

# Desenvolvimento de Aplicativo Mobile

Material base desenvolvido pelo  
Prof. Raphael Barreto

---

[raphael.b.oliveira@docente.senai.br](mailto:raphael.b.oliveira@docente.senai.br)

**2026**

SENAI - SAPUCAÍ

# Renderização de Listas em React Native

Firjan SENAI - 2026

DESENVOLVIMENTO MOBILE



# Sumário da Aula

## 1 Introdução às Listas

Diferenças entre React e React Native para renderização de listas

## 2 FlatList Component

Propriedades essenciais: data, renderItem e keyExtractor

## 3 Componentes Customizados

Criação do componente Task e integração com Expo Icons

## 4 Boas Práticas

ScrollView vs FlatList e otimização de performance

## 5 Implementação Prática

Código completo e funcional da aplicação de tarefas

CAPÍTULO 01

# Introdução: Renderização de Listas

Renderização de Listas em React Native



## Minhas Tarefas

Adicione um item



Fazer café



Estudar React Native



Academia

## Por Que Listas São Diferentes no React Native?

No React Native, a renderização de listas possui componentes específicos que oferecem melhor performance e funcionalidades otimizadas para dispositivos móveis. Ao contrário do React web, onde usamos apenas o método map(), o React Native fornece componentes dedicados como FlatList e SectionList.

Esses componentes foram projetados especificamente para lidar com grandes volumes de dados em telas pequenas, oferecendo recursos como virtualização automática, eventos de scroll personalizados e gerenciamento eficiente de memória.

# Comparação: React vs React Native

## React (Web)

Utiliza o método `map()` diretamente no JSX para iterar sobre arrays e renderizar elementos. Funciona bem para listas pequenas, mas pode ter problemas de performance com grandes volumes de dados.

```
tasks.map(item => (
  <Text key={item.id}>
    {item.text}
  </Text>
))
```

## React Native (Mobile)

Utiliza componentes especializados como `FlatList` que oferecem virtualização, melhor performance e eventos específicos para dispositivos móveis. Ideal para qualquer tamanho de lista.

```
<FlatList
  data={tasks}
  keyExtractor={item => item.id}
  renderItem={({item}) => (
    <Task text={item.text} />
  )}
/>
```

Acesse: [FlatList · React Native](#)

# Tipos de Componentes de Lista

## FlatList

Componente padrão para renderizar listas simples e homogêneas. Oferece virtualização automática e é o mais utilizado para a maioria dos casos.

- Performance otimizada
- Virtualização automática
- Eventos de scroll

## SectionList

Extensão da FlatList para listas com sessões categorizadas. Perfeito quando seus dados possuem agrupamentos lógicos.

- Cabeçalhos de sessão
- Dados agrupados
- Navegação por categoria

CAPÍTULO 02

# FlatList: O Componente Essencial

Renderização de Listas em React Native

# Estrutura Básica de Dados

## Preparando os Dados

Antes de renderizar uma lista, precisamos estruturar nossos dados adequadamente. No exemplo da aplicação de tarefas, criamos um array de objetos onde cada tarefa possui propriedades específicas.

A estrutura inclui um identificador único (id), um indicador de conclusão (completed) e o texto descritivo da tarefa. Essa organização facilita a manipulação e renderização dos dados.

```
export default function RootLayout() {  
  const tasks = [  
    { id: 1, completed: true, text: "Fazer café" },  
    { id: 2, completed: false, text: "Estudar React Native" },  
    { id: 3, completed: false, text: "Academia" }  
  ]  
  ...  
}
```



# Propriedades Fundamentais da FlatList

A FlatList possui três propriedades essenciais que você precisa dominar para renderizar listas corretamente. Essas propriedades formam a base de qualquer implementação de lista no React Native.



## 1 data

Recebe o array de dados que será renderizado. Deve ser sempre um array de objetos.

## 2 renderItem

Função que define como cada item será renderizado na tela. Recebe o item atual como parâmetro.

## 3 keyExtractor

Extrai uma chave única para cada item. Opcional se houver um campo 'id' nos dados.

