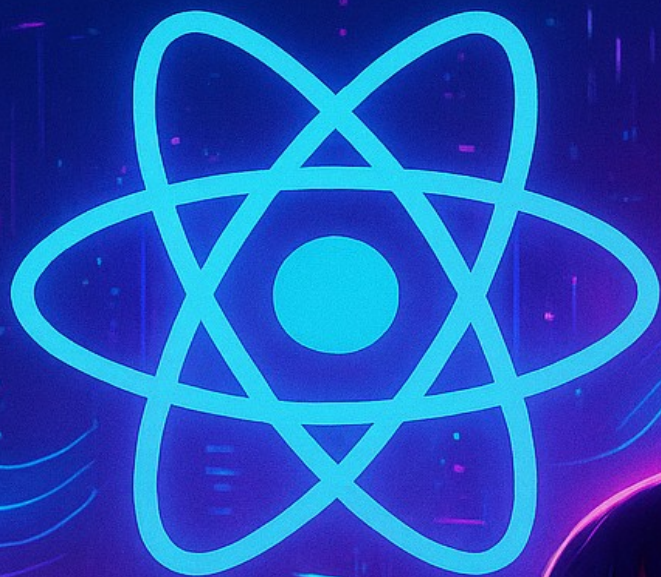


REACT

DO ZERO AO INTERMEDIÁRIO

Aprendizado prático com
exemplos, dicas e desafios



**ANA RAQUEL
DE HOLANDA**

[Desenvolvedora Java
apaixonada por música
e animes ♪ 🌸]

2025





INTRODUÇÃO

Se você nunca escreveu uma única linha de React, este e-book é para você. Ele foi criado com uma linguagem simples, visual, prática e cheia de analogias, ícones, caixas de alerta e exemplos diretos ao ponto 🎯.



Você aprenderá:

- A estrutura de um app React com Vite
- Componentes, props e estado (useState)
- Eventos, listas, formulários
- Renderização condicional
- Hooks e boas práticas








Objetivo: Tornar você capaz de **criar um projeto real de ToDo List** com React e salvar as tarefas localmente usando o `localStorage`.











O e-book foi construído **passo a passo**, para que você não se perca.

Você não precisa saber JavaScript profundamente, apenas ter **vontade de aprender** e praticar cada lição como uma jornada de desbloqueio de poderes de desenvolvedora.











































Vamos começar? 🚀

Sumário

 INTRODUÇÃO.....	2
■ Capítulo 1 – Introdução ao React.....	7
<i>i</i> O que você vai aprender aqui?.....	7
🧠 1.1 – O que é o React?.....	7
🔥 1.2 – Por que aprender React?.....	8
🧱 1.3 – O que você precisa saber antes de começar?.....	8
🔧 1.4 – O que você vai construir com este e-book?.....	8
📌 Dica rápida: SPA.....	8
📁 Resumo do Capítulo.....	8
■ Próximo capítulo:.....	9
■ Capítulo 2 – Como o React Funciona por Dentro 	9
<i>i</i> Neste capítulo, você aprenderá:.....	9
🧩 2.1 – O que é JSX?.....	9
🧱 2.2 – O que é um Componente?.....	9
🧠 2.3 – Virtual DOM.....	10
👥 2.4 – Tipos de Componentes.....	10
🔄 2.5 – Ciclo de Vida (resumido).....	10
✅ Exemplo completo:.....	10
📢 Aviso Importante.....	11
📁 Resumo do Capítulo.....	11
■ Próximo capítulo:.....	11
■ Capítulo 3 – Criando seu Primeiro Projeto React com Vite 	11
<i>i</i> Neste capítulo, você aprenderá:.....	11
🔧 3.1 – Preparando o ambiente.....	12
⚡ 3.2 – Criando o projeto com Vite.....	12
📁 3.3 – Instalando as dependências.....	12
🟢 3.4 – Rodando o projeto.....	12
📦 3.5 – Estrutura de arquivos.....	13
🔧 3.6 – Editando o App.....	13
✅ Conclusão do Capítulo.....	13
■ Próximo capítulo:.....	13
■ Capítulo 4 – Trabalhando com useState 	13
<i>i</i> Neste capítulo, você vai aprender:.....	13
💡 4.1 – O que é estado no React?.....	14
🔧 4.2 – Importando e usando useState.....	14
⚠️ Importante:.....	14
🔄 4.3 – O ciclo de atualização.....	15
🧠 4.4 – Exemplo com texto dinâmico.....	15
✅ Resultado esperado.....	15
💡 Dica:.....	15
■ Próximo capítulo:.....	16
■ Capítulo 5 – Lidando com Eventos e Inputs 	16
<i>i</i> Neste capítulo, você vai aprender:.....	16
🖱️ 5.1 – Manipulando eventos de clique.....	16
✍️ 5.2 – Controlando campos de input.....	16
📝 5.3 – Criando formulários com validação simples.....	17
⚠️ Dica de ouro:.....	17
📖 5.4 – Tipos de eventos mais usados.....	17

	Resumo.....	18
	Próximo capítulo:.....	18
	Capítulo 6 – Renderização Condicional 	18
	Neste capítulo, você vai aprender:.....	18
	6.1 – if fora do JSX.....	18
	6.2 – Operador Lógico &&.....	19
	6.3 – Operador Ternário ? :.....	19
	6.4 – Exemplo completo com estado.....	19
	Dicas importantes:.....	19
	Operadores úteis.....	20
	Resumo.....	20
	Próximo capítulo:.....	20
	Capítulo 7 – Renderizando Listas com .map() e a Prop key 	20
	Neste capítulo, você vai aprender:.....	20
	7.1 – O método .map() em ação.....	20
	7.2 – O erro clássico: faltando key.....	21
	Dica:.....	21
	7.3 – Renderizando componentes com .map().....	21
	7.4 – Lista com botão de remover item.....	22
	Evite isso:.....	22
	Resumo.....	22
	Exercício prático:.....	22
	Próximo capítulo:.....	23
	Capítulo 8 – Manipulando Eventos no React  	23
	Neste capítulo, você aprenderá:.....	23
	8.1 – Evento de clique: onClick.....	23
	8.2 – Evento com parâmetro.....	23
	8.3 – Manipulando inputs.....	24
	8.4 – Trabalhando com formulários.....	24
	8.5 – Eventos de teclado, mouse, etc.....	24
	8.6 – Exemplo prático: botão de like.....	25
	Dica prática:.....	25
	Resumo.....	25
	Exercício proposto:.....	25
	Próximo capítulo:.....	25
	Capítulo 9 – Estado entre Componentes e Lifting State Up  	25
	Neste capítulo, você aprenderá:.....	26
	9.1 – Comunicação entre componentes.....	26
	Pai → Filho (props).....	26
	Filho → Pai (função por prop).....	26
	9.2 – O que é "Lifting State Up"?.....	26
	Exemplo clássico: input + lista.....	27
	Regra de ouro:.....	27
	Erro comum:.....	28
	Exercício prático:.....	28
	Resumo.....	28
	Próximo capítulo:.....	28
	Capítulo 10 – Renderização Condicional  	28
	Neste capítulo, você aprenderá:.....	28
	10.1 – O que é renderização condicional?.....	28
	Exemplo básico com if.....	29
	Operador ternário condição ? valor1 : valor2.....	29

