

# Desenvolvimento de Aplicativo Mobile

Material base desenvolvido pelo  
Prof. Raphael Barreto

---

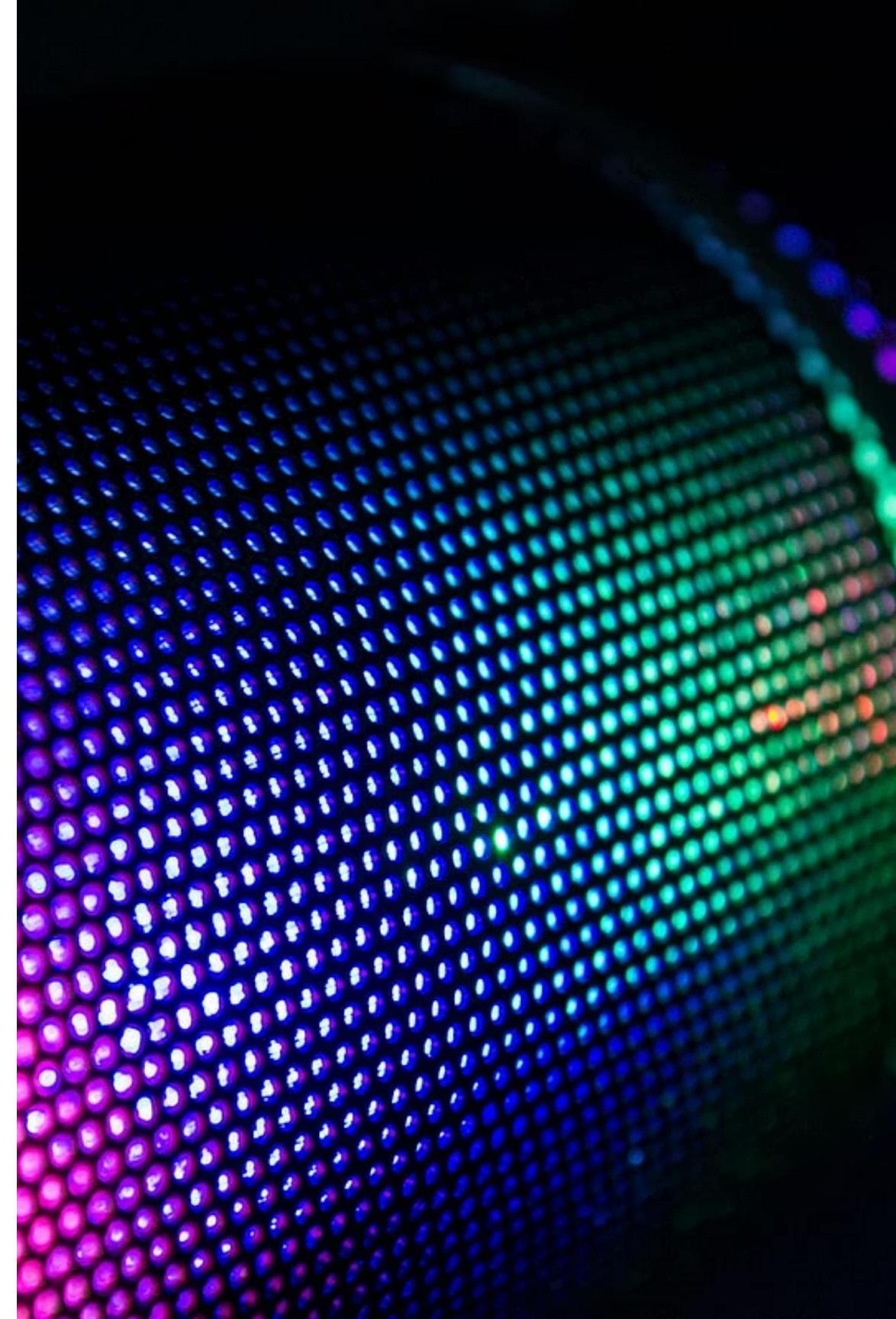
[raphael.b.oliveira@docente.senai.br](mailto:raphael.b.oliveira@docente.senai.br)

