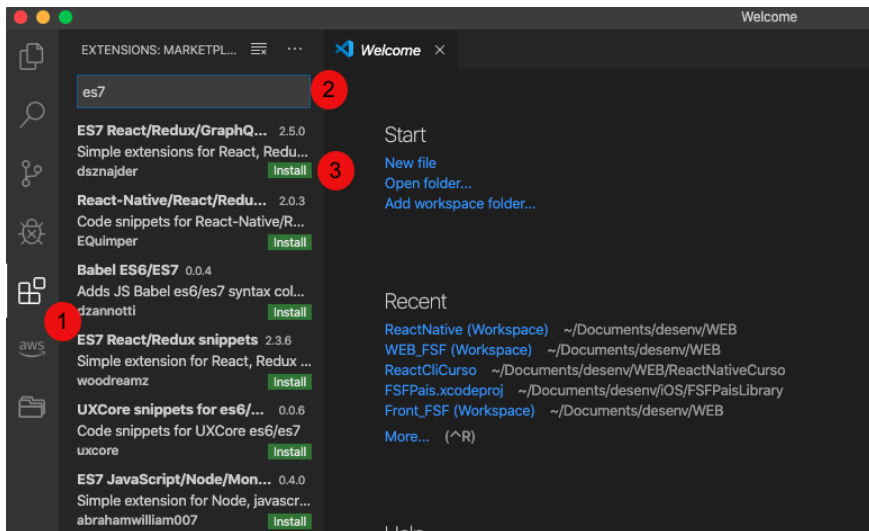
## 6 Editor de código

Você pode desenvolver o React Native em qualquer editor de código disponível no mercado, como o Sublime, o Atom etc.

Porém, a sugestão do Expo (e a nossa também) é utilizar o Visual Studio Code da Microsoft, que é bem leve, possibilita a instalação de módulos que ajudam muito no desenvolvimento e, o melhor, é gratuito. Para instalá-lo, vá até a página do Visual Code, clique em download e faça a instalação.

Para adicionar um módulo, abra o Visual Studio Code (também conhecido como VS Code) e clique no botão *componentes* na lateral direita; depois, digite o nome do componente que deseja instalar no campo de busca e clique no botão *install*.

Figura 10 – Instalando módulos no VS Code



Os módulos que ajudam no desenvolvimento são:

- ES7 React/Redux/GraphQL/React-Native snippets;
- React Native – Full Pack;
- React-definition; e
- React Native Tools (que permitem debug pelo próprio VS Code).

## 7 Ferramentas de desenvolvimento

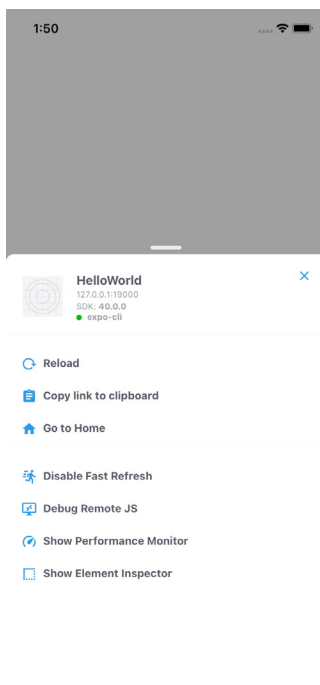
Ao executar um aplicativo React Native, você tem à disposição um menu com várias opções que ajudam no desenvolvimento de aplicativos. Cada device ou emulador tem a sua própria forma de abrir esse menu. Acompanhe cada uma das opções:

- no device Android/iOS, agite o celular com o aplicativo aberto;
- no emulador Android, clique em *Ctrl + L* ou *⌘ + L*; e

- no emulador iOS, clique em  D.

Ao executar um dos passos, o seguinte menu aparecerá, conforme a figura 11.

**Figura 11 – Menu de desenvolvimento Expo**



Essas ferramentas disponibilizadas no menu ajudam muito no dia a dia do desenvolvimento, com funcionalidades para a atualização de conteúdo, debug e leitura de erros. Vamos nos aprofundar nas principais dessas funcionalidades.

## 7.1 Fast Refresh

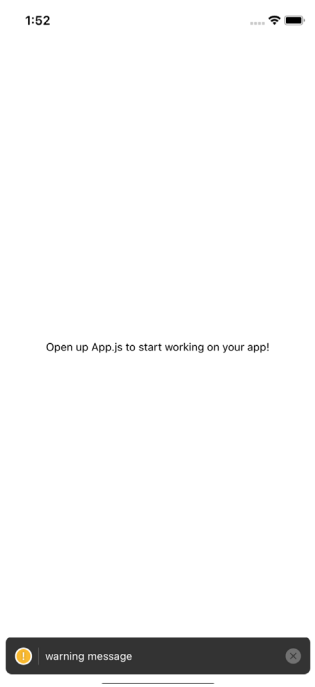
O Fast Refresh é uma das ferramentas mais interessantes para o desenvolvimento em React Native. Com ela, ao realizar uma alteração

no código-fonte e salvar, as mudanças serão refletidas no aplicativo sem que haja a necessidade de realizar uma nova compilação, o que faz com que os desenvolvedores ganhem muito tempo. Essa opção já vem ativada por padrão, mas, caso você queira desativá-la, é só clicar em *Disable Fast Refresh*.

## 7.2 In-app errors e warnings

Em tempo de execução, quando ocorrer qualquer erro ou warning, será exibida uma janela pop-up em vermelho ou amarelo com as informações do que ocorreu e de qual linha está relacionada ao erro. Caso você queira forçar a exibição de alguma informação com esse recurso, basta utilizar os comandos `console.warn()` e `console.error()`. Para ignorar as informações de warning, utilize a linha `console.disableYellowBox = true`.

Figura 12 – Mensagem de warning

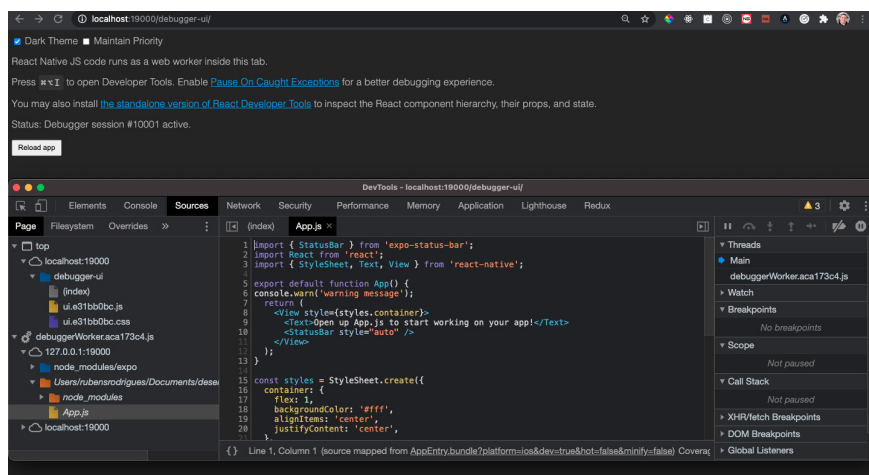


## 7.3 Debug

Como várias outras linguagens, o React Native permite realizar a depuração, ou debug, diretamente no device ou no emulador. Para isso, é necessário utilizar as ferramentas de desenvolvimento do Chrome.

O primeiro passo, no menu, é clicar em *Debug Remote JS*. Você perceberá que, ao clicar no botão, o Chrome será automaticamente aberto e exibirá uma tela preta com o texto “React Native JS code runs as a web worker inside this tab”. Ao clicar com o botão direito e depois em *inspecionar elementos*, você poderá ter acesso ao console da aplicação com os logs que colocou no aplicativo, bem como ao código-fonte na aba *Source*.

Figura 13 – Tela de debug



## Considerações finais

Agora que você chegou ao final do capítulo, com o ambiente montado e o primeiro projeto criado, vem à mente a seguinte pergunta: onde seria melhor rodá-lo, nos celulares ou nos simuladores?

O celular é uma boa opção, porque nada melhor do que rodar seu aplicativo e observar a performance dele diretamente no ambiente final. E também há outros pontos positivos, como rodar o aplicativo em um iPhone sem precisar do macOS; rodar o debug diretamente no celular, sem a necessidade de ele estar cabeado; e ter todos os recursos de desenvolvimento do React Native, como Fast Refresh, warnings e informações de erros.

O ponto de vantagem dos simuladores é que, hoje, eles são muito parecidos com os celulares reais. Portanto, você tem a possibilidade de testar seu aplicativo em um “celular” que você não possui, com formato de tela e resoluções diferentes.

A resposta da pergunta, então, é: depende. Você deve analisar as duas possibilidades e ver a que melhor se adéqua às suas necessidades.

## Referências

EXPO. Installation. **Expo**, [s. d.]. Disponível em: <https://docs.expo.io/get-started/installation/>. Acesso em: 1 abr. 2021.

LECHETA, Ricardo R. **Google Android**: aprenda a criar aplicações para dispositivos móveis com o Android SDK. São Paulo: Novatec, 2015.

WATCHMAN. Installation. **Watchman**, [s. d.]. Disponível em: <https://facebook.github.io/watchman/docs/install#buildinstall>. Acesso em: 1 abr. 2021.

YARN. 2 – Installation. **Yarn**, [s. d.]. Disponível em: <https://yarnpkg.com/getting-started/install>. Acesso em: 1 abr. 2021.



# O Flexbox e estilos

Finalmente, chegou a hora de colocarmos a mão na massa! Mas, para isso, precisamos aprender alguns conceitos importantes no desenvolvimento do React Native, como a estrutura de um componente, na qual temos de declarar cada funcionalidade e o modo como fazemos os layouts, utilizando um trio que nos ajudará bastante: o JSX, o StyleSheet e o Flexbox. O JSX é muito parecido com o HTML; com ele, vamos inserir as informações e variáveis na tela. O StyleSheet é muito parecido com o CSS, e permite que coloquemos estilos nos componentes a serem inseridos com o JSX. Já o Flexbox é uma ferramenta

utilizada dentro do StyleSheet, e nos ajuda muito na organização dos itens na tela, o que é uma grande dificuldade no desenvolvimento mobile, já que trabalhamos com diversos tipos, tamanhos e formas de celulares e precisamos aproveitar o máximo de espaço possível.

## 1 Estrutura de um componente

O React, de forma geral, trabalha muito com o conceito de componentes. A ideia é diminuirmos a quantidade de código em cada “classe” e criarmos componentes especialistas que podem ser reutilizados em toda sua aplicação.

Então, por exemplo, podemos criar um componente botão, com o formato e o design que será utilizado em todo o aplicativo, e utilizá-lo em várias telas, em vez de colocar o leiaute em todos os botões. Isso ajuda muito a economizar código e, principalmente, ajuda na manutenção, porque se for necessário mudar alguma coisa, será preciso mexer somente no componente, o que será replicado em todos os lugares.

Mas qual é a estrutura do componente? Bom, ele é declarado em partes, de acordo com a seguinte ordem:

- Parte 1 – Importando as bibliotecas e os componentes;
- Parte 2 – Declaração da function e retorno do código JSX;
- Parte 3 – Criação de um StyleSheet; e
- Parte 4 – Exportação do componente.

Vamos, agora, analisar o código de Hello World, que é gerado na criação do projeto. Logo no início, é preciso importar as bibliotecas e/ou outros componentes que serão utilizados.

```
import { StatusBar } from 'expo-status-bar';
import React from 'react';
import { StyleSheet, Text, View } from 'react-native';
```

Na sequência, vamos iniciar a declaração do nosso componente e, no seu return, declarar o código JSX, que será exibido na tela. Mais adiante, abordaremos melhor o JSX.

```
function App() {
  return (
    <View style={styles.container}>
      <Text>Hello World</Text>
      <StatusBar style="auto" />
    </View>
  );
}
```

Depois das definições das classes, serão inseridos os StyleSheets com as definições de estilo utilizado no código.

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

E, finalmente, você precisará exportar o seu componente para que ele possa ser utilizado em todo o código.

```
export default App;
```

Essa exportação é importante para que você consiga utilizar seu componente em outras partes do código, como o menu. Também é

possível fazer essa declaração no início do componente, que ficaria da seguinte forma:

```
export default function App() {
```

Esse formato será utilizado em todos os componentes a partir de agora, por isso é importante entendermos todos os conceitos.



### IMPORTANTE

Você pode encontrar outras formas de declarar um componente, por exemplo, utilizando classes. Isso acontece porque, nas versões anteriores do React Native, era indicado fazer a declaração dessa maneira. Porém, desde a versão 0.59 do React Native e a inclusão do React Hooks, a forma exposta aqui é a mais indicada.

## 2 Conhecendo o JSX

Ao falarmos de componentes, citamos o código JSX, que ajuda a declarar os itens visuais do aplicativo. Mas como ele funciona? Bom, quando você analisa a declaração de variável, o que você enxerga?

```
const element = <Text style={styles.textInfo}>Hello World</Text>;
```

Esse código não é nem uma string, nem um código HTML. Como declarado na documentação oficial do React, essa declaração:

É chamada JSX e é uma extensão de sintaxe para JavaScript. Recomendamos usar JSX com o React para descrever como a UI deveria parecer. JSX pode lembrar uma linguagem de template, mas vem com todo o poder do JavaScript (REACT, s. d.).

Mas como isso é, na verdade, JavaScript?

O React Native utiliza a ferramenta Babel para decodificar o seu código. Então, nesse caso, a linha anterior é uma interpretação do seguinte código:

```
const element = React.createElement(Text, {  
  style: styles.textInfo  
}, "Hello World");
```

Não é obrigatório o uso do JSX, porque você pode declarar o código JavaScript diretamente, mas imagine o quão complexo seria fazer isso sem o Babel. Outro ponto interessante, que você deve ter notado, é que as tags utilizadas no JSX são importadas no começo do código.

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

Mas o que são essas tags? Primeiro, elas não tags, e sim componentes do React Native que, em tempo de execução, utilizam os componentes nativos do sistema operacional. Isso ajuda a tornar o React Native muito mais rápido e fluido.

Existem vários desses componentes – por exemplo, o Image, o Button, o View e o FlatList – que são utilizados no desenvolvimento em React Native, mas vamos explorá-los apenas mais adiante.

### 3 Text e aplicação de estilos com o StyleSheet

Os aplicativos em React Native podem ser muito bonitos, utilizando imagens e cores que exploram o melhor do design. Uma das ferramentas utilizadas para aplicar o estilo é o StyleSheet. Ele é muito similar ao CSS, que utilizamos no HTML, mas possui pequenas diferenças.

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

Para criar o estilo, a primeira diferença é que, em vez da tag `style`, usada no CSS tradicional, utilizamos o componente `StyleSheet` do React Native e o método `create`. Como parâmetro do método, passaremos um objeto JavaScript, sendo que suas propriedades serão as funcionalidades que vamos aplicar no estilo. Essas funcionalidades são muito parecidas e apresentam resultados semelhantes aos da versão HTML.

Vamos usar como exemplo o *font-size* e o *background-color*, que possuem a mesma funcionalidade do CSS tradicional, e cuja única diferença, no React Native, está na escrita: *font-size* se torna *fontSize*, e *background-color* se torna *backgroundColor*. Ou seja, sempre que houver um traço na forma tradicional, no React Native é preciso juntar as duas palavras, deixando a primeira letra da segunda palavra maiúscula – é o famoso modelo de escrita camelo, ou *camelCase*.

