Aula 23 – React: Estado e Ciclo de Vida com Hooks

No universo do desenvolvimento web moderno, criar interfaces interativas e dinâmicas é um desafio constante. Imagine construir um aplicativo onde nada muda, onde os dados são estáticos e a interação do usuário não gera qualquer resposta. Seria como um livro sem páginas para virar, ou um jogo onde seus comandos não afetam o personagem. Para que nossas aplicações "ganhem vida", precisamos de mecanismos que permitam aos componentes reagir a eventos, armazenar informações e se adaptar ao longo do tempo.

É exatamente essa a função do estado e do ciclo de vida em React. Eles são os pilares que transformam componentes estáticos em elementos dinâmicos e responsivos, capazes de gerenciar dados e interagir de forma inteligente com o usuário e com o restante da aplicação. Compreender esses conceitos é o que separa um desenvolvedor que apenas replica código de um que realmente entende como construir aplicações robustas e eficientes.

Nesta aula, embarcaremos em uma jornada para desvendar como o React gerencia essas mudanças. Exploraremos os Hooks useState e useEffect, ferramentas poderosas que simplificam a forma como lidamos com o estado e os efeitos colaterais em componentes funcionais. Ao final, você será capaz de criar componentes que armazenam e atualizam dados de forma eficaz, reagem a eventos externos e gerenciam seu próprio "ciclo de vida", garantindo que suas aplicações sejam não apenas funcionais, mas também performáticas e fáceis de manter. Prepare-se para dar um salto qualitativo na sua forma de desenvolver com React, construindo interfaces que realmente respondem e evoluem.

Desvendando o Estado: A Memória do seu Componente

Pense em um aplicativo de lista de tarefas. Quando você adiciona uma nova tarefa, marca uma como concluída ou filtra as pendentes, o aplicativo precisa "lembrar" dessas informações para exibir a interface correta. Essa capacidade de um componente de armazenar e gerenciar dados que podem mudar ao longo do tempo é o que chamamos de **estado**. Sem ele, seus componentes seriam como atores que esquecem suas falas a cada nova cena, incapazes de manter a coerência da história.

Historicamente, o estado era gerenciado principalmente em componentes de classe. No entanto, com a evolução do React e a introdução dos Hooks, a forma como lidamos com o estado em componentes funcionais foi revolucionada. Os Hooks nos permitem adicionar funcionalidades de estado e ciclo de vida a componentes funcionais, tornando-os tão poderosos quanto os de classe, mas com uma sintaxe mais limpa e intuitiva. Essa mudança simplificou muito o desenvolvimento, especialmente para quem está começando com React e utilizando ferramentas modernas como o Vite para um setup rápido.

O useState é o Hook fundamental para adicionar estado a componentes funcionais. Ele é como um pequeno caderno de anotações que seu componente carrega consigo, onde ele pode registrar informações importantes e consultá-las sempre que precisar. Quando uma anotação é alterada, o React automaticamente percebe e redesenha o componente para refletir a nova informação. Isso garante que a interface do usuário esteja sempre sincronizada com os dados mais recentes, sem que você precise manipular o DOM diretamente.

O Hook useState: Seu Primeiro Passo para o Estado

Para começar a usar o useState, você precisa importá-lo do React e chamá-lo dentro do seu componente funcional. A função useState retorna um par de valores: o estado atual e uma função para atualizá-lo. É uma convenção desestruturar esse par usando um array, o que torna o código mais legível e direto.

```
import React, { useState } from 'react';

function Contador() {
  // 'count' é o estado atual, 'setCount' é a função para atualizá-lo
  // O valor inicial do estado é 0
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>Você clicou {count} vezes</p>
      <button onClick={() => setCount(count + 1)}>
        Clique aqui
      </button>
    </div>
  );
}

export default Contador;
```

Neste exemplo, count armazena o número de cliques. Quando o botão é pressionado, setCount é chamada com um novo valor (count + 1), o que faz o componente Contador ser renderizado novamente com o valor atualizado. Isso demonstra a simplicidade e o poder do useState para gerenciar dados dinâmicos.

Gerenciando Múltiplos Estados e Atualizações Funcionais

À medida que seus componentes crescem em complexidade, é comum precisar gerenciar múltiplos pedaços de estado. O useState permite que você declare várias variáveis de estado independentes dentro do mesmo componente, cada uma com seu próprio valor e função de atualização. Isso é como ter vários cadernos de anotações, cada um dedicado a um tipo específico de informação, sem que um interfira no outro.

Por exemplo, em um formulário de login, você pode precisar de um estado para o nome de usuário, outro para a senha e talvez um terceiro para uma mensagem de erro. Cada um desses estados pode ser gerenciado de forma isolada, tornando o código mais modular e fácil de entender. A flexibilidade do useState é um dos motivos pelos quais ele se tornou a espinha dorsal para a gestão de dados em componentes funcionais.

Além disso, ao atualizar o estado, é importante considerar como o novo valor é derivado do valor anterior. Em situações onde a atualização do estado depende do estado anterior (como no nosso contador), é uma boa prática passar uma função para a função de atualização do estado (setCount). Essa função recebe o estado anterior como argumento e retorna o novo estado. Isso garante que você esteja sempre trabalhando com o valor mais recente do estado, especialmente em cenários onde múltiplas atualizações podem ocorrer rapidamente.

Atualizações Funcionais com useState

Imagine que você tem um contador e, por algum motivo, quer incrementá-lo duas vezes seguidas em um único evento. Se você simplesmente chamasse setCount(count + 1) duas vezes, o resultado não seria o esperado, pois ambas as chamadas veriam o mesmo valor count inicial. A solução é usar a forma funcional:

```
import React, { useState } from 'react';

function DuploContador() {
  const [count, setCount] = useState(0);

  const handleDuploIncremento = () => {
    // Primeira atualização usando o valor anterior (prevCount)
    setCount(prevCount => prevCount + 1);
    // Segunda atualização, garantindo que ela use o valor mais recente da primeira
    setCount(prevCount => prevCount + 1);
  };

  return (
    <div>
      <p>Contador: {count}</p>
      <button onClick={handleDuploIncremento}>
        Incrementar Duas Vezes
      </button>
    </div>
  );
}

export default DuploContador;
```

Esta abordagem é crucial para evitar bugs sutis relacionados a atualizações assíncronas do estado e garante a robustez da sua aplicação. É uma prática recomendada que se alinha com a filosofia de performance e previsibilidade do React.

Ciclo de Vida do Componente: A Jornada de um Elemento React

Assim como nós nascemos, crescemos e, eventualmente, partimos, um componente React também tem um "ciclo de vida". Ele é montado (aparece na tela), pode ser atualizado (seus dados ou propriedades mudam) e, finalmente, é desmontado (removido da tela). Entender essas fases é fundamental para saber quando executar certas operações, como buscar dados de uma API, configurar assinaturas de eventos ou limpar recursos.

Tradicionalmente, componentes de classe utilizavam métodos específicos para cada fase do ciclo de vida, como componentDidMount, componentDidUpdate e componentWillUnmount. Embora eficazes, esses métodos podiam levar a um código disperso, onde lógicas relacionadas (como a configuração e a limpeza de um evento) ficavam em métodos diferentes, dificultando a manutenção e a compreensão.

A introdução do Hook useEffect revolucionou a forma como lidamos com o ciclo de vida em componentes funcionais. Ele oferece uma maneira unificada e mais declarativa de lidar com "efeitos colaterais" – operações que não são diretamente parte da renderização, mas que interagem com o mundo exterior ou com o próprio componente após a renderização. Pense no useEffect como um "zelador" do seu componente, que cuida de tarefas importantes nos bastidores, garantindo que tudo esteja em ordem em cada fase da vida do componente.

