

</>



full stack

do zero aos primeiros projetos

@umporcentoprog



Fullstack Pro



O acesso a esse arquivo é exclusivo para você que adquiriu o Fullstack Pro. Valorize o seu dinheiro, não compartilhe com ninguém!

Olá! Muito obrigado por adquirir o pacote **FullStack Pro**, aqui nessa página do Notion, você terá acesso a todos os seus benefícios do pacote pro, acesso a bots do Telegram, links úteis, modelo de currículo, etc...

Tudo aqui foi pensado para complementar a sua jornada durante a leitura do e-book Fullstack Completo: Do zero aos primeiros projetos.

Para utilizar é muito simples, basta entrar nas categorias abaixo, e explorar os recursos que disponibilizamos para você!

Segue o link para acesso ao seu pacote de benefícios do pacote Fullstack Pro:

<https://umporcentoprogramador.notion.site/Fullstack-Pro-34f515e3751b44f3a8869174b02b28f0?pvs=74>

sinopse

Descubra o caminho para se tornar um programador fullstack completo com o ebook "Programador FullStack Completo: Do Zero aos Primeiros Projetos". Este guia abrangente e descomplicado foi criado para te levar desde os fundamentos até a construção de projetos reais, abordando tanto o front-end quanto o back-end de aplicações web.

Dividido em 14 módulos detalhados, este ebook começa com os conceitos básicos de desenvolvimento web, passando por HTML, CSS e JavaScript, e evolui para técnicas avançadas de estilização e interação com o DOM. Você aprenderá a utilizar frameworks e bibliotecas como Bootstrap e React, e mergulhará no mundo dos bancos de dados e do Node.js para criar servidores e APIs robustas.

Cada módulo inclui explicações claras, exemplos práticos e exercícios desafiadores para reforçar seu aprendizado. Além disso, projetos práticos ao longo do ebook, como a construção de um portfólio pessoal, um sistema de gerenciamento de biblioteca e um gerenciador de despesas, garantem que você aplique o conhecimento adquirido de forma prática e relevante.

Este ebook é mais do que um simples manual técnico; é um guia para transformar sua paixão pela programação em habilidades práticas e demandadas no mercado. Prepare-se para elevar seu nível de programação e dar os primeiros passos rumo a uma carreira de sucesso como programador fullstack. 

sumário

Front-end Básico	05
Javascript	88
Javascript com DOM	132
Git	192
Bootstrap	227
Banco de Dados	258
Back-end Básico com Node.js	296
React	343
Estilização avançada	384
TypeScript	430
API com Nest.js	467
React Com TypeScript	516
Projeto final	558
Docker	579

módulo 1

frontend básico

Explore os fundamentos do desenvolvimento web e crie suas primeiras páginas com HTML, CSS e muito mais! 

13 tópicos neste módulo

Módulo 1: Frontend Básico

No Capítulo 1: Frontend Básico, você aprenderá os fundamentos do desenvolvimento web, começando com uma introdução ao que é o desenvolvimento web e os componentes principais do frontend e backend. Depois, serão introduzidas às tecnologias fundamentais do frontend, como HTML, CSS e JavaScript, e entenderá a importância do desenvolvimento web. Será apresentado o HTML, incluindo sua estrutura básica e as tags mais importantes, bem como o CSS, incluindo sua sintaxe básica, seletores, propriedades principais e técnicas de layout. Ao final do capítulo, você aplicará o conhecimento adquirido criando uma página web completa utilizando HTML e CSS.

1.1 - Introdução ao Desenvolvimento Web

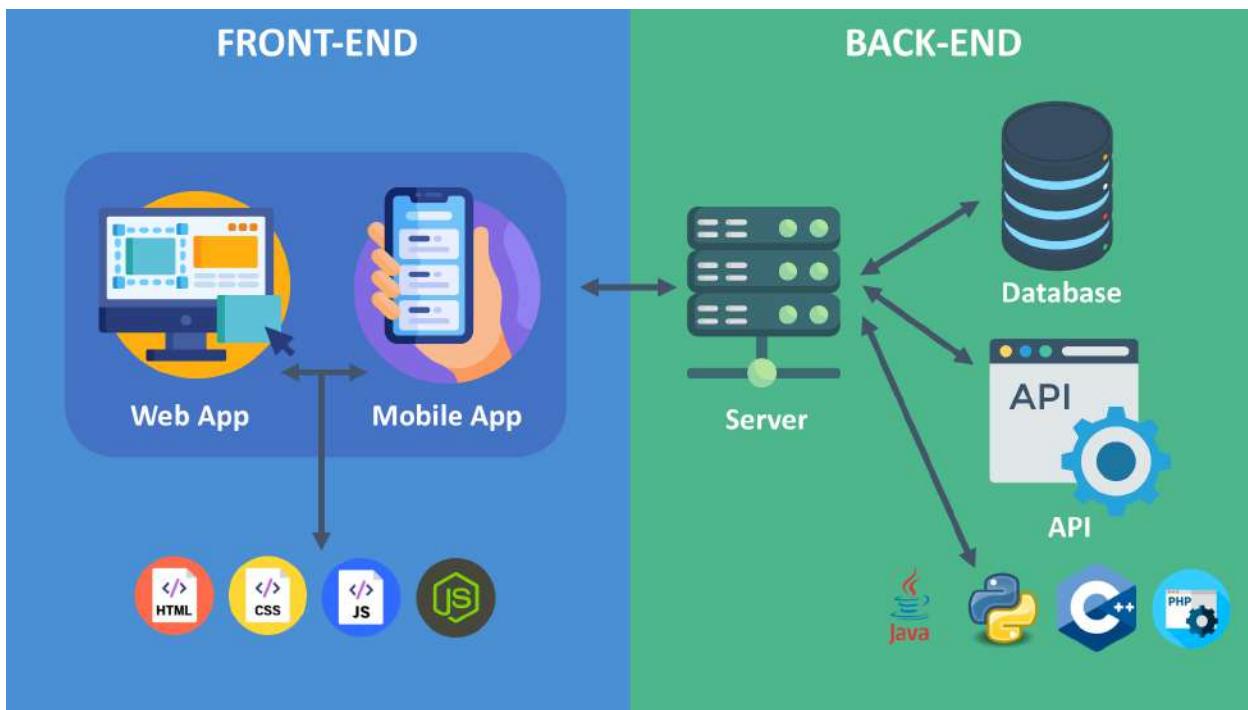
O que é Desenvolvimento Web

O desenvolvimento web é o processo de criar e manter sites e aplicativos acessíveis através da internet. É um campo que abrange uma ampla gama de atividades, desde a construção de páginas web estáticas simples até a criação de aplicações web complexas com interatividade avançada.

Componentes do Desenvolvimento Web

O desenvolvimento web pode ser dividido em duas grandes áreas:

1. **Frontend** : Também conhecido como desenvolvimento do lado do cliente, refere-se à parte do site ou aplicativo que os usuários veem e interagem diretamente. Envolve o uso de tecnologias como HTML, CSS e JavaScript para criar a interface e a experiência do usuário.
2. **Backend** : Conhecido como desenvolvimento do lado do servidor, é a parte que lida com a lógica de negócios, banco de dados, e a comunicação entre o frontend e o backend. Embora este tópico não seja o foco do nosso capítulo, é importante saber que o frontend interage com o backend para fornecer uma experiência completa ao usuário.



Tecnologias Fundamentais do Frontend

Para começar a construir páginas web, você deve estar familiarizado com três tecnologias principais:

- 1. HTML (HyperText Markup Language)** 📄: A estrutura básica de qualquer página web. HTML é usado para criar o esqueleto de uma página, incluindo textos, imagens, links e outros elementos.
- 2. CSS (Cascading Style Sheets)** 🎨: Usado para estilizar e formatar o conteúdo HTML. CSS permite definir a aparência da sua página, como cores, fontes, espaçamentos e layout.
- 3. JavaScript** 🔧: Adiciona interatividade e dinâmica à sua página web. Com JavaScript, você pode criar animações, manipular elementos da página em resposta a eventos e interagir com servidores para obter dados sem precisar recarregar a página.

Importância do Desenvolvimento Web

O desenvolvimento web é crucial porque:

- **Acessibilidade Global:** Websites e aplicativos podem ser acessados por pessoas ao redor do mundo, a qualquer hora.
- **Experiência do Usuário:** A qualidade de um site ou aplicativo pode afetar diretamente a satisfação e a retenção do usuário.
- **Oportunidades de Carreira:** Com o crescimento da tecnologia e a digitalização dos negócios, há uma demanda crescente por desenvolvedores web qualificados.

Exemplo Prático

Aqui está um exemplo básico de uma página HTML com um pouco de CSS e JavaScript:

```
<!DOCTYPE html>
<html lang="pt-BR">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Minha Primeira Página</title>
    <style>
        body {
            font-family: Arial, sans-serif;
            background-color: #f4f4f4;
            color: #333;
            text-align: center;
            padding: 20px;
        }
        .btn {
            background-color: #007BFF;
            color: #fff;
            border: none;
            padding: 10px 20px;
        }
    </style>
</head>
<body>
    <h1>Olá, Mundo!</h1>
    <p>Este é o meu primeiro projeto de Frontend.</p>
    <a href="#" class="btn">Clique aqui</a>
</body>
</html>
```

```

        font-size: 16px;
        cursor: pointer;
    }

```

```

</style>

```

```

</head>

```

```

<body>
    <h1>Bem-vindo ao Desenvolvimento Web!</h1>
    <p>Esta é a sua primeira página web.</p>
    <button class="btn" onclick="mostrarMensagem()>Clique aq
ui!</button>

```

```

<script>
    function mostrarMensagem() {
        alert('Olá, mundo!');
    }

```

```

</script>

```

```

</body>

```

```

</html>

```

Explicação do Código:

- O HTML define a estrutura da página, incluindo um título (`<title>`), uma mensagem de boas-vindas (`<h1>`), e um botão (`<button>`).
 - O CSS estiliza a página, definindo a fonte, a cor de fundo e o estilo do botão.
 - O JavaScript adiciona funcionalidade ao botão, exibindo uma mensagem de alerta quando clicado.
-

1.2 Fundamentos de HTML

O HTML (HyperText Markup Language) é a linguagem base para a criação de páginas web. Ele define a estrutura e o conteúdo de uma página, permitindo que os navegadores exibam texto, imagens, links e outros elementos. Neste tópico, vamos explorar os fundamentos do HTML, começando com a sua estrutura básica e as tags mais importantes.

1.2.1 Estrutura Básica do HTML

Uma página HTML é composta por uma série de elementos aninhados dentro de tags. A estrutura básica de um documento HTML é a seguinte:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-
  scale=1.0">
  <title>Título da Página</title>
</head>
<body>
  <h1>Olá, Mundo!</h1>
  <p>Este é um parágrafo de exemplo.</p>
</body>
</html>
```

Explicação:

- `<!DOCTYPE html>`: Declaração do tipo de documento, informando ao navegador que o documento está usando HTML5.
- `<html lang="en">`: Tag raiz que envolve todo o conteúdo da página, com um atributo `lang` para definir o idioma.
- `<head>`: Contém meta-informações sobre o documento, como codificação de caracteres e título.
- `<meta charset="UTF-8">`: Define a codificação de caracteres como UTF-8, garantindo que todos os caracteres sejam exibidos corretamente.
- `<meta name="viewport" content="width=device-width, initial-scale=1.0">`: Configura a viewport para garantir que a página seja renderizada corretamente em dispositivos móveis.
- `<title>`: Define o título da página, que aparece na aba do navegador.

- `<body>`: Contém o conteúdo visível da página, como cabeçalhos (`<h1>`), parágrafos (`<p>`), e outros elementos.

1.2.2 Tags e Atributos Importantes

HTML utiliza tags para criar elementos. Cada tag pode ter atributos que fornecem informações adicionais sobre o elemento. Vamos explorar algumas das tags e atributos mais comuns:

- **Tags de Cabeçalho:**

`<h1>` a `<h6>`: Definem cabeçalhos, do mais importante (`<h1>`) ao menos importante (`<h6>`).

```
<h1>Este é um Cabeçalho Principal</h1>
<h2>Este é um Subcabeçalho</h2>
```

- **Tags de Texto:**

`<p>`: Define um parágrafo.

`` e ``: Destacam texto com ênfase forte (negrito) ou leve (itálico).

```
<p>Este é um <strong>parágrafo</strong> com texto <em>enfatizado</em>. </p>
```

- **Tags de Link:**

`<a>`: Cria um link. O atributo `href` define o destino do link.

```
<a href="https://www.example.com">Visite o Exemplo</a>
```

- **Tags de Imagem:**

``: Insere uma imagem. O atributo `src` define a fonte da imagem, e `alt` fornece texto alternativo.

```

```

1.2.3 Estruturação de Conteúdo com HTML

O HTML permite criar uma estrutura organizada para o conteúdo, utilizando listas, tabelas e divisões:

- **Listas:**

- ``: Cria uma lista não ordenada (com marcadores).
- ``: Cria uma lista ordenada (numerada).
- ``: Define um item de lista.

```
<ul>
  <li>Item de Lista 1</li>
  <li>Item de Lista 2</li>
</ul>
```

- **Tabelas:**

- `<table>`: Cria uma tabela.
- `<tr>`: Define uma linha de tabela.
- `<td>`: Define uma célula de tabela.
- `<th>`: Define uma célula de cabeçalho.

```
<table>
  <tr>
    <th>Nome</th>
    <th>Idade</th>
  </tr>
  <tr>
    <td>João</td>
    <td>30</td>
  </tr>
</table>
```

- **Divisões e Spans:**

- `<div>`: Cria uma divisão de bloco.
- ``: Cria uma divisão em linha.

```
<div>
  Este é um bloco de texto.
  <span>Este é um texto em linha.</span>
</div>
```



Exercícios Práticos

1. **Crie um documento HTML básico que inclua um cabeçalho, um parágrafo e uma lista não ordenada.**
2. **Adicione uma imagem e um link ao seu documento HTML.**
3. **Crie uma tabela com três linhas e duas colunas, incluindo cabeçalhos.**
4. **Utilize `<div>` e `` para organizar diferentes partes de conteúdo na sua página.**
5. **Explique a diferença entre tags de bloco (`<div>`) e tags em linha (``).**

1.3 Fundamentos de CSS

O CSS (Cascading Style Sheets) é a linguagem utilizada para estilizar documentos HTML. Ele permite adicionar cores, fontes, layouts e muito mais, transformando uma página simples em uma apresentação visualmente atraente. Neste tópico, vamos explorar os fundamentos do CSS, incluindo sua sintaxe básica, seletores e propriedades principais.

1.3.1 Sintaxe Básica do CSS

O CSS é composto por regras que especificam como os elementos HTML devem ser exibidos. Cada regra é composta por um seletor e um bloco de declaração. A estrutura básica de uma regra CSS é a seguinte:

```
seletor {  
    propriedade: valor;  
}
```

Exemplo:

```
body {  
    background-color: #f0f0f0;  
    color: #333;  
    font-family: Arial, sans-serif;  
}
```

Explicação:

- **Seletor:** Especifica o elemento HTML ao qual a regra será aplicada.
- **Propriedade:** Define o aspecto do elemento que será estilizado (como `background-color` ou `color`).
- **Valor:** Especifica o valor da propriedade (como `#f0f0f0` ou `Arial, sans-serif`).

1.3.2 Adicionando um arquivo CSS no HTML

Para adicionar um arquivo CSS ao seu documento HTML, você precisa usar a tag `<link>` dentro do elemento `<head>`. A tag `<link>` deve ter os seguintes atributos:

- `rel="stylesheet"`: Indica que o arquivo é uma folha de estilo.
- `href="caminho/para/o/arquivo.css"`: Fornece o caminho para o arquivo CSS.

Aqui está um exemplo de como adicionar um arquivo CSS chamado `styles.css` ao seu documento HTML:

```
<!DOCTYPE html>
<html lang="pt-BR">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Minha Primeira Página Web</title>
    <link rel="stylesheet" href="styles.css">
  </head>
  <body>
    <!-- Conteúdo da página -->
  </body>
</html>
```

Explicação:

- A tag `<link>` é colocada dentro do elemento `<head>`.
- O atributo `href` deve conter o caminho para o arquivo CSS. No exemplo acima, o arquivo CSS está no mesmo diretório que o arquivo HTML, por isso usamos `href="styles.css"`.

Depois de adicionar a tag `<link>`, todas as regras CSS definidas no arquivo `styles.css` serão aplicadas ao seu documento HTML, permitindo que você estilize sua página de maneira mais eficiente e organizada.

1.3.3 Seletores de CSS

Os seletores são usados para selecionar os elementos HTML que serão estilizados. Existem vários tipos de seletores no CSS, incluindo:

- **Seletores de Tipo:** Selecionam elementos HTML pelo nome da tag.

```
p {
  color: blue;
}
```

- **Seletores de Classe:** Selecionam elementos que possuem um atributo de classe específico.

```
.minha-classe {  
    font-size: 18px;  
}
```

- **Seletores de ID:** Selecionam elementos que possuem um atributo de ID específico.

```
#meu-id {  
    margin: 20px;  
}
```

- **Seletores de Atributo:** Selecionam elementos que possuem um atributo específico.

```
input[type="text"] {  
    border: 1px solid #ccc;  
}
```

1.3.4 Propriedades e Valores Comuns

O CSS possui uma ampla variedade de propriedades que podem ser usadas para estilizar elementos HTML. Aqui estão algumas das mais comuns:

- **Propriedades de Cor:**

`color` : Define a cor do texto.

`background-color` : Define a cor de fundo.

```
h1 {  
    color: #ff0000;
```

```
background-color: #000000;  
}
```

- **Propriedades de Texto:**

`font-family` : Define a fonte do texto.

`font-size` : Define o tamanho da fonte. `text-align` : Define o alinhamento do texto.

```
p {  
  font-family: 'Arial', sans-serif;  
  font-size: 16px;  
  text-align: justify;  
}
```

- **Propriedades de Margem e Padding:**

`margin` : Define a margem externa do elemento. `padding` : Define o espaçamento interno do elemento.

```
div {  
  margin: 20px;  
  padding: 10px;  
}
```

- **Propriedades de Bordas:**

`border` : Define a borda do elemento.

```
img {  
  border: 2px solid #000;  
}
```

1.3.5 Modelos de Caixa (Box Model)

O modelo de caixa é um conceito fundamental no CSS. Cada elemento é representado como uma caixa retangular, composta por quatro partes principais:

- **Content:** O conteúdo real da caixa, como texto ou imagem.
- **Padding:** Espaço entre o conteúdo e a borda.
- **Border:** Borda ao redor do padding (se definida).
- **Margin:** Espaço entre a borda e os elementos vizinhos.

Exemplo:

```
div {  
    width: 300px;  
    padding: 10px;  
    border: 5px solid #ccc;  
    margin: 20px;  
}
```

1.3.6 Layouts e Posicionamento

O CSS permite criar layouts complexos utilizando diferentes técnicas de posicionamento:

- **Posicionamento Estático:** Valor padrão. Os elementos são posicionados de acordo com o fluxo normal do documento.
- **Posicionamento Relativo:** O elemento é posicionado em relação à sua posição normal.
- **Posicionamento Absoluto:** O elemento é posicionado em relação ao seu primeiro elemento pai posicionado.
- **Posicionamento Fixo:** O elemento é posicionado em relação à viewport do navegador.
- **Flexbox:** Sistema de layout unidimensional que facilita a criação de layouts flexíveis.

- **Grid Layout:** Sistema de layout bidimensional que facilita a criação de layouts complexos.

Exemplo de Flexbox:

```
.container {  
    display: flex;  
    justify-content: center;  
    align-items: center;  
    height: 100vh;  
}  
  
.item {  
    background-color: #ff0000;  
    padding: 20px;  
    margin: 10px;  
}
```



Exercícios Práticos

1. Crie uma regra CSS que altere a cor de fundo do corpo da página para azul e a cor do texto para branco.
2. Utilize seletores de classe para estilizar três elementos diferentes com diferentes tamanhos de fonte.
3. Crie uma caixa com 200px de largura, 200px de altura, 10px de padding, 5px de borda e 20px de margem.
4. Estilize uma lista não ordenada utilizando Flexbox para alinhar os itens ao centro.
5. Explique a diferença entre `margin` e `padding` no modelo de caixa.

1.4 Criando sua Primeira Página Web

Agora que você já entende os fundamentos de HTML e CSS, é hora de colocar esse conhecimento em prática criando sua primeira página web. Vamos construir uma página simples, mas completa, utilizando as tecnologias que aprendemos até agora. Este projeto integrará HTML para estrutura e CSS para estilização.

1.4.1 Estrutura Básica da Página

Começaremos criando a estrutura básica da nossa página web. Vamos adicionar um cabeçalho, um parágrafo e uma lista.

```
<!DOCTYPE html>
<html lang="pt-BR">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-
  scale=1.0">
  <title>Minha Primeira Página Web</title>
  <link rel="stylesheet" href="styles.css">
</head>
<body>
  <header>
    <h1>Bem-vindo à Minha Primeira Página Web</h1>
    <nav>
      <ul>
        <li><a href="#sobre">Sobre</a></li>
        <li><a href="#servicos">Serviços</a></li>
        <li><a href="#contato">Contato</a></li>
      </ul>
    </nav>
  </header>
  <main>
    <section id="sobre">
      <h2>Sobre Mim</h2>
```

```

<p>Meu nome é João e sou um desenvolvedor web em formaç
ão.</p>
</section>
<section id="servicos">
  <h2>Serviços</h2>
  <ul>
    <li>Desenvolvimento de Websites</li>
    <li>Design Responsivo</li>
    <li>Otimização de Performance</li>
  </ul>
</section>
</main>
<footer>
  <p>&copy; 2024 João. Todos os direitos reservados.</p>
</footer>
</body>
</html>

```

Explicação:

- **Cabeçalho:** Contém o título principal (`<h1>`) e um menu de navegação (`<nav>`) com links para diferentes seções da página.
- **Corpo Principal:** Inclui duas seções (`<section>`), uma para "Sobre Mim" e outra para "Serviços".
- **Rodapé:** Contém um parágrafo com informações de direitos autorais.

1.4.2 Estilizando a Página com CSS

Agora, vamos adicionar um arquivo CSS para estilizar nossa página. Crie um arquivo chamado `styles.css` e adicione o seguinte código:

```

/* Reset de Estilos */
* {
  margin: 0;
  padding: 0;
}

```

```
    box-sizing: border-box;
}

/* Estilos Gerais */
body {
    font-family: Arial, sans-serif;
    line-height: 1.6;
    background-color: #f4f4f4;
    color: #333;
    padding: 20px;
}

/* Cabeçalho */
header {
    background: #333;
    color: #fff;
    padding: 20px 0;
    text-align: center;
}

header h1 {
    margin-bottom: 10px;
}

header nav ul {
    list-style: none;
    padding: 0;
}

header nav ul li {
    display: inline;
    margin-right: 10px;
}

header nav ul li a {
    color: #fff;
```

```

    text-decoration: none;
}

/* Seções */
section {
    margin-bottom: 20px;
}

section h2 {
    margin-bottom: 10px;
}

section p, section ul {
    margin-bottom: 10px;
}

section ul {
    list-style-type: disc;
    margin-left: 20px;
}

/* Rodapé */
footer {
    text-align: center;
    padding: 10px 0;
    background: #333;
    color: #fff;
}

```

Explicação:

- **Reset de Estilos:** Remove as margens e paddings padrão e define o box-sizing para border-box para facilitar o controle dos tamanhos dos elementos.
- **Estilos Gerais:** Define a fonte padrão, a cor de fundo, a cor do texto e o espaçamento padrão para o corpo da página.

- **Cabeçalho:** Define a cor de fundo, a cor do texto e o alinhamento do texto para o cabeçalho.
- **Seções:** Define o espaçamento entre as seções e estiliza os cabeçalhos e listas dentro delas.
- **Rodapé:** Define a cor de fundo, a cor do texto e o alinhamento do texto para o rodapé.

1.4.3 Revisão e Melhorias

Revise o código HTML e CSS para garantir que tudo esteja funcionando corretamente. Experimente adicionar mais seções, alterar as cores, fontes e layouts para personalizar sua página. Aqui estão algumas sugestões de melhorias:

- **Adicione uma imagem de perfil na seção "Sobre Mim".**
- **Crie um formulário de contato na seção "Contato".**
- **Adicione ícones de redes sociais no rodapé.**



Exercícios Práticos

1. **Crie um novo arquivo HTML com a estrutura básica aprendida e adicione um cabeçalho, parágrafo e lista.**
2. **Estilize sua página usando um arquivo CSS separado, alterando as cores e fontes.**
3. **Adicione uma nova seção "Projetos" com uma lista de projetos fictícios que você gostaria de desenvolver.**
4. **Insira uma imagem de perfil na seção "Sobre Mim" e estilize-a com CSS.**
5. **Crie um formulário de contato na seção "Contato" com campos para nome, e-mail e mensagem, e estilize-o com CSS.**

1.5 Trabalhando com Imagens e Links

Neste tópico, vamos aprender como adicionar imagens e links às suas páginas web, tornando-as mais dinâmicas e interativas. As imagens ajudam a ilustrar conteúdo e a tornar a experiência do usuário mais agradável, enquanto os links permitem a navegação entre diferentes páginas e recursos na web.

1.5.1 Adicionando Imagens

Para adicionar uma imagem a uma página HTML, usamos a tag ``. Essa tag é uma tag auto-fechada, o que significa que ela não precisa de uma tag de fechamento. Vamos explorar os atributos mais comuns da tag ``:

- **src:** Especifica a URL da imagem.
- **alt:** Fornece um texto alternativo que é exibido se a imagem não puder ser carregada.

Exemplo:

```

```

Explicação:

- **src:** Este atributo aponta para a localização da imagem. Pode ser um caminho relativo (se a imagem estiver no mesmo diretório do arquivo HTML) ou um URL absoluto.
- **alt:** Este atributo fornece uma descrição textual da imagem, que é útil para acessibilidade e SEO.

1.5.2 Dimensões e Estilização de Imagens

Você pode controlar as dimensões de uma imagem utilizando os atributos `width` e `height`, ou aplicando estilos CSS:

Exemplo com Atributos HTML:

```

```

Exemplo com CSS inline:

```

```

Exemplo com CSS Externo:

```
/* styles.css */
img {
    width: 300px;
    height: auto; /* Mantém a proporção da imagem */
}
```

1.5.3 Adicionando Links

Os links são adicionados usando a tag `<a>`. O atributo mais importante dessa tag é o `href`, que especifica o destino do link.

Exemplo de Link para uma Página Externa:

```
<a href="https://www.exemplo.com">Visite o Exemplo</a>
```

Exemplo de Link para uma Página Interna:

```
<a href="sobre.html">Sobre Mim</a>
```

1.5.4 Links Internos e Âncoras

Você pode criar links que navegam para diferentes partes da mesma página usando âncoras. Para isso, você define um `id` no elemento de destino e cria um

link que aponta para esse `id`.

Exemplo de Link Interno:

```
<!-- Destino da âncora -->
<h2 id="secao-sobre">Sobre Mim</h2>
<p>Meu nome é João...</p>

<!-- Link para a âncora -->
<a href="#secao-sobre">Ir para a Seção Sobre Mim</a>
```

1.5.5 Imagens como Links

Você pode usar imagens como links simplesmente aninhando uma tag `` dentro de uma tag `<a>`.

Exemplo:

```
<a href="<https://www.exemplo.com>">
  
</a>
```

1.5.6 Atributos Adicionais para Links

Os links podem ter atributos adicionais que modificam seu comportamento:

- **target**: Define onde abrir o link. Por exemplo, `target="_blank"` abre o link em uma nova aba.
- **title**: Fornece um texto descritivo que aparece quando o usuário passa o cursor sobre o link.

Exemplo com Atributos Adicionais:

```
<a href="<https://www.exemplo.com>" target="_blank" title="Visite o Exemplo">Visite o Exemplo</a>
```



Exercícios Práticos

1. Adicione uma imagem à sua página HTML usando o atributo `src` e forneça uma descrição com o atributo `alt`.
 2. Estilize a imagem para ter 400px de largura e altura proporcional usando CSS.
 3. Crie um link para uma página externa (por exemplo, <https://www.google.com>) e um link para uma página interna chamada `contato.html`.
 4. Adicione âncoras à sua página para criar links internos que navegam para diferentes seções da página.
 5. Crie um link que usa uma imagem como elemento clicável, e adicione atributos `target="_blank"` e `title` para o link.
-

1.6 Trabalhando com Listas e Tabelas

Listas e tabelas são ferramentas poderosas no HTML para organizar e apresentar informações de forma clara e estruturada. Elas são usadas frequentemente para mostrar dados, criar menus de navegação, listas de tarefas, entre outros. Neste tópico, vamos explorar como criar e estilizar listas e tabelas no HTML.

1.6.1 Listas Não Ordenadas

As listas não ordenadas são usadas para agrupar itens que não possuem uma ordem específica. A tag `` (unordered list) é usada para criar uma lista não ordenada, e cada item da lista é definido pela tag `` (list item).

Exemplo de Lista Não Ordenada:

```
<ul>
  <li>Item 1</li>
  <li>Item 2</li>
```

```
<li>Item 3</li>
</ul>
```

1.6.2 Listas Ordenadas

As listas ordenadas são usadas para agrupar itens que possuem uma sequência específica. A tag `` (ordered list) é usada para criar uma lista ordenada, e cada item da lista é definido pela tag ``.

Exemplo de Lista Ordenada:

```
<ol>
  <li>Primeiro Item</li>
  <li>Segundo Item</li>
  <li>Terceiro Item</li>
</ol>
```

1.6.3 Listas Aninhadas

Você pode aninhar listas dentro de outras listas para criar hierarquias mais complexas.

Exemplo de Listas Aninhadas:

```
<ul>
  <li>Item 1</li>
  <li>Item 2
    <ul>
      <li>Subitem 1</li>
      <li>Subitem 2</li>
    </ul>
  </li>
  <li>Item 3</li>
</ul>
```

1.6.4 Estilizando Listas

As listas podem ser estilizadas utilizando CSS para alterar marcadores, espaçamento, cores e muito mais.

Exemplo de Estilização de Lista:

```
ul {  
    list-style-type: square; /* Define o tipo de marcador */  
    padding-left: 20px; /* Adiciona espaço à esquerda da lista */  
}  
  
ol {  
    list-style-type: decimal; /* Define o tipo de marcador para lista ordenada */  
    padding-left: 20px;  
}  
  
li {  
    margin-bottom: 10px; /* Adiciona espaço entre os itens da lista */  
}
```

1.6.5 Criando Tabelas

As tabelas são usadas para exibir dados tabulares de forma organizada. Uma tabela é criada com a tag `<table>`, e seus conteúdos são organizados em linhas (`<tr>`) e células (`<td>` para dados e `<th>` para cabeçalhos).

Exemplo de Tabela:

```
<table>  
  <thead>  
    <tr>
```

```

<th>Nome</th>
<th>Idade</th>
<th>Cidade</th>
</tr>
</thead>
<tbody>
<tr>
<td>João</td>
<td>30</td>
<td>São Paulo</td>
</tr>
<tr>
<td>Maria</td>
<td>25</td>
<td>Rio de Janeiro</td>
</tr>
</tbody>
</table>

```

1.6.6 Estruturando Tabelas

As tabelas podem ser divididas em cabeçalho (`<thead>`), corpo (`<tbody>`) e rodapé (`<tfoot>`).

Exemplo de Estrutura de Tabela Completa:

```

<table>
<thead>
<tr>
<th>Produto</th>
<th>Preço</th>
<th>Quantidade</th>
</tr>
</thead>
<tbody>

```

```

<tr>
  <td>Arroz</td>
  <td>R$ 20,00</td>
  <td>2</td>
</tr>
<tr>
  <td>Feijão</td>
  <td>R$ 10,00</td>
  <td>1</td>
</tr>
</tbody>
<tfoot>
  <tr>
    <td>Total</td>
    <td>R$ 30,00</td>
    <td>3</td>
  </tr>
</tfoot>
</table>

```

1.6.7 Estilizando Tabelas

As tabelas podem ser estilizadas para melhorar a legibilidade e a aparência. Você pode usar CSS para adicionar bordas, cores de fundo, espaçamento e muito mais.

Exemplo de Estilização de Tabela:

```

table {
  width: 100%;
  border-collapse: collapse; /* Remove espaços entre as células */
}

th, td {
  border: 1px solid #ddd; /* Adiciona bordas às células */
}

```

```
padding: 8px; /* Adiciona espaçamento interno */
text-align: left; /* Alinha o texto à esquerda */
}

th {
    background-color: #f2f2f2; /* Adiciona cor de fundo aos cab
eçalhos */
}

tr:nth-child(even) {
    background-color: #f9f9f9; /* Adiciona cor de fundo às linh
as pares */
}
```



Exercícios Práticos

- 1. Crie uma lista não ordenada com três itens, e uma lista ordenada com três passos.**
- 2. Crie uma tabela com três colunas (Produto, Preço, Quantidade) e três linhas de dados.**
- 3. Estilize a lista não ordenada para usar marcadores quadrados e adicionar espaço entre os itens.**
- 4. Estilize a tabela para ter bordas, espaço interno e cores de fundo alternadas nas linhas.**
- 5. Crie uma lista aninhada com uma lista não ordenada dentro de um item de lista ordenada.**

1.7 Estilizando a Página com CSS (Box Model, Backgrounds, Display)

Neste tópico, vamos explorar técnicas avançadas de estilização utilizando CSS, focando em conceitos fundamentais como o Box Model, a aplicação de backgrounds e as diferentes propriedades de display. Esses conceitos são cruciais para criar layouts flexíveis e responsivos.

1.7.1 Box Model

O Box Model é um conceito central no CSS que descreve a estrutura e o espaçamento de elementos HTML. Cada elemento é representado como uma caixa retangular, composta por quatro partes principais:

- **Content (Conteúdo):** A área onde o conteúdo (texto, imagem, etc.) é exibido.
- **Padding (Preenchimento):** Espaço entre o conteúdo e a borda do elemento.
- **Border (Borda):** A borda ao redor do padding e do conteúdo.
- **Margin (Margem):** Espaço entre a borda do elemento e os elementos vizinhos.

Exemplo de Box Model:

```
div {  
    width: 200px;  
    height: 100px;  
    padding: 10px;  
    border: 5px solid #333;  
    margin: 20px;  
}
```

Explicação:

- **Width e Height:** Define a largura e a altura do conteúdo.
- **Padding:** Adiciona espaço interno ao redor do conteúdo.
- **Border:** Define a borda ao redor do padding.
- **Margin:** Adiciona espaço externo ao redor da borda.

1.7.2 Backgrounds

Os backgrounds são usados para adicionar cores, imagens ou gradientes ao fundo de um elemento. Vamos explorar as propriedades mais comuns para manipular backgrounds:

- **background-color:** Define a cor de fundo de um elemento.
- **background-image:** Define uma imagem de fundo.
- **background-repeat:** Controla a repetição da imagem de fundo.
- **background-position:** Define a posição da imagem de fundo.
- **background-size:** Ajusta o tamanho da imagem de fundo.

Exemplo de Background:

```
body {  
    background-color: #f0f0f0;  
}  
  
div {  
    background-image: url('imagem.jpg');  
    background-repeat: no-repeat;  
    background-position: center;  
    background-size: cover;  
}
```

1.7.3 Display

A propriedade `display` define como um elemento é exibido na página. Existem vários valores para a propriedade `display`, mas os mais comuns são:

- **block:** O elemento é exibido como um bloco, ocupando a largura total disponível.
- **inline:** O elemento é exibido em linha com outros elementos, ocupando apenas a largura necessária.

- **inline-block**: O elemento é exibido em linha, mas permite definir largura e altura.
- **none**: O elemento não é exibido.

Exemplo de Propriedade Display:

```
.block {  
  display: block;  
}  
  
.inline {  
  display: inline;  
}  
  
.inline-block {  
  display: inline-block;  
}  
  
.none {  
  display: none;  
}
```

Aplicação Prática:

```
<div class="block">Este é um elemento block</div>  
<span class="inline">Este é um elemento inline</span>  
<span class="inline-block">Este é um elemento inline-block</span>  
<div class="none">Este elemento não será exibido</div>
```

1.7.4 Trabalhando com Layouts

Combinar o Box Model, backgrounds e propriedades de display permite criar layouts complexos e responsivos. Vamos ver um exemplo de layout básico utilizando essas técnicas:

HTML:

```
<div class="container">
  <header class="header">Cabeçalho</header>
  <nav class="nav">Navegação</nav>
  <main class="main">Conteúdo Principal</main>
  <aside class="aside">Barra Lateral</aside>
  <footer class="footer">Rodapé</footer>
</div>
```

CSS:

```
/* Reset de Estilos */
* {
  margin: 0;
  padding: 0;
  box-sizing: border-box;
}

/* Layout Container */
.container {
  display: grid;
  grid-template-areas:
    'header header header'
    'nav main aside'
    'footer footer footer';
  grid-gap: 10px;
  padding: 10px;
}

.header {
  grid-area: header;
```

```
background-color: #333;
color: #fff;
padding: 20px;
text-align: center;
}

.nav {
grid-area: nav;
background-color: #f4f4f4;
padding: 20px;
}

.main {
grid-area: main;
background-color: #fff;
padding: 20px;
}

.aside {
grid-area: aside;
background-color: #f4f4f4;
padding: 20px;
}

.footer {
grid-area: footer;
background-color: #333;
color: #fff;
padding: 20px;
text-align: center;
}
```



Exercícios Práticos

1. Crie uma `div` com uma largura de 300px, altura de 150px, padding de 20px, borda de 5px e margem de 10px.
 2. Adicione uma cor de fundo azul a uma `div` e uma imagem de fundo com repetição desativada.
 3. Crie um layout básico usando `display: block`, `display: inline` e `display: inline-block`.
 4. Experimente usar `display: none` para ocultar um elemento e depois torná-lo visível novamente.
 5. Crie um layout de página simples utilizando o Grid Layout, com cabeçalho, navegação, conteúdo principal, barra lateral e rodapé.
-

1.8 Elementos de Formulário em HTML

Os formulários são uma parte essencial de muitas aplicações web, permitindo que os usuários enviem informações para o servidor. Neste tópico, vamos explorar os elementos de formulário mais comuns em HTML e como estilizar e validar esses formulários para uma melhor experiência do usuário.

1.8.1 Estrutura Básica de um Formulário

A estrutura básica de um formulário em HTML é criada usando a tag `<form>`, que pode conter vários elementos de entrada como `<input>`, `<textarea>`, `<select>` e `<button>`. O formulário também deve especificar atributos como `action` (URL para onde os dados do formulário serão enviados) e `method` (método de envio, como GET ou POST).

Exemplo de Formulário Básico:

```
<form action="/submit" method="post">
  <label for="nome">Nome:</label>
  <input type="text" id="nome" name="nome" required>
```

```

<label for="email">Email:</label>
<input type="email" id="email" name="email" required>

<label for="mensagem">Mensagem:</label>
<textarea id="mensagem" name="mensagem" rows="4" cols="50"
required></textarea>

<button type="submit">Enviar</button>
</form>

```

Explicação:

- **<label>**: Define rótulos para os elementos do formulário, melhorando a acessibilidade.
- **<input>**: Elemento de entrada para dados, com tipos diferentes como `text`, `email`, `password`, etc.
- **<textarea>**: Área de texto para entradas maiores.
- **<button>**: Botão para enviar o formulário.

1.8.2 Tipos de Input Comuns

Os inputs são os elementos de formulário mais utilizados e possuem vários tipos que definem seu comportamento e aparência. Alguns dos tipos mais comuns incluem:

- **text**: Entrada de texto padrão.
- **password**: Entrada de senha, com caracteres mascarados.
- **email**: Validação de formato de email.
- **number**: Entrada de número com opções de incremento/decremento.
- **date**: Seleção de data através de um calendário.
- **checkbox**: Opção de seleção múltipla.
- **radio**: Opção de seleção única dentro de um grupo.

- **submit**: Botão para enviar o formulário.

Exemplo de Diversos Tipos de Input:

```

<form>
  <label for="username">Nome de Usuário:</label>
  <input type="text" id="username" name="username" required>

  <label for="senha">Senha:</label>
  <input type="password" id="senha" name="senha" required>

  <label for="email">Email:</label>
  <input type="email" id="email" name="email" required>

  <label for="idade">Idade:</label>
  <input type="number" id="idade" name="idade" min="1" max="100">

  <label for="data">Data de Nascimento:</label>
  <input type="date" id="data" name="data">

  <label for="newsletter">Inscrever-se na Newsletter:</label>
  <input type="checkbox" id="newsletter" name="newsletter">

  <p>Sexo:</p>
  <input type="radio" id="masculino" name="sexo" value="masculino">
  <label for="masculino">Masculino</label>
  <input type="radio" id="feminino" name="sexo" value="feminino">
  <label for="feminino">Feminino</label>

  <button type="submit">Enviar</button>
</form>

```

1.8.3 Elementos de Seleção

Os elementos de seleção permitem que o usuário escolha uma ou mais opções de uma lista predefinida. Os principais elementos de seleção são `<select>` e `<option>` para listas suspensas e `<datalist>` para listas de sugestões.

Exemplo de Elemento de Seleção:

```
<label for="pais">País:</label>
<select id="pais" name="pais">
  <option value="brasil">Brasil</option>
  <option value="eua">Estados Unidos</option>
  <option value="japao">Japão</option>
</select>
```

1.8.4 Elementos de Área de Texto

Para entradas de texto maiores, utilizamos a tag `<textarea>`, que permite ao usuário digitar várias linhas de texto.

Exemplo de Área de Texto:

```
<label for="comentarios">Comentários:</label>
<textarea id="comentarios" name="comentarios" rows="4" cols="50"></textarea>
```

1.8.5 Estilizando Formulários

Os formulários podem ser estilizados com CSS para melhorar a aparência e a usabilidade. Vamos ver um exemplo de estilização básica para um formulário.

CSS para Estilização de Formulários:

```
form {
  max-width: 600px;
  margin: 0 auto;
```

```
padding: 20px;
border: 1px solid #ccc;
background-color: #f9f9f9;
border-radius: 5px;
}

label {
  display: block;
  margin-bottom: 5px;
  font-weight: bold;
}

input, textarea, select, button {
  width: 100%;
  padding: 10px;
  margin-bottom: 10px;
  border: 1px solid #ccc;
  border-radius: 3px;
}

button {
  background-color: #5cb85c;
  color: white;
  border: none;
  cursor: pointer;
}

button:hover {
  background-color: #4cae4c;
}
```

1.8.6 Validação de Formulários

A validação de formulários é crucial para garantir que os dados enviados são corretos e completos. A validação pode ser feita tanto no lado do cliente (frontend) quanto no lado do servidor (backend). Em HTML5, podemos utilizar atributos como `required`, `min`, `max`, `pattern` e `type` para validação básica no cliente.

Exemplo de Validação de Formulário:

```
<form>
  <label for="email">Email:</label>
  <input type="email" id="email" name="email" required>

  <label for="idade">Idade:</label>
  <input type="number" id="idade" name="idade" min="18" max="100" required>

  <label for="senha">Senha:</label>
  <input type="password" id="senha" name="senha" pattern=".{6,}" title="A senha deve ter pelo menos 6 caracteres" required>

  <button type="submit">Enviar</button>
</form>
```



Exercícios Práticos

- 1. Crie um formulário que inclua campos para nome, email, telefone e mensagem, e adicione validação básica usando HTML5.**
- 2. Estilize o formulário para que ele tenha uma aparência profissional e amigável ao usuário.**
- 3. Adicione um campo de seleção de país com opções para Brasil, Estados Unidos e Japão.**
- 4. Crie um grupo de botões de rádio para selecionar o gênero (Masculino, Feminino, Outro).**

- 5. Implemente um campo de senha com validação que exija pelo menos 8 caracteres, incluindo letras e números.**
-

1.9 Projeto 1 - Portfólio

Neste projeto, vamos criar um portfólio pessoal que inclui uma seção sobre você, seus serviços, projetos e uma forma de contato. Vamos construir o projeto passo a passo, explicando cada parte e adicionando a estilização correspondente.

1.9.1 Estrutura Inicial do HTML

Começamos criando a estrutura básica do HTML para o nosso portfólio. Vamos criar o esqueleto da página HTML.

Passo 1: Head

```
<!DOCTYPE html>
<html lang="pt-BR">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <title>Portfólio Pessoal</title>
  <link rel="stylesheet" href="styles.css">
</head>
<body>
</body>
</html>
```

Explicação:

- **DOCTYPE:** Declaração do tipo de documento, informando ao navegador que estamos utilizando HTML5.
- **<html lang="pt-BR">:** Tag raiz que envolve todo o conteúdo da página, com um atributo `lang` para definir o idioma.

- **<meta charset="UTF-8">**: Define a codificação de caracteres como UTF-8, garantindo que todos os caracteres sejam exibidos corretamente.
- **<meta name="viewport" content="width=device-width, initial-scale=1.0">**: Configura a viewport para garantir que a página seja renderizada corretamente em dispositivos móveis.
- **<title>**: Define o título da página, que aparece na aba do navegador.
- **<link rel="stylesheet" href="styles.css">**: Inclui o arquivo CSS para estilização da página.

1.9.2 Criando e Estilizando o Cabeçalho e Navegação

Passo 2: Cabeçalho e Navegação

Dentro do body, vamos começar a inserir as seções da nossa página:

```
<header>
  <h1>Meu Portfólio</h1>
  <nav>
    <ul>
      <li><a href="#sobre">Sobre</a></li>
      <li><a href="#servicos">Serviços</a></li>
      <li><a href="#projetos">Projetos</a></li>
      <li><a href="#contato">Contato</a></li>
    </ul>
  </nav>
</header>
```

Explicação:

- **<header>**: Contém o título do portfólio e o menu de navegação.
- **<h1>**: Título principal da página.
- **<nav>**: Elemento de navegação, contendo uma lista de links.
- ****: Lista não ordenada para os itens de navegação.

- ****: Cada item da lista.
- ****: Links que navegam para diferentes seções da página.

Estilização do Cabeçalho e Navegação

Adicione o seguinte CSS ao arquivo `styles.css`:

```
/* Resetando margens e paddings */
* {
    margin: 0;
    padding: 0;
    box-sizing: border-box;
}

/* Cabeçalho */
header {
    background: #333;
    color: #fff;
    padding: 20px 0;
    text-align: center;
}

header h1 {
    margin-bottom: 10px;
}

header nav ul {
    list-style: none;
    padding: 0;
}

header nav ul li {
    display: inline;
    margin-right: 10px;
}
```

```
header nav ul li a {  
    color: #fff;  
    text-decoration: none;  
}
```

Explicação:

- **header**: Define a cor de fundo, cor do texto, padding e alinhamento do texto para o cabeçalho.
- **header h1**: Define a margem inferior do título do cabeçalho.
- **header nav ul**: Remove os estilos de lista padrão.
- **header nav ul li**: Exibe os itens da lista em linha e adiciona margem à direita.
- **header nav ul li a**: Define a cor do texto e remove o sublinhado dos links.

1.9.3 Criando e Estilizando as Seções

Passo 3: Seções de Conteúdo

Abaixo do header, insira a nossa tag main:

```
<main>  
    <section id="sobre">  
        <h2>Sobre Mim</h2>  
        <p>Meu nome é João e sou um desenvolvedor web apaixonado  
        por criar soluções criativas e eficientes.</p>  
    </section>  
    <section id="servicos">  
        <h2>Serviços</h2>  
        <ul>  
            <li>Desenvolvimento de Websites</li>  
            <li>Design Responsivo</li>  
            <li>Otimização de Performance</li>  
        </ul>  
    </section>
```

```

<section id="projetos">
  <h2>Projetos</h2>
  <div class="projeto">
    <h3>Projeto 1</h3>
    <p>Descrição do projeto 1.</p>
  </div>
  <div class="projeto">
    <h3>Projeto 2</h3>
    <p>Descrição do projeto 2.</p>
  </div>
</section>
</main>

```

Explicação:

- **<main>**: Elemento principal que contém as seções de conteúdo.
- **<section>**: Define as diferentes seções da página.
- **<h2>**: Cabeçalhos das seções.
- **<p>**: Parágrafos de texto.
- ****: Lista não ordenada para os serviços.
- **<div class="projeto">**: Divisões que contêm os projetos.

Estilização das Seções

Adicione o seguinte CSS ao arquivo `styles.css`:

```

/* Seções */
section {
  margin-bottom: 20px;
}

section h2 {
  margin-bottom: 10px;
  color: #333;
}

```

```
}

section p, section ul {
    margin-bottom: 10px;
}

section ul {
    list-style-type: disc;
    margin-left: 20px;
}

.projeto {
    margin-bottom: 20px;
    padding: 20px;
    border: 1px solid #ddd;
    border-radius: 5px;
    background-color: #fff;
}
```

Explicação:

- **section**: Define o espaçamento entre as seções.
- **section h2**: Define a margem inferior dos cabeçalhos das seções e a cor do texto.
- **section p, section ul**: Define a margem inferior dos parágrafos e listas nas seções.
- **section ul**: Define o tipo de marcador e a margem esquerda das listas.
- **.projeto**: Define a margem inferior, padding, borda, borda arredondada e cor de fundo para os projetos.

1.9.4 Criando e Estilizando o Formulário de Contato

Passo 4: Formulário de Contato

Abaixo da última section, adicione mais uma:

```
<section id="contato">
  <h2>Contato</h2>
  <form action="/submit" method="post">
    <label for="nome">Nome:</label>
    <input type="text" id="nome" name="nome" required>

    <label for="email">Email:</label>
    <input type="email" id="email" name="email" required>

    <label for="mensagem">Mensagem:</label>
    <textarea id="mensagem" name="mensagem" rows="4" required></textarea>

    <button type="submit">Enviar</button>
  </form>
</section>
```

Explicação:

- **<section id="contato">**: Seção que contém o formulário de contato.
- **<form>**: Elemento de formulário, com atributos `action` e `method`.
- **<label>**: Rótulos para os campos do formulário.
- **<input>**: Campos de entrada para nome e email.
- **<textarea>**: Área de texto para a mensagem.
- **<button>**: Botão para enviar o formulário.

Estilização do Formulário de Contato

Adicione o seguinte CSS ao arquivo `styles.css`:

```
/* Formulário */
form {
```

```

        display: flex;
        flex-direction: column;
    }

label {
    margin-bottom: 5px;
    font-weight: bold;
}

input, textarea, button {
    margin-bottom: 10px;
    padding: 10px;
    border: 1px solid #ccc;
    border-radius: 3px;
}

button {
    background-color: #5cb85c;
    color: white;
    border: none;
    cursor: pointer;
}

button:hover {
    background-color: #4cae4c;
}

```

Explicação:

- **form:** Define a exibição em flex e a direção das colunas para o formulário.
- **label:** Define a margem inferior e a negrito para os rótulos.
- **input, textarea, button:** Define a margem inferior, padding, borda e borda arredondada para os campos do formulário.
- **button:** Define a cor de fundo, cor do texto, borda e cursor para o botão.
- **button:hover:** Define a cor de fundo ao passar o mouse sobre o botão.

1.9.5 Criando e Estilizando o Rodapé

Passo 5: Rodapé

Adicione o `<footer>` abaixo do fechamento da tag `<main>`

```
<footer>
  <p>&copy; 2024 João. Todos os direitos reservados.</p>
</footer>
```

Explicação:

- `<footer>`: Contém o texto do rodapé, com informações de direitos autorais.

Estilização do Rodapé

Adicione o seguinte CSS ao arquivo `styles.css`:

```
/* Rodapé */
footer {
  text-align: center;
  padding: 10px 0;
  background: #333;
  color: #fff;
}
```

Explicação:

- `footer`: Define o alinhamento do texto, padding, cor de fundo e cor do texto para o rodapé.



Exercícios Práticos

1. Adicione uma imagem de perfil na seção "Sobre Mim" e estilize-a com CSS.

2. Implemente uma validação de formulário que verifique se todos os campos estão preenchidos antes de enviar.
 3. Crie um carrossel de imagens na seção "Projetos" utilizando JavaScript e CSS.
 4. Adicione animações de transição para quando os usuários passarem o mouse sobre os elementos do menu de navegação.
 5. Experimente alterar o layout do portfólio para ser responsivo usando media queries no CSS.
-

1.10 Layout Responsivo

O layout responsivo é essencial para garantir que as páginas web sejam visualmente atraentes e funcionais em **diferentes dispositivos**, desde desktops até smartphones. 📱💻 Neste tópico, vamos aprender como criar layouts responsivos usando CSS e técnicas como media queries e flexbox.

1.10.1 Introdução ao Layout Responsivo

Um layout responsivo adapta a estrutura e o conteúdo de uma página web para diferentes tamanhos de tela e resoluções. Isso melhora a experiência do usuário em dispositivos móveis, tablets e desktops. O CSS é a ferramenta principal para implementar um layout responsivo. 🌐

1.10.2 Media Queries

As media queries são uma parte fundamental do CSS para criar layouts responsivos. Elas permitem aplicar estilos CSS com base em características específicas do dispositivo, como a largura da tela.

Exemplo de Media Query:

```
/* Estilo padrão para dispositivos grandes */
body {
    font-size: 18px;
}

/* Estilos para dispositivos com largura máxima de 768px */
@media (max-width: 768px) {
    body {
        font-size: 16px;
    }
}
```

Explicação:

- **Estilo padrão:** Define o tamanho da fonte para dispositivos grandes.
- **Media Query:** Aplica um estilo diferente para dispositivos com largura máxima de 768px (como tablets e smartphones).

1.10.3 Utilizando Flexbox para Layout Responsivo

O Flexbox é uma técnica de layout poderosa que facilita a criação de layouts responsivos. Ele permite distribuir o espaço entre os itens de um contêiner e alinhar os itens de uma maneira previsível.

Exemplo de Flexbox:

```
<div class="container">
    <div class="item">Item 1</div>
    <div class="item">Item 2</div>
    <div class="item">Item 3</div>
</div>
```

```
.container {
    display: flex;
```

```
flex-wrap: wrap;  
}  
  
.item {  
  flex: 1 1 200px;  
  margin: 10px;  
  padding: 20px;  
  background-color: #f4f4f4;  
  text-align: center;  
}
```

Explicação:

- `display: flex`: Define o contêiner como um contêiner flexível.
- `flex-wrap: wrap`: Permite que os itens flexíveis sejam quebrados em várias linhas.
- `flex: 1 1 200px`: Define que cada item deve crescer e encolher igualmente, com uma largura mínima de 200px.
- `margin, padding, background-color, text-align`: Estiliza os itens para melhor visualização.

1.10.4 Layout Responsivo Completo

Vamos criar um layout responsivo completo usando o que aprendemos até agora. Esse layout terá um cabeçalho, uma barra de navegação, uma área de conteúdo principal e um rodapé.

HTML:

```
<!DOCTYPE html>  
<html lang="pt-BR">  
<head>  
  <meta charset="UTF-8">  
  <meta name="viewport" content="width=device-width, initial-  
  scale=1.0">
```

```

<title>Layout Responsivo</title>
<link rel="stylesheet" href="styles.css">
</head>
<body>
  <header>
    <h1>Layout Responsivo</h1>
    <nav>
      <ul>
        <li><a href="#home">Home</a></li>
        <li><a href="#sobre">Sobre</a></li>
        <li><a href="#servicos">Serviços</a></li>
        <li><a href="#contato">Contato</a></li>
      </ul>
    </nav>
  </header>
  <main>
    <section id="conteudo">
      <h2>Conteúdo Principal</h2>
      <p>Bem-vindo ao nosso site com layout responsivo!</p>
    </section>
  </main>
  <footer>
    <p>&copy; 2024 Layout Responsivo. Todos os direitos reservados.</p>
  </footer>
</body>
</html>

```

CSS:

```

/* Reset de Estilos */
* {
  margin: 0;
  padding: 0;
  box-sizing: border-box;
}

```

```
/* Estilos Gerais */
body {
    font-family: Arial, sans-serif;
    line-height: 1.6;
    background-color: #f4f4f4;
    color: #333;
    padding: 20px;
}

/* Cabeçalho */
header {
    background: #333;
    color: #fff;
    padding: 20px 0;
    text-align: center;
}

header h1 {
    margin-bottom: 10px;
}

header nav ul {
    list-style: none;
    padding: 0;
}

header nav ul li {
    display: inline;
    margin-right: 10px;
}

header nav ul li a {
    color: #fff;
    text-decoration: none;
}
```

```
/* Conteúdo Principal */
main {
    margin: 20px 0;
}

main h2 {
    margin-bottom: 10px;
}

main p {
    margin-bottom: 20px;
}

/* Rodapé */
footer {
    text-align: center;
    padding: 10px 0;
    background: #333;
    color: #fff;
}

/* Estilos Responsivos */
@media (max-width: 768px) {
    header nav ul li {
        display: block;
        margin: 5px 0;
    }

    .container {
        flex-direction: column;
    }

    .item {
        flex: 1 1 100%;
```

```
}
```

Explicação:

- **Media Query:** Define estilos específicos para dispositivos com largura máxima de 768px.
- **header nav ul li:** Exibe os itens da navegação em bloco, em vez de inline, para dispositivos móveis.
- **.container:** Define a direção da flexbox para coluna em dispositivos menores.
- **.item:** Ajusta a largura dos itens para 100% em dispositivos menores.



Exercícios Práticos

1. Crie um layout responsivo simples com um cabeçalho, barra de navegação, área de conteúdo principal e rodapé.
2. Utilize media queries para ajustar a navegação de uma exibição em linha para uma exibição em bloco em dispositivos menores.
3. Implemente um layout flexível usando Flexbox para uma galeria de imagens responsiva. A small icon of a camera with a flash.
4. Experimente alterar as cores e fontes para diferentes tamanhos de tela utilizando media queries. A small icon of a painter's palette.
5. Crie um layout de duas colunas que se transforme em uma única coluna em dispositivos móveis. A small icon of a smartphone.

1.11 Elementos Semânticos e Acessibilidade

Os elementos semânticos são uma parte fundamental do HTML5, pois fornecem significado ao conteúdo e melhoram a acessibilidade das páginas web. Neste

tópico, vamos explorar o uso de elementos semânticos e as práticas recomendadas para tornar seu site mais acessível. 

1.11.1 Introdução aos Elementos Semânticos

Os elementos semânticos descrevem claramente seu significado para o navegador e para o desenvolvedor. Exemplos de elementos semânticos incluem `<header>`, `<footer>`, `<article>`, `<section>`, e `<nav>`.

Exemplo de Elementos Semânticos:

```
<!DOCTYPE html>
<html lang="pt-BR">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-
  scale=1.0">
  <title>Document</title>
</head>
<body>
  <header>
    <h1>Cabeçalho Principal</h1>
  </header>
  <nav>
    <ul>
      <li><a href="#home">Home</a></li>
      <li><a href="#about">Sobre</a></li>
      <li><a href="#services">Serviços</a></li>
      <li><a href="#contact">Contato</a></li>
    </ul>
  </nav>
  <section>
    <article>
      <h2>Título do Artigo</h2>
      <p>Conteúdo do artigo...</p>
    </article>
  </section>
```

```
<footer>
    <p>&copy; 2024 Seu Nome. Todos os direitos reservados.</p>
</footer>
</body>
</html>
```

Explicação:

- **<header>**: Usado para o cabeçalho de uma página ou seção.
- **<nav>**: Define um conjunto de links de navegação.
- **<section>**: Representa uma seção genérica de conteúdo.
- **<article>**: Representa um conteúdo independente e autocontido.
- **<footer>**: Usado para o rodapé de uma página ou seção.

1.11.2 Melhorando a Acessibilidade

A acessibilidade web garante que todas as pessoas, independentemente de suas habilidades, possam usar a web. Aqui estão algumas práticas recomendadas para melhorar a acessibilidade do seu site:

- **Usar Elementos Semânticos:** Como vimos, elementos semânticos fornecem significado ao conteúdo.
- **Atributos Alt em Imagens:** Adicionar descrições alternativas para imagens usando o atributo `alt`.

Exemplo de Atributo Alt:

```

```

- **Rótulos de Formulário:** Usar a tag `<label>` para associar rótulos aos elementos de formulário.

Exemplo de Rótulos de Formulário:

```
<form>
  <label for="nome">Nome:</label>
  <input type="text" id="nome" name="nome">

  <label for="email">Email:</label>
  <input type="email" id="email" name="email">

  <button type="submit">Enviar</button>
</form>
```

- **Contraste de Cores:** Certificar-se de que há contraste suficiente entre o texto e o fundo para que o texto seja legível.

Exemplo de Contraste de Cores:

```
body {
  background-color: #ffffff;
  color: #333333;
}
```

- **Uso de ARIA (Accessible Rich Internet Applications):** Adicionar atributos ARIA para melhorar a acessibilidade de elementos interativos.

Exemplo de Atributos ARIA:

```
<button aria-label="Fechar">X</button>
```

1.11.3 Exemplos Práticos de Acessibilidade

Vamos aplicar algumas dessas práticas em um exemplo prático.

Exemplo de Página Acessível:

```
<!DOCTYPE html>
<html lang="pt-BR">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Página Acessível</title>
  <link rel="stylesheet" href="styles.css">
</head>
<body>
  <header>
    <h1>Bem-vindo ao Meu Site Acessível</h1>
    <nav>
      <ul>
        <li><a href="#home">Home</a></li>
        <li><a href="#sobre">Sobre</a></li>
        <li><a href="#servicos">Serviços</a></li>
        <li><a href="#contato">Contato</a></li>
      </ul>
    </nav>
  </header>
  <main>
    <section id="sobre">
      <h2>Sobre Mim</h2>
      <p>Meu nome é João e sou um desenvolvedor web dedicado a criar experiências inclusivas na web.</p>
    </section>
    <section id="servicos">
      <h2>Serviços</h2>
      <ul>
        <li>Desenvolvimento de Websites</li>
        <li>Design Responsivo</li>
        <li>Otimização de Performance</li>
      </ul>
    </section>
  </main>

```

```

<section id="contato">
  <h2>Contato</h2>
  <form action="/submit" method="post">
    <label for="nome">Nome:</label>
    <input type="text" id="nome" name="nome" required>

    <label for="email">Email:</label>
    <input type="email" id="email" name="email" required>

    <label for="mensagem">Mensagem:</label>
    <textarea id="mensagem" name="mensagem" rows="4" required></textarea>

    <button type="submit">Enviar</button>
  </form>
</section>
</main>
<footer>
  <p>&copy; 2024 João. Todos os direitos reservados.</p>
</footer>
</body>
</html>

```

CSS:

```

/* Reset de Estilos */
* {
  margin: 0;
  padding: 0;
  box-sizing: border-box;
}

/* Estilos Gerais */
body {
  font-family: Arial, sans-serif;
  line-height: 1.6;
}

```

```
background-color: #ffffff;
color: #333333;
padding: 20px;
}

/* Cabeçalho */
header {
    background: #333333;
    color: #ffffff;
    padding: 20px 0;
    text-align: center;
}

header h1 {
    margin-bottom: 10px;
}

header nav ul {
    list-style: none;
    padding: 0;
}

header nav ul li {
    display: inline;
    margin-right: 10px;
}

header nav ul li a {
    color: #ffffff;
    text-decoration: none;
}

/* Conteúdo Principal */
main {
    margin: 20px 0;
}
```

```
main h2 {
    margin-bottom: 10px;
}

main p {
    margin-bottom: 20px;
}

main ul {
    list-style-type: disc;
    margin-left: 20px;
}

main ul li {
    margin-bottom: 5px;
}

/* Formulário */
form {
    display: flex;
    flex-direction: column;
}

label {
    margin-bottom: 5px;
    font-weight: bold;
}

input, textarea, button {
    margin-bottom: 10px;
    padding: 10px;
    border: 1px solid #cccccc;
    border-radius: 3px;
}
```

```

button {
    background-color: #5cb85c;
    color: #ffffff;
    border: none;
    cursor: pointer;
}

button:hover {
    background-color: #4cae4c;
}

/* Rodapé */
footer {
    text-align: center;
    padding: 10px 0;
    background: #333333;
    color: #ffffff;
}

```



Exercícios Práticos

1. Adicione atributos `alt` descritivos a todas as imagens do seu site para melhorar a acessibilidade.
2. Use elementos semânticos como `<header>`, `<nav>`, `<section>`, `<article>`, e `<footer>` em uma página HTML.
3. Crie um formulário acessível usando rótulos `<label>` e elementos de entrada `<input>` e `<textarea>`.
4. Aplique contrastes de cores adequados entre o texto e o fundo para garantir legibilidade.
5. Implemente atributos ARIA em elementos interativos para melhorar a acessibilidade.

1.12 Layouts Avançados com Flexbox e Grid

Neste tópico, vamos explorar duas das mais poderosas ferramentas de layout no CSS: **Flexbox** e **Grid Layout**. Com essas ferramentas, você será capaz de criar layouts complexos e responsivos de maneira eficiente. 

1.12.1 Introdução ao Flexbox

Flexbox (Flexible Box Layout) é uma técnica de layout unidimensional que facilita a distribuição do espaço entre itens em um contêiner e o alinhamento desses itens. Flexbox é ideal para layouts que precisam se adaptar a diferentes tamanhos de tela.

Estrutura Básica do Flexbox:

```
<div class="container">
  <div class="item">Item 1</div>
  <div class="item">Item 2</div>
  <div class="item">Item 3</div>
</div>

.container {
  display: flex;
  flex-direction: row; /* Define a direção dos itens */
}

.item {
  flex: 1; /* Os itens crescem igualmente para preencher o espaço disponível */
  padding: 20px;
  background-color: #f4f4f4;
  margin: 10px;
```

```
    text-align: center;  
}
```

Propriedades Comuns do Flexbox:

- **display: flex;**: Define um contêiner flexível.
- **flex-direction:**: Define a direção dos itens (row, row-reverse, column, column-reverse).
- **justify-content:**: Alinha os itens ao longo do eixo principal (flex-start, flex-end, center, space-between, space-around).
- **align-items:**: Alinha os itens ao longo do eixo transversal (flex-start, flex-end, center, stretch, baseline).

Exemplo de Flexbox:

```
.container {  
  display: flex;  
  justify-content: space-between;  
  align-items: center;  
  height: 100vh; /* Altura da tela inteira */  
}  
  
.item {  
  flex: 1;  
  padding: 20px;  
  background-color: #f4f4f4;  
  margin: 10px;  
  text-align: center;  
}
```

Explicação:

- **justify-content: space-between;**: Espaça os itens igualmente ao longo do eixo principal.
- **align-items: center;**: Alinha os itens ao centro do eixo transversal.

1.12.2 Introdução ao Grid Layout

Grid Layout é uma técnica de layout bidimensional que permite a criação de layouts complexos com alinhamento preciso. É ideal para páginas com estruturas de grade, como galerias de imagens e dashboards.

Estrutura Básica do Grid:

```
<div class="grid-container">
  <div class="grid-item">Item 1</div>
  <div class="grid-item">Item 2</div>
  <div class="grid-item">Item 3</div>
  <div class="grid-item">Item 4</div>
  <div class="grid-item">Item 5</div>
  <div class="grid-item">Item 6</div>
</div>
```

```
.grid-container {
  display: grid;
  grid-template-columns: repeat(3, 1fr); /* Define 3 colunas
de tamanho igual */
  grid-gap: 10px; /* Espaçamento entre os itens */
}

.grid-item {
  background-color: #f4f4f4;
  padding: 20px;
  text-align: center;
}
```

Propriedades Comuns do Grid:

- **display: grid;**: Define um contêiner de grade.
- **grid-template-columns**: Define o número e o tamanho das colunas.

- **grid-template-rows**:: Define o número e o tamanho das linhas.
- **grid-gap**:: Define o espaçamento entre os itens.
- **grid-column**:: Define a posição do item em termos de colunas.
- **grid-row**:: Define a posição do item em termos de linhas.

Exemplo de Grid Layout:

```
.grid-container {
  display: grid;
  grid-template-columns: repeat(3, 1fr); /* Três colunas de tamanho igual */
  grid-template-rows: 100px 200px; /* Primeira linha de 100px e segunda de 200px */
  grid-gap: 10px; /* Espaçamento entre os itens */
}

.grid-item:nth-child(1) {
  grid-column: 1 / 3; /* O primeiro item ocupa duas colunas */
}

.grid-item:nth-child(2) {
  grid-row: 1 / 3; /* O segundo item ocupa duas linhas */
}

.grid-item {
  background-color: #f4f4f4;
  padding: 20px;
  text-align: center;
}
```

Explicação:

- **grid-column: 1 / 3;**: O primeiro item ocupa da coluna 1 à coluna 3 (duas colunas).

- **grid-row: 1 / 3;**: O segundo item ocupa da linha 1 à linha 3 (duas linhas).

1.12.3 Comparação entre Flexbox e Grid

Flexbox:

- Unidimensional (trabalha com uma linha ou coluna por vez).
- Ideal para layouts simples e alinhamentos ao longo de um único eixo.
- Mais fácil de aprender e implementar rapidamente.

Grid:

- Bidimensional (trabalha com linhas e colunas simultaneamente).
- Ideal para layouts complexos e controlados com precisão.
- Requer mais tempo para aprender, mas oferece maior flexibilidade.

1.12.4 Exemplo Prático de Layout com Flexbox e Grid

Vamos criar um exemplo prático de um layout responsivo usando Flexbox e Grid.

HTML:

```
<!DOCTYPE html>
<html lang="pt-BR">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-
  scale=1.0">
  <title>Layout Avançado</title>
  <link rel="stylesheet" href="styles.css">
</head>
<body>
  <header>
    <h1>Layout Avançado com Flexbox e Grid</h1>
    <nav>
```

```

<ul>
    <li><a href="#home">Home</a></li>
    <li><a href="#sobre">Sobre</a></li>
    <li><a href="#servicos">Serviços</a></li>
    <li><a href="#contato">Contato</a></li>
</ul>
</nav>
</header>
<main class="container">
    <section class="item">Item 1</section>
    <section class="item">Item 2</section>
    <section class="item">Item 3</section>
    <section class="item">Item 4</section>
    <section class="item">Item 5</section>
    <section class="item">Item 6</section>
</main>
<footer>
    <p>&copy; 2024 Layout Avançado. Todos os direitos reservados.</p>
</footer>
</body>
</html>

```

CSS:

```

/* Reset de Estilos */
* {
    margin: 0;
    padding: 0;
    box-sizing: border-box;
}

/* Estilos Gerais */
body {
    font-family: Arial, sans-serif;
    line-height: 1.6;
}

```

```
background-color: #f4f4f4;
color: #333;
padding: 20px;
}

/* Cabeçalho */
header {
    background: #333;
    color: #fff;
    padding: 20px 0;
    text-align: center;
}

header h1 {
    margin-bottom: 10px;
}

header nav ul {
    list-style: none;
    padding: 0;
}

header nav ul li {
    display: inline;
    margin-right: 10px;
}

header nav ul li a {
    color: #fff;
    text-decoration: none;
}

/* Container Flexível */
.container {
    display: flex;
    flex-wrap: wrap;
```

```

        justify-content: space-between;
    }

.item {
    flex: 1 1 30%;
    background-color: #f4f4f4;
    margin: 10px;
    padding: 20px;
    text-align: center;
}

/* Estilos Responsivos */
@media (max-width: 768px) {
    .container {
        display: grid;
        grid-template-columns: 1fr;
        grid-gap: 10px;
    }

    .item {
        flex: 1 1 100%;
    }
}

/* Rodapé */
footer {
    text-align: center;
    padding: 10px 0;
    background: #333;
    color: #fff;
}

```

Explicação:

- **Flexbox:** Usado para o layout principal em dispositivos grandes.

- **Grid:** Usado para o layout principal em dispositivos menores, através de media queries.
- **Responsividade:** A combinação de Flexbox e Grid permite uma adaptação suave entre diferentes tamanhos de tela.



Exercícios Práticos

1. Crie um layout de galeria de imagens usando Flexbox.
 2. Crie um layout de blog usando Grid Layout, com uma barra lateral e uma área de conteúdo principal.
 3. Aplique media queries para alternar entre Flexbox e Grid em diferentes tamanhos de tela.
 4. Experimente alinhar itens de diferentes maneiras usando `justify-content` e `align-items` no Flexbox.
 5. Crie um layout de dashboard usando Grid Layout, com cabeçalho, barra lateral, área de conteúdo principal e rodapé.
-

1.13 Projeto 2 - Biblioteca

Neste projeto, vamos criar uma aplicação web de biblioteca onde os usuários podem visualizar uma lista de livros e adicionar novos livros. Vamos dividir o projeto em partes, construindo e explicando cada uma delas detalhadamente.

1.13.1 Estrutura Inicial do HTML

Vamos começar criando a estrutura básica do HTML para a nossa aplicação de biblioteca.

Passo 1: Head

```
<!DOCTYPE html>
<html lang="pt-BR">
```

```

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Biblioteca Online</title>
  <link rel="stylesheet" href="styles.css">
</head>
<body>

```

Explicação:

- **DOCTYPE**: Declaração do tipo de documento, informando ao navegador que estamos utilizando HTML5.
- **<html lang="pt-BR">**: Tag raiz que envolve todo o conteúdo da página, com um atributo `lang` para definir o idioma.
- **<meta charset="UTF-8">**: Define a codificação de caracteres como UTF-8, garantindo que todos os caracteres sejam exibidos corretamente.
- **<meta name="viewport" content="width=device-width, initial-scale=1.0">**: Configura a viewport para garantir que a página seja renderizada corretamente em dispositivos móveis.
- **<title>**: Define o título da página, que aparece na aba do navegador.
- **<link rel="stylesheet" href="styles.css">**: Inclui o arquivo CSS para estilização da página.

Passo 2: Cabeçalho e Navegação

```

<header>
  <h1>Biblioteca Online</h1>
  <nav>
    <ul>
      <li><a href="#livros">Livros</a></li>
      <li><a href="#adicionar-livro">Adicionar Livro</a></li>
    </ul>

```

```
</nav>  
</header>
```

Explicação:

- **<header>**: Contém o título da biblioteca e o menu de navegação.
- **<h1>**: Título principal da página.
- **<nav>**: Elemento de navegação, contendo uma lista de links.
- ****: Lista não ordenada para os itens de navegação.
- ****: Cada item da lista.
- ****: Links que navegam para diferentes seções da página.

Estilização do Cabeçalho e Navegação

Adicione o seguinte CSS ao arquivo `styles.css`:

```
/* Resetando margens e paddings */  
* {  
    margin: 0;  
    padding: 0;  
    box-sizing: border-box;  
}  
  
/* Body */  
body {  
    font-family: Arial, sans-serif;  
    background-color: #f4f4f4;  
    margin: 0;  
    padding: 0px 80px;  
}  
  
/* Cabeçalho */  
header {  
    background: #333;  
    color: #fff;
```

```
padding: 20px 0;
text-align: center;
}

header h1 {
margin-bottom: 10px;
}

header nav ul {
list-style: none;
padding: 0;
}

header nav ul li {
display: inline;
margin-right: 10px;
}

header nav ul li a {
color: #fff;
text-decoration: none;
}
```

Explicação:

- **header**: Define a cor de fundo, cor do texto, padding e alinhamento do texto para o cabeçalho.
- **header h1**: Define a margem inferior do título do cabeçalho.
- **header nav ul**: Remove os estilos de lista padrão.
- **header nav ul li**: Exibe os itens da lista em linha e adiciona margem à direita.
- **header nav ul li a**: Define a cor do texto e remove o sublinhado dos links.

Passo 3: Seções de Conteúdo

```
<main>
  <section id="livros">
    <h2>Lista de Livros</h2>
    <ul id="lista-livros">
      <!-- Livros serão adicionados aqui dinamicamente -->
    </ul>
  </section>
</main>
```

Explicação:

- **<main>**: Elemento principal que contém as seções de conteúdo.
- **<section>**: Define as diferentes seções da página.
- **<h2>**: Cabeçalhos das seções.
- ****: Lista não ordenada para os livros.
- **<form>**: Formulário para adicionar novos livros.
- **<label>**: Rótulos para os campos do formulário.
- **<input>**: Campos de entrada para título, autor e ano do livro.
- **<button>**: Botão para enviar o formulário.

Estilização das Seções

Adicione o seguinte CSS ao arquivo `styles.css`:

```
/* Seções */
section {
  margin-bottom: 20px;
}

section h2 {
  margin-bottom: 10px;
  color: #333;
}
```

```

section ul {
    list-style-type: none;
    padding: 0;
}

section ul li {
    background-color: #fff;
    border: 1px solid #ddd;
    margin-bottom: 10px;
    padding: 10px;
    border-radius: 5px;
}

```

Explicação:

- **section**: Define o espaçamento entre as seções.
- **section h2**: Define a margem inferior dos cabeçalhos das seções e a cor do texto.
- **section ul**: Remove os estilos de lista padrão e padding.
- **section ul li**: Define a cor de fundo, borda, margem inferior, padding e borda arredondada para os itens da lista.

Passo 4: Formulário de Adicionar Livro

Abaixo do fechamento da section anterior, adicione o formulário:

```

<section id="adicionar-livro">
    <h2>Adicionar Novo Livro</h2>
    <form id="form-livro">
        <label for="titulo">Título:</label>
        <input type="text" id="titulo" name="titulo" required>

        <label for="autor">Autor:</label>
        <input type="text" id="autor" name="autor" required>

```

```

<label for="ano">Ano:</label>
<input type="number" id="ano" name="ano" required>

<button type="submit">Adicionar Livro</button>
</form>
</section>

```

Explicação:

- **<form id="form-livro">**: Formulário para adicionar novos livros.
- **<label>**: Rótulos para os campos do formulário.
- **<input>**: Campos de entrada para título, autor e ano do livro.
- **<button>**: Botão para enviar o formulário.

Estilização do Formulário

Adicione o seguinte CSS ao arquivo `styles.css`:

```

/* Formulário */
form {
    display: flex;
    flex-direction: column;
}

label {
    margin-bottom: 5px;
    font-weight: bold;
}

input, button {
    margin-bottom: 10px;
    padding: 10px;
    border: 1px solid #ccc;
    border-radius: 3px;
}

```

```
button {  
    background-color: #5cb85c;  
    color: white;  
    border: none;  
    cursor: pointer;  
}  
  
button:hover {  
    background-color: #4cae4c;  
}
```

Explicação:

- **form**: Define a exibição em flex e a direção das colunas para o formulário.
- **label**: Define a margem inferior e a negrito para os rótulos.
- **input, button**: Define a margem inferior, padding, borda e borda arredondada para os campos do formulário.
- **button**: Define a cor de fundo, cor do texto, borda e cursor para o botão.
- **button:hover**: Define a cor de fundo ao passar o mouse sobre o botão.

Passo 5: Rodapé

```
<footer>  
    <p>&copy; 2024 Biblioteca Online. Todos os direitos  
    reservados.</p>  
    </footer>  
</body>  
</html>
```

Explicação:

- **<footer>**: Contém o texto do rodapé, com informações de direitos autorais.

Estilização do Rodapé

Adicione o seguinte CSS ao arquivo `styles.css`:

```
/* Rodapé */
footer {
    text-align: center;
    padding: 10px 0;
    background: #333;
    color: #fff;
}
```

Explicação:

- **footer:** Define o alinhamento do texto, padding, cor de fundo e cor do texto para o rodapé.

1.13.3 Adicionando Funcionalidade com JavaScript

Vamos adicionar interatividade ao nosso projeto utilizando JavaScript. Vamos criar uma funcionalidade para adicionar livros à lista.

Passo 6: JavaScript para Adicionar Livros

Crie um arquivo chamado `scripts.js` e adicione o seguinte código:

```
document.addEventListener('DOMContentLoaded', function() {
    const form = document.getElementById('form-livro');
    const listaLivros = document.getElementById('lista-livros');

    form.addEventListener('submit', function(event) {
        event.preventDefault(); // Previne o envio padrão do formulário

        const titulo = document.getElementById('titulo').value;
        const autor = document.getElementById('autor').value;
        const ano = document.getElementById('ano').value;
```

```

        const livroItem = document.createElement('li');
        livroItem.textContent = `${titulo} por ${autor} (${ano})`;
        listaLivros.appendChild(livroItem);

        form.reset(); // Reseta o formulário
    });
}

```

Explicação:

- **DOMContentLoaded**: Garante que o código JavaScript seja executado apenas após o carregamento completo do DOM.
- **form.addEventListener('submit', function(event) { ... })**: Adiciona um listener para o evento de envio do formulário e cria um novo item de livro na lista ao enviar o formulário.
- **event.preventDefault()**: Previne o envio padrão do formulário.
- **form.reset()**: Reseta o formulário após adicionar o livro à lista.

Não se esqueça de adicionar o link para o arquivo JavaScript no seu HTML:

