Instituto Telles

Material do Aluno

Técnico em Desenvolvimento *Web* e Cibersegurança

Front-end: princípios

JavaScript

Instituto Telles

Parcerias:

SEDUC
Secretaria de Estado da Educação

SECTI
Secretaria de Estado de Ciência, Tecnologia e Inovação

GOVERNO DE GOIÁS
O ESTADO QUE DÁ CERTO

Sumário

Front-end: princípios

INTRODUÇÃO

O desenvolvimento *front-end* desempenha um papel crucial na experiência do usuário, já que é através dele que é feita a ponte entre a interface gráfica e o usuário, facilitando a interação com *websites* e aplicações *web*. Junto a isso, os princípios JavaScript são fundamentais, já que, geralmente, essa é a linguagem de programação utilizada no desenvolvimento *front-end*, permitindo a criação de interfaces interativas e dinâmicas.

No contexto do *front-end*, os princípios JavaScript abrangem uma variedade de conceitos e técnicas que visam otimizar o desempenho, a legibilidade e a manutenibilidade do código. Desde a manipulação do DOM (*Document Object Model* ou, em português, Modelo de Objeto de Documento) até o gerenciamento de eventos.

Além disso, a modularidade e a organização do código são aspectos-chave para garantir um desenvolvimento *front-end* sustentável. A utilização de padrões de projeto, como o MVC (*Model-View-Controller*) e o MVVM (*Model-View-ViewModel*), contribui para a estruturação eficiente das aplicações, facilitando a escalabilidade e a colaboração entre desenvolvedores.

Neste contexto, explorar os princípios JavaScript não apenas fortalece a base técnica dos desenvolvedores *front-end*, mas também estimula a criação de experiências de usuário mais fluidas e atrativas. Este material abordará esses princípios, ressaltando a sua importância e proporcionando perspectivas sobre como implementá-los para aprimorar a qualidade e a eficiência do desenvolvimento *front-end*.

CAPÍTULO 01

EXPLORANDO O JAVASCRIPT

O que esperar deste capítulo:

- Compreender os conceitos fundamentais da linguagem JavaScript;
- Entender o conceito de assincronia e como lidar com operações assíncronas em JavaScript;
- Utilizar as principais bibliotecas e *frameworks* JavaScript no desenvolvimento *front-end*.

1.1 O que é JavaScript?

Fonte: https://encurtador.com.br/dfhAI. Acesso em: 05 fev. 2024.

JavaScript é uma linguagem de programação de alto nível, interpretada e orientada a objetos, amplamente utilizada no desenvolvimento *web* para tornar as páginas mais interativas e dinâmicas. Um exemplo comum da aplicação dessa linguagem no desenvolvimento *front-end* é a **validação de formulários em páginas *web***. Suponha que você tem um formulário de registro em um *site*. Você pode usar JavaScript para validar se os campos foram preenchidos corretamente pelo usuário antes de enviar os dados para o servidor. Por exemplo, caso algum campo esteja vazio, uma mensagem de erro poderá ser exibida.

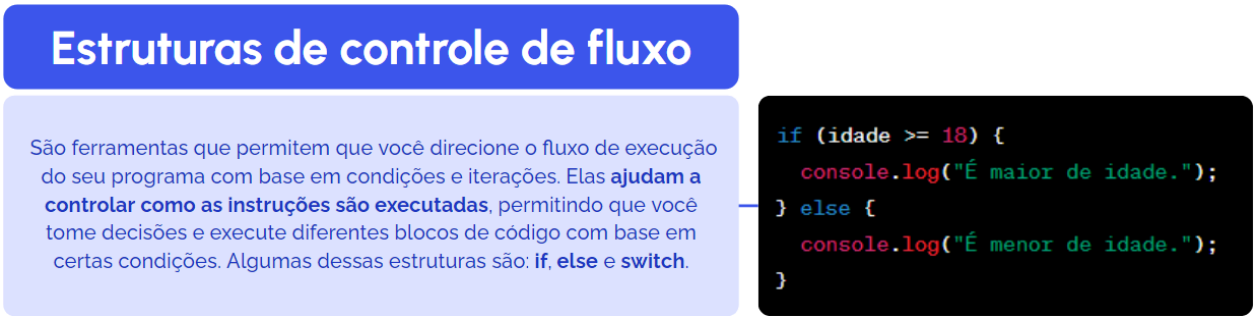
Apesar desse ter sido apenas um exemplo básico de como a linguagem JavaScript pode ser aplicada durante o desenvolvimento *web*, nós já conseguimos perceber seu papel

crucial na evolução da *web*, mostrando que essa linguagem transformou a experiência do usuário, permitindo a criação de aplicações mais ricas e responsivas. Aqui, nós vamos explorar em detalhes as características e aplicações do JavaScript e porque essa linguagem desempenha um papel vital na construção de *sites* modernos.

JavaScript é uma **linguagem de *script***, isso significa dizer que ela foi **projetada para a automação de tarefas e a interação com outros *softwares***, sendo originalmente desenvolvida para ser executada no navegador. Essa linguagem permite que os desenvolvedores adicionem funcionalidades dinâmicas às páginas *web*, interagindo com o usuário, manipulando o conteúdo da página e muito mais. Diferentemente de linguagens como HTML e CSS, JavaScript é uma linguagem de programação completa, proporcionando **controle lógico e estruturas de dados**.

1.1.1 Sintaxe básica: fundamentos para iniciantes

O primeiro passo para entender a linguagem JavaScript é familiarizar-se com sua sintaxe, que são as regras que definem como as instruções em uma linguagem de programação específica devem ser formatadas e organizadas para que sejam compreendidas e executadas pelo computador de forma correta. Vamos começar conhecendo três conceitos fundamentais: as **variáveis**, os **tipos de dados** e as **estruturas de controle de fluxo**.



1.1.2 Manipulação do DOM: interatividade em ação

Em JavaScript, o DOM (*Document Object Model*) é a **representação hierárquica dos elementos HTML de uma página**. Ele é como uma representação organizada e estruturada de todos os elementos na página *web* que você está criando. Nessa linguagem, a **manipulação do DOM** refere-se à capacidade de **interagir e modificar os elementos HTML** de uma página *web* dinamicamente. Ao usar JavaScript para interagir com o DOM, você está, basicamente, dando instruções para modificar ou manipular elementos na sua página *web*. Por exemplo, você pode dizer ao JavaScript para mudar a cor de um botão quando ele é clicado ou para adicionar um novo parágrafo ao seu documento quando uma condição é atendida. Tudo isso te permite mexer na estrutura para criar páginas interativas e dinâmicas.

A seguir, vamos conhecer alguns métodos e propriedades que a linguagem JavaScript fornece para manipular essa estrutura.

Seleção de elementos

O JavaScript pode ser usado para **selecionar e manipular elementos HTML**. Isso pode ser feito usando métodos como **getElementById**.

```
let elemento = document.getElementById("meuElemento");
```

Manipulação de eventos

Com esse método é possível **tornar a página reativa à ações do usuário**, como cliques ou teclas pressionadas. Você pode usar métodos como **addEventListener** para associar funções a eventos.

```
elemento.addEventListener("click", function() {
  alert("Clicou no elemento!");
});
```

Fazer a manipulação do DOM é, com certeza, uma habilidade muito importante para você que está começando a estudar programação. Selecionar e modificar diferentes partes do documento, como alterar textos, estilos e, até mesmo, responder a ações do usuário, abre

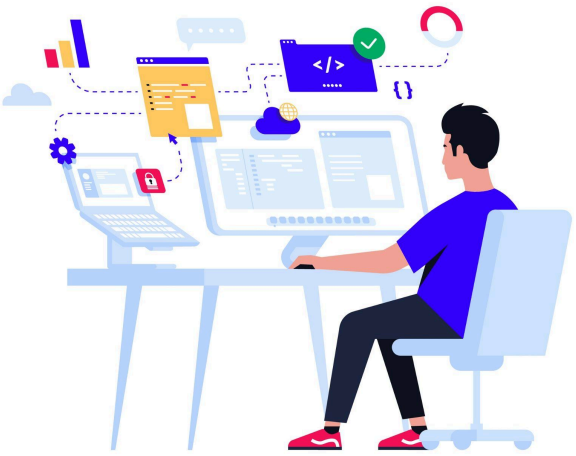
portas para a criação de páginas *web* mais interativas e personalizadas e ter esse conhecimento irá capacitá-lo a construir experiências *on-line* mais envolventes e funcionais.

1.1.3 Assincronia e requisições AJAX: dinamismo em tempo real

Em JavaScript, a **assincronia** refere-se à capacidade de **executar operações independentes sem bloquear a execução do restante do código**. Em outras palavras, enquanto uma operação está em andamento, o programa pode continuar executando outras tarefas em segundo plano. Isso é especialmente útil em situações em que algumas operações podem levar tempo, como carregar dados de uma fonte externa.

Ao trabalhar com assincronia, utilizamos o **async/await**, sintaxe que **simplifica o trabalho com código assíncrono**. Essa sintaxe é baseada em **promises** e permite escrever código assíncrono de forma mais síncrona, tornando-o mais legível e fácil de entender.

Promises são objetos que representam o **resultado eventual de uma operação assíncrona**, como a conclusão ou a falha de uma operação. Esses objetos permitem o encadeamento de ações a serem realizadas quando a operação for concluída.



Disponível em: https://encurtador.com.br/ekm24. Acesso em: 06 fev. 2024.

Já as **requisições AJAX** (*Asynchronous JavaScript and XML*) são uma maneira de **realizar operações assíncronas em JavaScript**, permitindo que você atualize partes específicas de uma página *web* sem precisar recarregar a página inteira. Atualmente, apesar do nome "XML" na sigla, o formato mais comum para transmitir dados é o JSON.