O Hook useEffect: Gerenciando Efeitos Colaterais

O useEffect é uma função que recebe outra função como argumento. Essa função será executada após cada renderização do componente, a menos que você especifique o contrário. É o local ideal para realizar operações como:

- Busca de dados (data fetching)
- Manipulação direta do DOM (embora geralmente evitada)
- Configuração de assinaturas de eventos (timers, listeners)
- Sincronização com sistemas externos

```
import React, { useState, useEffect } from 'react';

function Relogio() {
  const [time, setTime] = useState(new Date());

  useEffect(() => {
    // Este efeito será executado após a montagem e a cada atualização
    const timerID = setInterval(() => setTime(new Date()), 1000);

    // A função de retorno (cleanup) será executada antes da desmontagem
    // ou antes de uma nova execução do efeito
    return () => {
      clearInterval(timerID);
    };
  }, []); // Array de dependências vazio: o efeito só roda na montagem e desmontagem

  return (
    <div>
      <p>Hora atual: {time.toLocaleTimeString()}</p>
    </div>
  );
}

export default Relogio;
```

Neste exemplo, o useEffect configura um timer que atualiza a hora a cada segundo. O array vazio [] como segundo argumento indica que este efeito só deve ser executado uma vez, após a montagem inicial do componente, e sua função de limpeza será executada apenas na desmontagem. Isso é análogo ao componentDidMount e componentWillUnmount combinados.

A Profundidade do useEffect: Dependências e Limpeza

A beleza do useEffect reside na sua flexibilidade, especialmente no controle de quando um efeito deve ser reexecutado. O segundo argumento do useEffect é um array de dependências. Ele funciona como uma lista de "vigilância": se qualquer valor nessa lista mudar entre as renderizações, o efeito será reexecutado. Se o array estiver vazio ([]), o efeito será executado apenas uma vez, após a montagem inicial do componente, e a função de limpeza (se houver) será executada na desmontagem. Se o array for omitido, o efeito será executado após *cada* renderização.

Essa capacidade de controlar a reexecução é crucial para otimizar a performance e evitar comportamentos indesejados. Por exemplo, se você está buscando dados de uma API com base em um ID de usuário, você quer que a busca ocorra apenas quando o ID do usuário mudar, não a cada renderização do componente. O array de dependências permite expressar essa lógica de forma clara e concisa.

A função de limpeza, retornada pelo useEffect, é igualmente importante. Ela é executada antes que o componente seja desmontado ou antes que o efeito seja reexecutado (se as dependências mudarem). Essa limpeza é vital para evitar vazamentos de memória e garantir que recursos externos (como timers, assinaturas de eventos ou conexões de rede) sejam liberados adequadamente. É como fechar a torneira depois de usar a água, ou desligar as luzes ao sair de um cômodo.

useEffect com Dependências e Limpeza em Ação

Considere um componente que exibe informações de um usuário, cujo ID pode mudar:

```
import React, { useState, useEffect } from 'react';

function PerfilUsuario({ userId }) {
  const [userData, setUserData] = useState(null);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    setLoading(true);
    // Simula uma chamada de API
    const fetchUser = async () => {
      try {
        const response = await new Promise(resolve =>
          setTimeout(() => {
            resolve({
              id: userId,
              name: `Usuário ${userId}`,
              email: `user${userId}@example.com`
            });
          }, 1000)
        );
        setUserData(response);
      } catch (error) {
        console.error("Erro ao buscar usuário:", error);
        setUserData(null);
      } finally {
        setLoading(false);
      }
    };

    fetchUser();

    // Função de limpeza: opcional, mas boa prática para cancelar requisições, etc.
    return () => {
      // Aqui você poderia cancelar a requisição anterior se ela ainda estivesse pendente
      // Ex: controller.abort();
      console.log(`Limpando efeito para userId: ${userId}`);
    };
  }, [userId]); // O efeito será reexecutado sempre que 'userId' mudar

  if (loading) return <p>Carregando perfil...</p>;
  if (!userData) return <p>Nenhum usuário encontrado.</p>;

  return (
    <div>
      <h3>Perfil de {userData.name}</h3>
      <p>Email: {userData.email}</p>
    </div>
  );
}

export default PerfilUsuario;
```

Neste exemplo, o efeito de busca de dados só é executado quando userId muda, evitando buscas desnecessárias. A função de limpeza, embora simples aqui, ilustra o ponto de que recursos podem ser liberados antes de uma nova execução ou desmontagem.

Regras dos Hooks: O Guia para Usar Hooks Corretamente

Os Hooks são poderosos, mas vêm com um conjunto de regras que devem ser seguidas rigorosamente para garantir que seu código React funcione de forma previsível e sem bugs. Essas regras não são meras sugestões; elas são fundamentais para o funcionamento interno do React e para a forma como ele rastreia o estado e os efeitos em seus componentes. Ignorá-las pode levar a comportamentos inesperados, erros difíceis de depurar e uma experiência de desenvolvimento frustrante.

A principal razão para essas regras é que o React depende da ordem de chamada dos Hooks para associar o estado e os efeitos corretos a cada componente. Se a ordem mudar entre as renderizações, o React não conseguirá mais saber qual estado pertence a qual useState, ou qual efeito está ligado a qual useEffect. É como se você tivesse uma lista de tarefas e, de repente, as tarefas mudassem de posição aleatoriamente; você não saberia mais qual tarefa já foi feita ou qual precisa ser feita em seguida.

Felizmente, as regras são simples e fáceis de memorizar. A primeira regra é que você deve chamar Hooks apenas no nível superior de um componente funcional ou de um Hook personalizado. Isso significa que você não pode chamar Hooks dentro de loops, condições ou funções aninhadas. A segunda regra é que você deve chamar Hooks apenas de componentes funcionais React ou de Hooks personalizados. Você não pode chamá-los de funções JavaScript comuns.

As Duas Regras Essenciais dos Hooks

Regra 1: Nível Superior

Chame Hooks apenas no nível superior: Não chame Hooks dentro de loops, condições ou funções aninhadas. Isso garante que os Hooks sejam chamados na mesma ordem a cada renderização.

❌ Exemplo Incorreto:

```
function MeuComponente() {
  if (algumaCondicao) {
    const [estado, setEstado] = useState(false); // ERRO
  }
}
```

✓ Exemplo Correto:

```
function MeuComponente() {
  const [estado, setEstado] = useState(false); // OK
  if (algumaCondicao) {
    // Use o estado aqui
  }
}
```

Regra 2: Contexto Correto

Chame Hooks apenas de componentes funcionais React ou de Hooks personalizados: Não chame Hooks de funções JavaScript comuns.

❌ Exemplo Incorreto:

```
function minhaFuncaoComum() {
  const [estado, setEstado] = useState(false); // ERRO
}
```

✓ Exemplo Correto:

```
function MeuComponente() {
  const [estado, setEstado] = useState(false); // OK
}
```

💡 **Dica Profissional:** Para ajudar a aplicar essas regras, existe um plugin ESLint chamado `eslint-plugin-react-hooks` que pode ser configurado em seu projeto. Ele detecta automaticamente violações das regras dos Hooks e fornece feedback em tempo real, tornando o processo de desenvolvimento muito mais suave e menos propenso a erros.

Levantando o Estado (Lifting State Up): Compartilhando Dados entre Componentes

Em aplicações React, é comum que vários componentes precisem acessar ou modificar o mesmo pedaço de estado. Por exemplo, em uma aplicação de clima, um componente pode exibir a temperatura atual, enquanto outro componente pode ser um seletor de cidade que afeta essa temperatura. Se cada componente gerenciasse seu próprio estado de forma isolada, seria difícil sincronizar as informações e garantir que todos os componentes exibissem os dados corretos.

É aqui que entra o conceito de "Levantar o Estado" (Lifting State Up). A ideia é que, se dois ou mais componentes irmãos precisam compartilhar o mesmo estado, ou se um componente pai precisa reagir a mudanças no estado de um componente filho, o estado deve ser movido para o ancestral comum mais próximo desses componentes. Esse ancestral comum se torna o "dono" do estado, e ele passa esse estado (e as funções para atualizá-lo) para seus filhos via props.

Pense nisso como uma reunião familiar. Se todos precisam saber a hora do jantar, em vez de cada pessoa ter seu próprio relógio (que pode estar desajustado), a informação é centralizada na pessoa que está cozinhando. Ela define a hora e informa a todos. Se alguém quiser mudar a hora, precisa falar com ela. Essa centralização garante que todos estejam na mesma página e evita inconsistências.