# Estilização em React Native

Desenvolvimento Front-End Mobile

FIRJAN SENAI

2026



# Sumário

01

## Introdução à Estilização

Compreendendo StyleSheet e as diferenças do CSS tradicional

03

## Aplicando Estilos aos Componentes

Diferentes formas de aplicar estilos no React Native

05

## Trabalhando com Cores e Constantes

Criando e gerenciando paletas de cores

07

## Estilizando Elementos Interativos

Trabalhando com Pressable e estados visuais

02

## Criando Estilos com StyleSheet

Estrutura básica e sintaxe de estilização

04

## Organizando Estilos no Projeto

Boas práticas e estruturação de arquivos

06

## Layout com Flexbox

Diferenças e particularidades do Flex no mobile

08

## Projeto Prático: Lista de Tarefas

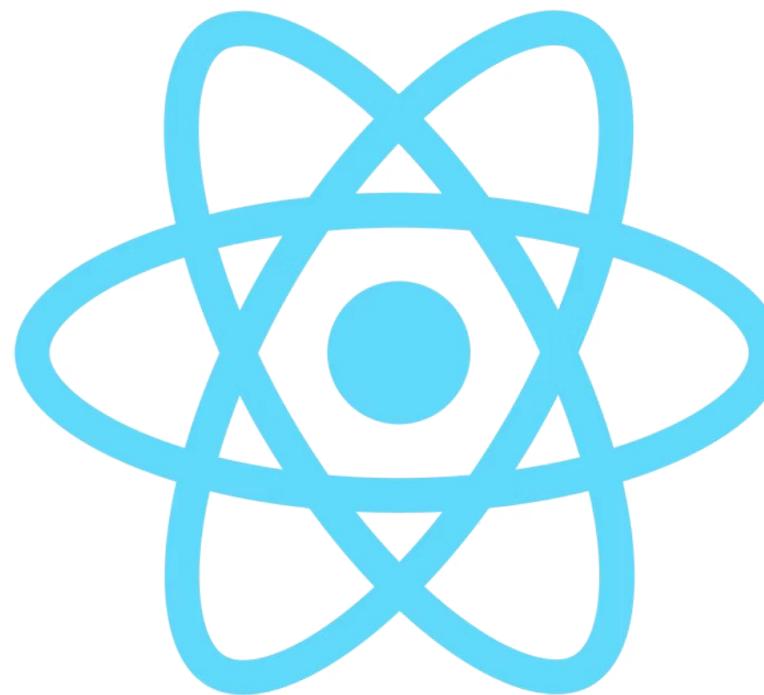
Construindo uma interface completa e estilizada

 CAPÍTULO 01

# Introdução à Estilização em React Native

Compreendendo o conceito de CSS-in-JS e StyleSheet

# CSS no React Native: Uma Nova Abordagem



No React Native, a estilização funciona de maneira diferente do desenvolvimento web tradicional. Não trabalhamos diretamente com HTML e CSS como no React para web. Em vez disso, utilizamos o **StyleSheet**, uma abstração que nos permite criar estilos usando JavaScript.

Esta abordagem é conhecida como **CSS-in-JS** e traz diversas vantagens para o desenvolvimento mobile, incluindo melhor performance, isolamento de estilos e facilidade de manutenção.