As **promises** e a sintaxe **async/await** estão relacionadas às requisições AJAX por meio da **manipulação assíncrona de operações** e isso se deve às características e aplicações desses dois conceitos, já que as **promises** são objetos que representam o resultado de uma operação assíncrona, enquanto o **async/await** é uma sintaxe mais amigável para se trabalhar com elas.

```
function fetchData() {
  return new Promise((resolve, reject) => {
    // Lógica para buscar dados
    if (sucesso) {
      resolve(dados);
    } else {
      reject("Erro ao buscar dados");
    }
  });
}

async function processarDados() {
  try {
    const dados = await fetchData();
    console.log("Dados processados:", dados);
  } catch (erro) {
    console.error("Erro:", erro);
  }
}
```

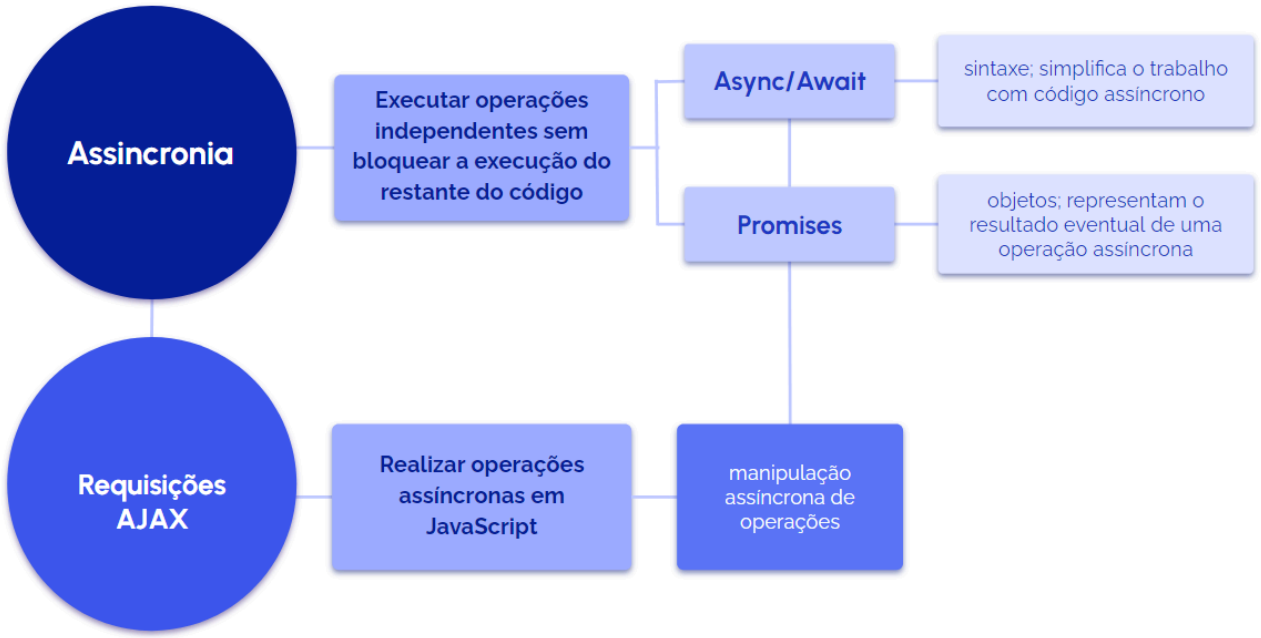
Exemplo da aplicação de **promises** e **async/await** em um código, utilizados para lidar com operações assíncronas de forma mais eficiente.

```
let xhr = new XMLHttpRequest();
xhr.open("GET", "dados.json", true);
xhr.onreadystatechange = function() {
  if (xhr.readyState == 4 && xhr.status == 200) {
    let dados = JSON.parse(xhr.responseText);
    console.log("Dados recebidos:", dados);
  }
};
xhr.send();
```

Exemplo da aplicação de requisições AJAX, usada para atualizar partes específicas da página sem recarregá-la.

Ao trabalhar com as requisições AJAX, você pode usar **promises** para lidar com o assincronismo e o **async/await** para simplificar ainda mais o código. A combinação de assincronia e requisições AJAX é fundamental para o desenvolvimento de aplicações *web* modernas, em que a interação com o servidor pode ocorrer de forma eficiente e sem interromper a interatividade do usuário.

Agora, fique com um mapa mental que sintetiza o que estudamos até aqui.



1.1.4 *Frameworks* e bibliotecas: potencializando o desenvolvimento

Além dos conceitos básicos, os *frameworks* e as bibliotecas JavaScript modernos, como **React**, **Angular** e **Vue.js,** têm revolucionado o desenvolvimento *web*. Essas ferramentas facilitam a criação de interfaces de usuário complexas, mantendo a escalabilidade do código. Veja mais sobre eles a seguir.

- **React**
  - **Componentização e Virtual DOM:**
    - facilita a criação de interfaces modulares e reutilizáveis;
    - utiliza um DOM virtual para otimizar a atualização da interface.

```
import React from 'react';

function MeuComponente(props) {
  return <div>{props.mensagem}</div>;
}
```

- **Angular**
  - **Orientação a objetos**:
    - é um ***framework* completo** para desenvolvimento *front-end*;
    - segue uma arquitetura similar ao **MVC** (*Model-View-Controller*) e é fundamentado em **orientação a objetos;**
    - utiliza o ***Shadow* DOM** para encapsular componentes e oferecer reusabilidade e manutenção simplificada.
- **Vue.js**
  - **Meio-termo**:
    - é um ***framework* progressivo**, o que significa que você pode usá-lo gradualmente em partes do seu projeto;
    - oferece uma sintaxe simples e intuitiva, tornando-o fácil de aprender e usar;
    - também utiliza o **Virtual DOM**, proporcionando eficiência em atualizações e um bom desempenho geral.

Frameworks e bibliotecas são essenciais no desenvolvimento *web* moderno, pois fornecem ferramentas poderosas para simplificar tarefas comuns e melhorar a eficiência do desenvolvimento. Muitas vezes, a escolha entre um *framework* e uma biblioteca depende dos requisitos específicos do projeto e das preferências da equipe de desenvolvimento.

1.2 Sintaxe básica em JavaScript: variáveis, tipos de dados e operadores

Quando falamos sobre sintaxe básica em JavaScript, estamos nos referindo às **regras e estruturas** fundamentais que precisamos seguir ao escrever código nesta linguagem de programação. É como as regras de gramática que seguimos ao escrever uma redação ou uma carta. Ou seja, seguir essas regras permite que você escreva um código funcional e compreensível, que poderá ser entendido por toda a comunidade de desenvolvedores.

Para que você consiga seguir corretamente essas regras, é necessário conhecer alguns conceitos importantes, como **variáveis**, **tipos de dados** e **operadores**. Vamos lá?

1.2.1 Variáveis e o uso de palavras-chave: armazenando informações

As variáveis são essenciais para qualquer linguagem de programação. Neste universo, elas são usadas para **armazenar e manipular dados** e funcionam como caixas em que podemos guardar informações importantes. Imagine que você está criando um jogo e precisa lembrar a pontuação do jogador. Aqui é onde as variáveis entram em cena!

Quando escrevemos um programa em JavaScript, nós utilizamos as variáveis para armazenar dados. Para que seja fácil de entender o que essas "caixas" estão guardando, essas variáveis são nomeadas, como "**pontuacaoJogador**". Se o jogador marcou dez pontos no jogo, é possível atualizar essa variável para dez. Depois, se ele marcar mais cinco pontos, é só aumentar esse número! Viu como é simples?

Disponível em: https://encurtador.com.br/bZ249. Acesso em: 08 fev. 2024.

Em JavaScript, você pode declarar as variáveis usando **palavras-chave**, que desempenham um papel importante na criação, declaração e manipulação de variáveis, já que cada uma tem um significado específico associado a ela. Algumas das principais palavras-chave utilizadas em JavaScript são:

- **var:** declara variáveis globalmente ou localmente. Porém, o seu escopo pode ser alterado inesperadamente, como a pontuação do nosso jogador, lembra?

  Exemplo:

  ```
  var pontuacaoJogador = 10;
  ```

- **let:** introduz a variável localmente em um bloco de código e permite reatribuição. Por exemplo, você poderia utilizar esta variável para definir o nome do jogador do seu *game*. Exemplo:

  ```
  let nomeJogador = "Alice";
  nomeJogador = "Bob"; // Reatribuição permitida
  ```

- **const:** declara uma constante cujo **valor não pode ser reatribuído**. Ou seja, variáveis declaradas com **var** e **let** podem ser reatribuídas, enquanto as que foram declaradas com **const** são imutáveis. Isso significa que o seu valor não pode ser alterado após a atribuição inicial.

  Exemplo:

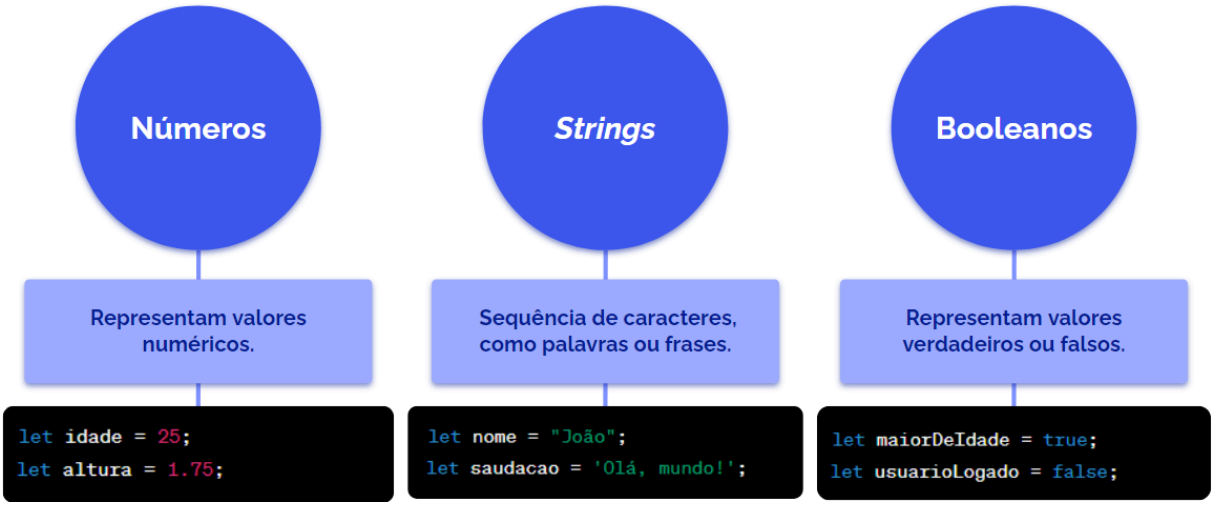
```
const PI = 3.14;


// Tentativa de reatribuição (gerará um erro)
// PI = 3.14159;


console.log("O valor de PI é:", PI);
```

