



CURSO

React JS

 @alexandrejunior.dev

Bem-vindo(a)!

O React JS é uma biblioteca frontend em JavaScript de código aberto com foco em criar interfaces de usuário em páginas web. Entre as grandes vantagens de se utilizar esse framework, podemos citar o fato dele transformar uma mesma tela em partes individuais (componentes), facilitando o trabalho do desenvolvedor sobre cada uma delas.

Criada em 2011 pelo Facebook (atual Meta), a biblioteca se tornou uma ferramenta de código aberto em 2013, ou seja, qualquer pessoa pode baixar, alterar e distribuir alterações no seu código-fonte. Ela foi desenvolvida a fim de otimizar a conexão entre atividades simultâneas operadas na rede social, como por exemplo a atualização de tarefas (status, chat e notificações).

Desde então, o React se tornou uma das bibliotecas JavaScript mais utilizadas em todo o mundo. Entre as organizações que usam a ferramenta estão Netflix, Airbnb e WhatsApp.

O React utiliza um conceito de componentes para organizar e estruturar a interface do usuário, e funciona em conjunto com outras bibliotecas e ferramentas, como o React Router para gerenciamento de rotas, e o Redux para gerenciamento de estado.

Neste curso, vamos explorar os principais conceitos e funcionalidades do React.js, desde a instalação e configuração, até o desenvolvimento de aplicações mais complexas.

Pré-requisitos

Fundamentos de lógica de programação e conceitos básicos de HTML, CSS e JavaScript.

Ferramentas

- Node.js (versão igual ou superior a 16)
- Visual Studio Code IDE (recomendado)

Repositório do Curso

O código-fonte dos exemplos realizados neste curso podem ser encontrados no GitHub, através do link:

<https://github.com/alexandresjunior>

Sumário

1. Linguagem JavaScript
2. Instalação e Configuração
3. Estrutura de um Componente
4. Propriedades e Estados
5. React Router Dom: Navegando entre Páginas
6. Context API: Gerenciamento de Estados Globais
7. Integração com APIs Externas com Axios
8. Conclusão

Linguagem JavaScript

Na maioria das interações online, temos um cliente/usuário que envia uma **requisição** (*request*) HTTP para um servidor, que envia de volta uma **resposta** (*response*) HTTP. Todo o código backend (seja em Java, Python, C# etc.) que escrevemos é executado em um servidor.

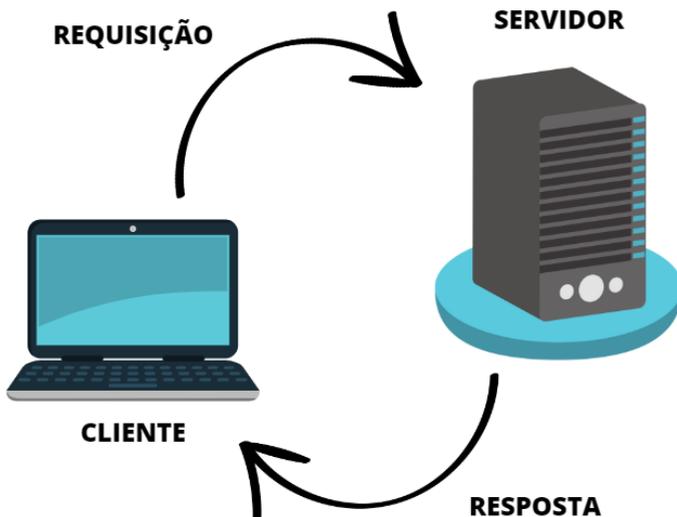


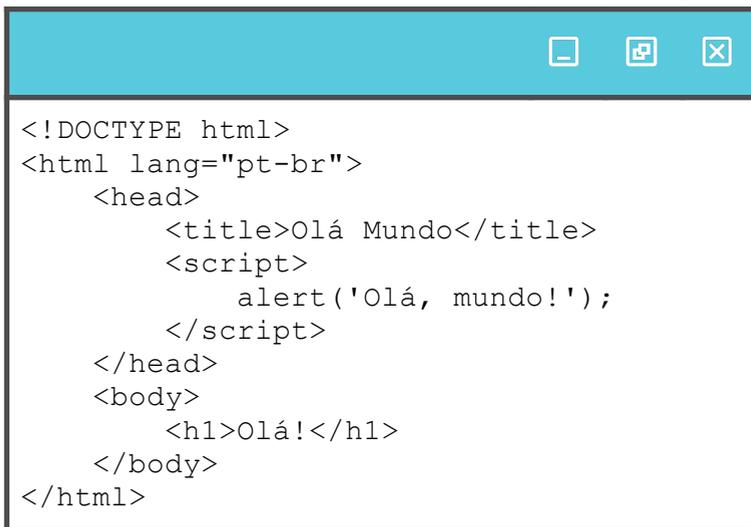
Figura 1: Esquema de interação online cliente-servidor.

O JavaScript nos permitirá executar o código no lado do **cliente**, sem que nenhuma interação com o

servidor seja necessária durante a execução, tornando nossos sites muito mais **interativos**.

Para adicionar um pouco de JavaScript à nossa página HTML, podemos adicionar um par de tags `<script>` em algum lugar da nossa página HTML.

Usamos essa tag para sinalizar ao navegador que tudo o que escrevemos entre as duas tags é o código JavaScript que desejamos executar quando um usuário visita nosso site ou executa uma ação.

A screenshot of a code editor window with a light blue header containing three icons: a square, a magnifying glass, and a close button. The main area is white and contains the following HTML code:

```
<!DOCTYPE html>
<html lang="pt-br">
  <head>
    <title>Olá Mundo</title>
    <script>
      alert('Olá, mundo!');
    </script>
  </head>
  <body>
    <h1>Olá!</h1>
  </body>
</html>
```

Figura 2: Uso de tags `<script>` para adição de código JavaScript em uma página HTML.

Nosso primeiro programa pode ser algo como isto:

```
alert('Hello, world!');
```

A função `alert` do JavaScript exibe uma mensagem para o usuário na forma de um **pop-up**. Para mostrar onde isso se encaixaria em uma página HTML na prática, a **Figura 2** mostra um exemplo de uma página HTML simples com um pouco de JavaScript embutido.

Eventos

Um recurso do JavaScript que o torna útil para a programação web é o suporte à **programação orientada a eventos**.

A programação orientada a eventos é um paradigma de programação centrado na detecção de **eventos** e nas **ações** que devem ser executadas quando um evento é detectado.

Um evento pode ser um botão sendo clicado, o cursor sendo movido, uma resposta sendo digitada ou uma página sendo carregada. Quase tudo que um usuário faz para interagir com uma página web pode ser considerado um evento. Em JavaScript, usamos **Event Listeners** que aguardam a ocorrência de determinados eventos e, em seguida, executam algum código.

Vamos começar transformando nosso JavaScript acima em uma função chamada `dizerOla`:

```
function dizerOla() {  
    alert('Olá, mundo!')  
}
```

Agora, vamos fazer esta função ser executada sempre que um botão for clicado. Para isso, criaremos um botão HTML em nossa página com o atributo **onclick**, que dá ao navegador as instruções do que deve acontecer quando o botão for clicado:

```
<button onclick="dizerOla()">  
    Enviar  
</button>
```

Essas alterações nos permitem esperar para executar partes do nosso código JavaScript até que um determinado evento ocorra.

Variáveis

JavaScript é uma linguagem de programação como Python, Java, C ou qualquer outra linguagem com a qual você já trabalhou, o que significa que possui muitos dos mesmos recursos de outras linguagens, incluindo variáveis. Existem três palavras-chave que podemos usar para atribuir valores em JavaScript:

- **var:** usado para definir uma variável globalmente.

```
var idade = 20;
```

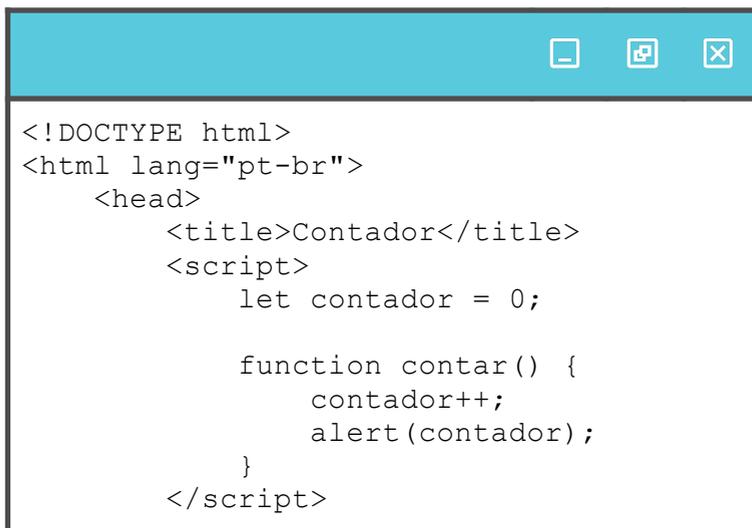
- **let:** usado para definir uma variável cujo escopo é limitado ao bloco atual, como uma função ou *loop*.

```
let contador = 1;
```

- **const:** usado para definir um valor que não mudará.

```
const pi = 3.14;
```

Na **Figura 3** temos um exemplo de como podemos usar uma variável, vamos dar uma olhada em uma página que monitora um contador:

A screenshot of a code editor window with a teal header bar containing window control icons (minimize, maximize, close). The main area is white and contains the following code:

```
<!DOCTYPE html>
<html lang="pt-br">
  <head>
    <title>Contador</title>
    <script>
      let contador = 0;

      function contar() {
        contador++;
        alert(contador);
      }
    </script>
```

```
</head>
<body>
  <h1>Olá, mundo!</h1>
  <button
onclick="contar()">Contar</button>
</body>
</html>
```

Figura 3: Uso de variáveis em JavaScript.

querySelector

Além de nos permitir exibir mensagens por meio de alertas, o JavaScript também nos permite alterar elementos na página. Para fazer isso, devemos primeiro introduzir uma função chamada **document.querySelector**. Esta função procura e retorna elementos do **Document Object Model (DOM)**.

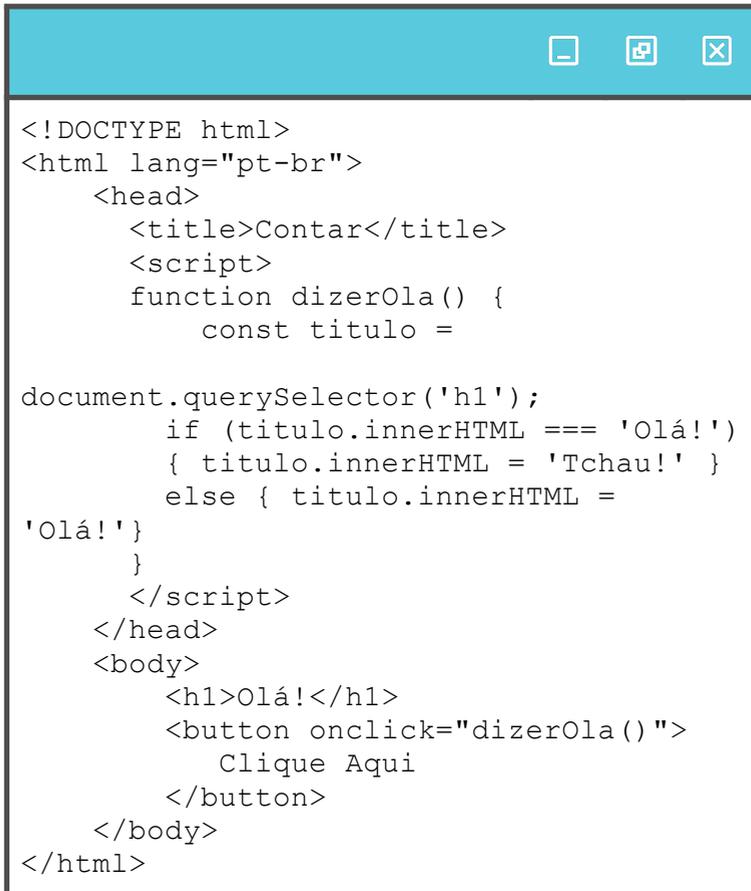
Por exemplo, usamos:

```
let titulo =
document.querySelector('h1');
```

para extrair um título. Então, para manipular o conteúdo desse elemento, podemos alterar sua propriedade **innerHTML**:

```
titulo.innerHTML = `Tchau!`;
```

Também podemos tirar vantagem das condicionais no JavaScript. Por exemplo, digamos que, em vez de sempre alterar nosso título para "Tchau!", desejamos alternar entre "Olá!" e "Tchau!", nossa página terá então a seguinte aparência:



```
<!DOCTYPE html>
<html lang="pt-br">
  <head>
    <title>Contar</title>
    <script>
      function dizerOla() {
        const titulo =
document.querySelector('h1');
        if (titulo.innerHTML === 'Olá!')
          { titulo.innerHTML = 'Tchau!' }
        else { titulo.innerHTML =
'Olá!' }
      }
    </script>
  </head>
  <body>
    <h1>Olá!</h1>
    <button onclick="dizerOla()">
      Clique Aqui
    </button>
  </body>
</html>
```

Figura 4: Exemplo de obtenção de elementos do DOM.

Observe que em JavaScript, usamos `===` como uma comparação mais forte entre dois itens que também verifica se os objetos são do mesmo tipo. Normalmente usaremos essa notação sempre que possível.

Manipulação do DOM

```

<!DOCTYPE html>
<html lang="pt-br">
  <head>
    <title>Contar</title>
    <script>
      let contador = 0;
      function contar() {
        contador++;

        document.querySelector('h1')
          .innerHTML =
contador;
      }
    </script>
  </head>
  <body>
    <h1>0</h1>
    <button onclick="contar()">
      Contar
    </button>
  </body>

```

```
</html>
```

Figura 5: Exemplo de manipulação de elementos do DOM. Vamos usar essa ideia de manipulação do DOM para melhorar nossa página de contador (ver **Figura 5**).

Podemos tornar essa página ainda mais interessante exibindo um alerta toda vez que o contador chegar a um múltiplo de dez. Neste alerta, queremos formatar uma string para customizar a mensagem, o que em JavaScript podemos fazer usando **template literals**.

Durante o seu uso, exige-se que haja acentos graves (') ao redor de toda a expressão e um **\$** e **chaves** ao redor de quaisquer substituições.

Por exemplo, vamos mudar nossa função de contagem:

```
function contar() {
    contador++;
    document.querySelector('h1')
        .innerHTML = contador;

    if (contador % 10 === 0) {
        alert(`Contador igual a
    ${contador}`)
    }
}
```

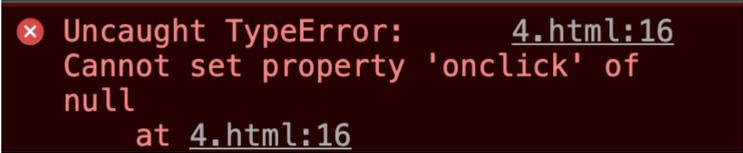
Agora, vamos ver algumas maneiras pelas quais podemos melhorar o design desta página.

Primeiro, assim como tentamos evitar o **estilo in-line** com CSS, queremos evitar o JavaScript in-line tanto quanto possível. Podemos fazer isso adicionando uma linha de script que altera o atributo **onclick** de um botão na página e remove esse atributo de dentro da tag button.

```
document.querySelector('button')
    .onclick = contar;
```

Uma coisa a observar sobre o que acabamos de fazer é que não estamos chamando a função **contar** adicionando os parênteses logo depois, mas apenas nomeando a função.