# Por Que StyleSheet?

## Performance Otimizada

Os estilos são validados e otimizados em tempo de desenvolvimento, resultando em melhor performance na aplicação final

## Type Safety

Detecção de erros mais eficiente, pois os estilos são objetos JavaScript validados pela ferramenta de desenvolvimento

## Reutilização

Facilita o compartilhamento de estilos entre componentes e a criação de design systems consistentes

# Diferenças Fundamentais

## CSS Tradicional (Web)

```
.image {  
  width: 30px;  
  height: 30px  
}
```

- Sintaxe CSS padrão
- Valores com unidades (px, %, em)
- Kebab-case (font-size)
- Arquivos .css separados

## StyleSheet (React Native)

```
const styles = StyleSheet.create({  
  image: {  
    width: 30,  
    height: 30  
  }  
});
```

- Objetos JavaScript
- Valores numéricos sem unidade
- camelCase (fontSize)
- Definidos no mesmo arquivo JS

🔧 CAPÍTULO 02

# Criando Estilos com StyleSheet

Estrutura básica e primeiros passos práticos

# Importando e Configurando StyleSheet

Para começar a trabalhar com estilos no React Native, primeiro precisamos importar o StyleSheet do pacote react-native. Veja a estrutura inicial do nosso componente:

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

Note que o **StyleSheet** está sendo importado junto com os outros componentes core do React Native. Esta é a primeira etapa essencial para trabalhar com estilização.

# Criando Sua Primeira "Folha de Estilos"

O método `StyleSheet.create()` é usado para definir nossos estilos. Ele recebe um objeto JavaScript onde cada propriedade representa um "estilo" que pode ser aplicado aos componentes.

Vamos começar criando um estilo para controlar o tamanho da nossa imagem:

```
const estilo = StyleSheet.create({
  image: {
    width: 50,
    height: 50
  }
});
```

- ❑ **Importante:** Os valores numéricos são interpretados como pixels por padrão. Para usar porcentagem, utilize strings: "50%"

# Aplicando Estilos aos Componentes

Depois de criar nosso objeto de estilos, precisamos aplicá-lo ao componente usando a propriedade `style`. Veja como fazemos isso com a imagem:

```
<Image source={logo} style={estilo.image} />
```

## Sintaxe de Aplicação

Usamos `style={nomeDoObjeto.nomeDaClasse}` para referenciar o estilo criado

## Múltiplos Estilos

Para aplicar mais de um estilo, use um array:  
`style={[estilo.base, estilo.adicional]}`

# Estilos Inline vs StyleSheet

## Estilo Inline

```
<Image  
  source={logo}  
  style={{  
    width: 30,  
    height: 30  
  }}  
/>
```

- ✓ Funciona perfeitamente
- ✗ Dificulta manutenção
- ✗ Código menos organizado
- ✗ Não é reutilizável

## StyleSheet (Recomendado)

```
const estilo = StyleSheet.create({  
  image: {  
    width: 30,  
    height: 30  
  }  
});  
  
<Image source={logo} style={estilo.image} />
```

- ✓ Código organizado
- ✓ Fácil manutenção
- ✓ Reutilizável
- ✓ Melhor performance

T CAPÍTULO 03

# Estilizando Textos e Tipografia

Trabalhando com fontes, tamanhos e cores no React Native

# Propriedades de Texto

Vamos criar um estilo completo para o título "Minhas Tarefas". No React Native, as propriedades CSS são escritas em **camelCase** em vez de kebab-case:

```
const estilo = StyleSheet.create({
  title: {
    fontSize: 30,          // font-size vira fontSize
    fontFamily: "Calibri", // font-family vira fontFamily
    fontWeight: 600,        // font-weight vira fontWeight
    color: colors.primary,
    marginLeft: 10
  }
});
```



1

CSS Web

font-size, font-family, margin-left

2

React Native

fontSize, fontFamily, marginLeft

# Aplicando o Estilo de Título

Agora aplicamos o estilo ao componente Text que contém nosso título:

```
<Text style={estilo.title}>  
  Minhas Tarefas  
</Text>
```

Com isso, nosso texto agora possui:

- Tamanho de fonte aumentado (30)
- Fonte Calibri
- Peso da fonte mais pesado (600)
- Cor personalizada
- Margem lateral de 10 pixels



## Dica de Produtividade

Use **Ctrl + Espaço** no VS Code dentro das propriedades de estilo para ver todas as opções disponíveis com autocompletar e documentação inline.

