Apostila do curso

Construindo interfaces web interativas e reutilizáveis com React.js

Celso Yoshikazu Ishida

2025

Apresentação

Olá! Bem-vindo ao livro curso que introduz ao mundo React para a programação de aplicativos.

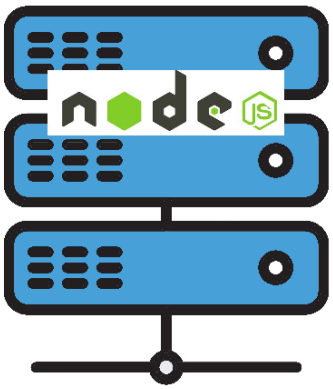
Vídeo: qrrjs0015

Este livro, revisado com auxílio de IA, resume os conceitos principais da linguagem e é complementado com os exercícios do curso básico de React ministrado pelo professor Celso Yoshikazu Ishida e os vídeos, materiais interativos e exercícios indicados pelos QrCode ao longo do livro. Este material interativo facilita na compreensão e complementa o livro que traz os conceitos importantes para a fixação. Os exercícios aparecem na ordem crescente de aprendizagem. Os exercícios são recomendados a serem feitos com o IDE + IA. O IDE (do inglês, é o Ambiente de Desenvolvimento Integrado) é um software de auxílio para a programação. Existem algumas recomendadas, veja a explicação: http://200.17.199.250/siteprototipo/curso/javascript/qrj001.html

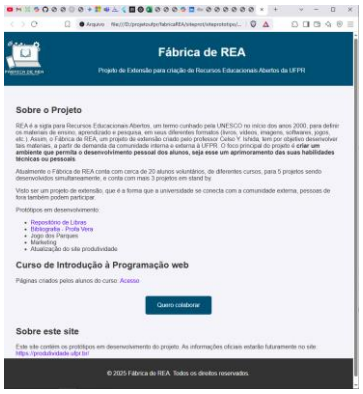
Utilizaremos um IDE e o auxílio de uma Inteligência Artificial para a programação de interfaces (Front-end), ou seja, a IA gerará código que serão executados na máquina do cliente e não no servidor da sua empresa. Aprenderemos a criar os programas em React através de prompts e mais do que orquestrar (pedir para a IA criar o código), encorajamos a você entender os detalhes desta linguagem poderosa.

Então, prepare-se para essa jornada! Vamos explorar juntos o universo do React e descobrir o poder que essa linguagem pode colocar nas suas mãos. Esta apostila deve ser lida em ordem e ela faz parte do curso. E esperamos que ao final deste, você não só entenderá o que é React, mas também será capaz de criar suas próprias páginas na web!

Vamos nessa?



E existe código Javascript que é executado no servidor (que é Node.js) e o Javascript que é executado no servidor. No livro de introdução à programação web vimos Javascript puro executado a partir do HTML. Neste livro estamos interessados na biblioteca React.JS que usa JavaScript para a lógica e uma variação do HTML (JSX) para a estrutura visual.



Vídeo: HTML e JavaScript X React.js QRRJS016

React: Otimizando a Construção de Interfaces

A afirmação de que "o React é uma biblioteca JavaScript que gera código HTML" está parcialmente correta, mas podemos ser mais precisos.

O React é, de fato, uma biblioteca JavaScript para construir interfaces de usuário (UIs). O diferencial dele não é simplesmente "gerar HTML", mas sim:

- Abstração do DOM: Em vez de você manipular o DOM diretamente (o que pode ser complexo e ineficiente em aplicações grandes), o React introduz um conceito chamado Virtual DOM. Ele mantém uma cópia leve e virtual do DOM em memória. Quando há uma mudança de estado na sua aplicação, o React atualiza essa cópia virtual, compara com o DOM real, e só então faz as modificações mínimas e mais eficientes no DOM do navegador. Isso torna o desenvolvimento mais rápido e a aplicação mais performática.
- Componentização: O React encoraja a construção de interfaces através de componentes reutilizáveis. Cada componente encapsula sua própria lógica (JavaScript) e sua própria estrutura visual (que se assemelha a HTML, mas é na verdade JSX – uma extensão de sintaxe do JavaScript que permite escrever estruturas de UI de forma declarativa, parecida com HTML).
- Natureza Declarativa: Em vez de dizer ao navegador "faça isso, depois aquilo" (abordagem imperativa), com React você descreve "como a UI deve parecer" dado um certo estado

(abordagem declarativa). O React se encarrega de fazer as atualizações necessárias para que a UI corresponda a essa descrição.

Em resumo, o React não apenas "gera HTML", ele oferece uma maneira mais eficiente e organizada de gerenciar e atualizar a interface do usuário, utilizando JavaScript e uma sintaxe que se parece com HTML (JSX) para descrever os componentes. Ele simplifica a interação com o DOM, abstraindo muitas das complexidades.

ÍNDICE

Sumário

HTML e Javascript

Para começar a aprender React JS de forma eficiente, você não precisa ser um expert em HTML e CSS, mas ter um domínio sólido dos fundamentos é crucial. React lida com a estrutura (HTML) e, em menor grau, com a apresentação (CSS) através do JSX e das suas convenções.

Aqui estão os tópicos de HTML e CSS que você precisa dominar antes de mergulhar em React. Se souber pelo menos 80% de cada conceito está ótimo.

A seguir os tópicos de HTML, leia e, se necessário, estude o assunto que você acha que precisa ser reforçado:

1. Estrutura Básica de um Documento HTML:
    - Entender a tag <!DOCTYPE html>, <html>, <head>, <body>.
    - Saber onde colocar o seu aplicativo React (geralmente dentro de um div com um ID específico, como).
2. Elementos HTML Comuns (Semânticos e Não Semânticos):
    - Estruturais: , .
    - Textuais: <h1> a <h6>, <p>, <a>, , .
    - Listas: <ul>, <ol>, <li>.
    - Imagens: <img> (com src e alt).
    - Formulários (Básico): <form>, <input> (tipos como text, password, checkbox, radio, submit), <textarea>, <select>, <option>, <label>, <button>. Entender a semântica e os atributos básicos (ex: name, id, value, placeholder).
3. Atributos HTML:
    - Compreender o uso de atributos como id, class, src, href, alt, title, data-*.
    - Importante: No React (JSX), a maioria dos atributos são escritos em *camelCase* (ex: className em vez de class, htmlFor em vez de for). Você aprenderá isso com React, mas saber o atributo original é fundamental.
4. Conceitos Semânticos:
    - Ter uma noção do porquê usar tags semânticas (ex: <header>, <nav>, <main>, <section>, <article>, <footer>).

A seguir os tópicos de CSS importantes para o React:

1. Seletores CSS:
    - Seletores de Tipo: p, h1, div.
    - Seletores de Classe: .minhaClasse.
    - Seletores de ID: #meuId.
    - Seletores de Atributo: [type="text"].
    - Combinadores: descendente (espaço), filho direto (>).
    - Pseudo-classes: :hover, :focus, :active, :first-child, :nth-child().
2. Propriedades CSS Fundamentais:
    - Box Model: margin, padding, border, content, box-sizing. Absolutamente essencial!
    - Display: block, inline, inline-block, none.
    - Posicionamento: static, relative, absolute, fixed, sticky (com top, right, bottom, left).
    - Tipografia: font-family, font-size, font-weight, line-height, text-align, color.
    - Backgrounds: background-color, background-image, background-repeat, background-position, background-size.
    - Dimensões: width, height, min-width, max-width, min-height, max-height.
    - Visual: opacity, box-shadow, border-radius, overflow.

3. Flexbox (Essencial para Layouts):
    - Container Props: display: flex, flex-direction, justify-content, align-items, flex-wrap, gap.
    - Item Props: flex-grow, flex-shrink, flex-basis, order, align-self.
    - Ser capaz de criar layouts responsivos e organizar elementos na página usando Flexbox é uma habilidade valiosa que você usará constantemente em React.
4. Grid CSS (Altamente Recomendado):
    - Embora Flexbox seja suficiente para muitos layouts, Grid é poderoso para layouts bidimensionais. Ter uma compreensão básica de display: grid, grid-template-columns, grid-template-rows, gap será um diferencial.
5. Unidades de Medida: px, %, em, rem, vw, vh.
6. Cascata, Especificidade e Herança:
    - Entender como as regras CSS são aplicadas e como o navegador decide qual estilo aplicar quando há conflitos. Isso é fundamental para depurar estilos em qualquer projeto.
7. Reset/Normalize CSS:
    - Ter uma ideia do que são e por que são usados para garantir consistência entre navegadores.