Vejamos como isso se dá no componente `Text`. Você deve ter percebido, nos nossos códigos anteriores, que ele serve para colocar um texto na tela. Mas como aplicamos um estilo nele, aumentando a fonte e trocando a cor do texto? O primeiro passo é criar um novo arquivo, chamado *AppStyle.js*, e declarar nosso estilo.

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

(Cont.)

```
    textInfo: {  
      fontSize: 30,  
      color: 'red'  
    }  
  });  
  export default styles;
```

No arquivo *App.js*, vamos importar esse novo componente com o estilo.

```
import styles from './AppStyles';
```

E, no final, vamos colocar o estilo no componente *Text*, utilizando a propriedade *style* e passando o nome do nosso estilo (no caso, o *textInfo*) junto com o nome *styles*<sup>1</sup> dentro de chaves. E, é claro, podemos colocar esse estilo em qualquer componente do nosso aplicativo.

```
<Text style={styles.textInfo}>{info}</Text>
```



## IMPORTANTE

Mesmo que haja um arquivo separado apenas com o estilo, na declaração apresentada anteriormente é preciso importar o componente *StyleSheet* do React Native para que o estilo funcione. No componente que realizou a importação, essa declaração não é mais necessária.

---

<sup>1</sup> O nome *styles* não é obrigatório, você pode escrever o nome que quiser. Só é importante que a variável, o nome da exportação no arquivo *AppStyles.js* e o nome colocado antes do *from* na importação sejam os mesmos.

É possível declarar o estilo diretamente no componente, indicando, dentro da propriedade *style*, as funcionalidades do estilo. Mas, como vamos utilizar um objeto dentro da variável no JSX, precisaremos utilizar duas chaves. Dessa forma, o código ficará assim:

```
<Text style={{fontSize: 30, color: 'red'}}>{info}</Text>
```

Também é possível juntar as duas maneiras de declaração. Nesse caso, passamos um *array* para a propriedade *style*, que automaticamente junta as duas informações e aplica os dois estilos. Nosso exemplo ficaria assim:

```
<Text style={[ styles.textInfo, {fontSize: 30, color: 'red'}}>{info}</Text>
```

O ideal é deixar sempre as declarações de estilos em arquivos separados, para que o código de estilo não se misture ao código de montagem do leiaute. Usamos a declaração de estilo diretamente no componente apenas nos casos em que é necessário alterar o estilo com uma informação variável, recebida por uma conexão ou por algum input do usuário. Por exemplo, em um aplicativo bancário, podemos mudar a cor do componente *Text* para vermelho ou verde, a depender do saldo em conta corrente do usuário.

## 4 O Flexbox

O Flexbox é uma ferramenta muito importante no desenvolvimento de aplicativos React Native, pois facilita muito o trabalho do desenvolvedor na hora de montar os leiautes. Ele se baseia essencialmente em dois conceitos: empilhamento e alinhamento.

O conceito de empilhamento remete a como será a disposição do conteúdo na tela. Considerando uma página de login, primeiro teremos



um logo no topo da tela, e, embaixo, uma caixa com os campos de input. Dentro dessa caixa, teremos primeiro o label com o nome *login*; depois, a caixa de texto, onde será inserido o usuário; depois, outro label com o texto *password*; depois, uma caixa de texto para inserir a senha; e, por último, os botões *login* e *esqueci a senha*. Percebeu como estamos empilhando todos os componentes? Este é o conceito: empilhamos os componentes para adequá-los na tela. E, é claro, devemos colocar espaçamentos e deixá-los com um design bem interessante.

O outro conceito é o de alinhamento, segundo o qual, como o próprio nome diz, alinhamos o conteúdo que foi empilhado de forma centralizada, na lateral esquerda, no topo da tela, enfim, de várias maneiras. Essa ideia ficará mais clara quando utilizarmos os exemplos, mais adiante. Aliás, para executar os exemplos, você pode empregar a ferramenta Yoga Layout, que é muito útil para quem está aprendendo.

O Flexbox também é utilizado para desenvolvimento web, mas com algumas diferenças. Como descrito na documentação de desenvolvimento do React Native:

O Flexbox funciona da mesma maneira tanto no React Native quanto no CSS na web, com poucas exceções. Os valores padrões são diferentes, por exemplo, *flexDirection* como *column* em vez de *row*, *alignContent* como *flex-start* em vez de *stretch*, *flexShrink* como *0* em vez de *1*, e o *flex* admitindo somente números inteiros. (REACT NATIVE, s. d., tradução nossa)

Para determinar as configurações a serem inseridas nos componentes, o Flexbox disponibiliza várias propriedades que trabalharão com os conceitos de alinhamento e empilhamento. As propriedades disponíveis são: *flex*, *flexDirection*, *justifyContent* e *alignItems*.

## 4.1 Propriedade *flex*

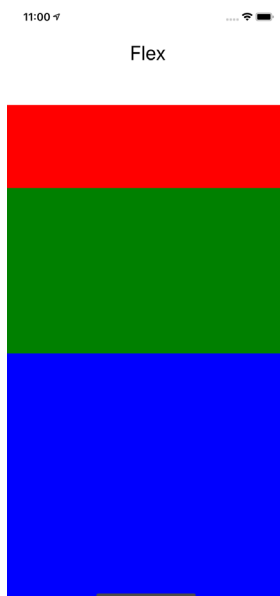
A propriedade *flex* é utilizada para definir como os itens serão preenchidos no espaço disponível em tela. Esse espaço será dividido de acordo com os números inseridos na opção.

```

StyleSheet.create({
  container: {
    flexDirection: 'column',
    flex: 1,
    marginTop: 60,
  },
  blockA: {
    flex: 1,
    backgroundColor: 'powderblue'
  },
  blockB: {
    flex: 2,
    backgroundColor: 'skyblue'
  },
  blockC: {
    flex: 3,
    backgroundColor: 'steelblue'
  }
});

```

**Figura 1 – Exemplo: *flex***



No `StyleSheet`, `blockA`, `blockB` e `blockC` recebem os valores 1, 2 e 3. Dessa maneira, o `flex` somará todos os valores do bloco (no caso,  $1 + 2 + 3 = 6$ ) e dividirá o espaço na tela em 6 partes. O `box1`, assim, fica com  $1/6$ , o `box2`, com  $2/6$  e o `box3`, com  $3/6$ , ou seja, a metade do contêiner.

## 4.2 Propriedade `flexDirection`

A propriedade `flexDirection` é utilizada para determinar em qual direção os itens da tela serão empilhados. Existem quatro tipos de opções: `row`, `column`, `row-reverse` e `column-reverse`.

Na `row`, os itens são alinhados da esquerda para a direita, na `column`, de cima para baixo. Ao adicionar o `reverse` na opção, inverte-se a característica do atributo; portanto, o `row-reverse` alinha os itens da direita para a esquerda, e o `column-reverse`, de baixo para cima.

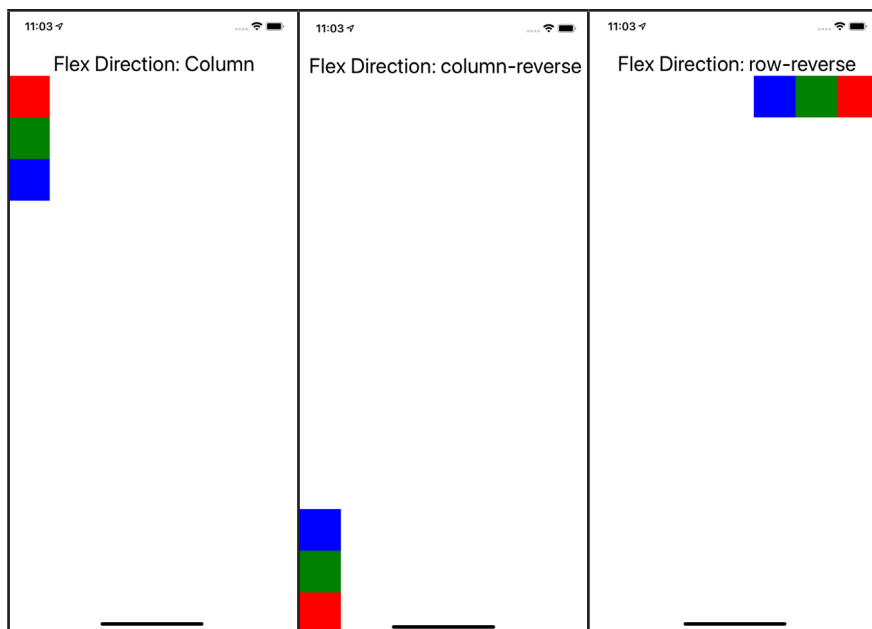
```
export default StyleSheet.create({
  container: {
    flex: 1,
    flexDirection: "row",
  },
  blockA: {
    width: 60, height: 60,
    backgroundColor: 'red'
  },
  blockB: {
    width: 60, height: 60,
    backgroundColor: 'green'
  },
  blockC: {
    width: 60, height: 60,
    backgroundColor: 'blue'
  }
});
```

Figura 2 – Exemplo: *flexDirection*



Ao alterarmos o contêiner para as outras opções do *flexDirection*, teremos:

Figura 3 – Outros exemplos do *flexDirection*



## 4.3 Propriedade *justifyContent*

O *justifyContent* controla os alinhamentos dos itens na tela dentro do espaço do contêiner. Esse atributo é trabalhado em conjunto com o *flexDirection*, e o alinhamento muda de acordo com a opção escolhida. Então, por exemplo, se escolhermos o atributo *row*, os itens serão alinhados horizontalmente, e, se escolhermos o atributo *column*, eles serão alinhados verticalmente.

A propriedade *justifyContent* apresenta as seguintes opções: *flex-start*, que alinha os itens no início do contêiner (esse é o valor padrão); *flex-end*, que alinha os itens no final do contêiner; *center*, que centraliza os itens; *space-between*, que realiza o alinhamento dos itens de forma igual, porém colando-os no topo e no rodapé; e *space-around*, que distribui o espaço entre os itens uniformemente, deixando o mesmo espaço no topo e no rodapé.

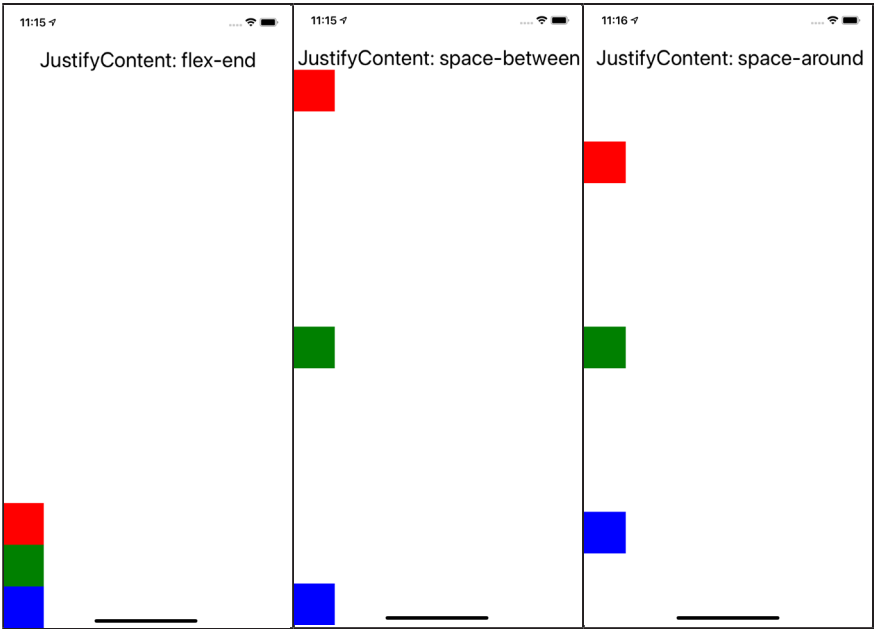
```
StyleSheet.create({
  container: {
    flex: 1,
    flexDirection: "column",
    justifyContent: 'flex-start'
  },
  blockA: {
    width: 60, height: 60,
    backgroundColor: 'red'
  },
  blockB: {
    width: 60, height: 60,
    backgroundColor: 'green'
  },
  blockC: {
    width: 60, height: 60,
    backgroundColor: 'blue'
  }
});
```

Figura 4 – Exemplo: *justifyContent*



Ao alterarmos o contêiner para as outras opções do *justifyContent*, teremos:

Figura 5 – Outros exemplos do *justifyContent*



## 4.4 Propriedade *alignItems*

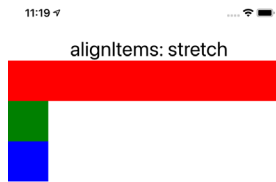
O *alignItems* é utilizado para alinhar os itens dentro do contêiner, o que é muito similar ao *justifyContent*; porém, o que muda entre eles é o eixo de alinhamento. Por exemplo, se o *flexDirection* estiver como *row*, o *justifyContent* realizará o alinhamento no eixo horizontal, e o *alignItem*, no eixo vertical. Porém, se você definir o *flexDirection* como *column*, o alinhamento se inverte, ou seja, o *justifyContent* trabalha no eixo vertical e o *alignItem*, no horizontal.

As opções do *alignItems* são: *flex-start*, *flex-end* e *center*, os quais têm a mesma funcionalidade que no *justifyContent*; o *baseline*, que alinha os itens da mesma forma que está declarada no contêiner pai; e o *stretch*, que estende o tamanho dos itens até o fim do contêiner.

Vale ressaltar que o *stretch* apenas funcionará se a largura ou a altura do item (dependendo do *flexDirection*) não estiver determinada.

```
StyleSheet.create({
  container: {
    flex: 1,
    flexDirection: "column",
    justifyContent: 'flex-start',
    alignItems: 'stretch'
  },
  blockA: {
    height: 60,
    backgroundColor: 'red'
  },
  blockB: {
    width: 60, height: 60,
    backgroundColor: 'green'
  },
  blockC: {
    width: 60, height: 60,
    backgroundColor: 'blue'
  }
})
```

Figura 6 – Exemplo: *alignItems*



## PARA SABER MAIS

Os atributos *flexDirection*, *justifyContent* e *alignItems* são os mais utilizados no desenvolvimento do React Native. Combinando os três valores, você conseguirá determinar o alinhamento de praticamente todos os componentes em um aplicativo, deixando-os preparados para todo tipo de tela. Porém, existem mais alguns atributos que são bem importantes, como *alignSelf*, *flexGrow* e *flexShrink*. Para saber mais sobre eles, vale a pena pesquisá-los no site do React Native.

## Considerações finais

Com o aprendizado deste capítulo, você já conseguirá aplicar um leiaute bem bacana utilizando o React Native. Com base nas dicas do JSX, nas propriedades do StyleSheet e em toda a força e flexibilidade



que o Flexbox proporciona, você agora é capaz de montar a tela de qualquer aplicativo que está no mercado. Aliás, falando em Flexbox, atente-se a como utilizar suas propriedades, pois elas indicam todos os caminhos para construir as telas.

Mas a perfeição, é claro, vem somente com a prática. Por isso, se inspire em alguns dos aplicativos mais bonitos e tente imaginar como suas telas foram montadas utilizando essas ferramentas. Em pouco tempo, você já conseguirá reproduzi-las.

## Referências

REACT. Introduzindo JSX. **React**, [s. d.]. Disponível em: <https://pt-br.reactjs.org/docs/introducing-jsx.html>. Acesso em: 1 abr. 2021.