```
<script src="scripts.js"></script>
```



Exercícios Práticos

1. Adicione uma validação ao formulário para garantir que todos os campos sejam preenchidos corretamente antes de adicionar um livro.
2. Estilize a lista de livros para que cada item tenha um botão de remoção.
3. Adicione uma funcionalidade para remover um livro da lista ao clicar no botão de remoção.
4. Implemente um sistema de classificação para os livros, permitindo que os usuários dêem uma nota aos livros adicionados.

5. Crie uma seção de pesquisa que permita aos usuários filtrar a lista de livros com base no título ou autor. 

Conclusão do Capítulo 1

Neste capítulo, exploramos os conceitos fundamentais de layout avançado utilizando **Flexbox** e **Grid Layout** no CSS. Aprendemos como utilizar as propriedades básicas e comuns dessas técnicas para criar layouts responsivos e complexos de maneira eficiente. Além disso, vimos exemplos práticos de como implementar esses layouts e a importância de adaptá-los para diferentes tamanhos de tela.

Também trabalhamos em um projeto prático de uma aplicação web de biblioteca, onde construímos a estrutura inicial do HTML, estilizamos com CSS e adicionamos interatividade com JavaScript. Este projeto nos permitiu aplicar os conhecimentos adquiridos de forma prática e ver como Flexbox e Grid podem ser usados juntos para criar layouts adaptáveis.

No próximo capítulo, iremos mergulhar nos fundamentos do **JavaScript básico**. Vamos aprender sobre variáveis, funções, loops, objetos, condicionais, etc... Prepare-se para levar suas habilidades de desenvolvimento web para o próximo nível! 

módulo 2

javascript

Desvende o poder do JavaScript e aprenda tudo sobre variáveis, funções e estruturas de controle. 

08 tópicos neste módulo

Módulo 2: JavaScript

Neste capítulo, vamos explorar os fundamentos da linguagem de programação JavaScript. JavaScript é uma das linguagens mais populares e amplamente utilizadas no desenvolvimento web, permitindo a criação de páginas interativas e dinâmicas. Vamos abordar desde a execução de scripts JavaScript usando Node.js até conceitos essenciais como variáveis, arrays e loops. Este capítulo fornecerá uma base sólida para quem deseja se aprofundar no desenvolvimento com JavaScript e aplicar esses conhecimentos em projetos reais. 🔥

2.1 Como rodar o JavaScript com Node.js

JavaScript é uma linguagem de programação essencial para o desenvolvimento web, permitindo a criação de páginas interativas e dinâmicas.

Originalmente concebida para adicionar interatividade às páginas da web, o JavaScript evoluiu para uma linguagem completa, capaz de desenvolver complexas aplicações web, mobile, e até mesmo servidores com o Node.js.

Antes de começarmos a escrever código JavaScript, precisamos entender como executá-lo. Vamos explorar como rodar JavaScript utilizando Node.js. 🚀✨

2.1.1 Introdução ao JavaScript

JavaScript é uma linguagem de script interpretada, amplamente utilizada para criar páginas web interativas. Ele é executado no navegador do cliente, mas também pode ser usado no lado do servidor com tecnologias como Node.js.

2.1.2 Utilizando Node.js para Executar JavaScript

Node.js é uma plataforma que permite rodar JavaScript no lado do servidor. Para utilizar Node.js, primeiro precisamos instalá-lo.

Passo 1: Instalando Node.js

1. Acesse o site oficial do Node.js: nodejs.org

2. Baixe e instale a versão recomendada para o seu sistema operacional.

Passo 2: Verificando a Instalação

Para verificar se o Node.js foi instalado corretamente, abra o terminal (Prompt de Comando no Windows ou Terminal no Mac/Linux) e execute o comando:

```
node -v
```

Você deve ver a versão do Node.js instalada, indicando que a instalação foi bem-sucedida.

2.1.3 Executando um Script JavaScript com Node.js

Agora que o Node.js está instalado, vamos criar um arquivo JavaScript e executá-lo.

Passo 3: Criando e Executando um Script JavaScript

- Crie um arquivo chamado `app.js` com o seguinte conteúdo:

```
console.log("Olá, mundo! Este script está sendo executado pelo  
o Node.js.");
```

- Abra o terminal do VsCode.
- Execute o seguinte comando:

```
node app.js
```

Você deve ver a mensagem "Olá, mundo! Este script está sendo executado pelo Node.js." exibida no terminal.

Explicação:

- **console.log()**: Exibe a mensagem "Olá, mundo! Este script está sendo executado pelo Node.js." no terminal.
- **node app.js**: Executa o arquivo JavaScript `app.js` utilizando o Node.js.

JavaScript não é apenas uma ferramenta para adicionar efeitos ou interatividade simples em sites; é uma **linguagem robusta, capaz de construir sistemas complexos e performáticos**. Seu papel no desenvolvimento web moderno é indiscutível, e seu domínio é essencial para qualquer desenvolvedor que deseja estar na vanguarda da tecnologia.

2.2 Variáveis

As variáveis são um dos conceitos fundamentais em qualquer linguagem de programação. Elas são usadas para armazenar dados que podem ser manipulados e utilizados ao longo do seu código. Vamos explorar como declarar e usar variáveis em JavaScript. 

2.2.1 Introdução às Variáveis

Em JavaScript, você pode declarar variáveis usando três palavras-chave: `var`, `let` e `const`. Cada uma tem suas características e é usada em diferentes situações.

2.2.2 Declarando Variáveis com `var`

A palavra-chave `var` é a maneira tradicional de declarar variáveis em JavaScript. No entanto, ela tem um escopo de função, o que pode levar a comportamentos inesperados.

Exemplo de Declaração com `var`:

```
var nome = "João";
console.log(nome); // João
```

Explicação:

- `var nome = "João";` : Declara uma variável chamada `nome` e atribui a ela o valor `"João"`.
- `console.log(nome);` : Exibe o valor da variável `nome` no console.

2.2.3 Declarando Variáveis com `let`

A palavra-chave `let` foi introduzida no ECMAScript 6 (ES6) e é recomendada para declarar variáveis que podem ter seus valores alterados. Ela tem um escopo de bloco, o que significa que a variável só existe dentro do bloco onde foi declarada.

Exemplo de Declaração com `let`:

```
let idade = 25;
console.log(idade); // 25
idade = 26;
console.log(idade); // 26
```

Explicação:

- `let idade = 25;` : Declara uma variável chamada `idade` e atribui a ela o valor `25`.
- `idade = 26;` : Altera o valor da variável `idade` para `26`.
- `console.log(idade);` : Exibe o valor atualizado da variável `idade` no console.

2.2.4 Declarando Variáveis com `const`

A palavra-chave `const` também foi introduzida no ES6 e é usada para declarar variáveis cujos valores não podem ser alterados após a atribuição inicial. Assim como `let`, `const` tem um escopo de bloco.

Exemplo de Declaração com `const`:

```
const cidade = "São Paulo";
console.log(cidade); // São Paulo
```

Explicação:

- `const cidade = "São Paulo";`: Declara uma constante chamada `cidade` e atribui a ela o valor `"São Paulo"`.
- `console.log(cidade);`: Exibe o valor da constante `cidade` no console.

2.2.5 Boas Práticas para Usar Variáveis

- **Use `let` e `const` ao invés de `var`:** `let` e `const` têm escopo de bloco, o que reduz o risco de erros.
- **Escolha nomes descritivos:** Nomeie suas variáveis de forma que descrevam claramente o que elas representam.
- **Prefira `const` para valores que não mudam:** Use `const` para declarar variáveis cujos valores não devem ser alterados.

2.2.6 Exemplos Práticos

Vamos praticar a declaração e o uso de variáveis em JavaScript. Crie um arquivo chamado `variaveis.js` e adicione o seguinte código:

Passo 1: Declarando Variáveis com `var`

```
var nome = "Ana";
console.log(nome); // Ana
nome = "Carlos";
console.log(nome); // Carlos
```

Passo 2: Declarando Variáveis com `let`

```
let idade = 30;
console.log(idade); // 30
idade = 31;
console.log(idade); // 31
```

Passo 3: Declarando Constantes com `const`

```
const pais = "Brasil";
console.log(pais); // Brasil
// pais = "Argentina"; // Isso causará um erro porque não podemos reatribuir uma constante
```

Passo 4: Executando o Código com Node.js

Para executar o código, abra o terminal e navegue até o diretório onde está o arquivo `variaveis.js`. Em seguida, execute o comando:

```
node variaveis.js
```

Você deve ver a saída do console conforme descrito nos exemplos acima.



Exercícios Práticos

1. Crie um arquivo chamado `meu_nome.js` e declare uma variável chamada `nome` que armazene seu nome. Exiba o valor no console. 
2. Crie uma variável chamada `idade` com a sua idade e exiba o valor no console. Altere o valor da variável e exiba o novo valor. 
3. Declare uma constante chamada `cidade` com o nome da sua cidade e exiba o valor no console. 
4. Experimente declarar uma constante e tentar alterar seu valor. Observe o erro que ocorre. 
5. Pratique criar variáveis com `let` e `const` em diferentes blocos de código (como dentro de funções ou loops) e observe como o escopo afeta seu uso. 

2.3 Arrays

Os arrays são uma estrutura de dados fundamental em JavaScript. Eles permitem armazenar múltiplos valores em uma única variável, facilitando a manipulação e o acesso a esses dados. Vamos explorar como criar e usar arrays em JavaScript. 



2.3.1 Introdução aos Arrays

Um array é uma lista ordenada de valores, onde cada valor pode ser acessado por um índice numérico. Os índices começam em zero, o que significa que o primeiro elemento de um array está na posição 0.

2.3.2 Criando Arrays

Existem várias maneiras de criar arrays em JavaScript. A forma mais comum é utilizando colchetes `[]`.

Exemplo de Criação de Arrays:

```
let frutas = ["Maçã", "Banana", "Laranja"];
console.log(frutas); // ["Maçã", "Banana", "Laranja"]
```

Explicação:

- `let frutas = ["Maçã", "Banana", "Laranja"];` : Declara um array chamado `frutas` que contém três elementos: "Maçã", "Banana" e "Laranja".
- `console.log(frutas);` : Exibe o conteúdo do array no console.

2.3.3 Acessando Elementos de um Array

Você pode acessar elementos de um array utilizando seus índices. Lembre-se de que os índices começam em zero.

Exemplo de Acesso a Elementos de um Array:

```
let primeiraFruta = frutas[0];
let segundaFruta = frutas[1];
```

```
let terceiraFruta = frutas[2];
console.log(primeiraFruta); // Maçã
console.log(segundaFruta); // Banana
console.log(terceiraFruta); // Laranja
```

Explicação:

- `frutas[0]` : Acessa o primeiro elemento do array.
- `frutas[1]` : Acessa o segundo elemento do array.
- `frutas[2]` : Acessa o terceiro elemento do array.

2.3.4 Modificando Elementos de um Array

Você pode modificar os elementos de um array atribuindo novos valores aos seus índices.

Exemplo de Modificação de Elementos de um Array:

```
frutas[1] = "Morango";
console.log(frutas); // ["Maçã", "Morango", "Laranja"]
```

Explicação:

- `frutas[1] = "Morango";` : Modifica o segundo elemento do array para "Morango".
- `console.log(frutas);` : Exibe o conteúdo atualizado do array no console.

2.3.5 Métodos de Array Comuns

JavaScript fornece vários métodos para manipular arrays. Aqui estão alguns dos mais comuns:

- `push()` : Adiciona um ou mais elementos ao final do array.
- `pop()` : Remove o último elemento do array.
- `shift()` : Remove o primeiro elemento do array.

- `unshift()` : Adiciona um ou mais elementos ao início do array.
- `length` : Retorna o número de elementos no array.

Exemplos de Métodos de Array:

```
let frutas = ["Maçã", "Banana", "Laranja"];

// Adicionando um elemento ao final do array
frutas.push("Uva");
console.log(frutas); // ["Maçã", "Banana", "Laranja", "Uva"]

// Removendo o último elemento do array
frutas.pop();
console.log(frutas); // ["Maçã", "Banana", "Laranja"]

// Removendo o primeiro elemento do array
frutas.shift();
console.log(frutas); // ["Banana", "Laranja"]

// Adicionando um elemento ao início do array
frutas.unshift("Abacaxi");
console.log(frutas); // ["Abacaxi", "Banana", "Laranja"]

// Obtendo o comprimento do array
let tamanho = frutas.length;
console.log(tamanho); // 3
```

Explicação:

- `push("Uva")` : Adiciona "Uva" ao final do array.
- `pop()` : Remove o último elemento do array.
- `shift()` : Remove o primeiro elemento do array.
- `unshift("Abacaxi")` : Adiciona "Abacaxi" ao início do array.
- `length` : Retorna o número de elementos no array.

2.3.6 Iterando sobre Arrays

Você pode iterar sobre os elementos de um array usando loops, como `for` ou `forEach`.

Exemplo de Iteração com `for`:

```
for (let i = 0; i < frutas.length; i++) {
    console.log(frutas[i]);
}
```

Explicação:

- `for (let i = 0; i < frutas.length; i++)`: Itera sobre cada índice do array `frutas`.
- `console.log(frutas[i])`: Exibe cada elemento do array no console.

Exemplo de Iteração com `forEach`:

```
frutas.forEach(function(fruta) {
    console.log(fruta);
});
```

Explicação:

- `frutas.forEach(function(fruta) { ... })`: Itera sobre cada elemento do array `frutas`.
- `console.log(fruta)`: Exibe cada elemento do array no console.

2.3.7 Exemplos Práticos

Vamos praticar a criação e manipulação de arrays em JavaScript. Crie um arquivo chamado `arrays.js` e adicione o seguinte código:

Passo 1: Criando um Array

```
let animais = ["Cachorro", "Gato", "Pássaro"];
console.log(animais); // ["Cachorro", "Gato", "Pássaro"]
```

Passo 2: Acessando Elementos de um Array

```
let primeiroAnimal = animais[0];
let segundoAnimal = animais[1];
let terceiroAnimal = animais[2];
console.log(primeiroAnimal); // Cachorro
console.log(segundoAnimal); // Gato
console.log(terceiroAnimal); // Pássaro
```

Passo 3: Modificando Elementos de um Array

```
animais[1] = "Peixe";
console.log(animais); // ["Cachorro", "Peixe", "Pássaro"]
```

Passo 4: Usando Métodos de Array

```
animais.push("Coelho");
console.log(animais); // ["Cachorro", "Peixe", "Pássaro", "Coelho"]

animais.pop();
console.log(animais); // ["Cachorro", "Peixe", "Pássaro"]

animais.shift();
console.log(animais); // ["Peixe", "Pássaro"]

animais.unshift("Leão");
console.log(animais); // ["Leão", "Peixe", "Pássaro"]

let tamanho = animais.length;
console.log(tamanho); // 3
```

Passo 5: Iterando sobre um Array

```
animais.forEach(function(animal) {  
  console.log(animal);  
});
```

Passo 6: Executando o Código com Node.js

Para executar o código, abra o terminal e navegue até o diretório onde está o arquivo `arrays.js`. Em seguida, execute o comando:

```
node arrays.js
```

Você deve ver a saída do console conforme descrito nos exemplos acima.



Exercícios Práticos

1. Crie um arquivo chamado `minhas_frutas.js` e declare um array chamado `frutas` que armazene o nome de cinco frutas. Exiba o array no console.
2. Acesse e exiba o primeiro e o último elemento do array `frutas` no console.
3. Modifique o segundo elemento do array `frutas` e exiba o array atualizado no console.
4. Adicione uma nova fruta ao final do array `frutas` e exiba o array no console.
5. Remova o primeiro elemento do array `frutas` e exiba o array no console.
6. Itere sobre o array `frutas` e exiba cada fruta no console usando o método `forEach`.

2.4 Loops

Os loops são estruturas de controle que permitem executar um bloco de código várias vezes. Eles são extremamente úteis quando precisamos repetir uma tarefa com diferentes valores. Vamos explorar os principais tipos de loops em JavaScript: `for`, `while` e `do...while`. 

2.4.1 Introdução aos Loops

Loops são usados para iterar sobre coleções de dados ou para repetir uma tarefa até que uma condição específica seja atendida. Existem vários tipos de loops em JavaScript, cada um com suas características e usos específicos.

2.4.2 Loop `for`

O loop `for` é um dos mais utilizados em JavaScript. Ele é ideal para iterar sobre arrays ou executar um bloco de código um número específico de vezes.

Estrutura do Loop `for`:

```
for (inicialização; condição; incremento) {  
    // Código a ser executado em cada iteração  
}
```

Exemplo de Loop `for`:

```
for (let i = 0; i < 5; i++) {  
    console.log("Iteração número: " + i);  
}
```

Explicação:

- `let i = 0;`: Inicialização da variável de controle do loop.
- `i < 5;`: Condição que deve ser verdadeira para que o loop continue executando.
- `i++;`: Incremento da variável de controle após cada iteração.
- `console.log("Iteração número: " + i);`: Código que será executado em cada iteração.

2.4.3 Loop `while`

O loop `while` executa um bloco de código enquanto uma condição especificada é verdadeira. É útil quando o número de iterações não é conhecido previamente.

Estrutura do Loop `while`:

```
while (condição) {  
    // Código a ser executado enquanto a condição for verdadeira  
    a  
}
```

Exemplo de Loop `while`:

```
let contador = 0;  
while (contador < 5) {  
    console.log("Contagem: " + contador);  
    contador++;  
}
```

Explicação:

- `let contador = 0;`: Inicialização da variável de controle.
- `contador < 5;`: Condição que deve ser verdadeira para que o loop continue executando.
- `console.log("Contagem: " + contador);`: Código que será executado em cada iteração.
- `contador++;`: Incremento da variável de controle após cada iteração.

2.4.4 Loop `do...while`

O loop `do...while` é semelhante ao `while`, mas garante que o bloco de código seja executado pelo menos uma vez antes de verificar a condição.

Estrutura do Loop `do...while`:

```
do {  
    // Código a ser executado  
} while (condição);
```

Exemplo de Loop `do...while`:

```
let numero = 0;  
do {  
    console.log("Número: " + numero);  
    numero++;  
} while (numero < 5);
```

Explicação:

- `let numero = 0;`: Inicialização da variável de controle.
- `console.log("Número: " + numero);`: Código que será executado em cada iteração.
- `numero++;`: Incremento da variável de controle após cada iteração.
- `while (numero < 5);`: Condição que deve ser verdadeira para que o loop continue executando.

2.4.5 Looping através de Arrays

Loops são frequentemente usados para iterar sobre arrays. Vamos ver como podemos usar os loops `for` e `forEach` para isso.

Exemplo de Iteração com `for`:

```
let frutas = ["Maçã", "Banana", "Laranja"];  
  
for (let i = 0; i < frutas.length; i++) {  
    console.log(frutas[i]);  
}
```

Explicação:

- `for (let i = 0; i < frutas.length; i++)`: Itera sobre cada índice do array `frutas`.
- `console.log(frutas[i])`: Exibe cada elemento do array no console.

Exemplo de Iteração com `forEach`:

```
frutas.forEach(function(fruta) {
  console.log(fruta);
});
```

Explicação:

- `frutas.forEach(function(fruta) { ... })`: Itera sobre cada elemento do array `frutas`.
- `console.log(fruta)`: Exibe cada elemento do array no console.

2.4.6 Exemplos Práticos

Vamos praticar o uso de loops em JavaScript. Crie um arquivo chamado `loops.js` e adicione o seguinte código:

Passo 1: Usando um Loop `for`

```
for (let i = 1; i <= 10; i++) {
  console.log("Número: " + i);
}
```

Passo 2: Usando um Loop `while`

```
let contador = 1;
while (contador <= 10) {
  console.log("Contagem: " + contador);
  contador++;
}
```

Passo 3: Usando um Loop `do...while`

```
let numero = 1;
do {
    console.log("Número: " + numero);
    numero++;
} while (numero <= 10);
```

Passo 4: Iterando sobre um Array com `for`

```
let animais = ["Cachorro", "Gato", "Pássaro"];
for (let i = 0; i < animais.length; i++) {
    console.log(animais[i]);
}
```

Passo 5: Iterando sobre um Array com `forEach`

```
animais.forEach(function(animal) {
    console.log(animal);
});
```

Passo 6: Executando o Código com Node.js

Para executar o código, abra o terminal e navegue até o diretório onde está o arquivo `loops.js`. Em seguida, execute o comando:

```
node loops.js
```

Você deve ver a saída do console conforme descrito nos exemplos acima.



Exercícios Práticos

1. Crie um arquivo chamado `minha_tabua.js` e use um loop `for` para exibir a tabuada do número 7 (de 1 a 10).
2. Use um loop `while` para exibir os números pares de 2 a 20 no console.
3. Crie um loop `do...while` que exiba os números de 1 a 5 no console.

4. Declare um array de cinco cores e use um loop `for` para exibir cada cor no console. 

5. Use o método `forEach` para iterar sobre um array de nomes e exibir uma saudação personalizada para cada nome. 

2.5 Condicionais

As condicionais são estruturas de controle que permitem executar diferentes blocos de código com base em certas condições. Em JavaScript, usamos `if`, `else if`, `else` e `switch` para criar essas estruturas. Vamos explorar como usar condicionais para tomar decisões no seu código. 

2.5.1 Introdução às Condicionais

As condicionais permitem que seu código tome decisões e execute diferentes ações com base em condições específicas. Isso é essencial para criar lógica dinâmica em seus programas.

2.5.2 Estrutura Básica do `if`

A declaração `if` é usada para executar um bloco de código se uma condição for verdadeira.

Estrutura do `if`:

```
if (condição) {  
    // Código a ser executado se a condição for verdadeira  
}
```

Exemplo de `if`:

```
let idade = 18;
```

```
if (idade >= 18) {  
    console.log("Você é maior de idade.");  
}
```

Explicação:

- `if (idade >= 18)` : Verifica se a condição (idade maior ou igual a 18) é verdadeira.
- `console.log("Você é maior de idade.");` : Executa este código se a condição for verdadeira.

2.5.3 Estrutura do `if...else`

A declaração `if...else` permite executar um bloco de código se a condição for verdadeira e outro bloco de código se a condição for falsa.

Estrutura do `if...else`:

```
if (condição) {  
    // Código a ser executado se a condição for verdadeira  
} else {  
    // Código a ser executado se a condição for falsa  
}
```

Exemplo de `if...else`:

```
let idade = 16;  
  
if (idade >= 18) {  
    console.log("Você é maior de idade.");  
} else {  
    console.log("Você é menor de idade.");  
}
```

Explicação:

- `if (idade >= 18)` : Verifica se a condição (idade maior ou igual a 18) é verdadeira.
- `console.log("Você é maior de idade.");` : Executa este código se a condição for verdadeira.
- `else` : Bloco de código que é executado se a condição for falsa.
- `console.log("Você é menor de idade.");` : Executa este código se a condição for falsa.

2.5.4 Estrutura do `if...else if...else`

A declaração `if...else if...else` permite verificar múltiplas condições e executar diferentes blocos de código com base nessas condições.

Estrutura do `if...else if...else`:

```
if (condição1) {
    // Código a ser executado se condição1 for verdadeira
} else if (condição2) {
    // Código a ser executado se condição2 for verdadeira
} else {
    // Código a ser executado se nenhuma das condições anteriores for verdadeira
}
```

Exemplo de `if...else if...else`:

```
let nota = 85;

if (nota >= 90) {
    console.log("Nota A");
} else if (nota >= 80) {
    console.log("Nota B");
} else if (nota >= 70) {
    console.log("Nota C");
```

```
    } else if (nota >= 60) {
        console.log("Nota D");
    } else {
        console.log("Nota F");
    }
```

Explicação:

- `if (nota >= 90)`: Verifica se a nota é maior ou igual a 90.
- `else if (nota >= 80)`: Verifica se a nota é maior ou igual a 80.
- `else if (nota >= 70)`: Verifica se a nota é maior ou igual a 70.
- `else if (nota >= 60)`: Verifica se a nota é maior ou igual a 60.
- `else`: Bloco de código que é executado se nenhuma das condições anteriores for verdadeira.

2.5.5 Estrutura do `switch`

A declaração `switch` é usada para executar um bloco de código entre vários blocos, com base no valor de uma expressão.

Estrutura do `switch`:

```
switch (expressão) {
    case valor1:
        // Código a ser executado se expressão === valor1
        break;
    case valor2:
        // Código a ser executado se expressão === valor2
        break;
    // mais casos...
    default:
        // Código a ser executado se nenhum dos casos anteriores
        // for verdadeiro
}
```

Exemplo de `switch`:

```
let dia = "segunda";

switch (dia) {
  case "segunda":
    console.log("Hoje é segunda-feira.");
    break;
  case "terça":
    console.log("Hoje é terça-feira.");
    break;
  case "quarta":
    console.log("Hoje é quarta-feira.");
    break;
  case "quinta":
    console.log("Hoje é quinta-feira.");
    break;
  case "sexta":
    console.log("Hoje é sexta-feira.");
    break;
  default:
    console.log("Hoje é fim de semana.");
}
```

Explicação:

- `switch (dia)`: Avalia a expressão `dia`.
- `case "segunda"`: Executa o bloco de código se `dia` for igual a "segunda".
- `break`: Interrompe a execução dos casos para evitar que outros casos sejam executados.
- `default`: Executa o bloco de código se nenhum dos casos anteriores for verdadeiro.

2.5.6 Exemplos Práticos

Vamos praticar o uso de condicionais em JavaScript. Crie um arquivo chamado `condicionais.js` e adicione o seguinte código:

Passo 1: Usando `if`

```
let idade = 20;

if (idade >= 18) {
    console.log("Você é maior de idade.");
}
```

Passo 2: Usando `if...else`

```
let idade = 15;

if (idade >= 18) {
    console.log("Você é maior de idade.");
} else {
    console.log("Você é menor de idade.");
}
```

Passo 3: Usando `if...else if...else`

```
let nota = 75;

if (nota >= 90) {
    console.log("Nota A");
} else if (nota >= 80) {
    console.log("Nota B");
} else if (nota >= 70) {
    console.log("Nota C");
} else if (nota >= 60) {
    console.log("Nota D");
} else {
    console.log("Nota F");
}
```

Passo 4: Usando `switch`

```
let dia = "quarta";

switch (dia) {
  case "segunda":
    console.log("Hoje é segunda-feira.");
    break;
  case "terça":
    console.log("Hoje é terça-feira.");
    break;
  case "quarta":
    console.log("Hoje é quarta-feira.");
    break;
  case "quinta":
    console.log("Hoje é quinta-feira.");
    break;
  case "sexta":
    console.log("Hoje é sexta-feira.");
    break;
  default:
    console.log("Hoje é fim de semana.");
}
```

Passo 5: Executando o Código com Node.js

Para executar o código, abra o terminal e navegue até o diretório onde está o arquivo `condicionais.js`. Em seguida, execute o comando:

```
node condicionais.js
```

Você deve ver a saída do console conforme descrito nos exemplos acima.



Exercícios Práticos

1. Crie um arquivo chamado `idade_verificacao.js` e use um `if...else` para verificar se a idade de uma pessoa permite que ela vote (`idade >= 16`). 
 2. Use um `if...else if...else` para categorizar a velocidade de um carro (`velocidade < 40`: "Muito devagar", `40 <= velocidade < 80`: "Velocidade normal", `velocidade >= 80`: "Muito rápido"). 
 3. Crie um script que use um `switch` para exibir o nome do mês com base em um número (1 para janeiro, 2 para fevereiro, etc.). 
 4. Crie um script que verifique se um número é par ou ímpar usando um `if...else`. 
 5. Use um `switch` para exibir uma mensagem de saudação em diferentes idiomas com base em um código de idioma (por exemplo, "en" para inglês, "es" para espanhol, "fr" para francês). 
-

2.6 Funções

As funções são blocos de código reutilizáveis que realizam uma tarefa específica. Elas ajudam a tornar o código mais modular, legível e fácil de manter. Vamos explorar como criar e usar funções em JavaScript. 

2.6.1 Introdução às Funções

Uma função é um bloco de código que pode ser definido uma vez e executado sempre que necessário. As funções podem aceitar parâmetros, que são valores passados para a função, e podem retornar um valor.

2.6.2 Declarando Funções

Você pode declarar uma função usando a palavra-chave `function`, seguida pelo nome da função, parênteses `()` e um bloco de código `{}`.

Exemplo de Declaração de Função:

```
function saudacao() {  
    console.log("Olá, mundo!");  
}
```

Explicação:

- `function saudacao()` : Declara uma função chamada `saudacao`.
- `console.log("Olá, mundo!");` : Bloco de código que será executado quando a função for chamada.

2.6.3 Chamando Funções

Para executar uma função, você precisa chamar seu nome seguido por parênteses `()`.

Exemplo de Chamada de Função:

```
saudacao(); // Olá, mundo!
```

Explicação:

- `saudacao();` : Chama a função `saudacao`, executando o bloco de código dentro dela.

2.6.4 Funções com Parâmetros

As funções podem aceitar parâmetros, que são valores passados para a função quando ela é chamada. Esses parâmetros podem ser usados dentro da função.

Exemplo de Função com Parâmetros:

```
function saudacao(nome) {  
    console.log("Olá, " + nome + "!");  
}
```

```
saudacao("Maria"); // Olá, Maria!  
saudacao("João"); // Olá, João!
```

Explicação:

- `function saudacao(nome)` : Declara uma função que aceita um parâmetro chamado `nome`.
- `console.log("Olá, " + nome + "!");` : Usa o parâmetro `nome` dentro da função.

2.6.5 Funções com Retorno

As funções podem retornar um valor usando a palavra-chave `return`. Esse valor pode ser usado posteriormente no código.

Exemplo de Função com Retorno:

```
function soma(a, b) {  
    return a + b;  
}  
  
let resultado = soma(3, 4);  
console.log(resultado); // 7
```

Explicação:

- `function soma(a, b)` : Declara uma função que aceita dois parâmetros, `a` e `b`.
- `return a + b;` : Retorna a soma de `a` e `b`.
- `let resultado = soma(3, 4);` : Chama a função `soma` com os valores `3` e `4`, armazenando o resultado na variável `resultado`.
- `console.log(resultado);` : Exibe o valor de `resultado` no console.

2.6.6 Funções Anônimas e Arrow Functions

As funções anônimas são funções sem nome, frequentemente usadas como argumentos para outras funções. As arrow functions são uma sintaxe mais curta

para escrever funções anônimas introduzida no ES6.

Exemplo de Função Anônima:

```
let saudacao = function(nome) {  
    console.log("Olá, " + nome + "!");  
};  
  
saudacao("Ana"); // Olá, Ana!
```

Exemplo de Arrow Function:

```
let saudacao = (nome) => {  
    console.log("Olá, " + nome + "!");  
};  
  
saudacao("Carlos"); // Olá, Carlos!
```

Explicação:

- `let saudacao = function(nome) { ... };`: Declara uma função anônima e a atribui à variável `saudacao`.
- `let saudacao = (nome) => { ... };`: Declara uma arrow function e a atribui à variável `saudacao`.

2.6.7 Exemplos Práticos

Vamos praticar a criação e uso de funções em JavaScript. Crie um arquivo chamado `funcoes.js` e adicione o seguinte código:

Passo 1: Declarando e Chamando Funções

```
function saudacao() {  
    console.log("Olá, mundo!");  
}
```

```
saudacao(); // Olá, mundo!
```

Passo 2: Funções com Parâmetros

```
function saudacao(nome) {
  console.log("Olá, " + nome + "!");
}

saudacao("Maria"); // Olá, Maria!
saudacao("João"); // Olá, João!
```

Passo 3: Funções com Retorno

```
function multiplicacao(a, b) {
  return a * b;
}

let resultado = multiplicacao(5, 7);
console.log(resultado); // 35
```

Passo 4: Funções Anônimas

```
let saudacao = function(nome) {
  console.log("Olá, " + nome + "!");
};

saudacao("Ana"); // Olá, Ana!
```

Passo 5: Arrow Functions

```
let saudacao = (nome) => {
  console.log("Olá, " + nome + "!");
};
```

```
saudacao("Carlos"); // Olá, Carlos!
```

Passo 6: Executando o Código com Node.js

Para executar o código, abra o terminal e navegue até o diretório onde está o arquivo `funcoes.js`. Em seguida, execute o comando:

```
node funcoes.js
```

Você deve ver a saída do console conforme descrito nos exemplos acima.



Exercícios Práticos

1. Crie um arquivo chamado `calculo_area.js` e declare uma função chamada `calculaArea` que aceite o comprimento e a largura de um retângulo e retorne a área. 
2. Crie uma função chamada `parOuImpar` que aceite um número e exiba "Par" se o número for par e "Ímpar" se o número for ímpar. 
3. Crie uma função anônima que aceite dois números e retorne a soma deles. Atribua essa função a uma variável chamada `soma` e chame-a com diferentes valores. 
4. Escreva uma arrow function que aceite um nome e exiba uma mensagem de boas-vindas personalizada. 
5. Crie uma função chamada `converterCelsiusParaFahrenheit` que aceite uma temperatura em Celsius e retorne a temperatura convertida para Fahrenheit. 

2.7 Objetos

Os objetos são uma das principais estruturas de dados em JavaScript, permitindo a criação de coleções de pares chave-valor. Eles são extremamente versáteis e

fundamentais para a programação orientada a objetos. Vamos explorar como criar e manipular objetos em JavaScript. 

2.7.1 Introdução aos Objetos

Em JavaScript, um objeto é uma coleção de propriedades, onde cada propriedade é definida como um par chave-valor. As chaves são strings (ou símbolos) e os valores podem ser de qualquer tipo, incluindo outros objetos.

2.7.2 Criando Objetos

Existem várias maneiras de criar objetos em JavaScript. A forma mais comum é utilizando a notação literal de objeto.

Exemplo de Criação de Objeto:

```
let pessoa = {  
    nome: "João",  
    idade: 30,  
    cidade: "São Paulo"  
};  
  
console.log(pessoa);
```

Explicação:

- `let pessoa = { ... };`: Declara um objeto chamado `pessoa` com propriedades `nome`, `idade` e `cidade`.
- `console.log(pessoa);`: Exibe o objeto `pessoa` no console.

2.7.3 Acessando Propriedades de um Objeto

Você pode acessar as propriedades de um objeto utilizando a notação de ponto `.` ou a notação de colchetes `[]`.

Exemplo de Acesso a Propriedades:

```
console.log(pessoa.nome); // João  
console.log(pessoa["idade"]); // 30
```

Explicação:

- `pessoa.nome`: Acessa a propriedade `nome` do objeto `pessoa` usando a notação de ponto.
- `pessoa["idade"]`: Acessa a propriedade `idade` do objeto `pessoa` usando a notação de colchetes.

2.7.4 Modificando Propriedades de um Objeto

Você pode modificar as propriedades de um objeto atribuindo novos valores às chaves existentes.

Exemplo de Modificação de Propriedades:

```
pessoa.idade = 31;  
pessoa["cidade"] = "Rio de Janeiro";  
  
console.log(pessoa);
```

Explicação:

- `pessoa.idade = 31;`: Modifica a propriedade `idade` do objeto `pessoa` para `31`.
- `pessoa["cidade"] = "Rio de Janeiro";`: Modifica a propriedade `cidade` do objeto `pessoa` para "Rio de Janeiro".

2.7.5 Adicionando e Removendo Propriedades de um Objeto

Você pode adicionar novas propriedades a um objeto ou remover propriedades existentes.

Exemplo de Adição de Propriedades:

```
pessoa.profissao = "Engenheiro";
console.log(pessoa);
```

Exemplo de Remoção de Propriedades:

```
delete pessoa.cidade;
console.log(pessoa);
```

Explicação:

- `pessoa.profissao = "Engenheiro";`: Adiciona a propriedade `profissao` ao objeto `pessoa`.
- `delete pessoa.cidade;`: Remove a propriedade `cidade` do objeto `pessoa`.

2.7.6 Métodos em Objetos

Os objetos podem ter métodos, que são funções definidas como propriedades.

Exemplo de Método em Objeto:

```
let pessoa = {
  nome: "João",
  idade: 30,
  saudacao: function() {
    console.log("Olá, meu nome é " + this.nome);
  }
};

pessoa.saudacao(); // Olá, meu nome é João
```

Explicação:

- `saudacao: function() { ... }`: Define um método `saudacao` no objeto `pessoa`.
- `this.nome`: Refere-se à propriedade `nome` do objeto `pessoa` dentro do método `saudacao`.

- `pessoa.saudacao();`: Chama o método `saudacao` do objeto `pessoa`.

2.7.7 Iterando sobre Propriedades de um Objeto

Você pode iterar sobre as propriedades de um objeto usando o loop `for...in`.

Exemplo de Iteração com `for...in`:

```
for (let chave in pessoa) {
  console.log(chave + ": " + pessoa[chave]);
}
```

Explicação:

- `for (let chave in pessoa) { ... }`: Itera sobre cada propriedade do objeto `pessoa`.
- `console.log(chave + ": " + pessoa[chave]);`: Exibe a chave e o valor de cada propriedade do objeto `pessoa`.

2.7.8 Exercícios Práticos

Vamos praticar a criação e manipulação de objetos em JavaScript. Crie um arquivo chamado `objetos.js` e adicione o seguinte código:

Passo 1: Criando um Objeto

```
let carro = {
  marca: "Toyota",
  modelo: "Corolla",
  ano: 2020
};

console.log(carro);
```

Passo 2: Acessando Propriedades de um Objeto

```
console.log(carro.marca); // Toyota  
console.log(carro["modelo"]); // Corolla
```

Passo 3: Modificando Propriedades de um Objeto

```
carro.ano = 2021;  
carro["modelo"] = "Camry";  
  
console.log(carro);
```

Passo 4: Adicionando e Removendo Propriedades de um Objeto

```
carro.cor = "Prata";  
console.log(carro);  
  
delete carro.ano;  
console.log(carro);
```

Passo 5: Métodos em Objetos

```
let pessoa = {  
    nome: "João",  
    idade: 30,  
    saudacao: function() {  
        console.log("Olá, meu nome é " + this.nome);  
    }  
};  
  
pessoa.saudacao(); // Olá, meu nome é João
```

Passo 6: Iterando sobre Propriedades de um Objeto

```
for (let chave in carro) {  
    console.log(chave + ": " + carro[chave]);
```

```
}
```

Passo 7: Executando o Código com Node.js

Para executar o código, abra o terminal e navegue até o diretório onde está o arquivo `objetos.js`. Em seguida, execute o comando:

```
node objetos.js
```

Você deve ver a saída do console conforme descrito nos exemplos acima.



Desafios para Praticar

1. Crie um arquivo chamado `livro.js` e declare um objeto chamado `livro` com as propriedades `titulo`, `autor` e `ano`.
2. Adicione um método chamado `descricao` ao objeto `livro` que exiba uma mensagem com o título e o autor do livro.
3. Modifique o valor da propriedade `ano` do objeto `livro` e exiba o objeto atualizado no console.
4. Adicione uma nova propriedade chamada `genero` ao objeto `livro` e remova a propriedade `ano`.
5. Use um loop `for...in` para iterar sobre todas as propriedades do objeto `livro` e exibir suas chaves e valores no console.

2.8 Strings

As strings são uma das principais estruturas de dados em JavaScript, usadas para representar e manipular texto. Vamos explorar como criar, manipular e usar strings em JavaScript.

2.8.1 Introdução às Strings

Uma string é uma sequência de caracteres usada para representar texto. Em JavaScript, você pode criar strings usando aspas simples `'`, aspas duplas `"` ou crases ``` para templates strings.

Exemplo de Criação de Strings:

```
let saudacao = "Olá, mundo!";
let nome = 'João';
let mensagem = `Bem-vindo, ${nome}!`;

console.log(saudacao); // Olá, mundo!
console.log(nome); // João
console.log(mensagem); // Bem-vindo, João!
```

Explicação:

- `let saudacao = "Olá, mundo!"`: Declara uma string usando aspas duplas.
- `let nome = 'João'`: Declara uma string usando aspas simples.
- `let mensagem = Bem-vindo, ${nome}!;`: Declara uma template string usando crases e interpolação de variável `${}`.

2.8.2 Métodos Comuns de Strings

JavaScript fornece vários métodos para manipular strings. Aqui estão alguns dos mais comuns:

- `length`: Retorna o comprimento da string.
- `toUpperCase()`: Converte a string para maiúsculas.
- `toLowerCase()`: Converte a string para minúsculas.
- `indexOf()`: Retorna o índice da primeira ocorrência de um valor especificado.
- `slice()`: Extrai uma parte da string e retorna a parte extraída.
- `replace()`: Substitui um valor especificado por outro valor em uma string.

- `split()`: Divide a string em um array de substrings.

Exemplos de Uso de Métodos de Strings:

```
let texto = "JavaScript é incrível!";
console.log(texto.length); // 21
console.log(texto.toUpperCase()); // JAVASCRIPT É INCRÍVEL!
console.log(texto.toLowerCase()); // javascript é incrível!
console.log(texto.indexOf("incrível")); // 14
console.log(texto.slice(0, 10)); // JavaScript
console.log(texto.replace("incrível", "fantástico")); // Java
Script é fantástico!
console.log(texto.split(" ")); // ["JavaScript", "é", "incrív
el!"]
```

Explicação:

- `texto.length`: Retorna o comprimento da string.
- `texto.toUpperCase()`: Converte a string para maiúsculas.
- `texto.toLowerCase()`: Converte a string para minúsculas.
- `texto.indexOf("incrível")`: Retorna o índice da primeira ocorrência da palavra "incrível".
- `texto.slice(0, 10)`: Extrai os primeiros 10 caracteres da string.
- `texto.replace("incrível", "fantástico")`: Substitui "incrível" por "fantástico" na string.
- `texto.split(" ")`: Divide a string em um array de substrings, usando o espaço como delimitador.

2.8.3 Concatenando Strings

Você pode concatenar (juntar) strings usando o operador `+` ou template strings.

Exemplo de Concatenar Strings:

```

let saudacao = "Olá";
let nome = "João";
let mensagem = saudacao + ", " + nome + "!";
console.log(mensagem); // Olá, João!

// Usando template strings
let mensagemTemplate = `${saudacao}, ${nome}!`;
console.log(mensagemTemplate); // Olá, João!

```

Explicação:

- `let mensagem = saudacao + ", " + nome + "!"`: Concatena as strings `saudacao`, `", "`, `nome` e `"!"`.
- `let mensagemTemplate = `${saudacao}, ${nome}!``: Usa template strings para criar a mensagem.

2.8.4 Escapando Caracteres em Strings

Para incluir aspas dentro de uma string, você pode usar o caractere de escape `\\"`.

Exemplo de Escapar Caracteres:

```

let frase = "Ele disse: \\\"JavaScript é incrível!\\\"";
console.log(frase); // Ele disse: "JavaScript é incrível!"

```

Explicação:

- `"Ele disse: \\\"JavaScript é incrível!\\\""`: Usa `\\\"` para incluir aspas duplas dentro da string.

2.8.5 Comparando Strings

Você pode comparar strings usando os operadores de comparação `==`, `===`, `!=` e `!==`.

Exemplo de Comparação de Strings:

```
let a = "JavaScript";
let b = "JavaScript";
let c = "javascript";

console.log(a == b); // true
console.log(a === b); // true
console.log(a == c); // false
console.log(a === c); // false
```

Explicação:

- `a == b`: Verifica se `a` e `b` têm o mesmo valor.
- `a === b`: Verifica se `a` e `b` têm o mesmo valor e tipo.
- `a == c`: Verifica se `a` e `c` têm o mesmo valor (diferencia maiúsculas e minúsculas).
- `a === c`: Verifica se `a` e `c` têm o mesmo valor e tipo (diferencia maiúsculas e minúsculas).

2.8.6 Exercícios Práticos

Vamos praticar a criação e manipulação de strings em JavaScript. Crie um arquivo chamado `strings.js` e adicione o seguinte código:

Passo 1: Criando Strings

```
let saudacao = "Olá, mundo!";
let nome = 'João';
let mensagem = `Bem-vindo, ${nome}!`;

console.log(saudacao);
console.log(nome);
console.log(mensagem);
```

Passo 2: Usando Métodos de Strings

```
let texto = "JavaScript é incrível!";

console.log(texto.length);
console.log(texto.toUpperCase());
console.log(texto.toLowerCase());
console.log(texto.indexOf("incrível"));
console.log(texto.slice(0, 10));
console.log(texto.replace("incrível", "fantástico"));
console.log(texto.split(" "));
```

Passo 3: Concatenando Strings

```
let saudacao = "Olá";
let nome = "João";
let mensagem = saudacao + ", " + nome + "!";

console.log(mensagem);

let mensagemTemplate = `${saudacao}, ${nome}!`;
console.log(mensagemTemplate);
```

Passo 4: Escapando Caracteres

```
let frase = "Ele disse: \\\"JavaScript é incrível!\\\"";
console.log(frase);
```

Passo 5: Comparando Strings

```
let a = "JavaScript";
let b = "JavaScript";
let c = "javascript";

console.log(a == b);
console.log(a === b);
```

```
console.log(a == c);
console.log(a === c);
```

Passo 6: Executando o Código com Node.js

Para executar o código, abra o terminal e navegue até o diretório onde está o arquivo `strings.js`. Em seguida, execute o comando:

```
node strings.js
```

Você deve ver a saída do console conforme descrito nos exemplos acima.



Desafios para Praticar

1. Crie um arquivo chamado `mensagem_boas_vindas.js` e declare uma string que contenha uma mensagem de boas-vindas personalizada. Use métodos de strings para modificar e exibir a mensagem no console.
2. Declare uma string que represente uma frase e use o método `split` para dividi-la em palavras. Exiba o array de palavras no console.
3. Crie uma string que contenha uma citação famosa e use o método `replace` para substituir uma palavra por outra. Exiba a citação modificada no console.
4. Declare duas strings que representem nomes de pessoas e use operadores de comparação para verificar se os nomes são iguais. Exiba o resultado no console.
5. Crie uma string que contenha um parágrafo de texto e use o método `slice` para extrair uma frase específica. Exiba a frase extraída no console.

2.9 Conclusão do Capítulo

Neste capítulo, exploramos diversos conceitos fundamentais do JavaScript. Aprendemos sobre funções, incluindo a criação de funções com parâmetros e retornos, funções anônimas e arrow functions. Praticamos a criação e manipulação de objetos, uma das principais estruturas de dados em JavaScript, e vimos como adicionar, modificar e remover propriedades, bem como definir

métodos e iterar sobre as propriedades de um objeto. Também exploramos strings, incluindo métodos comuns para manipulá-las, concatenar strings, escapar caracteres e comparar strings.

Realizamos vários exercícios práticos para consolidar nosso entendimento, criando e executando códigos JavaScript que aplicam esses conceitos.

No próximo capítulo, mergulharemos no JavaScript com DOM (Document Object Model). Aprenderemos como o JavaScript pode interagir com os elementos de uma página web, permitindo a manipulação dinâmica do conteúdo e a criação de experiências interativas para o usuário. Prepare-se para descobrir como trazer suas páginas web à vida com JavaScript e DOM!

módulo 3

javascript com DOM

Interaja com sua página web de forma dinâmica
manipulando o DOM com JavaScript  

08 tópicos neste módulo

Módulo 3: JavaScript com DOM

Neste capítulo, vamos explorar como o JavaScript pode ser usado para interagir com o Document Object Model (DOM) para criar páginas web dinâmicas e interativas. O DOM é uma interface de programação que representa a estrutura de um documento HTML ou XML como uma árvore de nós. Cada elemento do HTML é um nó no DOM, e podemos usar JavaScript para manipular esses nós de diversas maneiras. Vamos aprender como selecionar elementos, modificar o conteúdo, adicionar interatividade e muito mais! 🌐✨

3.1 Introdução ao DOM

Cada elemento HTML, atributo e texto é um nó no DOM. Com JavaScript, podemos acessar e manipular esses nós para criar experiências de usuário mais dinâmicas e interativas. Vamos explorar os conceitos básicos do DOM e como interagir com ele usando JavaScript.

3.1.1 O que é o DOM?

O DOM é uma interface de programação que permite que scripts acessem e atualizem o conteúdo, a estrutura e o estilo de documentos HTML. Quando um navegador carrega uma página web, ele cria uma representação do documento como um objeto DOM, que pode ser manipulado por linguagens de script como JavaScript.

3.1.2 Estrutura do DOM

A estrutura do DOM é hierárquica, com um único nó raiz representando o documento completo. Abaixo do nó raiz, cada elemento HTML, atributo e texto é representado como um nó separado.

Exemplo de Estrutura do DOM:

```
<!DOCTYPE html>
<html>
  <head>
```

```
<title>Minha Página</title>
</head>
<body>
  <h1>Bem-vindo</h1>
  <p>Este é um parágrafo.</p>
</body>
</html>
```

Representação em Árvore do DOM:

- `html`
 - `head`
 - `title`
 - `#text` : "Minha Página"
 - `body`
 - `h1`
 - `#text` : "Bem-vindo"
 - `p`
 - `#text` : "Este é um parágrafo."

3.2 Selecionando Elementos

Selecionar elementos do DOM é uma habilidade essencial para qualquer desenvolvedor web. Com JavaScript, temos várias maneiras de selecionar e manipular esses elementos. Vamos explorar diferentes métodos de seleção de elementos e aprender como usá-los de maneira eficaz. 

3.2.1 Selecionando Elementos pelo ID

A maneira mais direta de selecionar um elemento específico é pelo seu ID. IDs devem ser únicos dentro de uma página HTML.

Exemplo de Seleção pelo ID:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Exemplo de Seleção por ID</title>
  </head>
  <body>
    <h1 id="titulo">Bem-vindo</h1>
    <script>
      let titulo = document.getElementById('titulo');
      console.log(titulo); // <h1 id="titulo">Bem-vindo</h1>
    </script>
  </body>
</html>
```

Explicação:

- `document.getElementById('titulo')` : Seleciona o elemento com o ID "titulo".

3.2.2 Selecionando Elementos pela Classe

Se você quiser selecionar vários elementos que compartilham a mesma classe, use `getElementsByClassName`. Isso retorna uma coleção de todos os elementos que possuem a classe especificada.

Exemplo de Seleção pela Classe:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Exemplo de Seleção por Classe</title>
  </head>
  <body>
```

```

<p class="texto">Parágrafo 1</p>
<p class="texto">Parágrafo 2</p>
<script>
    let paragrafos = document.getElementsByClassName('text
o');
    console.log(paragrafos); // HTMLCollection [ <p.texto>,
<p.texto> ]
</script>
</body>
</html>

```

Explicação:

- `document.getElementsByClassName('texto')`: Seleciona todos os elementos com a classe "texto".

3.2.3 Selecionando Elementos pela Tag

Para selecionar todos os elementos de um determinado tipo de tag (como `<p>`, `<div>`, etc.), use `getElementsByTagName`.

Exemplo de Seleção pela Tag:

```

<!DOCTYPE html>
<html>
    <head>
        <title>Exemplo de Seleção por Tag</title>
    </head>
    <body>
        <p>Parágrafo 1</p>
        <p>Parágrafo 2</p>
        <script>
            let todosParagrafos = document.getElementsByTagName(
('p'));
            console.log(todosParagrafos); // HTMLCollection [ <p>,
<p> ]
        </script>

```

```
</body>
</html>
```

Explicação:

- `document.getElementsByTagName('p')`: Seleciona todos os elementos `<p>`.

3.2.4 Selezionando Elementos com Query Selector

Os métodos `querySelector` e `querySelectorAll` oferecem uma maneira mais flexível de selecionar elementos, permitindo usar seletores CSS.

Exemplo de Seleção com Query Selector:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Query Selector</title>
  </head>
  <body>
    <p class="texto">Parágrafo 1</p>
    <p class="texto">Parágrafo 2</p>
    <script>
      // Seleciona o primeiro elemento que corresponde ao se
      let primeiroParagrafo = document.querySelector('.text
      o');
      console.log(primeiroParagrafo); // <p class="texto">Par
      ágrafo 1</p>

      // Seleciona todos os elementos que correspondem ao sel
      let todosParagrafos = document.querySelectorAll('.text
      o');
      console.log(todosParagrafos); // NodeList [ <p.texto>,
      <p.texto> ]
```

```
</script>
</body>
</html>
```

Explicação:

- `document.querySelector('.texto')` : Seleciona o primeiro elemento que corresponde ao seletor CSS ".texto".
- `document.querySelectorAll('.texto')` : Seleciona todos os elementos que correspondem ao seletor CSS ".texto".

3.2.5 Exercícios Práticos

Vamos praticar a seleção de elementos do DOM em JavaScript. Crie arquivos HTML e adicione o seguinte código:

Passo 1: Selezionando Elementos pelo ID

```
<!DOCTYPE html>
<html>
  <head>
    <title>Seleção pelo ID</title>
  </head>
  <body>
    <h1 id="titulo">Bem-vindo</h1>
    <script>
      let titulo = document.getElementById('titulo');
      console.log(titulo); // <h1 id="titulo">Bem-vindo</h1>
    </script>
  </body>
</html>
```

Passo 2: Selezionando Elementos pela Classe

```
<!DOCTYPE html>
<html>
```

```
<head>
  <title>Seleção pela Classe</title>
</head>
<body>
  <p class="texto">Parágrafo 1</p>
  <p class="texto">Parágrafo 2</p>
  <script>
    let paragrafos = document.getElementsByClassName('text
o');
    console.log(paragrafos); // HTMLCollection [ <p.texto>,
<p.texto> ]
  </script>
</body>
</html>
```

Passo 3: Selecionando Elementos pela Tag

```
<!DOCTYPE html>
<html>
  <head>
    <title>Seleção pela Tag</title>
  </head>
  <body>
    <p>Parágrafo 1</p>
    <p>Parágrafo 2</p>
    <script>
      let todosParagrafos = document.getElementsByTagName(
'p');
      console.log(todosParagrafos); // HTMLCollection [ <p>,
<p> ]
    </script>
  </body>
</html>
```

Passo 4: Selecionando Elementos com Query Selector

```

<!DOCTYPE html>
<html>
  <head>
    <title>Query Selector</title>
  </head>
  <body>
    <p class="texto">Parágrafo 1</p>
    <p class="texto">Parágrafo 2</p>
    <script>
      let primeiroParagrafo = document.querySelector('.text
o');
      console.log(primeiroParagrafo); // <p class="texto">Par
ágrafo 1</p>

      let todosParagrafos = document.querySelectorAll('.text
o');
      console.log(todosParagrafos); // NodeList [ <p.texto>,
<p.texto> ]
    </script>
  </body>
</html>

```

Passo 5: Executando o Código no Navegador

Para executar o código, abra o arquivo HTML no seu navegador.



Desafios para Praticar

- Crie um arquivo chamado `selecao_desafios.html` e selecione um elemento pelo ID, modifique seu conteúdo e exiba o resultado no console.** 🎯
- Selecione todos os elementos com uma classe específica e modifique seu estilo (por exemplo, altere a cor do texto).** 🎨
- Use `querySelector` para selecionar um elemento específico e adicione uma nova classe a ele.** ✨

4. Use `querySelectorAll` para selecionar todos os elementos de uma tag específica e altere seu conteúdo. 
 5. Combine seletores CSS em `querySelector` para selecionar elementos de maneira mais precisa (por exemplo, selecione um elemento com uma classe específica dentro de um determinado ID). 
-

3.3 Manipulando Conteúdo

Agora que sabemos como selecionar elementos do DOM, vamos aprender a manipulá-los. Isso inclui modificar o conteúdo, os atributos e o estilo dos elementos. Manipular o DOM nos permite criar páginas web dinâmicas e interativas. 

3.3.1 Modificando o Conteúdo de Texto

Podemos alterar o conteúdo de texto de um elemento usando as propriedades `textContent` e `innerHTML`. A diferença é que `textContent` define apenas o texto enquanto `innerHTML` pode incluir HTML.

Exemplo de Modificação de Conteúdo de Texto:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Modificando Conteúdo de Texto</title>
  </head>
  <body>
    <h1 id="titulo">Bem-vindo</h1>
    <p id="descricao">Este é um parágrafo.</p>

    <script>
      let titulo = document.getElementById('titulo');
      titulo.textContent = 'Olá, Mundo!';
    </script>
```

```

        let descricao = document.getElementById('descricao');
        descricao.innerHTML = 'Este é um <strong>parágrafo</strong> modificado.';
    </script>
</body>
</html>

```

Explicação:

- `titulo.textContent = 'Olá, Mundo!'`: Modifica o conteúdo de texto do elemento `titulo`.
- `descricao.innerHTML = 'Este é um parágrafo modificado.'`: Modifica o conteúdo HTML do elemento `descricao`.

3.3.2 Modificando Atributos

Você pode adicionar ou modificar atributos de um elemento usando os métodos `setAttribute` e `getAttribute`.

Exemplo de Modificação de Atributos:

```

<!DOCTYPE html>
<html>
<head>
    <title>Modificando Atributos</title>
</head>
<body>
    

    <script>
        let imagem = document.getElementById('imagem');
        imagem.setAttribute('src', 'imagem2.jpg');
        imagem.setAttribute('alt', 'Imagen 2');
    </script>

```

```
</body>
</html>
```

Explicação:

- `imagem.setAttribute('src', 'imagem2.jpg')`: Modifica o atributo `src` do elemento `imagem`.
- `imagem.setAttribute('alt', 'Imagen 2')`: Modifica o atributo `alt` do elemento `imagem`.

3.3.3 Modificando Estilos

Podemos alterar os estilos CSS de um elemento usando a propriedade `style`.

Exemplo de Modificação de Estilos:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Modificando Estilos</title>
  </head>
  <body>
    <p id="paragrafo">Este é um parágrafo estilizado.</p>

    <script>
      let paragrafo = document.getElementById('paragrafo');
      paragrafo.style.color = 'blue';
      paragrafo.style.fontSize = '20px';
      paragrafo.style.backgroundColor = 'yellow';
    </script>
  </body>
</html>
```

Explicação:

- `paragrafo.style.color = 'blue'` : Modifica a cor do texto do elemento `paragrafo` para azul.
- `paragrafo.style.fontSize = '20px'` : Modifica o tamanho da fonte do texto do elemento `paragrafo` para 20 pixels.
- `paragrafo.style.backgroundColor = 'yellow'` : Modifica a cor de fundo do elemento `paragrafo` para amarelo.

3.3.4 Adicionando e Removendo Classes

Adicionar e remover classes é uma maneira eficiente de aplicar estilos CSS aos elementos. Podemos usar os métodos `classList.add`, `classList.remove` e `classList.toggle`.

Exemplo de Adição e Remoção de Classes:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Adicionando e Removendo Classes</title>
    <style>
      .destacado {
        color: red;
        font-weight: bold;
      }
    </style>
  </head>
  <body>
    <p id="paragrafo">Este é um parágrafo.</p>

    <script>
      let paragrafo = document.getElementById('paragrafo');
      paragrafo.classList.add('destacado'); // Adiciona a classe 'destacado'
      console.log(paragrafo.className); // "destacado"
    </script>
  </body>
</html>
```

```

        paragrafo.classList.remove('destacado'); // Remove a classe 'destacado'
        console.log(paragrafo.className); // ""

        paragrafo.classList.toggle('destacado'); // Adiciona a classe 'destacado'
        console.log(paragrafo.className); // "destacado"

        paragrafo.classList.toggle('destacado'); // Remove a classe 'destacado'
        console.log(paragrafo.className); // ""
    </script>
</body>
</html>

```

Explicação:

- `paragrafo.classList.add('destacado')`: Adiciona a classe `destacado` ao elemento `paragrafo`.
- `paragrafo.classList.remove('destacado')`: Remove a classe `destacado` do elemento `paragrafo`.
- `paragrafo.classList.toggle('destacado')`: Adiciona ou remove a classe `destacado` do elemento `paragrafo`, dependendo de sua presença.



Exercícios Práticos

Vamos praticar a manipulação de conteúdo, atributos e estilos em JavaScript. Crie arquivos HTML e adicione o seguinte código:

Passo 1: Modificando Conteúdo de Texto

```

<!DOCTYPE html>
<html>
<head>
    <title>Modificando Conteúdo de Texto</title>

```

```

</head>
<body>
    <h1 id="titulo">Bem-vindo</h1>
    <p id="descricao">Este é um parágrafo.</p>

    <script>
        let titulo = document.getElementById('titulo');
        titulo.textContent = 'Olá, Mundo!';

        let descricao = document.getElementById('descricao');
        descricao.innerHTML = 'Este é um <strong>parágrafo</strong> modificado.';
    </script>
</body>
</html>

```

Passo 2: Modificando Atributos

```

<!DOCTYPE html>
<html>
    <head>
        <title>Modificando Atributos</title>
    </head>
    <body>
        

        <script>
            let imagem = document.getElementById('imagem');
            imagem.setAttribute('src', 'imagem2.jpg');
            imagem.setAttribute('alt', 'Imagen 2');
        </script>
    </body>
</html>

```

Passo 3: Modificando Estilos

```

<!DOCTYPE html>
<html>
  <head>
    <title>Modificando Estilos</title>
  </head>
  <body>
    <p id="paragrafo">Este é um parágrafo estilizado.</p>

    <script>
      let paragrafo = document.getElementById('paragrafo');
      paragrafo.style.color = 'blue';
      paragrafo.style.fontSize = '20px';
      paragrafo.style.backgroundColor = 'yellow';
    </script>
  </body>
</html>

```

Passo 4: Adicionando e Removendo Classes

```

<!DOCTYPE html>
<html>
  <head>
    <title>Adicionando e Removendo Classes</title>
    <style>
      .destacado {
        color: red;
        font-weight: bold;
      }
    </style>
  </head>
  <body>
    <p id="paragrafo">Este é um parágrafo.</p>

    <script>
      let paragrafo = document.getElementById('paragrafo');

```

```

    paragrafo.classList.add('destacado'); // Adiciona a cla
    sse 'destacado'
    console.log(paragrafo.className); // "destacado"

    paragrafo.classList.remove('destacado'); // Remove a cl
    asse 'destacado'
    console.log(paragrafo.className); // ""

    paragrafo.classList.toggle('destacado'); // Adiciona a
    classe 'destacado'
    console.log(paragrafo.className); // "destacado"

    paragrafo.classList.toggle('destacado'); // Remove a cl
    asse 'destacado'
    console.log(paragrafo.className); // ""

```

</script>

</body>

</html>



Desafios para Praticar

- Crie um arquivo chamado `manipulacao_conteudo.html` e modifique o conteúdo de texto e HTML de um elemento.**
- Modifique os atributos de uma imagem, alterando o `src` e o `alt`, e exiba o resultado no console.**
- Adicione e remova estilos de um elemento usando a propriedade `style`, como cor de fundo, cor do texto e tamanho da fonte.**
- Crie uma função que adicione uma classe a um elemento ao clicar em um botão e remova a classe ao clicar novamente (usando `classList.toggle`).**
- Combine várias manipulações (conteúdo, atributos, estilos) em uma função que altera completamente a aparência e o conteúdo de um elemento ao clicar em um botão.**

3.4 Eventos e Interatividade

Eventos são ações ou ocorrências que acontecem na página web, como cliques do mouse, pressionamento de teclas, ou carregamento de uma página. Em JavaScript, podemos usar eventos para tornar nossas páginas mais interativas e dinâmicas. Vamos explorar como lidar com eventos e criar interações na nossa página. 🚀✨

3.4.1 Introdução aos Eventos

Os eventos são a base da interatividade em JavaScript. Cada vez que algo acontece no navegador, como um clique do mouse ou uma tecla pressionada, um evento é disparado. Podemos usar JavaScript para "escutar" esses eventos e responder a eles.

3.4.2 Adicionando Event Listeners

Um Event Listener é uma função que espera por um evento específico e executa um código quando esse evento ocorre. Para adicionar um Event Listener a um elemento, usamos o método `addEventListener`.

Exemplo de Adição de Event Listener:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Adicionando Event Listeners</title>
  </head>
  <body>
    <button id="meuBotao">Clique em mim!</button>
    <script>
      let botao = document.getElementById('meuBotao');
```

```

botao.addEventListener('click', function() {
    alert('Botão clicado!');
});
</script>
</body>
</html>

```

Explicação:

- `botao.addEventListener('click', function() { ... });`: Adiciona um Event Listener que escuta por cliques no botão e executa a função fornecida quando o botão é clicado.

3.4.3 Eventos Comuns

Existem muitos eventos diferentes que podemos usar em JavaScript. Aqui estão alguns dos mais comuns:

- `click`: Disparado quando um elemento é clicado.
- `mouseover`: Disparado quando o ponteiro do mouse passa sobre um elemento.
- `mouseout`: Disparado quando o ponteiro do mouse deixa um elemento.
- `keydown`: Disparado quando uma tecla é pressionada.
- `keyup`: Disparado quando uma tecla é solta.

Exemplo de Uso de Diferentes Eventos:

```

<!DOCTYPE html>
<html>
    <head>
        <title>Eventos Comuns</title>
    </head>
    <body>
        <button id="botaoClique">Clique em mim!</button>
        <p id="texto">Passe o mouse aqui!</p>
        <input id="campoTexto" type="text" placeholder="Digite al

```

```
go...">

<script>
    let botao = document.getElementById('botaoClique');
    let texto = document.getElementById('texto');
    let campoTexto = document.getElementById('campoTexto');

    // Evento de clique
    botao.addEventListener('click', function() {
        alert('Botão clicado!');
    });

    // Evento de mouseover
    texto.addEventListener('mouseover', function() {
        texto.style.color = 'blue';
    });

    // Evento de mouseout
    texto.addEventListener('mouseout', function() {
        texto.style.color = 'black';
    });

    // Evento de keydown
    campoTexto.addEventListener('keydown', function(event)
    {
        console.log('Tecla pressionada: ' + event.key);
    });

    // Evento de keyup
    campoTexto.addEventListener('keyup', function(event) {
        console.log('Tecla liberada: ' + event.key);
    });
</script>
</body>
</html>
```

Explicação:

- `click`: Adiciona um alert quando o botão é clicado.
- `mouseover` e `mouseout`: Mudam a cor do texto quando o mouse passa sobre ele e quando o mouse deixa o elemento.
- `keydown` e `keyup`: Exibem no console a tecla pressionada e liberada no campo de texto.

3.4.4 Removendo Event Listeners

Podemos remover Event Listeners usando o método `removeEventListener`. Para isso, precisamos de uma referência à função que foi usada na adição do listener.

Exemplo de Remoção de Event Listener:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Removendo Event Listeners</title>
  </head>
  <body>
    <button id="botaoAdicionar">Adicionar Listener</button>
    <button id="botaoRemover">Remover Listener</button>
    <button id="meuBotao">Clique em mim!</button>

    <script>
      let botaoAdicionar = document.getElementById('botaoAdicionar');
      let botaoRemover = document.getElementById('botaoRemover');
      let meuBotao = document.getElementById('meuBotao');

      function mostrarAlerta() {
        alert('Botão clicado!');
      }
```

```

    // Adiciona o listener
    botaoAdicionar.addEventListener('click', function() {
        meuBotao.addEventListener('click', mostrarAlerta);
    });

    // Remove o listener
    botaoRemover.addEventListener('click', function() {
        meuBotao.removeEventListener('click', mostrarAlerta);
    });
</script>
</body>
</html>

```

Explicação:

- `meuBotao.addEventListener('click', mostrarAlerta)`: Adiciona o listener `mostrarAlerta` ao botão.
- `meuBotao.removeEventListener('click', mostrarAlerta)`: Remove o listener `mostrarAlerta` do botão.



Exercícios Práticos

Vamos praticar a manipulação de eventos e criar interatividade em JavaScript. Crie arquivos HTML e adicione o seguinte código:

Passo 1: Adicionando Event Listener

```

<!DOCTYPE html>
<html>
    <head>
        <title>Adicionando Event Listener</title>
    </head>
    <body>
        <button id="meuBotao">Clique em mim!</button>
        <script>

```

```

let botao = document.getElementById('meuBotao');
botao.addEventListener('click', function() {
    alert('Botão clicado!');
});
</script>
</body>
</html>

```

Passo 2: Usando Diferentes Eventos

```

<!DOCTYPE html>
<html>
    <head>
        <title>Usando Diferentes Eventos</title>
    </head>
    <body>
        <button id="botaoClique">Clique em mim!</button>
        <p id="texto">Passe o mouse aqui!</p>
        <input id="campoTexto" type="text" placeholder="Digite algo...">

        <script>
            let botao = document.getElementById('botaoClique');
            let texto = document.getElementById('texto');
            let campoTexto = document.getElementById('campoTexto');

            // Evento de clique
            botao.addEventListener('click', function() {
                alert('Botão clicado!');
            });

            // Evento de mouseover
            texto.addEventListener('mouseover', function() {
                texto.style.color = 'blue';
            });
        </script>
    </body>
</html>

```

```

        // Evento de mouseout
        texto.addEventListener('mouseout', function() {
            texto.style.color = 'black';
        });

        // Evento de keydown
        campoTexto.addEventListener('keydown', function(event)
        {
            console.log('Tecla pressionada: ' + event.key);
        });

        // Evento de keyup
        campoTexto.addEventListener('keyup', function(event) {
            console.log('Tecla liberada: ' + event.key);
        });
    </script>
</body>
</html>

```

Passo 3: Removendo Event Listener

```

<!DOCTYPE html>
<html>
    <head>
        <title>Removendo Event Listener</title>
    </head>
    <body>
        <button id="botaoAdicionar">Adicionar Listener</button>
        <button id="botaoRemover">Remover Listener</button>
        <button id="meuBotao">Clique em mim!</button>

        <script>
            let botaoAdicionar = document.getElementById('botaoAdicionar');
            let botaoRemover = document.getElementById('botaoRemover');

```

```

let meuBotao = document.getElementById('meuBotao');

function mostrarAlerta() {
    alert('Botão clicado!');
}

// Adiciona o listener
botaoAdicionar.addEventListener('click', function() {
    meuBotao.addEventListener('click', mostrarAlerta);
});

// Remove o listener
botaoRemover.addEventListener('click', function() {
    meuBotao.removeEventListener('click', mostrarAlerta);
});

</script>
</body>
</html>

```



Desafios para Praticar

1. Crie um arquivo chamado `eventos_interatividade.html` e adicione um Event Listener a um botão que altera o texto de um parágrafo ao ser clicado.
2. Adicione Event Listeners para `mouseover` e `mouseout` em uma imagem, alterando seu estilo ao passar o mouse sobre ela e ao sair.
3. Crie um campo de texto que exiba a contagem de caracteres digitados em tempo real usando os eventos `keydown` e `keyup`.
4. Implemente um botão que adiciona e remove uma classe de um parágrafo, alternando seu estilo ao ser clicado.
5. Combine vários eventos para criar uma interatividade mais complexa, como um formulário que exibe uma mensagem de sucesso ao ser enviado, com validação dos campos.

3.5 Manipulação de Estilos

Além de manipular o conteúdo e responder a eventos, JavaScript permite que você altere os estilos dos elementos dinamicamente. Isso é útil para criar efeitos visuais, realçar informações importantes e melhorar a interatividade da página.

Vamos explorar como manipular estilos usando JavaScript. 🌟🌟

3.5.1 Alterando Estilos com a Propriedade `style`

A maneira mais direta de alterar o estilo de um elemento é usando a propriedade `style`. Podemos modificar qualquer propriedade CSS diretamente em JavaScript.

Exemplo de Alteração de Estilos:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Alterando Estilos</title>
    <style>
      #paragrafo {
        color: black;
        font-size: 16px;
        background-color: white;
      }
    </style>
  </head>
  <body>
    <p id="paragrafo">Este é um parágrafo estilizado.</p>

    <script>
      let paragrafo = document.getElementById('paragrafo');
      paragrafo.style.color = 'blue';
    </script>
  </body>
</html>
```

```
    paragrafo.style.fontSize = '20px';
    paragrafo.style.backgroundColor = 'lightgray';

```

</script>

```
</body>
</html>
```

Explicação:

- `paragrafo.style.color = 'blue';` : Altera a cor do texto para azul.
- `paragrafo.style.fontSize = '20px';` : Altera o tamanho da fonte para 20 pixels.
- `paragrafo.style.backgroundColor = 'lightgray';` : Altera a cor de fundo para cinza claro.

3.5.2 Adicionando e Removendo Classes

Adicionar e remover classes é uma maneira eficaz de aplicar estilos predefinidos aos elementos. Isso permite que você mantenha seu CSS separado do JavaScript, promovendo uma melhor organização do código.

Exemplo de Adição e Remoção de Classes:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Adicionando e Removendo Classes</title>
    <style>
      .destacado {
        color: red;
        font-weight: bold;
      }
    </style>
  </head>
  <body>
    <p id="paragrafo">Este é um parágrafo.</p>
    <button id="botaoAdicionar">Adicionar Classe</button>
```

```

<button id="botaoRemover">Remover Classe</button>

<script>
    let paragrafo = document.getElementById('paragrafo');
    let botaoAdicionar = document.getElementById('botaoAdicionar');
    let botaoRemover = document.getElementById('botaoRemover');

    botaoAdicionar.addEventListener('click', function() {
        paragrafo.classList.add('destacado');
    });

    botaoRemover.addEventListener('click', function() {
        paragrafo.classList.remove('destacado');
    });
</script>
</body>
</html>

```

Explicação:

- `paragrafo.classList.add('destacado');`: Adiciona a classe `destacado` ao elemento `paragrafo`.
- `paragrafo.classList.remove('destacado');`: Remove a classe `destacado` do elemento `paragrafo`.

3.5.3 Alternando Classes com `toggle`

O método `classList.toggle` adiciona uma classe ao elemento se ela não estiver presente, ou remove a classe se ela estiver presente.

Exemplo de Alternância de Classes:

```

<!DOCTYPE html>
<html>

```

```

<head>
    <title>Alternando Classes</title>
    <style>
        .destacado {
            color: red;
            font-weight: bold;
        }
    </style>
</head>
<body>
    <p id="paragrafo">Este é um parágrafo.</p>
    <button id="botaoToggle">Alternar Classe</button>

    <script>
        let paragrafo = document.getElementById('paragrafo');
        let botaoToggle = document.getElementById('botaoToggle');

        botaoToggle.addEventListener('click', function() {
            paragrafo.classList.toggle('destacado');
        });
    </script>
</body>
</html>

```

Explicação:

- `paragrafo.classList.toggle('destacado');`: Alterna a classe `destacado` no elemento `paragrafo`.

3.5.4 Alterando Vários Estilos de Uma Vez

Podemos definir múltiplos estilos de uma vez usando uma string CSS.

Exemplo de Definição de Múltiplos Estilos:

```

<!DOCTYPE html>
<html>
  <head>
    <title>Definindo Múltiplos Estilos</title>
    <style>
      #paragrafo {
        transition: all 0.5s;
      }
    </style>
  </head>
  <body>
    <p id="paragrafo">Este é um parágrafo estilizado.</p>
    <button id="botaoEstilos">Alterar Estilos</button>

    <script>
      let paragrafo = document.getElementById('paragrafo');
      let botaoEstilos = document.getElementById('botaoEstilos');

      botaoEstilos.addEventListener('click', function() {
        paragrafo.style.cssText = 'color: green; font-size: 25px; background-color: lightyellow;';
      });
    </script>
  </body>
</html>

```

Explicação:

- `paragrafo.style.cssText = 'color: green; font-size: 25px; background-color: lightyellow;'`: Define múltiplos estilos no elemento `paragrafo`.



Exercícios Práticos

Vamos praticar a manipulação de estilos em JavaScript. Crie arquivos HTML e adicione o seguinte código:

Passo 1: Alterando Estilos com a Propriedade `style`

```
<!DOCTYPE html>
<html>
  <head>
    <title>Alterando Estilos</title>
    <style>
      #paragrafo {
        color: black;
        font-size: 16px;
        background-color: white;
      }
    </style>
  </head>
  <body>
    <p id="paragrafo">Este é um parágrafo estilizado.</p>
    <button id="botaoEstilos">Alterar Estilos</button>

    <script>
      let paragrafo = document.getElementById('paragrafo');
      let botaoEstilos = document.getElementById('botaoEstilos');

      botaoEstilos.addEventListener('click', function() {
        paragrafo.style.color = 'blue';
        paragrafo.style.fontSize = '20px';
        paragrafo.style.backgroundColor = 'lightgray';
      });
    </script>
  </body>
</html>
```

Passo 2: Adicionando e Removendo Classes

```

<!DOCTYPE html>
<html>
  <head>
    <title>Adicionando e Removendo Classes</title>
    <style>
      .destacado {
        color: red;
        font-weight: bold;
      }
    </style>
  </head>
  <body>
    <p id="paragrafo">Este é um parágrafo.</p>
    <button id="botaoAdicionar">Adicionar Classe</button>
    <button id="botaoRemover">Remover Classe</button>

    <script>
      let paragrafo = document.getElementById('paragrafo');
      let botaoAdicionar = document.getElementById('botaoAdicionar');
      let botaoRemover = document.getElementById('botaoRemover');

      botaoAdicionar.addEventListener('click', function() {
        paragrafo.classList.add('destacado');
      });

      botaoRemover.addEventListener('click', function() {
        paragrafo.classList.remove('destacado');
      });
    </script>
  </body>
</html>

```

Passo 3: Alternando Classes com `toggle`

```

<!DOCTYPE html>
<html>
  <head>
    <title>Alternando Classes</title>
    <style>
      .destacado {
        color: red;
        font-weight: bold;
      }
    </style>
  </head>
  <body>
    <p id="paragrafo">Este é um parágrafo.</p>
    <button id="botaoToggle">Alternar Classe</button>

    <script>
      let paragrafo = document.getElementById('paragrafo');
      let botaoToggle = document.getElementById('botaoToggle');

      botaoToggle.addEventListener('click', function() {
        paragrafo.classList.toggle('destacado');
      });
    </script>
  </body>
</html>

```

Passo 4: Alterando Vários Estilos de Uma Vez

```

<!DOCTYPE html>
<html>
  <head>
    <title>Definindo Múltiplos Estilos</title>
    <style>
      #paragrafo {

```

```

        transition: all 0.5s;
    }

```

```

</style>
</head>
<body>
<

p id="paragrafo">Este é um parágrafo estilizado.</p>
<button id="botaoEstilos">Alterar Estilos</button>

<script>
    let paragrafo = document.getElementById('paragrafo');
    let botaoEstilos = document.getElementById('botaoEstilos');

    botaoEstilos.addEventListener('click', function() {
        paragrafo.style.cssText = 'color: green; font-size: 2
5px; background-color: lightyellow;';
    });
</script>
</body>
</html>

```



Desafios para Praticar

- Crie um arquivo chamado `manipulacao_estilos.html` e altere os estilos de um elemento ao clicar em um botão, incluindo cor do texto, cor de fundo e tamanho da fonte.**
- Adicione Event Listeners para alterar os estilos de um elemento ao passar o mouse sobre ele e ao sair do elemento.**
- Implemente um botão que adiciona e remove uma classe de um elemento, alternando seu estilo ao ser clicado.**
- Use `classList.toggle` para alternar entre duas classes de estilo ao clicar em um botão.**

5. Crie uma função que altera múltiplos estilos de um elemento ao clicar em um botão, usando uma string CSS. 
-

3.6 Navegação no DOM

Navegar pelo DOM é essencial para manipular elementos de maneira eficiente. Isso envolve mover-se entre os nós do DOM para acessar elementos pai, filho e irmãos. Vamos explorar como navegar no DOM e entender a hierarquia dos elementos.  

3.6.1 Acessando Elementos Pai

Para acessar o elemento pai de um nó, usamos a propriedade `parentNode`. Isso nos permite subir um nível na hierarquia do DOM.

Exemplo de Acesso ao Elemento Pai:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Acessando Elementos Pai</title>
  </head>
  <body>
    <div id="pai">
      <p id="filho">Este é um parágrafo.</p>
    </div>

    <script>
      let filho = document.getElementById('filho');
      let pai = filho.parentNode;
      console.log(pai); // <div id="pai">...</div>
    </script>
```

```
</body>
</html>
```

Explicação:

- `filho.parentNode`: Acessa o elemento pai do parágrafo `filho`.

3.6.2 Acessando Elementos Filhos

Para acessar os elementos filhos de um nó, usamos a propriedade `childNodes` ou `children`. A diferença é que `childNodes` retorna todos os nós filhos, incluindo nós de texto, enquanto `children` retorna apenas os elementos filhos.

Exemplo de Acesso aos Elementos Filhos:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Acessando Elementos Filhos</title>
  </head>
  <body>
    <div id="pai">
      <p>Filho 1</p>
      <p>Filho 2</p>
      <p>Filho 3</p>
    </div>

    <script>
      let pai = document.getElementById('pai');
      let filhos = pai.children;
      console.log(filhos); // HTMLCollection [ <p>, <p>, <p> ]
    </script>
  </body>
</html>
```

Explicação:

- `pai.children`: Acessa os elementos filhos do `div` com ID `pai`.

3.6.3 Acessando Elementos Irmãos

Para acessar os elementos irmãos de um nó, usamos as propriedades `nextSibling` e `previousSibling` para acessar os irmãos adjacentes. Para acessar apenas os elementos irmãos, usamos `nextElementSibling` e `previousElementSibling`.

Exemplo de Acesso aos Elementos Irmãos:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Acessando Elementos Irmãos</title>
  </head>
  <body>
    <p id="primeiro">Primeiro parágrafo</p>
    <p id="segundo">Segundo parágrafo</p>
    <p id="terceiro">Terceiro parágrafo</p>

    <script>
      let primeiro = document.getElementById('primeiro');
      let segundo = primeiro.nextElementSibling;
      let terceiro = segundo.nextElementSibling;
      console.log(segundo); // <p id="segundo">Segundo parágrafo</p>
      console.log(terceiro); // <p id="terceiro">Terceiro parágrafo</p>
    </script>
  </body>
</html>
```

Explicação:

- `primeiro.nextElementSibling`: Acessa o próximo elemento irmão do `parágrafo` com ID `primeiro`.
- `segundo.nextElementSibling`: Acessa o próximo elemento irmão do `parágrafo` com ID `segundo`.

3.6.4 Criando e Removendo Nós

Podemos adicionar novos elementos ao DOM usando `createElement` e `appendChild`, e remover elementos com `removeChild`.