  resultado: O valor de PI é: 3.14

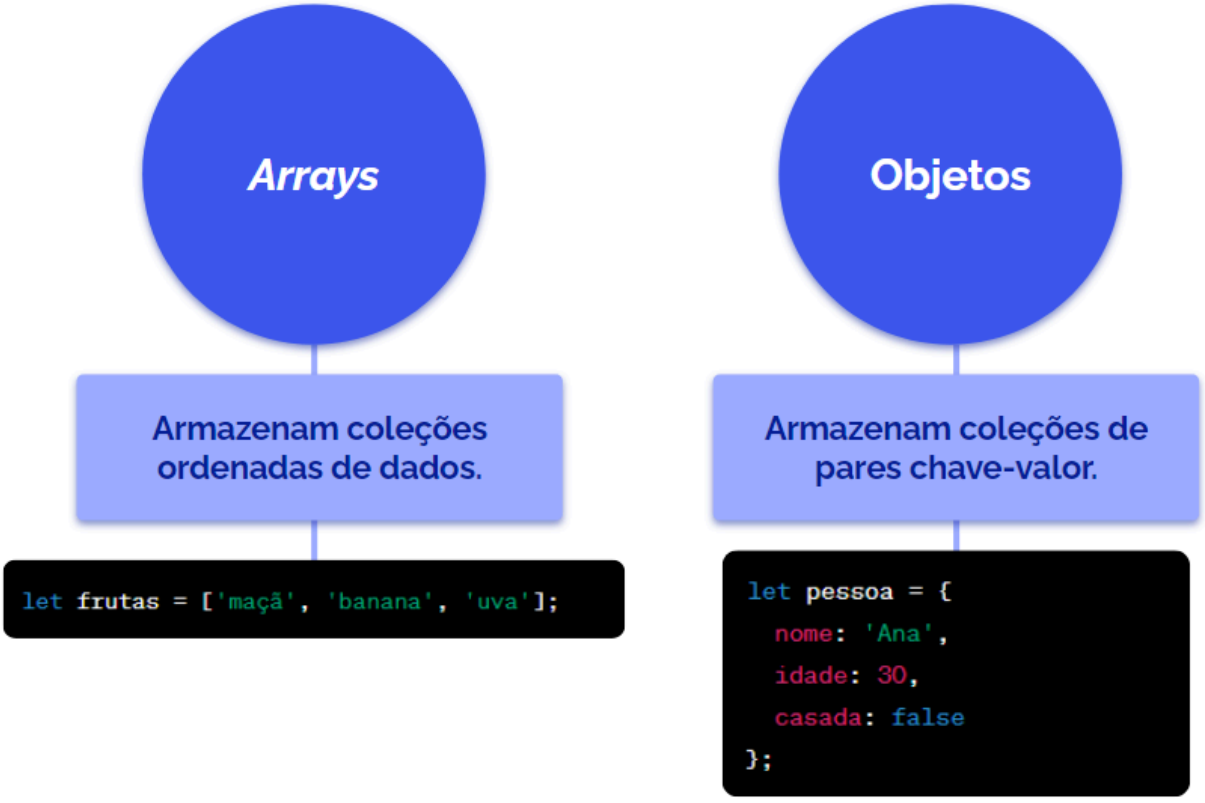
As palavras-chave desempenham um papel crucial na estruturação e execução do código. Elas são utilizadas para definir a sintaxe, a estrutura e o comportamento do código. O uso correto das variáveis e das palavras-chave é essencial para garantir que o código seja claro, semântico e que seja executado conforme o esperado. Por isso, conhecer esses conceitos é fundamental para o seu aprendizado e desenvolvimento das suas habilidades ao utilizar a linguagem JavaScript.

1.2.2 Tipos de dados: compreendendo a variedade

Ao falar sobre tipos de dados em JavaScript, estamos nos referindo aos diferentes tipos de informações que podemos manipular em nossos programas. Assim como na nossa vida cotidiana, na qual encontramos diferentes tipos de coisas ao nosso redor (como números e palavras), o mesmo acontece quando se trata da linguagem JavaScript. Por isso, temos tipos de dados específicos para cada tipo de informação. Acompanhe o esquema a seguir.



Apesar desses serem os tipos de dados mais básicos em JavaScript, também existem outros tipos mais complexos, como *arrays* e objetos, que permitem organizar e manipular informações de maneira mais estruturada.



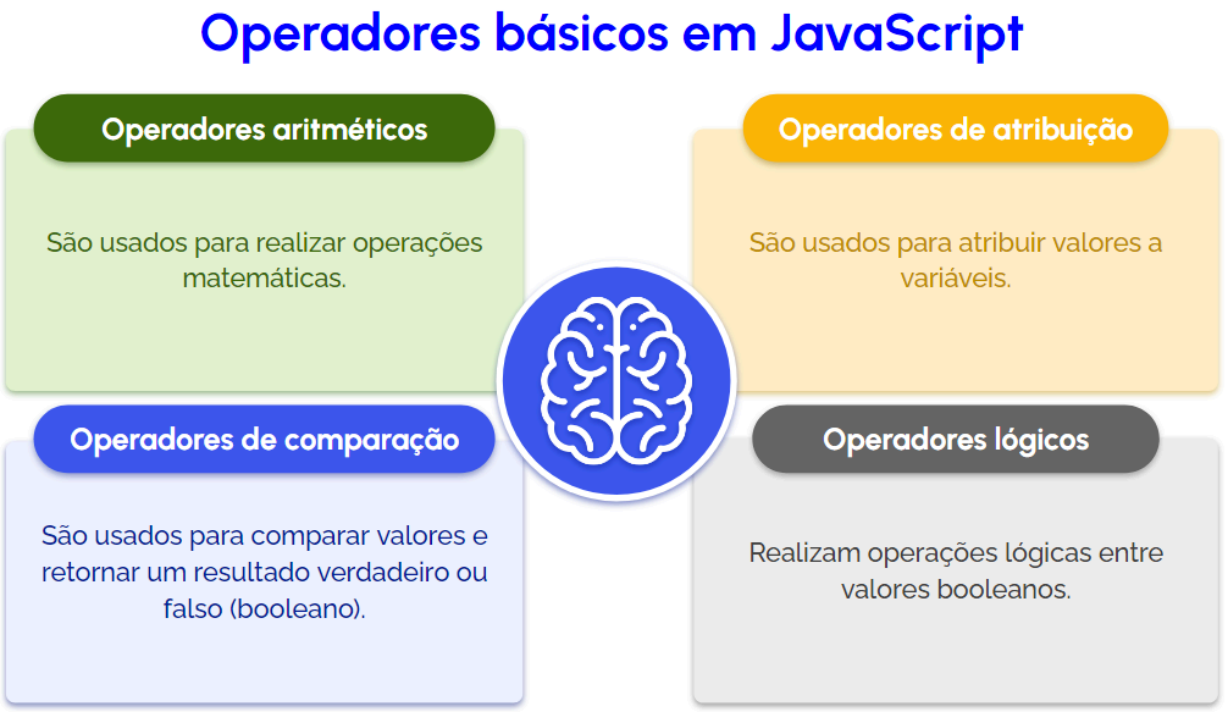
Entender esses tipos de dados é fundamental para começar a programar em JavaScript, pois nos permite trabalhar com diferentes tipos de informações e realizar diversas tarefas em nossos programas.

1.2.3 Operadores: manipulando dados

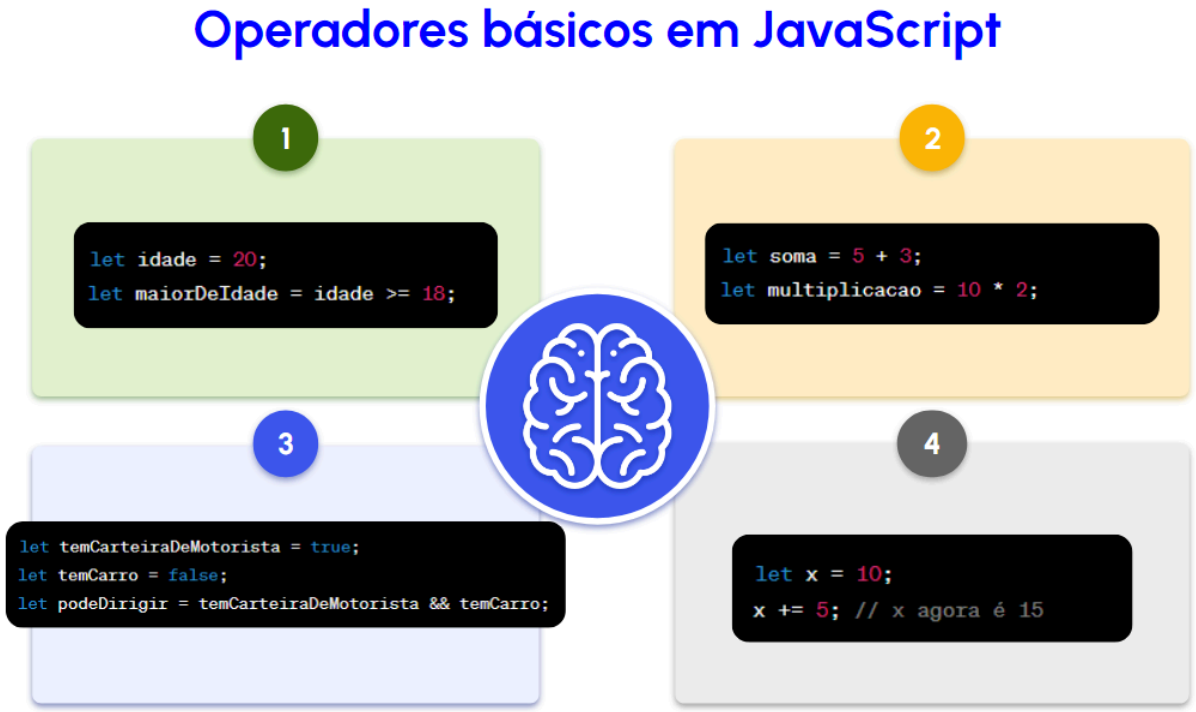
Em JavaScript, os operadores são como ferramentas que nos ajudam a fazer diferentes tipos de ações ou cálculos em nossos programas. Assim como usamos operadores na matemática para somar, subtrair, multiplicar ou dividir números, em JavaScript, usamos operadores para realizar ações similares e muito mais.

Disponível em: https://encurtador.com.br/BHOZ2. Acesso em: 05 fev. 2024.

Veja no esquema a seguir alguns dos principais operadores usados em JavaScript.



Agora, para você colocar em prática seus conhecimentos sobre os operadores em JavaScript, temos um desafio! No quadro a seguir estão exemplos de aplicação dos operadores. Tente responder corretamente qual é o operador que está sendo utilizado no código.



Gabarito: 1) Operadores de comparação; 2) Operadores aritméticos; 3) Operadores lógicos; 4) Operadores de atribuição.

