

React Native

Fundamentos

Por Dr. Mateus dos Santos



**INSTITUTO
FEDERAL**

Sul de Minas Gerais

React Native

Fundamentos

Por Dr. Mateus dos Santos

Material didático produzido pelo professor das disciplinas de Programação para Dispositivos Móveis dos cursos do IFSULDEMINAS Campus Poços de Caldas.

Dr. Mateus dos Santos
mateus.santos@ifsuldeminas.edu.br



<https://www.facebook.com/mateuscelular>



<https://twitter.com/mateuscelular>



(35) 9 9977 7799



**INSTITUTO
FEDERAL**

Sul de Minas Gerais

Sumário

Introdução	6
Preparação para Desenvolvimento	8
Conhecendo o Projeto	13
Componentes	21
Componentes Estilizados	21
Layout com Flexbox	32
Renderização Condicional	45
Estado do Componente	49
Botão	53
Navegação entre Telas	60
Drawer Navigator	65
Bottom Tab Navigator	71
WebView - Inserção de PDF	77
WebView - Inserção de Vídeo	79
Referências	81

Introdução

O desenvolvimento de aplicativos voltados para dispositivos móveis, em especial, tablets e smartphones, é realizado (basicamente) de forma nativa ou híbrida.

Desenvolvimento nativo requer do desenvolvedor conhecimento na linguagem de programação e plataforma adotada pelo fabricante. Nos dias atuais (2020), Google e Apple são as empresas que dominam o mercado de vendas, em específico, com seus sistemas operacionais Android e iOS. Existem vantagens para o desenvolvedor ao criar seus aplicativos nativamente na linguagem e plataforma adotadas pelo fabricante. Entretanto, também há desvantagens com a programação nativa. Podemos citar o custo de aprendizagem na linguagem e plataforma. Outro fator negativo é a impossibilidade do aplicativo criado não funcionar em dispositivos com sistemas operacionais diferentes.

Já o desenvolvimento híbrido, permite que o aplicativo criado seja compatível com sistemas operacionais e plataformas diferentes. O desenvolvimento do aplicativo fará o uso de outros recursos, como API (Interface de Programação de Aplicativos), bibliotecas, plugins entre outros recursos tecnológicos para poder se comunicar com o sistema operacional nativo. Existem diversas plataformas

e frameworks que atualmente são utilizados para o desenvolvimento híbrido.

Neste material didático, iremos abordar o desenvolvimento híbrido através do React Native. A plataforma é de propriedade do Facebook, usa tecnologia JavaScript e baseada no React.

O React é a biblioteca JavaScript bastante utilizada hoje para o desenvolvimento web. Segundo Escudelario e Pinho (2020), o React compete com os frameworks Angular e Vue.js, sobressaindo, em termos de vantagens, por: a) facilidade no processo de aprendizagem; b) manutenção pelo Facebook no qual há diversas tecnologias de código aberto disponibilizadas e , ainda, testes na própria rede social do grupo; c) baseado em componentes; d) fato de ter sido adotado por outras grandes empresas.

Este material didático foi construído seguindo uma ordem lógica de prática de desenvolvimento. Desta forma, a sequência organizada nos capítulos favorecem um processo evolutivo de aprendizagem.

As informações e orientações presentes neste material foram, em sua maioria, extraídas do site oficial do React Native, disponível em: <https://reactnative.dev/>.

Preparação para Desenvolvimento

Há diversas formas de se preparar um ambiente de desenvolvimento para React Native. Você poderá usar um emulador ou mesmo um dispositivo físico para testar o aplicativo (app) que está desenvolvendo. Poderá usar a IDE (Ambiente de Desenvolvimento Integrado) oficial da Google (Android Studio), a oficial da Apple (Xcode¹) ou mesmo usar uma outra de sua preferência. É claro que o sistema operacional do seu computador que irá utilizar para desenvolver o aplicativo poderá também possibilitar algumas configurações diferenciadas.

Neste capítulo, iremos orientá-lo a preparar o ambiente para uso de uma IDE “genérica”, ou seja, não utilizando o Android Studio ou o Xcode. A opção disso neste capítulo se deve por dois fatores: a) Android Studio requer uma arquitetura computacional robusta, para poder executar com satisfação seu processo de criação; b) Xcode requer o uso do sistema operacional MacOS. O uso de uma IDE “genérica” tende a facilitar e agilizar o seu aprendizado nesta tecnologia.

A seguir, iremos apresentar os passos que deve seguir para deixar seu ambiente de desenvolvimento preparado:

¹ Necessário possuir o sistema operacional MacOS, presente nos computadores da Apple.

1. **Instalação do Node.js LTS:** Node.js é um interpretador de JavaScript assíncrono de código aberto orientado a eventos. Acesse o site <https://nodejs.org/en/download/> e baixe o Node.js na versão LTS compatível com o sistema operacional de seu computador;
2. **Instalação do Expo:** Expo é uma ferramenta utilizada para o desenvolvimento de apps com React Native. Sua utilidade é o acesso as API's do dispositivo, sem a necessidade que você, desenvolvedor, se preocupe com isso. A documentação oficial do Expo você encontra neste link: <https://expo.io/>. Para instalar em seu computador, acesse o terminal ou *prompt* de comando do seu computador. Execute o seguinte comando:

```
npm install -g expo-cli
```

Talvez seja necessário executar o comando citado com permissão de administrador, através do **sudo** em sistemas unix;

3. **Criação de um projeto:** finalizando a instalação do Expo e, ainda pelo terminal (ou *prompt*), criaremos um projeto chamado “Projeto01” através do seguinte comando:

```
expo init Projeto01
```

4. **Acessando a pasta do projeto:** após o comando anterior para criar o projeto, acesse a pasta dele através do comando:

```
cd Projeto01
```

5. **Execute o projeto:** após acessar a pasta do projeto, execute-o através do comando:

```
npm start
```

ou

```
expo start
```

6. **Baixe um app no seu dispositivo:** ao executar o passo anterior, será aberta uma aba em seu navegador com seu projeto em execução. Antes de interagirmos com esta aba, baixe da loja de aplicativos um app chamado “expo client”. Este app faz o intermédio do seu projeto com os recursos nativos do seu dispositivo. A Imagem 1 ilustra o app;

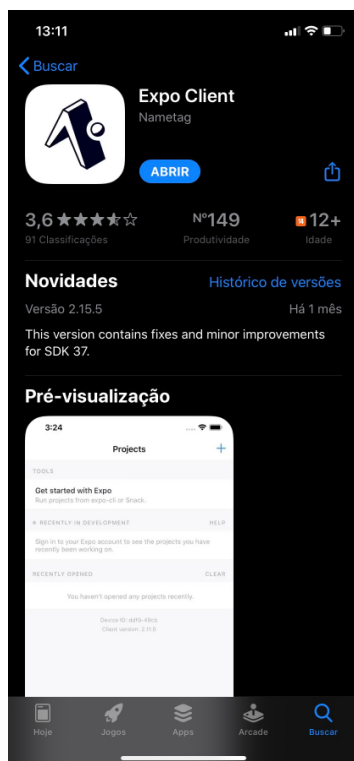


Imagem 1: app Expo Client

7. **Abra seu projeto no seu dispositivo:** após instalar o app (expo client), certifique-se que seu computador e seu dispositivos estão **conectados a mesma rede sem fio**. Abra a câmera do seu dispositivo para captura de código QR Code (dependendo do modelo e sistema operacional, basta apenas abrir a câmera, em outros casos, use o próprio Expo Client para fazer a captura). O seu projeto será aberto então em seu dispositivo. O Expo Client pedirá um breve cadastro para abrir suas funcionalidades;

8. **Use uma IDE para codificar:** você pode escolher alguma IDE para poder codificar e programar. Recomendamos, neste material, o Visual Studio Code, disponível para download em: <https://code.visualstudio.com/>. Assim que abrir a IDE, escolha a opção “abrir” e aponte para a pasta do projeto. O arquivo principal no qual você começará a codificar, chama-se “App.js”.

Conhecendo o Projeto

Após ter criado o projeto via terminal e, aberto ele pelo Visual Studio Code (ou similar), você verá a estrutura de pastas e arquivos demonstrados na Imagem 2.

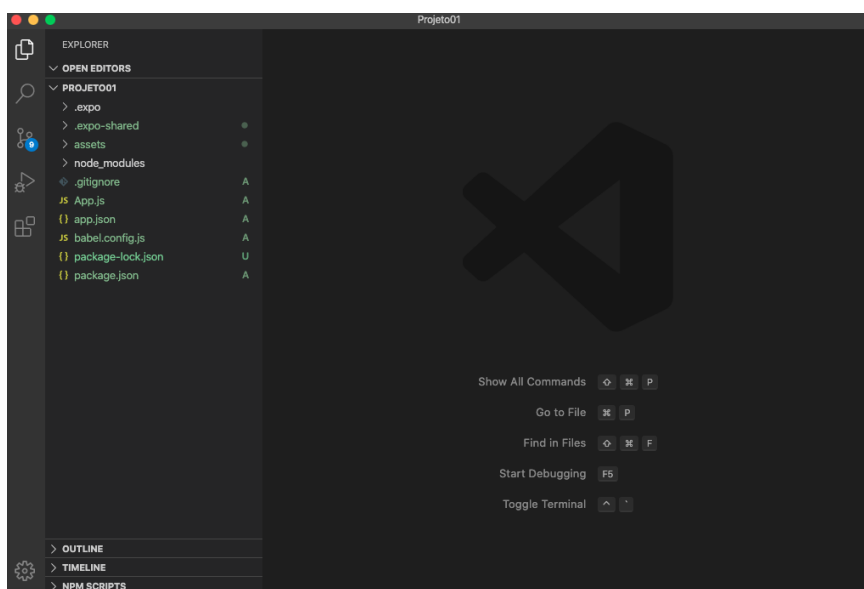


Imagem 2: estrutura de projeto

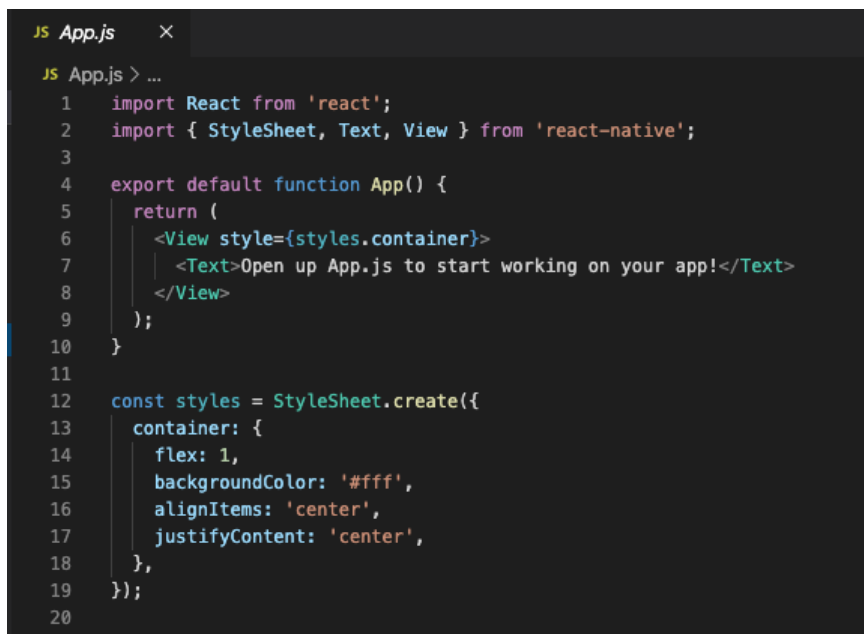
Esta estrutura está organizada da seguinte forma:

- *.expo* e *.expo-shared*: pastas ocultas no quais estão os arquivos de configuração e informação do Expo;
- *assets*: local onde estão as imagens do seu aplicativo. Novas imagens devem ser inseridas aqui;

- *node_modules*: local onde estão as dependências do projeto;
- *.gitignore*: local onde são informados os arquivos que serão ignorados ao subir o projeto para controle de versionamento no git;
- *App.js*: arquivo principal do projeto. Ele é ponto de entrada onde você iniciará a codificação;
- *app.json*: arquivo onde estão os metadados do aplicativo, como nome, versão dependências, ícone, plataforma etc;
- *babel.config.js*: arquivo de configuração Babel, responsável por traduzir o JSX² para sintaxe web (HTML, CSS e JS);
- *package-lock.json* e *package.json*: scripts para build na plataforma Android e iOS.