REACT NATIVE. Layout with Flexbox. **React Native Dev**, [s. d.]. Disponível em: <https://reactnative.dev/docs/flexbox>. Acesso em: 1 abr. 2021.

# Componentes básicos do React Native

Como você deve ter notado durante nossos exemplos, um dos recursos mais importantes do React Native são os componentes presentes na biblioteca *react-native*. São eles que fazem a mediação entre o código JavaScript e o componente utilizado nativamente nos ambientes Android e iOS, e isso explica o porquê da grande fluidez que os aplicativos React Native apresentam.

Vamos falar sobre alguns desses principais componentes, como *TextInput*, *Image*, *ScrollView* e *Touchable*s. Mas, antes, vamos entender um pouco mais sobre o conceito de *props*, que ajuda muito no

compartilhamento de informações entre os componentes, e sobre o React Hooks, uma funcionalidade relativamente nova, mas que mudou completamente a forma de programar em React Native.

## 1 Utilizando a *props*

A *props* é uma ferramenta de fácil entendimento. Ela nada mais é do que uma variável passada como “parâmetro” quando utilizamos o componente ou uma biblioteca, sendo possível trabalhar com essa informação dentro desse componente.

Vamos criar um exemplo para isso ficar mais claro. Utilizando os conceitos do MVC (model-view-controller, em português: arquitetura modelo-visão-controle), criaremos dois componentes, um chamado *HomeController* e o outro, *HomeView*. No *HomeController*, vamos colocar todo o controle do componente e das informações recebidas e, no *HomeView*, somente os códigos relacionados à camada visual.

Para começar, nosso *HomeController* será:

```
import React from 'react';
import { registerRootComponent } from 'expo';
import HomeView from './HomeView';

const HomeController = () => {

  let information = "Teste Info";
  //Passando a variável information como a props info
  return <HomeView info={information}/>;
}

// Somente utilizamos a declaração registerRootComponent
// quando o componente for o determinado como o inicial no
// EntryPoint do Expo, que é o caso do HomeController.
// Caso não seja, usamos a linha "export default
HomeController";
export default registerRootComponent(HomeController);
```

E nosso *HomeView* será:

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

const HomeView = (props) => {

  //Utilizando a props info que recebemos na inicialização
  do componente
  return <View style={{marginTop: 130}}>
    <Text>{props.info} - 2</Text>
  </View>
}

export default HomeView;
```

No final, vamos alterar o arquivo *app.json* para adicionar a propriedade *entryPoint* e dizer qual o caminho do arquivo que o nosso aplicativo vai iniciar. No nosso caso, será: *src/Home/HomeController*.

```
"icon": "./assets/icon.png",
"entryPoint": "./src/Home/HomeController",
```



## IMPORTANTE

Não é obrigatório alterar o componente inicial do Expo, mas acreditamos ser uma boa prática colocar todos os códigos dentro da pasta *src* para deixá-los mais organizados. Por esse motivo, acabamos realizando essa alteração.

## 2 React Hooks

Nas primeiras versões do React Native, os componentes eram separados em dois tipos: os *stateful* e os *stateless*. A diferença entre eles era que o primeiro precisava obrigatoriamente ser construído utilizando classes, permitindo o acesso aos *states* (sobre os quais explicamos mais adiante), e precisava receber as informações dos ciclos de vida do componente (quando o componente é iniciado, quando é carregado, quando termina de carregar etc). O *stateless* era mais simples, declarado como funções; não permitia acesso aos *states* e ao ciclo de vida e, geralmente, recebia *props* para tratar as informações.

Porém, a partir da versão 0.59 do React Native, foi disponibilizada uma nova funcionalidade chamada React Hooks, na qual não era mais necessário decidir entre os dois tipos de componentes, porque todos os tipos teriam acesso aos *states* e aos ciclos de vida.

O React Hooks tem como objetivo facilitar a reutilização de lógica de estados entre componentes e a redução de componentes complexos (permitindo a separação em componentes menores). Mas a principal vantagem é que, utilizando somente funções, o aprendizado é muito mais fácil, pois não precisamos mais nos preocupar com as cláusulas *this* e *binds*, por exemplo.



### IMPORTANTE

Como o próprio React diz em sua documentação principal (REACT JS, s. d.), não há previsão de o modelo anterior ser retirado; assim, as duas versões podem ser utilizadas. Porém, o que percebemos é que, cada vez mais, os exemplos, tutoriais e os ReadMes das bibliotecas estão adotando a nova versão com o React Hooks.

Mas o que é uma hook? Segundo a documentação do React, “um hook é uma função especial que permite que você ‘se conecte’ às features do React” (REACT JS, s. d.). Porém, além das features do React, você pode criar as suas próprias features com os chamados Custom Hooks.

Agora, vamos nos aprofundar nas três principais hooks, que são as mais utilizados no dia a dia do desenvolvimento, e vamos ver como elas podem nos ajudar.

## 2.1 *useState*

Vamos começar pelo *useState*, porém, antes, precisamos entender o que é o conceito de *states*, que já foi mencionado, mas ainda não explicado. O *state* é o grande diferencial do React em relação às outras linguagens. Ele é uma variável declarada de uma forma especial, cujo valor, ao ser alterado, envia um comando para que o componente seja renderizado novamente na tela.

No *useState*, você deve passar um valor inicial, e ele retornará um *array* com duas posições: na primeira, uma variável com valor do *state* e, na segunda, uma função que possibilita atualizar o valor atual do *state*. Vamos inserir um *useState* no nosso *HomeController*:

```
//Importar o useState no componente
import React, {useState} from 'react';
import { registerRootComponent } from 'expo';
import HomeView from './HomeView';

const HomeController = () => {

  //Declarando o state information
  const [information, setInformation] = useState('');

  //função chamada no View
  const onClicked = (value) => {
    //Alterando a informação do useState
    setInformation(value);
  }
}
```

(Cont.)

```

    //Passando a variável information como o props info e a
    função onClicked
    return <HomeView
        info={information}
        onClicked={onClicked}
    />;
}
...

```

No exemplo apresentado, declaramos o *state information* por meio da hook *useState*, e alteramos o valor do *state* dentro da função *onClicked*, que enviamos ao *HomeView*. Ao chamar essa função, o componente será renderizado novamente, porque alteramos a informação do *information*.

## 2.2 *useEffect*

O *useEffect* utiliza um conceito chamado “operações de efeitos colaterais”, as quais são empregadas quando queremos usar um código adicional durante a atualização do DOM (document object model, ou modelo de documento por objetos) na tela. Imagine uma tela que exibe as informações de lugares próximos, conectando-se com uma API para buscar essas informações. Você somente vai chamar a API quando o seu componente for inicializado, e não toda vez que for necessário atualizar alguma informação na tela. Para isso, utilizamos o *useEffect*.

Outra função bem importante dos *useEffects* é receber os ciclos de vida do componente, que são enviados a eles sempre que há uma mudança. Vamos declarar alguns *useEffects* no nosso *HomeController* para entender como eles funcionam.

Preste atenção, ao final, no segundo parâmetro da hook *useEffect*, pois é ele que fará a diferença quanto ao momento em que a função será chamada.

```
//Importar o useState e o useEffect no componente
import React, {useState, useEffect} from 'react';
...
//Declarando o state information
const [information, setInformation] = useState('');

//Esse useEffect é invocado sempre que há uma renderização
do componente
useEffect(() => {
  console.log("Executa na renderização do componente");
  return () => {
    console.log("Executa antes de realizar a
renderização");
  }
});

//Esse useEffect é invocado sempre que o componente é
montado
useEffect(() => {
  console.log("Executa na montagem do componente");
  return () => {
    console.log("Executa na desmontagem do
componente");
  };
}, []);

//Esse useEffect é invocado sempre que o valor do count é
alterado
useEffect(() => {
  console.log("Executa na alteração do information");
  return () => {
    console.log("Executa antes de executar o render ao
alterar o valor do information");
  }
}, [information]);
```

(Cont.)



```

//função chamada na View
const onClicked = (value) => {
  console.log("Clicado no botão");
  //Alterando a informação do useState
  setInformation(value);
}

//Passando a variável information como a props info e a
função onClicked
return <HomeView
  info={information}
  onClicked={onClicked}
/>;
//Aqui ele é executado na desmontagem do
componente

```

Se olharmos o console do aplicativo (figura 1), veremos as informações que mostram todo o ciclo de vida desse componente, ao rodar na primeira vez, e ao alterar a informação do *state information*.

**Figura 1 – Console do aplicativo ao executar o *useEffects***

```

Executa na renderização do componente
Executa na montagem do componente
Executa na alteração do information
Clicado no botão
Executa antes de realizar a renderização
Executa antes de executar o render ao alterar o valor do information
Executa na renderização do componente
Executa na alteração do information

```

## 2.3 *useRef*

O *useRef* é utilizado para criar uma informação ou um objeto que precisa ser armazenado durante todo o ciclo de vida de um componente. Um uso muito comum do *useRef* é para armazenar uma referência de um componente ou instanciar classes utilizadas durante o *useEffect*.

```
import React, {useState, useEffect, useRef} from 'react';
...
const exampleModel = useRef(null)

//Esse useEffect é invocado sempre que o componente é
montado
useEffect(() => {
  console.log("Executa na montagem do componente");
  //Inicializando a classe Exemplo Model e chamando o
método doSomething
  exampleModel.current = new ExampleModel();
  exampleModel.current.doSomething();
  return () => {
    //Aqui ele é executado na desmontagem do
componente
    console.log("Executa na desmontagem do
componente");
  };
}, []);
```

Você pode perceber que, ao buscarmos ou alterarmos uma informação no *useRef*, utilizamos o *current* depois da variável. Utilizamos esse recurso para não correremos o risco de a informação ser duplicada durante o ciclo de vida ou para não pegarmos a informação errada.



## PARA SABER MAIS

Como falamos no início, você tem a opção de criar a sua própria hooks, que chamamos de Custom Hooks. Caso você queira saber mais sobre isso, há um vasto material no site de documentação do React.

## 3 Componentes do React Native

O React Native possui uma biblioteca padrão com vários componentes que nos ajudam a programar no dia a dia, utilizando os

mesmos componentes que são disponibilizados pelo sistema operacional quando desenvolvemos com as linguagens nativas dos devices.

### 3.1 *TextInput*

O *TextInput* nada mais é do que um campo de texto no qual o usuário insere informações por meio do teclado, como nome, e-mail, entre outras. Ele possui diversas configurações, como a autocorreção (*autoCorrection*), o texto de *placeholder*, se é um campo de senha (*secureTextEntry*) e se o teclado utilizado será customizado para aceitar somente números ou ser mais voltado à escrita de e-mails.

O *TextInput* possui, também, alguns métodos bem importantes, como o *onChangeText*, para receber a informação quando o texto é digitado, e o *onFocus*, quando o campo de texto recebe o foco.

```
return <View style={{marginTop: 130}}>
  <TextInput
    style={styles.input}
    onChangeText={props.onChangeNumber}
    value={props.value}
    placeholder="useless placeholder"
    keyboardType="numeric"
  />
</View>
```

### 3.2 *TouchableOpacity*, *TouchableHighlight* e *TouchableWithoutFeedback*

Os componentes *Touchable*s são utilizados para inserir toques nas views atreladas. Eles podem ser dos tipos: *TouchableOpacity*, que, quando pressionado, tira a opacidade das views; *TouchableHighlight*, que, quando pressionado, também tira opacidade das views, mas coloca uma cor de base para destacá-las; ou *TouchableWithoutFeedback*,

que permite o toque sem que se mexa no layout. Lembrando que é necessário ter ao menos uma view circundada pelos *Touchable*s para ele funcionar.

O método mais importante é o *onPress*, que serve para adicionar um método que será chamado ao tocarmos no componente.

```
return <View style={{marginTop: 130}}>
  <TouchableOpacity onPress={() => props.
onClicked('clicado')}>
    <Text>Click 1</Text>
  </TouchableOpacity>
  <TouchableHighlight onPress={() => props.
onClicked('clicado1')}>
    <Text>Click 1</Text>
  </TouchableHighlight>
  <TouchableWithoutFeedback onPress={() => props.
onClicked('clicado2')}>
    <Text>Click 1</Text>
  </TouchableWithoutFeedback>
</View>
```

### 3.3 Image

O componente *Image* permite a inserção de imagens de vários tipos, como imagens dos assets, imagens disponíveis na internet, imagens do disco local (como da galeria), entre outras.

```

return <View style={{marginTop: 130}}>
  <Image
    style={styles.tinyLogo}
    source={require('../../assets/icon.png')}
  />
  <Image
    style={styles.tinyLogo}
    source={{
      uri: 'https://reactnative.dev/img/tiny_logo.
png',
    }}
  />
  <Image
    style={styles.tinyLogo}
    source={{
      uri: 'data:image/
png;base64,iVBORw0KGgoAAAANSUhEUgAAADMAAAAzCAYAAAA6oTAqAP/
eMrC5UTVAAAAABJRU5ErkJggg==' ,
    }}
  />
</View>

```

### 3.4 SafeAreaView

O componente *SafeAreaView* permite que os itens dentro dele sejam renderizados nas áreas “seguras” do aplicativo, colocando automaticamente espaços em certas áreas – como barra de navegação, barra de status e limitações físicas das telas (os cantos arredondados de device, a área onde fica a câmera ou o botão *home*, no caso do iPhone X ou superior).

```

return <View style={styles.container}>
  <SafeAreaView style={styles.safe}>
    <View style={{backgroundColor: 'blue', flex: 1,
width: '100%'}}></View>
  </SafeAreaView>
</View>

```

### 3.5 *ImageBackground*

O componente *ImageBackground* permite inserir uma imagem de background de fundo na tela, fazendo com que todos os componentes dentro dele sejam colocados por cima da imagem. Ele possui as mesmas configurações e tipos do campo *Image* normal.

```
return <View style={styles.container}>
  <ImageBackground source={{
    uri: 'https://reactnative.dev/img/tiny_logo.png',
  }} style={styles.image}>
    <Text style={styles.text}>Inside</Text>
  </ImageBackground>
</View>
```

### 3.6 *ActivityIndicator*

O componente *ActivityIndicator* permite criar um indicador de atividade muito utilizado quando fazemos um processamento paralelo, como uma conexão com uma API ou uma operação no I/O no disco ou no banco de dados interno do aplicativo.

```
return <View style={styles.container}>
  <ActivityIndicator size="large" />
  <ActivityIndicator size="small" color="#0000ff" />
</View>
```

### 3.7 *Pressable*

Um dos componentes mais recentes, o *Pressable*, permite receber vários tipos de avisos relacionados ao toque na tela, por exemplo, quando o usuário inicia o toque (*onPressIn*) ou deixa de tocar (*onPressOut*), ou quando o usuário pressiona a tela rapidamente (*onPress*) ou faz um

toque com duração maior (*onLongPress*). Também é possível determinar o tempo de duração para que o *Pressable* nos informe se o toque feito é *onPress* ou *LongPress*, este último, utilizando a configuração *delayLongPress*; por padrão, o tempo para isso é de 500 milissegundos.

O *Pressable* também permite alterar o estilo e qualquer informação no botão quando o componente está ou não pressionado, passando uma variável booleana chamada *pressed* tanto no *style* quanto no seu *children*.

