

O que é o DOM?

DOM significa **Document Object Model** (Modelo de Objeto de Documento).

Ele é uma **interface de programação** que os navegadores usam para representar páginas web de forma estruturada e interativa.

Em termos simples:

O DOM é a maneira como o navegador **vê** e **organiza** a página web — como uma árvore de objetos — para que **programas**, como o JavaScript, possam **ler**, **alterar**, **adicionar** ou **remover** elementos da página dinamicamente.

Como o DOM funciona?

Quando você carrega uma página web (HTML), o navegador:

1. **Lê** o código HTML.
2. **Interpreta** a estrutura dos elementos (`<html>`, `<head>`, `<body>`, `<div>`, etc.).
3. **Cria uma árvore de nós** (nós = nodes) em memória para representar essa estrutura.

Essa árvore é chamada de **Árvore DOM**.

Cada parte do HTML vira um objeto:

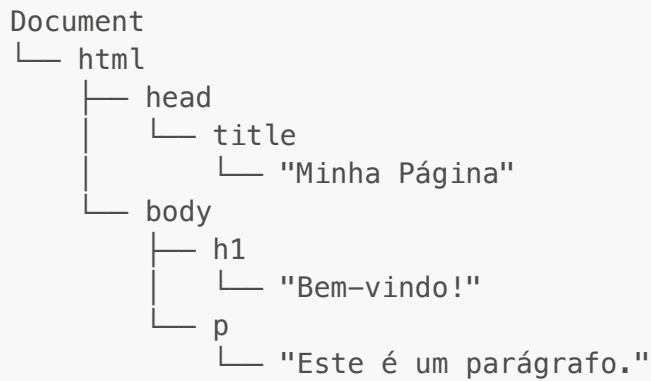
- As **tags** viram **elementos**.
 - Os **textos** dentro das tags viram **nós de texto**.
 - Os **atributos** (como `id`, `class`, etc.) são tratados como **propriedades** desses objetos.
-

Exemplo de Árvore DOM

Imagine esse HTML simples:

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

O navegador cria essa estrutura em memória:



Ou seja, o **Document** é o **nó raiz**, e tudo dentro dele é organizado como uma **árvore de objetos**.

Tipos de nós (Nodes) no DOM

No DOM, existem vários tipos de nós:

Tipo de Nó	Exemplo
Document	Representa o documento inteiro
Element	Tags HTML, como <code><p></code> , <code><h1></code> , <code><div></code> , etc.
Text	Texto dentro de elementos
Attribute	Atributos das tags (como <code>class</code> , <code>id</code>)
Comment	Comentários no HTML (<code><!-- Comentário --></code>)

O que é o DOM em Aplicações Web?

O **DOM** (Document Object Model) é uma interface de programação para documentos HTML e XML. Ele representa a estrutura de um documento de forma hierárquica, permitindo que os desenvolvedores manipulem dinamicamente o conteúdo, a estrutura e o estilo de uma página web. Em outras palavras, o DOM cria uma **representação em árvore** de todos os elementos de uma página (como `<div>`, `<p>`, ``, etc.) e permite que esses elementos sejam manipulados diretamente via JavaScript.

Como usamos o DOM nas aplicações web?

Em aplicações web modernas, o DOM é a ponte entre o **navegador** e o **JavaScript**, permitindo que interajamos com o conteúdo da página após ela ser carregada. Podemos **criar, modificar, remover e atualizar** elementos, atributos e até o estilo visual de uma página sem a necessidade de recarregar a página inteira. Isso torna a navegação e interação mais rápidas e fluídas, proporcionando uma **experiência de usuário mais dinâmica e responsiva**.

Para que usamos o DOM nas aplicações?

O DOM é utilizado para **dinamizar e interagir com o conteúdo da página**, tornando-a mais interativa. Alguns dos principais usos incluem:

1. **Manipulação de conteúdo dinâmico:** Podemos atualizar textos, imagens ou outros elementos da página sem precisar recarregar toda a página. Por exemplo, em sistemas de chat, novas mensagens podem ser exibidas dinamicamente sem a necessidade de atualizar a página.
2. **Criação e remoção de elementos:** O DOM nos permite criar novos elementos (como botões, listas, formulários) e removê-los de forma programática. Isso é útil para criar interfaces interativas e sistemas de carregamento dinâmico de conteúdo.
3. **Alteração de estilos e classes:** Através do DOM, podemos modificar as propriedades de estilo de um elemento, como cor, tamanho, margem, entre outros. Também podemos adicionar ou remover classes CSS, permitindo que as páginas mudem visualmente em tempo real com base em interações do usuário.
4. **Manipulação de formulários:** O DOM permite que a gente altere os valores de campos de formulários (como inputs, selects, checkboxes), e também valide e envie dados sem recarregar a página (por exemplo, com Ajax).
5. **Eventos e interações:** O DOM é essencial para capturar e responder a eventos de interação do usuário, como **cliques, teclados, movimentos do mouse**, etc. Ele permite a **geração de respostas dinâmicas**, como exibir alertas, mudar o conteúdo de um campo ou fazer animações.

Onde usamos o DOM no desenvolvimento de software?

O DOM é **onipresente** no desenvolvimento de **aplicações web**. Qualquer página web que tenha JavaScript usa o DOM para permitir interatividade. Aqui estão alguns exemplos de onde usamos o DOM:

1. **Aplicações interativas (SPA):** Em aplicações de página única (Single Page Applications - SPAs), como aquelas feitas com **React, Vue.js** ou **Angular**, o DOM é manipulado constantemente para atualizar a interface sem a necessidade de recarregar a página inteira. Cada vez que o estado da aplicação muda, o DOM é atualizado para refletir as novas informações.
2. **Sites dinâmicos e interativos:** Websites que têm conteúdo que muda em tempo real, como **sites de notícias, dashboards, sistemas de e-commerce** ou **blogs**, frequentemente manipulam o DOM para carregar novos artigos, atualizar informações em tempo real, exibir produtos baseados nas escolhas do usuário, etc.
3. **Jogo e entretenimento interativo:** Muitos jogos baseados em navegador ou animações interativas dependem do DOM para mover objetos na tela, interagir com o usuário e responder a eventos.
4. **Acessibilidade e controle de formulário:** O DOM é fundamental para a criação de **formulários acessíveis**, onde podemos manipular o foco (seleção de campo), exibir mensagens de erro de validação, ou até mesmo personalizar os campos dinamicamente com base nas respostas do usuário.

Como o DOM no dia a dia do desenvolvimento de software?

Explicação Detalhada sobre **Selecionando Elementos** no DOM

Selecionar elementos é a **primeira etapa** para interagir com qualquer página web através do DOM. Antes de podermos modificar ou manipular qualquer coisa na página, precisamos **localizar os elementos** que queremos afetar. O DOM oferece várias maneiras de **selecionar** esses elementos com base em seus atributos, como **ID**, **classe**, **nome da tag** ou outros atributos personalizados.

Por que é importante?

Quando você trabalha com DOM, o processo de **seleção** é o primeiro passo para poder alterar o conteúdo, estilo ou comportamento de um elemento específico na página.

Como Selecionamos Elementos?

O método `getElementById()` é utilizado para selecionar um único elemento que tenha um **ID específico**. Esse método retorna o primeiro (e geralmente único) elemento que tem o ID fornecido.

- **Uso:** `document.getElementById('id')`
- **Retorna:** O elemento que possui o ID especificado ou `null` se não encontrar nenhum elemento com esse ID.

Exemplo:

```
<!DOCTYPE html>
<html lang="pt-BR">
<head>
  <meta charset="UTF-8">
  <title>Exemplo getElementById</title>
</head>
<body>
  <h1 id="titulo">Olá, Mundo!</h1>
  <button id="botao">Clique aqui</button>

  <script>
    const titulo = document.getElementById('titulo');
    console.log(titulo); // Exibe o elemento <h1 id="titulo">

    const botao = document.getElementById('botao');
    console.log(botao); // Exibe o elemento <button id="botao">
  </script>
</body>
</html>
```

Neste exemplo, o `getElementById()` retorna os elementos `<h1>` e `<button>` que possuem os IDs "titulo" e "botao", respectivamente.