Exemplo de Criação e Remoção de Nós:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Criando e Removendo Nós</title>
  </head>
  <body>
    <div id="container">
      <p>Parágrafo existente</p>
    </div>
    <button id="adicionar">Adicionar Parágrafo</button>
    <button id="remover">Remover Parágrafo</button>

    <script>
      let container = document.getElementById('container');
      let botaoAdicionar = document.getElementById('adicionar');
      let botaoRemover = document.getElementById('remover');

      botaoAdicionar.addEventListener('click', function() {
        let novoParagrafo = document.createElement('p');
        novoParagrafo.textContent = 'Novo parágrafo';
        container.appendChild(novoParagrafo);
      });
    </script>
  </body>
</html>
```

```

botaoRemover.addEventListener('click', function() {
  let ultimoParagrafo = container.lastElementChild;
  if (ultimoParagrafo) {
    container.removeChild(ultimoParagrafo);
  }
});
</script>
</body>
</html>

```

Explicação:

- `document.createElement('p')`: Cria um novo elemento `<p>`.
- `container.appendChild(novoParagrafo)`: Adiciona o novo parágrafo ao `div` com ID `container`.
- `container.removeChild(ultimoParagrafo)`: Remove o último parágrafo filho do `div` com ID `container`.



Exercícios Práticos

Vamos praticar a navegação no DOM em JavaScript. Crie arquivos HTML e adicione o seguinte código:

Passo 1: Acessando Elementos Pai

```

<!DOCTYPE html>
<html>
  <head>
    <title>Acessando Elementos Pai</title>
  </head>
  <body>
    <div id="pai">
      <p id="filho">Este é um parágrafo.</p>
    </div>
  </body>
</html>

```

```
<script>
  let filho = document.getElementById('filho');
  let pai = filho.parentNode;
  console.log(pai); // <div id="pai">...</div>
</script>
</body>
</html>
```

Passo 2: Acessando Elementos Filhos

```
<!DOCTYPE html>
<html>
  <head>
    <title>Acessando Elementos Filhos</title>
  </head>
  <body>
    <div id="pai">
      <p>Filho 1</p>
      <p>Filho 2</p>
      <p>Filho 3</p>
    </div>

    <script>
      let pai = document.getElementById('pai');
      let filhos = pai.children;
      console.log(filhos); // HTMLCollection [ <p>, <p>, <p> ]
    </script>
  </body>
</html>
```

Passo 3: Acessando Elementos Irmãos

```
<!DOCTYPE html>
<html>
```

```

<head>
  <title>Acessando Elementos Irmãos</title>
</head>
<body>
  <p id="primeiro">Primeiro parágrafo</p>
  <p id="segundo">Segundo parágrafo</p>
  <p id="terceiro">Terceiro parágrafo</p>

  <script>
    let primeiro = document.getElementById('primeiro');
    let segundo = primeiro.nextElementSibling;
    let terceiro = segundo.nextElementSibling;
    console.log(segundo); // <p id="segundo">Segundo parágrafo</p>
    console.log(terceiro); // <p id="terceiro">Terceiro parágrafo</p>
  </script>
</body>
</html>

```

Passo 4: Criando e Removendo Nós

```

<!DOCTYPE html>
<html>
  <head>
    <title>Criando e Removendo Nós</title>
  </head>
  <body>
    <div id="container">
      <p>Parágrafo existente</p>
    </div>
    <button id="adicionar">Adicionar Parágrafo</button>
    <button id="remover">Remover Parágrafo</button>

    <script>
      let container = document.getElementById('container');

```

```

let botaoAdicionar = document.getElementById('adicionar');
let botaoRemover = document.getElementById('remover');

botaoAdicionar.addEventListener('click', function() {
  let novoParagrafo = document.createElement('p');
  novoParagrafo.textContent = 'Novo parágrafo';
  container.appendChild(novoParagrafo);
});

botaoRemover.addEventListener('click', function() {
  let ultimoParagrafo = container.lastElementChild;
  if (ultimoParagrafo) {
    container.removeChild(ultimoParagrafo);
  }
});
</script>
</body>
</html>

```



Desafios para Praticar

- Crie um arquivo chamado `navegacao_dom.html` e acesse o elemento pai de um parágrafo, alterando seu estilo.**
- Acesse todos os elementos filhos de um `div` e altere seu conteúdo de texto.**
- Implemente a navegação entre elementos irmãos, alterando o estilo do próximo e do anterior ao clicar em um botão.
- Crie uma função que adicione novos elementos ao DOM e outra que remova elementos existentes ao clicar em botões.**
- Combine várias manipulações (acesso a pai, filhos, irmãos) em uma função que altera a estrutura do DOM dinamicamente.**

3.7 Criando e Removendo Elementos

Manipular o DOM não se limita apenas a alterar elementos existentes. Podemos também criar novos elementos e adicioná-los ao DOM, bem como remover elementos. Esta habilidade é essencial para criar interfaces dinâmicas e interativas. Vamos explorar como criar e remover elementos no DOM. 

3.7.1 Criando Elementos

Para criar um novo elemento, usamos o método `document.createElement`. Depois de criar o elemento, podemos configurá-lo adicionando conteúdo ou atributos, e então adicioná-lo ao DOM com métodos como `appendChild` ou `insertBefore`.

Exemplo de Criação de Elementos:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Criando Elementos</title>
  </head>
  <body>
    <div id="container">
      <p>Parágrafo existente</p>
    </div>
    <button id="adicionar">Adicionar Parágrafo</button>

    <script>
      let container = document.getElementById('container');
      let botaoAdicionar = document.getElementById('adicionar');

      botaoAdicionar.addEventListener('click', function() {
        let novoParagrafo = document.createElement('p');
        novoParagrafo.textContent = 'Novo parágrafo criado dinamicamente!';
        container.appendChild(novoParagrafo);
      });
    </script>
  </body>
</html>
```

```
    });
  </script>
</body>
</html>
```

Explicação:

- `document.createElement('p')`: Cria um novo elemento `<p>`.
- `novoParagrafo.textContent = 'Novo parágrafo criado dinamicamente!'`: Define o conteúdo de texto do novo parágrafo.
- `container.appendChild(novoParagrafo)`: Adiciona o novo parágrafo ao `div` com ID `container`.

3.7.2 Removendo Elementos

Para remover um elemento do DOM, usamos o método `removeChild`. Primeiro, selecionamos o elemento pai e, em seguida, removemos o elemento filho desejado.

Exemplo de Remoção de Elementos:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Removendo Elementos</title>
  </head>
  <body>
    <div id="container">
      <p id="paragrafoRemover">Parágrafo que será removido</p>
    >
    </div>
    <button id="remover">Remover Parágrafo</button>

    <script>
      let container = document.getElementById('container');
```

```

let botaoRemover = document.getElementById('remover');
let paragrafoRemover = document.getElementById('paragrafoRemover');

botaoRemover.addEventListener('click', function() {
  if (paragrafoRemover) {
    container.removeChild(paragrafoRemover);
  }
});

</script>
</body>
</html>

```

Explicação:

- `container.removeChild(paragrafoRemover)`: Remove o parágrafo com ID `paragrafoRemover` do `div` com ID `container`.

3.7.3 Inserindo Elementos Antes de Outro Elemento

Podemos usar o método `insertBefore` para inserir um novo elemento antes de um elemento existente.

Exemplo de Inserção de Elemento Antes de Outro:

```

<!DOCTYPE html>
<html>
  <head>
    <title>Inserindo Elementos Antes</title>
  </head>
  <body>
    <div id="container">
      <p id="primeiro">Primeiro parágrafo</p>
      <p id="segundo">Segundo parágrafo</p>
    </div>
    <button id="inserirAntes">Inserir Parágrafo Antes</button>
  </body>

```

```

>

<script>
    let container = document.getElementById('container');
    let botaoInserirAntes = document.getElementById('inserirAntes');
    let segundo = document.getElementById('segundo');

    botaoInserirAntes.addEventListener('click', function()
    {
        let novoParagrafo = document.createElement('p');
        novoParagrafo.textContent = 'Parágrafo inserido antes do segundo!';
        container.insertBefore(novoParagrafo, segundo);
    });
</script>
</body>
</html>

```

Explicação:

- `container.insertBefore(novoParagrafo, segundo)`: Insere o novo parágrafo antes do parágrafo com ID `segundo` no `div` com ID `container`.

3.7.4 Substituindo Elementos

Podemos substituir um elemento existente por um novo elemento usando o método `replaceChild`.

Exemplo de Substituição de Elementos:

```

<!DOCTYPE html>
<html>
    <head>
        <title>Substituindo Elementos</title>
    </head>

```

```

<body>
  <div id="container">
    <p id="paragrafoOriginal">Parágrafo original</p>
  </div>
  <button id="substituir">Substituir Parágrafo</button>

  <script>
    let container = document.getElementById('container');
    let botaoSubstituir = document.getElementById('substituir');

    let paragrafoOriginal = document.getElementById('paragrafoOriginal');

    botaoSubstituir.addEventListener('click', function() {
      let novoParagrafo = document.createElement('p');
      novoParagrafo.textContent = 'Parágrafo substituto!';
      container.replaceChild(novoParagrafo, paragrafoOriginal);
    });
  </script>
</body>
</html>

```

Explicação:

- `container.replaceChild(novoParagrafo, paragrafoOriginal)` : Substitui o parágrafo com ID `paragrafoOriginal` pelo novo parágrafo no `div` com ID `container`.



Exercícios Práticos

Vamos praticar a criação, remoção, inserção e substituição de elementos no DOM em JavaScript. Crie arquivos HTML e adicione o seguinte código:

Passo 1: Criando Elementos

```

<!DOCTYPE html>
<html>
  <head>
    <title>Criando Elementos</title>
  </head>
  <body>
    <div id="container">
      <p>Parágrafo existente</p>
    </div>
    <button id="adicionar">Adicionar Parágrafo</button>

    <script>
      let container = document.getElementById('container');
      let botaoAdicionar = document.getElementById('adicionar');

      botaoAdicionar.addEventListener('click', function() {
        let novoParagrafo = document.createElement('p');
        novoParagrafo.textContent = 'Novo parágrafo criado dinamicamente!';
        container.appendChild(novoParagrafo);
      });
    </script>
  </body>
</html>

```

Passo 2: Removendo Elementos

```

<!DOCTYPE html>
<html>
  <head>
    <title>Removendo Elementos</title>
  </head>
  <body>
    <div id="container">

```

```

        <p id="paragrafoRemover">Parágrafo que será removido</p>
    >
    </div>
    <button id="remover">Remover Parágrafo</button>

    <script>
        let container = document.getElementById('container');
        let botaoRemover = document.getElementById('remover');
        let paragrafoRemover = document.getElementById('paragrafoRemover');

        botaoRemover.addEventListener('click', function() {
            if (paragrafoRemover) {
                container.removeChild(paragrafoRemover);
            }
        });
    </script>
</body>
</html>

```

Passo 3: Inserindo Elementos Antes de Outro

```

<!DOCTYPE html>
<html>
    <head>
        <title>Inserindo Elementos Antes</title>
    </head>
    <body>
        <div id="container">
            <p id="primeiro">Primeiro parágrafo</p>
            <p id="segundo">Segundo parágrafo</p>
        </div>
        <button id="inserirAntes">Inserir Parágrafo Antes</button>
    >
    <script>

```

```

let container = document.getElementById('container');
let botaoInserirAntes = document.getElementById('inserirAntes');
let segundo = document.getElementById('segundo');

botaoInserirAntes.addEventListener('click', function()
{
    let novoParagrafo = document.createElement('p');
    novoParagrafo.textContent = 'Parágrafo inserido antes
do segundo!';
    container.insertBefore(novoParagrafo, segundo);
});
</script>
</body>
</html>

```

Passo 4: Substituindo Elementos

```

<!DOCTYPE html>
<html>
    <head>
        <title>Substituindo Elementos</title>
    </head>
    <body>
        <div id="container">
            <p id="paragrafoOriginal">Parágrafo original</p>
        </div>
        <button id="substituir">Substituir Parágrafo</button>

        <script>
            let container = document.getElementById('container');
            let botaoSubstituir = document

            .getElementById('substituir');
            let paragrafoOriginal = document.getElementById('paragr
afoOriginal');

```

```
botaoSubstituir.addEventListener('click', function() {
  let novoParagrafo = document.createElement('p');
  novoParagrafo.textContent = 'Parágrafo substituto!';
  container.replaceChild(novoParagrafo, paragrafoOriginal);
});
</script>
</body>
</html>
```



Desafios para Praticar

1. Crie um arquivo chamado [criacao_remocao_elementos.html](#) e implemente um botão que crie novos elementos ao ser clicado. 
2. Implemente um botão que remova o último elemento filho de um container ao ser clicado. 
3. Crie uma função que insere um novo elemento antes de um elemento específico ao clicar em um botão. 
4. Implemente uma funcionalidade que substitua um elemento existente por um novo elemento ao clicar em um botão. 
5. Combine várias manipulações para criar uma interface dinâmica onde elementos são criados, removidos, inseridos e substituídos com base em ações do usuário. 

3.8 Projeto - Gerador de Senhas

Neste tópico, vamos aplicar tudo o que aprendemos até agora em um projeto prático. Vamos criar um gerador de senhas dinâmico e interativo, onde o usuário poderá definir o comprimento da senha e gerar uma nova senha clicando em um

botão. Este projeto vai consolidar seus conhecimentos sobre manipulação do DOM, eventos e interatividade. 

3.8.1 Estrutura HTML Inicial

Vamos começar criando a estrutura HTML básica para o nosso projeto.

Precisamos de um campo de entrada para o comprimento da senha, um botão para gerar a senha e um lugar para exibir a senha gerada.

HTML Inicial:

```
<!DOCTYPE html>
<html>
<head>
    <title>Gerador de Senhas</title>
    <style>
        body {
            font-family: Arial, sans-serif;
            display: flex;
            justify-content: center;
            align-items: center;
            height: 100vh;
            background-color: #f0f0f0;
        }
        .container {
            background: white;
            padding: 20px;
            border-radius: 8px;
            box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
            text-align: center;
        }
        input, button {
            padding: 10px;
            margin: 10px;
            border-radius: 5px;
            border: 1px solid #ccc;
        }
    </style>
</head>
<body>
    <div class="container">
        <input type="text" placeholder="Comprimento da senha" />
        <button>Gerar Senha</button>
        <span>Senha gerada: <span id="password"></span></span>
    </div>
</body>
</html>
```

```

        button {
            cursor: pointer;
            background-color: #007bff;
            color: white;
            border: none;
        }
        button:hover {
            background-color: #0056b3;
        }
        #senha {
            margin-top: 20px;
            font-weight: bold;
            font-size: 1.2em;
        }
    
```

</style>

</head>

<body>

<div class="container">

<h1>Gerador de Senhas</h1>

<input type="number" id="comprimento" placeholder="Comprimento da senha" min="1">

<button id="gerar">Gerar Senha</button>

<div id="senha"></div>

</div>

<script src="script.js"></script>

</body>

</html>

Explicação:

- **Estrutura HTML:** Criamos um layout básico com um campo de entrada, um botão e uma área para exibir a senha gerada.
- **CSS:** Adicionamos alguns estilos para melhorar a aparência da página.

3.8.2 Lógica do JavaScript

Agora vamos adicionar a lógica JavaScript para gerar a senha. Vamos capturar o comprimento da senha do campo de entrada, gerar uma senha aleatória com esse comprimento e exibir a senha gerada.

Script JavaScript:

```
document.getElementById('gerar').addEventListener('click', function() {
    let comprimento = document.getElementById('comprimento').value;
    let senha = gerarSenha(comprimento);
    document.getElementById('senha').textContent = senha;
});

function gerarSenha(comprimento) {
    let caracteres = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789!@#$%^&*()_+[]{}|;:,.<>?';
    let senha = '';
    for (let i = 0; i < comprimento; i++) {
        let indice = Math.floor(Math.random() * caracteres.length);
        senha += caracteres[indice];
    }
    return senha;
}
```

Explicação:

- **Capturando o evento de clique:** Adicionamos um Event Listener ao botão para capturar o evento de clique.
- **Função `gerarSenha`:** Função que gera uma senha aleatória com o comprimento especificado.

3.8.3 Testando o Projeto

Abra o arquivo HTML no navegador e teste a funcionalidade. Insira um comprimento para a senha e clique no botão "Gerar Senha". A senha gerada deve aparecer na área designada.



Desafios para Melhorar o Projeto

- 1. Adicione Opções para Incluir Caracteres Especiais:** Adicione checkboxes para permitir que o usuário escolha se deseja incluir letras maiúsculas, letras minúsculas, números e caracteres especiais na senha.
- 2. Copiar Senha para a Área de Transferência:** Adicione um botão para copiar a senha gerada para a área de transferência.
- 3. Validar Comprimento da Senha:** Adicione validação para garantir que o comprimento da senha seja um número válido e maior que zero.
- 4. Estilizar a Senha Gerada:** Adicione estilos adicionais para realçar a senha gerada, como mudar a cor ou o tamanho da fonte.
- 5. Adicionar Feedback Visual:** Adicione mensagens de feedback, como "Senha gerada com sucesso!" ou "Comprimento inválido!".

3.8.4 Código Completo com Melhorias

HTML com Opções Adicionais:

```
<!DOCTYPE html>
<html>
<head>
    <title>Gerador de Senhas</title>
    <style>
        body {
            font-family: Arial, sans-serif;
            display: flex;
            justify-content: center;
            align-items: center;
            height: 100vh;
            background-color: #f0f0f0;
```

```

        }
    .container {
        background: white;
        padding: 20px;
        border-radius: 8px;
        box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
        text-align: center;
    }
    input, button {
        padding: 10px;
        margin: 10px;
        border-radius: 5px;
        border: 1px solid #ccc;
    }
    button {
        cursor: pointer;
        background-color: #007bff;
        color: white;
        border: none;
    }
    button:hover {
        background-color: #0056b3;
    }
    #senha {
        margin-top: 20px;
        font-weight: bold;
        font-size: 1.2em;
    }

```

</style>

</head>

<body>

<div class="container">

<h1>Gerador de Senhas</h1>

<input type="number" id="comprimento" placeholder="Comprimento da senha" min="1">

<div>

```

        <input type="checkbox" id="maiusculas" checked> Letras Maiúsculas
        <input type="checkbox" id="minusculas" checked> Letras Minúsculas
        <input type="checkbox" id="numeros" checked> Números
        <input type="checkbox" id="especiais" checked> Caracteres Especiais
    </div>
    <button id="gerar">Gerar Senha</button>
    <div id="senha"></div>
    <button id="copiar">Copiar Senha</button>
    <div id="feedback"></div>
</div>
<script src="script.js"></script>
</body>
</html>

```

Script JavaScript com Melhorias:

```

document.getElementById('gerar').addEventListener('click', function() {
    let comprimento = document.getElementById('comprimento').value;
    let maiusculas = document.getElementById('maiusculas').checked;
    let minusculas = document.getElementById('minusculas').checked;
    let numeros = document.getElementById('numeros').checked;
    let especiais = document.getElementById('especiais').checked;

    if (comprimento <= 0) {
        document.getElementById('feedback').textContent = 'Co'
    }
})

```

```

        comprimento inválido!';
        return;
    }

    let senha = gerarSenha(comprimento, maiusculas, minusculas,
        numeros, especiais);
    document.getElementById('senha').textContent = senha;
    document.getElementById('feedback').textContent = 'Senha
gerada com sucesso!';
});

document.getElementById('copiar').addEventListener('click', f
unction() {
    let senha = document.getElementById('senha').textContent;
    navigator.clipboard.writeText(senha).then(function() {
        document.getElementById('feedback').textContent = 'Se
nha copiada para a área de transferência!';
    });
});

function gerarSenha(comprimento, maiusculas, minusculas, nume
ros, especiais) {
    let caracteres = '';
    if (maiusculas) caracteres += 'ABCDEFGHIJKLMNOPQRSTUVWXYZ';
    if (minusculas) caracteres += 'abcdefghijklmnopqrstuvwxyz';
    if (numeros) caracteres += '0123456789';
    if (especiais) caracteres += '!@#$%^&*()_+{}[];,.<>?';

    let senha = '';
    for (let i = 0; i < comprimento; i++) {
        let indice = Math.floor(Math.random() * caracteres.le
ngth);
        senha += caracteres[indice];
    }
}

```

```
    return senha;  
}
```

Explicação:

- **Checkboxes**: Permitem ao usuário selecionar os tipos de caracteres que desejam incluir na senha.
- **Validação do Comprimento**: Garante que o comprimento da senha seja válido.
- **Copiar Senha**: Adiciona a funcionalidade de copiar a senha gerada para a área de transferência.
- **Feedback Visual**: Exibe mensagens de feedback para o usuário.



Desafios para Praticar

1. **Personalizar as Mensagens de Feedback**: Torne as mensagens de feedback mais descritivas e amigáveis.
2. **Adicionar Opções de Comprimento Padrão**: Adicione botões para comprimentos de senha padrão (por exemplo, 8, 12, 16 caracteres).
3. **Salvar Preferências do Usuário**: Utilize o `localStorage` para salvar as preferências do usuário (por exemplo, tipos de caracteres selecionados).
4. **Melhorar a Interface do Usuário**: Adicione ícones e estilizações adicionais para melhorar a experiência do usuário.
5. **Gerar Senhas Seguras**: Adicione lógica para garantir que a senha gerada atenda a certos critérios de segurança (por exemplo, deve incluir pelo menos uma letra maiúscula, uma letra minúscula, um número e um caractere especial).

Conclusão do Capítulo

Neste capítulo, exploramos como manipular o DOM usando JavaScript. Aprendemos a criar, remover, inserir e substituir elementos, além de navegar entre elementos pai, filhos e irmãos. Praticamos essas habilidades através de exemplos

práticos e desafios que reforçaram seu entendimento sobre a manipulação do DOM.

Também desenvolvemos um projeto - um gerador de senhas dinâmico e interativo. Este projeto consolidou nossos conhecimentos ao aplicar a criação e manipulação de elementos, eventos e interatividade em JavaScript.

No próximo capítulo, vamos mergulhar no mundo do Git. Aprenderemos a usar Git para controle de versão, o que é fundamental para qualquer desenvolvedor que deseja colaborar em projetos e manter um histórico organizado do código.

Prepare-se para explorar as funcionalidades do Git e como ele pode melhorar seu fluxo de trabalho de desenvolvimento.

módulo 4

git

Controle de versão como um profissional! Aprenda Git
e mantenha seus projetos organizados e seguros. 

05 tópicos neste módulo

Módulo 4: Git

Neste capítulo, vamos explorar o Git, uma das ferramentas de controle de versão mais populares e poderosas disponíveis para desenvolvedores. 🔥

4.1 Introdução ao Git e Controle de Versão

Git é um sistema de controle de versão distribuído que permite aos desenvolvedores rastrear alterações no código-fonte durante o desenvolvimento de software. Ele facilita a colaboração entre equipes, mantém o histórico de alterações e ajuda a gerenciar diferentes versões do código. Vamos explorar os conceitos básicos do Git e entender como ele funciona. 📁🔧

4.1.1 O que é Controle de Versão?

Controle de versão é a prática de gerenciar e rastrear mudanças no código-fonte ao longo do tempo. Ele permite que várias pessoas trabalhem no mesmo projeto simultaneamente, sem sobreescriver o trabalho uma da outra. As principais vantagens do controle de versão incluem:

- **Rastreamento de Histórico:** Permite ver quem fez quais alterações e quando.
- **Recuperação de Versões Anteriores:** Facilita a reversão para versões anteriores do código se algo der errado.
- **Colaboração:** Facilita o trabalho em equipe, permitindo que várias pessoas contribuam para o mesmo projeto de forma organizada.

4.1.2 O que é Git?

Git é uma ferramenta de controle de versão distribuído, criada por Linus Torvalds em 2005. Ele permite que cada desenvolvedor tenha uma cópia completa do repositório de código em sua máquina local. Algumas características do Git incluem:

- **Distribuído:** Cada clone do repositório é um repositório completo, com todo o histórico de alterações.

- **Rápido e Eficiente:** Projetado para ser rápido, mesmo com repositórios grandes.
- **Seguro:** Usa criptografia para garantir a integridade do histórico de alterações.

4.1.4 Instalando o Git

Para começar a usar o Git, você precisa instalá-lo em sua máquina. Siga os passos abaixo para instalar o Git:

1. **Acesse o site oficial do Git:** [Git !\[\]\(59cf1ffae348a8d681360b1cddbe21d6_img.jpg\)](#)
2. **Baixe o instalador:** Disponível para Windows, macOS e Linux. 
3. **Instale o Git:** Siga as instruções do instalador.
4. **Verifique a instalação:** Abra o terminal ou prompt de comando e execute o comando:

```
git --version
```

Se tudo estiver correto, você verá a versão do Git instalada. 

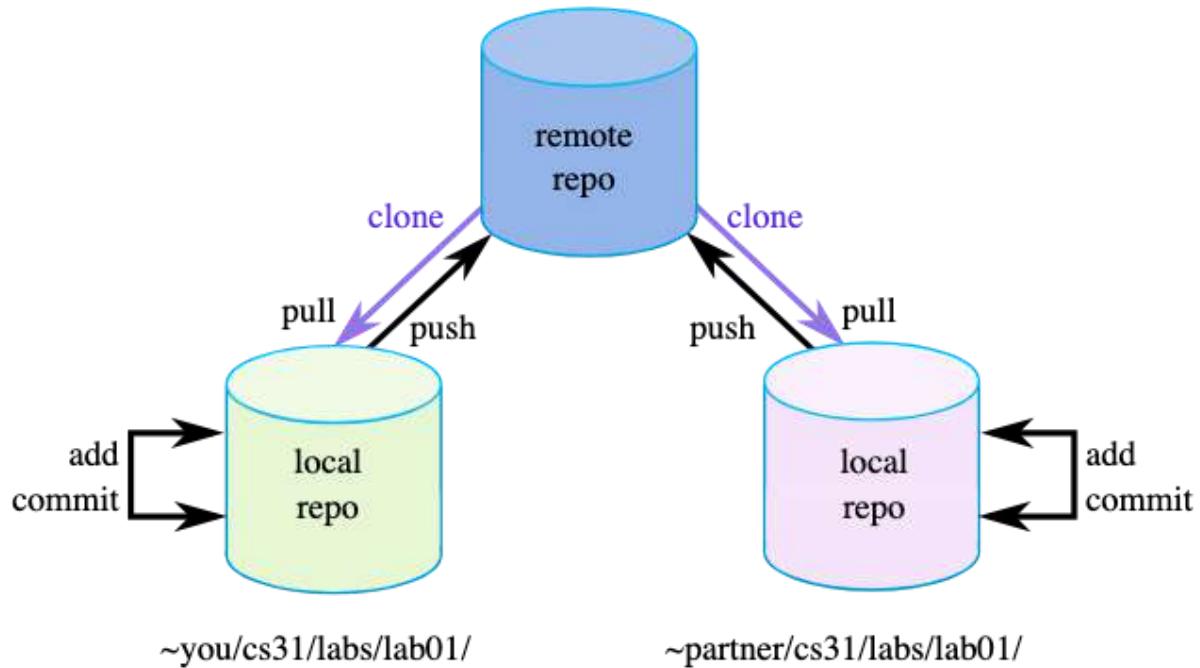
4.2 Conceitos Básicos do Git

4.2.1 Repositório (Local e Remoto)

Um repositório Git é onde todo o histórico do seu projeto é armazenado. Existem dois tipos de repositórios:

1. **Repositório Local:** É o repositório que está no seu computador. Ele contém todo o histórico do projeto e permite que você faça operações como commits, branches e merges sem precisar de uma conexão com a internet. Em um repositório local, você pode criar branches, fazer commits e visualizar o histórico das alterações.

2. Re却itório Remoto: É uma versão do seu re却itório que está hospedada em um servidor (como GitHub, GitLab ou Bitbucket). Ele permite a colaboração entre diferentes desenvolvedores. Você pode sincronizar seu re却itório local com o remoto através de operações como push, pull e fetch. O re却itório remoto é essencial para o trabalho colaborativo, permitindo que várias pessoas trabalhem no mesmo projeto simultaneamente.

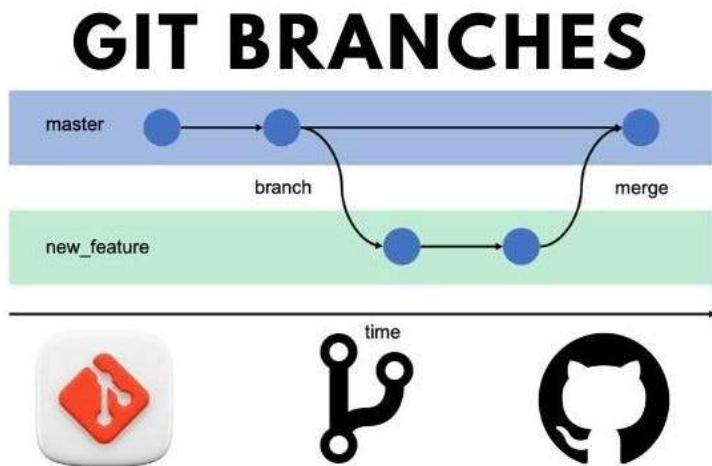


4.2.2 Branches (Ramificações)

Branches são uma das funcionalidades mais poderosas do Git. Elas permitem que você crie "linhas do tempo" alternativas no seu projeto. Cada branch é uma linha separada de desenvolvimento, e você pode alternar entre elas facilmente.

- **Branch Principal (main ou master):** É a linha principal de desenvolvimento. Geralmente, contém a versão mais estável do seu projeto. Todas as outras branches geralmente derivam dessa branch principal.
- **Branches de Funcionalidade:** São branches criadas para desenvolver novas funcionalidades ou corrigir bugs sem afetar a branch principal. Isso permite

que você trabalhe em novas features ou fixes de maneira isolada, sem interromper o desenvolvimento principal.



4.2.3 Commits

Um commit é uma "fotografia" do seu projeto em um momento específico no tempo. Ele contém uma mensagem que descreve as alterações feitas e um identificador único (SHA-1 Hash). Os commits permitem que você acompanhe a evolução do seu projeto e reverta para versões anteriores, se necessário. Cada commit aponta para um ou mais commits anteriores, formando um histórico linear ou ramificado das alterações.

- **Histórico de Commits:** O histórico de commits é uma lista sequencial de todos os commits feitos em um repositório. Ele permite que você veja a evolução do projeto e entenda quem fez quais mudanças e por quê.

4.2.4 SHA-1 Hash

Cada commit no Git é identificado por um hash SHA-1, um identificador único de 40 caracteres. Esse hash é gerado com base no conteúdo do commit, garantindo a integridade dos dados. Isso significa que, se o conteúdo de um commit for alterado, o hash também será alterado, tornando fácil detectar alterações não autorizadas.

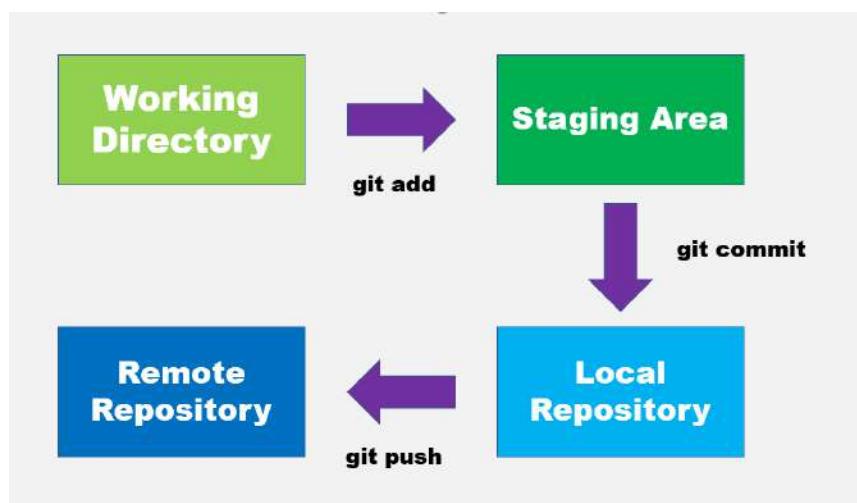
Exemplo de SHA-1 Hash:

```
e83c5163316f89bfbde7d9ab23ca2e25604af290
```

4.2.5 Árvore de Diretórios

O Git trabalha com três áreas principais:

1. **Working Directory (Diretório de Trabalho)**: Onde você faz alterações nos arquivos. É o estado atual do seu projeto no sistema de arquivos.
2. **Staging Area (Área de Preparação)**: Onde você adiciona os arquivos que deseja incluir no próximo commit. Os arquivos na staging area são preparados para serem commitados. Pense na staging area como um local onde você pode revisar e organizar suas mudanças antes de fazer um commit.
3. **Repository (Repositório)**: Onde os commits são armazenados. É o histórico completo do seu projeto. No repositório, todos os commits são armazenados de maneira segura e eficiente.



Esses conceitos formam a base do funcionamento do Git. Entender como cada componente interage com os outros é essencial para usar o Git de maneira eficaz e eficiente. No próximo tópico, vamos explorar os comandos básicos do Git para que você possa colocar esses conceitos em prática.

4.3 Configurando o Git

Para usar o Git de forma eficiente, é importante configurá-lo corretamente no seu ambiente de desenvolvimento. Vamos aprender como configurar o Git, personalizar algumas opções e entender os arquivos de configuração. 

4.3.1 Configuração Global

A configuração global do Git define opções que se aplicam a todos os repositórios no seu sistema. Você pode configurar seu nome de usuário e e-mail, que serão usados para identificar as alterações que você fizer.

- **Configurar Nome de Usuário e E-mail:**

```
git config --global user.name "Seu Nome"  
git config --global user.email "seu.email@example.com"
```

- **Verificar Configurações:**

Para verificar as configurações atuais, use o comando:

```
git config --global --list
```

4.3.2 Configuração Local

A configuração local do Git se aplica apenas ao repositório atual. Você pode substituir as configurações globais em um repositório específico.

- **Configurar Nome de Usuário e E-mail Localmente:**

```
cd seu_repositorio  
git config user.name "Nome Local"  
git config user.email "local.email@example.com"
```

- **Verificar Configurações Locais:**

Para verificar as configurações locais, use o comando:

```
git config --list
```

4.3.3 Arquivo de Configuração

O Git armazena suas configurações em arquivos de configuração. Existem três níveis de arquivos de configuração:

- **Sistema** (`/etc/gitconfig`): Configurações aplicáveis a todos os usuários no sistema.
- **Global** (`~/.gitconfig`): Configurações aplicáveis ao usuário atual.
- **Local** (`.git/config`): Configurações aplicáveis ao repositório atual.

Você pode editar esses arquivos diretamente usando um editor de texto ou usando comandos Git.

4.3.4 Alias de Comandos

Aliases são atalhos personalizados para comandos Git. Eles ajudam a economizar tempo e a tornar o uso do Git mais eficiente.

- **Criar Aliases:**

```
git config --global alias.st status  
git config --global alias.ci commit  
git config --global alias.co checkout  
git config --global alias.br branch
```

- **Usar Aliases:**

Agora você pode usar os aliases criados:

```
git st  
git ci -m "Mensagem de commit"  
git co nome_da_branch  
git br
```

4.4 Comandos Básicos do Git

Neste tópico, vamos explorar os comandos básicos do Git que são essenciais para gerenciar seu repositório e colaborar com outros desenvolvedores. Vamos aprender sobre `init`, `clone`, `status`, `add`, `commit`, `log`, entre outros. 🚀✨

4.4.1 `init`

O comando `init` é usado para criar um novo repositório Git. Ele inicializa um repositório vazio no diretório atual.

Usando o Comando `init`:

- **Iniciar um repositório Git:**

```
git init
```

Exemplo Prático:

```
# Navegar até o diretório do projeto  
cd meu_projeto  
  
# Inicializar um repositório Git  
git init
```

4.4.2 `clone`

O comando `clone` é usado para criar uma cópia local de um repositório remoto. Isso é útil para começar a trabalhar em um projeto existente.

Usando o Comando `clone`:

- **Clonar um repositório remoto:**

```
git clone URL_do_repositorio
```

Exemplo Prático:

```
# Clonar um repositório do GitHub  
git clone <https://github.com/usuario/repo.git>
```

4.4.3 `status`

O comando `status` mostra o estado atual do repositório, incluindo arquivos modificados, adicionados e não rastreados.

Usando o Comando `status`:

- **Verificar o estado do repositório:**

```
git status
```

Exemplo Prático:

```
# Verificar o estado do repositório  
git status
```

4.4.4 `add`

O comando `add` é usado para adicionar arquivos ao estágio (staging area). Isso prepara os arquivos para o próximo commit.

Usando o Comando `add`:

- **Adicionar um arquivo ao estágio:**

```
git add nome_do_arquivo
```

Para adicionar todos os arquivos modificados:

```
git add .
```

Exemplo Prático:

```
# Adicionar um arquivo ao estágio  
git add index.html  
  
# Adicionar todos os arquivos modificados ao estágio  
git add .
```

4.4.5 `commit`

O comando `commit` é usado para salvar as mudanças feitas no repositório local. Cada commit cria uma nova instantânea do seu código, permitindo que você rastreie o histórico de alterações.

Usando o Comando `commit`:

- **Fazer um commit:**

```
git commit -m "Mensagem descrevendo a alteração"
```

Exemplo Prático:

```
# Fazer um commit com uma mensagem descritiva  
git commit -m "Adicionar arquivo index.html"
```

4.4.6 `log`

O comando `log` mostra o histórico de commits do repositório, incluindo mensagens de commit, autores e datas.

Usando o Comando `log`:

- Verificar o histórico de commits:

```
git log
```

Exemplo Prático:

```
# Verificar o histórico de commits  
git log
```

4.4.7 `diff`

O comando `diff` mostra as diferenças entre arquivos ou commits. Isso é útil para revisar mudanças antes de fazer um commit.

Usando o Comando `diff`:

- Verificar diferenças entre o estado atual e o último commit:

```
git diff
```

Exemplo Prático:

```
# Verificar diferenças entre arquivos modificados e o último commit  
git diff
```

4.4.8 **branch**

O comando `branch` é usado para criar, listar e gerenciar branches. As branches permitem que você trabalhe em diferentes linhas de desenvolvimento simultaneamente.

Usando o Comando `branch`:

- **Criar uma nova branch:**

```
git branch nome_da_branch
```

- **Listar todas as branches:**

```
git branch
```

Exemplo Prático:

```
# Criar uma nova branch  
git branch nova_funcionalidade  
  
# Listar todas as branches  
git branch
```

4.4.9 **checkout**

O comando `checkout` é usado para mudar para outra branch ou restaurar arquivos de um commit específico.

Usando o Comando `checkout`:

- **Mudar para outra branch:**

```
git checkout nome_da_branch
```

Exemplo Prático:

```
# Mudar para a branch "nova_funcionalidade"  
git checkout nova_funcionalidade
```

4.4.10 `merge`

O comando `merge` é usado para combinar mudanças de diferentes branches em uma única branch.

Usando o Comando `merge`:

- **Mesclar uma branch em outra:**

```
git merge nome_da_branch
```

Exemplo Prático:

```
# Mesclar a branch "nova_funcionalidade" na branch atual  
git merge nova_funcionalidade
```



Exercícios Práticos

Vamos praticar os comandos básicos do Git. Execute as seguintes operações:

1. **Inic平ize um novo repositório Git e faça alguns commits.**
2. **Clone um repositório existente e faça alguns commits e pushes.**
3. **Crie uma nova branch e mescle-a na branch principal.**
4. **Use os comandos `status`, `diff` e `log` para revisar seu trabalho.**

Exemplo de Código:

```
# Inicializar um repositório Git
git init

# Criar um arquivo index.html
echo "<!DOCTYPE html><html><head><title>Meu Projeto</title></
head><body><h1>Olá, Mundo!</h1></body></html>" > index.html

# Adicionar o arquivo ao estágio
git add index.html

# Fazer um commit
git commit -m "Adicionar arquivo index.html"

# Verificar o estado do repositório
git status

# Verificar diferenças entre arquivos modificados e o último
commit
git diff

# Verificar o histórico de commits
git log

# Criar uma nova branch
git branch nova_funcionalidade

# Mudar para a nova branch
git checkout nova_funcionalidade

# Fazer algumas alterações e commits na nova branch
echo "<p>Bem-vindo ao meu projeto!</p>" >> index.html
git add index.html
git commit -m "Adicionar parágrafo de boas-vindas ao index.ht
ml"
```

```
# Voltar para a branch principal
git checkout main

# Mesclar a branch "nova_funcionalidade" na branch principal
git merge nova_funcionalidade

# Clonar um repositório existente
git clone <https://github.com/usuario/repo.git>

# Navegar até o repositório clonado
cd repo

# Fazer algumas alterações e commits
echo "<p>Este é um novo parágrafo.</p>" >> index.html
git add index.html
git commit -m "Adicionar novo parágrafo ao index.html"

# Enviar mudanças para o repositório remoto
git push origin main
```

4.5 Branching e Merging

Trabalhar com branches (ramos) é uma das funcionalidades mais poderosas do Git. Permite que você crie linhas separadas de desenvolvimento para trabalhar em diferentes funcionalidades ou correções de bugs de forma isolada. Depois, você pode mesclar (merge) essas branches de volta na branch principal. Vamos explorar como criar e gerenciar branches e como realizar a mesclagem. 🌱✨

4.5.1 Criando Branches

Uma branch permite que você tenha uma cópia do seu repositório para fazer alterações sem afetar o trabalho na branch principal.

Comando para criar uma nova branch:

```
git branch nome_da_branch
```

Exemplo Prático:

```
# Criar uma nova branch chamada "nova_funcionalidade"  
git branch nova_funcionalidade
```

4.5.2 Mudando para Outra Branch

Para trabalhar em uma branch diferente, você precisa mudar para ela usando o comando `checkout`.

Comando para mudar para outra branch:

```
git checkout nome_da_branch
```

Exemplo Prático:

```
# Mudar para a branch "nova_funcionalidade"  
git checkout nova_funcionalidade
```

4.5.3 Criar e Mudar para uma Nova Branch

Você pode combinar a criação de uma branch e a mudança para ela em um único comando usando `checkout` com a opção `-b`.

Comando para criar e mudar para uma nova branch:

```
git checkout -b nome_da_branch
```

Exemplo Prático:

```
# Criar e mudar para a branch "nova_funcionalidade"  
git checkout -b nova_funcionalidade
```

4.5.4 Listando Branches

Você pode listar todas as branches no seu repositório usando o comando `branch`.

Comando para listar branches:

```
git branch
```

Exemplo Prático:

```
# Listar todas as branches  
git branch
```

4.5.5 Mesclando Branches

Depois de terminar de trabalhar em uma branch, você pode mesclá-la de volta na branch principal ou em outra branch. Use o comando `merge` para combinar as mudanças.

Comando para mesclar branches:

```
git merge nome_da_branch
```

Exemplo Prático:

```
# Mudar para a branch principal  
git checkout main
```

```
# Mesclar a branch "nova_funcionalidade" na branch principal  
git merge nova_funcionalidade
```

4.5.6 Resolução de Conflitos

Às vezes, ao mesclar branches, você pode encontrar conflitos se houver mudanças conflitantes nos mesmos arquivos. Git marcará esses conflitos e permitirá que você os resolva manualmente.

Passos para resolver conflitos:

- 1. Abra os arquivos conflitantes e edite para resolver os conflitos.**
- 2. Marque os conflitos como resolvidos:**

```
git add nome_do_arquivo
```

- 3. Complete a mesclagem:**

```
git commit -m "Resolver conflitos de mesclagem"
```

Exemplo Prático:

```
# Mesclar a branch "nova_funcionalidade" e resolver conflitos  
git checkout main  
git merge nova_funcionalidade  
  
# Se houver conflitos, edite os arquivos conflitantes  
# Marque os conflitos como resolvidos  
git add nome_do_arquivo  
  
# Complete a mesclagem  
git commit -m "Resolver conflitos de mesclagem"
```



Exercícios Práticos

Vamos praticar o uso de branches e mesclagem no Git. Execute as seguintes operações:

- 1. Crie uma nova branch e faça alguns commits.**
- 2. Mude para a branch principal e mescle a nova branch nela.**
- 3. Resolva quaisquer conflitos que possam surgir durante a mesclagem.**
- 4. Liste todas as branches no repositório.**

Exemplo de Código:

```
# Criar e mudar para uma nova branch
git checkout -b nova_funcionalidade

# Fazer algumas alterações e commits na nova branch
echo "<p>Nova funcionalidade adicionada.</p>" >> index.html
git add index.html
git commit -m "Adicionar nova funcionalidade ao index.html"

# Mudar para a branch principal
git checkout main

# Mesclar a branch "nova_funcionalidade" na branch principal
git merge nova_funcionalidade

# Se houver conflitos, edite os arquivos conflitantes
# Marque os conflitos como resolvidos
git add nome_do_arquivo

# Complete a mesclagem
git commit -m "Resolver conflitos de mesclagem"

# Listar todas as branches
git branch
```

4.6 Trabalhando com Repositórios Remotos

Trabalhar com repositórios remotos é uma parte essencial do uso do Git, especialmente quando colaboramos com outros desenvolvedores. Repositórios remotos são versões do seu projeto que estão hospedadas na internet ou em uma rede local. Vamos explorar como adicionar repositórios remotos, enviar mudanças, obter atualizações e colaborar de forma eficaz.  

4.6.1 Adicionando Repositórios Remotos

Para trabalhar com um repositório remoto, primeiro você precisa adicionar a URL do repositório remoto ao seu projeto local.

Comando para adicionar um repositório remoto:

```
git remote add nome_remoto URL_do_repositorio
```

Exemplo Prático:

```
# Adicionar um repositório remoto chamado "origin"
git remote add origin <https://github.com/usuario/repo.git>
```

4.6.2 Verificando Repositórios Remotos

Você pode listar todos os repositórios remotos associados ao seu projeto local usando o comando `remote -v`.

Comando para listar repositórios remotos:

```
git remote -v
```

Exemplo Prático:

```
# Listar todos os repositórios remotos  
git remote -v
```

4.6.3 Enviando Mudanças para o Re却itório Remoto (**push**)

O comando **push** é usado para enviar suas mudanças locais para o re却itório remoto. Isso permite que outros desenvolvedores vejam suas alterações.

Comando para enviar mudanças:

```
git push nome_remoto nome_da_branch
```

Exemplo Prático:

```
# Enviar mudanças para a branch "main" no re却itório remoto  
"origin"  
git push origin main
```

4.6.4 Obtendo Atualizações do Re却itório Remoto (**pull**)

O comando **pull** é usado para atualizar seu re却itório local com as mudanças mais recentes do re却itório remoto. Isso garante que você esteja trabalhando com a versão mais atual do código.

Comando para obter atualizações:

```
git pull nome_remoto nome_da_branch
```

Exemplo Prático:

```
# Atualizar a branch "main" com as mudanças do re却itório re  
moto "origin"  
git pull origin main
```

4.6.5 Clonando um Repositório Remoto (`clone`)

O comando `clone` é usado para criar uma cópia local de um repositório remoto. Isso é útil para começar a trabalhar em um projeto existente.

Comando para clonar um repositório remoto:

```
git clone URL_do_repositorio
```

Exemplo Prático:

```
# Clonar um repositório do GitHub  
git clone <https://github.com/usuario/repo.git>
```

4.6.6 Visualizando as Branches Remotas

Você pode listar todas as branches remotas usando o comando `branch -r`.

Comando para listar branches remotas:

```
git branch -r
```

Exemplo Prático:

```
# Listar todas as branches remotas  
git branch -r
```

4.6.7 Colaboração com Forks e Pull Requests

Em plataformas como GitHub, GitLab e Bitbucket, você pode colaborar com outros desenvolvedores usando forks e pull requests. Um fork é uma cópia do repositório que permite que você faça alterações livremente. Um pull request é uma solicitação para mesclar suas mudanças no repositório original.

Passos para colaborar com forks e pull requests:

- 1. Fork o repositório original.**
- 2. Clone o fork para sua máquina local.**
- 3. Crie uma branch para suas alterações.**
- 4. Faça commits e pushes para o seu fork.**
- 5. Abra um pull request para o repositório original.**



Exercícios Práticos

Vamos praticar o uso de repositórios remotos no Git. Execute as seguintes operações:

- 1. Adicione um repositório remoto ao seu projeto local.**
- 2. Envie suas mudanças locais para o repositório remoto usando `push`.**
- 3. Obtenha atualizações do repositório remoto usando `pull`.**
- 4. Clone um repositório remoto e faça algumas alterações.**
- 5. Liste todas as branches remotas do repositório clonado.**

Exemplo de Código:

```
# Adicionar um repositório remoto chamado "origin"
git remote add origin <https://github.com/usuario/repo.git>

# Enviar mudanças para a branch "main" no repositório remoto
# "origin"
git push origin main

# Atualizar a branch "main" com as mudanças do repositório remoto "origin"
git pull origin main

# Clonar um repositório do GitHub
git clone <https://github.com/usuario/repo.git>
```

```
# Navegar até o repositório clonado
cd repo

# Fazer algumas alterações e commits
echo "<p>Este é um novo parágrafo.</p>" >> index.html
git add index.html
git commit -m "Adicionar novo parágrafo ao index.html"

# Enviar mudanças para o repositório remoto
git push origin main

# Listar todas as branches remotas
git branch -r
```

4.7 Manipulação e Reescrita de Histórico

Manipular e reescrever o histórico de commits no Git pode ser uma tarefa delicada, mas é extremamente útil para manter um histórico de commits limpo e organizado. Vamos explorar como usar comandos como `rebase`, `reset` e `reflog` para manipular e reescrever o histórico de commits. 

4.7.1 Rebase

O comando `rebase` é usado para aplicar commits de uma branch em outra base. É uma alternativa ao `merge` que pode resultar em um histórico de commits mais linear.

Usando o Comando `rebase`:

- **Mudar para a branch que você quer rebasear:**

```
git checkout minha_branch
```

- **Rebasear a branch atual na base de outra branch:**

```
git rebase nome_da_branch
```

Exemplo Prático:

```
# Mudar para a branch "minha_branch"  
git checkout minha_branch
```

```
# Rebasear "minha_branch" na base da branch "main"  
git rebase main
```

4.7.2 Interativo (Interactive) Rebase

O rebase interativo (`rebase -i`) permite que você edite commits individuais, reordene commits, combine (squash) commits e muito mais.

Usando o Comando `rebase -i`:

- **Iniciar um rebase interativo:**

```
git rebase -i HEAD~n
```

(Substitua `n` pelo número de commits que você deseja editar)

- **Editar o rebase interativo:**

```
pick f1e2d3a Mensagem do commit 1  
pick a2b3c4d Mensagem do commit 2  
pick c3d4e5f Mensagem do commit 3
```

Você pode mudar `pick` para `edit`, `squash`, etc.

Exemplo Prático:

```
# Iniciar um rebase interativo para os últimos 3 commits  
git rebase -i HEAD~3
```

4.7.3 Reset

O comando `reset` é usado para desfazer commits. Existem três modos principais de reset: `--soft`, `--mixed` e `--hard`.

Usando o Comando `reset`:

- **Modo Soft (`--soft`)**: Mantém as mudanças no índice (staging area).

```
git reset --soft HEAD~1
```

- **Modo Mixed (`--mixed`)**: Mantém as mudanças no diretório de trabalho.

```
git reset --mixed HEAD~1
```

- **Modo Hard (`--hard`)**: Remove todas as mudanças do índice e do diretório de trabalho.

```
git reset --hard HEAD~1
```

Exemplo Prático:

```
# Desfazer o último commit e manter as mudanças no índice  
git reset --soft HEAD~1  
  
# Desfazer o último commit e manter as mudanças no diretório  
# de trabalho  
git reset --mixed HEAD~1
```

```
# Desfazer o último commit e remover todas as mudanças  
git reset --hard HEAD~1
```

4.7.4 Reflog

O comando `reflog` é usado para visualizar o histórico de referências do Git. Ele permite que você veja ações anteriores e recupere commits perdidos.

Usando o Comando `reflog`:

- **Ver o histórico de referências:**

```
git reflog
```

- **Recuperar um commit perdido:**

```
git reset --hard SHA_do_commit
```

Exemplo Prático:

```
# Ver o histórico de referências  
git reflog  
  
# Recuperar um commit perdido usando o SHA  
git reset --hard f1e2d3a
```



Exercícios Práticos

Vamos praticar a manipulação e reescrita de histórico no Git. Execute as seguintes operações:

1. Use `rebase` para criar um histórico de commits mais linear.
2. Faça um rebase interativo para editar mensagens de commits e combinar commits.

3. Use `reset` para desfazer commits em diferentes modos (`--soft`, `--mixed`, `--hard`).
4. Use `reflog` para visualizar o histórico de referências e recuperar um commit perdido.

Exemplo de Código:

```
# Mudar para a branch "minha_branch"
git checkout minha_branch

# Rebasear "minha_branch" na base da branch "main"
git rebase main

# Iniciar um rebase interativo para os últimos 3 commits
git rebase -i HEAD~3

# Desfazer o último commit e manter as mudanças no índice
git reset --soft HEAD~1

# Desfazer o último commit e manter as mudanças no diretório
# de trabalho
git reset --mixed HEAD~1

# Desfazer o último commit e remover todas as mudanças
git reset --hard HEAD~1

# Ver o histórico de referências
git reflog

# Recuperar um commit perdido usando o SHA
git reset --hard f1e2d3a
```

4.8 Boas Práticas e Convenções

Seguir boas práticas e convenções ao usar Git é essencial para manter um repositório organizado, legível e fácil de gerenciar. Neste tópico, vamos explorar algumas das melhores práticas para trabalhar com Git, incluindo a escrita de mensagens de commit claras, uso de branches, e convenções de nomenclatura.



4.8.1 Mensagens de Commit Claras

Escrever mensagens de commit claras e descriptivas é crucial para manter um histórico de commits útil. Mensagens de commit devem responder às perguntas "O que foi alterado?" e "Por que foi alterado?".

Boas Práticas para Mensagens de Commit:

- 1. Seja Descritivo:** Explique o que foi alterado e por que.
- 2. Use o Tempo Verbal Imperativo:** Por exemplo, "Adicionar nova funcionalidade" ao invés de "Adicionada nova funcionalidade".
- 3. Mantenha a Linha de Assunto Curta:** A linha de assunto deve ter no máximo 50 caracteres.
- 4. Adicione um Corpo se Necessário:** Se a mudança for complexa, adicione um corpo à mensagem de commit.

Exemplo de Mensagem de Commit:

```
Adicionar funcionalidade de login de usuário
```

```
Implementar autenticação baseada em JWT para gerenciar sessões de usuário.
```

```
Adicionar rotas para login e logout. Atualizar a documentação com exemplos de uso.
```

4.8.2 Uso de Branches

Trabalhar com branches é uma prática recomendada para isolar novas funcionalidades, correções de bugs e experimentos. Isso evita conflitos e mantém a branch principal (`main` ou `master`) estável.

Boas Práticas para Uso de Branches:

- 1. Use Nomes Descritivos:** O nome da branch deve refletir o propósito da branch.
- 2. Mantenha Branches Curto Vidas:** Mescle as branches assim que a funcionalidade ou correção estiver pronta.
- 3. Use Branches de Feature e Hotfix:** Crie branches específicas para novas funcionalidades (`feature/nome_da_funcionalidade`) e correções de bugs (`hotfix/nome_da_correcao`).

Exemplo de Nomes de Branches:

```
feature/adicionar-login  
hotfix/corrigir-bug-no-formulario
```

4.8.3 Convenções de Nomenclatura

Seguir convenções de nomenclatura para branches, commits e arquivos ajuda a manter o repositório organizado e legível.

Boas Práticas para Nomenclatura:

- Branches:** Use nomes descritivos e separados por hifens (`-`).
- Commits:** Mantenha as mensagens de commit curtas e descritivas.
- Arquivos:** Use nomes de arquivos que descrevam seu conteúdo e evite caracteres especiais.

Exemplo de Nomenclatura:

```
branches: feature/adicionar-login, hotfix/corrigir-bug-no-formulario
```

```
commits: Adicionar funcionalidade de login de usuário  
arquivos: login.component.js, user.service.js
```

4.8.4 Rebase vs Merge

Decidir entre rebase e merge depende do fluxo de trabalho da equipe. Ambos têm suas vantagens e desvantagens.

Merge:

- **Pro:** Mantém o histórico completo de commits.
- **Contra:** Pode resultar em um histórico de commits mais complexo.

Rebase:

- **Pro:** Cria um histórico de commits mais linear.
- **Contra:** Pode ser arriscado se não for usado corretamente.

Exemplo de Uso de Rebase e Merge:

```
# Merge  
git checkout main  
git merge feature/adicionar-login  
  
# Rebase  
git checkout feature/adicionar-login  
git rebase main
```

4.8.5 Revisão de Código

Revisão de código é uma prática importante para garantir a qualidade do código. Use pull requests para solicitar revisões antes de mesclar mudanças na branch principal.

Boas Práticas para Revisão de Código:

1. **Use Pull Requests:** Solicite revisões de outros desenvolvedores.

- 2. Forneça Feedback Construtivo:** Seja claro e específico ao fornecer feedback.
- 3. Revise Regularmente:** Faça revisões de código regularmente para manter a qualidade.



Exercícios Práticos

Vamos praticar as boas práticas e convenções no Git. Execute as seguintes operações:

- 1. Crie uma branch descritiva para uma nova funcionalidade.**
- 2. Faça commits com mensagens claras e descritivas.**
- 3. Use rebase para criar um histórico de commits linear.**
- 4. Solicite uma revisão de código usando um pull request.**

Exemplo de Código:

```
# Criar uma branch descritiva para uma nova funcionalidade
git checkout -b feature/adicionar-login

# Fazer commits com mensagens claras e descritivas
echo "<p>Formulário de login.</p>" >> login.html
git add login.html
git commit -m "Adicionar formulário de login"

# Fazer mais alterações e commits
echo "console.log('Login');" >> login.js
git add login.js
git commit -m "Adicionar lógica de login"

# Rebase para criar um histórico de commits linear
git checkout main
git pull origin main
git checkout feature/adicionar-login
git rebase main

# Mesclar a branch de funcionalidade na branch principal
```

```
git checkout main
git merge feature/adicionar-login

# Solicitar uma revisão de código usando um pull request
# (Este passo é realizado na interface do GitHub, GitLab, etc.)
```

Conclusão do Capítulo

Neste capítulo, exploramos os conceitos fundamentais e avançados do Git, uma ferramenta essencial para o controle de versão e colaboração em projetos de desenvolvimento. Aprendemos a criar e gerenciar branches, mesclar mudanças, resolver conflitos e trabalhar com repositórios remotos. Também discutimos boas práticas para manter um histórico de commits limpo e organizado, além de realizar a reescrita e manipulação do histórico de commits.

Os principais tópicos abordados incluíram:

- **Criação e Mudança de Branches:** Como isolar novas funcionalidades e correções de bugs.
- **Mesclagem de Branches:** Como integrar mudanças de diferentes branches.
- **Resolução de Conflitos:** Como lidar com conflitos durante a mesclagem.
- **Trabalho com Repositórios Remotos:** Adição, envio e obtenção de atualizações de repositórios remotos.
- **Manipulação de Histórico:** Uso de comandos como `rebase` e `reset` para manter um histórico de commits limpo.
- **Boas Práticas e Convenções:** Escrita de mensagens de commit claras, uso de branches e convenções de nomenclatura.

No próximo capítulo, vamos mergulhar no mundo do Bootstrap. Você aprenderá a usar este poderoso framework front-end para criar sites responsivos e estilizados de maneira rápida e eficiente. Prepare-se para explorar componentes, grid system, e muito mais!

Continue praticando os conceitos aprendidos e até o próximo capítulo!

módulo 5

bootstrap

Crie páginas web responsivas e atraentes
rapidamente utilizando o framework Bootstrap. 

06 tópicos neste módulo

Módulo 5: Bootstrap

5.1 Introdução ao Bootstrap

5.1.1 O que é Bootstrap?

Bootstrap é um dos frameworks front-end mais populares do mundo, utilizado para desenvolver interfaces web responsivas e modernas com facilidade. Criado pelo Twitter, o Bootstrap oferece uma ampla variedade de componentes prontos, como botões, formulários, modais, e um poderoso sistema de grid para criar layouts responsivos. Neste capítulo, vamos explorar como configurar o Bootstrap em seu projeto, entender seu sistema de grid, e usar os componentes básicos para construir páginas web elegantes e funcionais. 

Principais Características do Bootstrap:

- **Responsividade:** Sites criados com Bootstrap são automaticamente responsivos, ajustando-se a diferentes tamanhos de tela, desde dispositivos móveis até desktops.
- **Componentes Prontos:** Bootstrap inclui uma ampla gama de componentes prontos, como botões, formulários, navegações, e muito mais.
- **Customização:** Embora venha com estilos padrão, Bootstrap é altamente customizável, permitindo que você ajuste o visual e o comportamento dos componentes conforme suas necessidades.

5.2 Configuração do Bootstrap

Neste tópico, vamos nos aprofundar na configuração do Bootstrap em seu projeto. Veremos como incluir Bootstrap via CDN e como baixar e hospedar os arquivos localmente. Também aprenderemos a configurar os arquivos necessários para começar a usar o Bootstrap de maneira eficiente. 

5.2.1 Usando Bootstrap via CDN

A maneira mais rápida de começar a usar o Bootstrap é incluí-lo via CDN (Content Delivery Network). Isso significa que você está carregando os arquivos CSS e JavaScript diretamente de servidores externos confiáveis, o que pode melhorar o desempenho do seu site.

Passos para incluir Bootstrap via CDN:

- Adicione o link CSS no `<head>` do seu HTML:

```
<link rel="stylesheet" href="">
```

- Adicione os scripts JavaScript no final do `<body>` do seu HTML:

```
<script src=""></script>
<script src=""></script>
<script src=""></script>
```

Exemplo Completo de Configuração via CDN:

```
<!DOCTYPE html>
<html lang="pt-br">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Projeto com Bootstrap</title>
    <link rel="stylesheet" href="">
  </head>
```

```

<body>
  <div class="container">
    <header class="my-4">
      <h1 class="text-center">Bem-vindo ao Meu Projeto</h1>
    <1>
    </header>
    <main>
      <p class="lead">Esta é uma página de exemplo usando Bootstrap.</p>
      <button class="btn btn-primary">Clique Aqui</button>
    <>
    </main>
    <footer class="mt-4 text-center">
      <p>&copy; 2023 Meu Projeto</p>
    </footer>
  </div>
  <script src="https://code.jquery.com/jquery-3.5.1.slim.min.js"></script>
  <script src="https://cdn.jsdelivr.net/npm/@popperjs/core@2.9.2/dist/umd/popper.min.js"></script>
  <script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.5.2/js/bootstrap.min.js"></script>
</body>
</html>

```

5.2.2 Baixando Bootstrap e Hospedando Localmente

Se preferir hospedar os arquivos do Bootstrap localmente, você pode baixar os arquivos e incluí-los no seu projeto. Isso pode ser útil se você precisar trabalhar offline ou deseja customizar os arquivos do Bootstrap.

Passos para baixar e hospedar Bootstrap localmente:

- 1. Baixe os arquivos do Bootstrap:**

- Acesse o site oficial do Bootstrap: [Bootstrap Download](#)

- Baixe a versão mais recente do Bootstrap.

2. Extraia os arquivos:

- Extraia os arquivos baixados para uma pasta no seu projeto.

3. Inclua os arquivos CSS e JavaScript no seu HTML:

Adicione o link CSS no `<head>`:

```
<link rel="stylesheet" href="caminho/para/bootstrap.min.css">
```

Adicione os scripts JavaScript no final do `<body>`:

```
<script src="caminho/para/jquery.min.js"></script>
<script src="caminho/para/popper.min.js"></script>
<script src="caminho/para/bootstrap.min.js"></script>
```

Exemplo Completo de Configuração Local:

```
<!DOCTYPE html>
<html lang="pt-br">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Projeto com Bootstrap</title>
    <link rel="stylesheet" href="css/bootstrap.min.css">
  </head>
  <body>
    <div class="container">
      <header class="my-4">
        <h1 class="text-center">Bem-vindo ao Meu Projeto<h1>
      </header>
      <main>
        <p class="lead">Esta é uma página de exemplo usando Bootstrap.</p>
      </main>
    </div>
  </body>
</html>
```

```
<button class="btn btn-primary">Clique Aqui</button>
</main>
<footer class="mt-4 text-center">
  <p>&copy; 2023 Meu Projeto</p>
</footer>
</div>
<script src="js/jquery.min.js"></script>
<script src="js/popper.min.js"></script>
<script src="js/bootstrap.min.js"></script>
</body>
</html>
```

5.3 Sistema de Grid do Bootstrap

O sistema de grid do Bootstrap é uma das suas funcionalidades mais poderosas, permitindo criar layouts responsivos e flexíveis com facilidade. Vamos explorar como o sistema de grid funciona e como usá-lo para criar layouts complexos que se adaptam a diferentes tamanhos de tela. 

5.3.1 Introdução ao Sistema de Grid

O sistema de grid do Bootstrap é baseado em uma estrutura de 12 colunas que se ajustam automaticamente conforme o tamanho da tela muda. Utilizando classes predefinidas, você pode definir como os elementos se comportam em diferentes resoluções, desde dispositivos móveis até desktops.

COL-3		COL-3		COL-3			COL-3		
COL-4			COL-4			COL-4			
COL-6						COL-6			
COL-2		COL-2		COL-2		COL-2		COL-2	
COL-1									

Estrutura Básica:

- Container:** O contêiner centraliza o conteúdo e fornece um padding horizontal.
- Row:** As linhas (`row`) são usadas para agrupar colunas.
- Column:** As colunas (`col`) definem o layout dentro das linhas.

Exemplo de Estrutura Básica:

```
<div class="container">
  <div class="row">
    <div class="col">
      Coluna 1
    </div>
    <div class="col">
      Coluna 2
    </div>
    <div class="col">
      Coluna 3
    </div>
  </div>
</div>
```

5.3.2 Containers

Os containers são elementos fundamentais no sistema de grid do Bootstrap. Eles são usados para centralizar e dar padding ao conteúdo da página. Existem dois

tipos de containers:

1. `.container` : Tem um width fixo que se ajusta conforme o viewport aumenta.
2. `.container-fluid` : Ocupa 100% da largura do viewport.

Exemplo de Uso de Containers:

```
<!-- Container fixo -->
<div class="container">
    <p>Conteúdo centralizado e com padding horizontal.</p>
</div>

<!-- Container fluido -->
<div class="container-fluid">
    <p>Conteúdo que ocupa 100% da largura do viewport.</p>
</div>
```

5.3.3 Linhas (Rows) e Colunas (Columns)

As linhas (`row`) são usadas para criar grupos horizontais de colunas. Dentro das linhas, as colunas (`col`) definem a estrutura do layout.

Exemplo de Linhas e Colunas:

```
<div class="container">
    <div class="row">
        <div class="col-sm">
            Coluna 1
        </div>
        <div class="col-sm">
            Coluna 2
        </div>
        <div class="col-sm">
            Coluna 3
        </div>
```

```
</div>  
</div>
```

5.3.4 Colunas Responsivas

O Bootstrap permite que você defina diferentes tamanhos de colunas para diferentes tamanhos de tela usando classes específicas como `.col-sm`, `.col-md`, `.col-lg`, e `.col-xl`.

Exemplo de Colunas Responsivas:

```
<div class="container">  
  <div class="row">  
    <div class="col-12 col-sm-6 col-md-4 col-lg-3">  
      Coluna Responsiva  
    </div>  
    <div class="col-12 col-sm-6 col-md-4 col-lg-3">  
      Coluna Responsiva  
    </div>  
    <div class="col-12 col-sm-6 col-md-4 col-lg-3">  
      Coluna Responsiva  
    </div>  
    <div class="col-12 col-sm-6 col-md-4 col-lg-3">  
      Coluna Responsiva  
    </div>  
  </div>  
</div>
```

5.3.5 Colunas Aninhadas

Você pode aninhar colunas dentro de outras colunas para criar layouts mais complexos.

Exemplo de Colunas Aninhadas:

```
<div class="container">
  <div class="row">
    <div class="col-md-8">
      Coluna Principal
      <div class="row">
        <div class="col-6">
          Subcoluna 1
        </div>
        <div class="col-6">
          Subcoluna 2
        </div>
      </div>
      <div class="col-md-4">
        Coluna Secundária
      </div>
    </div>
  </div>
</div>
```

5.3.6 Offset, Ordem e Alinhamento

O Bootstrap oferece classes para controlar o offset (deslocamento), a ordem e o alinhamento das colunas.

Exemplo de Offset:

```
<div class="container">
  <div class="row">
    <div class="col-md-4 offset-md-4">
      Coluna Centralizada com Offset
    </div>
  </div>
</div>
```

Exemplo de Ordem:

```
<div class="container">
  <div class="row">
    <div class="col-md-4 order-md-2">
      Coluna 1 (Ordem 2)
    </div>
    <div class="col-md-4 order-md-1">
      Coluna 2 (Ordem 1)
    </div>
  </div>
</div>
```

Exemplo de Alinhamento:

```
<div class="container">
  <div class="row align-items-center">
    <div class="col-md-4">
      Alinhado ao Centro
    </div>
    <div class="col-md-4">
      Alinhado ao Centro
    </div>
  </div>
</div>
```

5.3.7 Documentação

[Documentação oficial do grid system](#)



Exercícios Práticos

Vamos praticar o uso do sistema de grid do Bootstrap. Execute as seguintes operações:

1. Crie um layout básico usando containers, linhas e colunas.
 2. Crie um layout responsivo que se adapte a diferentes tamanhos de tela.
 3. Experimente com colunas aninhadas para criar layouts mais complexos.
 4. Use offset, ordem e alinhamento para ajustar o layout conforme necessário.
-

5.4 Componentes Básicos do Bootstrap

Bootstrap oferece uma vasta coleção de componentes prontos para usar que facilitam a criação de interfaces de usuário elegantes e funcionais. Neste tópico, vamos explorar alguns dos componentes básicos mais utilizados, como botões, cards, navbars e muito mais. 

5.4.1 Botões

Os botões são elementos essenciais em qualquer aplicação web. Bootstrap fornece uma variedade de estilos para botões, permitindo que você escolha o que melhor se adapta ao seu design.



Documentação - Botões

Exemplo de Botões Básicos:

```
<button class="btn btn-primary">Primário</button>
<button class="btn btn-secondary">Secundário</button>
<button class="btn btn-success">Sucesso</button>
<button class="btn btn-danger">Perigo</button>
<button class="btn btn-warning">Aviso</button>
<button class="btn btn-info">Informação</button>
<button class="btn btn-light">Claro</button>
```

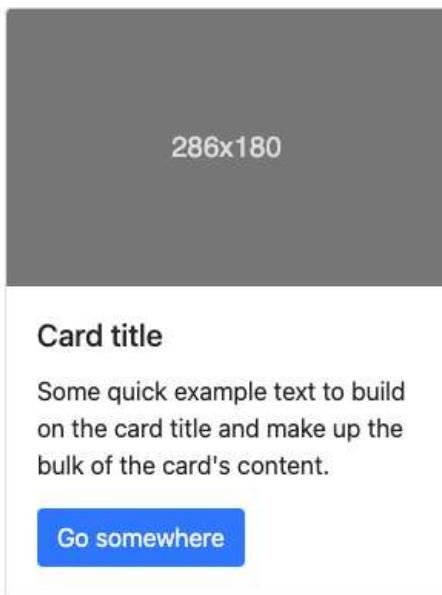
```
<button class="btn btn-dark">Escuro</button>
<button class="btn btn-link">Link</button>
```

Botões com Tamanhos Diferentes:

```
<button class="btn btn-primary btn-lg">Grande</button>
<button class="btn btn-primary">Padrão</button>
<button class="btn btn-primary btn-sm">Pequeno</button>
```

5.4.2 Cards

Os cards são contêineres flexíveis e extensíveis que incluem opções para cabeçalhos, rodapés, uma ampla variedade de conteúdo e muito mais.



Documentação - Cards

Exemplo de um Card Básico:

```
<div class="card" style="width: 18rem;">
  
```

```

<div class="card-body">
  <h5 class="card-title">Título do Card</h5>
  <p class="card-text">Algum texto de exemplo rápido para c
onstruir o título do card e fazer preencher o conteúdo do car
d.</p>
  <a href="#" class="btn btn-primary">Ir em algum lugar</a>
</div>
</div>

```

Cards com Vários Conteúdos:

```

<div class="card" style="width: 18rem;">
  
  <div class="card-body">
    <h5 class="card-title">Título do Card</h5>
    <p class="card-text">Algum texto de exemplo rápido para c
onstruir o título do card e fazer preencher o conteúdo do car
d.</p>
  </div>
  <ul class="list-group list-group-flush">
    <li class="list-group-item">Item 1</li>
    <li class="list-group-item">Item 2</li>
    <li class="list-group-item">Item 3</li>
  </ul>
  <div class="card-body">
    <a href="#" class="card-link">Link do Card</a>
    <a href="#" class="card-link">Outro Link</a>
  </div>
</div>

```

5.4.3 Navbars

Navbars são componentes de navegação responsivos que incluem suporte para branding, navegação e muito mais. Elas podem ser customizadas para atender às

necessidades de qualquer projeto.



Documentação - Navbars

Exemplo de uma Navbar Básica:

```
<nav class="navbar navbar-expand-lg navbar-light bg-light">
  <a class="navbar-brand" href="#">Navbar</a>
  <button class="navbar-toggler" type="button" data-toggle="collapse" data-target="#navbarNav" aria-controls="navbarNav" aria-expanded="false" aria-label="Toggle navigation">
    <span class="navbar-toggler-icon"></span>
  </button>
  <div class="collapse navbar-collapse" id="navbarNav">
    <ul class="navbar-nav">
      <li class="nav-item active">
        <a class="nav-link" href="#">Home <span class="sr-only">(atual)</span></a>
      </li>
      <li class="nav-item">
        <a class="nav-link" href="#">Recursos</a>
      </li>
      <li class="nav-item">
        <a class="nav-link" href="#">Preços</a>
      </li>
      <li class="nav-item">
        <a class="nav-link disabled" href="#">Desativado</a>
      </li>
    </ul>
  </div>
</nav>
```

Navbars com Formulários e Botões:

```
<nav class="navbar navbar-expand-lg navbar-light bg-light">
  <a class="navbar-brand" href="#">Navbar</a>
  <button class="navbar-toggler" type="button" data-toggle="collapse" data-target="#navbarNav" aria-controls="navbarNav" aria-expanded="false" aria-label="Toggle navigation">
    <span class="navbar-toggler-icon"></span>
  </button>
  <div class="collapse navbar-collapse" id="navbarNav">
    <ul class="navbar-nav">
      <li class="nav-item active">
        <a class="nav-link" href="#">Home <span class="sr-only" style="font-size: 1em; font-weight: bold;">(atual)</span></a>
      </li>
      <li class="nav-item">
        <a class="nav-link" href="#">Recursos</a>
      </li>
      <li class="nav-item">
        <a class="nav-link" href="#">Preços</a>
      </li>
      <li class="nav-item">
        <a class="nav-link disabled" href="#">Desativado</a>
      </li>
    </ul>
    <form class="form-inline my-2 my-lg-0 ml-auto">
      <input class="form-control mr-sm-2" type="search" placeholder="Buscar" aria-label="Buscar">
      <button class="btn btn-outline-success my-2 my-sm-0" type="submit">Buscar</button>
    </form>
  </div>
</nav>
```

5.4.4 Alertas

Os alertas são componentes usados para fornecer feedback contextual para ações do usuário.

This is a primary alert—check it out!

This is a secondary alert—check it out!

This is a success alert—check it out!

This is a danger alert—check it out!

This is a warning alert—check it out!

This is a info alert—check it out!

This is a light alert—check it out!

This is a dark alert—check it out!

Documentação - Alerts

Exemplo de Alertas:

```
<div class="alert alert-primary" role="alert">
  Este é um alerta primário—verifique-o!
</div>
<div class="alert alert-secondary" role="alert">
  Este é um alerta secundário—verifique-o!
</div>
<div class="alert alert-success" role="alert">
  Este é um alerta de sucesso—verifique-o!
</div>
<div class="alert alert-danger" role="alert">
  Este é um alerta de perigo—verifique-o!
</div>
```

```
<div class="alert alert-warning" role="alert">
  Este é um alerta de aviso—verifique-o!
</div>
<div class="alert alert-info" role="alert">
  Este é um alerta de informação—verifique-o!
</div>
<div class="alert alert-light" role="alert">
  Este é um alerta claro—verifique-o!
</div>
<div class="alert alert-dark" role="alert">
  Este é um alerta escuro—verifique-o!
</div>
```



Exercícios Práticos

Vamos praticar o uso dos componentes básicos do Bootstrap. Execute as seguintes operações:

1. **Crie uma página que inclua botões estilizados do Bootstrap.**
2. **Adicione um card com imagem, título, texto e links.**
3. **Crie uma navbar com links de navegação e um formulário de busca.**

5.5 Utilitários e Classes de Ajuda

Bootstrap oferece uma série de utilitários e classes de ajuda que facilitam a estilização rápida e eficaz de elementos HTML. Esses utilitários incluem classes para margens, paddings, alinhamento, visibilidade, entre outros. Vamos explorar algumas das classes de utilitários mais úteis e aprender a aplicá-las em nossos projetos. 🔧✨

5.5.1 Margens e Paddings

As classes de margem e padding do Bootstrap ajudam a ajustar o espaçamento ao redor dos elementos. Elas seguem a convenção `m` para margens e `p` para padding, seguidas por um valor que indica o tamanho do espaçamento.

Sintaxe Geral:

```
<!-- Margens -->
<div class="m-0">Margem 0</div>
<div class="mt-1">Margem superior 1</div>
<div class="mr-2">Margem direita 2</div>
<div class="mb-3">Margem inferior 3</div>
<div class="ml-4">Margem esquerda 4</div>
<div class="mx-5">Margens horizontais 5</div>
<div class="my-2">Margens verticais 2</div>

<!-- Paddings -->
<div class="p-0">Padding 0</div>
<div class="pt-1">Padding superior 1</div>
<div class="pr-2">Padding direito 2</div>
<div class="pb-3">Padding inferior 3</div>
<div class="pl-4">Padding esquerdo 4</div>
<div class="px-5">Paddings horizontais 5</div>
<div class="py-2">Paddings verticais 2</div>
```

Exemplo de Uso de Margens e Paddings:

```
<div class="container">
  <div class="row">
    <div class="col-12 mt-3">
      <div class="p-3 mb-2 bg-light text-dark">Box com padding e margem</div>
    </div>
  </div>
</div>
```

5.5.2 Alinhamento de Texto

As classes de alinhamento de texto permitem que você ajuste o alinhamento do texto dentro dos elementos.

Sintaxe Geral:

```
<!-- Alinhamento de Texto -->
<p class="text-left">Texto alinhado à esquerda</p>
<p class="text-center">Texto centralizado</p>
<p class="text-right">Texto alinhado à direita</p>
<p class="text-justify">Texto justificado</p>
<p class="text nowrap">Texto sem quebra de linha</p>
```

Exemplo de Alinhamento de Texto:

```
<div class="container">
  <div class="row">
    <div class="col-12">
      <p class="text-center">Este texto está centralizado.</p>
    </div>
  </div>
</div>
```

5.5.3 Visibilidade

As classes de visibilidade permitem que você mostre ou oculte elementos em diferentes tamanhos de tela.

Sintaxe Geral:

```
<!-- Visibilidade -->
<div class="d-none">Sempre escondido</div>
<div class="d-block">Sempre visível</div>
```

```
<div class="d-sm-none">Escondido em telas pequenas</div>
<div class="d-md-block d-lg-none">Visível apenas em telas médias</div>
```

Exemplo de Controle de Visibilidade:

```
<div class="container">
  <div class="row">
    <div class="col-12 d-none d-md-block">
      Este conteúdo é visível apenas em telas médias e maiores.
    </div>
  </div>
</div>
```

5.5.4 Alinhamento de Elementos

Bootstrap também oferece classes para alinhar elementos horizontalmente e verticalmente.

Sintaxe Geral:

```
<!-- Alinhamento Horizontal -->
<div class="d-flex justify-content-start">Alinhado à esquerda
</div>
<div class="d-flex justify-content-center">Alinhado ao centro
</div>
<div class="d-flex justify-content-end">Alinhado à direita</div>
<div class="d-flex justify-content-between">Espaçado entre os itens</div>
<div class="d-flex justify-content-around">Espaçado ao redor dos itens</div>

<!-- Alinhamento Vertical -->
```

```
<div class="d-flex align-items-start">Alinhado ao topo</div>
<div class="d-flex align-items-center">Alinhado ao centro</di
v>
<div class="d-flex align-items-end">Alinhado ao fundo</div>
```

Exemplo de Alinhamento de Elementos:

```
<div class="container">
  <div class="d-flex justify-content-center align-items-ce
ntr" style="height: 100vh;">
    <div class="p-3 mb-2 bg-light text-dark">Centralizado Hor
izontal e Verticalmente</div>
  </div>
</div>
```

5.5.5 Cores de Fundo e Texto

O Bootstrap oferece classes para definir cores de fundo e texto de forma rápida.

Sintaxe Geral:

```
<!-- Cores de Fundo -->
<div class="bg-primary text-white">Fundo Primário</div>
<div class="bg-secondary text-white">Fundo Secundário</div>
<div class="bg-success text-white">Fundo de Sucesso</div>
<div class="bg-danger text-white">Fundo de Perigo</div>
<div class="bg-warning text-dark">Fundo de Aviso</div>
<div class="bg-info text-white">Fundo de Informação</div>
<div class="bg-light text-dark">Fundo Claro</div>
<div class="bg-dark text-white">Fundo Escuro</div>

<!-- Cores de Texto -->
<p class="text-primary">Texto Primário</p>
<p class="text-secondary">Texto Secundário</p>
<p class="text-success">Texto de Sucesso</p>
```

