

# React Native

## 1 Introdução

Neste material, vamos desenvolver uma aplicação que permite ao usuário **gerenciar a sua lista de lembretes**. Ele poderá

- cadastrar lembretes (**Create**)
- listar lembretes (**Read**)
- atualizar lembretes (**Update**)
- remover lembretes (**Delete**)

Ou seja, um **CRUD básico de lembretes**. Observe que esta versão da aplicação opera **inteiramente em memória volátil**.

## 2 Desenvolvimento

### Criando a aplicação

Comece criando a aplicação. Observe que optamos por utilizar um template com suporte à linguagem Typescript.

```
npx create-expo-app -t expo-template-blank-typescript lembretes
```

Uma pasta chamada lembretes terá sido criada para você. Use

```
code lembretes
```

para abrir uma instância do VS Code vinculada a ela. No VS Code, clique Terminal >> New Terminal, obtendo um terminal interno. Como vamos testar a aplicação no navegador num primeiro momento, instale as dependências para tal com

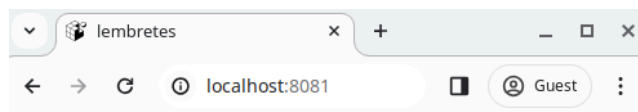
```
npx expo install react-native-web react-dom @expo/metro-runtime
```

Coloque a aplicação para funcionar com

```
npm start
```

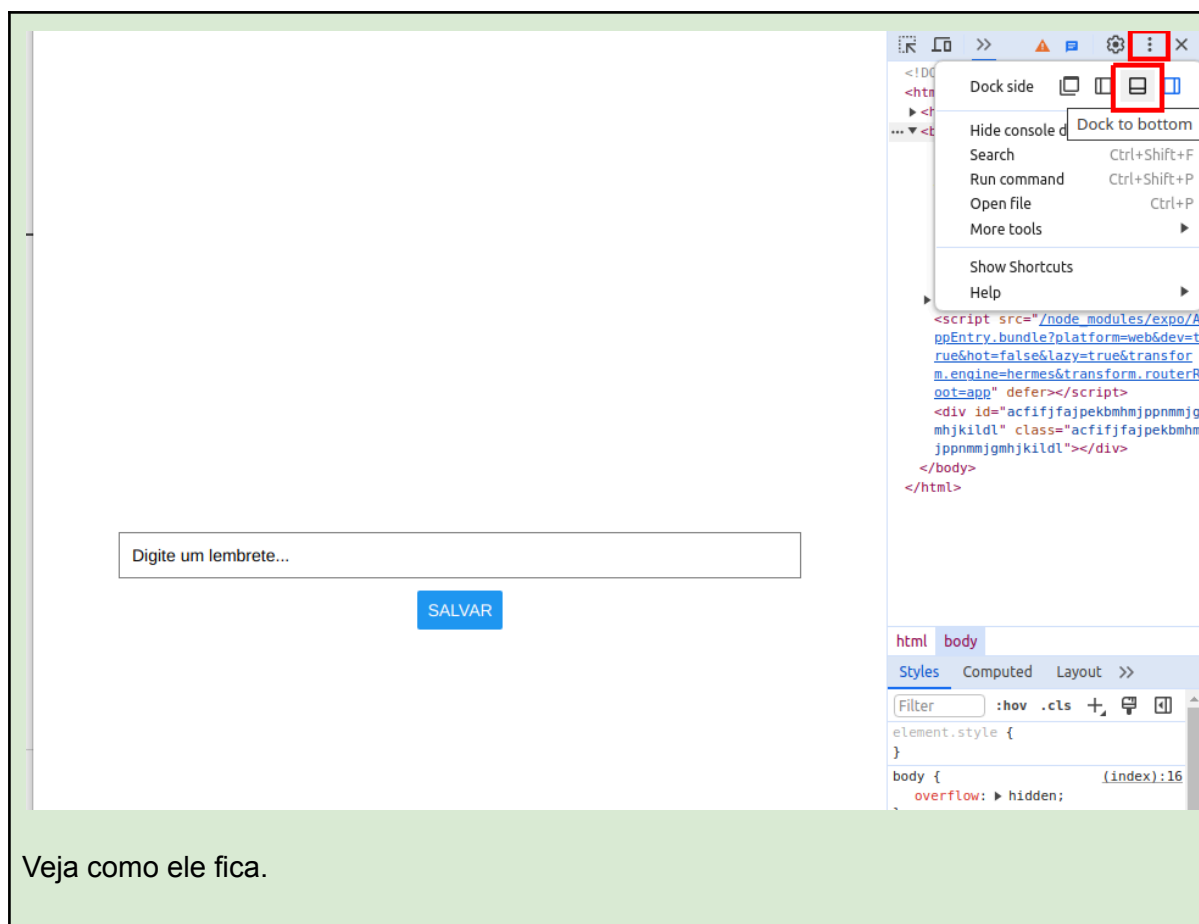
No terminal, aperte **w** para vê-la funcionar no navegador.

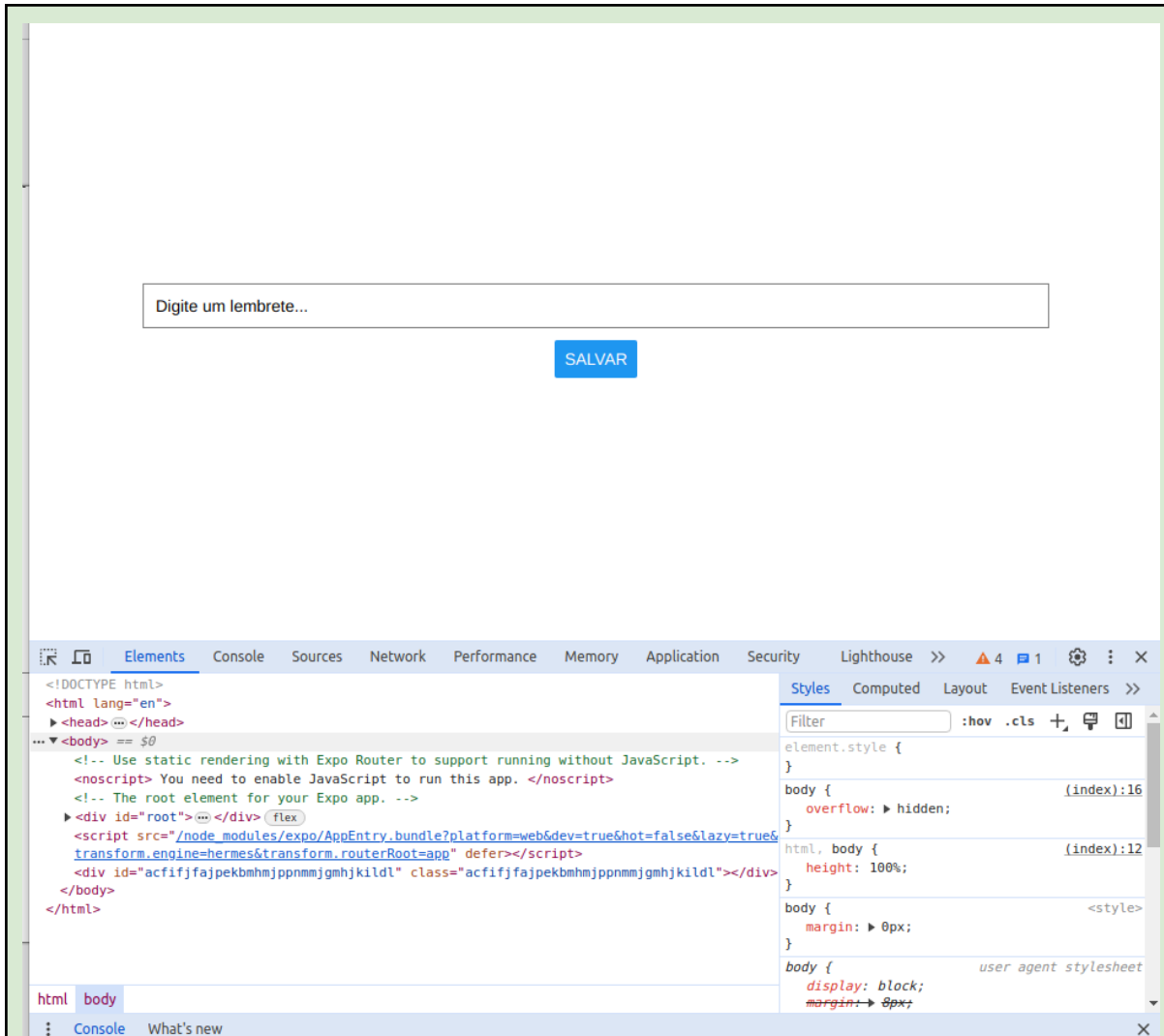
Veja o resultado esperado.



Open up App.tsx to start working on your app!

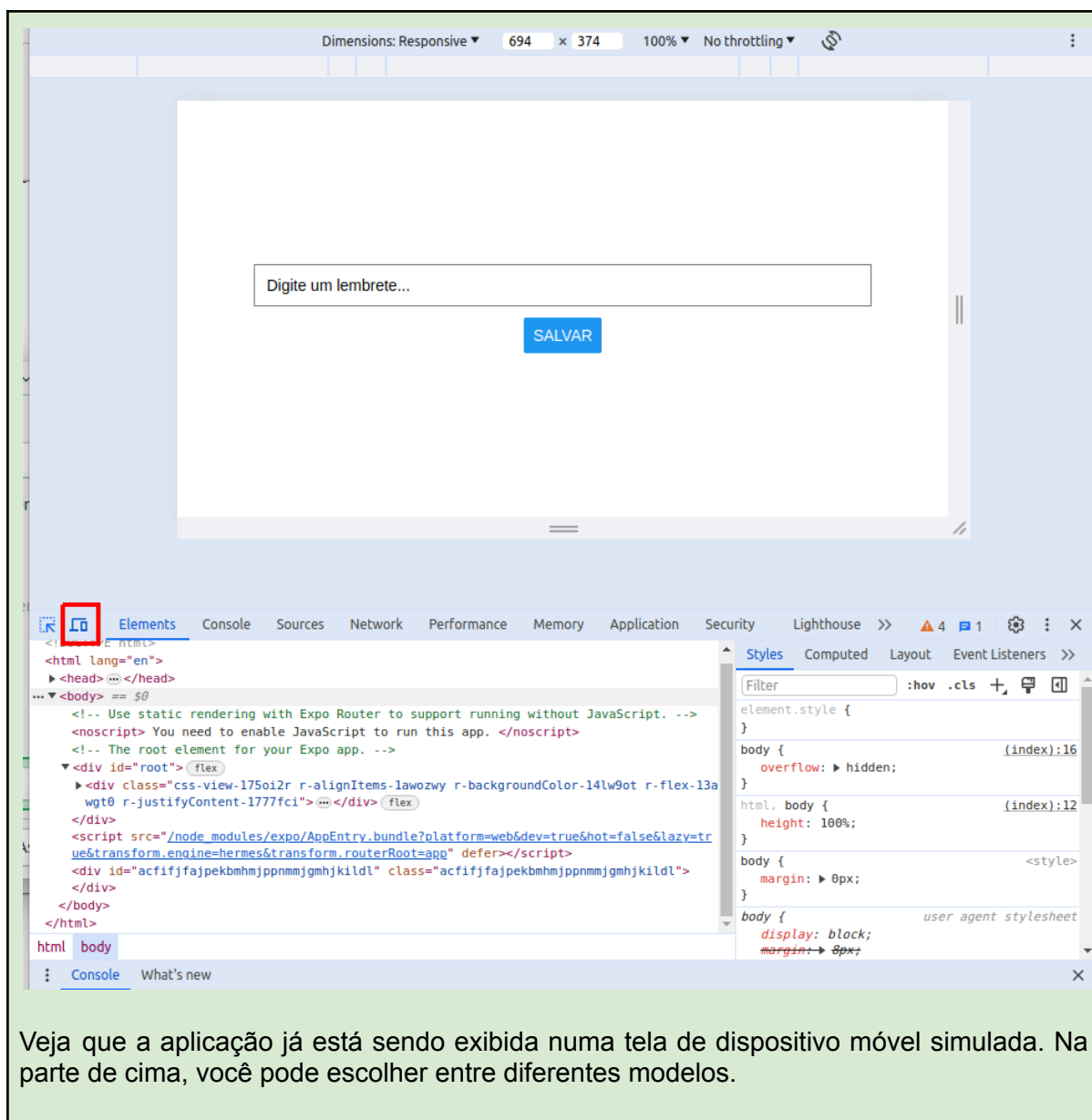
**Nota.** Para testar no navegador utilizando uma espécie de simulador de dispositivos móveis, no Google Chrome, abra o Chrome Dev Tools apertando CTRL + SHIFT + I. A seguir, posicione-o na parte de baixo para simplificar.



