Desenvolvimento de Aplicativo Mobile

Material base desenvolvido pelo
Prof. Raphael Barreto

raphael.b.oliveira@docente.senai.br

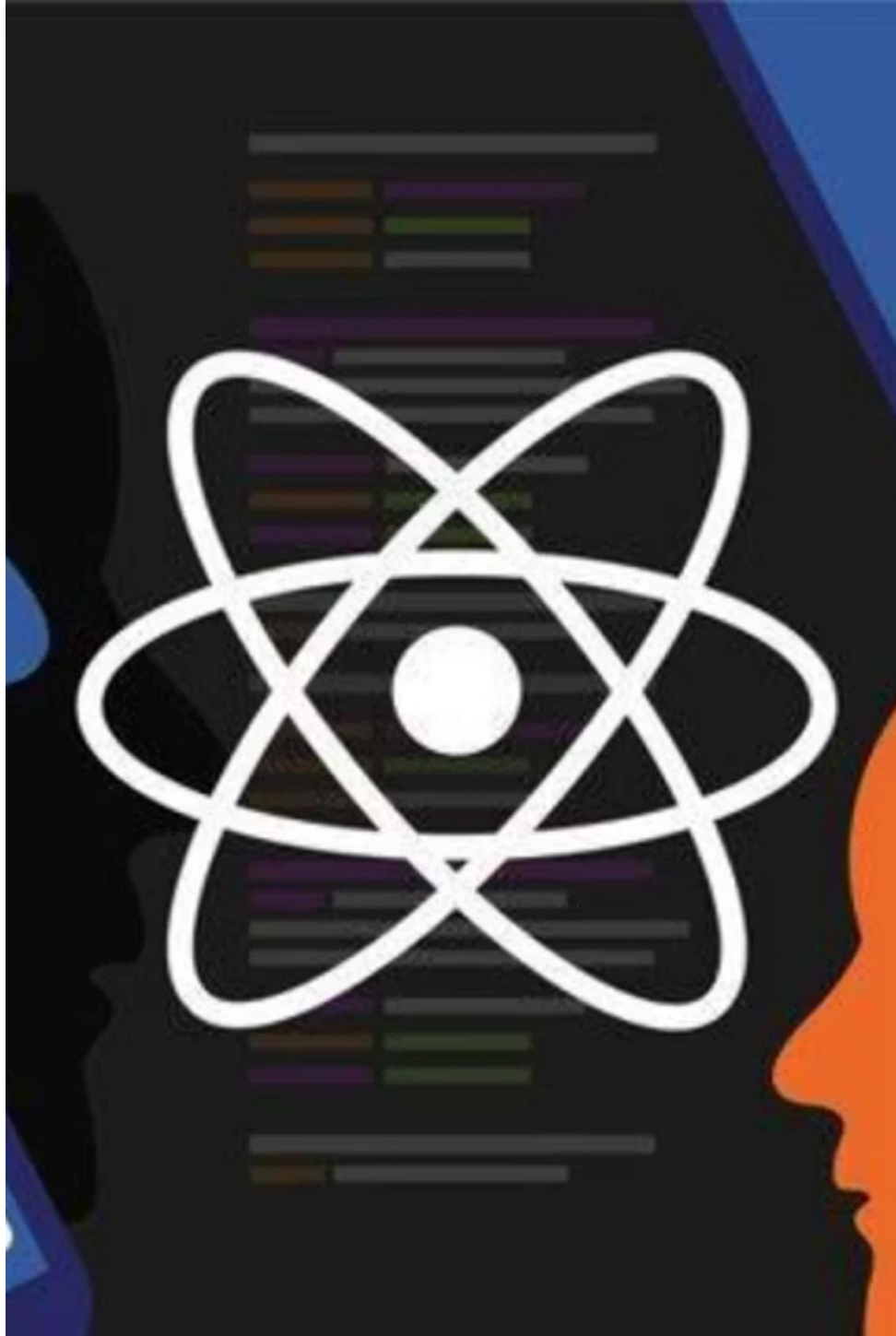
2026
SENAI - SAPUCAÍ

Fazendo a Lista de Tarefas Funcionar com React

Desenvolvendo uma aplicação completa com conceitos fundamentais do React Native

SENAI 2026

DESENVOLVIMENTO FRONT-END



Sumário da Aula

1. **Introdução ao Projeto**
   Visão geral da lista de tarefas e objetivos
2. **Gerenciamento de Estado**
   Implementando useState para tarefas e input
3. **Input Controlado**
   Diferenças entre React Web e React Native
4. **Adicionando Tarefas**
   Criação da função addTask e manipulação de arrays
5. **Completando Tarefas**
   Estado local e interatividade dos componentes
6. **Código Completo**
   Revisão final da aplicação funcional

CAPÍTULO 01

Introdução ao Projeto

Preparando nossa aplicação de lista de tarefas

O Que Vamos Construir

Nesta aula, vamos finalizar a implementação de uma lista de tarefas totalmente funcional. O objetivo é criar uma aplicação onde o usuário pode digitar uma tarefa no campo de input, apertar o botão de adicionar, e ver a tarefa aparecer instantaneamente na lista abaixo.

Além disso, cada tarefa poderá ser marcada como completa ou incompleta com um simples toque, mudando visualmente seu estado. Este projeto demonstra conceitos fundamentais do React Native que você utilizará em aplicações profissionais.

Limpeza Inicial do Código

Componentes Removidos

Antes de começarmos, precisamos fazer uma limpeza no código. Existem dois componentes importados que não estamos mais utilizando e que podem ser removidos:

- **Button**: Substituído por Pressable para maior personalização visual
- **ScrollView**: Trocada por View porque FlatList já é scrollável

Esta limpeza mantém nosso código organizado e evita avisos desnecessários no console.