Após conhecer a estrutura básica de um projeto React Native, vamos conhecer agora o arquivo principal de “*start*” do aplicativo (*App.js*) . A Imagem 3 ilustra este arquivo aberto.

² Segundo Escudelar e Pinho (2020), a interface gráfica do React Native é desenvolvida em JSX, que por sua vez, é uma extensão da sintaxe do JavaScript que lembra HTML/XML. Nesta estrutura, é possível escrever HTML, CSS e JavaScript em um único arquivo.



```

JS App.js  X
JS App.js > ...
1  import React from 'react';
2  import { StyleSheet, Text, View } from 'react-native';
3
4  export default function App() {
5    return (
6      <View style={styles.container}>
7        <Text>Open up App.js to start working on your app!</Text>
8      </View>
9    );
10 }
11
12 const styles = StyleSheet.create({
13   container: {
14     flex: 1,
15     backgroundColor: '#fff',
16     alignItems: 'center',
17     justifyContent: 'center',
18   },
19 });
20

```

Imagem 3: arquivo app.js

As duas primeiras linhas referem-se ao processo de importação. A primeira importando do React, lembrando que é a tecnologia usada para desenvolvimento web. Já a segunda, faz a importação do React Native, voltado para plataforma móvel. Quando você adicionar objetos ao seu aplicativo, não esqueça de verificar se a importação foi feita corretamente.

O React (web ou móvel) atua no princípio do paradigma de componente. Durante a prática de desenvolvimento, você irá observar melhor o uso deste paradigma. Para criar seus componentes, você poderá fazer através de *classe* (Imagem 4) ou de *função* (Imagem 5).

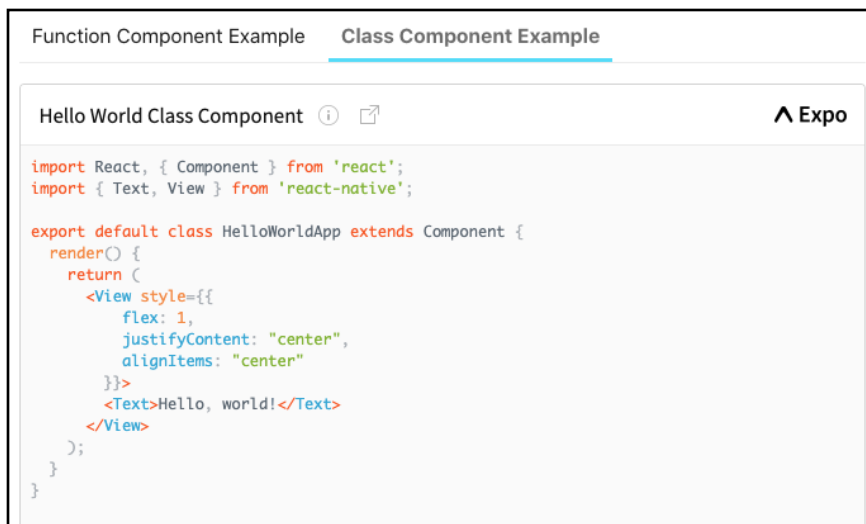


Imagem 4: codificando em forma de classe
Fonte: Site oficial React Native

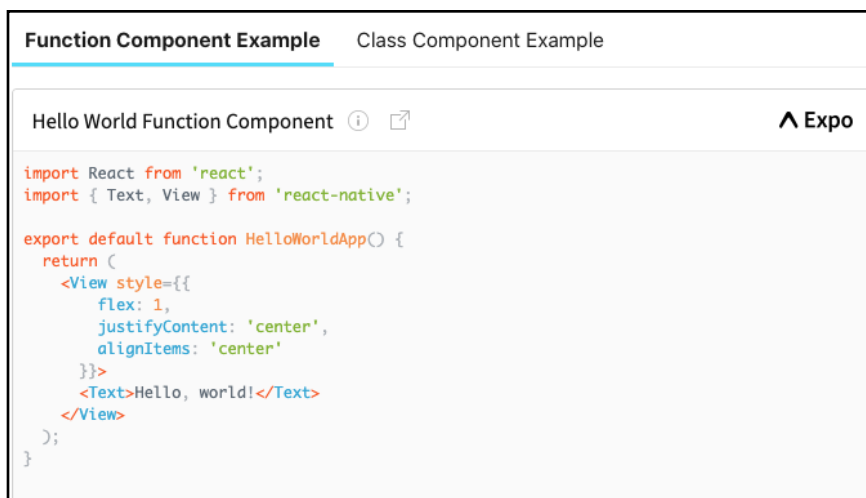


Imagem 5: codificando em forma de função
Fonte: Site oficial React Native

Na criação do seu primeiro projeto, observe que ele foi criado pelo paradigma de função. Neste código há dois escopos: um da função app (*export default function App*) e outra para definição de um estilo (*const styles*).

No escopo da função, observe que ela retorna uma estrutura de <View> e <Text>. O View faz analogia a tag <div> de um código HTM. Já o Text faz analogia a tag <p> de um código HTML. Observe que há uma frase dentro do escopo da tag Text. Quando você escaneou o QR Code está frase deve ter aparecido na tela principal do app em seu dispositivo. A frase equivale a uma mensagem inicial de boas vindas, assim como o famoso “hello world”.

Se você alterar o conteúdo desta frase e salvar verá que automaticamente ela será alterada no dispositivo. **Faça este teste!** Diferentemente da programação nativa onde você precisa executar um “run”, no React Native basta apenas salvar o código na IDE para ver o resultado. É simples e rápido. Caso tenha acontecido algum problema, refaça o processo de digitalização do QR Code pelo seu dispositivo.

Lembre-se das regras: 1) dispositivo e computador precisam estar na mesma rede de internet; 2) terminal ou *prompt* precisam estar aberto após a execução do comando “*expo start*”; 3) navegador precisa com a página do expo precisa também estar aberta.

Agora que você viu como alterar o texto inicial dado na criação do projeto, vamos forçar um erro para que possa ver também como detectar erros e corrigir. Toda

codificação baseada em tag, é necessário sempre abrir e fechar a tag. **Apague o fecha tag do Text** (`</Text>`). Isso é um erro grave, mas estamos fazendo isso apenas como um teste. Assim que apagar o fecha tag Text, salve o código. Observe que o app que esta aberto em ser dispositivo acusou uma tela de erro (Imagem 6) e, também, no navegador aparecerá a mesma informação de erro (Imagem 7).



Imagem 6: mensagem de erro no dispositivo

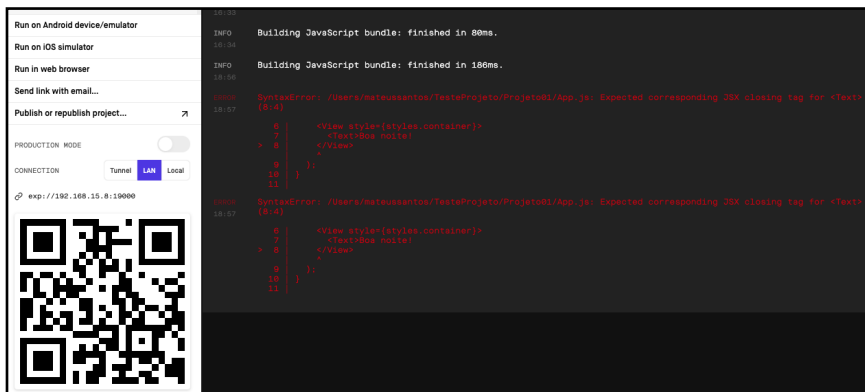


Imagem 7: mensagem de erro no navegador

Desta forma, observe que será possível você monitorar os possíveis erros que irão surgir tanto pelo navegador quanto pelo dispositivo. **Volte o fecha tag Text**, salve o código da IDE e observe novamente que seu app voltará com a mensagem inicial em poucos segundos.

Uma outra observação importante: a função requer o retorno de 1 objeto apenas. Desta forma, caso você tenha mais de um elemento para exibir, será necessário usar um container para carregá-los. Por exemplo, se você for colocar duas frases em duas tag's TEXT, desta forma:

```
export default function App() {  
  return (  
    <Text>Frase 1</Text>  
    <Text>Frase 2</Text>  
  );  
};
```

Verá que o trecho de código acima não será executado. Isso porque a função (*function*) deverá retornar apenas 1 objeto em seu retorno (*return*). Quando você criou o projeto, automaticamente a tag View veio por default. A View é um container, você pode colocar diversos outros objetos dentro dela e, por sua vez, ela pode ser retorno de uma função. Fique atento a isso!

Voltando ao segundo escopo, presente no código App.js, você verá que há uma definição de estilos, assim como está acostumado a fazer em CSS. A constante *style* define algumas propriedades. Observe que a tag View usa a propriedade *style*, usando assim os estilos definidos naquele segundo escopo. Acostume-se a ver uma mistura de padrões de códigos (tag, estilos, propriedades, funções etc), pois o React Native trabalha desta forma.

Componentes

No código inicial que você trabalhou ao criar um novo projeto, atuou com as tag's Text e View. Estas tag's são, na verdade, componentes nativos do React Native. Quando você desenvolve nativamente em Android, há componentes proprietários do Java ou Kotlin. Quando desenvolve para iOS, há componentes proprietários do Objective-C ou Swift. O JavaScript permite você programar nativamente em React Native sem se preocupar como ele se irá comportar no dispositivo do seu cliente, ou seja, independente se o aparelho dele usa sistema operacional Android ou iOS. A Imagem 8 ilustra isso.