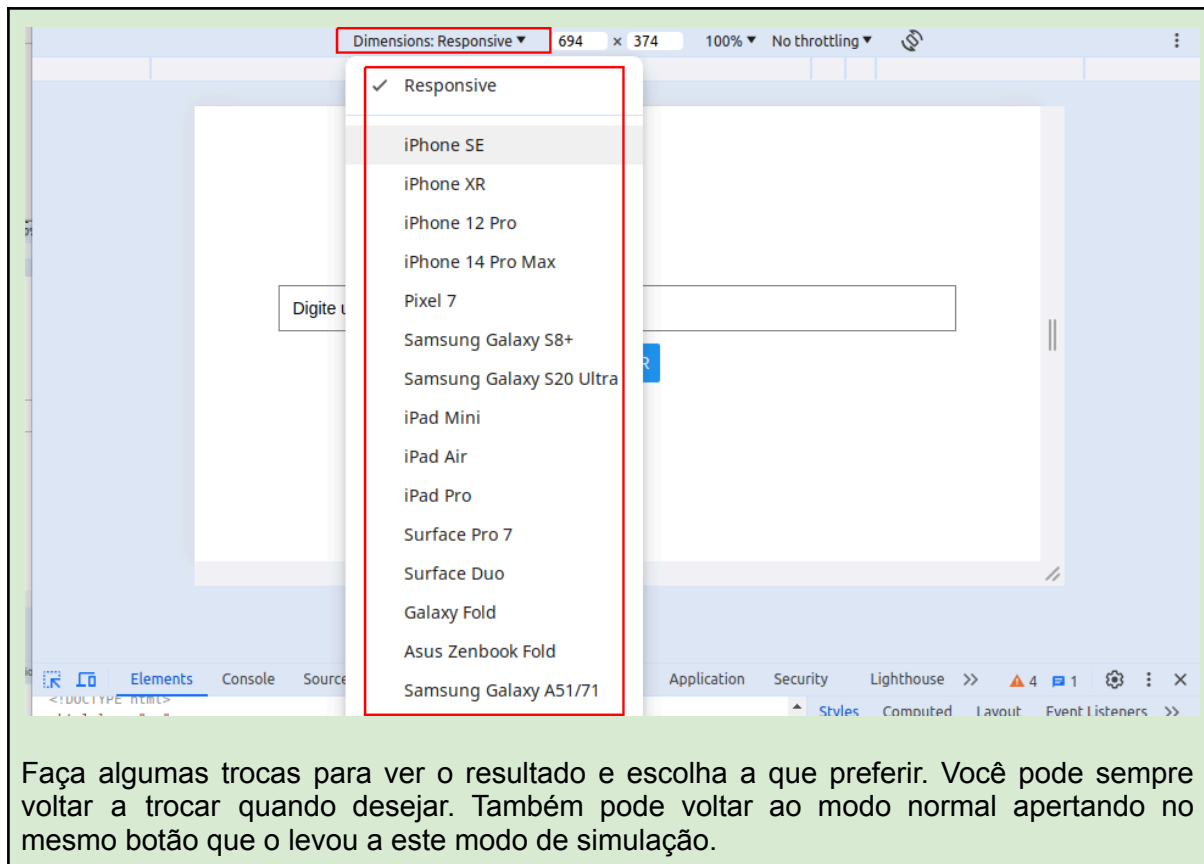


The screenshot shows a web browser window with a simple form. The form has a text input field containing the placeholder text "Digite um lembrete..." and a blue button labeled "SALVAR". Below the browser window, the Chrome DevTools interface is visible. The 'Elements' panel on the left shows the HTML structure of the page, including the root element and a script tag. The 'Styles' panel on the right shows the default styles for the body element, such as 'display: block' and 'margin: 0px'.

Agora, clique como destacado a seguir para abrir o simulador de dispositivos móveis.



Veja que a aplicação já está sendo exibida numa tela de dispositivo móvel simulada. Na parte de cima, você pode escolher entre diferentes modelos.



Faça algumas trocas para ver o resultado e escolha a que preferir. Você pode sempre voltar a trocar quando desejar. Também pode voltar ao modo normal apertando no mesmo botão que o levou a este modo de simulação.

Comece pelo arquivo **App.tsx**, fazendo uma “limpeza” inicial.

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

export default function App() {
  return (
    <View style={styles.container}>

    </View>
  );
}

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

## Componente textual para digitar os lembretes

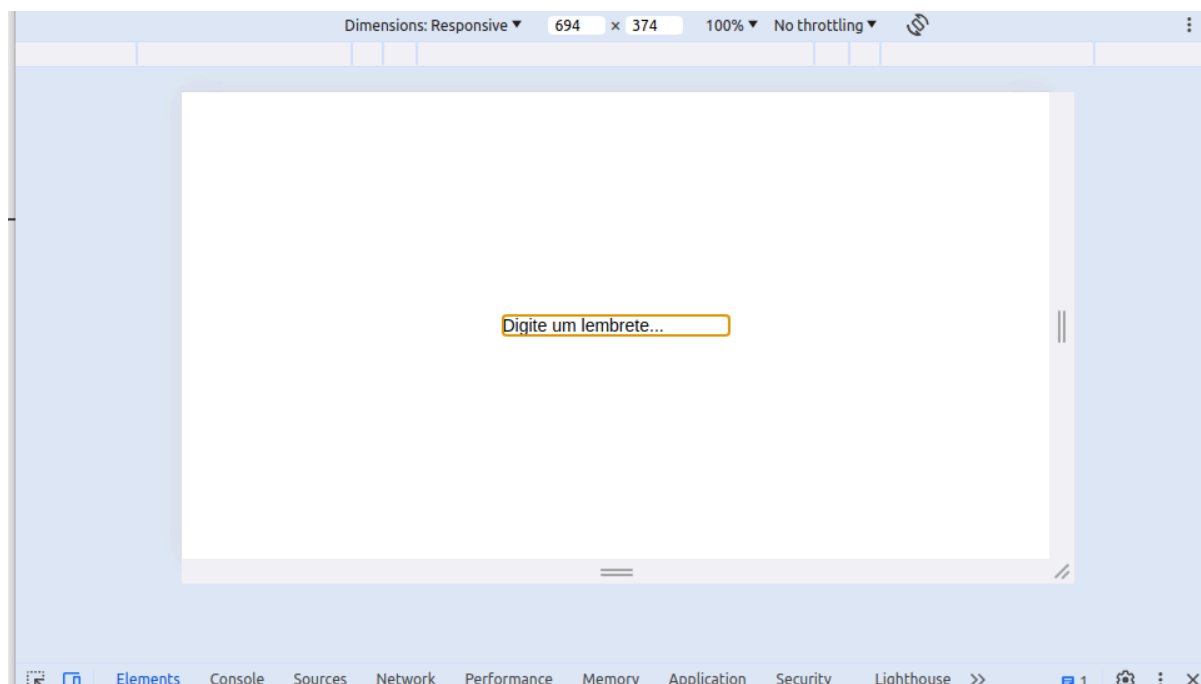
A seguir, vamos utilizar um componente do tipo **TextInput** para permitir que o usuário digite um lembrete. Veja um pouco da sua documentação e alguns exemplos.

<https://reactnative.dev/docs/textinput>

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

export default function App() {
  return (
    <View style={styles.container}>
      <TextInput
        placeholder='Digite um lembrete...'
      />
    </View>
  );
}
...
```

Veja o resultado esperado. O TextInput nasce sem praticamente nenhum “estilo”.



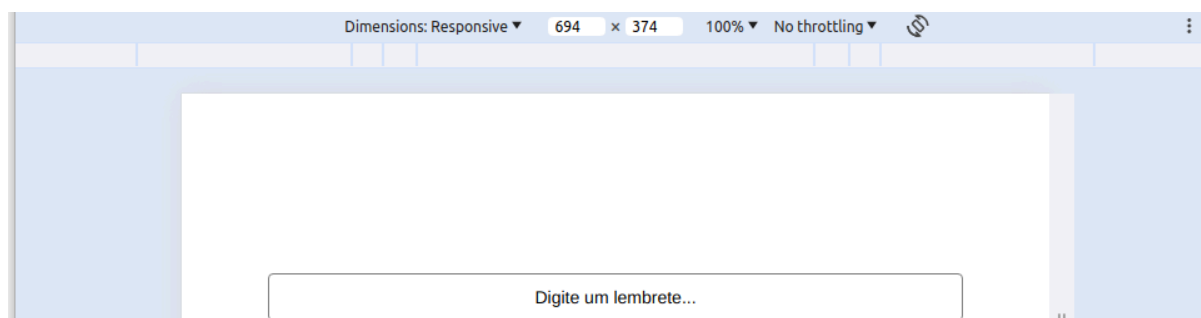
Vamos criar um objeto de estilos para aplicar sobre ele.