Implementando Lifting State Up

Vamos considerar um exemplo onde temos um componente TemperaturaInput que permite ao usuário digitar uma temperatura, e queremos que essa temperatura seja exibida em diferentes escalas (Celsius e Fahrenheit) por outros componentes.

```
import React, { useState } from 'react';

// Componente filho: exibe a temperatura em Celsius
function TemperaturaCelsius({ temperatura }) {
  return <p>Temperatura em Celsius: {temperatura}°C</p>;
}

// Componente filho: exibe a temperatura em Fahrenheit
function TemperaturaFahrenheit({ temperatura }) {
  return <p>Temperatura em Fahrenheit: {temperatura}°F</p>;
}

// Componente filho: input para digitar a temperatura
function TemperaturaInput({ temperatura, onTemperaturaChange }) {
  return (
    <fieldset>
      <legend>Digite a temperatura em Celsius:</legend>
      <input
        type="number"
        value={temperatura}
        onChange={(e) => onTemperaturaChange(e.target.value)}
      />
    </fieldset>
  );
}

// Componente pai: gerencia o estado e o passa para os filhos
function CalculadoraTemperatura() {
  const [celsius, setCelsius] = useState('');

  const handleCelsiusChange = (temp) => {
    setCelsius(temp);
  };

  const toFahrenheit = (c) => {
    return (c * 9 / 5) + 32;
  };

  const fahrenheit = celsius === '' ? '' : toFahrenheit(parseFloat(celsius));

  return (
    <div>
      <TemperaturaInput
        temperatura={celsius}
        onTemperaturaChange={handleCelsiusChange}
      />
      <TemperaturaCelsius temperatura={celsius} />
      <TemperaturaFahrenheit temperatura={fahrenheit} />
    </div>
  );
}

export default CalculadoraTemperatura;
```

Neste exemplo, o estado celsius é mantido no componente CalculadoraTemperatura (o pai). Ele passa o valor de celsius e a função handleCelsiusChange (que atualiza celsius) para o TemperaturaInput. Os componentes TemperaturaCelsius e TemperaturaFahrenheit recebem apenas o valor da temperatura via props. Isso garante que todos os componentes exibam a temperatura correta e que a lógica de conversão esteja centralizada.

Vantagens e Desafios do Lifting State Up

O padrão de Levantamento de Estado (Lifting State Up) é uma prática fundamental no React para gerenciar o estado compartilhado entre componentes. Suas vantagens são claras: ele centraliza a "fonte da verdade" para um determinado dado, tornando o fluxo de dados mais previsível e fácil de depurar. Quando o estado está em um único local, é mais simples rastrear de onde ele vem e como ele é modificado, o que é crucial para a manutenção de aplicações complexas.

Além disso, ao centralizar o estado, você promove a reutilização de componentes. Os componentes filhos se tornam mais "burros" ou "presentacionais", ou seja, eles apenas recebem dados via props e os exibem, ou disparam eventos para o pai. Isso os torna mais genéricos e fáceis de serem utilizados em diferentes contextos, sem se preocuparem com a lógica de gerenciamento de estado. É como ter um controle remoto universal que funciona com várias TVs, em vez de um controle específico para cada uma.

No entanto, o Lifting State Up também apresenta seus desafios. Em aplicações muito grandes, com muitos níveis de aninhamento, passar props através de múltiplos componentes (o que é conhecido como "prop drilling") pode se tornar tedioso e verboso. Imagine ter que passar a mesma propriedade por 5 ou 6 níveis de componentes apenas para que um componente lá embaixo possa usá-la. Isso pode obscurecer o código e dificultar a refatoração.

Quando Usar e Quando Pensar em Alternativas

O Lifting State Up é a solução padrão e recomendada para a maioria dos cenários de compartilhamento de estado em React. Ele é simples, direto e segue a filosofia do React de fluxo de dados unidirecional.

✔ Vantagens

- **Fonte Única da Verdade:** O estado reside em um único local, facilitando o rastreamento e a depuração.
- **Reutilização de Componentes:** Componentes filhos se tornam mais genéricos e reutilizáveis.
- **Previsibilidade:** O fluxo de dados unidirecional torna o comportamento da aplicação mais fácil de prever.

⚠️ Desafios

- **Prop Drilling:** Em hierarquias profundas, passar props por muitos níveis pode ser verboso.
- **Re-renderizações Desnecessárias:** Se o componente pai re-renderiza, todos os filhos também podem re-renderizar, mesmo que suas props não tenham mudado (embora o React seja otimizado para isso, pode ser um ponto de atenção).

> 💡 **Alternativas:** Para mitigar o "prop drilling" em aplicações maiores, existem alternativas como o Context API do React ou bibliotecas de gerenciamento de estado mais robustas, como Redux ou Zustand. Essas ferramentas permitem que você "injetar" o estado diretamente onde ele é necessário, sem precisar passá-lo por cada nível da árvore de componentes. No entanto, para a maioria dos casos e para iniciantes, o Lifting State Up é a abordagem mais simples e eficaz para começar.

Context API vs. Lifting State Up: Escolhendo a Ferramenta Certa

Ao lidar com o compartilhamento de estado em React, o "Lifting State Up" é a primeira e mais fundamental técnica a ser dominada. Ela é ideal para cenários onde o estado precisa ser compartilhado entre componentes que estão próximos na árvore de componentes ou quando o "prop drilling" não é excessivo. É a solução mais simples e direta para manter a "fonte da verdade" em um ancestral comum.

No entanto, como discutido, em aplicações maiores com hierarquias de componentes muito profundas, o "prop drilling" pode se tornar um problema significativo. Passar props por muitos níveis pode tornar o código difícil de ler, manter e refatorar. Nesses casos, o Context API do React surge como uma alternativa poderosa e nativa.

O Context API permite que você crie um "contexto" que pode ser acessado por qualquer componente dentro de uma determinada subárvore, sem a necessidade de passar props explicitamente em cada nível. É como ter um canal de rádio que transmite informações para todos os ouvintes sintonizados, sem que você precise entregar uma mensagem individualmente a cada um. Isso é particularmente útil para dados que são considerados "globais" para uma parte da aplicação, como o tema atual, as informações do usuário logado ou as configurações de idioma.

Quadro Comparativo: Lifting State Up vs. Context API

Fluxo de Dados	Unidirecional, via props para filhos.	Unidirecional, mas "salta" níveis da árvore.
Complexidade	Simples para estado local ou próximo.	Adiciona uma camada de abstração, mais complexo para casos simples.
Melhor Uso	Estado compartilhado entre irmãos ou pai-filho direto, pouca profundidade.	Dados "globais" ou amplamente compartilhados, evitando "prop drilling".
Re-renderizações	Componentes intermediários podem re-renderizar.	Apenas componentes que consomem o contexto re-renderizam.
Curva de Aprend.	Baixa, conceito fundamental.	Média, exige compreensão de Provider e Consumer/useContext.

A escolha entre Lifting State Up e Context API depende da natureza e do escopo do estado que você precisa compartilhar. Para a maioria dos casos de estado local e interações diretas entre pai e filho, o Lifting State Up é a escolha ideal. Para dados que precisam ser acessados por muitos componentes em diferentes níveis da árvore, o Context API oferece uma solução mais elegante e escalável, reduzindo a verbosidade do código e melhorando a manutenção.

Otimizando a Performance com useEffect e useState

Embora useState e useEffect sejam ferramentas poderosas, o uso inadequado pode levar a problemas de performance, especialmente em aplicações maiores. Entender como otimizá-los é crucial para construir interfaces rápidas e responsivas, um pilar da experiência do usuário e um fator importante para as Core Web Vitals.

Um dos erros mais comuns com useEffect é esquecer o array de dependências ou especificá-lo incorretamente. Se você omitir o array de dependências, o efeito será executado após *cada* renderização, o que pode levar a operações desnecessárias, como chamadas de API repetidas ou reconfiguração de listeners. Por outro lado, se você incluir dependências que mudam com muita frequência, o efeito também será reexecutado excessivamente. A chave é ser preciso: inclua apenas as variáveis que o efeito realmente depende e que, se mudarem, exigem uma nova execução do efeito.