Por Que Evitar ScrollView com FlatList?

A FlatList já é uma ScrollView por natureza. Colocar uma ScrollView dentro de outra não é uma boa prática em dispositivos móveis, onde o espaço é limitado. Embora funcione, o React Native emite avisos e pode causar problemas de performance.

CAPÍTULO 02

Gerenciamento de Estado

Transformando dados estáticos em estado reativo

De Variável Comum para Estado

A Transformação Necessária

Atualmente, nossa lista de tarefas é apenas uma variável comum no código. Isso significa que quando adicionamos novos itens, a interface não se atualiza automaticamente para refletir essas mudanças.

Para resolver isso, precisamos transformar essa lista em uma **variável de estado** usando o hook useState. Quando o estado muda, o React Native automaticamente re-renderiza a interface do usuário.

Esta é a essência da reatividade no React: conectar dados a componentes visuais.

Por Que Usar Estado?

O estado permite que a UI reaja automaticamente às mudanças nos dados. Sem ele, teríamos que manualmente forçar a atualização da tela toda vez que algo mudasse.

Criando o Estado das Tarefas

Vamos renomear nossa lista atual para **initialTasks**, mantendo algumas tarefas iniciais para visualização. Em seguida, criaremos o estado propriamente dito que irá gerenciar todas as tarefas da aplicação.

```
...
import { useState } from 'react';

const initialTasks = [
  { id: 1, completed: true, text: "Fazer café" },
  { id: 2, completed: false, text: "Estudar React Native" },
  { id: 3, completed: false, text: "Academia" }
]

export default function RootLayout() {
  const [tasks, setTasks] = useState(initialTasks)
  return (
...
```

Note que o **useState** é importado diretamente do React, não do React Native. Isso acontece porque o React Native é construído sobre o React, permitindo usar todos os hooks que você já conhece.

