REACT JS PARA INICIANTES ABSOLUTOS

React

CRIE INTERFACES MODERNAS

Capítulo 1

Introdução ao React e Configuração do Ambiente

Introdução ao React e Configuração do Ambiente

O que é o React?

O **React** é uma biblioteca **JavaScript** criada pelo Facebook em 2013 para construir interfaces de usuário. Ele foi projetado para resolver problemas de desempenho e complexidade em aplicações web dinâmicas. Diferente de frameworks como Angular e Vue, o React foca na criação de componentes reutilizáveis. Componentes são blocos de construção que podem ser combinados para criar interfaces complexas.

- **Componentização:** React permite dividir a interface em componentes menores, que são mais fáceis de manter e reutilizar.
- **Reatividade:** Sempre que um dado ou um estado muda, o React atualiza automaticamente a interface, tornando-a sempre sincronizada.
- **Virtual DOM:** Para melhorar a performance, o React usa uma representação em memória do DOM real. Isso significa que ele calcula as mudanças necessárias e atualiza o DOM de forma eficiente, o que deixa as aplicações mais rápidas.

Introdução ao React e Configuração do Ambiente

Instalando o Node.js e o npm

Para usar o **React,** você precisa instalar o **Node.js** e o **npm** (Node Package Manager). O Node.js é uma plataforma para rodar JavaScript fora do navegador, e o npm é o sistema de gerenciamento de pacotes do Node, que facilita a instalação e atualização de bibliotecas, incluindo o React.

Passo a Passo:

Baixe e instale o Node.js no site oficial: https://nodejs.org.

Após a instalação, abra o terminal (ou o prompt de comando) e verifique se o Node e o npm foram instalados com sucesso:

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

Esses comandos devem exibir a versão instalada de cada ferramenta.

Introdução ao React e Configuração do Ambiente

Primeiros passos com JSX

JSX (JavaScript XML) é uma extensão de sintaxe que permite escrever **HTML** dentro do JavaScript. No React, os componentes retornam **JSX** para definir o layout da interface. Vejamos um exemplo:

```bash
// Arquivo: App.js
import React from 'react';

function App() {
  return (
    <div>
      <h1>Olá, React!</h1>
      <p>Estamos começando nossa jornada com o React.</p>
    </div>
  );
}

export default App;
```

Neste exemplo:

- Criamos um **componente App** usando uma função.
- **O componente** retorna uma estrutura **JSX**, que se parece com **HTML,** mas é realmente JavaScript.
- **export default App;** torna o **componente App** disponível para ser usado em outros arquivos.

Para que o **componente App** seja exibido na página, precisamos renderizá-lo. Isso acontece no arquivo **index.js:**

```bash
// Arquivo: index.js
import React from 'react';
import ReactDOM from 'react-dom/client';
import App from './App';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<App />);
```

- **ReactDOM.createRoot s**eleciona o elemento **HTML** com o **id root** e prepara o **React** para renderizar dentro dele.
- **root.render(<App />);** exibe o **componente** App na tela.

Introdução ao React e Configuração do Ambiente

Recapitulando:

Neste primeiro capítulo, aprendemos:

- O que é React e por que ele é usado para criar interfaces dinâmicas e performáticas**.**
- Como instalar o **Node.js** e configurar o ambiente com **o Create React App.**
- A estrutura básica de um projeto React e como renderizar o **componente** principal.

Você agora tem o ambiente necessário para criar seus primeiros componentes e interfaces! No próximo capítulo, vamos aprofundar em como construir componentes e explorar a sintaxe **JSX.**

Capítulo 2

Componentes e JSX - Construindo Blocos de Interface no React

Componentes e JSX - Construindo Blocos de Interface no React

O que são Componentes no React?

No React, componentes são como os blocos de construção das interfaces. Eles são peças independentes e reutilizáveis, responsáveis por uma parte específica da interface. Imagine que você está criando uma página de perfil: o cabeçalho, a foto de perfil e a seção de posts podem ser componentes separados, cada um com seu próprio código e funcionalidades. Isso torna o desenvolvimento mais organizado e permite reaproveitar esses componentes em outras partes do projeto.

Entender componentes e **JSX** é o primeiro passo para criar interfaces reativas e organizadas.

Criando seu Primeiro Componente

- vamos criar um **componente** simples que exiba uma mensagem de boas-vindas.

```
// Arquivo: WelcomeMessage.js
import React from 'react';

function WelcomeMessage() {
  return (
    <h1>Bem-vindo ao meu primeiro componente em React!</h1>
  );
}

export default WelcomeMessage;
```

Neste exemplo:

- Usamos uma função chamada **WelcomeMessage** para criar o **componente.**
- Esse **componente** retorna um elemento **<h1>** com o texto da nossa mensagem.
- No final, exportamos o **componente** para usá-lo em outros arquivos.