```
export default function App() {
  return (
    <View style={styles.container}>
      <TextInput
        style={styles.input}
        placeholder='Digite um lembrete...'
      />
    </View>
  );
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#fff',
    alignItems: 'center',
    justifyContent: 'center',
  },
  input: {
    width: '80%',
    borderColor: 'gray',
    borderWidth: 1,
    marginBottom: 10,
    padding: 10,
    textAlign: 'center',
    borderRadius: 4
  },
});
```

Observe como já melhorou bastante.



## Variável de estado para um lembrete e componente controlado

O componente `TextInput` tem a seguinte característica: quando o usuário digita um valor, desejamos armazená-lo em uma variável que, a qualquer momento, possa ser utilizada na adição de um novo lembrete. Além disso, o valor dessa variável será utilizado para especificar o que o componente exibe. Ou seja, ela será uma **variável de estado** e o componente será **controlado**. Para isso, usamos o hook **`useState`** do React.

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

import {
  useState
} from 'react'

export default function App() {
  const [lembrete, setLembrete] = useState('')
  return (
    <View style={styles.container}>
      <TextInput
        style={styles.input}
        placeholder='Digite um lembrete...'
        value={lembrete}
        onChangeText={setLembrete}
      />
    </View>
  );
}
```

Se desejar testar, basta exibir o lembrete num componente textual logo abaixo. Ao editar o campo, a tela deve ser atualizada automaticamente, conforme você digita.

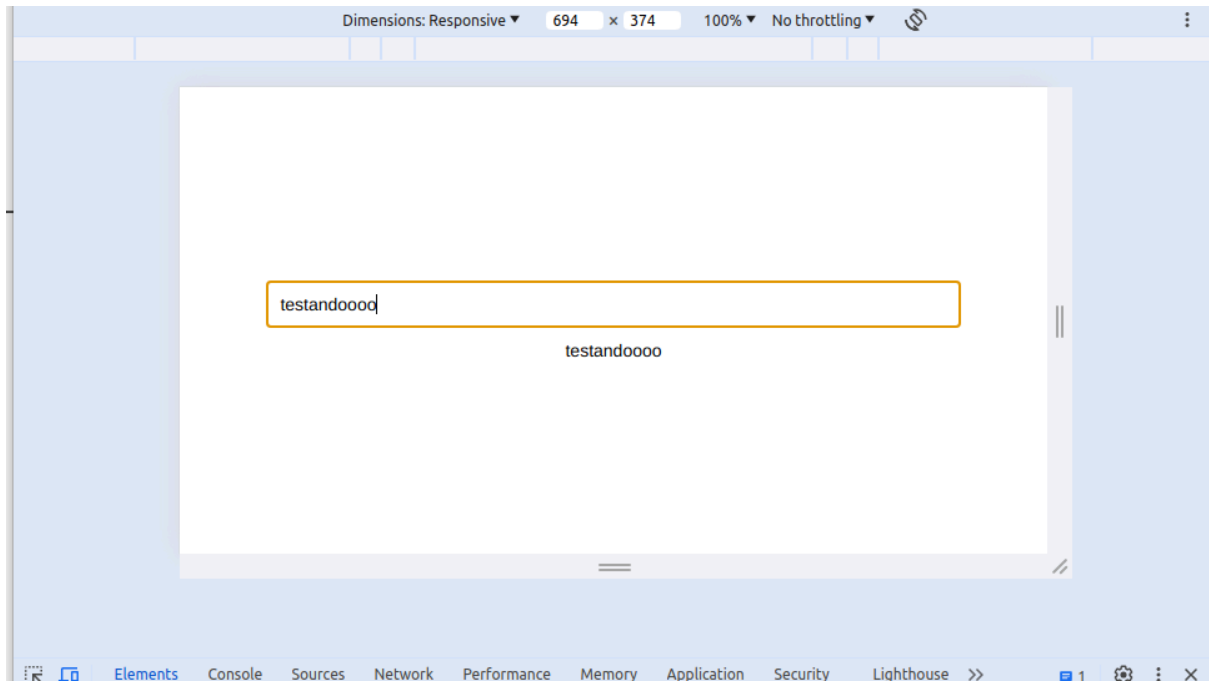
```
import {
  StyleSheet,
  Text,
  TextInput,
  View
} from 'react-native';

import {
  useState
} from 'react'

export default function App() {
  const [lembrete, setLembrete] = useState('')
  return (
    <View style={styles.container}>
      <TextInput
        style={styles.input}
        placeholder='Digite um lembrete...'
        value={lembrete}
        onChangeText={setLembrete}
      />
      <Text>{lembrete}</Text>
    </View>
  );
}

...
```

Veja o resultado.



A seguir, remova o componente textual utilizado no teste.

## Botão para adicionar lembretes

O próximo componente visual da tela será um botão que, quando clicado, adiciona um novo lembrete a uma lista de lembretes, que também precisa ser criada. Poderíamos usar um Button. Veja a sua documentação.

<https://reactnative.dev/docs/button>

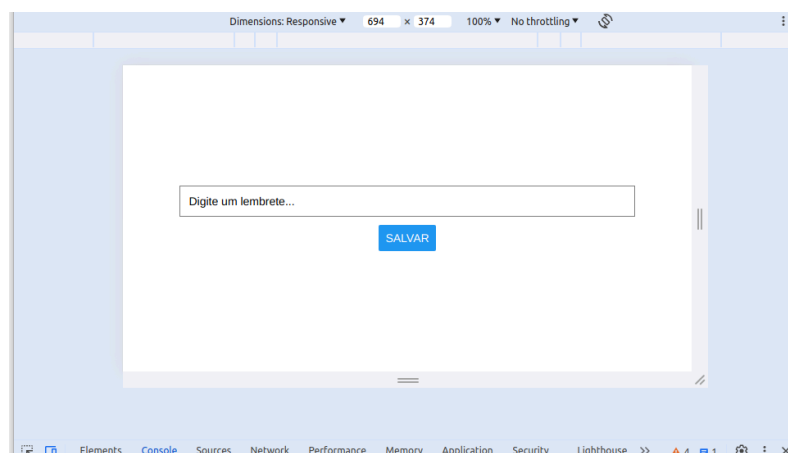
Veja como fica seu uso.

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

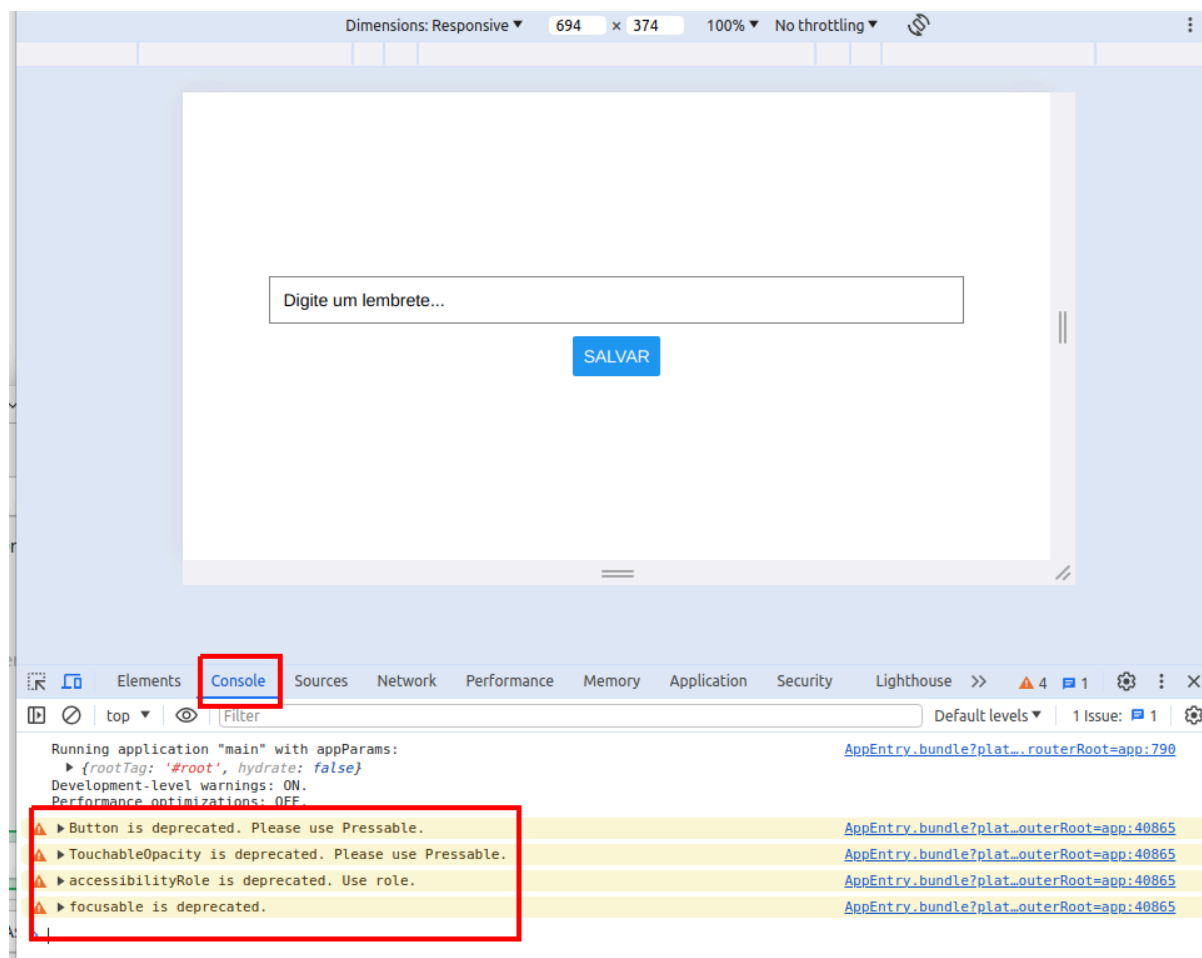
import {
  useState
} from 'react'

export default function App() {
  const [lembrete, setLembrete] = useState('')
  return (
    <View style={styles.container}>
      <TextInput
        style={styles.input}
        placeholder='Digite um lembrete...'
        value={lembrete}
        onChangeText={setLembrete}
      />
      <Button title='Salvar' />
    </View>
  );
}
```

E o resultado visual.



Entretanto, veja o que a mensagem exibida no **console** do Chrome Dev Tools nos diz.



Ela mostra que Button é um componente já obsoleto. E ainda sugere que utilizemos o componente Pressable. Observe que há, ainda, outras mensagens indicando que outros componentes também são obsoletos. Elas decorrem do uso de Button. Investigando a documentação de Pressable, acessível por meio do link a seguir, descobrimos que ele pode fazer o papel de Button e que, aparentemente, é mais flexível e tem mais opções, inclusive sendo capaz de detectar diferentes tipos de toque por parte do usuário.. Inspecione você mesmo.

<https://reactnative.dev/docs/pressable>

Assim, vamos substituir o Button por um Pressable. Por trazer mais opções, ele também dá um pouco mais de trabalho num primeiro momento.

```

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

...
export default function App() {
  const [lembrete, setLembrete] = useState('')
  return (
    <View style={styles.container}>
      <TextInput
        style={styles.input}
        placeholder='Digite um lembrete...'
        value={lembrete}
        onChangeText={setLembrete}
      />
      <Pressable>
        <Text>
          Salvar lembrete
        </Text>
      </Pressable>
    </View>
  );
}
...

```

A seguir, aplicamos objetos de estilos aos novos componentes, trocando a cor de fundo do Pressable e a cor de fonte do Text, além de ajustes de padding, borda, margem e centralização de texto.