Isso especifica que desejamos chamar essa função somente quando o botão for clicado. Isso funciona porque, como o Python, o JavaScript oferece suporte à **programação funcional**, portanto, **as funções podem ser tratadas como valores**. A alteração acima por si só não é suficiente, pois ao inspecionarmos a página e olharmos para o console do nosso navegador, temos:

A screenshot of a browser's developer console showing a JavaScript error. The text is as follows:

```
✖ Uncaught TypeError: 4.html:16  
  Cannot set property 'onclick' of  
  null  
    at 4.html:16
```

Figura 6: Erro no console do navegador: não encontrou o elemento do tipo *button*.

Este erro surge porque quando o JavaScript procurou por um elemento usando **`document.querySelector('button')`**, ele não encontrou nada. Isso acontece porque leva um pouco de tempo para a página carregar e nosso código JavaScript foi executado antes que o botão fosse renderizado.

Para contornar essa questão, podemos especificar que o código seja executado somente depois que a página for carregada usando a função **`addEventListener`**.

Esta função recebe dois argumentos:

1. Um evento para ser rastreado (por exemplo: **`onclick`**);
2. Uma função a ser executada quando o evento for detectado (por exemplo: a função **`dizerOla`**).

Podemos usar essa função para executar o código apenas quando todo o conteúdo for carregado:

```
document.addEventListener(
```

```
    'DOMContentLoaded',  
    function() {  
        // Seu código aqui  
    }  
);
```

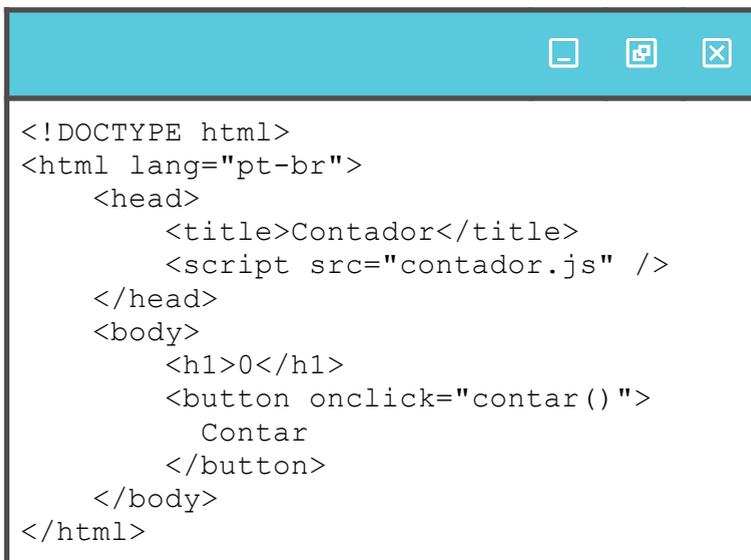
No exemplo acima, usamos uma **função anônima**, que é uma função que nunca recebe um nome. Juntando tudo isso, nosso JavaScript agora tem a seguinte aparência:

```
let contador = 0;  
  
function contar() {  
    contador++;  
    document.querySelector('h1')  
        .innerHTML = contador;  
  
    if (contador % 10 === 0) {  
        alert(`Contador igual a  
${contador}`)  
    }  
}  
  
document.addEventListener('DOMContentLoaded',  
    function() {  
        document.querySelector('button')  
            .onclick = contar;  
    });
```

Outra maneira de melhorar nosso design é mover nosso JavaScript para um arquivo separado. A maneira como fazemos isso é muito semelhante a como colocamos nosso CSS em um arquivo separado para estilização:

1. Escreva todo o seu código JavaScript em um arquivo separado que termine em **.js** (por exemplo: **contador.js**).
2. Adicione o atributo **src** à tag `<script>` que aponte para este novo arquivo.

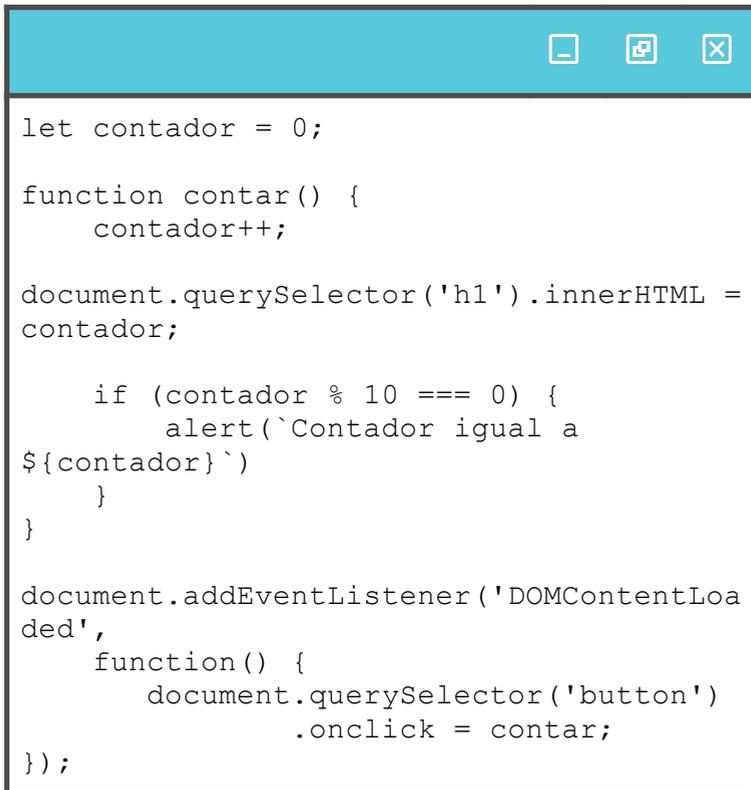
Para nossa página de contador, poderíamos ter um arquivo chamado **contador.html** assim:

A screenshot of a code editor window with a teal header bar containing three icons: a window icon, a refresh icon, and a close icon. The main area of the window displays HTML code for a counter page. The code includes a DOCTYPE declaration, an HTML lang attribute set to 'pt-br', a head section with a title 'Contador' and a script tag pointing to 'contador.js', and a body section with an h1 element containing '0' and a button with the text 'Contar' and an onclick event handler 'contar()'.

```
<!DOCTYPE html>
<html lang="pt-br">
  <head>
    <title>Contador</title>
    <script src="contador.js" />
  </head>
  <body>
    <h1>0</h1>
    <button onclick="contar()">
      Contar
    </button>
  </body>
</html>
```

Figura 7: Importação de arquivo JavaScript escrito isoladamente do arquivo HTML.

E um arquivo chamado **contador.js** que se parece com isso:

A screenshot of a code editor window with a teal header bar containing minimize, maximize, and close icons. The main area is white and contains the following JavaScript code:

```
let contador = 0;

function contar() {
    contador++;

    document.querySelector('h1').innerHTML =
    contador;

    if (contador % 10 === 0) {
        alert(`Contador igual a
    ${contador}`)
    }
}

document.addEventListener('DOMContentLoaded',
function() {
    document.querySelector('button')
        .onclick = contar;
});
```

Figura 8: Script contador.js.

Ter o JavaScript em um arquivo separado é útil por vários motivos, dentre os quais temos:

- **Apelo visual:** nossos arquivos HTML e JavaScript individuais tornam-se mais legíveis.
- **Acesso entre arquivos HTML:** agora podemos ter vários arquivos HTML que compartilham o mesmo JavaScript.
- **Colaboração:** agora podemos facilmente ter uma pessoa trabalhando no JavaScript enquanto outra trabalha no HTML.
- **Importação:** Podemos importar bibliotecas JavaScript que outras pessoas já escreveram. Por exemplo, o Bootstrap tem sua própria biblioteca JavaScript que você pode incluir para tornar seu site mais interativo.

Vamos começar com outro exemplo de página um pouco mais interativa. Criaremos agora uma página (ver **Figura 9**) onde um usuário pode digitar seu nome para obter uma saudação personalizada.

Algumas observações sobre a página da **Figura 9**:

- Usamos o atributo **autofocus** no campo de entrada nome para indicar que o cursor deve estar ativo dentro desse campo assim que a página for carregada.
- Usamos `#nome` dentro de `document.querySelector` para encontrar um

elemento com um **id** de valor igual a **nome**. Podemos usar todos os mesmos seletores nesta função como podemos em CSS.

- Usamos o atributo **value** de um campo de entrada para encontrar o que está digitado no momento.

```
<!DOCTYPE html>
<html lang="pt-br">
<head>
  <title>Olá</title>
  <script>
    document.addEventListener(
      'DOMContentLoaded',
      function() {
        document.querySelector('form')
          .onsubmit = function() {
            const nome =
document.querySelector('#nome')
              .value;
            alert(`Olá, ${nome}`);
          };
      });
  </script>
</head>
<body>
  <form>
    <input autofocus id="nome"
      placeholder="Nome"
      type="text">
    <input type="submit">
```

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

Figura 9: Exemplo com saudação personalizada.

- Podemos fazer mais do que apenas adicionar HTML à nossa página usando JavaScript. Também podemos alterar o estilo de uma página.

Na página HTML da **Figura 10**, usamos botões para mudar a cor do nosso título.



```
<!DOCTYPE html>
<html lang="pt-br">
<head>
  <title>Cores</title>
  <script>
    document.addEventListener(
      'DOMContentLoaded',
      function() {
        document
          .querySelectorAll('button')
          .forEach(function(button) {
            button.onclick =
              function() {
                document
                  .querySelector("#ola")
                    .style.color =
```

```
        button.dataset.color;
    }
    });
});
</script>
</head>
<body>
  <h1 id="ola">Olá</h1>
  <button data-color="red">
    Vermelho
  </button>
  <button data-color="blue">
    Azul
  </button>
  <button data-color="green">
    Verde
  </button>
</body>
</html>
```

Figura 10: Exemplo com botões que alteram a cor de acordo com a opção selecionada.

Algumas observações sobre a página acima:

- Mudamos o estilo de um elemento usando o atributo **style.ALGUMACOISA**.
- Usamos o atributo **data-ALGUMACOISA** para atribuir dados a um elemento HTML. Posteriormente, podemos acessar esses dados em JavaScript usando a propriedade **dataset** do elemento.

- Usamos a função **querySelectorAll** para obter uma **lista** (semelhante a uma lista Python ou um array JavaScript) com todos os elementos que correspondem à consulta.
- A função **forEach** em JavaScript aceita outra função como argumento de entrada e aplica essa função a cada elemento de uma **lista** ou **array**.

Console JavaScript

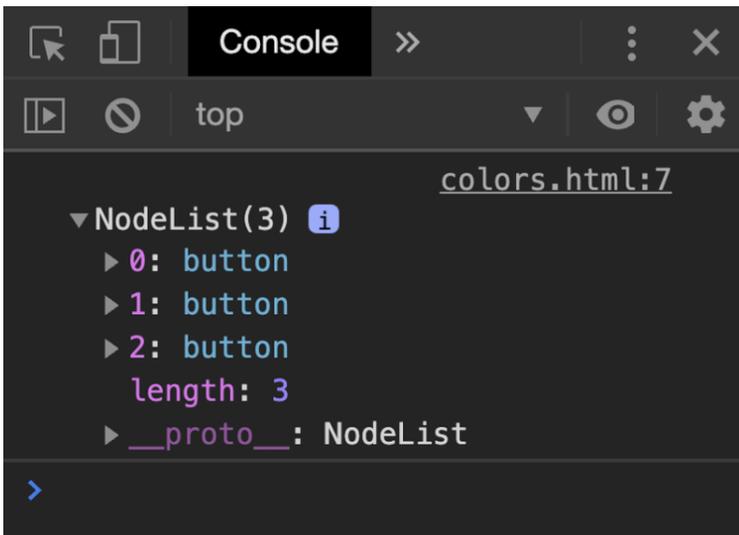


Figura 11: Uso do console para depurações (*debug*).

O console é uma ferramenta útil para testar pequenos pedaços do nosso código e depurar. Você pode escrever e executar o código JavaScript no console, que pode ser encontrado inspecionando o elemento

em seu navegador e clicando na aba **Console** (o processo exato pode mudar de navegador para navegador).

Uma ferramenta útil para depuração é a impressão no console, que você pode fazer usando a função **console.log**. Por exemplo, na página **cores.html** vista anteriormente, podemos adicionar a seguinte linha:

```
console.log(  
    document.querySelectorAll('button')  
);
```

O que nos retorna no console o objeto da **Figura 11**.

Arrow Functions

Além da tradicional notação de função que já vimos, o JavaScript agora nos dá a capacidade de usar **arrow functions (funções de seta)** onde temos uma entrada (ou apenas um par de parênteses caso não haja entrada) seguida por => e de algum código a ser executado.

Por exemplo, podemos alterar nossa função contar vista anteriormente para usar uma função de seta anônima:

```
const contar = () => {  
    contador++;
```

```
document.querySelector('h1')
    .innerHTML = contador;

if (contador % 10 === 0) {
    alert(`Contador igual a
    ${contador}`)
}
}
```

Ou ainda podemos alterar nosso script **cores.html** para usar uma função de seta anônima:



```
<!DOCTYPE html>
<html lang="pt-br">
<head>
  <title>Cores</title>
  <script>
    document.addEventListener(
      'DOMContentLoaded',
      () => {
        document
          .querySelectorAll('button')
          .forEach((button) => {
            button.onclick = () => {
              document

.querySelector("#ola")
                .style.color =
                button.dataset.color;
            }
          })
      }
    )
  </script>
</head>
<body>
  <div>
    <h1>Cores</h1>
    <button>Clique aqui</button>
  </div>
</body>
</html>
```

```
        });  
    });  
    </script>  
</head>  
<body>  
    <h1 id="ola">Olá</h1>  
    <button data-color="red">  
        Vermelho  
    </button>  
    <button data-color="blue">  
        Azul  
    </button>  
    <button data-color="green">  
        Verde  
    </button>  
</body>  
</html>
```

Figura 12: Exemplo de uso de *arrows functions*.

Para ter uma ideia sobre alguns outros eventos que podemos usar, vamos ver como podemos implementar nosso seletor de cores usando um menu suspenso em vez de três botões separados.

Podemos detectar mudanças em um elemento do tipo **select** usando o atributo **onchange**. Em JavaScript, esta é uma palavra-chave que muda com base no contexto em que é usada. No caso de um manipulador de eventos, **this** refere-se ao objeto que acionou o evento (ver **Figura 13**).

Existem muitos outros eventos que podemos detectar em JavaScript, incluindo os mais comuns abaixo:

- onclick
- onmouseover
- onkeydown
- onkeyup
- onload
- onblur
- entre outros...

A screenshot of a code editor window with a teal header bar containing three icons: a square, a square with a plus sign, and a square with an X. The main area is white and contains the following code:

```
<!DOCTYPE html>
<html lang="pt-br">
<head>
  <title>Cores</title>
  <script>
    document.addEventListener(
      'DOMContentLoaded',
      () => {
        document
          .querySelector('select')
          .onchange = () => {
            document
              .querySelector("#ola")
              .style.color =
                this.color;
          }
      });
  </script>
</head>
```

```
<body>
  <h1 id="ola">Olá</h1>
  <select>
    <option value="black">
      Preto
    </option>
    <option value="red">
      Vermelho
    </option>
    <option value="blue">
      Azul
    </option>
    <option value="green">
      Verde
    </option>
  </select>
</body>
</html>
```

Figura 13: Exemplo de uso do atributo *onchange*.

Lista de Tarefas

```
<!DOCTYPE html>
<html lang="pt-br">
  <head>
    <title>Tarefas</title>
    <script src="tarefas.js" />
  </head>
  <body>
    <h1>Tarefas</h1>
```

```
<ul id="tarefas"></ul>
<form>
  <input id="tarefa"
    placeholder="Nova Tarefa"
    type="text">
  <input id="submit"
    type="submit">
</form>
</body>
</html>
```

Figura 14: HTML do exemplo Lista de Tarefas.