```
<p class="text-danger">Texto de Perigo</p>
<p class="text-warning">Texto de Aviso</p>
<p class="text-info">Texto de Informação</p>
<p class="text-light bg-dark">Texto Claro</p>
<p class="text-dark">Texto Escuro</p>
```

Exemplo de Uso de Cores de Fundo e Texto:

```
<div class="container">
  <div class="row">
    <div class="col-12 bg-primary text-white p-3">
      Este é um texto com fundo primário e texto branco.
    </div>
  </div>
</div>
```



Exercícios Práticos

Vamos praticar o uso dos utilitários e classes de ajuda do Bootstrap. Execute as seguintes operações:

- 1. Ajuste margens e paddings de elementos usando as classes de utilitários.**
- 2. Alinhe texto de diferentes formas usando as classes de alinhamento de texto.**
- 3. Controle a visibilidade de elementos em diferentes tamanhos de tela.**
- 4. Aplique alinhamento horizontal e vertical em elementos.**
- 5. Utilize classes de cores de fundo e texto para estilizar elementos.**

5.6 Projeto: Portfólio com Bootstrap

Vamos construir um portfólio bonito e funcional usando Bootstrap. Este projeto vai consolidar todos os conceitos e componentes que aprendemos até agora, desde o sistema de grid até a customização de componentes. Vamos dividir a criação do portfólio em várias seções: Cabeçalho, Sobre Mim, Projetos, e Contato. 

5.6.1 Estrutura do Projeto

Antes de começarmos, vamos criar a estrutura básica do nosso projeto. Certifique-se de que seu diretório de trabalho esteja configurado da seguinte forma:

```
portfolio/
├── css/
│   └── custom.css
├── img/
│   └── perfil.jpg
└── index.html
```

5.6.2 Cabeçalho (Header)

Vamos começar criando a seção do cabeçalho, que incluirá uma barra de navegação e uma imagem de fundo.

HTML:

```
<!DOCTYPE html>
<html lang="pt-br">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Portfólio</title>
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css">
    <link rel="stylesheet" href="css/custom.css">
  </head>
  <body>
```

```

<!-- Cabeçalho -->
<header class="bg-dark text-white text-center py-5">
    <nav class="navbar navbar-expand-lg navbar-dark bg-dark">
        <a class="navbar-brand" href="#">Meu Portfólio</a>
        <button class="navbar-toggler" type="button" data-toggle="collapse" data-target="#navbarNav" aria-controls="navbarNav" aria-expanded="false" aria-label="Toggle navigation">
            <span class="navbar-toggler-icon"></span>
        </button>
        <div class="collapse navbar-collapse" id="navbarNav">
            <ul class="navbar-nav ml-auto">
                <li class="nav-item"><a class="nav-link" href="#sobre">Sobre Mim</a></li>
                <li class="nav-item"><a class="nav-link" href="#projetos">Projetos</a></li>
                <li class="nav-item"><a class="nav-link" href="#contato">Contato</a></li>
            </ul>
        </div>
    </nav>
    <div class="container">
        <h1 class="display-4">Bem-vindo ao Meu Portfólio</h1>
        <p class="lead">Desenvolvedor Full Stack apaixonado por criar soluções incríveis.</p>
    </div>
</header>

```

CSS (custom.css):

```

body {
    font-family: Arial, sans-serif;
}

```

```
header {
    background: url('../img/perfil.jpg') no-repeat center center;
    background-size: cover;
    height: 100vh;
    display: flex;
    flex-direction: column;
    justify-content: center;
    align-items: center;
    color: white;
    text-shadow: 1px 1px 5px rgba(0, 0, 0, 0.7);
}

header .navbar {
    position: fixed;
    top: 0;
    width: 100%;
    z-index: 1000;
}

header .container {
    z-index: 2;
}

header::before {
    content: '';
    position: absolute;
    top: 0;
    left: 0;
    right: 0;
    bottom: 0;
    background: rgba(0, 0, 0, 0.5);
    z-index: 1;
}
```

5.6.3 Sobre Mim (About Me)

A seção "Sobre Mim" vai incluir uma imagem de perfil e uma breve descrição.

HTML:

```
<!-- Sobre Mim -->
<section id="sobre" class="py-5 bg-light">
  <div class="container">
    <div class="row">
      <div class="col-md-4">
        
      </div>
      <div class="col-md-8">
        <h2>Sobre Mim</h2>
        <p>Olá! Meu nome é [Seu Nome], sou um desenvolvedor Full Stack com paixão por criar soluções inovadoras e eficientes. Tenho experiência em diversas tecnologias e frameworks, e estou sempre buscando aprender e me aprimorar.</p>
        <p>Minhas habilidades incluem HTML, CSS, JavaScript, React, Node.js, entre outras. Adoro trabalhar em equipe e acredito que a colaboração é a chave para o sucesso de qualquer projeto.</p>
      </div>
    </div>
  </div>
</section>
```

CSS (custom.css):

```
#sobre img {
  border: 5px solid #fff;
}
```

5.6.4 Projetos (Projects)

A seção de projetos vai exibir cartões com uma imagem, título e descrição para cada projeto.

HTML:

```
<!-- Projetos -->
<section id="projetos" class="py-5">
    <div class="container">
        <h2 class="text-center mb-4">Projetos</h2>
        <div class="row">
            <div class="col-md-4">
                <div class="card mb-4">
                    
                    <div class="card-body">
                        <h5 class="card-title">Projeto 1</h5>
                        <p class="card-text">Descrição breve do Projeto
                            1.</p>
                    </div>
                </div>
            </div>
            <div class="col-md-4">
                <div class="card mb-4">
                    
                    <div class="card-body">
                        <h5 class="card-title">Projeto 2</h5>
                        <p class="card-text">Descrição breve do Projeto
                            2.</p>
                    </div>
                </div>
            </div>
            <div class="col-md-4">
                <div class="card mb-4">
                    
```

```

        <div class="card-body">
            <h5 class="card-title">Projeto 3</h5>
            <p class="card-text">Descrição breve do Projeto
            3.</p>
        </div>
    </div>
</div>
</div>
</div>
</div>
</div>
</section>

```

CSS (custom.css):

```

#projetos .card {
    box-shadow: 0 4px 8px rgba(0, 0, 0, 0.1);
    transition: transform 0.3s;
}

#projetos .card:hover {
    transform: scale(1.05);
}

```

5.6.5 Contato (Contact)

A seção de contato vai incluir um formulário simples para que visitantes possam enviar mensagens.

HTML:

```

<!-- Contato -->
<section id="contato" class="py-5 bg-light">
    <div class="container">
        <h2 class="text-center mb-4">Contato</h2>
        <div class="row">
            <div class="col-md-8 offset-md-2">
                <form>

```

```

        <div class="form-group">
            <label for="nome">Nome</label>
            <input type="text" class="form-control" id="nome"
e" placeholder="Seu nome">
        </div>
        <div class="form-group">
            <label for="email">Email</label>
            <input type="email" class="form-control" id="email"
placeholder="Seu email">
        </div>
        <div class="form-group">
            <label for="mensagem">Mensagem</label>
            <textarea class="form-control" id="mensagem" rows="4"
placeholder="Sua mensagem"></textarea>
        </div>
        <button type="submit" class="btn btn-primary">Enviar</button>
    </form>
</div>
</div>
</div>
</section>

<!-- Rodapé -->
<footer class="bg-dark text-white text-center py-4">
    <p>&copy; 2023 Meu Portfólio</p>
</footer>
</body>
</html>

```

CSS (custom.css):

```

#contato .form-control {
    margin-bottom: 1rem;
}

```



Exercícios Práticos

Vamos praticar a criação do portfólio com Bootstrap. Execute as seguintes operações:

- **Adicione uma seção de depoimentos:**

- Crie uma nova seção abaixo da seção "Projetos".
- Adicione depoimentos de clientes ou colegas usando cartões do Bootstrap.

- **Adicione um efeito de rolagem suave:**

- Implemente um efeito de rolagem suave para os links da barra de navegação.
- Utilize jQuery ou JavaScript para adicionar a funcionalidade de rolagem suave.

- **Melhore a acessibilidade:**

- Adicione atributos `aria-label` nos links e botões para melhorar a acessibilidade.
- Certifique-se de que o contraste das cores esteja adequado para leitores de tela.

- **Adicione animações:**

- Adicione animações de entrada usando a biblioteca Animate.css ou CSS personalizado.
- Implemente animações para elementos específicos, como os cartões de projetos e a imagem de perfil.

- **Otimize o layout para dispositivos móveis:**

- Verifique a responsividade do portfólio em diferentes tamanhos de tela.
- Ajuste o layout e o espaçamento para garantir uma ótima experiência em dispositivos móveis.

módulo 6

banco de dados

Domine a arte de gerenciar dados com bancos de dados e SQL. 

06 tópicos neste módulo

Módulo 6: Banco de Dados

6.1 Introdução aos Bancos de Dados

Os bancos de dados são fundamentais para a maioria das aplicações, permitindo o armazenamento, a recuperação e a manipulação eficiente de dados. Neste tópico, vamos explorar os conceitos básicos de bancos de dados, entender como eles funcionam, e discutir os diferentes tipos de bancos de dados disponíveis.



6.1.1 O que é um Banco de Dados?

Um banco de dados é uma coleção organizada de dados, geralmente armazenados e acessados eletronicamente a partir de um sistema de computador. Os bancos de dados permitem que grandes volumes de informações sejam armazenados de maneira estruturada e eficiente, facilitando a consulta, a atualização e a exclusão de dados.

Principais Conceitos:

- **Tabelas:** Estruturas que organizam dados em linhas e colunas. Cada tabela representa uma entidade, como "Usuários" ou "Pedidos".
- **Registros:** Cada linha em uma tabela representa um registro único.
- **Campos:** Cada coluna em uma tabela representa um campo de dados específico, como "Nome", "Email" ou "Idade".
- **Chave Primária:** Um campo ou combinação de campos que identifica de forma exclusiva cada registro em uma tabela.
- **Chave Estrangeira:** Um campo que cria um relacionamento entre duas tabelas, referenciando a chave primária de outra tabela.

6.1.2 Tipos de Bancos de Dados

Existem vários tipos de bancos de dados, cada um com suas características e casos de uso específicos. Os mais comuns são:

- **Bancos de Dados Relacionais:** Armazenam dados em tabelas que podem ser relacionadas entre si. Exemplos incluem MySQL, PostgreSQL, Oracle e SQL Server.
- **Bancos de Dados NoSQL:** Projetados para armazenar e recuperar dados de forma rápida e eficiente em grandes volumes. Exemplos incluem MongoDB, Cassandra e Redis.
- **Bancos de Dados em Memória:** Armazenam dados na memória RAM para acesso rápido. Exemplos incluem Redis e Memcached.

6.1.3 SQL - Structured Query Language

SQL (Structured Query Language) é a linguagem padrão usada para interagir com bancos de dados relacionais. Com SQL, você pode realizar várias operações em um banco de dados, incluindo a criação de tabelas, a inserção de dados, a consulta de dados, a atualização de registros e a exclusão de dados.

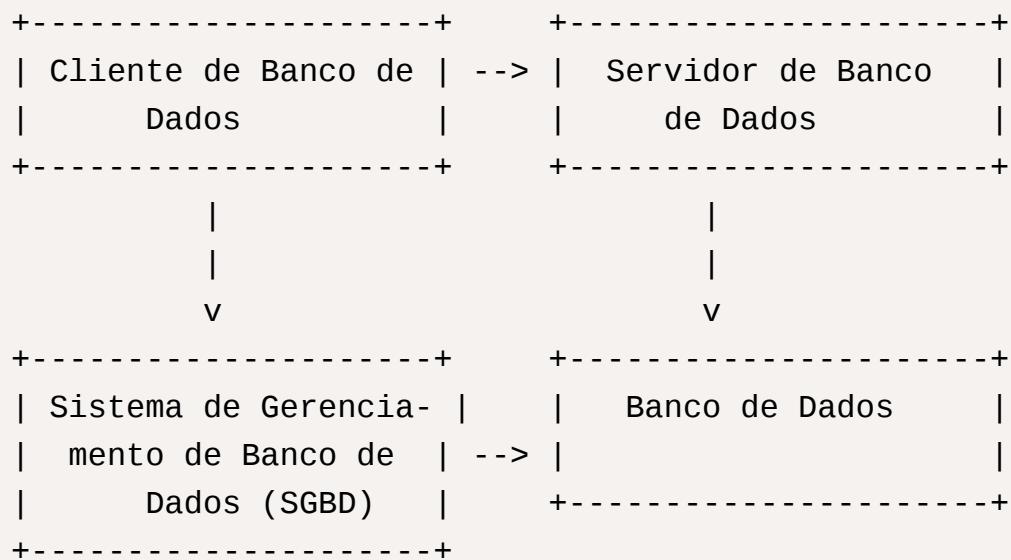
6.1.4 Arquitetura de um Banco de Dados

A arquitetura de um banco de dados envolve várias camadas e componentes que trabalham juntos para gerenciar e fornecer acesso aos dados.

Componentes Principais:

- **Sistema de Gerenciamento de Banco de Dados (SGBD):** Software que gerencia o banco de dados, lida com a criação, leitura, atualização e exclusão de dados, além de fornecer mecanismos de segurança e integridade.
- **Servidor de Banco de Dados:** Hardware onde o SGBD é executado.
- **Cliente de Banco de Dados:** Interface ou aplicação que se comunica com o SGBD para realizar operações no banco de dados.

Diagrama de Arquitetura de Banco de Dados:



6.2 Como rodar um banco de dados com Docker

Usar Docker para rodar bancos de dados é uma maneira eficiente de configurar e gerenciar ambientes de desenvolvimento. Neste tópico, vamos aprender como configurar um banco de dados usando Docker. 

6.2.1 O que é Docker?

Docker é uma plataforma que permite criar, distribuir e executar aplicações em contêineres. Contêineres são unidades leves e portáteis que incluem tudo o que a aplicação precisa para funcionar, como o código, as bibliotecas e as dependências.

6.2.2 Instalando Docker

Antes de começar, precisamos instalar o Docker. Você pode baixar e instalar o Docker Desktop a partir do site oficial: [Docker Download](#).

Instalação no Windows e Mac:

1. Baixe o instalador do Docker Desktop.
2. Execute o instalador e siga as instruções na tela.
3. Após a instalação, reinicie o computador, se necessário.

6.2.3 Configurando um Banco de Dados com Docker

Vamos configurar um banco de dados MySQL usando Docker. A seguir, estão os passos para rodar um contêiner MySQL:

- **Puxe a imagem do MySQL do Docker Hub:**

```
docker pull mysql:latest
```

- **Execute o contêiner MySQL:**

```
docker run --name meu_mysql -e MYSQL_ROOT_PASSWORD=minha_senha -p 3306:3306 -d mysql:latest
```

- `-name meu_mysql`: Nome do contêiner.
- `e MYSQL_ROOT_PASSWORD=minha_senha`: Define a senha do usuário root.
- `d mysql:latest` : Executa o contêiner em segundo plano usando a última versão do MySQL.

- **Verifique se o contêiner está rodando:**

```
docker ps
```

6.2.4 Conectando ao Banco de Dados

Após configurar o contêiner do MySQL, você pode se conectar ao banco de dados usando um cliente MySQL ou a linha de comando:

Usando a linha de comando:

```
docker exec -it meu_mysql mysql -u root -p
```

Digite a senha que você definiu anteriormente (`minha_senha`).

Usando um cliente MySQL:

Você pode usar um cliente MySQL, como MySQL Workbench, DBeaver, ou qualquer outro de sua preferência, para se conectar ao banco de dados. Use as seguintes configurações de conexão:

- **Host:** 127.0.0.1
- **Porta:** 3306
- **Usuário:** root
- **Senha:** minha_senha

6.3 Criação e Manipulação de Tabelas

A criação de tabelas é uma parte fundamental no design de bancos de dados. As tabelas armazenam os dados de forma estruturada, permitindo fácil acesso e manipulação. Neste tópico, vamos nos concentrar na criação de tabelas e definição de colunas, tipos de dados e restrições. 

6.3.1 Estrutura de uma Tabela

Uma tabela no banco de dados é composta por linhas (registros) e colunas (campos). Cada coluna tem um tipo de dado específico e pode ter restrições que garantem a integridade dos dados. Vamos explorar como definir essa estrutura.

6.3.2 Tipos de Dados Comuns

Os tipos de dados definem o tipo de valor que cada coluna pode armazenar. Aqui estão alguns dos tipos de dados mais comuns usados no MySQL:

- **INT**: Armazena números inteiros.
- **VARCHAR(n)**: Armazena strings de texto com comprimento variável até **n** caracteres.
- **TEXT**: Armazena strings de texto com comprimento variável (mais longo que VARCHAR).
- **DATE**: Armazena datas (ano, mês e dia).
- **TIMESTAMP**: Armazena data e hora.
- **DECIMAL(m, d)**: Armazena números decimais com **m** dígitos, dos quais **d** são decimais.

6.3.3 Criando uma Tabela

Vamos criar uma tabela chamada **usuarios** com várias colunas e diferentes tipos de dados.

Comando SQL para criar a tabela `usuarios` :

```
CREATE TABLE usuarios (
    id INT AUTO_INCREMENT PRIMARY KEY,
    nome VARCHAR(50) NOT NULL,
    email VARCHAR(50) NOT NULL,
    senha VARCHAR(255) NOT NULL,
    data_criacao TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

- **id INT AUTO_INCREMENT PRIMARY KEY** : Cria uma coluna **id** que é um número inteiro, auto-incrementado e chave primária.

- `nome VARCHAR(50) NOT NULL`: Cria uma coluna `nome` que armazena uma string de até 50 caracteres e não permite valores nulos.
- `email VARCHAR(50) NOT NULL`: Cria uma coluna `email` semelhante à coluna `nome`.
- `senha VARCHAR(255) NOT NULL`: Cria uma coluna `senha` para armazenar senhas criptografadas.
- `data_criacao TIMESTAMP DEFAULT CURRENT_TIMESTAMP`: Cria uma coluna `data_criacao` que armazena a data e hora de criação do registro, com o valor padrão definido como o momento atual.

6.3.4 Restrições e Chaves

As restrições ajudam a garantir a integridade dos dados. Aqui estão algumas das restrições mais comuns:

- **PRIMARY KEY**: Identifica de forma única cada registro na tabela.
- **NOT NULL**: Garante que a coluna não pode ter valores nulos.
- **UNIQUE**: Garante que todos os valores na coluna são únicos.
- **FOREIGN KEY**: Cria um relacionamento entre tabelas.

Exemplo de Tabela com Restrições:

```
CREATE TABLE produtos (
    id INT AUTO_INCREMENT PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    descricao TEXT,
    preco DECIMAL(10, 2) NOT NULL,
    data_criacao TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    categoria_id INT,
    FOREIGN KEY (categoria_id) REFERENCES categorias(id)
);
```

- `categoria_id INT`: Cria uma coluna que armazenará o ID da categoria do produto.

- `FOREIGN KEY (categoria_id) REFERENCES categorias(id)`: Define `categoria_id` como uma chave estrangeira que referencia a coluna `id` na tabela `categorias`.

6.3.5 Relacionamentos em Bancos de Dados

6.3.5.1 Relacionamento Um-para-Um

Um relacionamento um-para-um ocorre quando um registro em uma tabela está associado a no máximo um registro em outra tabela, e vice-versa. Esse tipo de relacionamento é menos comum, mas é útil em casos onde você quer dividir uma tabela muito grande em partes menores por motivos de desempenho ou segurança.

Exemplo:

```
CREATE TABLE pessoa (
    id INT PRIMARY KEY,
    nome VARCHAR(50)
);

CREATE TABLE passaporte (
    id INT PRIMARY KEY,
    numero VARCHAR(20),
    pessoa_id INT,
    FOREIGN KEY (pessoa_id) REFERENCES pessoa(id)
);
```

Neste exemplo, cada pessoa pode ter apenas um passaporte, e cada passaporte está associado a apenas uma pessoa.

6.3.5.2 Relacionamento Um-para-Muitos

Um relacionamento um-para-muitos ocorre quando um registro em uma tabela pode estar associado a múltiplos registros em outra tabela. Este é um dos relacionamentos mais comuns e reflete situações do mundo real, como uma categoria que pode ter vários produtos.

Exemplo:

```
CREATE TABLE categoria (
    id INT PRIMARY KEY,
    nome VARCHAR(50)
);

CREATE TABLE produto (
    id INT PRIMARY KEY,
    nome VARCHAR(50),
    categoria_id INT,
    FOREIGN KEY (categoria_id) REFERENCES categoria(id)
);
```

Aqui, uma categoria pode ter vários produtos, mas cada produto pertence a apenas uma categoria.

6.3.5.3 Relacionamento Muitos-para-Muitos

Um relacionamento muitos-para-muitos ocorre quando múltiplos registros em uma tabela podem estar associados a múltiplos registros em outra tabela. Para modelar este tipo de relacionamento, usamos uma tabela intermediária (também chamada de tabela de junção) que contém chaves estrangeiras de ambas as tabelas.

Exemplo:

```
CREATE TABLE autor (
    id INT PRIMARY KEY,
    nome VARCHAR(50)
);

CREATE TABLE livro (
    id INT PRIMARY KEY,
    titulo VARCHAR(50)
);
```

```
CREATE TABLE autor_livro (
    autor_id INT,
    livro_id INT,
    PRIMARY KEY (autor_id, livro_id),
    FOREIGN KEY (autor_id) REFERENCES autor(id),
    FOREIGN KEY (livro_id) REFERENCES livro(id)
);
```

Neste caso, um autor pode ter escrito vários livros e um livro pode ter sido escrito por vários autores.



Exercícios Práticos

Vamos praticar a criação de tabelas e definição de colunas no MySQL:

1. Crie uma tabela chamada `clientes` com os seguintes campos:

- `id` (INT, AUTO_INCREMENT, PRIMARY KEY)
- `nome` (VARCHAR(100), NOT NULL)
- `email` (VARCHAR(100), NOT NULL, UNIQUE)
- `telefone` (VARCHAR(20))
- `data_cadastro` (TIMESTAMP, DEFAULT CURRENT_TIMESTAMP)

2. Crie uma tabela chamada `pedidos` com os seguintes campos:

- `id` (INT, AUTO_INCREMENT, PRIMARY KEY)
- `cliente_id` (INT, NOT NULL)
- `produto_id` (INT, NOT NULL)
- `quantidade` (INT, NOT NULL)
- `data_pedido` (TIMESTAMP, DEFAULT CURRENT_TIMESTAMP)
- **Adicione uma chave estrangeira para `cliente_id` referenciando a tabela `clientes`.**

3. Crie uma tabela chamada `categorias` com os seguintes campos:

- `id` (INT, AUTO_INCREMENT, PRIMARY KEY)
 - `nome` (VARCHAR(50), NOT NULL)
-

6.4 Inserção, Atualização e Exclusão de Dados

Agora que sabemos como criar tabelas, é hora de aprender a manipular os dados dentro delas. Vamos abordar como inserir, atualizar e excluir dados usando comandos SQL. Essas operações são fundamentais para a gestão dos dados em qualquer aplicação. 

6.4.1 Inserindo Dados

Para inserir dados em uma tabela, utilizamos o comando `INSERT INTO`. Esse comando permite adicionar novos registros a uma tabela específica.

Sintaxe Geral:

```
INSERT INTO nome_da_tabela (coluna1, coluna2, coluna3) VALUES  
(valor1, valor2, valor3);
```

Exemplo:

Vamos inserir dados na tabela `usuarios` criada anteriormente:

```
INSERT INTO usuarios (nome, email, senha) VALUES ('João Silva', 'joao@example.com', 'senha123');  
  
INSERT INTO usuarios (nome, email, senha) VALUES ('Maria Oliveira', 'maria@example.com', 'senha456');  
  
INSERT INTO usuarios (nome, email, senha) VALUES ('Carlos Santos', 'carlos@example.com', 'senha789');
```

Cada comando insere um novo registro na tabela `usuarios` com os valores fornecidos para as colunas `nome`, `email` e `senha`.

6.4.2 Atualizando Dados

Para atualizar dados em uma tabela, usamos o comando `UPDATE`. Esse comando permite modificar os valores existentes em uma ou mais colunas de um ou mais registros.

Sintaxe Geral:

```
UPDATE nome_da_tabela SET coluna1 = valor1, coluna2 = valor2  
WHERE condição;
```

Exemplo:

Vamos atualizar o email de João Silva na tabela `usuarios`:

```
UPDATE usuarios SET email = 'novoemail@example.com' WHERE nom  
e = 'João Silva';
```

Este comando atualiza o valor da coluna `email` para 'novoemail@example.com' onde a coluna `nome` é igual a 'João Silva'.

6.4.3 Excluindo Dados

Para excluir dados de uma tabela, utilizamos o comando `DELETE`. Esse comando permite remover registros específicos de uma tabela.

Sintaxe Geral:

```
DELETE FROM nome_da_tabela WHERE condição;
```

Exemplo:

Vamos excluir o registro de Carlos Santos na tabela `usuarios`:

```
DELETE FROM usuarios WHERE nome = 'Carlos Santos';
```

Este comando remove o registro onde a coluna `nome` é igual a 'Carlos Santos'.



Exercícios Práticos

Vamos praticar a inserção, atualização e exclusão de dados no MySQL:

1. Insira novos registros na tabela `clientes` criada anteriormente:

- `nome` : 'Ana Paula'
- `email` : 'ana@example.com'
- `telefone` : '1234-5678'

2. Atualize o telefone de 'Ana Paula' para '8765-4321'

```
UPDATE clientes SET telefone = '8765-4321' WHERE nome = 'Ana Paula';
```

3. Exclua o registro de 'Ana Paula' da tabela `clientes`

```
DELETE FROM clientes WHERE nome = 'Ana Paula';
```

6.5 Consultas e Recuperação de Dados

Consultar e recuperar dados de um banco de dados é uma habilidade essencial para qualquer desenvolvedor. Utilizando o comando `SELECT`, podemos extrair informações específicas de uma tabela ou de várias tabelas relacionadas. Neste tópico, vamos explorar como realizar consultas básicas, usar filtros, ordenar resultados e juntar tabelas. A small icon depicting a magnifying glass over a bar chart.

6.5.1 Consultas Básicas com SELECT

O comando `SELECT` é utilizado para recuperar dados de uma tabela. A sintaxe básica é:

```
SELECT coluna1, coluna2 FROM nome_da_tabela;
```

Para selecionar todas as colunas de uma tabela, usamos `*`:

```
SELECT * FROM usuarios;
```

Exemplo:

Vamos recuperar todos os dados da tabela `usuarios`:

```
SELECT * FROM usuarios;
```

6.5.2 Filtrando Resultados com WHERE

Podemos filtrar os resultados usando a cláusula `WHERE`. Isso permite que a consulta retorne apenas os registros que atendem a determinadas condições.

Sintaxe Geral:

```
SELECT coluna1, coluna2 FROM nome_da_tabela WHERE condição;
```

Exemplo:

Vamos recuperar os usuários cujo nome é 'Maria Oliveira':

```
SELECT * FROM usuarios WHERE nome = 'Maria Oliveira';
```

6.5.3 Ordenando Resultados com ORDER BY

Para ordenar os resultados de uma consulta, usamos a cláusula `ORDER BY`.

Podemos ordenar em ordem ascendente (`ASC`) ou descendente (`DESC`).

Sintaxe Geral:

```
SELECT coluna1, coluna2 FROM nome_da_tabela ORDER BY coluna1  
ASC;
```

Exemplo:

Vamos recuperar todos os usuários ordenados pelo nome em ordem alfabética:

```
SELECT * FROM usuarios ORDER BY nome ASC;
```

6.5.4 Limitando Resultados com LIMIT

Podemos limitar o número de resultados retornados usando a cláusula `LIMIT`.

Sintaxe Geral:

```
SELECT coluna1, coluna2 FROM nome_da_tabela LIMIT número;
```

Exemplo:

Vamos recuperar apenas os dois primeiros usuários:

```
SELECT * FROM usuarios LIMIT 2;
```

6.5.5 Junção de Tabelas com JOIN

As junções são usadas para combinar registros de duas ou mais tabelas com base em uma condição relacionada. O tipo mais comum de junção é a `INNER JOIN`.

Sintaxe Geral:

```
SELECT a.coluna1, b.coluna2  
FROM tabela1 a  
INNER JOIN tabela2 b ON a.coluna_comum = b.coluna_comum;
```

Exemplo:

Vamos criar uma tabela `pedidos` e realizar uma junção com a tabela `usuarios` para recuperar informações sobre os pedidos feitos por cada usuário.

Criação da tabela `pedidos`:

```
CREATE TABLE pedidos (
    id INT AUTO_INCREMENT PRIMARY KEY,
    usuario_id INT,
    produto VARCHAR(50),
    quantidade INT,
    data_pedido TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (usuario_id) REFERENCES usuarios(id)
);
```

Inserção de dados na tabela `pedidos`:

```
INSERT INTO pedidos (usuario_id, produto, quantidade) VALUES
(1, 'Notebook', 1);
INSERT INTO pedidos (usuario_id, produto, quantidade) VALUES
(2, 'Mouse', 2);
INSERT INTO pedidos (usuario_id, produto, quantidade) VALUES
(1, 'Teclado', 1);
```

Consulta com JOIN:

```
SELECT usuarios.nome, pedidos.produto, pedidos.quantidade
FROM usuarios
INNER JOIN pedidos ON usuarios.id = pedidos.usuario_id;
```

Esta consulta retorna o nome dos usuários junto com os produtos que compraram e a quantidade de cada produto.



Exercícios Práticos

Vamos praticar a consulta e recuperação de dados no MySQL:

1. Recupere todos os registros da tabela `clientes`.
 2. Recupere todos os pedidos feitos por 'João Silva'.
 3. Recupere os emails de todos os clientes ordenados pelo nome em ordem descendente.
 4. Recupere os três primeiros registros da tabela `produtos`.
 5. Realize uma junção entre as tabelas `clientes` e `pedidos` para mostrar o nome do cliente e o produto comprado.
-

6.6 Funções Agregadas: COUNT, SUM, AVG e Mais

Funções agregadas são essenciais para realizar cálculos em grupos de dados, permitindo que você obtenha estatísticas valiosas sobre seus dados. Vamos explorar algumas das funções agregadas mais comuns no MySQL, como `COUNT`,

`SUM`, `AVG`, `MIN` e `MAX`. 

6.6.1 Função COUNT

A função `COUNT` retorna o número de registros em uma tabela ou o número de valores não nulos em uma coluna específica.

Sintaxe Geral:

```
SELECT COUNT(coluna) FROM nome_da_tabela;
```

Exemplo:

Vamos contar o número de usuários na tabela `usuarios`:

```
SELECT COUNT(*) FROM usuarios;
```

6.6.2 Função SUM

A função `SUM` retorna a soma total dos valores em uma coluna numérica.

Sintaxe Geral:

```
SELECT SUM(coluna) FROM nome_da_tabela;
```

Exemplo:

Vamos somar a quantidade total de produtos vendidos na tabela `pedidos` :

```
SELECT SUM(quantidade) FROM pedidos;
```

6.6.3 Função AVG

A função `AVG` retorna a média dos valores em uma coluna numérica.

Sintaxe Geral:

```
SELECT AVG(coluna) FROM nome_da_tabela;
```

Exemplo:

Vamos calcular a média de produtos vendidos por pedido na tabela `pedidos` :

```
SELECT AVG(quantidade) FROM pedidos;
```

6.6.4 Função MIN e MAX

As funções `MIN` e `MAX` retornam o menor e o maior valor em uma coluna, respectivamente.

Sintaxe Geral:

```
SELECT MIN(coluna) FROM nome_da_tabela;  
SELECT MAX(coluna) FROM nome_da_tabela;
```

Exemplo:

Vamos encontrar o preço mínimo e máximo na tabela `produtos` :

```
SELECT MIN(preco) AS preco_minimo, MAX(preco) AS preco_maximo  
FROM produtos;
```

6.6.5 Agrupando Resultados com GROUP BY

A cláusula `GROUP BY` é usada para agrupar registros que têm valores idênticos em colunas especificadas e, em seguida, aplicar funções agregadas nesses grupos.

Sintaxe Geral:

```
SELECT coluna1, AGG_FUNC(coluna2) FROM nome_da_tabela GROUP BY coluna1;
```

Exemplo:

Vamos agrupar os pedidos por `usuario_id` e calcular o total de produtos comprados por cada usuário:

```
SELECT usuario_id, SUM(quantidade) AS total_comprado FROM pedidos GROUP BY usuario_id;
```



Exercícios Práticos

Vamos praticar o uso de funções agregadas no MySQL:

- 1. Conte o número total de clientes na tabela `clientes`.**
- 2. Calcule a soma total de preços dos produtos na tabela `produtos`.**
- 3. Calcule a média de preços dos produtos na tabela `produtos`.**
- 4. Encontre o menor e o maior valor de `quantidade` na tabela `pedidos`.**
- 5. Agrupe os pedidos por `cliente_id` e calcule o total de produtos comprados por cada cliente.**

6.7 Gerenciamento de Transações

O gerenciamento de transações é uma parte crucial no desenvolvimento de sistemas que interagem com bancos de dados. Transações permitem que várias operações de banco de dados sejam executadas de forma segura e confiável, garantindo que todas as operações dentro da transação sejam completadas com sucesso ou nenhuma delas seja aplicada. Vamos explorar os conceitos de transações, como usá-las e sua importância na integridade dos dados. 

6.7.1 O que é uma Transação?

Uma transação é uma sequência de uma ou mais operações de banco de dados tratadas como uma única unidade lógica. Uma transação deve obedecer às propriedades ACID para garantir a integridade e confiabilidade dos dados:

- **Atomicidade (Atomicity)**: Todas as operações dentro da transação devem ser concluídas com sucesso; se qualquer operação falhar, todas as alterações feitas são desfeitas.
- **Consistência (Consistency)**: A transação deve levar o banco de dados de um estado consistente para outro estado consistente.
- **Isolamento (Isolation)**: As operações dentro de uma transação devem ser isoladas de outras transações simultâneas.
- **Durabilidade (Durability)**: Uma vez que a transação é confirmada, as mudanças feitas são permanentes, mesmo que ocorra uma falha do sistema.

6.7.2 Iniciando, Confirmando e Revertendo Transações

No MySQL, usamos comandos específicos para gerenciar transações: `START TRANSACTION`, `COMMIT` e `ROLLBACK`.

Sintaxe Geral:

```
START TRANSACTION;  
-- operações SQL  
COMMIT; -- confirma a transação  
-- ou  
ROLLBACK; -- reverte a transação
```

6.7.3 Exemplo de Uso de Transações

Vamos ver um exemplo de como usar transações para garantir a integridade dos dados ao transferir fundos entre duas contas bancárias.

Exemplo: Transferência de Fundos

Suponha que temos uma tabela `contas` com as seguintes colunas: `id`, `saldo`.

Estrutura da tabela `contas`:

```
CREATE TABLE contas (  
    id INT AUTO_INCREMENT PRIMARY KEY,  
    saldo DECIMAL(10, 2) NOT NULL  
)
```

Inserindo dados na tabela `contas`:

```
INSERT INTO contas (saldo) VALUES (1000.00);  
INSERT INTO contas (saldo) VALUES (2000.00);
```

Vamos transferir R\$500 da conta com `id = 1` para a conta com `id = 2`.

Transação de Transferência de Fundos:

```
START TRANSACTION;  
  
-- Subtrair 500 da conta 1  
UPDATE contas SET saldo = saldo - 500.00 WHERE id = 1;
```

```
-- Adicionar 500 à conta 2
UPDATE contas SET saldo = saldo + 500.00 WHERE id = 2;

-- Confirmar a transação
COMMIT;
```

Se qualquer uma das operações falhar, podemos reverter a transação:

```
START TRANSACTION;

-- Subtrair 500 da conta 1
UPDATE contas SET saldo = saldo - 500.00 WHERE id = 1;

-- Adicionar 500 à conta 2
UPDATE contas SET saldo = saldo + 500.00 WHERE id = 2;

-- Verificação de erro
-- Se ocorrer um erro, reverter a transação
IF (erro_ocorreu) THEN
    ROLLBACK;
ELSE
    COMMIT;
END IF;
```



Exercícios Práticos

Vamos praticar o gerenciamento de transações no MySQL:

1. Crie uma tabela chamada `estoque` com as seguintes colunas:

- `id` (INT, AUTO_INCREMENT, PRIMARY KEY)
- `produto` (VARCHAR(100))
- `quantidade` (INT)

```
CREATE TABLE estoque (
    id INT AUTO_INCREMENT PRIMARY KEY,
    produto VARCHAR(100) NOT NULL,
    quantidade INT NOT NULL
);
```

2. Insira alguns produtos na tabela `estoque`.

```
INSERT INTO estoque (produto, quantidade) VALUES ('Produto A', 100);
INSERT INTO estoque (produto, quantidade) VALUES ('Produto B', 200);
```

3. Inicie uma transação para vender 10 unidades do 'Produto A' e adicionar 10 unidades ao 'Produto B'. Se qualquer operação falhar, reverta a transação.

```
START TRANSACTION;

UPDATE estoque SET quantidade = quantidade - 10 WHERE produto = 'Produto A';
UPDATE estoque SET quantidade = quantidade + 10 WHERE produto = 'Produto B';

-- Suponha que a segunda operação falha
ROLLBACK; -- Reverta a transação
```

4. Agora, execute a transação novamente, mas desta vez com sucesso.

```
START TRANSACTION;

UPDATE estoque SET quantidade = quantidade - 10 WHERE produto = 'Produto A';
UPDATE estoque SET quantidade = quantidade + 10 WHERE produto = 'Produto B';
```

```
COMMIT; -- Confirme a transação
```

6.8 Se Aprofundando com os Joins

Os joins são uma parte fundamental do SQL que permite combinar dados de duas ou mais tabelas com base em uma condição relacionada. Dominar os joins é essencial para realizar consultas complexas e obter uma visão abrangente dos dados. Vamos explorar os diferentes tipos de joins e como usá-los efetivamente.



6.8.1 Tipos de Joins

Existem vários tipos de joins, cada um com um propósito específico:

- **INNER JOIN**: Retorna os registros que têm correspondências em ambas as tabelas.
- **LEFT JOIN (ou LEFT OUTER JOIN)**: Retorna todos os registros da tabela da esquerda e os registros correspondentes da tabela da direita. Se não houver correspondência, os resultados da tabela da direita serão `NULL`.
- **RIGHT JOIN (ou RIGHT OUTER JOIN)**: Retorna todos os registros da tabela da direita e os registros correspondentes da tabela da esquerda. Se não houver correspondência, os resultados da tabela da esquerda serão `NULL`.
- **FULL JOIN (ou FULL OUTER JOIN)**: Retorna todos os registros quando há uma correspondência em uma das tabelas. Se não houver correspondência, os resultados serão `NULL` para a tabela que não tem correspondência.

6.8.2 INNER JOIN

O `INNER JOIN` retorna apenas os registros que têm correspondências em ambas as tabelas.

Sintaxe Geral:

```
SELECT a.coluna1, b.coluna2  
FROM tabela1 a  
INNER JOIN tabela2 b ON a.coluna_comum = b.coluna_comum;
```

Exemplo:

Vamos supor que temos as tabelas `usuarios` e `pedidos`. Queremos recuperar os nomes dos usuários e os produtos que compraram.

```
SELECT usuarios.nome, pedidos.produto  
FROM usuarios  
INNER JOIN pedidos ON usuarios.id = pedidos.usuario_id;
```

6.8.3 LEFT JOIN

O `LEFT JOIN` retorna todos os registros da tabela da esquerda e os registros correspondentes da tabela da direita. Se não houver correspondência, os resultados da tabela da direita serão `NULL`.

Sintaxe Geral:

```
SELECT a.coluna1, b.coluna2  
FROM tabela1 a  
LEFT JOIN tabela2 b ON a.coluna_comum = b.coluna_comum;
```

Exemplo:

Vamos recuperar todos os usuários e os produtos que compraram, incluindo usuários que não fizeram nenhuma compra.

```
SELECT usuarios.nome, pedidos.produto  
FROM usuarios  
LEFT JOIN pedidos ON usuarios.id = pedidos.usuario_id;
```

6.8.4 RIGHT JOIN

O `RIGHT JOIN` retorna todos os registros da tabela da direita e os registros correspondentes da tabela da esquerda. Se não houver correspondência, os resultados da tabela da esquerda serão `NULL`.

Sintaxe Geral:

```
SELECT a.coluna1, b.coluna2  
FROM tabela1 a  
RIGHT JOIN tabela2 b ON a.coluna_comum = b.coluna_comum;
```

Exemplo:

Vamos recuperar todos os produtos comprados e os usuários que os compraram, incluindo produtos que ainda não foram comprados.

```
SELECT usuarios.nome, pedidos.produto  
FROM usuarios  
RIGHT JOIN pedidos ON usuarios.id = pedidos.usuario_id;
```

6.8.5 FULL JOIN

O `FULL JOIN` retorna todos os registros quando há uma correspondência em uma das tabelas. Se não houver correspondência, os resultados serão `NULL` para a tabela que não tem correspondência. **Nota:** O `FULL JOIN` não é suportado por todos os sistemas de gerenciamento de banco de dados.

Sintaxe Geral:

```
SELECT a.coluna1, b.coluna2  
FROM tabela1 a  
FULL JOIN tabela2 b ON a.coluna_comum = b.coluna_comum;
```

Exemplo:

Vamos recuperar todos os usuários e produtos, incluindo usuários que não fizeram nenhuma compra e produtos que não foram comprados.

```
SELECT usuarios.nome, pedidos.produto
FROM usuarios
FULL JOIN pedidos ON usuarios.id = pedidos.usuario_id;
```



Exercícios Práticos

Vamos praticar o uso de diferentes tipos de joins no MySQL:

1. Crie duas tabelas chamadas `alunos` e `matriculas`.

- `alunos` : `id` (INT, AUTO_INCREMENT, PRIMARY KEY), `nome` (VARCHAR(100))
- `matriculas` : `id` (INT, AUTO_INCREMENT, PRIMARY KEY), `aluno_id` (INT), `curso` (VARCHAR(100))

```
CREATE TABLE alunos (
    id INT AUTO_INCREMENT PRIMARY KEY,
    nome VARCHAR(100) NOT NULL
);

CREATE TABLE matriculas (
    id INT AUTO_INCREMENT PRIMARY KEY,
    aluno_id INT,
    curso VARCHAR(100) NOT NULL,
    FOREIGN KEY (aluno_id) REFERENCES alunos(id)
);
```

2. Insira alguns dados nas tabelas `alunos` e `matriculas`.

```
INSERT INTO alunos (nome) VALUES ('Ana Paula'), ('Carlos Santos');
INSERT INTO matriculas (aluno_id, curso) VALUES (1, 'Matemática'), (2, 'Física'), (1, 'Biologia');
```

- 3. Use um `INNER JOIN` para recuperar os nomes dos alunos e os cursos em que estão matriculados.**

```
SELECT alunos.nome, matriculas.curso  
FROM alunos  
INNER JOIN matriculas ON alunos.id = matriculas.aluno_id;
```

- 4. Use um `LEFT JOIN` para recuperar todos os alunos, incluindo aqueles que não estão matriculados em nenhum curso.**

```
SELECT alunos.nome, matriculas.curso  
FROM alunos  
LEFT JOIN matriculas ON alunos.id = matriculas.aluno_id;
```

- 5. Use um `RIGHT JOIN` para recuperar todos os cursos, incluindo aqueles que não têm alunos matriculados.**

```
SELECT alunos.nome, matriculas.curso  
FROM alunos  
RIGHT JOIN matriculas ON alunos.id = matriculas.aluno_id;
```

- 6. Tente usar um `FULL JOIN` (se suportado) para recuperar todos os alunos e cursos, incluindo aqueles que não têm correspondência.**

```
SELECT alunos.nome, matriculas.curso  
FROM alunos  
FULL JOIN matriculas ON alunos.id = matriculas.aluno_id;
```

6.9 Índices e Performance

Os índices são estruturas de dados especiais que melhoram a velocidade das operações de consulta em uma tabela de banco de dados. Eles são fundamentais

para otimizar a performance das suas consultas, especialmente em tabelas grandes. Vamos explorar o que são índices, como criá-los, e como eles impactam a performance do banco de dados. 

6.9.1 O que são Índices?

Um índice é uma estrutura de dados que melhora a velocidade de operações de leitura em uma tabela de banco de dados. Pense em um índice como o índice de um livro, que permite encontrar rapidamente a página onde um determinado tópico é discutido.

6.9.2 Tipos de Índices

Existem vários tipos de índices, cada um com suas vantagens e desvantagens:

- **Índice Primário (Primary Index)**: Criado automaticamente na chave primária de uma tabela.
- **Índice Único (Unique Index)**: Garante que todos os valores em uma coluna sejam únicos.
- **Índice Composto (Composite Index)**: Criado em mais de uma coluna, útil para consultas que filtram por múltiplas colunas.
- **Índice de Texto Completo (Full-Text Index)**: Usado para buscas de texto completo.
- **Índice de Cobertura (Covering Index)**: Inclui todas as colunas necessárias para uma consulta, evitando acessos adicionais à tabela.

6.9.3 Criando Índices

Vamos ver como criar diferentes tipos de índices no MySQL.

Índice Primário:

Já é criado automaticamente ao definir uma coluna como chave primária.

```
CREATE TABLE usuarios (
    id INT AUTO_INCREMENT PRIMARY KEY,
```

```
        nome VARCHAR(50),  
        email VARCHAR(50)  
    );
```

Índice Único:

```
CREATE UNIQUE INDEX idx_email_unico ON usuarios (email);
```

Índice Composto:

```
CREATE INDEX idx_nome_email ON usuarios (nome, email);
```

Índice de Texto Completo:

```
CREATE FULLTEXT INDEX idx_texto_completo ON artigos (conteudo);
```

6.9.4 Usando Índices para Melhorar Performance

Índices melhoram a velocidade de busca, mas também podem impactar negativamente a performance de inserções, atualizações e exclusões, pois o índice precisa ser atualizado. Por isso, é importante usar índices de forma estratégica.

6.9.5 Consultando com Índices

Vamos ver como índices podem melhorar a performance de consultas. Suponha que temos uma tabela `produtos`:

```
CREATE TABLE produtos (  
    id INT AUTO_INCREMENT PRIMARY KEY,  
    nome VARCHAR(100),  
    preco DECIMAL(10, 2),
```

```
        categoria_id INT  
    );
```

Criamos um índice na coluna `nome` para acelerar as buscas por nome de produto:

```
CREATE INDEX idx_nome_produto ON produtos (nome);
```

Agora, uma consulta que usa a coluna `nome` será mais rápida:

```
SELECT * FROM produtos WHERE nome = 'Notebook';
```

6.9.6 Monitorando e Mantendo Índices

Monitorar o uso de índices é importante para garantir que eles estejam realmente melhorando a performance. Ferramentas de análise de consulta, como o `EXPLAIN` no MySQL, ajudam a entender como uma consulta está sendo executada e se os índices estão sendo utilizados.

Usando EXPLAIN:

```
EXPLAIN SELECT * FROM produtos WHERE nome = 'Notebook';
```



Exercícios Práticos

Vamos praticar a criação e uso de índices no MySQL:

1. Crie uma tabela chamada `clientes` com as seguintes colunas:

- `id` (INT, AUTO_INCREMENT, PRIMARY KEY)
- `nome` (VARCHAR(100))
- `email` (VARCHAR(100), UNIQUE)
- `telefone` (VARCHAR(20))

```
CREATE TABLE clientes (
    id INT AUTO_INCREMENT PRIMARY KEY,
    nome VARCHAR(100),
    email VARCHAR(100) UNIQUE,
    telefone VARCHAR(20)
);
```

2. Crie um índice na coluna `nome` para melhorar a busca por nome de cliente.

```
CREATE INDEX idx_nome_cliente ON clientes (nome);
```

3. Crie um índice composto nas colunas `nome` e `email`.

```
CREATE INDEX idx_nome_email_cliente ON clientes (nome, email);
```

4. Use a ferramenta `EXPLAIN` para verificar se os índices estão sendo utilizados em uma consulta.

```
EXPLAIN SELECT * FROM clientes WHERE nome = 'Ana Paula';
```

5. Monitore a performance de consultas usando índices e ajuste conforme necessário.

6.10 Projeto - Sistema de Gerenciamento de Biblioteca

Neste projeto, vamos aplicar vários dos conceitos aprendidos até agora para criar um Sistema de Gerenciamento de Biblioteca. Vamos construir um banco de dados para gerenciar livros, membros, empréstimos e devoluções. Este projeto integrará a criação e manipulação de tabelas, consultas, transações, índices e mais. 

6.10.1 Estrutura do Projeto

Vamos criar as seguintes tabelas para o nosso sistema:

1. **livros**: Armazena informações sobre os livros disponíveis na biblioteca.
2. **membros**: Armazena informações sobre os membros da biblioteca.
3. **emprestimos**: Armazena informações sobre os empréstimos de livros.

6.10.2 Criando as Tabelas

Tabela livros:

```
CREATE TABLE livros (
    id INT AUTO_INCREMENT PRIMARY KEY,
    titulo VARCHAR(100) NOT NULL,
    autor VARCHAR(100) NOT NULL,
    ano_publicacao YEAR,
    genero VARCHAR(50),
    disponibilidade BOOLEAN DEFAULT TRUE
);
```

Tabela membros:

```
CREATE TABLE membros (
    id INT AUTO_INCREMENT PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL,
    telefone VARCHAR(20),
    data_cadastro TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Tabela emprestimos:

```
CREATE TABLE emprestimos (
    id INT AUTO_INCREMENT PRIMARY KEY,
    livro_id INT,
    membro_id INT,
```

```
    data_emprestimo DATE,  
    data_devolucao DATE,  
    status VARCHAR(20) DEFAULT 'emprestado',  
    FOREIGN KEY (livro_id) REFERENCES livros(id),  
    FOREIGN KEY (membro_id) REFERENCES membros(id)  
);
```

6.10.3 Inserindo Dados

Vamos inserir alguns dados nas tabelas `livros` e `membros`.

Inserindo dados na tabela `livros`:

```
INSERT INTO livros (titulo, autor, ano_publicacao, genero)  
VALUES  
( 'Dom Casmurro', 'Machado de Assis', 1899, 'Romance' ),  
( 'O Alquimista', 'Paulo Coelho', 1988, 'Ficção' ),  
( '1984', 'George Orwell', 1949, 'Distopia' );
```

Inserindo dados na tabela `membros`:

```
INSERT INTO membros (nome, email, telefone)  
VALUES  
( 'Ana Paula', 'ana@example.com', '1234-5678' ),  
( 'Carlos Santos', 'carlos@example.com', '8765-4321' );
```

6.10.4 Realizando Empréstimos

Vamos criar transações para gerenciar os empréstimos e devoluções de livros.

Transação para empréstimo de um livro:

```
START TRANSACTION;  
  
-- Atualizar disponibilidade do livro
```

```
UPDATE livros SET disponibilidade = FALSE WHERE id = 1;

-- Registrar o empréstimo
INSERT INTO emprestimos (livro_id, membro_id, data_emprestimo)
VALUES (1, 1, CURDATE());

COMMIT;
```

Transação para devolução de um livro:

```
START TRANSACTION;

-- Atualizar disponibilidade do livro
UPDATE livros SET disponibilidade = TRUE WHERE id = 1;

-- Atualizar status do empréstimo
UPDATE emprestimos SET data_devolucao = CURDATE(), status =
'devolvido' WHERE livro_id = 1 AND membro_id = 1 AND status =
'emprestado';

COMMIT;
```

6.10.5 Consultas Úteis

Vamos criar algumas consultas para obter informações importantes sobre o sistema de biblioteca.

Listar todos os livros disponíveis:

```
SELECT * FROM livros WHERE disponibilidade = TRUE;
```

Listar todos os livros emprestados e seus respectivos membros:

```
SELECT livros.titulo, membros.nome, emprestimos.data_emprestimo
FROM emprestimos
INNER JOIN livros ON emprestimos.livro_id = livros.id
INNER JOIN membros ON emprestimos.membro_id = membros.id
WHERE emprestimos.status = 'emprestado';
```

Listar todos os membros com empréstimos ativos:

```
SELECT DISTINCT membros.nome, membros.email
FROM membros
INNER JOIN emprestimos ON membros.id = emprestimos.membro_id
WHERE emprestimos.status = 'emprestado';
```

6.10.6 Índices para Melhorar Performance

Vamos criar alguns índices para melhorar a performance das consultas.

Índice na coluna **titulo** da tabela **livros**:

```
CREATE INDEX idx_titulo ON livros (titulo);
```

Índice na coluna **email** da tabela **membros**:

```
CREATE UNIQUE INDEX idx_email_membros ON membros (email);
```

Índice composto nas colunas **livro_id** e **membro_id** da tabela **emprestimos**:

```
CREATE INDEX idx_emprestimos ON emprestimos (livro_id, membro_id);
```



Exercícios Práticos

Vamos praticar criando e manipulando dados no nosso Sistema de Gerenciamento de Biblioteca:

1. Insira mais livros na tabela `livros`.

```
INSERT INTO livros (titulo, autor, ano_publicacao, genero)
VALUES
('Harry Potter e a Pedra Filosofal', 'J.K. Rowling', 1997,
'Fantasia'),
('O Senhor dos Anéis', 'J.R.R. Tolkien', 1954, 'Fantasia');
```

2. Insira mais membros na tabela `membros`.

```
INSERT INTO membros (nome, email, telefone)
VALUES
('Maria Oliveira', 'maria@example.com', '5678-1234'),
('João Silva', 'joao@example.com', '4321-8765');
```

3. Realize um empréstimo para um novo livro e um novo membro.

```
START TRANSACTION;
UPDATE livros SET disponibilidade = FALSE WHERE id = 4;
INSERT INTO emprestimos (livro_id, membro_id, data_emprestimo) VALUES (4, 3, CURDATE());
COMMIT;
```

4. Liste todos os empréstimos ativos.

```
SELECT livros.titulo, membros.nome, emprestimos.data_emprestimo
FROM emprestimos
INNER JOIN livros ON emprestimos.livro_id = livros.id
INNER JOIN membros ON emprestimos.membro_id = membros.id
WHERE emprestimos.status = 'emprestado';
```

módulo 7

backend básico com node.js

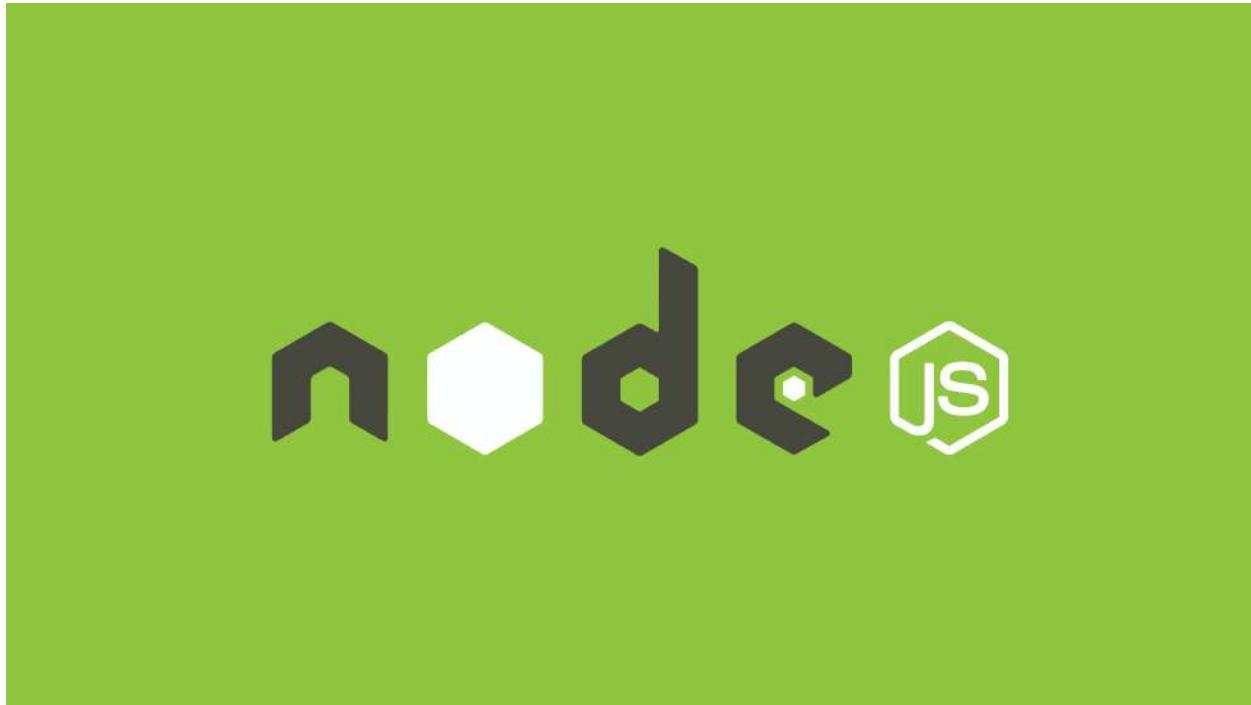
Construa a base do seu servidor com Node.js,
Express e EJS, e entenda a arquitetura MVC. 

09 tópicos neste módulo

Módulo 7: Backend básico com Node.js

7.1 Introdução ao Back-end e Modelos de Servidor

Neste capítulo, vamos mergulhar no desenvolvimento back-end usando Node.js. Vamos construir uma aplicação utilizando o padrão MVC (Model-View-Controller) com o framework Express e o motor de templates EJS. Nosso projeto será uma Aplicação de Restaurante que gerencia pedidos, e vamos construir esse projeto de forma progressiva, ensinando os conceitos de Node.js ao longo do caminho.



7.1.1 O que é Node.js e como ele funciona?

Node.js é um runtime de JavaScript baseado no motor V8 do Google Chrome, que permite executar código JavaScript fora do navegador. Ele é amplamente utilizado

para criar servidores web e aplicações de rede de alta performance devido à sua arquitetura assíncrona e orientada a eventos.

Vantagens do Node.js:

- **Alta Performance:** Graças ao motor V8 e à arquitetura não-bloqueante.
- **JavaScript no Back-end:** Permite que você use a mesma linguagem no front-end e no back-end.
- **Ampla Comunidade:** Uma vasta coleção de módulos disponíveis através do npm (Node Package Manager).

7.1.1.1 Funcionamento Interno do Node.js

Node.js é uma plataforma de desenvolvimento que permite executar código JavaScript fora do navegador. Para entender como o Node.js funciona internamente, é importante explorar alguns dos componentes-chave que ele utiliza: o motor V8, a biblioteca libuv, o Event Loop e as operações assíncronas.

V8: O Motor JavaScript

O V8 é o motor JavaScript de código aberto desenvolvido pelo Google, utilizado pelo navegador Chrome. Ele é responsável por compilar o código JavaScript diretamente para código de máquina, o que resulta em uma execução extremamente rápida. No contexto do Node.js, o V8 permite que o JavaScript seja executado no lado do servidor com alta performance.

Libuv: Biblioteca de I/O Assíncrono

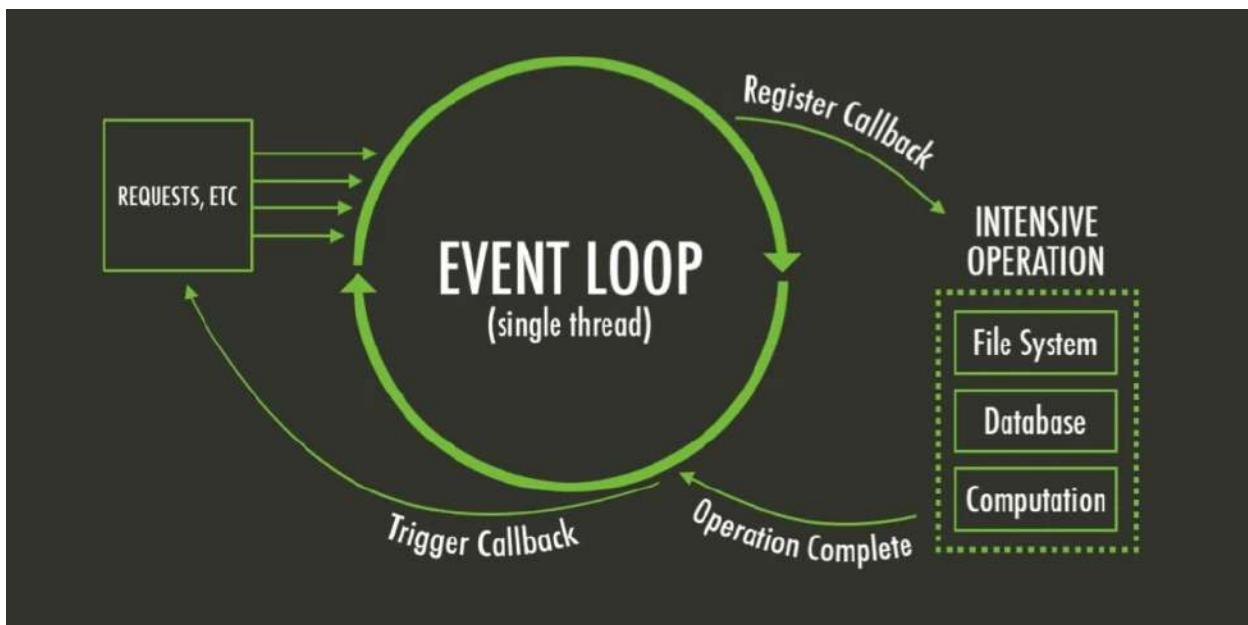
Libuv é uma biblioteca multi-plataforma que fornece ao Node.js uma API para operações assíncronas, como operações de sistema de arquivos, rede, e timers. Ela é crucial para a arquitetura não-bloqueante do Node.js, permitindo que o servidor lide com muitas conexões e operações simultâneas sem ficar bloqueado esperando uma operação ser concluída.

Event Loop: O Coração do Node.js

O Event Loop é o mecanismo interno do Node.js que lida com operações assíncronas. Ele permite que o Node.js realize operações de entrada/saída (I/O) de forma não-bloqueante. Quando uma operação assíncrona é iniciada (como ler um arquivo ou fazer uma requisição HTTP), ela é delegada à libuv, que a executa em segundo plano. Enquanto isso, o Event Loop continua a processar outras tarefas.

Como o Event Loop funciona:

- Timers:** Trata de callbacks agendados com `setTimeout` e `setInterval`.
- I/O Callbacks:** Processa callbacks de operações de I/O que foram completadas.
- Idle, Prepare:** Fases internas usadas em contextos específicos pela libuv.
- Poll:** Recupera novos eventos de I/O; se não houver eventos, o Event Loop pode adormecer.
- Check:** Processa callbacks agendados com `setImmediate`.
- Close Callbacks:** Executa callbacks de eventos de fechamento, como `socket.on('close')`.



Operações Assíncronas

A arquitetura do Node.js é projetada para ser assíncrona e não-bloqueante. Isso significa que quando uma operação de I/O é iniciada, o Node.js não espera que ela termine para continuar executando outras instruções. Em vez disso, ele registra um callback que será chamado quando a operação for concluída. Esse modelo é ideal para aplicações que necessitam de alta escalabilidade e capacidade de lidar com muitas conexões simultâneas, como servidores web.

7.1.1.2 Principais Funcionalidades do Node.js (APIs internas)

Node.js possui diversas APIs internas que facilitam o desenvolvimento de aplicações de rede e servidores. Algumas das principais são:

- **HTTP**: Permite criar servidores web e manipular requisições e respostas HTTP.
- **FS (File System)**: Oferece uma API para interagir com o sistema de arquivos, permitindo ler, escrever, e excluir arquivos.
- **Path**: Fornece utilitários para trabalhar com caminhos de arquivos e diretórios.
- **OS**: Fornece informações sobre o sistema operacional, como a quantidade de memória livre e a arquitetura da CPU.
- **Events**: Implementa um sistema de eventos, permitindo criar e manipular eventos personalizados.
- **Stream**: Facilita a manipulação de dados de fluxo contínuo, como leitura e escrita de arquivos e comunicação de rede.
- **Buffer**: Manipula dados binários de forma eficiente.
- **Crypto**: Proporciona funcionalidades de criptografia, como hash e cifragem.

7.1.2 Instalando Node.js e npm

Antes de começarmos a construir nossa Aplicação de Restaurante, precisamos instalar o Node.js e o npm.

1. Baixe e instale o Node.js:

- Acesse [Node.js Download](#) e baixe a versão recomendada para seu sistema operacional.
- Siga as instruções de instalação.

2. Verifique a instalação:

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

Se os comandos acima retornarem as versões do Node.js e npm, a instalação foi bem-sucedida. 🎉

7.1.3 Criando o Projeto Node.js

Vamos começar criando a estrutura do nosso projeto.

- **Crie uma pasta para o projeto:**

```
mkdir api-restaurante  
cd api-restaurante
```

- **Incialize um novo projeto Node.js:**

```
npm init -y
```

Isso cria um arquivo `package.json` com as configurações básicas do projeto.

7.1.4 Instalando Dependências

Vamos instalar as dependências necessárias para nossa aplicação: Express para criar o servidor, EJS como motor de templates, e algumas outras ferramentas úteis.

```
npm install express ejs sequelize body-parser
```

- **express:** Framework minimalista para Node.js.

- **ejs**: Motor de templates para gerar HTML a partir de dados.
- **sequelize**: ORM para comunicação com banco de dados.

7.1.5 Criando a Estrutura de Pastas

Vamos organizar nosso projeto seguindo o padrão MVC.

```
mkdir models views controllers routes
```

- **models**: Contém os modelos de dados.
- **views**: Contém as views (páginas EJS).
- **controllers**: Contém os controladores que gerenciam a lógica de negócios.
- **routes**: Contém as definições de rotas da aplicação.

7.1.6 Configurando o Servidor Express

Vamos configurar nosso servidor Express no arquivo `app.js`.

Crie o arquivo `app.js`:

```
const express = require('express');
const bodyParser = require("body-parser");

const app = express();
const port = 3000;

app.use(bodyParser.urlencoded({ extended: false }));
app.use(bodyParser.json());

// Configuração do EJS
app.set('view engine', 'ejs');

// Rota inicial
```

```
app.get('/', (req, res) => {
    res.send('Bem-vindo à Aplicação de Restaurante!');
});

// Inicia o servidor
app.listen(port, () => {
    console.log(`Servidor rodando em <http://localhost>:${port}`);
});
```

Execute o servidor:

```
node app.js
```

Abra o navegador e acesse <http://localhost:3000> para ver a mensagem de boas-vindas. 🎉



Exercícios Práticos

Vamos praticar os conceitos abordados até agora:

- 1. Instale Node.js e npm em sua máquina.**
- 2. Crie um novo projeto Node.js com a estrutura de pastas conforme descrito.**
- 3. Instale as dependências necessárias: express, ejs e sequelize.**
- 4. Configure o servidor Express e crie uma rota inicial que exiba uma mensagem de boas-vindas.**

Com essas etapas concluídas, estamos prontos para continuar construindo nossa API de Restaurante. 🚀✨

7.3 Entendendo Módulos e npm

Para construir aplicações Node.js de forma eficaz, é essencial entender o conceito de módulos e o uso do npm (Node Package Manager). Neste tópico, vamos explorar como os módulos funcionam em Node.js, como usar o npm para gerenciar dependências e como isso se aplica ao nosso projeto de Aplicação de Restaurante. 

7.3.1 O que são Módulos?

Em Node.js, um módulo é simplesmente um arquivo JavaScript que pode exportar funcionalidades, como objetos, funções ou variáveis, para serem usados em outros arquivos. Módulos ajudam a organizar o código, tornando-o mais modular e reutilizável.

Exemplo de Módulo Simples:

Arquivo `math.js`:

```
// Funções de adição e subtração
function add(a, b) {
    return a + b;
}

function subtract(a, b) {
    return a - b;
}

// Exportando as funções
module.exports = {
    add,
    subtract
};
```

Usando o Módulo `math.js` em outro arquivo:

Arquivo `app.js`:

```
// Importando o módulo math
const math = require('./math');

console.log(math.add(2, 3)); // 5
console.log(math.subtract(5, 2)); // 3
```

7.3.2 Tipos de Módulos

Node.js suporta diferentes tipos de módulos:

- **Módulos Internos:** Módulos fornecidos pelo Node.js, como `fs` (sistema de arquivos), `http` (servidor HTTP) e `path` (trabalhar com caminhos de arquivos).
- **Módulos de Terceiros:** Módulos instalados via npm, como `express`, `mongoose`, etc.
- **Módulos Locais:** Módulos criados por você dentro do seu projeto, como `math.js` no exemplo acima.

7.3.3 Usando npm para Gerenciar Dependências

O npm é o gerenciador de pacotes padrão para Node.js. Ele permite instalar, atualizar e remover pacotes, além de gerenciar as dependências do seu projeto.

Comandos Básicos do npm:

- **Iniciar um Projeto Node.js:**

```
npm init -y
```

Isso cria um arquivo `package.json` que armazena informações sobre o projeto e suas dependências.

- **Instalar um Pacote:**

```
npm install express
```

Isso instala o pacote `express` e adiciona uma entrada para ele no arquivo `package.json`.

- **Instalar Dependências de um Projeto:**

Se você clonar um projeto Node.js que já possui um `package.json`, pode instalar todas as dependências listadas nele usando:

```
npm install
```

- **Remover um Pacote:**

```
npm uninstall express
```

Isso remove o pacote `express` do projeto e do `package.json`.

7.3.4 Estrutura de um Arquivo `package.json`

O arquivo `package.json` contém informações importantes sobre o projeto e suas dependências. Vamos dar uma olhada em um exemplo simplificado:

```
{
  "name": "api-restaurante",
  "version": "1.0.0",
  "description": "Aplicação para gerenciar pedidos de um restaurante",
  "main": "app.js",
  "scripts": {
    "start": "node app.js"
  },
  "dependencies": {
    "express": "^4.17.1",
    "ejs": "^3.1.5"
  },
  "author": "Seu Nome",
```

```
    "license": "MIT"  
}
```

- **name**: Nome do projeto.
- **version**: Versão do projeto.
- **description**: Descrição do projeto.
- **main**: Arquivo principal do projeto.
- **scripts**: Scripts que podem ser executados usando `npm run <script>`.
- **dependencies**: Pacotes que o projeto depende para funcionar.
- **author**: Autor do projeto.
- **license**: Licença do projeto.

Vamos praticar o uso de módulos e npm:

1. Crie um novo módulo chamado `calculadora.js` com funções de multiplicação e divisão.
 2. Crie um arquivo `app.js` para importar e usar o módulo `calculadora`.
 3. Inicialize um novo projeto Node.js e instale o pacote `express`.
 4. Adicione um script `start` no `package.json` para iniciar o servidor com `node app.js`.
-

7.5 Conceitos de MVC e Organização do Projeto

Neste tópico, vamos nos aprofundar nos conceitos de MVC (Model-View-Controller) e ver como organizar nosso projeto Aplicação de Restaurante usando esses conceitos. Vamos organizar a estrutura de pastas dentro de um diretório `src` para manter nosso projeto bem organizado. 

7.5.1 O que é MVC?

MVC é um padrão de arquitetura de software que separa a aplicação em três componentes principais:

- **Model (Modelo):** Representa os dados e a lógica de negócios da aplicação. Ele define como os dados são armazenados, manipulados e recuperados.
- **View (Visão):** Responsável pela apresentação dos dados ao usuário. Em aplicações web, isso geralmente se refere aos templates HTML.
- **Controller (Controlador):** Intermediário entre o Model e a View. Ele recebe as solicitações do usuário, processa-as (geralmente interagindo com o Model) e retorna os dados apropriados à View.

Essa separação ajuda a organizar o código, facilita a manutenção e melhora a escalabilidade da aplicação.

7.5.2 Estrutura de Pastas do Projeto

Vamos organizar nosso projeto seguindo o padrão MVC. Aqui está a estrutura de pastas que vamos usar dentro do diretório `src`:

```
api-restaurante/
├── src/
│   ├── config/
│   │   └── database.js
│   ├── controllers/
│   │   └── pedidoController.js
│   ├── models/
│   │   └── pedido.js
│   ├── routes/
│   │   └── pedidos.js
│   └── views/
│       └── pedidos/
└── app.js
├── .env
└── package.json
└── server.js
```

- **src/**: Diretório principal onde todo o código-fonte da aplicação está localizado.
 - **config/**: Contém configurações da aplicação, como a configuração do banco de dados.
 - **controllers/**: Contém os controladores que gerenciam a lógica de negócios.
 - **models/**: Contém os modelos de dados.
 - **routes/**: Contém as definições de rotas da aplicação.
 - **views/**: Contém os templates EJS para renderizar HTML.
 - **app.js**: Arquivo principal que configura o aplicativo Express.

7.5.3 Detalhando Cada Componente

1. Configuração (config):

O diretório

`config` contém arquivos de configuração, como a configuração do banco de dados. Isso ajuda a centralizar as configurações e facilita a manutenção.

2. Controladores (controllers):

Os controladores são responsáveis por lidar com as solicitações HTTP, processar os dados (geralmente interagindo com os modelos) e retornar as respostas apropriadas. Cada controlador é normalmente associado a um modelo específico.

3. Modelos (models):

Os modelos representam a estrutura dos dados e contêm a lógica de negócios associada. Eles são responsáveis por interagir com o banco de dados, recuperando e manipulando os dados conforme necessário.

4. Público (public):

O diretório

`public` contém arquivos estáticos, como CSS, JavaScript e imagens. Esses arquivos são acessíveis diretamente pelo cliente (navegador).

5. Rotas (routes):

As rotas definem os endpoints da aplicação e mapeiam as solicitações HTTP para

os métodos dos controladores. Isso ajuda a separar a lógica de roteamento da lógica de negócios.

6. Visões (views):

As visões são responsáveis pela apresentação dos dados ao usuário. Usaremos EJS como motor de templates para gerar HTML dinâmico e Bootstrap para estilizar as páginas.

7.6 Utilizando ORM com Sequelize e MySQL

Neste tópico, vamos entender o que é um ORM (Object-Relational Mapping), como instalar e configurar o Sequelize para trabalhar com MySQL e como criar o modelo de pedido para nossa Aplicação de Restaurante. 

7.6.1 O que é um ORM?

ORM (Object-Relational Mapping) é uma técnica de programação que permite converter dados entre sistemas incompatíveis usando linguagens de programação orientadas a objetos. Em outras palavras, um ORM permite que você interaja com o banco de dados usando objetos em vez de escrever consultas SQL diretamente. Isso facilita o desenvolvimento e manutenção do código.

Vantagens do ORM:

- **Abstração:** Simplifica as operações com o banco de dados, permitindo que os desenvolvedores foquem na lógica de negócios.
- **Produtividade:** Reduz a quantidade de código boilerplate necessário para interagir com o banco de dados.
- **Portabilidade:** Facilita a troca de bancos de dados sem modificar o código da aplicação.

7.6.2 Instalando e Configurando o Sequelize

Vamos usar o Sequelize como nosso ORM para interagir com o banco de dados MySQL.

Instale as dependências necessárias para o projeto:

```
npm install express sequelize mysql2 dotenv body-parser
```

7.6.3 Configurando a Conexão com o Banco de Dados

Vamos criar o arquivo de configuração do banco de dados `src/config/database.js`.

Crie o arquivo `src/config/database.js`:

```
const { Sequelize } = require("sequelize");
require("dotenv").config();

const sequelize = new Sequelize(
  process.env.DB_NAME,
  process.env.DB_USER,
  process.env.DB_PASS,
  {
    host: process.env.DB_HOST,
    dialect: "mysql",
  }
);

sequelize
  .authenticate()
  .then(() => {
    console.log("Conectado ao MySQL com sucesso!");
    return sequelize.sync();
  })
  .catch((err) => {
    console.error("Não foi possível conectar ao MySQL:", err);
  });

```

```
module.exports = sequelize;
```

Como rodar o banco com Docker:

Iniciando conteiner:

```
docker run --name meu_mysql -e MYSQL_ROOT_PASSWORD=minha_senha -p 3306:3306 -d mysql:latest
```

Conectando e criando banco:

1. Entre no container do MySQL:

```
docker exec -it meu_mysql bash
```

2. Acesse o MySQL:

```
mysql -u root -p
```

3. Crie o banco de dados:

```
CREATE DATABASE api_restaurante;
```

Configure o seu arquivo `.env`:

```
DB_NAME=api_restaurante  
DB_USER=root  
DB_PASS=minha_senha  
DB_HOST=127.0.0.1  
DB_PORT=3306
```

7.6.4 Criando o Modelo de Pedido

Vamos criar o modelo de dados para os pedidos usando Sequelize.

Crie o arquivo `src/models/pedido.js`:

```
const { DataTypes } = require('sequelize');
const sequelize = require('../config/database');

const Pedido = sequelize.define('Pedido', {
  cliente: {
    type: DataTypes.STRING,
    allowNull: false
  },
  itens: {
    type: DataTypes.JSON,
    allowNull: false
  },
  total: {
    type: DataTypes.FLOAT,
    allowNull: false
  },
  status: {
    type: DataTypes.STRING,
    defaultValue: 'pendente'
  }
}, {
  tableName: 'pedidos'
});

module.exports = Pedido;
```

- **Importação do Sequelize e DataTypes:** Importa as classes necessárias do Sequelize para definir o modelo.
- **Definição do Modelo `Pedido`:** Cria o modelo `Pedido` com os campos `cliente`, `itens`, `total` e `status`.

- **Exportação do Modelo:** Exporta o modelo `Pedido` para ser usado em outras partes da aplicação.

Sincronize o modelo com o banco de dados no arquivo `src/app.js`:

```
const express = require('express');
const bodyParser = require("body-parser");

const sequelize = require('./config/database');

const app = express();
const port = 3000;

app.use(bodyParser.urlencoded({ extended: false }));
app.use(bodyParser.json());

app.set('view engine', 'ejs');
app.set("views", "src/views");

sequelize.sync()
  .then(() => {
    console.log('Banco de dados sincronizado');
    app.listen(port, () => {
      console.log(`Servidor rodando em <http://localhost>:${port}`);
    });
  })
  .catch(err => console.error('Erro ao sincronizar banco de dados:', err));
```

7.6.5 Rodando o projeto

Adicione o seguinte script no seu arquivo `package.json`:

```
{  
  "dependencies": {  
    "dotenv": "^16.4.5",  
    "express": "^4.19.2",  
    "mysql2": "^3.11.0",  
    "sequelize": "^6.37.3"  
  },  
  "scripts": {  
    "start": "node src/app.js"  
  }  
}
```

Agora, é só abrir o terminal e rodar:

```
npm run start
```



Exercícios Práticos

Para praticar a configuração do Sequelize e a criação de um modelo, siga estas etapas:

1. **Instale as dependências Sequelize, MySQL2 e dotenv.**
2. **Configure o arquivo `.env` com as credenciais do banco de dados.**
3. **Crie o arquivo `src/config/database.js` para configurar a conexão com o banco de dados.**
4. **Crie o modelo de pedido em `src/models/pedido.js`.**
5. **Sincronize o modelo com o banco de dados em `src/app.js`.**

7.7 Usando EJS para Renderizar Views

Neste tópico, vamos explorar o EJS (Embedded JavaScript) para renderizar as views da nossa aplicação. Vamos criar uma view para gerenciar pedidos, incluindo um formulário para criar pedidos e uma lista para visualizar os pedidos existentes, usando Bootstrap para melhorar a aparência. 

7.7.1 O que é EJS?

EJS (Embedded JavaScript) é um motor de templates para Node.js que permite gerar HTML com código JavaScript embutido. Com EJS, você pode inserir lógica de programação diretamente nas páginas HTML, facilitando a criação de conteúdo dinâmico.

Vantagens do EJS:

- **Simplicidade:** Fácil de aprender e usar.
- **Flexibilidade:** Permite misturar JavaScript com HTML.
- **Integração:** Funciona bem com Express e outros frameworks Node.js.

7.7.2 Configurando EJS

Instale o EJS como dependência:

```
npm install ejs
```

7.7.3 Criando a View de Pedidos

Vamos criar a view para gerenciar pedidos, incluindo um formulário para criar novos pedidos e uma tabela para listar os pedidos existentes.

Crie a view de pedidos em `src/views/pedidos/index.ejs`:

```
<!DOCTYPE html>
<html lang="en">
  <head>
```

```

<meta charset="UTF-8" />
<meta name="viewport" content="width=device-width, initial-scale=1.0" />
<title>Pedidos</title>
<link
    rel="stylesheet"
    href="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css"
/>
<link rel="stylesheet" href="/css/style.css" />
</head>
<body>
    <nav class="navbar navbar-expand-lg navbar-light bg-light">
        <a class="navbar-brand" href="/">Aplicação Restaurante</a>
        <div class="collapse navbar-collapse" id="navbarNav">
            <ul class="navbar-nav">
                <li class="nav-item">
                    <a class="nav-link" href="/pedidos">Pedidos</a>
                </li>
            </ul>
        </div>
    </nav>
    <div class="container">
        <div class="row mt-4">
            <div class="col-md-6">
                <h2>Pedidos</h2>
                <table class="table table-striped">
                    <thead>
                        <tr>
                            <th>ID</th>
                            <th>Cliente</th>
                            <th>Total</th>
                            <th>Status</th>
                        </tr>

```