```

...
export default function App() {
  const [lembrete, setLembrete] = useState('')
  return (
    <View style={styles.container}>
      <TextInput
        style={styles.input}
        placeholder='Digite um lembrete...'
        value={lembrete}
        onChangeText={setLembrete}
      />
      <Pressable
        style={styles.button}>
        <Text
          style={styles.buttonText}>
            Salvar lembrete
          </Text>
        </Pressable>
      </View>
    );
  }

  const styles = StyleSheet.create({
    container: {
      flex: 1,
      backgroundColor: '#fff',
      alignItems: 'center',
      justifyContent: 'center',
    },
    input: {
      width: '80%',
      borderColor: 'gray',
      borderWidth: 1,
      marginBottom: 10,
      padding: 10,
    },
    button: {
      width: '80%',
      backgroundColor: '#0096F3', //material design blue 500
      padding: 12,
      borderRadius: 4,
    },
    buttonText: {
      color: 'white',
      textAlign: 'center'
    }
  });
}

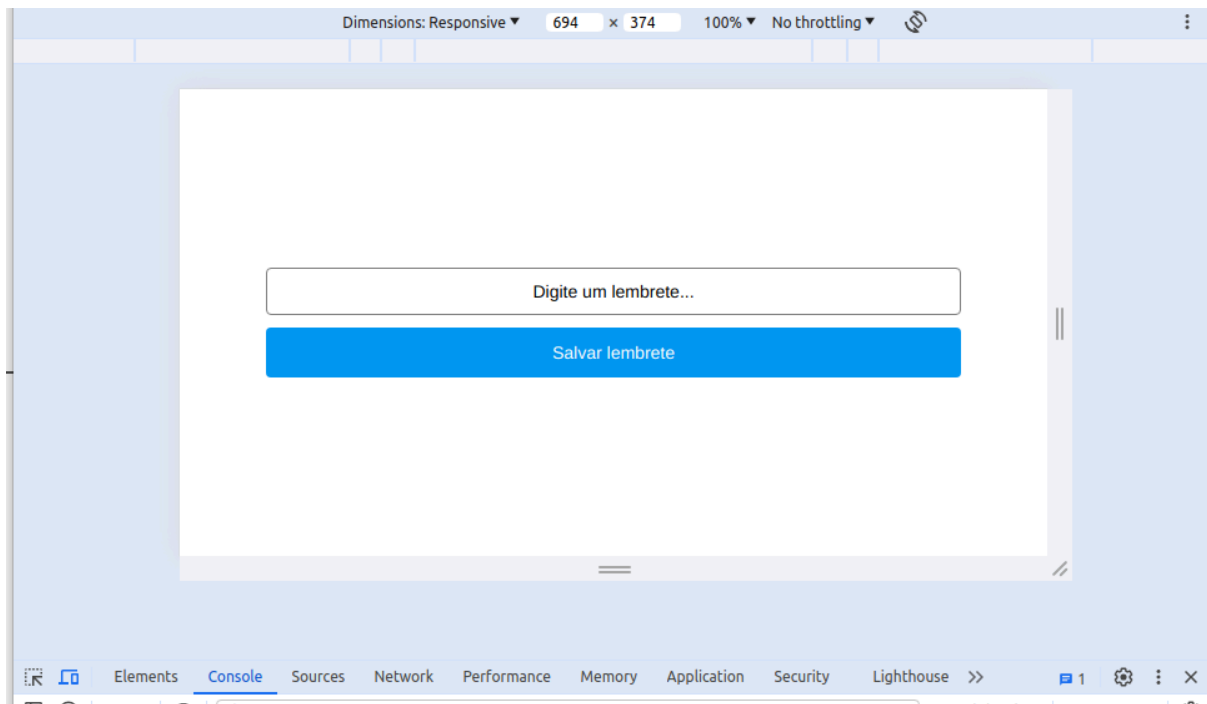
```



**Nota.** A cor que escolhemos para o botão é um azul que faz parte da especificação Material Design do Google. Seu nome é Material Blue 500. Veja mais a seguir.

<https://m3.material.io/>

Veja o resultado.



## Lista para armazenar os lembretes

Precisamos de uma lista para armazenar os lembretes. A sua atualização tem implicações visuais, já que ela é exibida na tela. Ou seja, quando atualizada, a tela deve ser atualizada. Por isso, ela será uma variável de estado. Ela será uma lista de objetos do tipo Lembrete. Como estamos utilizando Typescript, vamos começar definindo o tipo, obtendo benefícios providos por um sistema de tipos estático.

```

...
import {
  useState
} from 'react'

interface Lembrete{
  id: string;
  texto: string;
}

export default function App() {
  const [lembrete, setLembrete] = useState('')
  const [lembretes, setLembretes] = useState<Lembrete[]>([])
  return (
    <View style={styles.container}>
  ...

```

## Função para adição de lembretes

A seguir, podemos especificar uma função que será executada quando o botão for clicado. Ela fará o seguinte:

- construir um objeto Lembrete com id igual à data atual do sistema e texto igual ao valor existente na variável de estado chamada lembrete
- utilizar o hook associado à lista de lembretes a fim de adicionar o lembrete à lista, garantindo a atualização da tela
- atualizar a variável lembrete armazenando a cadeia vazia, limpando portando o campo em que o usuário digita seus lembretes

**Nota.** A função `setLembretes` recebe uma função que se encarrega de fazer a atualização, ao invés de simplesmente a coleção a ser atribuída. Isso acontece pois

- O React pode agrupar diversas chamadas à função `setState`, a fim de aprimorar o desempenho da aplicação e isso pode causar o uso de um estado desatualizado, caso a atualização de interesse estiver sendo feita em função de um estado anterior.

Quando passamos uma função como parâmetro, o React garante que seu argumento é o estado mais recente.

Observe a implementação da função de adição de lembretes e seu vínculo ao botão.

```
...
export default function App() {
  const [lembrete, setLembrete] = useState('')
  const [lembretes, setLembretes] = useState<Lembrete[]>([])

  const adicionar = () => {
    const novoLembrete: Lembrete =
      {id: Date.now().toString(), texto: lembrete}
    setLembretes(lembretesAtual => [
      novoLembrete, //primeiro o novo lembrete
      ...lembretesAtual //extrai todos os
      lembretes já existentes com o operador spread
    ])
    //limpa o campo em que o usuário digita o lembrete
    setLembrete('')
  }

  return (
    <View style={styles.container}>
      <TextInput
        style={styles.input}
        placeholder='Digite um lembrete...'
        value={lembrete}
        onChangeText={setLembrete}
      />
      <Pressable
        style={styles.button}
        onPress={adicionar}>
        <Text
          style={styles.buttonText}>
            Salvar lembrete
          </Text>
        </Pressable>
      </View>
    );
  }
  ...
}
```

## FlatList para exibição da lista de lembretes

Um componente eficiente para a exibição de uma coleção de itens é um FlatList. Veja mais sobre ele.

<https://reactnative.dev/docs/flatlist>

Para utilizá-lo na exibição de lembretes, vamos começar especificando dois props:

- data: a coleção de itens que desejamos exibir
- renderItem: uma função que, dado um objeto recebido como parâmetro, devolve uma JSX que diz como ele deve ser exibido na lista.

Num primeiro momento, para cada lembrete, exibimos apenas componente Text simples com o seu texto.

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

import {
  ...

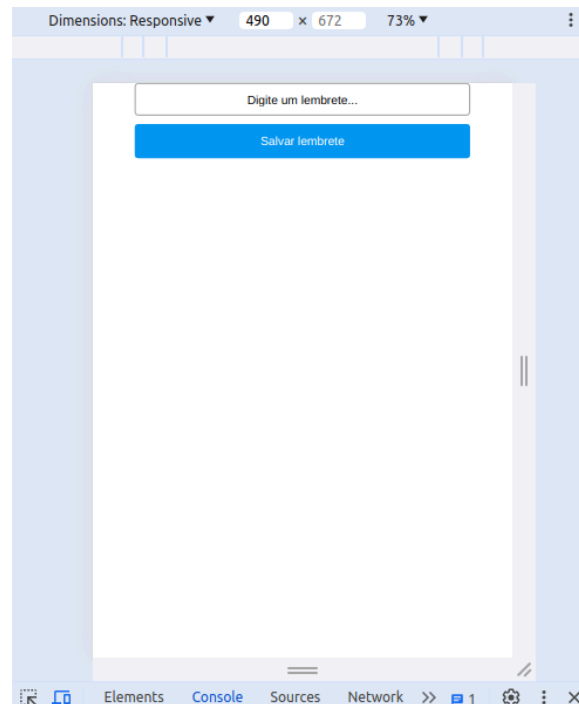
  <Pressable
    style={styles.button}
    onPress={adicionar}>
    <Text
      style={styles.buttonText}>
      Salvar lembrete
    </Text>
  </Pressable>
  <FlatList
    data={lembretes}
    renderItem={
      lembrete => (
        <View>
          <Text>{lembrete.item.texto}</Text>
        </View>
      )
    }
  </FlatList>
}
```

```

    />
  </View>
...

```

Observe que a FlatList toma todo o espaço remanescente e, por essa razão, os componentes de entrada de dados (TextInput e Pressable) “subiram” na tela.



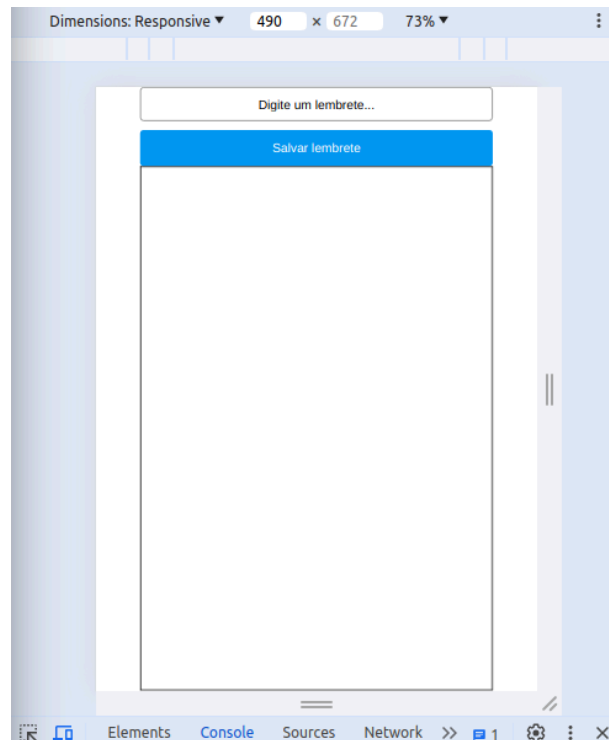
Adicione uma borda à FlatList para verificar o espaço que ela está ocupando.

```

...
<FlatList
  style={{borderWidth: 1, borderColor: 'black', width: '80%'}}
  data={lembretes}
  renderItem={
    lembrete => (
      <View>
        <Text>{lembrete.item.texto}</Text>
      </View>
    )
  }
...

```

Observe.