Para reunir algumas das coisas que vimos até aqui, vamos criar uma lista de tarefas inteiramente em JavaScript, começando pela página HTML. Observe abaixo como deixamos espaço para uma lista não ordenada, mas ainda não adicionamos nada a ela. Note também que adicionamos um link para **tarefas.js** onde escreveremos nosso JavaScript.

Na **Figura 14** está o nosso script **tarefas.js**. Algumas observações sobre o que você verá abaixo:

- Este código é um pouco diferente do que vimos até aqui. Neste caso, apenas consultamos nosso botão de envio e campo de tarefa de entrada uma vez no início e armazenamos esses dois valores nas variáveis **submit** e **novaTarefa**.
- Podemos ativar/desativar um botão definindo seu atributo **disabled** como **true/false**.

- Em JavaScript, usamos o atributo **length** para encontrar o comprimento de objetos como **strings** e **arrays**.
- No final do script, adicionamos a linha **return false**. Isso evita o envio padrão do formulário, que envolve recarregar a página atual ou redirecionar para uma nova.
- Em JavaScript, podemos criar elementos HTML usando a função **createElement**. Podemos então adicionar esses elementos ao DOM usando a função **append**.

```
// Espera a página carregar
document.addEventListener(
  'DOMContentLoaded',
  function() {
    /* Seleção dos botões
     * de 'submit' e 'tarefa' */
    const submit =

document.querySelector('#submit');
    const novaTarefa =

document.querySelector('#tarefa');

    /* Desabilita o botão
     * 'submit' por padrão */
    submit.disabled = true;

    /* Observa um valor ser
```

```
    * digitado no campo de entrada */
novaTarefa.onkeyup = () => {
    if (novaTarefa.value.length >
0){
        submit.disabled = false;
    }
    else {
        submit.disabled = true;
    }
}
// Rastreia o envio do formulário
document.querySelector('form')
    .onsubmit = () => {
    /* Obtém a tarefa enviada
    * pelo usuário */
    const tarefa = novaTarefa.value;

    /* Criação de um item
    * de lista com a tarefa */
    const li =
        document.createElement('li');
    li.innerHTML = tarefa;

    /* Adição do item na
    * lista não-ordenada */
    document
        .querySelector('#tarefas')
        .append(li);

    // Limpa o campo de entrada
    novaTarefa.value = '';

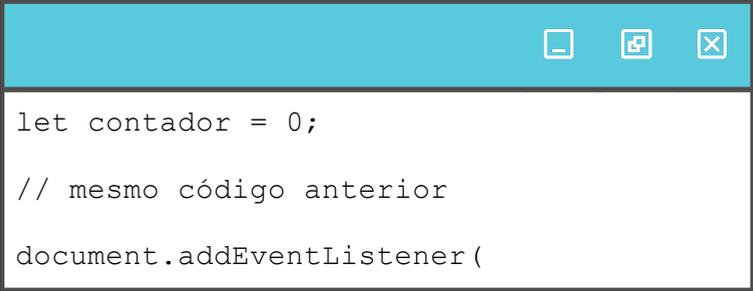
    /* Desabilita o botão
    * de envio novamente */
    submit.disabled = true;
```

```
        /* Evita o envio padrão
         * do formulário */
        return false;
    }
});
```

Figura 15: Script do exemplo Lista de Tarefas.

Intervalos

Além de especificar quais funções são executadas quando um evento é acionado, também podemos definir funções para serem executadas após um determinado período de tempo. Por exemplo, vamos retornar ao exemplo da página de contador e adicionar um intervalo para que, mesmo que o usuário não clique em nada, o contador seja incrementado a cada segundo. Para isso, utilizamos a função **setInterval**, que recebe como argumento de entrada uma função a ser executada e um tempo (em milissegundos) entre as execuções da função.



```
let contador = 0;

// mesmo código anterior

document.addEventListener(
```

```
'DOMContentLoaded',  
function() {  
    document.querySelector('button')  
        .onclick = contar;  
  
    setInterval(count, 1000);  
});
```

Figura 16: Exemplo de uso da função *setInterval*.

Armazenamento Local

Uma coisa a notar sobre todos os nossos sites até agora é que toda vez que recarregamos a página, todas as nossas informações são perdidas. A cor do título volta a ser preta, o contador volta a 0 (zero) e todas as tarefas são apagadas. Às vezes é isso que pretendemos, mas outras vezes queremos poder armazenar informações que podemos usar quando um usuário retornar ao site.

Uma maneira de fazer isso é usando armazenamento local, isto é, armazenando informações no navegador do usuário que podemos acessar posteriormente. Essas informações são armazenadas como um conjunto de **pares chave-valor**. Para usar o armazenamento local, empregaremos duas funções principais:

- **localStorage.getItem(key):** esta função procura uma entrada no armazenamento local com uma

determinada **chave (key)** e retorna o valor associado a essa chave.

- **localStorage.setItem(key, value):** esta função define uma entrada no armazenamento local, associando a **chave (key)** a um novo **valor (value)**.

Vejamos como podemos usar essas novas funções para atualizar nosso contador.

```
/* Verifica se há algum
 * valor no armazenamento local */
if (!localStorage.getItem('contador')) {
  /* Caso não haja, inicializamos
   * o contador com 0 (zero) */
  localStorage.setItem('contador', 0);
}

function contar() {
  /* Obtém o valor do contador
   * armazenado localmente */
  let contador =
    localStorage.getItem('contador');

  // Atualiza contador
  contador++;
  document.querySelector('h1')
    .innerHTML = contador;

  // Armazena o contador localmente
  localStorage.setItem(
```

```
        'contador', contador);
    }

    document.addEventListener(
        'DOMContentLoaded',
        function() {
            /* Atualiza o valor do elemento
             * 'h1' com o contador */
            document.querySelector('h1')
                .innerHTML =

localStorage.getItem('contador');

document.querySelector('button')
    .onclick = contar;
});
```

Figura 17: Exemplo de uso do armazenamento local.

APIs

Vimos ao longo deste capítulo que um objeto JavaScript nos permite armazenar pares chave-valor. Por exemplo, eu poderia criar um objeto JavaScript representando Harry Potter:

```
let pessoa = {
    nome: 'Harry',
    sobrenome: 'Potter'
};
```

Uma maneira pela qual os objetos JavaScript são realmente úteis é na transferência de dados de um site para outro, principalmente ao usar APIs.

Uma API, ou *Application Programming Interface*, é uma forma de comunicação estruturada entre duas aplicações diferentes. Por exemplo, podemos querer que nosso aplicativo obtenha informações do Google Maps, Amazon ou algum serviço meteorológico.

Podemos fazer isso fazendo chamadas para a API de um serviço, que retornará dados estruturados para nós, geralmente no **formato JSON** (*JavaScript Object Notation*). Por exemplo, um voo em formato JSON pode ter esta aparência:

```
{
  "origem": "Recife",
  "destino": "São Paulo",
  "duracao": 180
}
```

Os valores dentro de um JSON não precisam ser apenas strings e números como no exemplo acima. Também podemos armazenar listas, ou mesmo outros objetos JavaScript:

```
{
  "origem": {
```

```
        "cidade": "Recife",
        "codigo": "REC"
    },
    "destino": {
        "cidade": "São Paulo",
        "codigo": "SPO"
    },
    "duracao": 180
}
```

Para mostrar como podemos usar APIs em nossas aplicações, vamos trabalhar na construção de uma aplicação onde passamos o CEP do usuário e obtemos o seu endereço completo (rua, número, bairro etc), utilizando a **API pública da ViaCEP**. Ao visitar o site, você verá a documentação da API, que geralmente é um bom ponto de partida quando você deseja usar uma API.

Podemos testar esta API visitando a URL:

```
https://viacep.com.br/ws/01001000/json/
```

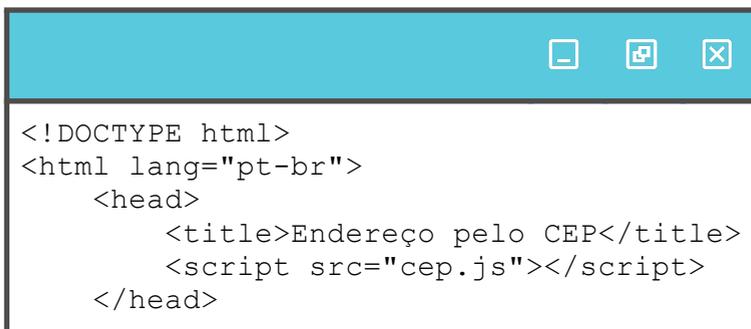
Onde 01001-000 é um CEP de exemplo que retorna o endereço:

```
{
  "cep": "01001-000",
  "logradouro": "Praça da Sé",
  "complemento": "lado ímpar",
  "bairro": "Sé",
```

```
"localidade": "São Paulo",
"uf": "SP",
"ibge": "3550308",
"gia": "1004",
"ddd": "11",
"siafi": "7107"
}
```

Vamos dar uma olhada em como implementar essa API em um aplicativo criando um novo arquivo HTML chamado `cep.html` e vinculá-lo a um arquivo JavaScript, mas deixando o **corpo (body)** vazio (ver Figura 18).

Agora, usaremos algo chamado **AJAX**, ou *Asynchronous JavaScript And XML*, que nos permite acessar informações de páginas externas mesmo após o carregamento da nossa página. Para fazer isso, usaremos a função **fetch** que nos permitirá enviar uma solicitação HTTP. A função **fetch** retorna um objeto do tipo **Promise**.



```
<!DOCTYPE html>
<html lang="pt-br">
  <head>
    <title>Endereço pelo CEP</title>
    <script src="cep.js"></script>
  </head>
```

```
<body>
    <!-- corpo inicialmente vazio
-->
</body>
</html>
```

Figura 18: Página HTML com corpo (*body*) vazio.

Não vamos falar sobre os detalhes do que é um **Promise** agora, mas podemos pensar nele como um valor que aparecerá em algum momento, mas não necessariamente imediatamente.

Lidamos com objetos do tipo **Promise** definindo seu comportamento através do atributo **.then** que descreve o que deve ser feito quando recebemos a resposta da requisição. O trecho de código da Figura 19 imprime a nossa resposta no console.

```
document.addEventListener(
  'DOMContentLoaded',
  function() {
    // Envia uma requisição GET pela URL
    fetch('https://viacep.com.br/ws/01001000
/json/')
    // Formata a resposta no padrão JSON
    .then(response => response.json())
    .then(data => {
```

```
        // Imprime no console
        console.log(data);
    }
    );
});
```

Figura 19: Chamada à API externa com a função *fetch*.

Um ponto importante sobre o código acima é que o argumento de **.then** é sempre uma função. Embora pareça que estamos criando as variáveis **response** e **data**, essas variáveis são apenas os parâmetros de duas funções anônimas.

Ao invés de simplesmente imprimir esses dados no console (ver **Figura 20**), podemos usar o JavaScript para exibir uma mensagem na tela, como no código da **Figura 21**.

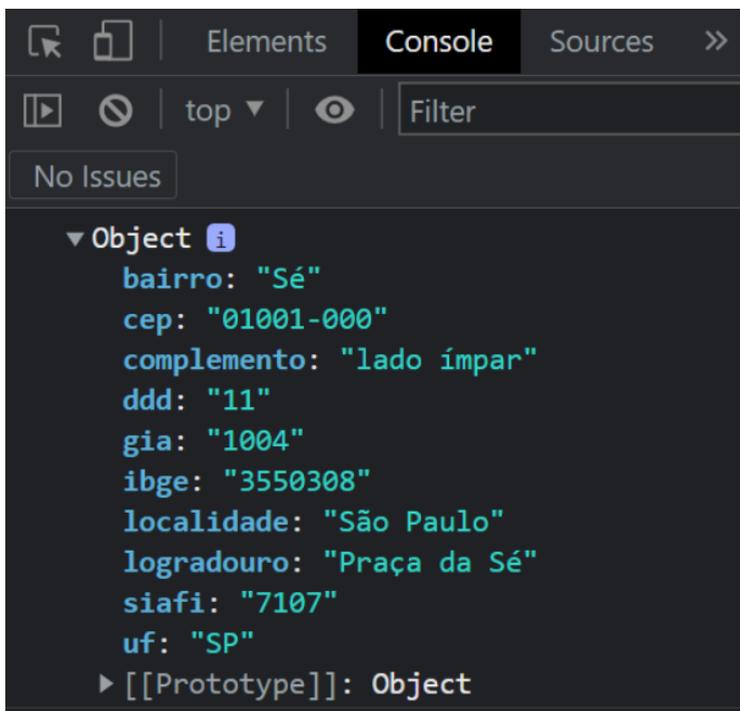
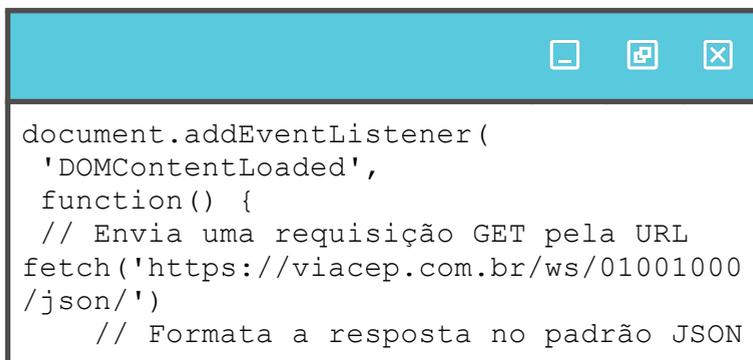


Figura 20: Resposta da chamada à API no console.