Ter uma base sólida em HTML e CSS antes de mergulhar em React é fundamental. Embora React use JSX (que parece HTML) e você vá estilizar componentes, o entendimento dos fundamentos do HTML e CSS garante que você saiba o que está sendo renderizado e como está sendo estilizado no navegador.

O que você sabe? Que tal fazer o teste para descobrir?  QRRJS017

Se não souber algum assunto não se preocupe, vamos aprendendo à medida que vamos utilizando os conceitos. E, perceba que em alguns momentos é bom parar para estudar algum tópico específico.

O que é React.js

React.js é uma **biblioteca JavaScript de código aberto** usada para construir interfaces de usuário (UI) interativas e dinâmicas. Foi criado com foco no desenvolvimento de **aplicações de página única (SPAs)**.

Características do React.js - QRRJS018

O React é considerado uma biblioteca (e não um framework) principalmente por causa do seu foco restrito e flexibilidade. Aqui está uma explicação sucinta:

- **Foco específico**: React se concentra apenas na camada de visualização (UI) da aplicação. Ele não impõe regras sobre como lidar com rotas, estado global, ou comunicação com APIs — você escolhe as ferramentas que quiser.
- **Flexibilidade**: Como biblioteca, React permite que você combine com outras bibliotecas ou frameworks para montar sua arquitetura. Frameworks, por outro lado, geralmente oferecem uma estrutura mais completa e direcionada (padronizada).

Principais características do React.js:

- **Baseado em componentes:** A UI é dividida em pequenos blocos reutilizáveis chamados **componentes**.
- **Declarativo:** Você descreve como a interface deve parecer com base no estado atual, e o React cuida de atualizar a tela conforme necessário.
- **Virtual DOM:** O React usa uma versão virtual da árvore DOM para detectar mudanças e atualizá-las de forma eficiente no navegador.
- **Unidirecional:** Os dados fluem em uma única direção, do componente pai para os filhos (**fluxo de dados unidirecional**).
- **JSX:** Uma sintaxe que mistura JavaScript com HTML para tornar a construção de interfaces mais intuitiva.

Tópicos importantes

Resumo dos tópicos QRRJS019

Aqui está um resumo dos tópicos essenciais sobre ReactJS e que serão abordados neste livro interativo:

Fundamentos do React

1. Componentes – São os blocos de construção do React. Aprender a criar componentes funcionais e componentes de classe, além de entender a composição e reutilização de componentes.

2. JSX (JavaScript XML) – Uma extensão do JavaScript que permite escrever código semelhante ao HTML dentro do JavaScript.

3. Props e State – Mecanismo para passar dados de um componente pai para um componente filho, permitindo reutilização e configuração, enquanto o State gerencia dados internos que podem ser alterados dinamicamente.

Interatividade e Estado

4. Eventos e Manipulação de Estado – Aprender a capturar eventos de usuário (ex: ‘onClick’, ‘onChange’, onSubmit, dentre outros) e modificar estados dinamicamente.

5. Hooks – Entender ‘useState’ para gerenciar estados, ‘useEffect’ para lidar com efeitos colaterais, e ‘useContext’ para compartilhamento de dados entre componentes.

Navegação e Estruturação

6. React Router – Biblioteca para criar rotas e gerenciar a navegação em Single Page Applications (SPAs), permitindo múltiplas "páginas" e passagem de parâmetros via URL.

7. Gerenciamento de Estado Global – Ferramentas como Redux ou Context API ajudam a compartilhar estados entre diferentes componentes sem precisar passar props manualmente.

Estilização e Consumo de APIs

8. Componentes Estilizados – Aprender a estilizar componentes com CSS tradicional, CSS Modules, ou bibliotecas como Styled Components.

9. Consumindo APIs – Realizar requisições HTTP usando ‘fetch’ ou ‘axios’ para obter e manipular dados dinâmicos de um backend.

Melhores Práticas e Performance

10. Otimização e Performance – Técnicas como memoization, ‘React.memo’, ‘useMemo’, e ‘useCallback’ para evitar re-renderizações desnecessárias e melhorar a eficiência da aplicação.

Esses tópicos dão uma base inicial para quem já entende HTML, CSS e JavaScript, permitindo avançar no desenvolvimento de aplicações modernas com ReactJS. Agora, que tal iniciar a conhecer mais sobre o React?

Como criar projeto React

Um app é um projeto. E o projeto é criado por um programa de linha de comando. Existem duas opções:

npm create vite@latest

npx create-react-app nomeMeuApp

Veja na como criar um projeto e quais os arquivos criados:

QRRJS004 COMO CRIAR UM PROJETO COM VITE.

Agora veja a explicação da estrutura de pastas e a ordem de execução inicial do projeto

QRRJS020 Análise dos arquivos Vite

Exercício inicial

Aqui vai um **exercício inicial prático de React** ideal para quem está começando:

Descrição: Criar um componente que exibe um número e botões para **incrementar**, **decrementar** e **zerar** esse número.

O que você vai aprender com isso:

- Criar e usar **componentes funcionais**
- Usar o **hook `useState`**
- Lidar com **eventos** (`onClick`)
- Atualizar a **interface dinamicamente**

Exemplo de código esperado:

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

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

  return (
```

```
    <div style={{ textAlign: 'center', marginTop: '50px' }}>
      <h1>Contador</h1>
      <h2>{contador}</h2>
      <button onClick={() => setContador(contador + 1)}>+</button>
      <button onClick={() => setContador(contador - 1)}>-</button>
      <button onClick={() => setContador(0)}>Zerar</button>
    </div>
  );
}

export default Contador;
```

Resumo do que o iniciante deve atentar durante esse exercício e que será estudado durante o curso:

Tópico	Estudar o quê
Componentes	Criação, exportação/importação, estrutura básica
JSX	Sintaxe, diferenças com HTML, expressões `{}`
useState	Criação de estado, atualização, uso dentro do JSX
Eventos	`onClick`, passar funções para eventos
Reatividade	Como o React atualiza a tela com `useState`

Veja a resolução do primeiro exemplo: QRRJS001

Agora, tente fazer o exercício, mas com algumas funcionalidades adicionais:

- **Desabilitar** o botão de decremento se o valor for 0.
- Estilizar os botões com CSS.
- Mostrar uma mensagem quando o contador for múltiplo de 10: `"Você chegou a um múltiplo de 10!"`

Obs: Para testar componentes o que se pede, tenha sempre um projeto inicial React em Vite para fazer os testes. Deixe o mais simples possível retirando os logos e observações da página. Só deixe uma frase ou botão como referência visual para novas inclusões ou modificação dos exercícios. Você pode usar um projeto por grupo de exercícios.

Componentes no React

Os componentes são a essência do React. Eles funcionam como blocos de comandos que, quando combinados, formam interfaces interativas e reutilizáveis. Essa abordagem modular torna o desenvolvimento mais organizado e facilita a manutenção do código. Um componente pode ser uma função ou classe que retorna um elemento da interface.

- Composição: Um componente pode incluir outros componentes dentro dele, criando hierarquias funcionais.

- Reutilização: Componentes podem ser reutilizados em diferentes partes do aplicativo, reduzindo redundância no código.

Tipos de Componentes

1. Componentes Funcionais

- São baseados em funções JavaScript.

- Simples e fáceis de escrever.

- Utilizam props para receber dados.

- Podem utilizar *hooks* como 'useState' para gerenciar estado.

Exemplo:

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

2. Componentes de Classe

- Baseados em classes ES6.

- Utilizam o método 'render()' para definir o que será exibido.

- Podem ter estado interno ('this.state') e métodos do ciclo de vida.

Exemplo:

```
class Saudacao extends React.Component {
  render() {
    return <h1>Olá, {this.props.nome}!</h1>;
  }
}
```

Conclusão

Aprender sobre componentes é essencial para dominar o React. Ao compreender seus tipos e suas aplicações, você estará preparado para criar interfaces mais organizadas, modulares e escaláveis.

Agora vamos para os exercícios?

QRRJS005

JSX – A Sintaxe Extensível do React

JSX (JavaScript XML) é uma extensão de sintaxe para JavaScript, não sendo uma linguagem de template separada, mas uma ferramenta que permite escrever código com uma aparência similar a HTML dentro de arquivos JavaScript. O React utiliza o JSX extensivamente, permitindo que os desenvolvedores descrevam a estrutura da interface de usuário de forma declarativa e familiar.

1. Sintaxe Básica

A principal característica do JSX é a capacidade de misturar HTML (ou XML) com JavaScript. Por exemplo:

JavaScript

```
const elemento = <h1>Olá, Mundo!</h1>;
```

Neste exemplo, `<h1>Olá, Mundo!</h1>` não é uma *string*; é um elemento React que o JSX compila para chamadas de função `React.createElement()`.

Para incorporar expressões JavaScript dentro do JSX, utiliza-se chaves `{}`. Isso permite inserir variáveis, chamadas de função ou qualquer outra expressão JavaScript válida diretamente na marcação:

```
const nome = "React";
const elemento = <h1>Olá, {nome}!</h1>;

function formatarSaudacao(usuario) {
  return usuario ? <h1>Bem-vindo, {usuario}!</h1> : <h1>Olá, Estranho!</h1>;
}
const usuarioLogado = "Alice";
const saudacao = formatarSaudacao(usuarioLogado);
```

2. Regras Essenciais

Embora o JSX se pareça com HTML, existem algumas regras e diferenças importantes a serem observadas:

- Atributos de Elementos:
  - Nomes de atributos em JSX são escritos em camelCase, diferentemente do HTML que usa *kebab-case*. Por exemplo, `onclick` em HTML se torna `onClick` em JSX, e `tabindex` se torna `tabIndex`.
  - A exceção mais notável e importante é o atributo `class` do HTML. Em JSX, ele deve ser substituído por `className`. Isso ocorre porque `class` é uma palavra reservada em JavaScript para definir classes.
  -

```
// HTML
<div class="container" onclick="handleClick()"></div>

// JSX
<div className="container" onClick={() => handleClick()}></div>
```

- Fechamento de Tags:

- Todas as tags devem ser explicitamente fechadas. Tags vazias (sem conteúdo interno) devem ser autocontidas, terminando com />.

JavaScript

```
// HTML válido
<img src="logo.png">
<input type="text">

// JSX válido
<img src="logo.png" alt="Logo" />
<input type="text" />
```

- Elementos Adjacentes:
  - Um componente React só pode retornar um único elemento raiz. Se você precisar retornar múltiplos elementos adjacentes, eles devem ser envolvidos por um elemento pai (como uma div) ou, preferencialmente, por um Fragmento React (<></> ou <React.Fragment></React.Fragment>). O Fragmento permite agrupar elementos sem adicionar nós extras ao DOM real.

```
// Inválido em JSX (retornando múltiplos elementos adjacentes)
// <h1>Título</h1>
// <p>Parágrafo</p>

// Válido com um elemento pai
<div>
  <h1>Título</h1>
  <p>Parágrafo</p>
</div>

// Válido com um Fragmento
<>
  <h1>Título</h1>
  <p>Parágrafo</p>
</>
```

3. Propósito e Benefícios

O principal propósito do JSX é tornar a escrita de componentes de UI mais intuitiva e legível. Ao permitir que a lógica de renderização e a marcação visual coexistam no mesmo arquivo JavaScript, o JSX promove uma abordagem de componentes mais coesa e compreensível. Ele elimina a necessidade de separar a lógica do *template* em arquivos diferentes, facilitando a manutenção e o desenvolvimento de interfaces complexas e dinâmicas.

Em suma, o JSX é uma ferramenta poderosa que melhora significativamente a experiência de desenvolvimento com React, combinando o poder do JavaScript com a familiaridade da sintaxe HTML.

Agora, vamos para os exercícios?


QRRJS003

JSX é opcional?

Sim. Você **pode usar React sem JSX**, mas quase ninguém faz isso, porque JSX deixa o código mais limpo e legível. Ferramentas como o Babel são responsáveis por traduzir o JSX para JavaScript puro antes do código ser executado.

JSX pode parecer estranho no começo, especialmente por parecer uma mistura de HTML com JavaScript. Mas com um pouco de prática, você vai perceber como ele é poderoso e facilita a criação de componentes reutilizáveis, dinâmicos e fáceis de manter.

Agora, que tal explorar como os Props e o *State* influenciam o comportamento dos componentes?

Props no React

No React, um dos pilares fundamentais da construção de interfaces dinâmicas e reutilizáveis é o conceito de Props (abreviação de "properties"). As Props permitem que componentes troquem informações entre si, promovendo modularidade e flexibilidade na estrutura do código.

O que são Props?

Props são objetos passados para um componente pai a um componente filho, permitindo que ele receba e utilize dados externos. Em React, essa comunicação é unidirecional (*one-way data flow*), ou seja, os dados fluem do componente pai para o componente filho, garantindo previsibilidade e um código mais organizado.

Principais observações sobre Props

1. Imutáveis: As Props são somente leitura no componente que as recebe, ou seja, não podem ser alteradas diretamente dentro do componente filho.

2. Passagem de Dados: Qualquer tipo de informação pode ser passado via Props, como strings, números, arrays, objetos e até funções.

3. Reutilização de Componentes: A utilização de Props ajuda a tornar os componentes mais reutilizáveis, permitindo que sejam usados em diferentes contextos apenas alterando os valores que recebem.

4. Desestruturação: Em vez de acessar as Props via ‘props.nome’, é comum utilizar a técnica de desestruturação para simplificar o código (‘const { nome, idade } = props’).

5. Default Props: É possível definir valores padrão para Props caso não sejam passadas, garantindo que o componente funcione corretamente sem dependências externas.

Exemplo básico de uso de Props

```
function Saudacao(props) {
  return <h1>Olá, {props.nome}!</h1>;
}
function App() {
  return <Saudacao nome="Carlos" />;
}
```

Neste exemplo, o componente ‘Saudacao’ recebe a propriedade ‘nome’ e exibe uma saudação personalizada. O valor "Carlos" é passado para ‘Saudacao’ através do componente ‘App’, mostrando como Props funcionam na prática.

Compreender Props é essencial para construir interfaces interativas e escaláveis no React. Este conceito forma a base para comunicação entre componentes e é frequentemente combinado com *State*, outro pilar importante da biblioteca.

Vamos para os exercícios? QRRJS006

Pronto para explorar como *State* e Props podem trabalhar juntos?

State no React

O conceito de State é fundamental no React, pois permite o gerenciamento dinâmico de dados internos de um componente. Diferente das Props, que são imutáveis e definidas pelo componente pai, o State pode ser modificado durante o ciclo de vida do componente, tornando-se essencial para interatividade e atualização da interface.

É uma estrutura de dados interna do componente React que permite que ele gerencie informações que podem mudar ao longo do tempo e afetam a renderização da interface. Quando o state de um componente é atualizado, o React detecta a mudança e re-renderiza o componente automaticamente. O state é utilizado com `useState` em componentes funcionais ou com `this.setState` em componentes de classe.

Principais conceitos sobre State

- Mutabilidade controlada: O State pode ser alterado dinamicamente, mas apenas através da função 'setState' (em componentes de classe) ou do Hook 'useState' (em componentes funcionais). Essas abordagens garantem que as alterações sejam rastreadas corretamente e acionem re-renderizações.

- Reatividade: Sempre que o State muda, o React automaticamente atualiza o componente e seus filhos relevantes, garantindo que a interface reflita os dados mais recentes sem necessidade de manipulação direta do DOM.

- Imutabilidade: Embora o State possa ser alterado, boas práticas recomendam nunca modificá-lo diretamente. Por exemplo, ao trabalhar com arrays ou objetos, sempre crie novas versões deles em vez de modificar o estado existente.

- Ciclo de vida e efeitos colaterais: O State pode ser atualizado dentro de eventos do ciclo de vida do componente ou com Hooks como 'useEffect', permitindo lidar com mudanças em resposta a ações do usuário ou chamadas de API.

Observações Importantes

1. State é local: Cada componente gerencia seu próprio estado, tornando-o isolado e independente dos demais.

2. Evite estados desnecessários: Sempre avalie se um valor precisa estar no State ou se pode ser derivado de Props ou variáveis internas.

3. Estado global e gerenciamento avançado: Para estados compartilhados entre vários componentes, ferramentas como Context API, Redux, ou Zustand podem ser mais adequadas.

Exemplo básico

Aqui está um exemplo básico de uso do **State** em um componente funcional do React usando o Hook `useState`:

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

function Contador() {
  // Definição do estado 'contador' com valor inicial 0
  const [contador, setContador] = useState(0);

  return (
    <div>
      <h2>Contador: {contador}</h2>
      <button onClick={() => setContador(contador + 1)}>Incrementar</button>
      <button onClick={() => setContador(contador - 1)}>Diminuir</button>
      <button onClick={() => setContador(0)}>Resetar</button>
    </div>
  );
}

export default Contador;
```