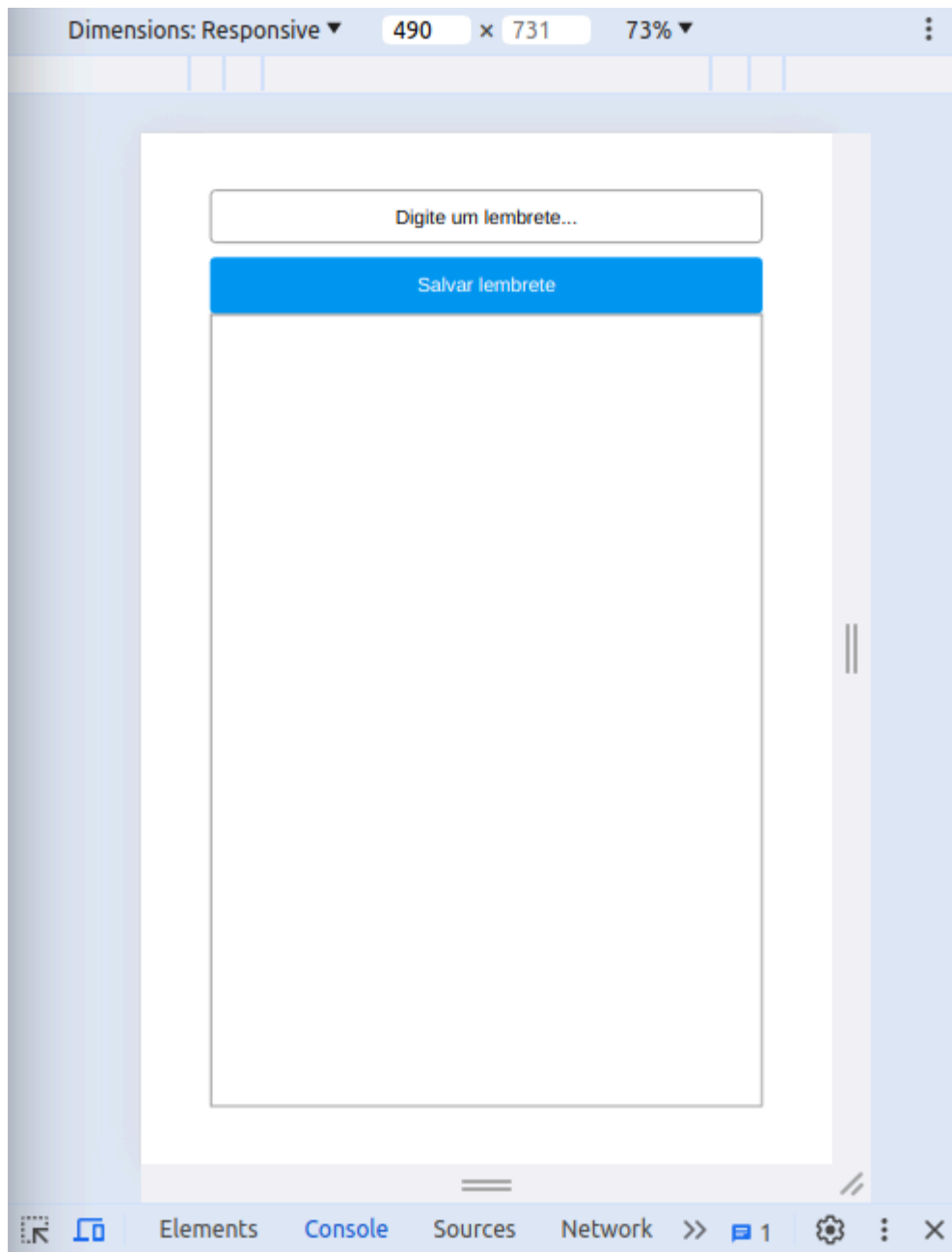
Se desejar, remova completamente as bordas da lista. Neste material vamos optar por mantê-la. Porém, os estilos da lista passarão a ser feitos por meio de um objeto isolado, como os demais.

```
...
<FlatList
  style={styles.list}
  data={lembretes}
...
const styles = StyleSheet.create({
  container: {
    ...
  },
  input: {
    ...
  },
  button: {
    ...
  },
  buttonText: {
    ...
  },
  list: {
    width: '80%',
    borderColor: 'gray',
    borderWidth: 1,
  },
});
```

Também vamos **adicionar um pouco de padding ao contêiner principal**, a fim de descolar os conteúdos das bordas. Podemos também **remover a centralização vertical do contêiner principal**, dado que o conteúdo está todo ocupado. No React Native, as Views têm display flex por padrão. Além disso, o eixo principal é o vertical. Assim, **a propriedade responsável por centralizar verticalmente é a justify-content**. Vamos removê-la.

```
...  
container: {  
  flex: 1,  
  backgroundColor: '#fff',  
  alignItems: 'center',  
  paddingVertical: 40  
},  
...
```

Resultado.

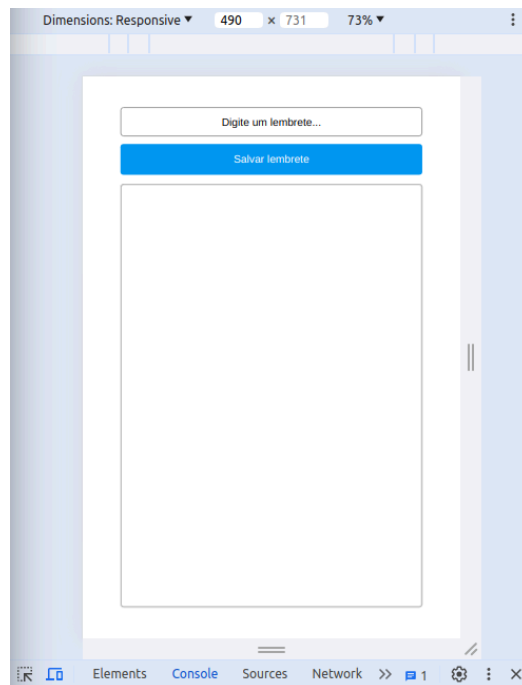


A list ainda está colada no botão. Vamos colocar um pouco de **margem superior** nela para descolar. Talvez queiramos **arredondar as bordas também**. Escolha de acordo com o seu gosto.

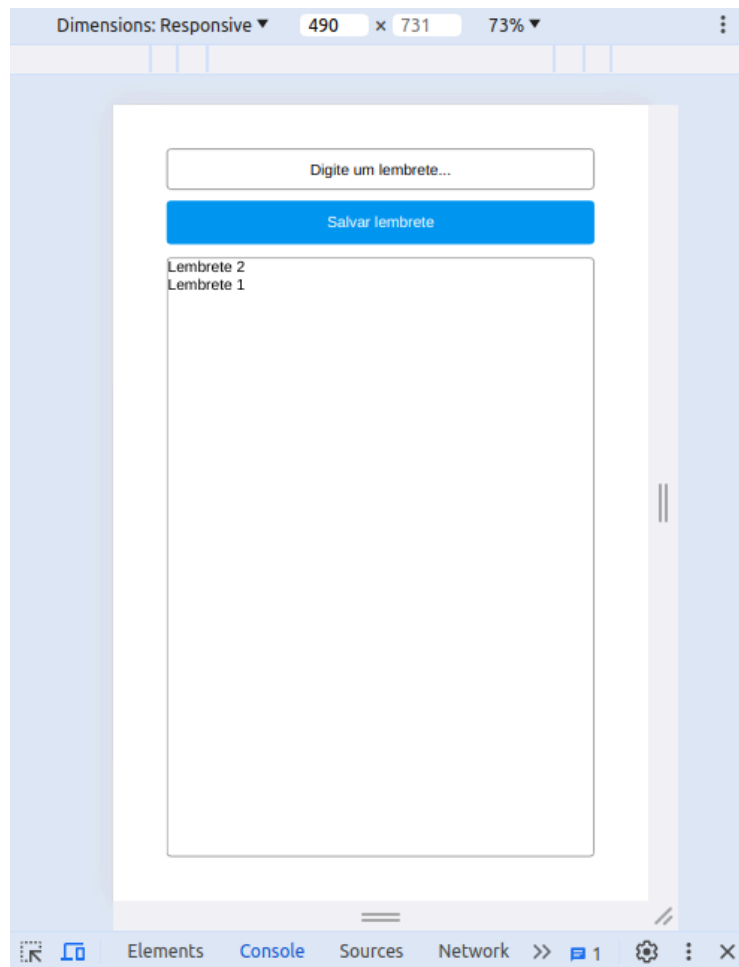


```
...  
list: {  
  marginTop: 12,  
  width: '80%',  
  borderColor: 'gray',  
  borderWidth: 1,  
  borderRadius: 4,  
}  
...
```

Veja o resultado.



Adicione alguns lembretes para ver o resultado.