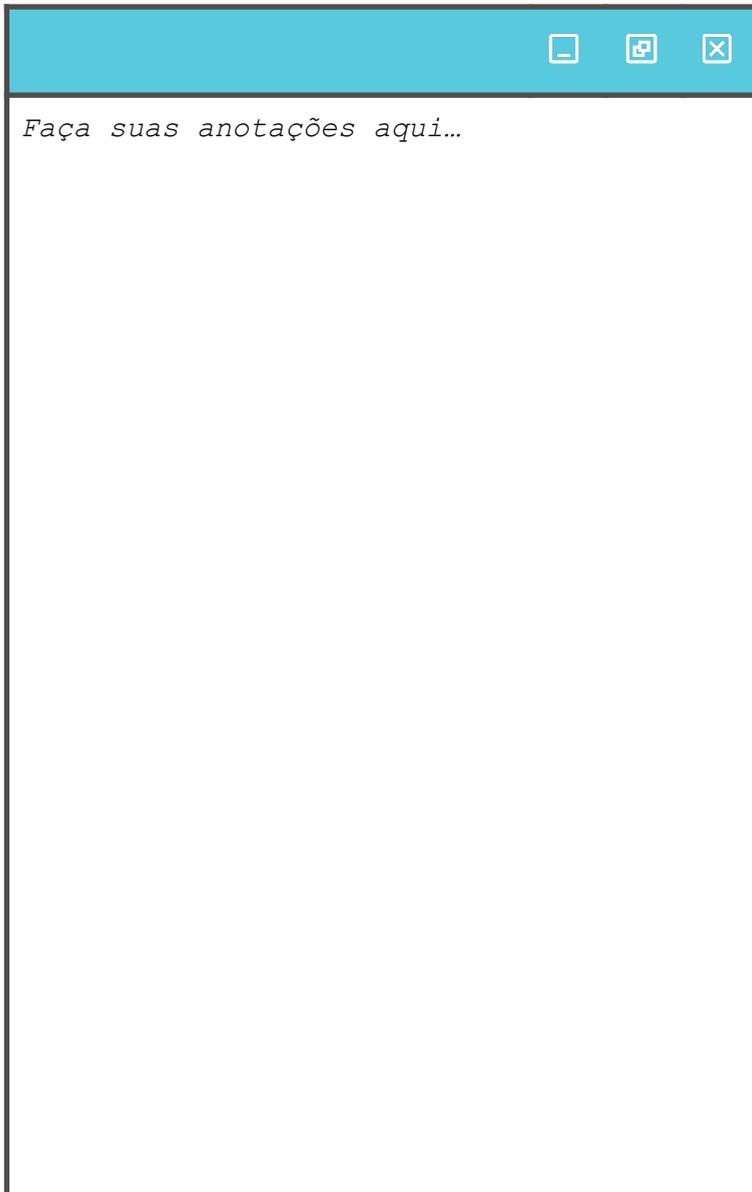


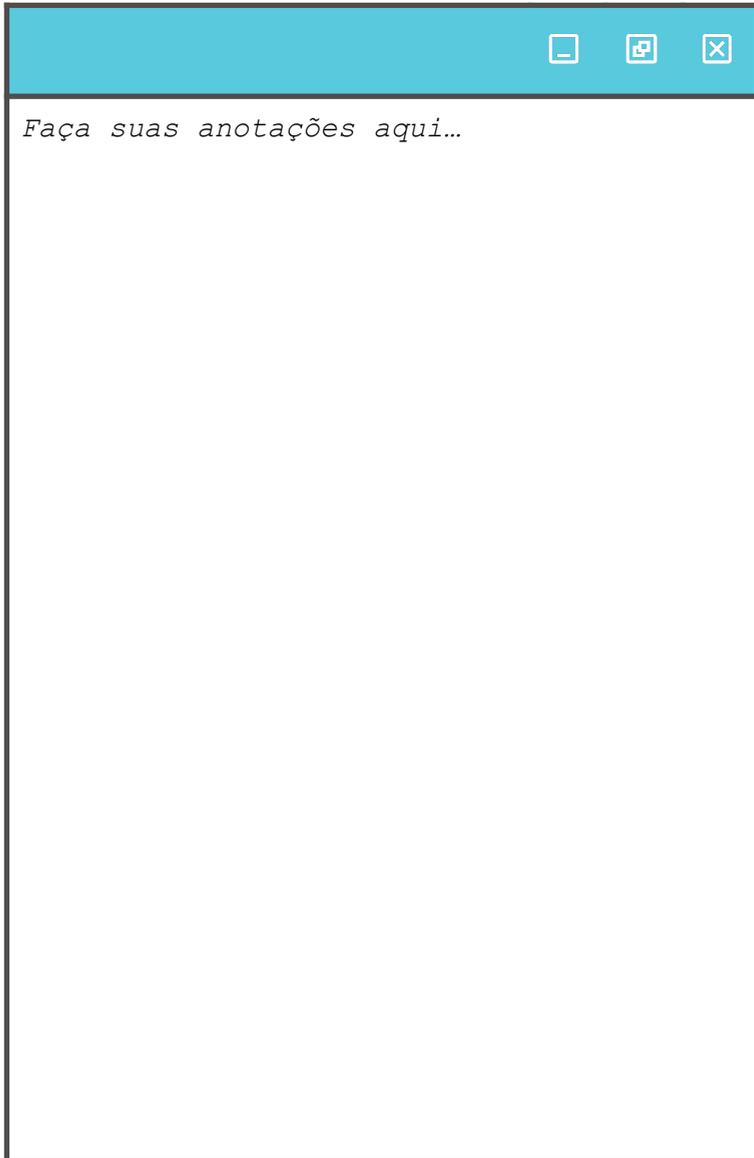
```
.then(response => response.json())
.then(data => {
  // Exibe mensagem na tela
  document.querySelector('body')
    .innerHTML =
    `Rua  ${data.logradouro} -
    ${data.localidade},
    ${data.uf}`;
  }
  );
});
```

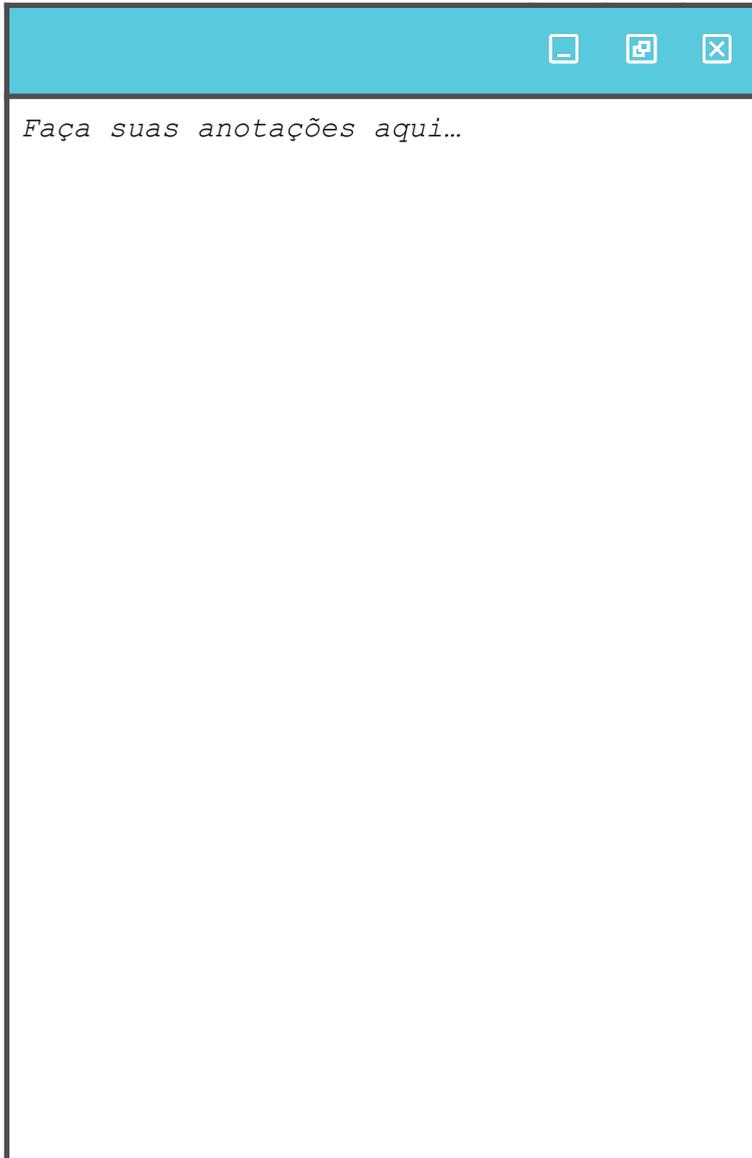
Figura 21: Resposta da chamada à API no console.



Figura 22: Resposta da chamada à API na tela do *browser*.







Instalação e configuração

Para começar a utilizar o React.js, é necessário ter o **Node.js** e o **NPM (Node Package Manager)** instalados em sua máquina. O Node.js é uma plataforma para desenvolvimento em JavaScript do lado do servidor, e o NPM é um gerenciador de pacotes que permite instalar e gerenciar as dependências de um projeto.



Para instalar o Node.js e o NPM, basta acessar o site oficial e fazer o download da versão mais recente: <https://nodejs.org/>.

Para criar um projeto em React, a versão do node deverá ser **igual ou superior a versão 14**. Contudo, para este curso recomendamos a **versão 16.18.0**.

Podemos verificar a versão do node instalada executando a seguinte linha de comando no terminal:

```
node --version
```

Após a instalação do Node.js e do NPM, é possível criar um novo projeto React utilizando o comando no terminal:

```
npx create-react-app nome-do-projeto
```

Este comando irá baixar as dependências e configurar um projeto React básico. onde **npx** é uma ferramenta executora de pacotes que vem com o **npm 5.2+** e **meu-projeto-react** é o nome do projeto.

Uma aplicação React não lida com lógica de backend ou bancos de dados; ela apenas cria um pipeline de compilação de frontend, para que você possa usá-lo com qualquer backend que desejar (Java com Spring Boot, por exemplo).

Para inicializar nossa aplicação via terminal, devemos entrar na pasta recém-criada **nome-do-projeto**, através do comando **cd** (*change directory*) e em seguida, executamos o comando **npm start**.

```
cd nome-do-projeto  
npm start
```

O **npm** é o gerenciador de pacotes padrão para o desenvolvimento de projetos em JavaScript, que permite que você aproveite um vasto ecossistema de pacotes de terceiros e os instale ou atualize facilmente.

Ao iniciar a aplicação, ela deverá ser a página a ser aberta no navegador ou através da URL <http://localhost:3000>:

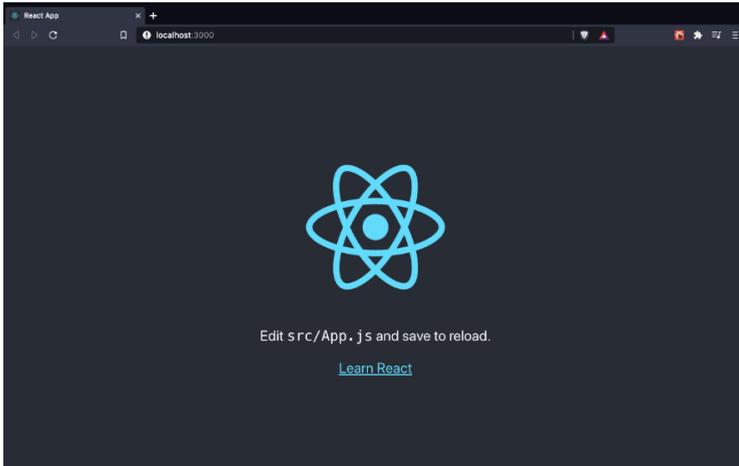
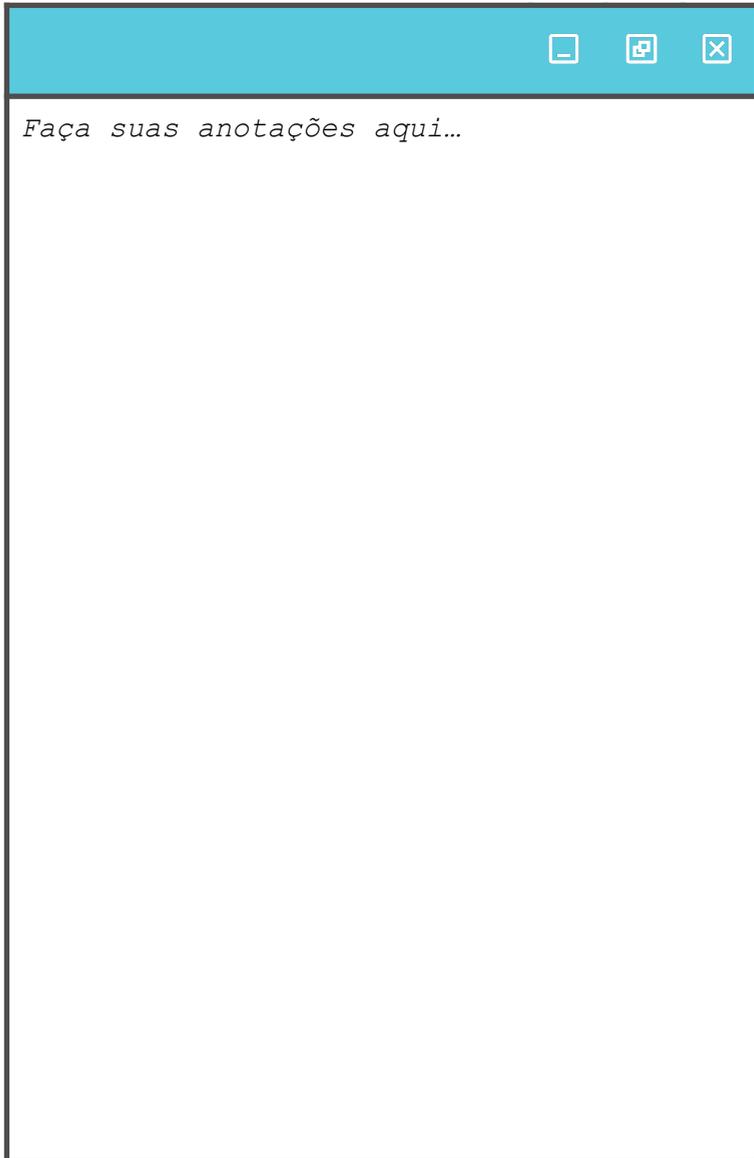


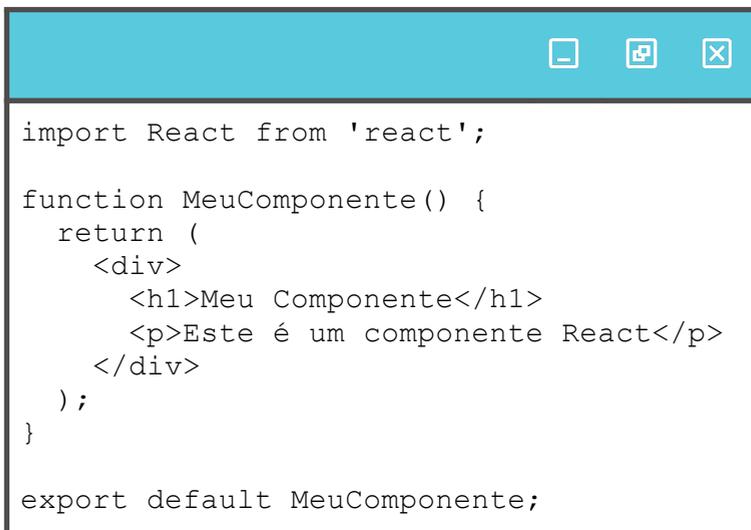
Figura 23: Tela inicial de um projeto React recém-criado.





Estrutura de um Componente

Um componente React é basicamente uma função JavaScript que retorna um elemento React. O elemento React é uma representação virtual da interface do usuário, e é responsável por renderizar o HTML final no navegador. A estrutura básica de um componente React é mostrado na **Figura 24**.