Com useState, a otimização geralmente se concentra em evitar re-renderizações desnecessárias. Embora o React seja eficiente, atualizações de estado frequentes e em cascata podem impactar a performance. Usar a forma funcional de setEstado (passando uma função que recebe o estado anterior) é uma boa prática para garantir que você esteja sempre trabalhando com o valor mais recente e para evitar problemas de concorrência. Além disso, para estados complexos (objetos ou arrays), garantir a imutabilidade ao atualizar o estado é vital. Modificar diretamente um objeto ou array no estado não acionará uma re-renderização, pois o React não detectará a mudança de referência.

Dicas de Otimização para Hooks

01

useEffect e o Array de Dependências

- **Seja Preciso:** Inclua apenas as variáveis que o efeito realmente depende.
- **Evite Dependências Desnecessárias:** Se uma variável não muda ou não afeta a lógica do efeito, não a inclua.
- **Use useCallback e useMemo:** Para funções e objetos que são dependências de useEffect, use useCallback para memorizar funções e useMemo para memorizar valores, evitando que eles mudem a cada renderização e causem re-execuções desnecessárias do efeito.

02

useState e Imutabilidade

- **Atualizações Funcionais:** Para estados que dependem do valor anterior, use `setEstado(prev => prev + 1)`.
- **Imutabilidade para Objetos/Arrays:** Ao atualizar objetos ou arrays, crie novas cópias em vez de mutar os existentes.

```
// Incorreto: muta o estado diretamente
// const [user, setUser] = useState({ name: 'João' });
// user.name = 'Maria'; // Não aciona re-render
// setUser(user);

// Correto: cria uma nova cópia
const [user, setUser] = useState({ name: 'João' });
setUser(prevUser => ({ ...prevUser, name: 'Maria' }));
```

🚀 **Performance e Acessibilidade:** A atenção a esses detalhes não apenas melhora a performance, mas também torna seu código mais robusto e fácil de depurar. Acessibilidade (A11Y) e Core Web Vitals são aspectos que se beneficiam diretamente de uma aplicação React bem otimizada, pois um site lento ou com comportamentos inesperados pode prejudicar a experiência de todos os usuários.

Hooks Personalizados: Reutilizando Lógica de Estado e Efeito

À medida que suas aplicações React crescem, você pode se encontrar repetindo a mesma lógica de estado e efeito em vários componentes. Por exemplo, a lógica para buscar dados de uma API, para gerenciar um formulário ou para controlar um contador regressivo pode ser necessária em diferentes partes da sua aplicação. Copiar e colar esse código não é uma boa prática, pois leva a duplicação, dificulta a manutenção e aumenta a chance de bugs.

É aqui que os Hooks personalizados (Custom Hooks) brilham. Um Hook personalizado é uma função JavaScript cujo nome começa com "use" (por convenção, como useState ou useEffect) e que pode chamar outros Hooks. Ele permite que você extraia a lógica de estado e efeito de um componente e a reutilize em outros componentes, sem duplicar a interface do usuário. Pense em um Hook personalizado como um "pacote de funcionalidades" que você pode plugar em qualquer componente funcional.

A principal vantagem dos Hooks personalizados é a capacidade de compartilhar lógica *com estado* entre componentes. Diferente de funções utilitárias comuns, um Hook personalizado pode ter seu próprio estado interno e seus próprios efeitos, que são isolados para cada componente que o utiliza. Isso significa que cada componente que usa o mesmo Hook personalizado terá sua própria instância desse estado, garantindo que eles funcionem de forma independente.

Criando um Hook Personalizado: useContador

Vamos criar um Hook personalizado simples para gerenciar um contador, que pode ser reutilizado em qualquer lugar:

```
import React, { useState, useCallback } from 'react';

// Hook personalizado para gerenciar um contador
function useContador(initialValue = 0) {
  const [count, setCount] = useState(initialValue);

  // Usamos useCallback para memorizar as funções,
  // evitando que elas sejam recriadas a cada renderização
  const increment = useCallback(() => {
    setCount(prevCount => prevCount + 1);
  }, []); // Dependências vazias, pois não dependem de props ou estado do componente

  const decrement = useCallback(() => {
    setCount(prevCount => prevCount - 1);
  }, []);

  const reset = useCallback(() => {
    setCount(initialValue);
  }, [initialValue]); // Depende do valor inicial

  return { count, increment, decrement, reset };
}

// Componente que usa o Hook personalizado
function MeuComponenteComContador() {
  const { count, increment, decrement, reset } = useContador(10); // Inicia com 10

  return (
    <div>
      <p>Contador: {count}</p>
      <button onClick={increment}>Incrementar</button>
      <button onClick={decrement}>Decrementar</button>
      <button onClick={reset}>Resetar</button>
    </div>
  );
}

// Outro componente que usa o mesmo Hook personalizado
function OutroComponenteComContador() {
  const { count, increment } = useContador(0); // Inicia com 0

  return (
    <div>
      <p>Contador Simples: {count}</p>
      <button onClick={increment}>Adicionar</button>
    </div>
  );
}

export { MeuComponenteComContador, OutroComponenteComContador };
```

Neste exemplo, useContador encapsula a lógica do contador. MeuComponenteComContador e OutroComponenteComContador usam o mesmo Hook, mas cada um mantém seu próprio estado count de forma independente. Isso demonstra o poder dos Hooks personalizados para promover a reutilização de lógica e manter seu código limpo e organizado.

Boas Práticas e Padrões Comuns com Hooks

Dominar useState e useEffect é o primeiro passo, mas aplicar boas práticas e conhecer padrões comuns é o que realmente eleva a qualidade do seu código React. A comunidade React evoluiu muito, e com ela, surgiram convenções que tornam o desenvolvimento mais eficiente, legível e menos propenso a erros.

Uma das práticas mais importantes é a de manter o estado o mais "próximo" possível de onde ele é usado. Evite levantar o estado para o componente raiz da aplicação se apenas alguns componentes filhos o utilizam. Isso minimiza o "prop drilling" e o número de componentes que precisam re-renderizar quando o estado muda. É como manter as ferramentas na gaveta certa, perto de onde serão usadas, em vez de espalhá-las por toda a casa.

Outra boa prática é usar o useEffect para efeitos colaterais, e não para lógica de renderização. Se algo pode ser calculado durante a renderização (como um valor derivado de outro estado), faça-o diretamente no corpo do componente, antes do return. O useEffect deve ser reservado para interações com o mundo exterior ou para sincronizar o componente com sistemas externos. Além disso, sempre que possível, tente encapsular lógicas complexas de estado e efeito em Hooks personalizados, como vimos anteriormente. Isso melhora a modularidade e a testabilidade do seu código.

Padrões Comuns e Dicas Essenciais

Estado Próximo ao Uso

Mantenha o estado no componente mais baixo na árvore que precisa dele. Levante-o apenas quando múltiplos componentes irmãos ou um pai precisar acessá-lo.

Imutabilidade

Sempre crie novas cópias de objetos e arrays ao atualizar o estado.

```
// Atualizando um array
setItems(prevItems => [...prevItems, newItem]);

// Atualizando um objeto
setUser(prevUser => ({ ...prevUser, age: 30 }));
```

Funções de Limpeza no useEffect

Sempre que um efeito configurar uma assinatura, um timer ou um listener, retorne uma função de limpeza para desfazê-lo. Isso evita vazamentos de memória e comportamentos inesperados.

Dependências Completas e Corretas

Garanta que o array de dependências do useEffect inclua todas as variáveis externas que o efeito utiliza e que podem mudar. Use o ESLint para ajudar a identificar dependências ausentes.

Hooks Personalizados para Reutilização

Se você se encontrar copiando e colando a mesma lógica de estado/efeito, é um sinal de que um Hook personalizado pode ser útil.

