Professor Rodrigo Bossini
https://sites.google.com/site/professorrodrigobossini/

Introdução a React & React Native

1 Introdução

Neste material iremos desenvolver uma aplicação que permite ao usuário armazenar lembretes de atividades que pretende realizar posteriormente. Trata-se de uma aplicação simples que ilustra os principais conceitos do desenvolvimento de aplicações com React Native.

2 Desenvolvimento

2.1 (Criando o App) O primeiro passo é criar um novo aplicativo React Native, o que pode ser feito com o comando

```
expo init  app-lembretes
```

Utilize o template **blank**.

2.2 (Navegando até o diretório do projeto) Abra um terminal ou prompt de comando e navegue até o diretório em que a sua aplicação foi criada com o comando

```
cd diretorio
```

2.3 (Abrindo o VS Code) Abra uma instância do VS Code vinculada a esse diretório com o comando

```
code .
```

Neste instante você já pode fechar o terminal que usou para abrir o VS Code.

2.4 (Abrindo um terminal no VS Code) No VS Code, clique em Terminal >> New Terminal para trabalhar com um terminal integrado ao IDE, o que pode ser bastante conveniente.

2.5 (Executando a aplicação Expo) No terminal integrado do VS Code, digite o comando a seguir para iniciar o ambiente Expo.

```
npm start
```

2.6 (Executando a aplicação em um dispositivo ou emulador) No terminal, você pode digitar "**a**" para iniciar um emulador Android. Também dá para clicar em "Run on Android device/emulator no cliente do Expo.

2.7 (Interface gráfica da aplicação) A aplicação terá **duas regiões principais**: uma que será utilizada para o usuário inserir novos lembretes e a outra, logo abaixo, para mostrar uma lista de lembretes. Assim, abra o arquivo **App.js** e faça o ajuste exibido na Listagem 2.7.1. Note que há um componente **View** para cada parte de interesse. Também iremos retirar os estilos incluídos por padrão no template.

Listagem 2.7.1

```
export default function App() {
return (
<View>
<View>
{/* usuário irá inserir lembretes aqui*/}
</View>
<View>
{/*Aqui será exibida a lista de lembretes*/}
</View>
</View>
);
}
```

2.8(Componentes para entrada de lembretes) A seguir, vamos adicionar os componentes que viabilizam a inserção de lembretes. Um **InputText** permite que o usuário insira texto. E um **Button** fará com que seja possível a inserção. Veja a Listagem 2.8.1.

Listagem 2.8.1

```
export default function App() {
return (
<View style={styles.container}>
<View>
{/* usuário irá inserir lembretes aqui*/}
<TextInput />
<Button
title="+"
/>
</View>
<View>
{/*Aqui será exibida a lista de lembretes*/}
</View>
</View>
);
}
```

Verifique o resultado no emulador. Note que o Button é visível mas o TextInput não. Vamos ajustar isso com algumas configurações semelhantes a CSS. Veja a Listagem 2.8.2.

Listagem 2.8.2

```
export default function App() {
return (
<View style={{padding: 50}}>
<View>
{/* usuário irá inserir lembretes aqui*/}
<TextInput placeholder="Lembrar..." style={{borderBottomColor: 'black', borderBottomWidth: 1, marginBottom: 4,
padding: 2}} />
<Button
title="+"
/>
</View>
<View>
{/*Aqui será exibida a lista de lembretes*/}
</View>
</View>
);
}
```

Note que estamos usando algo análogo a CSS “inline”, o que pode se tornar bastante inconveniente e difícil de se manter. Em breve passaremos a aplicar melhores práticas.

2.9 (Algumas propriedades “CSS”) Em geral, os componentes React usam o módulo chamado **Flexible Box**, muitas vezes chamado de “**flex model**”. As regras de disposição visual dos componentes são ditadas de acordo com esse modelo. Por padrão, os filhos de um componente são colocados um abaixo do outro, como está acontecendo com o Button e o TextInput. Podemos alterar isso ajustando atributos do flex model. Veja, na Tabela 2.9.1, as propriedades que iremos aplicar e seu significado.