# Propriedade data: Fornecendo os Dados

A propriedade **data** é onde você passa o array de objetos que deseja renderizar. É o ponto de partida para qualquer `FlatList`.

O React Native espera que essa propriedade contenha um array válido. Se estiver vazio, a lista simplesmente não renderizará nenhum item, mas também não causará erro.

- **Importante:** A propriedade `data` deve sempre receber um array, mesmo que vazio. Nunca passe `null` ou `undefined`.

```
<FlatList  
  data={tasks}  
  renderItem={({item}) => (<Text>{item.text}</Text>) }  
/>
```

Neste exemplo, estamos passando nosso array 'tasks' para a propriedade `data`. A `FlatList` automaticamente iterará sobre cada elemento.

# Propriedade renderItem: Customizando a Aparência

```
<FlatList  
  data={tasks}  
  renderItem={({item}) => (  
    <View>  
      <Text>{item.text}</Text>  
    </View>  
)}  
>
```

## Como Funciona

A função `renderItem` é chamada para cada elemento do array. Ela recebe um objeto com várias propriedades, sendo a principal delas o `item`, que contém os dados do elemento atual.

Você pode retornar qualquer componente React Native válido: `Text`, `View`, `Image`, ou até componentes customizados complexos. A flexibilidade é total.

Além de 'item', o objeto também fornece 'index' (posição do item) e 'separators' (métodos para controlar separadores).

# Propriedade keyExtractor: Identificação Única

Assim como no React web, cada item de uma lista precisa de uma chave única para otimização de renderização. No React Native, isso é ainda mais crítico devido às limitações de performance mobile.

## Comportamento Padrão

Se seus objetos possuem uma propriedade chamada `id`, a `FlatList` automaticamente a utiliza como chave. Nesse caso, você não precisa especificar o `keyExtractor`.

```
// ID automático
const tasks = [
  { id: 1, text: "Tarefa" }
]
```

## KeyExtractor Customizado

Se sua chave única tem outro nome ou precisa ser calculada, use o `keyExtractor` para especificar como extraí-la.

```
const tasks = [
  { completed: true, text: "Fazer café" },
  { completed: false, text: "Estudar React Native" },
  { completed: false, text: "Academia" }
];

<FlatList
  data={tasks}
  keyExtractor={({item}) => item.taskId}
  renderItem={({item}) => (<Text>{item.text} </Text>)}
/>
```