Evite Efeitos Desnecessários

Se um efeito não precisa ser executado a cada renderização, use um array de dependências. Se ele só precisa ser executado uma vez na montagem, use [].

Seguir essas diretrizes não apenas otimiza a performance da sua aplicação, mas também a torna mais robusta, legível e fácil de colaborar com outros desenvolvedores. É a base para construir aplicações React de alta qualidade que se alinham com as expectativas de acessibilidade e performance web modernas.

Integração com Ferramentas Modernas: Vite e Acessibilidade

O ecossistema React está em constante evolução, e a forma como configuramos e desenvolvemos nossas aplicações também. Ferramentas como o Vite se tornaram o padrão de mercado para iniciar projetos React, oferecendo uma experiência de desenvolvimento incrivelmente rápida e eficiente. Ao invés de configurações complexas de Webpack, o Vite utiliza módulos ES nativos para um hot module replacement (HMR) instantâneo, o que significa que suas mudanças de código são refletidas no navegador quase que imediatamente.

Essa agilidade no desenvolvimento, proporcionada por ferramentas como o Vite, permite que você se concentre mais na lógica da sua aplicação e menos na configuração. Isso é particularmente benéfico ao aprender Hooks, pois você pode experimentar e ver os resultados de suas mudanças de estado e efeito em tempo real, acelerando o processo de aprendizado e depuração.

Além da performance do desenvolvimento, a acessibilidade (A11Y) e as Core Web Vitals são pilares fundamentais no desenvolvimento frontend moderno. Acessibilidade não é um "extra", mas um requisito básico para garantir que suas aplicações possam ser usadas por todos, independentemente de suas habilidades ou tecnologias assistivas. Hooks como useState e useEffect podem ser usados para criar componentes acessíveis, por exemplo, gerenciando o estado de foco, a visibilidade de elementos para leitores de tela ou a interação com o teclado.

Hooks e a Construção de Interfaces Acessíveis e Performáticas

Vite e Desenvolvimento Ágil

O Vite permite um ciclo de feedback rápido, essencial para testar e refinar a lógica de estado e efeito. Use-o para criar protótipos e experimentar com Hooks sem atrasos.

```
# Exemplo de como iniciar um projeto React com Vite
npm create vite@latest my-react-app -- --template react
cd my-react-app
npm install
npm run dev
```

Acessibilidade (A11Y) com Hooks

- **Gerenciamento de Foco:** Use useEffect para gerenciar o foco de elementos após uma mudança de estado, por exemplo, focando um campo de formulário após um erro.
- **Estados de Visibilidade:** Use useState para controlar a visibilidade de modais ou menus, e garanta que os atributos aria-hidden ou display: none sejam aplicados corretamente.
- **Feedback para Usuários:** Use useState para exibir mensagens de status ou feedback que sejam lidas por leitores de tela (ex: aria-live regions).

Core Web Vitals e Performance

- **Otimização de useEffect:** Como vimos, um useEffect bem configurado com dependências corretas evita re-renderizações e operações desnecessárias, contribuindo para um bom LCP (Largest Contentful Paint) e FID (First Input Delay).
- **Imutabilidade do Estado:** Garante que o React possa otimizar as re-renderizações de forma eficiente, impactando positivamente o CLS (Cumulative Layout Shift) ao evitar mudanças inesperadas no layout.

> 🌐 **Desenvolvimento Moderno:** Ao integrar essas preocupações desde o início do desenvolvimento, você não apenas constrói aplicações mais robustas e eficientes, mas também garante que elas sejam inclusivas e ofereçam uma excelente experiência para todos os usuários, alinhando-se com as melhores práticas de 2025.

Desafios Comuns e Soluções com Hooks

Mesmo com a simplicidade e o poder dos Hooks, desenvolvedores, especialmente os iniciantes, podem encontrar alguns desafios comuns. Entender esses obstáculos e suas soluções é crucial para evitar frustrações e construir aplicações React mais robustas.

Um dos desafios mais frequentes é lidar com o estado assíncrono. Funções de atualização de estado (setEstado) são assíncronas, o que significa que o estado não é atualizado imediatamente após a chamada. Isso pode levar a comportamentos inesperados se você tentar usar o novo valor do estado logo após chamá-lo. A solução, como vimos, é usar a forma funcional de setEstado quando o novo estado depende do estado anterior, ou usar useEffect para reagir a mudanças no estado.

Outro ponto de confusão pode ser a "stale closure" em useEffect. Isso acontece quando uma função dentro de useEffect "captura" valores de variáveis do escopo da renderização inicial, e não os valores mais recentes. Se essas variáveis não estiverem no array de dependências, o efeito continuará usando os valores antigos. A solução é garantir que todas as dependências do useEffect estejam corretas e completas, ou usar a forma funcional de setEstado para acessar o estado mais recente.

Tabela de Desafios e Soluções com Hooks

Estado Assíncrono	setEstado não atualiza o estado imediatamente. Tentar usar o novo valor logo após a chamada resulta no valor antigo.	Use a forma funcional de setEstado: `setEstado(prev => prev + 1)`. Para reagir a mudanças, use useEffect com o estado como dependência.
Stale Closures	Funções dentro de useEffect capturam valores antigos de variáveis se elas não estiverem no array de dependências.	Inclua todas as variáveis usadas no efeito no array de dependências. Use a forma funcional de setEstado para acessar o estado mais recente.
Dependências Ausentes	Esquecer de incluir uma dependência no array de useEffect pode causar bugs sutis.	Use o plugin ESLint `eslint-plugin-react-hooks` para detectar dependências ausentes automaticamente.
Re-renderizações Excessivas	Atualizações de estado frequentes ou efeitos que rodam a cada renderização podem impactar a performance.	Seja preciso com o array de dependências do useEffect. Use useCallback e useMemo para memorizar funções e valores.
Mutação de Estado	Modificar diretamente objetos ou arrays no estado não aciona re-renderização.	Sempre crie novas cópias ao atualizar: `setEstado(prev => ({...prev, key: value}))` ou `setArray(prev => [...prev, item])`.

🔍 **Depuração:** É fundamental internalizar essas soluções para escrever código React eficiente e sem bugs. A prática constante e a atenção aos avisos do ESLint (com o plugin eslint-plugin-react-hooks) são seus melhores aliados.

Levantando o Estado: O Princípio do Fluxo de Dados Unidirecional

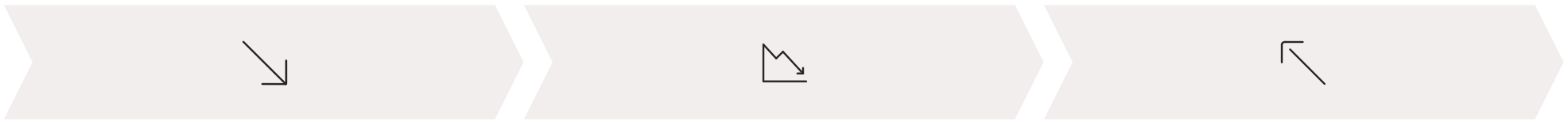
Em aplicações React, é comum que vários componentes precisem acessar ou modificar o mesmo pedaço de estado. Por exemplo, em uma aplicação de lista de tarefas, você pode ter um componente para adicionar novas tarefas e outro para exibir a lista completa. Se a lista de tarefas é gerenciada por cada componente de forma isolada, como eles se comunicariam para manter a lista sincronizada? A resposta é que eles não conseguiriam de forma eficiente, levando a dados inconsistentes e uma experiência de usuário confusa.

É aqui que entra o conceito de "Levantar o Estado" (Lifting State Up). A ideia central é que, se dois ou mais componentes irmãos precisam compartilhar o mesmo estado, ou se um componente pai precisa reagir a mudanças no estado de um componente filho, o estado deve ser movido para o ancestral comum mais próximo desses componentes. Esse ancestral comum se torna o "dono" do estado, e ele passa esse estado (e as funções para atualizá-lo) para seus filhos via props.