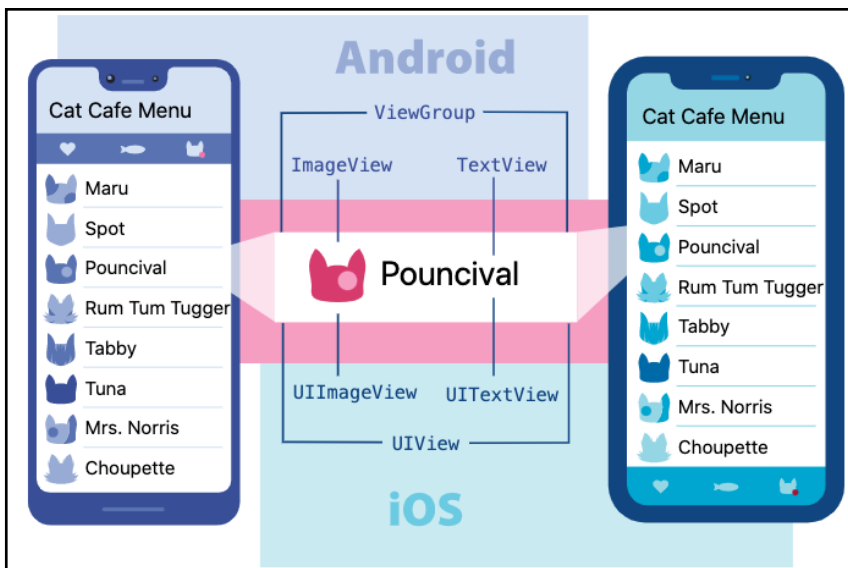


Imagem 8: transparência dos componentes independente da plataforma

Fonte: Documentação Oficial React Native

A Imagem 9 ilustra uma tabela comparativa entre alguns componentes e suas plataformas. A documentação oficial do React Native você poderá consultar neste link: <https://reactnative.dev/docs/components-and-apis>.

REACT NATIVE UI COMPONENT	ANDROID VIEW	IOS VIEW	WEB ANALOG	DESCRIPTION
<code><View></code>	<code><ViewGroup></code>	<code><UIView></code>	A non-scrolling <code><div></code>	A container that supports layout with flexbox, style, some touch handling, and accessibility controls
<code><Text></code>	<code><TextView></code>	<code><UITextView></code>	<code><p></code>	Displays, styles, and nests strings of text and even handles touch events
<code><Image></code>	<code><ImageView></code>	<code><UIImageView></code>	<code></code>	Displays different types of images
<code><ScrollView></code>	<code><ScrollView></code>	<code><UIScrollView></code>	<code><div></code>	A generic scrolling container that can contain multiple components and views
<code><TextInput></code>	<code><EditText></code>	<code><UITextField></code>	<code><input type="text"></code>	Allows the user to enter text

Imagem 9: comparativo dos principais componentes

Fonte: Documentação Oficial React Native

Conforme já dito aqui neste material, o React Native trabalha com o paradigma de componentes. VoC6e pode (e vai certamente) criar seus próprios componentes para desenvolver seus projetos.

Vamos demonstrar na sequência a criação de um componente, que posteriormente você irá usar no seu código principal (App.js).

Para facilitar a organização, crie uma pasta onde você coloca seus componentes criados. No Visual Studio, ao lado do nome do projeto há um ícone para criação de pastas. Imagem 10 ilustra isso.

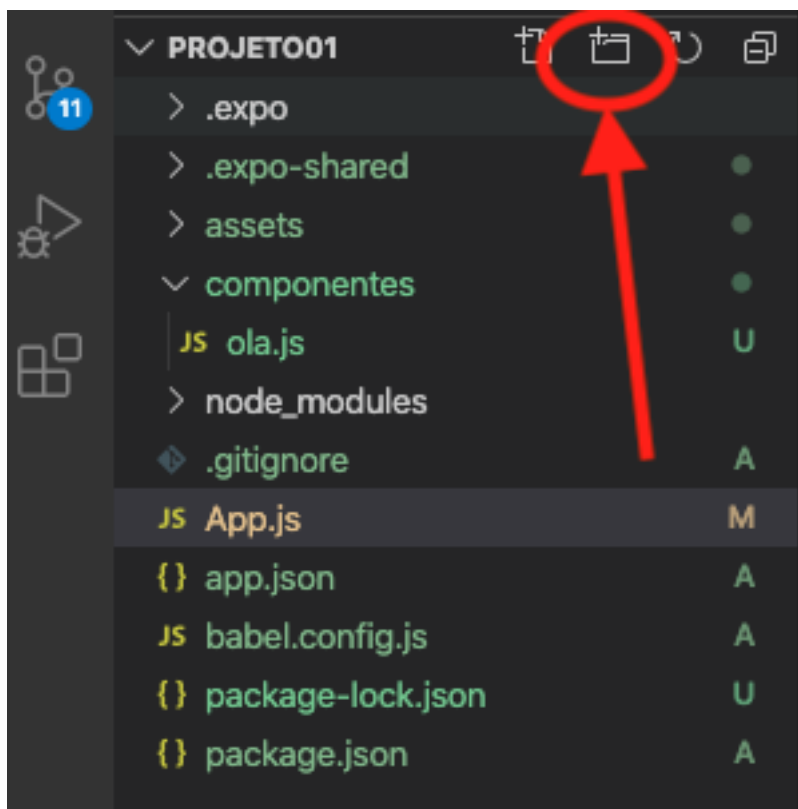
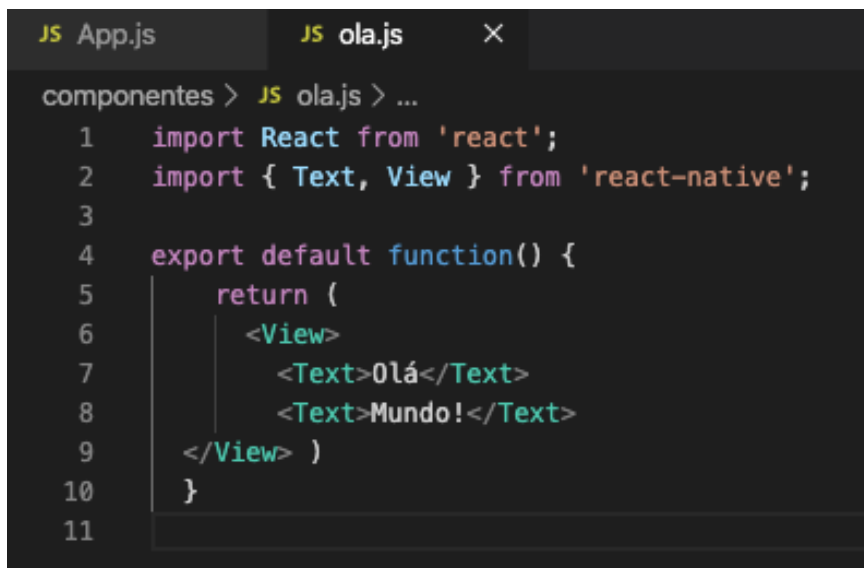


Imagem 10: criando nova pasta

Crie uma pasta chamada “componentes”, na sequência, crie um arquivo dentro desta pasta e salve com o nome “ola.js”. Lembrando que o nome da pasta e do componente você poderá escolher livremente, conforme escopo do seu projeto.

No arquivo novo que acabou de criar, codifique ele conforme a Imagem 11 mostra.

A screenshot of a code editor interface. At the top, there are two tabs: 'JS App.js' and 'JS ola.js'. The 'JS ola.js' tab is active, and a close button 'X' is visible to its right. Below the tabs, the text 'componentes > JS ola.js > ...' is displayed. The main area shows the following code:

```
1  import React from 'react';
2  import { Text, View } from 'react-native';
3
4  export default function() {
5    return (
6      <View>
7        <Text>Olá</Text>
8        <Text>Mundo!</Text>
9      </View> )
10 }
11
```

Imagem 11: criando novo componente

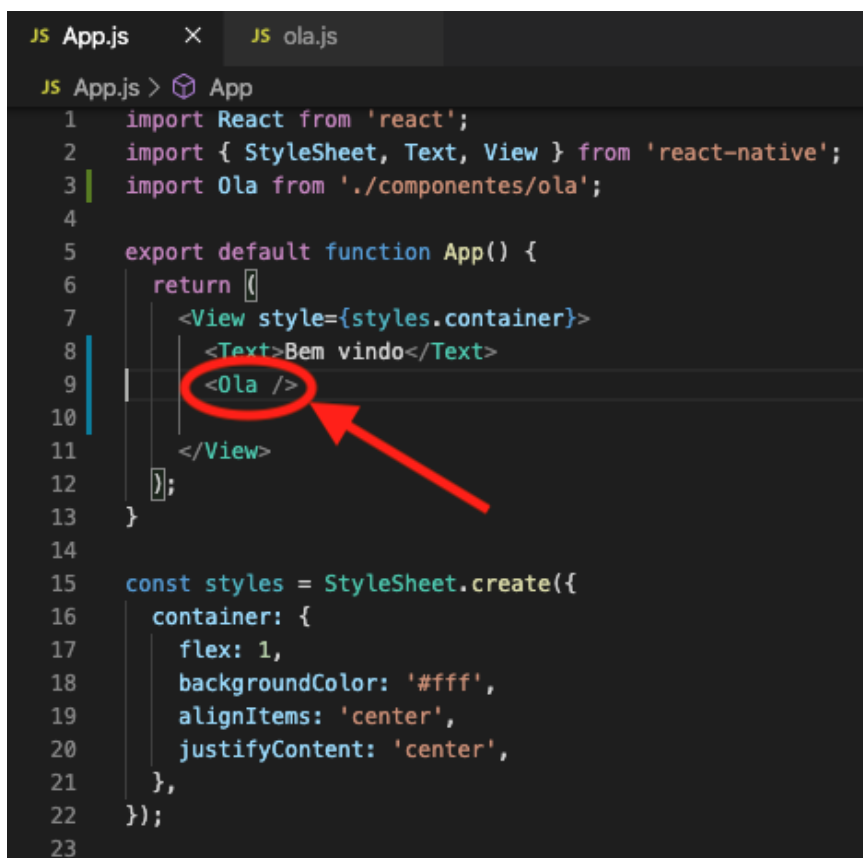
As duas primeiras linhas de *import*, nós já vimos o significado delas. Uma faz importação do React (web) e outro especificamente para dispositivos móveis (*native*).

Da linha 4 a linha 10 temos o escopo de uma função, note que ela tem o termo “*export*” significando que a função irá retornar algo, no caso, um componente. Como irá retornar, observe o uso da palavra “*return*”, lembrando que este retorno deverá ser um único objeto. Em razão disso, foi usado o View como um container para “carregar” dois View’s. Após codificar este arquivo, não esqueça de salvar.

Após criar seu componente, volte para o arquivo principal (App.js). Primeiramente, será necessário importar este componente para o arquivo principal

```
import Ola from './componentes/ola';
```

Feito a importação, basta usá-lo como se fosse uma tag nativa. No caso, ficara como `<Ola />`. Imagem 12 ilustra isso.



```
JS App.js  X  JS ola.js
JS App.js > App
1  import React from 'react';
2  import { StyleSheet, Text, View } from 'react-native';
3  import Ola from './componentes/ola';
4
5  export default function App() {
6    return (
7      <View style={styles.container}>
8        <Text>Bem vindo</Text>
9        <Ola />
10     </View>
11   );
12 }
13
14
15 const styles = StyleSheet.create({
16   container: {
17     flex: 1,
18     backgroundColor: '#fff',
19     alignItems: 'center',
20     justifyContent: 'center',
21   },
22 });
23
```

Imagem 12: usando novo componente

Agora que criamos nos primeiro componente, vamos melhorá-lo possibilitando o uso das propriedades dele. Assim como você já deve estar acostumado com o uso de propriedades em HTML, aqui iremos usar o mesmo princípio.