✓	Renderização parcial com &&.....	29
✓	Return antecipado (early return).....	29
🔧	Aplicando no ToDoList:.....	29
💡	Extras comuns em SPAs:.....	30
🎯	Dica visual:.....	30
🧠	Resumo.....	30
➡️ SOON	Próximo capítulo:.....	30
📘	Capítulo 11 – Trabalhando com Formulários e Inputs Controlados 🛠️🔧	30
📖	Neste capítulo, você vai aprender:.....	30
🎯	11.1 – Inputs controlados: o que são?.....	31
✓	Exemplo básico.....	31
⚠️	Por que usar e.preventDefault()?.....	31
🔧	11.2 – Múltiplos campos no estado.....	31
📦	11.3 – Inputs de checkbox, radio, select.....	32
✓	Checkbox.....	32
✓	Radio.....	32
✓	Select.....	32
✓	11.4 – Exemplo prático: formulário de tarefa.....	32
🧠	Resumo.....	33
⚙️	Desafio!.....	33
➡️ SOON	Próximo capítulo:.....	33
📘	Capítulo 12 – useEffect: Efeitos Colaterais em Componentes Funcionais ⚙️🌐	33
📖	Neste capítulo você vai aprender:.....	33
🧠	12.1 – O que são efeitos colaterais?.....	34
✓	12.2 – Sintaxe do useEffect.....	34
✓	Executa sempre que as dependências mudam.....	34
✓	Se as dependências forem [], roda apenas uma vez após o primeiro render.....	34
⚠️	Exemplo: título da página.....	34
🌐	12.3 – Buscando dados de uma API.....	34
🧠	12.4 – Limpando efeitos (cleanup).....	35
📌	Dicas práticas.....	35
🧠	Resumo.....	35
💡	Desafio:.....	35
➡️ SOON	Próximo capítulo:.....	36
📘	Capítulo 13 – Componentes Reutilizáveis, props.children e Composição 🌱🌿	36
📖	Neste capítulo você vai aprender:.....	36
🔄	13.1 – A importância da reutilização.....	36
✓	Exemplo: Botão reutilizável.....	36
👦	13.2 – O props.children: conte todo o seu conteúdo!.....	36
🧩	13.3 – Composição: componentes como blocos de Lego.....	37
⚠️	Evite "componentes burrinhos demais".....	37
🧱	13.4 – Layouts compostos com children.....	38
🧠	Resumo.....	38
💡	Desafio prático:.....	38
➡️ SOON	Próximo capítulo:.....	38
📘	Capítulo 14 – Formulários Controlados em React 💬📄	39
📖	Você vai aprender:.....	39
🔧	14.1 – Estado controlado: o que é isso?.....	39
📄	Exemplo: Campo de texto controlado.....	39
✓	14.2 – Inputs checkbox e select.....	39
🛑	14.3 – Prevenindo o envio padrão.....	40
🔍	14.4 – Dicas e boas práticas.....	40

	Resumo.....	41
	Desafio prático:.....	41
	Próximo capítulo:.....	41
	Capítulo 15 – Comunicação Entre Componentes  ↔	41
	Você vai aprender:.....	41
	15.1 – O problema da separação.....	41
	Isso dá problema:.....	42
	15.2 – Lifting State Up (Subir o estado).....	42
	Exemplo Prático:.....	42
	15.3 – Callback como prop.....	42
	15.4 – Compartilhando estado entre irmãos.....	43
	Dica Prática.....	43
	Resumo.....	43
	Desafio prático:.....	43
	Próximo capítulo:.....	43
	Capítulo 16 – Refs em React: Acessando Elementos Diretamente  	44
	Você vai aprender:.....	44
	16.1 – O que é uma ref?.....	44
	Com o hook useRef() você pode:.....	44
	Importante.....	44
	16.2 – Acessando um input com ref.....	44
	16.3 – Guardando valores sem re-render.....	45
	16.4 – Quando usar useRef.....	45
	Dica Profissional.....	45
	Desafio prático.....	46
	Próximo capítulo:.....	46
	Capítulo 17 – Renderização Condicional em React  	46
	Você vai aprender:.....	46
	17.1 – Renderização condicional: o conceito.....	46
	17.2 – Usando if (fora do JSX).....	46
	17.3 – Operador ternário ? :.....	47
	17.4 – Operador lógico &&.....	47
	17.5 – Evite isso!.....	47
	17.6 – Renderizando listas com .map().....	47
	Revisão do Capítulo.....	48
	Desafio prático.....	48
	BIBLIOGRAFIA UTILIZADA.....	48

Capítulo 1 – Introdução ao React

O que você vai aprender aqui?

Neste capítulo, você entenderá:

- O que é o React e por que ele é tão popular 💡
 - Onde ele é usado e por quem 🏢
 - O que você precisa saber para começar 🚀
-

1.1 – O que é o React?

React é uma **biblioteca JavaScript** criada pelo **Facebook** para construir **interfaces de usuário (UI)** de forma rápida, organizada e reativa.

💬 Imagine que seu site é como um LEGO: React permite construir cada pedacinho (botões, caixas, listas) como peças independentes chamadas **componentes**.

1.2 – Por que aprender React?

Mais usado do mercado:

React está presente em grandes empresas como Netflix, Instagram, Uber e Mercado Livre.

Alta empregabilidade:

É uma das stacks mais exigidas em vagas de front-end.

Reutilização de código:

Você cria um componente uma vez e usa onde quiser.

Comunidade gigante:

Muita documentação, bibliotecas, fóruns e tutoriais.

1.3 – O que você precisa saber antes de começar?

Pré-requisitos mínimos:

- HTML + CSS (só o básico)
- JavaScript (variáveis, funções, arrays e objetos)
- Lógica de programação (if, for, funções)

Se você está insegura com JS, tudo bem! Vamos explicando na prática com exemplos visuais. Mas recomendo depois estudar mais JS para avançar com tranquilidade. 😊

1.4 – O que você vai construir com este e-book?

Durante o e-book, você criará **vários mini-projetos**, e ao final, um projeto completo de **ToDo List com salvamento no navegador** usando **Hooks**, **componentes reutilizáveis** e **estilização moderna**.

Dica rápida: SPA

React é focado em **SPA – Single Page Applications**, ou seja, uma única página que se atualiza dinamicamente sem recarregar o navegador inteiro.

Resumo do Capítulo

Conceito	Descrição
React	Biblioteca JS para criar interfaces (UI)
Criado por	Facebook
Usa componentes	Blocos reutilizáveis de código
Tipo de aplicação	SPA (Single Page Application)
Por que aprender	Alta demanda, comunidade forte, produtividade

Próximo capítulo:

Capítulo 2 – O que é React e como ele funciona por dentro

Capítulo 2 – Como o React Funciona por Dentro

Neste capítulo, você aprenderá:

- O que são **componentes** e **JSX**
 - Como o React usa o **DOM Virtual**
 - A diferença entre **classe** e **função**
 - O ciclo de vida básico de um componente
-

🧩 2.1 – O que é JSX?

JSX (JavaScript XML) é a **linguagem visual** que usamos para escrever a interface no React. Ela **parece HTML**, mas é **JavaScript disfarçado**.

💬 Exemplo:

```
function Saudacao() {  
  return <h1>Olá, Ana! 🌻</h1>;  
}
```

✅ Isso *não é HTML* – é JSX, e precisa ser **convertido** em código JavaScript puro por trás dos panos com uma ferramenta chamada **Babel**.

🧱 2.2 – O que é um Componente?

Um **componente** é uma **função** que retorna uma parte da interface do usuário.

Cada botão, título, card, formulário — tudo é um componente.

💡 **Exemplo prático:**

```
function BotaoLegal() {  
  return <button>🔔 Clique Aqui</button>;  
}
```

Você pode usar esse botão onde quiser assim:

```
<BotaoLegal />
```

🧠 2.3 – Virtual DOM

O **DOM Virtual** é como um "**espelho invisível**" da sua página na memória.

Quando algo muda, o React:

1. Compara o DOM real com o DOM virtual 🕒
 2. Atualiza **somente o que mudou**, e não a página toda ⚡
 3. Isso torna seu app **rápido e eficiente**
-

👤 2.4 – Tipos de Componentes

React permite dois tipos principais:

Tipo	Exemplo	Uso atual
Componente de Função	<code>function</code>	✅ Recomendado
Componente de Classe	<code>class</code>	❌ Antigo, não recomendado
💬 Hoje usamos Hooks com funções para tudo. Esqueça classes.		