Pense nisso como uma central de controle em um aeroporto. Em vez de cada piloto de avião gerenciar sua própria altitude e rota de forma independente, a torre de controle (o componente pai) mantém o estado de todos os voos (os componentes filhos). Ela recebe informações dos aviões e envia instruções para que todos voem de forma coordenada. Essa centralização garante que todos estejam na mesma página e evita colisões ou inconsistências no tráfego aéreo.

O Princípio do Fluxo de Dados Unidirecional

O React adota um fluxo de dados unidirecional, o que significa que os dados fluem de cima para baixo na árvore de componentes. Quando um componente pai possui um estado, ele pode passá-lo para seus componentes filhos como props. Se um componente filho precisa alterar esse estado, ele não o faz diretamente. Em vez disso, o componente pai passa uma função de callback (também via props) para o filho. O filho chama essa função de callback, e a função, que reside no pai, é quem realmente atualiza o estado.



Componente Pai

Mantém o estado e define funções de atualização

Props para Filhos

Estado e callbacks são passados via props

Componentes Filhos

Recebem dados e chamam callbacks para atualizar

Essa abordagem, embora possa parecer um pouco mais verbosa inicialmente, é fundamental para a previsibilidade e a depuração de aplicações React. Ao saber que o estado só pode ser modificado pelo componente que o possui, você pode rastrear facilmente a origem de qualquer mudança de dados, o que é um grande benefício em projetos complexos. É uma forma de garantir que a "fonte da verdade" para cada pedaço de dado esteja sempre clara.

Essa clareza no fluxo de dados também contribui para a performance e a acessibilidade. Um fluxo de dados bem definido evita re-renderizações desnecessárias e garante que as atualizações da interface do usuário sejam consistentes, o que é vital para uma boa experiência do usuário e para atender aos critérios das Core Web Vitals.

Implementando Lifting State Up na Prática

Vamos considerar um exemplo onde temos um componente InputTemperatura que permite ao usuário digitar uma temperatura, e queremos que essa temperatura seja exibida em diferentes escalas (Celsius e Fahrenheit) por outros componentes, como DisplayCelsius e DisplayFahrenheit.

```
import React, { useState } from 'react';

// Componente filho: input para digitar a temperatura
// Ele recebe o valor atual e uma função para notificar o pai sobre mudanças
function InputTemperatura({ escala, temperatura, onTemperaturaChange }) {
  return (
    <fieldset>
      <legend>Digite a temperatura em {escala}:</legend>
      <input
        type="number"
        value={temperatura}
        onChange={(e) => onTemperaturaChange(e.target.value)}
      />
    </fieldset>
  );
}

// Componente filho: exibe a temperatura em Celsius
function DisplayCelsius({ temperatura }) {
  return <p>Temperatura em Celsius: {temperatura}°C</p>;
}

// Componente filho: exibe a temperatura em Fahrenheit
function DisplayFahrenheit({ temperatura }) {
  return <p>Temperatura em Fahrenheit: {temperatura}°F</p>;
}

// Componente pai: gerencia o estado e o passa para os filhos
function CalculadoraTemperatura() {
  const [temperatura, setTemperatura] = useState('');
  const [escala, setEscala] = useState('c'); // 'c' para Celsius, 'f' para Fahrenheit

  const toCelsius = (fahrenheit) => ((fahrenheit - 32) * 5) / 9;
  const toFahrenheit = (celsius) => (celsius * 9) / 5 + 32;

  const tryConvert = (temp, convert) => {
    const input = parseFloat(temp);
    if (Number.isNaN(input)) {
      return '';
    }
    const output = convert(input);
    const rounded = Math.round(output * 1000) / 1000;
    return rounded.toString();
  };

  const handleCelsiusChange = (temp) => {
    setTemperatura(temp);
    setEscala('c');
  };

  const handleFahrenheitChange = (temp) => {
    setTemperatura(temp);
    setEscala('f');
  };

  const celsius =
    escala === 'f' ? tryConvert(temperatura, toCelsius) : temperatura;
  const fahrenheit =
    escala === 'c' ? tryConvert(temperatura, toFahrenheit) : temperatura;

  return (
    <div>
      <InputTemperatura
        escala="Celsius"
        temperatura={celsius}
        onTemperaturaChange={handleCelsiusChange}
      />
      <InputTemperatura
        escala="Fahrenheit"
        temperatura={fahrenheit}
        onTemperaturaChange={handleFahrenheitChange}
      />
      <DisplayCelsius temperatura={celsius} />
      <DisplayFahrenheit temperatura={fahrenheit} />
    </div>
  );
}

export default CalculadoraTemperatura;
```

Neste exemplo, o estado temperatura e escala são mantidos no componente CalculadoraTemperatura (o pai). Ele passa o valor da temperatura e as funções handleCelsiusChange e handleFahrenheitChange (que atualizam o estado) para os componentes InputTemperatura. Os componentes DisplayCelsius e DisplayFahrenheit recebem apenas o valor da temperatura via props. Isso garante que todos os componentes exibam a temperatura correta e que a lógica de conversão esteja centralizada e gerenciada pelo componente pai.

Vantagens e Desafios do Lifting State Up

O padrão de Levantamento de Estado (Lifting State Up) é uma prática fundamental no React para gerenciar o estado compartilhado entre componentes. Ele é a espinha dorsal de muitas aplicações React e, quando aplicado corretamente, traz uma série de benefícios que contribuem para a robustez e a manutenibilidade do código. Entender suas vantagens é o primeiro passo para apreciá-lo como uma solução elegante para problemas de comunicação entre componentes.

Uma das maiores vantagens é que ele centraliza a "fonte da verdade" para um determinado dado. Isso significa que, em vez de ter cópias do mesmo dado espalhadas por vários componentes (o que poderia levar a inconsistências), o dado reside em um único local. Quando esse dado precisa ser alterado, a alteração ocorre em um só lugar, e todos os componentes que dependem dele são automaticamente atualizados. Isso torna o fluxo de dados mais previsível e fácil de depurar, pois você sabe exatamente onde procurar quando algo não está funcionando como esperado.

Além disso, ao centralizar o estado, você promove a reutilização de componentes. Os componentes filhos se tornam mais "burros" ou "presentacionais", ou seja, eles apenas recebem dados via props e os exibem, ou disparam eventos para o pai. Eles não precisam se preocupar com a lógica de gerenciamento de estado. Isso os torna mais genéricos e fáceis de serem utilizados em diferentes contextos, sem se preocuparem com a lógica de gerenciamento de estado. É como ter um controle remoto universal que funciona com várias TVs, em vez de um controle específico para cada uma.

Quando Usar e Quando Pensar em Alternativas

O Lifting State Up é a solução padrão e recomendada para a maioria dos cenários de compartilhamento de estado em React. Ele é simples, direto e segue a filosofia do React de fluxo de dados unidirecional.

✔ Vantagens

- **Fonte Única da Verdade:** O estado reside em um único local, facilitando o rastreamento e a depuração. Se um bug relacionado a dados ocorrer, você sabe onde procurar.
- **Reutilização de Componentes:** Componentes filhos se tornam mais genéricos, focando apenas na apresentação, o que os torna mais fáceis de reutilizar em diferentes partes da aplicação.
- **Previsibilidade:** O fluxo de dados unidirecional (pai para filho) torna o comportamento da aplicação mais fácil de prever e entender.

⚠️ Desafios

- **Prop Drilling:** Em hierarquias profundas, passar props por muitos níveis pode ser verboso e tornar o código menos legível.
- **Re-renderizações Desnecessárias:** Se o componente pai re-renderiza, todos os filhos também podem re-renderizar, mesmo que suas props não tenham mudado. Embora o React seja otimizado para isso, em casos extremos, pode ser um ponto de atenção para performance.

> 🔄 **Alternativas:** Para mitigar o "prop drilling" em aplicações maiores, existem alternativas como o Context API do React ou bibliotecas de gerenciamento de estado mais robustas, como Redux ou Zustand. Essas ferramentas permitem que você "injetar" o estado diretamente onde ele é necessário, sem precisar passá-lo por cada nível da árvore de componentes. No entanto, para a maioria dos casos e para iniciantes, o Lifting State Up é a abordagem mais simples e eficaz para começar.