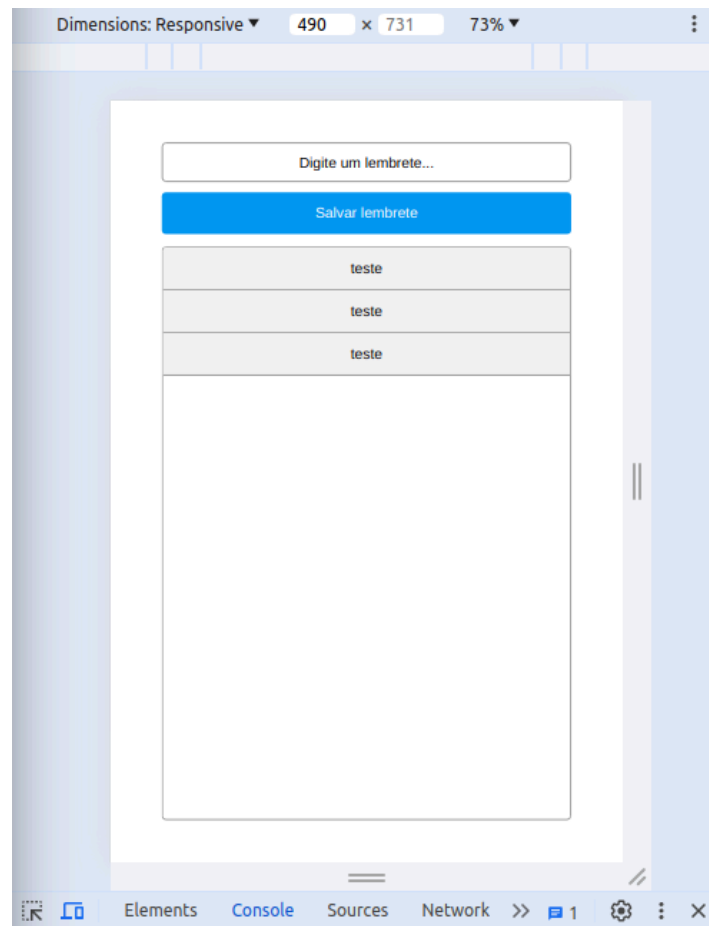
Eles são apresentados como texto puro, afinal, estamos utilizando um componente Text. Vamos melhorar um pouco o seu visual.

```

...
<FlatList
  style={styles.list}
  data={lembretes}
  renderItem={
    lembrete => (
      <View>
        <Text
          style={styles.listItem}>
            {lembrete.item.texto}
        </Text>
      </View>
    )
  }
/>
...
,
list: {
  marginTop: 12,
  width: '80%',
  borderColor: 'gray',
  borderWidth: 1,
  borderRadius: 4,
},
listItem: {
  padding: 12,
  borderBottomWidth: 1,
  borderBottomColor: 'gray',
  backgroundColor: '#f0f0f0',
  textAlign: 'center',
}
});

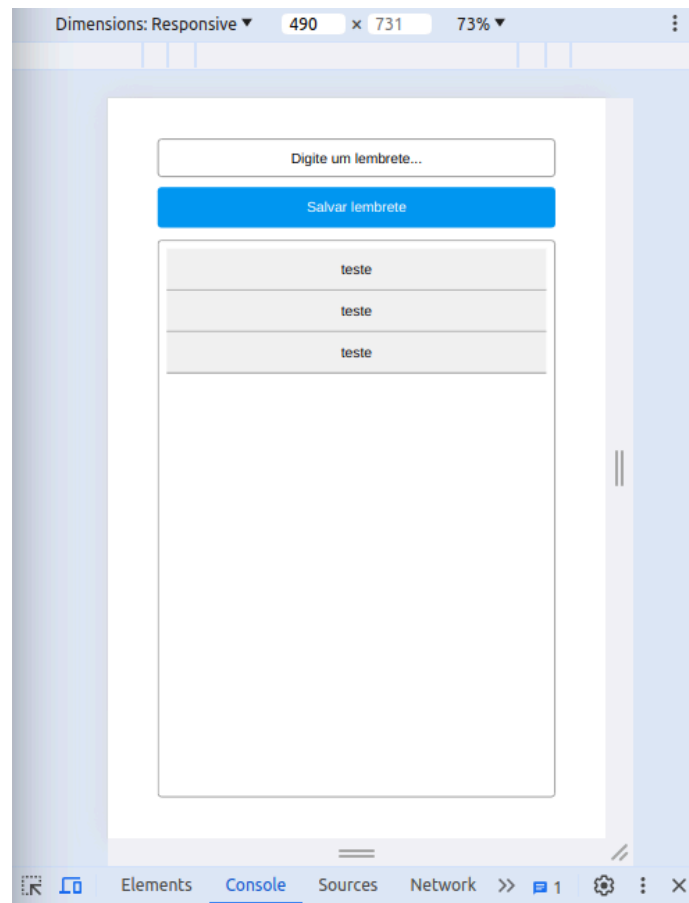
```

Veja como já melhorou.



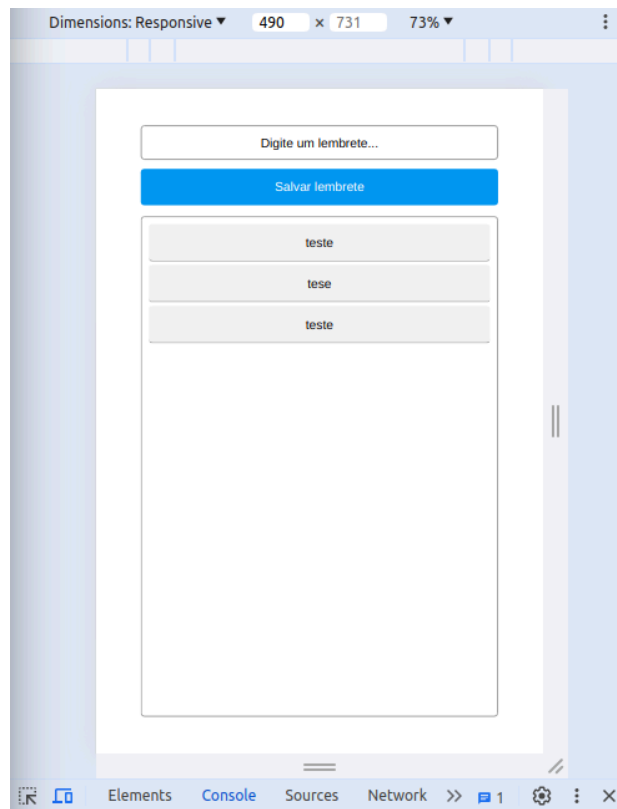
Apesar disso, os lembretes estão colados nas bordas da lista. Vamos adicionar padding a ela para resolver.

```
list: {  
  marginTop: 12,  
  width: '80%',  
  borderColor: 'gray',  
  borderWidth: 1,  
  borderRadius: 4,  
  padding: 8  
},
```



Como os lembretes estão colados entre si, vamos adicionar um pouco de margem inferior a cada um deles. Também podemos arredondar as suas bordas.

```
listItem: {  
  padding: 12,  
  borderBottomWidth: 1,  
  borderBottomColor: 'gray',  
  backgroundColor: '#f0f0f0',  
  textAlign: 'center',  
  marginBottom: 4,  
  borderRadius: 4  
}
```



Ainda sobre a FlatList, quando exibimos uma lista de itens, é fundamental que seus itens possuam um identificador. Isso ajuda o React a

- fazer atualizações mais eficientes
- manter a consistência entre atualizações de estado

Podemos associar um id a cada item da lista por meio do props **keyExtractor** da FlatList. Utilizaremos o id que cada lembrete já possui. Quando criado, um lembrete armazena um número textualmente que representa a data (com precisão de milissegundo) em que foi criado. A menos que dois lembretes sejam criados no mesmo milissegundo, esse valor serve como identificador de lembrete. Suficiente para o momento.

```

...
<FlatList
  style={styles.list}
  keyExtractor={(item) => item.id}
  data={lembretes}
  renderItem={
    lembrete => (
      <View>
        <Text
          style={styles.listItem}>
            {lembrete.item.texto}
        </Text>
      </View>
    )
  }
/>
...

```

Visualmente, nada muda. Mas a aplicação está garantidamente operando de maneira correta e eficiente.

## Ícones para remoção e atualização

O Expo oferece ícones de diferentes famílias de ícones famosas. Uma família famosa é provida pelo framework Ionic. Ela se chama ionicons. Veja.

<https://ionic.io/ionicons>

Há também a FontAwesome. Observe.

<https://fontawesome.com/>

Outra família de ícones popular é provida pela especificação AntDesign. Veja.

<https://ant.design/components/icon>

E assim por diante. Utilizando o pacote **@expo/vector-icons** do Expo podemos utilizar ícones dessas famílias. Veja a sua documentação.

<https://www.npmjs.com/package/@expo/vector-icons>

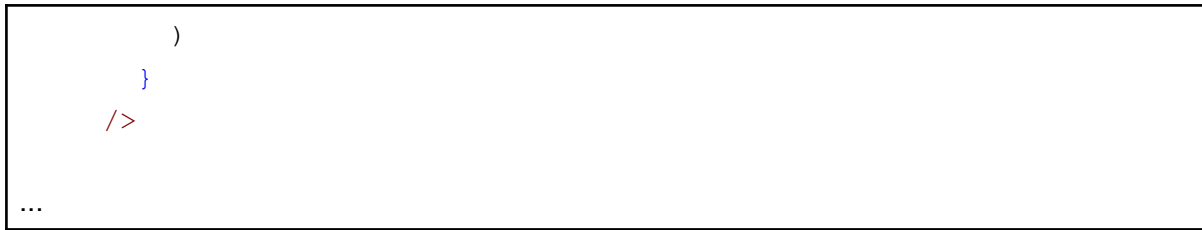
Nesta aplicação, vamos utilizar ícones de AntDesign. Por isso, faça a instrução import necessária.

```
import {
  FlatList,
  Pressable,
  StyleSheet,
  Text,
  TextInput,
  View
} from 'react-native';
import { AntDesign } from '@expo/vector-icons';
...
```

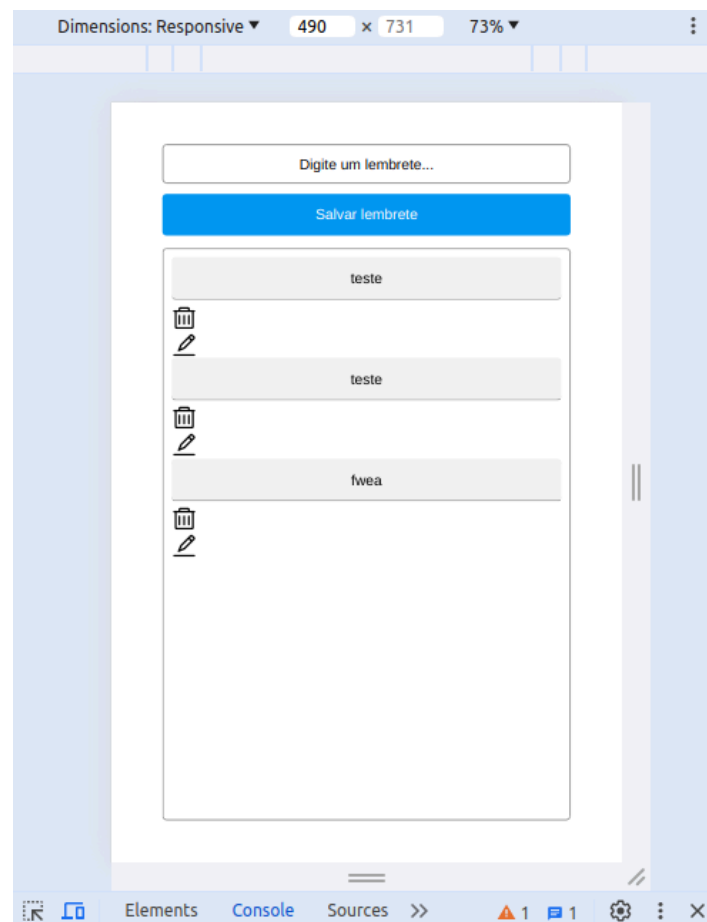
A seguir, criamos um Pressable para cada ação. Cada Pressable engloba o ícone correspondente.

```
...
<FlatList
  style={styles.list}
  keyExtractor={(item) => item.id}
  data={lembretes}
  renderItem={
    lembrete => (
      <View>
        <Text
          style={styles.listItem}>
            {lembrete.item.texto}
        </Text>
        <Pressable>
          <AntDesign
            name="delete"
            size={24} />
        </Pressable>
        <Pressable>
          <AntDesign
            name="edit"
            size={24} />
        </Pressable>
      </View>
    )
  }
/>
```





Veja que os filhos da View raiz de cada item na lista ficam empilhados.



Isso acontece pois, por padrão, o eixo principal de um View React Native é vertical. Vamos trocar para horizontal.

```
...
listItem: {
  padding: 12,
  borderBottomWidth: 1,
  borderBottomColor: 'gray',
  backgroundColor: '#f0f0f0',
  textAlign: 'center',
  marginBottom: 4,
  borderRadius: 4,
  flexDirection: 'row'
}
...
```

Observe, entretanto, que nada acontece. Isso ocorre pois aplicamos a regra ao componente Text, e não ao componente View, que é a raiz de cada item na lista. Por isso, vamos ajustar a aplicação para que ela estilos aplicados à raiz e também ao texto, de maneira separada:

- o objeto listItemText abrigará estilos para o componente Text
- o objeto listItem abrigará estilos para o componente View, que fica na raiz

É uma pequena refatoração do que tínhamos.