Após criar nosso componente, você percebeu que o uso dele dentro do arquivo principal foi feito através da sintaxe de uma tag. Propriedades de uma tag você insere antes do sinal de fechar tag (“/” sinal de maior). Neste caso, a propriedade que você irá inserir na tag será, na verdade, uma passagem da parâmetro para o componente criado.

Vamos fazer na prática isso, insira uma propriedade “nome” na tag `<Ola>` (seu componente criado). Você poderá criar suas próprias propriedades, sem se preocupar com os nomes delas. Poderá ser “*name*” ou “nome”, ou ainda qualquer outro nome que escolher. Não há regra para isso!

```
<Ola nome = "Maria" />
```

Neste caso, estamos passando a String “Maria” em forma de parâmetro para nosso componente. Ou, mais tecnicamente, estamos passando a String como uma propriedade do componente Ola.

Agora volte ao ser arquivo Ola.js. Vamos tratar a propriedade que foi criada de modo que seu componente consiga fazer algo com ela.

O primeiro passo é allear os parâmetros de entrada da sua função. Anteriormente, quando você criou, não havia especificação de parâmetros de entrada de função (parênteses vazios). Acione o nome “*props*”. Na verdade, poderia ser qualquer nome, mas por convenção, usa-se “*props*”. Diferentemente de outras linguagens que você esteja acostumado, não é necessário especificar o tipo do parâmetro, é livre e flexível!

```
export default function(props) {
```

Dentro da função, para ter acesso ao conteúdo definido nas propriedades, você usará “*props.nome*”. Se tivesse definido a propriedade como “name” seria “*props.name*” e assim por diante. Será sempre o parâmetro de entrada seguido de ponto e o nome da propriedade que você usou o componente. Para que este conteúdo seja exibido, é necessário que você coloque o parâmetro dentro de chaves. A seguir, há um exemplo:

```
<Text>Propriedade recebida é {props.nome}</Text>
```

Componentes Estilizados

Neste capítulo iremos fazer o uso de estilos CSS para personalizar melhor nosso aplicativo. Até então, nos capítulos anteriores, havíamos usado os componentes (nativos ou criados) sem nos preocupar em criar algum tipo de estilo.

Crie uma pasta em seu projeto chamada Estilos (na prática, nomes são de sua livre escolha). Crie dentro desta pasta um arquivo chamado “Estilos.js”.

Neste arquivo de estilos criado, a primeira coisa que precisamos fazer é a importação do módulo StyleSheet do pacote React Native.

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

Na sequência, crie a exportação do seu estilo através do código:

```
export default StyleSheet.create
```

O código completo (até então) ficará conforme a Imagem 13.

```

Estilos > JS Estilos.js > [🔗] default
1  import { StyleSheet } from 'react-native';
2
3  export default StyleSheet.create(
4    {
5
6    }
7  )

```

Imagem 13: criando estilos

Vamos estilizar a propriedade *text*, uma vez que estamos neste momento trabalhando apenas com propriedades simples de exibição textual. As propriedades do estilo *text* devem ficar dentro de chaves, separadas por vírgula. Vamos alterar as propriedades de tamanho do texto, estilo (negrito), borda e cor de borda, além do *padding* (preenchimento). A Imagem 14 mostra a codificação sugerida.

```

1  import { StyleSheet } from 'react-native';
2
3  export default StyleSheet.create({
4    {
5      text: {
6        fontSize: 18,
7        fontWeight: 'bold',
8        borderColor: 'blue',
9        borderWidth: 2,
10       padding:10,
11      }
12    }
13  })

```

Imagem 14: estilo text personalizado

Após criar o estilo, vamos usá-lo em nosso projeto. Dando sequência ao projeto criado nos capítulos anteriores, abra o arquivo `Ola.js`.

Primeiramente devemos fazer a importação do arquivos de estilos criado. O processo é parecido de quando você fez a importação do componente. Usou-se dois pontos sequencias porque lembre-se que o arquivo `Ola.js` está dentro da pasta `componentes` e não no raiz do projeto.

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

Posteriormente, adicione o estilo na tag (componente) desejado. No caso, vamos adicionar no terceiro `Text`. Você deve fazer a indicação do estilo dentro de chaves e chamar pelo nome do arquivo seguido de ponto e nome da propriedade estilizada.

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

Da forma como fizemos, a qualquer momento dentro do nosso projeto poderemos reaproveitar os estilos personalizados neste arquivo a parte. Esta é apenas uma forma de se trabalhar com estilos, uma forma no qual definimos a estilização genérica de nossa aplicação.

Uma outra forma é fazer a estilização específica para algum determinado componente. Observe que o arquivo principal (`App.js`) já faz isso. Note que neste arquivo há a importação do *StyleSheet* e há a estilização através da *const*

style. Neste caso, há a estilização do container, mas você pode, por exemplo, adicionar a propriedade *text*, estiliza-la e depois usá-la em algum componente (tag). A Imagem 15 ilustra esta modificação.

```
js App.js > [e] styles > text
1  import React from 'react';
2  import { StyleSheet, Text, View } from 'react-native';
3  import Ola from './componentes/ola';
4
5  export default function App() {
6    return (
7      <View style={styles.container}>
8        <Text style={styles.text}>Bem vindo !!</Text>
9        <Ola nome = "xico" />
10     </View>
11   );
12 }
13
14
15 const styles = StyleSheet.create({
16   container: {
17     flex: 1,
18     backgroundColor: '#fff',
19     alignItems: 'center',
20     justifyContent: 'center',
21   },
22   text: {
23     fontSize: 18,
24     fontWeight: 'bold',
25     borderColor: 'blue',
26     borderWidth: 2,
27     padding: 10,
28   },
29 },
30 });
31
```

Imagem 15: alterando estilo na classe principal

Layout com Flexbox

O componente pode especificar a disposição de seus filhos dentro de um layout responsivo. O Flexbox É uma alternativa para viabilizar isso. Usa os mesmos princípios do CSS, com algumas particularidades.

As propriedades de largura (*width*) e altura (*height*) serão importantes também na hora de definir a ocupação do objeto no layout. Podem ser usadas dentro da propriedade *style*. Importante destacar que as medidas são *unitless*, ou seja, sem densidade específica. Na prática significa que os objetos serão sempre do mesmo tamanho, independente do tamanho físico da tela do dispositivo.

A Imagem 16 ilustra o uso das propriedades de largura e altura. Note que foram criadas três *Views* variando as propriedades de largura, altura e cor. As alterações foram feitas *inline*, ou seja, dentro de cada tag.

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

export default function () {
  return (
    <View>
      <View style={{width: 50, height: 50, backgroundColor: 'powderblue'}}>
      <View style={{width: 100, height: 100, backgroundColor: 'skyblue'}}>
      <View style={{width: 150, height: 150, backgroundColor: 'steelblue'}}>
    </View>
  );
};
```

Imagem 16: alterando propriedades largura e altura

A Imagem 17 ilustra o resultado da codificação da Imagem 16.

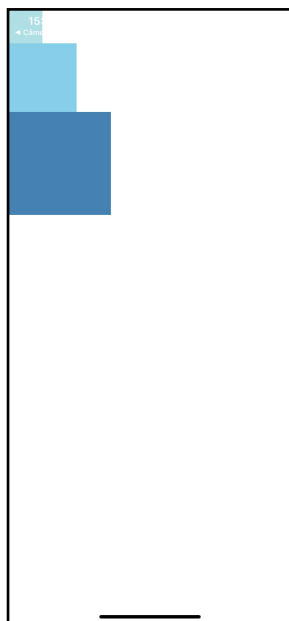


Imagem 17: resultado da alteração da imagem 16

Você pode usar a propriedade *flex* no componente para que ele expanda e defina dinamicamente sua ocupação na tela. Normalmente, você usará o valor 1. Se você tiver dois ou mais componentes dentro de uma View e definir o valor *flex* para eles em 1, cada um deles ocupará a mesma proporção na tela. Dinamicamente possuem “peso” igual e a ocupação será dividida.

Alterando o código da Imagem 16, vamos adicionar a propriedade *flex* nas quatro View's: uma na View principal e nas outras três View's internas. Nestas internas, vamos retirar as propriedades de largura e altura. A Imagem 18

ilustrar código alterado e a Imagem 19 ilustra o resultado gerado.

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

export default function () {
  return (
    <View style={{flex:1}}>
      <View style={{flex:1, backgroundColor: 'powderblue'}} />
      <View style={{flex:1, backgroundColor: 'skyblue'}} />
      <View style={{flex:1, backgroundColor: 'steelblue'}} />
    </View>
  );
};
```

Imagem 18: trabalhando com flex

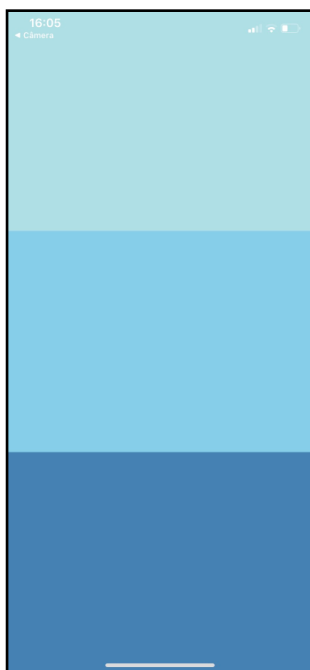


Imagem 19: resultado gerado com o código da imagem 18

Se você atribuir valores diferentes para o *flex*, obviamente, terá resultados diferentes. No exemplo anterior codificado, iremos alterar duas View's internas atribuindo valores de *flex* 2 e 3. Assim, teremos três View's internas com valores de *flex* 1, 2 e 3. A Imagem 20 ilustra esta alteração.