2.5 – Ciclo de Vida (resumido)


Um componente React tem 3 fases principais:

Fase	Descrição
Montagem	Quando o componente aparece na tela
Atualização	Quando o estado ou props mudam
Desmontagem	Quando o componente sai da tela

Usamos **Hooks** como `useEffect` para controlar essas fases.

Exemplo completo:

```
function BemVinda(props) {  
  return <h2>Bem-vinda, {props.nome}! 🌸</h2>;  
}
```

 props são os **dados externos** que você envia para o componente.

Uso:

```
<BemVinda nome="Ana" />
```

Aviso Importante

 O JSX **sempre precisa retornar um único elemento-pai**.

Errado:

```
return <h1>Oi</h1><p>Seja bem-vinda</p>; ❌
```

Correto:

```
return (  
  <div>  
    <h1>Oi</h1>  
    <p>Seja bem-vinda</p>  
  </div>  
) ; ✅
```

Resumo do Capítulo

Conceito	Definição
JSX	Sintaxe parecida com HTML usada dentro do React
Componente	Função que retorna UI
DOM Virtual	Representação leve do DOM usada para atualizações rápidas
Props	Informações passadas de pai para filho
Hooks	Funções especiais do React para gerenciar estados e ciclos

Próximos capítulos:

Capítulo 3 – Criando seu primeiro projeto React com Vite

Capítulo 3 – Criando seu Primeiro Projeto React com Vite


Neste capítulo, você aprenderá:

- Como instalar o ambiente
 - Como criar um projeto com **Vite**
 - Como rodar seu primeiro app React localmente
 - Como organizar os arquivos
-

3.1 – Preparando o ambiente

Antes de tudo, você precisa de 3 ferramentas instaladas:

Ferramenta	Link para download	Motivo
Node.js	nodejs.org	Executar o React
Git	git-scm.com	Versionamento e GitHub
VS Code	code.visualstudio.com	Editor de código leve e poderoso

 Verifique se o Node está funcionando:

```
node -v  
npm -v
```

3.2 – Criando o projeto com Vite

O Vite é uma ferramenta **super rápida** para criar projetos React.

Execute no terminal:

```
npm create vite@latest meu-todo-list
```

Siga os passos:

1. Dê o nome do projeto: `meu-todo-list`
 2. Selecione: **React**
 3. Escolha: **JavaScript** (ou **TypeScript** se preferir)
-

3.3 – Instalando as dependências

Entre na pasta do projeto e instale:

```
cd meu-todo-list
npm install
```

3.4 – Rodando o projeto

Agora execute:

```
npm run dev
```

Abra o navegador e acesse o endereço que aparece, geralmente:

```
http://localhost:5173/
```

 Você verá a **página inicial do Vite + React!**


3.5 – Estrutura de arquivos

Entenda os diretórios principais:

```
meu-todo-list/
├── public/           # Arquivos públicos (favicon, index.html)
├── src/              # Onde vamos programar!
│   ├── App.jsx      # Componente principal
│   └── main.jsx      # Ponto de entrada da aplicação
└── package.json     # Lista de dependências e scripts
```

3.6 – Editando o App

Abra o arquivo `src/App.jsx` e modifique o conteúdo:

```
function App() {
  return (
    <div>
      <h1>Olá, Ana!  </h1>
      <p>Este é seu primeiro app React com Vite!</p>
    </div>
  );
}
```

```
export default App;
```

Salve e veja o navegador atualizar **automaticamente**. 

Conclusão do Capítulo

Etapa	Resultado esperado
Criou com Vite	Projeto pronto em segundos

Etapa	Resultado esperado
Rodou o app localmente	Acesso via localhost:5173
Editou o App.jsx	Viu a mudança em tempo real

Próximo capítulo:

Capítulo 4 – Introdução aos Estados com useState 🧠

Excelente, Ana! Vamos seguir com o **Capítulo 4**: a base da interatividade em React começa aqui. 💡

Capítulo 4 – Trabalhando com useState 🧠

Neste capítulo, você vai aprender:

- O que é **estado (state)**
 - Como usar o **hook useState**
 - Criar seu primeiro botão interativo
 - Entender o ciclo de renderização
-

4.1 – O que é estado no React?

O **estado** é como a "memória" de um componente.


Ele guarda informações que **podem mudar com o tempo**, como:


- Itens de uma lista
- Nome digitado em um input
- Contador de cliques

✅ Toda vez que o estado muda, o React **renderiza novamente** o componente com os novos valores.

4.2 – Importando e usando useState

Abra o arquivo App.jsx e altere assim:

```
//  Adicione esse import no topo
import { useState } from 'react';

function App() {
  //  useState retorna [estado, funçãoParaAtualizar]
  const [contador, setContador] = useState(0);
```

```
return (
  <div>
    <h1>Contador 🇧🇷</h1>
    <p>Valor atual: {contador}</p>

    {/* Botão para incrementar */}
    <button onClick={() => setContador(contador + 1)}>
      ➕ Aumentar
    </button>

    {/* Botão para resetar */}
    <button onClick={() => setContador(0)}>
      ↺ Resetar
    </button>
  </div>
);
}

export default App;
```

⚠ Importante:

- `useState(0)` → O estado **inicial** é 0
 - `setContador()` → Serve para **mudar o valor**
 - `{contador}` → Usamos **chaves** para mostrar o valor no HTML
-

🔄 4.3 – O ciclo de atualização

Quando você clica no botão:

1. A função `setContador` muda o valor do estado
2. O componente **App** é **re-renderizado**
3. O React mostra a nova contagem na tela

✅ **Tudo acontece automaticamente.** Você não precisa atualizar nada manualmente.

🧠 4.4 – Exemplo com texto dinâmico

Vamos adicionar um campo de texto controlado pelo estado:

```
function App() {
  const [nome, setNome] = useState('');

  return (
    <div>
      <h1>Olá, {nome || 'visitante'}! 🙌</h1>

      <input
        type="text"
        placeholder="Digite seu nome"
        value={nome}
        onChange={(e) => setNome(e.target.value)}
      />
    </div>
  );
}
```

```
    />  
  </div>  
);  
}
```

✅ Resultado esperado

Ação	Resultado
Clicou em +	Contador aumenta
Clicou em ↺	Contador volta para zero
Digitou no input	Nome aparece dinamicamente



Dica:

- React **não** permite alterar o DOM diretamente
 - Sempre use os **hooks** como `useState`, `useEffect`, etc.
-



Próximo capítulo:

Capítulo 5 – Lidando com Eventos e Inputs 🧩

Vamos construir interações mais ricas com formulário e validações.



Capítulo 5 – Lidando com Eventos e Inputs 🎯



Neste capítulo, você vai aprender:

- Como lidar com **eventos de clique e digitação**
 - Controlar inputs com `useState`
 - Prevenir o comportamento padrão dos formulários
 - Validar campos simples
-



5.1 – Manipulando eventos de clique

No React, você adiciona eventos com `onClick`, `onChange`, etc. A diferença é que tudo é feito com **camelCase**:

```
<button onClick={minhaFuncao}>Clique aqui</button>
```



Exemplo prático:

```
function App() {
  const handleClick = () => {
    alert('Você clicou no botão! 🚀');
  };

  return <button onClick={handleClick}>Clique aqui</button>;
}
```

🔧 5.2 – Controlando campos de input

Inputs em React funcionam melhor como **inputs controlados** — ou seja, com valor vindo do `useState`.

```
function App() {
  const [email, setEmail] = useState('');

  return (
    <div>
      <input
        type="email"
        value={email}
        onChange={(e) => setEmail(e.target.value)}
        placeholder="Digite seu e-mail"
      />
      <p>Você digitou: {email}</p>
    </div>
  );
}
```

🔄 Toda digitação aciona `setEmail`, que **atualiza o estado**, que por sua vez **atualiza a tela**.