Note que **color** do **title** está importando do arquivo **colors.js** que deveremos criar e importar em **\_layout.jsx**

```
import { colors } from "../constants/colors";
```



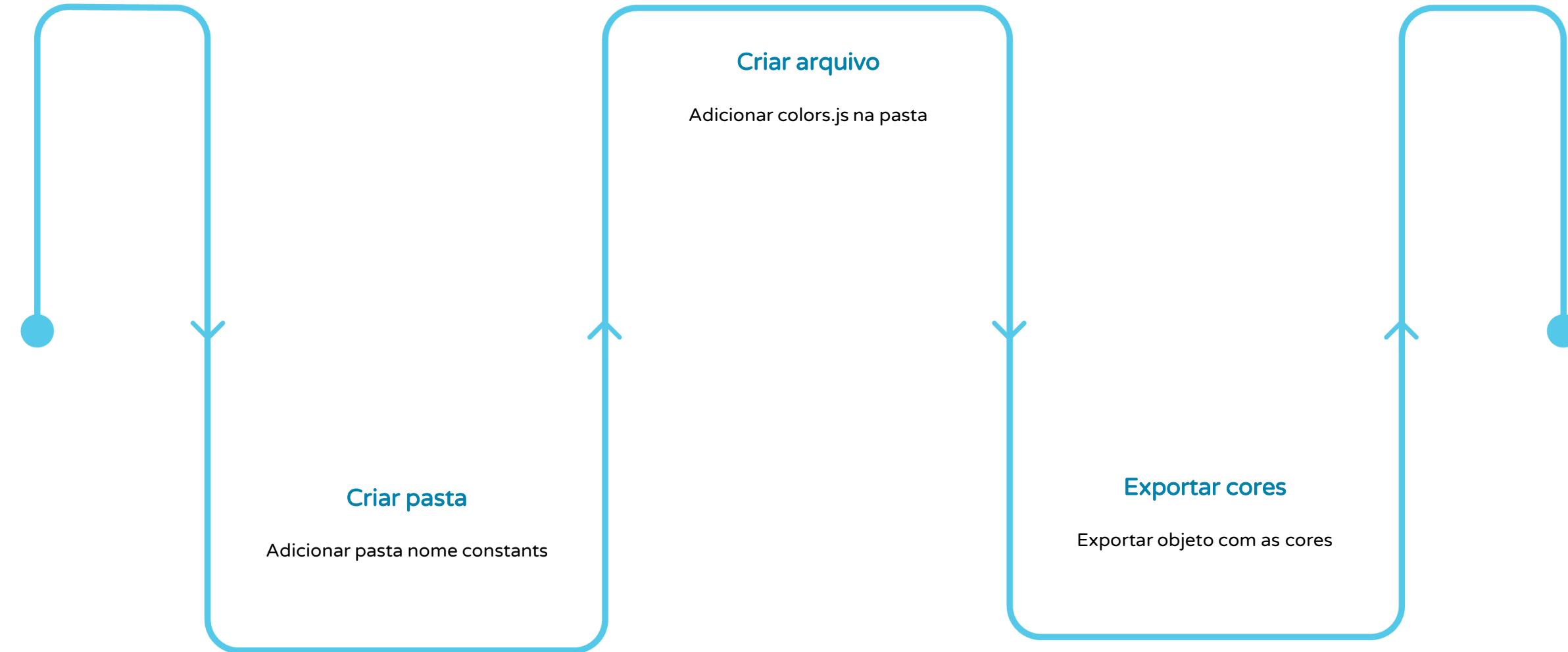
CAPÍTULO 04

# Trabalhando com Cores e Constantes

Organizando paletas de cores para design consistente

# Criando um Arquivo de Constantes de Cores

Uma boa prática é centralizar as cores do projeto em um arquivo separado. Isso facilita manutenção e garante consistência visual em toda aplicação.