```

</thead>
<tbody>
  <% pedidos.forEach(pedido => { %>
    <tr>
      <td><%= pedido.id %></td>
      <td><%= pedido.cliente %></td>
      <td><%= pedido.total %></td>
      <td><%= pedido.status %></td>
    </tr>
  <% }) %>
  </tbody>
</table>
</div>
<div class="col-md-6">
  <h2>Criar Pedido</h2>
  <form action="/pedidos" method="POST">
    <div class="form-group">
      <label for="cliente">Cliente:</label>
      <input
        type="text"
        class="form-control"
        id="cliente"
        name="cliente"
        required
      />
    </div>
    <div class="form-group">
      <label for="itens">Itens:</label>
      <textarea
        class="form-control"
        id="itens"
        name="itens"
        rows="3"
        required
      ></textarea>
    </div>

```

```

<div class="form-group">
  <label for="total">Total:</label>
  <input
    type="number"
    class="form-control"
    id="total"
    name="total"
    required
  />
</div>
<button type="submit" class="btn btn-primary">Criar Pedido</button>
</form>
</div>
</div>
<script src="https://code.jquery.com/jquery-3.5.1.slim.min.js"></script>
<script src="https://cdn.jsdelivr.net/npm/@popperjs/core@2.9.2/dist/umd/popper.min.js"></script>
<script src="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/js/bootstrap.min.js"></script>
<script src="/js/script.js"></script>
</body>
</html>

```



Exercícios Práticos

Para praticar a criação de views com EJS e Bootstrap, siga estas etapas:

- 1. Instale o EJS como dependência**
- 2. Configure o Express para usar EJS como motor de templates no arquivo `src/app.js`**
- 3. Crie a view de pedidos em `src/views/pedidos/index.ejs`**

7.8 Implementando Rotas e Controllers

Neste tópico, vamos explorar como criar rotas e controllers para nossa aplicação. Vamos criar o arquivo `PedidoController.js` com a lógica necessária para conectar a view com o model e depois configurar as rotas para gerenciar os pedidos. 

7.8.1 O que são Rotas e Controllers?

- **Rotas:** Definem os endpoints da aplicação e mapeiam as solicitações HTTP para os métodos dos controllers. As rotas são responsáveis por direcionar as requisições para as funções apropriadas nos controllers.
- **Controllers:** Contêm a lógica de negócios da aplicação. Eles processam as solicitações recebidas das rotas, interagem com os models (para acessar e manipular dados) e retornam as respostas apropriadas às views.

7.8.2 Criando o Controller de Pedidos

Vamos criar o arquivo `pedidoController.js` dentro do diretório `src/controllers/` para gerenciar a lógica de negócios dos pedidos.

Crie o arquivo `src/controllers/pedidoController.js` usando uma classe:

```
const Pedido = require('../models/pedido');

class PedidoController {
    // Método para obter todos os pedidos e renderizar a view
    async getAllPedidos(req, res) {
        try {
            const pedidos = await Pedido.findAll();
            res.render('pedidos/index', { title: 'Pedidos', pedidos });
        } catch (err) {
            res.status(500).json({ error: err.message });
        }
    }
}
```

```

    }
}

// Método para criar um novo pedido
async createPedido(req, res) {
  const { cliente, itens, total } = req.body;
  try {
    await Pedido.create({ cliente, itens, total });
    res.redirect('/pedidos');
  } catch (err) {
    res.status(500).json({ error: err.message });
  }
}

module.exports = new PedidoController();

```

- Importa o modelo `Pedido` para interagir com o banco de dados.
- Define a classe `PedidoController` para gerenciar a lógica de negócios dos pedidos.
- O método `getAllPedidos` obtém todos os pedidos e renderiza a view com esses dados.
- O método `createPedido` cria um novo pedido com base nos dados fornecidos no corpo da requisição.
- Exporta uma instância da classe `PedidoController` para ser usada em outras partes da aplicação.

7.8.3 Criando as Rotas de Pedidos

Vamos criar o arquivo de rotas `pedidos.js` dentro do diretório `src/routes/` para definir os endpoints da aplicação relacionados aos pedidos.

Crie o arquivo `src/routes/pedidos.js`:

```

const express = require("express");
const pedidoController = require("../controllers/pedidoController");

const router = express.Router();

router.get("/", (req, res) => pedidoController.getAllPedidos
(req, res));

router.post("/", (req, res) => pedidoController.createPedido
(req, res));

module.exports = router;

```

- Importa o módulo `express` e o controlador de pedidos.
- Cria um roteador Express.
- Define a rota GET para obter todos os pedidos usando o método `getAllPedidos` do controlador.
- Define a rota POST para criar um novo pedido usando o método `createPedido` do controlador.
- Exporta o roteador para ser utilizado em outras partes da aplicação.

7.8.4 Atualizando o Arquivo `app.js`

Vamos atualizar o arquivo `src/app.js` para usar as rotas de pedidos que acabamos de criar.

Atualize o arquivo `src/app.js`:

```

const express = require('express');
const bodyParser = require("body-parser");
const sequelize = require('./config/database');
const pedidosRouter = require('./routes/pedidos');

```

```

const app = express();
const port = 3000;

app.use(bodyParser.urlencoded({ extended: false }));
app.use(bodyParser.json());

// Configurando EJS
app.set('view engine', 'ejs');
app.set("views", "src/views");

// Definindo diretório para arquivos estáticos
app.use(express.static('src/public'));

// Usando as rotas de pedidos
app.use('/pedidos', pedidosRouter);

sequelize.sync()
  .then(() => {
    console.log('Banco de dados sincronizado');
    app.listen(port, () => {
      console.log(`Servidor rodando em <http://localhost>:${port}`);
    });
  })
  .catch(err => console.error('Erro ao sincronizar banco de dados:', err));

```

7.9 Testando o Projeto e as Rotas/Views

Neste tópico, vamos discutir como testar o projeto para garantir que as rotas e views estejam funcionando corretamente. A testagem é uma etapa crucial no desenvolvimento de software, pois ajuda a identificar e corrigir problemas antes que o projeto seja lançado.

7.9.1 Testando o Projeto Manualmente

Para realizar testes manuais, siga estas etapas:

1. Inicie o Servidor

- Certifique-se de que o servidor esteja rodando. No terminal, execute:

```
npm run start
```

2. Acesse as Rotas no Navegador

- Abra o navegador e acesse a rota principal dos pedidos:

```
http://localhost:3000/pedidos
```

- Verifique se a lista de pedidos está sendo exibida corretamente e se os dados são carregados do banco de dados.

3. Crie um Novo Pedido

- Na página de pedidos, preencha o formulário de criação de pedido com os dados necessários (cliente, itens, total) e clique no botão "Criar Pedido".
- Verifique se o novo pedido aparece na lista de pedidos após a submissão do formulário.



Exercícios Práticos

Para praticar a criação de rotas e controllers e incrementar o projeto, siga estas etapas:

1. **Crie o arquivo `src/controllers/PedidoController.js` com a lógica para gerenciar pedidos usando classes.**
2. **Crie o arquivo de rotas `src/routes/pedidos.js` para definir os endpoints relacionados aos pedidos.**
3. **Atualize o arquivo `src/app.js` para usar as rotas de pedidos.**

7.10 Melhorando o Projeto

Para tornar o projeto mais robusto e impressionante para o seu portfólio, considere implementar as seguintes melhorias:

- **Novas features:**
 - Atualização de Pedidos: Implemente uma funcionalidade para permitir a atualização dos pedidos existentes. Isso pode incluir a modificação dos itens do pedido, a quantidade, o cliente, e o status do pedido.
 - Deleção de Pedidos: Adicione a capacidade de deletar pedidos do sistema. Isso deve garantir que os pedidos possam ser removidos quando não forem mais necessários ou foram criados por engano.
 - CRUD de Restaurantes: Desenvolva um CRUD completo (Create, Read, Update, Delete) para gerenciar os dados dos restaurantes. Isso incluiria a criação de novos restaurantes, leitura de informações dos restaurantes existentes, atualização dos dados dos restaurantes e deletar restaurantes.
- **Autenticação e Autorização:**
 - Adicione funcionalidades de login e registro de usuários.
 - Implemente diferentes níveis de acesso com base em permissões de usuário (ex.: administrador, cliente).
- **Validação de Dados:**
 - Utilize bibliotecas como `Joi` ou `Validator` para validar os dados de entrada.
 - Garanta que apenas dados válidos sejam salvos no banco de dados.
- **Documentação da API:**
 - Use ferramentas como `Swagger` ou `Postman` para documentar os endpoints da API.

- Inclua exemplos de requisições e respostas para facilitar o uso da API por outros desenvolvedores.

- **Paginação e Filtros:**

- Implemente paginação para grandes listas de dados.
 - Adicione filtros de busca para facilitar a localização de pedidos específicos.
-

7.11 APIs

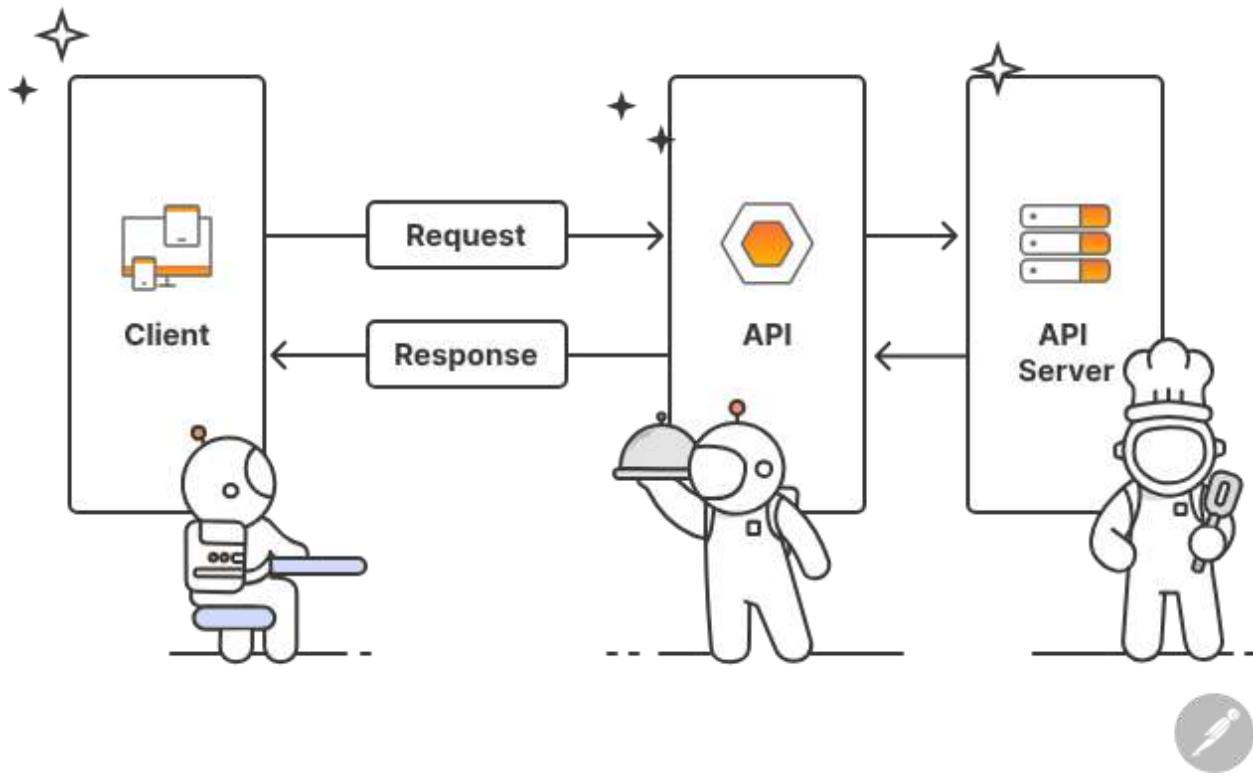
O que é uma API?

Vamos começar com uma analogia simples para entender o conceito de API. Imagine que você está em um restaurante . Quando você escolhe o que quer comer, você não vai direto para a cozinha preparar o seu prato, certo? Em vez disso, você fala com o garçom, que leva o seu pedido para a cozinha (que é o servidor). A cozinha prepara a comida (os dados) e o garçom a traz de volta para você. Essa interação entre você, o garçom e a cozinha é muito parecida com a forma como uma API funciona.

API: O Garçom da Internet

- **Cliente (Você):** Quem faz o pedido.
- **Garçom (API):** Quem leva o pedido para a cozinha e traz a comida de volta.
- **Cozinha (Servidor):** Quem prepara a comida e a entrega ao garçom.

Assim como o garçom é responsável por receber o seu pedido e trazê-lo de volta, uma API é responsável por receber solicitações de um cliente (como um navegador ou um aplicativo) e retornar os dados solicitados, processados pelo servidor.



⟳ Como Funciona uma API?

Quando você usa um aplicativo no seu celular, como um app de lista de tarefas, o aplicativo precisa se comunicar com um servidor para armazenar, atualizar ou excluir suas tarefas. Isso é feito através de uma API. A API age como um intermediário que permite que diferentes sistemas (o aplicativo e o servidor) conversem entre si.

Por exemplo, no caso de uma lista de tarefas:

- Adicionar uma tarefa:** Quando você adiciona uma nova tarefa, o aplicativo envia uma solicitação (request) ao servidor via API, usando o método **POST**. O servidor então armazena essa nova tarefa no banco de dados e retorna uma resposta (response) confirmado que a tarefa foi criada com sucesso.
- Ver todas as tarefas:** Quando você quer ver todas as suas tarefas, o aplicativo faz uma solicitação **GET** via API para o servidor, que busca todas as tarefas armazenadas e as envia de volta ao aplicativo.

3. **Atualizar uma tarefa:** Se você quiser marcar uma tarefa como concluída, o aplicativo usa o método **PUT** ou **PATCH** para atualizar essa tarefa via API.
4. **Excluir uma tarefa:** Quando você decide excluir uma tarefa, o aplicativo faz uma solicitação **DELETE** via API, e o servidor remove essa tarefa do banco de dados.



Tipos de APIs

Existem diferentes tipos de APIs, e cada uma tem sua própria maneira de funcionar. Vamos focar nos dois tipos mais comuns:

- **APIs RESTful:** Pense em uma API RESTful como um cardápio de restaurante , onde cada prato tem um endereço (URL) específico e você pode fazer diferentes pedidos (ações) como GET (pegar), POST (criar), PUT (atualizar) e DELETE (excluir).
- **APIs GraphQL:** Imagine um bufê onde você monta seu prato exatamente como quiser, escolhendo só os ingredientes (dados) que precisa. Isso é semelhante ao GraphQL, onde você pode solicitar exatamente os dados que deseja, sem receber nada a mais.



Principais Componentes de uma API

Para entender como usar uma API, é importante conhecer os componentes principais:

- **Endpoint/Rotas:** É como o endereço do restaurante . Onde você vai para fazer o pedido. Exemplo: `/tarefas` seria o endpoint para acessar as tarefas.
- **Métodos HTTP:** São as ações que você pode realizar:
 - **GET:** Pegar dados . (Ex: Ver todas as tarefas)
 - **POST:** Criar algo novo . (Ex: Adicionar uma nova tarefa)
 - **PUT/PATCH:** Atualizar algo . (Ex: Marcar uma tarefa como concluída)
 - **DELETE:** Excluir algo . (Ex: Remover uma tarefa)

- **Request (Solicitação):** É o seu pedido para o garçom (API). Contém informações sobre o que você quer.
- **Response (Resposta):** É o que você recebe de volta. O resultado do que foi processado pelo servidor.

APIs e a Web: Cliente-Servidor

A web funciona baseada no modelo cliente-servidor. O **cliente** é quem faz a solicitação, como o seu navegador ou um aplicativo. O **servidor** é quem processa essa solicitação e envia a resposta de volta ao cliente.

Imagine que você está em casa e quer pedir uma pizza 🍕. Você faz o pedido (cliente) ligando para a pizzaria (servidor). A pizzaria prepara a pizza e um entregador a leva até você. Esse ciclo é muito parecido com como uma API opera:

1. **Cliente:** Faz a solicitação (pede a pizza).
2. **Servidor:** Processa o pedido (prepara a pizza).
3. **API (Garçom/Entregador):** Leva o pedido do cliente ao servidor e a resposta de volta ao cliente (entrega a pizza).

Por que Usar APIs?

- **Interoperabilidade:** APIs permitem que diferentes sistemas e aplicações se comuniquem, como quando você usa um app de clima que obtém dados de um servidor externo.
- **Reutilização:** Uma vez que uma API é criada, ela pode ser reutilizada por diferentes aplicações e desenvolvedores, economizando tempo e esforço.
- **Modularidade:** APIs permitem que sistemas sejam divididos em partes menores, mais fáceis de gerenciar e atualizar.

Exemplo Prático: API de Lista de Tarefas

Suponha que você está construindo um aplicativo de lista de tarefas. Você poderia ter uma API com os seguintes endpoints:

1. **POST /tarefas**: Adicionar uma nova tarefa à lista.
2. **GET /tarefas**: Obter todas as tarefas.
3. **PUT /tarefas/:id**: Atualizar uma tarefa existente.
4. **DELETE /tarefas/:id**: Excluir uma tarefa da lista.

Cada um desses endpoints representa uma ação específica que o cliente pode realizar, e a API é responsável por gerenciar essas interações entre o cliente e o servidor.

Resumindo

Uma API é como o garçom que conecta o cliente à cozinha em um restaurante, facilitando a comunicação entre diferentes sistemas. Entender APIs é essencial para qualquer desenvolvedor moderno, pois elas são a base da maioria das aplicações web e móveis que usamos todos os dias.

7.12 API com Node.js (API de Tarefas)

Introdução ao Projeto

Neste tópico, vamos desenvolver uma API utilizando Node.js e Express para gerenciar uma lista de tarefas. O objetivo deste projeto é fornecer uma aplicação backend funcional que permita criar, ler, atualizar e excluir tarefas. Através deste projeto, você aprenderá a configurar um servidor Express, definir rotas, implementar controllers e conectar a API a um banco de dados. Além disso, você terá a oportunidade de praticar conceitos importantes como a organização de código, a criação de modelos de dados e a validação de entradas do usuário. Vamos começar a construir nossa API de tarefas e explorar o poder do Node.js no desenvolvimento de aplicações backend robustas e escaláveis.

Criando o Projeto Node.js

Para iniciar o projeto, vamos criar uma nova pasta e inicializar um projeto Node.js:

```
mkdir api-tarefas  
cd api-tarefas  
npm init -y
```

Instalando Dependências

Vamos instalar as dependências necessárias para o nosso projeto:

- **Express**: Framework para Node.js que facilita a criação de servidores web.
- **Sequelize**: ORM (Object-Relational Mapper) para interagir com o banco de dados.
- **MySQL2**: Driver para conectar ao banco de dados MySQL.
- **dotenv**: Para gerenciar variáveis de ambiente.

```
npm install express sequelize mysql2 dotenv
```

Criando a Estrutura de Pastas

Vamos criar a estrutura de pastas do nosso projeto:

```
mkdir -p src/{models,controllers,routes,repositories,config}
```

- **models**: Contém os modelos do Sequelize que representam as tabelas do banco de dados.
- **controllers**: Contém a lógica de negócios e as funções que serão chamadas pelas rotas.
- **routes**: Contém as definições de rotas que mapeiam URLs para os controllers.

- **repositories**: Contém a lógica de acesso aos dados, como consultas ao banco de dados.
- **config**: Contém arquivos de configuração, como a configuração do banco de dados.

Vamos criar a estrutura de pastas e arquivos finais para o nosso projeto:

```
api-tarefas/
├── node_modules/
└── src/
    ├── config/
    │   └── database.js
    ├── controllers/
    │   └── tarefaController.js
    ├── models/
    │   └── tarefa.js
    ├── repositories/
    │   └── tarefaRepository.js
    ├── routes/
    │   └── tarefas.js
    └── app.js
├── .env
└── package.json
└── README.md
```

Configurando o Projeto

Vamos criar os arquivos principais do projeto:

1. Configuração do banco de dados (`src/config/database.js`):

```
const { Sequelize } = require('sequelize');
require('dotenv').config();
```

```

const sequelize = new Sequelize(process.env.DB_NAME, process.
env.DB_USER, process.env.DB_PASS, {
  host: process.env.DB_HOST,
  dialect: 'mysql'
});

module.exports = sequelize;

```

- O código importa o módulo `Sequelize` do pacote `sequelize`.
- Utiliza o módulo `dotenv` para carregar variáveis de ambiente a partir de um arquivo `.env`.
- Cria uma nova instância do Sequelize, utilizando as variáveis de ambiente (`DB_NAME`, `DB_USER`, `DB_PASS`, `DB_HOST`) para configurar a conexão com o banco de dados MySQL.
- Define `host` e `dialect` como configurações da conexão.
- Exporta a instância do Sequelize para ser utilizada em outras partes do projeto.

2. Modelo de Tarefa (`src/models/tarefa.js`):

```

const { DataTypes } = require('sequelize');
const sequelize = require('../config/database');

const Tarefa = sequelize.define('Tarefa', {
  titulo: {
    type: DataTypes.STRING,
    allowNull: false
  },
  descricao: {
    type: DataTypes.TEXT,
    allowNull: true
  },
  concluida: {

```

```

        type: DataTypes.BOOLEAN,
        defaultValue: false
    }
});

module.exports = Tarefa;

```

- O código importa o objeto `DataTypes` do pacote `sequelize`, que é utilizado para definir os tipos de dados dos campos do modelo.
- Importa a instância `sequelize` configurada no arquivo `database.js` para conectar ao banco de dados.
- Define o modelo `Tarefa` com três campos: `titulo`, `descricao` e `concluida`.
 - `titulo`: Campo de texto obrigatório (`STRING`) que não pode ser nulo (`allowNull: false`).
 - `descricao`: Campo de texto opcional (`TEXT`) que pode ser nulo (`allowNull: true`).
 - `concluida`: Campo booleano (`BOOLEAN`) que indica se a tarefa está concluída, com valor padrão `false`.
- Exporta o modelo `Tarefa` para ser utilizado em outras partes do projeto.

3. Repositório de Tarefas (`src/repositories/tarefaRepository.js`):

```

const Tarefa = require('../models/tarefa');

class TarefaRepository {
    async getAllTarefas() {
        try {
            return await Tarefa.findAll();
        } catch (error) {
            throw new Error('Erro ao buscar todas as tarefas: ' + error.message);
        }
    }
}

```

```

    }

    async createTarefa(tarefaData) {
        try {
            return await Tarefa.create(tarefaData);
        } catch (error) {
            throw new Error('Erro ao criar nova tarefa: ' + error.message);
        }
    }

    // ... Implementar restante dos métodos (update, delete, get by id)
}

module.exports = new TarefaRepository();

```

- O arquivo importa o modelo `Tarefa` do diretório `src/models/tarefa`.
- Define a classe `TarefaRepository` para gerenciar a lógica de acesso aos dados das tarefas.
- O método `getAllTarefas` busca todas as tarefas do banco de dados utilizando o método `findAll` do Sequelize.
- O método `createTarefa` cria uma nova tarefa no banco de dados utilizando o método `create` do Sequelize.
- Em caso de erro, ambos os métodos lançam uma exceção com uma mensagem de erro específica.
- Exporta uma instância da classe `TarefaRepository` para ser utilizada em outras partes do projeto.

4. Controller de Tarefas (`src/controllers/tarefaController.js`):

```

const TarefaRepository = require('../repositories/tarefaRepository');

```

```

class TarefaController {
  async getAllTarefas(req, res) {
    try {
      const tarefas = await TarefaRepository.getAllTarefas();
      res.json(tarefas);
    } catch (error) {
      res.status(500).json({ error: error.message });
    }
  }

  async createTarefa(req, res) {
    const { titulo, descricao, concluida } = req.body;
    try {
      const novaTarefa = await TarefaRepository.createTarefa(
        { titulo, descricao, concluida });
      res.status(201).json(novaTarefa);
    } catch (error) {
      res.status(500).json({ error: error.message });
    }
  }

  // ...Implementar restante dos métodos (update, delete, get by id)
}

module.exports = new TarefaController();

```

- O código acima define um **controller** para gerenciar as tarefas na aplicação.
- Os **controllers** são responsáveis por processar as solicitações HTTP, interagir com os **repositories** e retornar as respostas apropriadas.
- Neste caso, o **TarefaController** possui dois métodos principais:
 - **getAllTarefas** : Busca todas as tarefas do banco de dados e retorna uma resposta JSON com a lista de tarefas.

- `createTarefa`: Cria uma nova tarefa com base nos dados fornecidos no corpo da requisição e retorna a tarefa criada em formato JSON.

Comparação API vs. MVC:

- **API:**
 - A API é focada em fornecer endpoints que permitem aos clientes (como navegadores ou aplicativos) interagir com os dados através de solicitações HTTP.
 - Utiliza métodos HTTP (GET, POST, PUT, DELETE) para realizar operações CRUD (Create, Read, Update, Delete).
 - As respostas são geralmente em formato JSON, facilitando a integração com diferentes tipos de clientes.
- **MVC (Model-View-Controller):**
 - No MVC, os **controllers** geralmente interagem com **views** para renderizar páginas HTML que serão exibidas ao usuário.
 - O foco é mais na renderização de interfaces de usuário completas, incluindo templates HTML e lógica de apresentação.
 - As respostas podem incluir HTML, CSS e JavaScript, além de dados JSON para interações assíncronas.

5. Rotas de Tarefas (`src/routes/tarefas.js`):

```
const express = require('express');
const tarefaController = require('../controllers/tarefaController');

const router = express.Router();

router.get('/', (req, res) => tarefaController.getAllTarefas(req, res));
router.post('/', (req, res) => tarefaController.createTarefa(req, res));
```

```
module.exports = router;
```

- Importa o módulo `express` e o `tarefaController`.
- Cria um roteador Express para gerenciar as rotas.
- Define a rota GET para obter todas as tarefas usando o método `getAllTarefas` do controlador.
- Define a rota POST para criar uma nova tarefa usando o método `createTarefa` do controlador.
- Exporta o roteador para ser utilizado em outras partes da aplicação.

6. Arquivo principal (`src/app.js`):

```
const express = require('express');
const bodyParser = require('body-parser');
const sequelize = require('./config/database');
const tarefasRouter = require('./routes/tarefas');

const app = express();
const port = 3000;

app.use(bodyParser.urlencoded({ extended: false }));
app.use(bodyParser.json());

app.use('/tarefas', tarefasRouter);

sequelize.sync()
  .then(() => {
    console.log('Banco de dados sincronizado');
    app.listen(port, () => {
      console.log(`Servidor rodando em <http://localhost:${port}>`);
    });
  });

```

```
    })
    .catch(err => console.error('Erro ao sincronizar banco de dados:', err));
```

- Importa os módulos `express`, `body-parser`, `sequelize` e as rotas de tarefas.
- Cria uma instância do Express (`app`) e define a porta na qual o servidor vai rodar (`port`).
- Configura o middleware `bodyParser` para lidar com dados enviados nas requisições HTTP.
- Define a rota `/tarefas` para usar o roteador de tarefas.
- Sincroniza o banco de dados com o Sequelize e inicia o servidor na porta especificada.
- Exibe mensagens no console indicando o sucesso ou erro na sincronização do banco de dados e no início do servidor.

7. Arquivo `.env`:

```
DB_NAME=api_tarefas
DB_USER=root
DB_PASS=minha_senha
DB_HOST=127.0.0.1
DB_PORT=3306
```

Adicionando Script no `package.json`

Para facilitar o início do projeto, vamos adicionar um script no `package.json` para iniciar o servidor. Abra o arquivo `package.json` e adicione o script `"start"` na seção `"scripts"`:

```
{
  "scripts": {
```

```
        "start": "node src/app.js"
    }
}
```

Iniciando o Projeto

Com o script adicionado, você pode iniciar o servidor simplesmente executando o seguinte comando no terminal:

```
npm start
```

Este comando vai iniciar o servidor Node.js e você poderá acessar sua API através do navegador ou de ferramentas como Insomnia.

Testando Requisições no Insomnia

Para testar as requisições da sua API, você pode utilizar uma ferramenta como o Insomnia. Aqui estão alguns passos para você começar a testar suas requisições:

1. Instalar o Insomnia:

- Baixe e instale o Insomnia a partir do site oficial.

2. Criar um novo Workspace:

- Abra o Insomnia e crie um novo Workspace para organizar suas requisições.

3. Adicionar uma nova Request:

- Clique em "New Request" e dê um nome para sua requisição.
- Selecione o método HTTP apropriado (GET, POST, PUT, DELETE).

4. Configurar a URL da API:

- Insira a URL do endpoint que você deseja testar. Por exemplo, para obter todas as tarefas, você pode usar `http://localhost:3000/tarefas`.

5. Adicionar Dados (se necessário):

- Se você estiver fazendo uma requisição POST ou PUT, adicione os dados necessários no corpo da requisição em formato JSON.
- Exemplo de corpo para criar uma nova tarefa:

```
{  
  "titulo": "Estudar Node.js",  
  "descricao": "Completar o módulo de API",  
  "concluida": false  
}
```

6. Enviar a Requisição:

- Clique em "Send" para enviar a requisição e verificar a resposta do servidor.
- Veja os resultados na seção de resposta do Insomnia.

7. Salvar e Reutilizar:

- Salve suas requisições para reutilizá-las posteriormente. Isso facilita o teste e a validação contínua da sua API.

Melhorias Sugeridas para o Projeto

- **Autenticação e Autorização:**
 - Implemente autenticação de usuários utilizando tokens JWT.
 - Adicione regras de autorização para garantir que apenas usuários autorizados possam acessar ou modificar determinadas tarefas.
- **Rotas de atualização e deleção:**
 - Implemente as rotas de atualização de tarefas e deleção de tarefas, usando os métodos HTTP PUT e DELETE.
- **Validação de Dados:**
 - Utilize bibliotecas como Joi ou Validator para validar dados de entrada.
 - Garanta que apenas dados válidos sejam salvos no banco de dados.

- **Tratamento de Erros:**
 - Crie uma camada de middleware para tratamento de erros.
 - Garanta que mensagens de erro amigáveis sejam retornadas ao cliente em caso de falha.
- **Paginação e Filtros:**
 - Implemente paginação para grandes listas de dados.
 - Adicione filtros de busca para facilitar a localização de tarefas específicas.
- **Documentação da API:**
 - Use ferramentas como Swagger ou Postman para documentar os endpoints da API.
 - Inclua exemplos de requisições e respostas para facilitar o uso da API por outros desenvolvedores.

módulo 8

react

Crie interfaces de usuário interativas
e eficientes com React.



09 tópicos neste módulo

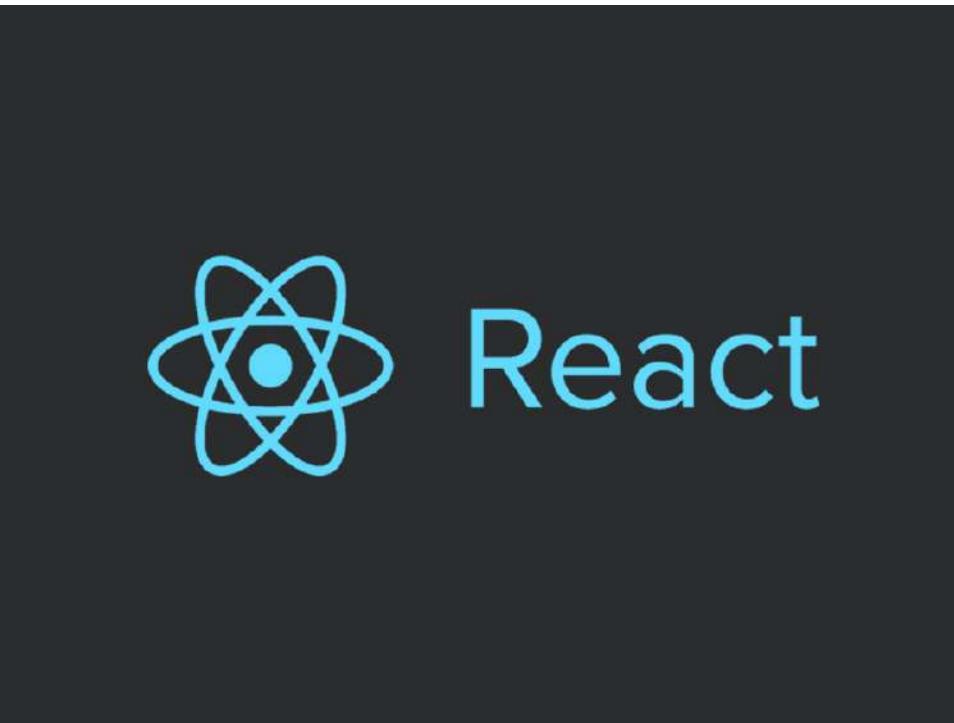
Módulo 8: React

Neste capítulo, vamos explorar o React, uma das bibliotecas JavaScript mais populares para a construção de interfaces de usuário interativas e dinâmicas. Abordaremos desde a introdução ao React, passando pelos conceitos fundamentais como componentes, estado, props e JSX, até a configuração do ambiente de desenvolvimento com Vite. Além disso, discutiremos a renderização de listas e a importância das chaves, bem como a criação e utilização de componentes reutilizáveis. Ao final, você estará apto a desenvolver uma aplicação de gerenciamento de tarefas completa, utilizando as melhores práticas de desenvolvimento com React.

8.1 Introdução ao React

Bem-vindo ao capítulo sobre React!

React é uma das bibliotecas mais populares para a construção de interfaces de usuário interativas e dinâmicas na web. Criado pelo Facebook, ele revolucionou a forma como desenvolvemos aplicações front-end, proporcionando uma abordagem declarativa e baseada em componentes para a construção de interfaces complexas.



8.1.1 O que é React?

React é uma biblioteca JavaScript **open-source** focada em construir interfaces de usuário (UI). Ele permite que você crie componentes reutilizáveis que podem ser compostos para criar interfaces de usuário sofisticadas. React adota uma abordagem baseada em componentes, onde cada parte da interface pode ser encapsulada em um pequenos blocos de código reutilizáveis, tornando o desenvolvimento mais modular e fácil de gerenciar.

8.1.2 Por que usar React?

- **Componentização:** No React, a UI é dividida em componentes, pequenas peças de código reutilizáveis que juntas formam uma aplicação completa. Isso promove a **reutilização e manutenção** de código, tornando mais fácil adicionar novas funcionalidades ou modificar as existentes. É como poder criar suas próprias tags HTML, com seus atributos e funcionalidades próprias para usar da maneira que quiser.

- **Virtual DOM:** React utiliza uma árvore de objetos leve conhecida como **Virtual DOM**. Quando o estado de um componente muda, o Virtual DOM cria uma nova árvore de elementos e compara com a anterior, atualizando apenas as partes da página que realmente mudaram. Isso resulta em uma performance muito melhor em comparação com a manipulação direta do DOM.
- **Fluxo de dados unidirecional:** O React adota um fluxo de dados unidirecional, onde os dados fluem de cima para baixo na árvore de componentes. Isso simplifica o entendimento do estado da aplicação e torna a depuração de erros mais intuitiva.
- **Comunidade e ecossistema:** O React tem uma enorme comunidade de desenvolvedores e um ecossistema rico, incluindo ferramentas como **React Router** para gerenciamento de rotas, **Redux** para gerenciamento de estado global, e muitas outras bibliotecas que podem ser facilmente integradas.

8.1.3 Conceitos Fundamentais

- **Componentes:** A base de tudo no React. Um componente pode ser uma função ou uma classe que retorna um pedaço de UI. Eles podem receber **props** (propriedades) e manter **estado** (state) para gerenciar informações dinâmicas.
- **JSX:** React utiliza JSX, uma extensão de sintaxe que permite escrever código similar a HTML dentro de arquivos JavaScript. JSX é transformado em chamadas de função JavaScript que criam os elementos do React.
- **Props:** São as propriedades que você passa para os componentes. Elas são como os argumentos de uma função e permitem que os componentes sejam configurados ou personalizados.
- **State:** O estado de um componente é um objeto que contém informações que podem mudar ao longo do tempo. Quando o estado de um componente muda, o React automaticamente atualiza a UI para refletir essa mudança.

8.1.4 Projeto: Aplicação de Gerenciamento de Tarefas

Neste capítulo, vamos construir uma **Aplicação de Gerenciamento de Tarefas**. Durante o desenvolvimento deste projeto, vamos explorar os conceitos de React em profundidade, começando com a configuração do ambiente, passando pela criação de componentes, gerenciamento de estado, manipulação de eventos, e muito mais.

A ideia é que você comprehenda como estruturar uma aplicação React desde o início, aprendendo as melhores práticas e adquirindo as habilidades necessárias para criar aplicações modernas e interativas.

O que você aprenderá ao final deste capítulo:

1. Como configurar um ambiente de desenvolvimento React.
2. Os fundamentos de JSX e componentes React.
3. Como gerenciar estado e props para criar UIs dinâmicas.
4. A importância do ciclo de vida dos componentes e hooks.
5. Como dividir uma aplicação em componentes reutilizáveis.
6. Melhores práticas para estruturação e manutenção de projetos React.

Este será um passo importante na sua jornada como desenvolvedor full-stack, permitindo que você construa interfaces de usuário poderosas e escaláveis.

8.2 Configurando o Ambiente de Desenvolvimento com Vite

Antes de começarmos a construir nossa **Aplicação de Gerenciamento de Tarefas** com React, precisamos configurar o ambiente de desenvolvimento. Vamos usar o **Vite**, uma ferramenta de build moderna e ultrarrápida que é ótima para iniciar projetos React.

Por que Vite?

- **Velocidade:** Vite é incrivelmente rápido tanto na inicialização do projeto quanto no processo de hot-reloading.
- **Configuração mínima:** Vite oferece uma configuração mínima, permitindo que você comece a desenvolver rapidamente sem se preocupar com muitas configurações.
- **Supporte moderno:** Ele suporta as mais recentes versões do ES6+ e tem uma integração perfeita com React.

Passo 1: Instalando o Node.js

Antes de começar, você precisa ter o Node.js instalado no seu sistema. Se ainda não o tem, baixe e instale o Node.js a partir do site oficial: [Node.js](#).

Você pode verificar se o Node.js está instalado corretamente rodando o seguinte comando no terminal:

```
node -v
```

Isso deve retornar a versão instalada do Node.js.

Passo 2: Criando o Projeto com Vite

Agora vamos criar o projeto React usando o Vite.

1. **Abra o terminal e navegue até o diretório onde deseja criar o projeto.**
2. **Execute o seguinte comando para criar um novo projeto React com Vite:**

```
npm create vite@latest meu-gerenciador-de-tarefas --template react
```

Aqui estamos usando o `npm create` para inicializar um novo projeto com Vite, especificando o template para React.

3. **Navegue até o diretório do projeto:**

```
cd meu-gerenciador-de-tarefas
```

4. Instale as dependências do projeto:

```
npm install
```

Isso instalará todas as dependências necessárias que vêm com o template React do Vite.

Passo 3: Estrutura Básica do Projeto

Após a instalação, a estrutura do projeto será algo como:

```
meu-gerenciador-de-tarefas/
├── node_modules/
├── public/
│   └── index.html
└── src/
    ├── assets/
    ├── components/
    ├── App.css
    ├── App.jsx
    ├── index.css
    └── main.jsx
    ├── .gitignore
    ├── index.html
    ├── package.json
    └── vite.config.js
```

- `public/`: Contém arquivos públicos como o `index.html`.
- `src/`: Contém o código-fonte da aplicação React.
 - `components/`: Diretório onde armazenaremos os componentes React.
 - `App.jsx`: Componente principal da aplicação.

- `main.jsx`: Ponto de entrada do React.
- `vite.config.js`: Arquivo de configuração do Vite.

Passo 4: Rodando o Servidor de Desenvolvimento

Agora que o projeto está configurado, vamos rodar o servidor de desenvolvimento para ver a aplicação em ação.

1. No terminal, execute o comando:

```
npm run dev
```

2. Abra o navegador e acesse o endereço:

```
<http://localhost:5173>
```

Você deve ver a página inicial padrão do Vite com React, confirmando que tudo está funcionando corretamente.

Passo 5: Configuração Adicional (Opcional)

- **EditorConfig**: Configure um arquivo `.editorconfig` para garantir a consistência do estilo de código entre diferentes editores.
- **ESLint**: Configure o ESLint para manter a qualidade do código e detectar problemas durante o desenvolvimento.
- **Prettier**: Integre o Prettier para formatar o código automaticamente de acordo com padrões definidos.

Exemplo de `.editorconfig`:

```
root = true  
  
[*]
```

```
indent_style = space
indent_size = 2
end_of_line = lf
charset = utf-8
trim_trailing_whitespace = true
insert_final_newline = true
```

8.3: Entendendo o JSX

Vamos começar a nossa jornada com React mergulhando no JSX, que é a base para construir interfaces no React. Você vai aprender o que é o JSX, como ele funciona e, ao longo deste capítulo, vamos começar a customizar o projeto padrão que criamos com o Vite, preparando-o para a nossa Aplicação de Gerenciamento de Tarefas.

8.3.1 O que é JSX?

JSX (JavaScript XML) é uma extensão de sintaxe para JavaScript que permite escrever código que se parece com HTML, mas com superpoderes! 

Em vez de usar `createElement` ou outras chamadas de API para criar elementos DOM, o JSX permite que você escreva elementos de forma declarativa, usando uma sintaxe semelhante ao HTML. O React usa esse código para criar e atualizar a interface do usuário de forma eficiente.

Por exemplo, com JSX, em vez de escrever algo como:

```
const element = React.createElement('h1', null, 'Olá, mundo');
```

Você pode escrever:

```
const element = <h1>Olá, mundo!</h1>;
```

Essa sintaxe é muito mais simples e legível, especialmente quando você está construindo interfaces complexas.

8.3.2 Como o JSX Funciona?

Embora o JSX pareça HTML, ele é transformado em código JavaScript puro por trás das cenas. Quando você escreve `<h1>Olá, mundo!</h1>`, o React compila isso para:

```
const element = React.createElement('h1', null, 'Olá, mundo!');
```

Isso significa que você pode usar todas as funcionalidades do JavaScript dentro do JSX, como variáveis, loops e condicionais. Tudo isso torna o JSX uma ferramenta poderosa para construir UIs dinâmicas.

8.3.3 Vamos Customizar o Projeto Padrão do Vite!

Agora que você tem uma ideia básica do que é o JSX, vamos começar a mexer no nosso projeto React criado com o Vite. Primeiro, abra o projeto no seu editor de código favorito.

8.3.4 Limpando o Código Inicial

O projeto criado pelo Vite vem com algum código padrão que podemos simplificar para focar no que é realmente importante. Vamos começar limpando o conteúdo do arquivo `App.jsx`.

Passo 1: Abra o arquivo `App.jsx` no editor.

Passo 2: Apague todo o conteúdo dentro da função `App` e substitua por algo mais simples, como:

```
import './App.css'

function App() {
  return (
    <div>
      <h1>Gerenciador de Tarefas <img alt="pencil icon" style={{vertical-align: middle}}/></h1>
      <p>Bem-vindo ao seu gerenciador de tarefas. Vamos começar a organizar seu dia!</p>
    </div>
  );
}

export default App;
```

Explicação:

Aqui, estamos usando o JSX para criar uma estrutura básica da nossa aplicação. Estamos declarando um título e um parágrafo dentro de uma `div`. Isso será o ponto de partida visual do nosso Gerenciador de Tarefas.

8.3.5 Estilizando a Página

Vamos adicionar um pouco de estilo para deixar nossa aplicação mais agradável. Abra o arquivo `App.css` e adicione o seguinte código:

```
div {
  text-align: center;
  margin-top: 50px;
}

h1 {
  font-size: 2.5rem;
  color: #4CAF50;
}
```

```
p {  
    font-size: 1.2rem;  
    color: #555;  
}
```

Explicação:

Aqui, centralizamos o conteúdo da página e ajustamos o estilo do título e do parágrafo. Isso dá um toque mais profissional à nossa aplicação.

O Que Vem a Seguir? ➔

Neste tópico, você aprendeu o que é o JSX e começou a customizar o projeto React criado com o Vite. No próximo tópico, vamos explorar como usar **componentes** em React, que são blocos de construção reutilizáveis para sua interface. Vamos continuar desenvolvendo nosso Gerenciador de Tarefas, tornando-o cada vez mais funcional e interativo.

8.4: Listas e Chaves 📁

Quando estamos construindo interfaces dinâmicas, como nossa Aplicação de Gerenciamento de Tarefas, é comum precisar exibir múltiplos itens, como uma lista de tarefas. Neste tópico, vamos aprender como renderizar listas de forma eficiente usando **chaves** (keys) para ajudar o React a gerenciar essas listas.

8.4.1 Renderizando Listas no React 📋

Quando você tem uma coleção de itens, como um array de tarefas, e quer exibi-los em uma lista, o JSX facilita isso utilizando a função `map()`. Vamos criar uma lista de tarefas fictícias e renderizá-las no nosso componente.

8.4.2 Criando uma Lista de Tarefas 🔪

Primeiro, vamos adicionar algumas tarefas fictícias diretamente no código para ilustrar o conceito:

```
import './App.css'

function App() {
  const tarefas = ['Estudar React', 'Fazer exercícios', 'Ler um livro'];

  return (
    <div>
      <h1>Gerenciador de Tarefas <img alt="pencil icon" style={{verticalAlign: 'middle'}}/></h1>
      <ul>
        {tarefas.map((tarefa, index) => (
          <li key={index}>{tarefa}</li>
        ))}
      </ul>
    </div>
  );
}

export default App;
```

Explicação:

- `tarefas.map()` : Aqui, estamos usando o método `map()` para iterar sobre cada item do array `tarefas`. Para cada tarefa, criamos um elemento `` (item de lista) que será renderizado na tela.
- `key={index}` : A propriedade `key` é muito importante em listas. Ela ajuda o React a identificar quais itens foram alterados, adicionados ou removidos. Neste caso, usamos o índice do array (`index`) como chave.

8.4.3 Entendendo a Importância das Chaves 🔑

O React utiliza a propriedade `key` para otimizar o processo de renderização, garantindo que cada item da lista tenha uma identidade única. Isso ajuda o React a determinar rapidamente quais itens precisam ser atualizados, evitando renderizações desnecessárias e melhorando a performance da aplicação.

Por que não usar o índice como chave?

Embora no exemplo acima tenhamos usado o índice como chave, essa prática não é recomendada em aplicações reais onde a lista pode ser reordenada ou itens podem ser removidos/adicionados. Em casos mais complexos, o uso de índices pode levar a bugs difíceis de rastrear. O ideal é usar um identificador único associado a cada item, como um ID.

8.4.4 Exemplo com IDs únicos:

```
import './App.css'

function App() {
  const tarefas = [
    { id: 1, nome: 'Estudar React' },
    { id: 2, nome: 'Fazer exercícios' },
    { id: 3, nome: 'Ler um livro' },
  ];

  return (
    <div>
      <h1>Gerenciador de Tarefas <img alt="pencil icon" /></h1>
      <ul>
        {tarefas.map((tarefa) => (
          <li key={tarefa.id}>{tarefa.nome}</li>
        )))
      </ul>
    </div>
  );
}
```

```
export default App;
```

Explicação:

- **IDs Únicos**: Cada tarefa agora possui um `id` único, que usamos como `key` na renderização da lista. Isso é mais seguro e recomendado para garantir que o React sempre saiba exatamente qual item está sendo manipulado.

8.4.5 Customizando o Estilo da Lista 🎨

Vamos adicionar um pouco de estilo à nossa lista para que ela fique mais visualmente agradável:

```
ul {  
  list-style-type: none;  
  padding: 0;  
}  
  
li {  
  background: #f0f0f0;  
  margin: 5px 0;  
  padding: 10px;  
  border-radius: 5px;  
  font-size: 1.2rem;  
  color: #333;  
}
```

Explicação:

- `ul { list-style-type: none; }`: Removemos os marcadores padrão dos itens da lista (``).
- `li { background: #f0f0f0; }`: Adicionamos um fundo cinza claro e um espaçoamento entre os itens para que eles fiquem visualmente organizados.

8.5: Componentes e Props

Chegamos a um dos conceitos mais importantes e poderosos do React:

Componentes e Props. Entender como criar e utilizar componentes de forma eficiente é fundamental para construir aplicações React modulares, reutilizáveis e fáceis de manter. Neste tópico, vamos mergulhar fundo nesse conceito e continuar desenvolvendo nossa **Aplicação de Gerenciamento de Tarefas**.

8.5.1 O que são Componentes?

No React, **componentes** são como blocos de construção. Cada componente é uma função ou classe que retorna um pedaço da interface de usuário (UI). Eles permitem dividir a UI em partes independentes e reutilizáveis, tornando o código mais organizado e fácil de manter.

Imagine que cada parte da sua interface como o cabeçalho, a lista de tarefas, o rodapé pode ser um componente separado. Isso torna mais fácil modificar, reutilizar e até testar essas partes da interface.

8.5.2 Criando o Primeiro Componente

Vamos começar criando um componente simples chamado `Tarefa`, que será responsável por renderizar uma única tarefa na nossa lista.

Passo 1: Crie um novo arquivo chamado `Tarefa.jsx` na pasta `src/components`:

```
function Tarefa({ nome }) {
  return (
    <li>{nome}</li>
  );
}

export default Tarefa;
```

Explicação:

- `function Tarefa({ nome }) { ... }`: Esse é um componente funcional que recebe `props` como argumento. Aqui, estamos desestruturando `nome` diretamente das `props` (falaremos mais sobre isso em breve).
- `{nome}`: O componente retorna um elemento de lista (``) que contém o nome da tarefa.

8.5.3 O Que São Props?

Props (propriedades) são a forma como passamos dados de um componente pai para um componente filho. Elas são imutáveis, o que significa que, uma vez que o valor é passado para o componente filho, ele não pode ser alterado pelo próprio componente filho.

No exemplo acima, passamos a prop `nome` para o componente `Tarefa`, que a utiliza para exibir o nome da tarefa.

8.5.4 Usando o Componente Tarefa na Lista

Agora que temos o componente `Tarefa`, vamos usá-lo no nosso componente principal (`App.jsx`) para renderizar a lista de tarefas.

Passo 2: Abra o arquivo `App.jsx` e modifique-o para utilizar o novo componente `Tarefa`:

```
import './App.css'
import React from 'react';
import Tarefa from './components/Tarefa';

function App() {
  const tarefas = [
    { id: 1, nome: 'Estudar React' },
    { id: 2, nome: 'Fazer exercícios' },
    { id: 3, nome: 'Ler um livro' },
  ];
}
```

```

    return (
      <div>
        <h1>Gerenciador de Tarefas <img alt="pencil icon" style={{vertical-align: middle}}/></h1>
        <ul>
          {tarefas.map((tarefa) => (
            <Tarefa key={tarefa.id} nome={tarefa.nome} />
          )))
        </ul>
      </div>
    );
}

export default App;

```

Explicação:

- `<Tarefa key={tarefa.id} nome={tarefa.nome} />`: Aqui, estamos usando o componente `Tarefa` para cada item da lista. Estamos passando a prop `nome` com o valor de `tarefa.nome`. A `key` continua sendo utilizada para ajudar o React a identificar cada item da forma única.

8.5.5 Reutilizando Componentes

Uma das grandes vantagens de usar componentes no React é a **reutilização**. Imagine que você precise exibir a mesma tarefa em diferentes partes da aplicação, mas com estilos diferentes. Com componentes, você pode facilmente reutilizar o mesmo código em vários lugares, apenas alterando as `props` conforme necessário.

Vamos adicionar um novo componente para o título da lista de tarefas, demonstrando como componentes podem ser reutilizados.

Passo 3: Crie um novo componente chamado `Titulo.jsx`:

```

function Titulo({ texto }) {
  return (

```

```

        <h2>{texto}</h2>
    );
}

export default Titulo;

```

Passo 4: Utilize o componente `Titulo` em `App.jsx`:

```

import './App.css'
import React from 'react';
import Tarefa from './components/Tarefa';
import Titulo from './components/Titulo';

function App() {
  const tarefas = [
    { id: 1, nome: 'Estudar React' },
    { id: 2, nome: 'Fazer exercícios' },
    { id: 3, nome: 'Ler um livro' },
  ];

  return (
    <div>
      <h1>Gerenciador de Tarefas <img alt="pencil icon" style={{ verticalAlign: 'middle' }} /></h1>
      <Titulo texto="Lista de Tarefas" />
      <ul>
        {tarefas.map((tarefa) => (
          <Tarefa key={tarefa.id} nome={tarefa.nome} />
        )));
      </ul>
    </div>
  );
}

export default App;

```

Explicação:

- `<Titulo texto="Lista de Tarefas" />`: Criamos um componente `Titulo` que recebe um texto como prop e o renderiza dentro de um elemento `<h2>`. Isso demonstra como os componentes podem ser reutilizados e customizados de acordo com a necessidade.
-

8.6: Estado e Ciclo de Vida

Agora que você já entende como criar componentes e passar dados entre eles usando **props**, é hora de aprender sobre o **estado** (state) e o **ciclo de vida** dos componentes em React. Esses conceitos são fundamentais para criar aplicações dinâmicas e interativas, como o nosso **Gerenciador de Tarefas**.

8.6.1 O que é Estado (State)?

No React, **estado** é um objeto que armazena informações sobre o componente e pode ser alterado ao longo do tempo. Diferente das **props**, que são imutáveis e passadas de um componente pai para um componente filho, o **estado** é gerenciado internamente por cada componente e pode ser atualizado dinamicamente.

Quando o estado de um componente muda, o React automaticamente re-renderiza o componente para refletir essas mudanças na interface. Isso é o que torna as interfaces em React tão reativas e dinâmicas!

8.6.2 Adicionando Estado ao Componente

Vamos adicionar um estado ao nosso componente `App` para gerenciar as tarefas. Usaremos o hook `useState` para criar e manipular o estado no React.

Passo 1: No arquivo `App.jsx`, importe `useState` e crie o estado para as tarefas:

```
import './App.css';
import React, { useState } from 'react';
```

```

import Tarefa from './components/Tarefa';
import Titulo from './components/Titulo';

function App() {
  const [tarefas, setTarefas] = useState([
    { id: 1, nome: 'Estudar React' },
    { id: 2, nome: 'Fazer exercícios' },
    { id: 3, nome: 'Ler um livro' },
  ]);

  return (
    <div>
      <h1>Gerenciador de Tarefas <img alt="pencil icon" style={{ verticalAlign: 'middle' }} /></h1>
      <Titulo texto="Lista de Tarefas" />
      <ul>
        {tarefas.map((tarefa) => (
          <Tarefa key={tarefa.id} nome={tarefa.nome} />
        ))}
      </ul>
    </div>
  );
}

export default App;

```

Explicação:

- `useState`: É um hook que permite adicionar estado a componentes funcionais. Ele retorna um array com duas posições: o estado atual (`tarefas`) e uma função para atualizá-lo (`setTarefas`).
- **Estado Inicial:** Passamos um array de tarefas como estado inicial. Agora, podemos atualizar essa lista dinamicamente, e o React vai re-renderizar a interface automaticamente.

8.6.3 Ciclo de Vida dos Componentes 🌱

O ciclo de vida de um componente em React refere-se ao conjunto de eventos que ocorrem desde a sua criação até a sua destruição. Para componentes funcionais, usamos o hook `useEffect` para lidar com esses eventos.

Principais fases do ciclo de vida:

- 1. Montagem (Mounting):** Quando o componente é criado e inserido no DOM.
- 2. Atualização (Updating):** Quando o componente é atualizado devido a mudanças no estado ou nas props.
- 3. Desmontagem (Unmounting):** Quando o componente é removido do DOM.

8.6.4 Usando o useEffect para Gerenciar o Ciclo de Vida 🔄

Vamos ver um exemplo simples de como usar o `useEffect` para executar código durante as fases do ciclo de vida do componente.

Passo 2: Adicione o `useEffect` ao `App.jsx`:

```
import './App.css';
import React, { useState, useEffect } from 'react';
import Tarefa from './components/Tarefa';
import Titulo from './components/Titulo';

function App() {
  const [tarefas, setTarefas] = useState([
    { id: 1, nome: 'Estudar React' },
    { id: 2, nome: 'Fazer exercícios' },
    { id: 3, nome: 'Ler um livro' },
  ]);

  useEffect(() => {
    console.log('Componente montado ou atualizado!');
  }, [tarefas]);

  return (
    <div>
      <h1>{Titulo}</h1>
      <ul>
        {tarefas.map(tarefa => (
          <li>{tarefa.nome}</li>
        ))}
      </ul>
    </div>
  );
}

export default App;
```

```

<div>
  <h1>Gerenciador de Tarefas <img alt="pencil icon" style="vertical-align: middle;"/></h1>
  <Titulo texto="Lista de Tarefas" />
  <ul>
    {tarefas.map((tarefa) => (
      <Tarefa key={tarefa.id} nome={tarefa.nome} />
    )))
  </ul>
</div>
);
}

export default App;

```

Explicação:

- **useEffect**: O `useEffect` executa uma função após o componente ser renderizado. No exemplo acima, ele está observando o estado `tarefas`. Sempre que o estado `tarefas` mudar, o código dentro do `useEffect` será executado.
- **Dependências**: O array `[tarefas]` é passado como uma lista de dependências. Isso significa que o efeito só será executado quando `tarefas` mudar. Se você deixar o array vazio (`[]`), o efeito será executado apenas uma vez, quando o componente for montado.

8.6.5 Atualizando o Estado

Agora que entendemos como o estado funciona, vamos adicionar uma funcionalidade básica para adicionar novas tarefas.

Passo 3: Adicione um formulário para inserir novas tarefas:

```

import './App.css'
import React, { useState } from 'react';
import Tarefa from './components/Tarefa';
import Titulo from './components/Titulo';

```

```

function App() {
  const [tarefas, setTarefas] = useState([
    { id: 1, nome: 'Estudar React' },
    { id: 2, nome: 'Fazer exercícios' },
    { id: 3, nome: 'Ler um livro' },
  ]);

  const [novaTarefa, setNovaTarefa] = useState('');

  const adicionarTarefa = () => {
    const nova = { id: tarefas.length + 1, nome: novaTarefa };
    setTarefas([...tarefas, nova]);
    setNovaTarefa('');
  };

  return (
    <div>
      <h1>Gerenciador de Tarefas <img alt="pencil icon" style={{ verticalAlign: 'middle' }} /></h1>
      <h2>Lista de Tarefas</h2>
      <ul>
        {tarefas.map((tarefa) => (
          <Tarefa key={tarefa.id} nome={tarefa.nome} />
        ))}
      </ul>
      <input
        type="text"
        value={novaTarefa}
        onChange={(e) => setNovaTarefa(e.target.value)}
        placeholder="Adicionar nova tarefa"
      />
      <button onClick={adicionarTarefa}>Adicionar</button>
    </div>
  );
}

```

```
export default App;
```

Explicação:

- `setTarefas([...tarefas, nova])`: Aqui, estamos atualizando o estado das tarefas, adicionando uma nova tarefa ao array existente.
- `setNovaTarefa('')`: Após adicionar a tarefa, limpamos o campo de input.

Exercícios Práticos



Agora é hora de colocar a mão na massa! Aqui estão alguns exercícios para solidificar o que você aprendeu:

1. **Adicionar Remoção de Tarefas:** Adapte o código acima para permitir que o usuário remova uma tarefa da lista ao clicar em um botão "Remover" ao lado de cada tarefa.
2. **Editar Tarefas:** Implemente uma funcionalidade que permita ao usuário editar o nome de uma tarefa existente. O nome editado deve ser salvo no estado do componente.
3. **Adicionar um Estilo Condicional:** Adicione uma funcionalidade que permita marcar uma tarefa como concluída. As tarefas concluídas devem ser exibidas com um estilo diferente (por exemplo, riscado ou em uma cor mais clara).
4. **Criar um Componente de Contagem de Tarefas:** Crie um componente separado que mostre quantas tarefas ainda não foram concluídas, atualizando automaticamente à medida que o usuário marca as tarefas como concluídas ou as adiciona.

8.8: Eventos em React



Neste tópico, vamos aprofundar nossos conhecimentos sobre **eventos em React**. Já que estamos construindo uma **Aplicação de Gerenciamento de Tarefas**,

vamos explorar como lidar com diferentes tipos de eventos para tornar a interação do usuário mais dinâmica e fluida. Usaremos o código que desenvolvemos até agora como base e adicionaremos novas funcionalidades interativas.

8.8.1 Capturando Eventos de Clique

Vamos começar com o evento mais comum: o **clique**. Nós já vimos um exemplo básico de clique ao adicionar uma nova tarefa, mas agora vamos expandir essa funcionalidade.

Melhoria: Exibir uma mensagem de confirmação quando uma tarefa for adicionada.

Atualize a função `adicionarTarefa` para incluir uma mensagem de confirmação após a adição de uma tarefa:

```
const adicionarTarefa = () => {
  const nova = { id: tarefas.length + 1, nome: novaTarefa };
  setTarefas([...tarefas, nova]);
  alert(`Tarefa "${nova.nome}" adicionada com sucesso!`);
  setNovaTarefa('');
};
```

Explicação:

- `alert` : Exibimos uma mensagem confirmando a adição da nova tarefa, utilizando o nome da tarefa que foi adicionada.

8.8.2 Capturando Eventos de Teclado

Além dos cliques, eventos de teclado são muito importantes em interfaces interativas. Vamos capturar o evento de pressionar a tecla "Enter" para adicionar a tarefa, tornando o processo mais fluido.

Passo 1: Atualizar o campo de input para lidar com o evento de tecla.

Adicione o evento `onKeyDown` ao input:

```
<input  
  type="text"  
  value={novaTarefa}  
  onChange={(e) => setNovaTarefa(e.target.value)}  
  onKeyDown={(e) => e.key === 'Enter' && adicionarTarefa()}  
  placeholder="Adicionar nova tarefa"  
/>
```

Explicação:

- `onKeyDown`: Este evento captura as teclas pressionadas enquanto o usuário está no campo de input.
- `e.key === 'Enter' && adicionarTarefa()`: Verificamos se a tecla pressionada é "Enter" (`e.key === 'Enter'`). Se for, a função `adicionarTarefa` é chamada, permitindo que o usuário adicione uma tarefa apenas pressionando Enter.

8.8.3 Capturando Eventos de Mudança de Input

Capturar mudanças nos campos de input é uma prática comum para atualizar o estado de forma dinâmica enquanto o usuário interage com a aplicação. Já estamos utilizando o evento `onChange` para atualizar o estado `novaTarefa`, mas vamos reforçar esse conceito.

Recapitulando:

```
<input  
  type="text"  
  value={novaTarefa}  
  onKeyDown={(e) => e.key === 'Enter' && adicionarTarefa()}  
  onChange={(e) => setNovaTarefa(e.target.value)}  
  placeholder="Adicionar nova tarefa"  
/>
```

Explicação:

- `onChange`: Este evento é disparado toda vez que o valor do input muda. A função associada (`setNovaTarefa`) atualiza o estado `novaTarefa` com o valor digitado pelo usuário.

8.8.4 Criando Eventos Customizados ✨

Eventos customizados são uma maneira poderosa de criar comportamentos específicos para sua aplicação. Vamos adicionar a funcionalidade de **marcar uma tarefa como concluída**.

Passo 1: Adicionar um botão "Concluir" para cada tarefa.

Atualize o componente `Tarefa` para incluir um botão que marca a tarefa como concluída:

```
function Tarefa({ nome, onConcluir }) {
  return (
    <li>
      {nome}
      <button onClick={onConcluir}>Concluir</button>
    </li>
  );
}

export default Tarefa;
```

Passo 2: Atualizar o `App.jsx` para lidar com a conclusão da tarefa.

Adicione uma função que marca a tarefa como concluída:

```
const concluirTarefa = (id) => {
  setTarefas(tarefas.map((tarefa) =>
    tarefa.id === id ? { ...tarefa, concluida: true } : tarefa
  ));
};
```

E passe essa função como prop para o componente `Tarefa`:

```
<ul>
  {tarefas.map((tarefa) => (
    <Tarefa
      key={tarefa.id}
      nome={tarefa.nome}
      onConcluir={() => concluirTarefa(tarefa.id)}
    />
  ))}
</ul>
```

Explicação:

- `onConcluir`: Passamos uma função como prop para o componente `Tarefa`, que é chamada quando o usuário clica no botão "Concluir".
- `concluirTarefa`: Esta função percorre a lista de tarefas e marca como concluída a tarefa cujo ID corresponde ao que foi passado.

8.8.5 Feedback Visual ao Concluir Tarefas 🎨

Vamos adicionar um feedback visual para indicar que uma tarefa foi concluída. Para isso, vamos alterar o estilo da tarefa concluída, por exemplo, riscando o texto.

Passo 1: Atualizar o componente `Tarefa` para aplicar o estilo condicional.

Modifique o componente `Tarefa` para aceitar a prop `concluida` e aplicar o estilo condicionalmente:

```
function Tarefa({ nome, concluida, onConcluir }) {
  return (
    <li style={{ textDecoration: concluida ? 'line-through' :
  'none' }}>
```

```

        {nome}
      <button onClick={onConcluir}>Concluir</button>
    </li>
  );
}

export default Tarefa;

```

Passo 2: Atualizar o `App.jsx` para passar a prop `concluida` para o componente `Tarefa`.

Modifique o código no `App.jsx` para incluir a prop `concluida`:

```

<ul>
  {tarefas.map((tarefa) => (
    <Tarefa
      key={tarefa.id}
      nome={tarefa.nome}
      concluida={tarefa.concluida}
      onConcluir={() => concluirTarefa(tarefa.id)}
    />
  )))
</ul>

```

Explicação:

- `textDecoration: concluida ? 'line-through' : 'none'`: Aplicamos o estilo de risco (`line-through`) ao texto da tarefa se ela estiver marcada como concluída.
- `concluida={tarefa.concluida}`: Passamos a prop `concluida` para o componente `Tarefa`, que determinará se o estilo de risco deve ser aplicado.

Exercícios Práticos



1. **Editar Tarefas:** Adicione a funcionalidade de editar o nome de uma tarefa existente. Ao clicar em "Editar", o nome da tarefa deve ser carregado no input para ser modificado.

2. **Remover Tarefas:** Crie um botão "Remover" ao lado de cada tarefa que permite ao usuário excluir a tarefa da lista.
 3. **Marcar Tarefas como Importantes:** Adicione uma funcionalidade que permite ao usuário marcar tarefas como importantes. Tarefas importantes podem ser destacadas visualmente na lista.
 4. **Filtro de Tarefas:** Implemente um filtro para exibir apenas tarefas concluídas, não concluídas ou todas.
 5. **Contagem de Tarefas:** Adicione um contador que exibe o número total de tarefas e quantas delas foram concluídas.
-

8.9 Consumindo APIs com React

Neste subcapítulo, vamos explorar como consumir APIs em uma aplicação React. Vamos aprender a usar duas abordagens populares para fazer requisições HTTP: a API nativa `fetch` e a biblioteca `axios`. Em seguida, aplicaremos esses conceitos na nossa aplicação de todo-list, conectando-a à API de tarefas que criamos no capítulo de backend.

8.9.1 O que é o Fetch?

O `fetch` é uma API nativa do JavaScript que permite fazer requisições HTTP de forma assíncrona. Com ele, podemos enviar e receber dados de servidores externos, como APIs, sem bloquear a execução do restante do código. O `fetch` é amplamente utilizado por ser simples e já vir embutido nos navegadores modernos.

Exemplo Básico de Fetch

Antes de integrarmos o `fetch` em nossa aplicação de todo-list, vamos ver um exemplo simples de como ele funciona.

```
fetch('https://api.github.com/users')
  .then(response => response.json())
```

```
.then(data => console.log(data))
.catch(error => console.error('Erro:', error));
```

Explicação:

1. `fetch('https://api.github.com/users')`: Esta linha faz uma requisição GET para a URL especificada. O `fetch` retorna uma Promise, que é uma maneira de lidar com operações assíncronas.
2. `.then(response => response.json())`: Aqui, usamos o método `.then()` para manipular a resposta da requisição. A resposta é convertida para JSON, o que é necessário porque as APIs geralmente retornam dados nesse formato.
3. `.then(data => console.log(data))`: Neste segundo `.then()`, lidamos com os dados convertidos e os exibimos no console.
4. `.catch(error => console.error('Erro:', error))`: O método `.catch()` captura qualquer erro que ocorra durante a requisição ou no processamento dos dados, exibindo uma mensagem de erro no console.

8.9.2 O que é o Axios?

O `axios` é uma biblioteca JavaScript popular que simplifica as requisições HTTP, oferecendo mais funcionalidades que o `fetch`, como suporte a navegadores mais antigos, tratamento de erros mais consistente e um formato de código mais limpo. Embora o `fetch` seja nativo e leve, o `axios` é frequentemente preferido em projetos maiores pela sua flexibilidade e funcionalidades adicionais.

Exemplo Básico de Axios

Agora, vamos ver um exemplo de como usar o `axios` para fazer a mesma requisição que fizemos com o `fetch`.

Primeiro, precisamos instalar o `axios` no nosso projeto:

```
npm install axios
```

Agora, o exemplo em si:

```
import axios from 'axios';

axios.get('https://api.github.com/users')
  .then(response => console.log(response.data))
  .catch(error => console.error('Erro:', error));
```

Explicação:

1. `axios.get(https://api.github.com/users)` : O `axios` fornece métodos específicos para diferentes tipos de requisição, como `get`, `post`, `put`, `delete`. Aqui, estamos usando o `axios.get()` para fazer uma requisição GET.
2. `.then(response => console.log(response.data))` : Assim como no `fetch`, usamos `.then()` para lidar com a resposta. Porém, no `axios`, a resposta completa está dentro de um objeto `response`, e os dados reais são acessados através de `response.data`.
3. `.catch(error => console.error('Erro:', error))` : O `axios` também oferece suporte a `.catch()` para tratar erros, de maneira similar ao `fetch`.

Conclusão sobre Fetch e Axios

Ambas as ferramentas têm seus méritos:

- **Fetch** é ideal para projetos leves ou quando se quer evitar dependências externas, mas requer mais código para manipular erros e configurar cabeçalhos.
- **Axios** é mais robusto, oferecendo uma sintaxe mais elegante e suporte a mais funcionalidades, tornando-o preferido para aplicações maiores ou mais complexas.

No próximo passo, vamos aplicar o que aprendemos sobre `fetch` e `axios` na nossa aplicação de todo-list. Os exemplos apresentados serão feitos com `axios`, mas fique a vontade para usar `fetch` caso queira.

8.9.3 Implementando o Consumo de API na Aplicação Todo-List

Agora que você já aprendeu a consumir APIs usando `fetch` e `axios`, vamos implementar essas técnicas na nossa aplicação Todo-List. Vamos substituir as

funções de adicionar e listar tarefas para que elas interajam com a API que criamos no backend usando Node.js.

Observação: É necessário ter a API rodando em <http://localhost:3000> para que essa aplicação funcione

Primeiro, comece editando o componente da tarefa para mostrar a descrição da tarefa (uma vez que a API que desenvolvemos também armazena a descrição)

```
function Tarefa({ nome, concluida, descricao, onConcluir }) {
  return (
    <li style={{ textDecoration: concluida ? "line-through" : "" }}>
      <span>{nome} - </span>
      <span>{descricao}</span>
      <button onClick={onConcluir}>Concluir</button>
    </li>
  );
}

export default Tarefa;
```

Para o App.jsx, aqui está o código completo com as mudanças necessárias:

```
import "./App.css";
import React, { useState, useEffect } from "react";
import axios from "axios";
import Tarefa from "./components/Tarefa";
import Titulo from "./components/Titulo";

function App() {
  const [tarefas, setTarefas] = useState([]);
  const [novaTarefa, setNovaTarefa] = useState("");
  const [descricao, setDescricao] = useState("");

  // Função para adicionar uma nova tarefa
  const adicionarTarefa = () => {
    const novaTarefaObj = {
      nome: "Comprar leite",
      descricao: "Fazer compras na padaria",
      concluida: false
    };

    setTarefas([...tarefas, novaTarefaObj]);
    setNovaTarefa("");
  };

  // Função para marcar uma tarefa como concluída
  const marcarConcluida = (id) => {
    const tarefasModificadas = tarefas.map((tarefa) => {
      if (tarefa.id === id) {
        tarefa.concluida = true;
      }
      return tarefa;
    });

    setTarefas(tarefasModificadas);
  };

  // Função para excluir uma tarefa
  const excluirTarefa = (id) => {
    const tarefasModificadas = tarefas.filter((tarefa) => tarefa.id !== id);

    setTarefas(tarefasModificadas);
  };

  // Função para limpar a lista de tarefas
  const limparTarefas = () => {
    setTarefas([]);
  };

  // Função para buscar tarefas
  const buscarTarefas = () => {
    axios.get("http://localhost:3000/tarefas")
      .then((res) => {
        const tarefasRecuperadas = res.data.map((tarefa) => {
          return {
            ...tarefa,
            id: tarefa._id
          };
        });

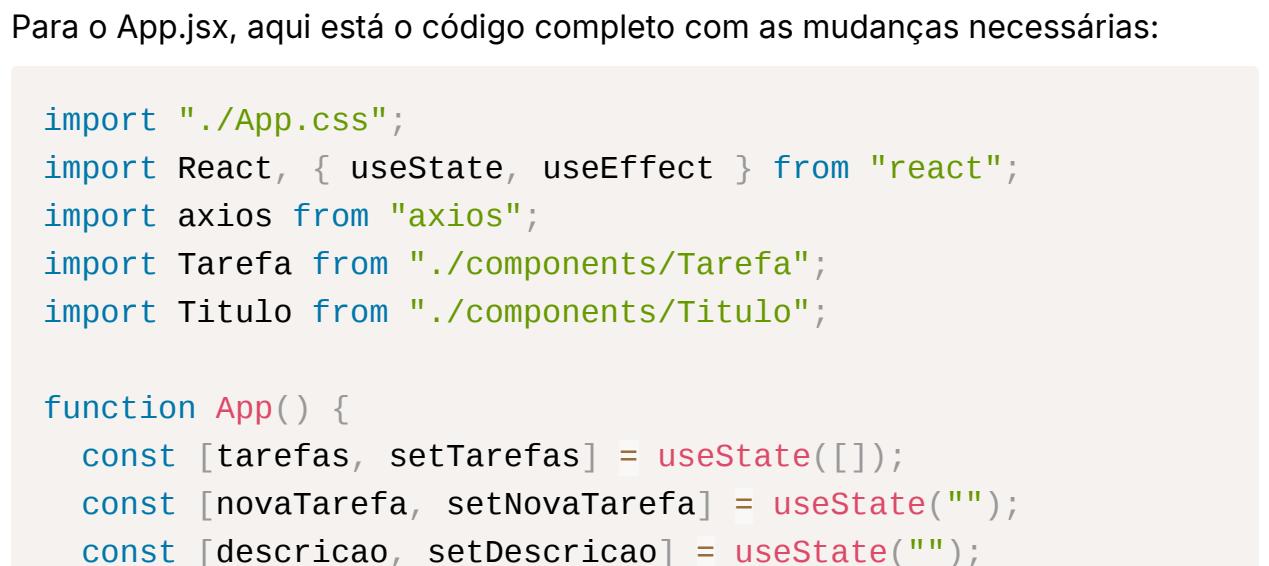
        setTarefas(tarefasRecuperadas);
      })
      .catch((err) => console.error(err));
  };

  // Função para adicionar uma nova descrição
  const adicionarDescricao = (e) => {
    e.preventDefault();
    setDescricao(e.target.value);
  };

  // Função para limpar a descrição
  const limparDescricao = () => {
    setDescricao("");
  };

  // Função para enviar a descrição para o backend
  const enviarDescricao = (e) => {
    e.preventDefault();
    const descricaoObj = {
      descricao: descricao
    };

    axios.post("http://localhost:3000/tarefas", descricaoObj)
      .then((res) => {
        const tarefaCriada = res.data;
        const tarefasNovas = [...tarefas, tarefaCriada];
        setTarefas(tarefasNovas);
        setDescricao("");
      })
      .catch((err) => console.error(err));
  };

  return (
    <div>
      <h1>Lista de Tarefas</h1>
      <Tit
        
```

```

const buscarTarefas = async () => {
  try {
    const response = await axios.get("http://localhost:3000/tarefas");
    setTarefas(response.data);
  } catch (error) {
    console.error("Erro ao buscar tarefas:", error);
  }
};

const adicionarTarefa = async () => {
  if (novaTarefa.trim() === '') return;

  try {
    await axios.post('http://localhost:3000/tarefas', {
      titulo: novaTarefa,
      descricao,
      concluida: false,
    });
    buscarTarefas();
    setNovaTarefa('');
    setDescricao('');
  } catch (error) {
    console.error('Erro ao adicionar tarefa:', error);
  }
};

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

return (
  <div>
    <h1>Gerenciador de Tarefas <img alt="pencil icon" /></h1>

```

```

<Titulo texto="Lista de Tarefas" />
<ul>
  {tarefas.map((tarefa) => (
    <Tarefa
      key={tarefa.id}
      nome={tarefa.titulo}
      descricao={tarefa.descricao}
    />
  ))}
</ul>
<input
  type="text"
  value={novaTarefa}
  onChange={(e) => setNovaTarefa(e.target.value)}
  placeholder="Adicionar nova tarefa"
/>
<input
  type="text"
  value={descricao}
  onChange={(e) => setDescricao(e.target.value)}
  placeholder="Descrição"
/>
<button onClick={adicionarTarefa}>Adicionar</button>
</div>
);
}

export default App;

```

Explicação do Código

1. Importações e Estados

```

import "./App.css";
import React, { useState, useEffect } from "react";

```

```
import axios from "axios";
import Tarefa from "./components/Tarefa";
import Titulo from "./components/Titulo";
```

Aqui estamos importando `axios` para facilitar o consumo da API, juntamente com os componentes para renderizar na tela.

2. Estados do componente

```
const [tarefas, setTarefas] = useState([]);
const [novaTarefa, setNovaTarefa] = useState("");
const [descricao, setDescricao] = useState("");
```

Aqui, estamos declarando os estados do componente. O estado `tarefas` armazenará a lista de tarefas obtida da API, enquanto `novaTarefa` e `descricao` serão utilizados para controlar os inputs do usuário ao adicionar novas tarefas e suas descrições.

3. Função para Buscar Tarefas

```
const buscarTarefas = async () => {
  try {
    const response = await axios.get('<http://localhost:3000/tarefas>');
    setTarefas(response.data);
  } catch (error) {
    console.error('Erro ao buscar tarefas:', error);
  }
};
```

Essa função utiliza `axios` para fazer uma requisição GET à API e obter a lista de tarefas. Ao receber a resposta, atualizamos o estado `tarefas` com os dados recebidos.

4. Função para Adicionar uma Nova Tarefa

```
const adicionarTarefa = async () => {
  if (novaTarefa.trim() === '') return;

  try {
    await axios.post('http://localhost:3000/tarefas', {
      titulo: novaTarefa,
      descricao,
      concluida: false,
    });
    buscarTarefas();
    setNovaTarefa('');
    setDescricao('');
  } catch (error) {
    console.error('Erro ao adicionar tarefa:', error);
  }
};
```

Aqui, usamos `axios` para enviar uma requisição POST à API, criando uma nova tarefa com o título fornecido pelo usuário. Se a requisição for bem-sucedida, uma nova busca é realizada para atualizar a listagem.

4. useEffect para Carregar Tarefas ao Iniciar o App

```
useEffect(() => {
  buscarTarefas();
}, []);
```

Esse `useEffect` garante que as tarefas sejam carregadas da API assim que o componente for montado. Ele chama a função `buscarTarefas` uma vez, ao carregar a aplicação.

5. Renderização dos Componentes

```
return (
  <div>
```

```

<h1>Gerenciador de Tarefas <img alt="pencil icon" style="vertical-align: middle;"/></h1>
<Titulo texto="Lista de Tarefas" />
<ul>
  {tarefas.map((tarefa) => (
    <Tarefa
      key={tarefa.id}
      nome={tarefa.titulo}
      descricao={tarefa.descricao}
    />
  ))}
</ul>
<input
  type="text"
  value={novaTarefa}
  onChange={(e) => setNovaTarefa(e.target.value)}
  placeholder="Adicionar nova tarefa"
/>
<input
  type="text"
  value={descricao}
  onChange={(e) => setDescricao(e.target.value)}
  placeholder="Descrição"
/>
<button onClick={adicionarTarefa}>Adicionar</button>
</div>
);

```

Aqui, renderizamos a lista de tarefas no formato `` e permitimos que o usuário adicione novas tarefas por meio de um campo de input para o título, um campo de input para a descrição e um botão.

Exercícios Práticos

- 1. Adicionar Autenticação:** Implemente um sistema de login e registro para que os usuários possam salvar suas tarefas e acessá-las de qualquer dispositivo.

2. **Persistência de Dados:** Utilize uma API ou banco de dados local para salvar as tarefas, garantindo que os dados não sejam perdidos ao recarregar a página.
3. **Atualização de Tarefas:** Implemente a funcionalidade de atualizar tarefas, permitindo que o usuário edite o título e a descrição de uma tarefa existente.
4. **Checkagem de tarefas:** Implemente a funcionalidade de marcar a tarefa como concluída, mas integrado com a API.
5. **Interface Responsiva:** Melhore a responsividade da interface para garantir que o aplicativo funcione bem em dispositivos móveis e tablets.
6. **Dark Mode:** Implemente um modo escuro para melhorar a experiência do usuário em ambientes com pouca luz.
7. **Filtros:** Adicione opções de filtro e ordenação para que os usuários possam organizar suas tarefas de maneiras diferentes.
8. **Deleção de Tarefas:** Adicione a capacidade de deletar tarefas da lista.
9. **Tratamento de Erros:** Implemente mensagens de erro mais amigáveis para o usuário.
10. **Animações:** Utilize animações para tornar a interface mais dinâmica e agradável de usar.
11. **Documentação Completa:** Crie uma documentação completa do projeto, incluindo instruções de instalação, uso e contribuição.

Agora que você sabe como consumir APIs em React, pode implementar funcionalidades mais avançadas e transformar sua aplicação em um gerenciador de tarefas completo e profissional.

Subindo o Projeto no GitHub

Recomendo que você suba esse projeto no seu GitHub!

Subir o projeto no GitHub não só ajuda a manter um backup do seu trabalho, mas também permite que outras pessoas vejam o seu trabalho, colaborem e dêem feedback sobre seu código.

módulo 9

estilização avançada

Aprimore o visual das suas aplicações com
styled-components e TailwindCSS. 

12 tópicos neste módulo

Capítulo 9: Estilização Avançada

9.1 Introdução ao Styled-components

Bem-vindo ao mundo do styled-components! 

O **styled-components** é uma biblioteca incrível para estilização no React que utiliza o poder do JavaScript para criar componentes CSS personalizados. Em vez de usar classes CSS tradicionais e arquivos separados, com styled-components você pode escrever o estilo diretamente dentro dos seus componentes React. Isso traz várias vantagens para o desenvolvimento de interfaces modernas e dinâmicas.

O que é Styled-components?

Styled-components é uma biblioteca baseada em **CSS-in-JS**, que significa “CSS dentro do JavaScript”. Em vez de escrever seu CSS em arquivos .css separados, você cria componentes estilizados diretamente no arquivo JavaScript. Isso torna o código mais modular e facilita a reutilização e o compartilhamento de estilos entre os componentes.

- **Escopo Local:** Uma das maiores vantagens do styled-components é que os estilos são automaticamente escopados para o componente ao qual pertencem. Isso elimina problemas comuns como conflitos de nomes de classes e vazamento de estilos.
- **Temas e Provedores de Tema:** Styled-components permite que você defina temas globais para sua aplicação, facilitando a manutenção e a consistência dos estilos em toda a interface.
- **Manutenção Simplificada:** Ao manter o estilo e a lógica de um componente juntos, você melhora a legibilidade do código e facilita a manutenção a longo prazo.

Por que usar Styled-components?

1. **Modularidade e Reutilização:** Cada componente possui seu próprio estilo, tornando mais fácil reutilizá-los em diferentes partes da aplicação ou até em projetos diferentes.
 2. **Interação com Props:** Você pode facilmente modificar estilos com base nas props do componente, criando interfaces altamente dinâmicas e reativas.
 3. **Melhor Experiência de Desenvolvimento:** Com a ajuda de ferramentas de desenvolvimento como o **VS Code**, você terá suporte a autocompletar e verificação de tipos para os estilos, o que melhora significativamente a produtividade.
 4. **Estilo Condicional:** O styled-components permite aplicar estilos condicionais de forma limpa e fácil, sem a necessidade de manipular classes manualmente
-

9.2 Configuração do Styled-components e Melhoria na Estrutura do Projeto

Neste tópico, vamos aplicar **styled-components** ao projeto existente e melhorar a estrutura do código. A ideia é estilizar os componentes e deixar o código mais modular e fácil de manter. Vamos transformar o projeto atual utilizando styled-components e algumas boas práticas de estruturação.

O que é um componente estilizado?

Um componente estilizado é um componente React que utiliza a biblioteca `styled-components` para aplicar estilos CSS diretamente no JavaScript. Isso permite que os estilos sejam encapsulados no próprio componente, promovendo a modularidade e a reutilização de código. Com `styled-components`, você pode definir estilos dinâmicos baseados nas props do componente, aplicar temas globais e evitar conflitos de nomes de classes. Isso resulta em um código mais limpo, organizado e fácil de manter.

9.2.1 Refatorando os Componentes com Styled-components

Vamos começar refatorando os componentes `Titulo` e `Tarefa` para usar styled-components.

Instalando Styled-components

Caso ainda não tenha instalado, no terminal, rode:

```
npm install styled-components
```

Refatorando o Componente `Titulo`

Vamos criar uma versão estilizada do componente `Titulo`.

`src/components/Titulo.jsx`:

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

const StyledTitulo = styled.h2`  
  font-size: 2rem;  
  color: #333;  
  margin-bottom: 20px;  
`;

function Titulo({ texto }) {  
  return (  
    <StyledTitulo>{texto}</StyledTitulo>  
  );  
}

export default Titulo;
```

Aqui, criamos um componente `StyledTitulo` usando `styled-components`, encapsulando o estilo e tornando-o reutilizável.