Entender a sintaxe básica de JavaScript é fundamental para qualquer desenvolvedor *web*. Agora, você sabe que as variáveis armazenam dados, os tipos de dados definem a natureza desses dados e os operadores permitem a manipulação e a interação. Com esses conceitos fundamentais em mente, os desenvolvedores podem construir uma base sólida para explorar aspectos mais avançados da linguagem, se tornando capazes de criar aplicações *web* dinâmicas e interativas. Munido desses conhecimentos, o próximo passo é aplicá-los em projetos reais, aprofundando-se na linguagem e expandindo as habilidades de programação.

1.3 Escrevendo scripts JavaScript no HTML

Integrar *scripts* JavaScript em documentos HTML é uma prática fundamental para adicionar interatividade e dinamismo às páginas *web*. Para facilitar o seu entendimento sobre este assunto, produzimos um **passo a passo** que certamente irá ajudá-lo nos estudos. Nele, vamos explorar como incorporar e executar *scripts* JavaScript em documentos HTML. Para isso, **abra uma plataforma de teste de código** de sua preferência, como o CodePen (https://codepen.io/) ou o Visual Studio Code (https://code.visualstudio.com/) e mãos à obra!

1

Inclusão de scripts no HTML: incluir a tag **<script>** no arquivo HTML. Essa tag é usada para incorporar scripts diretamente no documento.

```html
<!DOCTYPE html>
<html lang="pt-br">
<head>
    <meta charset="UTF-8">
    <title>Meu Documento HTML</title>
</head>
<body>

    <!-- Inclusão do script -->
    <script>
        // Seu código JavaScript vai aqui
        alert("Olá, mundo!");
    </script>

</body>
</html>
```

O código acima incorpora um script diretamente no corpo da página HTML. No entanto, é comum incluir scripts na seção **<head>** ou no final do corpo (**</body>**) para garantir que a página seja carregada antes da execução do script.

2

Localização do *script*:

- dentro do **<head>**
- antes do **</body>**

```html
<head>
    <!-- Outros elementos do cabeçalho -->
    <script>
        // Seu código JavaScript aqui
    </script>
</head>
```

Dentro do <head>: colocar o *script* no cabeçalho é útil quando o *script* precisa ser carregado antes da renderização da página. No entanto, isso pode afetar o tempo de carregamento da página.

```html
<body>
    <!-- Conteúdo da página -->
    <script>
        // Seu código JavaScript aqui
    </script>
</body>
```

Antes do </body>: colocar o *script* no final do corpo da página é uma prática comum para garantir que o conteúdo da página seja carregado antes do *script*. Isso pode melhorar o desempenho percebido da página.

3 ***Scripts* externos:** para manter o código mais organizado e facilitar a sua reutilização, é possível criar *scripts* externos em arquivos separados.

```html
<!DOCTYPE html>
<html lang="pt-br">
<head>
    <meta charset="UTF-8">
    <title>Meu Documento HTML</title>
    <!-- Inclusão de script externo -->
    <script src="caminho/do/seu/script.js"></script>
</head>
<body>
    <!-- Conteúdo da página -->
</body>
</html>
```

No exemplo acima, o *script* é armazenado em um arquivo chamado **script.js**. Isso facilita a manutenção do código e permite a reutilização em várias páginas.

4 **Manipulação do DOM:** para manipular o DOM de forma segura, aguarde o carregamento completo da página usando o evento **DOMContentLoaded**.

```html
<script>
    document.addEventListener("DOMContentLoaded", function() {
        // Seu código JavaScript que manipula o DOM vai aqui
        let elemento = document.getElementById("meuElemento");
        elemento.innerHTML = "Olá, mundo!";
    });
</script>
```

Ao esperar pelo evento **DOMContentLoaded**, você garante que o DOM esteja totalmente carregado antes de manipulá-lo. Faça o teste!

5 **Boas práticas:** para otimizar o desempenho, use os atributos **async** ou **defer** quando incluir *scripts* externos.

```html
<script async src="caminho/do/seu/script.js"></script>
```

Async: esse atributo indica que o *script* é assíncrono e não bloqueará o carregamento da página enquanto é baixado.

```html
<script defer src="caminho/do/seu/script.js"></script>
```

Defer: esse atributo adia a execução do *script* até que o HTML seja totalmente analisado.

Vamos refletir?

E aí, você conseguiu colocar em prática todos os passos? O que achou da experiência? Escrever *scripts* JavaScript no HTML é uma habilidade fundamental para desenvolvedores *web*. Entender a localização e o momento da execução do *script* é crucial para garantir uma experiência do usuário otimizada. Ao seguir boas práticas e compreender as nuances da integração de *scripts*, você poderá criar páginas *web* mais dinâmicas e interativas.

DESAFIO PRÁTICO

Desenvolver e implementar *scripts* JavaScript simples em uma página HTML

Descrição

A empresa Web Interact possui um *site* institucional que fornece informações sobre os seus produtos e serviços. No entanto, a firma percebeu que a página inicial estava estática e desejava torná-la mais dinâmica, com a capacidade de exibir mensagens personalizadas e interagir com os visitantes de forma mais amigável.

Objetivos

- Adicionar uma mensagem de boas-vindas personalizada na página inicial;
- Implementar um formulário de contato simples;
- Incluir um botão que altera dinamicamente o tema da página.

Orientações

- **Estrutura HTML e CSS:**

- comece criando a estrutura básica da página usando HTML;
- defina a aparência visual do painel com CSS. Pense em cores, fontes e em um *layout*.

- **Mensagem de boas-vindas:**
  - crie um elemento na página (por exemplo, um **<p>**) para exibir a mensagem de boas-vindas;
  - use a linguagem JavaScript para personalizar essa mensagem com o nome do administrador (fictício).

- **Formulário de contato:**
  - crie um formulário HTML com campos para nome, *e-mail* e mensagem;
  - implemente a lógica para exibir uma mensagem de confirmação após o envio do formulário.

- **Alteração de tema:**
  - adicione um botão (um **<button>**, por exemplo) que, quando clicado, altera as cores de fundo e texto da página;
  - use a linguagem JavaScript para aplicar essas mudanças de forma dinâmica.

- **JavaScript:**
  - utilize eventos (como **click**) para interagir com os elementos da página;
  - use variáveis (**var**, **let** ou **const**) para armazenar informações relevantes.

- **Teste e ajuste:**
  - teste o painel em diferentes navegadores e dispositivos;
  - verifique se todas as funcionalidades estão funcionando conforme o esperado.

DESAFIO PRÁTICO II

Otimizando o desempenho de um *site* de *e-commerce*

Descrição

Você foi contratado para melhorar o desempenho de um *site* de *e-commerce* que está enfrentando problemas de lentidão e uma alta taxa de rejeição. O seu objetivo é otimizar a velocidade de carregamento das páginas, melhorar a experiência do usuário e aumentar as conversões.

Objetivos

- Realizar a análise de desempenho do *site*;
- Otimizar as imagens, utilizando formatos mais eficientes e reduzindo o tamanho sem comprometer a qualidade visual;
- Minimizar o tempo de carregamento da página;
- Configurar o *cache* do navegador para armazenar recursos estáticos;
- Realizar testes e monitoramento do *site*.

Orientações

- **Análise de desempenho:**
  - realize uma análise detalhada do *site* para identificar gargalos de desempenho;
  - verifique o tempo de carregamento das páginas, o tamanho dos arquivos, a quantidade de requisições e outros fatores relevantes.
- **Otimização de imagens:**
  - reduza o tamanho das imagens sem comprometer a qualidade visual;
  - utilize formatos de imagem mais eficientes, como WebP.

- **Minificação de arquivos CSS e JavaScript:**
    - minimize os arquivos CSS e JavaScript para reduzir o tempo de carregamento;
    - remova códigos desnecessários e organize o código de forma eficiente.
- ***Cache* e compactação:**
    - configure o *cache* do navegador para armazenar recursos estáticos.
    - ative a compactação *Gzip* para reduzir o tamanho dos arquivos transferidos.
- **Testes e monitoramento:**
    - realize testes de desempenho antes e depois das otimizações;
    - monitore o *site* regularmente para garantir que as melhorias sejam mantidas.

Nesse capítulo, nós aprendemos que JavaScript é uma linguagem de programação essencial no desenvolvimento *web*, permitindo a criação de páginas interativas e dinâmicas. Entendemos que a sintaxe básica do JavaScript inclui conceitos como **variáveis**, **tipos de dados** e **operadores**, fornecendo os fundamentos necessários para a **manipulação eficiente de informações**. Além disso, estudamos que a **integração de *scripts*** JavaScript no HTML desempenha um papel crucial na adição de funcionalidades às páginas *web*, seja incorporando código diretamente no documento ou utilizando *scripts* externos.

Também vimos que, ao compreender a localização apropriada para os *scripts* e aplicar boas práticas, os desenvolvedores podem criar experiências de usuário mais eficientes e responsivas, explorando também conceitos avançados, como **manipulação do DOM** e **operações assíncronas**. Assim como aprendemos, a habilidade de escrever *scripts* JavaScript de forma eficaz é fundamental para construir aplicações *web* modernas e envolventes.

ATIVIDADE DE FIXAÇÃO

1. O que é JavaScript? Qual é o seu papel fundamental no desenvolvimento *web*?
2. Explique a diferença entre as palavras-chave **var**, **let** e **const** na declaração de variáveis em JavaScript.
3. A manipulação do DOM em JavaScript contribui para a criação de páginas *web* interativas? Explique.
4. Qual é a importância de incluir *scripts* JavaScript no final do corpo (**</body>**) em vez de no cabeçalho (**<head>**) do HTML?
5. Como você declara e utiliza uma função em JavaScript? Dê um exemplo simples.
6. Descreva o conceito de tipos de dados em JavaScript, fornecendo exemplos de pelo menos três tipos diferentes.
7. O que são operadores em JavaScript e como eles são utilizados para manipular dados?
8. Explique a diferença entre os atributos **async** e **defer** ao incluir *scripts* externos em HTML.
9. Como você espera o carregamento completo da página antes de executar *scripts* JavaScript que manipulam o DOM? Dê um exemplo.
10. Qual é a finalidade dos eventos DOMContentLoaded e como eles são relevantes no contexto da manipulação do DOM em JavaScript?

CAPÍTULO 02

EXPLORANDO AS FUNÇÕES E INTERAÇÃO EM JAVASCRIPT

O que esperar deste capítulo:

- Compreender os conceitos fundamentais relacionados à criação, à chamada e ao retorno de funções em JavaScript;
- Desenvolver habilidades para interagir dinamicamente com o DOM usando funções JavaScript;
- Aplicar funções JavaScript na resolução de problemas práticos, promovendo a resolução eficiente e organizada de desafios.