CAPÍTULO 03

# Criando Componentes Customizados

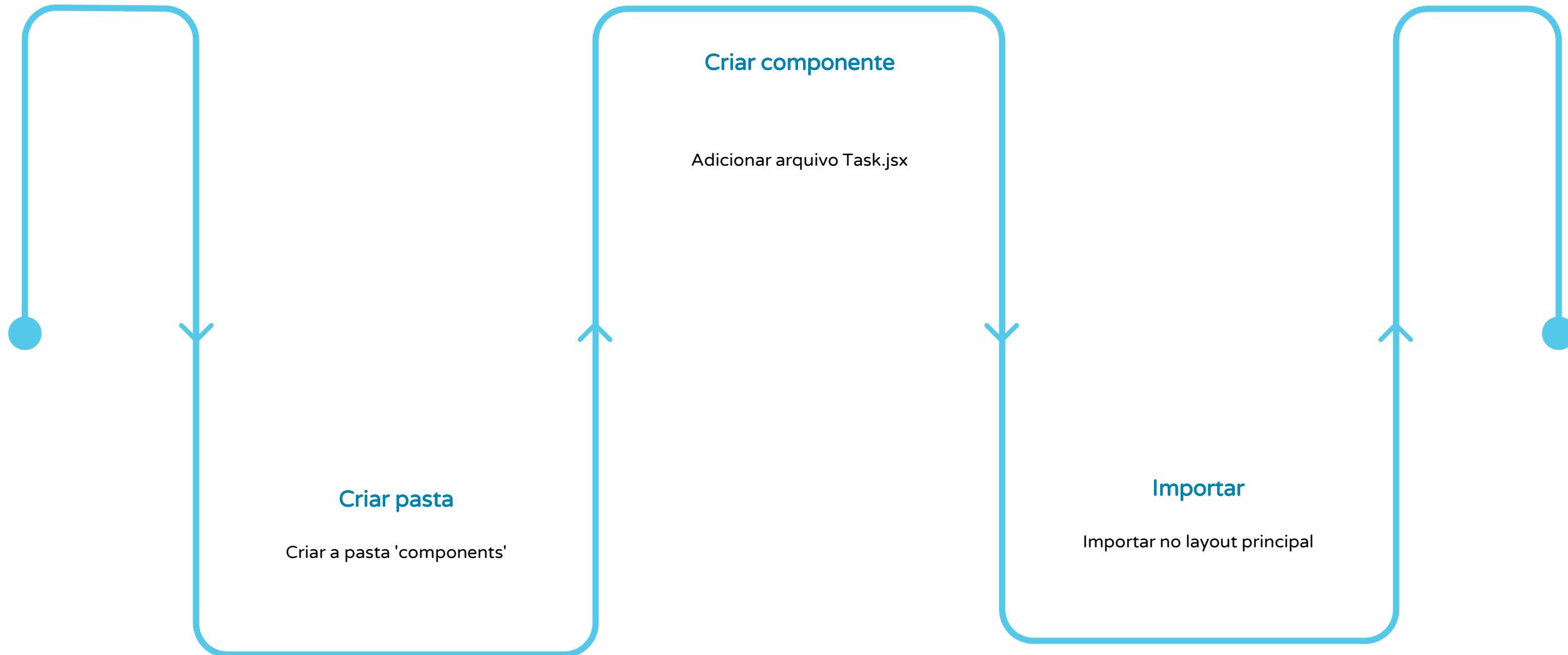
Renderização de Listas em React Native

# Estrutura do Componente Task

Criar componentes separados para os itens da lista é uma prática recomendada que melhora a organização, reutilização e manutenção do código. Vamos construir um componente Task completo.

O componente Task será responsável por renderizar cada tarefa individualmente, incluindo um ícone de check, o texto da tarefa e a interatividade necessária.

# Criando a Estrutura de Pastas



Organizar o projeto com uma estrutura de pastas clara facilita a manutenção e escalabilidade da aplicação. A pasta 'components' centralizará todos os componentes reutilizáveis.

# Código Base do Componente Task

## Estrutura Inicial

Todo componente React Native funcional segue uma estrutura padrão:  
imports, função do componente, JSX de retorno e estilos.

O componente Task receberá props, especificamente o texto da tarefa, e renderizará uma interface visual completa para cada item.

```
import { View, Text, Pressable, StyleSheet } from "react-native"

export default function Task({ text }) {
  return (
    <View style={style.rowContainer}>
      /* Conteúdo aqui */
    </View>
  )
}

const style = StyleSheet.create({
  rowContainer: {
    display: "flex",
    flexDirection: "row",
    alignItems: "center",
    gap: 10,
    marginBottom: 10
  }
})
```

# Integrando Expo Vector Icons

O Expo já vem com uma biblioteca completa de ícones pré-instalada, incluindo conjuntos populares como Ionicons, MaterialIcons, FontAwesome e muitos outros. Não é necessário instalar pacotes adicionais.

## Como Encontrar Ícones

Acesse a documentação do Expo Vector Icons para buscar e visualizar todos os ícones disponíveis.

1. Visite [icons.expo.fyi](https://icons.expo.fyi)
2. Pesquise pelo nome do ícone
3. Copie o código de importação

Acesse: [@expo/vector-icons@15.0.3](https://github.com/expo/expo/tree/main/packages/expo-vector-icons)

## Implementação

```
import { Ionicons } from "@expo/vector-icons"

// Dentro do componente:
<Ionicons
  name="checkmark-circle"
  size={32}
  color={colors.primary}
/>
```

Cada biblioteca de ícones (Ionicons, MaterialIcons, etc.) é importada separadamente e oferece centenas de ícones prontos para uso.