```
return (
  <View style={styles.container}>
    <Pressable
      onPress={() => {
        console.log("On Press")
      }}
      onLongPress={() => {
        console.log("On Long Pressed")
      }}
      onPressIn={() => {
        console.log("On Press In")
      }}
      onPressOut={() => {
        console.log("On Press Out")
      }}
      style={({ pressed }) => [
        {
          backgroundColor: pressed
            ? 'rgb(210, 230, 255)'
            : 'white'
        }
      ]}>
      {({ pressed }) => (
        <Text style={styles.text}>
          {pressed ? 'Pressed!' : 'Press Me'}
        </Text>
      )}
    </Pressable>
  </View>
);
```

## 3.8 ScrollView

O componente *ScrollView* permite que, quando vários itens são inseridos em uma tela e ultrapassam o tamanho vertical, automaticamente seja criada uma barra que realiza a rolagem até o fim do conteúdo. É importante dizer que, para o *ScrollView* funcionar, é necessário que você determine seu tamanho interno (não recomendado) ou que todos os itens possuam uma altura determinada para que esse componente consiga realizar a configuração.

```
return (  
  <SafeAreaView style={styles.container}>  
    <ScrollView style={styles.scrollview}>  
      <View style={styles.box100}></View>  
      <View style={styles.box100}></View>  
      <View style={styles.box100}></View>  
      <View style={styles.box100}></View>  
      <View style={styles.box100}></View>  
      <View style={styles.box100}></View>  
      <View style={styles.box100}></View>  
    </ScrollView>  
  </SafeAreaView>  
) ;
```



### IMPORTANTE

Apesar dos diversos tipos de componentes disponibilizados, vale reforçar que, como diz o site de documentação do React Native, “você não está limitado aos componentes e APIs carregados no React Native. O React Native tem uma comunidade de milhares de desenvolvedores” (2021, tradução nossa). Um bom lugar para procurar por essas bibliotecas extras é o site do GitHub, que possui milhares de bibliotecas disponíveis – sendo a grande maioria open source –, as quais podem ser instaladas utilizando o npm e o yarn.



## Considerações finais

As hooks e os componentes são a base de desenvolvimento dos aplicativos. Os exemplos que usamos neste livro são os básicos e mais utilizados no dia a dia, mas temos muitas outras opções, como as hooks *useMemo* e *useLayoutEffects*, ou os componentes *StatusBar*, *Switch* e *RefreshControl*. Por isso, vale a pena sempre ficar de olho na documentação do React e do React Native, a fim de aprender mais e ficar por dentro das novidades.

Outro ponto importante é que você deve ter percebido que o React e o React Native compartilham muitas features, como a *props*, o *states* e o React Hooks. Porém, tome cuidado, porque os componentes de visualização são bem diferentes entre eles. Aliás, essa é a grande diferença entre os dois.

E, como sempre, a perfeição vem com a prática. Então, agora é a hora de você pensar em desenvolver um aplicativo próprio, ou tentar replicar uma tela de um aplicativo de que goste. Isso vai ajudá-lo muito a entender a dinâmica de construção de telas e a utilização de componentes.

## Referências

REACT JS. Introducing Hooks. **React Docs**, 2021. Disponível em: <https://reactjs.org/docs/hooks-intro.html>. Acesso em: 21 abr. 2021.

REACT NATIVE. Core components and APIs. **React Native Docs**, 2021. Disponível em: <https://reactnative.dev/docs/components-and-apis>. Acesso em: 21 abr. 2021.

# Trabalhando com listas e componentes

Conforme avançamos no desenvolvimento do React, utilizamos muitas bibliotecas que facilitam bastante nosso trabalho com códigos prontos e já testados, os quais, é claro, permitem adicionar cada vez mais funcionalidades no nosso app.

Para começar, vamos aprender sobre o *PropTypes*, um verificador de tipos passados para um componente, que aponta erros caso esteja faltando alguma das propriedades requeridas. Também vamos aprender sobre dois componentes muito importantes, o *FlatList* e o *SectionList*, que ajudam a criar listas, amplamente utilizadas nos aplicativos, permitindo criar seções para separar os itens. Mas de que serve uma lista em

um aplicativo, se não conseguirmos criar páginas para ver os detalhes ou buscar mais informações na internet? Esses, portanto, serão nossos próximos passos, utilizando o Axios, que é uma biblioteca de conexão, e o React Navigation, que permite organizar a paginação no aplicativo.

## 1 *PropTypes*

Quando “inicializamos” um componente, devemos passar todas as propriedades necessárias para que ele consiga executar sua função. Mas quais propriedades são realmente necessárias? E como se certificar de que elas sejam enviadas e que os tipos estejam corretos?

Conforme está descrito na documentação do React (2021), existem algumas formas de realizar essa checagem, como o *Flow* e o *TypeScript*. Além dessas, existe uma forma que era nativa há até algumas versões do React e que, apesar de hoje estar separada em uma biblioteca, ainda é muito utilizada nos códigos; trata-se do chamado *PropTypes*.

Para começar a usar esse código, vamos instalar a biblioteca por meio do comando: `yarn add prop-types`. Analise o código a seguir:

```
import React from 'react';
import {View, Text} from 'react-native'
import PropTypes from 'prop-types';
const TestComponent = (props) => {
  //Multiplicar as variáveis
  let result = props.var1 * props.var2;

  //Montando o retorno com o resultado
  return (
    <View>
      <Text style={{marginTop: 50}}>
        Show result = {result}
      </Text>
    </View>
  )
}
```

(Cont.)

```

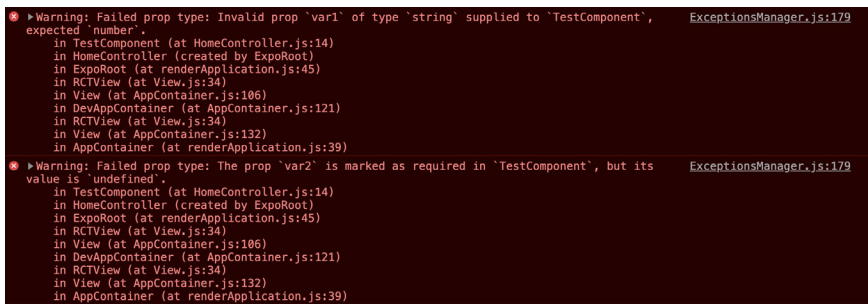
}

//Checagem das props enviadas
TestComponent.propTypes = {
  var1: PropTypes.number.isRequired,
  var2: PropTypes.number.isRequired
};
export default TestComponent;

```

Podemos perceber que estamos utilizando o *PropTypes* para nos certificar de que o componente, quando chamado, possua duas propriedades numéricas: *var1* e *var2*. Caso o componente seja chamado sem esses requisitos ou caso passemos uma string, receberemos um warning no console avisando que algo está errado (figura 1).

**Figura 1 – Console exibindo os erros no caso de os requisitos do *PropTypes* não serem cumpridos**



```

*Warning: Failed prop type: Invalid prop `var1` of type `string` supplied to `TestComponent`,
expected `number`. ExceptionsManager.js:179
    in TestComponent (at HomeController.js:14)
    in HomeController (created by ExpoRoot)
    in ExpoRoot (at renderApplication.js:45)
    in RCTView (at View.js:34)
    in View (at AppContainer.js:106)
    in DevAppContainer (at AppContainer.js:121)
    in RCTView (at View.js:34)
    in View (at AppContainer.js:132)
    in AppContainer (at renderApplication.js:39)

*Warning: Failed prop type: The prop `var2` is marked as required in `TestComponent`, but its
value is `undefined`. ExceptionsManager.js:179
    in TestComponent (at HomeController.js:14)
    in HomeController (created by ExpoRoot)
    in ExpoRoot (at renderApplication.js:45)
    in RCTView (at View.js:34)
    in View (at AppContainer.js:106)
    in DevAppContainer (at AppContainer.js:121)
    in RCTView (at View.js:34)
    in View (at AppContainer.js:132)
    in AppContainer (at renderApplication.js:39)

```

O *PropTypes* também permite que seja declarado um valor padrão a uma propriedade, e, caso você não passe o valor dessa propriedade na utilização do componente, será utilizado esse valor padrão. Para isso, adicionaremos o código logo abaixo da declaração do *PropTypes*:

```

TestComponent.defaultProps = {
  var1: 1,
  var2: 1
}

```

Essas definições ajudam muito não somente a evitar problemas e bugs, mas também a saber facilmente, quando se utiliza um componente de outra pessoa, o que deve ser passado, para que tudo funcione normalmente sem a necessidade de ficar verificando o código. Quando trabalhamos com o React, em que tudo acaba virando um componente, isso ajuda muito no desenvolvimento.



### IMPORTANTE

O *PropTypes* permite a definição de diversos tipos de variáveis, desde as mais simples e básicas (como número, string e booleana) até as mais complexas (como objetos, tipos das propriedades no objeto enviado, array, funções e até mesmo se deve haver uma *children*). Para se aprofundar nesse assunto e conhecer todos os tipos, vá até a página do GitHub na biblioteca *PropTypes*.

## 2 *FlatList* e *SectionList*

O *FlatList* é um componente do React Native que permite criar uma lista de itens que, caso estes ultrapassem o tamanho determinado, gera uma barra de rolagem. Como podemos perceber nos aplicativos, esse é um recurso muito utilizado para exibir as informações, pois as disponibiliza de uma forma fácil e rápida.

Você pode estar se perguntando: mas o *ScrollView* não faz a mesma coisa? Sim, é verdade, o *ScrollView* também cria uma lista de componentes e gera rolagem, porém a grande diferença entre eles é a performance. Enquanto o *ScrollView* já carrega todos os itens de uma única vez, o *FlatList* somente carrega os itens que serão exibidos na tela, e ainda limpa a memória. Então, quando temos 50 itens para serem exibidos em uma lista, por exemplo, o *FlatList* apresenta uma performance muito superior.

No dia a dia, acabamos usando o *ScrollView* apenas para montar um componente que pode ser maior que o tamanho da tela do celular, gerando a necessidade de uma rolagem. Quando trabalhamos com itens repetidos, o melhor caso é utilizar o *FlatList*, porque, além de sua performance, podemos utilizar alguns outros recursos atrelados a ele, como o *RefreshList*.

Vamos analisar o código:

```
import React, {useState} from 'react';
import { SafeAreaView, FlatList, Text, TouchableOpacity,
RefreshControl } from 'react-native';

const FlatListExample = (props) => {
  const [selectedID, setSelectedID] = useState(0);
  /* Nessa função, colocamos o JSX equivalente para cada
item da lista */
  const renderItem = ({ item, index }) => {
    let backgroundInfo = { backgroundColor: '#edffff'
};
    if (item.id === selectedID) {
      backgroundInfo = { backgroundColor: '#ff0000'
};
    }
    return (
      <TouchableOpacity onPress={() =>
setSelectedID(item.id)}
        style={[{ padding: 20 }, backgroundInfo]}>
        <Text style={{ fontSize: 32, }}>{item.
title}</Text>
      </TouchableOpacity>
    );
  }
  /* Colocamos o SafeArea para somente utilizar as áreas
clicáveis do celular
e flex:1 para utilizar a tela inteira */
  return (
    <SafeAreaView style={{ flex: 1 }} >
      <FlatList
```

(Cont.)

```

        //No data, colocamos o array com os itens
a serem exibidos
        data={props.itens}
        //No renderItem, colocamos a função onde
será exibido o item
        renderItem={renderItem}
        //No Keyextractor, colocamos o valor único
para cada item
        keyExtractor={item => item.id}
        //No extradata, colocamos as informações
extras que atualizam o FlatList
        extraData={selectedID}
        //No refreshControl, colocamos o componente
para atualizar a tela
        refreshControl={
            <RefreshControl
                refreshing={props.refreshing}
                onRefresh={() => props.onRefresh()}
            />
        }
    />
</SafeAreaView>
);
}
export default FlatListExample;

```

O código para usar o *FlatList* é bem simples e fácil, mas é preciso destacar alguns pontos:

- É possível criar uma função dentro do *RenderItem* e fazer a codificação lá dentro, mas isso não é muito indicado, porque o código fica bem confuso. O ideal é deixar uma função separada somente para essa renderização.
- O *Scroll* não preserva as variáveis internas dentro do *RenderItem*. Então, caso você precise dessa informação, o ideal é usar alguma ferramenta que permita o armazenamento, como o próprio *array*, o *useState*, o *useRef* ou o *React-Redux*, do qual falaremos mais adiante.

- Como afirma a documentação do React Native, o *FlatList* “é um *PureComponent*, o que significa que ele não será atualizado se suas props permanecerem iguais” (REACT NATIVE, 2021, tradução nossa). Isso significa que, mesmo alterando o valor do *useState* no exemplo anterior, a lista não será atualizada. Para forçar a atualização, utilizamos a propriedade *extraData* e passamos o state.
- O componente *RefreshList* utilizado no exemplo é bem interessante, pois permite executar uma função para atualizar a lista somente arrastando os itens de cima para baixo, o que é o comportamento padrão nos sistemas operacionais. Caso você passe *true* para a propriedade *refreshing*, ela exibirá um *ActivityIndicator* rodando no topo da tabela.



## NA PRÁTICA

Sempre que trabalhamos com o React, se inserimos vários itens semelhantes na tela (utilizando o *FlatList*, o *ScrollView* ou até mesmo inserindo diretamente um *array* de itens no JSX), será apresentado um warning no console para alertar que existem itens duplicados. Esse warning parece não ser importante, mas podem acontecer erros muito estranhos no aplicativo relacionados aos itens duplicados. Por isso, uma dica bem importante é sempre colocar um id único em casos como esses, e, se você estiver usando o *FlatList*, utilizar a propriedade *keyExtractor* para realizar essa alteração.

---

O *SectionList* é um componente bem parecido com o *FlatList*, mas ele tem uma funcionalidade a mais: separar os itens por seções, como o próprio nome diz. Vamos analisar o código:



```

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

const SectionListExample = (props) => {
  /* Observe que cada item é um objeto com a data dentro */
  const data = [
    {
      title: "Eletrodomésticos",
      data: [
        { id: 1, title: "Geladeira" },
        { id: 2, title: "Fogão" },
        { id: 3, title: "Micro-ondas" }
      ],
    },
    {
      title: "Eletrônicos",
      data: [
        { id: 4, title: "TV" },
        { id: 5, title: "Videogame" },
        { id: 6, title: "Notebook" }
      ]
    },
  ];

  /* Nessa função, colocamos o JSX equivalente para cada
  item da lista */
  const renderItem = ({ item, index }) => {
    return (
      <View style={{padding: 20, backgroundColor:
        '#edffff'}}>
        <Text style={{fontSize: 32,}}>{item.title}</
Text>
        </View>
      </View>
    );
  }

  const renderSection = ({ section: { title } }) => {
    return (
      <Text style={{fontSize: 40, margin:
        10}}>{title}</Text>
    );
  }
}

```

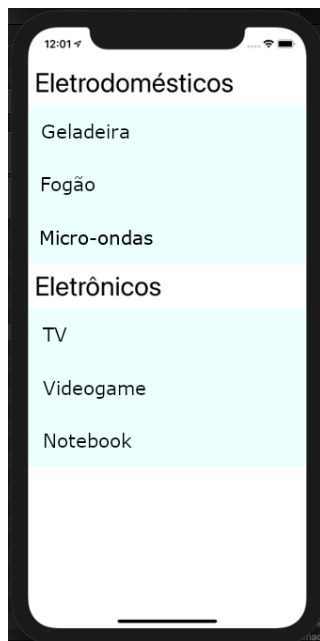
(Cont.)