Estado para o Campo de Texto

Capturando o Input do Usuário

Além do estado das tarefas, precisamos de outro estado para capturar o que o usuário está digitando no campo de input. Este é um padrão comum chamado **input controlado**.

```
...
export default function RootLayout() {
  const [tasks, setTasks] = useState(initialTasks)
  const [text,setText] = useState("")
  return (
...
```

Com este estado, podemos conectar o valor digitado pelo usuário à nossa lógica de adicionar tarefas. O estado começa vazio e será preenchido conforme o usuário digita.

CAPÍTULO 03

Input Controlado no React Native

Diferenças importantes entre Web e Mobile

Input Controlado: Web vs Mobile

No React para web, controlamos inputs usando **value** e **onChange**, acessando o valor através de `e.target.value`. Essa abordagem funciona porque estamos lidando diretamente com o DOM do navegador.

onChange={(e) => setText(e.target.value)}

No React Native, não temos DOM! Estamos renderizando componentes nativos do iOS e Android, não elementos HTML. Por isso, a sintaxe é ligeiramente diferente, mas o conceito permanece o mesmo.

Implementando o TextInput Controlado

A Sintaxe Correta

No React Native, usamos duas propriedades no componente TextInput:

- **value**: Recebe o estado atual do texto
- **onChangeText**: Recebe a função que atualiza o estado

Note que usamos **onChangeText** ao invés de **onChange**. Além disso, não precisamos acessar `e.target.value` - o próprio componente passa o texto diretamente para nossa função.

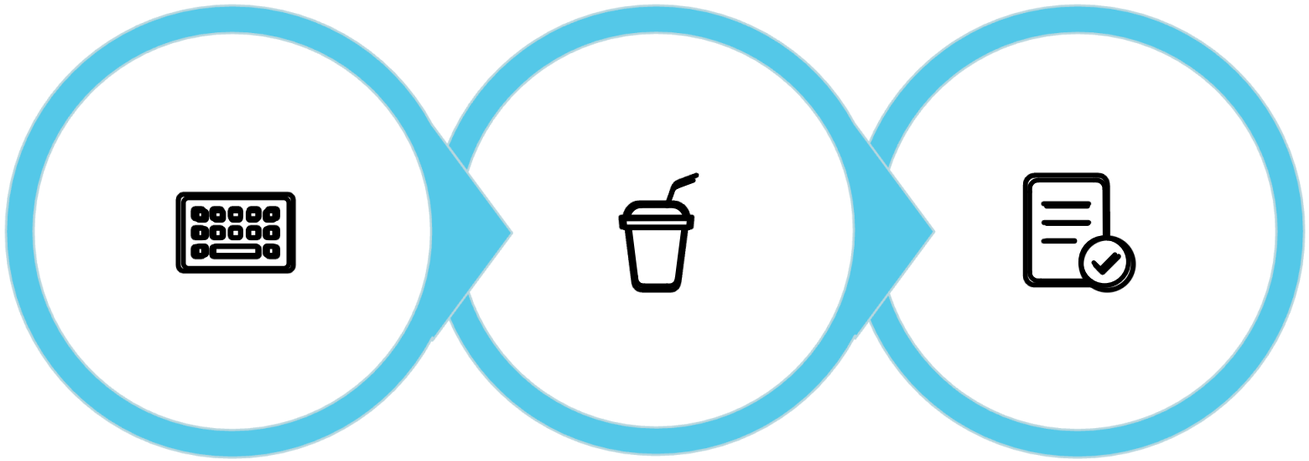
Em **Pressable**, substituimos o **"oi"** por **text**, para que o **alert** possa receber o texto digitado pelo usuário.