Componentes e JSX - Construindo Blocos de Interface no React

Entendendo o JSX

JSX é a sintaxe que permite escrever **HTML** dentro do **JavaScript,** algo único no React. No exemplo acima, usamos o **JSX** para retornar o **<h1>** dentro da função. O **JSX** é parecido com **HTML**, mas possui algumas diferenças. Por exemplo:

- Para definir classes, usamos **className** em vez de **class.**
- Todo código **JSX** precisa estar dentro de uma única tag pai.

Renderizando Componentes no React

Agora que temos o **componente WelcomeMessage,** vamos renderizá-lo na nossa aplicação principal.

```
// Arquivo: App.js
import React from 'react';
import WelcomeMessage from './WelcomeMessage';

function App() {
  return (
    <div>
      <WelcomeMessage />
    </div>
  );
}

export default App;
```

Aqui, importamos o **componente WelcomeMessage** e o utilizamos dentro do componente principal App. No React, você pode pensar nos componentes como se fossem tags **HTML** personalizadas:
<WelcomeMessage /> é uma tag que agora faz parte da sua biblioteca de componentes.

Componentes e JSX - Construindo Blocos de Interface no React

Criando Componentes com Parâmetros: Props

As **props** são a forma de passar dados para um componente. Vamos modificar nosso componente para exibir o nome do usuário.

Renderizando Componentes no React

Agora que temos o componente WelcomeMessage, vamos renderizá-lo na nossa aplicação principal.

```
// Arquivo: WelcomeMessage.js
import React from 'react';

function WelcomeMessage(props) {
  return (
    <h1>Bem-vindo, {props.nome}!</h1>
  );
}

export default WelcomeMessage;
```

Ao usar **{props.nome},** fazemos com que o **componente** exiba o nome que passamos para ele como propriedade. Para usá-lo:

```
// Arquivo: App.js
import React from 'react';
import WelcomeMessage from './WelcomeMessage';

function App() {
  return (
    <div>
      <WelcomeMessage nome="João" />
    </div>
  );
}

export default App;
```

Agora, ao renderizar **WelcomeMessage** e passar a **prop nome="João"**, o componente exibirá "Bem-vindo, João!".

Componentes e JSX - Construindo Blocos de Interface no React

Recapitulando:

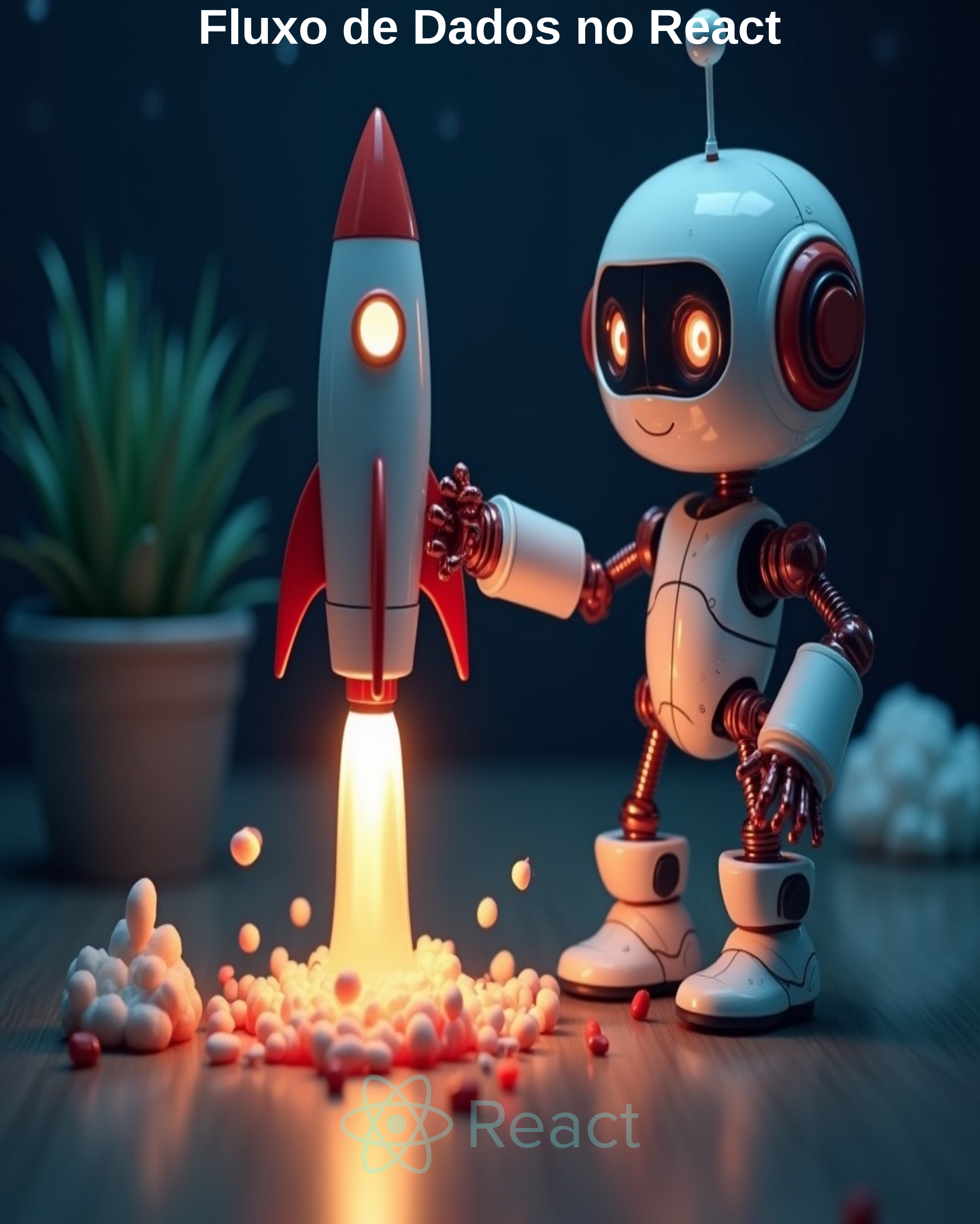
Neste primeiro capítulo, aprendemoscomo montar a base para um componente React e o poder do **JSX** para construir interfaces.