📝 5.3 – Criando formulários com validação simples

React não tem um sistema nativo de formulários, mas você pode criar com HTML + `useState`.

```
function App() {
  const [nome, setNome] = useState('');
  const [mensagem, setMensagem] = useState('');

  const handleSubmit = (e) => {
    e.preventDefault(); // ⚠️ Previne recarregamento da página
    if (nome.trim() === '') {
      setMensagem('⚠️ Nome é obrigatório!');
    } else {
      setMensagem(`✅ Olá, ${nome}! Formulário enviado.`);
    }
  };

  return (
    <form onSubmit={handleSubmit}>
      <input
        type="text"
        placeholder="Seu nome"
        value={nome}
        onChange={(e) => setNome(e.target.value)}
      />
      <button type="submit">Enviar</button>
      <p>{mensagem}</p>
    </form>
  );
}
```



```
    </form>
  );
}
```

⚠ Dica de ouro:

- Sempre use `e.preventDefault()` para **impedir o comportamento padrão** de `<form>`
 - Valide campos com `trim()` para evitar espaços vazios
-

📖 5.4 – Tipos de eventos mais usados

Evento	Descrição
<code>onClick</code>	Quando o usuário clica
<code>onChange</code>	Quando o valor de input muda
<code>onSubmit</code>	Quando um formulário é enviado
<code>onKeyDown</code>	Quando uma tecla é pressionada
<code>onBlur</code>	Quando um campo perde o foco

🧠 Resumo

- ✓ Eventos em React funcionam com funções JavaScript
 - ✓ Use `useState` para inputs controlados
 - ✓ Sempre capture `e.target.value` nos inputs
 - ✓ Nunca se esqueça do `e.preventDefault()` em formulários
-

🎯 Próximo capítulo:

Capítulo 6 – Renderização Condicional e Operadores Lógicos

Você aprenderá a mostrar ou esconder partes da tela de forma inteligente com `if`, `&&` e `?`.

📖 Capítulo 6 – Renderização Condicional 🌙

📘 Neste capítulo, você vai aprender:

- Como mostrar ou esconder elementos com base em condições
- Usar operadores `if`, `&&` e `?` : dentro do JSX

- Boas práticas para manter o código limpo
-

6.1 – if fora do JSX

Você pode usar `if` normalmente, **fora do JSX**, para decidir o que renderizar.

```
function App() {
  const [logado, setLogado] = useState(false);

  let mensagem;
  if (logado) {
    mensagem = <h1>Bem-vindo(a)! 🎉</h1>;
  } else {
    mensagem = <h1>Por favor, faça login.</h1>;
  }

  return (
    <div>
      {mensagem}
      <button onClick={() => setLogado(!logado)}>Alternar</button>
    </div>
  );
}
```

6.2 – Operador Lógico &&

Use condição `&&` JSX para renderizar algo **somente se a condição for verdadeira**:

```
{logado && <p>Você está conectado. ✅</p>}
```

Se `logado` for `false`, nada será exibido. É ótimo para componentes opcionais.

6.3 – Operador Ternário ? :

Ideal para renderizações rápidas com **duas opções**:

```
<p>{logado ? '✅ Online' : '❌ Offline'}</p>
```

É compacto e expressivo, mas evite ternários aninhados — prefira `if` para casos mais complexos.

6.4 – Exemplo completo com estado

```
function App() {
  const [idade, setIdade] = useState(0);

  return (
    <div>
      <input
        type="number"
        value={idade}
        onChange={(e) => setIdade(Number(e.target.value))}
        placeholder="Sua idade"
      />
    </div>
  );
}
```

```

    />

    {idade >= 18 ? (
      <p>✅ Maior de idade</p>
    ) : (
      <p>⚠️ Menor de idade</p>
    )}
  </div>
);
}

```

⚠️ Dicas importantes:

- ✅ Use `&&` para exibir apenas se for verdadeiro
 - ✅ Use `? :` para escolher entre duas opções
 - ✅ Prefira `if` fora do JSX quando as condições forem mais complexas
 - ✅ Nunca escreva lógica pesada diretamente no JSX
-

📖 Operadores úteis

Operador	Uso	Exemplo
<code>&&</code>	Se for verdadeiro, renderiza	<code>{condição && <p>Algo</p>}</code>
<code>? :</code>	Se for verdadeiro ou falso	<code>{condição ? A : B}</code>
<code>if</code>	Fora do JSX, para lógica complexa	<code>if (condição) { ... }</code>

🧠 Resumo

- ✅ React pode renderizar dinamicamente com base em estado
 - ✅ Use `&&` para exibição condicional
 - ✅ Use `? :` para escolhas binárias
 - ✅ Use `if` para lógica fora do JSX
-

🚀 Próximo capítulo:

Capítulo 7 – Trabalhando com Listas e a Prop key

Aprenda a renderizar coleções com `.map()`, como o React identifica cada item com `key`, e os erros mais comuns.

Capítulo 7 – Renderizando Listas com `.map()` e a Prop `key`

Neste capítulo, você vai aprender:


- Como renderizar listas dinamicamente no JSX
 - A importância da prop `key`
 - Erros comuns ao trabalhar com arrays
-

7.1 – O método `.map()` em ação

O `.map()` permite **iterar sobre arrays** e gerar JSX para cada item:

```
const nomes = ["Ana", "Lucas", "João"];

function App() {
  return (
    <ul>
      {nomes.map((nome) => (
        <li>{nome}</li>
      ))}
    </ul>
  );
}
```

 Cada item será transformado em um ``.

7.2 – O erro clássico: faltando `key`

Quando renderizamos listas, o React precisa identificar **cada item de forma única**.

```
<ul>
  {nomes.map((nome, index) => (
    <li key={index}>{nome}</li> // ⚠ usar index é aceitável, mas pode causar bugs
  ))}
</ul>
```

Dica:

Use um **ID único** como `key` sempre que possível:

```
const usuarios = [
  { id: 1, nome: "Ana" },
  { id: 2, nome: "Lucas" },
];

<ul>
  {usuarios.map((user) => (
    <li key={user.id}>{user.nome}</li>
  ))}
```

💡 7.3 – Renderizando componentes com .map()

Você pode renderizar componentes inteiros:

```
function Pessoa({ nome }) {
  return <p>👤 {nome}</p>;
}

function App() {
  const nomes = ["Ana", "Carlos", "Joana"];

  return (
    <div>
      {nomes.map((nome, index) => (
        <Pessoa key={index} nome={nome} />
      ))}
    </div>
  );
}
```

📦 7.4 – Lista com botão de remover item

```
function App() {
  const [lista, setLista] = useState(["React", "Vue", "Angular"]);

  const removerItem = (indexParaRemover) => {
    const novalista = lista.filter((_, index) => index !== indexParaRemover);
    setLista(novalista);
  };

  return (
    <ul>
      {lista.map((item, index) => (
        <li key={index}>
          {item}
          <button onClick={() => removerItem(index)}>🗑️</button>
        </li>
      ))}
    </ul>
  );
}
```

⚠️ Evite isso:

<li key={Math.random()}>{item} // ❌ NUNCA use valores aleatórios como `key`!

Isso faz o React recriar tudo do zero a cada renderização, perdendo performance e estado.


Resumo

- ✓ Use `.map()` para gerar elementos dinamicamente
 - ✓ Use `key` para identificar cada item da lista
 - ✓ Prefira IDs únicos ao invés de índices como `key`
 - ✓ Componentes também podem ser renderizados em listas
 - ✓ Evite `Math.random()` como chave
-

Exercício prático:

Crie um componente `ListaDeTarefas` que renderize:



- Um array de tarefas
- Um botão para marcar como feita
- Um botão para remover cada tarefa

 Este exercício será usado no Capítulo 8 para manipulação de estado mais complexa!

Próximo capítulo:

Capítulo 8 – Manipulando Eventos no React

Aprenda a lidar com cliques, formulários, inputs e interações dinâmicas no seu app React!