A screenshot of a code editor window with a light blue header bar containing three icons: a square, a square with a plus sign, and a square with an X. The main area is white and contains the following JavaScript code:

```
import React from 'react';

function MeuComponente() {
  return (
    <div>
      <h1>Meu Componente</h1>
      <p>Este é um componente React</p>
    </div>
  );
}

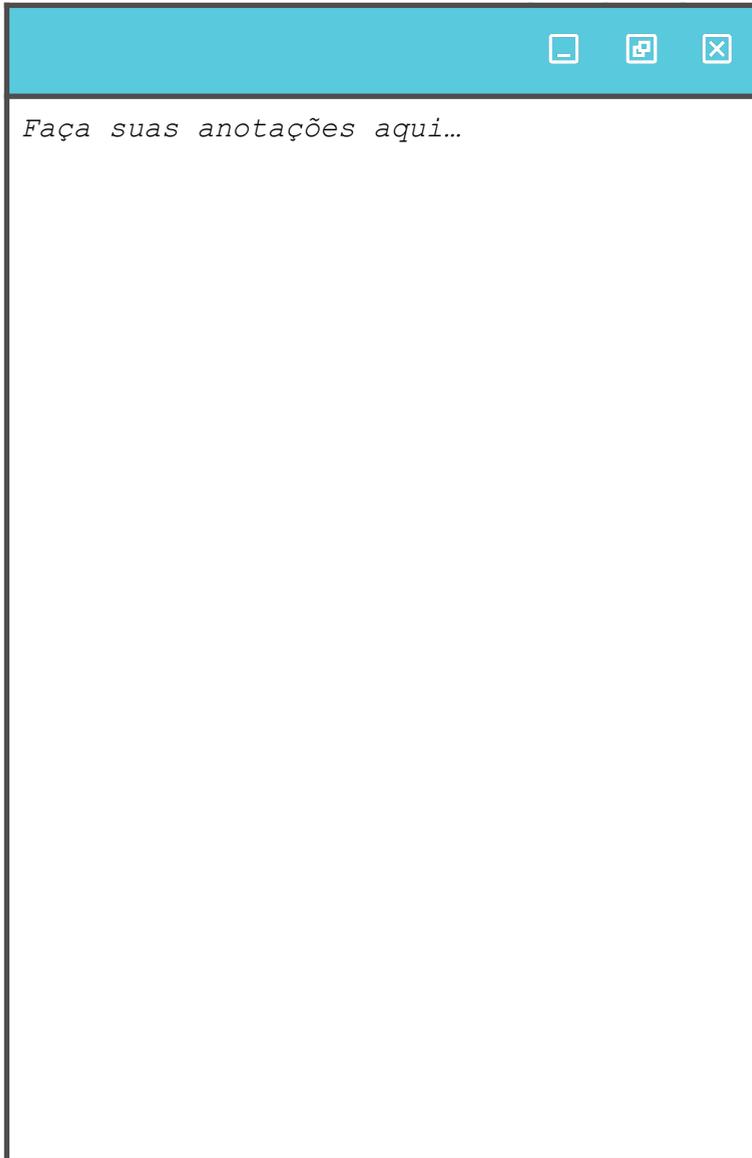
export default MeuComponente;
```

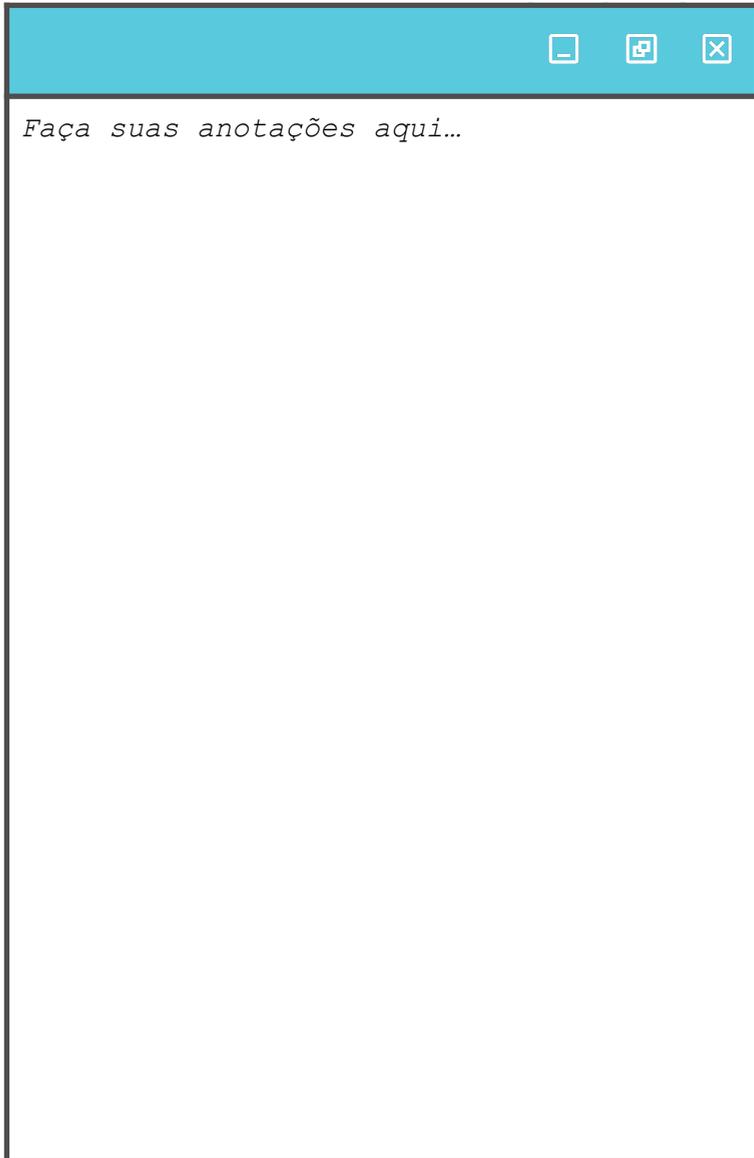
Figura 24: Estrutura básica de um componente React.

Na primeira linha do código, é importada a biblioteca React. Em seguida, é definida a função **MeuComponente**, que retorna um elemento React. O elemento é definido utilizando a sintaxe de HTML

dentro de parênteses. É possível utilizar JavaScript dentro do HTML utilizando chaves.

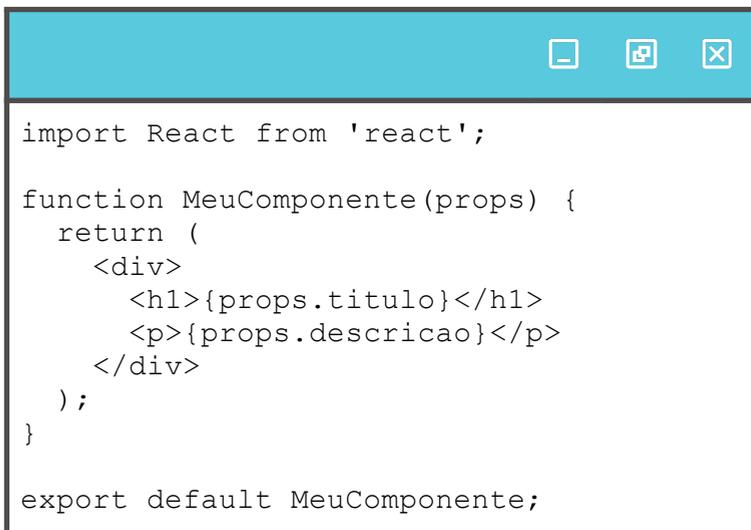
Por fim, o componente é exportado utilizando a palavra-chave **export default**. Isso permite que o componente seja importado e utilizado em outras partes do código.





Propriedades e Estados

Os componentes React podem receber propriedades (**props**) como parâmetro. As props são utilizadas para passar informações de um componente pai para um componente filho. As props são **somente leitura** e não devem ser modificadas pelo componente filho.



```
import React from 'react';

function MeuComponente(props) {
  return (
    <div>
      <h1>{props.titulo}</h1>
      <p>{props.descricao}</p>
    </div>
  );
}

export default MeuComponente;
```

Figura 25: Uso de propriedades em um componente React.

No exemplo da **Figura 25**, o componente **MeuComponente** recebe duas propriedades: **titulo** e **descricao**. Estas propriedades são utilizadas dentro do elemento React utilizando chaves.

Além das props, os componentes React podem utilizar **estados (states)** para gerenciar informações dinâmicas. O estado é uma variável que é utilizada dentro do componente e **pode ser modificada** pelo próprio componente.

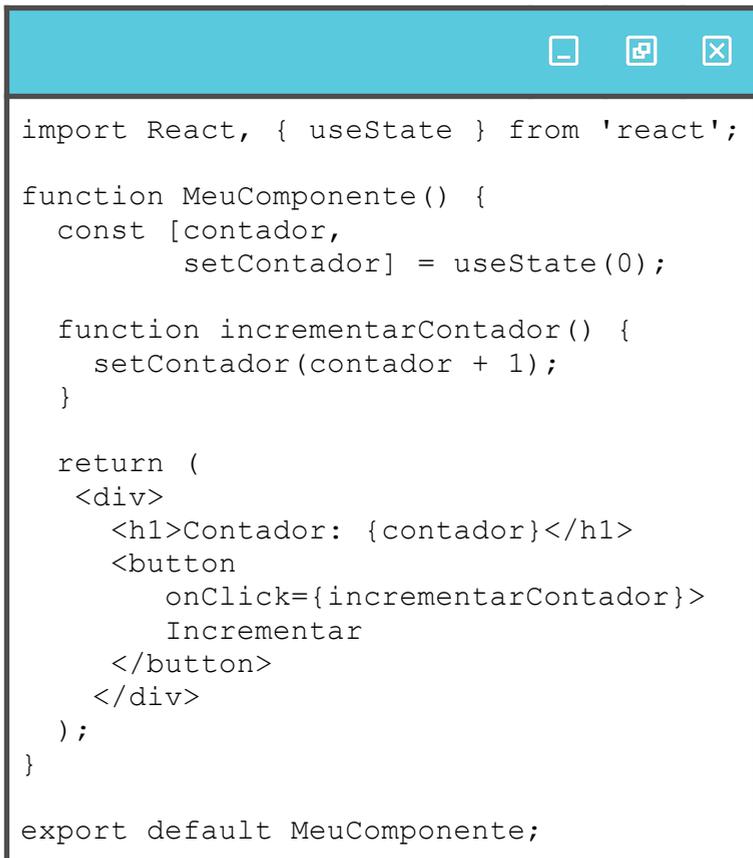
Para utilizar o estado em um componente React, é necessário utilizar o hook **useState** da biblioteca React. Este **hook** retorna um array com duas posições: a primeira posição contém o **valor atual do estado**, e a segunda posição contém uma **função para atualizar o estado**.

No exemplo da **Figura 26**, é utilizado o hook **useState** para criar a variável **contador** e a função **setContador**. A variável contador é inicializada com o valor 0. A função setContador é utilizada dentro da função **incrementarContador**, que é chamada ao clicar no botão. Esta função atualiza o valor do estado contador para o valor atual mais 1.

O **useEffect** é um hook do React que permite executar **efeitos colaterais** em componentes funcionais, que são ações que **afetam o ambiente fora do escopo do componente**, como chamadas à API, atualizações no localStorage ou manipulação do DOM.

O **useEffect** é executado **após cada renderização do componente**. Ele recebe dois parâmetros: uma **função** que contém o efeito colateral a ser executado

e uma **lista de dependências opcional**. A lista de dependências permite que o efeito seja executado apenas quando determinadas propriedades ou estados do componente forem alterados.

A screenshot of a code editor window with a light blue header containing three icons: a minus sign, a plus sign, and an 'X' sign. The main area is white and contains the following JavaScript code:

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

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

  function incrementarContador() {
    setContador(contador + 1);
  }

  return (
    <div>
      <h1>Contador: {contador}</h1>
      <button
        onClick={incrementarContador}>
        Incrementar
      </button>
    </div>
  );
}

export default MeuComponente;
```

Figura 26: Uso de estados em um componente React.

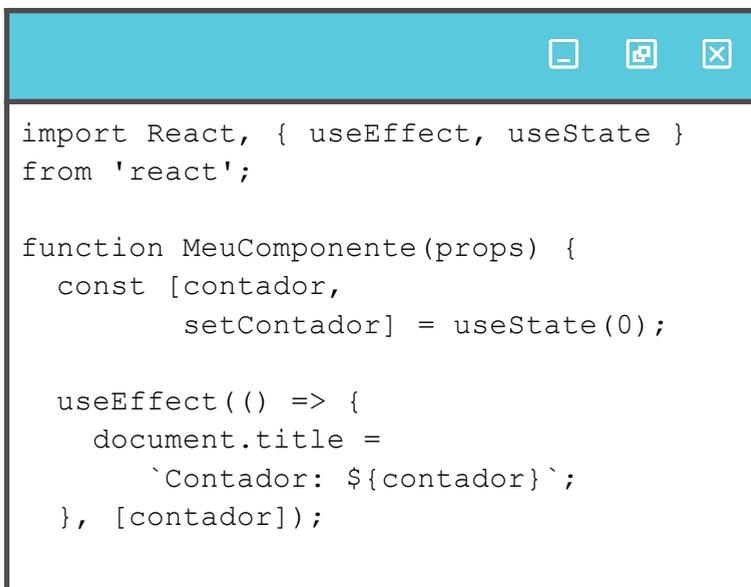
```
import React, { useEffect, useState }
from 'react';

function MeuComponente() {
  const [usuarios,
        setUsuarios] = useState([]);
  useEffect(() => {
    fetch('https://api.meusite.com/usuarios'
  )
    .then(res => res.json())
    .then(data => setUsuarios(data));
  }, []);

  return (
    <div>
      <h1>Lista de Usuários</h1>
      <ul> {usuarios.map(usuario => (
        <li key={usuario.id}>
          {usuario.nome}
        </li>
      ))}
      </ul>
    </div>
  );
}
export default MeuComponente;
```

Figura 27: Uso do *useEffect* em um componente React. No exemplo da **Figura 27**, o componente **MeuComponente** utiliza o **useEffect** para buscar a lista de usuários da API e atualizar o estado **usuarios**. A função passada como primeiro parâmetro ao **useEffect** é executada após a montagem do componente. O segundo parâmetro é uma **lista vazia**, indicando que o efeito deve ser **executado apenas na montagem do componente**.

É possível utilizar outras dependências na lista do segundo parâmetro do **useEffect**. Se alguma dependência for alterada, o efeito será executado novamente. Por exemplo:

A screenshot of a code editor window with a light blue header bar containing three window control icons (minimize, maximize, close). The main area is white and contains the following JavaScript code:

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

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

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

```
function incrementarContador() {
  setContador(contador + 1);
}

return (
  <div>
    <h1>Contador: {contador}</h1>
    <button
      onClick={incrementarContador}>
      Incrementar
    </button>
  </div>
);
}

export default MeuComponente;
```

Figura 28: Definição de dependências no *useEffect*.

No exemplo da **Figura 28**, o componente **MeuComponente** utiliza o **useEffect** para atualizar o título da página com o valor do contador. A função passada como primeiro parâmetro ao **useEffect** é executada após cada atualização do contador. O segundo parâmetro é uma lista contendo o estado contador, indicando que o efeito deve ser executado sempre que o contador for alterado.

O **useEffect** deve ser utilizado com cautela, pois a execução de efeitos colaterais pode ter impactos

negativos na performance da aplicação. Por isso, é recomendável utilizar o **useEffect** apenas para efeitos colaterais que **afetam o ambiente fora do escopo do componente**, e **não para atualizações de estado do componente**.

Além disso, é importante sempre **limpar os efeitos colaterais quando o componente é desmontado**. Para isso, é possível retornar uma função de limpeza na função passada como primeiro parâmetro ao **useEffect**. A função de limpeza é executada quando o componente é desmontado ou quando o efeito é limpo.

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

  return () => {
    document.title = 'Minha Aplicação';
  };
}, [contador]);
```

No exemplo acima, a função de limpeza do **useEffect** é responsável por redefinir o título da página para o valor original da aplicação. Ela é executada quando o componente é desmontado ou quando o efeito é limpo. Isso garante que o título não ficará com o valor do contador após o componente ser desmontado.

Outra recomendação é utilizar as dependências do **useEffect** de forma adequada. Se nenhuma dependência for passada, o efeito será executado a cada renderização do componente. Se todas as dependências forem passadas, o efeito será executado apenas quando todas as dependências forem alteradas. Se algumas dependências forem passadas, o efeito será executado quando as dependências selecionadas forem alteradas.