Com esses passos, você pode criar componentes reutilizáveis e organizados para formar a base da sua interface. Dividir a interface em componentes menores e reutilizáveis facilita a manutenção do código e permite expandir a aplicação rapidamente.

Nos próximos capítulos, vamos explorar como adicionar ainda mais funcionalidades a esses componentes, trazendo dinamismo e interatividade!

Capítulo 3

Estado e Propriedades - Controlando o Fluxo de Dados no React



Estado e Propriedades - Controlando o Fluxo de Dados no React

Neste capítulo, vamos abordar um dos conceitos mais importantes no React: o estado e as propriedades (props). Entender como eles funcionam é essencial para criar componentes interativos e que respondam dinamicamente aos dados da sua aplicação.

Propriedades (Props) - Passando Informações para Componentes

As propriedades, ou props, são dados que você passa de um componente "pai" para um componente "filho". Imagine que você tem um componente de botão que precisa mostrar diferentes rótulos, como “Enviar”, “Cancelar” ou “Excluir”. Com props, você pode personalizar o botão conforme a necessidade de cada página, sem precisar recriar o componente.

Vamos ver isso em ação:

```jsx
// Arquivo: Botao.js
import React from 'react';

function Botao(props) {
  return (
    <button>{props.label}</button>
  );
}

export default Botao;
```

Nesse exemplo:

- O **componente Botao** recebe uma prop chamada **label** e a exibe dentro do botão.
- O valor da prop será definido pelo **componente** que chamar **Botao**.

Estado e Propriedades - Controlando o Fluxo de Dados no React

Para usar o componente **Botao** com diferentes rótulos:

```jsx
// Arquivo: App.js
import React from 'react';
import Botao from './Botao';

function App() {
  return (
    <div>
      <Botao label="Enviar" />
      <Botao label="Cancelar" />
    </div>
  );
}

export default App;
```

Aqui**, Botao** vai exibir “Enviar” no primeiro botão e “Cancelar” no segundo. Isso mostra como as props permitem controlar o conteúdo dos componentes de fora para dentro.

Estado (State) - Controlando Dados Internos do Componente

O estado é um conjunto de dados que o próprio componente controla internamente e que pode mudar ao longo do tempo, permitindo que a interface se adapte às ações do usuário. Diferente das props, o estado é gerenciado dentro do componente e não pode ser diretamente alterado de fora.

Vamos ver como o estado funciona com um exemplo simples:

```jsx
// Arquivo: Contador.js
import React, { useState } from 'react';

function Contador() {
  const [contagem, setContagem] = useState(0);

  return (
    <div>
      <p>Contagem: {contagem}</p>
      <button onClick={() => setContagem(contagem + 1)}>Aumentar</button>
    </div>
  );
}

export default Contador;
```

Estado e Propriedades - Controlando o Fluxo de Dados no React

Neste exemplo:

Usamos a função **useState** para criar o estado **contagem**, que inicia com valor **0.**
setContagem é a função que usamos para atualizar o valor de **contagem.**
Quando o botão é clicado**, setContagem** aumenta o valor da **contagem** em **1.**
Isso faz com que o **componente** seja **reativo.** Sempre que o estado muda, o React renderiza novamente o **componente,** mostrando o valor atualizado.

Diferença entre Props e Estado

- **Props** são imutáveis, ou seja, o componente não pode alterá-las. Elas são definidas pelo componente pai.
- **Estado** é mutável e controlado dentro do próprio componente, permitindo que ele responda a eventos e ações do usuário.