2. `getElementsByClassName()`

O método `getElementsByClassName()` é usado para selecionar todos os elementos que têm uma determinada **classe**. Esse método retorna uma coleção de elementos com a classe indicada.

- **Uso:** `document.getElementsByClassName('nome-da-classe')`
- **Retorna:** Uma coleção (não um único elemento) de todos os elementos que têm a classe especificada. Para acessar um elemento específico dessa coleção, você deve usar um índice.

Exemplo:

```
<!DOCTYPE html>
<html lang="pt-BR">
<head>
  <meta charset="UTF-8">
  <title>Exemplo getElementByClassName</title>
</head>
<body>
  <p class="paragrafo">Parágrafo 1</p>
  <p class="paragrafo">Parágrafo 2</p>
  <p class="paragrafo">Parágrafo 3</p>

  <script>
    const paragrafos = document.getElementsByClassName('paragrafo');
    console.log(paragrafos); // Retorna uma coleção de todos os
    elementos <p> com a classe "paragrafo"

    // Acessando um item específico na coleção
    console.log(paragrafos[1]); // Exibe o segundo parágrafo
  </script>
</body>
</html>
```

Neste caso, todos os parágrafos que têm a classe `"paragrafo"` são selecionados. A variável `paragrafos` contém uma **coleção de elementos**, então você precisa usar um índice para acessar um item específico.

3. `querySelector()`

O método `querySelector()` é uma maneira muito poderosa e flexível de selecionar um **único** elemento, baseado em um **seletor CSS**. Ele funciona de forma semelhante a como você selecionaria elementos com CSS, permitindo selecionar por **ID**, **classe**, **nome de tag**, entre outros.

- **Uso:** `document.querySelector('seletor')`
- **Retorna:** O **primeiro** elemento que corresponde ao seletor CSS especificado ou `null` se não encontrar nenhum elemento correspondente.

Exemplo:

```

<!DOCTYPE html>
<html lang="pt-BR">
<head>
  <meta charset="UTF-8">
  <title>Exemplo querySelector</title>
</head>
<body>
  <div class="caixa">Caixa 1</div>
  <div class="caixa">Caixa 2</div>
  <div class="caixa">Caixa 3</div>

  <script>
    const caixa = document.querySelector('.caixa');
    console.log(caixa); // Exibe o **primeiro** <div class="caixa">
  </script>
</body>
</html>

```

Aqui, o `querySelector()` retorna o **primeiro** `<div>` que tem a classe `"caixa"`. Se você quiser selecionar todos os elementos com a classe, deve usar `querySelectorAll()`.

4. `querySelectorAll()`

O método `querySelectorAll()` seleciona **todos** os elementos que correspondem ao seletor CSS fornecido. Ele retorna uma lista de **todos os elementos** que combinam com o seletor.

- **Uso:** `document.querySelectorAll('seletor')`
- **Retorna:** Uma lista **NodeList** de todos os elementos que correspondem ao seletor.

Exemplo:

```

<!DOCTYPE html>
<html lang="pt-BR">
<head>
  <meta charset="UTF-8">
  <title>Exemplo querySelectorAll</title>
</head>
<body>
  <p class="texto">Texto 1</p>
  <p class="texto">Texto 2</p>
  <p class="texto">Texto 3</p>

  <script>
    const textos = document.querySelectorAll('.texto');
    console.log(textos); // Retorna uma NodeList de todos os <p>
    com a classe "texto"

    // Acessando um item específico
    console.log(textos[1]); // Exibe o segundo parágrafo com a

```

```
classe "texto"
</script>
</body>
</html>
```

Neste exemplo, `querySelectorAll()` seleciona todos os elementos `<p>` que têm a classe `"texto"`, e retorna uma **NodeList** que você pode percorrer.

Resumo de Seleção de Elementos

Método	Descrição	Retorna
<code>getElementById()</code>	Seleciona um elemento pelo ID .	Um único elemento (ou <code>null</code> se não encontrar)
<code>getElementsByName()</code>	Seleciona todos os elementos por classe .	Uma coleção de elementos
<code>querySelector()</code>	Seleciona o primeiro elemento que corresponde ao seletor.	Um único elemento (ou <code>null</code> se não encontrar)
<code>querySelectorAll()</code>	Seleciona todos os elementos que correspondem ao seletor.	Uma lista de elementos (NodeList)

Exemplos Simples e Didáticos:

1. Selecionando um elemento por ID:

```
<div id="paragrafo1">Texto do Parágrafo 1</div>
<script>
  const paragrafo1 = document.getElementById('paragrafo1');
  console.log(paragrafo1); // Exibe o elemento <div id="paragrafo1">
</script>
```

2. Selecionando elementos por classe:

```
<div class="item">Item 1</div>
<div class="item">Item 2</div>
<div class="item">Item 3</div>
<script>
  const items = document.getElementsByClassName('item');
  console.log(items); // Exibe todos os elementos com a classe "item"
</script>
```

3. Usando `querySelector()` para selecionar o primeiro elemento:

```
<p class="alerta">Alerta 1</p>
<p class="alerta">Alerta 2</p>
<script>
  const alerta = document.querySelector('.alerta');
  console.log(alerta); // Exibe o **primeiro** <p> com a classe
  "alerta"
</script>
```

4. Usando `querySelectorAll()` para selecionar todos os elementos com uma classe:

```
<p class="noticia">Notícia 1</p>
<p class="noticia">Notícia 2</p>
<script>
  const noticias = document.querySelectorAll('.noticia');
  console.log(noticias); // Exibe todos os <p> com a classe "noticia"
</script>
```

A seleção de elementos no DOM é essencial para qualquer manipulação dinâmica de conteúdo em uma página web. A escolha do método de seleção depende do que você precisa fazer:

- Use `getElementById()` para **IDs únicos**.
- Use `getElementsByClassName()` ou `querySelectorAll()` para selecionar **múltiplos elementos** com uma **classe**.
- Use `querySelector()` para **selecionar o primeiro elemento** correspondente a um seletor CSS.

Manipulando conteúdo

- `textContent`
- `innerHTML`
- `value`
- Manipulando conteúdo de **div**, **input** e **parágrafos**.

Vai ficar simples, bonito e didático:

```
<!DOCTYPE html>
<html lang="pt-BR">
<head>
  <meta charset="UTF-8">
  <title>Exemplos de Manipulação de Conteúdo</title>
  <style>
    body {
      font-family: Arial, sans-serif;
      margin: 20px;
    }
  </style>
</head>
<body>
```



```

    .caixa {
        margin: 20px 0;
        padding: 10px;
        border: 1px solid #ccc;
        background-color: #f9f9f9;
    }
    button {
        margin: 5px;
        padding: 5px 10px;
        cursor: pointer;
    }
</style>
</head>
<body>

    <h1 id="titulo">Título Original</h1>

    <div class="caixa" id="caixa">
        Aqui é a caixa de texto inicial.
    </div>

    <input type="text" id="campoTexto" value="Valor Inicial">
    <br><br>

    <button onclick="mudarTitulo()">Alterar Título (textContent)</button>
    <button onclick="alterarCaixa()">Alterar Caixa (innerHTML)</button>
    <button onclick="alterarCampo()">Alterar Campo (value)</button>
    <button onclick="mostrarCampo()">Mostrar Valor do Campo</button>

    <p id="resultado"></p>

    <script>
        function mudarTitulo() {
            const titulo = document.getElementById('titulo');
            titulo.textContent = 'Título alterado com textContent!';
        }

        function alterarCaixa() {
            const caixa = document.getElementById('caixa');
            caixa.innerHTML = '<strong>Texto mudado com HTML!</strong> Agora
tem <em>formatação</em>.';
        }

        function alterarCampo() {
            const campo = document.getElementById('campoTexto');
            campo.value = 'Novo valor definido via JavaScript';
        }

        function mostrarCampo() {
            const campo = document.getElementById('campoTexto');
            const resultado = document.getElementById('resultado');
            resultado.textContent = 'Valor atual do campo: ' + campo.value;
        }
    </script>

```

```
</script>

</body>
</html>
```

que essa página faz?