Refatorando o Componente `Tarefa`

Agora, vamos refatorar o componente `Tarefa`.

`src/components/Tarefa.jsx:`

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

const StyledTarefa = styled.li`  
background: ${({props}) => (props.concluida ? '#d4edda' : '#f0f0f0')};  
color: ${({props}) => (props.concluida ? '#155724' : '#333')};  
margin: 5px 0;  
padding: 10px;  
border-radius: 5px;  
font-size: 1.2rem;  
text-decoration: ${({props}) => (props.concluida ? 'line-through' : 'none')};  
display: flex;  
justify-content: space-between;  
align-items: center;  
`;  
  
const ConcluirButton = styled.button`  
background: #28a745;  
color: white;  
border: none;  
padding: 5px 10px;  
border-radius: 3px;  
cursor: pointer;  
  
&:hover {  
background: #218838;  
}  
`;
```

```

function Tarefa({ nome, concluida, descricao, onConcluir }) {
  return (
    <StyledTarefa concluida={concluida}>
      <span>{nome} - </span>
      <span>{descricao}</span>
      <ConcluirButton onClick={onConcluir}>Concluir</Concluir
    Button>
    </StyledTarefa>
  );
}

export default Tarefa;

```

Neste exemplo, usamos o `styled-components` para criar `StyledTarefa` e `ConcluirButton`, aplicando estilos condicionais com base nas props.

9.2.2. Melhorando a Estrutura do Projeto

Agora, vamos organizar melhor o projeto, separando responsabilidades e facilitando futuras manutenções.

Criando um Componente de Layout

Vamos criar um componente para o layout da aplicação, centralizando o conteúdo e aplicando alguns estilos globais.

`src/components/Layout.jsx:`

```

import styled from 'styled-components';

const Container = styled.div`
  max-width: 600px;
  margin: 50px auto;
  padding: 20px;
  background: #fff;
  border-radius: 10px;

```

```

        box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
        text-align: center;
    };

    function Layout({ children }) {
        return <Container>{children}</Container>;
    }

    export default Layout;

```

Esse componente `Layout` vai centralizar o conteúdo e aplicar um fundo branco com sombra, melhorando a aparência da aplicação.

Integrando o Layout no `App`

Agora, vamos integrar o layout no nosso componente `App`.

`App.jsx:`

```

import { useState } from 'react';
import axios from "axios";
import Tarefa from './components/Tarefa';
import Titulo from './components/Titulo';
import Layout from './components/Layout';

function App() {
    const [tarefas, setTarefas] = useState([]);
    const [novaTarefa, setNovaTarefa] = useState("");
    const [descricao, setDescricao] = useState("");

    const buscarTarefas = async () => {
        try {
            const response = await axios.get("http://localhost:3000/tarefas");
            setTarefas(response.data);
        } catch (error) {

```

```

        console.error("Erro ao buscar tarefas:", error);
    }
};

const adicionarTarefa = async () => {
    if (novaTarefa.trim() === '') return;

    try {
        await axios.post('http://localhost:3000/tarefas', {
            titulo: novaTarefa,
            descricao,
            concluida: false,
        });
        buscarTarefas();
        setNovaTarefa('');
        setDescricao('');
    } catch (error) {
        console.error('Erro ao adicionar tarefa:', error);
    }
};

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

return (
    <Layout>
        <h1>Gerenciador de Tarefas <img alt="pencil icon" style={{verticalAlign: 'middle'}}/></h1>
        <Titulo texto="Lista de Tarefas" />
        <ul>
            {tarefas.map((tarefa) => (
                <Tarefa
                    key={tarefa.id}
                    nome={tarefa.nome}
                    descricao={tarefa.descricao}
                    concluida={tarefa.concluida}

```

```

        onConcluir={() => concluirTarefa(tarefa.id)}
      />
    )}
</ul>
<input
  type="text"
  value={novaTarefa}
  onKeyDown={(e) => e.key === 'Enter' && adicionarTarefa(
    a()
  )
  onChange={(e) => setNovaTarefa(e.target.value)}
  placeholder="Adicionar nova tarefa"
/>
<button onClick={adicionarTarefa}>Adicionar</button>
</Layout>
);
}

export default App;

```

Agora, todo o conteúdo do `App` é renderizado dentro do componente `Layout`, o que centraliza o layout e melhora a estética.

9.2.3 Removendo o CSS Antigo

Agora que estamos usando styled-components, podemos remover o antigo arquivo `App.css` para evitar conflitos.

```
rm src/App.css
```

9.2.4 Testando as Melhorias

Depois de fazer todas essas mudanças, rode novamente o projeto:

```
npm run dev
```

9.3 Styled-components - Temas e Provedores de Tema

⭐ Vamos adicionar uma nova camada de personalização ao seu projeto! ⭐

Neste tópico, vamos explorar como utilizar **temas** no `styled-components` para aplicar estilos globais de forma dinâmica e consistente. Com o uso de temas, você poderá facilmente alternar entre diferentes estilos (como temas claro e escuro) sem precisar modificar manualmente cada componente. Isso trará mais flexibilidade e escalabilidade ao seu projeto.

9.3.1 Introdução aos Temas com Styled-components

No `styled-components`, os temas permitem que você defina estilos globais que podem ser aplicados de forma consistente em toda a aplicação. Esses temas são gerenciados pelo `ThemeProvider`, um componente especial que vem embutido na biblioteca.

Com temas, você pode definir cores, tamanhos de fonte, espaçamentos e outros valores estilísticos uma única vez, e então utilizá-los em todos os componentes. Vamos configurar um tema global e aplicá-lo ao projeto.

9.3.2 Configurando o Tema Global

9.3.2.1 Criando o Tema

Primeiro, vamos criar um arquivo de tema. Esse arquivo conterá todas as variáveis que queremos usar em nosso projeto.

`src/themes/defaultTheme.js`:

```

const defaultTheme = {
  colors: {
    primary: '#4CAF50',
    secondary: '#8BC34A',
    background: '#f0f0f0',
    text: '#333',
    buttonText: '#fff',
    buttonBackground: '#28a745',
  },
  fontSizes: {
    small: '0.875rem',
    medium: '1rem',
    large: '1.5rem',
  },
  spacings: {
    small: '8px',
    medium: '16px',
    large: '24px',
  },
};

export default defaultTheme;

```

Nesse exemplo, definimos cores e tamanhos de fonte que usaremos no projeto. Essas variáveis podem ser facilmente ajustadas ou expandidas conforme necessário.

9.3.2.2 Aplicando o Tema com ThemeProvider

Agora, vamos usar o `ThemeProvider` para aplicar esse tema à nossa aplicação.

App.jsx:

```

import { useState } from 'react';
import axios from "axios";
import { ThemeProvider } from 'styled-components';

```

```

import defaultTheme from './themes/defaultTheme';
import Tarefa from './components/Tarefa';
import Titulo from './components/Titulo';
import Layout from './components/Layout';

function App() {
  const [tarefas, setTarefas] = useState([]);
  const [novaTarefa, setNovaTarefa] = useState("");
  const [descricao, setDescricao] = useState("");

  const buscarTarefas = async () => {
    try {
      const response = await axios.get("http://localhost:3000/tarefas");
      setTarefas(response.data);
    } catch (error) {
      console.error("Erro ao buscar tarefas:", error);
    }
  };

  const adicionarTarefa = async () => {
    if (novaTarefa.trim() === '') return;

    try {
      await axios.post('http://localhost:3000/tarefas', {
        titulo: novaTarefa,
        descricao,
        concluida: false,
      });
      buscarTarefas();
      setNovaTarefa('');
      setDescricao('');
    } catch (error) {
      console.error('Erro ao adicionar tarefa:', error);
    }
  };
}

```

```

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

return (
  <ThemeProvider theme={defaultTheme}>
    <Layout>
      <h1>Gerenciador de Tarefas <EditIcon/></h1>
      <Title texto="Lista de Tarefas" />
      <ul>
        {tarefas.map((tarefa) => (
          <Tarefa
            key={tarefa.id}
            nome={tarefa.nome}
            concluida={tarefa.concluida}
            onConcluir={() => concluirTarefa(tarefa.id)}
          />
        )));
      </ul>
      <input
        type="text"
        value={novaTarefa}
        onKeyDown={(e) => e.key === 'Enter' && adicionarTarefa()}
        onChange={(e) => setNovaTarefa(e.target.value)}
        placeholder="Adicionar nova tarefa"
      />
      <button onClick={adicionarTarefa}>Adicionar</button>
    </Layout>
  </ThemeProvider>
);
}

export default App;

```

Aqui, envolvemos toda a aplicação com o `ThemeProvider`, passando o `defaultTheme` como prop. Agora, todos os componentes dentro do `ThemeProvider` terão acesso às variáveis do tema.

9.3.3 Usando Variáveis de Tema nos Componentes

Vamos ajustar nossos componentes `Titulo` e `Tarefa` para utilizar as variáveis do tema.

9.3.3.1 Atualizando o Componente `Titulo`

`src/components/Titulo.jsx:`

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

const StyledTitulo = styled.h2`
  font-size: ${({ theme }) => theme.fontSizes.large};
  color: ${({ theme }) => theme.colors.text};
  margin-bottom: ${({ theme }) => theme.spacings.large};
`;

function Titulo({ texto }) {
  return (
    <StyledTitulo>{texto}</StyledTitulo>
  );
}

export default Titulo;
```

9.3.3.2 Atualizando o Componente `Tarefa`

`src/components/Tarefa.jsx:`

```

import styled from 'styled-components';

const StyledTarefa = styled.li`  

  background: ${({props}) => (props.concluida ? props.theme.colors.secondary : props.theme.colors.background)};  

  color: ${({props}) => (props.concluida ? '#155724' : props.theme.colors.text)};  

  margin: ${({ theme }) => theme.spacings.small} 0;  

  padding: ${({ theme }) => theme.spacings.medium};  

  border-radius: 5px;  

  font-size: ${({ theme }) => theme.fontSizes.medium};  

  text-decoration: ${({props}) => (props.concluida ? 'line-through' : 'none')};  

  display: flex;  

  justify-content: space-between;  

  align-items: center;  

`;  

const ConcluirButton = styled.button`  

  background: ${({ theme }) => theme.colors.buttonBackground};  

  color: ${({ theme }) => theme.colors.buttonText};  

  border: none;  

  padding: 5px 10px;  

  border-radius: 3px;  

  cursor: pointer;  

  &:hover {  

    background: #218838;  

  }  

`;  

function Tarefa({ nome, concluida, descricao, onConcluir }) {  

  return (  

    <StyledTarefa concluida={concluida}>

```

```
        <span>{nome} - </span>
        <span>{descricao}</span>
        <ConcluirButton onClick={onConcluir}>Concluir</Concluir
      Button>
    </StyledTarefa>
  );
}

export default Tarefa;
```

Agora, nossos componentes estão utilizando as variáveis de tema, o que facilita a manutenção e a personalização do estilo.

Exercícios Práticos

Para consolidar o que foi aprendido, aqui estão alguns exercícios para você aprimorar ainda mais o projeto:

- 1. Criar Tema Escuro:** Crie um tema escuro e adicione uma funcionalidade no `App` para alternar entre o tema claro e o escuro.
- 2. Refatorar Mais Componentes:** Refatore outros componentes ou crie novos usando as variáveis de tema, garantindo que todo o projeto esteja consistente em termos de estilo.
- 3. Adicionar Novas Variáveis ao Tema:** Adicione mais variáveis ao tema, como espaçamentos, bordas, sombras, e utilize-as para refinar o layout.
- 4. Aplicar Temas Condicionais:** Modifique a aplicação para aplicar temas diferentes com base em um estado do aplicativo, como o horário do dia ou preferências do usuário.
- 5. Testar Responsividade:** Ajuste os temas e os componentes para garantir que funcionem bem em diferentes tamanhos de tela, utilizando as variáveis do tema para controlar a responsividade.

9.4 Styled-components - Animações e Estilizações Avançadas

🎨 Vamos transformar sua aplicação com animações e estilizações avançadas!



Neste tópico, vamos adicionar animações e aplicar técnicas de estilização avançada para melhorar ainda mais a experiência do usuário na sua aplicação. Vamos estilizar o formulário de tarefas e adicionar transições e animações para tornar a interface mais interativa e visualmente agradável. Além disso, você encontrará exercícios para praticar e incrementar o projeto.

9.4.1 Estilizando o Formulário de Tarefas

Vamos começar aplicando um estilo mais agradável ao formulário de tarefas. Usaremos styled-components para isso.

9.4.1.1 Criando Componentes Estilizados para o Formulário

Primeiro, vamos criar componentes estilizados para o input e o botão.

src/components/Formulario.jsx:

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

const FormContainer = styled.div`  
  margin-top: 20px;  
`;

const InputTarefa = styled.input`  
  padding: 10px;  
  font-size: 1rem;  
  border: 2px solid ${({ theme }) => theme.colors.primary};  
  border-radius: 5px;  
  margin-right: 10px;  
  width: 250px;  
  transition: border-color 0.3s ease;
```

```

&:focus {
  border-color: ${({ theme }) => theme.colors.secondary};
  outline: none;
}

const AdicionarButton = styled.button`
padding: 10px 20px;
font-size: 1rem;
color: ${({ theme }) => theme.colors.buttonText};
background-color: ${({ theme }) => theme.colors.buttonBackground};
border: none;
border-radius: 5px;
cursor: pointer;
transition: background-color 0.3s ease, transform 0.2s ease;
`;

&:hover {
  background-color: #218838;
  transform: scale(1.05);
}

&:active {
  transform: scale(0.95);
}
`;

function Formulario({ novaTarefa, setNovaTarefa, descricao, setDescricao, adicionarTarefa }) {
  return (
    <FormContainer>
      <InputTarefa
        type="text"
        value={novaTarefa}
      >

```

```

        onChange={(e) => setNovaTarefa(e.target.value)}
        placeholder="Adicionar nova tarefa"
      />
      <InputTarefa
        type="text"
        value={descricao}
        onChange={(e) => setDescricao(e.target.value)}
        placeholder="Descricao"
      />
      <AdicionarButton onClick={adicionarTarefa}>Adicionar</A
      dicionarButton>
    </FormContainer>
  );
}

export default Formulario;

```

9.4.2 Integrando o Formulário Estilizado no App

Agora, vamos substituir o formulário antigo pelo novo componente estilizado.

App.jsx:

```

import { useState } from 'react';
import axios from "axios";
import { ThemeProvider } from 'styled-components';
import defaultTheme from './themes/defaultTheme';
import Tarefa from './components/Tarefa';
import Titulo from './components/Titulo';
import Layout from './components/Layout';
import Formulario from './components/Formulario';

function App() {
  const [tarefas, setTarefas] = useState([]);
  const [novaTarefa, setNovaTarefa] = useState("");

```

```

const [descricao, setDescricao] = useState("");

const buscarTarefas = async () => {
  try {
    const response = await axios.get("http://localhost:3000/tarefas");
    setTarefas(response.data);
  } catch (error) {
    console.error("Erro ao buscar tarefas:", error);
  }
};

const adicionarTarefa = async () => {
  if (novaTarefa.trim() === '') return;

  try {
    await axios.post('http://localhost:3000/tarefas', {
      titulo: novaTarefa,
      descricao,
      concluida: false,
    });
    buscarTarefas();
    setNovaTarefa('');
    setDescricao('');
  } catch (error) {
    console.error('Erro ao adicionar tarefa:', error);
  }
};

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

return (
  <ThemeProvider theme={defaultTheme}>
    <Layout>

```

```

<h1>Gerenciador de Tarefas <img alt="pencil icon" style="vertical-align: middle;"/></h1>
<Titulo texto="Lista de Tarefas" />
<ul>
  {tarefas.map((tarefa) => (
    <Tarefa
      key={tarefa.id}
      nome={tarefa.nome}
      concluida={tarefa.concluida}
      descricao={tarefa.descricao}
      onConcluir={() => concluirTarefa(tarefa.id)}
    />
  )))
</ul>
<Formulario
  novaTarefa={novaTarefa}
  setNovaTarefa={setNovaTarefa}
  descricao={descricao}
  setDescricao={setDescricao}
  adicionarTarefa={adicionarTarefa}
/>
</Layout>
</ThemeProvider>
);

}

export default App;

```

9.4.3 Adicionando Animações ao Projeto

Agora que o formulário está estilizado, vamos adicionar algumas animações para tornar a interface mais fluida e interativa.

9.4.3.1 Animação ao Adicionar Tarefa

Podemos adicionar uma animação para destacar a nova tarefa adicionada.

src/components/Tarefa.jsx:

```
import styled, { keyframes } from 'styled-components';

const fadeIn = keyframes` 
  from {
    opacity: 0;
    transform: translateY(-10px);
  }
  to {
    opacity: 1;
    transform: translateY(0);
  }
`;

const StyledTarefa = styled.li`
  background: ${props => (props.concluida ? props.theme.colors.secondary : props.theme.colors.background)};
  color: ${props => (props.concluida ? '#155724' : props.theme.colors.text)};
  margin: ${({ theme }) => theme.spacings.small} 0;
  padding: ${({ theme }) => theme.spacings.medium};
  border-radius: 5px;
  font-size: ${({ theme }) => theme.fontSizes.medium};
  text-decoration: ${props => (props.concluida ? 'line-through' : 'none')};
  display: flex;
  justify-content: space-between;
  align-items: center;
  animation: ${fadeIn} 0.5s ease-out;
`;

const ConcluirButton = styled.button`
  background: ${({ theme }) => theme.colors.buttonBackground}
```

```

d};

color: ${({ theme }) => theme.colors.buttonText};
border: none;
padding: 5px 10px;
border-radius: 3px;
cursor: pointer;

&:hover {
  background: #218838;
}

;

function Tarefa({ nome, concluida, onConcluir }) {
  return (
    <StyledTarefa concluida={concluida}>
      <span>{nome} - </span>
      <span>{descricao}</span>
      <ConcluirButton onClick={onConcluir}>Concluir</Concluir
      Button>
    </StyledTarefa>
  );
}

export default Tarefa;

```

Exercícios Práticos

- 1. Adicionar Animações de Exclusão:** Adicione uma animação para quando uma tarefa for removida da lista.
- 2. Criar Feedback Visual ao Clicar:** Adicione uma animação para fornecer feedback visual quando o usuário clicar nos botões, como um efeito de pressionamento.

3. **Aprimorar a Experiência de Foco:** Melhore a experiência ao focar no campo de input, adicionando efeitos como mudança de cor ou tamanho da borda.
 4. **Animação ao Concluir Tarefas:** Adicione uma animação para quando uma tarefa for marcada como concluída, como um efeito de desvanecimento ou deslizamento.
 5. **Testar e Melhorar a Responsividade:** Garanta que as animações e estilos aplicados funcionem bem em dispositivos móveis, ajustando conforme necessário.
-

9.5 Introdução ao TailwindCSS



Bem-vindo ao universo do TailwindCSS!



O **TailwindCSS** é um framework CSS altamente utilitário que oferece uma abordagem inovadora para a estilização de interfaces web. Em vez de escrever classes CSS customizadas e lidar com arquivos separados, com TailwindCSS, você aplica classes utilitárias diretamente nos seus elementos HTML. Isso permite uma construção rápida e eficiente de interfaces modernas, ao mesmo tempo em que mantém seu código CSS altamente otimizado e livre de estilos não utilizados.

Oferece funcionalidades parecidas aos do framework Bootstrap, abordado nos capítulos anteriores, principalmente no que diz respeito as classes utilitárias (`flex`, `mt-5`, etc). O Tailwind vai fundo nessa abordagem ao oferecer centenas de classes para que você não precise escrever uma linha sequer de CSS.

O que é TailwindCSS?

Como já foi dito, o TailwindCSS é um framework CSS baseado em utilitários, onde você aplica classes pré-definidas diretamente nos elementos HTML para estilizar sua interface. Cada classe corresponde a uma única regra CSS, como margens, padding, cores, tipografia e muito mais. Isso resulta em um código mais conciso e direto, facilitando a criação de layouts responsivos e consistentes.

- **Classes Utilitárias:** TailwindCSS fornece uma vasta gama de classes utilitárias que cobrem praticamente todas as propriedades CSS. Isso elimina a necessidade de escrever CSS customizado para estilos comuns, permitindo que você construa rapidamente interfaces complexas.
- **Customização Flexível:** Apesar de ser baseado em classes utilitárias, TailwindCSS é altamente personalizável. Você pode configurar temas, cores, espaçamentos e outras propriedades através do arquivo de configuração, adaptando o framework às necessidades do seu projeto.
- **Desempenho e Otimização:** Com TailwindCSS, você pode facilmente purgar classes não utilizadas, resultando em arquivos CSS menores e tempos de carregamento mais rápidos. Isso é especialmente útil em projetos maiores, onde a performance é crucial.

Por que usar TailwindCSS?

1. **Produtividade Aumentada:** Com classes utilitárias prontas para uso, você pode estilizar rapidamente elementos e focar mais no design e na funcionalidade da sua aplicação.
2. **Consistência e Reutilização:** Ao usar uma abordagem baseada em utilitários, você garante uma consistência visual em toda a aplicação. As classes utilitárias também promovem a reutilização de estilos, reduzindo a duplicação de código.
3. **Design Responsivo Facilmente:** Responsividade com TailwindCSS é muito fácil! Ele oferece classes responsivas que facilitam a adaptação do layout para diferentes tamanhos de tela. Isso simplifica a criação de interfaces móveis e acessíveis.
4. **Comunidade e Ecossistema:** TailwindCSS possui uma comunidade ativa e uma vasta quantidade de recursos, plugins e extensões que podem ser integrados ao seu fluxo de trabalho, ampliando ainda mais suas capacidades.

9.6 Configuração do TailwindCSS



Para começar a usar o TailwindCSS, primeiro precisamos configurá-lo em nosso projeto. Vamos passar pelo processo de instalação, configuração e integração com a nossa aplicação. Isso garantirá que estamos prontos para aproveitar todas as funcionalidades que o framework oferece.

Para essa parte do capítulo, vamos construir juntos uma aplicação para mostrar conselhos aleatórios consumindo uma API pública, usando na prática o que aprendemos no capítulo de React.

9.6.1 Instalando o TailwindCSS

Para instalar o TailwindCSS, você precisa ter o Node.js e o npm (ou yarn) configurados no seu ambiente de desenvolvimento. Siga os passos abaixo para adicionar o TailwindCSS ao seu projeto:

Passo 1: Iniciar o Projeto

Se você ainda não possui um projeto, pode criar um novo usando o Create React App:

```
npm create vite@latest app-conselhos -- --template react
cd app-conselhos
```

Passo 2: Instalar o TailwindCSS

Instale o TailwindCSS e seus dependentes executando o comando:

```
npm install -D tailwindcss postcss autoprefixer
npx tailwindcss init
```

Este comando cria os arquivos `tailwind.config.js` na raiz do seu projeto.

Passo 3: Configurar o PostCSS

O **PostCSS** é uma ferramenta que processa e transforma CSS usando plugins, permitindo funcionalidades como autoprefixação e otimização. No contexto do

TailwindCSS, o PostCSS é essencial para interpretar as diretivas do Tailwind e gerar o CSS final que será utilizado na aplicação.

Para isso, crie um arquivo na raiz do projeto chamado `postcss.config.js` e adicione o seguinte conteúdo nele:

```
export default {
  plugins: {
    tailwindcss: {},
    autoprefixer: {}
  }
}
```

Passo 4: Configurar o TailwindCSS

Abra o arquivo `tailwind.config.js` e adicione os caminhos dos seus arquivos onde você utilizará as classes Tailwind:

```
export default {
  content: [
    "./src/**/*.{html,js,jsx,ts,tsx}",
  ],
  theme: {
    extend: {},
  },
  plugins: []
}
```

Passo 5: Incluir o TailwindCSS no CSS Principal

No arquivo `src/index.css`, substitua todo o conteúdo do arquivo pelo seguinte:

```
@tailwind base;
@tailwind components;
```

```
@tailwind utilities;
```

Passo 6: Testar a Configuração

Substitua o conteúdo arquivo `src/App.jsx` pelo código a seguir:

```
function App() {
  return (
    <h1 className="text-red-400 text-center text-3xl">Tailwind CSS!</h1>
  );
}

export default App;
```

Agora, execute o servidor de desenvolvimento para garantir que tudo está funcionando corretamente:

```
npm run dev
```

Se tudo estiver configurado corretamente e você ver um texto vermelho bem grande escrito “Tailwind Configurado” no centro da tela, seu projeto estará pronto para usar o TailwindCSS.

Dica: Se você usa o Vscode, procure pela extensão `Tailwind CSS Intellisense`. Ela fornecerá uma gama de funcionalidades para ajudar você a desenvolver com o framework.

9.7 Utilizando Utilitários do TailwindCSS

Agora que temos o TailwindCSS configurado e entendemos os conceitos básicos, vamos explorar alguns dos utilitários mais poderosos que ele oferece. Esses utilitários vão desde classes para layout e tipografia até espaçamentos e cores, permitindo a construção rápida e eficiente de interfaces modernas.

Para este capítulo, recomendo que você consulte a documentação oficial do TailwindCSS, disponível no link a seguir: [TailwindCSS Documentation](#). Isso facilitará a utilização de qualquer classe ou propriedade CSS com a qual você tenha dúvidas, permitindo que você descubra rapidamente como utilizá-las. Com o TailwindCSS, as possibilidades são infinitas.

9.7.1 Utilitários de Layout

Os utilitários de layout do TailwindCSS permitem que você crie layouts complexos de forma simples e direta. Vamos explorar algumas das classes mais utilizadas para layout.

Classes de Display

As classes de display controlam como um elemento é exibido na página:

```
<div class="block">Block</div>
<div class="inline">Inline</div>
<div class="inline-block">Inline Block</div>
<div class="flex">Flex</div>
<div class="grid">Grid</div>
```

Flexbox

TailwindCSS oferece uma série de utilitários para trabalhar com Flexbox, facilitando a criação de layouts flexíveis e responsivos:

```
<div class="flex">
  <div class="flex-1">Flex 1</div>
  <div class="flex-1">Flex 2</div>
</div>
<div class="flex items-center justify-center">
  <div>Centered</div>
</div>
<div class="flex items-center flex-wrap">
  <div>Wrap</div>
```

```
</div>
<div class="flex gap-4 flex-col">
  <div>Colunas</div>
</div>
```

Grid

Além de Flexbox, o TailwindCSS possui suporte completo para layouts de grade (grid), permitindo a criação de layouts ainda mais complexos:

```
<!-- 3 colunas, cada div ocupando uma coluna -->
<div class="grid grid-cols-3 gap-4">
  <div class="h-12 bg-red-400">Grid 1</div>
  <div class="h-12 bg-green-400">Grid 2</div>
  <div class="h-12 bg-blue-400">Grid 3</div>
</div>

<!-- 12 colunas, com a primeira div ocupando 10 e as outras duas ocupando as restantes -->
<div class="grid grid-cols-12 gap-4">
  <div class="col-span-10 h-12 bg-red-400">Grid 1</div>
  <div class="col-span-1 h-12 bg-green-400">Grid 2</div>
  <div class="col-span-1 h-12 bg-blue-400">Grid 3</div>
</div>
```

9.7.2 Utilitários de Tipografia

Os utilitários de tipografia no TailwindCSS permitem controlar a aparência do texto com precisão, desde o tamanho e peso da fonte até o espaçamento entre linhas.

Tamanho da Fonte

Controle o tamanho da fonte utilizando classes como:

```
<p class="text-xs">Texto Muito Pequeno</p>
<p class="text-sm">Texto Pequeno</p>
```

```
<p class="text-base">Texto Normal</p>
<p class="text-lg">Texto Grande</p>
<p class="text-xl">Texto Muito Grande</p>
<p class="text-2xl">Texto Maior Ainda</p>
```

Peso da Fonte

Ajuste o peso da fonte com as classes correspondentes:

```
<p class="font-thin">Thin</p>
<p class="font-light">Light</p>
<p class="font-normal">Normal</p>
<p class="font-bold">Bold</p>
<p class="font-black">Black</p>
```

Espaçamento entre Linhas

Altere o espaçamento entre linhas para melhorar a legibilidade:

```
<p class="leading-none">Nenhum espaçamento</p>
<p class="leading-tight">Espaçamento apertado</p>
<p class="leading-normal">Espaçamento normal</p>
<p class="leading-loose">Espaçamento solto</p>
```

Transformação de Texto

Utilize classes para transformar o texto, como capitalização ou formatação de caso:

```
<p class="uppercase">Uppercase</p>
<p class="lowercase">Lowercase</p>
<p class="capitalize">Capitalize</p>
<p class="normal-case">Normal Case</p>
```

9.7.3 Utilitários de Espaçamento

TailwindCSS facilita o controle de margens e padding com classes utilitárias para ajustar rapidamente o espaçamento ao redor dos elementos.

Margem

Aplique margens de diferentes tamanhos com classes específicas:

```
<div class="m-4">Margem de 4 unidades</div>
<div class="mt-2">Margem superior de 2 unidades</div>
<div class="mr-2">Margem direita de 2 unidades</div>
<div class="mb-2">Margem inferior de 2 unidades</div>
<div class="ml-2">Margem esquerda de 2 unidades</div>
```

Padding

Ajuste o padding interno dos elementos de forma semelhante:

```
<div class="p-4">Padding de 4 unidades</div>
<div class="pt-2">Padding superior de 2 unidades</div>
<div class="pr-2">Padding direito de 2 unidades</div>
<div class="pb-2">Padding inferior de 2 unidades</div>
<div class="pl-2">Padding esquerdo de 2 unidades</div>
```

Espaçamento Uniforme

Você também pode aplicar espaçamentos uniformes ao redor de um elemento:

```
<div class="m-4 p-4">Margem e padding de 4 unidades</div>
```

9.7.4 Utilitários de Cores

TailwindCSS oferece uma ampla gama de classes para aplicar cores de fundo, texto e bordas.

Cores de Fundo

Aplique cores de fundo aos elementos:

```
<div class="bg-red-500">Fundo vermelho</div>
<div class="bg-green-500">Fundo verde</div>
<div class="bg-blue-500">Fundo azul</div>
```

Cores de Texto

Altere a cor do texto com classes específicas:

```
<p class="text-red-500">Texto vermelho</p>
<p class="text-green-500">Texto verde</p>
<p class="text-blue-500">Texto azul</p>
```

Cores de Borda

Defina cores de borda para seus elementos, com diferentes tamanhos e direções:

```
<div class="border border-red-500">Borda vermelha</div>
<div class="border border-green-500">Borda verde</div>
<div class="border border-blue-500">Borda azul</div>
```

Cores de Sombra

TailwindCSS também inclui utilitários para aplicar sombras (`shadow-`) aos elementos, que ajudam a criar profundidade e destaque visual. As sombras podem ser configuradas com diferentes intensidades e também podem ser personalizadas.

Exemplos de sombras com cores:

- `shadow-sm`: Aplica uma sombra leve ao elemento.

```
<div class="shadow-sm bg-white p-4">
  Sombra Leve
</div>
```

- `shadow-lg`: Aplica uma sombra maior e mais proeminente.

```
<div class="shadow-lg bg-white p-4">
  Sombra Grande
</div>
```

- `shadow-none` : Remove qualquer sombra aplicada ao elemento.

```
<div class="shadow-none bg-white p-4">  
  Sem Sombra  
</div>
```

Com os utilitários de cores do TailwindCSS, você tem o controle total sobre a paleta de cores da sua aplicação, permitindo criar designs que são ao mesmo tempo esteticamente agradáveis e funcionais. As classes predefinidas e a facilidade de customização garantem que você possa adaptar as cores para qualquer projeto, mantendo a simplicidade e a eficiência no desenvolvimento.

9.7.5 Medidas Relativas e Customizadas no TailwindCSS

TailwindCSS oferece uma poderosa flexibilidade para definir tamanhos e dimensões dos elementos utilizando classes utilitárias. Além das medidas predefinidas, como `w-1/2` (50% de largura) ou `h-16` (4rem de altura), você também pode usar medidas customizadas para ajustar com precisão as dimensões dos seus elementos.

Uso de Medidas Customizadas com o Prefixo `[]`

Uma das funcionalidades mais interessantes do TailwindCSS é a capacidade de definir medidas específicas utilizando o prefixo `[]`. Isso permite que você aplique valores exatos de largura

(`w-[]`), altura (`h-[]`), margens (`m-[]`), padding (`p-[]`), e outras propriedades CSS, diretamente nas classes utilitárias.

Exemplo de Largura Customizada

Se você precisar definir uma largura exata de 12px para um elemento, você pode usar a classe `w-[12px]`. O valor entre colchetes é interpretado diretamente como uma unidade de medida CSS, proporcionando uma precisão adicional na estilização.

```
<div class="w-[12px] h-[24px] bg-blue-500">  
  Elemento com largura de 12px e altura de 24px  
</div>
```

Neste exemplo:

- `w-[12px]` : Define a largura do elemento como exatamente 12 pixels.
- `h-[24px]` : Define a altura do elemento como exatamente 24 pixels.
- `bg-blue-500` : Aplica um fundo azul ao elemento.

Outras Unidades de Medida Customizadas

Além de pixels (`px`), você pode usar outras unidades de medida relativas, como:

- `em` : Relativo ao tamanho da fonte do elemento pai.
- `rem` : Relativo ao tamanho da fonte raiz (geralmente 16px por padrão).
- `%` : Relativo ao tamanho do contêiner pai.
- `vw` e `vh` : Relativo à largura e altura da viewport, respectivamente.

Por exemplo, para definir um elemento que ocupa 50% da largura da tela, você pode usar:

```
<div class="w-[50%] h-[50vh] bg-green-500">  
  Elemento que ocupa 50% da largura e 50% da altura da viewport  
</div>
```

Neste caso:

- `w-[50%]` : Define a largura como 50% da largura do contêiner pai.
- `h-[50vh]` : Define a altura como 50% da altura da viewport.

Vantagens das Medidas Customizadas

O uso de medidas customizadas no TailwindCSS é especialmente útil quando você precisa de um controle mais granular sobre as dimensões dos elementos, sem precisar recorrer a CSS personalizado. Isso mantém seu código mais limpo e

integrado ao sistema de utilitários do TailwindCSS, ao mesmo tempo em que oferece a flexibilidade necessária para designs complexos.

Dicas de Uso:

- Utilize medidas customizadas quando as classes predefinidas não oferecem a precisão necessária.
- Combine unidades diferentes (`px`, `rem`, `%`, etc.) para obter layouts mais responsivos e adaptáveis.

Com essa abordagem, TailwindCSS permite que você mantenha o controle total sobre as dimensões e espaçamentos, utilizando medidas customizadas e relativas diretamente nas classes utilitárias, facilitando a criação de interfaces detalhadas e precisas.

9.7.6 Classes Utilitárias para Responsividade

Responsividade é um aspecto crucial no desenvolvimento de interfaces modernas, e o TailwindCSS facilita a criação de layouts que se adaptam a diferentes tamanhos de tela com suas classes utilitárias responsivas. Com TailwindCSS, você pode aplicar estilos específicos para diferentes pontos de quebra (breakpoints) sem a necessidade de escrever CSS adicional.

Pontos de Quebra (Breakpoints) no TailwindCSS

TailwindCSS utiliza os seguintes pontos de quebra por padrão:

- `sm`: Aplica estilos a partir de telas pequenas (640px).
- `md`: Aplica estilos a partir de telas médias (768px).
- `lg`: Aplica estilos a partir de telas grandes (1024px).
- `xl`: Aplica estilos a partir de telas extra grandes (1280px).
- `2xl`: Aplica estilos a partir de telas muito grandes (1536px).

Esses pontos de quebra podem ser utilizados como prefixos nas classes utilitárias para aplicar estilos específicos a diferentes tamanhos de tela.

Exemplos de Uso Responsivo

1. Ajustando a Largura em Diferentes Tamanhos de Tela

Você pode ajustar a largura de um elemento conforme o tamanho da tela aumenta, usando as classes responsivas:

```
<div class="w-full sm:w-1/2 md:w-1/3 lg:w-1/4 xl:w-1/5 bg-blue-500">  
  Elemento Responsivo  
</div>
```

Neste exemplo:

- `w-full`: Em telas menores que 640px, o elemento ocupa 100% da largura.
- `sm:w-1/2`: A partir de 640px, o elemento ocupa 50% da largura.
- `md:w-1/3`: A partir de 768px, o elemento ocupa 33,33% da largura.
- `lg:w-1/4`: A partir de 1024px, o elemento ocupa 25% da largura.
- `xl:w-1/5`: A partir de 1280px, o elemento ocupa 20% da largura.

2. Mudando o Layout com Flexbox

As classes de Flexbox podem ser combinadas com breakpoints para criar layouts que se adaptam dinamicamente:

```
<div class="flex flex-col sm:flex-row bg-gray-200 p-4">  
  <div class="w-full sm:w-1/3 bg-red-500 p-2">Coluna 1</div>  
  <div class="w-full sm:w-1/3 bg-green-500 p-2">Coluna 2</div>  
>  
  <div class="w-full sm:w-1/3 bg-blue-500 p-2">Coluna 3</div>  
</div>
```

Neste exemplo:

- `flex-col`: Em telas menores que 640px, os itens são dispostos em uma coluna.
- `sm:flex-row`: A partir de 640px, os itens são dispostos em uma linha.

3. Controlando a Visibilidade

Você pode mostrar ou ocultar elementos dependendo do tamanho da tela usando classes como `hidden`, `block`, ou `inline-block`:

```
<div class="hidden sm:block md:inline-block lg:hidden">  
  Elemento Responsivo  
</div>
```

Neste exemplo:

- `hidden`: O elemento está oculto em telas menores que 640px.
- `sm:block`: A partir de 640px, o elemento é exibido como um bloco.
- `md:inline-block`: A partir de 768px, o elemento é exibido como inline-block.
- `lg:hidden`: A partir de 1024px, o elemento volta a ser oculto.

Unidades de Medida Relativas para Responsividade

Além das classes utilitárias baseadas em breakpoints, o TailwindCSS permite o uso de unidades de medida relativas como `%`, `vw`, e `vh`, que se adaptam naturalmente ao tamanho da tela:

- `w-[50vw]`: Define a largura como 50% da largura da viewport.
- `h-[75vh]`: Define a altura como 75% da altura da viewport.

Essas unidades são extremamente úteis para criar layouts fluidos que se adaptam perfeitamente a diferentes dispositivos.

Vantagens das Classes Responsivas no TailwindCSS

- **Simplicidade**: As classes responsivas eliminam a necessidade de escrever media queries complexas, tornando o código mais limpo e fácil de manter.
- **Flexibilidade**: Você pode aplicar estilos diferentes para cada ponto de quebra, permitindo a criação de layouts altamente adaptáveis.
- **Rapidez**: A aplicação direta de classes utilitárias torna o desenvolvimento de layouts responsivos rápido e eficiente.

Dicas de Uso:

- Use classes responsivas para ajustar o layout em diferentes dispositivos sem precisar escrever CSS adicional.
- Combine breakpoints com flexbox e grid para criar layouts complexos e adaptáveis.
- Experimente unidades de medida relativas (`vw`, `vh`) para designs que fluem naturalmente com o tamanho da tela.

Com as classes utilitárias para responsividade do TailwindCSS, você tem o poder de criar interfaces que se adaptam perfeitamente a qualquer dispositivo, garantindo uma excelente experiência de usuário em diferentes plataformas e tamanhos de tela.

9.8 Customização e Temas com TailwindCSS

Você percebeu que, ao instalar o TailwindCSS e executa o comando `npx tailwindcss init`, um arquivo chamado `tailwind.config.js` é gerado na raiz do seu projeto. Este arquivo é fundamental para personalizar e estender o comportamento padrão do TailwindCSS, permitindo que você adapte o framework às necessidades específicas do seu projeto.

O que é o `tailwind.config.js`?

O `tailwind.config.js` é um arquivo de configuração onde você pode definir e modificar diversas propriedades do TailwindCSS. Ele oferece um alto grau de flexibilidade, permitindo desde ajustes simples como adicionar novas cores ou espaçamentos, até customizações mais avançadas, como criar temas e adicionar plugins personalizados.

Principais Seções do `tailwind.config.js`

1. Content (Conteúdo):

- Esta seção define os arquivos onde o TailwindCSS deve procurar por classes CSS. Isso ajuda a manter o arquivo CSS final o mais enxuto possível, removendo classes que não são utilizadas no seu código.
- Exemplo:

```
export default {
  content: [
    "./src/**/*.{js,jsx,ts,tsx}",
  ],
}
```

2. Theme (Tema):

- A seção `theme` é onde você pode estender ou substituir as configurações padrão do TailwindCSS. Aqui, você pode definir cores, tamanhos de fonte, espaçamentos, bordas, e muito mais.
- Exemplo:

```
export default {
  theme: {
    extend: {
      colors: {
        primary: '#4CAF50',
        secondary: '#8BC34A',
      },
      spacing: {
        '72': '18rem',
      },
    },
  },
}
```

3. Plugins:

- A seção `plugins` permite que você adicione funcionalidades extras ao TailwindCSS, como plugins de tipografia, formulários ou animações personalizadas.

- Exemplo:

```
export default {
  plugins: [
    require('@tailwindcss/forms'),
    require('@tailwindcss/typography'),
  ],
}
```

Como Utilizar o `tailwind.config.js`?

O `tailwind.config.js` é altamente intuitivo e segue uma estrutura de chave-valor simples. Quando você quiser customizar uma parte do TailwindCSS, basta adicionar as propriedades desejadas na seção `theme` ou em outras seções específicas do arquivo.

Por exemplo, se você deseja adicionar novas cores ao seu projeto, basta incluir essas cores no `extend` dentro da seção `theme`:

```
export default {
  theme: {
    extend: {
      colors: {
        customBlue: '#1e3a8a',
        customGreen: '#10b981',
      },
    },
  },
}
```

Depois de salvar as alterações, você pode utilizar essas novas cores diretamente em suas classes utilitárias:

```
<div class="bg-customBlue text-white">
  Texto com fundo azul personalizado
</div>
```

```
</div>
<div class="bg-customGreen text-white">
  Texto com fundo verde personalizado
</div>
```

Vantagens do `tailwind.config.js`

- **Personalização:** Permite ajustar o TailwindCSS de acordo com as necessidades do projeto, sem modificar diretamente os arquivos de origem.
- **Manutenção:** Centraliza as configurações de estilo em um único local, facilitando a manutenção e a consistência do design.
- **Flexibilidade:** Com o `tailwind.config.js`, você pode adaptar o TailwindCSS para projetos de qualquer escala, desde pequenos sites até grandes aplicações.

9.8 Projeto - App de conselhos

Neste tópico, vamos criar um aplicativo web em React que consome uma API pública de conselhos (<https://api.adviceslip.com/advice>) e exibe o conselho em um layout estilizado usando TailwindCSS. Vamos estruturar o código em blocos para facilitar a compreensão. O aplicativo consistirá em um card centralizado que exibe o ID e o texto do conselho, com um botão para gerar um novo conselho ao ser clicado. Utilizaremos um número aleatório no path da API para evitar o cache das requisições.

Passo 1: Estrutura Inicial do Projeto

Vamos utilizar a estrutura do projeto configurada anteriormente. Todo o código será escrito em um único arquivo `App.jsx`. Certifique-se de que o TailwindCSS está corretamente configurado no seu projeto de acordo com o que foi explicado.

Passo 2: Definindo os Estados e Funções

Nesta parte, configuraremos os estados e as funções necessárias para buscar e armazenar os dados do conselho.

```

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

function App() {
  const [advice, setAdvice] = useState('');
  const [adviceId, setAdviceId] = useState('')

  const fetchAdvice = async () => {
    try {
      const randomNumber = Math.floor(Math.random() * 100);
      const response = await fetch(`https://api.adviceslip.com/advice/${randomNumber}`);
      const data = await response.json();

      setAdvice(data.slip.advice);
      setAdviceId(data.slip.id);
    } catch (error) {
      console.error("Erro ao buscar conselho:", error);
    }
  };

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

```

Explicação:

- **Estados (`useState`):** `advice` armazena o texto do conselho e `adviceId` armazena o ID do conselho.
- `fetchAdvice`: Esta função faz a requisição à API. Um número aleatório entre 0 e 100 é gerado e adicionado ao path da URL, garantindo que cada requisição seja única.
- `useEffect`: Executa a função `fetchAdvice` uma vez, logo após o componente ser montado, para buscar o primeiro conselho.

Passo 3: Estrutura HTML e Layout do Componente

Agora, vamos criar o layout do card utilizando TailwindCSS. Este card será centralizado na tela, com bordas arredondadas, sombra, e exibirá o ID do conselho, o texto do conselho e um botão para gerar um novo.

```
return (
  <div className="flex justify-center items-center h-screen bg-gray-100">
    <div className="bg-white p-6 rounded-lg shadow-lg w-80 text-center">
      <span className="text-gray-500 text-sm">Conselho #{adviceId}</h1>
      <p className="text-lg font-semibold my-4">{advice}</p>
    <br/>
    <button
      onClick={fetchAdvice}
      className="bg-blue-500 text-white px-4 py-2 rounded-full hover:bg-blue-600 transition duration-300"
    >
      Gerar Novo Conselho
    </button>
  </div>
</div>
);

}

export default App;
```

Explicação:

- **Contêiner Principal (`div`)**: A `div` de nível superior usa `flex` para centralizar o card na tela vertical e horizontalmente. O fundo da tela é cinza claro (`bg-gray-100`).
- **Card (`div`)**: O card é um elemento `div` com fundo branco (`bg-white`), padding (`p-6`), bordas arredondadas (`rounded-lg`), e sombra (`shadow-lg`). A largura do

card é definida como 80 unidades (`w-80`).

- **Id do conselho (`span`):** Exibe o ID do conselho com uma cor cinza suave (`text-gray-500`) e um tamanho de fonte pequeno (`text-sm`).
- **Texto do Conselho (`p`):** O texto do conselho é exibido com um tamanho de fonte maior (`text-lg`), estilo semibold (`font-semibold`), e um espaçamento vertical (`my-4`).
- **Botão (`button`):** O botão para gerar um novo conselho usa um fundo azul (`bg-blue-500`), texto branco (`text-white`), padding (`px-4 py-2`), bordas arredondadas (`rounded-full`), e um efeito de hover para escurecer o fundo (`hover:bg-blue-600`). A transição suave é adicionada com `transition duration-300`.

Resultado Esperado

Quando o usuário abrir o aplicativo, verá um card centralizado com o ID e o texto de um conselho. Ao clicar no botão "Gerar Novo Conselho", um novo conselho será buscado e exibido no card.

Desafios para Melhorar o App

Agora que você criou a versão básica do app de conselhos, aqui estão três desafios para você aprimorar o projeto:

1. Adicionar Animações ao Card:

- Adicione uma animação ao card quando um novo conselho for gerado, como um fade-in ou um efeito de deslize. Consulte a documentação de animações do TailwindCSS para inspiração: [TailwindCSS Animations](#).

2. Melhorar a Responsividade:

- Ajuste o layout do card para melhorar a responsividade em diferentes tamanhos de tela. Experimente usar breakpoints para modificar o tamanho do card em dispositivos móveis e tablets. Veja mais sobre breakpoints em: [TailwindCSS Responsive Design](#).

3. Personalizar Temas de Cores:

- Crie um tema claro e escuro para o aplicativo. Adicione um botão que permita ao usuário alternar entre os temas, alterando as cores de fundo, texto e botões. Consulte a documentação para ajudar: [TailwindCSS Dark Mode](#).
-

Com esses desafios, você poderá explorar ainda mais as capacidades do TailwindCSS e aprimorar suas habilidades em criação de interfaces interativas e responsivas. Boa sorte e divirta-se desenvolvendo!

módulo 10

typescript

Dê um upgrade ao seu JavaScript com tipagem estática e mais segurança no código usando TypeScript. 🔑

06 tópicos neste módulo

Capítulo 10: TypeScript

Introdução

Neste capítulo, exploraremos os fundamentos do TypeScript, uma linguagem de programação que estende o JavaScript com tipagem estática opcional e recursos avançados de orientação a objetos, vamos passar por todos os conceitos mais importantes da linguagem, para prepará-lo para os próximos capítulos, onde iremos colocar o TypeScript na prática com NestJS e React!

10.1 Introdução ao TypeScript e Configuração de Ambiente

Bem-vindo ao mundo do TypeScript!

O TypeScript é uma linguagem de programação que se baseia no JavaScript, oferecendo tipagem estática opcional e recursos de orientação a objetos. Ele foi criado para ajudar os desenvolvedores a escreverem códigos mais seguros e mais fáceis de manter, detectando erros antes mesmo de rodar o código. Neste capítulo, vamos introduzir o TypeScript e configurar o ambiente de desenvolvimento para que você possa começar a explorar suas funcionalidades.

10.1.1 O que é TypeScript?

O TypeScript é um superset (superconjunto) de JavaScript, o que significa que todo código JavaScript válido também é código TypeScript válido. A principal diferença é que o TypeScript adiciona tipos ao JavaScript, permitindo a verificação de tipos em tempo de compilação.

Principais vantagens do TypeScript:

- **Tipagem Estática:** Permite definir tipos de variáveis, funções e objetos, ajudando a prevenir erros comuns de tipo.
- **Verificação de Erros em Tempo de Compilação:** Identifica erros antes mesmo de o código ser executado, aumentando a confiabilidade do código.

- **Melhor Suporte a IDEs:** Com TypeScript, as ferramentas de desenvolvimento, como o VSCode, oferecem um autocompletar mais inteligente e mensagens de erro mais úteis.
- **Escalabilidade:** Ideal para grandes projetos, onde a consistência e a manutenção são cruciais.

Funcionamento Interno do TypeScript

Para entender melhor como o TypeScript funciona internamente, é necessário explorar alguns componentes chave do compilador TypeScript e o processo de transformação de código TypeScript em código JavaScript.

Compilador TypeScript (`tsc`)

O `tsc` é o compilador oficial do TypeScript. Ele lê arquivos TypeScript (`.ts`) e os converte em arquivos JavaScript (`.js`). O `tsc` realiza várias tarefas durante esse processo:

1. Leitura e Análise de Arquivos:

O compilador começa lendo o arquivo de configuração

`tsconfig.json`, que define o comportamento do compilador e os arquivos a serem incluídos na compilação. Em seguida, ele lê os arquivos TypeScript especificados.

2. Parsing (Análise Sintática):

O compilador converte o código TypeScript em uma estrutura de dados intermediária chamada Árvore de Sintaxe Abstrata (AST). A AST representa a estrutura do código em uma forma que o compilador pode entender e manipular.

3. Checagem de Tipos:

Uma das principais vantagens do TypeScript é a verificação de tipos. Durante essa fase, o compilador verifica se os tipos são usados corretamente. Ele compara os tipos declarados com os tipos inferidos pelo compilador e sinaliza qualquer discrepância ou erro.

4. Transformações:

O compilador então aplica várias transformações na AST para convertê-la em código JavaScript. Essas transformações podem incluir:

- Remoção de tipos e interfaces, já que o JavaScript não possui tipagem estática.
- Conversão de sintaxe TypeScript específica, como enums e generics, para sintaxe JavaScript equivalente.
- Adição de código auxiliar para suportar recursos avançados do TypeScript, como decorators.

5. Emissão de Código:

Finalmente, o compilador gera o código JavaScript a partir da AST transformada e escreve os arquivos `.js` no diretório especificado (geralmente `./dist`).

Ferramentas de Suporte

Além do compilador `tsc`, o TypeScript oferece várias ferramentas que melhoram a experiência de desenvolvimento:

- **Language Service:**

O serviço de linguagem do TypeScript fornece recursos avançados de IDE, como autocompletar, navegação de código, renomeação de símbolos, e verificação de erros em tempo real. Ele é integrado em editores como Visual Studio Code.

- **Linter:**

Ferramentas como

`TSLint` (agora descontinuado) e `ESLint` com suporte para TypeScript ajudam a garantir que o código siga padrões de estilo e boas práticas, além de detectar problemas potenciais antes mesmo da compilação.

Interoperação com JavaScript

O TypeScript é projetado para interoperar perfeitamente com JavaScript. Isso significa que você pode gradualmente migrar um projeto JavaScript para

TypeScript, convertendo um arquivo de cada vez. O TypeScript permite a importação de módulos JavaScript e o uso de bibliotecas JavaScript existentes, fornecendo definições de tipos através de arquivos `@types`.

10.1.2 Configuração do Ambiente de Desenvolvimento

Antes de começarmos a escrever código TypeScript, precisamos configurar o ambiente de desenvolvimento. Vamos instalar as ferramentas necessárias e criar uma estrutura básica de projeto.

Instalando o TypeScript

Primeiro, vamos instalar o TypeScript globalmente no seu sistema. Abra o terminal e execute o seguinte comando:

```
npm install -g typescript
```

Depois de instalado, você pode verificar a versão instalada com:

```
tsc --version
```

Criando um Projeto TypeScript

Agora, vamos configurar um projeto TypeScript básico.

1. Crie uma pasta para o seu projeto:

```
mkdir meu-projeto-typescript  
cd meu-projeto-typescript
```

2. Initialize o projeto com o Node.js:

```
npm init -y
```

3. Crie o arquivo de configuração do TypeScript:

O arquivo `tsconfig.json` é onde você configura as opções do compilador TypeScript. Para criar esse arquivo automaticamente, execute:

```
tsc --init
```

Isso gerará um arquivo `tsconfig.json` com as configurações básicas.

Entendendo o `tsconfig.json`

O arquivo `tsconfig.json` contém várias opções de configuração. Vamos destacar as principais:

```
{
  "compilerOptions": {
    "target": "es6", // Especifica a versão do JavaScript para a qual o TypeScript será compilado
    "module": "commonjs", // Define o sistema de módulos
    "strict": true, // Habilita verificações estritas de tipo
    "outDir": "./dist", // Diretório onde o código compilado será gerado
    "rootDir": "./src", // Diretório raiz para os arquivos TypeScript
    "esModuleInterop": true, // Garante a compatibilidade com módulos ES
    "forceConsistentCasingInFileNames": true // Enforce case-sensitive imports
  },
  "include": ["src/**/*"], // Inclui todos os arquivos TypeScript dentro do diretório src
  "exclude": ["node_modules", "**/*.test.ts"] // Exclui node_modules e arquivos de teste
}
```

Estrutura Básica do Projeto

Vamos criar uma estrutura básica para o projeto:

1. Crie os diretórios necessários:

```
mkdir src
```

2. Crie um arquivo TypeScript básico:

Dentro da pasta `src`, crie um arquivo `index.ts`:

`src/index.ts`:

```
const saudacao: string = 'Olá, TypeScript!';
console.log(saudacao);
```

Compilando e Executando o Código TypeScript

Para compilar o código TypeScript, utilize o comando:

```
npx tsc
```

Isso gerará um arquivo JavaScript equivalente na pasta `dist` (conforme definido no `tsconfig.json`).

Para executar o código JavaScript compilado:

```
node dist/index.js
```

Agora você verá a mensagem "Olá, TypeScript!" no terminal.

10.1.3 Rodando o Código com `ts-node-dev`

Para facilitar o desenvolvimento, podemos utilizar a ferramenta `ts-node-dev`. Esta ferramenta permite rodar o código TypeScript diretamente sem a necessidade de

compilar manualmente a cada mudança, além de reiniciar o servidor automaticamente quando houver alterações no código.

Instalando o `ts-node-dev`

Primeiro, vamos instalar o `ts-node-dev` como uma dependência de desenvolvimento:

```
npm install ts-node-dev --save-dev
```

Configurando o Script no `package.json`

Em seguida, adicione um script no `package.json` para facilitar a execução do projeto:

```
"scripts": {  
  "dev": "ts-node-dev --respawn --transpile-only src/index.ts"  
}
```

Executando o Projeto com `ts-node-dev`

Agora, você pode rodar o projeto utilizando o comando:

```
npm run dev
```

Sempre que você fizer alterações no código TypeScript, o `ts-node-dev` irá reiniciar automaticamente o servidor, refletindo as mudanças de forma instantânea. Isso agiliza o ciclo de desenvolvimento e facilita a depuração do código.

10.2 Tipos Básicos em TypeScript

Vamos explorar os principais tipos básicos que o TypeScript oferece. Entender esses tipos é fundamental para tirar o máximo proveito dessa linguagem e garantir que seu código seja mais seguro e fácil de manter.

10.2.1 Boolean

O tipo `boolean` é usado para representar valores lógicos, ou seja, `true` ou `false`. Este tipo é útil em operações condicionais e controle de fluxo.

```
let isActive: boolean = true;
```

No exemplo acima, a variável `isActive` pode ser usada para verificar se uma determinada funcionalidade está ativa ou não.

10.2.2 Number

O tipo `number` é usado para representar valores numéricos, tanto inteiros quanto de ponto flutuante.

```
let idade: number = 30;
let preco: number = 19.99;
```

O TypeScript não diferencia entre números inteiros e de ponto flutuante; ambos são tratados como `number`.

10.2.3 String

O tipo `string` é usado para representar cadeias de caracteres. Você pode usar aspas simples (`'`) ou duplas (`"`) para definir strings, além das templates strings com crases (`\``).

```
let nome: string = "Maria";
let saudacao: string = `Olá, ${nome}!`;
```

Templates strings permitem interpolar variáveis dentro da string, tornando o código mais legível.

10.2.4 Array

Arrays em TypeScript podem armazenar uma lista de valores de um determinado tipo. Existem duas formas principais de declarar arrays:

```
let frutas: string[] = ["maçã", "banana", "laranja"];
let numeros: Array<number> = [1, 2, 3, 4, 5];
```

No exemplo acima, `frutas` é um array de strings, enquanto `numeros` é um array de números.

10.2.5 Tuple

Tuplas permitem que você defina um array com um número fixo de elementos, onde os tipos dos elementos são conhecidos e podem ser diferentes.

```
let pessoa: [string, number] = ["João", 25];
```

Aqui, `pessoa` é uma tupla onde o primeiro elemento é uma `string` e o segundo é um `number`.

10.2.6 Enum

Enums são usados para definir um conjunto de valores nomeados. Isso ajuda a tornar o código mais legível e autodescritivo.

```
enum Cor {
  Vermelho,
  Verde,
  Azul,
}
```

```
let minhaCor: Cor = Cor.Verde;
```

Você também pode usar enums baseados em strings:

```
enum Status {  
    Sucesso = "SUCESSO",  
    Falha = "FALHA",  
    Pendente = "PENDENTE",  
}  
  
let meuStatus: Status = Status.Sucesso;
```

10.2.7 Any

O tipo `any` permite que uma variável armazene valores de qualquer tipo. Embora seja flexível, seu uso deve ser cuidadoso, pois desativa a verificação de tipos do TypeScript.

```
let valor: any = "Olá";  
valor = 42; // Isso é permitido com `any`
```

10.2.8 Void

O tipo `void` é usado principalmente em funções que não retornam valor.

```
function logMensagem(mensagem: string): void {  
    console.log(mensagem);  
}
```

10.2.9 Null e Undefined

Os tipos `null` e `undefined` são usados para indicar a ausência de valor. Em TypeScript, `null` é frequentemente usado para indicar um valor vazio, enquanto `undefined` é usado para indicar que uma variável foi declarada, mas ainda não recebeu um valor.

```
let valorNulo: null = null;  
let valorIndefinido: undefined = undefined;
```

Esses tipos são úteis para lidar com casos em que valores podem estar ausentes ou não definidos.

Exercícios Práticos

Agora que você conhece os tipos básicos do TypeScript, é hora de praticar!

- Declare Variáveis com Diferentes Tipos:** Crie variáveis utilizando cada um dos tipos abordados: `boolean`, `number`, `string`, `array`, `tuple`, `enum`, `any`, `void`, `null`, e `undefined`.
- Explore Enums:** Crie um enum para representar diferentes estados de um pedido (por exemplo, `Pendente`, `Enviado`, `Entregue`) e use-o em uma variável para rastrear o status de um pedido.
- Manipule Arrays e Tuplas:** Experimente adicionar, remover e acessar elementos em arrays e tuplas. Veja como o TypeScript ajuda a evitar erros de tipo.
- Use `any` e `void` de Forma Segura:** Crie exemplos onde o tipo `any` pode ser útil, mas também identifique onde ele pode causar problemas. Crie funções com retorno `void` e entenda quando usá-las.

10.3 Arrays

Os arrays são uma das estruturas de dados mais utilizadas em qualquer linguagem de programação, e com TypeScript não é diferente. No TypeScript, os

arrays são tipados, o que significa que você pode (e deve) especificar o tipo de dados que o array irá armazenar. Isso ajuda a evitar erros ao tentar armazenar ou acessar elementos.

Neste tópico, vamos nos aprofundar nas diferentes formas de trabalhar com arrays no TypeScript, desde a criação básica até operações mais avançadas.

10.3.1 Declaração e Inicialização de Arrays

A forma mais simples de declarar um array é especificar o tipo dos elementos que ele conterá:

```
let numeros: number[] = [1, 2, 3, 4, 5];
let frutas: string[] = ["maçã", "banana", "laranja"];
```

Você também pode usar a sintaxe de `Array<T>` para declarar arrays:

```
let ids: Array<number> = [101, 102, 103];
```

Ambas as formas são equivalentes, mas a primeira é mais comum e mais fácil de ler.

10.3.2 Acessando Elementos de um Array

Os elementos de um array podem ser acessados utilizando o índice correspondente, que começa em `0`:

```
let primeiraFruta: string = frutas[0]; // "maçã"
console.log(primeiraFruta); // Saída: "maçã"
```

Se você tentar acessar um índice que não existe, o TypeScript avisará que o valor pode ser `undefined`:

```
let frutaInexistente: string | undefined = frutas[10];
```

10.3.3 Adicionando e Removendo Elementos

Você pode adicionar elementos a um array utilizando o método `push`:

```
frutas.push("uva");
```

E remover o último elemento com `pop`:

```
let ultimaFruta: string | undefined = frutas.pop();
console.log(ultimaFruta); // Saída: "uva"
```

Para adicionar ou remover elementos no início do array, use `unshift` e `shift`, respectivamente:

```
frutas.unshift("morango"); // Adiciona "morango" ao início
let primeiraRemovida: string | undefined = frutas.shift(); // Remove o primeiro elemento
```

10.3.4 Iterando sobre Arrays

Há várias maneiras de iterar sobre os elementos de um array em TypeScript:

- **For loop tradicional:**

```
for (let i = 0; i < frutas.length; i++) {
    console.log(frutas[i]);
}
```

- **For...of loop:**

```
for (let fruta of frutas) {
    console.log(fruta);
}
```

- **Método forEach:**

```
frutas.forEach((fruta) => console.log(fruta));
```

Cada método tem sua utilidade dependendo do que você deseja fazer.

10.3.5 Arrays Multidimensionais

O TypeScript permite a criação de arrays multidimensionais, ou seja, arrays dentro de arrays:

```
let matriz: number[][] = [
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9],
];
console.log(matriz[1][2]); // Saída: 6
```

Aqui, `matriz` é um array 2D, e você pode acessar elementos usando dois índices.

10.3.6 Métodos Úteis para Arrays

O TypeScript herda todos os métodos de manipulação de arrays do JavaScript.

Aqui estão alguns dos mais úteis:

- **concat:** Junta dois ou mais arrays:

```
let outrosNumeros: number[] = [6, 7, 8];
let todosOsNumeros: number[] = numeros.concat(outrosNumeros);
```

- **slice:** Extrai uma seção de um array sem modificar o original:

```
let algunsNumeros: number[] = todosOsNumeros.slice(1, 4);
```

- **splice:** Remove ou adiciona elementos em uma posição específica:

```
todosOsNumeros.splice(2, 1); // Remove 1 elemento na posição 2
```

```
todosOsNumeros.splice(2, 0, 9); // Adiciona o número 9 na posição 2
```

- **map:** Cria um novo array aplicando uma função a cada elemento do array original:

```
let numerosDobrados: number[] = numeros.map((numero) => numero * 2);
```

- **filter:** Cria um novo array com todos os elementos que passaram em um teste específico:

```
let numerosMaioresQueTres: number[] = numeros.filter((numero) => numero > 3);
```

- **reduce:** Aplica uma função a um acumulador e a cada elemento do array (da esquerda para a direita), resultando em um único valor:

```
let soma: number = numeros.reduce((total, numero) => total + numero, 0);
```

10.3.7 Exercícios Práticos

Para consolidar o que você aprendeu sobre arrays, tente resolver os exercícios a seguir:

1. **Crie e Manipule Arrays Simples:** Crie um array de números e adicione, remova e modifique elementos usando os métodos discutidos.

- 2. Itere Sobre Arrays de Diferentes Formas:** Experimente iterar sobre um array usando `for`, `for...of`, e `forEach`, e observe as diferenças.
 - 3. Trabalhe com Arrays Multidimensionais:** Crie um array 2D que represente uma matriz de 3×3 . Acesse e modifique alguns dos elementos.
 - 4. Utilize Métodos de Manipulação de Arrays:** Use métodos como `map`, `filter`, e `reduce` para transformar um array de números. Por exemplo, dobre todos os valores de um array e depois filtre apenas os números maiores que 10.
 - 5. Combine Arrays:** Crie dois arrays de strings e combine-os usando `concat`. Em seguida, extraia uma subseção do array combinado usando `slice`.
-

10.4 Funções

As funções são um dos pilares fundamentais da programação, permitindo a reutilização de código e a organização lógica dos programas. No TypeScript, as funções recebem um tratamento especial, com a possibilidade de definir explicitamente os tipos de seus parâmetros e retorno, proporcionando um maior controle e segurança sobre o comportamento da função.

Neste tópico, vamos explorar como criar e usar funções no TypeScript, além de destacar as principais diferenças em relação ao JavaScript.

10.4.1 Definindo Funções com Tipos

No TypeScript, você pode (e deve) definir os tipos dos parâmetros e do valor de retorno de uma função. Isso ajuda a evitar erros comuns, como passar valores incorretos para uma função ou retornar um tipo de dado inesperado.

Sintaxe básica:

```
function saudacao(nome: string): string {  
    return `Olá, ${nome}!`;  
}
```

```
const mensagem: string = saudacao("Maria");
console.log(mensagem); // Saída: "Olá, Maria!"
```

No exemplo acima, a função `saudacao` recebe um parâmetro `nome` do tipo `string` e retorna uma `string`. Se você tentar passar um valor de outro tipo, o TypeScript vai gerar um erro.

10.4.2 Parâmetros Opcionais e Valores Padrão

No TypeScript, você pode definir parâmetros opcionais usando o símbolo `?` após o nome do parâmetro. Parâmetros opcionais são aqueles que o usuário não é obrigado a fornecer ao chamar a função.

```
function apresentar(nome: string, idade?: number): string {
    if (idade) {
        return `Meu nome é ${nome} e eu tenho ${idade} anos.`;
    } else {
        return `Meu nome é ${nome}.`;
    }
}

console.log(apresentar("João")); // Saída: "Meu nome é João."
console.log(apresentar("João", 25)); // Saída: "Meu nome é João e eu tenho 25 anos."
```

Você também pode definir valores padrão para parâmetros:

```
function multiplicar(a: number, b: number = 2): number {
    return a * b;
}

console.log(multiplicar(5)); // Saída: 10
console.log(multiplicar(5, 3)); // Saída: 15
```

10.4.3 Funções Anônimas e Arrow Functions

Funções anônimas são funções que não têm nome e são frequentemente usadas como argumentos para outras funções. No TypeScript, você pode atribuir uma função anônima a uma variável.

Função Anônima:

```
const somar = function (a: number, b: number): number {  
    return a + b;  
};  
  
console.log(somar(10, 20)); // Saída: 30
```

Arrow Functions:

As arrow functions são uma sintaxe mais concisa para escrever funções anônimas. Elas também mantêm o contexto do `this` do local onde foram definidas.

```
const subtrair = (a: number, b: number): number => {  
    return a - b;  
};  
  
console.log(subtrair(10, 5)); // Saída: 5
```

Quando a função tem apenas uma expressão e retorna um valor, você pode omitir as chaves `{}` e o `return`:

```
const dividir = (a: number, b: number): number => a / b;  
  
console.log(dividir(10, 2)); // Saída: 5
```

10.4.4 Tipos de Função

No TypeScript, você pode definir explicitamente o tipo de uma função, incluindo os tipos de seus parâmetros e de seu retorno.

Definindo um Tipo de Função:

```
let calculadora: (a: number, b: number) => number;

calculadora = (x, y) => x + y;
console.log(calculadora(10, 20)); // Saída: 30

calculadora = (x, y) => x * y;
console.log(calculadora(10, 20)); // Saída: 200
```

Aqui, `calculadora` é uma variável que pode armazenar qualquer função que receba dois números e retorne um número.

10.4.5 Funções que Retornam `void`, `never` e `unknown`

- `void`: Usado para funções que não retornam um valor.

```
function logMensagem(mensagem: string): void {
    console.log(mensagem);
}
```

- `never`: Usado para funções que nunca retornam, como funções que lançam exceções ou entram em loops infinitos.

```
function erroFatal(mensagem: string): never {
    throw new Error(mensagem);
}
```

- `unknown`: Usado para funções que podem retornar qualquer tipo, mas requer verificação de tipo antes de uso.

```
function retornarValor(valor: unknown): string {
    if (typeof valor === "string") {
        return valor;
```

```
    } else {
        return "Valor não é uma string";
    }
}
```

10.4.6 Sobrecarga de Funções

O TypeScript permite a sobrecarga de funções, onde você pode definir várias assinaturas para a mesma função, adaptando-a para diferentes tipos de argumentos.

```
function juntar(a: string, b: string): string;
function juntar(a: number, b: number): number;
function juntar(a: any, b: any): any {
    return a + b;
}

console.log(juntar(1, 2)); // Saída: 3
console.log(juntar("Olá, ", "mundo!")); // Saída: "Olá, mundo!"
```

Aqui, a função `juntar` pode ser usada tanto para concatenar strings quanto para somar números.

10.4.7 Exercícios Práticos

Agora é hora de praticar o que aprendemos sobre funções no TypeScript:

- 1. Crie Funções com Tipagem Completa:** Defina funções que recebam parâmetros de diferentes tipos e retornem valores tipados. Experimente usar valores padrão e parâmetros opcionais.
- 2. Explore Funções Anônimas e Arrow Functions:** Reescreva funções tradicionais como arrow functions e observe as diferenças de sintaxe e comportamento.

3. **Implemente Tipos de Funções:** Crie variáveis que armazenam funções com tipos definidos e troque as implementações das funções sem alterar os tipos.
 4. **Sobrecarga de Funções:** Implemente uma função com múltiplas assinaturas que possa ser chamada com diferentes tipos de argumentos e produza resultados variados.
 5. **Trabalhe com `void`, `never` e `unknown`:** Crie exemplos de funções que retornam `void` e `never`. Experimente usar `unknown` e veja como ele difere de `any`.
-

10.5 Interfaces e Tipos Customizados

Interfaces e tipos customizados são ferramentas poderosas no TypeScript que permitem definir a forma dos objetos, garantir consistência no código e melhorar a legibilidade. Esses conceitos são fundamentais para escrever código escalável e fácil de manter.

Neste tópico, vamos nos aprofundar em como criar e utilizar interfaces e tipos customizados, explorando suas diferenças e quando usar cada um.

10.5.1 Interfaces

Interfaces são usadas para definir a estrutura de um objeto, especificando quais propriedades ele deve ter e quais tipos essas propriedades devem ser. Interfaces são especialmente úteis para garantir que objetos e classes sigam um contrato específico.

Definindo uma Interface:

```
interface Usuario {  
    nome: string;  
    idade: number;  
    email?: string; // Propriedade opcional  
}
```

```
const usuario1: Usuario = {
    nome: "Ana",
    idade: 28,
};
```

Aqui, `Usuario` é uma interface que define que todo objeto desse tipo deve ter `nome` e `idade` como obrigatórios e `email` como opcional.

Implementando Interfaces em Classes:

Interfaces também podem ser implementadas em classes para garantir que a classe siga uma determinada estrutura:

```
class Pessoa implements Usuario {
    nome: string;
    idade: number;

    constructor(nome: string, idade: number) {
        this.nome = nome;
        this.idade = idade;
    }

    saudar(): string {
        return `Olá, meu nome é ${this.nome} e tenho ${this.idade} anos.`;
    }
}

const pessoa1 = new Pessoa("Carlos", 35);
console.log(pessoa1.saudar()); // Saída: "Olá, meu nome é Carlos e tenho 35 anos."
```

10.5.2 Extendendo Interfaces

Interfaces podem ser estendidas para criar novas interfaces que herdam as propriedades de interfaces existentes. Isso permite criar estruturas mais complexas de forma modular.