Explicação

- O estado `contador` é inicializado com `useState(0)`, definindo seu valor inicial como `0`.
- O botão **Incrementar** atualiza o estado aumentando `contador` em `1`.
- O botão **Diminuir** reduz o valor do estado em `1`.
- O botão **Resetar** redefine o estado para `0`.
- Sempre que `contador` muda, o React re-renderiza o componente, refletindo o novo valor na tela.

Esse é um exemplo simples, mas demonstra bem como o **State** permite a atualização dinâmica dos dados em resposta às interações do usuário!

State X variável

Uma variável em JavaScript pode armazenar dados temporários, mas não aciona uma re-renderização automática do componente quando seu valor muda. Se você modificar uma variável sem atualizar o state, o React não atualizará a interface automaticamente.

```
function MeuComponente() {
   let contador = 0;

   return (
     <div>
       <p>Contador: {contador}</p>
       <button onClick={() => contador++}>Incrementar</button>
     </div>
```

```
    );
  }
```

No exemplo acima, o botão altera a variável `contador`, mas a interface não se atualiza porque o React não detecta essa mudança.

Resumindo: use o state quando precisar que a interface se atualize automaticamente com mudanças nos dados. Se for apenas uma variável temporária que não afeta a renderização, um `let` ou `const` pode ser suficiente.

Vamos aprender através dos exercícios? QRRJS007

Eventos e Manipulação de Estado no React

Em aplicações React, os eventos desempenham um papel essencial na interação do usuário com a interface. Eles permitem capturar ações como cliques, digitação e movimentos do mouse, possibilitando a modificação dinâmica do estado para atualizar componentes em tempo real. Dominar esse conceito é crucial para desenvolver aplicações interativas e responsivas.

Principais conceitos sobre Eventos no React

- Eventos sintéticos: React utiliza um sistema de eventos próprio chamado Synthetic Events, que unifica eventos nativos dos navegadores para garantir compatibilidade e desempenho otimizado.
- Manipuladores de eventos: Funções são usadas para reagir a eventos disparados por elementos da interface. O evento é passado automaticamente como argumento para a função manipuladora.
- Atualização de Estado: Em resposta a um evento, o estado de um componente pode ser modificado com 'setState' (em componentes de classe) ou 'useState' (em componentes funcionais), garantindo que a interface reflita os dados mais recentes.

Eventos mais comuns

- 'onClick': Acionado quando o usuário clica em um elemento.
- 'onChange': Disparado quando há uma mudança em inputs, como campos de texto ou seleções.
- 'onSubmit': Executado ao enviar formulários.
- 'onMouseOver' / 'onMouseOut': Ativados quando o mouse passa sobre ou sai de um elemento.
- 'onKeyPress' / 'onKeyDown': Monitoram teclas pressionadas no teclado.

Observações importantes

1. Evite atualizações diretas do estado: Use sempre a função apropriada ('setState' ou 'useState') para modificar o estado.
2. Gerenciamento eficiente: Se muitos eventos estão sendo disparados, pode ser útil utilizar técnicas como debouncing ou throttling para otimizar o desempenho.
3. Componentes controlados: Para capturar valores de input e atualizar estados em tempo real, recomenda-se usar componentes controlados—onde o estado interno determina o valor do campo.

Ao compreender a manipulação de eventos e estado, será possível criar interfaces dinâmicas e intuitivas no React!

Vamos para os exercícios aprender? QRRJS008

Hooks no React

Os Hooks revolucionaram a maneira como os componentes funcionais gerenciam estados e efeitos colaterais no React. Antes dos Hooks, a manipulação de estado e ciclo de vida era restrita a componentes de classe. Agora, os Hooks oferecem uma abordagem mais intuitiva e modular para lidar com essas necessidades em componentes funcionais.

'useState' – Gerenciamento de Estado

O 'useState' é um Hook fundamental que permite que um componente funcional tenha estado próprio. Ele retorna um valor de estado e uma função para atualizá-lo, garantindo que a interface reaja a mudanças desse estado.

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

const incrementar = () => {
  setContador(contador + 1);
};
```

Observação importante: O React não altera diretamente a variável de estado, mas recria o componente com o novo estado, garantindo uma renderização eficiente.

'useEffect' – Lidando com Efeitos Colaterais

O 'useEffect' permite que lidemos com efeitos colaterais dentro de componentes funcionais, como chamadas de API, assinaturas de eventos e manipulações do DOM.

```
useEffect(() => {
  console.log("O componente foi montado ou atualizado!");
}, [contador]);
```

Aqui, o 'useEffect' é acionado sempre que 'contador' mudar. Se um array de dependências vazio '[]' for passado, o efeito só será executado uma vez na montagem do componente.

'useContext' – Compartilhamento de Dados Entre Componentes

O 'useContext' simplifica a transmissão de dados em árvores de componentes, evitando o problema de "prop drilling", onde propriedades precisam ser repassadas manualmente por vários níveis.

```
const TemaContext = React.createContext("claro");

const MeuComponente = () => {
  const tema = useContext(TemaContext);
  return <div style={{ background: tema === "claro" ? "#fff" : "#333" }}>Tema Atual:
{tema}</div>;
```

```
};
```

Com ‘useContext’, um componente pode acessar valores globais sem a necessidade de repassar propriedades explicitamente.

Observações Importantes

- Hooks só podem ser usados dentro de componentes funcionais ou em Hooks personalizados, nunca dentro de classes.

- A ordem de chamada dos Hooks é crucial – nunca use Hooks dentro de loops ou condicionais, pois isso pode comprometer a lógica de renderização do React.

- Melhor organização e reutilização – Com Hooks, podemos criar funções reutilizáveis para lógica de estado e efeitos, reduzindo redundância e melhorando a manutenção do código.

Vamos para os exercícios? QRRJS009

React Router

O React Router é uma biblioteca essencial para construir aplicações com múltiplas páginas e navegação dinâmica no React. Sem ele, uma aplicação React funciona como uma página única, onde a atualização da interface ocorre sem alterar a URL. Com o React Router, podemos criar rotas que permitem navegar entre diferentes partes da aplicação sem recarregar a página inteira, proporcionando uma experiência mais fluida.

Instalação e Configuração

Para utilizar o React Router, primeiro instalamos a biblioteca:

```
npm install react-router-dom
```

Depois, envolvemos nossa aplicação com o 'BrowserRouter':

```
import { BrowserRouter } from "react-router-dom";

const App = () => (
  <BrowserRouter>
    {/* Componentes da aplicação */}
  </BrowserRouter>
);
```

Definição de Rotas com 'Route' e 'Routes'

Usamos 'Routes' e 'Route' para mapear caminhos (URLs) a componentes:

```
import { Routes, Route } from "react-router-dom";

const App = () => (
  <Routes>
    <Route path="/" element={<Home />} />
    <Route path="/sobre" element={<Sobre />} />
  </Routes>
);
```

O atributo 'path' define a URL, e 'element' especifica o componente a ser renderizado quando o usuário acessa esse caminho.

Navegação com 'Link' e 'NavLink'

Em vez de usar '<a>' (que recarrega a página), utilizamos '<Link>' para navegação interna:

```
import { Link } from "react-router-dom";

const Navbar = () => (
  <nav>
    <Link to="/">Home</Link>
```

```
    <Link to="/sobre">Sobre</Link>
  </nav>
);
```

O ‘<NavLink>‘ é semelhante ao ‘<Link>‘, mas adiciona uma classe CSS ativa automaticamente quando a rota está selecionada.

Navegação Programática com ‘useNavigate’

Podemos mudar de página via código usando ‘useNavigate’:

```
import { useNavigate } from "react-router-dom";

const Pagina = () => {
  const navigate = useNavigate();

  return (
    <button onClick={() => navigate("/sobre")}>Ir para Sobre</button>
  );
};
```

Isso é útil para redirecionamentos após uma ação, como login bem-sucedido.

Observações Importantes

- React Router não recarrega a página, apenas altera os componentes visíveis.
- O caminho "/" normalmente representa a página inicial da aplicação.
- Use ‘exact’ para evitar conflitos – Sem ele, uma rota ‘"/"‘ pode coincidir com ‘"/sobre"‘ porque ambas começam igual.
- Para parâmetros dinâmicos na URL, usamos ‘useParams()’, como em ‘/user/:id’.
- Quando a página não existe, podemos criar uma rota "404" personalizada.

Com o React Router, sua aplicação React pode ter uma navegação fluida, intuitiva e dinâmica, tornando a experiência do usuário muito mais agradável! Vamos para os exercícios? QRRJS010

Introdução ao Gerenciamento de Estado Global

O gerenciamento de estado global é essencial para manter a organização e a eficiência de aplicações React complexas. Em projetos menores, podemos gerenciar estados locais diretamente dentro dos componentes com ‘useState’. No entanto, conforme a aplicação cresce, compartilhar estados entre múltiplos componentes pode se tornar um desafio, especialmente quando diversos elementos precisam acessar e modificar os mesmos dados.

Ferramentas como Redux e Context API surgem para resolver esse problema, permitindo uma forma mais centralizada e controlada de gerenciar estados.

Explorando ‘useContext’ e Suas Aplicações

O ‘useContext’ é um Hook do React que facilita o compartilhamento de dados entre componentes, evitando a necessidade de repassar propriedades manualmente de um componente pai para vários níveis de filhos. Isso resolve o problema conhecido como "prop drilling", onde propriedades precisam ser passadas repetidamente por diversas camadas da árvore de componentes.

Como Funciona o ‘useContext’?

Para utilizar o ‘useContext’, seguimos três etapas principais:

- Criar o Contexto
- Fornecer o Contexto aos componentes
- Consumir o Contexto dentro de um componente

Veja um exemplo prático:

```
import { createContext, useContext } from "react";

// 1 Criamos um contexto com um valor padrão
const TemaContext = createContext("claro");

const ExibirTema = () => {
  // 3 Usamos ‘useContext’ para acessar o valor do contexto
  const tema = useContext(TemaContext);
  return <div style={{ background: tema === "claro" ? "#fff" : "#333", color: tema
=== "claro" ? "#000" : "#fff" }}>Tema Atual: {tema}</div>;
};

const App = () => {
  return (
    // 2 Fornecemos o contexto aos componentes filhos
    <TemaContext.Provider value="escuro">
```

```
      <ExibirTema />
    </TemaContext.Provider>
  );
};

export default App;
```

Aqui, o ‘TemaContext.Provider’ fornece o valor ‘"escuro"‘ para ‘ExibirTema’, eliminando a necessidade de repassar essa informação manualmente.

Aplicações Comuns do ‘useContext’

- Gerenciamento de temas (modo claro/escuro)
- Autenticação e controle de usuário
- Configurações globais da aplicação
- Gerenciamento de idiomas

Observações e Boas Práticas

- Evite usar Contexto para estados que mudam frequentemente, pois pode causar re-renderizações desnecessárias.
- Combine ‘useContext’ com ‘useReducer’ para uma melhor organização do estado global, especialmente em aplicações grandes.
- Use contextos separados para diferentes tipos de dados – não misture temas, autenticação e preferências em um único contexto.

Boas práticas com Context API:

- **Crie múltiplos contextos:** Evite colocar todo o seu estado em um único contexto. Separe o estado em contextos lógicos para melhor organização e para evitar re-renderizações desnecessárias.
- **Use `useReducer` com Context:** Para estados mais complexos que exigem múltiplas ações e lógicas de atualização, combinar `useReducer` com a Context API é uma ótima prática. Isso oferece um padrão semelhante ao Redux, mas sem a necessidade de uma biblioteca externa.
- **Evite o "prop drilling":** A Context API foi criada justamente para resolver o problema de passar props por muitos níveis de componentes.
- **Cuidado com as re-renderizações:** Uma mudança em um provedor de contexto re-renderiza todos os consumidores desse contexto. Em aplicações muito grandes com um contexto global que muda frequentemente, isso pode ser um problema de performance.

Redux – Gerenciamento Avançado de Estado

O Redux é uma biblioteca externa que oferece uma solução mais robusta para gerenciar estados globais. Ele segue o princípio de um "estado único e imutável", onde todas as mudanças são feitas através de ações e reducers.

Fluxo básico do Redux:

- Store – Armazena o estado global.
- Actions – São eventos que descrevem mudanças no estado.
- Reducers – Funções puras que determinam como o estado é atualizado.

Exemplo simplificado de Redux com um contador:

```
const contadorReducer = (state = 0, action) => {
  if (action.type === "INCREMENTAR") {
    return state + 1;
  }
  return state;
};

// Criando a store
const store = createStore(contadorReducer);

// Despachando uma ação
store.dispatch({ type: "INCREMENTAR" });
```

Vantagens: Melhor controle, previsibilidade e escalabilidade.

Limitação: Requer mais configuração e aprendizado inicial.

Boas práticas com Redux (especialmente com Redux Toolkit):

- **Use Redux Toolkit (RTK):** É a forma oficial e recomendada de usar Redux hoje. Ele simplifica enormemente a configuração, reduz o boilerplate e inclui bibliotecas úteis como Redux Thunk (para ações assíncronas) e Immer (para imutabilidade).
- **Modularize seus slices:** Divida seu estado em "slices" lógicos, cada um com seu próprio reducer, ações e selectors. Isso mantém o código organizado e escalável.
- **Normalized State:** Para dados relacionais, normalize seu estado para facilitar a busca, atualização e evitar duplicação.
- **Imutabilidade:** O Redux depende da imutabilidade do estado. O RTK e o Immer ajudam muito com isso.
- **Utilize selectors:** Use reselect ou o `createSelector` do RTK para criar selectors memoizados. Isso evita re-renderizações desnecessárias de componentes que consomem o estado do Redux e otimiza o desempenho.
- **Middleware para efeitos colaterais:** Use middlewares como Redux Thunk ou Redux Saga para lidar com lógica assíncrona e efeitos colaterais.

Redux e Context API podem ser combinados: Algumas aplicações usam ‘Context API’ para estados menos críticos e ‘Redux’ para controle mais refinado.

Boas práticas: Manter ações e reducers organizados, evitar estados duplicados e minimizar re-renderizações desnecessárias.

Observações Importantes

Escolha a ferramenta certa para cada caso:

- Use ‘Context API’ para estados simples e globais.
- Use ‘Redux’ para aplicações maiores e estados mais complexos.

Vamos resolver exercícios? QRRJS011

Componentes Estilizados

A estilização de componentes em React é uma parte fundamental do desenvolvimento de interfaces modernas e organizadas. Existem diferentes maneiras de estilizar seus componentes, incluindo CSS tradicional, CSS Modules e Styled Components. Vamos explorar cada uma dessas abordagens e ver exemplos práticos.

CSS Tradicional

O método mais simples de aplicar estilos em componentes React é utilizando folhas de estilo CSS externas e atribuindo classes aos elementos dentro do JSX.

#Exemplo:

```css
/* styles.css */
.button {
  background-color: #007bff;
  color: white;
  padding: 10px 20px;
  border: none;
  cursor: pointer;
}
.button:hover {
  background-color: #0056b3;
}
```

Arquivo JSX:

```jsx
import React from 'react';
import './styles.css';

function Button() {
  return <button className="button">Clique Aqui</button>;
}

export default Button;
```

Esse método é simples e eficaz, mas pode levar a conflitos globais e dificuldades na manutenção em projetos grandes.

CSS Modules

Com CSS Modules, podemos criar os estilos para cada componente individualmente, evitando interferências globais.

#Exemplo:

```
/* Button.module.css */
.button {
  background-color: #28a745;
  color: white;
  padding: 10px 20px;
  border-radius: 5px;
  border: none;
  cursor: pointer;
}
```

JSX:

```
import React from 'react';
import styles from './Button.module.css';

function Button() {
  return <button className={styles.button}>Clique Aqui</button>;
}

export default Button;
```

Aqui, os estilos são localizados dentro do módulo, garantindo que apenas esse componente use a classe definida.

Styled Components

O Styled Components permite escrever estilos diretamente dentro do componente, utilizando Template Literals do JavaScript. Exemplo:

```
import styled from 'styled-components';

const Button = styled.button’
  background-color: #dc3545;
  color: white;
  padding: 10px 20px;
  border-radius: 5px;
  border: none;
  cursor: pointer;

  &:hover {
    background-color: #a71d2a;
  }
‘;

function App() {
  return <Button>Clique Aqui</Button>;
}
```

```
export default App;
```

Com Styled Components, os estilos ficam totalmente encapsulados dentro do próprio componente, permitindo um maior controle dinâmico baseado em propriedades.

Observações Importantes

- Escopo e Modularização: CSS Modules e Styled Components evitam conflitos globais e tornam o código mais modular.
- Manutenção e Reutilização: Styled Components facilita a reutilização de estilos sem a necessidade de folhas de estilo externas.
- Condicional e Dinamismo: Com Styled Components, é possível alterar estilos baseado em propriedades, como 'props.primary'.

Cada abordagem tem suas vantagens, e a escolha depende das necessidades do projeto. Agora que já vimos os conceitos principais, vamos aprofundar fazendo exercícios? QRRJS012

Consumindo APIs – Obtendo e Manipulando Dados em React

Uma das grandes vantagens do React é sua capacidade de lidar com dados dinâmicos, proporcionando aplicações interativas e atualizadas em tempo real. Para isso, muitas vezes precisamos obter informações de um backend utilizando APIs (Application Programming Interfaces). Neste capítulo, vamos explorar como consumir APIs usando 'fetch' e 'axios', duas abordagens populares para realizar requisições HTTP.

O que são APIs?

APIs (Application Programming Interfaces) permitem que aplicações troquem informações de forma estruturada, possibilitando que um sistema acesse dados ou funcionalidades de outro. Elas são usadas para buscar dados de um servidor, enviar informações ou interagir com serviços externos, como previsão do tempo, notícias ou redes sociais.

Uma forma prática de testar uma API é utilizar o navegador. Se a API for pública e disponibilizar dados em formato JSON, basta inserir a URL diretamente na barra de endereços e pressionar Enter. Se a API estiver configurada corretamente, o navegador retornará os dados, geralmente em um formato legível ou estruturado.

Por exemplo, ao acessar http://200.17.199.251:5002/librasapi/contextosdes , você pode visualizar um retorno como:

```
[
  {
    "id": 1,
    "descricao": "Geral"
  },
  {
    "id": 3,
    "descricao": "Novas Tecnologias"
  }
]
```

Para APIs que exigem autenticação ou métodos mais avançados, ferramentas como Postman, Insomnia ou até o console do navegador podem ser usadas para testes mais aprofundados. Vamos explorar como fazer essas requisições no React com 'fetch' e 'axios'.

Métodos de Requisição HTTP

Os métodos mais usados ao consumir APIs são:

- GET – Obtém dados de um servidor.
- POST – Envia dados para um servidor.
- PUT/PATCH – Atualiza informações existentes.
- DELETE – Remove dados do servidor.

‘fetch’ vs ‘axios’

Ambas as ferramentas servem para realizar requisições HTTP, mas têm diferenças significativas:

- ‘fetch’ é uma API nativa do JavaScript, baseada em Promises, permitindo fazer requisições de forma assíncrona.
- ‘axios’ é uma biblioteca externa, oferecendo funcionalidades adicionais, como tratamento automático de JSON e suporte a cancelamento de requisição.

Exemplo atualizado com ‘fetch’

```
fetch('http://200.17.199.251:5002/librasapi/contextosdes')
  .then(response => {
    if (!response.ok) {
      throw new Error(`HTTP error! status: ${response.status}`);
    }
    return response.json(); // Converte a resposta para JSON
  })
  .then(data => {
    console.log('Dados recebidos:', data); // O array completo para depuração
    data.forEach(contexto => {
      console.log('ID do Contexto:', contexto.id);
      console.log('Descrição do Contexto:', contexto.descricao);
    });
  })
  .catch(error => console.error('Erro ao buscar dados:', error));
```

Exemplo atualizado com ‘axios’

```
import axios from 'axios';

axios.get('http://200.17.199.251:5002/librasapi/contextosdes')
  .then(response => {
    // Verifica se response.data é um array e se tem pelo menos um elemento
    if (Array.isArray(response.data) && response.data.length > 0) {
      const primeiroContexto = response.data[0]; // Pega o primeiro objeto do array
      console.log('ID:', primeiroContexto.id);
      console.log('Descrição:', primeiroContexto.descricao); // Use 'descricao'
    } else {
      console.log('Nenhum contexto encontrado ou formato inesperado.');
    }
  })
  .catch(error => console.error('Erro ao buscar dados:', error));
```

Agora, além de obter os dados, os exemplos mostram como imprimir os campos individuais no console, tornando a manipulação das informações mais acessível.

Essas abordagens permitem que aplicações React integrem dados dinâmicos, tornando-as muito mais funcionais e atraentes para os usuários.

Vamos para os exercícios? QRRJS013

Boas práticas

Resumo de boas práticas para iniciantes em programação com React.js:

1. Organize seu projeto desde o início

- **Separe por pastas**: `components/`, `pages/`, `assets/`, `services/`, `hooks/` etc.
- Evite arquivos gigantes. Divida sua aplicação em componentes reutilizáveis.

2. Comece com componentes funcionais

- Prefira **funções com hooks** em vez de classes.
- Use **arrow functions** para simplificar a sintaxe:

```
const MeuComponente = () => {
  return <div>Olá!</div>;
};
```

3. Use `useState` para controlar dados internos

- Ideal para lidar com campos de formulário, contadores, visibilidade, etc.

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

4. Use `useEffect` para efeitos colaterais

- Como chamadas de API, manipulação de DOM ou sincronização com estados.

```
useEffect(() => {
  console.log("Componente montado");
}, []);
```

5. Evite misturar lógica e apresentação

- Mantenha funções de lógica separadas do JSX, sempre que possível.
- Crie arquivos utilitários ou hooks personalizados para lógica mais complexa.

6. Use `props` para tornar componentes reutilizáveis

- Passe dados e funções como propriedades para componentes filhos.

```
const Saudacao = ({ nome }) => <p>Olá, {nome}!</p>;
```

7. Dê nomes claros e significativos

- Evite nomes como `data1`, `x`, `comp2`. Use `listaUsuarios`, `BotaoEnviar`, etc.

8. Mantenha o JSX limpo e legível

- Formate seu código com indentação correta.

- Use fragmentos (`<>...</>`) para evitar divs desnecessárias.
- Use `className` no lugar de `class`.

9. Controle o estado global com moderação

- Use `Context API` apenas quando necessário.
- Prefira estado local (`useState`) sempre que possível para evitar complexidade.

10. Use `key` ao renderizar listas

- Isso ajuda o React a identificar quais itens mudaram.