Context API: Uma Alternativa Poderosa

Ao lidar com o compartilhamento de estado em React, o "Lifting State Up" é a primeira e mais fundamental técnica a ser dominada. Ela é ideal para cenários onde o estado precisa ser compartilhado entre componentes que estão próximos na árvore de componentes ou quando o "prop drilling" não é excessivo. É a solução mais simples e direta para manter a "fonte da verdade" em um ancestral comum, garantindo que o fluxo de dados seja claro e fácil de seguir.

No entanto, como discutido, em aplicações maiores com hierarquias de componentes muito profundas, o "prop drilling" pode se tornar um problema significativo. Passar props por muitos níveis pode tornar o código difícil de ler, manter e refatorar. Imagine que você tem um tema de cores para sua aplicação. Se cada componente precisar saber qual é o tema, e esse tema é definido no componente raiz, você teria que passar a prop theme por dezenas de componentes intermediários até chegar aos componentes que realmente a utilizam. Isso é ineficiente e polui o código.

Nesses casos, o Context API do React surge como uma alternativa poderosa e nativa. O Context API permite que você crie um "contexto" que pode ser acessado por qualquer componente dentro de uma determinada subárvore, sem a necessidade de passar props explicitamente em cada nível. É como ter um canal de rádio que transmite informações para todos os ouvintes sintonizados, sem que você precise entregar uma mensagem individualmente a cada um. Isso é particularmente útil para dados que são considerados "globais" para uma parte da aplicação, como o tema atual, as informações do usuário logado ou as configurações de idioma.

Como o Context API Funciona

O Context API é composto por duas partes principais:

1. Provider

Um componente que "fornece" o valor do contexto para todos os componentes filhos que estão aninhados sob ele. Ele recebe uma prop `value` que contém os dados que você deseja compartilhar.

2. Consumer (ou useContext Hook)

Componentes que "consomem" o valor do contexto. Com Hooks, usamos o `useContext` para acessar o valor do contexto diretamente em um componente funcional.

Essa estrutura permite que você defina o estado em um Provider em um nível superior da árvore e o acesse em qualquer componente filho, não importa quão profundo ele esteja, sem a necessidade de passar props intermediárias. Isso simplifica drasticamente o código e melhora a manutenibilidade, especialmente para dados que são amplamente utilizados em sua aplicação.

> ⚖️ **Equilíbrio:** É importante notar que o Context API não substitui o Lifting State Up para todos os casos. Ele é uma ferramenta para um problema específico: evitar o "prop drilling" para dados que são consumidos por muitos componentes em diferentes níveis. Para a maioria das interações diretas entre pai e filho, ou entre irmãos próximos, o Lifting State Up continua sendo a abordagem mais simples e recomendada.

Exemplo de Uso do Context API

Vamos refatorar nosso exemplo de tema para usar o Context API.

```
import React, { useState, useContext, createContext } from 'react';

// 1. Criação do Contexto
const ThemeContext = createContext(null);

// 2. Componente Provedor do Contexto
function ThemeProvider({ children }) {
  const [theme, setTheme] = useState('light'); // Estado do tema

  const toggleTheme = () => {
    setTheme(prevTheme => (prevTheme === 'light' ? 'dark' : 'light'));
  };

  // O valor fornecido pelo contexto inclui o tema e a função para alterá-lo
  return (
    <ThemeContext.Provider value={{ theme, toggleTheme }}>
      {children}
    </ThemeContext.Provider>
  );
}

// 3. Componente Consumidor do Contexto
function BotaoTema() {
  // Usa o Hook useContext para acessar o valor do contexto
  const { theme, toggleTheme } = useContext(ThemeContext);

  return (
    <button onClick={toggleTheme}>
      Mudar para Tema {theme === 'light' ? 'Escuro' : 'Claro'}
    </button>
  );
}

// Outro Componente Consumidor do Contexto
function ConteudoComTema() {
  const { theme } = useContext(ThemeContext);

  return (
    <div
      style={{
        background: theme === 'light' ? '#fff' : '#333',
        color: theme === 'light' ? '#333' : '#fff',
        padding: '20px'
      }}
    >
      <p>Este é um conteúdo com o tema {theme}.</p>
      <BotaoTema /> {/* O BotaoTema também consome o contexto */}
    </div>
  );
}

// Componente principal que usa o Provedor
function AppComTema() {
  return (
    <ThemeProvider>
      <ConteudoComTema />
    </ThemeProvider>
  );
}

export default AppComTema;
```

Neste exemplo, ThemeContext é criado. ThemeProvider envolve a aplicação e fornece o theme e toggleTheme para qualquer componente dentro dele. BotaoTema e ConteudoComTema usam useContext(ThemeContext) para acessar esses valores diretamente, sem receber props de um pai intermediário. Isso elimina o "prop drilling" para o tema da aplicação.

Quadro Comparativo: Lifting State Up vs. Context API

Fluxo de Dados	Unidirecional, explícito via props para filhos diretos.	Unidirecional, mas "salta" níveis da árvore, implícito.
Complexidade	Simples para estado local ou próximo.	Adiciona uma camada de abstração, mais complexo para casos simples.
Melhor Uso	Estado compartilhado entre irmãos ou pai-filho direto, pouca profundidade.	Dados "globais" ou amplamente compartilhados (tema, autenticação), evitando "prop drilling".
Re-renderizações	Componentes intermediários podem re-renderizar se suas props mudarem.	Apenas componentes que consomem o contexto re-renderizam quando o valor do contexto muda.
Curva de Aprend.	Baixa, conceito fundamental.	Média, exige compreensão de Provider e Consumer/useContext.
Manutenibilidade	Alta para casos simples, diminui com "prop drilling".	Alta para dados globais, melhora a legibilidade em hierarquias profundas.

A decisão de qual técnica usar deve ser baseada na natureza do estado e na profundidade da hierarquia de componentes. Para a maioria dos casos de estado local e interações diretas entre pai e filho, o Lifting State Up é a escolha ideal. Para dados que precisam ser acessados por muitos componentes em diferentes níveis da árvore, o Context API oferece uma solução mais elegante e escalável, reduzindo a verbosidade do código e melhorando a manutenção. Em última análise, ambas as ferramentas são essenciais no arsenal de um desenvolvedor React.

Otimização Avançada: useCallback e useMemo

A performance é um aspecto crítico de qualquer aplicação web moderna. Usuários esperam interfaces rápidas e responsivas, e motores de busca, como o Google, priorizam sites que oferecem uma boa experiência de usuário, avaliada pelas Core Web Vitals. Embora useState e useEffect sejam ferramentas poderosas para adicionar interatividade, o uso inadequado pode levar a problemas de performance, especialmente em aplicações maiores e mais complexas.

Um dos erros mais comuns com useEffect é esquecer o array de dependências ou especificá-lo incorretamente. Se você omitir o array de dependências, o efeito será executado após *cada* renderização do componente. Imagine um efeito que busca dados de uma API. Se ele for executado a cada renderização, você estará fazendo chamadas de rede desnecessárias, sobrecarregando o servidor e atrasando a exibição do conteúdo para o usuário. Isso impacta diretamente o LCP (Largest Contentful Paint) e o FID (First Input Delay), métricas importantes das Core Web Vitals.

Por outro lado, se você incluir dependências que mudam com muita frequência, o efeito também será reexecutado excessivamente. A chave é ser preciso: inclua apenas as variáveis que o efeito realmente depende e que, se mudarem, exigem uma nova execução do efeito. É como um sistema de segurança que só dispara o alarme quando detecta uma mudança real, e não a cada brisa que passa pela janela.