```
interface Endereco {  
    rua: string;  
    cidade: string;  
}  
  
interface UsuarioComEndereco extends Usuario, Endereco {  
    telefone: string;  
}  
  
const usuario2: UsuarioComEndereco = {  
    nome: "Pedro",  
    idade: 40,  
    rua: "Rua ABC",  
    cidade: "São Paulo",  
    telefone: "1234-5678",  
};
```

Aqui, `UsuarioComEndereco` herda as propriedades de `Usuario` e `Endereco`, além de adicionar uma nova propriedade `telefone`.

10.5.3 Tipos Customizados (`type`)

O TypeScript também permite criar tipos customizados usando a palavra-chave `type`. Esses tipos são mais flexíveis do que interfaces e podem ser usados para definir combinações de tipos, unir diferentes tipos (`union types`) ou mesmo criar tipos literais.

Definindo um Tipo Customizado:

```
type ID = string | number;  
  
let usuarioId: ID;
```

```
usuarioId = 123; // Válido
usuarioId = "abc123"; // Válido
```

Combinando Tipos ([Union Types](#)):

Tipos customizados são úteis para definir tipos que podem ser uma combinação de outros:

```
type StatusPedido = "pendente" | "enviado" | "entregue";

function atualizarStatus(status: StatusPedido): void {
    console.log(`Status atualizado para: ${status}`);
}

atualizarStatus("enviado"); // Válido
atualizarStatus("cancelado"); // Erro: Tipo '"cancelado"' não
é atribuível ao tipo 'StatusPedido'.
```

10.5.4 Diferenças entre Interfaces e Tipos Customizados

Embora interfaces e tipos customizados possam ser usados de maneira intercambiável em muitos casos, existem algumas diferenças importantes:

- **Interfaces podem ser implementadas por classes, enquanto tipos não.**
- **Interfaces podem ser estendidas com outras interfaces usando [extends](#), enquanto tipos usam a interseção ([&](#)).**
- **Interfaces são preferíveis para definir estruturas de objetos, enquanto tipos são mais versáteis para combinações de tipos e literais.**

Exemplo de Interseção ([&](#)):

```
type Admin = {
    nome: string;
    permissao: string;
};
```

```
type UserAdmin = Usuario & Admin;

const admin: UserAdmin = {
  nome: "Carlos",
  idade: 30,
  permissao: "superuser",
};
```

10.5.5 Exercícios Práticos

Agora que você aprendeu sobre interfaces e tipos customizados, é hora de praticar!

- 1. Crie e Implemente Interfaces:** Defina uma interface para um objeto de `Produto` com propriedades como `nome`, `preco`, e `categoria`. Em seguida, crie uma classe que implemente essa interface.
- 2. Extenda Interfaces:** Crie uma interface para `Cliente` e outra para `Pedido`. Extenda `Cliente` para adicionar propriedades específicas de um cliente com endereço e faça o mesmo para `Pedido`, adicionando informações de pagamento.
- 3. Crie Tipos Customizados:** Crie um tipo customizado para IDs que podem ser `string` ou `number`. Use esse tipo em uma função que busca usuários por ID.
- 4. Trabalhe com Union Types:** Crie um tipo que possa representar um `Status` de pedido como "pendente", "enviado" ou "entregue". Crie uma função que aceita esse tipo e retorna uma mensagem personalizada.
- 5. Compare Interfaces e Tipos:** Crie exemplos práticos onde tanto interfaces quanto tipos customizados poderiam ser usados. Decida qual é mais adequado para cada situação e explique por quê.

10.6 Orientação a Objetos

A programação orientada a objetos (POO) é um paradigma amplamente utilizado que organiza o código em torno de "objetos", que são instâncias de classes. TypeScript, como uma extensão tipada do JavaScript, oferece suporte robusto para POO, incluindo conceitos como classes, herança, encapsulamento e polimorfismo.

Neste tópico, vamos explorar em detalhes esses conceitos e ver como implementá-los em TypeScript.

10.6.1 Classes e Objetos

Em TypeScript, uma classe é um molde para criar objetos. Cada objeto criado a partir de uma classe pode ter suas próprias propriedades e métodos.

Definindo uma Classe:

```
class Animal {  
    nome: string;  
    idade: number;  
  
    constructor(nome: string, idade: number) {  
        this.nome = nome;  
        this.idade = idade;  
    }  
  
    emitirSom(): string {  
        return `${this.nome} faz um som!`;  
    }  
}  
  
const cachorro = new Animal("Rex", 5);  
console.log(cachorro.emitirSom()); // Saída: "Rex faz um so  
m!"
```

Aqui, `Animal` é uma classe com duas propriedades (`nome` e `idade`) e um método (`emitirSom`). A palavra-chave `new` é usada para criar uma instância da classe.

10.6.2 Herança

Herança é um conceito da POO onde uma classe pode herdar propriedades e métodos de outra classe. Em TypeScript, isso é feito com a palavra-chave `extends`.

Definindo uma Classe que Herda Outra:

```
class Cachorro extends Animal {
    raca: string;

    constructor(nome: string, idade: number, raca: string) {
        super(nome, idade);
        this.raca = raca;
    }

    emitirSom(): string {
        return `${this.nome}, o ${this.raca}, late!`;
    }
}

const meuCachorro = new Cachorro("Rex", 5, "Labrador");
console.log(meuCachorro.emitirSom()); // Saída: "Rex, o Labra-
dor, late!"
```

Aqui, `Cachorro` herda de `Animal`, mas também adiciona uma nova propriedade `raca` e sobrescreve o método `emitirSom`.

10.6.3 Encapsulamento

Encapsulamento é o princípio de esconder os detalhes internos de um objeto e expor apenas o que é necessário. Em TypeScript, isso é feito usando modificadores de acesso como `public`, `private`, e `protected`.

- `public`: O padrão, acessível de qualquer lugar.

- `private`: Acessível apenas dentro da própria classe.
- `protected`: Acessível dentro da classe e subclasses.

Exemplo de Encapsulamento:

```
class Pessoa {
    private nome: string;

    constructor(nome: string) {
        this.nome = nome;
    }

    public getNome(): string {
        return this.nome;
    }

    private ajustarNome(nome: string): void {
        this.nome = nome.trim();
    }
}

const pessoa1 = new Pessoa(" Maria ");
console.log(pessoa1.getNome()); // Saída: " Maria "
// pessoa1.ajustarNome("Maria") -> Erro: Método privado
```

Neste exemplo, `nome` é privado e não pode ser acessado diretamente de fora da classe. O método `ajustarNome` também é privado, enquanto `getNome` é público.

10.6.4 Polimorfismo

Polimorfismo permite que métodos em classes derivadas tenham a mesma assinatura, mas comportamentos diferentes. Em TypeScript, isso é alcançado através da herança e sobrescrita de métodos.

Exemplo de Polimorfismo:

```

class Gato extends Animal {
    emitirSom(): string {
        return `${this.nome} mia!`;
    }
}

const meuGato = new Gato("Mimi", 3);
console.log(meuGato.emitirSom()); // Saída: "Mimi mia!"

```

Apesar de `emitirSom` ser definido em `Animal`, ele é sobreescrito tanto em `Cachorro` quanto em `Gato`, demonstrando polimorfismo.

10.6.5 Classes Abstratas e Interfaces

Classes abstratas não podem ser instanciadas diretamente e são usadas como base para outras classes. Elas podem conter métodos abstratos que devem ser implementados nas classes derivadas.

Exemplo de Classe Abstrata:

```

abstract class Forma {
    abstract calcularArea(): number;

    exibirArea(): void {
        console.log(`A área é: ${this.calcularArea()}`);
    }
}

class Quadrado extends Forma {
    lado: number;

    constructor(lado: number) {
        super();
        this.lado = lado;
    }
}

```

```

    calcularArea(): number {
      return this.lado * this.lado;
    }
}

const meuQuadrado = new Quadrado(4);
meuQuadrado.exibirArea(); // Saída: "A área é: 16"

```

Interfaces também podem ser usadas em conjunto com classes para definir contratos que as classes devem seguir.

Exemplo de Interface:

```

interface Movel {
  mover(distancia: number): void;
}

class Carro implements Movel {
  mover(distancia: number): void {
    console.log(`O carro moveu ${distancia} metros.`);
  }
}

const meuCarro = new Carro();
meuCarro.mover(10); // Saída: "O carro moveu 10 metros."

```

10.6.6 Exercícios Práticos

Agora que você compreende os principais conceitos de orientação a objetos no TypeScript, é hora de colocar em prática:

- 1. Crie Classes com Propriedades e Métodos:** Defina uma classe `Produto` com propriedades como `nome`, `preco`, e `desconto`, além de métodos que calculam o

preço final após o desconto.

2. **Implemente Herança:** Crie uma classe `ProdutoDigital` que herda de `Produto`, adicionando uma nova propriedade `linkDownload` e sobrescrevendo o método que calcula o preço final.
 3. **Use Encapsulamento:** Reescreva a classe `Produto` para tornar `preco` e `desconto` privados, criando métodos públicos para acessar e modificar essas propriedades de forma controlada.
 4. **Aplique Polimorfismo:** Crie uma classe `ProdutoFisico` que também herda de `Produto`, mas tem uma implementação diferente do cálculo de preço final, considerando o custo de envio.
 5. **Trabalhe com Classes Abstratas:** Defina uma classe abstrata `ContaBancaria` com um método abstrato `calcularJuros` e implemente classes concretas como `ContaPoupanca` e `ContaCorrente`.
-

10.7 Generics

Generics são uma funcionalidade avançada do TypeScript que permite criar componentes reutilizáveis que funcionam com vários tipos de dados. Em vez de definir um tipo específico para uma função, classe ou interface, os Generics permitem que você defina uma "variável de tipo" que pode ser usada com qualquer tipo de dado.

Neste tópico, vamos explorar como usar Generics no TypeScript, desde a definição básica até casos mais complexos, mostrando como eles podem ajudar a criar código mais flexível e reutilizável.

10.7.1 Introdução aos Generics

Os Generics permitem que você crie funções, classes e interfaces que funcionem com tipos variados. A sintaxe básica usa `< >` para definir um tipo genérico.

Exemplo Básico de Generics em Função:

```
function identico<T>(valor: T): T {
    return valor;
}

console.log(identico<string>("Olá")); // Saída: "Olá"
console.log(identico<number>(10)); // Saída: 10
```

Aqui, `T` é um tipo genérico que permite que a função `identico` trabalhe com qualquer tipo de dado. Você pode especificar o tipo ao chamar a função, ou o TypeScript pode inferir o tipo automaticamente.

10.7.2 Generics em Funções

Generics são particularmente úteis em funções que precisam trabalhar com diferentes tipos de dados sem perder a tipagem.

Exemplo de Função Genérica com Arrays:

```
function pegarPrimeiroElemento<T>(arr: T[]): T {
    return arr[0];
}

const numeros = [1, 2, 3];
const palavras = ["a", "b", "c"];

console.log(pegarPrimeiroElemento(numeros)); // Saída: 1
console.log(pegarPrimeiroElemento(palavras)); // Saída: "a"
```

Nesta função, `T` representa o tipo dos elementos do array. A função `pegarPrimeiroElemento` pode ser usada com qualquer tipo de array.

10.7.3 Generics em Interfaces e Classes

Generics também podem ser usados em interfaces e classes para definir contratos e estruturas flexíveis.

Interface Genérica:

```
interface Par<K, V> {
    chave: K;
    valor: V;
}

const item: Par<string, number> = { chave: "idade", valor: 30 };
console.log(item); // Saída: { chave: "idade", valor: 30 }
```

Classe Genérica:

```
class Caixa<T> {
    conteudo: T;

    constructor(conteudo: T) {
        this.conteudo = conteudo;
    }

    exibirConteudo(): void {
        console.log(this.conteudo);
    }
}

const caixaDeNumero = new Caixa<number>(123);
const caixaDeString = new Caixa<string>("Teste");

caixaDeNumero.exibirConteudo(); // Saída: 123
caixaDeString.exibirConteudo(); // Saída: "Teste"
```

Aqui, a classe `Caixa` pode armazenar qualquer tipo de dado, graças ao uso de Generics.

10.7.4 Restrições de Generics

Você pode impor restrições aos Generics para garantir que eles trabalhem apenas com tipos que atendam a certos critérios. Isso é feito com a palavra-chave `extends`.

Exemplo de Restrição de Generics:

```
function obterPropriedade<T, K extends keyof T>(objeto: T, chave: K) {
    return objeto[chave];
}

const pessoa = { nome: "Carlos", idade: 30 };
console.log(obterPropriedade(pessoa, "nome")); // Saída: "Carlos"
console.log(obterPropriedade(pessoa, "altura")); // Erro: 'altura' não existe no tipo 'Pessoa'
```

Neste exemplo, `K` é restrito a ser uma chave do tipo `T`, garantindo que você só possa acessar propriedades existentes no objeto.

10.7.5 Generics e Tipos de União

Generics podem ser combinados com tipos de união (`union types`) para criar funções e classes ainda mais flexíveis.

Exemplo com União de Tipos:

```
function exibirValor<T extends string | number>(valor: T): void {
    console.log(`O valor é: ${valor}`);
}

exibirValor("Olá"); // Saída: "O valor é: Olá"
exibirValor(123); // Saída: "O valor é: 123"
```

```
// exibirValor(true); // Erro: Tipo 'boolean' não é atribuível  
// a 'string | number'
```

Aqui, a função `exibirValor` aceita apenas valores que sejam `string` ou `number`.

10.7.6 Exercícios Práticos

Agora que você entende os fundamentos dos Generics, é hora de praticar:

- 1. Crie uma Função Genérica:** Implemente uma função que possa trabalhar com arrays de diferentes tipos, como strings e números. Experimente adicionar restrições para garantir que os elementos tenham propriedades específicas.
- 2. Implemente uma Interface Genérica:** Defina uma interface para uma estrutura de dados `Par` que armazene pares de chave-valor de qualquer tipo. Crie alguns objetos que implementem essa interface com diferentes tipos.
- 3. Desenvolva uma Classe Genérica:** Crie uma classe `Lista` que permita adicionar, remover e exibir elementos de qualquer tipo. Teste a classe com diferentes tipos de dados.
- 4. Use Restrições em Generics:** Escreva uma função que aceita objetos e retorna o valor de uma propriedade especificada. Garanta que a função só aceite chaves válidas para o objeto.
- 5. Combine Generics com Tipos de União:** Crie uma função genérica que aceite tanto strings quanto números e realize uma operação específica com base no tipo do dado. Por exemplo, se for uma string, a função pode concatenar, e se for um número, a função pode somar.

módulo 11

api com nest.js

Construa APIs robustas e escaláveis com
Nest.js, TypeORM e JWT. 

09 tópicos neste módulo

Capítulo 11: APIs com Nest.js

11.1 Introdução ao Nest.js

Nest.js é um poderoso framework para construir aplicações do lado do servidor usando Node.js. Inspirado por frameworks como Angular, ele adota uma abordagem modular e orientada a objetos, facilitando a criação de aplicações escaláveis, testáveis e de fácil manutenção.

O Nest.js é construído em cima do Express.js (ou Fastify) e usa TypeScript como linguagem principal, embora também suporte JavaScript. Sua arquitetura é baseada em módulos, controladores e serviços, o que permite organizar o código de forma estruturada e eficiente.

Por que usar o Nest.js?

- Escalabilidade:** O Nest.js facilita a construção de aplicações modulares que podem crescer à medida que as necessidades do projeto aumentam.
- Mantenibilidade:** Com uma arquitetura limpa e uso extensivo de boas práticas, como injeção de dependência e separação de responsabilidades, o Nest.js torna o código mais fácil de entender e manter.
- Suporte para TypeScript:** Com suporte completo para TypeScript, o Nest.js oferece tipagem estática, o que ajuda a capturar erros em tempo de desenvolvimento e melhorar a produtividade.
- Ecosistema Rico:** O Nest.js oferece uma ampla gama de ferramentas e pacotes integrados, como o suporte nativo para GraphQL, WebSockets, microservices, e muito mais.
- Comunidade Ativa:** Com uma comunidade vibrante e em constante crescimento, o Nest.js tem uma grande quantidade de recursos, tutoriais e pacotes de terceiros para ajudar no desenvolvimento.

Visão Geral da Arquitetura

- **Módulos:** São a base da aplicação Nest.js. Um módulo é uma unidade coesa que agrupa um conjunto de funcionalidades relacionadas. Cada aplicação Nest.js tem pelo menos um módulo, o AppModule, que é o módulo raiz.
- **Controladores (Controllers):** São responsáveis por lidar com as requisições HTTP e enviar as respostas. Eles se comunicam com os serviços para processar as requisições.
- **Serviços (Services):** Contêm a lógica de negócio e são injetados nos controladores para processar as requisições. Eles são marcados como `@Injectable()` para permitir a injeção de dependência.
- **Repositórios:** Gerenciam a comunicação com o banco de dados, geralmente usando bibliotecas como TypeORM ou Mongoose.
- **Pipes:** São usados para transformar ou validar dados antes de serem processados por controladores ou serviços.
- **Guards:** Servem para proteger rotas, controlando o acesso a determinadas partes da aplicação, geralmente usados para implementar autenticação e autorização.

No decorrer deste capítulo, vamos construir uma API de conversão de moedas, onde vamos explorar cada um desses conceitos na prática. A cada tópico, você verá como incrementar o projeto, aprendendo como cada parte do Nest.js se encaixa para formar uma aplicação robusta.

11.2 Configuração do Ambiente

Para começar a desenvolver uma aplicação com Nest.js, é essencial configurar o ambiente de desenvolvimento corretamente. Neste tópico, vamos passar pelo processo de instalação do Nest CLI, a criação de um novo projeto, e como estruturar o ambiente de desenvolvimento para maximizar a produtividade.

11.2.1 Instalando o Nest CLI

O **Nest CLI** é uma ferramenta poderosa que facilita a criação e gerenciamento de projetos Nest.js. Ele automatiza várias tarefas, como a geração de módulos, serviços, controladores e muito mais.

Para instalar o Nest CLI globalmente no seu sistema, execute o seguinte comando no terminal:

```
npm install -g @nestjs/cli
```

Após a instalação, você pode verificar se tudo está funcionando corretamente executando:

```
nest --version
```

Este comando deve retornar a versão do Nest CLI instalada, confirmando que a instalação foi bem-sucedida.

11.2.2 Criando um Novo Projeto Nest.js

Com o Nest CLI instalado, vamos criar um novo projeto. Execute o comando abaixo para gerar a estrutura inicial da aplicação:

```
nest new api-conversao-moedas
```

O CLI irá solicitar que você escolha o gerenciador de pacotes (NPM ou Yarn) para o projeto. Selecione o que você preferir.

```
> nest new api-conversao-moedas
⚡  We will scaffold your app in a few seconds..

? Which package manager would you ❤️ to use? (Use arrow keys)
> npm
yarn
pnpm
```

Após a conclusão do processo, navegue até o diretório do projeto:

```
cd api-conversao-moedas
```

Agora, você tem a estrutura básica de um projeto Nest.js pronta para ser usada.

11.2.3 Estrutura do Projeto

A estrutura padrão de um projeto Nest.js criado com o CLI é organizada da seguinte forma:

```
api-conversao-moedas/
├── src/
│   ├── app.controller.ts
│   ├── app.module.ts
│   ├── app.service.ts
│   └── main.ts
├── test/
└── node_modules/
    ├── .eslintrc.js
    ├── .gitignore
    ├── nest-cli.json
    ├── package.json
    ├── tsconfig.json
    └── README.md
```

Vamos entender brevemente cada um desses arquivos:

- **src/**: Contém o código-fonte da aplicação. É onde você passará a maior parte do tempo desenvolvendo.
 - **app.controller.ts**: Um exemplo básico de controlador.
 - **app.module.ts**: O módulo raiz da aplicação.
 - **app.service.ts**: Um serviço básico de exemplo.

- **main.ts**: O arquivo de entrada da aplicação, responsável por iniciar o servidor Nest.js.
- **test/**: Diretório para testes automatizados.
- **node_modules/**: Diretório onde todas as dependências do projeto são instaladas.
- **.eslintrc.js**: Arquivo de configuração para o ESLint, uma ferramenta de linting para identificar e reportar padrões problemáticos encontrados no código JavaScript/TypeScript.
- **nest-cli.json**: Arquivo de configuração do Nest CLI.
- **package.json**: Contém informações sobre o projeto, como scripts, dependências e versões.
- **tsconfig.json**: Arquivo de configuração do TypeScript.

11.2.4 Rodando o Servidor

Para iniciar o servidor Nest.js e ver a aplicação rodando, execute o seguinte comando:

```
npm run start:dev
```

Este comando inicializa o servidor em modo de desenvolvimento, permitindo hot reloads sempre que você fizer alterações no código. Por padrão, a aplicação estará acessível em <http://localhost:3000>.

Ao acessar essa URL, você deve ver a mensagem `Hello World!`, indicando que o servidor está funcionando corretamente.

11.3 Módulos

No Nest.js, os **módulos** são uma parte essencial da arquitetura. Eles ajudam a organizar o código em partes coesas e modulares, facilitando a manutenção,

escalabilidade e reutilização. Cada aplicação Nest.js é composta de pelo menos um módulo, chamado de **módulo raiz** (`AppModule`), mas conforme a aplicação cresce, outros módulos podem ser criados para agrupar funcionalidades relacionadas.

11.3.1 O que é um Módulo?

Um módulo no Nest.js é uma classe marcada com o decorador `@Module()`. Essa classe encapsula um conjunto de controladores, serviços e outros módulos que trabalham juntos para fornecer uma funcionalidade específica.

O decorador `@Module()` recebe um objeto de configuração que define os elementos que compõem o módulo:

- **controllers:** As classes que gerenciam as requisições e respostas HTTP.
- **providers:** Serviços e outras classes que serão injetados como dependências nos controladores.
- **imports:** Outros módulos cujos provedores serão usados neste módulo.
- **exports:** Provedores que devem ser acessíveis a outros módulos que importam este módulo.

11.3.2 Gerenciamento de Dependências

No Nest.js, as dependências são gerenciadas automaticamente pelo framework através de **injeção de dependência**. Isso significa que, ao invés de criar instâncias de classes manualmente, você declara as dependências de uma classe e o Nest.js se encarrega de injetá-las onde necessário.

Por exemplo, se um controlador depende de um serviço, você apenas injeta esse serviço no construtor do controlador, e o Nest.js cuida de fornecer a instância correta.

11.3.3 Criando um Módulo para o Projeto

Vamos agora criar um módulo chamado `CurrencyModule`, que será responsável pela lógica de conversão de moedas na nossa API.

Primeiro, no terminal, use o Nest CLI para gerar o módulo:

```
nest generate module currency
```

O comando acima criará uma nova pasta `currency` dentro da pasta `src/`, contendo um arquivo `currency.module.ts`.

currency.module.ts:

```
import { Module } from '@nestjs/common';

@Module({
  controllers: [],
  providers: [],
  exports: []
})
export class CurrencyModule {}
```

Este arquivo contém a estrutura básica de um módulo. Vamos configurar o módulo para começar a construir nossa API de conversão de moedas.

Estrutura inicial do projeto:

```
api-conversao-moedas/
  └── src/
    ├── app.controller.ts
    ├── app.module.ts
    ├── app.service.ts
    └── currency/
      └── currency.module.ts
    └── main.ts
```

Agora, precisamos importar o `CurrencyModule` no `AppModule` para integrá-lo ao módulo raiz da aplicação.

app.module.ts:

```
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';
import { CurrencyModule } from './currency/currency.module';

@Module({
  imports: [CurrencyModule],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}
```

Ao adicionar o `CurrencyModule` ao array `imports`, estamos dizendo ao Nest.js que o módulo raiz deve incluir todas as funcionalidades definidas no `CurrencyModule`.

11.4 Services

Os serviços são uma das principais partes de uma aplicação Nest.js, responsáveis por manter a lógica de negócio da aplicação. Eles são classes que podem ser injetadas em controladores e outros serviços, permitindo que a lógica seja centralizada e reutilizada em toda a aplicação.

11.4.1 O que é um Service?

Um **service** no Nest.js é uma classe que é anotada com o decorator `@Injectable()`. Isso indica ao Nest.js que a classe pode ser injetada em outros componentes do sistema, como controladores, através da injeção de dependência.

A principal função de um serviço é encapsular a lógica de negócio, mantendo os controladores focados em lidar com as requisições e respostas HTTP.

11.4.2 Criando um Service

Vamos criar um serviço chamado `CurrencyService`, que será responsável por lidar com a conversão de moedas na nossa API.

No terminal, dentro do diretório do projeto, execute o comando:

```
nest generate service currency
```

Esse comando criará dois arquivos na pasta `currency`:

- **currency.service.ts**: Aqui é onde a lógica de negócio será implementada.
- **currency.service.spec.ts**: Um arquivo para testes automatizados.

Aqui está um exemplo básico de como podemos implementar o `CurrencyService`:

currency.service.ts:

```
import { Injectable } from '@nestjs/common';

@Injectable()
export class CurrencyService {
  private currencies = [
    { code: 'USD', rate: 1.0 },
    { code: 'EUR', rate: 0.85 },
    { code: 'BRL', rate: 5.25 },
  ];

  getAvailableCurrencies(): string[] {
    return this.currencies.map((currency) => currency.code);
  }

  convertCurrency(amount: number, from: string, to: string):
```

```

number {
  const fromRate = this.currencies.find(
    (currency) => currency.code === from,
  )?.rate;

  const toRate = this.currencies.find(
    (currency) => currency.code === to,
  )?.rate;

  if (!fromRate || !toRate) {
    throw new Error('Currency not found');
  }

  const convertedAmount = (amount / fromRate) * toRate;
  return convertedAmount;
}
}

```

Neste serviço, temos duas funções principais:

- `getAvailableCurrencies` : Retorna uma lista com os códigos das moedas disponíveis.
- `convertCurrency` : Realiza a conversão de um valor entre duas moedas diferentes, usando taxas de câmbio simuladas.

Essas taxas de câmbio são armazenadas em um array dentro do serviço, e a lógica de conversão é simples, mas ilustrativa. Em uma aplicação real, essas taxas de câmbio provavelmente seriam recuperadas de um banco de dados ou de uma API externa.

11.4.3 Injetando Dependências

Os serviços são injetados em outras partes da aplicação, como controladores, através da injeção de dependência. No próximo tópico, vamos explorar como fazer isso criando controladores para lidar com as requisições HTTP.

11.4.4 Exercícios Práticos

1. **Expanda o CurrencyService:** Adicione uma função ao serviço que permita atualizar as taxas de câmbio. Simule a atualização das taxas de câmbio com valores diferentes.
2. **Adicione Mais Moedas:** Expanda a lista de moedas suportadas pelo serviço adicionando pelo menos mais três moedas com suas respectivas taxas de câmbio.
3. **Teste a Lógica de Conversão:** Adicione testes no arquivo `currency.service.spec.ts` para verificar se a lógica de conversão está funcionando corretamente.

Com o `CurrencyService` criado e implementado, você agora tem um componente fundamental da API que gerencia a lógica de conversão de moedas. No próximo tópico, vamos criar controladores que usarão este serviço para lidar com as requisições dos usuários. 

11.5 Controllers

No Nest.js, **controllers** são responsáveis por lidar com as requisições HTTP e enviar as respostas apropriadas. Eles são a camada que interage diretamente com o cliente, geralmente chamando os serviços para realizar operações de negócios e retornando os resultados.

11.5.1 O que é um Controller?

Um **controller** é uma classe que é anotada com o decorador `@Controller()` e que define um conjunto de rotas. Cada método dentro de um controller é mapeado para uma rota específica e um método HTTP (GET, POST, PUT, DELETE, etc.).

Os controladores no Nest.js seguem o padrão de convenção MVC (Model-View-Controller), onde o controlador age como intermediário entre o modelo (dados) e a visão (UI ou API).

11.5.2 Criando um Controller

Vamos criar um controlador chamado `currencyController` para gerenciar as operações de conversão de moedas na nossa API.

No terminal, execute o comando:

```
nest generate controller currency
```

Este comando criará um arquivo `currency.controller.ts` dentro da pasta `currency`.

Vamos agora implementar o `CurrencyController` para lidar com as rotas de conversão de moedas e listar as moedas disponíveis.

currency.controller.ts:

```
import { Controller, Get, Query } from '@nestjs/common';
import { CurrencyService } from './currency.service';

@Controller('currency')
export class CurrencyController {
    constructor(private readonly currencyService: CurrencyService) {}

    @Get('convert')
    convertCurrency(
        @Query('amount') amount: number,
        @Query('from') from: string,
        @Query('to') to: string,
    ): { convertedAmount: number } {
        const convertedAmount = this.currencyService.convertCurrency(
            amount,
            from,
            to
        );
        return { convertedAmount };
    }
}
```

```

        amount,
        from,
        to,
    );
    return { convertedAmount };
}

@Get('list')
getAvailableCurrencies(): string[] {
    return this.currencyService.getAvailableCurrencies();
}

```

Neste controlador, criamos duas rotas principais:

1. `GET /currency/convert`: Esta rota realiza a conversão de um valor de uma moeda para outra. Ela aceita três parâmetros query (`amount`, `from`, `to`) e retorna o valor convertido.
 - Exemplo de uso: `GET /currency/convert?amount=100&from=USD&to=EUR`
 - Resposta: `{ "convertedAmount": 85 }`
2. `GET /currency/list`: Esta rota retorna uma lista de todas as moedas disponíveis para conversão.
 - Exemplo de uso: `GET /currency/list`
 - Resposta: `["USD", "EUR", "BRL"]`

11.5.3 Integração com o Service

O `CurrencyController` depende do `CurrencyService` para realizar as operações de conversão e listagem de moedas. Essa dependência é injetada no controlador através do construtor, utilizando a injeção de dependência fornecida pelo Nest.js. Com isso, o controlador não precisa se preocupar com a lógica de negócio em si, que está encapsulada no serviço. Isso segue o princípio de separação de

responsabilidades, onde o controlador lida apenas com as requisições HTTP, e o serviço lida com a lógica de negócio.

11.5.4 Exercícios Práticos

- 1. Adicione Mais Funcionalidades ao Controlador:** Crie uma nova rota que permita atualizar as taxas de câmbio. Essa rota deve aceitar uma moeda e uma nova taxa como parâmetros e deve atualizar a taxa no `CurrencyService`.
 - 2. Crie um Controlador de Teste:** Crie um controlador chamado `HealthController` com uma rota `GET /health` que retorna o status da aplicação, simulando uma checagem de saúde do sistema.
 - 3. Teste as Rotas:** Use uma ferramenta como Postman ou Insomnia para testar as rotas criadas. Verifique se a conversão está funcionando conforme esperado.
 - 4. Melhore o Tratamento de Erros:** Modifique o controlador para capturar e retornar erros de forma mais estruturada. Por exemplo, retorne uma resposta 404 se a moeda não for encontrada.
-

11.6 TypeORM

O **TypeORM** é um ORM para TypeScript, que permite trabalhar com bancos de dados de maneira mais estruturada e orientada a objetos.

11.6.1 O que é um ORM?

Um ORM (Object-Relational Mapping) é uma técnica que permite mapear, transformar e manipular dados de tabelas de banco de dados como se fossem objetos em linguagens de programação orientadas a objetos. Isso simplifica a interação com o banco de dados, permitindo que você se concentre na lógica de negócio sem se preocupar com a complexidade das operações SQL.

11.6.2 Instalação do TypeORM

Para usar o TypeORM em um projeto Nest.js, precisamos instalar algumas dependências:

1. TypeORM em si.
2. O driver do banco de dados que você deseja usar (neste exemplo, usaremos o MySQL).

Execute o seguinte comando para instalar as dependências necessárias:

```
npm install @nestjs/typeorm typeorm mysql2
```

11.6.3 Criando um Schema

No TypeORM, as tabelas do banco de dados são representadas como classes chamadas **entidades**. Vamos criar uma entidade para representar uma moeda.

Primeiro, crie um arquivo chamado `currency.entity.ts` dentro do diretório `src/currency`:

currency.entity.ts:

```
import { Entity, Column, PrimaryGeneratedColumn } from 'typeorm';

@Entity()
export class Currency {
    @PrimaryGeneratedColumn()
    id: number;

    @Column()
    code: string;

    @Column('decimal', { precision: 5, scale: 2 })
}
```

```
    rate: number;  
}
```

Aqui, estamos criando uma entidade `Currency` com três colunas:

- `id`: Uma chave primária auto-gerada.
- `code`: O código da moeda (ex: USD, EUR).
- `rate`: A taxa de câmbio da moeda.

Configurar o TypeORM no Módulo

Depois de criar a entidade, você deve configurar o TypeORM no módulo principal (ou em um módulo dedicado) para garantir que ele reconheça as entidades.

Aqui está um exemplo de como configurar o TypeOrmModule no módulo CurrencyModule:

```
import { Module } from '@nestjs/common';  
import { TypeOrmModule } from '@nestjs/typeorm';  
import { CurrencyService } from './currency.service';  
import { CurrencyController } from './currency.controller';  
import { Currency } from './currency.entity';  
  
@Module({  
  imports: [TypeOrmModule.forFeature([Currency])],  
  providers: [CurrencyService],  
  controllers: [CurrencyController],  
})  
export class CurrencyModule {}
```

11.6.4 Injetando o Repositório no Service

Agora que temos nossa entidade, precisamos injetar o repositório correspondente no serviço `CurrencyService` para que possamos realizar operações de banco de dados.

Vamos modificar o `CurrencyService` para usar o repositório do TypeORM:

`currency.service.ts`:

```
import { Injectable } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { Repository } from 'typeorm';
import { Currency } from './currency.entity';

@Injectable()
export class CurrencyService {
  constructor(
    @InjectRepository(Currency)
    private currencyRepository: Repository<Currency>,
  ) {}

  async getAvailableCurrencies(): Promise<string[]> {
    const currencies = await this.currencyRepository.find();
    return currencies.map(currency => currency.code);
  }

  async convertCurrency(
    amount: number,
    from: string,
    to: string,
  ): Promise<number> {
    const fromCurrency = await this.currencyRepository.findOne(
      {
        where: { code: from },
      });
    const toCurrency = await this.currencyRepository.findOne(
      {
        where: { code: to },
      });
    if (!fromCurrency || !toCurrency) {
```

```

        throw new Error('Currency not found');
    }

    const convertedAmount = (amount / fromCurrency.rate) * to
    Currency.rate;
    return convertedAmount;
}

async createCurrency(code: string, rate: number): Promise<C
urrency> {
    const currency = this.currencyRepository.create({ code, r
ate });
    await this.currencyRepository.save(currency);

    return currency;
}
}

```

- 1. Injeção do Repositório:** Estamos injetando o repositório `Currency` usando o decorador `@InjectRepository`. Isso permite que o serviço acesse o banco de dados através do repositório.
- 2. Operações de Banco de Dados:** Usamos os métodos `find` e `findOne` do repositório para buscar moedas no banco de dados.
- 3. Lógica de Conversão:** A lógica de conversão foi modificada para trabalhar com os dados reais do banco de dados, ao invés de uma lista simulada.
- 4. Método para adicionar:** Criamos um novo método para adicionar `Currency` no banco.

Aqui, modificamos o `CurrencyService` para usar o repositório do TypeORM para buscar as moedas no banco de dados. Os métodos agora são assíncronos porque as operações de banco de dados retornam `Promises`.

Por último, vamos atualizar o nosso controller para ele também retornar as promises e também adicionar uma rota POST para criar `Currency`:

```
import { Body, Controller, Get, Post, Query } from '@nestjs/common';
import { CurrencyService } from './currency.service';
import { Currency } from './currency.entity';

@Controller('currency')
export class CurrencyController {
    constructor(private readonly currencyService: CurrencyService) {}

    @Get('convert')
    async convertCurrency(
        @Query('amount') amount: number,
        @Query('from') from: string,
        @Query('to') to: string,
    ): Promise<{ convertedAmount: number }> {
        const convertedAmount = await this.currencyService.convertCurrency(
            amount,
            from,
            to,
        );

        return { convertedAmount };
    }

    @Get('list')
    async getAvailableCurrencies(): Promise<string[]> {
        return await this.currencyService.getAvailableCurrencies();
    }

    @Post()
```

```
async createCurrency(
  @Body('code') code: string,
  @Body('rate') rate: number,
): Promise<Currency> {
  return await this.currencyService.createCurrency(code, rate);
}
}
```

11.6.3 Configuração a conexão com o banco no módulo principal

Agora que tudo está pronto, só falta configurar a conexão com o banco de dados.

No seu arquivo app.module.ts (módulo principal), adicione a seguinte configuração:

```
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';
import { CurrencyModule } from './currency/currency.module';
import { TypeOrmModule } from '@nestjs/typeorm';
import { Currency } from './currency/currency.entity';

@Module({
  imports: [
    CurrencyModule,
    TypeOrmModule.forRoot({
      host: 'localhost',
      type: 'mysql',
      port: 3306,
      username: 'root',
      password: 'minha_senha',
      database: 'currency_db',
      synchronize: true,
      entities: [Currency],
    })
  ]
})
```

```

    },
],
controllers: [AppController],
providers: [AppService],
})
export class AppModule {}

```

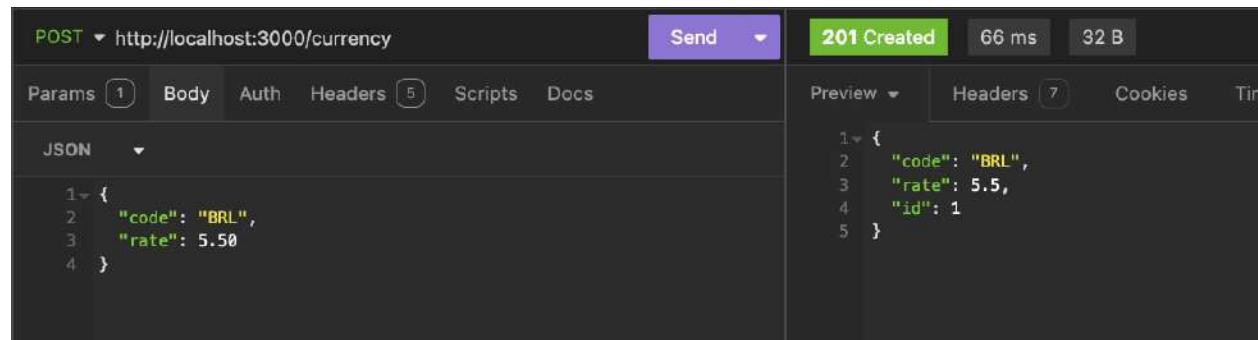
Essa alteração configura o TypeORM para se conectar a um banco de dados MySQL local chamado `currency_db`. A opção `synchronize: true` faz com que o TypeORM sincronize as entidades com o banco de dados automaticamente, criando as tabelas necessárias.

Caso você não saiba como rodar um banco de dados local, visite o módulo de Banco de Dados desse e-book, que lá ensinamos a rodar qualquer banco de dados com o Docker!

11.6.8 Testando Rotas

Em um client como Postman ou Insomnia, vamos testar as rotas criadas:

POST /currency



The screenshot shows a Postman interface with the following details:

- Method: POST
- URL: `http://localhost:3000/currency`
- Status: **201 Created**
- Time: 66 ms
- Size: 32 B
- Params: 1
- Body: JSON
- Headers: 7
- Cookies: 0
- Preview: Shows a JSON response with code "BRL", rate 5.5, and id 1.

```

1+ {
2   "code": "BRL",
3   "rate": 5.5,
4   "id": 1
5 }

```

GET /currency/list

GET ▾ http://localhost:3000/currency/list

Send ▾

200 OK 53 ms 7 B

Params 1 Body Auth Headers 4 Scripts Docs

No Body ▾

```

1> [
2   "BRL"
3 ]

```

GET /currency/convert

GET ▾ http://localhost:3000/currency/convert

Send ▾

200 OK 26 ms 38 B

Params 3 Body Auth Headers 5 Scripts Docs

URL PREVIEW
http://localhost:3000/currency/convert?amount=10&from=BRL&to=USD

QUERY PARAMETERS Import from URL Bulk Edit

+ Add Delete all Description

# amount	10	▼	✓	✖
# from	BRL	▼	✓	✖
# to	USD	▼	✓	✖

PATH PARAMETERS

```

1> {
2   "convertedAmount": 1.81818181818181
3 }

```

11.6.7 Exercícios Práticos

- Adicione Mais Campos à Entidade:** Adicione um campo de descrição à entidade `Currency` e modifique os métodos para incluir esse campo na lógica.
- Integre com uma API Externa:** Simule a atualização das taxas de câmbio puxando dados de uma API externa e salvando-os no banco de dados.
- Crie um Novo Serviço:** Crie um serviço que gerencie as configurações de taxas de câmbio, permitindo que os administradores atualizem as taxas diretamente na aplicação.
- Adicione Relacionamentos:** Expanda a entidade `Currency` para ter um relacionamento com outra entidade, como um histórico de conversões.

11.7 Validação de Dados (`class-validator` e `class-transformer`)

A validação de dados é uma etapa crucial no desenvolvimento de APIs, especialmente quando se trata de garantir que os dados recebidos pelo servidor estejam no formato e tipo corretos. No Nest.js, as bibliotecas `class-validator` e `class-transformer` são amplamente utilizadas para realizar essa validação de forma declarativa e eficiente.

11.7.1 O que são `class-validator` e `class-transformer` ?

- `class-validator` : Esta biblioteca permite adicionar validações aos campos das classes que representam as entidades ou DTOs (Data Transfer Objects). Ela oferece uma variedade de decorators que podem ser aplicados diretamente às propriedades das classes para validar seus valores.
- `class-transformer` : Trabalha junto com o `class-validator` para transformar os dados recebidos (geralmente em formato JSON) em instâncias de classes. Isso é útil para garantir que os dados sejam manipulados como objetos do TypeScript, com todos os métodos e propriedades definidos.

11.7.2 Instalando as Dependências

Primeiro, precisamos instalar as dependências necessárias:

```
npm install class-validator class-transformer
```

11.7.3 Aplicando Validações em um DTO

Um **DTO** (Data Transfer Object) é uma classe que define a estrutura dos dados que devem ser enviados ou recebidos pela API. Vamos criar um DTO para validar os dados de uma conversão de moeda.

Crie um arquivo chamado `convert-currency.dto.ts` dentro da pasta `src/currency`.

convert-currency.dto.ts:

```
import { IsNumber, IsString, IsNotEmpty } from 'class-validator';

export class ConvertCurrencyDto {
    @IsNumber()
    @IsNotEmpty()
    amount: number;

    @IsString()
    @IsNotEmpty()
    from: string;

    @IsString()
    @IsNotEmpty()
    to: string;
}
```

Aqui estamos usando três decorators principais:

- `@IsNumber()` : Verifica se o valor é um número.
- `@IsString()` : Verifica se o valor é uma string.
- `@IsNotEmpty()` : Verifica se o campo não está vazio.

11.7.4 Integrando o DTO no Controller

Agora que temos nosso DTO configurado, precisamos integrá-lo no `CurrencyController` para que os dados sejam validados automaticamente antes de serem processados.

Modifique o método `convertCurrency` no `currency.controller.ts` para usar o DTO:

currency.controller.ts:

```
import { Body, Controller, Get, Post, Query } from '@nestjs/common';
import { CurrencyService } from './currency.service';
import { Currency } from './currency.entity';
import { ConvertCurrencyDto } from './convert-currency.dto';

@Controller('currency')
export class CurrencyController {
  constructor(private readonly currencyService: CurrencyService) {}

  @Get('convert')
  async convertCurrency(
    @Query() query: ConvertCurrencyDto,
  ): Promise<{ convertedAmount: number }> {
    const { amount, from, to } = query;
    const convertedAmount = await this.currencyService.convertCurrency(
      amount,
      from,
      to,
    );
    return { convertedAmount };
  }

  @Get('list')
  async getAvailableCurrencies(): Promise<string[]> {
    return await this.currencyService.getAvailableCurrencies();
  }

  @Post()
  async createCurrency()
```

```
    @Body('code') code: string,
    @Body('rate') rate: number,
  ): Promise<Currency> {
  return await this.currencyService.createCurrency(code, rate);
}
}
```

Dessa forma, a validação será aplicada nos parâmetros da rota.

Adicionando o `ValidationPipe` no `main.ts`

O `ValidationPipe` aplica as regras de validação do `class-validator` globalmente, o que significa que qualquer dado recebido que esteja associado a um DTO será validado automaticamente. Vamos adicionar isso ao arquivo `main.ts`.

`main.ts`:

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import { ValidationPipe } from '@nestjs/common';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);

  // Ativando o ValidationPipe globalmente
  app.useGlobalPipes(new ValidationPipe({
    whitelist: true, // Remove propriedades não declaradas no DTO
    forbidNonWhitelisted: true, // Lança erro se propriedades não declaradas forem encontradas
    transform: true, // Transforma os payloads para os tipos especificados nos DTOs
  }));

  await app.listen(3000);
}
```

```
}
```

```
bootstrap();
```

Explicação das Configurações:

- `whitelist: true`: Propriedades que não estão explicitamente definidas no DTO serão removidas automaticamente, o que é útil para evitar a entrada de dados inesperados.
- `forbidNonWhitelisted: true`: Lança um erro se o cliente enviar propriedades não listadas no DTO, melhorando a segurança e garantindo que somente dados esperados sejam processados.
- `transform: true`: Transforma os dados de entrada para os tipos definidos no DTO. Por exemplo, se um campo é definido como `number` no DTO, o `ValidationPipe` converterá a string recebida (`"123"`) em um número (`123`).

Agora, caso você mande algum campo errado, a API retornará um erro:

The screenshot shows a Postman request to `http://localhost:3000/currency/convert`. The query parameters are: `amount=10&from=BRL&to=USD`. The `amount` field is highlighted in red, indicating it's the cause of the error. The response is a `400 Bad Request` with the following JSON body:

```
1: {
2:   "message": [
3:     "amount must be a number conforming to the specified constraints"
4:   ],
5:   "error": "Bad Request",
6:   "statusCode": 400
7: }
```

11.7.5 Exercícios Práticos

1. **Adicione Mais Validações:** Adicione novas validações ao DTO, como restrições de tamanho para as strings (ex: `@Length(3, 3)` para os códigos de moeda) e limites para os números (ex: `@Min(1)` para `amount`).

2. Crie um Novo DTO para Criação de Currency: Crie um DTO para a rota POST, validando os campos, e integre ele com o body.

3. Teste as Validações: Crie casos de teste que enviem dados inválidos para a API e verifique se os erros são retornados conforme esperado.

11.8 Autenticação e Autorização

Neste tópico, vamos implementar autenticação e autorização na nossa API de conversão de moedas utilizando o Nest.js. Vamos criar um módulo de usuários, configurar o schema de usuários, implementar rotas para `signup` e `login`, utilizando bcrypt para hashear as senhas e JWT para autenticação. Em seguida, aplicaremos um guard para proteger as rotas relacionadas às operações de conversão de moedas.

O que é autenticação?

Autenticação é o processo de verificar a identidade de um usuário que tenta acessar um sistema. Isso geralmente é feito através de credenciais de login, como um nome de usuário e senha. Uma vez autenticado, o usuário pode acessar recursos e funcionalidades que são permitidos para ele.

O que é autorização?

Autorização é o processo de conceder ou negar permissão para que um usuário autenticado acesse recursos específicos ou execute ações particulares dentro de um sistema. Enquanto a autenticação verifica quem é o usuário, a autorização determina o que esse usuário pode ou não pode fazer.

Dependências Necessárias

Para configurar a autenticação e autorização com JWT, precisamos instalar as seguintes dependências:

1. **bcrypt**: Para hashear as senhas dos usuários.
2. **@nestjs/jwt**: Para trabalhar com JWT no Nest.js.
3. **@nestjs/passport** e **passport-jwt**: Para integração do Nest.js com Passport.js, facilitando o uso de estratégias de autenticação.

Instalação das Dependências

No terminal, dentro do seu projeto, execute os seguintes comandos:

```
npm install bcrypt @nestjs/jwt @nestjs/passport passport-jwt
```

11.8.1 Criando o Módulo de Usuários

Primeiro, vamos criar o módulo de usuários que conterá todo o código relacionado à autenticação e gerenciamento de usuários.

```
nest generate module users
```

11.8.2 Criando o Schema de Usuários

Vamos definir a entidade `User` usando TypeORM. Crie um arquivo `user.entity.ts` dentro da pasta `src/users`.

`src/users/user.entity.ts`:

```
import { Entity, Column, PrimaryGeneratedColumn } from 'typeorm';

@Entity()
export class User {
  @PrimaryGeneratedColumn()
  id: number;
```

```
@Column()
username: string;

@Column()
password: string;
}
```

Vamos agora, adicionar essa entidade no arquivo `users.module.ts`:

```
import { Module } from '@nestjs/common';
import { TypeOrmModule } from '@nestjs/typeorm';
import { User } from './user.entity';

@Module({
  imports: [TypeOrmModule.forFeature([User])],
})
export class UsersModule {}
```

Não esqueça de também adicionar a entidade User no array de entidades no arquivo `app.module.ts`.

11.8.3 Criando o Serviço de Usuários

Agora, vamos criar o serviço de usuários, que lidará com a lógica de criação e autenticação de usuários. O serviço usará bcrypt para hashear as senhas.

```
nest generate service users
```

src/users/users.service.ts:

```
import { Injectable } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { Repository } from 'typeorm';
import { User } from './user.entity';
import * as bcrypt from 'bcrypt';
```

```

@Injectable()
export class UsersService {
  constructor(
    @InjectRepository(User)
    private usersRepository: Repository<User>,
  ) {}

  async createUser(username: string, password: string): Promise<User> {
    const hashedPassword = await bcrypt.hash(password, 10);
    const newUser = this.usersRepository.create({ username, p
assword: hashedPassword });
    return this.usersRepository.save(newUser);
  }

  async findByUsername(username: string): Promise<User | unde
fined> {
  return this.usersRepository.findOne({ where: { username } });
}

  async validateUser(username: string, pass: string): Promise<User | null> {
    const user = await this.findByUsername(username);
    if (user && await bcrypt.compare(pass, user.password)) {
      return user;
    }
    return null;
  }
}

```

- O serviço `UsersService` é responsável pela lógica de criação e autenticação de usuários.

- Utiliza a injeção de dependência para acessar o repositório de usuários do TypeORM.
- O método `createUser` cria um novo usuário hasheado a senha com bcrypt antes de salvar no banco de dados.
- O método `findByUsername` busca um usuário no banco de dados pelo nome de usuário.
- O método `validateUser` verifica se o nome de usuário e a senha fornecidos são válidos, comparando a senha fornecida com a senha hasheada armazenada no banco de dados.

Por que Utilizamos Hash para Senhas?

O hash é uma técnica utilizada para transformar dados (como uma senha) em uma sequência de caracteres única e de comprimento fixo, geralmente irreversível. Quando falamos sobre segurança de senhas, o hash é uma prática essencial por várias razões:

1. **Irreversibilidade:** Uma vez que uma senha é transformada em um hash, ela não pode ser facilmente revertida ao seu estado original. Isso significa que mesmo que um atacante consiga acessar os hashes das senhas, ele não poderá obter as senhas originais facilmente.
2. **Segurança:** Armazenar senhas em texto plano (sem hash) é extremamente arriscado. Se um banco de dados for comprometido, todas as senhas dos usuários estarão imediatamente expostas. Com o hash, mesmo que os hashes sejam obtidos por um invasor, a segurança das senhas é muito maior.
3. **Comparação Segura:** Em vez de comparar senhas em texto plano, podemos comparar os hashes das senhas. Isso significa que a senha original nunca precisa ser exposta ou transmitida em texto plano, aumentando a segurança geral.
4. **Salting:** Muitas implementações de hash, como bcrypt, permitem adicionar um "salt" — um valor aleatório adicionado à senha antes de ser hasheada. Isso garante que mesmas senhas idênticas tenham hashes diferentes, dificultando ainda mais a vida dos atacantes.

Ao utilizar o bcrypt em nosso serviço de usuários, estamos garantindo que as senhas dos usuários sejam armazenadas de forma segura, protegendo-as contra acessos não autorizados e possíveis violações de dados.

11.8.4 Criando Rotas para Signup e Login

Vamos criar um controlador para gerenciar as rotas de `signup` e `login`. Para o login, utilizaremos JWT para gerar tokens de autenticação.

```
nest generate controller users
```

src/users/users.controller.ts:

```
import { Controller, Post, Body, UnauthorizedException } from
'@nestjs/common';
import { UsersService } from './users.service';
import { JwtService } from '@nestjs/jwt';

@Controller('auth')
export class UsersController {
    constructor(
        private readonly usersService: UsersService,
        private readonly jwtService: JwtService,
    ) {}

    @Post('signup')
    async signup(@Body() body: { username: string; password: string }) {
        const user = await this.usersService.createUser(
            body.username,
            body.password,
        );
        return { message: 'User created successfully', userId: user.id };
    }
}
```

```

    }

    @Post('login')
    async login(@Body() body: { username: string; password: string }) {
        const user = await this.usersService.validateUser(
            body.username,
            body.password,
        );
        if (!user) {
            throw new UnauthorizedException('Invalid credentials');
        }

        const payload = { username: user.username, sub: user.id };
        return {
            access_token: this.jwtService.sign(payload),
        };
    }
}

```

- O `UsersController` contém as rotas para `signup` e `login` dos usuários.
- O método `signup` cria um novo usuário utilizando o serviço `UsersService` e retorna uma mensagem de sucesso junto com o ID do usuário.
- O método `login` valida as credenciais do usuário utilizando o serviço `UsersService`.
- Se as credenciais forem válidas, o método `login` gera um token JWT utilizando o `JwtService` e o retorna.
- Se as credenciais não forem válidas, o método `login` retorna uma mensagem de credenciais inválidas.

11.8.5 Configurando JWT

Vamos configurar JWT para gerenciar tokens de autenticação.

```
nest generate module auth
```

src/auth/auth.module.ts:

```
import { Module } from '@nestjs/common';
import { JwtModule } from '@nestjs/jwt';
import { PassportModule } from '@nestjs/passport';
import { UsersModule } from 'src/users/users.module';

@Module({
  imports: [
    PassportModule,
    JwtModule.register({
      secret: 'your-secret-key', // Use uma chave mais segura
      em produção
      signOptions: { expiresIn: '60m' },
    }),
    UsersModule,
  ],
  exports: [JwtModule],
})
export class AuthModule {}
```

Adicione a importação do módulo jwt no módulo de usuários:

src/users/users.module.ts:

```
import { Module } from '@nestjs/common';
import { TypeOrmModule } from '@nestjs/typeorm';
import { User } from './user.entity';
import { UsersController } from './users.controller';
import { JwtModule } from '@nestjs/jwt';
```

```
@Module({
  imports: [
    TypeOrmModule.forFeature([User]),
    JwtModule.register({
      secret: 'your-secret-key',
    }),
  ],
  providers: [UsersService],
  controllers: [UsersController],
})
export class UsersModule {}
```

O que é JWT e por que estamos utilizando

JWT (JSON Web Token) é uma forma de enviar informações entre duas partes usando um objeto JSON. Essas informações são seguras porque o JWT é assinado digitalmente. Você pode assinar um JWT com um segredo (usando o algoritmo HMAC) ou com um par de chaves pública/privada (usando RSA ou ECDSA).

Vantagens do JWT

1. **Compacto:** Os tokens JWT são compactos, o que os torna ideais para uso em ambientes onde o desempenho é crítico, como autenticação web em tempo real.
2. **Autocontido:** O token carrega todas as informações necessárias sobre o usuário, eliminando a necessidade de consultas adicionais ao banco de dados.
3. **Seguro:** O JWT é assinado usando uma chave secreta ou um par de chaves pública/privada, garantindo que os dados não foram alterados durante a transmissão.

Por que estamos utilizando JWT?

Estamos utilizando JWT para implementar a autenticação e autorização na nossa API por várias razões:

- 1. Autenticação Stateless:** Com JWT, a autenticação é stateless, o que significa que não precisamos armazenar o estado da sessão no servidor. Isso facilita a escalabilidade da aplicação.
- 2. Facilidade de Uso:** JWT é fácil de usar e integrar com outras tecnologias e frameworks, como Nest.js.
- 3. Segurança:** Os tokens JWT são assinados, garantindo que os dados não foram alterados. Além disso, os tokens podem ser configurados com uma expiração, melhorando a segurança.
- 4. Interoperabilidade:** JWT é um padrão e pode ser utilizado em diferentes contextos e com diferentes tecnologias, proporcionando uma forma consistente de autenticação em sistemas heterogêneos.

Com o JWT, podemos garantir que apenas usuários autenticados possam acessar determinados recursos da API, implementando facilmente a autenticação e autorização.

11.8.6 Criando um Guard para Proteger Rotas

Vamos criar um guard para proteger as rotas de currency, exigindo que o usuário esteja autenticado para acessá-las.

src/auth/jwt-auth.guard.ts:

```
import {  
  CanActivate,  
  ExecutionContext,  
  Injectable,  
  UnauthorizedException,  
} from '@nestjs/common';  
import { JwtService } from '@nestjs/jwt';  
import { Request } from 'express';  
  
@Injectable()
```

```

export class JwtAuthGuard implements CanActivate {
  constructor(private jwtService: JwtService) {}

  async canActivate(context: ExecutionContext): Promise<boolean> {
    const request = context.switchToHttp().getRequest();
    const token = this.extractTokenFromHeader(request);

    if (!token) {
      throw new UnauthorizedException();
    }
    try {
      const payload = await this.jwtService.verifyAsync(token,
        {
          secret: 'your-secret-key',
        });
      request['user'] = payload;
    } catch {
      throw new UnauthorizedException();
    }
    return true;
  }

  private extractTokenFromHeader(request: Request): string | undefined {
    const [type, token] = request.headers.authorization?.split(' ') ?? [];
    return type === 'Bearer' ? token : undefined;
  }
}

```

Aplicando o guard nas rotas do nosso CurrencyController, assim bloqueando o acesso de usuários que não enviarem o JWT, ou que enviem um token inválido:

src/currency/currency.controller.ts:

```
import { Body, Controller, Get, Post, Query, UseGuards } from
  '@nestjs/common';
import { CurrencyService } from './currency.service';
import { Currency } from './currency.entity';
import { ConvertCurrencyDto } from './convert-currency.dto';
import { JwtAuthGuard } from 'src/auth/jwt-auth.guard';

@UseGuards(JwtAuthGuard)
@Controller('currency')
export class CurrencyController {
  constructor(private readonly currencyService: CurrencyService) {}

  @Get('convert')
  async convertCurrency(
    @Query() query: ConvertCurrencyDto,
  ): Promise<{ convertedAmount: number }> {
    const { amount, from, to } = query;
    const convertedAmount = await this.currencyService.convertCurrency(
      amount,
      from,
      to,
    );
    return { convertedAmount };
  }

  @Get('list')
  async getAvailableCurrencies(): Promise<string[]> {
    return await this.currencyService.getAvailableCurrencies();
  }
}
```

```
@Post()
async createCurrency(
  @Body('code') code: string,
  @Body('rate') rate: number,
): Promise<Currency> {
  return await this.currencyService.createCurrency(code, rate);
}
```

Não esqueça de no módulo de currency, importar o módulo de autenticação:

src/currency/currency.module.ts:

```
import { Module } from '@nestjs/common';
import { TypeOrmModule } from '@nestjs/typeorm';
import { CurrencyService } from './currency.service';
import { CurrencyController } from './currency.controller';
import { Currency } from './currency.entity';
import { AuthModule } from 'src/auth/auth.module';

@Module({
  imports: [TypeOrmModule.forFeature([Currency]), AuthModule],
  providers: [CurrencyService],
  controllers: [CurrencyController],
})
export class CurrencyModule {}
```

11.8.7 Testando as rotas

Criação de usuários:

POST ▼ http://localhost:3000/auth/signup Send 201 Created 168 ms 50 B

Params 3 Body Auth Headers 5 Scripts Docs

JSON

```
1= {  
2   "username": "teste",  
3   "password": "senha123"  
4 }  
5
```

Preview ▾ Headers 7 Cookies Timeline Mock

```
1= {  
2   "message": "User created successfully",  
3   "userId": 1  
4 }
```

Login com a senha correta:

POST ▼ http://localhost:3000/auth/login Send 201 Created 237 ms 160 B Just Now ▾

Params 3 Body Auth Headers 5 Scripts Docs

JSON

```
1= {  
2   "username": "teste",  
3   "password": "senha123"  
4 }  
5
```

Preview ▾ Headers 7 Cookies Timeline Mock Response

```
1= {  
2   "access_token":  
3     "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6InRlc3RLIiw-  
4c3Vl1joxLC3pYXQ10je3MjI4MDA2MDN9.MNetNdin-  
hxVecyAgmx0MeJNTDjv1U-dWb8pLE40oE"  
3 }
```

Login com a senha incorreta:

POST ▼ http://localhost:3000/auth/login Send 401 Unauthorized 118 ms 73 B

Params 3 Body Auth Headers 5 Scripts Docs

JSON

```
1= {  
2   "username": "teste",  
3   "password": "senha1234"  
4 }  
5
```

Preview ▾ Headers 7 Cookies Timeline Mock

```
1= {  
2   "message": "Invalid credentials",  
3   "error": "Unauthorized",  
4   "statusCode": 401  
5 }
```

Tentando acessar a rota GET `currency/list` sem passar o token:

The screenshot shows a Postman request for `GET http://localhost:3000/currency/list`. The response status is **401 Unauthorized** with a duration of **44 ms** and a size of **43 B**. The Headers tab shows the response body:

```
1= {
2   "message": "Unauthorized",
3   "statusCode": 401
4 }
```

Agora com o token:

The screenshot shows a Postman request for `GET http://localhost:3000/currency/list`. The response status is **200 OK** with a duration of **117 ms** and a size of **13 B**. The Headers tab shows the response body:

```
1= [
2   "BRL",
3   "USD"
4 ]
```

11.9 Swagger

Agora que nossa API de conversão de moedas está tomando forma, é essencial documentá-la de maneira clara e acessível para que outros desenvolvedores possam entender e utilizar os endpoints que estamos criando. É aqui que entra o **Swagger**.

O Swagger é uma ferramenta poderosa que permite gerar uma documentação interativa para APIs RESTful. Ele fornece uma interface web onde você pode explorar e testar todos os endpoints disponíveis na sua API, sem a necessidade de um cliente HTTP separado como Postman ou Insomnia. Isso facilita o desenvolvimento, a manutenção e a colaboração em projetos.

11.9.1 Instalando e Configurando o Swagger

Primeiro, vamos instalar o pacote necessário para integrar o Swagger ao Nest.js:

```
npm install @nestjs/swagger
```

Com os pacotes instalados, o próximo passo é configurar o Swagger no seu aplicativo Nest.js. Vamos configurar o Swagger no arquivo `main.ts` para que ele esteja disponível assim que o servidor for iniciado.

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import { ValidationPipe } from '@nestjs/common';
import { DocumentBuilder, SwaggerModule } from '@nestjs/swagger';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);

  // Configurando o Swagger
  const config = new DocumentBuilder()
    .setTitle('API de Conversão de Moedas')
    .setDescription('Documentação da API de Conversão de Moedas')
    .setVersion('1.0')
    .addTag('Conversão')
    .build();

  const document = SwaggerModule.createDocument(app, config);
  SwaggerModule.setup('api', app, document);

  // Ativando o ValidationPipe globalmente
  app.useGlobalPipes(
    new ValidationPipe({
```

```

        whitelist: true, // Remove propriedades não declaradas
no DTO
            forbidNonWhitelisted: true, // Lança erro se propriedad
es não declaradas forem encontradas
            transform: true, // Transforma os payloads para os tipo
s especificados nos DTOS
        ),
);
}

await app.listen(3000);
}
bootstrap();

```

No código acima, estamos utilizando o `DocumentBuilder` para definir os detalhes da nossa documentação, como título, descrição, versão, e até mesmo tags para organizar nossos endpoints. A documentação gerada pelo Swagger estará disponível na rota `/api` do nosso servidor.

11.9.2 Anotações e Decoradores do Swagger

Para que o Swagger consiga gerar uma documentação detalhada dos nossos endpoints, precisamos adicionar algumas anotações (decorators) aos nossos controladores, rotas, e DTOs (Data Transfer Objects).

Aqui está um exemplo de como você pode anotar no CurrencyController:

```

import { Body, Controller, Get, Post, Query, UseGuards } from
'@nestjs/common';
import { CurrencyService } from './currency.service';
import { Currency } from './currency.entity';
import { ConvertCurrencyDto } from './convert-currency.dto';
import { JwtAuthGuard } from 'src/auth/jwt-auth.guard';
import { ApiOperation, ApiQuery } from '@nestjs/swagger';

```

```

@UseGuards(JwtAuthGuard)
@Controller('currency')
export class CurrencyController {
    constructor(private readonly currencyService: CurrencyService) {}

    @ApiOperation({ summary: 'Converte uma moeda para outra' })
    @ApiQuery({ name: 'from', description: 'Moeda de origem', example: 'USD' })
    @ApiQuery({ name: 'to', description: 'Moeda de destino', example: 'BRL' })
    @ApiQuery({ name: 'amount', description: 'Valor a ser convertido', example: 100 })
    @Get('convert')
    async convertCurrency(
        @Query() query: ConvertCurrencyDto,
    ): Promise<{ convertedAmount: number }> {
        const { amount, from, to } = query;
        const convertedAmount = await this.currencyService.convertCurrency(
            amount,
            from,
            to,
        );

        return { convertedAmount };
    }

    ...restante do código
}

```

Neste exemplo, estamos utilizando os decoradores `@ApiTags`, `@ApiOperation`, e `@ApiQuery` para adicionar informações úteis à nossa documentação. Isso inclui descrições dos endpoints, parâmetros de consulta e exemplos de valores.

11.9.3 Explorando a Documentação

Após configurar e anotar seus controladores, você pode iniciar o servidor e acessar a documentação gerada pelo Swagger em <http://localhost:3000/api>.

Essa interface fornecerá uma visão geral dos endpoints disponíveis, juntamente com a capacidade de testar os endpoints diretamente da interface.

The screenshot shows a web browser displaying the API documentation for a currency converter. The title is "API de Conversão de Moedas". The main section is titled "default". It lists two endpoints: a general root endpoint "/ (GET)" and a specific conversion endpoint "/currency/convert (GET)". The "/currency/convert" endpoint is described as "Converte uma moeda para outra". Below the endpoint, there are three parameters: "amount" (query), "to" (query), and "from" (query). Each parameter has a description and an example value. A "Try it out" button is present. The "Responses" section shows a single response code: 200. At the bottom, there is another endpoint entry for "/currency/list (GET)".

Exercícios Práticos

- 1. Adicione Swagger a Novos Endpoints:** Anote todos os endpoints existentes na API com decoradores do Swagger, incluindo os métodos de autenticação e as operações.
- 2. Adicione Documentação aos DTOs:** Pesquise sobre o decorator @ApiProperty, e o adicione em seus DTOs.
- 3. Personalize a Documentação:** Adicione mais detalhes à documentação, como exemplos de respostas, status codes e descrições mais completas.

4. **Teste a API via Swagger:** Utilize a interface Swagger para fazer chamadas à API e verificar se a documentação reflete corretamente o comportamento do servidor.
-

módulo 12

react com typescript

Combine o melhor de React e TypeScript para
criar aplicações eficientes e tipadas. 

05 tópicos neste módulo

Capítulo 12: React com TypeScript

12.1 Configuração de Ambiente

Neste capítulo, vamos configurar o ambiente para iniciar o desenvolvimento de nossa **Wikipedia pessoal** utilizando **React** com **TypeScript**. Se você já está familiarizado com o React em JavaScript, essa etapa será uma extensão natural, mas com a adição da tipagem estática poderosa que o TypeScript oferece. 🎉

12.1.1 Criando o Projeto com Vite

Para criar nosso projeto, utilizaremos o **Vite**, uma ferramenta moderna de build que suporta nativamente TypeScript e é super rápida. Vamos criar um projeto Vite configurado com React e TypeScript.

1. Criação do Projeto:

- Abra o terminal e execute o comando abaixo:

```
npm create vite@latest wikipedia-pessoal --template react-ts
```

- Navegue até o diretório do projeto:

```
cd wikipedia-pessoal
```

- Instale as dependências:

```
npm install
```

2. Estrutura do Projeto:

- Após a criação, você verá uma estrutura básica com TypeScript já configurado. Aqui estão alguns dos principais arquivos e pastas:

- `index.tsx`: Ponto de entrada da aplicação.
- `App.tsx`: Componente principal do React.
- `tsconfig.json`: Arquivo de configuração do TypeScript.

3. Executando o Projeto:

- Para garantir que tudo está funcionando, execute o comando abaixo para iniciar o servidor de desenvolvimento:

```
npm run dev
```

- Acesse <http://localhost:5173> no navegador para ver a aplicação rodando.

12.2 Componentes

Neste tópico, vamos estruturar as pastas e arquivos do nosso projeto **Wikipedia pessoal** e criar os primeiros componentes. Vamos utilizar o **Chakra UI**, uma biblioteca de componentes React que facilita a criação de interfaces de usuário modernas e acessíveis. Além disso, configuraremos o Chakra UI para estilizar nossa aplicação.

12.2.1 Estrutura de Pastas e Arquivos

Vamos começar organizando nosso projeto em uma estrutura que facilite a manutenção e expansão. Aqui está a estrutura inicial sugerida:

```
wikipedia-pessoal/
├── public/
└── src/
    ├── components/
    │   ├── Header.tsx
    │   ├── SearchBar.tsx
    │   └── ArticleList.tsx
```

```
|   |   └── ArticleItem.tsx
|   └── pages/
|       ├── Home.tsx
|       └── Article.tsx
|   └── App.tsx
|       └── main.tsx
└── .eslintrc.json
└── tsconfig.json
└── package.json
```

- **components/**: Contém componentes reutilizáveis como `Header`, `SearchBar`, `ArticleList`, e `ArticleItem`.
- **pages/**: Armazena as páginas principais da aplicação, como `Home` e `Article`.
- **App.tsx**: Componente principal da aplicação.

12.2.2 Instalando e Configurando o Chakra UI

Para criar uma interface agradável e responsiva, vamos utilizar o **Chakra UI**.

1. Limpando o CSS:

- Apague o conteúdo do arquivo `index.css`;
- Apague o conteúdo do arquivo `App.css`;

2. Instalando o Chakra UI:

- Abra o terminal e execute o seguinte comando:

```
npm install @chakra-ui/react @emotion/react @emotion/style
d framer-motion
```

3. Configurando o Chakra UI:

- No `main.tsx`, envolva a aplicação com o `ChakraProvider` para aplicar o tema padrão do Chakra:

```
import React from 'react'
import ReactDOM from 'react-dom/client'
import App from './App.tsx'
import './index.css'
import { ChakraProvider } from '@chakra-ui/react'

ReactDOM.createRoot(document.getElementById('root')!).render(
  <ChakraProvider>
    <React.StrictMode>
      <App />
    </React.StrictMode>,
  </ChakraProvider>,
)
)
```

12.2.3 Criando os Primeiros Componentes

Vamos criar os componentes iniciais que compõem a estrutura da nossa Wikipedia pessoal.

1. Componente Header:

- Crie o arquivo `Header.tsx` na pasta `components`:

```
import { Box, Heading } from '@chakra-ui/react';

const Header = () => {
  return (
    <Box as="header" bg="teal.500" p={4} color="white">
      <Heading as="h1" size="lg">
        Wikipedia Pessoal 🌐
      </Heading>
    </Box>
  );
};
```

```
export default Header;
```

- O `Header` é um componente simples que exibe o título da aplicação dentro de uma barra superior.

2. Componente SearchBar:

- Crie o arquivo `SearchBar.tsx` na pasta `components` :

```
import { Box, Input, Button } from '@chakra-ui/react';

const SearchBar = () => {
  return (
    <Box display="flex" mt={4} mb={4}>
      <Input placeholder="Search for articles..." />
      <Button ml={2} colorScheme="teal">
        Search
      </Button>
    </Box>
  );
};

export default SearchBar;
```

- A `SearchBar` permite ao usuário procurar por artigos e tem um botão estilizado com o Chakra.

3. Componente ArticleList e ArticleItem:

- Crie o arquivo `ArticleItem.tsx` na pasta `components` :

```
import { Box, Text } from '@chakra-ui/react';

const ArticleItem: React.FC<any> = ({ title, summary }) =>
{
  return (
    <Box>
      <Text>{title}</Text>
      <Text>{summary}</Text>
    </Box>
  );
};

export default ArticleItem;
```

```

        <Box p={4} shadow="md" borderWidth="1px" mb={4}>
            <Text fontWeight="bold">{title}</Text>
            <Text mt={2}>{summary}</Text>
        </Box>
    );
};

export default ArticleItem;

```

- O `ArticleItem` exibe o título e o resumo de um artigo.
- Crie o arquivo `ArticleList.tsx` na pasta `components`:

```

import { Box } from '@chakra-ui/react';
import ArticleItem from './ArticleItem';

const ArticleList = () => {
    const articles = [
        { id:1, title: 'React com TypeScript', summary: 'Aprenda a usar TypeScript com React.' },
        { id:2, title: 'Estilização com Chakra UI', summary: 'Guia para estilização com Chakra UI.' },
    ];

    return (
        <Box>
            {articles.map((article, index) => (
                <ArticleItem id={article.id} key={index} title={article.title} summary={article.summary} />
            ))}
        </Box>
    );
};

export default ArticleList;

```

- O `ArticleList` renderiza uma lista de `ArticleItem` a partir de um array de artigos.

4. Componente Principal:

- Finalmente, configure o `App.tsx` para usar esses componentes:

```
import React from 'react';
import { Container } from '@chakra-ui/react';
import Header from './components/Header';
import SearchBar from './components/SearchBar';
import ArticleList from './components/ArticleList';

const App: React.FC = () => {
  return (
    <Container maxW="container.md">
      <Header />
      <SearchBar />
      <ArticleList />
    </Container>
  );
};

export default App;
```

Exercícios Práticos

1. **Adicionar Mais Componentes:** Crie mais componentes que possam ser úteis na Wikipedia, como um componente de rodapé.
2. **Personalizar o Tema:** Explore como customizar o tema do Chakra UI e aplique um tema personalizado à aplicação.
3. **Adicionar Responsividade:** Ajuste o layout dos componentes para melhorar a experiência em dispositivos móveis.

12.3 Tipagem de Props

Agora que já configuramos o ambiente e criamos os primeiros componentes, é hora de entender como funciona a **tipagem de props** no TypeScript. Props são os argumentos que passamos para os componentes React, e com TypeScript, podemos garantir que essas props sejam do tipo correto, tornando nosso código mais seguro e robusto. 🎯

12.3.1 O que são Props?

No React, **props** (propriedades) são uma forma de passar dados de um componente pai para um componente filho. Com TypeScript, podemos definir o tipo dessas props para evitar erros e garantir que o componente receba os dados que espera.

12.3.2 Definindo Tipos de Props

Vamos começar tipando as props de um componente simples. Vamos revisar o componente `ArticleItem` que criamos anteriormente e adicionar tipagem às suas props.

Passo 1: Criando uma Interface para as Props

Em TypeScript, usamos **interfaces** para definir a forma dos objetos. Vamos criar uma interface `ArticleItemProps` que descreve as props que o componente `ArticleItem` deve receber.

```
interface ArticleItemProps {  
  title: string;  
  summary: string;  
}
```

Passo 2: Tipando as Props no Componente

Agora, podemos usar essa interface para tipar as props do componente `ArticleItem`.

```
import { Box, Text } from '@chakra-ui/react';

interface ArticleItemProps {
  title: string;
  summary: string;
}

const ArticleItem: React.FC<ArticleItemProps> = ({ title, summary }) => {
  return (
    <Box p={4} shadow="md" borderWidth="1px" mb={4}>
      <Text fontWeight="bold">{title}</Text>
      <Text mt={2}>{summary}</Text>
    </Box>
  );
};

export default ArticleItem;
```

Aqui, `React.FC<ArticleItemProps>` indica que `ArticleItem` é um componente funcional do React que aceita props do tipo `ArticleItemProps`. Se tentarmos passar uma prop que não esteja definida na interface, o TypeScript nos dará um erro durante o desenvolvimento. 

12.3.3 Tornando Props Opcionais

Às vezes, algumas props podem ser opcionais. Podemos fazer isso em TypeScript adicionando um `?` ao nome da prop na interface.

Vamos imaginar que a prop `summary` é opcional:

```
interface ArticleItemProps {
  title: string;
```

```
    summary?: string;  
}
```

Agora, o `summary` não é obrigatório ao usar o componente `ArticleItem`. Se `summary` não for passado, ele será `undefined` dentro do componente.

12.3.4 Tipagem com Props Complexas

E se uma prop for mais complexa, como um array ou um objeto? Podemos definir esses tipos diretamente na interface.

Por exemplo, vamos criar um componente `Author` que receba um objeto `author` como prop.

Crie o arquivo `Author.tsx`:

```
import { Box, Text } from "@chakra-ui/react";  
  
interface Author {  
  name: string;  
  bio: string;  
}  
  
interface AuthorProps {  
  author: Author;  
}  
  
const AuthorComponent: React.FC<AuthorProps> = ({ author }) => {  
  return (  
    <Box p={4} shadow="md" borderWidth="1px" mb={4}>  
      <Text fontWeight="bold">{author.name}</Text>  
      <Text mt={2}>{author.bio}</Text>  
    </Box>  
  );  
};
```

```
export default AuthorComponent;
```

Aqui, estamos aninhando a interface `Author` dentro da interface `AuthorProps`. Isso mostra como podemos construir tipos complexos em TypeScript.

Exercícios Práticos

- Adicionar Tipagem:** Adicione tipagem a todos os componentes que você já criou na sua aplicação Wikipedia. Certifique-se de que todas as props estejam corretamente tipadas.
 - Props Opcionais:** Identifique uma prop que possa ser opcional em um dos seus componentes e torne-a opcional usando o `?`.
 - Componentes Complexos:** Crie um componente que receba um array de objetos como prop e renderize uma lista a partir desses dados.
-

12.4 Hooks

Agora que já aprendemos sobre tipagem de props e construímos a estrutura básica dos nossos componentes, vamos começar a dar vida à nossa Wikipedia pessoal usando **hooks** no React. Os hooks permitem que você adicione **estado** e outras funcionalidades reativas aos componentes funcionais. 

12.4.1 Introdução aos Hooks

Os **hooks** são funções especiais que permitem usar recursos do React, como estado e ciclo de vida, em componentes funcionais. Com TypeScript, podemos tipar esses hooks para garantir que tudo funcione da forma esperada.

Os hooks mais comuns são:

- `useState`
- `useEffect`

- `useContext`

Vamos começar com o `useState` para gerenciar o estado dos nossos componentes.

12.4.2 Usando `useState` com TypeScript

O `useState` é um hook que permite adicionar estado a um componente funcional. Com TypeScript, precisamos especificar o tipo de dado que o estado armazenará.

Vamos adicionar um hook `useState` ao nosso componente para gerenciar a exibição de um artigo na nossa Wikipedia.

```
import { useState } from 'react';
import { Box, Button, Text } from '@chakra-ui/react';

interface ArticleItemProps {
  title: string;
  summary: string;
}

const ArticleItem: React.FC<ArticleItemProps> = ({ title, summary }) => {
  // Usando useState com TypeScript para gerenciar a visibilidade do conteúdo
  const [isSummaryVisible, setSummaryVisible] = useState<boolean>(false);

  const toggleSummaryVisibility = () => {
    setSummaryVisible(!isSummaryVisible);
  };

  return (
    <Box p={4} shadow="md" borderWidth="1px" mb={4}>
      <Text fontWeight="bold">{title}</Text>
      {isSummaryVisible && <Text mt={2}>{summary}</Text>}
    
```

```

        <Button mt={4} onClick={toggleSummaryVisibility}>
          {isSummaryVisible ? 'Ocultar Conteúdo' : 'Mostrar Conteúdo'}
        </Button>
      </Box>
    );
}

export default ArticleItem;

```

Aqui, usamos `useState` para controlar a visibilidade do conteúdo do artigo. O estado `isSummaryVisible` é inicializado como `false` e é alternado entre `true` e `false` quando o botão é clicado. A tipagem `<boolean>` garante que o estado só aceitará valores booleanos. ✓

12.4.3 Usando `useEffect` com TypeScript

O `useEffect` é outro hook importante, usado para **efeitos colaterais** como chamadas de API, assinaturas e alterações diretas no DOM.

Vamos simular uma chamada de API para carregar o conteúdo do artigo quando o componente for montado.

```

import { useState, useEffect } from 'react';
import { Box, Button, Text } from '@chakra-ui/react';

interface ArticleItemProps {
  title: string;
  fetchSummary: () => Promise<string>;
}

const ArticleItem: React.FC<ArticleItemProps> = ({ title, fetchSummary }) => {
  const [summary, setSummary] = useState<string>('');
  const [isSummaryVisible, setSummaryVisible] = useState<boolean>(false);

```

```

useEffect(() => {
  // Simulando uma chamada de API para carregar o conteúdo
  do artigo
  const loadSummary = async () => {
    const fetchedSummary = await fetchSummary();
    setSummary(fetchedSummary);
  };

  loadSummary();
}, [fetchSummary]); // Dependência do useEffect

const toggleSummaryVisibility = () => {
  setSummaryVisible(!isSummaryVisible);
};

return (
  <Box p={4} shadow="md" borderWidth="1px" mb={4}>
    <Text fontWeight="bold">{title}</Text>
    {isSummaryVisible && <Text mt={2}>{summary}</Text>}
    <Button mt={4} onClick={toggleSummaryVisibility}>
      {isSummaryVisible ? 'Ocultar Conteúdo' : 'Mostrar Con-
teúdo'}
    </Button>
  </Box>
);
};

export default ArticleItem;

```

Aqui, usamos `useEffect` para carregar o conteúdo do artigo quando o componente é montado. O conteúdo é carregado através da função `fetchSummary`, que simula uma chamada de API. O TypeScript garante que o tipo de retorno seja uma `Promise<string>`.

Agora vamos mudar o arquivo `ArticleList.tsx` para simular a chamada a uma API externa:

```
import { Box } from '@chakra-ui/react';
import ArticleItem from './ArticleItem';

// Função para simular a chamada de API para obter o resumo
const fetchSummary = async (title: string) => {
    // Simulação de uma chamada de API
    return `Resumo para o artigo: ${title}`;
};

const ArticleList = () => {
    const articles = [
        { id:1, title: 'React com TypeScript', summary: 'Aprenda a usar TypeScript com React.' },
        { id:2, title: 'Estilização com Chakra UI', summary: 'Guia para estilização com Chakra UI.' },
    ];

    return (
        <Box>
            {articles.map((article, index) => (
                <ArticleItem
                    key={index}
                    title={article.title}
                    fetchSummary={() => fetchSummary(article.title)}
                />
            ))}
        </Box>
    );
};

export default ArticleList;
```

Exercícios Práticos

- 1. Adicionar Estado:** Adicione hooks `useState` a outros componentes da sua Wikipedia pessoal para gerenciar estados como "favoritos" ou "artigos lidos".
- 2. Simulação de API:** Crie uma função que simula a obtenção de dados de um servidor e use o `useEffect` para carregar dados em um componente assim que ele for montado.
- 3. Exercício Extra:** Implemente um botão que permite ao usuário marcar um artigo como "favorito" e utilize `useState` para armazenar essa informação.

Com esses conhecimentos, você agora pode gerenciar o estado e os efeitos dos seus componentes de maneira eficiente e tipada no TypeScript! 

No próximo tópico, vamos explorar **Gerenciamento de Estado** de maneira mais aprofundada, utilizando contextos e outros recursos avançados do React e TypeScript.

Capítulo 12.5: Roteamento com React Router

Em uma aplicação web moderna, a navegação entre diferentes páginas ou seções da aplicação é uma necessidade comum. O **React Router** é a biblioteca padrão usada para gerenciar o roteamento em aplicações React. Ele permite que você defina rotas, crie links de navegação, e controle como e onde os componentes são renderizados com base na URL.

Neste capítulo, você aprenderá como configurar e utilizar o React Router para criar uma experiência de navegação fluida na sua aplicação **Wikipedia pessoal**.

12.5.1 O que é o React Router?

O **React Router** é uma biblioteca poderosa que permite que você implemente o roteamento em uma aplicação React de maneira declarativa. Com ele, você pode definir rotas que mapeiam URLs para componentes específicos. Quando o usuário navega para uma URL específica, o React Router renderiza o componente correspondente, criando uma experiência de navegação baseada em páginas.

Algumas das principais funcionalidades do React Router incluem:

- **Definição de Rotas:** Mapeie URLs para componentes React.
- **Navegação Declarativa:** Use componentes de navegação como `<Link>` e `<NavLink>` para criar links que permitem ao usuário navegar entre diferentes partes da aplicação.
- **Redirecionamentos e Proteção de Rotas:** Controle o acesso às rotas com base em condições, como autenticação de usuário.
- **Roteamento Aninhado:** Defina rotas dentro de rotas para criar layouts e páginas complexas.

12.5.2 Instalando o React Router

Antes de começar a usar o React Router, precisamos instalá-lo em nosso projeto. Você pode fazer isso através do npm:

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

12.5.3 Configurando o React Router

Vamos começar configurando o React Router na sua aplicação Wikipedia pessoal. Isso envolve definir as rotas e configurar os componentes de navegação.

Passo 1: Configurando o BrowserRouter

O componente `BrowserRouter` é o principal contêiner de roteamento e precisa envolver a aplicação para que o React Router funcione corretamente.

No arquivo `main.tsx`, atualize o código para incluir o `BrowserRouter`:

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import App from './App.tsx';
import { ChakraProvider } from '@chakra-ui/react';
import { BrowserRouter } from 'react-router-dom';

ReactDOM.createRoot(document.getElementById('root')!).render(
  <ChakraProvider>
    <BrowserRouter>
```

```
<React.StrictMode>
  <App />
</React.StrictMode>
</BrowserRouter>
</ChakraProvider>
);
```

Passo 2: Definindo as Rotas

Agora, vamos definir as rotas da aplicação. Crie ou edite o arquivo `App.tsx` para incluir a configuração básica de rotas.

```
import React from 'react';
import { Routes, Route } from 'react-router-dom';
import { Container } from '@chakra-ui/react';
import Home from './pages/Home';
import Article from './pages/Article';
import Header from './components/Header';

const App: React.FC = () => {
  return (
    <Container maxW="container.md">
      <Header />
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/article/:id" element={<Article />} />
      </Routes>
    </Container>
  );
};

export default App;
```

Aqui:

- O componente `Routes` atua como um contêiner que agrupa todas as rotas definidas.