Perfeito, Ana! Vamos avançar para uma das engrenagens centrais de qualquer app React: **eventos!**  

Capítulo 8 – Manipulando Eventos no React

Neste capítulo, você aprenderá:

- Como lidar com eventos no React (cliques, inputs, formulários)
 - Como escrever funções de manipulação (handlers)
 - A diferença entre eventos no React e no HTML tradicional
-

✓ 8.1 – Evento de clique: `onClick`

```
function App() {  
  function handleClick() {  
    alert("🖱️ Você clicou no botão!");  
  }  
}
```

```

    }

    return <button onClick={handleClick}>Clique aqui</button>;
  }

```

⚠️ **Atenção: não coloque () após o nome da função** no `onClick`, pois isso executa a função automaticamente!

```

// Correto
onClick={handleClick}

// Errado ❌
onClick={handleClick()}

```

🛠️ 8.2 – Evento com parâmetro

Para passar parâmetros:

```

function App() {
  const saudacao = (nome) => alert(`Olá, ${nome}!`);

  return <button onClick={() => saudacao("Ana")}>Dizer Olá</button>;
}

```

🔧 8.3 – Manipulando inputs

```

function App() {
  const [nome, setNome] = useState("");

  function handleChange(evento) {
    setNome(evento.target.value);
  }

  return (
    <div>
      <input type="text" onChange={handleChange} />
      <p>Você digitou: {nome}</p>
    </div>
  );
}

```

🧠 `evento.target.value` é o valor atual do campo de texto!

✉️ 8.4 – Trabalhando com formulários

```

function Formulario() {
  const [email, setEmail] = useState("");

  function handleSubmit(e) {
    e.preventDefault(); // ✅ evita o recarregamento da página
    alert(`✉️ Email enviado: ${email}`);
  }

  return (
    <form onSubmit={handleSubmit}>

```

```
    <input
      type="email"
      value={email}
      onChange={(e) => setEmail(e.target.value)}
      required
    />
    <button type="submit">Enviar</button>
  </form>
);
}
```

⚙️ 8.5 – Eventos de teclado, mouse, etc.

React suporta os principais eventos DOM:

Evento HTML	Evento React
onclick	onClick
onchange	onChange
onsubmit	onSubmit
onkeydown, onkeyup	onKeyDown, etc.
onmouseenter	onMouseEnter

📦 8.6 – Exemplo prático: botão de like

```
function BotaoLike() {
  const [likes, setLikes] = useState(0);

  return (
    <button onClick={() => setLikes(likes + 1)}>
      ❤️ Curtidas: {likes}
    </button>
  );
}
```

💡 Dica prática:

Você pode declarar funções diretamente no JSX, mas **se forem muito complexas**, extraia para fora para melhorar a leitura do código.

🧠 Resumo

- ✅ Use `onClick`, `onChange`, `onSubmit` e outros para interagir com o usuário
 - ✅ Evite executar funções diretamente com `()` no JSX
 - ✅ `event.target.value` é o coração da leitura de inputs
 - ✅ Sempre use `e.preventDefault()` em formulários
 - ✅ Eventos no React seguem o padrão `camelCase` (diferente do HTML)
-

Exercício proposto:

Atualize o componente `ListaDeTarefas`:

- Adicione um `input` de texto e um botão "Adicionar"
 - O botão deve adicionar o novo item à lista
 - Adicione um campo para mostrar quantas tarefas existem
-



Próximo capítulo:

Capítulo 9 – Estado com múltiplos componentes e `lifting state up`

Como compartilhar dados entre componentes e organizar seu estado globalmente.

Capítulo 9 – Estado entre Componentes e Lifting State Up



Neste capítulo, você aprenderá:

- Como **passar dados de um componente para outro**
 - O que é o "**Lifting State Up**"
 - Quando e por que mover o `useState` para o componente pai
-



9.1 – Comunicação entre componentes



Pai → Filho (props)

Componentes filhos recebem **props** (propriedades) do componente pai.

```
function Saudacao({ nome }) {  
  return <p>Olá, {nome}!</p>;  
}  
  
function App() {  
  return <Saudacao nome="Ana" />;  
}
```



App envia `nome="Ana"` para `Saudacao`.

Filho → Pai (função por prop)

Se o filho precisa modificar algo no pai, ele pode **receber uma função** como prop:

```
function BotaoIncrementar({ aoClicar }) {  
  return <button onClick={aoClicar}>Adicionar</button>;  
}  
  
function App() {  
  const [contador, setContador] = useState(0);  
  
  return (  
    <>  
      <p>Contador: {contador}</p>  
      <BotaoIncrementar aoClicar={() => setContador(contador + 1)} />  
    </>  
  );  
}
```

🔗 O pai passa setContador (ou uma função que a utiliza) para o filho!

9.2 – O que é "Lifting State Up"?

“Lifting state up” é quando você move o `useState` para um componente mais acima na hierarquia, permitindo que outros componentes o acessem via props.

Exemplo clássico: input + lista

Imagine que você tem dois componentes:

```
function InputTarefa({ aoAdicionar }) {  
  const [texto, setTexto] = useState("");  
  
  const enviar = () => {  
    aoAdicionar(texto); // chama função do pai  
    setTexto("");  
  };  
  
  return (  
    <>  
      <input  
        value={texto}  
        onChange={(e) => setTexto(e.target.value)}  
        placeholder="Nova tarefa"  
      />  
      <button onClick={enviar}>➕</button>  
    </>  
  );  
}  
  
function Lista({ tarefas }) {  
  return (  
    <ul>  
      {tarefas.map((tarefa, i) => (  
        <li key={i}>{tarefa}</li>  
      ))}  
    </ul>  
  );  
}
```

```
    );  
  }  
}
```

Agora o componente pai:

```
function App() {  
  const [tarefas, setTarefas] = useState([]);  
  
  function adicionarTarefa(nova) {  
    setTarefas([...tarefas, nova]);  
  }  
  
  return (  
    <div>  
      <InputTarefa aoAdicionar={adicionarTarefa} />  
      <Lista tarefas={tarefas} />  
    </div>  
  );  
}
```

✅ App controla o estado e o compartilha entre InputTarefa e Lista.

🧠 Regra de ouro:

O estado deve estar no componente mais próximo que precisa ler e modificá-lo.
Se dois componentes usam o mesmo dado, mova o estado para o pai comum.

⚠️ Erro comum:

❌ Ter múltiplos estados duplicados em componentes diferentes (ex: tarefas em Lista e InputTarefa) leva à **inconsistência de dados**.

🔧 Exercício prático:

Transforme sua ToDo List em:

- Componente InputNovaTarefa
 - Componente ListaTarefas
 - Componente pai App controlando o estado
 - O botão "Adicionar" adiciona uma tarefa na lista
 - Cada tarefa da lista exibe um botão "Remover" usando uma função passada do pai
-

🧠 Resumo

- ✅ Use props para **enviar dados do pai para o filho**
- ✅ Use funções como props para **permitir o filho modificar o pai**

- ✓ "Lifting state up" evita estados duplicados e melhora a comunicação
 - ✓ Componentes devem ser **responsáveis apenas pela UI**, não pela lógica compartilhada
-



Próximo capítulo:

Capítulo 10 – Renderização Condicional: mostrando ou escondendo elementos com lógica



Capítulo 10 – Renderização Condicional



Neste capítulo, você aprenderá:

- Como **mostrar ou ocultar elementos com lógica**
 - Técnicas com `if`, operador `&&`, `? : e return early`
 - Padrões usados em SPAs com autenticação, loading e mensagens
-



10.1 – O que é renderização condicional?

Renderizar condicionalmente significa: **mostrar algo apenas se uma condição for verdadeira**.



Exemplo básico com `if`

```
function Saudacao({ logado }) {  
  if (logado) {  
    return <p>Bem-vinda, Ana! 🌸</p>;  
  } else {  
    return <p>Por favor, faça login. 🔒</p>;  
  }  
}
```



Operador ternário condição `? valor1 : valor2`


Mais compacto e muito usado:


```
<p>{logado ? "Bem-vinda, Ana!" : "Por favor, faça login."}</p>
```



Renderização parcial com `&&`



Se condição for verdadeira, o elemento à direita será renderizado.

```
{logado && <p>Você está conectada! 
```

 Se logado for false, **nada é renderizado**.