- Tem um **título** (**h1**) que muda usando **textContent**.
- Tem uma **div** ("**caixa**") que muda usando **innerHTML** com tags HTML (negrito, itálico).
- Tem um **campo de texto** (**input**) que:
 - Pode ter o **valor alterado** com JavaScript (**value**).
 - Pode ter o **valor mostrado** em um parágrafo.

3. Alterando Estilos

Claro! Vou fazer uma explicação bem detalhada sobre **Alterando Estilos com DOM**, com exemplos simples e didáticos para ficar bem claro.



Alterando Estilos com o DOM

O que significa "alterar estilos"?

Quando alteramos estilos pelo DOM, estamos **mudando a aparência de elementos HTML** usando **JavaScript**, sem precisar editar diretamente o arquivo CSS.

Isso permite que sua página web **responda dinamicamente** às ações do usuário, como um clique ou uma digitação.



Como Alterar Estilos?

Existem **duas formas principais** de alterar o estilo de um elemento:

1. Usar a propriedade **.style**

Você pode acessar e modificar o estilo **diretamente** usando **.style**.

O nome das propriedades de CSS muda para o formato **camelCase** no JavaScript.

CSS	JavaScript (style)
background-color	backgroundColor
font-size	fontSize
text-align	textAlign

2. Usar `.classList` para adicionar/remover classes

Em vez de mexer no estilo individualmente, você pode **adicionar** ou **remover classes CSS** usando `classList`.

Assim, o estilo continua organizado no seu arquivo `.css` e o JS apenas gerencia **qual classe** o elemento possui.

Exemplos Didáticos

1. Alterando diretamente com `.style`

```
<button id="botao">Clique para mudar cor</button>

<script>
  const botao = document.getElementById('botao');
  botao.onclick = function() {
    botao.style.backgroundColor = 'blue';
    botao.style.color = 'white';
    botao.style.fontSize = '20px';
  };
</script>
```

Explicação:

- Quando o botão for clicado, o fundo dele vai mudar para azul, o texto para branco e o tamanho da fonte vai aumentar.
-

2. Alterando estilo com `classList.add` e CSS

```
<style>
  .estilo-destacado {
    background-color: yellow;
    color: red;
    font-weight: bold;
  }
</style>

<button id="destacar">Destacar Texto</button>
<p id="texto">Este é um texto normal.</p>

<script>
  const botao = document.getElementById('destacar');
  const texto = document.getElementById('texto');

  botao.addEventListener('click', function() {
```

```
texto.classList.add('estilo-destacado');
});
</script>
```

Explicação:

- Ao clicar no botão, adicionamos a classe **estilo-destacado** ao `<p>`, mudando todo o seu estilo de uma vez só.
- Essa abordagem deixa o JavaScript mais limpo e o CSS organizado.

3. Alternar estilos com `classList.toggle`

```
<style>
  .ativo {
    background-color: green;
    color: white;
    padding: 10px;
  }
</style>

<button id="alternar">Ativar/Desativar Estilo</button>

<script>
  const botao = document.getElementById('alternar');

  botao.addEventListener('click', function() {
    botao.classList.toggle('ativo');
  });
</script>
```

Explicação:

- Toda vez que clicar no botão, ele **ativa ou desativa** a classe **ativo**.
- **toggle** é útil para criar efeitos de liga/desliga.

Como?	Quando usar?	Exemplo
<code>.style.propriedade</code>	Quando mudar UM estilo específico	<code>element.style.color = 'blue'</code>

Como?	Quando usar?	Exemplo
<code>.classList.add('classe')</code>	Para aplicar vários estilos definidos no CSS	<code>element.classList.add('ativo')</code>
<code>.classList.remove('classe')</code>	Para remover um estilo CSS aplicado	<code>element.classList.remove('ativo')</code>
<code>.classList.toggle('classe')</code>	Para alternar entre estilos (ligar/desligar)	<code>element.classList.toggle('ativo')</code>

Alterar estilos com DOM é essencial para criar páginas **dinâmicas e interativas**.

Usamos isso para:

- **Destacar** elementos ao passar o mouse
- **Mostrar/ocultar** áreas de conteúdo
- **Responder** a cliques e ações do usuário
- **Adaptar** a aparência da página dinamicamente

Se quiser, posso também montar um **mini-projeto prático** aplicando essas técnicas para fixar ainda mais. Quer? 🎯

4. Criando e Removendo Elementos

Criando e Removendo Elementos no DOM

Quando estamos desenvolvendo uma aplicação web, muitas vezes precisamos **adicionar novos elementos** à página dinamicamente (por exemplo, um novo item de lista, uma nova mensagem no chat, uma nova linha numa tabela) ou então **remover elementos** (como excluir uma notificação depois que o usuário lê).

O **DOM** nos dá comandos muito simples para fazer isso:

- **Criar elementos:** usando `document.createElement()`
- **Adicionar elementos:** usando `appendChild()`, `prepend()`, `insertBefore()`
- **Remover elementos:** usando `remove()` ou `removeChild()`

Como Criar um Novo Elemento

1. Criar o Elemento

Usamos `document.createElement('tag')` para **criar** um novo elemento (como uma `div`, um `p`, um `li`, etc.).

```
const novoParagrafo = document.createElement('p');
```

Agora temos um **elemento vazio** — só criamos ele, ainda não colocamos na página.

2. Adicionar Conteúdo ao Elemento

Podemos preencher esse novo elemento usando `textContent`, `innerHTML` ou outras propriedades.

```
novoParagrafo.textContent = 'Este é um parágrafo novo criado via JavaScript!';
```

3. Inserir o Elemento na Página

Depois que criamos e preenchemos o elemento, precisamos **inserir ele no DOM**, usando:

- `appendChild()`: adiciona como **último filho** de outro elemento.
- `prepend()`: adiciona como **primeiro filho** de outro elemento.

Exemplo:

```
const container = document.getElementById('container'); // Seleciona onde vamos colocar
container.appendChild(novoParagrafo); // Adiciona no final
```

Exemplo Completo de Criação:

```
<!DOCTYPE html>
<html lang="pt-BR">
<head>
  <meta charset="UTF-8">
  <title>Exemplo de Criação</title>
</head>
<body>

  <div id="container">
    <h1>Bem-vindo!</h1>
  </div>

  <script>
```

```
const novoParagrafo = document.createElement('p'); // Criar elemento
novoParagrafo.textContent = 'Parágrafo criado dinamicamente!'; //
Adicionar conteúdo
const container = document.getElementById('container'); // Selecionar
local
container.appendChild(novoParagrafo); // Inserir na página
</script>

</body>
</html>
```

Como Remover um Elemento

Existem duas formas muito usadas:

1. `element.remove()`

Remove **diretamente** o elemento.

```
const elemento = document.getElementById('meuElemento');
elemento.remove();
```

2. `parentNode.removeChild(element)`

Se você tiver o **pai** do elemento, pode pedir para o pai remover o filho:

```
const elemento = document.getElementById('meuElemento');
elemento.parentNode.removeChild(elemento);
```

(Essa forma é mais antiga, mas ainda funciona muito bem.)