Imagine que você está criando uma lista de tarefas. A lista em si pode ser passada como props para um componente "Tarefas", mas o estado seria responsável por monitorar mudanças, como quando uma tarefa é marcada como concluída.

Estado e Propriedades - Controlando o Fluxo de Dados no React

Passando Estado como Props para Componentes Filhos

Em algumas situações, você precisará compartilhar o estado de um componente com seus "filhos". Vamos ver como fazer isso usando o exemplo de um contador.

```
// Arquivo: App.js
import React, { useState } from 'react';
import Contador from './Contador';

function App() {
  const [valor, setValor] = useState(0);

  return (
    <div>
      <Contador valor={valor} incrementar={() => setValor(valor + 1)} />
    </div>
  );
}

export default App;
```

Assim, **Contador** depende das props para exibir a contagem e incrementar o valor. Esse é um padrão comum no React para separar a lógica de controle e a exibição.

Estado e Propriedades - Controlando o Fluxo de Dados no React

Recapitulando o Fluxo de Dados com Estado e Props

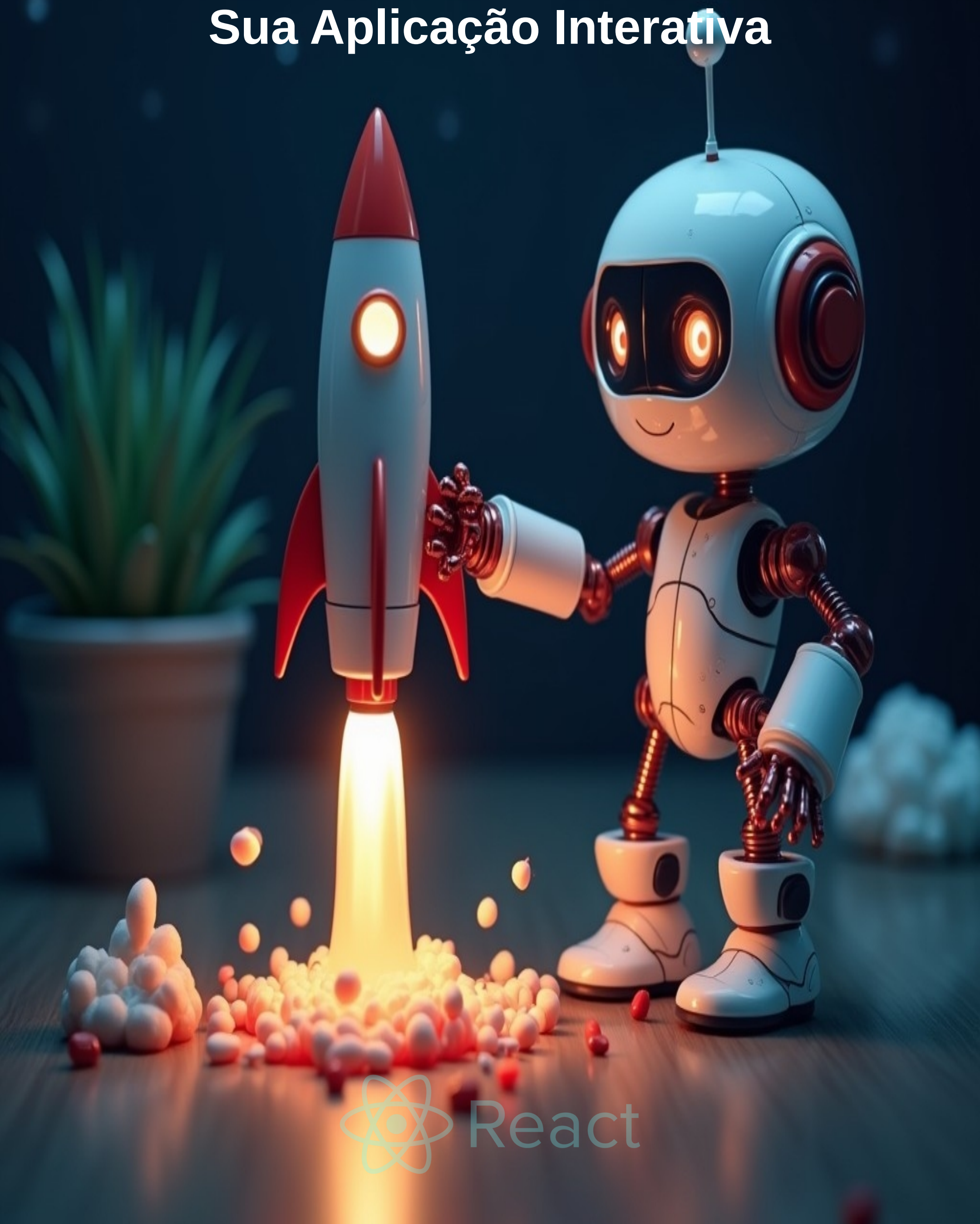
- Props são passadas do componente pai para o filho, permitindo que o componente exiba dados e chame funções definidas fora dele.
- Estado é gerenciado dentro do componente e pode mudar dinamicamente, acionando re-renderizações automáticas.