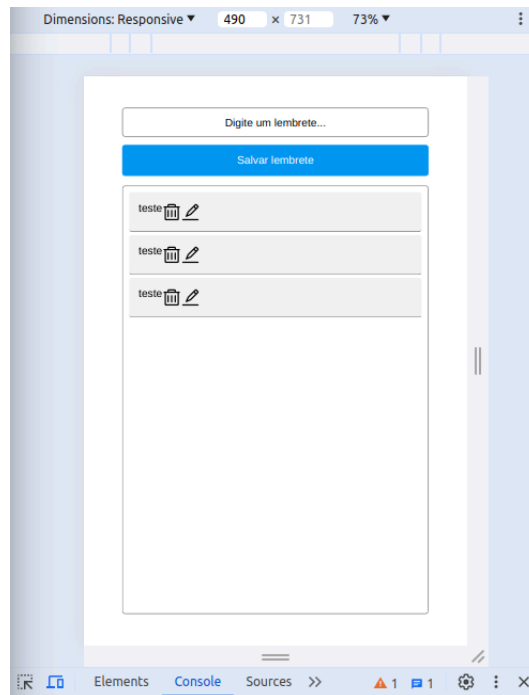
```
...
<FlatList
  style={styles.list}
  keyExtractor={(item) => item.id}
  data={lembretes}
  renderItem={
    lembrete => (
      <View
        style={styles.listItem}>
        <Text
          style={styles.listItemText}>
            {lembrete.item.texto}
        </Text>
        <Pressable>
          <AntDesign
            name="delete"
            size={24} />
        </Pressable>
      </View>
    )
  }
/>
```

```

        <Pressable>
          <AntDesign
            name="edit"
            size={24} />
          </Pressable>
        </View>
      )
    }
  />
...
const styles = StyleSheet.create({
  container: {
    ...
  },
  input: {
    ...
  },
  button: {
    ...
  },
  buttonText: {
    ...
  },
  list: {
    ...
  },
  listItem: {
    padding: 12,
    borderBottomWidth: 1,
    borderBottomColor: 'gray',
    backgroundColor: '#f0f0f0',
    marginBottom: 4,
    borderRadius: 4,
    flexDirection: 'row'
  },
  listItemText: {
    textAlign: 'center',
  }
});

```

Veja o resultado.



Precisamos, claro, ajustar para que o lembrete ocupe uma parte maior e os botões fiquem mais para o canto. Façamos isso:

- O lembrete ocupará apenas 70% da tela
- Os botões serão agrupados por um View, que ocupa 30% da tela
- Dentro do View dos botões, distribuimos o espaço remanescente de maneira uniforme antes, entre e depois deles (evenly)
- Também centralizamos verticalmente os filhos do item na lista (Text e View que agrupa os botões)

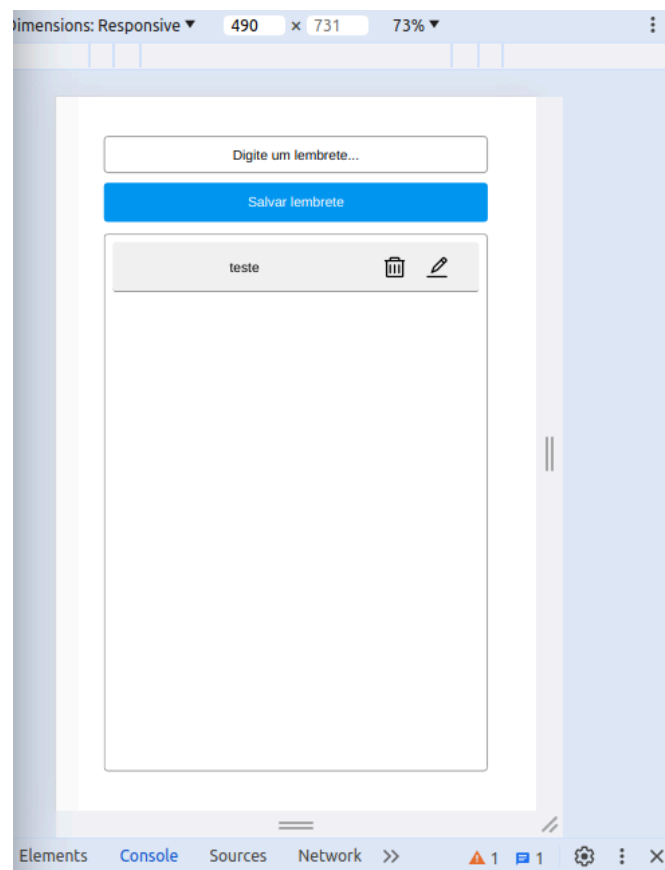
```

...
<FlatList
  style={styles.list}
  keyExtractor={({item}) => item.id}
  data={lembretes}
  renderItem={
    lembrete => (
      <View
        style={styles.listItem}>
        <Text
          style={styles.listItemText}>
            {lembrete.item.texto}
        </Text>
        <View
          style={styles.listItemButtons}>
            <Pressable>
              <AntDesign
                name="delete"
                size={24} />
            </Pressable>
            <Pressable>
              <AntDesign
                name="edit"
                size={24} />
            </Pressable>
          </View>
        </View>
      )
    )
  }
...
,
listItem: {
  padding: 12,
  borderBottomWidth: 1,
  borderBottomColor: 'gray',
  backgroundColor: '#f0f0f0',
  marginBottom: 4,
  borderRadius: 4,
  flexDirection: 'row',
  alignItems: 'center'
},

```

```
listItemText:{  
  width: '70%',  
  textAlign: 'center',  
},  
listItemButtons: {  
  flexDirection: 'row',  
  justifyContent: 'space-evenly',  
  width: '30%',  
}
```

Veja o resultado.



## Removendo um lembrete

Vamos começar escrevendo uma função que

- se chama remover
- recebe o objeto lembrete (do tipo Lembrete) a ser removido
- antes de prosseguir, exibe um Alert para confirmar que o usuário quer, de fato, remover aquele lembrete, exibindo o texto dele para o usuário ver o lembrete que está removendo
- busca por ele na lista (variável de estado lembretes) usando seu id
- remove da lista
- atualiza a variável de estado lembretes

```
...
export default function App() {
  const [lembrete, setLembrete] = useState('')
  const [lembretes, setLembretes] = useState<Lembrete[]>([])

  const adicionar = () => {
    ...
  }

  const remover = (lembrete: Lembrete) => {
    console.log('chamou')
    Alert.alert(
      //título
      'Remover Lembrete',
      //mensagem central
      `Deseja remover este lembrete? ${lembrete.texto}`,
      //coleção de botões
      [
        {
          text: 'Cancelar',
          style: 'cancel'
        },
        {
          text: 'Remover',
          style: 'destructive',
          onPress: () => {
            setLembretes(
              lembretesAtual => lembretesAtual.filter(item => item.id !==
                lembrete.id
              ))
          }
        }
      ]
    )
  }

  return (
    ...
  )
}
```

É importante observar que **o componente Alert não funciona no navegador**. Assim, se você estiver testando a aplicação no navegador (e não num emulador ou dispositivo real), **refaça a implementação do método remover sem a verificação do Alert**.

```
...  
const remover = (lembrete: Lembrete) => {  
  //remove sem alert  
  setLembretes(lembretesAtual => lembretesAtual.filter(item =>  
    item.id !== lembrete.id));  
}  
...
```

**Nota.** É natural que diversos recursos que estamos utilizando não funcionem no navegador, afinal, o alvo principal do React Native não são os navegadores e sim os dispositivos móveis. Os testes no navegador são apenas preliminares, ajudam no começo. Conforme avança em seus estudos, o melhor é adotar o uso de um emulador ou dispositivo real. Entretanto, há bibliotecas para React Native que viabilizam o uso de construções semelhantes aos Alerts. Veja essa, por exemplo:

<https://www.npmjs.com/package/react-native-awesome-alerts>

Um bom exercício é tentar utilizá-la. E você sempre pode buscar por mais conteúdos no **npm registry**.

<https://www.npmjs.com/>

Busque por react-native e verifique os diversos pacotes que podem ser de interesse.

Faça um teste adicionando alguns lembretes e clicando na latinha de lixo de alguns deles.



## Atualizando um lembrete

A atualização de um lembrete acontecerá da seguinte forma

- quando o ícone de edição de um lembrete for clicado, seu texto será copiado para o mesmo campo textual usado para a adição. Neste momento, a aplicação entra em “modo de edição”, o que é representado por uma variável de estado booleana.

- quando a aplicação estiver em modo de edição, o texto exibido no botão será alterado para “Atualizar lembrete”

- quando o botão for clicado, o lembrete será atualizado na lista e a aplicação sai do modo de edição. Neste momento, o texto do botão volta ao padrão.

- decidimos qual função chamar (adicionar, que já existe, ou atualizar, que vamos criar agora) em função do modo da aplicação (em edição ou não)

Comece declarando uma variável de estado que responde se a aplicação está em modo de edição ou não.

```
...
export default function App() {
  const [lembrete, setLembrete] = useState('')
  const [lembretes, setLembretes] = useState<Lembrete[]>([])
  const [emModoDeEdicao, setEmModoDeEdicao] = useState(false)
  ...
}
```

A seguir, escreva a função atualizar, ainda vazia.

```
...
const remover = (lembrete: Lembrete) => {
  //remove sem alert
  setLembretes(lembretesAtual => lembretesAtual.filter(item =>
item.id !== lembrete.id));
}

const atualizar = () => {

}

return (
...
)
```

No botão que adiciona, decida qual texto exibir e qual função associar em função do modo da aplicação (em edição ou não).

```
return (
  <View style={styles.container}>
    <TextInput
      style={styles.input}
      placeholder='Digite um lembrete...'
      value={lembrete}
      onChangeText={setLembrete}
    />
    <Pressable
      style={styles.button}
      onPress={emModoDeEdicao ? atualizar : adicionar}>
      <Text
        style={styles.buttonText}>
        {
          emModoDeEdicao ? 'Atualizar lembrete' : 'Salvar lembrete'
        }
      </Text>
    </Pressable>
  </FlatList>
  ...
)
```