```
return (  
  <SafeAreaView style={{ flex: 1 }} >  
    <SectionList  
      //No sections, colocamos o array com os itens a  
      serem exibidos  
      sections={data}  
      // No renderItem, colocamos a função onde será  
      exibido o item  
      renderItem={renderItem}  
      //No Keyextractor, colocamos o valor único para  
      cada item  
      keyExtractor={(item, index) => item + index}  
      renderSectionHeader={renderSection}  
    />  
  </SafeAreaView>  
);  
}  
export default SectionListExample;
```

No *SectionList*, o segredo é como montamos o *array* das informações. Enquanto no *FlatList* você pode passar um *array* com os objetos independentemente de como estão suas propriedades, no *SectionList* você também precisa passar um *array* de objetos, mas, dentro dele, é preciso ter ao menos duas propriedades: uma com o nome da section e outra com um *array* de objetos, cuja propriedade deve se chamar *data*.

Rodando o código anterior, o aplicativo ficará conforme a figura 2.

Figura 2 – Captura de tela com a lista de itens e os *SectionLists*



### 3 React-Navigation

A navegação entre as telas do aplicativo é uma funcionalidade essencial, pois precisamos gerenciar os vários fluxos de tela, além de enviar informações entre uma tela e outra. Para realizar essa funcionalidade, precisamos utilizar a biblioteca *React-Navigation*.

O primeiro passo é realizar a instalação, digitando os comandos no terminal ou prompt de comando: `yarn add @react-navigation/native` e, na sequência, `yarn add @react-navigation/stack`. Logo depois, precisamos instalar algumas bibliotecas de suporte à react-navigation. Para isso, vamos rodar o comando no terminal:

```
expo install react-native-gesture-handler react-native-reanimated react-native-screens react-native-safe-area-context @react-native-community/masked-view
```

O conceito principal do React-Navigation é a *stack*, ou pilha, a qual se comporta – assim como os conceitos de pilha no desenvolvimento de software – como uma fila na qual o primeiro a entrar é o último a sair. Portanto, quando se coloca um componente na tela, ele ficará na primeira posição da fila; quando o próximo componente é chamado, ele ficará visível na tela (primeira posição), e o componente anterior irá para a segunda posição. Ao chamarmos o comando de voltar, ele retira o componente que está na primeira posição, e o componente que está na segunda posição vai para o início da fila e, conseqüentemente, é exibido na tela.

Compreendido esse conceito, temos de optar por um dos três modelos de organização de telas disponíveis no *React-Navigation*: o *drawer* (menu lateral), o *tabs* (abas) ou simplesmente a disposição de todas as telas na mesma *stack*.

O *drawer* não é propriamente o menu de navegação, mas ele separa várias *stacks* em grupos e permite que você navegue entre esses conjuntos de *stacks* sem colocá-los em pilha, ou seja, sem colocar um menu *voltar*. Como você deve ter notado, esse é o modo como os aplicativos que possuem um menu lateral funcionam: dentro de cada *stack*, há as telas que vão e voltam, e, no menu, você pode navegar entre essas várias *stacks* sem que elas necessariamente estejam interligadas.

Já o *tabs* é muito parecido com o *drawer*, mas utiliza o conceito de abas, as quais geralmente ficam na parte inferior do aplicativo.



## PARA SABER MAIS

Você pode notar que instalamos a biblioteca `@react-navigation/stack` logo após o `@react-navigation/native`. Isso ocorreu porque somente utilizaremos a *stack* para o nosso exemplo. Segundo o site React Navigation (2021a; 2021b), caso você deseje utilizar o *drawer* ou o *tab*, será necessário instalar, respectivamente, o `@react-navigation/drawer` e o `@react-navigation/bottom-tabs`. Para mais informações, acesse a página de documentação do React Navigation.

Vamos, então, criar um exemplo simples somente gerando uma stack de telas. Para isso, vamos criar o *App.js*, que realizará a organização do menu, conforme o código:

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

/* Componente com a tela 1, recebendo o parâmetro
navigation */
function Screen1({ navigation }) {
  return (
    <View style={{ flex: 1, alignItems: 'center',
justifyContent: 'center' }}>
      <Text>Screen 1</Text>
      {/* Vai para a tela 2 */}
      <Button title="Go To Screen 2" onPress={() =>
navigation.navigate('Screen2')} />
    </View>
  );
}

/* Componente com a tela 2, recebendo o parâmetro
navigation */
function Screen2({ navigation }) {
  return (
    <View style={{ flex: 1, alignItems: 'center',
justifyContent: 'center' }}>
      <Text>Screen 2</Text>
    </View>
  );
}

const Stack = createStackNavigator();

function App() {
```

(Cont.)

```

    /* Abaixo criamos um contêiner navigation com uma
    stack com as telas 1 e 2 */
    return (
      <NavigationContainer>
        <Stack.Navigator>
          <Stack.Screen name="Screen1"
component={Screen1} />
          <Stack.Screen name="Screen2"
component={Screen2} />
        </Stack.Navigator>
      </NavigationContainer>
    );
  }
  export default App;

```

É importante destacar alguns pontos do código anterior:

- Sempre que você chama um componente por meio do *Stack.Screen*, ele passa como propriedade a instância de *navigation*, que é utilizada para chamar outra tela ou para voltar à tela anterior.
- Para chamar uma tela, usamos o *navigation.navigate* e passamos como parâmetro o nome do componente, que é o mesmo que está na propriedade *name* do *Stack.Screen*.

Mas como podemos fazer para passar informações de uma tela a outra? Para isso, vamos passar essa informação como parâmetro no *navigate* e, no outro componente, vamos receber uma nova *props*, o *route*, que receberá a informação. Nesse caso, nosso código será:

```

/* Componente com a tela 1, recebendo o parâmetro
navigation */
function Screen1({ navigation }) {
  return (
    <View style={{ flex: 1, alignItems: 'center',
justifyContent: 'center' }}>

```

(Cont.)

```

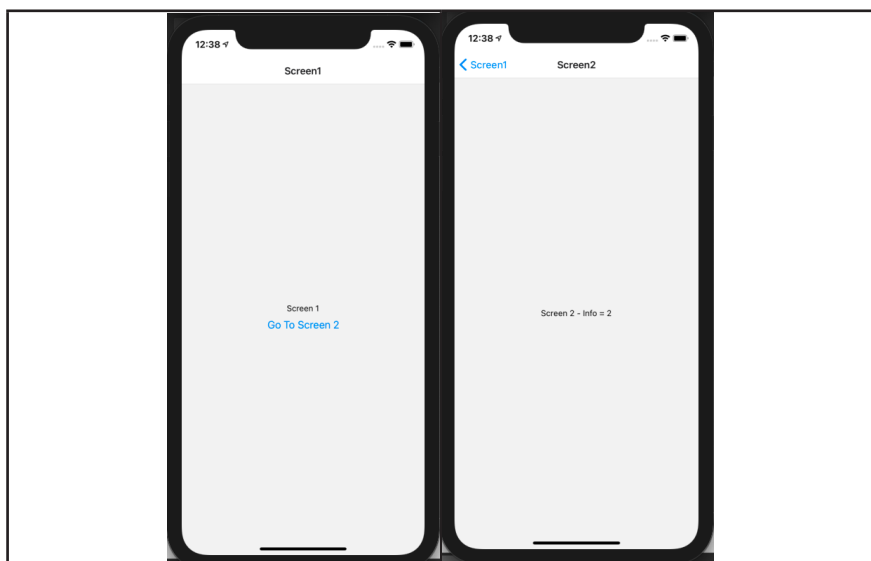
        <Text>Screen 1</Text>
        /* Vai para a tela 2 */
        <Button title="Go To Screen 2" onPress={() =>
navigation.navigate('Screen2', {info: 2})} />
        </View>
    );
}
/* Componente com a tela 2, recebendo o parâmetro
navigation */
function Screen2({ navigation, route }) {

    const {info} = route.params;
    return (
        <View style={{ flex: 1, alignItems: 'center',
justifyContent: 'center' }}>
            <Text>Screen 2 - Info = {info}</Text>
        </View>
    );
}
}

```

Nosso aplicativo ficará conforme a figura 3.

**Figura 3 – Realizando a navegação de telas**



## 4 Consumindo fonte de dados

Hoje, vivemos em um mundo altamente conectado e, por isso, praticamente não existem mais aplicativos que não estejam ligados a servidores por meio das já famosas APIs, sendo um servidor próprio ou de um serviço na internet, como as APIs do Google, do Firebase, do Facebook, entre outras.

Para realizar essa conexão, o JavaScript possui duas principais bibliotecas: a Fetch e a Axios. A Fetch, nativa, é a mais simples e funciona bem para conexões sem muitos requisitos. Já a Axios é uma biblioteca mais robusta, com uma série de features que a Fetch não possui, como realizar acessos simultâneos, verificar o progresso de download, fazer a transformação automática em JSON, entre outros.

Por esse motivo, vamos adotar a Axios no nosso desenvolvimento. Para isso, o primeiro passo é instalar a biblioteca usando o seguinte comando no terminal ou prompt de comando: *yarn add axios*.

Uma forma bem interessante de utilizar a Axios é criando uma service que faz a configuração inicial da conexão, determinando a URL principal e/ou os headers necessários para a conexão. E então, sempre que for chamar a Axios, você utiliza esse service já pronto.

Vamos à criação desse service:

```
import axios from "axios";

const api = axios.create({
  baseURL: "https://api.myserver.com",
});

export default api;
```

Agora, vamos criar o arquivo que realizará a conexão:



```

import react, {useEffect} from 'react';
import api from '../services/api'

const TestAxios = () => {
  useEffect(() => {
    api.get("/users/")
      .then((response) => doSomething(response.
data))
      .catch((err) => {
        console.error("ops! connection error" +
err);
      });

    // Enviando informação de posts para o servidor.
    const response = await api.post("/posts", { title:
title });
    }, [])

    return (
      <Text>
        Connection example
      </Text>
    )
  }
}

export default TestAxios;

```

Você pode perceber que chamamos a conexão dentro do *useEffect*, que é inicializado na construção do componente. Isso é bem importante, pois evita que a conexão seja realizada sempre que um componente for alterado.

## Considerações finais

Nunca é demais falar que o React Native funciona muito voltado ao desenvolvimento por componentes, separando os códigos em funções

para cada tipo de funcionalidade; nesse ponto de vista, se encaixam perfeitamente as bibliotecas. Por isso, ao desenvolvermos, muitas vezes recorreremos à instalação de bibliotecas parceiras ou do próprio React, o que tem um lado bom e um lado ruim.

O lado bom é que sempre temos uma enorme comunidade para trazer bibliotecas prontas, cada vez mais atualizadas e com melhor performance. Mas esse também é o ponto ruim, pois é preciso ficar sempre atento a atualizações, já que, se o código fica muito tempo desatualizado, é necessário entrar em cada uma das bibliotecas para ver quais são os procedimentos de migração de versão, o que pode ser meio trabalhoso, principalmente se há muitas versões para migrar.

Neste capítulo, começamos justamente a utilizar as primeiras bibliotecas, como PropTypes, React-Navigation e Axios. Todas essas bibliotecas têm em comum suas funções especialistas em verificar as propriedades enviadas, a navegação de telas e a busca de informação na internet.

## Referências

REACT. Typechecking with PropTypes. **React Docs**, 2021. Disponível em: <https://reactjs.org/docs/typechecking-with-proptypes.html>. Acesso em: 1 maio 2021.

REACT NATIVE. FlatList. **React Native Docs**, 2021. Disponível em: <https://react-native.dev/docs/flatlist>. Acesso em: 1 maio 2021.

REACT NAVIGATION. Tab Navigation. **React Navigation Docs**, 2021a. Disponível em: <https://reactnavigation.org/docs/tab-based-navigation>. Acesso em: 1 maio 2021.

REACT NAVIGATION. Drawer Navigation. **React Navigation Docs**, 2021b. Disponível em: <https://reactnavigation.org/docs/drawer-based-navigation>. Acesso em: 1 maio 2021.

# Trabalhando com geocoordenadas, mapas e notificações

Sempre é bom relembrar como os smartphones mudaram as nossas vidas nos últimos anos, incluindo sensores e funcionalidades que, apesar de já existirem há algum tempo, passaram a ser integradas em um único aparelho. Duas dessas principais features são o GPS e a câmera do celular; com o boom de aplicativos, podemos aproveitá-las ainda mais.

O bom e velho GPS, além de mostrar os melhores e/ou mais rápidos caminhos, passou a incluir diversos recursos, como a possibilidade de alguém compartilhar sua posição com familiares, de avisar a escola do

filho que está chegando para buscá-lo ou de compartilhar se há muito trânsito ou algum problema em uma via. Já as câmeras, por meio dos aplicativos, hoje incluem recursos como realidade aumentada e filtros.

E são esses recursos que vamos aprender neste capítulo, visando dar a você a liberdade de pensar em mais funcionalidades inventivas com essas features.

## 1 Como buscar as informações das geocoordenadas

Os smartphones permitem buscar as posições dos usuários com o aplicativo aberto (foreground) ou em segundo plano ou fechado (background). Para isso, eles utilizam o GPS ou a rede 3G com triangulação de antenas e retornam as coordenadas de latitude e longitude, que são únicas para cada lugar do mundo.

Após receber essa posição, podemos pedir para que o sistema operacional nos avise sempre que há uma modificação significativa do usuário, para que possamos atualizar a posição dele em um mapa ou desempenhar uma ação.

Para obter essa posição, utilizaremos uma biblioteca que já se comunica com esses sensores e obtém essa informação de forma muito fácil. Por isso, o primeiro passo é instalar essa biblioteca, a *expo-location*, por meio dos seguintes comandos: *expo install expo-location*.

Conforme está descrito na documentação da *expo-location*: “em aplicativos gerenciados ou bare, o *Location* requer o *Permission.Location*” (EXPO, 2021a, tradução nossa). Ou seja, antes de acessar a posição do usuário, precisamos que ele nos dê permissão, seja em foreground ou background. Na verdade, em background há a necessidade de mais permissões, que estão descritas no site da biblioteca (EXPO, 2021a).