Essa estrutura cria um fluxo de dados previsível e bem organizado, onde as props fluem para baixo e o estado é controlado localmente, ou, em alguns casos, por componentes pais. Essa abordagem modular e organizada é o que torna o React uma ferramenta poderosa para criar interfaces dinâmicas e reativas.

Neste capítulo, você aprendeu como o React gerencia dados e eventos dentro dos componentes. Nos próximos capítulos, vamos expandir esses conceitos, ensinando como lidar com eventos de usuário e conectar sua aplicação ao mundo exterior com APIs.

Capítulo 4

Manipulação de Eventos: Tornando Sua Aplicação Interativa



Manipulação de Eventos - Tornando Sua Aplicação Interativa

Neste capítulo, vamos transformar nossos componentes em partes interativas da interface, permitindo que respondam a ações do usuário. No React, a manipulação de eventos é uma das formas mais diretas de tornar a experiência do usuário dinâmica e envolvente.

O que São Eventos no React?
Os eventos são ações que ocorrem quando o usuário interage com a interface — como cliques, digitação ou movimentação do cursor. No React, podemos usar eventos para definir como a interface deve reagir a essas ações. Por exemplo, imagine uma lista de tarefas onde o usuário pode clicar para marcar uma tarefa como concluída: esse clique é um evento que desencadeia uma mudança na interfac
Pense em um evento como o interruptor de luz de um cômodo. O interruptor (evento) responde à ação do usuário (ligar ou desligar), alterando o estado do cômodo (escuro ou iluminado). Da mesma forma, eventos em React respondem às ações dos usuários, mudando o estado ou a aparência de um componente.

Estado e Propriedades - Controlando o Fluxo de Dados no React

Como Adicionar Eventos no React?

Adicionar eventos em React é muito parecido com o que fazemos com JavaScript tradicional, mas usando a sintaxe **JSX**. Vamos explorar o evento de clique, um dos mais usados.

```jsx
// Arquivo: Botao.js
import React from 'react';

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

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

export default Botao;
```

Neste exemplo:

- Criamos uma função **handleClick** que será executada sempre que o botão for clicado.
- O evento **onClick** é associado ao botão, e chamará a função **handleClick** sempre que o usuário clicar.

No React, eventos como **onClick**, **onChange**, **onMouseEnter** seguem o padrão camelCase, diferente do JavaScript HTML padrão (que usa letras minúsculas, como **onclick**).

Estado e Propriedades - Controlando o Fluxo de Dados no React

Atualizando o Estado com Eventos

Agora que sabemos como associar um evento a um componente, vamos usá-lo para atualizar o estado. Digamos que queremos criar um botão que conte o número de cliques. Para isso, vamos combinar o evento **onClick** com o estado **useState**.

```
// Arquivo: Contador.js
import React, { useState } from 'react';

function Contador() {
  const [contagem, setContagem] = useState(0);

  const handleClick = () => {
    setContagem(contagem + 1);
  };

  return (
    <div>
      <p>Você clicou {contagem} vezes</p>
      <button onClick={handleClick}>Clique para contar</button>
    </div>
  );
}

export default Contador;
```

Neste exemplo:

- **useState** define a variável **contagem** com um valor inicial de **0**.
- **setContagem** é a função que atualiza **contagem**.
- Quando **handleClick** é acionado pelo **onClick**, **setContagem** aumenta a contagem, e o React renderiza novamente o componente, exibindo a contagem atualizada.

Isso mostra como podemos usar eventos para mudar o estado e, automaticamente, atualizar a interface para o usuário.

Estado e Propriedades - Controlando o Fluxo de Dados no React

Manipulando Eventos de Entrada do Usuário

Outro uso importante dos eventos é em campos de formulário, onde os eventos de entrada, como **onChange**, capturam o que o usuário digita.

Vamos criar um exemplo onde exibimos o nome do usuário enquanto ele digita:

```
// Arquivo: Saudacao.js
import React, { useState } from 'react';

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

  const handleChange = (event) => {
    setNome(event.target.value);
  };

  return (
    <div>
      <input type="text" placeholder="Digite seu nome" onChange={handleChange} />
      <p>Olá, {nome}!</p>
    </div>
  );
}

export default Saudacao;
```

Aqui:

- **handleChange** atualiza o estado **nome** toda vez que o usuário digita algo
- **event.target.value** representa o valor atual do campo de texto.
- A interface mostra o nome conforme ele é digitado, criando uma experiência interativa e em tempo real para o usuário.