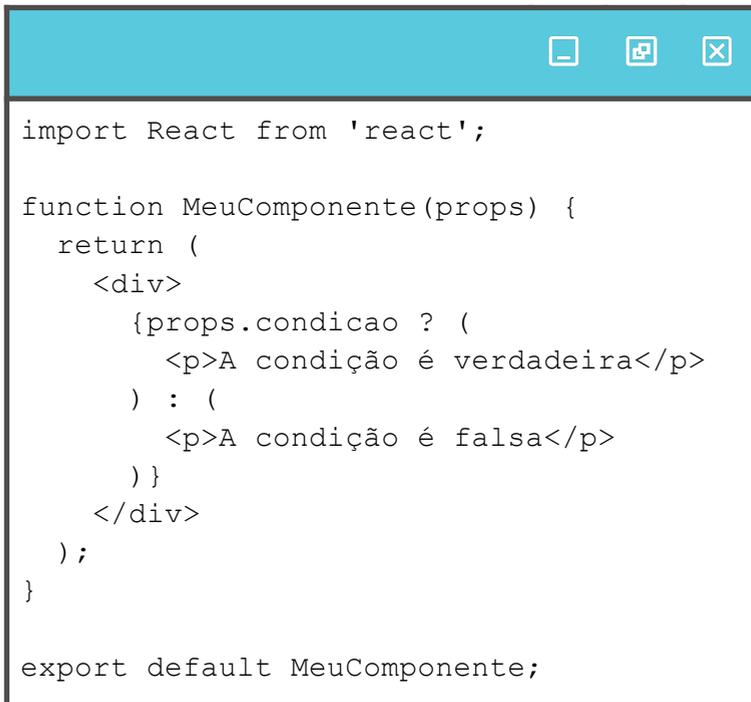
Por fim, é importante lembrar que o **useEffect** é uma ferramenta poderosa, mas não é a única forma de se trabalhar com efeitos colaterais no React. Em alguns casos, pode ser mais adequado utilizar outras soluções, como o **useLayoutEffect** ou a ContextAPI (vista mais adiante). Por isso, é importante avaliar o caso de uso e escolher a solução mais adequada para cada situação.

Renderização condicional

Em algumas situações, pode ser necessário renderizar elementos diferentes com base em alguma condição. Para isso, é possível utilizar a renderização condicional.

No exemplo da **Figura 29**, o componente **MeuComponente** recebe uma propriedade **condicao**. Se a condição for verdadeira, é renderizado o

parágrafo "A condição é verdadeira". Caso contrário, é renderizado o parágrafo "A condição é falsa".

A screenshot of a code editor window with a teal header bar containing three icons: a square, a square with a plus sign, and a square with an X. The main area is white and contains the following code:

```
import React from 'react';

function MeuComponente(props) {
  return (
    <div>
      {props.condicao ? (
        <p>A condição é verdadeira</p>
      ) : (
        <p>A condição é falsa</p>
      )}
    </div>
  );
}

export default MeuComponente;
```

Figura 29: Exemplo de renderização condicional

Ciclo de vida de um componente

Os componentes React, ao serem declarados como **classes** (em desuso) e não funções, possuem um ciclo de vida que consiste em diversas fases, desde a inicialização até a remoção. Durante este ciclo de vida,

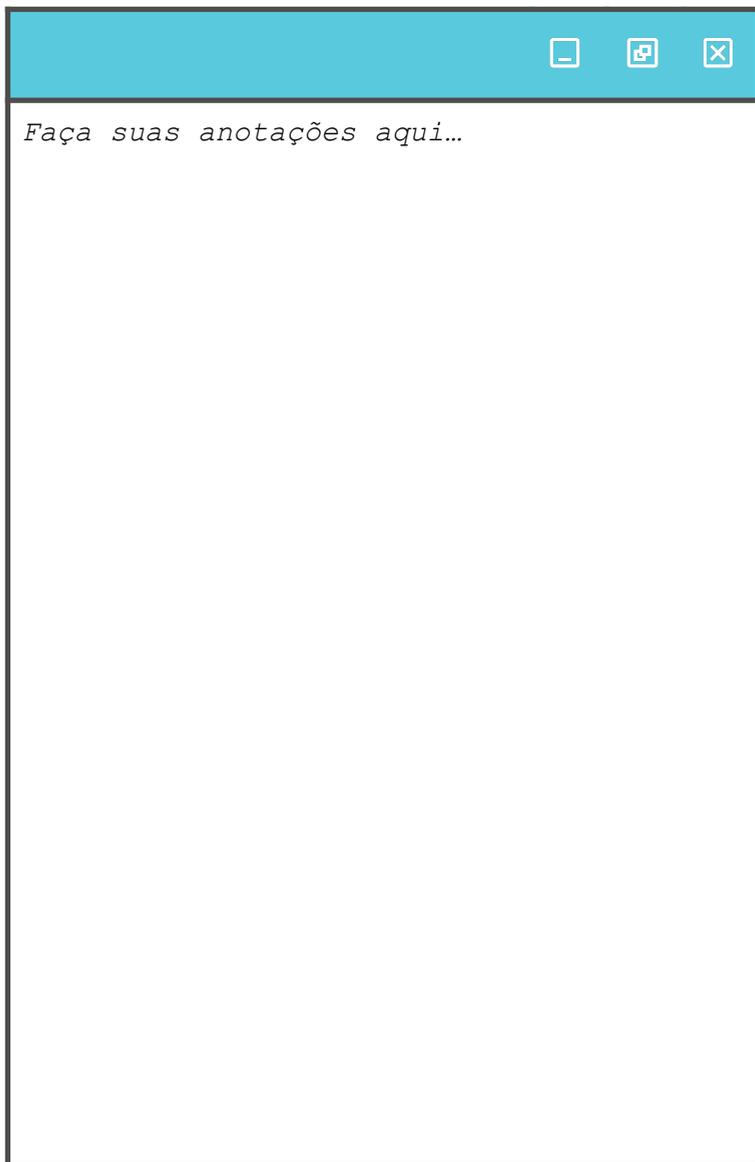
é possível utilizar métodos específicos para executar ações em cada fase.

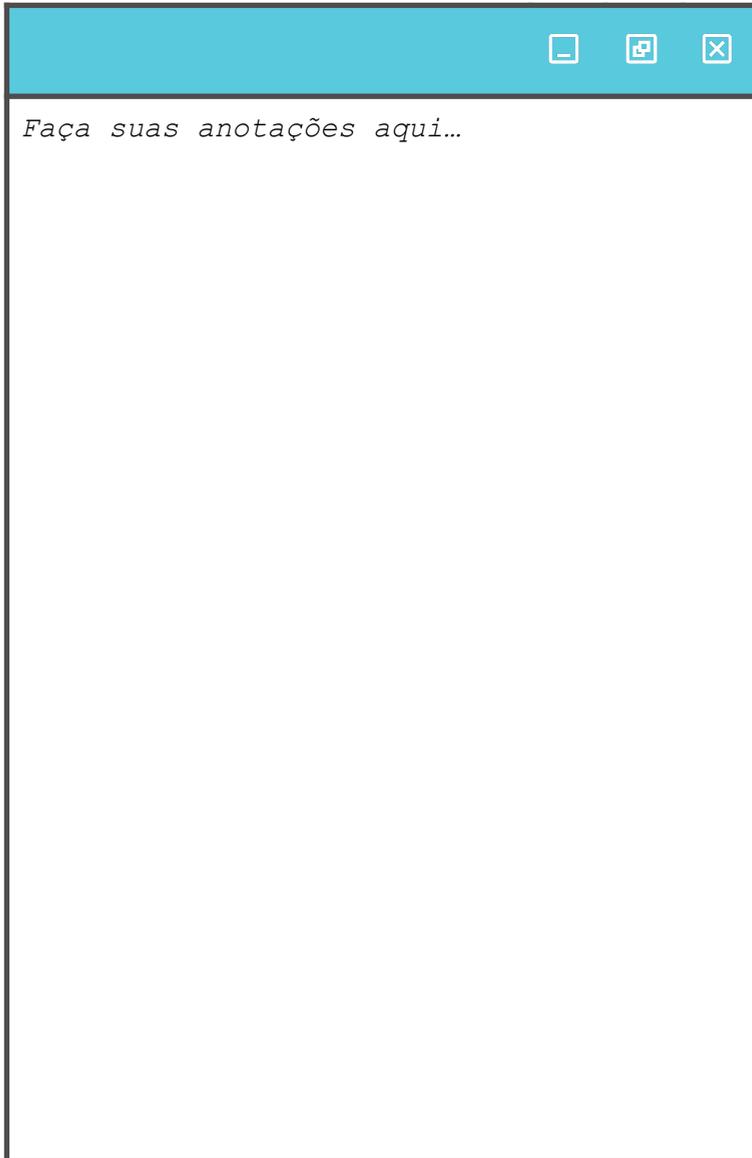
```
import React from 'react';
class MeuComponente extends
    React.Component
{
  constructor(props) {
    super(props);
    console.log('Componente iniciado');
  }
  componentDidMount() {
    console.log('Componente montado');
  }
  componentWillUnmount() {
    console.log('Componente
desmontado');
  }
  render() {
    return (
      <div>
        <h1>Meu Componente</h1>
        <p>Este é um componente
React</p>
      </div>
    );
  }
}
```

```
    }  
  }  
  export default MeuComponente;
```

Figura 30: Métodos específicos para executar ações em cada fase do ciclo de vida de um componente.

No exemplo acima, é utilizado uma classe para definir o componente **MeuComponente**. Este componente utiliza três métodos do ciclo de vida: o **construtor constructor**, que é executado no momento da inicialização do componente, o método **componentDidMount**, que é executado após o componente ser montado no DOM, e o método **componentWillUnmount**, que é executado antes do componente ser desmontado.





React Router Dom: Navegando entre páginas

O React Router Dom é uma biblioteca que permite fazer a navegação entre as páginas em uma aplicação React. Com o React Router Dom, é possível definir as **rotas** da aplicação e as páginas que serão exibidas para cada rota.

Para utilizar o React Router Dom, é necessário instalar a biblioteca através do gerenciador de pacotes do Node.js.

O comando para instalar o React Router Dom é:

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

Com o React Router Dom instalado, é possível definir as rotas da aplicação utilizando os componentes **BrowserRouter**, **Switch** e **Route**. O **BrowserRouter** é um componente que deve ser colocado em volta da aplicação para definir o contexto da navegação. O **Switch** é um componente que agrupa as rotas e garante que apenas uma rota será exibida por vez. Já o **Route** é um componente que define uma rota e a página que será exibida para essa rota.

Por exemplo, para definir duas rotas em uma aplicação, uma para a página inicial e outra para uma página de contato, pode-se utilizar o seguinte código:

```
import React from 'react';
import { BrowserRouter, Switch, Route }
from 'react-router-dom';
import PaginaInicial from
'./PaginaInicial';
import PaginaContato from
'./PaginaContato';

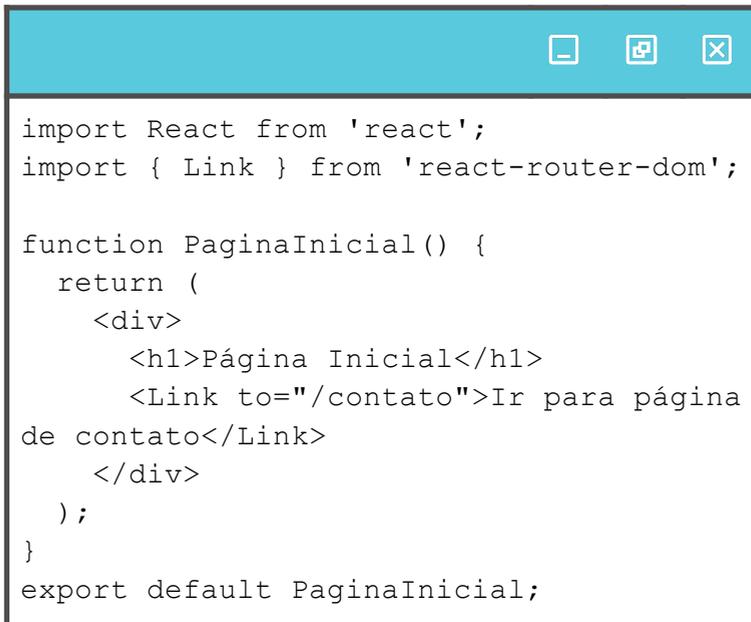
function App() {
  return (
    <BrowserRouter>
      <Switch>
        <Route
          exact path="/"
          component={PaginaInicial} />
        <Route
          path="/contato"
          component={PaginaContato} />
      </Switch>
    </BrowserRouter>
  );
}

export default App;
```

Figura 31: Definição de rotas com React Router Dom.

No exemplo da **Figura 31**, o **BrowserRouter** é utilizado para definir o contexto da navegação. O **Switch** agrupa as rotas, e o **Route** define duas rotas: uma rota para a página inicial ("/") e outra rota para a página de contato ("/contato"). Para cada rota, é definida a página que será exibida através da *prop* **component**.

Para navegar entre as páginas, é possível utilizar o componente **Link**, que é um componente que gera um link para uma rota, e quando clicado, navega para a página correspondente. Para navegar para a página de contato, pode-se utilizar o código da **Figura 32**.

A screenshot of a code editor window with a light blue header bar containing three window control icons (minimize, maximize, close). The main area is white and contains the following code:

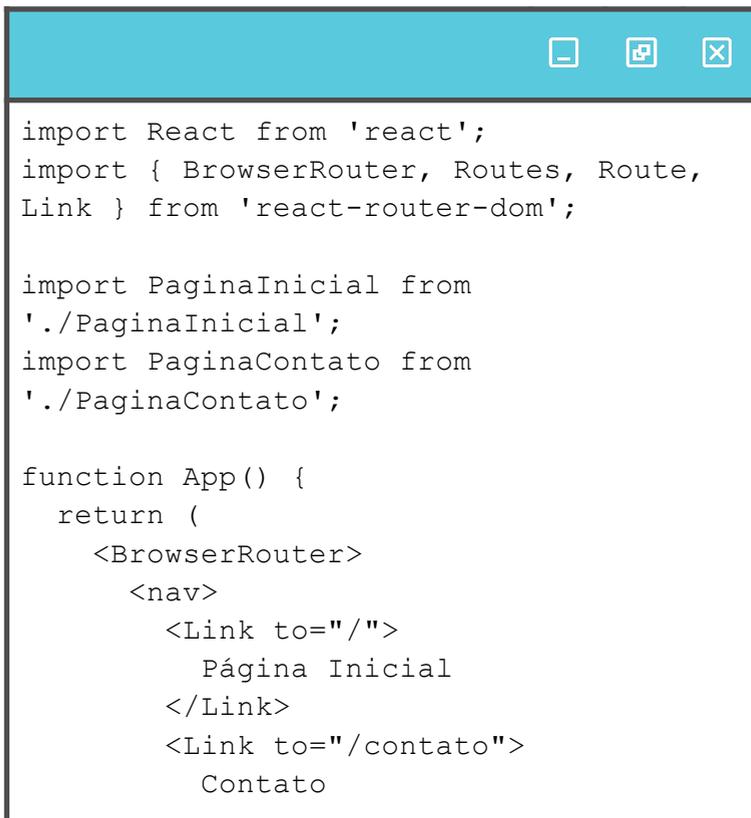
```
import React from 'react';
import { Link } from 'react-router-dom';

function PaginaInicial() {
  return (
    <div>
      <h1>Página Inicial</h1>
      <Link to="/contato">Ir para página
de contato</Link>
    </div>
  );
}
export default PaginaInicial;
```

Figura 32: Uso do componente *Link* para navegação.

No exemplo da **Figura 32**, o `Link` é utilizado para gerar um link para a rota `"/contato"`. Quando clicado, o link navega para a página de contato.