@expo/vector-icons@15.0.3

Q checkmark-circle



**checkmark-circle**

Ionicons



**checkmark-circle-outline**

Ionicons



**checkmark-circle-sharp**

Ionicons

# Adicionando Interatividade com Pressable

O componente Pressable permite criar elementos interativos que respondem ao toque do usuário. É mais flexível que o TouchableOpacity e oferece melhor controle sobre estados de pressionamento.

```
< Pressable >
  <Ionicons
    name="checkmark-
circle"
    size={32}
    color={colors.primary}
  />
</Pressable >
```

## Vantagens do Pressable

- Eventos de toque mais detalhados (onPressIn, onPressOut)
- Suporte a long press nativo
- Feedback visual customizável
- Melhor performance que TouchableOpacity

# Task.jsx Completo

Aqui está o código completo do componente Task, integrando todos os elementos discutidos: estrutura, ícones, interatividade e estilos.

```
import { Ionicons } from "@expo/vector-icons"
import { Pressable, StyleSheet, Text, View } from "react-native"
import { colors } from "../styles/colors"

export default function Task({ text }) {
  return (
    <View style={style.rowContainer}>
      <Pressable>
        <Ionicons
          name="checkmark-circle"
          size={32}
          color={colors.primary}
        />
      </Pressable>
      <Text>{text}</Text>
    </View>
  )
}

const style = StyleSheet.create({
  rowContainer: {
    display: "flex",
    flexDirection: "row",
    alignItems: "center",
    gap: 10,
    marginBottom: 10
  }
})
```

Para chamar task no index, vc deverá atualizar `renderItem={({ item }) => <Task text={item.text} />}`

# index.jsx Completo

```
import { Image, Pressable, ScrollView, StyleSheet, Text, TextInput, View } from 'react-native';
import logo from "../assets/images/check.png";
import { colors } from "../styles/colors";
import { FlatList } from 'react-native';
import Task from '../components/task';

export default function RootLayout() {
  const tasks = [
    { id: 1, completed: true, text: "Fazer café" },
    { id: 2, completed: false, text: "Estudar React Native" },
    { id: 3, completed: false, text: "Academia" }
  ];

  return (
    <View style={estilo.mainContainer}>
      <View style={estilo.rowContainer}>
        <Image source={logo} style={estilo.image} />
        <Text style={estilo.title}>
          Minhas Tarefas
        </Text>
      </View>
      <View style={estilo.rowContainer}>
        <TextInput style={estilo.input} placeholder="Adicione um item" />
        <Pressable
          onPress={() => alert("Olá")}
          style={({ pressed }) => [
            estilo.button,
            {
              backgroundColor: pressed
                ? "blue"
                : colors.primary
            }
          ]}
        >
          <Text style={estilo.buttonText}>+</Text>
        </Pressable>
      </View>
      <FlatList
        data={tasks}
        keyExtractor={(item) => item.id}
        renderItem={({ item }) => <Task text={item.text} />}
      />
    </View>
  );
}
```

```
const estilo = StyleSheet.create({
  image: {
    width: 30,
    height: 30
  },
  title: {
    fontSize: 30,           // font-size vira fontSize
    fontFamily: "Calibri", // font-family vira fontFamily
    fontWeight: 600,        // font-weight vira fontWeight
    color: colors.primary,
    marginLeft: 10
  },
  rowContainer: {
    display: "flex",
    flexDirection: "row",
    alignItems: "center",
    gap: 10,
    marginBottom: 20
  },
  input: {
    height: 40,
    paddingHorizontal: 16,
    borderColor: "gray",
    borderWidth: 1,
    borderRadius: 20,
    flexGrow: 1
  },
  button: {
    width: 40,
    height: 40,
    borderRadius: 20,
    backgroundColor: colors.primary,
    display: "flex",
    alignItems: "center",
    justifyContent: "center"
  },
  buttonText: {
    color: "white",
    fontSize: 24,
    lineHeight: 24,
    textAlign: "center"
  },
  mainContainer: {
    padding: 20
  }
});
```

CAPÍTULO 04

# Boas Práticas e Otimizações

Renderização de Listas em React Native

# ScrollView vs FlatList: Entendendo as Diferenças

## Por Que FlatList é Superior

A FlatList implementa virtualização automática, renderizando apenas os elementos visíveis na tela mais alguns extras para scroll suave. Isso economiza memória e melhora drasticamente a performance.