- Cada `Route` define um caminho (`path`) e o componente que deve ser renderizado quando a URL corresponde a esse caminho.
- Usamos `path="/"` para mapear a rota raiz (homepage) e `path="/article/:id"` para mapear a rota de visualização de um artigo específico, onde `:id` é um parâmetro dinâmico que será usado para identificar o artigo.

Passo 3: Criando Componentes de Página

Agora que temos as rotas definidas, vamos criar os componentes que serão renderizados em cada rota.

1. Página Home

```
// src/pages/Home.tsx
import React from 'react';
import { Box, Heading, Text } from '@chakra-ui/react';
import ArticleList from '../components/ArticleList';
import SearchBar from '../components/SearchBar';

const Home: React.FC = () => {
  return (
    <Box>
      <Heading as="h2" size="xl" mb={4}>
        Bem-vindo à Wikipedia Pessoal
      </Heading>
      <Text mb={4}>Pesquise e visualize seus artigos favoritos!</Text>
      <SearchBar />
      <ArticleList />
    </Box>
  );
};

export default Home;
```

1. Página Article

```

// src/pages/Article.tsx
import React from 'react';
import { useParams } from 'react-router-dom';
import { Box, Heading, Text } from '@chakra-ui/react';

const Article: React.FC = () => {
  const { id } = useParams<{ id: string }>();

  return (
    <Box>
      <Heading as="h2" size="xl" mb={4}>
        Artigo {id}
      </Heading>
      <Text>
        Aqui você verá o conteúdo do artigo com ID: {id}.
      </Text>
    </Box>
  );
};

export default Article;

```

- Na página `Home`, exibimos uma lista de artigos usando o componente `ArticleList`.
- Na página `Article`, usamos `useParams` do React Router para capturar o parâmetro `id` da URL, que representa o identificador do artigo.

12.5.4 Navegação entre Páginas

Para que o usuário possa navegar entre as páginas, precisamos criar links de navegação. Vamos usar o componente `<Link>` do React Router.

Passo 1: Criando Links de Navegação

Atualize o componente `ArticleItem` para incluir um link para a página do artigo:

```

import { Box, Text } from '@chakra-ui/react';
import { Link } from 'react-router-dom';

interface ArticleItemProps {
  id: number;
  title: string;
  summary: string;
}

const ArticleItem: React.FC<ArticleItemProps> = ({ id, title, summary }) => {
  return (
    <Box p={4} shadow="md" border="1px solid #ccc" mb={4}>
      <Link to={`/article/${id}`}>
        <Text fontWeight="bold" color="teal.500">
          {title}
        </Text>
      </Link>
      <Text mt={2}>{summary}</Text>
    </Box>
  );
};

export default ArticleItem;

```

Aqui, o `<Link>` cria um link de navegação que redireciona o usuário para a página de visualização do artigo ao clicar no título.

12.5.5 Roteamento Aninhado e Parâmetros de Rota

O React Router também permite criar rotas aninhadas, onde uma rota "pai" contém rotas "filhas". Isso é útil para criar layouts ou páginas que têm seções específicas que devem ser renderizadas dentro de uma estrutura maior.

Vamos expandir o exemplo para incluir uma rota aninhada na página de artigo que exibe seções adicionais.

Passo 1: Definindo uma Rota Aninhada

Atualize o componente `Article` para incluir uma rota aninhada:

```
// src/pages/Article.tsx
import React from 'react';
import { useParams, Routes, Route, Link } from 'react-router-dom';
import { Box, Heading, Text } from '@chakra-ui/react';

const Article: React.FC = () => {
  const { id } = useParams<{ id: string }>();

  return (
    <Box>
      <Heading as="h2" size="xl" mb={4}>
        Artigo {id}
      </Heading>
      <Text mb={4}>
        Aqui você verá o conteúdo do artigo com ID: {id}.
      </Text>
      <Link to="details" style={{ color: 'teal' }}>Ver Detalhes</Link>

      <Routes>
        <Route path="details" element={<ArticleDetails />} />
      </Routes>
    </Box>
  );
};

const ArticleDetails: React.FC = () => {
  return (
    <Box mt={4}>
      <Heading as="h3" size="lg" mb={2}>
        Detalhes do Artigo
      </Heading>
    </Box>
  );
}
```

```
</Heading>
<Text>Aqui estão os detalhes adicionais do artigo.</Text>
      </Box>
    );
};

export default Article;
```

Passo 2: Acessando a Rota Aninhada

Agora, ao acessar a página de um artigo e clicar em "Ver Detalhes", a rota aninhada será renderizada dentro da página do artigo.

Exercícios Práticos

- Adicionar Rotas:** Adicione rotas adicionais à sua aplicação, como uma página "Sobre" ou uma página de "Contato".
- Roteamento Aninhado:** Crie uma estrutura de rotas aninhadas para uma seção específica da sua aplicação, como um painel de administração com subseções.
- Redirecionamentos:** Configure redirecionamentos automáticos para garantir que os usuários sejam levados a páginas específicas com base em condições (como estar logado).

12.6.1 Context API

À medida que as aplicações React crescem em tamanho e complexidade, um problema comum que você pode encontrar é a necessidade de compartilhar dados entre componentes que estão em diferentes partes da árvore de componentes. Quando um dado ou função precisa ser acessado por muitos componentes, passá-lo manualmente por meio de props de um componente para outro pode rapidamente se tornar complicado e difícil de gerenciar. Esse problema é conhecido como **prop drilling**.

O Problema do Prop Drilling

Prop drilling ocorre quando você precisa passar dados de um componente pai para um componente filho distante através de uma série de componentes intermediários que não têm interesse real nos dados. Esses componentes intermediários existem apenas para "transportar" as props, o que leva a código repetitivo, difícil de manter e sujeito a erros.

Imagine que você tem uma aplicação onde o nome do usuário autenticado precisa ser acessado em vários componentes. Sem o Context API, você teria que passar esse dado manualmente através de todos os componentes intermediários que não precisam realmente do nome do usuário, mas ainda assim precisam aceitá-lo como prop e repassá-lo adiante.

Por exemplo:

```
const App = () => {
  const username = 'John Doe';
  return <Layout username={username} />;
};

const Layout = ({ username }) => {
  return <Header username={username} />;
};

const Header = ({ username }) => {
  return <UserProfile username={username} />;
};

const UserProfile = ({ username }) => {
  return <div>Welcome, {username}!</div>;
};
```

Neste exemplo, o `username` precisa ser passado através do `Layout` e do `Header` apenas para que o `UserProfile` possa usá-lo, mesmo que `Layout` e `Header` não façam uso direto dessa prop. Isso é um exemplo claro de prop drilling.

Esse problema se agrava à medida que a aplicação cresce e mais dados precisam ser compartilhados entre componentes. A solução para isso é o uso do **Context API** do React.

O que é o Context API?

O **Context API** é uma ferramenta nativa do React que permite compartilhar valores (como dados ou funções) entre componentes, sem a necessidade de passar props manualmente por cada nível da árvore de componentes. Ele permite criar um "contexto" que pode ser acessado por qualquer componente dentro da árvore, eliminando a necessidade de prop drilling.

Com o Context API, você pode:

- Criar um contexto para armazenar dados globais que podem ser acessados em qualquer lugar da aplicação.
- Evitar prop drilling, simplificando a passagem de dados entre componentes.
- Melhorar a manutenção e a legibilidade do código, especialmente em aplicações grandes.

Quando Usar o Context API?

O Context API é útil quando você tem dados ou funções que precisam ser acessados por muitos componentes em diferentes partes da árvore de componentes. Exemplos comuns incluem:

- **Autenticação de Usuário:** Dados de autenticação como o nome do usuário, tokens de acesso, etc.
- **Temas Globais:** Gerenciamento de temas, como modo claro/escuro.
- **Configurações de Aplicação:** Preferências do usuário, como o idioma da interface.

Vamos criar um exemplo prático de como usar o Context API para gerenciar a autenticação do usuário na sua aplicação Wikipedia pessoal.

Exemplo Prático: Gerenciando Autenticação com Context API

Passo 1: Criando o Contexto de Autenticação

Primeiro, precisamos criar um contexto que armazenará as informações de autenticação do usuário e fornecerá funções para fazer login e logout.

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

// Definindo o tipo para o contexto de autenticação
interface AuthContextType {
  user: string | null;
  login: (username: string) => void;
  logout: () => void;
}

// Criando o contexto com um valor padrão
const AuthContext = createContext<AuthContextType | undefined>(undefined);

export const AuthProvider: React.FC<{ children: React.ReactNode }> = ({ children } ) => {
  const [user, setUser] = useState<string | null>(null);

  const login = (username: string) => {
    setUser(username);
  };

  const logout = () => {
    setUser(null);
  };

  return (
    <AuthContext.Provider value={{ user, login, logout }}>
      {children}
    </AuthContext.Provider>
  );
};
```

```
// Custom Hook para usar o contexto de autenticação
export const useAuth = () => {
  const context = useContext(AuthContext);
  if (context === undefined) {
    throw new Error('useAuth deve ser usado dentro de um Auth
Provider');
  }
  return context;
};
```

Neste código:

- `AuthContext` é o contexto que criamos para armazenar e compartilhar as informações de autenticação.
- `AuthProvider` é um componente que envolve outros componentes e fornece o contexto de autenticação para todos eles. Ele gerencia o estado de autenticação (`user`) e define funções para fazer login e logout.
- `useAuth` é um custom hook que facilita o acesso ao contexto de autenticação em qualquer componente da aplicação.

Passo 2: Integrando o AuthProvider no Projeto

Agora que temos o contexto de autenticação, precisamos integrá-lo na nossa aplicação. Envolvemos o `AuthProvider` em torno do componente principal da aplicação no `main.tsx`.

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import App from './App.tsx';
import './index.css';
import { ChakraProvider } from '@chakra-ui/react';
import { BrowserRouter } from 'react-router-dom';
import { AuthProvider } from './context/AuthContext';

ReactDOM.createRoot(document.getElementById('root')!).render(
  <ChakraProvider>
    <AuthProvider>
```

```
<BrowserRouter>
  <React.StrictMode>
    <App />
  </React.StrictMode>
</BrowserRouter>
</AuthProvider>
</ChakraProvider>
);
```

Com essa configuração, qualquer componente dentro do `App` pode acessar o contexto de autenticação diretamente, sem necessidade de prop drilling.

Passo 3: Usando o Contexto em Componentes

Vamos criar um componente `UserProfile` que mostra o nome do usuário logado e um botão para logout, utilizando o contexto de autenticação.

```
import React from 'react';
import { Box, Button, Text } from '@chakra-ui/react';
import { useAuth } from '../context/AuthContext';

const UserProfile: React.FC = () => {
  const { user, logout } = useAuth();

  return (
    <Box p={4} shadow="md" border="1px solid #ccc" mb={4}>
      {user ? (
        <>
          <Text>Bem-vindo, {user}!</Text>
          <Button mt={2} colorScheme="teal" onClick={logout}>
            Logout
          </Button>
        </>
      ) : (
        <Text>Nenhum usuário logado</Text>
      )}
    </Box>
  );
}
```

```
    );
}

export default UserProfile;
```

Neste exemplo:

- O `useAuth` é usado para acessar o estado de autenticação (nome do usuário) e as funções `login` e `logout` diretamente no componente `UserProfile`.
- Não há necessidade de passar o `user` como prop por uma cadeia de componentes intermediários, simplificando o código e evitando prop drilling.

Passo 4: Exibindo o UserProfile

Por fim, vamos adicionar o `UserProfile` ao nosso `App.tsx` para exibir o perfil do usuário na interface principal.

```
import React from 'react';
import { Box, Container } from '@chakra-ui/react';
import Header from './components/Header';
import SearchBar from './components/SearchBar';
import ArticleList from './components/ArticleList';
import UserProfile from './components/UserProfile';

const App: React.FC = () => {
  return (
    <Container maxW="container.md">
      <Header />
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/article/:id" element={<Article />} />
      </Routes>
    </Container>

  );
}
```

```
export default App;
```

Benefícios do Context API

- **Elimina Prop Drilling:** Com o Context API, você pode compartilhar dados entre componentes sem passar props por cada nível da árvore, tornando o código mais limpo e fácil de manter.
- **Centraliza Dados Globais:** Ideal para gerenciar estados globais como autenticação, temas, ou configurações de aplicação.
- **Flexibilidade e Facilidade:** O Context API é flexível e fácil de implementar, permitindo que você adicione e acesse dados globais em qualquer parte da aplicação.

Exercícios Práticos

1. **Gerenciamento Global de Estado:** Crie um contexto para gerenciar temas (modo claro/escuro) na sua aplicação e aplique o tema globalmente.
2. **Evitar Prop Drilling:** Identifique partes do seu código onde ocorre prop drilling e refatore para usar Context API.
3. **Exploração do Context API:** Experimente combinar múltiplos contextos para gerenciar estados globais diferentes, como autenticação e favoritos.

12.6.2 Gerenciando Estados Complexos com `useReducer`

Quando começamos a construir aplicações mais complexas, muitas vezes precisamos lidar com estados que envolvem múltiplas variáveis, ou que precisam ser atualizados de maneiras diferentes dependendo de diversas ações. Para esses cenários, o `useState` pode começar a parecer limitado ou difícil de gerenciar. É aqui que o `useReducer` se torna uma ferramenta poderosa e útil.

O que é o `useReducer` ?

O `useReducer` é um hook do React que fornece uma maneira alternativa de gerenciar o estado dos componentes. Ele é inspirado no conceito de **reducers**

das arquiteturas de fluxo de dados, como o Redux, e é particularmente útil quando o estado do componente é complexo e depende de várias ações para ser atualizado.

Em vez de simplesmente atualizar o estado com uma nova variável ou objeto (como fazemos com `useState`), com `useReducer`, você define uma função **reducer** que determina como o estado deve ser atualizado com base em uma **ação** específica.

A estrutura básica do `useReducer` envolve três partes principais:

1. **Estado Inicial:** O estado inicial que define o valor de partida do seu componente.
2. **Reducer:** Uma função que recebe o estado atual e uma ação e retorna o novo estado.
3. **Ações:** As operações que descrevem as mudanças que devem ser feitas no estado.

Quando Usar `useReducer` ?

Você deve considerar o uso do `useReducer` quando:

- O estado do seu componente é complexo e envolve múltiplas propriedades que precisam ser atualizadas de maneira coordenada.
- Você precisa lidar com múltiplas ações que afetam o estado de maneiras diferentes.
- Você quer tornar o código mais previsível e fácil de depurar, especialmente em componentes grandes.

Vamos construir um exemplo prático para ilustrar como isso funciona.

Exemplo Prático: Gerenciamento de Artigos Favoritos

Vamos usar `useReducer` para gerenciar uma lista de artigos favoritos em sua aplicação Wikipedia pessoal. O objetivo é permitir que o usuário adicione e remova artigos dos favoritos de maneira eficiente.

Passo 1: Criando a Função Reducer

Primeiro, precisamos criar uma função **reducer** que gerencie as ações de adicionar e remover artigos dos favoritos. A função reducer vai receber o estado atual e uma ação, e com base no tipo da ação, retornará um novo estado.

```
// Definindo o tipo de um artigo
interface Article {
  id: number;
  title: string;
}

// Definindo o tipo do estado dos favoritos, que é uma lista
// de artigos
type FavoritesState = Article[];

// Definindo as ações que podem ser realizadas no estado dos
// favoritos
type FavoritesAction =
  | { type: 'ADD_FAVORITE'; article: Article } // Ação para
  adicionar um artigo
  | { type: 'REMOVE_FAVORITE'; articleId: number }; // Ação
  para remover um artigo

// Implementando a função reducer
const favoritesReducer = (state: FavoritesState, action: FavoritesAction): FavoritesState => {
  switch (action.type) {
    case 'ADD_FAVORITE':
      // Retorna uma nova lista de favoritos com o novo artigo
      // adicionado
      return [...state, action.article];
    case 'REMOVE_FAVORITE':
      // Retorna uma nova lista de favoritos, removendo o artigo
      // com o ID correspondente
      return state.filter(article => article.id !== action.articleId);
    default:
```

```
        return state; // Retorna o estado atual por padrão se
a ação não for reconhecida
    }
};
```

Nesta função:

- `state` é o estado atual, que no nosso caso é uma lista de artigos favoritos.
- `action` é um objeto que descreve a ação que deve ser realizada. Cada ação tem um tipo (`type`) que descreve o que deve ser feito (por exemplo, adicionar ou remover um favorito) e possivelmente carrega outros dados relevantes (como o artigo a ser adicionado ou o ID do artigo a ser removido).
- A função usa uma estrutura de controle `switch` para decidir como o estado deve ser atualizado com base no tipo da ação.

Passo 2: Usando `useReducer` no Componente

Agora que temos a função reducer, podemos usá-la dentro de um componente para gerenciar a lista de artigos favoritos.

```
import React, { useReducer } from 'react';
import { Box, Button, Text } from '@chakra-ui/react';

const FavoriteArticles: React.FC = () => {
    // Inicializando o useReducer com o reducer e um estado ini
    cial (lista vazia)
    const [favorites, dispatch] = useReducer(favoritesReducer,
    []);

    // Função para adicionar um artigo aos favoritos
    const addFavorite = (article: Article) => {
        dispatch({ type: 'ADD_FAVORITE', article });
    };

    // Função para remover um artigo dos favoritos
    const removeFavorite = (articleId: number) => {
        dispatch({ type: 'REMOVE_FAVORITE', articleId });
    };
}
```

```

};

return (
  <Box>
    <Text fontWeight="bold">Artigos Favoritos</Text>
    <ul>
      {favorites.map(article => (
        <li key={article.id}>
          {article.title}
          <Button
            ml={2}
            size="sm"
            onClick={() => removeFavorite(article.id)}
            colorScheme="red"
          >
            Remover
          </Button>
        </li>
      ))}
    </ul>
    <Button
      mt={4}
      colorScheme="teal"
      onClick={() => addFavorite({ id: 1, title: 'React com
TypeScript' })}
    >
      Adicionar "React com TypeScript" aos Favoritos
    </Button>
  </Box>
);
};

export default FavoriteArticles;

```

Neste exemplo:

- `useReducer` é inicializado com a nossa função `favoritesReducer` e um estado inicial (uma lista vazia).
- O `dispatch` é uma função que usamos para enviar ações para o reducer. Quando `dispatch` é chamado com uma ação, o reducer decide como atualizar o estado com base naquela ação.
- As funções `addFavorite` e `removeFavorite` são exemplos de como você pode despachar ações para modificar o estado.

Benefícios do `useReducer`

- **Organização:** Ao encapsular a lógica de atualização do estado em uma função reducer, o código fica mais organizado e modular.
- **Previsibilidade:** Como o reducer é uma função pura (não tem efeitos colaterais), é mais fácil de testar e depurar, garantindo que a mesma ação sempre produza o mesmo resultado dado o mesmo estado.
- **Escalabilidade:** À medida que o estado se torna mais complexo, `useReducer` permite lidar com diferentes tipos de ações de uma maneira clara e estruturada.

Exercícios Práticos

1. **Integração do componente desenvolvido:** Implemente o componente `FavoriteArticles` na aplicação para terminar a funcionalidade de favoritar artigos
2. **Gerenciamento Global de Estado:** Crie um contexto para gerenciar temas (modo claro/escuro) na sua aplicação e aplique o tema globalmente.
3. **Gerenciamento Complexo de Estado:** Use `useReducer` para gerenciar o estado de uma lista de tarefas ou artigos lidos.
4. **Exploração do Context API:** Experimente combinar múltiplos contextos para gerenciar estados globais diferentes, como autenticação e favoritos.

12.7 Formulários

Em muitas aplicações web, a interação do usuário se dá através de formulários. Seja para autenticação, envio de informações ou busca de dados, os formulários

são uma parte crucial da interface do usuário. Neste tópico, vamos criar uma tela de login para a sua aplicação **Wikipedia pessoal** utilizando React e Chakra UI. Vamos focar na estrutura do formulário e na validação dos dados inseridos pelo usuário.

12.7.1 Criando a Tela de Login

Vamos começar criando um componente de tela de login. Esse componente incluirá um formulário simples que solicitará o nome de usuário e a senha do usuário.

Passo 1: Criando o Componente de Login

Crie um novo arquivo `Login.tsx` na pasta `pages` para o componente de login.

```
// src/pages/Login.tsx
import React, { useState } from 'react';
import { Box, Button, FormControl, FormLabel, Input, FormError
rMessage } from '@chakra-ui/react';
import { useAuth } from '../context/AuthContext';
import { useNavigate } from 'react-router-dom';

const Login: React.FC = () => {
  const { login } = useAuth();
  const navigate = useNavigate();
  const [username, setUsername] = useState('');
  const [password, setPassword] = useState('');
  const [errors, setErrors] = useState<{ username?: string; p
assword?: string }>({});

  const handleLogin = () => {
    const validationErrors: { username?: string; password?: s
tring } = {};

    if (username.trim() === '') {
      validationErrors.username = 'Nome de usuário é obrigató
rio';
    }
  }
}
```

```

    if (password.trim() === '') {
      validationErrors.password = 'Senha é obrigatória';
    }

    if (Object.keys(validationErrors).length > 0) {
      setErrors(validationErrors);
    } else {
      // Se a validação passar, fazemos login e navegamos para a home
      login(username);
      navigate('/');
    }
  };

  return (
    <Box maxW="sm" mx="auto" mt={10} p={4} shadow="md" border
Width="1px">
    <FormControl id="username" isInvalid={!errors.username} mb={4}>
      <FormLabel>Nome de Usuário</FormLabel>
      <Input
        type="text"
        value={username}
        onChange={(e) => setUsername(e.target.value)}
      />
      {errors.username && <FormErrorMessage>{errors.username}</FormErrorMessage>}
    </FormControl>

    <FormControl id="password" isInvalid={!errors.password} mb={4}>
      <FormLabel>Senha</FormLabel>
      <Input
        type="password"
        value={password}
        onChange={(e) => setPassword(e.target.value)}
      />

```

```

        />
      {errors.password && <FormErrorMessage>{errors.password}</FormErrorMessage>}
    </FormControl>

    <Button colorScheme="teal" onClick={handleLogin} width="full">
      Login
    </Button>
  </Box>
);
};

export default Login;

```

Neste código:

- Utilizamos o `useState` para gerenciar os valores do nome de usuário e da senha.
- O `useAuth` é utilizado para acessar a função `login` do contexto de autenticação.
- Implementamos validação básica no lado do cliente, onde verificamos se os campos de nome de usuário e senha foram preenchidos.
- O `useNavigate` do React Router é usado para redirecionar o usuário para a página inicial após um login bem-sucedido.

Passo 2: Integrando a Rota de Login

Vamos agora adicionar uma rota para a tela de login em nossa aplicação. Para isso, atualize o arquivo `App.tsx`.

```

import React from 'react';
import { Routes, Route } from 'react-router-dom';
import { Container } from '@chakra-ui/react';
import Home from './pages/Home';
import Article from './pages/Article';

```

```

import Login from './pages/Login'; // Importando a tela de login
import Header from './components/Header';

const App: React.FC = () => {
  return (
    <Container maxW="container.md">
      <Header />
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/login" element={<Login />} />
        <Route path="/article/:id" element={<Article />} />
      </Routes>
    </Container>
  );
};

export default App;

```

Com isso, ao acessar a URL `/login`, a tela de login será exibida.

12.7.2 Estrutura do Formulário e Validação

Um aspecto importante ao trabalhar com formulários é garantir que os dados inseridos pelos usuários sejam válidos. Vamos discutir algumas práticas e técnicas para estruturação e validação de formulários em React.

Estrutura do Formulário

A estrutura básica de um formulário em React pode ser construída utilizando componentes de entrada (como `<Input>` e `<Textarea>`) e controlando o estado dos dados inseridos com `useState`. Além disso, é importante fornecer feedback ao usuário quando ele insere dados inválidos, utilizando componentes como `<FormErrorMessage>` do Chakra UI.

Validação do Formulário

A validação pode ocorrer tanto no lado do cliente quanto no lado do servidor. No exemplo acima, implementamos uma validação simples no lado do cliente, verificando se os campos obrigatórios estão preenchidos. Em uma aplicação real, você pode querer expandir isso para incluir:

- **Validação de Formato:** Como verificar se um email é válido ou se uma senha atende aos requisitos mínimos.
- **Validação de Dados Cruzados:** Como garantir que a confirmação da senha corresponde à senha original.
- **Validação Assíncrona:** Como verificar se um nome de usuário já está em uso, chamando uma API.

Para validações mais complexas, você pode considerar o uso de bibliotecas como **Formik** ou **React Hook Form** combinadas com **Yup** para validação de esquemas.

Exercício Prático

Implementar uma Rota Protegida: Agora que você tem uma tela de login funcional, como exercício, implemente uma rota protegida em sua aplicação. Essa rota deve exigir que o usuário esteja autenticado antes de acessá-la. Se o usuário não estiver autenticado, ele deve ser redirecionado para a tela de login. Utilize o contexto de autenticação que criamos no tópico anterior para verificar o estado de autenticação.

Com isso, você agora tem uma compreensão sólida de como criar e validar formulários em React, e está preparado para implementar funcionalidades mais avançadas, como rotas protegidas, em sua aplicação.

módulo 13

projeto final

Aplique todo o seu conhecimento em um projeto completo de gerenciador de despesas. 

35 tópicos neste módulo

Capítulo 13: Projeto final

O projeto final deste e-book será um **Gerenciador de Despesas** 💰, onde você colocará em prática todo o conhecimento adquirido ao longo dos capítulos anteriores. Neste capítulo, vamos fornecer um guia detalhado de como planejar, estruturar e desenvolver esse projeto, sem incluir código diretamente. Vamos abordar desde a definição dos objetivos até a implementação das funcionalidades avançadas, fornecendo dicas, materiais complementares e orientações para cada etapa.

Você pode estar se perguntando o motivo de não ter nenhum código nesse capítulo, e a resposta é simples: "Você precisa praticar por conta própria! 💪", simplesmente copiar os códigos não é aprender, por isso veja esse capítulo como um desafio para você enfrentar, além de que irá se parecer muito mais com um cenário de um emprego real, onde as demandas chegam e você precisa desenvolver.

A seguir está todo o planejamento do projeto, com tudo que você precisa fazer de forma detalhada, se esforce, pois esse pode virar um ótimo projeto para seu portfólio. 🚀

13.1 Planejamento do Projeto

Antes de começar a codificar, é essencial realizar um bom planejamento. Esta fase é fundamental para garantir que o projeto seja bem estruturado e atenda às expectativas. Vamos abordar três pontos principais: **Definição de objetivos**, **Escolha das tecnologias** e **Modelagem do banco de dados**.

13.1.1 Definição de Objetivos

O primeiro passo no planejamento é definir claramente os objetivos do projeto. O Gerenciador de Despesas deve permitir aos usuários:

- Registrar despesas e receitas, categorizando-as de forma eficiente. 💰

- Visualizar um histórico detalhado de transações. 
- Gerar relatórios e gráficos para analisar gastos. 
- Configurar orçamentos mensais e metas financeiras. 
- Exportar e importar dados para backup ou migração. 

Uma inspiração que você pode consultar é o Mobills.

Dica: Pense nas funcionalidades principais que devem estar presentes desde o início (MVP - Produto Mínimo Viável) e em funcionalidades que podem ser adicionadas posteriormente.

13.1.2 Escolha das Tecnologias

Escolher as tecnologias adequadas é crucial para o sucesso do projeto. Aqui estão algumas sugestões baseadas nos conhecimentos adquiridos ao longo do e-book:

- **Back-end:** Node.js com Nest.js para criar uma API robusta e modular. 
- **Front-end:** React com TypeScript para construir uma interface de usuário interativa e bem tipada. 
- **Banco de Dados:** MySQL ou PostgreSQL para gerenciar de forma eficiente os dados transacionais. 
- **Estilização:** Styled-components e TailwindCSS para garantir um design moderno e responsivo. 
- **Autenticação:** JWT (JSON Web Tokens) para autenticação segura dos usuários. 
- **Documentação:** Swagger para documentar e testar as APIs. 

Dica: Escolha tecnologias que você já conhece bem, mas também considere explorar novas ferramentas que podem agregar valor ao projeto.

13.1.3 Modelagem do Banco de Dados

A modelagem do banco de dados é uma das partes mais importantes do planejamento. Ela define como os dados serão armazenados e relacionados. Aqui está uma sugestão básica de como modelar o banco de dados para o Gerenciador de Despesas:

- **Tabela de Usuários (`users`)**: Armazena informações dos usuários, como nome, e-mail, senha (hash) e data de criação.
- **Tabela de Transações (`transactions`)**: Armazena as despesas e receitas, com colunas para valor, categoria, data, tipo (despesa ou receita) e o usuário relacionado.
- **Tabela de Categorias (`categories`)**: Armazena as diferentes categorias de despesas e receitas.
- **Tabela de Orçamentos (`budgets`)**: Armazena os orçamentos mensais dos usuários, vinculando categorias e limites de gasto.

Dica: Utilize diagramas de entidade-relacionamento (ER) para visualizar a estrutura do banco de dados antes de implementá-lo.

Com esse planejamento em mãos, você estará pronto para avançar para as etapas de implementação, onde começaremos a construir o projeto. No próximo tópico, vamos abordar a **Estruturação do Back-end com Node.js e Nest.js**. 

13.2 Estruturação do Back-end com Node.js e Nest.js

Nesta seção, vamos planejar a estruturação do back-end do projeto **Gerenciador de Despesas** utilizando **Node.js** e **Nest.js**. Este guia servirá como um roadmap das funcionalidades e componentes que devem ser implementados no back-end do projeto.

13.2.1 Configuração Inicial

- **Objetivo:** Estabelecer a base do projeto, configurando o ambiente de desenvolvimento com Node.js e Nest.js.

- **Funcionalidades:** O projeto deverá estar preparado para utilizar um banco de dados relacional, como MySQL ou PostgreSQL, para armazenar todas as informações do sistema.
- **Passos Importantes:**
 - Configuração do ambiente de desenvolvimento.
 - Escolha e configuração do banco de dados.
 - Organização inicial do projeto com módulos, serviços e controladores.

13.2.2 Criação de Módulos e Serviços

- **Objetivo:** Modularizar o projeto de forma que cada responsabilidade esteja separada em um módulo específico.
- **Funcionalidades:**
 - Criação de módulos para diferentes entidades do sistema, como **Transações, Usuários, Categorias e Orçamentos**.
 - Cada módulo deve conter um serviço responsável por lidar com a lógica de negócio específica.
- **Considerações:**
 - Mantenha os módulos organizados e evite a sobrecarga de responsabilidade em um único módulo.
 - Os serviços devem encapsular toda a lógica de negócios, facilitando a manutenção e os testes.

13.2.3 Criação de Controllers

- **Objetivo:** Definir os pontos de entrada da aplicação, onde as requisições serão recebidas e processadas.
- **Funcionalidades:**

- Cada módulo deve ter um controlador associado que será responsável por gerenciar as rotas HTTP.
 - O controlador deve delegar a lógica de negócio para os serviços, garantindo a separação de responsabilidades.
- **Considerações:**
 - As rotas devem ser bem definidas e documentadas para facilitar a integração com o front-end e outros serviços.

13.2.4 Validação de Dados com o class-validator

- **Objetivo:** Garantir que os dados recebidos pela API estejam no formato correto e atendam aos requisitos de validação.
- **Funcionalidades:**
 - Implementar validações para os dados recebidos, como formato de e-mail, presença de campos obrigatórios, e valores numéricos dentro de uma faixa permitida.
- **Considerações:**
 - Utilize DTOs (Data Transfer Objects) para definir claramente a estrutura dos dados de entrada e aplicar as validações necessárias.
 - A validação de dados ajuda a proteger o sistema contra entradas inválidas e potenciais vulnerabilidades.

13.2.5 Conexão com Banco de Dados (TypeORM)

- **Objetivo:** Integrar o projeto com um banco de dados relacional, utilizando um ORM (Object-Relational Mapping) para gerenciar a persistência dos dados.
- **Funcionalidades:**
 - Configuração inicial do ORM para conectar ao banco de dados escolhido.
 - Definição de entidades que mapeiam as tabelas do banco de dados.

- Implementação de repositórios para gerenciar as operações de leitura e escrita no banco de dados.
- **Considerações:**
 - A estrutura do banco de dados deve ser cuidadosamente planejada para suportar as funcionalidades do sistema, como gerenciamento de transações, usuários e categorias.

13.2.6 Documentação das Rotas e Payloads com Swagger

- **Objetivo:** Documentar a API de forma clara e acessível, facilitando a integração com outras partes do sistema e com desenvolvedores externos.
 - **Funcionalidades:**
 - Configuração do Swagger para gerar automaticamente a documentação das rotas e dos parâmetros aceitos pela API.
 - Disponibilização de um endpoint de documentação acessível para facilitar o uso e a consulta durante o desenvolvimento.
 - **Considerações:**
 - A documentação deve ser completa e sempre atualizada conforme a API evolui, garantindo que todos os desenvolvedores tenham uma referência confiável.
-

13.3 Criação do Front-end com React e TypeScript

Nesta seção, vamos expandir o planejamento do front-end do **Gerenciador de Despesas** utilizando **React** e **TypeScript**. O foco é oferecer um guia detalhado para a criação dos componentes e páginas que irão compor a interface do usuário, além de definir as configurações e estrutura necessárias.

13.3.1 Configuração Inicial

- **Objetivo:** Estabelecer a base do projeto, incluindo a instalação e configuração das principais dependências e ferramentas.
- **Pontos Importantes:**
 - Inicialize um projeto React com TypeScript utilizando **Vite** ou **Create React App**.
 - Instale e configure bibliotecas adicionais, como **React Router** para rotas e **styled-components** para estilização.
 - Configure um sistema de gerenciamento de estado, como **Context API** ou **Redux**, dependendo da complexidade do projeto.

13.3.2 Estrutura de Pastas e Componentes

- **Objetivo:** Organizar o projeto de forma modular, garantindo que os componentes e estilos estejam bem estruturados.
- **Pontos Importantes:**
 - Defina uma estrutura de pastas que separe componentes, estilos, rotas e páginas de forma clara e intuitiva.
 - Crie componentes básicos reutilizáveis, como **Botões**, **Formulários**, **Modal**, entre outros.
 - Garanta que os componentes sejam tipados corretamente, utilizando os recursos do TypeScript para validar as props e estados.

13.3.3 Configuração de Rotas

- **Objetivo:** Implementar a navegação entre as diferentes páginas da aplicação, utilizando o **React Router**.
- **Pontos Importantes:**
 - Defina as rotas principais da aplicação, como **Dashboard**, **Registro de Despesas**, **Relatórios**, e **Configurações**.

- Utilize **Rotas Protegidas** para garantir que apenas usuários autenticados possam acessar determinadas páginas.
- Implemente rotas dinâmicas para páginas que possam exigir parâmetros na URL, como detalhes de uma despesa específica.

13.3.4 Configuração do styled-components

- **Objetivo:** Estabelecer um sistema de estilização consistente para o projeto, utilizando **styled-components**.
- **Pontos Importantes:**
 - Configure um **ThemeProvider** para gerenciar temas globais, como paleta de cores, tipografia e espaçamentos.
 - Crie componentes estilizados para os elementos principais da interface, garantindo uma aparência moderna e responsiva.
 - Considere a utilização de animações e transições para melhorar a experiência do usuário.

13.3.5 Criação dos Componentes

- **Objetivo:** Desenvolver os componentes fundamentais que irão compor as páginas da aplicação.
- **Pontos Importantes:**
 - Crie componentes como **Card de Despesa**, **Gráfico de Relatórios**, **Formulário de Cadastro de Despesa**, entre outros.
 - Garanta que todos os componentes sejam reutilizáveis e extensíveis, facilitando a manutenção e adição de novas funcionalidades.
 - Utilize as vantagens do TypeScript para assegurar que as props e os estados estejam devidamente tipados, evitando erros comuns.

13.3.6 Criação das Páginas

- **Objetivo:** Desenvolver as páginas principais do Gerenciador de Despesas, conectando os componentes criados anteriormente.
 - **Páginas Principais:**
 - **Dashboard:** Página inicial que oferece uma visão geral das despesas recentes, metas e gráficos de desempenho.
 - **Registro de Despesas:** Formulário para adicionar novas despesas, com campos para valor, data, categoria e descrição.
 - **Relatórios:** Página dedicada à visualização de relatórios detalhados, com gráficos e filtros para análise das despesas.
 - **Configurações:** Página para que o usuário possa ajustar preferências, como categorias de despesas, notificações e outros parâmetros personalizados.
 - **Histórico de Transações:** Página para visualização detalhada e gerenciamento do histórico completo de despesas.
-

13.4 Integração Front-end e Back-end

Neste tópico, o foco será na integração do front-end e back-end do **Gerenciador de Despesas**. Vamos discutir como realizar essa comunicação de forma eficiente, garantir a segurança e tratar os possíveis erros que possam surgir.

13.4.1 Comunicação via API

- **Objetivo:** Estabelecer a comunicação entre o front-end (React) e o back-end (Node.js/Nest.js) utilizando APIs RESTful.
- **Pontos Importantes:**
 - **Endpoints:** Identifique os principais endpoints que o front-end precisará consumir, como criação, leitura, atualização e exclusão de despesas, além

de autenticação e relatórios.

- **Métodos HTTP:** Utilize os métodos HTTP adequados para cada operação (GET para leitura, POST para criação, PUT/PATCH para atualização, DELETE para exclusão).
- **Bibliotecas de Requisição:** Escolha uma biblioteca como **Axios** ou a API nativa **fetch** para realizar as requisições do front-end ao back-end.
- **Formatação de Dados:** Garanta que o formato de dados entre o front-end e back-end seja consistente, utilizando JSON como padrão.
- **Assincronismo:** Utilize funções assíncronas para lidar com as requisições, tratando respostas e erros de forma adequada.

13.4.2 Tratamento de Erros

- **Objetivo:** Implementar uma estratégia robusta para capturar e lidar com erros tanto no front-end quanto no back-end.
- **Pontos Importantes:**
 - **Validação de Erros no Back-end:** No back-end, utilize middlewares para captura de erros e retorne respostas padronizadas, incluindo códigos de status HTTP apropriados (ex.: 400 para erros de validação, 401 para erros de autenticação, 500 para erros internos do servidor).
 - **Exibição de Erros no Front-end:** No front-end, capture os erros durante as requisições e exiba mensagens claras para o usuário, utilizando componentes de notificação ou modal para feedback imediato.
 - **Logs e Monitoramento:** Considere implementar logs tanto no front-end quanto no back-end para monitorar a ocorrência de erros e realizar um diagnóstico mais preciso.
 - **Fallback e Retentativas:** Em situações onde a falha possa ser transitória (ex.: problemas de rede), implemente retentativas automáticas ou ofereça opções para o usuário tentar novamente.

13.4.3 Autenticação de Usuários

- **Objetivo:** Proteger as rotas sensíveis e os dados dos usuários através de um sistema de autenticação seguro.
- **Pontos Importantes:**
 - **JWT (JSON Web Tokens):** Utilize JWT para autenticação. O back-end deve emitir um token JWT após o login do usuário, que será armazenado no front-end (em **localStorage** ou **cookies**).
 - **Rotas Protegidas:** Implemente rotas protegidas no front-end, onde o usuário só poderá acessar determinadas páginas se estiver autenticado. Verifique a presença e validade do token JWT antes de permitir o acesso.
 - **Middleware de Autenticação:** No back-end, use middlewares para verificar o token JWT em todas as rotas que requerem autenticação. Caso o token seja inválido ou esteja ausente, retorne uma resposta de erro apropriada (ex.: 401 Unauthorized).
 - **Renovação de Token:** Considere a implementação de renovação de tokens para manter a sessão do usuário ativa por mais tempo, sem precisar de um novo login a cada requisição.
 - **LocalStorage:** Utilize o **localStorage** para armazenar o token JWT após o login do usuário. O localStorage é uma opção adequada para armazenar tokens, pois persiste entre as sessões do navegador.

13.5 Implementação de Funcionalidades Avançadas

Neste tópico, vamos discutir as funcionalidades mais avançadas que podem ser implementadas no **Gerenciador de Despesas** para torná-lo uma aplicação robusta e completa. Estas funcionalidades irão além das operações básicas, oferecendo aos usuários uma experiência mais rica e informativa.

13.5.1 Registrando suas despesas

- **Objetivo:** Permitir que os usuários registrem suas despesas de maneira simples e intuitiva.
- **Pontos Importantes:**
 - **Formulário de Registro:** Crie um formulário onde o usuário possa inserir detalhes da despesa, como categoria, valor, data e uma breve descrição.
 - **Categorias Personalizadas:** Ofereça a possibilidade de criar categorias personalizadas, permitindo maior flexibilidade na organização das despesas.
 - **Validação de Dados:** Implemente validações para garantir que os dados inseridos sejam consistentes e completos, como impedir o registro de valores negativos ou sem uma categoria atribuída.

13.5.2 Adição de gráficos e relatórios

- **Objetivo:** Fornecer uma visualização clara das despesas através de gráficos e relatórios.
- **Pontos Importantes:**
 - **Gráficos Interativos:** Utilize bibliotecas como Chart.js ou D3.js para criar gráficos que mostrem a distribuição das despesas por categoria, evolução mensal, etc.
 - **Relatórios Personalizados:** Permita que os usuários gerem relatórios com filtros personalizados, como por período, categoria, ou até mesmo por palavras-chave na descrição.
 - **Exportação de Relatórios:** Ofereça opções para exportar relatórios em formatos como PDF ou Excel, permitindo que os usuários mantenham um histórico offline de suas finanças.

13.5.3 Importação e exportação de dados

- **Objetivo:** Facilitar a importação e exportação de dados financeiros, para que o usuário possa migrar suas informações ou realizar backups.

- **Pontos Importantes:**

- **Formatos Suportados:** Implemente a importação e exportação em formatos comuns, como CSV e JSON.
- **Mapeamento de Campos:** Durante a importação, ofereça um mapeamento de campos para garantir que os dados sejam integrados corretamente ao sistema.
- **Backups Automáticos:** Considere adicionar uma funcionalidade de backup automático, onde o usuário possa configurar backups regulares dos seus dados.

13.5.4 Pesquisa, categorização e filtragem avançada de despesas

- **Objetivo:** Facilitar a busca e organização das despesas registradas, oferecendo ferramentas avançadas de pesquisa e categorização.

- **Pontos Importantes:**

- **Pesquisa Avançada:** Adicione uma barra de pesquisa que permita buscar despesas por nome, descrição, categoria ou valor.
- **Filtros Dinâmicos:** Implemente filtros que possam ser combinados, como filtrar despesas de um determinado mês que sejam superiores a um certo valor e pertencentes a uma categoria específica.
- **Tags:** Ofereça a funcionalidade de tags para uma categorização ainda mais granular, permitindo que o usuário adicione palavras-chave às despesas.

13.5.5 Configuração de metas e orçamentos mensais

- **Objetivo:** Ajudar os usuários a gerenciar melhor suas finanças através da configuração de metas e orçamentos.

- **Pontos Importantes:**

- **Definição de Metas:** Permita que os usuários estabeleçam metas de economia ou limites de gastos para determinadas categorias.

- **Alertas e Notificações:** Implemente alertas que avisem o usuário quando ele estiver próximo de atingir o limite de uma meta ou orçamento.
- **Resumo Mensal:** Crie um resumo mensal que mostre como o usuário está em relação às suas metas e orçamentos, destacando áreas de preocupação ou sucesso.

13.5.6 Histórico de transações detalhado

- **Objetivo:** Fornecer aos usuários um histórico completo de suas transações, permitindo uma análise detalhada de suas finanças ao longo do tempo.
 - **Pontos Importantes:**
 - **Linha do Tempo:** Apresente as transações em uma linha do tempo, facilitando a visualização da ordem cronológica dos gastos.
 - **Análise de Padrões:** Ofereça insights automáticos sobre padrões de despesas, como gastos recorrentes ou meses de maior consumo.
 - **Exportação de Histórico:** Assim como os relatórios, permita que os usuários exportem seu histórico completo em formatos como CSV ou PDF.
-

Conclusão

O desenvolvimento do **Gerenciador de Despesas** é um projeto desafiador e enriquecedor que permitirá a você aplicar e aprimorar habilidades em diversas tecnologias e práticas de desenvolvimento de software. Do back-end robusto com **Node.js** e **Nest.js** ao front-end interativo com **React** e **TypeScript**, cada etapa do projeto contribui para a construção de uma aplicação completa e funcional. 

Ter um projeto desse porte em seu portfólio é extremamente valioso. Ele não apenas demonstra suas habilidades técnicas, mas também sua capacidade de planejar, organizar e executar um projeto do início ao fim. Um portfólio bem elaborado pode abrir portas para novas oportunidades e mostrar aos empregadores ou clientes o seu verdadeiro potencial. 

Lembre-se de que cada linha de código e cada funcionalidade implementada são passos em direção ao seu crescimento profissional. Não tenha medo de enfrentar desafios e aprender com os erros. A jornada do desenvolvimento é cheia de altos e baixos, mas a recompensa vem com a experiência adquirida e o sucesso alcançado. Mantenha-se motivado e continue se esforçando para alcançar seus objetivos. 

Boa sorte em sua jornada de desenvolvimento e que este projeto seja apenas o começo de muitas realizações e conquistas em sua carreira. Nunca pare de aprender, de inovar e de se desafiar. O futuro é brilhante para aqueles que se dedicam e acreditam em seu potencial. 

módulo 14

docker

Containerize suas aplicações e
simplifique o deploy com Docker. 

06 tópicos neste módulo

Capítulo 14: Docker

14.1 Introdução ao Docker 🚢

Docker é uma ferramenta **revolucionária** que mudou a forma como desenvolvemos, testamos e implantamos aplicações. Com ele, você pode criar ambientes consistentes, portáteis e isolados para suas aplicações, facilitando o desenvolvimento e a produção. Vamos explorar o que torna o Docker tão especial e por que você deve considerar usá-lo em seus projetos.

14.1.1 O que é Docker? 🔧

Docker é uma **plataforma de contêineres** que permite empacotar sua aplicação e todas as suas dependências em um contêiner. Isso significa que você pode rodar a mesma aplicação em diferentes ambientes, como na sua máquina local, em um servidor de produção, ou na nuvem, sem precisar se preocupar com configurações e dependências do sistema. 💻

Um **contêiner** é uma unidade leve, portátil e independente que contém tudo o que a aplicação precisa para funcionar: código, runtime, bibliotecas e configurações do sistema. É como uma **mini máquina virtual**, mas muito mais leve e eficiente. 😎

14.1.2 Por que usar Docker? 🤔

Docker oferece vários benefícios que o tornam uma ferramenta essencial para desenvolvedores e equipes de DevOps:

- **Portabilidade:** Com Docker, você pode garantir que sua aplicação rodará da mesma maneira em qualquer ambiente. Se funciona no seu contêiner, funcionará em qualquer lugar! 🌎
- **Isolamento:** Cada contêiner é isolado dos outros, permitindo que você execute múltiplas aplicações ou versões da mesma aplicação no mesmo sistema, sem conflitos. 🧩

- **Escalabilidade:** Docker facilita o escalonamento de aplicações, permitindo que você inicie múltiplas instâncias dos seus contêineres quando precisar de mais capacidade de processamento. 
- **Facilidade no Desenvolvimento:** Docker permite que desenvolvedores configurem rapidamente ambientes de desenvolvimento consistentes, eliminando o clássico problema de "funciona na minha máquina". 

14.1.3 A Evolução do Docker

Docker foi lançado em 2013 e rapidamente se tornou popular por sua abordagem inovadora de contêineres. Antes do Docker, a virtualização era a solução comum para isolar ambientes, mas com uma sobrecarga significativa. Docker mudou o jogo ao oferecer uma forma mais leve e eficiente de isolamento.

Desde seu lançamento, Docker evoluiu para suportar uma ampla gama de funcionalidades, desde **orquestração de contêineres** com Kubernetes até **integração contínua** com ferramentas de DevOps. Hoje, é uma das ferramentas mais utilizadas em ambientes de desenvolvimento e produção. 

14.1.4 Quando Usar Docker?

Docker é ideal para diversos cenários, como:

- **Desenvolvimento Local:** Configure ambientes que replicam a produção para garantir que o código funcione em todos os lugares.
- **Implantação na Nuvem:** Docker é amplamente suportado por provedores de nuvem, facilitando a implantação de suas aplicações em qualquer lugar.
- **Testes Automatizados:** Crie ambientes de teste isolados para validar mudanças de código sem afetar a produção.
- **Microserviços:** Em arquiteturas de microserviços, Docker permite que cada serviço seja executado em seu próprio contêiner, facilitando o desenvolvimento, teste e implantação. 

Docker não é apenas uma ferramenta técnica; é uma nova maneira de pensar sobre como construímos, testamos e implantamos software. Ao longo deste capítulo, vamos explorar como Docker pode transformar o seu fluxo de trabalho, tornando-o mais eficiente e consistente. 

14.2 Instalação e Configuração

Antes de começarmos a usar o Docker, é importante garantir que ele esteja instalado corretamente no seu sistema. Vamos ver como realizar a instalação e configuração inicial para que tudo esteja pronto para começar a explorar o mundo dos contêineres. 

14.2.1 Instalando o Docker

A instalação do Docker é bem direta, mas varia de acordo com o sistema operacional que você está utilizando. Vamos abordar os principais sistemas:

1. Windows

- Docker Desktop é a opção mais comum para usuários do Windows. Ele fornece uma interface gráfica amigável para gerenciar contêineres.
- Certifique-se de que a virtualização esteja habilitada na BIOS do seu computador.
- Após a instalação, o Docker Desktop iniciará automaticamente.

2. macOS

- No macOS, você também pode usar o Docker Desktop. Ele é fácil de instalar e configurar.
- Assim como no Windows, o Docker Desktop para macOS também oferece uma interface gráfica.

3. Linux

- No Linux, a instalação do Docker é geralmente feita através do terminal, utilizando o gerenciador de pacotes da sua distribuição (apt, yum, dnf, etc.).
- Como o Docker foi originalmente desenvolvido para Linux, ele roda de forma nativa e com alta performance nesse sistema.

Para todos os sistemas, a instalação envolve **baixar o instalador oficial do Docker** e seguir as instruções na tela. O site oficial do Docker fornece guias detalhados para cada sistema operacional: [Docker Installation Guides](#). 

14.2.2 Verificando a Instalação

Depois de instalar o Docker, é importante verificar se tudo está funcionando corretamente. Isso pode ser feito com um simples comando no terminal:

```
docker --version
```

Esse comando retorna a versão do Docker instalada, confirmando que a instalação foi bem-sucedida. Se você ver algo como `Docker version 20.xx.xx, build xxxxxxxx`, significa que tudo está funcionando como esperado. 

Além disso, você pode testar a instalação executando o comando a seguir para rodar um contêiner simples do Docker:

```
docker run hello-world
```

Esse comando baixa uma imagem de teste do Docker e executa um contêiner que exibe uma mensagem de sucesso. Se você vir a mensagem "Hello from Docker!", isso confirma que o Docker está funcionando corretamente. 

14.2.3 Configuração Inicial

Após instalar o Docker, você pode ajustar algumas configurações básicas para otimizar seu uso:

- **Configuração de Recursos:** No Docker Desktop (Windows/macOS), você pode ajustar a quantidade de CPU, memória e outros recursos disponíveis para os contêineres. Isso é útil se você estiver executando contêineres pesados ou múltiplos contêineres simultaneamente.
- **Compartilhamento de Drives:** No Docker Desktop, certifique-se de que os drives corretos estão compartilhados. Isso permite que o Docker acesse arquivos e pastas no seu sistema operacional.
- **Atualizações Automáticas:** É importante manter o Docker atualizado para obter os últimos recursos e correções de segurança. Você pode configurar o Docker Desktop para verificar e instalar atualizações automaticamente.

14.2.4 Pronto para Avançar?

Com o Docker instalado e configurado, você está pronto para começar a explorar e utilizar todo o potencial dos contêineres. No próximo tópico, vamos nos aprofundar nos **fundamentos do Docker**, entendendo o que são contêineres, imagens, e como tudo isso se encaixa no seu fluxo de trabalho.

Prepare-se para dar os primeiros passos no Docker e ver como ele pode transformar a maneira como você desenvolve e implanta aplicações! 

14.3 Fundamentos do Docker

Agora que o Docker está instalado e configurado, vamos explorar os fundamentos que fazem dele uma ferramenta tão poderosa no mundo do desenvolvimento de software. 

14.3.1 Containers

Containers são a espinha dorsal do Docker. Eles são como pequenas "caixas" que encapsulam tudo o que uma aplicação precisa para rodar: código, bibliotecas, variáveis de ambiente, e tudo mais. O que diferencia os contêineres

de uma máquina virtual é que eles compartilham o kernel do sistema operacional, tornando-os muito mais leves e rápidos. ✨

- **Isolamento:** Cada contêiner roda de forma isolada dos outros, o que significa que você pode ter múltiplos contêineres rodando diferentes versões de uma aplicação sem conflitos.
- **Portabilidade:** Como os contêineres incluem todas as dependências necessárias, eles podem ser executados em qualquer lugar onde o Docker esteja instalado, seja no seu computador, em um servidor de nuvem, ou até mesmo em um Raspberry Pi!

14.3.2 Imagens

Imagens são os "blueprints" dos contêineres. Elas contêm as instruções necessárias para criar um contêiner. Pense em uma imagem como um modelo que define o que o contêiner deve ter e como deve se comportar.

- **Camadas:** Imagens no Docker são construídas em camadas. Cada comando no Dockerfile (um arquivo que define a imagem) adiciona uma nova camada. Isso torna as imagens altamente reutilizáveis e eficientes.
- **Repositórios de Imagem:** Imagens podem ser armazenadas em repositórios (registries), como o Docker Hub, onde você pode baixar e compartilhar imagens com a comunidade.

14.3.3 Registries

Registries são como bibliotecas onde as imagens do Docker são armazenadas e gerenciadas. O **Docker Hub** é o registry mais popular e público, mas você pode configurar seu próprio registry privado para armazenar imagens internas da sua empresa ou projeto.

- **Docker Hub:** É o registry padrão do Docker e contém milhões de imagens públicas que você pode usar para diferentes tipos de aplicações e serviços.
- **Registries Privados:** Para maior controle e segurança, você pode configurar registries privados, onde apenas pessoas autorizadas podem acessar as

imagens.

14.3.4 Docker Engine

O **Docker Engine** é o coração do Docker. É o que realmente faz o Docker funcionar, permitindo que você construa e rode contêineres.

- **Componentes:** O Docker Engine é composto por um daemon, que gerencia os contêineres, imagens, redes, volumes e outros objetos Docker. O CLI (Command Line Interface) é a ferramenta que você usa para interagir com o daemon.
- **Plataforma de Contêineres:** O Docker Engine torna possível executar contêineres de maneira rápida e eficiente em diferentes ambientes, desde o seu ambiente local de desenvolvimento até a produção em larga escala.

14.3.5 Docker Desktop

Docker Desktop é a interface gráfica que facilita o uso do Docker em sistemas Windows e macOS. Ele torna a administração e o gerenciamento de contêineres e imagens mais acessível, especialmente para iniciantes.

- **Gerenciamento Simplificado:** Docker Desktop permite gerenciar contêineres, imagens, volumes e redes através de uma interface gráfica, o que é útil para quem prefere evitar o terminal.
- **Integração:** Ele também integra outras ferramentas importantes do Docker, como o Docker Compose, que facilita o gerenciamento de múltiplos contêineres para ambientes mais complexos.

14.4 Trabalhando com Imagens Docker

Neste tópico, vamos explorar como criar e gerenciar imagens Docker. Vamos entender a estrutura de um **Dockerfile**, construir uma imagem a partir desse

arquivo e aprender a gerenciar imagens Docker. 

14.4.1 Dockerfile: Estrutura e Sintaxe



O **Dockerfile** é um arquivo de texto simples que contém uma lista de comandos que o Docker utiliza para montar uma imagem. Vamos criar um Dockerfile para uma aplicação Node.js com TypeScript.

Um Dockerfile típico pode ser dividido nas seguintes seções:

1. **Instrução FROM:** Especifica a imagem base a partir da qual a nova imagem será construída.
2. **Instrução MAINTAINER:** Define o mantenedor da imagem (embora esta prática esteja sendo substituída pelo uso de labels).
3. **Instrução RUN:** Executa comandos no container durante o processo de construção.
4. **Instrução COPY/ADD:** Copia arquivos e diretórios para a imagem.
5. **Instrução CMD/ENTRYPOINT:** Define o comando que será executado quando um container é iniciado a partir da imagem.
6. **Instrução EXPOSE:** Declara a porta na qual o container escutará durante a execução.
7. **Instrução ENV:** Define variáveis de ambiente.
8. **Instrução VOLUME:** Cria um ponto de montagem para volumes.

Exemplo de Dockerfile:

```
# Usando uma imagem base oficial do Node.js com suporte a TypeScript
FROM node:20

# Configurando o diretório de trabalho
```

```
WORKDIR /app

# Instalando dependências
COPY package*.json ./
RUN npm install

# Copiando o código da aplicação
COPY . .

# Compilando TypeScript
RUN npm run build

# Expondo a porta que a aplicação utilizará
EXPOSE 3000

# Comando para iniciar a aplicação
CMD ["npm", "start"]
```

Explicação dos comandos:

- `FROM`: Especifica a imagem base, que neste caso é o Node.js.
- `WORKDIR`: Define o diretório de trabalho dentro do contêiner.
- `COPY`: Copia os arquivos do projeto para o contêiner.
- `RUN`: Executa comandos dentro do contêiner, como instalar dependências e compilar o TypeScript.
- `EXPOSE`: Informa ao Docker qual porta a aplicação vai utilizar.
- `CMD`: Especifica o comando que será executado quando o contêiner iniciar.

14.4.2 Criando e Construindo Imagens Docker

Com o Dockerfile pronto, podemos construir nossa imagem Docker.

O comando `docker build` cria uma imagem a partir de um Dockerfile e de um contexto de construção. O contexto é o conjunto de arquivos no diretório atual ou

em um caminho especificado.

Comando para construir uma imagem:

```
docker build -t minha-app-ts .
```

Aqui, `-t minha-app-ts` define o nome da imagem e `.` indica o diretório atual onde o Dockerfile está localizado.

Otimizando Imagens

Para criar imagens eficientes e pequenas, siga estas práticas recomendadas:

- 1. Minimize a quantidade de camadas:** Combine múltiplas instruções RUN em uma única instrução.
- 2. Use imagens base menores:** Escolha imagens base menores como `alpine` sempre que possível.
- 3. Aproveite o cache de construção:** Estruture seu Dockerfile para maximizar o uso do cache, colocando instruções que mudam menos frequentemente no topo.

14.4.3 Gerenciamento de Imagens Docker

Depois de criar a imagem, você pode querer gerenciar as imagens disponíveis no seu ambiente Docker.

Baixando (pull) uma imagem do Docker Hub :

O comando `docker pull` baixa uma imagem de um registry (como o Docker Hub) para o seu host local.

```
docker pull meu-repo/minha-app:1.0
```

Enviando (push) uma imagem para o Docker Hub :

O comando `docker push` envia uma imagem do host local para um registry.

```
docker push meu-repo/minha-app:1.0
```

Listando Imagens 📋:

```
docker images
```

Renomeando (tag) uma imagem 🟡:

O comando `docker tag` cria uma nova tag para uma imagem existente.

```
docker tag minha-app-ts meu-repo/minha-app:1.0
```

Removendo uma imagem 🗑:

```
docker rmi minha-app-ts
```

Esses comandos ajudam você a gerenciar suas imagens Docker de maneira eficiente, facilitando o compartilhamento e o controle de versões.

14.4.4 Exercícios Práticos 🎯

1. **Crie um Dockerfile** para uma aplicação Node.js que utiliza TypeScript, como no exemplo, mas adicionando suas próprias configurações e scripts de build.
2. **Construa a imagem** usando o Dockerfile que você criou e execute o contêiner para garantir que tudo funcione corretamente.
3. **Experimente gerenciar as imagens criadas**, renomeando-as, enviando para um repositório, e removendo as que não forem mais necessárias.
4. **Modifique o Dockerfile** para usar uma imagem base mais leve, como `node:alpine`, e compare o tamanho das imagens resultantes.

5. **Documente e explique** cada etapa do seu Dockerfile e como ela contribui para o funcionamento da aplicação.
-

14.5 Trabalhando com Containers Docker

Os **containers Docker** são as instâncias em execução de suas imagens Docker. Eles são a base de todo o conceito de Docker, permitindo que você execute aplicações de maneira isolada e consistente, independentemente do ambiente. Neste tópico, vamos nos aprofundar em como criar, gerenciar e interagir com containers Docker usando diversos comandos essenciais.

14.5.1 Criando Containers

Para criar um container, você utiliza o comando `docker run`. Este comando combina várias operações em uma única linha: ele cria, inicia e, opcionalmente, interage com o container.

Exemplo:

```
docker run -d --name meu-container -p 8080:80 minha-app-ts
```

Explicação:

- `-d`: Roda o container em background (modo detachado).
- `--name`: Atribui um nome ao container.
- `-p 8080:80`: Mapeia a porta 8080 do host para a porta 80 do container.
- `minha-app-ts`: Nome da imagem que será usada para criar o container.

14.5.2 Comando `docker run`

O comando `docker run` é o mais utilizado no gerenciamento de containers. Ele pode ser configurado com diversas opções:

Exemplo:

```
docker run -it --rm ubuntu bash
```

Explicação:

- `it`: Inicia o container em modo interativo, permitindo interação via terminal.
- `-rm`: Remove o container automaticamente após ele ser parado.
- `ubuntu bash`: Usa a imagem `ubuntu` e executa o `bash` dentro do container.

14.5.3 Comando `docker stop`

Quando precisar parar um container em execução, utilize o comando `docker stop`.

Exemplo:

```
docker stop meu-container
```

Esse comando envia um sinal SIGTERM para o processo principal do container, permitindo que ele finalize suas operações de forma ordenada.

14.5.4 Comando `docker ps`

Para listar todos os containers em execução, utilize o comando `docker ps`.

Exemplo:

```
docker ps
```

Esse comando exibe uma lista com informações detalhadas dos containers ativos, como seus IDs, nomes, status e portas mapeadas.

Para listar todos os containers, incluindo os que estão parados, utilize:

```
docker ps -a
```

14.5.5 Comando `docker start` ►

O comando `docker start` reinicia containers que foram previamente parados.

Exemplo:

```
docker start meu-container
```

Isso retoma o container no mesmo estado em que ele estava antes de ser parado.

14.5.6 Comando `docker rm` 🗑

Quando não precisar mais de um container, você pode removê-lo com o comando `docker rm`.

Exemplo:

```
docker rm meu-container
```

Para remover todos os containers parados, utilize:

```
docker container prune
```

14.5.7 Comando `docker inspect` 🕵️

O comando `docker inspect` permite obter informações detalhadas sobre um container ou uma imagem.

Exemplo:

```
docker inspect meu-container
```

Isso retorna um JSON com detalhes completos do container, incluindo suas configurações de rede, volumes, variáveis de ambiente e muito mais.

14.5.8 Comando `docker logs`

Para visualizar os logs de um container, utilize o comando `docker logs`.

Exemplo:

```
docker logs meu-container
```

Você pode também seguir os logs em tempo real com a opção `-f`:

```
docker logs -f meu-container
```

14.5.9 Comando `docker exec`

O comando `docker exec` é utilizado para executar comandos dentro de um container que já está em execução.

Exemplo:

```
docker exec -it meu-container bash
```

Isso abre um terminal interativo dentro do container, permitindo que você execute comandos diretamente.

14.5.10 Exercícios Práticos

1. **Crie um container** usando uma imagem Docker existente e experimente as diferentes opções do comando `docker run`.
2. **Interrompa e reinicie** containers usando os comandos `docker stop` e `docker start`.
3. **Remova containers** que não estão mais em uso e utilize `docker container prune` para limpar seu ambiente.

4. **Inspecione um container** em execução para visualizar detalhes como configuração de rede e volumes montados.
 5. **Visualize e siga logs** de um container em tempo real, usando o comando `docker logs`.
 6. **Execute comandos** dentro de um container em execução com `docker exec` e explore o sistema de arquivos do container.
-

14.6 Docker Compose: Orquestrando seus Containers



O **Docker Compose** é uma ferramenta poderosa que permite definir e gerenciar múltiplos containers como um único serviço. Ele é especialmente útil para aplicações que envolvem vários serviços interconectados, como uma aplicação web com um banco de dados. Neste tópico, vamos nos aprofundar na utilização do Docker Compose, desde a estrutura básica de um arquivo `docker-compose.yml` até os comandos essenciais para gerenciar seus serviços.

14.6.1 Arquivo docker-compose.yml

O arquivo `docker-compose.yml` é onde você define a configuração dos seus serviços Docker. Ele utiliza a sintaxe YAML, que é simples e legível.

Estrutura Básica



A estrutura básica de um arquivo `docker-compose.yml` inclui definições de serviços, volumes, redes, e outras configurações específicas. Vamos explorar essa estrutura com um exemplo:

```
version: '3'  
services:  
  web:  
    image: 'node:14'
```

```

ports:
  - '3000:3000'
volumes:
  - './app:/usr/src/app'
environment:
  - NODE_ENV=development
command: 'npm start'

db:
  image: 'mysql:5.7'
  ports:
    - '3306:3306'
  environment:
    MYSQL_ROOT_PASSWORD: 'example'
    MYSQL_DATABASE: 'mydb'
  volumes:
    - db_data:/var/lib/mysql

volumes:
  db_data:

```

Explicação do Exemplo:

- **version:** Define a versão do Docker Compose que você está usando.
- **services:** Define os serviços que serão gerenciados pelo Docker Compose.
 - **web:** Um serviço usando a imagem `node:14` para uma aplicação Node.js.
 - **ports:** Mapeia a porta 3000 do container para a porta 3000 do host.
 - **volumes:** Monta o diretório local `./app` no diretório `/usr/src/app` do container.
 - **environment:** Define variáveis de ambiente para o serviço.
 - **command:** Comando a ser executado quando o serviço é iniciado.
 - **db:** Um serviço usando a imagem `mysql:5.7` para um banco de dados MySQL.

- **ports**: Mapeia a porta 3306 do container para a porta 3306 do host.
 - **environment**: Define variáveis de ambiente, como senha do root e nome do banco de dados.
 - **volumes**: Monta um volume chamado `db_data` no diretório `/var/lib/mysql` do container.
- **volumes**: Define volumes persistentes para os dados do serviço.

14.6.2 Comando `docker-compose up`

O comando `docker-compose up` é utilizado para iniciar todos os serviços definidos no arquivo `docker-compose.yml`. Ele cria e inicia os containers em ordem, conforme especificado no arquivo.

Exemplo:

```
docker-compose up
```

Opções úteis:

- `-d`: Inicia os serviços em modo detachado (background).
- `--build`: Recompila as imagens antes de iniciar os serviços.

```
docker-compose up -d --build
```

Esse comando é útil quando você faz alterações no Dockerfile ou em qualquer configuração relacionada à construção das imagens.

14.6.3 Comando `docker-compose down`

O comando `docker-compose down` é utilizado para parar e remover todos os containers, redes e volumes criados pelo `docker-compose up`.

Exemplo:

```
docker-compose down
```

Opções úteis:

- `v`: Remove os volumes associados aos serviços.

```
docker-compose down -v
```

Isso é útil quando você deseja limpar completamente todos os recursos criados pelo Docker Compose, incluindo os dados armazenados em volumes.

14.6.4 Outros Comandos Úteis

Além dos comandos principais `up` e `down`, o Docker Compose oferece uma série de outros comandos úteis:

- `docker-compose ps`: Lista todos os containers gerenciados pelo Docker Compose.

```
docker-compose ps
```

- `docker-compose logs`: Exibe os logs de todos os serviços.

```
docker-compose logs
```

- `docker-compose exec`: Executa comandos em um container em execução, semelhante ao `docker exec`.

```
docker-compose exec web bash
```

- `docker-compose restart`: Reinicia um ou mais serviços.

```
docker-compose restart web
```

- `docker-compose pull`: Puxa as imagens mais recentes para todos os serviços definidos no arquivo `docker-compose.yml`.

```
docker-compose pull
```

14.6.5 Exercícios Práticos

1. **Crie um arquivo** `docker-compose.yml` para uma aplicação simples, como uma aplicação web em Node.js conectada a um banco de dados MySQL.
2. **Inicie os serviços** utilizando `docker-compose up` e experimente rodá-los em modo detachado.
3. **Verifique os logs** de um serviço em execução usando `docker-compose logs`.
4. **Execute comandos dentro de um container** com `docker-compose exec`.
5. **Reinicie um serviço** com `docker-compose restart` e veja como o estado do serviço é mantido.
6. **Remova todos os serviços e recursos** utilizando `docker-compose down` com a opção `-v` para limpar os volumes.

14.7 Volumes: Gerenciando Dados Persistentes com Docker

Volumes no Docker são uma maneira eficiente de persistir dados fora do ciclo de vida dos containers. Eles permitem que os dados sobrevivam a reinicializações de containers e são essenciais para aplicações que precisam armazenar informações de forma duradoura. Neste tópico, vamos explorar os diferentes tipos de volumes, como criar e usar volumes, além de exemplos práticos de uso.

14.7.1 Tipos de Volumes

No Docker, existem principalmente três tipos de volumes:

1. **Volumes Gerenciados:** São volumes criados e gerenciados pelo Docker. Eles são armazenados fora do diretório de dados do container, normalmente em um local específico no host, mas são completamente gerenciados pelo Docker.

Exemplo:

```
docker volume create my_volume
```

2. **Bind Mounts:** Permitem que você mapeie um diretório específico do host para um diretório dentro do container. Ao contrário dos volumes gerenciados, os bind mounts dão mais controle ao usuário, mas requerem que o diretório no host exista.

Exemplo:

```
docker run -v /path/host:/path/container my_image
```

3. **tmpfs Mounts:** São volumes temporários armazenados na memória do host. Eles são usados para armazenar dados voláteis que não precisam ser persistidos após o término do container.

Exemplo:

```
docker run --mount type=tmpfs,tmpfs-size=64M,destination=/app tmpfs-container
```

14.7.2 Criando e Usando Volumes

Volumes gerenciados pelo Docker são a forma mais comum de persistir dados. Eles podem ser criados explicitamente antes de usar ou automaticamente ao iniciar um container com a flag `-v` ou `--mount`.

Criando um Volume

Você pode criar um volume com o comando:

```
docker volume create my_volume
```

Esse comando cria um volume chamado `my_volume`, que pode ser usado em qualquer container.

Usando um Volume em um Container

Uma vez que o volume está criado, você pode montá-lo em um container:

```
docker run -d --name my_container -v my_volume:/app/data my_image
```

Nesse exemplo, o diretório `/app/data` dentro do container está montado no volume `my_volume`. Qualquer dado salvo nesse diretório será persistido, mesmo que o container seja destruído.

14.7.3 Exemplos de Uso

Vamos ver alguns exemplos práticos de como volumes podem ser utilizados para diferentes propósitos:

Exemplo 1: Persistindo Dados de um Banco de Dados

Imagine que você está rodando um banco de dados MySQL em um container Docker. Para garantir que os dados sejam preservados mesmo que o container seja reiniciado ou recriado, você pode montar um volume:

```
docker run -d --name mysql_db \
-e MYSQL_ROOT_PASSWORD=my_password \
-v mysql_data:/var/lib/mysql \
mysql:latest
```

Nesse exemplo, todos os dados do MySQL serão armazenados no volume `mysql_data`, garantindo sua persistência.

Exemplo 2: Desenvolvimento com Volumes

Durante o desenvolvimento de uma aplicação, é comum mapear o diretório de código-fonte do host diretamente no container. Isso permite que você edite os arquivos no host e veja as mudanças refletidas instantaneamente no container.

```
docker run -d --name dev_container \
-v /path/to/source:/usr/src/app \
-p 3000:3000 \
node:14 \
npm start
```

Nesse caso, o código-fonte da aplicação é montado diretamente no container, facilitando o desenvolvimento iterativo.

14.7.4 Gerenciando Volumes

Docker oferece comandos para gerenciar volumes, como listar, inspecionar, e remover volumes que não são mais necessários:

- **Listando Volumes:**

```
docker volume ls
```

- **Inspecionando um Volume:**

```
docker volume inspect my_volume
```

- **Removendo um Volume:**

```
docker volume rm my_volume
```

Esses comandos são úteis para manter seu ambiente Docker organizado e garantir que volumes antigos ou não utilizados não ocupem espaço desnecessário.

14.7.5 Exercícios Práticos

1. **Crie um volume** usando o comando `docker volume create` e monte-o em um container de banco de dados.
 2. **Experimente criar um bind mount** entre um diretório do host e um diretório dentro do container para ver como as mudanças no host são refletidas no container.
 3. **Liste e inspecione os volumes** criados para entender melhor como o Docker gerencia o armazenamento.
 4. **Remova volumes antigos** e desnecessários do seu ambiente Docker para manter a organização.
-

14.8 Networking: Conectando Containers no Docker

O Docker Networking é um dos pilares que permite a comunicação entre containers e outros serviços na infraestrutura. Com uma compreensão sólida de networking no Docker, você será capaz de criar ambientes de rede complexos e garantir que seus containers possam se comunicar de forma eficiente e segura. Neste tópico, vamos explorar os conceitos fundamentais do Docker Networking, os diferentes tipos de redes disponíveis, como configurar redes personalizadas, além de exemplos práticos.

14.8.1 Conceitos Fundamentais de Networking no Docker

No Docker, a rede é um conjunto de regras e diretrizes que permite a comunicação entre containers e entre containers e o mundo exterior. O Docker fornece suporte nativo a múltiplos drivers de rede, cada um com características e usos específicos.

Tipos de Redes Docker

1. **Bridge Network** (Rede Padrão) :

- **Uso:** É a rede padrão criada pelo Docker, ideal para comunicação entre containers no mesmo host.
- **Comportamento:** Containers conectados à mesma bridge network podem se comunicar entre si usando seus nomes de host.

Exemplo:

```
docker network create my_bridge_network
```

2. Host Network:

- **Uso:** Utiliza diretamente a rede do host. Ideal para quando o container precisa de baixa latência e não requer isolamento de rede.
- **Comportamento:** Containers que usam a host network compartilham o mesmo stack de rede que o host.

Exemplo:

```
docker run --network host my_image
```

3. Overlay Network

- **Uso:** Utilizada em clusters de Docker Swarm para permitir a comunicação entre containers em diferentes hosts.
- **Comportamento:** Facilita a comunicação distribuída em uma configuração de cluster.

Exemplo:

```
docker network create --driver overlay my_overlay_network
```

4. Macvlan Network:

- **Uso:** Atribui um endereço MAC a cada container, tornando-o um dispositivo na rede física.
- **Comportamento:** Ideal para casos onde os containers precisam aparecer como dispositivos físicos na rede.

Exemplo:

```
docker network create -d macvlan --subnet=192.168.1.0/24 -  
-gateway=192.168.1.1 -o parent=eth0 my_macvlan_network
```

14.8.2 Criando e Gerenciando Redes Docker

Criando uma Nova Rede

Para criar uma nova rede Docker, você pode usar o comando `docker network create`. A seguir, criamos uma bridge network:

```
docker network create my_custom_network
```

Containers conectados a essa rede podem se comunicar entre si usando seus nomes de host.

Conectando Containers a uma Rede

Para conectar um container a uma rede específica no momento da criação:

```
docker run -d --name my_container --network my_custom_network  
my_image
```

Você também pode conectar um container a uma rede existente:

```
docker network connect my_custom_network my_existing_container
```

Desconectando e Removendo Redes

Para desconectar um container de uma rede:

```
docker network disconnect my_custom_network my_container
```

Para remover uma rede que não está em uso:

```
docker network rm my_custom_network
```

14.8.3 Exemplo Prático: Configurando a Comunicação entre Containers

Vamos criar dois containers que se comunicam usando uma rede personalizada:

- 1. Crie uma rede personalizada:**

```
docker network create my_app_network
```

- 2. Inicie um container de banco de dados na rede:**

```
docker run -d --name my_database --network my_app_network  
postgres
```

- 3. Inicie uma aplicação web na mesma rede:**

```
docker run -d --name my_webapp --network my_app_network my  
_webapp_image
```

Agora, a aplicação web pode se conectar ao banco de dados usando o nome do container `my_database` como host.

14.8.4 Segurança em Docker Networking

Docker oferece várias opções para garantir a segurança na comunicação entre containers:

- **Isolamento de Redes:** Containers em redes diferentes não podem se comunicar entre si, a menos que estejam explicitamente conectados à mesma rede.
- **IPTables:** Docker configura automaticamente regras de firewall no host para gerenciar o tráfego entre containers e redes externas.
- **TLS:** Para proteger a comunicação entre containers em diferentes hosts, você pode usar certificados TLS.

14.8.5 Exercícios Práticos

1. **Crie uma rede bridge personalizada** e conecte dois containers a ela. Teste a comunicação entre eles usando comandos básicos como `ping`.
 2. **Implemente uma overlay network** em um cluster Docker Swarm e experimente a comunicação entre containers em diferentes hosts.
 3. **Explore a segurança** configurando regras de IPTables ou usando Macvlan networks para isolar a comunicação entre containers.
 4. **Conecte um container a múltiplas redes** e observe como ele pode atuar como um ponto de comunicação entre duas redes diferentes.
-

14.9 Registries e Repositórios de Imagem

Os Docker registries e repositórios de imagem são componentes fundamentais para o gerenciamento de imagens Docker. Eles permitem que você armazene, distribua e compartilhe suas imagens de maneira eficiente, facilitando a colaboração e o deployment de aplicações em diferentes ambientes. Vamos explorar como funcionam os registries e repositórios, desde os conceitos básicos até a criação e uso de registries privados.

14.9.1 O que são Docker Registries?

Um **Docker Registry** é um serviço que armazena e distribui imagens Docker. Ele pode ser hospedado na nuvem, como o Docker Hub, ou em servidores privados, oferecendo flexibilidade no gerenciamento das imagens.

- **Docker Hub:** É o registry oficial mantido pela Docker. Ele oferece acesso a milhares de imagens públicas, além de permitir a criação de repositórios privados.
- **Registries Privados:** São registries hospedados em servidores próprios ou em provedores de nuvem, permitindo maior controle sobre a distribuição de imagens.

Vantagens dos Docker Registries

- **Distribuição:** Facilitam o compartilhamento de imagens entre equipes e ambientes.
- **Automação:** Integração com pipelines CI/CD para automatizar o build e deployment de imagens.
- **Segurança:** Controle sobre o acesso e permissão das imagens, garantindo que apenas usuários autorizados possam utilizá-las.

14.9.2 Repositórios de Imagem

Um **Repositório Docker** é um local dentro de um registry onde uma ou mais versões de uma imagem Docker são armazenadas. Ele organiza as imagens por nome e tag, facilitando o versionamento e o acesso.

- **Exemplo de Reppositório no Docker Hub:** `username/repository_name:tag`
 - `username` é o nome de usuário no Docker Hub.
 - `repository_name` é o nome do repositório.
 - `tag` é a versão específica da imagem.

Criando e Utilizando Repositórios

1. **Push de Imagens:** Após criar uma imagem Docker, você pode enviá-la para um repositório.

```
docker push username/repository_name:tag
```

2. **Pull de Imagens:** Para baixar uma imagem do repositório, use:

```
docker pull username/repository_name:tag
```

3. **Listando Imagens em um Re却toreio:** Você pode listar as versões disponíveis de uma imagem com:

```
docker search repository_name
```

14.9.3 Configurando um Docker Registry Privado

Além do Docker Hub, você pode configurar seu próprio Docker Registry privado, o que é útil para empresas que desejam manter controle total sobre suas imagens.

Passos para Configurar um Docker Registry Privado:

1. **Rodando o Docker Registry:**

- O Docker fornece uma imagem oficial para rodar um registry localmente.

```
docker run -d -p 5000:5000 --name registry registry:2
```

2. **Empurrando Imagens para o Registry Privado:**

- Tagueie a imagem com o endereço do registry:

```
docker tag my_image localhost:5000/my_image
```

- Envie a imagem para o registry:

```
docker push localhost:5000/my_image
```

3. **Baixando Imagens do Registry Privado:**

- Baixe a imagem do registry:

```
docker pull localhost:5000/my_image
```

14.9.4 Gerenciamento de Imagens em Registries

Gerenciando Tags e Versionamento

Usar tags para versionar suas imagens é uma prática recomendada. Isso ajuda a garantir que você saiba exatamente qual versão de uma aplicação ou serviço está rodando em produção.

```
docker tag my_image my_image:v1.0  
docker push username/repository_name:v1.0
```

Limpeza de Imagens e Repositórios

Manter seu registry organizado é essencial, especialmente quando você está lidando com múltiplas versões de imagens. Você pode remover imagens antigas para liberar espaço:

```
docker rmi repository_name:tag
```

14.9.5 Exercícios Práticos

1. **Crie uma conta no Docker Hub** e publique uma imagem de exemplo. Explore como versionar imagens com tags.
2. **Configure um Docker Registry privado** localmente e armazene uma imagem personalizada nele.
3. **Automatize o build e o push de imagens** para um registry privado usando um pipeline CI/CD.

4. Implemente políticas de retenção de imagens no seu registry privado para gerenciar o armazenamento de versões antigas.