Estado e Propriedades - Controlando o Fluxo de Dados no React

Passando Funções como Props para Manipular Eventos

Quando você precisa que um evento em um componente "filho" altere o estado em um componente "pai", pode passar uma função como prop.

Imagine que você quer um botão "Like" para aumentar um contador que está no componente "pai".

```jsx
// Arquivo: App.js
import React, { useState } from 'react';
import BotaoLike from './BotaoLike';

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

  const handleLike = () => {
    setLikes(likes + 1);
  };

  return (
    <div>
      <p>Total de likes: {likes}</p>
      <BotaoLike onLike={handleLike} />
    </div>
  );
}

export default App;
```

Agora, no componente **BotaoLike**:

```jsx
// Arquivo: BotaoLike.js
import React from 'react';

function BotaoLike(props) {
  return (
    <button onClick={props.onLike}>Curtir</button>
  );
}

export default BotaoLike;
```

Neste exemplo:

- **App** controla o estado **likes** e passa a função **handleLike** como prop para **BotaoLike**.
- Quando **BotaoLike** é clicado, ele executa **props.onLike**, chamando **handleLike** no componente **pai**, **App**, e aumentando a contagem de likes.

Essa abordagem permite separar a lógica de controle (no componente "pai") da exibição do botão (no componente "filho").

Estado e Propriedades - Controlando o Fluxo de Dados no React

Recapitulando:

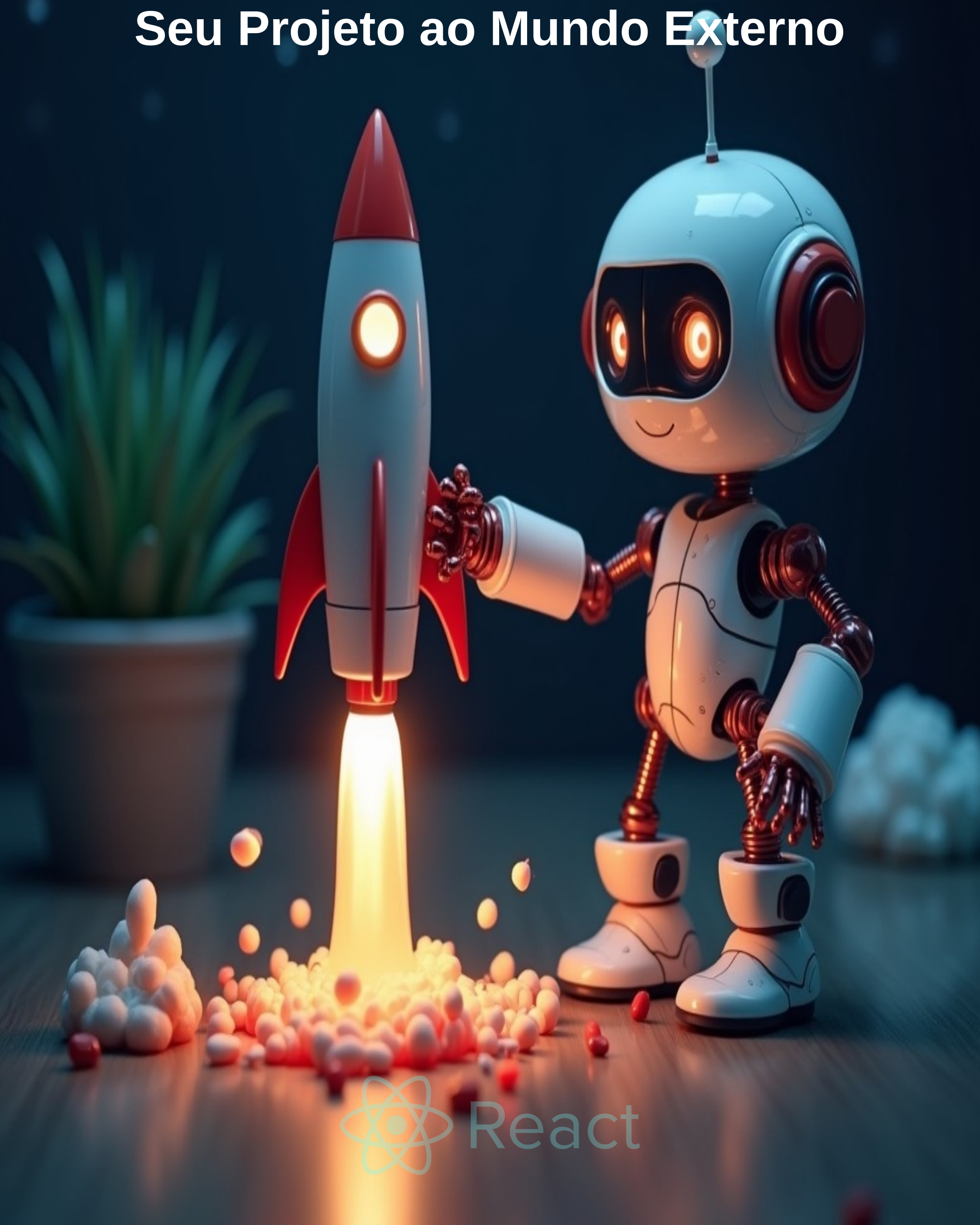
Neste capítulo, você aprendeu a:

- Associar eventos a componentes para tornar sua interface interativa.
- Atualizar o estado com eventos como onClick e onChange.
- Usar funções passadas como props para manipular o estado entre componentes.