Para entender como funciona todo esse processo, vamos analisar o código:

```
import React, { useState, useEffect } from 'react';
import { Text, View } from 'react-native';
import * as Location from 'expo-location';

export default function ExampleGeoCoordinates() {

  //Criando os states para buscar a informação
  const [position, setPosition] = useState(null);
  const [locationMessage, setLocationMessage] = useState('')

  //Vamos buscar a permissão e pegar a posição ao abrir a
  primeira vez o componente
  useEffect(() => {
    //Fazendo uma requisição assíncrona no useEffect
    (async () => {
      //Verifica se o usuário já deu a permissão e, caso
      não tenha dado, solicita a permissão
      let {status} = await Location.
requestForegroundPermissionsAsync();
      //Retorna o erro
      if (status !== 'granted') {
        setLocationMessage('Permissão negada');
        return;
      }

      //Com a permissão em ordem, busca a posição do
      usuário assincronamente
      let currentPosition = await Location.
getCurrentPositionAsync({});
      setPosition(currentPosition);
    })();
  }, []);
```

(Cont.)

```

    //Organiza o texto se está buscando
    let info = 'Waiting...';
    if(position){
        info = "Latitude = " + position.coords.latitude + " -
Longitude = " + position.coords.longitude +
        " - accuracy = " + position.coords.accuracy;
    } else if (locationMessage !== '') {
        info = locationMessage;
    }
    //Mostra o status/resultado na tela
    return (
        <View style={{ flex: 1}}>
            <Text style={{ fontSize: 25, color: 'red'
}}>{info}</Text>
        </View>
    );
}

```

Vale destacar alguns pontos importantes:

- O *Location.requestForegroundPermissionsAsync* busca as permissões necessárias para obter a posição do usuário com o aplicativo em foreground. Caso você precise buscar em background, utilize o comando *Location.getBackgroundPermissionsAsync()*.
- O método *Location.getCurrentPositionAsync* é o responsável por buscar as posições geográficas e pode ser chamado em qualquer ponto do código, como ao clicar um botão, por exemplo. Você pode passar como parâmetro um objeto com a propriedade *accuracy*, que determina a precisão da posição. Obviamente, localizações menos precisas têm tempo de resposta mais rápido, então vale a pena analisar o melhor custo-benefício para seu código.
- Outro método bem interessante é o *Location.getLastKnownPositionAsync*, que permite buscar a última posição conhecida do usuário. Ele é muito mais rápido que o *getCurrentPositionAsync* para buscar a posição, porém ela pode não estar atualizada. Como parâmetro,

you can pass an object with the *maxAge* property, with the maximum time in milliseconds that this location was obtained, and the *requiredAccuracy*, with the maximum radius of precision in meters. In both cases, if the last position is not within the maximum value passed, the location will return null.

Another very interesting way is to create a listener to notify us in case the user's position changes significantly. For this, we will use the *Location.watchPositionAsync* method, as follows:

```
Location.watchPositionAsync(  
    { accuracy: Location.Accuracy.Balanced, timeInterval:  
    1000, distanceInterval: 1 },  
    (location) => {  
        console.log(location)  
        setPosition(location);  
    })  
);
```

You can notice that, in the last parameter, a function will be passed that receives the location always updated, according to the user's movement. In the first parameter, an object is passed, in addition to the precision, it allows passing the *timeInterval*, with the minimum time interval, and the *distanceInterval*, with the minimum distance in meters for the function to be called.



## IMPORTANTE

A way to search for the position by expo-location has changed a lot in the latest version of Expo, due to changes in location permissions in Android and iOS systems. For this, follow two tips: it may be that you find some different versions of codes from the previous example, so make sure you are using version 41 of Expo. Another tip is to always keep an eye on the official Expo documentation to stay up to date.

## 2 Trabalhando com mapas

Não tem como trabalhar com a posição geográfica do usuário sem pensar em mapas. Afinal, não existe uma representação melhor da posição geográfica do que um pin em um mapa.

Para isso, utilizaremos a biblioteca React-Native-Maps, que simplifica muito o nosso trabalho. Para começar, vamos instalá-la utilizando o seguinte comando no terminal ou prompt de comando: *expo install react-native-maps*.

Vamos, então, colocar um mapa na tela utilizando o código:

```
import React from 'react';
import { View } from 'react-native';
import MapView, { Marker } from 'react-native-maps';
export default function ExampleGeoCoordinates() {

  //Função que obtém a informação de posição do marker
  alterado e coloca um
  //warning na tela
  const updatePosition = (coordinate) => {
    console.warn("Lat:" + coordinate.latitude + " Long: "
+ coordinate.longitude);
  }

  return (
    <View style={{flex: 1}}>
      /* Adicionando o mapa na tela inteira */
      <MapView style={{flex: 1}}>
```

(Cont.)



```

        //Colocando as informações de latitude e
longitude
        // e no delta a informação de aproximação do
mapa
        initialRegion={{
            latitude: -23.528766,
            longitude: -46.631241,
            latitudeDelta: 0.0092,
            longitudeDelta: 0.0091,
        }}
    >
    { /* Colocando um Marker no mapa */ }
    <Marker
        //determina que o marker é arrastável
        draggable
        //posição do marker
        coordinate={{ latitude: -23.528766,
longitude: -46.631241 }}
        //Título e descrição que aparecem quando
você toca no marker
        title={'Senac Tiradentes'}
        description={'Campus da Faculdade Senac
Tiradentes'}
        //Ao terminar de arrastar o marker, ele
chamará a função updatePosition
        onDragEnd={(e) => updatePosition(e.
nativeEvent.coordinate)}
    />
    </MapView>
</View>
);
}

```



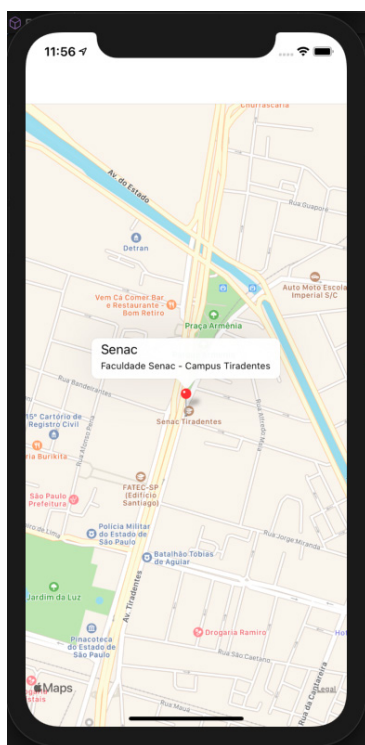
## PARA SABER MAIS

No código anterior, estamos trabalhando com os dois principais componentes do *react-native-maps*: o *MapView* e o *Marker*. Porém, essa biblioteca permite uma série de outras funções bem interessantes, como receber eventos do mapa, utilizar pins customizados, colocar polígonos

de acordo com os toques do usuário, entre outras. Para conhecê-las mais, vale a pena entrar na página da biblioteca *react-native-maps* no GitHub.

Nosso aplicativo ficará com a interface apresentada na figura 1.

**Figura 1 – Tela com o MapView**



Nos emuladores e no aplicativo de modo teste, nosso mapa funcionará sem problemas, mas, para enviá-lo para as lojas, é necessário gerar as chaves seguindo os passos, conforme descrito na página de documento do MapView do Expo (EXPO, 2021b).

## 2.1 Aplicativo Android

Vá ao Google Cloud API e registre um projeto. Na sequência, clique em “Ir para a visão geral de APIs” e selecione o botão “Ativar APIs e Serviços”. Escolha a opção “Maps SDK for Android” e clique em habilitar.

Deixe a impressão digital do certificado SHA-1 do seu aplicativo pronta. Se você estiver submetendo seu aplicativo para a Google Play Store, precisará criar um aplicativo standalone e enviá-lo para o Google Play ao menos uma vez, para que o Google gere as credenciais de assinatura do aplicativo. Para isso, siga estes passos:

1. vá a Google Play Console → (seu aplicativo) → Setup → App Integrity; e
2. copie o valor da impressão digital do certificado SHA-1.

Se você está instalando seu APK diretamente no device ou enviando-o para outra loja, precisa criar um aplicativo standalone e rodar o comando `expo fetch:android:hashes` no terminal ou prompt de comando, e copiar a impressão digital do certificado Google.

Criando uma chave de API:

1. vá para o gerenciador do Google Cloud Credential e clique em *Criar Credenciais e Chave de API*;
2. no modal, clique em *Restringir Chave*;
3. nas *Restrições de chave* → *Restrições de aplicativo*, assegure que a opção *Apps Android* esteja clicada;
4. clique em *Adicionar um item*, adicione o nome do pacote no *app.json* e coloque a impressão digital do certificado SHA-1 obtido no passo anterior;
5. clique em fechar e depois em salvar.

Para adicionar a Chave de API no seu projeto:

1. copie sua chave do aplicativo no campo *android.config.google-Maps.apiKey* do arquivo *app.json*; e
2. reconstrua o binário do aplicativo e envie-o para a Google Play, ou instale-o no device para testar se a configuração deu certo.

## 2.2 Aplicativo iOS

Vá ao Google Cloud API e registre um projeto. Na sequência, clique em “Ir para a visão geral de APIs” e selecione o botão “Ativar APIs e Serviços”. Escolha a opção “Maps SDK for Android” e clique em habilitar.

Criando uma chave de API:

1. vá para o gerenciador do *Google Cloud Credential* e clique em *Criar Credenciais e Chave de API*;
2. no modal, clique em *Restringir Chave*;
3. nas *Restrições de chave* → *Restrições de aplicativo*, assegure que a opção *Apps iOS* esteja clicada;
4. clique em *Adicionar um item*, copie o valor do campo *ios.bundleIdentifier* no arquivo *app.json* e cole-o no campo;
5. clique em fechar e depois em salvar.

Adicionar a chave da API no seu projeto:

1. copie sua chave do aplicativo no campo *android.config.google-Maps.apiKey* do arquivo *app.json*;
2. Em seu código, importe o `{PROVIDER_GOOGLE}` do *react-native-maps* e adicione a propriedade *provider=PROVIDER\_GOOGLE* para seu `<MapView>`. Essa propriedade funciona para iOS e Android; e

3. recrie seu aplicativo.

Uma maneira fácil de testar é fazer um build diretamente para o emulador.

## 3 Utilizando a câmera

Você consegue imaginar seu dia a dia sem a câmera do seu celular? Para tirar uma selfie, registrar um momento importante e até para escanear documentos, utilizamos a câmera quase diariamente.

O que vamos aprender agora é como buscar imagens do nosso device por meio da câmera, ou como buscá-las diretamente na galeria de fotos. Com a biblioteca *expo-image-picker*, isso fica muito fácil. Para isso, vamos instalar a biblioteca com o seguinte comando no terminal ou prompt de comando: *expo install expo-image-picker*.

Essa biblioteca possui dois métodos diferentes: o *launchImageLibraryAsync*, que busca as imagens na galeria de fotos, e a *launchCameraAsync*, que abre a câmera do celular. Nos dois métodos, precisamos passar um objeto com as propriedades do que é permitido nessa busca: o formato da imagem que queremos receber da biblioteca, se o usuário poderá editar a imagem, qual a qualidade da imagem que vamos receber e qual o tipo do arquivo que vamos buscar (imagem ou vídeo).

Antes de chamar essa biblioteca, precisamos solicitar ao usuário a permissão para acessar as fotos e/ou a câmera do celular dele. Para isso, utilizaremos os métodos *requestMediaLibraryPermissionsAsync* e *requestCameraPermissionsAsync*.

Vamos entender como utilizar a biblioteca no código:

```

import React, { useState, useEffect } from 'react';
import { Button, Image, View } from 'react-native';
import * as ImagePicker from 'expo-image-picker';

export default function ExampleCamera() {
  const [image, setImage] = useState(null);

  //State que roda ao montar o componente
  useEffect(() => {
    (async () => {

      //Checa permissão para acessar a biblioteca de
      fotos
      let infoLibrary = await ImagePicker.
      requestMediaLibraryPermissionsAsync();
      if (infoLibrary.status !== 'granted') {
        alert('Desculpe, precisamos da permissão para
        acessar a galeria');
      }

      //Checa permissão para acessar a câmera
      let infoCamera = await ImagePicker.
      requestCameraPermissionsAsync();
      if (infoCamera.status !== 'granted') {
        alert('Desculpe, precisamos da permissão para
        acessar a câmera');
      }
    })();
  }, []);

  //Função para controlar a busca da imagem
  const pickImage = async (type) => {

```

(Cont.)

```

        //Determina os parâmetros da imagem, como tipo de
        imagem permitido,
        //se permite editar a imagem, o formato da imagem e a
        qualidade
        let objectParams = {
            mediaTypes: ImagePicker.MediaTypeOptions.All,
            allowsEditing: true,
            aspect: [4, 3],
            quality: 1,
        }

        let result = null;
        if(type === 1){
            //Busca as imagens da galeria
            result = await ImagePicker.
            launchImageLibraryAsync(objectParams);
        } else {
            //Busca a imagem da câmera
            result = await ImagePicker.
            launchCameraAsync(objectParams);
        }
        if (!result.cancelled) {
            setImage(result.uri);
        }
    };

    return (
        <View style={{ flex: 1, alignItems: 'center',
        justifyContent: 'center' }}>
            <Button title="Abrir Câmera" onPress={() =>
            pickImage(2)} />
            <Button title="Abrir Galeria" onPress={() =>
            pickImage(1)} />
            {image && <Image source={{ uri: image }} style={{
            width: 200, height: 200 }} />}
        </View>
    );
}

```

Com essa biblioteca, fica muito fácil, não é verdade? Outra função interessante é que ela permite buscar várias fotos ao mesmo tempo

com a propriedade *allowsMultipleSelection* (no caso da galeria, é claro) e trazer a foto em *base64*, o que é muito útil quando precisamos enviá-la para algum servidor.

## 4 Notificações

As notificações dos aplicativos são ferramentas muito utilizadas para comunicação. Elas servem para avisar sobre o status de alguma funcionalidade, e ajudam no marketing e na retenção de clientes.

Em resumo, as notificações funcionam da seguinte maneira:

- No aplicativo, obtemos um token alfanumérico único para cada celular e o enviamos para os servidores. Esse token pode ser modificado caso ele expire ou caso o usuário desinstale o aplicativo.
- No servidor, enviamos uma mensagem para os servidores do Expo, que fica responsável por entregar a mesma mensagem aos sistemas operacionais.
- Ao receber a mensagem, o aplicativo, caso esteja aberto, é avisado; se o aplicativo estiver fechado, ele é avisado caso o usuário o abra tocando na notificação. Também é possível passar alguma informação na mensagem para que o aplicativo faça alguma ação específica.

Existem várias formas de enviar uma notificação, mas, como estamos abordando o Expo, utilizaremos o *ExpoPushToken*. Como ainda estamos em ambiente de desenvolvimento, vamos utilizar a ferramenta *Push Notification Tool*, que fará o envio da notificação. Mas, antes, precisamos obter o token único do nosso celular.

Para começar, vamos instalar a biblioteca *expo-notifications*. Para isso, rode o seguinte comando no seu terminal ou prompt de comando: *expo install expo-notifications*.



Agora, vamos analisar o código:

```
import Constants from 'expo-constants';
import * as Notifications from 'expo-notifications';
import React, { useEffect, useRef } from 'react';
import { View, Platform, Text } from 'react-native';

//Configura a notificação se o aplicativo estiver aberto
Notifications.setNotificationHandler({
  handleNotification: async () => ({
    shouldShowAlert: true,
    shouldPlaySound: true,
    shouldSetBadge: true,
  }),
});

export default function App() {
  const notificationListener = useRef();
  const responseListener = useRef();
  //Roda ao abrir o componente
  useEffect(() => {
    //Chama a função para registrar a notificação
    registerForPushNotificationsAsync().then(token => {
      console.log(token);
    });

    // Este listener é chamado quando uma notificação é
    recebida com o app em foreground
    notificationListener.current = Notifications.
    addNotificationReceivedListener(notification => {
      console.log(notification);
    });

    // Este listener é chamado quando o usuário toca ou
    interage com a notificação (funciona com o app em foreground,
    background ou fechado)
    responseListener.current = Notifications.
    addNotificationResponseReceivedListener(response => {
      console.log(response);
    });

    return () => {
```

(Cont.)

```

        //Remove os listeners das notificações
        Notifications.
removeNotificationSubscription(notificationListener.current);
        Notifications.
removeNotificationSubscription(responseListener.current);
    };
}, []);

//Função para registrar notificação
async function registerForPushNotificationsAsync() {
    let token;

    //Verifica se é um smartphone
    if (Constants.isDevice) {
        //Checa se o usuário permitiu notificações
        const { status: existingStatus } = await
Notifications.getPermissionsAsync();
        let finalStatus = existingStatus;
        if (existingStatus !== 'granted') {
            //Se não permitiu, solicita a notificação
            const { status } = await Notifications.
requestPermissionsAsync();
            finalStatus = status;
        }
        if (finalStatus !== 'granted') {
            alert('Falha ao obter o token da notificação
push');
            return;
        }

        //Busca o token único do celular
        token = (await Notifications.
getExpoPushTokenAsync()).data;
        console.log(token);
    } else {
        alert('Você precisa de um smartphone para receber
notificações');
    }
    if (Platform.OS === 'android') {

```

(Cont.)

```

        //Se for Android, configura os canais para receber
a notificação
        Notifications.
setNotificationChannelAsync('default', {
            name: 'default',
            importance: Notifications.AndroidImportance.
MAX,
            vibrationPattern: [0, 250, 250, 250],
            lightColor: '#FF231F7C',
        });
    }

    //Retorna o token
    return token;
}

return (
    <View
        style={{
            flex: 1,
            alignItems: 'center',
            justifyContent: 'space-around',
        }}>
        <Text> Testando Notificações </Text>
    </View>
);
}

```



## IMPORTANTE

O envio das notificações apenas funciona quando rodamos o aplicativo diretamente no smartphone.