```
{lista.map(item => <li key={item.id}>{item.nome}</li>)}
```

11. Evite re-renderizações desnecessárias

- Use `React.memo`, `useMemo` e `useCallback` quando perceber lentidão.
- Exemplo simples com `React.memo`:

```
const MeuComponente = React.memo(({ valor }) => {
  return <p>{valor}</p>;
});
```

12. Adote ferramentas de desenvolvimento

- Use extensões como:
  - **React Developer Tools** (inspecionar componentes)
  - **Prettier** e **ESLint** (formatação e correção automática)
- Utilize o **Vite** ou **Create React App** para iniciar projetos.

13. Estilize com estratégia

- Comece com CSS básico ou módulos (`.module.css`)
- Explore outras opções como:
  - `styled-components`
  - `Tailwind CSS`

14. Mantenha o código DRY (Don't Repeat Yourself)

- Reutilize componentes, funções e lógica.
- Crie hooks personalizados para lógicas reutilizáveis.

15. Comente quando for realmente necessário

- Use comentários apenas para explicar lógicas difíceis ou não óbvias.
- Excesso de comentários pode poluir o código.

Melhores Práticas e Performance em React – Técnica de Memoization

No desenvolvimento de aplicações React, a performance é um fator crucial para oferecer uma experiência fluida e responsiva aos usuários. Uma das técnicas mais eficazes para otimizar o desempenho da renderização é a memoization, um conceito que evita a recomputação desnecessária de funções e componentes, reduzindo o custo computacional e melhorando a eficiência da aplicação.

O que é Memoization?

A memoization é uma técnica de otimização que armazena o resultado de cálculos previamente realizados para reutilizá-los sempre que a mesma entrada for fornecida. Em React, essa abordagem é aplicada para evitar renderizações desnecessárias, garantindo que componentes e funções só sejam recalculados quando os dados realmente mudarem.

Benefícios da Memoization em React

1. Melhoria de performance: Reduz a carga de processamento ao evitar execuções repetitivas de funções.

2. Renderização eficiente: Evita que componentes sejam re-renderizados sem necessidade, economizando recursos.

3. Experiência do usuário aprimorada: Menos cálculos implicam em respostas mais rápidas da interface.

Ferramentas de Memoization em React

React oferece funções específicas para implementar memoization de forma eficaz:

- useMemo: Memoriza o retorno de funções computacionalmente caras, garantindo que apenas recalculações necessárias sejam feitas.
- useCallback: Memoriza funções para evitar que sejam recriadas em cada renderização, essencial ao lidar com handlers de eventos e callbacks em componentes filhos.
- React.memo: Otimiza componentes funcionais, impedindo a re-renderização quando suas props não mudam.

A aplicação correta dessas técnicas garante um React mais eficiente e responsivo, tornando o código mais otimizado e melhorando a experiência do usuário.

Aprofundando no ‘useMemo’

O ‘useMemo’ é um dos hooks mais poderosos de React para otimização de performance. Ele serve para memorizar o resultado de cálculos caros, evitando que sejam reexecutados desnecessariamente a cada renderização do componente.

Como funciona o ‘useMemo’?

A função ‘useMemo’ recebe dois argumentos principais:

1. Uma função de cálculo, que retorna o valor a ser memorizado.

2. Uma lista de dependências, indicando quando o valor memorizado deve ser recalculado.

Sempre que uma das dependências se altera, o cálculo é refeito. Se elas permanecem inalteradas, o valor previamente armazenado é reutilizado.

Exemplo prático

Imagine que temos uma função que faz um cálculo complexo baseado em um número fornecido pelo usuário:

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

const ExpensiveCalculation = ({ number }) => {
  const squaredNumber = useMemo(() => {
    console.log("Calculando...");
    return number  2;
  }, [number]);

  return <p>O quadrado de {number} é {squaredNumber}</p>;
};

const App = () => {
  const [num, setNum] = useState(0);

  return (
    <div>
      <input
        type="number"
        value={num}
        onChange={(e) => setNum(Number(e.target.value))}
      />
      <ExpensiveCalculation number={num} />
    </div>
  );
};

export default App;
```

O que acontece neste código?

- O cálculo do quadrado de ‘number’ só será executado se o número mudar.
- Caso contrário, o valor já calculado será reutilizado, evitando processamento desnecessário.

- Se o usuário digita rapidamente vários números no campo ‘<input>‘, apenas os valores diferentes provocam um novo cálculo, reduzindo impacto na performance.

Cuidados ao usar ‘useMemo’

Embora ‘useMemo’ seja útil, ele não deve ser usado indiscriminadamente. Algumas recomendações:

- Use quando o cálculo for computacionalmente caro e possa impactar a performance.
- Prefira utilizar em listas grandes, cálculos complexos e operações que exigem otimização.
- Evite memorização excessiva: Para funções simples, React já otimiza bem o código sem necessidade de ‘useMemo’.

Ao dominar ‘useMemo’, você melhora a eficiência dos componentes e cria aplicações mais rápidas e responsivas!

Explorando o ‘useCallback’

O ‘useCallback’ é um hook fundamental para otimização de performance em React. Ele serve para memorizar funções, evitando que sejam recriadas a cada renderização. Isso é especialmente útil ao passar funções para componentes filhos ou em eventos como ‘onClick’ e ‘onChange’.

Como funciona o ‘useCallback’?

O ‘useCallback’ recebe dois argumentos:

1. A função a ser memorizada, que será preservada na memória até que suas dependências mudem.

2. Uma lista de dependências, que determina quando a função deve ser recriada.

Se nenhuma dependência mudar, a mesma instância da função será reutilizada, evitando recriações desnecessárias.

Exemplo prático

Considere um cenário onde temos uma lista de itens e queremos filtrar os elementos sem recriar a função de filtragem a cada renderização:

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

const List = ({ filterFunction }) => {
  const items = ["React", "JavaScript", "HTML", "CSS", "Node.js"];
  return (
    <ul>
      {items.filter(filterFunction).map((item, index) => (
        <li key={index}>{item}</li>
      ))}
```

```
    </ul>
  );
};

const App = () => {
  const [search, setSearch] = useState("");

  // useCallback evita que a função seja recriada a cada renderização
  const filterItems = useCallback(
    (item) => item.toLowerCase().includes(search.toLowerCase()),
    [search]
  );

  return (
    <div>
      <input
        type="text"
        value={search}
        onChange={(e) => setSearch(e.target.value)}
      />
      <List filterFunction={filterItems} />
    </div>
  );
};

export default App;
```

O que acontece neste código?

- A função 'filterItems' é memorizada pelo 'useCallback', evitando sua recriação constante.
- A única vez que 'filterItems' será recalculada é quando 'search' mudar.
- Isso evita re-renderizações desnecessárias, garantindo melhor performance.

Quando usar 'useCallback'?

- Handlers de eventos (como 'onClick' ou 'onChange') em componentes pesados.
- Passagem de funções para componentes filhos, para evitar recriações e impactar a performance.
- Funções usadas em listas ou cálculos que podem ser otimizados.
- Evite uso excessivo! Para funções simples, React já gerencia bem a otimização sem necessidade de 'useCallback'.

Aprendendo sobre 'React.memo'

O 'React.memo' é uma função de alta ordem (HOC) usada para otimizar a renderização de componentes funcionais em React. Ele impede que um componente seja re-renderizado quando suas propriedades ('props') não mudam, reduzindo assim cálculos desnecessários e melhorando o desempenho da aplicação.

Como funciona o 'React.memo'?

Normalmente, um componente React re-renderiza sempre que seu componente pai é atualizado, mesmo que suas props não tenham mudado. O 'React.memo' resolve esse problema memorizando o componente e só permitindo que ele re-renderize se suas props realmente sofrerem alterações.

Exemplo prático

Imagine que temos um componente 'UserInfo', que recebe um nome de usuário via props:

```
import React from 'react';