Return antecipado (early return)


Padrão para **retornar diferente UIs** conforme estados:

```
function Painei({ carregando, erro, dados }) {  
  if (carregando) return <p>Carregando...   if (erro) return <p>Erro: {erro.message}   if (!dados) return null;  
  
  return <div>Dados: {dados.nome}</div>;  
}
```



Aplicando no ToDoList:

Vamos tornar o app mais amigável com mensagens condicionais!

```
function Lista({ tarefas }) {  
  if (tarefas.length === 0) {  
    return <p> Nenhuma tarefa no momento. Adicione uma nova!</p>;  
  }  
  
  return (  
    <ul>  
      {tarefas.map((t, i) => (  
        <li key={i}>{t}</li>  
      ))}  
    </ul>  
  );  
}
```



Extras comuns em SPAs:

Situação	Técnica comum
Tela de carregamento	<code>if (carregando) return <Loading /></code>
Sem resultados	<code>if (itens.length === 0)</code>
Login / Logout	<code>user ? <Painei/> : <LoginForm/></code>
Ocultar botão	<code>{isAdmin && <BotaoAdmin />}</code>



Dica visual:

Você pode usar **ícones e emojis** para melhorar a UX de elementos condicionais (ex: , , , , , , , , , , , , , , , , , , , , , , , , , , , , , , , , , , , , , , , , , , , , , , , , , , ,

Resumo

- ✓ Use `if`, ternário ou `&&` para renderizar elementos conforme estado
 - ✓ Componentes podem **retornar diferente JSX**
 - ✓ Renderização condicional melhora UX e evita erros visuais
 - ✓ Muito útil para **controle de fluxo**, como autenticação ou carregamento
-



Próximo capítulo:

Capítulo 11 – Trabalhando com formulários e inputs controlados: como criar interfaces que leem e modificam o estado do usuário



Capítulo 11 – Trabalhando com Formulários e Inputs Controlados




Neste capítulo, você vai aprender:

- O que são **inputs controlados**
 - Como usar `useState` para gerenciar campos de texto
 - Como lidar com **eventos de formulário**
 - Criar formulários reativos e seguros
-



11.1 – Inputs controlados: o que são?

Um **input controlado** é aquele cujo valor está sempre sincronizado com o **estado do componente**.

 Isso dá ao React controle total sobre os dados digitados — permitindo validação, limpeza, formatação etc.



Exemplo básico

```
import { useState } from "react";

function Formulario() {
  const [nome, setNome] = useState("");

  function handleSubmit(e) {
    e.preventDefault(); // previne o refresh
    alert(`Seu nome é: ${nome}`);
  }
}
```

```

return (
  <form onSubmit={handleSubmit}>
    <input
      type="text"
      value={nome}
      onChange={(e) => setNome(e.target.value)}
      placeholder="Digite seu nome"
    />
    <button type="submit">Enviar</button>
  </form>
);
}

```

⚠ Por que usar `e.preventDefault()`?

Porque o `<form>` tenta **recarregar a página** ao submeter. Isso quebra o SPA.
Com `preventDefault()`, mantemos a submissão **no controle do React**.

🔧 11.2 – Múltiplos campos no estado

```

const [form, setForm] = useState({ nome: "", email: "" });

function handleChange(e) {
  setForm({ ...form, [e.target.name]: e.target.value });
}

<form>
  <input name="nome" value={form.nome} onChange={handleChange} />
  <input name="email" value={form.email} onChange={handleChange} />
</form>

```

✅ Com `[e.target.name]`, o `handleChange` é genérico!

📦 11.3 – Inputs de checkbox, radio, select

✅ Checkbox

```

<input
  type="checkbox"
  checked={aceitaTermos}
  onChange={(e) => setAceitaTermos(e.target.checked)}
/>

```

✅ Radio

```

<input
  type="radio"
  name="plano"
  value="free"
  checked={plano === "free"}
  onChange={(e) => setPlano(e.target.value)}
/>

```

✓ Select

```
<select value={pais} onChange={(e) => setPais(e.target.value)}>
  <option value="BR">Brasil</option>
  <option value="PT">Portugal</option>
</select>
```

✓ 11.4 – Exemplo prático: formulário de tarefa

```
function AdicionarTarefa({ onAdd }) {
  const [texto, setTexto] = useState("");

  function handleSubmit(e) {
    e.preventDefault();
    if (texto.trim() === "") return;
    onAdd(texto);
    setTexto("");
  }

  return (
    <form onSubmit={handleSubmit}>
      <input
        type="text"
        value={texto}
        onChange={(e) => setTexto(e.target.value)}
        placeholder="Nova tarefa"
      />
      <button>Adicionar</button>
    </form>
  );
}
```

Resumo

- ✓ Um **input controlado** sempre tem `value` atrelado ao estado
 - ✓ Eventos como `onChange`, `onSubmit` dão controle completo
 - ✓ Use `useState` para guardar e atualizar campos
 - ✓ Trabalhar com formulários é essencial para interfaces ricas e reativas
-

Desafio!

Tente criar um formulário que:

- Aceita nome, idade e hobby
 - Exibe os dados inseridos logo abaixo
 - Mostra mensagem de erro se algum campo estiver vazio ao enviar
-



Próximo capítulo:

Capítulo 12 – Ciclo de Vida com `useEffect`: Como reagir a mudanças, buscar dados e sincronizar estados.



Capítulo 12 – `useEffect`: Efeitos Colaterais em Componentes Funcionais



Neste capítulo você vai aprender:

- O que é um **efeito colateral** (side effect)
 - Como o `useEffect` funciona
 - Quando e como usá-lo corretamente
 - Como **buscar dados de uma API**
 - Como **limpar efeitos** (cleanup)
-



12.1 – O que são efeitos colaterais?

São ações que afetam o mundo fora do React:



buscar dados de uma API



manipular o `localStorage`



configurar `setInterval`



interagir com o DOM diretamente



sincronizar estado com variáveis externas



12.2 – Sintaxe do `useEffect`

```
useEffect(() => {  
  // código a ser executado  
}, [dependências]);
```



Executa sempre que as dependências mudam



Se as dependências forem [], roda apenas uma vez após o primeiro render



Exemplo: título da página

```
import { useEffect, useState } from "react";
```

```
function Contador() {
  const [contador, setContador] = useState(0);

  useEffect(() => {
    document.title = `Contador: ${contador}`;
  }, [contador]);

  return (
    <button onClick={() => setContador(contador + 1)}>
      Incrementar
    </button>
  );
}
```

✅ O título do navegador muda toda vez que contador mudar!

12.3 – Buscando dados de uma API

```
import { useState, useEffect } from "react";

function Usuarios() {
  const [usuarios, setUsuarios] = useState([]);

  useEffect(() => {
    fetch("https://jsonplaceholder.typicode.com/users")
      .then((res) => res.json())
      .then((dados) => setUsuarios(dados));
  }, []);

  return (
    <ul>
      {usuarios.map((user) => (
        <li key={user.id}>{user.name}</li>
      ))}
    </ul>
  );
}
```

✅ Só busca os dados **uma vez** após o primeiro render.

12.4 – Limpando efeitos (cleanup)

Ideal para timers, sockets ou listeners de evento!

```
useEffect(() => {
  const intervalo = setInterval(() => {
    console.log("tick...");
  }, 1000);

  return () => {
    clearInterval(intervalo);
  };
}, []);
```

✅ O return dentro do useEffect é a **função de limpeza** (cleanup).