2.1 Criando um *script* JavaScript simples para interação

JavaScript é uma linguagem de programação essencial no desenvolvimento *web*, permitindo a criação de *scripts* que interagem dinamicamente com o conteúdo das páginas. Neste material, vamos explorar, por meio de um passo a passo, a criação de um *script* JavaScript simples, focado em interação, desde a incorporação no HTML até a manipulação básica do DOM. Para isso, **abra uma plataforma de teste de código** de sua preferência, como o CodePen (https://codepen.io/) ou Visual Studio Code (https://code.visualstudio.com/), e mãos à obra!

1

Incorporando o *script* no HTML: comece incorporando o *script* diretamente no documento HTML. Abra o seu editor de texto favorito e crie um arquivo HTML com a estrutura ao lado:

```
<!DOCTYPE html>
<html lang="pt-br">
<head>
    <meta charset="UTF-8">
    <title>Script de Interação</title>
</head>
<body>

    <!-- Conteúdo da Página -->

    <!-- Incorporação do Script -->
    <script>
        // Seu código JavaScript vai aqui
    </script>

</body>
</html>
```

2 **Selecionando elementos no DOM:** para interagir com a página, você precisa selecionar elementos no DOM. Para isso, crie uma função simples para alterar o conteúdo de um elemento. Adicione o seguinte código dentro da tag **<script>**:

```
document.addEventListener("DOMContentLoaded", function() {
    // Função para alterar o conteúdo de um elemento
    function alterarConteudo() {
        let elemento = document.getElementById("meuElemento");
        elemento.innerHTML = "Cliquei no botão!";
    }

    // Outras interações podem ser adicionadas aqui

});
```

3 **Adicionando um botão para interagir:** crie um botão no corpo da página que, quando clicado, aciona a função criamos no passo 2. Abaixo da função **alterarConteudo()**, adicione o seguinte código:

```
// Adicionando um botão
let botao = document.createElement("button");
botao.textContent = "Clique-me";
botao.addEventListener("click", alterarConteudo);
document.body.appendChild(botao);
```

4 **Estilizando o botão (opcional):** para uma experiência mais visual, você pode adicionar um pouco de estilo ao botão. Abaixo da criação do botão, adicione o seguinte código:

```
// Estilizando o botão (opcional)
botao.style.padding = "10px";
botao.style.fontSize = "16px";
botao.style.cursor = "pointer";
```

Vamos refletir?

Você conseguiu colocar em prática todos os passos?

O que achou da experiência?

A partir desse exercício, é possível perceber como a linguagem JavaScript pode ser utilizada para responder a eventos do usuário, manipular o DOM (Modelo de Objeto de Documento) e proporcionar uma experiência mais fluida e agradável aos visitantes de um *site*.

2.2 Funções JavaScript e as suas utilidades

As funções em JavaScript desempenham um papel crucial, proporcionando **modularidade**, **reutilização de código** e **organização estruturada em projetos *web***. Vamos explorar o conceito de funções e como elas podem ser utilizadas para aprimorar o desenvolvimento em JavaScript.

2.2.1 Entendendo o conceito de funções

Na linguagem JavaScript, uma função é um **bloco de código** que pode ser chamado e executado para realizar uma tarefa específica. Ela pode aceitar parâmetros, processá-los e retornar um resultado. Para criar uma função, utiliza-se a palavra-chave **function**. Veja um exemplo básico a seguir:

```
// Definindo uma função simples
function saudacao(nome) {
    return "Olá, " + nome + "!";
}

// Chamando a função
let mensagem = saudacao("João");
console.log(mensagem); // Saída: "Olá, João!"
```

Neste exemplo, é possível observar o seguinte:

- a função **saudacao** é declarada usando a palavra-chave **function**. Depois disso, ela recebe um parâmetro **nome**;
- o corpo da função contém uma instrução **return**, que concatena a *string* **"Olá, "** com o valor do parâmetro **nome** e adiciona **"!"** ao final;
- em seguida, a função **saudacao** é chamada com o argumento **"João"**. Isso significa que o código dentro da função será executado com o nome recebendo o valor "João";
- o resultado retornado pela função é atribuído à variável **mensagem**;
- por fim, a variável mensagem contém o **resultado da função**. Ou seja, a saudação personalizada com o nome "João": **Saída: "Olá, João!"**.
- O **console.log** é usado para imprimir o valor da variável mensagem no console.

E aí, conseguiu acompanhar a leitura desse código?

A sua jornada no universo da programação está apenas começando e, nesse caminho, você deve analisar os códigos como um detetive digital para descobrir o que cada linha está fazendo.

2.2.2 Parâmetros e retorno de funções

As funções podem aceitar parâmetros para torná-las mais flexíveis e reutilizáveis. Em programação, um parâmetro é uma **variável ou um valor** que é passado para uma função durante a sua chamada. Ele serve como uma espécie de entrada para a função, permitindo que ela receba dados específicos necessários para realizar uma tarefa ou cálculo.

Esses parâmetros são **definidos quando a função é declarada** e são **utilizados para transmitir informações à função quando ela é requisitada**. Além disso, os parâmetros podem **retornar valores** para serem utilizados em outras partes do código. Vamos dar uma olhada em um exemplo simples para ilustrar o conceito de parâmetros:

```
// Função que calcula a média de dois números
function calcularMedia(numero1, numero2) {
    let media = (numero1 + numero2) / 2;
    return media;
}

// Chamando a função e armazenando o resultado
let resultadoMedia = calcularMedia(10, 20);
console.log(resultadoMedia); // Saída: 15
```

Vamos analisar este exemplo passo a passo:

- a função **calcularMedia** é declarada, aceitando dois parâmetros: **numero1** e **numero2**;
- dentro da função, a média é calculada somando os dois números e dividindo ele por 2;
- o resultado deste cálculo é armazenado na variável **media** e, depois, é retornado pela instrução **return**;
- observe que a função **calcularMedia** é chamada com os argumentos 10 e 20. Isso significa que **numero1** recebe **10** e **numero2** recebe **20**;
- o resultado da função, que é a média desses dois números, é armazenado na variável **resultadoMedia**;
- na última linha do código, é possível ver que o valor armazenado na variável **resultadoMedia** é impresso no console usando **console.log** e, quando executado, retorna a saída esperada, que é **"15"**.

E aí, o que você achou da análise desse código? Nós vamos explorar a leitura e a interpretação dos códigos em JavaScript gradualmente para que você domine essa linguagem e se aventure cada vez mais no mundo da programação!

Disponível em: https://encurtador.com.br/ouEQT. Acesso em: 08 fev. 2024.

2.2.3 Funções anônimas e expressões de funções

Além da forma tradicional de declarar funções, a linguagem JavaScript permite a criação de funções anônimas e a expressões de funções.

Uma **expressão de função** é uma maneira de criar funções de forma mais concisa e pode ser usada em lugares em que as expressões são permitidas, como as funções anônimas.

As **funções anônimas** são funções que não têm um nome associado a elas. Muitas vezes, elas são criadas no momento em que são necessárias e podem ser passadas como argumentos para outras funções ou atribuídas a variáveis. Frequentemente, elas são usadas em situações em que a função é usada apenas uma vez. Observe o código a seguir:

```
// Função anônima atribuída a uma variável
let saudacaoAnonima = function(nome) {
    return "Olá, " + nome + "!";
};

// Chamando a função anônima
console.log(saudacaoAnonima("Maria")); // Saída: "Olá, Maria!"
```

Vamos entender passo a passo o que está contido neste código:

- primeiro, uma função anônima é definida e atribuída à variável **saudacaoAnonima**;
- a função aceita um parâmetro **nome**, que pode ser observado na segunda linha, e retorna uma *string* de saudação personalizada **"Olá, " + nome + "!"**;
- a função anônima é chamada com o argumento **"Maria"** durante a impressão no console usando **console.log**;
- o resultado da função, que é a saudação personalizada com o nome fornecido, é exibido no console. Logo, temos **"Olá, Maria"** como resultado dessa função anônima.

As funções anônimas são frequentemente usadas como ***callbacks*** em operações assíncronas, em expressões de função e em situações em que uma função é usada apenas localmente, sem a necessidade de um nome global. Essas formas de função oferecem **flexibilidade** na definição de funções em contextos específicos.

2.2.4 Escopo de variáveis em funções

O escopo em JavaScript refere-se à **visibilidade e acessibilidade de variáveis em diferentes partes do código**. É a região do código onde uma determinada variável é declarada e pode ser acessada. As funções possuem escopo próprio, o que significa que as variáveis declaradas dentro delas não são acessíveis fora. Isso é crucial para evitar conflitos de nomes e manter a integridade do código.

```
// Exemplo de escopo de variáveis em funções
function exemploEscopo() {
    let variavelLocal = "Eu sou local";

    // A variável local é acessível aqui
    console.log(variavelLocal); // Saída: "Eu sou local"
}

// A variável local não é acessível aqui
console.log(variavelLocal); // Erro: variavelLocal is not defined
```

Vamos à análise deste exemplo:

- a função **exemploEscopo** é definida e, dentro dela, uma variável local chamada **variavelLocal** é declarada e inicializada com o valor **"Eu sou local"**;
- a função imprime o valor dessa variável local no console;

- ao tentar acessar a variável local **variavelLocal** fora da função, ocorre um erro (**variavelLocal is not defined**). Isso ocorre porque a variável foi declarada dentro da função e, portanto, tem um escopo local, tornando-se inacessível fora da função.

2.2.5 *Callbacks* e assincronia

Callbacks e assincronia são conceitos fundamentais em JavaScript, especialmente quando se trabalha com operações que envolvem tempo, como requisições de rede, leitura de arquivos e eventos do usuário. Vamos entender cada um desses conceitos.

Em JavaScript, ***callback*** é uma **função que é passada como argumento para outra função e é executada após a conclusão de uma operação assíncrona ou de uma tarefa específica**. Para entender melhor esse conceito, imagine que você está organizando uma festa de aniversário.

Você contrata um mágico (função) para fazer um truque (operação assíncrona). Antes do mágico começar a se apresentar, você diz a ele: "Quando terminar, acenda uma luzinha para me avisar". Então, enquanto faz o truque, o mágico lembra da sua instrução e, ao terminar, acende a luzinha. Isso é um *callback*, uma espécie de luzinha que avisa que a tarefa assíncrona foi concluída.

Disponível em: https://encurtador.com.br/mvwx7. Acesso em: 06 fev. 2024.

Normalmente, o *callback* é utilizado para lidar com operações que não são imediatas, como a execução de código após uma requisição HTTP ou a leitura de um arquivo.