A partir da **versão 6**, o `React Router Dom` passou por algumas mudanças em relação à **sintaxe e utilização**. Na **Figura 33**, temos um exemplo de como fica o código para definir as rotas da aplicação utilizando a versão 6 do `React Router Dom`.



```
import React from 'react';
import { BrowserRouter, Routes, Route,
Link } from 'react-router-dom';

import PaginaInicial from
'./PaginaInicial';
import PaginaContato from
'./PaginaContato';

function App() {
  return (
    <BrowserRouter>
      <nav>
        <Link to="/">
          Página Inicial
        </Link>
        <Link to="/contato">
          Contato
        </Link>
      </nav>
    </BrowserRouter>
  );
}
```

```
        </Link>
    </nav>

    <Routes>
      <Route
        path="/"
        element={<PaginaInicial />} />
      <Route
        path="/contato"
        element={<PaginaContato />} />
    </Routes>
  </BrowserRouter>
);
}

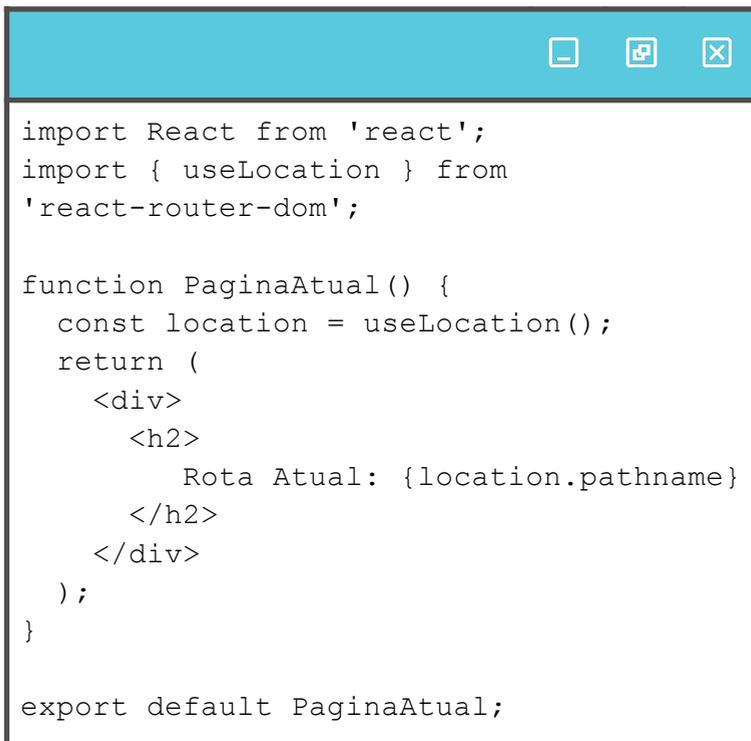
export default App;
```

Figura 33: Definição de rotas com a versão 6 do React Router Dom.

No exemplo da **Figura 33**, é utilizado o componente **BrowserRouter** para envolver a aplicação e o componente **Routes** para agrupar as rotas. As rotas são definidas através do componente **Route**. No lugar da *prop* **component**, utilizada na versão anterior, é utilizada a *prop* **element**, que recebe o componente a ser renderizado para a rota correspondente.

Da mesma forma que as versões anteriores, também é utilizado o componente **Link** para gerar links para as rotas, de forma semelhante à versão anterior.

É importante lembrar que a versão 6 do React Router Dom ainda está em beta e, portanto, pode sofrer mudanças no futuro. Para obter mais informações sobre a versão 6 do React Router Dom, consulte a documentação oficial.



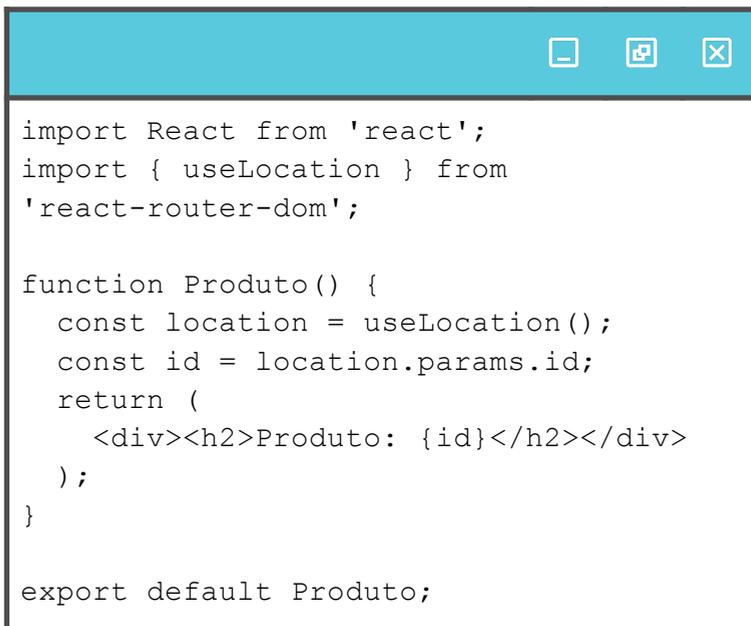
```
import React from 'react';
import { useLocation } from
'react-router-dom';

function PaginaAtual() {
  const location = useLocation();
  return (
    <div>
      <h2>
        Rota Atual: {location.pathname}
      </h2>
    </div>
  );
}

export default PaginaAtual;
```

Figura 34: Obtenção da rota atual com o *useLocation*.

O hook **useLocation** do React Router Dom é utilizado para obter informações sobre a localização atual do usuário na aplicação. Ele retorna um objeto com diversas informações, como a **rota atual**, **parâmetros de URL** e **estado da rota**. Para utilizar o **useLocation**, é necessário importá-lo do pacote **react-router-dom** e, em seguida, invocá-lo em um componente. A **Figura 34** ilustra como obter a rota atual a partir do **useLocation**. Note que a rota atual é obtida através da propriedade **pathname** do objeto retornado pelo **useLocation**.

A screenshot of a code editor window with a light blue header bar containing three icons: a square, a magnifying glass, and a close button. The main area is white and contains the following code:

```
import React from 'react';
import { useLocation } from
'react-router-dom';

function Produto() {
  const location = useLocation();
  const id = location.params.id;
  return (
    <div><h2>Produto: {id}</h2></div>
  );
}

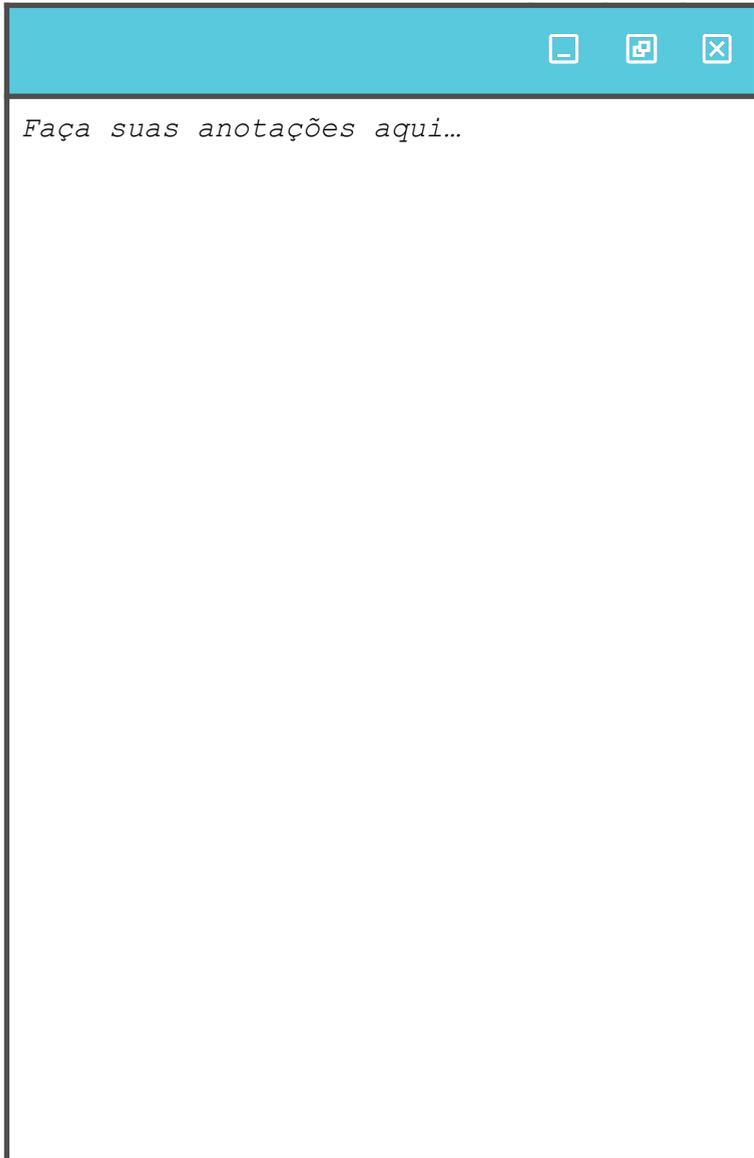
export default Produto;
```

Figura 35: Obtenção de parâmetros da URL com o *useLocation*.

Além disso, o **useLocation** também pode ser utilizado para obter **parâmetros de URL** e o **estado da rota**. Por exemplo, se a rota atual for **/produto/123**, é possível obter o **valor 123** do parâmetro de URL utilizando a propriedade **params** do objeto retornado pelo **useLocation**.

O exemplo da **Figura 35** ilustra como obter o valor de um parâmetro de URL. Neste exemplo, o valor do parâmetro **id** é obtido a partir da propriedade **params** do objeto retornado pelo **useLocation**.

Em resumo, o **useLocation** é um hook importante do **React Router Dom** que permite obter informações sobre a localização atual do usuário na aplicação, como a **rota atual**, **parâmetros de URL** e **estado da rota**.





Context API: Gerenciamento de Estados Globais

Uma das características do React é o gerenciamento de estados, que permite que a aplicação mantenha dados e informações que serão exibidas na tela. No entanto, em algumas situações, pode ser necessário compartilhar dados entre componentes que não estão diretamente relacionados, ou seja, que não são pais ou filhos um do outro. Para solucionar esse problema, React fornece a Context API, que é uma ferramenta para gerenciamento de estados globais.

A Context API permite que dados sejam compartilhados de forma eficiente entre componentes que estão em diferentes níveis da árvore de componentes do React. Para utilizar a Context API, é preciso criar um contexto com o método **createContext** do React. Esse contexto pode ser utilizado para compartilhar dados entre componentes que estão dentro dele.

Para adicionar dados ao contexto, é preciso utilizar o componente **Provider**, que recebe um objeto com os dados a serem compartilhados. Os componentes filhos (**children**) do Provider podem acessar esses dados utilizando o método **useContext**.

Por exemplo, vamos supor que queremos compartilhar o nome de usuário em toda a aplicação.

Para isso, podemos criar um contexto com o método `createContext`:

```
const UserContext =  
React.createContext();
```

Em seguida, podemos definir um componente `Provider` que irá fornecer o nome de usuário:

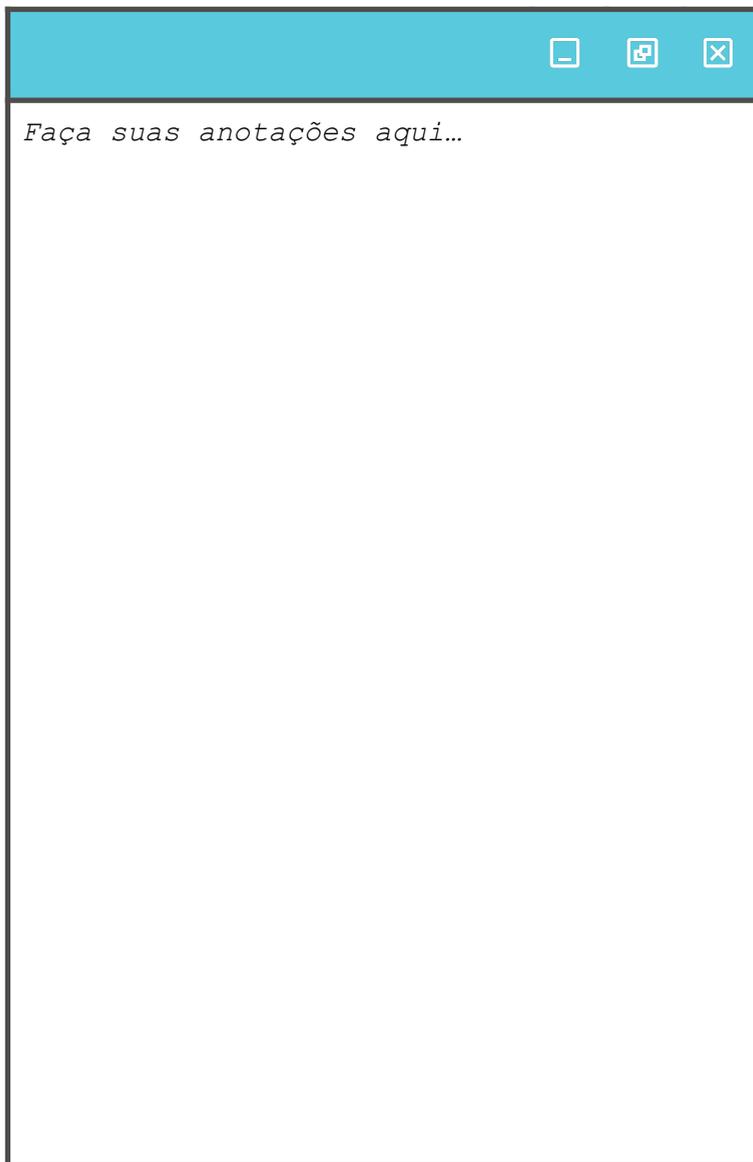
```
function UserProvider(props) {  
  const [username,  
        setUsername] = useState('John');  
  
  return (  
    <UserContext.Provider  
      value={{ username, setUsername }}>  
      {props.children}  
    </UserContext.Provider>  
  );  
}
```

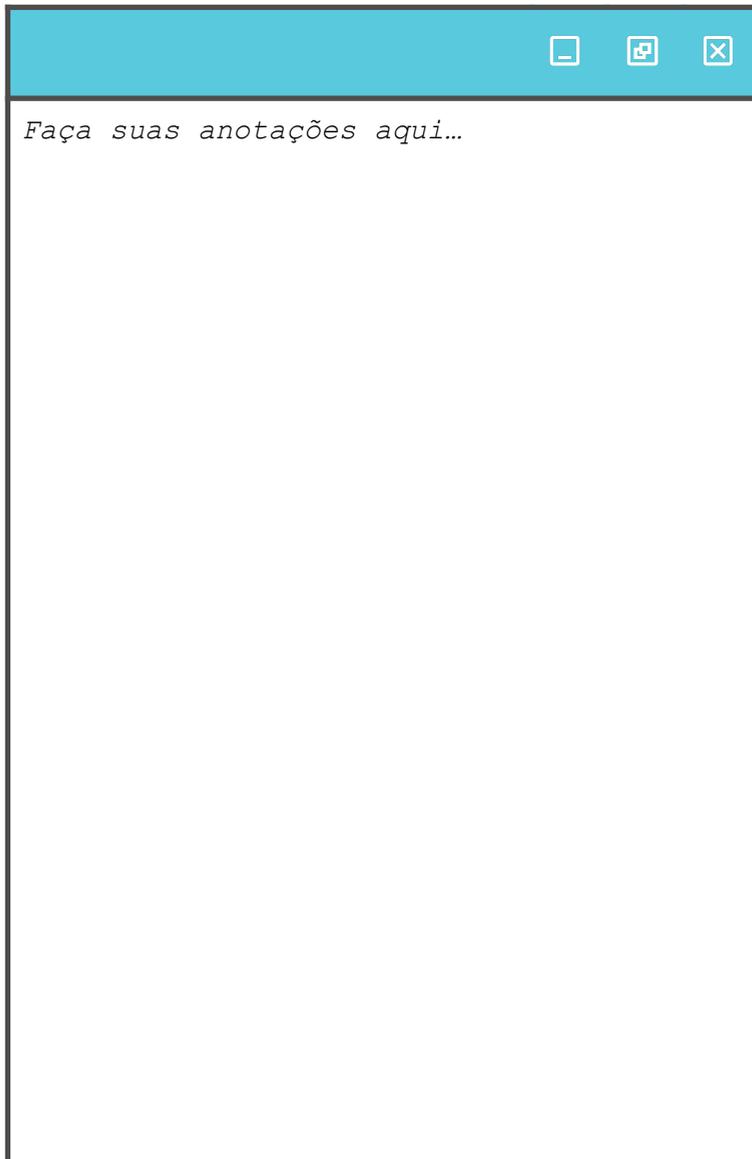
Agora, qualquer componente que esteja dentro do **UserContext.Provider** pode acessar o nome de usuário utilizando o método **useContext**:

```
function UserInfo() {  
  const { username } =  
  
  useContext(UserContext);  
  return <p>Username: {username}</p>;  
}
```

Assim, o componente **UserInfo** pode exibir o nome de usuário em qualquer lugar da aplicação, mesmo que ele não esteja diretamente relacionado com o componente **UserProvider**.