Eventos em React tornam a interface ativa e pronta para reagir às interações dos usuários. Agora você está pronto para usar esses conceitos para construir interfaces dinâmicas que respondem diretamente ao usuário!

Capítulo 5

Trabalhando com APIs: Conectando Seu Projeto ao Mundo Externo



Consumo de APIs - Conectando Sua Aplicação ao Mundo Externo

Agora que você já conhece a estrutura básica do React, sabe manipular eventos e gerenciar o estado, está pronto para dar um grande passo: conectar sua aplicação a uma API. Imagine o React como um veículo bem montado, pronto para cruzar várias rotas de desenvolvimento. Mas, para torná-lo verdadeiramente útil e dinâmico, ele precisa de combustível – e, nesse caso, o "combustível" são os dados externos fornecidos por APIs.

O que é uma API e Por que Usá-la?
Uma **API** (Interface de Programação de Aplicações) é uma ponte que permite que sua aplicação interaja com outros sistemas, trocando dados em tempo real. As APIs são responsáveis por trazer informações para sua aplicação, como dados de clima, perfis de redes sociais, cotações de mercado, e muito mais.

Analogia: Pense na API como um garçom que leva seu pedido (solicitação) para a cozinha (servidor) e traz a comida (dados) de volta para sua mesa (aplicação). Esse garçom facilita a comunicação entre você e a cozinha, assim como a API facilita o diálogo entre sua aplicação e a internet.

Consumo de APIs - Conectando Sua Aplicação ao Mundo Externo

Conectando-se a uma API com Fetch

Uma das formas mais comuns de consumir APIs em JavaScript e React é usando a função **fetch**, que permite fazer requisições HTTP. Vamos ver um exemplo prático onde nossa aplicação acessa uma API de piadas para exibir uma nova piada cada vez que o usuário clicar em um botão.

```jsx
// Arquivo: Piada.js
import React, { useState } from 'react';

function Piada() {
  const [piada, setPiada] = useState('');

  const buscarPiada = async () => {
    const resposta = await fetch('https://api.chucknorris.io/jokes/random');
    const dados = await resposta.json();
    setPiada(dados.value);
  };

  return (
    <div>
      <p>{piada}</p>
      <button onClick={buscarPiada}>Obter Piada</button>
    </div>
  );
}

export default Piada;
```

Neste exemplo:

- **buscarPiada** é uma função assíncrona que faz uma requisição HTTP para a API de piadas do Chuck Norris usando **fetch**.
- A função espera a resposta, converte para JSON, e então atualiza o estado **piada** com a nova piada obtida.
- Cada vez que o botão é clicado, a função **buscarPiada** é chamada, atualizando a piada exibida na tela.

Consumo de APIs - Conectando Sua Aplicação ao Mundo Externo

Usando useEffect para Carregar Dados Inicialmente

O React permite carregar dados automaticamente quando o componente é montado usando o useEffect. Isso é útil para que as informações da API sejam exibidas imediatamente quando o usuário abre a página.

Vamos modificar o exemplo acima para carregar uma piada assim que o componente é exibido.

```jsx
// Arquivo: Piada.js
import React, { useState, useEffect } from 'react';

function Piada() {
  const [piada, setPiada] = useState('');

  useEffect(() => {
    const buscarPiada = async () => {
      const resposta = await fetch('https://api.chucknorris.io/jokes/random');
      const dados = await resposta.json();
      setPiada(dados.value);
    };
    buscarPiada();
  }, []);

  return (
    <div>
      <p>{piada}</p>
      <button onClick={() => buscarPiada()}>Nova Piada</button>
    </div>
  );
}

export default Piada;
```

Neste exemplo:

- O **useEffect** chama **buscarPiada** quando o componente é montado, exibindo uma piada inicial.
- O array vazio **[]** ao final de **useEffect** indica que queremos que esse efeito seja executado apenas uma vez, na montagem do componente.

Esse padrão é muito usado para carregar dados de APIs quando o componente é carregado, especialmente em aplicações que consomem dados de mercado, redes sociais, entre outras fontes.