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

export default function () {
  return (
    <View style={{flex:1}}>
      <View style={{flex:1, backgroundColor: 'powderblue'}} />
      <View style={{flex:2, backgroundColor: 'skyblue'}} />
      <View style={{flex:3, backgroundColor: 'steelblue'}} />
    </View>
  );
};
```

Imagem 20: valores diferentes para o flex

Se somarmos 1, 2 e 3 teremos como resultado 6. A primeira View irá ocupar 1/6 da tela, a segunda View 2/6 da tela e a terceira View irá ocupar 3/6 da tela. Ou seja, independente do tamanho físico da tela do dispositivo, a proporção será a mesma na ocupação destes componentes na tela, tudo feito de forma dinâmica. A Imagem 21 ilustra o resultado desta alteração.

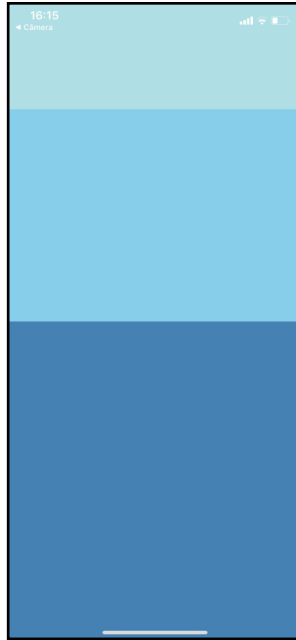


Imagem 20: resultado gerado com o código da imagem 19

FlexDirection é uma propriedade que você pode aplicar ao flex. Nela, você define a ordem em que os nós filhos serão montados no layout. No exemplo trabalhado até então, nós temos uma View principal (mãe) e três View's internas (filhas). Se aplicarmos o flexDirection na View principal poderemos alterar a disposição das três View's filhas. O flexDirection possui quatro valores que podem ser atribuídos:

- *Row*: nós filhos alinhados da esquerda para direita;
- *Column*: (padrão) nós filhos alinhados de cima para baixo;
- *Row-reverse*: nós filhos alinhados da direita para esquerda;
- *Column-reverse*: nós filhos alinhados de baixo para cima.

A imagem 21 ilustra o uso da propriedade `flexDirection`, sendo aplicada à View principal e com valor `row`. A Imagem 22 ilustra o resultado. A Imagem 23, mostra os exemplos do site oficial do React Native. A diferença do exemplo do site oficial é que eles aplicaram os valores de *width* e *height* nos nós filhos.

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

export default function () {
  return (
    <View style={{flex:1, flexDirection:"row"}}>
      <View style={{flex:1, backgroundColor: 'powderblue'}} />
      <View style={{flex:2, backgroundColor: 'skyblue'}} />
      <View style={{flex:3, backgroundColor: 'steelblue'}} />
    </View>
  );
};
```

Imagem 21: uso do `flexDirection`

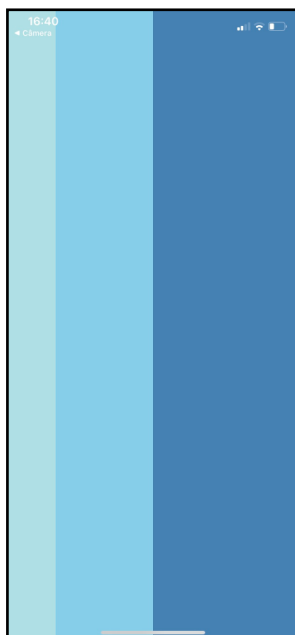


Imagem 22: resultado da alteração feito no código imagem 21

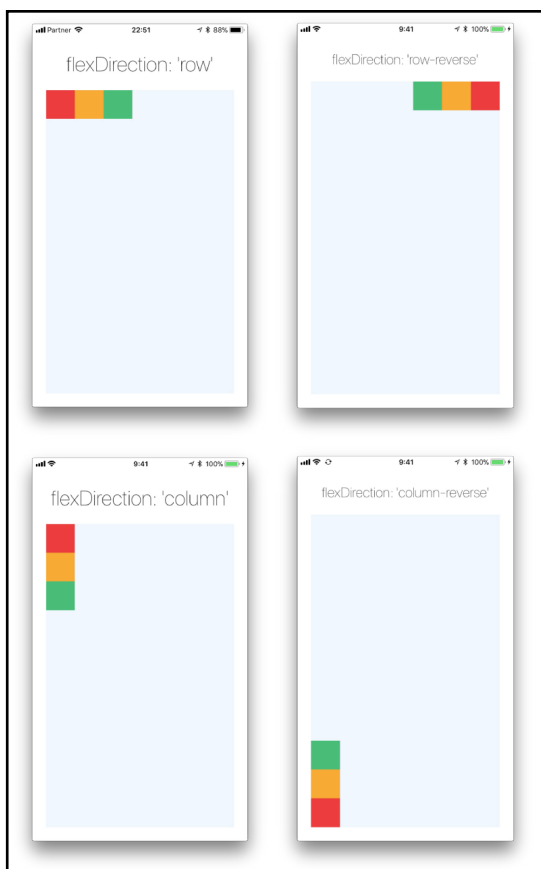


Imagem 23: diferenciação de uso dos valores do `flexDirection`

Fonte: <https://reactnative.dev/docs/flexbox>

JustifyContent é uma propriedade que você pode aplicar para alinhar os nós filhos dentro do container principal, isso usando ainda o princípio do *flex*. O alinhamento é aplicado ao eixo principal. Os valores do *justifyContent* são:

- *Flex-start*: (padrão) alinhamento no início do eixo principal;

- *Flex-end*: alinhamento no final do eixo principal;
- *Center*: centralizado no eixo principal;
- *Space-between*: espaço uniforme entre os nós filhos;
- *Space-around*: espaço uniforme entre os filhos, mas considera o espaço anterior ao primeiro nó e posterior ao último nó.



Imagem 24: diferenciação de uso dos valores do `justifyContent`
Fonte: <https://reactnative.dev/docs/flexbox>

A imagem 24 ilustra o uso da propriedade *justifyContent*. Esta imagem é do site oficial do React Native. As View's de cores vermelho, laranja e verde tiveram seus valores de altura e largura definidos em 50.

AlignItems é uma propriedade que você pode aplicar para alinhar os nós filhos dentro do container principal, isso usando ainda o princípio do *flex*. É muito parecido com o *justifyContent*, diferenciando que sua aplicação é feita no eixo transversal. Os valores do *alignItems* são:

- *Stretch*: (padrão) estica os nós filhos até o final;
- *Flex-start*: alinha no início do eixo transversal;
- *Flex-end*: alinha no fim do eixo transversal;
- *Center*: alinha no centro do eixo transversal;
- *Baseline*: alinha ao longo do eixo.

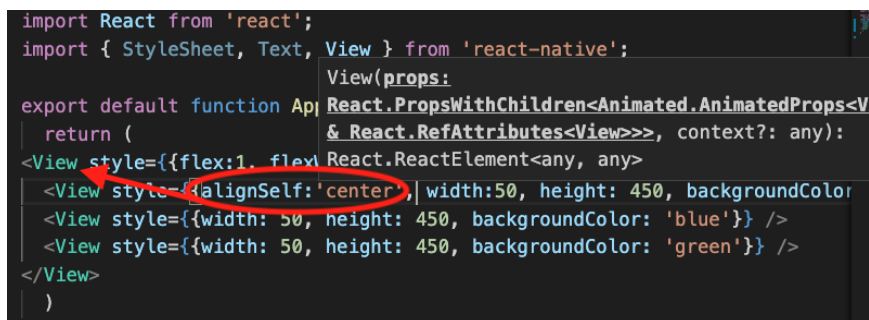


Imagem 25: diferenciação de uso dos valores do `alignItems`

Fonte: <https://reactnative.dev/docs/flexbox>

A Imagem 25 ilustra o uso dos valores diversos da propriedade *alignItems*.

AlignSelf é a possibilidade de você configurar o alinhamento de um nó filho. Observe que o *alignItems* atua no nó container, ou seja, naquele que possui componentes (objetos) internos. No entanto, caso queira desprender um nó filho da configuração de alinhamento do nó pai, você usará a propriedade *alignSelf*. A Imagem 26 ilustra o uso do *alignSelf*, observe que foi aplicado ao nó filho, a seta indica isso.



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

export default function App() {
  return (
    <View style={{flex:1, flex-direction: 'column', align-items: 'center', justify-content: 'center'}} >
      <View style={{alignSelf: 'center', width:50, height: 450, backgroundColor: 'blue'}} />
      <View style={{width: 50, height: 450, backgroundColor: 'blue'}} />
      <View style={{width: 50, height: 450, backgroundColor: 'green'}} />
    </View>
  )
}
```

Imagem 26: uso do *alignSelf*

FlexWrap é uma propriedade que permite que um objeto faça “quebra de linha” sem prejudicar os elementos que estão dentro dele. Se um componente (objeto) ultrapassar os limites do contêiner onde está, a visualização dele ficará desconfigurada. Com o uso da propriedade *flexWrap*, o componente será reorganizado em forma de “quebra de linha”. Você poderá usar os seguintes valores:

- *NoWrap*: sem quebra;
- *Wrap*: com quebra.

O código ilustrado na Imagem 27 ilustra o momento em que um componente (objeto) não respeita os limites de seu container. Observe que três View's internas estão com altura definida em 450. A Imagem 28 ilustra o resultado deste código. Observe que houve uma desconfiguração na organização dos elementos na tela do dispositivo.

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

export default function () {
  return (
    <View style={{flex:1, flexDirection:"column",
    justifyContent: 'center', alignItems:'stretch'}}>
      <View style={{width:50, height:450, backgroundColor: 'powderblue'}}>
      <View style={{width:50, height:450, backgroundColor: 'skyblue'}} />
      <View style={{width:50, height:450, backgroundColor: 'steelblue'}} />
    </View>
  );
};
```

Imagem 27: estouro dos limites do container

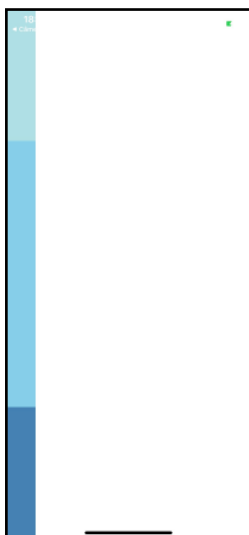


Imagem 28: resultado do código da Imagem 26

Na sequência, neste mesmo código, fizemos a inserção da propriedade `flexWrap` (Imagem 29). Observe a reorganização das `View`'s (Imagem 30).

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

export default function () {
  return (
    <View style={{flex:1, flexWrap:'wrap', flexDirection:'column',
    justifyContent: 'center', alignItems:'stretch'}}>
      <View style={{width:50, height:450, backgroundColor: 'powderblue'}} />
      <View style={{width:50, height:450, backgroundColor: 'skyblue'}} />
      <View style={{width:50, height:450, backgroundColor: 'steelblue'}} />
    </View>
  );
};
```

Imagem 29: uso do `flexWrap`

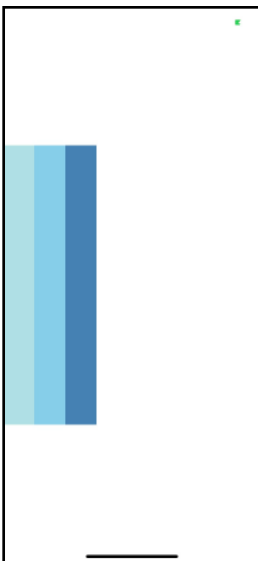


Imagem 30: resultado do código da Imagem 29

FlexGrow é uma propriedade que define o aumento ou melhor, a expansão de um nó filho dentro do espaço restante do container onde está inserido. O valor default dele é zero (0). Se, por exemplo, eu atribuir um valor 1 para um nó filho, ele irá ocupar todo o espaço restante dentro do container que está inserido.

Flex-Shrink é uma propriedade que define a redução de um nó filho em razão do espaço que tem dentro do container. O valor default dele é um (1), significando que ele pode ter uma redução caso necessário. No entanto, caso seja atribuído um valor zero (0) estaríamos informando que este nó é inflexível, ou seja, não poderá ter redução alguma e precisará manter o seu tamanho original.

Flex-Basis é uma propriedade que define o tamanho inicial do nó filho antes da distribuição do espaço restante do container onde está inserido.

Renderização Condicional

É possível renderizar os componentes obedecendo algum tipo de condição. Neste caso, uma possibilidade para estabelecer a condição é usar a estrutura *if-else*.

Uma maneira comum para se fazer isso é utilizando as propriedades de um componente, como já vimos, favorece o envio de dados através de parâmetros. Desta forma, você poderá ter uma função para receber os parâmetros (em forma de propriedade) e posteriormente tratá-la. Para este caso, a tratativa da condição é feita em um arquivo externo (extra).

Contudo, você pode também fazer a tratativa no próprio arquivo, fazendo uma chamada simples da função e passando os parâmetros.

O primeiro exemplo que vamos mostrar é a chamada da função dentro do próprio arquivo principal (App.js). A Imagem 31 ilustra um trecho de código onde tem uma função chamada “teste”. Esta função espera um parâmetro de entrada. Dentro dela, há uma condição sobre este parâmetro. Observe que dentro do *if* e do *else* há um retorno de um componente Text. Este componente terá uma renderização diferenciada em razão do parâmetro de entrada. Para um caso, terá tamanho e cor de fonte de um

jeito, para outro caso o componente terá outras configurações.