Segundo o site de documentação do Expo, no Push Notification Setup, “se você está usando um aplicativo bare ou construindo um aplicativo standalone com *expo build:ios* ou *expo build:android*, precisará

configurar as credenciais push" (EXPO, 2021c, tradução nossa). Ou seja, para colocar nossos aplicativos na loja, mesmo usando o aplicativo gerenciado, precisamos obter os certificados da Apple e do Firebase. Para isso, siga os passos indicados nos itens a seguir.

## 4.1 Aplicativo Android

Se você ainda não criou um projeto Firebase, vá até essa plataforma e clique em *Criar um projeto*. Dê-lhe um nome e escolha entre habilitar ou não o Google Analytics. No console, em *Adicionar um app para começar*, clique em *Android* e siga os passos. Tenha certeza de que o nome do Pacote Android é o mesmo que está no campo *android.package* do arquivo *app.json*. Faça o download do arquivo *google-services.json* e coloque-o no diretório raiz do seu aplicativo Expo.

No *app.json*, adicione um campo *android.googleServicesFile* com o caminho relativo para o arquivo *google-services.json* cujo download você acabou de fazer. Se você colocou o arquivo no diretório raiz, o caminho será:

```
"android": {
  "googleServicesFile": "./google-services.json",
  "package": "br.com.senac.testexpo",
  "adaptiveIcon": {
    "foregroundImage": "./assets/adaptive-icon.png",
    "backgroundColor": "#FFFFFF"
  }
}
```

Confirme que a chave da sua API no *google-services.json* tem a API restrictions correta na aba API Credentials no console do Google Cloud Platform. Para a notificação push funcionar corretamente, o Firebase requer que a chave da API esteja unrestricted, ou que tenha acesso para a API do Firebase Cloud Messaging e a API do Firebase Installations. A chave da API pode ser encontrada no campo *client.api\_key.current\_key* do *google-services.json*.

```
"client": [  
  {  
    "api_key": [  
      {  
        "current_key": "AIzaSyA0L60at-4oz_Gcbr5_  
OjeE07C9sS6kIVY"  
      }  
    ]  
  },  
]
```

Finalmente, crie um novo build rodando o `expo build:android`.

Para que o Expo consiga enviar notificações dos seus servidores usando suas credenciais, você precisa enviar sua chave secret server. Você pode encontrar essa chave no console do Firebase do seu projeto:

1. no topo da barra lateral, clique no ícone de engrenagem à direita do *Visão geral do projeto* e, depois, em *Configurações do projeto*;
2. clique na aba *Cloud Messaging*;
3. copie o token ao lado da chave do servidor;
4. rode o comando `expo push:android:upload --api-key <your-token-here>`, trocando o `<your-token-here>` pela string do token que você acabou de copiar. O Expo vai armazenar seu token com segurança nos seus servidores, onde ele somente será acessado quando você enviar uma notificação push.

## 4.2 Aplicativo iOS

No caso de aplicativos iOS que utilizam o *managed workflow*, quando você roda o `expo build:ios`, as credenciais são geradas e organizadas automaticamente.



## PARA SABER MAIS

Para enviar mensagens do seu servidor para os celulares, além do token único, é preciso se conectar com os servidores do Expo. Para isso, visite a página *Enviando notificações com a API do Expo Push* a fim de obter o SDK (kit de desenvolvimento de software) e as instruções de como enviar as notificações diretamente do seu servidor.

---

## Considerações finais

Utilizar os sensores do device no nosso aplicativo ajuda muito a passar uma imagem de qualidade e a deixá-lo com uma aparência mais profissional. Mas tome cuidado com o uso desse recurso em excesso! Por exemplo, somente busque a posição do usuário quando isso for realmente necessário. A Lei Geral de Proteção de Dados (Lei nº 13.709/2018) está em vigor, e, nesse contexto, obter uma informação tão importante exige uma justificativa bem plausível.

Outro ponto significativo refere-se às notificações. Elas são uma ferramenta importante para o marketing, mas “bombardear” seus usuários com mensagens pode acabar tendo um efeito contrário, trazendo mais insatisfação do que uma visão positiva. Por isso, analise bem qual o melhor momento de enviar notificações e quais mensagens são relevantes para seus usuários. E, se possível, permita que cada um escolha quais tipos de mensagens deseja receber, a fim de criar um filtro para receber a comunicação que lhe agrada mais. Afinal, não queremos que o usuário apague nosso aplicativo do celular, certo?

## Referências

EXPO. Location. **Expo Docs**, 2021a. Disponível em: <https://docs.expo.io/versions/latest/sdk/location/>. Acesso em: 9 maio 2021.

EXPO. MapView. **Expo Docs**, 2021b. Disponível em: <https://docs.expo.io/versions/latest/sdk/map-view/>. Acesso em: 9 maio 2021.

EXPO. Push Notification Setup. **Expo Docs**, 2021c. Disponível em: <https://docs.expo.io/push-notifications/overview/>. Acesso em: 9 maio 2021.

# Utilizando o React Redux

Como você já sabe, se estamos desenvolvendo um aplicativo em React, o ideal é sempre componentizar o máximo possível dos nossos códigos, para deixá-los mais enxutos e fáceis de compreender. Mas como fazemos quando um componente precisar alterar a informação que está em outro componente, ou quando precisamos que uma informação seja compartilhada entre vários componentes (como o token de validação de conexão de um usuário)?

Para essa questão, que é muito comum quando realmente usamos a compartimentação do código, foram inventadas as ferramentas de



compartilhamento de estados, o React Redux. Com essas ferramentas, sempre que uma informação for alterada, todos os componentes receberão essa atualização.

Além disso, vamos entender também como funciona o Redux-Saga, que junta o React Redux com toda a força do saga, facilitando muito o nosso trabalho.

## 1 O que é o React Redux?

Ao entrar no site de desenvolvimento do React Redux e clicar em “Por que usar React Redux”, lemos a seguinte justificativa:

O Redux é uma biblioteca independente que pode ser usada com qualquer camada de UI ou framework, incluindo React, Angular, Vue, Ember e Vanilla JS. Apesar de o Redux e o React serem normalmente usados juntos, eles são independentes um do outro (REACT REDUX, 2021, tradução nossa).

Com isso, podemos perceber que o React Redux é, na verdade, uma adaptação da biblioteca original Redux. Mas como essa biblioteca foi criada? Em 2015, os irmãos Dan Abramov e Andrew Clark eram membros do core team do React e passaram pelas dificuldades que abordamos no parágrafo de abertura deste capítulo. Para resolver o problema, eles utilizaram como base o Flux, que era como o Facebook gerenciava a informação no início do desenvolvimento do React. Por isso, muitos dos conceitos do Flux foram absorvidos nesta biblioteca.

Agora, com base na documentação do Redux (2021), vamos nos aprofundar em seus três princípios fundamentais:

- **Fonte única de verdade (SSOT):** o estado global da sua aplicação é armazenado em um objeto dentro de uma store simples. O conceito de SSOT refere-se a uma prática de estrutura de modelo de informação em que o elemento de dados é armazenado uma

vez, e todo o acesso e a atualização são realizados somente por referência, facilitando a implementação, o debug e a inspeção de uma aplicação. Você também pode gravar o último status desse state inteiro em um servidor ou na memória local de um celular; com isso, fica fácil compartilhar a informação quando o usuário muda de um smartphone para outro (inclusive na web) ou reabre o aplicativo e este precisa voltar ao ponto em que parou.

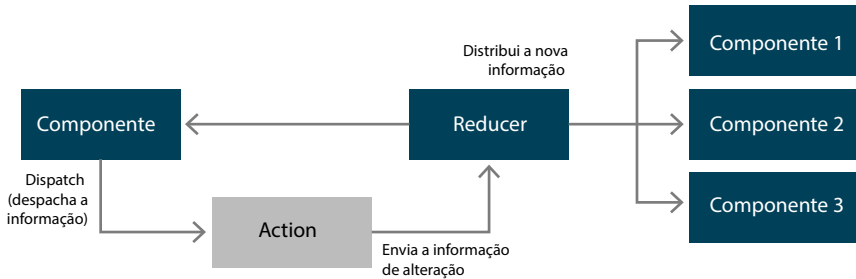
- **O estado é somente leitura:** a única maneira de mudar o state é chamando uma action com um objeto escrevendo seu tipo, garantindo que nenhum componente altere diretamente as informações do state. Como essas mudanças são centralizadas e ocorrem uma por vez, não há a possibilidade de sobreposição de informação.
- **Mudanças são realizadas por meio de funções puras:** para especificar como o state é alterado por actions, você escreverá reducers puros. Os reducers são funções puras que recebem o state anterior e a action, e retorna o state alterado. Você pode iniciar com um state simples e ir aumentando conforme o andamento do aplicativo.

Na prática, vamos criar para cada grupo de funcionalidade da nossa aplicação uma store, que organiza todo esse fluxo de armazenamento e troca de informação. Dentro de cada store, teremos dois arquivos:

- **Reducer:** é o arquivo que vai armazenar o state global daquela store e os reducers puros para alteração da informação.
- **Actions:** são os responsáveis por realizar o chamado das alterações no reducer. O action terá várias funções, e por meio dele serão passados os objetos com o tipo de alteração do reducer e as informações a serem alteradas.

A interação funciona do modo como apresentado na figura 1.

Figura 1 – Fluxograma React Redux



## 2 O Redux-Saga, colocando mais força no Redux

O Redux já é uma ferramenta importantíssima para o desenvolvimento, mas o Redux-Saga junta essa biblioteca com a força dos sagas e do ES6 function Generators, que são utilizados para organizar as chamadas assíncronas.

Os sagas disponibilizam alguns métodos, chamados effects, que nos auxiliam a organizar essas chamadas. Os mais utilizados com o Redux-Saga são:

- *take()*: pausa as operações até receber uma redux action;
- *call()*: executa uma função que, se possuir uma promise, pausa o saga até essa promise ser resolvida;
- *put()*: despacha uma redux action;
- *all()*: executa todas as chamadas, aguarda a execução e retorna todos os resultados; e
- *select()*: busca uma informação do estado global do Redux.

Para entender melhor todos os métodos, acompanhe o código a seguir:

```
//Atenção: um saga possui o '*' logo após o function
function* loadInfo() {
  try {
    // Aguarda até a Action START_PROMISSE ser executada
    yield take('START_PROMISE');

    // Busca a informação no Redux Principal
    const info = yield select(state => state.info);

    // Chama a função loadMainInfo e aguarda o
    processamento dela
    // para continuar processando o saga
    const mainInfo = yield call(loadMainInfo, info);

    // Rodamos as funções mainInfo2 e mainInfo3 e teremos
    a certeza
    // de que as duas funções serão executadas e
    receberemos as informações
    // O saga vai ficar aguardando o processamento das
    duas informações
    const [mainInfo2, mainInfo3] = yield all([
      call(loadMainInfo2, info),
      call(loadMainInfo3, info)
    ]);

    // Despacha a ação SAGA_RUN_SUCCESSFUL para o reducer
    // com as variáveis mainInfo, mainInfo2 e mainInfo3
    yield put({
      type: 'SAGA_RUN_SUCCESSFUL',
      payload: { mainInfo, mainInfo2, mainInfo3 }
    });
  } catch (error) {
```

(Cont.)

```

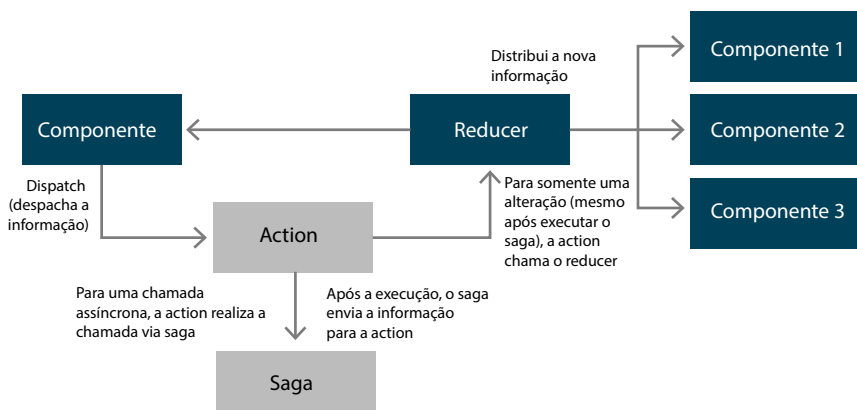
    });

    // Despacha a ação SAGA_RUN_SUCCESSFUL para o reducer
    // com as variáveis mainInfo, mainInfo2 e mainInfo3
    yield put({
      type: 'SAGA_RUN_SUCCESSFUL',
      payload: { mainInfo, mainInfo2, mainInfo3 }
    });
  } catch (error) {
    // Despacha a ação SAGA_RUN_FAILED para o reducer
    // com a variável error.message
    yield put({
      type: 'SAGA_RUN_FAILED',
      error: error.message
    });
  }
}
}

```

Na nossa organização de arquivos, colocaremos nossos sagas em um arquivo chamado *saga.js* junto os actions e reducers. Utilizando o Redux-Saga, nosso fluxo de informações ficará um pouco diferente, como podemos perceber na figura 2.