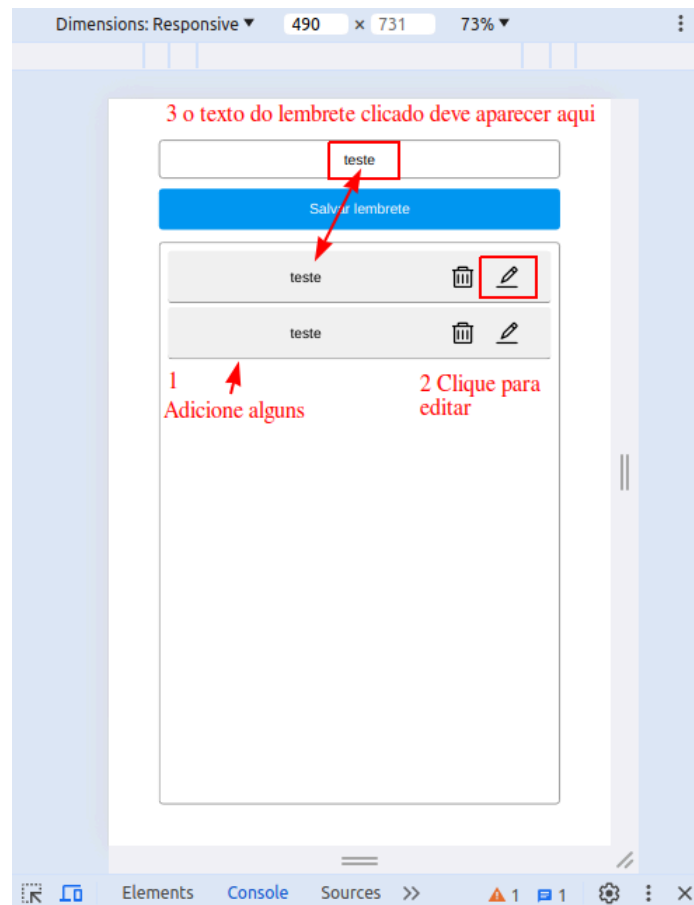
Ao botão de edição do lembrete, vincule uma função que atualiza o texto do lembrete (variável de estado). Isso fará com que o campo textual de Adição/Atualização de lembretes seja atualizado.

```

...
<FlatList
  style={styles.list}
  keyExtractor={(item) => item.id}
  data={lembretes}
  renderItem={
    lembrete => (
      <View
        style={styles.listItem}>
        <Text
          style={styles.listItemText}>
            {lembrete.item.texto}
        </Text>
        <View
          style={styles.listItemButtons}>
            <Pressable
              onPress={() => remover(lembrete.item)}>
                <AntDesign
                  name="delete"
                  size={24}/>
            </Pressable>
            <Pressable
              onPress={() => setLembrete(lembrete.item.texto)}>
                <AntDesign
                  name="edit"
                  size={24} />
            </Pressable>
          </View>
        </View>
      )
    )
  }
/>
...

```

Faça um teste rápido para ver se o texto de um lembrete aparece quando seu ícone de edição é clicado.



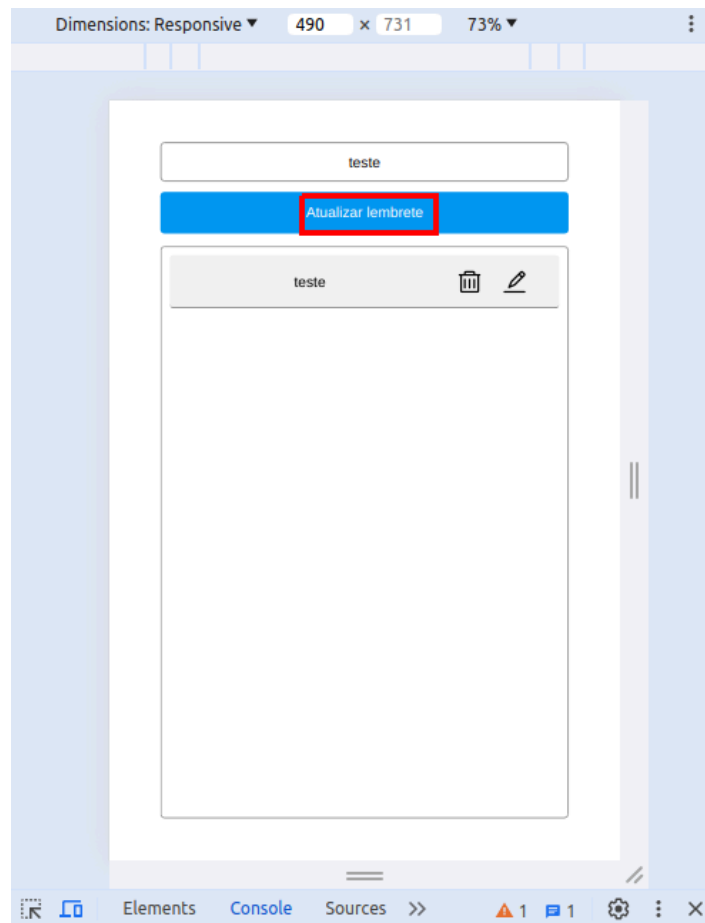
Ajuste a função para que ela coloque a aplicação em modo de edição.

```

...
<FlatList
  style={styles.list}
  keyExtractor={(item) => item.id}
  data={lembretes}
  renderItem={
    lembrete => (
      <View
        style={styles.listItem}>
        <Text
          style={styles.listItemText}>
            {lembrete.item.texto}
        </Text>
        <View
          style={styles.listItemButtons}>
            <Pressable
              onPress={() => remover(lembrete.item)}>
                <AntDesign
                  name="delete"
                  size={24}/>
              </Pressable>
              <Pressable
                onPress={() => {
                  setLembrete(lembrete.item.texto)
                  setEmModoDeEdicao(true)
                }}
              </Pressable>
            <AntDesign
              name="edit"
              size={24} />
            </Pressable>
          </View>
        </View>
      </View>
    )
  }
/>
...

```

Veja o que deve ocorrer quando clicarmos no ícone de edição de um lembrete.



Agora vamos ter de resolver o seguinte problema: para que a edição possa ocorrer, precisamos encontrar o lembrete na lista, o que somente pode ser feito caso tenhamos o seu id. Há algumas soluções possíveis:

- podemos declarar uma variável de estado para armazenar o id do lembrete cujo ícone de edição for clicado
- podemos refatorar a aplicação, de modo que o lembrete armazenado na variável de estado lembrete não mais seja apenas o texto, mas sim um objeto do tipo Lembrete composto por id e texto.

Vamos optar pela segunda opção.

Comece alterando a definição da variável de estado. Agora ela é do tipo Lembrete. Seu valor inicial é um objeto Lembrete válido. Seu texto é a cadeia vazia (para manter o campo textual limpo) e ele não tem id (afinal, o usuário não criou nada ainda).

```
...
export default function App() {
  const [lembrete, setLembrete] = useState<Lembrete>({texto: ''})
  ...
}
```

Essa linha não funciona, pois o id ainda é obrigatório.

```
5
6
7 interface Lembrete{
8   id: string;
9   texto: string;
10 }
11
12 export default function App() {
13   const [lembrete, setLembrete] = useState<Lembrete>({texto: ''})
14 }
```

Vamos ajustar, tornando o id um campo opcional.

```
...
interface Lembrete{
  id?: string;
  texto: string;
}
...
```

Há outros detalhes para arrumar. Na própria função adicionar, quando limpamos o campo lembrete, atribuímos uma cadeia vazia. Agora é necessário atribuir um objeto válido com campo texto igual à cadeia vazia.

```
...
const adicionar = () => {
  const novoLembrete: Lembrete = {id: Date.now().toString(), texto: lembrete.texto}
  console.log(novoLembrete)
  setLembretes(lembreresAtual => [
    novoLembrete, //primeiro o novo lembrete
    ...lembreresAtual //extraí todos os lembretes já existentes com o operador spread
  ])
}
//limpa o campo em que o usuário digita o lembrete
setLembrete({texto: ''})
}
...
```

O componente TextInput também não funciona. Agora precisamos utilizar a propriedade texto do objeto lembrete.

```
...
return (
  <View style={styles.container}>
    <TextInput
      style={styles.input}
      placeholder='Digite um lembrete...'
      value={lembrete.texto}
      onChangeText={ (novoTexto) => setLembrete({texto: novoTexto}) }
    />
    <Pressable
      style={styles.button}
      onPress={emModoDeEdicao ? atualizar : adicionar}>
  ...

```

A FlatList quebra em duas partes: no keyExtractor e na atribuição de novo lembrete quando o ícone de edição é clicado (causador de toda essa refatoração!). Vamos ajustar.



```

...
<FlatList
  style={styles.list}
  //embora possa ser null, usamos o operador ! para indicar ao
  //compilador que sabemos que, neste ponto, ele não é null
  keyExtractor={({item}) => item.id!}

  data={lembretes}
  renderItem={
    lembrete => (
      <View
        style={styles.listItem}>
        <Text
          style={styles.listItemText}>
            {lembrete.item.texto}
        </Text>
        <View
          style={styles.listItemButtons}>
            <Pressable
              onPress={() => remover(lembrete.item)}>
                <AntDesign
                  name="delete"
                  size={24}/>
              </Pressable>
              <Pressable
                onPress={() => {
                  setLembrete({id: lembrete.item.id, texto:
lembrete.item.texto})
                  setEmModoDeEdicao(true)
                }}
              >
                <AntDesign
                  name="edit"
                  size={24} />
              </Pressable>
            </View>
          </View>
        </View>
      )
    )
  }
/>
...

```

Faça novos testes adicionando e removendo. Essa parte já deve funcionar.

Agora vamos implementar a função atualizar. Observe os comentários no código.

```
...
const atualizar = () => {
  //para cada lembrete, verifica se o id é igual ao id do lembrete
  //em edição
  //se for, retorna o lembrete em edição, senão, retorna o lembrete
  //original
  const lembretesAtualizados = lembretes.map(item => {
    if(item.id === lembrete.id){
      return lembrete
    }
    return item
  })
  //atualiza a lista de lembretes
  setLembretes(lembretesAtualizados)
  //aplicação em modo de adição
  setEmModoDeEdicao(false)
  //limpa o campo em que o usuário digita o lembrete
  setLembrete({texto: ''})
}
...
```

A atualização ainda não funciona 100%, pela seguinte razão:

- quando adicionamos um lembrete, ele entra na lista com id e texto
- quando clicamos para editar, o lembrete (variável de estado) se torna aquele objeto clicado, com id e texto
- até aí tudo bem
- quando digitamos algo novo, o objeto lembrete (variável de estado que continha o objeto com id e texto a ser atualizado) é atualizada com um novo objeto, contendo apenas o texto novo, sem id
- quando tentamos atualizar, o lembrete (variável de estado) está sem id
- a atualização não acontece

Felizmente é fácil arrumar. A cada vez que o usuário digita, atualizamos o objeto lembrete (variável de estado), certo? Além de especificar o texto, também podemos especificar o id. Qual será o valor do id? O mesmo que já existe no próprio objeto lembrete, pois se ele ainda não existe, a aplicação está em modo de adição e ele continua não existindo. Sem problema. E se ele já existe, a aplicação está em modo de atualização e, a cada nova digitação por parte do usuário, o id é mantido. Observe.

```
...  
return (  
  <View style={styles.container}>  
    <TextInput  
      style={styles.input}  
      placeholder='Digite um lembrete...'  
      value={lembrete.texto}  
      onChangeText={  
        (novoTexto) => setLembrete({id: lembrete.id, texto:  
novoTexto}) }/>  
  </View>  
)  
...  

```

Neste ponto, é de se esperar que as operações CRUD estejam todas funcionando!

### ***Referências***

**React Native.** 2024. Disponível em <<https://reactnative.dev/>>. Acesso em 2024.