As funções em JavaScript desempenham um papel vital em operações assíncronas e *callbacks*. Elas podem ser utilizadas para controlar o fluxo de execução em operações que

levam tempo, como a leitura de arquivos ou requisições HTTP. Veja um exemplo de *callback* no código a seguir.

```
// Exemplo de callback em uma função assíncrona
function lerArquivo(caminho, callback) {
    // Lógica para ler o arquivo (simulada)
    let conteudo = "Conteúdo do arquivo...";
    callback(conteudo);
}

// Chamando a função com um callback
lerArquivo("arquivo.txt", function(resultado) {
    console.log("Conteúdo do arquivo:", resultado);
});
```

Aqui, podemos observar o seguinte:

- a função **lerArquivo** é definida para simular a leitura de um arquivo. Ela aceita dois parâmetros: **caminho** (o caminho do arquivo) e ***callback*** (uma função a ser chamada após a leitura);
- a lógica simulada para ler o arquivo atribui um conteúdo fictício à variável **conteudo** e chama a função *callback* com ele;
- a função **lerArquivo** é chamada com o caminho do arquivo (**"arquivo.txt"**) e uma função *callback* anônima;
- a função **callback** recebe o resultado (que é o conteúdo fictício do arquivo) e o imprime no console.

Parabéns! Você desbloqueou um novo nível em sua jornada de programação!

Depois de explorar as funções e interação em JavaScript e desvendar os segredos da leitura de linhas de código, agora você possui uma habilidade valiosa no universo da programação. Continue firme na sua jornada de aprendizado!

DESAFIO PRÁTICO

Criando um *script* JavaScript simples para interação

Descrição

Você está desenvolvendo uma página web e deseja criar um contador simples para exibir quantas vezes um botão foi clicado.

Objetivos

- Criar um botão na página HTML com um identificador único (por exemplo, **id="botaoContador"**);
- Selecionar o botão usando **document.querySelector** no seu arquivo JavaScript;
- Criar uma variável chamada **contador** e inicializá-la com o valor 0;
- Adicionar um ouvinte de eventos ao botão para detectar cliques;
- Incrementar o valor do contador em 1 dentro do ouvinte de eventos;
- Atualizar o texto exibido na página com o valor atual do contador.

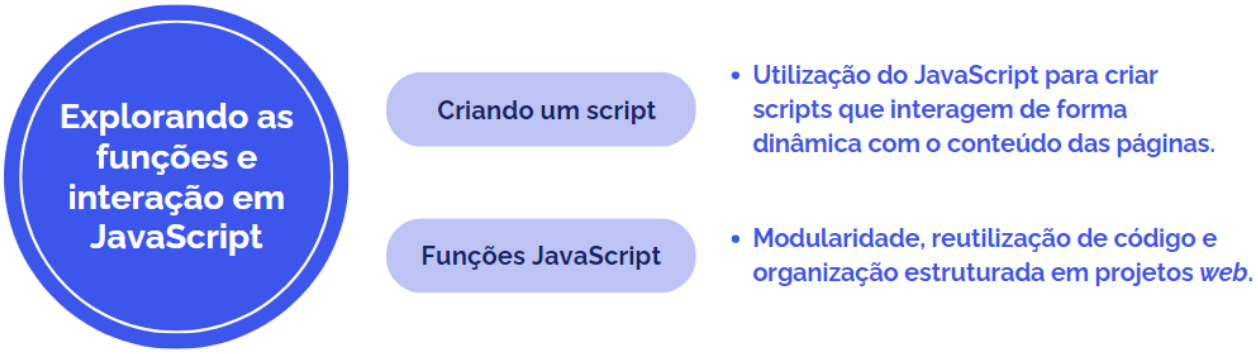
Orientações

- Implemente medidas de segurança rigorosas para garantir que as funções JavaScript possam ser executadas de forma segura e sem riscos de segurança;
- Ofereça a capacidade de integrar APIs externas diretamente nas funções JavaScript, proporcionando acesso a dados externos aos usuários;
- Crie um ambiente de desenvolvimento interativo que forneça recursos como sugestões de código, depuração e perfis de desempenho;
- Adicione funcionalidades de controle de versão para as funções JavaScript, permitindo aos usuários rastrear alterações ao longo do tempo.

RESUMO

Aprendemos que as funções em JavaScript representam um componente fundamental na construção de aplicações *web*, proporcionando modularidade e reusabilidade ao código. Neste material, nós conhecemos os conceitos essenciais de funções, destacando a sua declaração, os seus parâmetros e o retorno. Além disso, exploramos a versatilidade das funções em JavaScript, incluindo formas anônimas e expressões de funções.

Também discutimos o escopo de variáveis dentro de funções, destacando a importância da modularidade e prevenção de conflitos de nomes. Por fim, abordamos o papel crucial das funções em operações assíncronas, usando *callbacks* para controlar fluxos de execução em situações que envolvem tempo de espera, como a leitura de arquivos ou requisições HTTP. Dominar as funções em JavaScript é essencial para desenvolvedores que buscam criar um código claro, organizado e eficiente em projetos *web*.



ATIVIDADE DE FIXAÇÃO

1. O que é uma função em JavaScript e qual é a sua utilidade fundamental no desenvolvimento *web*?
2. Como se declara uma função em JavaScript? Dê um exemplo simples.
3. Explique a diferença entre parâmetros e argumentos em uma função JavaScript.
4. Qual é a importância do retorno de valores em funções e como isso contribui para a modularidade do código?
5. O que são funções anônimas em JavaScript e como elas diferem das funções nomeadas? Dê um exemplo prático.
6. Descreva a utilidade do escopo de variáveis em funções e o motivo dele ser crucial para evitar conflitos de nomes.
7. Como as funções podem ser utilizadas para lidar com operações assíncronas em JavaScript? Dê um exemplo de *callback* em uma situação prática.
8. Quais são as vantagens de utilizar expressões de funções em comparação com a declaração tradicional de funções?
9. Explique como o escopo de variáveis em funções contribui para a segurança e integridade do código em JavaScript.
10. Em que contexto as funções em JavaScript se destacam na criação de código mais modular, reutilizável e eficiente em projetos *web*?

CAPÍTULO 03

TOMADA DE DECISÃO EM JAVASCRIPT: EXPLORANDO AS ESTRUTURAS IF, ELSE, SWITCH E OS OPERADORES LÓGICOS

O que esperar deste capítulo:

- Aplicar corretamente as estruturas de decisão **if**, **else if**, **else** e **switch** em JavaScript;
- Usar operadores lógicos (**&&, ||, !**) para a avaliação de condições em estruturas de decisão;
- Utilizar estruturas condicionais para resolver problemas práticos em JavaScript.

3.1 Estruturas de decisão: if, else e switch

As estruturas de decisão em JavaScript são recursos fundamentais que permitem ao programador **controlar o fluxo de execução do código baseado em condições específicas**. Elas ajudam a **criar lógicas de tomada de decisão** dentro de um programa, permitindo que diferentes blocos de código sejam executados caso uma determinada condição seja verdadeira ou falsa.

Para ilustrar de forma mais clara, considere que o código em JavaScript se assemelha a um mapa de um jogo de *videogame*, em que as estruturas de decisão são como os portais que se abrem ou fecham conforme as escolhas do jogador, que, neste caso, é o programador. Assim como no jogo, se o jogador (condição) possuir a chave mágica (verdadeira), ele optará por uma porta; caso contrário (falsa), ele tentará outra.

Existem duas principais estruturas de decisão em JavaScript: **if-else** e **switch**. Vamos conhecer as características e as aplicações das duas neste material, para que você entenda como utilizá-las corretamente.

If-else	Switch
A estrutura **if** é usada para **avaliar uma condição**. Se ela for **verdadeira**, o bloco de código dentro do **if** é **executado**. Caso contrário, ou seja, se a condição for **falsa**, o bloco de código dentro do **else** (se presente) é **executado**.	Essa estrutura é usada quando há a necessidade de **avaliar diferentes valores de uma variável** e executar blocos de código com base nesses valores.

3.1.1 Estrutura if-else

Para começar, vamos acompanhar um exemplo do uso da estrutura **if-else** em um código:

```
let idade = 18;

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

Neste exemplo, o código dentro do bloco **if** será executado apenas se a condição **(idade >= 18)** for **verdadeira**.

Agora, observe esse outro exemplo:

```
let idade = 16;

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

Aqui, o código dentro do bloco **else** será executado se a condição **(idade >= 18)** for **falsa**.

Viu como é fácil? Agora que você já está craque no uso do **if-else**, vamos conhecer uma derivação dessa estrutura, o **else if**. Ela permite avaliar **condições adicionais caso a condição do if seja falsa**. Veja um exemplo:

```
let nota = 75;

if (nota >= 90) {
    console.log("Aprovado com distinção!");
} else if (nota >= 60) {
    console.log("Aprovado.");
} else {
    console.log("Reprovado.");
}
```

Neste caso, **múltiplas condições são avaliadas** sequencialmente, que são "Aprovado com distinção!", "Aprovado" e "Reprovado". Vamos entender melhor analisando cada parte do código:

- primeiro, declara-se uma variável chamada **nota** e atribui-se a ela o valor **75**;
- em seguida, há uma estrutura condicional que verifica **três cenários** com base no valor da variável **nota**:
    - se **nota** for **maior ou igual a 90**, o programa imprime **"Aprovado com distinção!"**;
    - se **nota** for **menor que 90 mas maior ou igual a 60**, o programa imprime **"Aprovado."**;
    - já se **nota** for **menor que 60**, o programa imprime **"Reprovado."**.

3.1.2 Estrutura switch

O **switch** é uma estrutura de **controle de fluxo** em programação que permite que você tome **decisões baseadas no valor de uma expressão**. Ela é especialmente útil quando se deseja **comparar uma expressão com vários valores diferentes**. Vamos compreender melhor essa estrutura vendo um exemplo de sua aplicação em um código.

```
let diaDaSemana = 3;

switch (diaDaSemana) {
    case 1:
        console.log("Segunda-feira");
        break;
    case 2:
        console.log("Terça-feira");
        break;
    case 3:
        console.log("Quarta-feira");
        break;
    case 4:
        console.log("Quinta-feira");
        break;
    case 5:
        console.log("Sexta-feira");
        break;
    case 6:
        console.log("Sábado");
        break;
    case 7:
        console.log("Domingo");
        break;
    default:
        console.log("Valor inválido para dia da semana");
}
```

Neste exemplo, **diaDaSemana** tem o valor **3**. Sendo assim, a estrutura **switch avalia a expressão e procura um caso correspondente**. Se encontrar um caso correspondente, ela executa o bloco de código associado a esse caso. No nosso exemplo, o console imprimirá **"Quarta-feira"**, pois **3** corresponde ao **caso 3**.