Tabela 2.9.1

Propriedade	**Significado**
flexDirection	Responsável pela disposição relativa dos elementos. O valor **row** indica que eles devem ser colocados lado a lado.
justifyContent	Usado para dizer como os filhos são alinhados ao longo do eixo principal. Pode ser usado para alinhá-los à esquerda, direita, centralizar etc. A propriedade **space-between** indica que o espaço que sobra deve ficar entre eles.
alignItems	Usado para dizer como os filhos são alinhados no eixo perpendicular ao principal. A propriedade **center** os centraliza.

Veja sua aplicação na Listagem 2.9.1.

Listagem 2.9.1

```
export default function App() {
return (
<View style={{padding: 50}}>
<View style={{flexDirection: 'row', justifyContent: 'space-between', alignItems: "center"}}>
{/* usuário irá inserir lembretes aqui*/}
<TextInput placeholder="Lembrar..." style={{ width: '80%', borderBottomColor: 'black', borderBottomWidth: 1,
padding: 2}} />
<Button
title="+"
/>
</View>
<View>
{/*Aqui será exibida a lista de lembretes*/}
</View>
</View>
);
}
```

2.10 (Definindo estilos como objetos JSON) Como comentamos, o uso de estilos “inline” tende a ser incoveniente e difícil de se manter. Passaremos agora a definir objetos que contém as propriedades de interesse e então a aplicá-los aos componentes visuais. Para tal, basta usar o método **create** de **StyleSheet**. Veja a Listagem 2.10.1.

Listagem 2.10.1

```
const styles = StyleSheet.create({
telaPrincipalView: {
padding: 50
},
lembreteView: {
flexDirection: 'row',
justifyContent: 'space-between',
alignItems: "center"
},
lembreteInputText: {
width: '80%',
borderBottomColor: 'black',
borderBottomWidth: 1,
padding: 2
}
});
```

Note que há um objeto JSON principal e que ele possui diversos objetos JSON aninhados. Cada um deles abriga uma coleção de estilos a ser aplicada a um ou mais componentes. A Listagem 2.10.2 mostra como utilizá-los.

Listagem 2.10.2

```
export default function App() {
return (
<View style={styles.telaPrincipalView}>
<View style={styles.lembreteView}>
{/* usuário irá inserir lembretes aqui*/}
<TextInput placeholder="Lembrar..." style={styles.lembreteInputText} />
<Button
title="+"
/>
</View>
<View>
{/*Aqui será exibida a lista de lembretes*/}
</View>
</View>
);
}
```

2.11 (Capturando o texto digitado e exibindo no log) A seguir, vamos lidar com o evento de clique no botão para captura os lembretes que o usuário digitar. Para isso, vamos precisar do operador **useState** novamente, já que o componente está definido com uma função. Além disso, definiremos duas funções: uma que executa sempre que o conteúdo do campo textual muda e outra, que executa quando o botão é clicado. Quando o clique acontecer, iremos somente exibir o texto digitado no log, que pode ser visto no terminal utilizado para executar o Expo. Veja a Listagem 2.11.1.

Listagem 2.11.1

```
import React, {useState} from 'react';

export default function App() {
const [lembrete, setLembrete] = useState('');

//captura o texto digitado
const capturarLembrete = (lembrete) => {
setLembrete(lembrete)
};

//para adicionar o que foi digitado
const adicionarLembrete = () => {
console.log (lembrete);
}
return (
<View style={styles.telaPrincipalView}>
<View style={styles.lembreteView}>
{/* usuário irá inserir lembretes aqui*/}
<TextInput
placeholder="Lembrar..."
style={styles.lembreteInputText}
onChangeText={capturarLembrete}
value={lembrete}

/>
<Button
title="+"
onPress={adicionarLembrete}
/>
</View>
<View>
{/*Aqui será exibida a lista de lembretes*/}
</View>
</View>
);
}
```

Deverá ser possível ver o log no próprio terminal em que o Expo foi iniciado.

2.12 (Armazenando os lembretes em uma lista) Quando o clique ocorre, desejamos armazenar aquilo que o usuário digitou em uma lista. Trata-se de mais um detalhe que fará parte do estado do componente. O primeiro passo é definir um novo par variável/função com **useState**, como na Listagem 2.12.1.