// Componente otimizado com React.memo
const UserInfo = React.memo(({ name }) => {
  console.log("Renderizando UserInfo...");
  return <p>Nome do usuário: {name}</p>;
});

const App = () => {
  const [count, setCount] = React.useState(0);

  return (
    <div>
      <button onClick={() => setCount(count + 1)}>Incrementar: {count}</button>
      <UserInfo name="Carlos" />
    </div>
  );
};

export default App;
```

O que acontece neste código?

- O botão altera o estado 'count', fazendo o componente pai re-renderizar.
- Sem 'React.memo', 'UserInfo' seria renderizado novamente, mesmo sem mudanças na prop 'name'.
- Com 'React.memo', 'UserInfo' só será renderizado se a prop 'name' mudar.
- Isso evita cálculos desnecessários e melhora a performance!

Quando usar 'React.memo'?

- Componentes que recebem muitas props, mas que raramente mudam.
- Listas grandes, onde cada item tem um cálculo complexo.
- Componentes pesados que não devem ser re-renderizados sem necessidade.
- Evite uso excessivo! Se um componente já depende de estados internos, 'React.memo' pode ser desnecessário.

Ao usar ‘React.memo’ corretamente, suas aplicações React se tornam mais eficientes, evitando desperdício de processamento e mantendo a UI rápida e responsiva!

Finalizando

Ao longo deste livro introdutório sobre React, percorremos uma jornada pelos fundamentos que tornam essa biblioteca uma escolha poderosa para o desenvolvimento de interfaces modernas e dinâmicas. Desde os primeiros exercícios com componentes, JSX até a exploração das melhores práticas.

Compreendemos como os componentes estruturam a aplicação, como os props e state influenciam o comportamento dinâmico, e exploramos a importância dos eventos e hooks para interações inteligentes. Passamos pelo Router, tornando possível a navegação fluida, e pelo Contexto, que simplifica o compartilhamento de estados globais. Além disso, aprendemos técnicas de estilização com CSS, integração com APIs, e aprimoramos a performance para garantir experiências rápidas e responsivas.

Agora que você tem esse conhecimento, está pronto para construir aplicações React. Pratique, explore novas bibliotecas e continue aprendendo—o ecossistema React está em constante evolução. O próximo passo depende de você. Então, que tal começar seu próprio projeto e ver essas ideias ganharem vida? Ou venha e participe de nossos projetos de extensão: produtividade.ufpr.br ou linkedin.com/in/celsoishida

Desafio Final

Vamos para o desafio final para finalizar este curso? QRRJS014

Desempenho React.js X Javascript puro

A diferença de performance entre HTML puro + JavaScript e React.js não é um conceito simples de "um é sempre melhor que o outro". Depende muito do contexto do projeto, do tamanho, da complexidade e da forma como é implementado.

Vamos analisar os pontos chave:

HTML puro + JavaScript (Vanilla JavaScript):

- Vantagens na performance:
  - Menos Overhead: Não há camadas adicionais, bibliotecas ou frameworks para carregar e interpretar. Isso significa que, em projetos simples e estáticos, o tempo de carregamento inicial e a performance podem ser ligeiramente melhores porque o navegador está apenas renderizando o HTML e executando o JavaScript diretamente.
  - Controle Total: Você tem controle direto sobre cada manipulação do DOM. Se você for um desenvolvedor experiente e otimizar cada linha de código, pode criar aplicações extremamente performáticas em JavaScript puro, pois você decide exatamente quando e como o DOM é atualizado.
  - Ideal para sites estáticos e pequenos: Para um blog simples, um site institucional com pouca interatividade, ou landing pages, o HTML + JavaScript puro é geralmente a opção mais leve e rápida.
- Desvantagens na performance (em projetos complexos):
  - Manipulação do DOM ineficiente: Manipular o Document Object Model (DOM) diretamente de forma repetitiva pode ser custoso. Se você precisar fazer muitas atualizações no DOM em um curto período, o JavaScript puro pode levar a lentidão e "jank" (travamentos visuais) na interface. Gerenciar o estado da aplicação e as atualizações do DOM manualmente em grandes projetos se torna um pesadíssimo desafio de performance e manutenção.
  - Gerenciamento de estado complexo: Em aplicações dinâmicas com muitos dados e interações, o gerenciamento do estado da aplicação em JavaScript puro se torna complicado e propenso a erros, impactando negativamente a performance conforme a aplicação cresce.
  - Manutenção e escalabilidade: À medida que a aplicação cresce, manter e otimizar o código JavaScript puro para performance se torna muito mais difícil.

React.js:

- Vantagens na performance (em projetos complexos e dinâmicos):
  - Virtual DOM: Esta é a principal vantagem de performance do React. Em vez de atualizar o DOM real diretamente a cada mudança, o React cria uma cópia virtual do DOM (Virtual DOM). Quando o estado de um componente muda, o React compara o Virtual DOM atual com o anterior, calcula as diferenças mínimas e aplica apenas as atualizações necessárias ao DOM real. Isso minimiza o número de manipulações diretas do DOM, que são operações caras, resultando em atualizações de UI muito mais rápidas e eficientes, especialmente em aplicações complexas com muitas atualizações.
  - Componentização e Reusabilidade: O React incentiva a construção de UIs em componentes independentes e reutilizáveis. Isso não afeta diretamente a performance em tempo de execução, mas melhora a produtividade do

desenvolvedor, tornando o código mais organizado e fácil de manter e otimizar.
    - Gerenciamento de Estado Otimizado: O React oferece ferramentas e padrões para gerenciar o estado da aplicação de forma eficiente (hooks como `useState`, `useEffect`, Context API, e bibliotecas como Redux). Um bom gerenciamento de estado leva a menos renderizações desnecessárias e, consequentemente, a uma melhor performance.
    - Ecossistema e Ferramentas: O vasto ecossistema do React oferece muitas ferramentas de otimização de performance (profilers, bibliotecas para otimizar renderizações, etc.) que facilitam a identificação e correção de gargalos de performance.
    - Ideal para aplicações complexas e dinâmicas: Para Single Page Applications (SPAs), painéis de controle, e-commerces com muitas interações, o React geralmente oferece uma performance de UI superior devido à sua abordagem otimizada de atualização do DOM.
- Desvantagens na performance (em projetos simples):
    - Overhead inicial: O React.js adiciona uma camada de abstração e exige que a biblioteca seja carregada e interpretada pelo navegador. Para aplicações muito simples e estáticas, esse overhead inicial pode fazer com que o tempo de carregamento seja um pouco maior do que com JavaScript puro.
    - Curva de aprendizado: O React tem uma curva de aprendizado steeper, especialmente para desenvolvedores iniciantes, devido a conceitos como JSX, Virtual DOM, gerenciamento de estado, etc.
    - Pode ser "overkill": Para sites que não precisam de muita interatividade ou gerenciamento de estado complexo, usar React pode ser um "overkill", adicionando complexidade e peso desnecessários ao projeto.

Em resumo:

- Para projetos pequenos, estáticos e com pouca interatividade: HTML + JavaScript puro pode ser mais leve e ter um tempo de carregamento inicial mais rápido.
- Para projetos grandes, dinâmicos e com muitas interações e atualizações de UI: React.js geralmente oferece uma performance de UI superior devido ao seu Virtual DOM e ao gerenciamento otimizado de atualizações, além de melhorar a produtividade e a manutenibilidade do código.

A escolha entre os dois deve ser baseada nas necessidades específicas do seu projeto. É importante lembrar que a performance também depende muito da qualidade da implementação do código, independentemente da tecnologia utilizada. Um código React mal otimizado pode ser mais lento que um JavaScript puro bem escrito, e vice-versa.

Como Começar: Preparando seu Computador

Passo 1: Instalar o Node.js

- **O que é:** Um programa que permite rodar JavaScript no seu computador (não só no navegador)
- **Onde baixar:** nodejs.org
- **Qual versão:** Sempre a LTS (mais estável)

Passo 2: Gerenciador de Pacotes (npm ou Yarn)

O que fazem: São como "lojas de aplicativos" para código JavaScript

- **npm:** Vem automaticamente com o Node.js
- **Yarn:** Alternativa criada pelo Facebook, às vezes mais rápida

Analogia: É como ter acesso a milhares de "ingredientes" (códigos prontos) que outros programadores criaram e compartilharam.

Passo 3: criar projeto

Com isso é possível criar os projetos através do Vite ou CRA conforme o vídeo do primeiro exercício.