```
function teste(props){  
  if(props <=3)  
  {  
    return(  
      <Text style={{fontSize:40, color:'blue'}} >É menor ou igual a 3</Text>  
    )  
  }else{  
    return(  
      <Text style={{fontSize:20, color:'white'}} >Maior que 3</Text>  
    )  
  }  
}
```

Imagem 31: renderização condicional: função dentro do mesmo arquivo

Para chamar a função criada, você poderá fazê-la dentro do contexto de outra tag ou sozinha em uma outra linha. A chamada da função deverá ser feita dentro de chaves. A Imagem 32 ilustra isso, na primeira linha, a função é chamada dentro do contexto de uma outra tag. Na segunda, sozinha em uma outra linha. Como a função retorna um componente Text, ele será montado conforme o local onde você fez a chamada da função.

```
<Text>Chamando dentro da tag: {teste(4)}</Text>  
{teste(3)}
```

Imagem 32: renderização condicional: chamada da função

O próximo exemplo que iremos mostrar é fazendo a renderização usando um arquivo externo. Neste formato, você pode fazer a condição na função principal (que possui o *export*) ou em funções secundárias. Para demonstrar,

criamos um arquivo externo chamado “cond.js”. Na função principal, colocamos a mesma função do exemplo anterior. Contudo, neste caso, a função tem a informação *expo default*. Observe também que o parâmetro comparado segue uma estrutura *props.nome*. *Props* já explicamos que refere-se ao parâmetro de entrada. Pode ser outro nome, mas por convenção usa-se o *props*. Já *nome*, refere-se a propriedade da tag (componente) que será chamada. A imagem 33 ilustra a codificação da função explicada.

```
export default function(props){  
  if(props.nome <=3)  
  {  
    return(  
      <Text style={{fontSize:40, color:'blue'}} >É menor ou igual a 3</Text>  
    )  
  }else{  
    return(  
      <Text style={{fontSize:20, color:'white'}} >Maior que 3</Text>  
    )  
  }  
}
```

Imagem 33: renderização condicional: chamada da função em um arquivo externo “cond.js”

A Imagem 34 ilustra a chamada da função no arquivo principal (App.js). Lembre-se que para isso, foi necessário importar no arquivo principal o arquivo “cond.js”. Relembrando que uma vez importado, *cond* torna-se um componente dentro do escopo do meu projeto. Como componente, podemos chama-lo através de uma tag. Observe na imagem que o valor para ser trabalhado na função foi chamado através de uma propriedade da tag *cond*. A propriedade chamada “nome”. Destacamos

também que poderia ser qualquer outro nome para esta propriedade, não é um atributo reservado.

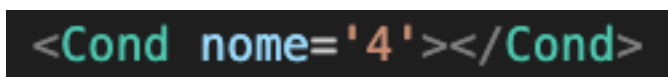
A imagem mostra um trecho de código JSX: `<Cond nome='4'></Cond>`. O texto "Cond" e o símbolo de fechamento "</Cond>" são em verde, enquanto o valor da propriedade "nome", que é "4", é em laranja.

Imagem 34: renderização condicional: chamada da função em um arquivo externo “cond.js”, usando a propriedade nome

Uma observação importante a ser feita refere-se ao tratamento da propriedade dentro da função. Observe que na Imagem 33, o parâmetro foi tratado como “*props.nome*”. Se você retirar o “nome”, provavelmente o código se manterá funcionando. Pelo fato de você não necessitar tipificar o parâmetro, ele acabará reconhecendo. Contudo, a forma correta (e que evitará erros) é você seguir a codificação feita no exemplo, utilizando exatamente a propriedade definida.

Uma outra possibilidade é você criar uma função dentro do arquivo externo. Desta forma, você encaminharia o valor recebido pelo *props* como parâmetro desta função. No exemplo acima, seria “*props.nome*”. A Imagem 35 ilustra esta possibilidade.

A imagem mostra um código JavaScript com uma função aninhada. O código é o seguinte:

```
export default function(props){  
  return (funcaoInterna(props.nome))  
}  
  
function funcaoInterna(dado){  
  if(dado <=3)  
  {  
    return(  
      <Text style={{fontSize:40, color:'blue'}} >É menor ou igual a 3</Text>  
    )  
  }else{  
    return(  
      <Text style={{fontSize:20, color:'white'}} >Maior que 3</Text>  
    )  
  }  
}
```

Imagem 35: função dentro de outra função

Estado do Componente

Os componentes possuem uma característica chamada estado. Nos capítulos anteriores nós vimos o uso das propriedades de um componente. Neste capítulo, trataremos sobre os estados dos componentes.

Nas propriedades do componente, os dados ou informações são tramitadas de forma externa. Nós vimos isso nos capítulos anteriores onde passávamos dados através de propriedades nas tag (componentes) que eram tratadas posteriormente como parâmetros de entrada de função.

Em estados, os dados são privados e controlados pelo próprio componente. Sua nomenclatura na codificação é *state*. Os dados de estado podem ser string, número, array e mesmo objeto.

Primeiramente vamos criar um novo componente. Desta vez, vamos criá-lo na forma de classe, em vez de função para praticarmos também este formato. Nosso novo componente chamara “Evento.js”. Dentro da classe Evento, criamos os estados através da palavra reservada *state*. Dentro do escopo de *state*, definimos um estado chamado “base” com valor inicial vazio (string sem valor). Dentro do escopo de *state*, poderíamos definir outros estados, separando assim os nomes por vírgula. Criamos um segundo estado chamado “base1”. Na Imagem 36

ilustramos a criação deste novo componente com a codificação detalhada até aqui.

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

class Evento extends React.Component{
  state = {
    base: '',
    base1: ''
  }
  render(){
  }
}
export default Evento;
```

Imagem 36: criando estado em um componente

Na sequência, vamos codificar o escopo de render, no qual tem o objetivo de renderizar o componente na tela do dispositivo. Neste render, faremos o retorno de uma View contendo outros componentes dentro dela, no caso, componente Text. Este Text faremos a exibição do estado do componente Evento. Para isso, usaremos o termo *this*, seguido de *state* e o nome do estado que queremos exibir. A Imagem 37 ilustra isso.

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

Imagem 37: exibindo valor de um estado em um componente Text

Para você ver o resultado codificado até aqui, terá que importar o componente `Evento` no arquivo principal (`App.js`) e depois usar a tag `Evento`, desta forma:

```
<Evento></Evento>
```

Se você testar a codificação até aqui não verá efeito algum, isso porque a string do estado *base* está vazia. Experimente alterar o valor da string *base* para ver o resultado.

Nossa próxima alteração, sobre a perspectiva do estado é fazer com que a ação do usuário altere um valor de estado. Para isso, vamos usar o componente *TextInput*, refere-se a uma caixa de texto para o usuário inserir algum valor.

No componente *TextInput*, vamos alterar duas propriedades ao codificar a tag: 1) *value*, para definir que o valor digitado pelo usuário será definido na própria caixa de texto; 2) *onChangeText*, nosso primeiro tratamento a eventos em um componente. Ação que será executada em tempo real quando o usuário fazer a digitação.

Deixe a string do estado “base” vazia novamente e faça o teste. A Imagem 38 ilustra a codificação explicada.

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

Imagem 38: uso do componente `TextInput` e com alteração de estado

Observações importantes: a propriedade `onChangeText` é escutado de evento, ou seja, ela monitora toda e qualquer alteração de texto que ocorra no componente no qual ela está configurada. Na codificação que fizemos, `onChangeText` irá alterar o estado do componente através do `setStatus`. Quando um componente tem seu estado alterado, ele chama novamente o render, para renderizar o componente na tela.

Botão

Neste capítulo iremos demonstrar o uso de botões nas telas do seu aplicativo em React Native. Especificamente, será mostrada a criação, a inserção de um evento simples de toque e a alteração de estado (*state*) no próprio componente e em outros.

Para fazer o uso do botão, bastará usar a tag `<Button>`, porém, antes disso é necessário que você adicione a biblioteca *Button* no *import* do React Native.

Button possui quatro propriedades principais, no qual certamente você irá utilizar:

- *Title*: título (rótulo) no botão;
- *Color*: cor do botão;
- *AccessibilityLabel*: texto (rótulo) complementar para acessibilidade;
- *OnPress*: define ação básica de toque no botão.

O primeiro exemplo que iremos mostrar é a ação do botão para apresentar um alerta em forma de caixa de diálogo. Este alerta é criado (definido) pela classe *Alert*. Será necessário também que você adicione a importação do *Alert* em React Native. A Imagem 39 ilustra este exemplo, observe bem a importação das bibliotecas *Button* e *Alert* e a codificação das quatro propriedades principais do botão.

```
import { StatusBar } from 'expo-status-bar';
import React from 'react';
import { StyleSheet, Text, View, Button, Alert } from 'react-native';
import Evento from './src/evento';

export default function App() {
  return (
    <View style={styles.container}>
      <Text name="tx" >Open up App.js to start working on your app!</Text>
      <Button
        title= "Botão Inicial"
        color= "blue"
        accessibilityLabel= "primeiro botão do app"
        onPress= (() => Alert.alert('Simple Button pressed'))
      > /Button
    </View>
  );
}
```

Imagem 39: uso básico do Button

No exemplo anterior, ao pressionar o botão um alerta é chamado para a tela, em forma de caixa de diálogo. Observe que a chamada do alerta foi feita *inline*, juntamente na mesma linha de código da propriedade *onPress*. No entanto, dependendo do seu projeto, você precisará ter uma codificação mais bem organizada ou, muitas vezes, precisará deixar a ação do botão de forma mais fácil de ser atualizada. Quando for este caso, você poderá deixar a ação botão definida em uma função a parte, cabendo assim somente a chamada da função dentro da propriedade *onPress*.

No exemplo a seguir, criamos uma função a parte chamada “funcaoBotaoPrincipal” no qual colocamos nela a criação do mesmo alerta do exemplo anterior. Neste exemplo, estamos apenas ilustrando como você faz a chamada de uma função dentro da propriedade *onPress*. A imagem 40 ilustra esta alteração.

```

export default function App() {
  return (
    <View style={styles.container}>
      <Text name="tx" >Open up App.js to start working on your app!</Text>
      <Button
        title= "Botão Inicial"
        color= "blue"
        accessibilityLabel= "primeiro botão do app"
        onPress= {funcaoBotaoPrincipal}
      > </Button>
    </View>
  );
}
function funcaoBotaoPrincipal(){
  Alert.alert('Simple Button pressed')
}

```

Imagem 40: uso do Button com chamada de função

Nos próximos exemplos, faremos alterações nas propriedades do botão e de outro componente através da ação de toque. Para isso, será necessário usarmos os conceitos de estado, explicados no capítulo anterior.