Dicas práticas

Situação	Dependências	Observações
Buscar dados 1x ao carregar	<code>[]</code>	Sem dependência
Reagir a mudança de props/estado	<code>[valor]</code>	Roda sempre que <code>valor</code> mudar
<code>setInterval</code> , <code>addEventListener</code>	<code>[]</code> + <code>cleanup</code>	Sempre limpe no <code>return</code> !
Lidar com <code>localStorage</code>	<code>[]</code> ou <code>[chave]</code>	Dependendo da leitura

Resumo

- ✓ `useEffect` permite **sincronizar** o React com efeitos externos
 - ✓ Rode uma vez com `[]` ou toda vez que variáveis mudarem
 - ✓ Sempre **limpe efeitos com return** se necessário
 - ✓ Essencial para **fetch**, **DOM direto**, **timers** e muito mais
-

Desafio:

Crie um componente que:

- Mostra o tempo em segundos desde que a página foi carregada
 - Usa `useEffect` e `setInterval`
 - Para de contar ao clicar em um botão “Pausar”
-

Próximo capítulo:

Capítulo 13 – Componentes reutilizáveis, `props.children` e composição.

Capítulo 13 – Componentes Reutilizáveis, `props.children` e Composição

Neste capítulo você vai aprender:

- Como **reutilizar componentes** de forma elegante
- O que é o `props.children`
- Como criar **layouts composicionais**
- Aplicar o padrão de “**Slot**” e “**Wrapper**” no React



13.1 – A importância da reutilização

Componentes devem ser **simples, coesos e reutilizáveis**.

Ao invés de duplicar estrutura, você pode parametrizá-los.



Exemplo: Botão reutilizável

```
function Botao({ texto, cor }) {  
  return (  
    <button style={{ backgroundColor: cor }}>  
      {texto}  
    </button>  
  );  
}
```

// Uso:

```
<Botao texto="Salvar" cor="green" />  
<Botao texto="Cancelar" cor="gray" />
```



13.2 – O props.children: conte todo o seu conteúdo!

O React permite **injetar JSX dentro de um componente**, como se fosse um slot.

```
function Card({ children }) {  
  return (  
    <div style={{ border: "1px solid gray", padding: "1rem" }}>  
      {children}  
    </div>  
  );  
}
```

// Uso:

```
<Card>  
  <h2>Título</h2>  
  <p>Conteúdo do card</p>  
</Card>
```



Isso permite **encaixar qualquer conteúdo** dentro do componente.



13.3 – Composição: componentes como blocos de Lego

Você pode combinar componentes como peças:

```
function Painel({ titulo, children }) {  
  return (  
    <section>  
      <h2>{titulo}</h2>  
      {children}  
    </section>  
  );  
}
```

```
function App() {
  return (
    <Painel titulo="Perfil">
      <p>Nome: Ana</p>
      <p>Email: ana@email.com</p>
    </Painel>
  );
}
```

⚠️ Evite "componentes burrinhos demais"

Componentes que não aceitam `children` ou `props` são muito específicos.

Exemplo fraco:

```
function Mensagem() {
  return <p>Olá, usuário!</p>;
}
```

✅ Melhorar usando props:

```
function Mensagem({ texto }) {
  return <p>{texto}</p>;
}
```



13.4 – Layouts compostos com `children`

Imagine um layout de página com header, conteúdo e rodapé:

```
function Layout({ header, children, footer }) {
  return (
    <>
      <header>{header}</header>
      <main>{children}</main>
      <footer>{footer}</footer>
    </>
  );
}
```

```
function App() {
  return (
    <Layout
      header={<h1>Meu Site</h1>}
      footer={<p>&copy; 2025</p>}
    >
      <p>Bem-vindo(a) à minha página!</p>
    </Layout>
  );
}
```

✅ Agora você tem um **layout flexível e modular**.



Resumo

- ✓ Use `props` para parametrizar comportamento e visual
 - ✓ Use `props.children` para **encaixar conteúdo**
 - ✓ Pense em **composição**, não herança
 - ✓ Reutilize componentes com estilos e lógicas diferentes
-



Desafio prático:

Crie um componente `PainelComTitulo` que recebe:

- Um `titulo` (prop)
 - Um conteúdo com `children`
 - Um botão “Fechar” que some com o painel
-



Próximo capítulo:

Capítulo 14 – Formulários controlados com `useState`: inputs, validações e manipulação.



Capítulo 14 – Formulários Controlados em React



Você vai aprender:

- Como usar o `useState` para inputs
 - Como lidar com **inputs de texto, checkbox e selects**
 - Como **validar e tratar envios**
 - O que são **eventos controlados** no React
-



14.1 – Estado controlado: o que é isso?

Em React, campos de formulário são **controlados pelo estado (`useState`)**, e não diretamente pelo DOM.

- ✓ Isso significa que o valor de um input está **sincronizado** com uma variável do React.
-



Exemplo: Campo de texto controlado

```
import { useState } from "react";
```

```
function FormularioNome() {
  const [nome, setNome] = useState("");

  function handleChange(e) {
    setNome(e.target.value);
  }

  return (
    <div>
      <label>Digite seu nome:</label>
      <input value={nome} onChange={handleChange} />
      <p>Olá, {nome}!</p>
    </div>
  );
}
```

✅ O valor digitado vai para `nome`, que re-renderiza a interface.

✅ 14.2 – Inputs checkbox e select

```
function Preferencias() {
  const [aceita, setAceita] = useState(false);
  const [cor, setCor] = useState("azul");

  return (
    <form>
      <label>
        <input
          type="checkbox"
          checked={aceita}
          onChange={(e) => setAceita(e.target.checked)}
        />
        Aceito os termos
      </label>

      <select value={cor} onChange={(e) => setCor(e.target.value)}>
        <option value="azul">Azul</option>
        <option value="verde">Verde</option>
        <option value="vermelho">Vermelho</option>
      </select>

      <p>Cor escolhida: {cor}</p>
    </form>
  );
}
```

🛑 14.3 – Prevenindo o envio padrão

React permite interceptar o envio do formulário para tratar os dados:

```
function EnviarFormulario() {
  const [email, setEmail] = useState("");

  function handleSubmit(e) {
    e.preventDefault(); // ⚠️ evita que a página recarregue
    alert(`Email enviado: ${email}`);
  }
}
```

```
}  
  
return (  
  <form onSubmit={handleSubmit}>  
    <input  
      type="email"  
      value={email}  
      onChange={(e) => setEmail(e.target.value)}  
      placeholder="Digite seu email"  
    />  
    <button type="submit">Enviar</button>  
  </form>  
)  
};  
}
```



14.4 – Dicas e boas práticas

- ✓ Sempre use `value` + `onChange` para inputs controlados
 - ✓ Prefira `e.target.value` ou `e.target.checked`
 - ✓ Trate erros e feedbacks ao usuário
 - ✓ Crie funções reutilizáveis para validação
-



Resumo

- `useState` controla os campos de formulário
 - Inputs de texto, checkbox e select precisam de `value` ou `checked`
 - Use `onSubmit` para interceptar e processar formulários
-



Desafio prático:

Crie um formulário com:

- Campo de nome (texto)
 - Checkbox “Deseja receber promoções?”
 - Select de gênero (masculino/feminino/outro)
 - Botão enviar que mostra os dados com `alert`
-



Próximo capítulo:

Capítulo 15 – Comunicação entre componentes: lifting state e callbacks como props.

Capítulo 15 – Comunicação Entre Componentes ↔


Você vai aprender:

- Como um componente **filho** envia dados para o **pai**
 - O que é o "**lifting state up**"
 - Como passar funções como props
 - Como compartilhar estado entre componentes irmãos
-

15.1 – O problema da separação

Imagine que você tem dois componentes:

- `CampoDeTexto`: recebe o nome
- `Mensagem`: exibe o nome

Mas... o estado está separado! 

Isso dá problema:

Cada componente tem seu próprio `useState`, e eles **não compartilham os dados**.

15.2 – Lifting State Up (Subir o estado)

A solução é **mover o estado para o componente pai**, e passar os dados por props.

 Isso se chama **lifting state up**.