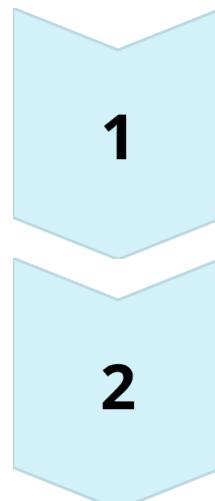
ScrollView renderiza todos os elementos de uma vez, independente de estarem visíveis ou não. Para listas pequenas (até 20 itens) isso é aceitável, mas para listas maiores causa problemas sérios de performance.

# O Problema de Nested ScrollViews



Um erro comum ao trabalhar com FlatList é colocá-la dentro de uma ScrollView. Isso cria um conflito de scroll e gera um warning importante no console.

- ❑ Warning: "VirtualizedLists should never be nested inside plain ScrollViews with the same orientation"



## ✗ Incorreto

ScrollView com FlatList dentro causa conflito de gestos

## ✓ Correto

FlatList em View normal, pois já possui scroll integrado

# Corrigindo o Warning: Solução Prática

## Código Problemático

```
<ScrollView>
  <View style={style.rowContainer}>
    <!-- Header -->
  </View>

  <FlatList
    data={tasks}
    renderItem={...}
  />
</ScrollView>
```

✗ Gera warning e problemas de scroll

## Código Correto

```
<View>
  <View style={style.rowContainer}>
    <!-- Header -->
  </View>

  <FlatList
    data={tasks}
    renderItem={...}
  />
</View>
```

✓ Funciona perfeitamente, sem warnings

OBS: Se replicarmos nossa tasks por várias vezes, veremos que ela irá scrollar automaticamente, devido ser nativo da FlatList

# Propriedades Avançadas da FlatList

Além das propriedades básicas, a FlatList oferece dezenas de recursos adicionais para casos de uso mais complexos. Explorar essas opções permite criar experiências de usuário sofisticadas.



## ListEmptyComponent

Componente exibido quando a lista está vazia. Evita necessidade de renderização condicional externa.



## ItemSeparatorComponent

Adiciona separadores visuais entre os itens automaticamente.



## ListHeaderComponent / ListFooterComponent

Elementos fixos no topo e rodapé da lista que rolam junto com o conteúdo.



## onRefresh / refreshing

Implementa pull-to-refresh nativo para atualizar dados da lista.

CAPÍTULO 05

# Implementação Completa

Renderização de Listas em React Native

# Arquivo \_layout.jsx Completo

Este é o código completo do arquivo principal que gerencia o layout da aplicação, incluindo a FlatList integrada com todos os componentes e funcionalidades.

```
import { Image, Pressable, ScrollView, StyleSheet, Text, TextInput, View } from 'react-native';
import logo from "../assets/images/check.png";
import { colors } from "../styles/colors";
import { FlatList } from 'react-native';
import Task from '../components/task';

export default function RootLayout() {
  const tasks = [
    { id: 1, completed: true, text: "Fazer café" },
    { id: 2, completed: false, text: "Estudar React Native" },
    { id: 3, completed: false, text: "Academia" }
  ];

  return (
    <View style={estilo.mainContainer}>
      <View style={estilo.rowContainer}>
        <Image source={logo} style={estilo.image} />
        <Text style={estilo.title}>
          Minhas Tarefas
        </Text>
      </View>
      <View style={estilo.rowContainer}>
        <TextInput style={estilo.input} placeholder="Adicione um item" />
        <Pressable
          onPress={() => alert("Olá")}
          style={({ pressed }) => [
            estilo.button,
            {
              backgroundColor: pressed
                ? "blue"
                : colors.primary
            }
          ]}
        >
          <Text style={estilo.buttonText}>+</Text>
        </Pressable>
      </View>
      <FlatList
        data={tasks}
        keyExtractor={(item) => item.id}
        renderItem={({ item }) => <Task text={item.text} />}
      />
    </View>
  );
}
```