Listagem 2.12.1

```
const [lembretes, setLembretes] = useState ([]);
```

Depois disso, precisamos ajustar o funcionamento da função que entra em execução quando o botão é clicado. Atualmente estamos apenas logando o que foi digitado. Desejamos adicionar o que foi digitado na lista. Usamos o operador **spread** do Javascript (…) para extrair todos os elementos do vetor existente adicionando-os a um novo logo a seguir, incluindo o novo lembrete digitado. A função setLembretes é usada para alterar o estado do componente. Ao final, exibimos o vetor no log, que pode ser visto no terminal. Veja a Listagem 2.12.2.

Listagem 2.12.2

```
//para adicionar o que foi digitado
const adicionarLembrete = () => {
setLembretes (lembretes => [...lembretes, lembrete]);
console.log (lembretes);
//console.log (lembrete);
}
```

2.13 (Exibindo a lista visualmente) A função map do Javascript permite especificar uma arrow function que indica o critério de mapeamento a ser aplicado aos elementos da coleção sobre a qual ela opera. Neste caso, estamos operando sobre uma coleção de strings (os lembretes), e desejamos que cada um deles seja mapeado para um componente React visual que o contém. Veja como fazê-lo na Listagem 2.13.1.

Listagem 2.13.1

```
<View>
{
/*Aqui será exibida a lista de lembretes*/
lembretes.map((lembrete) => <Text>{lembrete}</Text>)
}
</View>
```

Note que, no terminal, há uma mensagem de erro, parecida com aquele exibida pela Figura 2.13.1, que diz que cada componente visual de uma lista deve ter uma chave única.

Figura 2.13.1

```
Warning: Each child in a list should have a unique "key" prop.%s%s
 See https://fb.me/react-warning-keys for more information.%s,
```

Neste instante, esse não é um grande problema para a aplicação, mas já vamos ajustar para aprender como usar. Neste caso, diremos que a chave é o próprio texto de exibido por cada componente (muito embora não seja verdade que ele necessariamente seja único para cada um deles). Veja a Listagem 2.13.2.

Listagem 2.13.2

```
<View>
{
/*Aqui será exibida a lista de lembretes*/
lembretes.map((lembrete) => <Text key={lembrete}>{lembrete}</Text>)
}
</View>
```

Veja, na Figura 2.13.2, que o erro deixou de existir e que agora temos um Warning, que diz que chaves duplicadas foram encontradas (isso depende de você ter digitado valores repetidos ou não e armazenado eles na sua lista) e que que coisas ruins poderão acontecer no futuro.

Figura 2.13.2

```
Warning: Encountered two children with the same key, `%s`. Keys sh
ould be unique so that components maintain their identity across u
pdates. Non-unique keys may cause children to be duplicated and/or
 omitted — the behavior is unsupported and could change in a futur
e version.%s, Sghdh,
    in RCTView (at App.js:35)
```

2.14 (Estilizando os itens da lista) A seguir, vamos tornar mais agradáveis visualmente os itens da lista, aplicando estilos como feito anteriormente. Para isso, vamos definir uma nova **View** para englobar o componente **Text**, já que ele não permite a definição de muitas variações de estilos. Veja a Listagem 2.14.1.

Listagem 2.14.1

```
<View>
{
/*Aqui será exibida a lista de lembretes*/
lembretes.map((lembrete) =>
<View><Text key={lembrete}>{lembrete}</Text></View>)
}
</View>
```

A seguir, vamos definir um novo objeto JSON que agrupa as configurações a serem aplicadas a cada item na lista. Veja a Listagem 2.14.2.

Listagem 2.14.2

```
const styles = StyleSheet.create({
telaPrincipalView: {
padding: 50
},
lembreteView: {
flexDirection: 'row',
justifyContent: 'space-between',
alignItems: "center"
},
lembreteInputText: {
width: '80%',
borderBottomColor: 'black',
borderBottomWidth: 1,
padding: 2,
marginBottom: 8
},
itemNaLista: {
padding: 12,
backgroundColor: '#CCC',
borderColor: '#000',
borderWidth: 1,
marginBottom: 8,
borderRadius: 8
}

});
```

Resta, evidentemente, atribuir o estilo ao componente, como na Listagem 2.14.3. Além disso, ajuste a chave para que ela seja atribuída ao componente raiz de cada item da lista.