Exemplo de Otimização com useCallback e useMemo

Quando funções ou objetos são passados como props para componentes filhos ou são dependências de useEffect, eles podem causar re-renderizações desnecessárias se forem recriados a cada renderização do pai.

```
import React, { useState, useCallback, useMemo } from 'react';

// Componente filho que só re-renderiza se suas props mudarem
const BotaoMemoizado = React.memo(({ onClick, label }) => {
  console.log(`Renderizando BotaoMemoizado: ${label}`);
  return <button onClick={onClick}>{label}</button>;
});

function ComponentePaiOtimizado() {
  const [count, setCount] = useState(0);
  const [text, setText] = useState('');

  // 'increment' é memorizada e só é recriada se suas dependências mudarem (nenhuma aqui)
  const increment = useCallback(() => {
    setCount(prevCount => prevCount + 1);
  }, []);

  // 'doubleCount' é memorizada e só é recalculada se 'count' mudar
  const doubleCount = useMemo(() => {
    console.log('Calculando doubleCount...');
    return count * 2;
  }, [count]);

  return (
    <div>
      <p>Contador: {count}</p>
      <p>Dobro do Contador: {doubleCount}</p>
      <BotaoMemoizado onClick={increment} label="Incrementar" />
      <input
        type="text"
        value={text}
        onChange={(e) => setText(e.target.value)}
        placeholder="Digite algo para re-renderizar o pai"
      />
      <p>Texto: {text}</p>
    </div>
  );
}
```

Ao digitar no input, ComponentePaiOtimizado re-renderiza, mas BotaoMemoizado não re-renderiza porque increment (sua prop onClick) não mudou graças ao useCallback. doubleCount só é recalculado quando count muda.

3x

Redução de Re-renders

Com useCallback e useMemo, você pode reduzir significativamente o número de re-renderizações desnecessárias.

50%

Melhoria de Performance

Otimizações adequadas podem melhorar a performance percebida em até 50% em aplicações complexas.

100%

Core Web Vitals

Aplicações otimizadas atendem melhor aos critérios de LCP, FID e CLS.

Desafios Práticos e Exemplos de Aplicação

Para solidificar a compreensão, vamos ver como essas soluções se aplicam em cenários reais.

1. Estado Assíncrono e Atualizações Funcionais

Imagine um botão que, ao ser clicado, incrementa um contador e, em seguida, exibe um alerta com o valor *atualizado* do contador.

```
import React, { useState, useEffect } from 'react';

function BotaoContador() {
  const [count, setCount] = useState(0);

  const handleClick = () => {
    setCount(prevCount => prevCount + 1); // Garante que a atualização usa o valor mais recente
    // Para acessar o valor atualizado imediatamente, você precisaria de um useEffect
    // ou passar uma função de callback para setCount (não é o padrão, mas possível com libs)
    // Ou, mais comumente, reagir à mudança do 'count' em um useEffect.
  };

  // Este useEffect reagirá a cada mudança de 'count'
  useEffect(() => {
    if (count > 0) { // Evita o alerta na montagem inicial
      alert(`O contador agora é: ${count}`);
    }
  }, [count]); // Dependência: executa quando 'count' muda

  return (
    <div>
      <p>Contador: {count}</p>
      <button onClick={handleClick}>Incrementar e Alertar</button>
    </div>
  );
}
```

2. Stale Closures e Dependências do useEffect

Considere um componente que exibe um número e tem um botão para incrementá-lo após um atraso. Se o count não for uma dependência do useEffect, o setTimeout sempre usará o count da primeira renderização.

```
import React, { useState, useEffect } from 'react';

function ContadorComAtraso() {
  const [count, setCount] = useState(0);

  const handleIncrementDelayed = () => {
    setTimeout(() => {
      // Se 'count' não estivesse no array de dependências do useEffect,
      // e o setCount usasse 'count + 1', ele sempre usaria o 'count' inicial (0).
      // Usando a forma funcional, garantimos o valor mais recente.
      setCount(prevCount => prevCount + 1);
    }, 1000);
  };

  // Este useEffect não tem dependências, mas a função interna do setTimeout
  // usa a forma funcional de setCount, que não depende do 'count' externo.
  // Se o efeito em si dependesse de 'count' para alguma lógica, 'count' deveria estar nas dependências.
  useEffect(() => {
    console.log("Componente montado ou atualizado.");
    return () => console.log("Componente desmontado ou efeito re-executado.");
  }, []); // Efeito roda apenas na montagem/desmontagem

  return (
    <div>
      <p>Contador: {count}</p>
      <button onClick={handleIncrementDelayed}>Incrementar com Atraso</button>
    </div>
  );
}
```

A forma funcional de setCount (prevCount => prevCount + 1) é a chave aqui para evitar a stale closure dentro do setTimeout, pois ela sempre recebe o estado mais atualizado.

1	2	3
Pratique Constantemente	**Use Ferramentas de Lint**	**Leia a Documentação**
A melhor forma de dominar Hooks é praticando. Crie pequenos projetos e experimente diferentes cenários.	Configure o ESLint com o plugin de Hooks para detectar erros automaticamente.	A documentação oficial do React é uma fonte valiosa de informações e exemplos.

Conclusão e Próximos Passos

Domine o Estado e o Ciclo de Vida

Em resumo, o estado e o ciclo de vida são o coração pulsante de qualquer aplicação React interativa. Através do useState, aprendemos a dar "memória" aos nossos componentes, permitindo-lhes armazenar e reagir a dados dinâmicos. Com o useEffect, desvendamos como gerenciar efeitos colaterais e interagir com o mundo exterior, sincronizando nossos componentes com o ciclo de vida da aplicação. As regras dos Hooks garantem que essa magia aconteça de forma previsível, enquanto o conceito de "Lifting State Up" e o Context API nos equipam para compartilhar estado de forma eficiente em hierarquias complexas.

Em Prática

Para aplicar o que você aprendeu, comece a pensar em cada parte da sua interface como um componente que precisa de um estado. Se um botão precisa mudar de cor ao ser clicado, use useState. Se você precisa buscar dados de uma API quando um componente é carregado, use useEffect. Lembre-se de manter o estado o mais próximo possível de onde ele é usado e de sempre criar novas cópias ao atualizar objetos e arrays. A prática constante, aliada à atenção às dependências do useEffect e à imutabilidade do estado, solidificará seu conhecimento e o transformará em um desenvolvedor React mais proficiente e eficaz.

Autoavaliação

01

Qual Hook é utilizado para adicionar estado a componentes funcionais no React?

a) useReducer
b) useContext
c) useState
d) useEffect

02

Qual é a principal função do array de dependências no useEffect?

a) Definir o valor inicial do estado.
b) Indicar quais variáveis o efeito deve observar para ser reexecutado.
c) Especificar a função de limpeza do efeito.
d) Controlar a ordem de montagem dos componentes.

03

O que significa "Lifting State Up" em React?

a) Mover o estado de um componente pai para um componente filho.
b) Centralizar o estado em um ancestral comum para compartilhá-lo entre componentes.
c) Utilizar o Context API para gerenciar o estado global.
d) Elevar a prioridade de renderização de um componente.

04

Qual das seguintes opções é uma regra fundamental dos Hooks?

a) Chamar Hooks apenas dentro de loops.
b) Chamar Hooks apenas de funções JavaScript comuns.
c) Chamar Hooks apenas no nível superior de um componente funcional ou Hook personalizado.
d) Chamar Hooks condicionalmente, dependendo de uma lógica de negócio.

05

Descreva um cenário prático onde o uso de um Hook personalizado seria vantajoso, explicando como ele resolveria um problema de duplicação de lógica.

Gabarito: 1. c) 2. b) 3. b) 4. c)

Próxima Aula

Na **Aula 24 – React: Eventos e Formulários**, aprofundaremos como os componentes React interagem com as ações do usuário, explorando o tratamento de eventos e a construção de formulários controlados e não controlados, conectando diretamente com o gerenciamento de estado que aprendemos hoje.

Recursos Adicionais

- **Documentação Oficial do React (Hooks):** Para aprofundar nos detalhes técnicos e exemplos oficiais.
- **Artigos sobre Otimização de Performance em React:** Para explorar técnicas avançadas de otimização e Core Web Vitals.
- **ESLint com eslint-plugin-react-hooks:** Para configurar seu ambiente de desenvolvimento e garantir o uso correto dos Hooks.

NOTA IMPORTANTE: As informações regulatórias/legais/técnicas desta aula estão atualizadas até 2025. Consulte sempre fontes oficiais para verificar alterações.