A primeira coisa que iremos fazer é criar um componente extra que contenha um botão e uma mensagem de texto. Este componente extra será adicionado ao arquivo principal. Como vimos nos capítulos anteriores, ele será adicionado normalmente como se fosse um componente nativo.

Crie então um arquivo extra chamado “componente.js”, para sua melhor organização, habitue-se a criar arquivos extras dentro de pastas. Este arquivo que estamos criando, faremos em forma de classe. A Imagem 41 ilustra este arquivo novo criado, somente com as informações (codificações) básicas e obrigatórias.

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

class Componente extends React.Component{
  render(){

  }
}

export default Componente;
```

Imagem 41: criação de um novo componente

Após criar a estrutura básica deste novo arquivo, iremos criar dois componentes para serem retornados dentro de “render”. Iremos colocar um *Text* e um *Button* para este retorno.

```
render(){
  return(
    <View>
      <Text></Text>
      <Button></Button>
    </View>
  )
}
```

Vamos criar também o estado inicial para estes dois componentes. Nele, iremos alterar a propriedade cor e texto.


```
state={
  corInicial: "green",
  textoInicial: "Original"
}
```

Voltando aos componentes criados, vamos definir o estado inicial da propriedade cor para o botão e texto para o *Text*. Adicionamos também um rótulo para o botão. A Imagem 42 ilustra as modificações feitas até aqui.

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

class Componente extends React.Component{
  state={
    corInicial: "green",
    textoInicial: "Original"
  }
  render(){
    return(
      <View>
        <Text>{this.state.textoInicial}</Text>
        <Button title="Botão Adicional" color={this.state.corInicial}></Button>
      </View>
    )
  }
}

export default Componente;
```

Imagem 42: criação de estados iniciais

Para testarmos o que implementamos até aqui, devermos voltar ao nosso arquivo principal (App.js), adicionar a importação e usar, na forma de tag, o novo componente criado. A Imagem 43 ilustra isso.

```

import { StatusBar } from 'expo-status-bar';
import React from 'react';
import { StyleSheet, Text, View, Button, Alert } from 'react-native';
import Componente from './src/componente';

export default function App() {
  return (
    <View style={styles.container}>
      <Text name="tx" >Open up App.js to start working on your app!</Text>
      <Button
        title= "Botão Inicial !!"
        color= "blue"
        accessibilityLabel= "primeiro botão do app"
        onPress= {funcaoBotaoPrincipal}
      > </Button>
      <Componente></Componente>
    </View>
  );
}

```

Imagem 43: usando o novo componente no código principal

Próximo passo agora é criar uma função que altera o estado das duas propriedades e definir a ação do botão para a chamada a esta função. A seguir, mostramos a função (“pressionado”) sendo criada:

```

pressionado = () =>{
  this.setState({textoInicial:'Texto
Alterado'});
  this.setState({corInicial:'blue'});
}

```

Na sequência, adicionamos a chamada desta função na propriedade onPress do botão criado:

```
render(){  
    return(  
        <View>  
        <Text>{this.state.textoInicial}</Text>  
        <Button  
            title="Botão Adicional"  
            color={this.state.corInicial}  
            onPress={this.pressionado}></Button>  
        </View>  
    )  
}
```

Navegação entre Telas

É comum em um projeto de um aplicativo para dispositivos móveis o uso de várias telas. Dificilmente você terá um aplicativo de tela única. Existem possibilidades para você tratar isso dentro do React Native. No início deste capítulo, iremos apresentar esta criação através da biblioteca oficial React-Navigation.

Embora ela seja oficial, é comum acontecer atualizações na biblioteca que gerem transtornos de compatibilidade. Desta forma, sugerimos que fique muito atento a documentação oficial da biblioteca, disponível em: <https://reactnative.dev/docs/navigation#react-navigation>.

Antes de inicializar seu projeto, é preciso que você crie algumas dependências. Para isso, ainda no terminal (ou *prompt* de comando) e **antes** de dar o comando *Expo Start*, faça os seguintes procedimentos:

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

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

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

Se você estiver utilizando um dispositivo com iOS para testar, será preciso ainda fazer a seguinte instalação:

```
Cd iOS  
pod install  
Cd ..
```

Neste primeiro exemplo de navegação entre telas, iremos trabalhar com pilha, no qual cada tela será acrescida de forma de pilha, uma em cima de outra. Para isso, precisaremos criar um container de navegação e dentro dele uma pilha de navegação. Cada tela assumirá o papel de pilha de tela.

Feito os procedimentos para as instalações das dependências, você pode executar seu projeto e abrir a IDE para codificação. O primeiro passo será alteramos as importações para:

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

A primeira coisa que você precisa ter em mente é que será criada uma constante, que será posteriormente exportada no JavaScript. Dentro desta constante, criaremos o container de navegação. A Imagem 43 foi extraída da página oficial. Ela ilustra o “esqueleto” básico desta codificação que estamos fazendo.

```
import 'react-native-gesture-handler';
import * as React from 'react';
import { NavigationContainer } from '@react-navigation/native';

const App = () => {
  return (
    <NavigationContainer>
      { /* Rest of your app code */ }
    </NavigationContainer>
  );
};

export default App;
```

Imagem 43: codificação inicial básica para navegação entre telas

Fonte: <https://reactnative.dev/docs/navigation>

Seguiremos o exemplo da documentação oficial. Na Imagem 44 o código é restaurado para receber duas telas, ou melhor, duas pilhas de telas a serem incorporadas no navegador de telas. As telas são chamadas (referenciadas) como “Home” e “ProfileScreen”. Estas referências precisam ser as mesmas na hora de definição da constante que define as telas. É usada a propriedade *name* em ambas e a propriedade *title* em uma delas.

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

const Stack = createStackNavigator();

const MyStack = () => {
  return (
    <NavigationContainer>
      <Stack.Navigator>
        <Stack.Screen
          name="Home"
          component={HomeScreen}
          options={{ title: 'Welcome' }}
        />
        <Stack.Screen name="Profile" component={ProfileScreen} />
      </Stack.Navigator>
    </NavigationContainer>
  );
};
```

Imagem 44: inserção das telas na pilha

Fonte: <https://reactnative.dev/docs/navigation>

Por fim, precisamos criar as telas que serão exibidas na pilha. As telas são criadas no mesmo arquivo, através de um escopo de constante. A Imagem 45 ilustra este processo.

```
const HomeScreen = ({ navigation }) => {  
  return (  
    <Button  
      title="Go to Jane's profile"  
      onPress={() =>  
        navigation.navigate('Profile', { name: 'Jane' })  
      }  
    />  
  );  
};  
  
const ProfileScreen = () => {  
  return <Text>This is Jane's profile</Text>;  
};
```

Imagem 45: criação das telas na pilha

Fonte: <https://reactnative.dev/docs/navigation>

Drawer Navigator

Neste capítulo iremos tratar no *Drawer Navigator* que consiste no menu de gaveta deslizante, muito comum na maioria dos aplicativos. Esta funcionalidade, além de permitir um fácil acesso a outras telas através de um menu, possibilita um recurso animado para o usuário.

Iremos apresentar aqui as configurações e propriedades básicas. Com o *Drawer Navigator* você pode definir se ele irá deslizar da direita para esquerda ou da esquerda para direita (normal). Você pode definir que ele fique fixo também na tela, entre outras configurações.

O primeiro passo antes de começarmos é instalar o pacote de dependências ao seu projeto. Faça a instalação de dependências de **todas** as bibliotecas indicadas no capítulo anterior (Navegação entre Telas) **além** destas abaixo:

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

Desta forma, todas as instalações de dependências para o seu projeto estão disponíveis a seguir:

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

npm install @react-navigation/native @react-
navigation/stack

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

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

O próximo passo será fazer a importação correta da biblioteca para dentro do nosso projeto, em específico, dentro do arquivo App.js.

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

Neste exemplo, iremos criar duas telas (screen) através de constante (const) no próprio documento principal, de modo que elas possam depois serem incorporadas no Drawer Navigator. Os códigos a seguir demonstram a criação destas duas telas.

```

function Tela01(){
  return(
    <View style={{ flex: 1, justifyContent:
'center', alignItems: 'center' }}>
      <Text>Tela 01</Text>
      <Text> Insira seus demais componentes
aqui...</Text>
    </View>
  );
}

function Tela02(){
  return(
    <View style={{ flex: 1, justifyContent:
'center', alignItems: 'center' }}>
      <Text>Tela 02</Text>
      <Text> Insira seus demais componentes aqui...</
Text>
    </View>
  );
}

```

Na sequência, iremos criar nosso *Drawer Navigator*, que nada mais é que uma “tela” (container) que irá agregar outras telas. Criaremos uma constante que invocará o método de criação do *Drawer Navigator*. Com esta constante, teremos condições de criar um navigator (container) e depois agregar as *screens* (telas).

```
const Drawer = createDrawerNavigator();