Esta abordagem permite alterar cores globalmente e mantém o código mais limpo e profissional.

# Estrutura do Arquivo de Cores

Crie o arquivo `constants/colors.js` com a seguinte estrutura:

```
// constants/colors.js
export const colors = {
  primary: '#00A1FF',
  secondary: '#0082AD',
  darkGray: '#343739',
  mediumGray: '#747372',
  white: '#FFFFFF',
  gray: '#747372'
};
```

Para usar essas cores no seu componente, importe o objeto:

```
import { colors } from "../constants/colors";

const estilo = StyleSheet.create({
  title: {
    fontSize: 30,
    fontFamily: "Calibri",
    fontWeight: 600,
    color: colors.primary,
    marginLeft: 10
  }
});
```

Agora, sempre que precisar usar a cor primária, basta referenciar `colors.primary`, e se precisar mudá-la no futuro, você altera apenas um lugar!

# Vantagens de Usar Constantes



## Consistência

Todas as telas usam exatamente as mesmas cores, mantendo identidade visual uniforme



## Manutenibilidade

Altere uma cor em um único lugar e ela muda em todo o aplicativo automaticamente



## Colaboração

Equipes trabalham com mesmas referências, facilitando comunicação entre designers e desenvolvedores

 CAPÍTULO 05

# Layout com Flexbox

Organizando elementos na tela usando Flex

# Flexbox: Diferenças Cruciais no Mobile

## Web (CSS Tradicional)

Direção padrão: **row** (horizontal)

```
.container {  
  display: flex;  
  /* padrão: flex-direction: row */  
}
```

Elementos ficam lado a lado horizontalmente por padrão

## React Native

Direção padrão: **column** (vertical)

```
const styles = StyleSheet.create({  
  container: {  
    display: "flex"  
    // padrão: flexDirection: "column"  
  }  
});
```

Elementos ficam empilhados verticalmente por padrão

- ❑ **Por quê?** Em dispositivos móveis, a visualização vertical (scroll para baixo) é mais natural e comum que a horizontal.

# Criando um Container com Flex Row

Para alinhar a imagem e o texto horizontalmente, precisamos criar um container com `flexDirection: "row"`:

```
const estilo = StyleSheet.create({
  rowContainer: {
    display: "flex",
    flexDirection: "row",
    alignItems: "center",
    gap: 10,
    marginBottom: 20
  }
});
```

Agora envolvemos os elementos em uma View:

```
<View style={estilo.rowContainer}>
  <Image source={logo} style={estilo.image} />
  <Text style={estilo.title}>Minhas Tarefas</Text>
</View>
```

- `display: "flex"` - ativa o flexbox
- `flexDirection: "row"` - alinha horizontalmente
- `alignItems: "center"` - centraliza verticalmente
- `gap: 10` - espaçamento entre elementos

OBS: Não esqueça de importar o componente `View` no início do script

# Propriedades Flexbox Mais Utilizadas



## flexDirection

Define direção dos elementos: "row", "column", "row-reverse", "column-reverse"



## justifyContent

Alinhamento no eixo principal: "flex-start", "center", "flex-end", "space-between", "space-around"



## alignItems

Alinhamento no eixo cruzado: "flex-start", "center", "flex-end", "stretch"



## flexGrow

Define capacidade de crescimento do elemento para ocupar espaço disponível (número)



## gap

Espaçamento entre elementos filhos (substitui margins complexas)



## flexWrap

Quebra elementos em múltiplas linhas: "nowrap", "wrap", "wrap-reverse"