Exemplo Completo de Remoção:

```
<!DOCTYPE html>
<html lang="pt-BR">
<head>
  <meta charset="UTF-8">
  <title>Exemplo de Remoção</title>
</head>
<body>

  <div id="container">
    <p id="paragrafo">Este parágrafo será removido!</p>
```

```

<button id="removeBtn">Remover Parágrafo</button>
</div>

<script>
  const botao = document.getElementById('removeBtn');
  const paragrafo = document.getElementById('paragrafo');

  botao.addEventListener('click', function() {
    paragrafo.remove(); // Remove o parágrafo quando o botão for clicado
  });
</script>

</body>
</html>

```

Resumo Prático:

O que fazer?	Como fazer?
Criar um novo elemento	<code>document.createElement('tag')</code>
Adicionar conteúdo	<code>element.textContent = 'Texto'</code>
Inserir na página	<code>parentElement.appendChild(element)</code>
Remover um elemento direto	<code>element.remove()</code>
Remover como filho do pai	<code>parentElement.removeChild(element)</code>

Dicas Importantes:

- Sempre **crie** o elemento, **modifique** (adicione texto, classe, etc.), e só depois **adicione** na página.
- Se você for **remover**, verifique se o elemento realmente existe (`if (elemento)`) para evitar erros.
- Lembre que para adicionar mais estilos ao elemento, você pode usar `element.style` ou `classList.add()`.

5. Trabalhando com Eventos

O que são eventos?

Eventos são **ações ou ocorrências** que acontecem no navegador e que podemos **responder** via JavaScript.

Esses eventos podem ser:

- Um **clique** do mouse.
- Uma **tecla** pressionada.
- Um **formulário enviado**.
- O **carregamento** de uma página.
- A **mudança** de valor em um campo de texto.

Ou seja, eventos são **gatilhos** que disparam **códigos** JavaScript quando o usuário interage com a página.

Como "ouvir" eventos?

Para fazer algo acontecer quando um evento ocorre, usamos principalmente:

```
elemento.addEventListener('evento', função);
```

- **elemento**: É o elemento HTML que vai escutar o evento.
- **'evento'**: Nome do evento (ex: **'click'**, **'mouseover'**, **'keydown'**).
- **função**: Função que vai ser executada quando o evento acontecer.

Exemplos Simples e Didáticos

1. Evento de clique (**click**)

Quando o usuário **clica** em um botão.

```
<button id="meuBotao">Clique aqui</button>

<script>
const botao = document.getElementById('meuBotao');

botao.addEventListener('click', function() {
  alert('Você clicou no botão!');
});
</script>
```

➡ Quando você clicar no botão, um alerta vai aparecer na tela.

2. Evento de passar o mouse (**mouseover**)



Eventos de Mouse mais usados

Evento	Quando acontece
--------	-----------------

Evento	Quando acontece
<code>click</code>	Clicou no elemento
<code>dblclick</code>	Deu dois cliques rápidos (duplo clique)
<code>mousedown</code>	Apertou o botão do mouse (sem soltar)
<code>mouseup</code>	Soltou o botão do mouse
<code>mouseover</code>	Mouse entrou em cima do elemento
<code>mouseout</code>	Mouse saiu de cima do elemento
<code>mousemove</code>	Movendo o mouse sobre o elemento
<code>contextmenu</code>	Clicou com o botão direito (abrir menu)


Exemplos Simples e Didáticos

1. Click Simples (`click`)

```
<button id="botaoClick">Clique aqui</button>

<script>
const botao = document.getElementById('botaoClick');

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

 Um alerta aparece quando o botão é clicado.

2. Duplo Clique (`dblclick`)

```
<div id="caixa" style="width:150px; height:150px; background-
color:lightcoral;">
  Dê dois cliques aqui!
</div>

<script>
const caixa = document.getElementById('caixa');

caixa.addEventListener('dblclick', function() {
  caixa.style.backgroundColor = 'green';
});
```

```
});  
</script>
```

➡ Quando você der dois cliques na caixa, ela muda de cor.

3. Mouse Pressionado (**mousedown**) e Solto (**mouseup**)

```
<div id="caixaMouse" style="width:150px; height:150px; background-  
color:lightblue;">  
  Pressione e solte o mouse  
</div>  
  
<script>  
const caixaMouse = document.getElementById('caixaMouse');  
  
caixaMouse.addEventListener('mousedown', function() {  
  caixaMouse.style.backgroundColor = 'blue';  
});  
  
caixaMouse.addEventListener('mouseup', function() {  
  caixaMouse.style.backgroundColor = 'lightblue';  
});  
</script>
```

➡ Quando você **pressiona o botão do mouse**, a cor muda para azul. Quando você **solta**, volta ao azul claro.

4. Mouse Entrando e Saindo (**mouseover** e **mouseout**)

```
<div id="caixaHover" style="width:150px; height:150px; background-  
color:orange;">  
  Passe o mouse  
</div>  
  
<script>  
const caixaHover = document.getElementById('caixaHover');  
  
caixaHover.addEventListener('mouseover', function() {  
  caixaHover.textContent = 'Mouse em cima!';  
});  
  
caixaHover.addEventListener('mouseout', function() {  
  caixaHover.textContent = 'Passe o mouse';  
});  
</script>
```

➡ O texto da caixa muda quando o mouse passa por cima e volta quando sai.

5. Movendo o Mouse (**mousemove**)

```
<div id="area" style="width:300px; height:300px; border:2px solid black;">
  Mova o mouse aqui
</div>
<p id="posicao"></p>

<script>
const area = document.getElementById('area');
const posicao = document.getElementById('posicao');

area.addEventListener('mousemove', function(event) {
  posicao.textContent = `Posição do mouse: X=${event.offsetX},
Y=${event.offsetY}`;
});
</script>
```

➡ Enquanto você move o mouse dentro da área, aparece a posição do mouse em tempo real.

6. Botão Direito do Mouse (**contextmenu**)

```
<div id="areaDireito" style="width:200px; height:200px; background-color:lightgreen;">
  Clique com o botão direito
</div>

<script>
const areaDireito = document.getElementById('areaDireito');

areaDireito.addEventListener('contextmenu', function(event) {
  event.preventDefault(); // impede abrir o menu padrão do navegador
  alert('Você clicou com o botão direito!');
});
</script>
```

➡ Clicando com o botão direito na div, o navegador **não abre o menu padrão**, e mostra um alerta personalizado.

 **Dica extra: Saber qual botão do mouse foi clicado**

Se você quiser detectar **qual botão** do mouse foi pressionado:

```
<div id="caixaBotao" style="width:150px; height:150px; background-color:lightpink;">
  Clique aqui
</div>

<script>
const caixaBotao = document.getElementById('caixaBotao');

caixaBotao.addEventListener('mousedown', function(event) {
  if (event.button === 0) {
    alert('Botão esquerdo clicado');
  } else if (event.button === 2) {
    alert('Botão direito clicado');
  } else if (event.button === 1) {
    alert('Botão do meio (scroll) clicado');
  }
});
</script>
```

event.button:

- 0 → Botão esquerdo
- 1 → Botão do meio (scroll)
- 2 → Botão direito

Resumo Rápido

Evento	O que Detecta?
click	Clique normal
dblclick	Duplo clique
mousedown	Pressionar botão do mouse
mouseup	Soltar botão do mouse
mouseover	Mouse entrou no elemento
mouseout	Mouse saiu do elemento
mousemove	Movimento do mouse
contextmenu	Clique do botão direito

Vou criar para você um arquivo HTML completo, organizando **todos os principais eventos de mouse** de forma **bem visual e prática** para você testar.

Aqui está o código:

```
<!DOCTYPE html>
<html lang="pt-BR">
<head>
  <meta charset="UTF-8">
  <title>Eventos de Mouse – Exemplo Completo</title>
  <style>
    body {
      font-family: Arial, sans-serif;
      padding: 20px;
    }
    .caixa {
      width: 200px;
      height: 200px;
      margin: 20px;
      display: inline-block;
      text-align: center;
      line-height: 200px;
      font-weight: bold;
      color: white;
      cursor: pointer;
      user-select: none;
    }
    #click { background-color: #3498db; }
    #dblclick { background-color: #e67e22; }
    #mousedown { background-color: #2ecc71; }
    #mouseup { background-color: #9b59b6; }
    #mouseover { background-color: #1abc9c; }
    #mouseout { background-color: #f1c40f; color: black; }
    #mousemove { background-color: #e74c3c; }
    #contextmenu { background-color: #34495e; }
    #info {
      margin-top: 30px;
      font-size: 18px;
      font-weight: bold;
    }
  </style>
</head>
<body>

  <h1>Exemplos de Eventos de Mouse</h1>

  <div id="click" class="caixa">Click</div>
  <div id="dblclick" class="caixa">Double Click</div>
  <div id="mousedown" class="caixa">Mouse Down</div>
  <div id="mouseup" class="caixa">Mouse Up</div>
  <div id="mouseover" class="caixa">Mouse Over</div>
  <div id="mouseout" class="caixa">Mouse Out</div>
  <div id="mousemove" class="caixa">Mouse Move</div>
  <div id="contextmenu" class="caixa">Botão Direito</div>
```

```

<p id="info">Interaja com os quadrados para ver os eventos.</p>

<script>
  const info = document.getElementById('info');

  // Click
  document.getElementById('click').addEventListener('click',
function() {
  info.textContent = 'Você clicou no quadrado azul!';
});

  // Double Click
  document.getElementById('dblclick').addEventListener('dblclick',
function() {
  info.textContent = 'Você deu um duplo clique no quadrado
laranja!';
});

  // Mouse Down
  document.getElementById('mousedown').addEventListener('mousedown',
function() {
  info.textContent = 'Você pressionou o mouse no quadrado verde!';
});

  // Mouse Up
  document.getElementById('mouseup').addEventListener('mouseup',
function() {
  info.textContent = 'Você soltou o botão do mouse no quadrado
roxo!';
});

  // Mouse Over
  document.getElementById('mouseover').addEventListener('mouseover',
function() {
  info.textContent = 'O mouse entrou no quadrado verde água!';
});

  // Mouse Out
  document.getElementById('mouseout').addEventListener('mouseout',
function() {
  info.textContent = 'O mouse saiu do quadrado amarelo!';
});

  // Mouse Move
  document.getElementById('mousemove').addEventListener('mousemove',
function(event) {
  info.textContent = `Movendo o mouse no quadrado vermelho! (X:
${event.offsetX}, Y: ${event.offsetY})`;
});

  // Context Menu (Botão Direito)

document.getElementById('contextmenu').addEventListener('contextmenu',

```

```
function(event) {
    event.preventDefault(); // Impede abrir o menu padrão
    info.textContent = 'Você clicou com o botão direito no quadrado
cinza!';
};
</script>

</body>
</html>
```

✨ O que esse HTML faz:

- Cada quadrado representa um tipo de evento de mouse.
 - Quando você interage (clicar, passar mouse, pressionar, mover, etc.), o texto lá embaixo (**#info**) **muda** explicando o que aconteceu.
 - Super útil para **treinar eventos de mouse** de forma visual!
-

3. Evento de digitação (**keydown** e **keyup**)

1. Evento **keydown** — Quando a tecla é pressionada.

Esse evento é disparado assim que a tecla é pressionada, antes de ser liberada.

Exemplo: Mostrar o código da tecla pressionada

```
<input type="text" id="campoTexto" placeholder="Pressione uma tecla">
<p id="resultado"></p>

<script>
const campo = document.getElementById('campoTexto');
const resultado = document.getElementById('resultado');

campo.addEventListener('keydown', function(evento) {
    resultado.textContent = 'Você pressionou a tecla: ' + evento.key;
});
</script>
```

➡ **Explicação:** Quando o usuário digitar algo no campo de texto, o código da tecla pressionada aparecerá.

2. Evento **keyup** — Quando a tecla é solta.

Esse evento é disparado **quando a tecla é solta** após ser pressionada.

Exemplo: Mostrar a tecla pressionada após soltar

```
<input type="text" id="campoTexto2" placeholder="Digite e solte a tecla">
<p id="resultado2"></p>

<script>
const campo2 = document.getElementById('campoTexto2');
const resultado2 = document.getElementById('resultado2');

campo2.addEventListener('keyup', function(evento) {
  resultado2.textContent = 'Você soltou a tecla: ' + evento.key;
});
</script>
```

➡ **Explicação:** Depois de pressionar e soltar uma tecla, o nome da tecla será mostrado no parágrafo.

3. Evento **keypress** — Quando uma tecla é pressionada e gera um caractere.

Esse evento foi mais usado em versões antigas de JavaScript, mas é interessante saber que ele só detecta a digitação de **caracteres** (não detecta teclas como Shift, Caps Lock, etc.). Ele foi substituído por **keydown** e **keyup** em muitos casos.

Exemplo: Detectando caracteres digitados

```
<input type="text" id="campoTexto3" placeholder="Digite um caractere">
<p id="resultado3"></p>

<script>
const campo3 = document.getElementById('campoTexto3');
const resultado3 = document.getElementById('resultado3');

campo3.addEventListener('keypress', function(evento) {
  resultado3.textContent = 'Caractere digitado: ' + evento.key;
});
</script>
```

➡ **Explicação:** Quando o usuário digitar qualquer caractere, como uma letra ou número, o mesmo será exibido.

4. Detectando combinações de teclas — Exemplo de **Ctrl + C**

Vamos capturar a combinação de teclas pressionadas, como **Ctrl + C**, para mostrar que você pode ouvir várias teclas ao mesmo tempo.

Exemplo: Detectando o pressionamento de **Ctrl + C**

```
<p id="mensagem">Pressione "Ctrl + C" e veja o que acontece</p>

<script>
document.addEventListener('keydown', function(evento) {
  if (evento.ctrlKey && evento.key === 'c') {
    alert('Você pressionou Ctrl + C!');
  }
});
</script>
```

➡ **Explicação:** Quando o usuário pressiona **Ctrl + C** no teclado, um alerta é exibido.

5. Contando o número de caracteres digitados — Ao digitar no campo.

Exemplo: Contagem de caracteres enquanto digita

```
<input type="text" id="campoTexto4" placeholder="Digite algo">
<p id="contador">Caracteres digitados: 0</p>

<script>
const campo4 = document.getElementById('campoTexto4');
const contador = document.getElementById('contador');

campo4.addEventListener('input', function() {
  contador.textContent = 'Caracteres digitados: ' + campo4.value.length;
});
</script>
```

➡ **Explicação:** À medida que o usuário digita, o número de caracteres digitados é exibido no parágrafo. O evento **input** é muito usado quando você quer monitorar a entrada do usuário em tempo real.

6. Alterando o comportamento da tecla pressionada — Impedir a digitação de números.

Podemos usar o evento **keydown** para **prevenir** que o usuário digite números em um campo de texto, permitindo apenas letras.

Exemplo: Impedindo a digitação de números

```
<input type="text" id="campoTexto5" placeholder="Somente letras">
<p id="aviso"></p>
```

```

<script>
const campo5 = document.getElementById('campoTexto5');
const aviso = document.getElementById('aviso');

campo5.addEventListener('keydown', function(evento) {
  if (evento.key >= '0' && evento.key <= '9') {
    evento.preventDefault(); // Impede a digitação de números
    aviso.textContent = 'Números não são permitidos!';
  } else {
    aviso.textContent = '';
  }
});
</script>

```

➡ **Explicação:** Ao pressionar uma tecla numérica, o comportamento padrão é impedido (usando `preventDefault()`), e uma mensagem é exibida.

7. Autocompletar nomes de usuário — Dica dinâmica enquanto digita.

Exemplo: Sugestões dinâmicas enquanto digita o nome