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


...
    onPress={() => alert(text)}
...
```

Esta sintaxe mais limpa é uma das vantagens do React Native: menos código boilerplate e mais produtividade.

Testando o Input Controlado



Digitar texto

Atualizar estado

Re-renderizar

CAPÍTULO 04

Adicionando Tarefas Dinamicamente

Criando a função addTask e manipulando arrays

Estrutura da Função addTask

Objetivo da Função

A função addTask será chamada quando o usuário pressionar o botão "+".

Ela deve criar uma nova tarefa e adicioná-la à lista existente, mantendo todas as tarefas anteriores.

Em **Pressable**, **onPress** passará a receber a função addTast.

Sai:

```
onPress={() => alert(text)}
```

Entra:

```
onPress={addTask}
```

Criando a Nova Tarefa

Cada tarefa precisa seguir a mesma estrutura dos objetos que já temos na lista: um **id** único, um estado **completed** (inicialmente false), e o **text** digitado pelo usuário.

```
...
const [text, setText] = useState("")

const addTask = () => {
   const newTask = { id: tasks.length + 1, completed: false, text }
}
...
```

Gerando IDs Únicos

Para manter cada tarefa identificável, precisamos atribuir um ID único. Neste exemplo didático, usamos uma abordagem simples: pegamos o tamanho atual do array (`tasks.length`) e adicionamos 1.

> **Atenção em Produção**
>
> Em aplicações profissionais, seria mais adequado usar bibliotecas como UUID para gerar identificadores verdadeiramente únicos e aleatórios. Nossa abordagem funciona para este exercício, mas pode ter problemas se tarefas forem deletadas.

O campo **completed** sempre começa como `false` (tarefa não concluída), e o **text** vem do nosso estado de input. Como o nome da propriedade e o valor são iguais, podemos usar a sintaxe abreviada do ES6.

Atualizando o Estado de Forma Imutável

Após criar a nova tarefa, precisamos adicioná-la ao estado. Aqui está um conceito crucial do React: **nunca modificamos o estado diretamente**. Não podemos usar `tasks.push(newTask)` porque isso modifica o array original.

Em vez disso, criamos um novo array copiando todos os elementos existentes e adicionando a nova tarefa no final:

```
...
const addTask = () => {
   const newTask = { id: tasks.length + 1, completed: false, text }
}
   setTasks([...tasks, newTask])
...
```

O operador spread (`...`) copia todos os elementos do array original, e então adicionamos o novo item. Isso cria um array diferente na memória, permitindo que o React detecte a mudança e atualize a interface.

Limpando o Campo de Input

Melhorando a Experiência do Usuário

Após adicionar uma tarefa, é uma boa prática limpar o campo de texto automaticamente. Isso prepara a interface para a próxima entrada e fornece feedback visual de que a ação foi concluída com sucesso.

```
...
const addTask = () => {
   const newTask = { id: tasks.length + 1, completed: false, text }
}
   setTasks([...tasks, newTask])
   setText("")  // Limpa o input
}
...
```

Conectando a Função ao Botão

Agora que temos nossa função **addTask** completa, precisamos conectá-la ao botão. No componente **Task.jsx** iremos receber além do **text**, o **completed** :