□□ CAPÍTULO 06

# Estilizando Campos de Input

Criando interfaces de entrada de dados atraentes

# Estilo Completo para TextInput

O componente TextInput aceita diversos estilos para personalização. Veja um exemplo completo:

```
input: {  
  height: 40,  
  paddingHorizontal: 16,  
  borderColor: "gray",  
  borderWidth: 1,  
  borderRadius: 20,  
  flexGrow: 1  
}
```

Aplicando ao componente:

```
<TextInput  
  style={estilo.input}  
  placeholder="Adicione um item"  
/>
```

**height:** define altura do campo

**paddingHorizontal:** espaço interno lateral

**borderRadius:** cantos arredondados

**flexGrow:** ocupa espaço disponível

► CAPÍTULO 07

# Componentes Interativos: Pressable

Criando botões personalizados com feedback visual

# Por Que Usar Pressable em Vez de Button?



## Personalização Total

O Button tem aparência limitada e específica de cada plataforma. Com Pressable, você controla completamente o visual



## Feedback de Toque

Pressable oferece estados de interação (pressed, hover) que permitem criar experiências visuais responsivas ao toque



## Flexibilidade de Conteúdo

Pode conter qualquer componente interno: texto, imagens, ícones, ou combinações complexas

# Criando um Botão com Pressable

Vamos criar um botão circular estilizado para adicionar tarefas. Primeiro, definimos os estilos:

```
const estilo = StyleSheet.create({
  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"
  }
});
```

Agora implementamos o componente com funcionalidade de clique:

```
<Pressable
  onPress={() => alert("Oi")}
  style={estilo.button}
>
  <Text style={estilo.buttonText}>+</Text>
</Pressable>
```

# Adicionando Feedback Visual ao Pressable

Uma das grandes vantagens do Pressable é poder mudar a aparência quando o botão está sendo pressionado. Usamos a prop `pressed` para isso:

```
<Pressable
  onPress={() => alert("Oi")}
  style={({ pressed }) => [
    estilo.button,
    {
      backgroundColor: pressed
        ? "blue"
        : colors.primary
    }
  ]}
>
  <Text style={estilo.buttonText}>+</Text>
</Pressable>
```

## Como Funciona

O `style` pode receber uma função que retorna estilos. Esta função recebe um objeto com a propriedade `pressed`, que indica se o botão está sendo pressionado naquele momento.

## Array de Estilos

Retornamos um array com múltiplos estilos: o estilo base (`estilo.button`) e um objeto de estilo inline que muda baseado no estado `pressed`.

**OBS:** Para finalizar coloque o `input` e `botão` dentro de uma `View`, utilizando o estilo `rowContainer`

# Estrutura Final do Projeto

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

export default function RootLayout() {
  return (
    <ScrollView>
      <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("Oi")}
          style={({ pressed }) => [
            estilo.button,
            {
              backgroundColor: pressed
                ? "blue"
                : colors.primary
            }
          ]}
        >
          <Text style={estilo.buttonText}>+</Text>
        </Pressable>
      </View>
    </ScrollView>
  );
}

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"
  }
});
```

# Principais Aprendizados



## StyleSheet é CSS-in-JS

Usamos objetos JavaScript para definir estilos, não arquivos CSS separados



## camelCase é Obrigatório

Propriedades CSS viram camelCase: font-size → fontSize, background-color → backgroundColor



## Flexbox Padrão é Column

No mobile, flexDirection padrão é "column", não "row" como na web



## Organize Cores em Constantes

Centralize cores em arquivo separado para fácil manutenção e consistência



## Array para Múltiplos Estilos

Use arrays quando precisar aplicar mais de um estilo: style={[estilo1, estilo2]}



## Pressable para Interatividade

Use Pressable em vez de Button para ter controle total sobre aparência e estados

Com estes fundamentos, você está pronto para criar interfaces móveis profissionais e responsivas no React Native. Na próxima aula, vamos explorar componentes de lista para completar nossa aplicação de tarefas!



# 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**