Consumo de APIs - Conectando Sua Aplicação ao Mundo Externo

Usando useEffect para Carregar Dados Inicialmente

O React permite carregar dados automaticamente quando o componente é montado usando o useEffect. Isso é útil para que as informações da API sejam exibidas imediatamente quando o usuário abre a página.

Vamos modificar o exemplo acima para carregar uma piada assim que o componente é exibido.

```jsx
// Arquivo: Piada.js
import React, { useState, useEffect } from 'react';

function Piada() {
  const [piada, setPiada] = useState('');

  useEffect(() => {
    const buscarPiada = async () => {
      const resposta = await fetch('https://api.chucknorris.io/jokes/random');
      const dados = await resposta.json();
      setPiada(dados.value);
    };
    buscarPiada();
  }, []);

  return (
    <div>
      <p>{piada}</p>
      <button onClick={() => buscarPiada()}>Nova Piada</button>
    </div>
  );
}

export default Piada;
```

Neste exemplo:

- O **useEffect** chama **buscarPiada** quando o componente é montado, exibindo uma piada inicial.
- O array vazio **[]** ao final de **useEffect** indica que queremos que esse efeito seja executado apenas uma vez, na montagem do componente.

Esse padrão é muito usado para carregar dados de APIs quando o componente é carregado, especialmente em aplicações que consomem dados de mercado, redes sociais, entre outras fontes.

Consumo de APIs - Conectando Sua Aplicação ao Mundo Externo

Como o Consumo de APIs Está Presente no Mercado

Conectar uma aplicação a uma API é uma habilidade essencial no desenvolvimento de software moderno. As empresas dependem de dados em tempo real para fornecer funcionalidades dinâmicas e adaptáveis. Por exemplo, você verá APIs em projetos de e-commerce, previsão de clima, viagens, notícias, redes sociais e tantas outras áreas. Saber trabalhar com APIs aumenta seu valor como desenvolvedor, permitindo que você crie aplicações mais robustas, interativas e dinâmicas.

Se você está começando agora, saiba que a habilidade de consumir APIs coloca você em um nível de desenvolvedor profissional muito requisitado. O mercado valoriza profissionais que entendem de manipulação de dados e conseguem integrar diferentes sistemas. Conectar o React a uma API é um dos primeiros passos para se destacar e abrir portas para oportunidades em diversas indústrias, de startups a grandes empresas.

Consumo de APIs - Conectando Sua Aplicação ao Mundo Externo

Lidando com Erros e Garantindo uma Experiência Suave ao Usuário

Consumir dados externos tem seus desafios: nem sempre a API responderá como esperado. Para garantir que o usuário tenha uma experiência agradável, é importante lidar com erros e estados de carregamento.

Aqui vai um exemplo para aprimorar nossa aplicação de piadas, incluindo uma mensagem de "Carregando..." enquanto aguardamos a resposta da API e um tratamento para caso ocorra um erro.

```jsx
// Arquivo: Piada.js
import React, { useState, useEffect } from 'react';

function Piada() {
  const [piada, setPiada] = useState('');
  const [carregando, setCarregando] = useState(false);
  const [erro, setErro] = useState(null);

  const buscarPiada = async () => {
    setCarregando(true);
    setErro(null);
    try {
      const resposta = await fetch('https://api.chucknorris.io/jokes/random');
      if (!resposta.ok) {
        throw new Error('Erro ao buscar piada');
      }
      const dados = await resposta.json();
      setPiada(dados.value);
    } catch (erro) {
      setErro(erro.message);
    } finally {
      setCarregando(false);
    }
  };

  useEffect(() => {
    buscarPiada();
  }, []);

  return (
    <div>
      {carregando && <p>Carregando...</p>}
      {erro ? <p>{erro}</p> : <p>{piada}</p>}
      <button onClick={buscarPiada}>Nova Piada</button>
    </div>
  );
}

export default Piada;
```

Neste exemplo:

- O estado carregando exibe uma mensagem "Carregando..." enquanto esperamos a resposta.
- O estado **erro** guarda qualquer erro que possa ocorrer na requisição.
- **try...catch** lida com possíveis falhas na requisição, e **finally** garante que o carregamento termine, independentemente do sucesso ou falha da requisição.

Esse tratamento de erros cria uma experiência mais amigável para o usuário, mesmo quando algo dá errado.

Consumo de APIs - Conectando Sua Aplicação ao Mundo Externo

Recapitulando:

Neste capítulo, você aprendeu a:

- Conectar o React a uma API e exibir dados externos.
- Usar **fetch** e **useEffect** para carregar dados iniciais e responder a eventos do usuário.
- Tratar erros e estados de carregamento, oferecendo uma experiência mais agradável ao usuário.

Parabéns! Com o conhecimento de consumo de APIs, você está preparado para criar aplicações React que se conectam ao mundo real. Esse é um dos passos mais importantes para desenvolver habilidades requisitadas e essenciais no mercado de trabalho.