Listagem 2.14.3

```
<View>
{
/*Aqui será exibida a lista de lembretes*/
lembretes.map((lembrete) =>
<View key={lembrete} style={styles.itemNaLista}><Text>{lembrete}</Text></View>)
}
</View>
```

2.15 (Barra de rolagem vertical) Conforme novos itens são adicionados à lista, ela vai tomando a tela inteira verticalmente, pouco a pouco. Em um determinado instante, novos itens não mais caberão na tela e, no momento, não estamos utilizando nenhum recurso que faça com que uma barra de rolagem apareça para o usuário. O ajuste é simples: trocaremos a View raiz por um ScrollView, componente responsável por adicionar uma barra de rolagem, caso necessário. Ele será compilado para o componente nativo da plataforma que estiver em uso, como ocorre com os demais. Veja a Listagem 2.15.1. Não esqueça de viabilizar o uso do símbolo trazendo-o para o contexto com a instrução **import**.

Listagem 2.15.1

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

<ScrollView>
{
/*Aqui será exibida a lista de lembretes*/
lembretes.map((lembrete) =>
<View key={lembrete} style={styles.itemNaLista}><Text>{lembrete}</Text></View>)
}
</ScrollView>
```

2.16 (FlatList: Desempenho importa) Um componente do tipo ScrollView tem funcionamento possivelmente indesejável, dependendo das circunstâncias. Ocorre que esse tipo de componente renderiza todos os componentes que serão seus filhos, muito embora não necessariamente todos serão exibidos. Talvez a coleção com a qual ele lida contenha milhões de elementos mas ele vá exibir somente os cinco primeiros. Independente disso, o seu funcionamento será o mesmo: ele renderizará todos os componentes que forem seus filhos.

O componente **FlatList** funciona diferente. Ele somente renderiza os elementos que necessariamente serão exibidos na tela. Ou seja, a renderização se dá sob demanda, conforme o usuário utiliza a barra de rolagem. Veja seu uso na Listagem 2.16.1.

Listagem 2.16.1

```
{/*substuir o ScrollView e todo o seu conteúdo*/}
<FlatList
data={lembretes}/*coleção de lembretes*/
renderItem={ /*mapeamento*/
lembrete => ( /*dado um lembrete, gera uma view*/
<View style={styles.itemNaLista}>
<Text>{lembrete.item}</Text>
</View>
)
}
/>
```

Ainda temos problemas com a chave única que cada item de uma lista deveria ter. Ocorre que uma FlatList é capaz de gerenciar isso automaticamente, desde que a coleção que ela manipula seja uma coleção de objetos JSON e que cada um deles especifique a sua própria chave. No momento, nossa coleção é uma lista de strings. Vamos ajustar isso na função que entra em execução quando o botão é clicado. Ela adicionará um objeto JSON que possui uma propriedade chamada **key** e o texto digitado, ao invés de adicionar somente o texto. Além disso, vamos adicionar um novo item ao estado do componente: um **contador** que será usado com id sequencial para cada lembrete. Veja a Listagem 2.16.2.

Listagem 2.16.2

```
//novo item para o estado do componente
const [contadorLembretes, setContadorLembretes] = useState(0);


//ajuste na função que adiciona lembretes
const adicionarLembrete = () => {
setLembretes ((lembretes) =>{
console.log (lembretes);
//console.log (lembrete);
setContadorLembretes (contadorLembretes + 1);
return [...lembretes, {key: contadorLembretes.toString(), value: lembrete}];
} );
}

{/* a FlatList agora fica assim */}
<FlatList
data={lembretes}/*coleção de lembretes*/
renderItem={ /*mapeamento*/
lembrete => ( /*dado um lembrete, gera uma view*/
<View style={styles.itemNaLista}>
<Text>{lembrete.item.value}</Text>
</View>
)
}
/>
```

Referências

React – A JavaScript library for building user interfaces. 2020. Disponível em <https://reactjs.org/> Acesso em janeiro de 2020.

React Native · A framework for building native apps using React. 2020. Disponível em <https://facebook.github.io/react-native/>. Acesso em janeiro de 2020.