É importante notar que cada bloco de código de caso é seguido por uma instrução **break**. Isso é crucial para evitar a execução de outros casos após encontrar o caso correspondente. O bloco **default** é executado se nenhum dos casos corresponder ao valor da expressão.

Depois da análise do nosso exemplo, ficou fácil entender a estrutura **switch**, não é? Ela oferece uma maneira concisa e eficiente de lidar com várias condições.

3.2 Operadores lógicos para avaliação de condições

Depois de entender como as estruturas de decisão funcionam, agora vamos explorar os **operadores lógicos**, ferramentas essenciais em JavaScript para **avaliação de condições** que permitem que os desenvolvedores criem uma **lógica de controle de fluxo mais sofisticada**. Esses operadores permitem uma tomada de decisões **com base em múltiplas**

situações. É muito importante entender como esses operadores funcionam para construir uma lógica eficiente em seus programas.

Os operadores mais comuns de serem utilizados são o **&&** (**E lógico**), **||** (**OU lógico**) e **!** (**NÃO lógico**). Vamos entender como cada um deles deve ser usado para avaliar condições.

&&

Conhecido como **E lógico,** esse operador é usado para avaliar duas expressões booleanas. Ele retorna **true** se ambas as expressões forem verdadeiras e **false** se, pelo menos, uma delas for falsa. Sua sintaxe básica é a seguinte: **expressão1 && expressão2**.

```
let idade = 25;
let possuiCarteira = true;

if (idade >= 18 && possuiCarteira) {
    console.log("Você pode dirigir!");
} else {
    console.log("Você não pode dirigir.");
}
```

Neste exemplo, a mensagem será exibida apenas se a idade for **maior ou igual a 18** e a variável **possuiCarteira** for **verdadeira**.

||

Conhecido como "**OU lógico**", esse operador é caracterizado pelo uso de duas barras verticais. Ele é usado para avaliar duas expressões booleanas e retorna **true** se, **pelo menos, uma** das expressões for verdadeira e **false** apenas se **ambas** forem falsas. Sua sintaxe básica é a seguinte: **expressão1 || expressão2**.

```
let temExperiencia = false;
let possuiGraduacao = true;

if (temExperiencia || possuiGraduacao) {
    console.log("Você atende aos requisitos.");
} else {
    console.log("Você não atende aos requisitos.");
}
```

Neste caso, a mensagem será exibida se o candidato tiver experiência ou possuir uma graduação.

!

Conhecido como "**NÃO lógico**" ou "**negação**", esse operador lógico é usado para **inverter o valor** de uma expressão booleana. Ou seja, se a expressão for verdadeira, o uso da **!** tornará a expressão falsa. Se a expressão for falsa, a **!** tornará a expressão verdadeira. Sua sintaxe básica é: **!expressão**.

```
let possuiPermissao = false;

if (!possuiPermissao) {
  console.log("Acesso negado.");
} else {
  console.log("Acesso permitido.");
}
```

Neste exemplo, a condição **!possuiPermissao** verifica se a variável **possuiPermissao** é falsa, com isso, a mensagem "Acesso negado" é exibida. Se a variável fosse **true**, a mensagem "Acesso permitido" seria exibida.

Os operadores lógicos são ferramentas essenciais na programação, já que eles nos permitem realizar avaliações complexas e tomar decisões com base em condições. Ao dominar o uso desses operadores, você poderá criar lógicas condicionais poderosas e eficientes, proporcionando maior controle sobre o fluxo de seus programas. Este conhecimento é fundamental para construir aplicações robustas e eficazes.

DESAFIO PRÁTICO

Tomada de decisão em sistema de controle de estoque

Descrição

Você faz parte da equipe de desenvolvimento de *software* de uma empresa de varejo que lida com um grande inventário de produtos. Recentemente, a firma enfrentou desafios na gestão de estoque e a implementação de estruturas de decisão se tornou crucial para otimizar os processos.

A empresa possui um sistema de controle de estoque em que é necessário avaliar diferentes condições para tomar decisões adequadas. Por exemplo, é necessário verificar se um produto está em estoque antes de processar uma venda ou se um item está com baixo estoque para acionar um pedido de reposição.

Objetivos

- Implementar estruturas de decisão **if** e **else** para verificar se um produto está em estoque antes de permitir uma venda;
- Utilizar estruturas de decisão para avaliar se um item está com baixo estoque e acionar automaticamente um pedido de reposição;
- Utilizar a estrutura **switch** para lidar com diferentes cenários, como diferentes estados de um produto (disponível, em espera, fora de estoque).

Orientações

- Ao utilizar as estruturas de decisão, certifique-se de que as mensagens de *feedback* para os usuários sejam claras e informativas;
- Realize testes abrangentes para garantir que as estruturas de decisão estejam funcionando corretamente em diferentes cenários;
- Estruture o código de maneira que seja fácil adicionar ou modificar condições de decisão no futuro;
- Forneça documentação clara sobre como as estruturas de decisão foram implementadas para facilitar a manutenção futura.

Exploramos as estruturas fundamentais de controle de fluxo, como as instruções **if** e **else**, que permitem que o programa execute diferentes blocos de código com base em condições específicas.

Além disso, discutimos a estrutura **switch**, que é útil para casos em que existem múltiplas condições a serem avaliadas. Também estudamos os operadores lógicos em JavaScript, como **&& (E lógico)**, **|| (OU lógico)** e **! (NÃO lógico ou negação)**, que desempenham um papel crucial na avaliação de condições e na criação de lógica de controle de fluxo eficiente. Ao utilizar esses operadores de maneira inteligente, você poderá criar decisões complexas e adaptáveis nos seus códigos, proporcionando maior flexibilidade e sofisticação à lógica de programação. Tanto as estruturas **if**, **else** e **switch** como os operadores lógicos são ferramentas fundamentais para a construção de aplicações *web* que respondam dinamicamente a diferentes cenários, tornando-se essenciais para um desenvolvimento moderno no mundo da programação.

ATIVIDADE DE FIXAÇÃO

1. O que o operador **&&** faz em JavaScript? Em quais situações ele é mais útil?

2. Explique o funcionamento do operador **||** e forneça um exemplo prático da sua aplicação em condições.

3. Como o operador **!** pode ser utilizado para inverter o valor de uma expressão? Qual é sua utilidade em condições lógicas?

4. Diferencie as situações em que o operador **&&** e **||** seriam mais apropriados para a avaliação de múltiplas condições.

5. Em quais cenários o operador **!** seria particularmente útil na escrita de condições lógicas? Dê um exemplo prático.

6. Crie uma expressão lógica complexa que envolva tanto **&&** quanto **||** para simular uma condição realista em uma aplicação *web*.

7. Se tivermos duas variáveis booleanas, **temInternet** e **temEnergia**, como podemos usar os operadores lógicos para decidir se podemos realizar uma tarefa *on-line*?

8. Qual é a diferença entre utilizar os operadores **&&** e **||** em uma declaração condicional? Como essa escolha impacta o comportamento do código?

9. Explique como os operadores lógicos são úteis na construção de decisões mais complexas, como aquelas envolvendo mais de duas condições.

10. Qual é a diferença entre usar uma declaração condicional com **if** e **else** e **switch**? Forneça um exemplo de como o **switch** pode ser útil em uma situação específica.

CAPÍTULO 04

EXPLORANDO A EFICIÊNCIA EM JAVASCRIPT: ESTRUTURAS DE REPETIÇÃO, ARRAYS E ESTRUTURAS DE CONTROLE

O que esperar deste capítulo:

- Integrar conhecimentos adquiridos em estruturas de repetição e *arrays* em um projeto prático;
- Aplicar estruturas de controle, incluindo *loops*, para resolver problemas simples em JavaScript;
- Utilizar as estruturas de repetição **for**, **while** e **do...while** em JavaScript.

4.1 Estruturas de repetição: for, while, do...while

As estruturas de repetição, também conhecidas como ***loops***, são ferramentas poderosas em JavaScript, pois são utilizadas para **realizar tarefas repetitivas**, como executar um bloco de código várias vezes, com base em uma condição específica. Imagine que você tem que fazer a mesma coisa várias vezes, como cumprimentar cada amigo que chega na sua festa de aniversário. Usar as estruturas de repetição (*loops*) é como ter um robô que fará isso por você, percorrendo toda a lista de convidados e cumprimentando cada amigo automaticamente. Isso economiza tempo e esforço, permitindo que você curta a sua festa e, no mundo da programação, os *loops* tornam o seu programa muito mais eficiente.

Existem três principais estruturas de repetição: **for**, **while** e **do...while**. Vamos explorá-las, compreendendo como cada uma opera e as suas aplicações práticas na área da programação. Mas, antes, nós precisamos conhecer o conceito de **iteração**.

Disponível em: https://encurtador.com.br/bCDKN. Acesso em: 16 fev. 2024.

A iteração permite que um programa execute operações em cada item de uma coleção, como percorrer todos os elementos de um *array* para realizar uma determinada tarefa. Isso é fundamental para lidar de forma eficiente com conjuntos de dados, **automatizando a execução de ações** em cada elemento sem a necessidade de repetir manualmente o código. Veja a seguir um exemplo de como a iteração funciona em um código:

```
// Array de números
var numeros = [1, 2, 3, 4, 5];

// Iteração usando um loop for
// Inicializa uma variável i com 0; continua o loop enquanto i for menor que o
comprimento do array; incrementa i a cada iteração
for (var i = 0; i < numeros.length; i++) {
    // Imprime no console o índice da iteração e o valor do elemento correspondente
no array
    console.log("Elemento " + i + ": " + numeros[i]);
}

var numeros = [1, 2, 3, 4, 5];: Cria um array chamado números contendo os números
de 1 a 5.

for (var i = 0; i < numeros.length; i++) {: Inicia um loop for que inicializa uma
variável i com 0, continua enquanto i for menor que o comprimento do array numeros,
e incrementa i a cada iteração.

console.log("Elemento " + i + ": " + numeros[i]);: Dentro do loop, imprime no
console o índice da iteração (i) e o valor do elemento correspondente no array
(numeros[i]).

O loop percorre o array, e a cada iteração, imprime no console o índice da iteração
e o valor do elemento correspondente no array.

Saída :
Elemento 0: 1
Elemento 1: 2
Elemento 2: 3
Elemento 3: 4
Elemento 4: 5
```

Agora que você já está por dentro do que é iteração, vamos conhecer melhor as estruturas de repetição, começando pela estrutura **for**. Se liga no quadro a seguir!