# Estilos do Layout Principal

## StyleSheet do \_layout.jsx

```
const estilo = StyleSheet.create({
  image: {
    width: 30,
    height: 30
  },
  title: {
    fontSize: 30,
    fontFamily: "Calibri",
    fontWeight: 600,
    marginLeft: 10
  },
  rowContainer: {
    display: "flex",
    flexDirection: "row",
    alignItems: "center",
    gap: 10,
    marginBottom: 20
  },
  input: {
    height: 40,
    paddingHorizontal: 16,
    borderColor: "gray",
    borderWidth: 1,
    borderRadius: 20,
    flexGrow: 1
  },
  button: {
    width: 40,
    height: 40,
    borderRadius: 20,
    backgroundColor: colors.primary,
    display: "flex",
    alignItems: "center",
    justifyContent: "center"
  },
  buttonText: {
    color: "white",
    fontSize: 24,
    lineHeight: 24,
    textAlign: "center"
  },
  mainContainer: {
    padding: 20
  }
});
```

A estilização utiliza o `StyleSheet.create` para otimizar a criação de estilos e garantir melhor performance na renderização.

Os estilos definem o layout responsivo usando Flexbox, garantindo que a interface se adapte a diferentes tamanhos de tela.

# Resumo: Conceitos-Chave da Aula



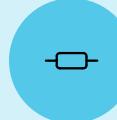
## FlatList é Essencial

Componente especializado para renderização eficiente de listas em React Native



## Três Propriedades Principais

data, renderItem e keyExtractor são fundamentais para qualquer lista



## Componentes Customizados

Criar componentes separados melhora organização e reutilização de código



## Performance Otimizada

FlatList implementa virtualização automática, renderizando apenas itens visíveis



## Evitar ScrollView Aninhada

Nunca colocar FlatList dentro de ScrollView, pois causa conflitos de scroll



## Expo Vector Icons

Biblioteca completa de ícones já integrada, sem necessidade de instalação adicional

# Próximos Passos: Tornando a Lista Funcional

## O Que Vem a Seguir

Atualmente, nossa aplicação renderiza uma lista estática e o botão apenas exibe um alerta. Na próxima aula, vamos implementar a funcionalidade completa da to-do list.

## Funcionalidades a Implementar

- Gerenciamento de estado com useState
- Capturar e armazenar valor do input
- Adicionar novas tarefas dinamicamente
- Marcar tarefas como concluídas
- Remover tarefas da lista

A base de renderização de listas que construímos hoje é fundamental para implementar essas funcionalidades interativas.

# Conclusão: Dominando Listas no React Native

Nesta aula, você aprendeu os fundamentos essenciais para trabalhar com listas no React Native. A FlatList é uma ferramenta poderosa que, quando bem utilizada, proporciona interfaces fluidas e performáticas mesmo com grandes volumes de dados.

Lembre-se sempre das três propriedades fundamentais: **data** para fornecer os dados, **renderItem** para definir a aparência de cada item, e **keyExtractor** quando necessário para identificação única.

A criação de componentes customizados, como nosso Task, demonstra boas práticas de organização de código que facilitarão a manutenção e evolução do projeto.

"A performance de uma aplicação mobile começa com escolhas corretas de componentes. FlatList vs ScrollView pode parecer uma decisão pequena, mas faz toda a diferença na experiência do usuário."

Continue praticando e explorando as propriedades avançadas da FlatList. Nos vemos na próxima aula para tornar nossa aplicação completamente funcional!



# Perguntas?

Entre em contato:

[raphael.b.oliveira@docente.senai.br](mailto:raphael.b.oliveira@docente.senai.br)

Obrigado pela atenção! Estou à disposição para esclarecer dúvidas e ajudar no seu aprendizado.

**Firjan SENAI**