**Figura 2 – Fluxograma React Redux**



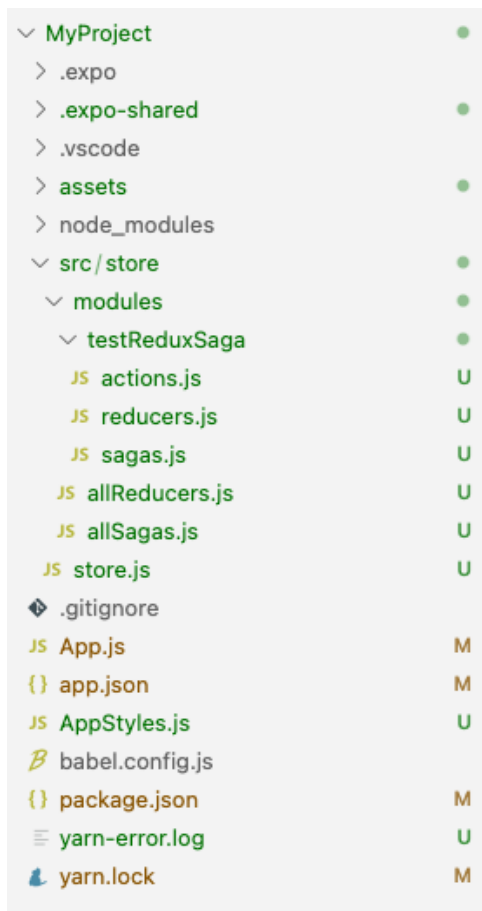
Podemos constatar que, com o Redux-Saga, todos os processamentos que exigem um processamento assíncrono (como uma API, uma gravação no BD interno ou um processamento mais demorado) deverão passar para o saga em vez de para o reducer diretamente, e, após a execução, chamarão a action para alterar as informações do reducer. Já os processamentos que somente vão alterar as informações do reducer continuarão no caminho normal, passando de action para reducer e compartilhando as informações.

### 3 Exemplo prático de Redux-Saga

Agora que já entendemos como funciona o conceito do Redux-Saga, vamos colocar a mão na massa e rodar um exemplo, pegando um projeto pronto ou criando um novo. Como sempre, o primeiro passo é rodar o seguinte comando no terminal ou prompt de comando para instalar as bibliotecas: `yarn add redux react-redux redux-saga immer`.

Agora, vamos criar a estrutura de pastas e arquivos para adequar o Redux-Saga. Criaremos uma pasta chamada *store* dentro da pasta *src*, onde ficará o arquivo de configuração *store* e a pasta *modules*. Nessa pasta, por sua vez, ficarão os outros arquivos de configuração do Redux-Saga (*allReducers.js* e *allSagas.js*) e, também, todos os módulos de que precisaremos. No nosso caso, será o módulo *testReduxSaga*, que terá os três arquivos já explicados anteriormente: *actions.js*, *reducer.js* e *saga.js*.

Figura 3 – Organização de arquivos React Redux



## IMPORTANTE

Vale lembrar que é superimportante separar os módulos de acordo com as suas funcionalidades. Então, em um aplicativo, podemos ter, por exemplo, o módulo de autenticação, o de pedidos ou o de busca de alguma informação específica. Essa divisão é importante para que o código não seja muito grande e fique mais fácil gerenciá-lo e encontrar erros. As informações entre os módulos podem ser acessadas no saga, utilizando o método select.

Agora, vamos colocar a mão na massa. Para começar, vamos criar um arquivo *store.js*:

```
/**
 * Realiza a configuração do React Redux e Redux-Saga
 */
//Importa os métodos e funções que usaremos no Redux-Saga
import createSagaMiddleware from 'redux-saga';
import { createStore, applyMiddleware } from 'redux';
//Importa os objetos com os reducers e os sagas
import allReducers from './modules/allReducers';
import allSagas from './modules/allSagas';
//Criando o Middleware Saga
const sagaMiddleware = createSagaMiddleware();
//Criando um array caso precisemos de mais middlewares
const middlewares = [sagaMiddleware];
//Criando a Redux Store
const store = createStore(allReducers, applyMiddleware(...
middlewares));
// Rodando redux-saga
sagaMiddleware.run(allSagas);
export default store
```

O store é um arquivo padrão de configuração, e dificilmente vamos alterá-lo. Agora, vamos juntar todos os reducers e sagas dos nossos módulos nos arquivos *allReducers.js* e *allSagas.js*; sempre que você criar um módulo, é necessário acrescentá-lo nesses arquivos. Para começar, vamos juntar todos os reducers no arquivo *allReducers.js*, utilizando o código:



```

/**
 * Objeto que junta todos os reducers que usaremos no
 aplicativo
 */

// Importa o combineReducers do redux
import { combineReducers } from 'redux';

//importa o reducer que está no módulo testReduxSaga
import testReduxSaga from './testReduxSaga/reducers';

//Junta os reducers
export default combineReducers({ testReduxSaga });

```

Agora, no *allSagas.js*, vamos juntar os sagas do nosso aplicativo, conforme o código abaixo:

```

/**
 * Objeto que junta todos os sagas que usaremos no aplicativo
 */
//Importa o all do redux-saga
import { all } from 'redux-saga/effects';

//Importa o saga dentro do módulo testReduxSaga
import testReduxSaga from './testReduxSaga/sagas';

//Junta todos os sagas importados
export default function* rootSaga() {
  return yield all([testReduxSaga]);
}

```

Para terminar nossa configuração, vamos alterar o primeiro arquivo do nosso projeto, que no nosso caso é o *App.js*, colocando o Provider do Redux-Saga, para que as informações do reducer possam ser acessadas em todos os componentes.

```

import React from 'react';
import { Text, View } from 'react-native';
import styles from './AppStyles';
//Importa o store
import store from './src/store/store';

//Importa o Provider do React Redux
import { Provider } from 'react-redux';

export default function App() {

  return (
    <React.StrictMode>
      { /* Adicionar o Provider com o store importado */ }
      <Provider store={store}>
        <View>
          <Text style={styles.titleText}>Hello World</Text>
        </View>
      </Provider>
    </React.StrictMode>
  )
}

```

Configurações feitas, vamos começar a programar nosso Redux-Saga. Primeiro, vamos criar os actions, que serão as funções chamadas pelo nosso componente para alterar o state ou chamar o saga.

```

/**
 * Objeto que reúne os actions que alteram as informações no
 reducer
 */
//Função que altera a informação do Info
export function setInfo(info) {
  console.log("Action changeInfo");
}

```

(Cont.)

```

    //Retorna o Redux Action testReduxSaga/GET_INFO_REDUCER e
    a nova informação do Info
    //para alterar o reducer
    return {
        type: 'testReduxSaga/GET_INFO_REDUCER',
        payload: { info },
    };
}
//Função que altera a informação do Info via saga
export function setInfoSaga() {

    console.log("Action getInfoSaga");
    //Retorna o Redux Action testReduxSaga/GET_INFO_SAGA
    //para chamar o saga
    return {
        type: 'testReduxSaga/GET_INFO_SAGA',
    };
}

```

Agora criaremos nosso primeiro saga, que chamará a função *Get/Info* e retornará uma informação após aguardar 3 segundos, simulando uma chamada assíncrona.

```

**
 * Objeto que reúne os sagas deste módulo
 */
//Importa os effects do Redux-Saga
import { takeLatest, call, put, all } from 'redux-saga/
effects';

// Importa as funções actions que serão chamadas pelo saga
import {
    setInfo,
} from './actions';

```

(Cont.)

```

//Função que retorna um texto após 3 segundos (somente para
teste)
function getInfo() {
    return new Promise((resolve) => {
        setTimeout(() => {
            resolve({
                info: "Info do Saga",
            });
        }, 3000);
    });
}
//Saga responsável por buscar a informação e alterar o reducer
function* getNewInfoSaga() {
    console.log("SAGA - getNewInfoSaga");
    //Chama o método assíncrono getInfo e retorna a informação
    após executar a informação
    const dataReturn = yield call(getInfo)

    console.log("SAGA - Retorno do Call GetInfo" + dataReturn.
info);
    //Chama a action ChangeInfo e despacha a ReduxAction
    yield put(setInfo(dataReturn.info));
}

//Junta todos os sagas deste objeto
export default all([
    //executa as operações recebidas e retorna o valor da
última
    // Atenção ao busInfo/GET_NEW_INFO, ele é o link chamado
no ACTION
    // para executar a Action
    takeLatest('testReduxSaga/GET_INFO_SAGA', getNewInfoSaga),
]);

```

No arquivo reducer, vamos colocar a variável *info* e a função que será responsável por alterar o state. Nessa função, vamos utilizar a biblioteca *immer* para juntar os states e gerar um novo.

```

/**
 * Objeto que organiza o reducer do módulo busInfo
 */
//Importa a função produce do immer
import produce from 'immer';
//Seta o state inicial
const INITIAL_STATE = {
  info: 'Info Inicial'
};
//Cria a função responsável por organizar o reducer
export default function getInfo(state = INITIAL_STATE, { type,
payload }) {
  //O producer é a uma função do immer que permite a
  //alteração do state de forma mais fácil utilizando o
  //draft. Após as alterações, somente é necessário
  //retorná-lo na função para que o reducer o entenda
  //como o novo state.
  return produce(state, (draft) => {
    switch (type) {
      //Verifica qual Redux Action foi chamada no action
      case 'testReduxSaga/GET_INFO_REDUCER': {
        console.log("REDUCER testReduxSaga/GET_INFO_
REDUCER = " + payload.info);
        //Altera a informação do Info no state
        draft.info = payload.info;
        break;
      }
      default:
    }
  });
}

```

Para rodar nosso aplicativo, vamos criar dois módulos separados, chamados *HomeComp1* e *HomeComp2*, e vamos utilizar o *useSelector* do Redux-Saga para ficar monitorando a variável *info* do nosso reducer. Em cada um deles, também vamos colocar um botão que chamará uma action. Vamos começar com o código *HomeComp1*, que chamará a action *setInfo*:

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

import { TouchableOpacity } from 'react-native';
//Importa as actions
import {
  setInfo,
} from './store/modules/testReduxSaga/actions';

//importa as funções useDispatch e useSelector do React Redux
import { useDispatch, useSelector } from 'react-redux';

export default function HomeComp1 (props) {

  //Busca a variável info do Reducer
  const info = useSelector((state) => state.testReduxSaga.
info);
  //Inicia o dispatch
  const dispatch = useDispatch();

  return (
    <View style={styles.mainContainer}>
      <Text style={styles.titleText}>{info}</Text>
      { /* Despacha a action setInfo */ }
      <TouchableOpacity style={styles.button}
onPress={() => dispatch(setInfo('Info do Reducer'))}>
        <Text> Alterar Reducer </Text>
      </TouchableOpacity>
    </View>
  )
}
```

E, no *HomeComp2*, vamos chamar a action *setInfoSaga*:

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

(Cont.)

```

import { TouchableOpacity } from 'react-native';
//Importa as actions
import {
  setInfoSaga,
} from './store/modules/testReduxSaga/actions';

//importa as funções useDispatch e useSelector do React Redux
import { useDispatch, useSelector } from 'react-redux';
export default function HomeComp2 (props) {

  //Busca a variável info do reducer
  const info = useSelector((state) => state.testReduxSaga.
info);
  //Inicia o dispatch
  const dispatch = useDispatch();

  return (
    <View style={styles.mainContainer}>
      <Text style={styles.titleText}>{info}</Text>
      { /* Despacha a action setInfoSaga */ }
      <TouchableOpacity style={styles.button}
onPress={() => dispatch(setInfoSaga('Info do Saga'))}>
        <Text> Alterar Saga </Text>
      </TouchableOpacity>
    </View>
  )
}

```

Para finalizar, no *App.js*, vamos chamar os dois componentes:

```

import React from 'react';
//Importa o store
import store from './src/store/store';

//Importa o Provider do React Redux
import { Provider } from 'react-redux';

import HomeComp1 from './src/HomeComp1';
import HomeComp2 from './src/HomeComp2';

```

(Cont.)

```
export default function App() {  
  
  return (  
    <React.StrictMode>  
      {/* Adicionar o Provider com o store importado*/}  
      <Provider store={store}>  
        <HomeComp1 />  
        <HomeComp2 />  
      </Provider>  
    </React.StrictMode>  
  )  
}
```

Nosso código ficará como a figura 4.

Figura 4 – Tela com os componentes montados





A grande diferença acontece quando algum dos botões é tocado. Independentemente do componente, a informação, ao ser atualizada, é replicada em ambos, o que é a grande força do React Redux. Se você clicar em *Alterar Reducer*, a informação é atualizada imediatamente, mas, se clicar em *Alterar Saga*, é realizada a chamada assíncrona que alterará a informação após 3 segundos. Se você abrir o console da página, conseguirá olhar o caminho da informação, conforme desenhamos na figura 5 e apresentamos no log do aplicativo.

Figura 5 – Resultado e caminho da informação





## PARA SABER MAIS

Neste capítulo, somente explicamos os conceitos básicos do Redux-Saga, os quais, porém, são utilizados na maioria das aplicações. Mas existem outras funções que não abordamos, como conexões concorrentes utilizando `takeLatest` e `takeEvery`, os forks e os spans, entre outras. Para conhecer mais, vale a pena consultar o site da documentação do Redux-Saga disponível na internet.

---

## Considerações finais

Utilizar as bibliotecas React Redux e Redux-Saga é algo que nos leva a pensar um pouco diferente sobre a forma como desenvolvemos uma aplicação em React. Mas, acredite, depois que você aprender a trabalhar com elas e a colocá-las em prática, não vai querer trabalhar de outra maneira.

Vale ressaltar que, para utilizar o Redux no React, não é obrigatória a utilização do Redux-Saga; pode-se somente utilizar o React Redux e seguir no primeiro fluxo. Porém, o Redux-Saga é uma biblioteca muito empregada nas aplicações de produção e facilita muito nosso trabalho, principalmente quando lidamos com acesso a APIs ou a banco de dados locais. E tudo fica tão integrado que praticamente colocamos toda a camada de modelo de negócio nos sagas e seguimos somente com a controller e a view nos componentes separados.

Com todo o conteúdo que estudamos durante a nossa jornada, você já está apto a criar aplicações profissionais. Agora, torcemos para que você desenvolva a próxima aplicação que mudará nosso mundo, como os aplicativos vêm fazendo na nossa vida.

## Referências

REACT REDUX. Why Use React Redux? **React Redux**, 2021. Disponível em: <https://react-redux.js.org/introduction/why-use-react-redux>. Acesso em: 9 maio 2021.

REDUX. Three Principles#. **ReDux Docs**, 2021. Disponível em: <https://redux.js.org/understanding/thinking-in-redux/three-principles>. Acesso em: 16 maio 2021.

## Sobre o autor

**José Rubens Rodrigues** possui experiência na área de TI desde 2001. Trabalhou, nos primeiros anos, com análise e desenvolvimento para mainframes em empresas do setor bancário, como Banco Real, Unibanco, Itaú e Alfa. Em 2005, passou a desenvolver aplicativos e games para celulares em uma multinacional do setor. Desenvolveu aplicativos para as plataformas BREW, J2ME, iOS e Android. Foi responsável pelo desenvolvimento de alguns dos aplicativos mais baixados da Vivo, como Vivo Cupido e Truco. Em 2010, cofundou a Intuitive Appz para desenvolver games e aplicativos para as plataformas mobile, web e PC, utilizando tecnologias como realidade aumentada e Kinect, em interações digitais para empresas como Foroni, Disney, Mattel, Volkswagen, MAN e Mercedes-Benz. Em 2013, cofundou a startup School Guardian, solução que traz segurança e agilidade na saída escolar, utilizada em mais de 300 escolas no Brasil, nos Estados Unidos, no Canadá, no Paraguai e no Uruguai, beneficiando mais de 80 mil alunos e 140 mil responsáveis, com mais de 3 milhões de chamadas realizadas. É responsável por toda a tecnologia da empresa, coordenando a equipe e desenvolvendo em React-Native, Kotlin, Swift, ReactJs, Node e PHP/Laravel. Desde 2019, atua como professor de empreendedorismo e inovação nos cursos de graduação e MBA da faculdade FIAP e como professor de tecnologias móveis no curso de TDS.