```

<input type="text" id="campoUsuario" placeholder="Digite o nome de usuário">
<ul id="sugestoes"></ul>

<script>
const campoUsuario = document.getElementById('campoUsuario');
const sugestoes = document.getElementById('sugestoes');
const nomes = ['joao', 'jose', 'julia', 'maria', 'marcus'];

campoUsuario.addEventListener('input', function() {
  const filtro = campoUsuario.value.toLowerCase();
  sugestoes.innerHTML = ''; // Limpa as sugestões anteriores

  if (filtro) {
    nomes.filter(nome =>
nome.toLowerCase().includes(filtro)).forEach(function(nome) {
      const item = document.createElement('li');
      item.textContent = nome;
      sugestoes.appendChild(item);
    });
  }
});
</script>

```

➡ **Explicação:** À medida que o usuário digita no campo, ele verá sugestões de nomes com base no que digitou até aquele momento. Isso é uma forma simples de autocompletar.

Resumo dos Eventos de Digitação

- **keydown**: Dispara quando uma tecla é pressionada.
- **keyup**: Dispara quando uma tecla é solta.
- **keypress**: Dispara quando uma tecla é pressionada e gera um caractere (não é mais tão usado em browsers modernos).
- **input**: Captura qualquer alteração no valor de um campo de entrada, como digitar, colar ou apagar.

Esses exemplos cobrem uma variedade de cenários de digitação, desde a simples captura de teclas até interações mais avançadas como contagem de caracteres e autocompletar. Esses eventos são muito úteis para tornar as interfaces mais dinâmicas e responsivas.

Exemplos

- Contagem de caracteres enquanto digita.
- Impedir digitação de números.
- Exibir sugestões dinâmicas com base no texto digitado.
- Mostrar a tecla pressionada.

```
<!DOCTYPE html>
<html lang="pt-br">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Eventos de Digitação</title>
  <style>
    body {
      font-family: Arial, sans-serif;
      margin: 20px;
    }

    #campoTexto {
      margin-bottom: 10px;
      padding: 8px;
      font-size: 16px;
      width: 100%;
    }

    #contador {
      font-size: 14px;
      color: #333;
    }

    #aviso {
      color: red;
      font-size: 14px;
    }
  </style>
</html>
```

```

#sugestoes {
  list-style-type: none;
  padding: 0;
}

#sugestoes li {
  padding: 5px;
  background-color: #f0f0f0;
  margin-top: 5px;
  border-radius: 4px;
}

#sugestoes li:hover {
  background-color: #dcdcdc;
  cursor: pointer;
}
</style>
</head>
<body>

<h1>Formulário de Digitação com Eventos</h1>

<!-- Input para digitar o nome -->
<input type="text" id="campoTexto" placeholder="Digite seu nome" />

<p id="contador">Caracteres digitados: 0</p>
<p id="aviso"></p>

<!-- Sugestões dinâmicas de nomes -->
<p><strong>Sugestões de nomes:</strong></p>
<ul id="sugestoes"></ul>

<!-- Exibir a tecla pressionada -->
<p><strong>Tecla pressionada:</strong> <span id="tecla"></span></p>

<script>
  const campoTexto = document.getElementById('campoTexto');
  const contador = document.getElementById('contador');
  const aviso = document.getElementById('aviso');
  const sugestoes = document.getElementById('sugestoes');
  const teclaDisplay = document.getElementById('tecla');

  const nomes = ['João', 'José', 'Juliana', 'Maria', 'Marcus',
'Carlos', 'Cláudia'];

  // Evento para contar caracteres enquanto digita
  campoTexto.addEventListener('input', function() {
    contador.textContent = 'Caracteres digitados: ' +
campoTexto.value.length;

    // Exibir sugestões dinâmicas de nomes
    const filtro = campoTexto.value.toLowerCase();
    sugestoes.innerHTML = ''; // Limpar sugestões anteriores

```

```

        if (filtro) {
            nomes.filter(nome =>
nome.toLowerCase().includes(filtro)).forEach(function(nome) {
                const item = document.createElement('li');
                item.textContent = nome;
                sugestoes.appendChild(item);
            });
        }
    });

    // Evento para impedir digitação de números
    campoTexto.addEventListener('keydown', function(evento) {
        if (evento.key >= '0' && evento.key <= '9') {
            evento.preventDefault(); // Impede digitar números
            aviso.textContent = 'Números não são permitidos!';
        } else {
            aviso.textContent = ''; // Limpa a mensagem
        }

        // Exibir a tecla pressionada
        teclaDisplay.textContent = evento.key;
    });

    // Evento de digitação - mostrar tecla pressionada
    campoTexto.addEventListener('keyup', function(evento) {
        teclaDisplay.textContent = evento.key; // Exibe a tecla que foi
pressionada
    });
</script>

</body>
</html>

```

Explicação do código:

1. Contagem de caracteres digitados:

- O evento `input` é usado para monitorar qualquer alteração no campo de texto. Toda vez que o usuário digita, a contagem de caracteres é atualizada dinamicamente.

2. Impedindo a digitação de números:

- O evento `keydown` é usado para detectar quando o usuário pressiona uma tecla. Se a tecla pressionada for um número (0 a 9), usamos `preventDefault()` para impedir a digitação e exibimos uma mensagem de aviso.

3. Sugestões dinâmicas enquanto digita:

- Enquanto o usuário digita, as sugestões de nomes são filtradas com base no que foi digitado no campo de texto. Isso é feito de maneira simples, comparando o texto digitado

com uma lista de nomes predefinidos.

4. Exibindo a tecla pressionada:

- Usamos o evento **keyup** para mostrar a tecla que foi pressionada. O valor da tecla é exibido em tempo real na tela.

Como funciona no navegador:

1. Digite qualquer texto no campo de entrada:

- A contagem de caracteres será atualizada conforme você digita.
- Se digitar números, uma mensagem de erro será exibida.

2. Sugestões de nomes aparecem abaixo do campo de texto conforme você digita algo que se assemelha a um nome na lista de sugestões.

3. Teclas pressionadas aparecem em tempo real embaixo do campo de texto, tanto enquanto você digita (**keydown**) quanto quando solta a tecla (**keyup**).

Esse exemplo abrange diversos eventos de digitação comuns em JavaScript, tornando a interação com o usuário mais rica e dinâmica. Pode ser facilmente adaptado e expandido para diferentes cenários em formulários ou interfaces de entrada de dados.

4. Evento de envio de formulário (**submit**)

No contexto do DOM (Document Object Model) e do desenvolvimento web, um **evento de envio de formulário** é acionado quando o usuário tenta **submeter** um formulário na página. Esse evento ocorre quando o botão de envio (tipicamente `<button type="submit">`) é clicado ou quando o usuário pressiona a tecla **Enter** em um campo de entrada de texto dentro do formulário.

Objetivo do Evento de Envio

O evento de envio de formulário permite que o desenvolvedor **intercepte a ação de envio** do formulário antes que ela seja completada. Isso é útil para:

- **Validar dados do formulário:** Verificar se os campos obrigatórios foram preenchidos corretamente.
- **Evitar o envio de dados incorretos:** Garantir que o formulário não seja enviado com dados inválidos.
- **Enviar os dados via Ajax:** Realizar uma requisição assíncrona ao servidor sem precisar recarregar a página.
- **Adicionar lógica de negócio:** Executar funções específicas antes do envio, como contagem de campos, formatação de dados ou qualquer outra ação programática.

Como Funciona o Evento de Envio?

O evento de envio (**submit**) é acionado quando o formulário é **submetido**. Podemos associar esse evento a uma função JavaScript que será executada quando o usuário tentar enviar o formulário.

Método do Formulário:

- **submit()**: Este método pode ser chamado para enviar o formulário programaticamente.
- **event.preventDefault()**: Em conjunto com o evento **submit**, usamos esse método para **impedir o envio padrão** do formulário, permitindo que o desenvolvedor realize validações ou outras ações antes do envio real.

Exemplo 1: Envio Simples de Formulário

Aqui temos um exemplo básico de um formulário simples, que ao ser enviado, exibe uma mensagem de alerta e impede o envio padrão do formulário.

HTML:

```
<!DOCTYPE html>
<html lang="pt-br">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <title>Exemplo de Envio de Formulário</title>
</head>
<body>

  <h2>Formulário de Contato</h2>
  <form id="formulario" action="processar_formulario.php"
method="post">
    <label for="nome">Nome:</label>
    <input type="text" id="nome" name="nome" required><br><br>

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

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

  <script src="script.js"></script>
</body>
</html>
```

JavaScript (em **script.js**):

```
// Selecionando o formulário
const formulario = document.getElementById('formulario');
```



```
// Adicionando o evento de envio
formulario.addEventListener('submit', function(event) {
    // Impedindo o envio padrão do formulário
    event.preventDefault();

    // Exibindo uma mensagem de alerta
    alert('O formulário foi enviado!');

    // Aqui, você pode realizar outras ações antes do envio, como
    // validações, Ajax, etc.
});
```

O que acontece no exemplo acima?

- O evento `submit` é acionado quando o botão de envio é clicado.
- A função associada ao evento intercepta o envio com o método `event.preventDefault()`, impedindo que o formulário seja enviado para o servidor imediatamente.
- Uma mensagem de alerta é exibida, indicando que o evento de envio foi capturado.

Este é um exemplo básico, mas já demonstra o controle total sobre o envio de formulários.

Exemplo 2: Validação de Dados Antes de Enviar

Agora, vamos usar o evento de envio para **validar os dados** do formulário antes de permitir que ele seja enviado.

HTML:

```
<!DOCTYPE html>
<html lang="pt-br">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Validação de Formulário</title>
</head>
<body>

    <h2>Formulário de Inscrição</h2>
    <form id="formulario">
        <label for="nome">Nome:</label>
        <input type="text" id="nome" name="nome"><br><br>

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

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

```
<script src="script.js"></script>
</body>
</html>
```

JavaScript (em `script.js`):

```
const formulario = document.getElementById('formulario');

// Adicionando o evento de envio
formulario.addEventListener('submit', function(event) {
  // Capturando os valores dos campos
  const nome = document.getElementById('nome').value;
  const email = document.getElementById('email').value;

  // Verificando se os campos estão vazios
  if (!nome || !email) {
    alert('Por favor, preencha todos os campos.');
```

event.preventDefault(); // Impede o envio do formulário

```
  } else {
    alert('Formulário enviado com sucesso!');
  }
});
```

O que acontece no exemplo acima?

- A função associada ao evento `submit` é chamada quando o formulário é enviado.
- A função verifica se os campos de nome e email estão preenchidos. Se algum campo estiver vazio, o envio é **impedido** com `event.preventDefault()`, e uma mensagem de alerta é mostrada ao usuário.
- Se ambos os campos estiverem preenchidos, o formulário é "enviado" (ou seja, a ação padrão de envio ocorre, mas pode ser substituída por lógica adicional como uma requisição Ajax).

Exemplo 3: Envio de Formulário com Ajax

Por fim, podemos usar o evento `submit` para enviar o formulário sem recarregar a página, utilizando **Ajax** (ou Fetch API). Esse exemplo permite enviar os dados para o servidor sem atualizar a página, proporcionando uma experiência de usuário mais fluida.

HTML:

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

```

scale=1.0">
  <title>Envio de Formulário com Ajax</title>
</head>
<body>

  <h2>Formulário de Feedback</h2>
  <form id="formulario">
    <label for="feedback">Seu Feedback:</label><br>
    <textarea id="feedback" name="feedback" rows="4" cols="50">
  </textarea><br><br>

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

  <div id="mensagem"></div>

  <script src="script.js"></script>
</body>
</html>

```

JavaScript (em **script.js**):

```

const formulario = document.getElementById('formulario');

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

  const feedback = document.getElementById('feedback').value;

  if (!feedback) {
    alert('Por favor, insira seu feedback!');
    return;
  }

  // Enviando dados via Fetch API (Ajax)
  fetch('processar_feedback.php', {
    method: 'POST',
    body: JSON.stringify({ feedback: feedback }),
    headers: {
      'Content-Type': 'application/json'
    }
  })
  .then(response => response.json())
  .then(data => {
    document.getElementById('mensagem').textContent = data.mensagem;
  })
  .catch(error => {
    alert('Erro ao enviar feedback.');
```

O que acontece no exemplo acima?

- O evento `submit` do formulário é interceptado.
- Os dados do formulário (feedback) são coletados.
- Usamos **Fetch API** para enviar os dados ao servidor em formato JSON, sem recarregar a página.
- O servidor processa os dados e retorna uma resposta que é exibida na página, na div `#mensagem`.

Conclusão

O **evento de envio de formulário** (`submit`) é uma parte essencial na criação de formulários dinâmicos e interativos em páginas web. Ele permite validar dados, realizar ações antes do envio, e enviar dados ao servidor sem a necessidade de recarregar a página (por exemplo, com Ajax).

Com a manipulação desse evento, você pode:

1. Validar os dados antes de enviá-los.
2. Impedir o envio do formulário caso os dados não sejam válidos.
3. Enviar os dados via Ajax para um servidor sem recarregar a página.

Essa abordagem ajuda a criar formulários mais rápidos e eficientes, melhorando a experiência do usuário.

5. Evento de mudança de valor (`change`)

Detecta quando o valor de um campo muda.

```
<select id="opcoes">
  <option value="1">Opção 1</option>
  <option value="2">Opção 2</option>
</select>

<script>
const select = document.getElementById('opcoes');

select.addEventListener('change', function() {
  alert('Você escolheu: ' + select.value);
});
</script>
```

➡ Quando você muda a opção selecionada, aparece um alerta mostrando o valor.

Resumo Visual:

Evento	Quando acontece	Exemplo
--------	-----------------	---------

Evento	Quando acontece	Exemplo
<code>click</code>	Quando um elemento é clicado	Clicar em botão
<code>mouseover</code>	Quando o mouse passa por cima	Passar mouse sobre uma div
<code>keydown</code>	Quando uma tecla é pressionada	Digitar em um campo
<code>submit</code>	Quando um formulário é enviado	Clicar em "enviar"
<code>change</code>	Quando um valor é alterado	Selecionar outra opção

Algumas dicas importantes:

- Sempre tente usar `addEventListener` (em vez de colocar direto no HTML).
- Você pode remover eventos com `removeEventListener` se quiser parar de ouvir.
- O objeto `event` (às vezes chamado de `e` ou `evento`) traz detalhes sobre o que aconteceu, como qual tecla foi pressionada, ou qual botão foi clicado.

6. Manipulando Formulários

Objetivo: O DOM oferece formas práticas de **interagir e manipular** os elementos de formulários, como campos de texto, selects e botões, permitindo **validar dados**, **capturar valores** e até **enviar os dados** sem recarregar a página.

Como fazer:

- `value`: Captura ou altera o valor de campos de formulário como `input`, `textarea`, `select`.
- `submit()`: Envia o formulário programaticamente.
- `reset()`: Restaura os valores padrão dos campos.