```
export default function Task({ text, completed }) {
```

Em index.jsx, iremos também passar além do text, o completed:

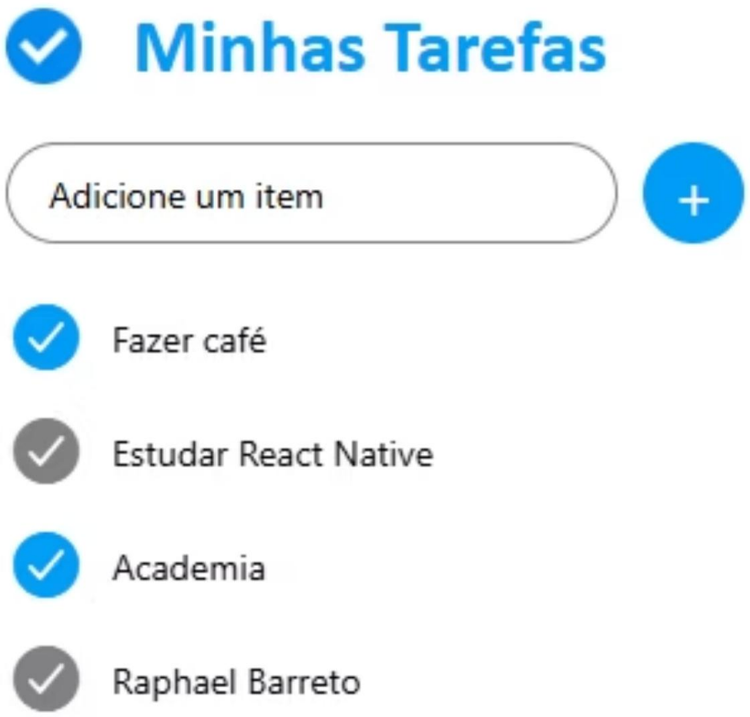
```
<FlatList
        data={tasks}
        keyExtractor={(item) => item.id.toString()}
        renderItem={({ item }) => <Task text={item.text} completed={item.completed} />}
      />
```

Quando o usuário pressiona o botão, a função é executada: uma nova tarefa é criada, adicionada à lista, e o campo é limpo. A FlatList automaticamente re-renderiza mostrando a nova tarefa porque o estado **tasks** mudou.

Trocando a cor de fundo dos ícones para itens completos

Em Task.jsx, iremos criar um ternário para trocar a cor de fundo do ícone, se o item da lista **completed** estiver com o valor **true**, iremos mostrar a cor de fundo primária, senão, se o valor for **false,** iremos mostrar a cor de fundo cinza.

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



Renderização Condicional do Ícone no componente Task

Agora podemos usar o estado **completed** para mudar a aparência visual da tarefa. Utilizamos um operador ternário para escolher a cor do ícone:

```
...
export default function Task({ text, initialCompleted }) {
    const [completed, setCompleted] = useState(initialCompleted)
    return (
        <View style={style.rowContainer}>
            <Pressable onPress={}>
                <Ionicons
                    name="checkmark-circle"
                    size={32}
                    color={completed ? colors.primary : "gray"}
                />
            </Pressable>
            <Text>{text}</Text>
        </View>)
}
...
```

Tarefas completas aparecem com o ícone na cor primária azul do SENAI, enquanto tarefas incompletas ficam em cinza. Esta diferenciação visual clara ajuda o usuário a entender rapidamente o estado de suas tarefas.

Estado Local no Componente Task

Com a função onPress criada, poderemos mudar o estado da tela conforme o estado do item **completed (true ou false)**. Então para cada tarefa, teremos um estado, que iremos chamar de **completed** e **setCompleted**. O estado inicial receberá completed que iremos chamar de **initialCompleted.**

Cada tarefa individual precisa gerenciar seu próprio estado de conclusão. Quando o usuário toca em uma tarefa, apenas aquela tarefa específica deve mudar, não todas as outras.

Por isso criamos um estado local dentro do componente Task.

Este é um conceito importante: **cada instância do componente tem seu próprio estado independente**. Três tarefas na lista significam três estados completed separados.

Passando Props para o Componente Task

Em _layout.jsx, iremos mudar o nome da prop da **FlatList** de completed para **initialCompleted**

Dados Necessários

Atualmente, o componente Task recebe apenas o texto da tarefa. Para exibir visualmente se uma tarefa está completa ou não, precisamos passar também o estado **completed**.

Modificamos o renderItem da FlatList para passar ambas as propriedades:

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

Por Que initialCompleted?

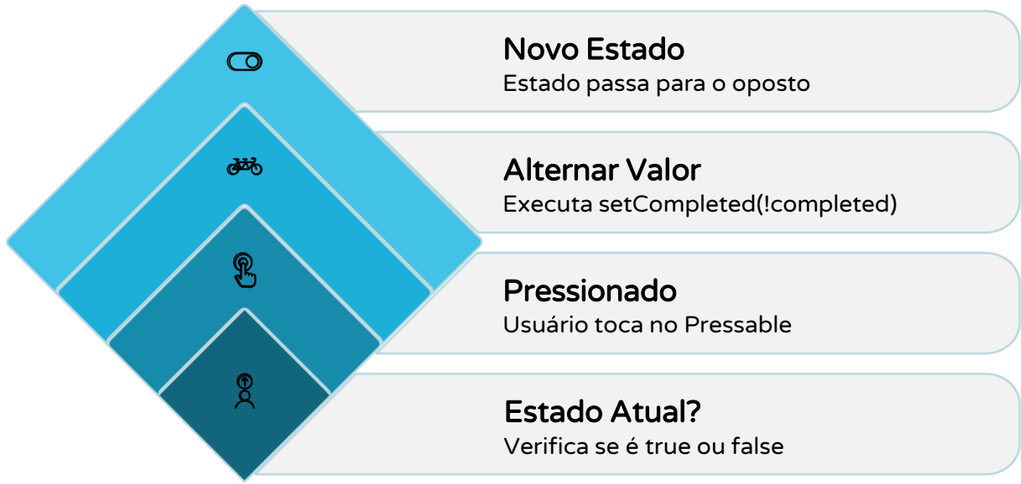
Chamamos de **initialCompleted** porque este será o valor inicial do estado interno do componente Task.

Alternando o Estado de Conclusão no componente Task

Implementando a Interatividade

Para permitir que o usuário marque/desmarque tarefas, envolvemos o conteúdo em um Pressable e implementamos a lógica de alternância:

```
...
<Pressable onPress={() => setCompleted(!completed)}>
   <Ionicons
     name="checkmark-circle"
     size={32}
     color={completed ? colors.primary : "gray"}
    />
</Pressable>
...
```



Aqui o que for **true** vai virar **false** e o que for **false** vai virar **true.** Vamos aplicar um setCompleted no oposto que temos **setCompleted(!completed)**

Aprendizagem com essa aula

Aprendemos com essa aula dois conceitos importantes, o primeiro é como fazer o input controlado :

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

O outro, é que estamos utilizando o **react** também, tudo o que fazemos com react, continua valendo aqui. O **estado** continua da mesa forma, atualizar o **array** também:

```
...
export default function RootLayout() {
  const [tasks, setTasks] = useState(initialTasks)
  const [text, setText] = useState("")

  const addTask = () => {
    const newTask = { id: tasks.length + 1, completed: false, text }
    setTasks([...tasks, newTask])
    setText("")
  }
...
```

Conceitos Fundamentais Aprendidos

Estado Reativo

Uso de useState para gerenciar dados que mudam e atualizar automaticamente a interface

Input Controlado

Diferenças entre React Web e React Native na captura de valores de formulário

Imutabilidade

Atualização correta de arrays sem modificar o estado original diretamente

Estado Local

Cada componente pode ter seu próprio estado independente dos demais

Props

Passagem de dados do componente pai para componentes filhos

FlatList

Renderização eficiente de listas com performance otimizada

React é React: Web ou Native

A grande mensagem desta aula é que **o React continua sendo o mesmo**, seja na web ou no mobile. Os conceitos fundamentais não mudam: componentes, props, estado, ciclo de vida com useEffect - tudo funciona da mesma forma.

As diferenças estão principalmente na camada de apresentação: usamos componentes nativos do React Native ao invés de elementos HTML, e adaptamos alguns eventos para o contexto móvel. Mas a lógica, a arquitetura e os padrões de desenvolvimento permanecem consistentes.

Por isso desenvolvedores React conseguem transicionar rapidamente para React Native. Você não está aprendendo uma tecnologia completamente nova - está aplicando conhecimentos existentes em um novo contexto. Na próxima aula, exploraremos navegação em aplicativos, um tema que realmente difere entre web e mobile!

Perguntas?

Entre em contato:

raphael.b.oliveira@docente.senai.br

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

Firjan SENAI