A Context API é uma ferramenta poderosa para gerenciamento de estados globais em aplicações React. No entanto, é importante lembrar que o seu uso deve ser cuidadoso, para evitar que a aplicação fique complexa demais e difícil de entender.





Integração com API Externas com Axios

O Axios é uma biblioteca JavaScript que é utilizada para realizar requisições HTTP a serviços web, como APIs externas. Ele oferece uma API simples e fácil de usar para realizar requisições com diversos métodos HTTP, como **GET**, **POST**, **PUT** e **DELETE**.

Para utilizá-lo em uma aplicação React, primeiro é necessário instalá-lo através do gerenciador de pacotes **npm**:

```
npm install axios
```

Em seguida, é necessário importá-lo no componente que irá realizar as requisições. O exemplo da **Figura 36** ilustra como realizar uma requisição GET para a API do GitHub e exibir os dados do usuário na tela.

Neste exemplo, é utilizado o hook **useEffect** para realizar a requisição GET para a API do GitHub quando o componente é montado. O resultado da requisição é armazenado no estado **user** utilizando o método **setUser**. Caso o valor de **user** seja nulo, a mensagem "Carregando.." é exibida na tela. Caso contrário, as informações do usuário, como nome e imagem de perfil, são exibidas na tela.

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

function GithubUser(props) {
  const [user, setUser] =
  useState(null);

  useEffect(() => {
    async function fetchUser() {
      const response = await
  axios.get(`https://api.github.com/users/
  ${props.username}`);
      setUser(response.data);
    }

    fetchUser();
  }, [props.username]);

  if (!user) {
    return <div>Carregando...</div>;
  }

  return (
    <div>
      <h2>{user.name}</h2>
      <img src={user.avatar_url}
        alt={user.name} />
    </div>
  );
}
```

```
    </div>
  );
}

export default GithubUser;
```

Figura 36: Chamada à API do GitHub com Axios.

Além de GET, é possível realizar requisições com outros métodos HTTP utilizando o Axios. Por exemplo, para realizar uma requisição POST com dados no corpo da requisição, basta utilizar o método **axios.post**:

```
axios.post('https://example.com/api/users', {
  name: 'John Doe',
  email: 'johndoe@example.com',
})
.then(response => {
  console.log(response.data);
})
.catch(error => {
  console.log(error);
});
```

Em resumo, o Axios é uma biblioteca JavaScript útil para realizar requisições HTTP a serviços web, como APIs externas. É fácil de usar e oferece uma API

simples e intuitiva para realizar requisições com diversos métodos HTTP.

O **fetch** é uma API nativa do JavaScript que também pode ser utilizada para realizar requisições HTTP a serviços web, como APIs externas. Ela também é simples e fácil de usar, e oferece uma API mais moderna e intuitiva que a antiga *API XMLHttpRequest*.

Abaixo, segue um exemplo que realiza a mesma requisição GET para a API do GitHub utilizando o *fetch*:

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

function GithubUser(props) {
  const [user, setUser] =
  useState(null);

  useEffect(() => {
    async function fetchUser() {
      const response = await
fetch(`https://api.github.com/users/${pr
ops.username}`);
      const data = await
response.json();
      setUser(data);
    }
  });
}
```

```
    }

    fetchUser();
  }, [props.username]);

  if (!user) {
    return <div>Carregando...</div>;
  }

  return (
    <div>
      <h2>{user.name}</h2>
      <img src={user.avatar_url}
          alt={user.name} />
    </div>
  );
}

export default GithubUser;
```

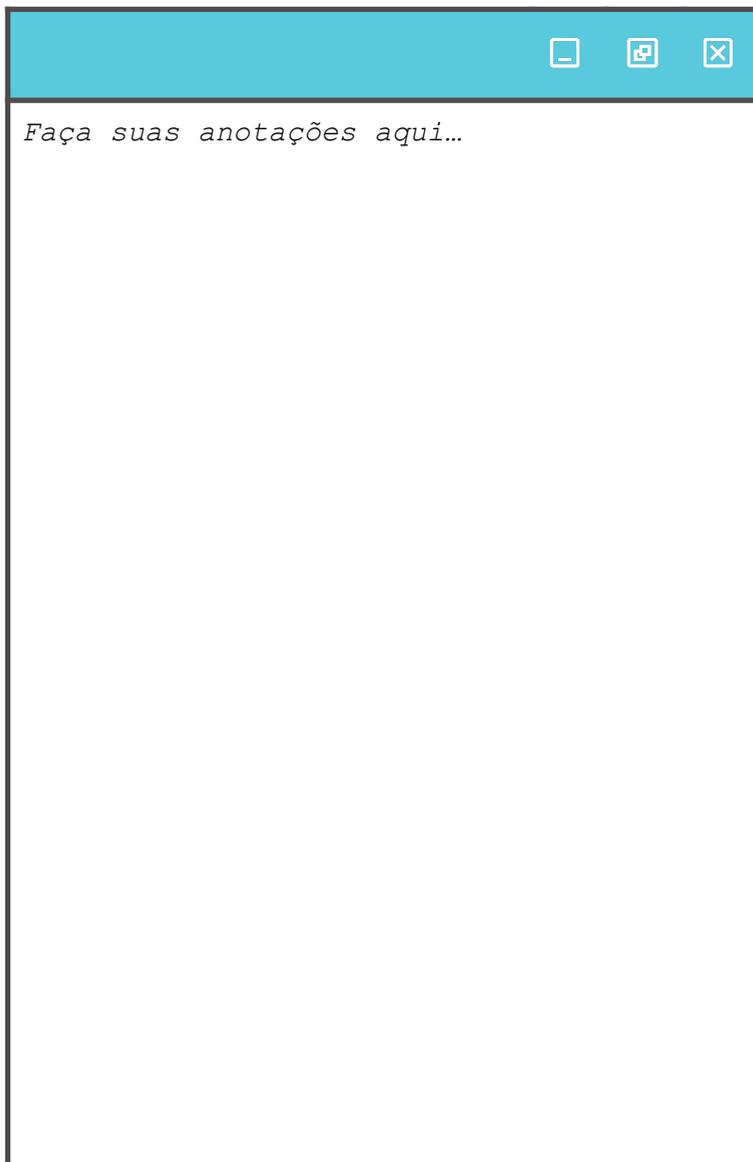
Figura 37: Chamada à API do GitHub com Axios.

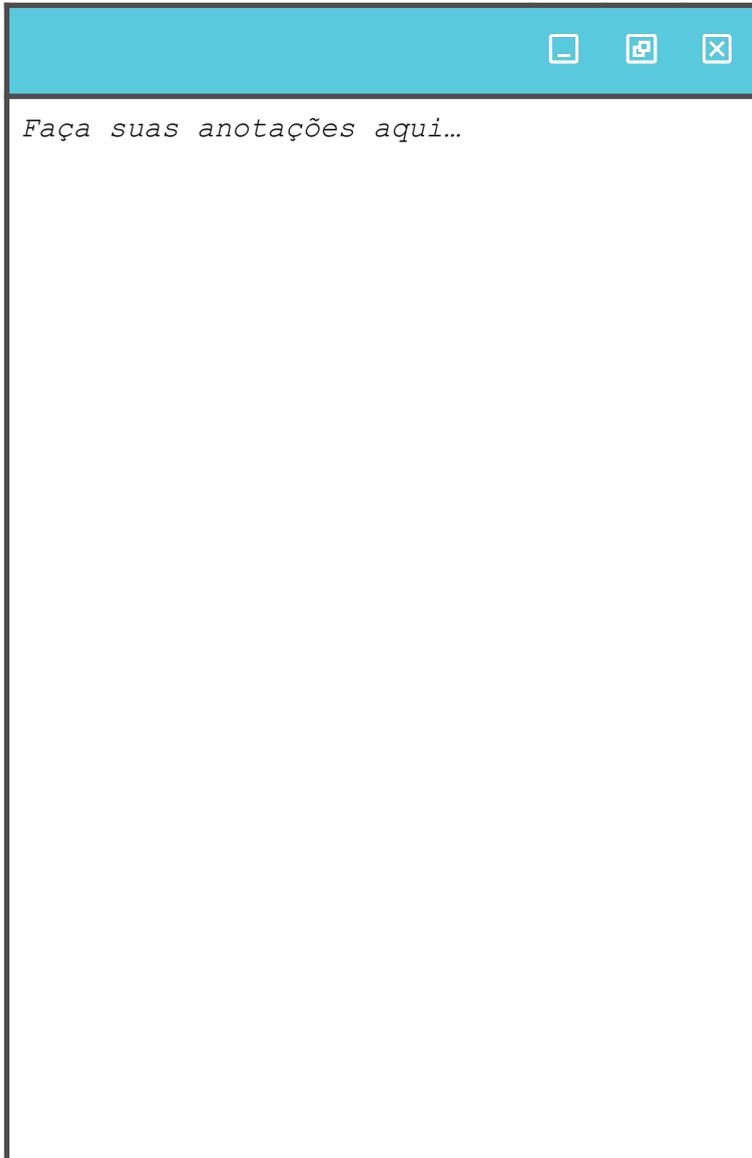
No exemplo da **Figura 37**, é utilizado o hook **useEffect** para realizar a requisição GET para a API do GitHub quando o componente é montado. O resultado da requisição é convertido para o formato JSON utilizando o método **response.json()**, e o resultado é armazenado no estado **user** utilizando o método **setUser**. Caso o valor de **user** seja nulo, a mensagem "Carregando..." é exibida na tela. Caso contrário, as

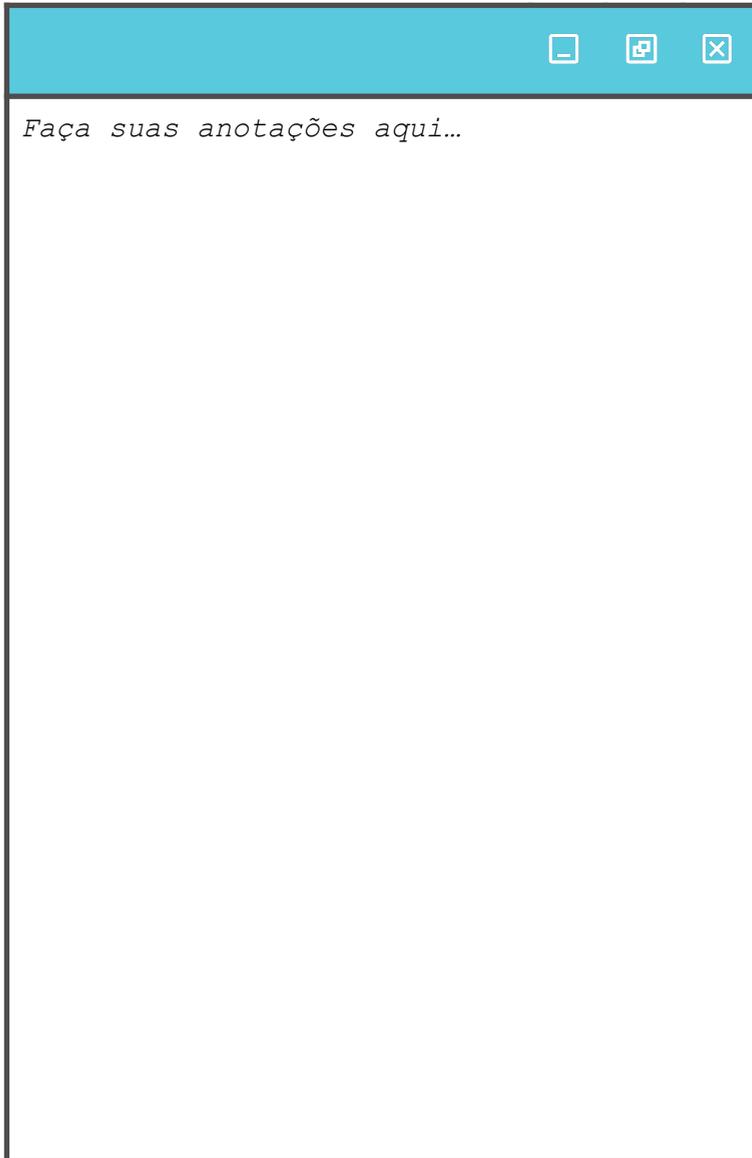
informações do usuário, como nome e imagem de perfil, são exibidas na tela.

Em relação ao Axios, a API fetch é nativa do JavaScript e **não requer a instalação de bibliotecas adicionais**. Além disso, ela oferece suporte a todas as funcionalidades disponíveis no Axios, como a especificação de cabeçalhos HTTP e a utilização de outros métodos HTTP. Entretanto, a API fetch pode apresentar algumas dificuldades em relação à manipulação de erros, e pode exigir o uso de código adicional para lidar com casos de erro na requisição.

Em resumo, tanto o Axios quanto a API fetch são opções viáveis para realizar requisições HTTP em aplicações React. Cada uma tem suas vantagens e desvantagens, e a escolha dependerá do contexto específico da aplicação e das preferências do desenvolvedor.







Conclusão

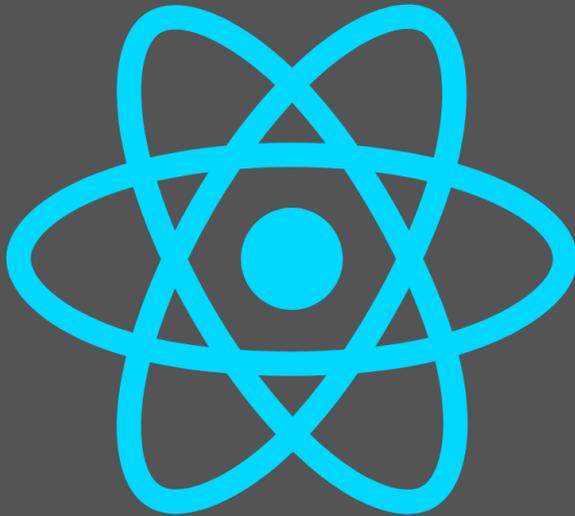
O React.js é uma poderosa biblioteca JavaScript para desenvolvimento de interfaces de usuário. Durante o curso, aprendemos sobre os principais conceitos e recursos do React, incluindo componentes, estado e propriedades, ciclo de vida de componentes, gerenciamento de eventos, roteamento e muito mais. Com a sua utilização, vimos que é possível criar interfaces complexas e dinâmicas com facilidade.

Ao longo do curso, pudemos colocar em prática nossos conhecimentos por meio de exercícios e projetos práticos, o que nos permitiu desenvolver aplicações reais usando React. Além disso, fomos apresentados a boas práticas e padrões de projeto para desenvolvimento de aplicações React, o que nos permitirá criar aplicações mais robustas, escaláveis e fáceis de manter.

Como resultado, podemos concluir que este curso foi uma experiência valiosa e enriquecedora para todos os participantes. Aprendemos uma das tecnologias mais populares e procuradas do mercado, o que aumenta nossas chances de empregabilidade e nos permitirá desenvolver aplicações web modernas e de alta qualidade. Esperamos que, após o término deste curso, possamos continuar a aprimorar nossos conhecimentos em React e desenvolver aplicações ainda mais avançadas e interessantes.



www.treinarecife.com.br



SAIBA MAIS EM:

www.alexandrejunior.dev