Exemplo:

```
// Capturando o valor de um campo de input
const inputNome = document.getElementById('nome');
console.log(inputNome.value);

// Validando se o campo não está vazio antes de enviar o formulário
const formulario = document.getElementById('formulario');
formulario.addEventListener('submit', function(event) {
  if (inputNome.value === '') {
    alert('Nome não pode estar vazio!');
    event.preventDefault(); // Impede o envio do formulário
  }
});
```

Resumo das Etapas:

Etapa	O que faz	Método/Propriedade Principal
Selecionando Elementos	Seleciona elementos da página para manipulação	<code>getElementById()</code> , <code>querySelector()</code>
Manipulando o Conteúdo	Modifica o texto ou HTML dentro de elementos	<code>textContent</code> , <code>innerHTML</code> , <code>value</code>
Alterando Estilos	Altera estilos diretamente no elemento ou adiciona/remover classes CSS	<code>style</code> , <code>classList.add()</code> , <code>classList.remove()</code>
Criando e Removendo Elementos	Cria novos elementos ou remove existentes	<code>createElement()</code> , <code>appendChild()</code> , <code>removeChild()</code>
Trabalhando com Eventos	Adiciona eventos de interação do usuário à página	<code>addEventListener()</code> , <code>removeEventListener()</code>
Manipulando Formulários	Interage com campos de formulários, valida dados e envia sem recarregar	<code>value</code> , <code>submit()</code> , <code>reset()</code>

Essas etapas são fundamentais para criar interfaces dinâmicas e interativas em aplicações web modernas. Cada uma delas tem seu papel específico, mas elas geralmente são usadas juntas para criar a experiência de usuário fluida e responsiva.

O DOM é **fundamental** no desenvolvimento de aplicações web dinâmicas e interativas. Ele oferece a **flexibilidade necessária** para criar interfaces de usuário ricas, responder rapidamente às ações dos usuários e atualizar o conteúdo da página de forma fluida, sem precisar recarregar a página. Em termos práticos, no **dia a dia do desenvolvimento**, o DOM é usado para:

- Criar e manipular elementos dinamicamente.
- Alterar o conteúdo e estilo de elementos.
- Trabalhar com eventos e interações do usuário.
- Manipular dados de formulários e inputs.

Manipulação do DOM

Usando JavaScript, você pode **acessar** e **manipular** o DOM.

Exemplos comuns:

- **Selecionar** elementos:

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

- **Alterar** texto:

```
titulo.textContent = 'Novo título!';
```

- **Criar** novos elementos:

```
const novoParagrafo = document.createElement('p');  
novoParagrafo.textContent = 'Outro parágrafo!';  
document.body.appendChild(novoParagrafo);
```

- **Modificar atributos:**

```
titulo.setAttribute('class', 'titulo-principal');
```

- **Remover elementos:**

```
titulo.remove();
```

DOM e Eventos

Além de alterar a estrutura da página, você pode **ouvir** eventos no DOM, como cliques, teclado, etc.

Exemplo:

```
const botao = document.querySelector('button');  
botao.addEventListener('click', function() {  
  alert('Você clicou no botão!');  
});
```

Isso permite criar **interatividade** entre o usuário e a página.

Algumas observações importantes:

- O DOM é uma **representação viva** da página: se você alterar algo no DOM via JavaScript, a alteração aparece **imediatamente** na tela.
- DOM não é exclusivo de HTML. Também pode representar documentos XML.
- A performance de manipulação do DOM pode impactar o desempenho de sites, especialmente se houver **muitas alterações** em elementos grandes.

Resumindo:

Conceito	Resumo Rápido
O que é DOM	Modelo em árvore da página web em objetos
Função principal	Permitir que programas leiam e modifiquem a estrutura do documento
Baseado em	HTML ou XML
Manipulação principal	Feita com JavaScript
Exemplos de uso	Alterar textos, adicionar elementos, ouvir eventos

Anatomia de um Elemento no DOM

Cada **elemento** no DOM (por exemplo, um `<p>`, `<div>`, ``) tem várias **propriedades** e **métodos**.

Exemplo:

Imagine que temos no HTML:

```
<p id="meuParagrafo" class="texto">Olá, mundo!</p>
```

Quando acessamos no JavaScript:

```
const p = document.getElementById('meuParagrafo');
```

O `p` é um objeto que possui:

- `id`: "meuParagrafo"
- `className`: "texto"
- `innerHTML`: "Olá, mundo!" (inclui HTML interno se existir)
- `textContent`: "Olá, mundo!" (apenas o texto)
- `style`: acesso ao estilo CSS inline (`p.style.color = "red"`)
- `children`: lista de filhos se tiver
- `parentElement`: o elemento pai (`body` nesse caso)
- Métodos como `.appendChild()`, `.remove()`, `.cloneNode()`, etc.

Cada elemento é super rico em informações e funcionalidades.

API do DOM

API (Application Programming Interface) do DOM é o **conjunto de objetos, métodos e eventos** que você usa para manipular documentos.

Principais APIs do DOM:

API	Função
Document API	Manipular o documento (acessar elementos, criar novos, etc.)
Element API	Manipular elementos específicos (atributos, filhos, etc.)
Events API	Capturar e tratar eventos como clique, teclado, mouse
Traversal API	Navegar entre nós (pais, filhos, irmãos)
MutationObserver API	Detectar mudanças no DOM automaticamente

Exemplo de Traversal API:

```
const div = document.querySelector('div');

console.log(div.parentElement); // Elemento pai
console.log(div.children);      // Lista de elementos filhos
console.log(div.nextElementSibling); // Próximo elemento irmão
console.log(div.previousElementSibling); // Elemento irmão anterior
```

DOM vs HTML Estático

Diferença importante:


HTML Estático	DOM Dinâmico
Fixo no carregamento	Pode mudar com JavaScript
Só leitura visual	Acesso completo a objetos
Não interativo	Eventos e animações

Então quando você **modifica o DOM**, **não está alterando o arquivo HTML original**, mas sim a **representação carregada pelo navegador**.

Eventos e Propagação no DOM

Quando ocorre um evento (tipo **click**), ele pode seguir três fases:

1. **Capturing Phase**: o evento vai do **document** até o alvo.
2. **Target Phase**: o evento chega no elemento que foi clicado.
3. **Bubbling Phase**: o evento "sobe" de volta pelo DOM.

 **Bubbling** é o comportamento padrão: eventos "borbulham" de dentro para fora.

Exemplo:

Se você clicar em um `<button>` dentro de um `<div>`, o evento será capturado primeiro no botão, depois "subirá" até o `div`, e depois até o `body`, e assim por diante.

Capturando evento no bubbling:

```
document.querySelector('div').addEventListener('click', function() {  
  console.log('Div clicada!');  
});
```

DOM Virtual

Você pode ouvir falar de "**Virtual DOM**" em frameworks como **React**, **Vue**, etc.

O que é o Virtual DOM?

- É uma **cópia** do DOM real, mantida na memória.
- Quando você faz alterações, o framework **compara** o Virtual DOM com o DOM atual (isso se chama **diffing**).
- Depois, ele atualiza **apenas o que mudou** no DOM real, evitando alterações custosas.

● Resultado: **Sites mais rápidos e responsivos!**

Desafios do DOM

Apesar de ser super poderoso, manipular o DOM **direto** (sem frameworks) tem desafios:

- **Complexidade**: Em páginas grandes, navegar e manipular o DOM pode ser difícil.
- **Performance**: Muitas alterações rápidas podem deixar a página lenta.
- **Cross-browser**: Pequenas diferenças entre navegadores precisam ser tratadas.
- **Estado**: Gerenciar o que está na tela (estado da UI) pode ficar bagunçado.

Por isso, frameworks modernos **abstraem** essa complexidade para nós!

Um resuminho visual para fechar 🧠:

```
[HTML] --> [Navegador lê] --> [Cria o DOM em árvore] --> [JavaScript  
acessa o DOM] --> [DOM altera a página em tempo real]
```

Se quiser, eu também posso te mostrar **exemplos mais avançados**, tipo:

- Criar uma **lista dinâmica** com input de usuário;
- Criar **animações** manipulando o DOM;
- Fazer **delegação de eventos** (evento para múltiplos elementos);