function MyDrawer(){
  return(
    <Drawer.Navigator>
      <Drawer.Screen name='Tela01'
component={Tela01}></Drawer.Screen>
      <Drawer.Screen name='Tela02'
component={Tela02}></Drawer.Screen>
    </Drawer.Navigator>
  );
}
```

Após criar o Drawer Navigator, precisaremos alterar a função principal do nosso projeto, no caso a função **App()**. Iremos criar um container de navegação e colocaremos o nosso *navigator* com as *screens*.

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

Após criarmos e testarmos nosso Drawer Navigator, vamos na sequência mostrar algumas propriedades nos quais podemos usá-las para personalizar melhor o visual:

- **drawerType**: tipo de animação será feita pela gaveta do *drawer*. Há três tipos: **front**, no qual a gaveta vai sobrepor a tela de baixo; **back**, no qual ela dá a sensação de empurrar a tela de baixo e **slide**, dando a sensação dum slide empurrando a tela de baixo;
- **drawerPosition**: define a lateral onde virá a gaveta do *drawer*, da esquerda para direita (**left**) ou da direita para esquerda (**right**);
- **openByDefault**: define se a gaveta do *drawer* virá aberta (**true**) quando a tela for renderizadas ou não (**false**);
- **backBehavior**: define qual tela da gaveta *drawer* virá marcada quando ela for aberta. Ordem (**order**) para definir o primeiro item da lista de telas; histórico (**history**) para definir a última tela acessada pelo usuário; nenhuma (**none**) para não definir e rota inicial (**initialRoute**) para definir alguma em específico;
- **drawerStyle**: define algumas configurações especiais da gaveta *drawer*, como tamanho de sua ocupação, cores entre outros;
- **overlayColor**: define a cor que a tela de baixo ficará quando a gaveta *drawer* for deslizada;

A Imagem 46 ilustra o esquema de montagem final, para que vocês entendam o processo. Cada tela (screen) é adicionada a um *Drawer Navigator*, este por sua vez, é adicionado a um *Navigator Container*. Caso precise fazer alguma personalização no *drawer*, será feito através do *Drawer Navigator* em forma de propriedade. A documentação oficial está disponível em: <https://reactnavigation.org/docs/drawer-navigator/>

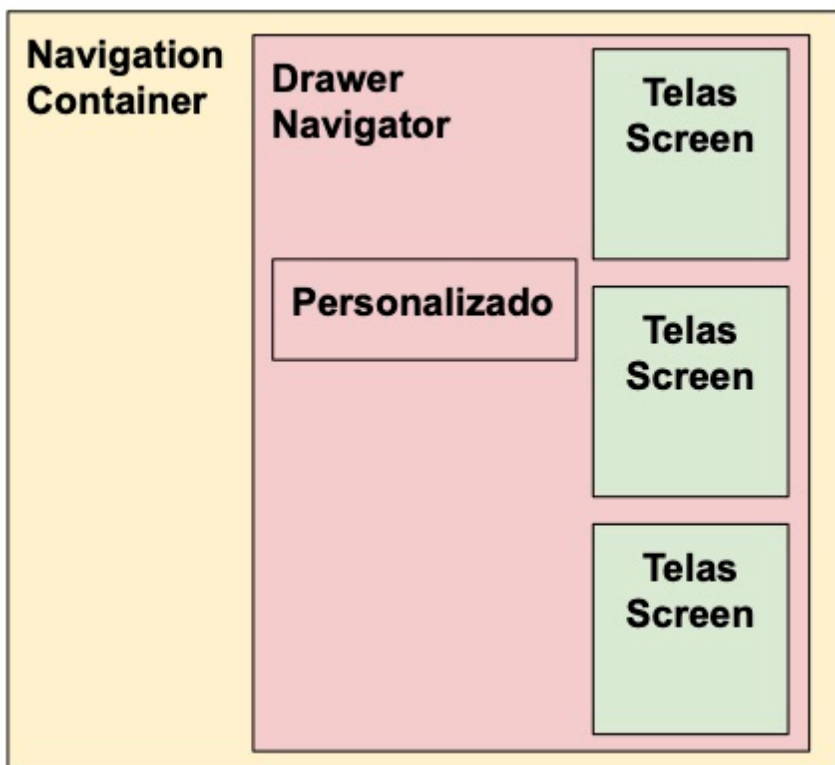


Imagem 46: esquema do drawer

Bottom Tab Navigator

Neste capítulo vamos apresentar o componente *Bottom Tab Navigator*, que representa aquela barra que fica na parte inferior da tela. Pode-se usar ícones ou não para representar. A interação nesta barra representa navegação (alteração entre telas).

Antes de iniciarmos a explicação do *Bottom Tab Navigator*, iremos relembrar a Imagem 46 do capítulo anterior. Observe que o *Drawer Navigator* está inserido dentro de um *Navigator Container*. Este container não é exclusivo do *drawer*, ele comporta também o *Bottom Tab Navigator* e o próprio *Stack Navigator*. Você precisará ficar atento ao arranjo da estrutura de seu projeto, pois um container não pode ser inserido dentro de outro.

Existem formas diferentes de você usar (codificar) estes componentes de navegação, mas há também um modo mais simplificado e comum para os três (*Drawer*, *Bottom Tab Navigator* e *Stack*). Neste modo, você cria uma constante que indica o construtor da navegação. Posteriormente, você faz o arranjo em duas estruturas: uma chamando sua constante “ponto” (.) *Navigator* e outra inserindo as telas através de sua constante “ponto” (.) *Screen*.

O *Bottom Tab Navigator* você poderá observar e acompanhar a documentação oficial através destes links: <https://reactnavigation.org/docs/tab-based-navigation> e

também este: <https://reactnavigation.org/docs/bottom-tab-navigator/>.

Primeiro passo será você fazer a importação das bibliotecas a serem usadas. Neste caso, vamos precisar do Navigator Container e do Bottom Tab Navigator.

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

Para que possamos ver o resultado nesta navegação, será preciso que você crie as demais telas. Neste exemplo, iremos criar três telas para navegar. No código abaixo, apresenta a codificação de uma, faça o mesmo para as demais, alterando obviamente o nome das telas.

```
function Tela01(){  
  return(  
    <View style={{flex:1,  
justifyContent:'center', alignItems:'center',  
backgroundColor:'#F5DEB3'}}>  
      <Text>Tela 01</Text>  
      <Text> Estamos dentro da Tela 01</Text>  
    </View>  
  );  
};
```


Agora vamos criar a nossa contente que irá chamar o método construtor de Bottom Tab Navigator.

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

Na sequência, vamos alterar a estrutura da função principal (App). Vamos criar um Navigator Container, inserir o Navigator (através da nossa constante) e inserir as três telas (também através de nossa constante).

```
export default function App() {  
  return (  
    <NavigationContainer>  
      <Tab.Navigator>  
        <Tab.Screen name="Tela01"  
component={Tela01}></Tab.Screen>  
        <Tab.Screen name="Tela02"  
component={Tela02}></Tab.Screen>  
        <Tab.Screen name="Tela03"  
component={Tela03}></Tab.Screen>  
      </Tab.Navigator>  
    </NavigationContainer>  
  );  
}
```

Até este ponto é possível você testar e observar a navegação e troca de telas .

O próximo passo será personalizar as abas desta barra. Vamos inserir ícones e trata-los para quando estão selecionados ou não. Neste exemplo, iremos usar uma biblioteca do Ion Icons e alterar as propriedades do Navigator. A documentação desta biblioteca você pode acessar neste link: <https://ionicons.com/>. Acessando e selecionando algum ícone, irá ver o nome do ícone. Na documentação há uma explicação sobre outras formas de usar estes objetos.

Primeiramente, vamos inserir a importação da biblioteca do Ion Icons.

```
import Ionicons from 'react-native-vector-  
icons/Ionicons';
```

Posteriormente, vamos alterar duas propriedades do Navigator, a `screenOptions` e a `tabBarOptions`. Na primeira iremos definir ícones para as abas. Definiremos um ícone para quando a aba estiver selecionada (`focused`) e para quando a aba não estiver selecionada. Por isso, será necessário uma estrutura encadeada de condições (`if`), uma para saber qual aba estamos tratando e outra para tratar qual será o ícone atribuído. Temos a opção ainda de definir tamanho e cor dos elementos. Caso nossa opção seja pelo padrão, basta retorna-los então sem qualquer tipo de definição. O código a seguir apresenta esta implementação.

```

<Tab.Navigator
  screenOptions={({route}) => ({
    tabBarIcon: ({focused, color, size})
=> {
      let iconName;
      if(route.name=== 'Tela01'){
        iconName = focused
          ? 'ios-information-circle'
          : 'ios-information-circle-
outline';
      }else if (route.name === 'Tela02')
      {
        iconName = focused
          ? 'ios-list-box'
          : 'ios-list';
      }else if (route.name === 'Tela03'){
        iconName = focused
          ? 'ios-add-circle'
          : 'ios-add-circle-outline';
      }
      return <Ionicons name={iconName}
size={size} color={'blue'} />;
    },
  )}) >

```

Agora iremos alterar a segunda propriedade, definido a cor do texto da aba para quando ela estiver ativa ou inativa. O código abaixo apresenta esta implementação. Observe que ela foi feita dentro da tag <Tab.Navigator> lembrando

que Tab é a nossa constante e Navigator a estrutura de navegação.

```
tabBarOptions={{  
  activeTintColor: 'tomato',  
  inactiveTintColor: 'gray',  
}}
```

WebView - Inserção PDF

Neste capítulo vamos trabalhar com o componente WebView para seu uso como um container para um documento em PDF. O WebView era um componente nativo da biblioteca “react-native”, ou seja, você poderia usar normalmente assim como faz com Text ou View. No entanto, ela foi descontinuada da biblioteca principal e agora para você usar terá que fazer a instalação de uma biblioteca específica. Esta biblioteca específica você pode verificar neste link: <https://github.com/react-native-community/react-native-webview>.

Primeiro passo é instalar então a biblioteca do WebView antes de você inicializar o projeto. Faça isso no terminal (ou prompt) através do seguinte comando:

```
npm install react-native-webview --save
```

Na sequência, você precisa fazer o link da biblioteca instalada ao seu projeto. Apesar de ter instalado dentro do seu projeto, ela não é “puxada” de forma automática, por ser uma biblioteca “não-oficial”. Para fazer o link, execute o seguinte comando ainda no terminal:

```
npx react-native link
```

Atenção: os procedimentos de instalação que são mostrados atualmente (setembro/2020) na biblioteca do WebView estão desatualizados. Tanto a forma de instalar quanto a forma de fazer o link. Para verificar a forma correta, leia o procedimento oficial em: <https://reactnative.dev/docs/linking-libraries-ios>.

No seu código, faça a importação da biblioteca, através do procedimento:

```
import {WebView} from 'react-native-webview';
```

Para usar o componente WebView é bem simples. Basta inseri-lo no seu layout e definir a propriedade *source*. Nesta propriedade, usaremos a configuração da *uri*, ou seja, vamos buscar o documento PDF na web de forma automática e trazê-lo para dentro (montado) do nosso aplicativo. Na sequência, mostramos como isso é feito, trazendo como exemplo um artigo em PDF da Jornada Científica do IFSULDEMINAS.

```
<WebView source={{uri: 'https://jornada.ifsuldeminas.edu.br/index.php/jcmch4/jcmch4/paper/viewFile/3012/2420'}}></WebView>
```

WebView - Inserção Vídeo

Neste capítulo vamos usar o mesmo componente WebView para mostrar vídeos do Youtube. No primeiro exemplo, vamos mostrar a tela toda, usando a propriedade `uri`. No segundo exemplo, vamos extrair e usar somente o vídeo do Youtube, através de codificação direta em Html.

Para o primeiro exemplo, além de definir a propriedade `source`, vamos definir a propriedade `flex` (já vimos isso aqui neste material) e habilitar o uso de JavaScript. Para você carregar uma página inteira do Youtube no seu aplicativo, use como base o seguinte código para o componente WebView:

```
<WebView style={{flex:1}}
  javascriptEnabled={true} source={{uri:
  'https://youtu.be/EdzMSFm7NAk'}} ></WebView>
```

Para o segundo exemplo, vamos usar somente o vídeo, desconsiderando as demais informações que são apresentadas pelo Youtube. Para isso, em vez de usar a propriedade `uri` vamos usar a `html`. Neste caso, vamos codificar em html fazendo a chamada do vídeo e, é claro, fazendo a chamada pelo link. Para este exemplo, nós podemos fazer o uso de outras configurações como largura e altura do vídeo, exibição em *fullscreen* ou não.

```
<WebView style={{flex:1}}
  javaScriptEnabled={true} source={{html:
"<html><boby><iframe width='800'
height='400' src='https://www.youtube.com/
embed/EdzMSFm7NAk' frameborder='0'
allowfullscreen> </iframe></body></html>"}}}
></WebView>
```


Referências

ESCUDELÁRIO, B.; PINHO, D.; React Native: Desenvolvimento de Aplicativos Mobile com React. Casa de Código, 2020.

REACT NATIVE. Site oficial. Acessado em maio/2020. Disponível em: <https://reactnative.dev/>.