Exemplo Prático:

```
// Componente Pai
function Saudacao() {
  const [nome, setNome] = useState("");

  return (
    <div>
      <CampoDeTexto nome={nome} setNome={setNome} />
      <Mensagem nome={nome} />
    </div>
  );
}
```

```
// Componente Filho
function CampoDeTexto({ nome, setNome }) {
  return (
    <input
      value={nome}
      onChange={(e) => setNome(e.target.value)}
      placeholder="Digite seu nome"
    />
  );
}

// Componente Filho
function Mensagem({ nome }) {
  return <p>Bem-vindo(a), {nome}!</p>;
}
```


15.3 – Callback como prop

Você pode passar uma função para o filho e chamá-la com um valor:

```
function Pai() {
  function lidarComClique(valor) {
    alert(`Você clicou em: ${valor}`);
  }

  return <Filho aoClicar={lidarComClique} />;
}

function Filho({ aoClicar }) {
  return (
    <button onClick={() => aoClicar("botão do filho")}>
      Clique em mim
    </button>
  );
}
```


 O filho **não precisa saber o que acontece, só informa o pai.**

15.4 – Compartilhando estado entre irmãos

O padrão é:

1. O estado vai para o componente pai
 2. Os dados fluem para os dois irmãos por props
-

Dica Prática

 Nunca duplique estado! Isso causa inconsistências.

 Centralize o estado no menor **ancestral comum** entre os componentes que precisam compartilhar dados.



Resumo

- `useState` deve ser **centralizado** no componente mais alto possível
 - Dados fluem **de cima para baixo (top-down)** via props
 - Ações fluem **de baixo para cima (bottom-up)** via funções passadas como props
-



Desafio prático:

Crie três componentes:

- `FormularioEmail`: campo para digitar o email
- `BotaoEnviar`: envia o email para o pai
- `MensagemFinal`: exibe o email enviado

Use `useState` no componente pai para controlar e distribuir os dados.



Próximo capítulo:

Capítulo 16 – Refs em React: acessando elementos diretamente.



Capítulo 16 – Refs em React: Acessando Elementos Diretamente 🔍 ⚙️



Você vai aprender:

- O que são **refs**
 - Quando usar refs em vez de state
 - Como usar `useRef` para acessar elementos DOM
 - Como armazenar valores mutáveis sem causar re-renderização
-



16.1 – O que é uma ref?

Uma **ref** é uma referência a um **elemento DOM** ou a **qualquer valor mutável** que você não quer que cause um re-render.



Com o hook `useRef()` você pode:

- Acessar um elemento diretamente (ex: focar um input)

- Guardar valores entre renderizações sem perdê-los
-

Importante

Você **não deve usar refs** para **substituir o estado (state)**. Use refs quando:

- Você precisa **acessar diretamente** um elemento
 - Você quer **armazenar valores mutáveis** que **não afetam a renderização**
-

16.2 – Acessando um input com ref

```
import { useRef } from "react";

function FocoNoInput() {
  const inputRef = useRef(null);

  function focarInput() {
    inputRef.current.focus(); // ⚡ DOM direto!
  }

  return (
    <>
      <input ref={inputRef} type="text" placeholder="Digite algo" />
      <button onClick={focarInput}>Focar no input</button>
    </>
  );
}
```

✅ `inputRef.current` aponta para o **elemento HTML real**.


16.3 – Guardando valores sem re-render

```
import { useRef, useState } from "react";

function ContadorSemReRender() {
  const contador = useRef(0);
  const [atual, setAtual] = useState(0);

  function incrementar() {
    contador.current++;
    console.log("Valor real: ", contador.current);
  }

  return (
    <>
      <p>Estado: {atual}</p>
      <button onClick={incrementar}>Incrementar sem re-render</button>
      <button onClick={() => setAtual(atual + 1)}>Forçar render</button>
    </>
  );
}
```

 **useRef não causa re-render.** Perfeito para contadores internos, tempo, ou variáveis de controle.

16.4 – Quando usar useRef

Situação	useState	useRef
Precisa de re-render	✓ Sim	✗ Não
Apenas guardar valor temporário	✗ Não	✓ Sim
Acessar DOM diretamente	✗ Não	✓ Sim

Dica Profissional

Use **useRef** para:

- Focar ou selecionar elementos
 - Controlar timers (`setInterval`)
 - Evitar re-render com valores de controle interno
 - Guardar o valor anterior de uma variável
-

Desafio prático

Crie um componente `TimerManual` com:

- Um botão que inicia um timer (com `setInterval`)
 - Um botão que para o timer
 - Um **useRef** para guardar o ID do timer
-

Próximo capítulo:

Capítulo 17 – Renderização Condicional: `if`, ternário e `&&`

Capítulo 17 – Renderização Condicional em React

Você vai aprender:

- Como mostrar componentes de forma condicional

- Usar `if`, operador ternário e `&&` no JSX
 - Técnicas para evitar código desorganizado
-

17.1 – Renderização condicional: o conceito

Assim como no JavaScript tradicional, no React também decidimos **quando** mostrar algo dependendo de uma **condição**.

No JSX, temos três formas principais de fazer isso:

17.2 – Usando `if` (fora do JSX)

```
function Saudacao({ logado }) {  
  if (logado) {  
    return <h2>Bem-vindo de volta! 🙌</h2>;  
  }  
  
  return <h2>Por favor, faça login 🔒</h2>;  
}
```

📌 Forma mais legível. Ideal para **escolhas complexas**.

17.3 – Operador ternário `?` :

```
function Mensagem({ online }) {  
  return (  
    <p>  
      Status: {online ? "🟢 Online" : "🔴 Offline"}  
    </p>  
  );  
}
```

⚠️ Cuidado com ternários aninhados – eles dificultam a leitura!

17.4 – Operador lógico `&&`

```
function BoasVindas({ logado }) {  
  return (  
    <>  
      <h1>Olá visitante!</h1>  
      {logado && <p>Você está logado ✅</p>}  
    </>  
  );  
}
```

📌 Perfeito para mostrar **algo apenas se uma condição for verdadeira**.

! 17.5 – Evite isso!

```
{condicao
  ? condicao2
  ? <Comp1 />
  : <Comp2 />
  : <Comp3 />}
```

🧑 Muito confuso! Prefira separar a lógica em variáveis:

```
let conteudo;

if (condicao) {
  if (condicao2) {
    conteudo = <Comp1 />;
  } else {
    conteudo = <Comp2 />;
  }
} else {
  conteudo = <Comp3 />;
}
```

✓ 17.6 – Renderizando listas com .map()

```
const tarefas = ["Estudar React", "Ler documentação", "Criar projeto"];
```

```
function ListaDeTarefas() {
  return (
    <ul>
      {tarefas.map((tarefa, i) => (
        <li key={i}>📌 {tarefa}</li>
      ))}
    </ul>
  );
}
```

🔑 A **chave (key)** única evita problemas de performance e bugs na reconciliação.

🎓 Revisão do Capítulo

Técnica	Quando usar
if	Para lógica mais clara e separada do JSX
? :	Para expressões simples e inline
&&	Mostrar algo se a condição for verdadeira

🧪 Desafio prático

Crie um componente que:

- Mostra "Boa noite 🌙" se `horaAtual >= 18`
- Senão mostra "Bom dia ☀️"

- Use `new Date().getHours()` para pegar a hora
-



BIBLIOGRAFIA UTILIZADA

- **Documentação Oficial React**
<https://pt-br.react.dev/learn>
 - **MDN Web Docs – JavaScript**
<https://developer.mozilla.org/pt-BR/docs/Web/JavaScript>
 - **Vite – Build tool**
<https://vitejs.dev>
 - **StackOverflow e GitHub Discussions**
Comunidade ativa com dúvidas reais de iniciantes
 - **Canal do YouTube – Dev em Dobro, Rocketseat e Curso em Vídeo**
Vídeos que reforçam o conteúdo prático de forma visual
-