For

Essa estrutura é ideal para iterações onde o **número de repetições é conhecido**. Ela possui três partes essenciais: **inicialização**, **condição de continuação** e **incremento/decremento**. Vejamos um exemplo:

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

Neste exemplo, o **loop for** imprimirá cinco vezes a mensagem, indicando a iteração atual.

Para entender melhor a aplicação dessa estrutura de repetição, vamos analisar linha por linha o código mostrado no exemplo anterior.

- De início, nós vemos um código JavaScript em que existe um *loop* for que **itera cinco vezes**;
- Na primeira linha, percebe-se a inicialização de uma variável chamada **i** com o **valor 0**. **O *loop* continuará enquanto i for menor que 5**. A cada iteração, o valor de **i** é incrementado em 1;
- Dentro do *loop*, na segunda linha, estamos usando **console.log** para imprimir uma mensagem no console. Essa mensagem inclui o texto **"Iteração número"** seguido do valor atual de **i + 1**. Portanto, o código imprimirá "**Iteração número 1**", "**Iteração número 2**", ..., até "**Iteração número 5**".

Basicamente, esse código está contando de 1 a 5 no console. É uma maneira simples de demonstrar como um *loop* **for** funciona em JavaScript.

While

Essa estrutura é útil quando o **número de repetições não é conhecido antecipadamente**, dependendo de uma condição. A condição é verificada antes da execução do bloco de código. Exemplo:

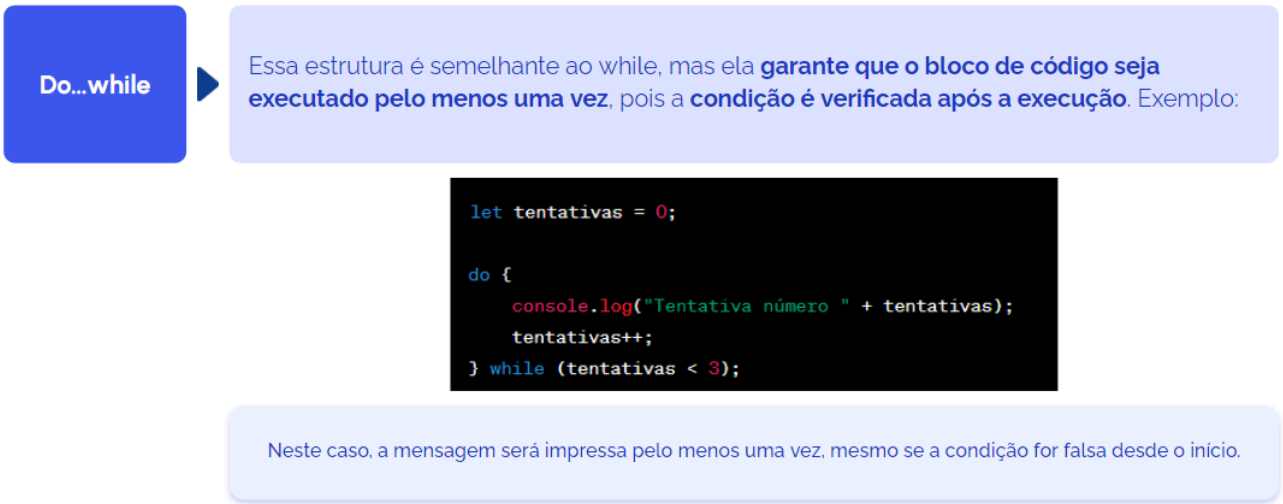
```
let contador = 0;

while (contador < 3) {
    console.log("Iteração número " + contador);
    contador++;
}
```

O **loop while** imprimirá a mensagem enquanto a condição contador < 3 for verdadeira.

Novamente, vamos entender o exemplo anterior analisando o código linha por linha:

- esse código itera até que a variável contador atinja o valor de 3;
- **let contador = 0;**: aqui, é inicializada uma variável chamada **contador** com o **valor 0**;
- **while (contador < 3) {**: o *loop* **while** continuará sendo executado enquanto o valor de **contador** for **menor que 3**;
- **console.log("Iteração número " + contador);**: dentro do *loop*, está sendo usado **console.log** para imprimir uma mensagem no console. Essa mensagem inclui o texto **"Iteração número" seguido do valor atual de contador**. Portanto, o código imprimirá "**Iteração número 0**", "**Iteração número 1**" e "**Iteração número 2**";
- **contador++;**: após cada iteração, o valor de contador é incrementado em 1.

Após essa análise, é possível perceber que esse código está contando de 0 a 2 no console.



Vamos entender cada passo da aplicação da estrutura de repetição **do...while** com base no exemplo mostrado anteriormente:

- **let tentativas = 0;**: aqui, estamos inicializando uma variável chamada **tentativas** com o valor 0;
- **do { ... } while (tentativas < 3);**: o *loop* **do...while** continuará sendo executado enquanto o valor de tentativas for menor que 3;
- **console.log("Tentativa número " + tentativas);**: dentro do *loop*, estamos usando **console.log** para imprimir uma mensagem no console. Essa mensagem inclui o texto **"Tentativa número"** seguido do valor atual de tentativas. Ou seja, o código vai imprimir "**Tentativa número 0**", "**Tentativa número 1**" e "**Tentativa número 2**";

- **tentativas++;**: após cada iteração, o valor de tentativas é incrementado em 1.

Se liga!

A diferença entre um loop do...while e um loop while é que o do...while executa o bloco de código pelo menos uma vez, **mesmo que a condição seja falsa desde o início**.

As estruturas de repetição em JavaScript são ferramentas essenciais, pois possibilitam a **execução repetida de blocos de código com base em condições específicas**. Elas oferecem flexibilidade e controle para **lidar com tarefas repetitivas** de maneira eficiente.

Ao compreender e aplicar esses conceitos, você poderá criar códigos mais dinâmicos e otimizados, melhorando a eficiência e a legibilidade dos seus programas em JavaScript.

4.2 Trabalhando com *arrays* e *loops*

Manipular *arrays* e implementar *loops* é uma habilidade fundamental em JavaScript, sendo essencial para lidar com conjuntos de dados e automatizar tarefas repetitivas. Os ***loops***, também conhecidos como **estruturas de repetição**, são utilizadas para executar um bloco de código várias vezes, tendo como base uma condição específica. Já o ***array*** é uma **estrutura de dados** que permite **armazenar e organizar coleções de elementos**. Pense nessa estrutura como uma caixa, na qual você pode guardar várias coisas. Em JavaScript, esses elementos podem ser de qualquer tipo de dado, como números, *strings*, objetos ou, até mesmo, outros *arrays*. Um *array* é definido utilizando colchetes – **[]** –, e os elementos dentro dele são separados por vírgulas.

Ao combinar *arrays* e *loops*, você pode criar um **sistema eficiente para lidar com muitos dados de uma vez**. Por exemplo, se tivermos uma lista de tarefas, um *array* poderia

armazenar essas tarefas em caixas específicas e um *loop* poderia executar uma ação para cada tarefa da lista, como mostrar uma tarefa na tela ou marcar uma tarefa como concluída.

Disponível em: https://encurtador.com.br/gkW17. Acesso em: 16 fev. 2024.

Usar essa combinação é como ter um assistente pessoal que organiza suas coisas (*arrays*) e realiza tarefas repetitivas para você (*loops*). Isso não apenas torna seu código mais limpo e fácil de entender, mas também economiza tempo, permitindo que você lide com grandes conjuntos de dados de maneira eficiente.

Agora que você já conheceu como essas duas estruturas funcionam, vamos entender como trabalhar efetivamente com os *arrays* usando as estruturas de repetição, a fim de você compreender melhor esses dois conceitos.

Entendendo arrays em JavaScript

Arrays são estruturas de dados que armazenam elementos de forma ordenada. Para criar um array utilizamos **colchetes**. Exemplo:

```
let frutas = ['Maçã', 'Banana', 'Morango', 'Uva'];
```

Utilizando o loop for para percorrer arrays

A estrutura for é frequentemente empregada para **iterar sobre os elementos de um array**. Exemplo:

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

Neste exemplo, o loop for percorre cada elemento do array **frutas**, exibindo-os no console.

Implementando o loop forEach para arrays

O método **forEach** proporciona uma abordagem mais elegante e legível para percorrer arrays. Exemplo:

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

O **forEach** executa uma função para cada elemento do array, simplificando o código.

Trabalhando com loops for... of para arrays

O loop **for... of** é uma opção moderna e concisa para **percorrer arrays**. Exemplo:

```
for (let fruta of frutas) {
    console.log(fruta);
}
```

Este loop simplifica ainda mais a iteração sobre os elementos do array **frutas**.

Aplicando manipulações em arrays

Manipular arrays envolve **adicionar**, **remover** e **modificar** elementos. Veja um exemplo de adição:

```
frutas.push('Pêssego');
```

Este código adiciona **'Pêssego'** ao final do array **frutas**.

Filtrando e mapeando arrays

Métodos como **filter** e **map** permitem **filtrar** e **transformar** elementos de um array. Analise esse exemplo de uso do **map**:

```
let frutasUpperCase = frutas.map(function(fruta) {
    return fruta.toUpperCase();
});
```

Neste caso, o **map** cria um novo array com todas as frutas em letras maiúsculas (UpperCase).

4.3 Resolvendo problemas utilizando as estruturas de controle

Em JavaScript, assim como em outras linguagens de programação, as estruturas de controle são ferramentas essenciais para resolver problemas de maneira eficaz. Elas ajudam a **controlar o fluxo** do programa, permitindo que você **tome decisões** e **repita ações** conforme for necessário. Veja no mapa mental a seguir algumas das principais estruturas de controle da linguagem JavaScript:

Vamos refletir

Você já conhecia alguma dessas estruturas de controle?
Elas fornecem a flexibilidade necessária para criar lógica condicional, loops e iterações, permitindo que você desenvolva programas mais poderosos e eficientes em JavaScript.

Analisando um exemplo

Agora que você já tem todas as informações sobre as estruturas de controle em mãos, **imagine que vamos criar um programa simples, que verifica números, por exemplo, se um número é par, ímpar ou se é um número é primo.** Vamos analisar, passo a passo, esta situação, aplicando as estruturas de controle para resolver problemas comuns durante o desenvolvimento deste programa em JavaScript.

1 **Definindo o problema:** começamos identificando o problema que desejamos resolver, que é **criar um programa simples que verifica números.** Primeiro, queremos que ele **analise se um número é par ou ímpar**. Qual estrutura de controle você utilizaria neste caso?

2 **Hora da decisão:** para **verificar se um número é par ou ímpar**, nosso programa precisa tomar algumas decisões, é nessa hora que devemos utilizar a **estrutura if**, pois ela é fundamental para tomar **decisões condicionais**. Então, vamos usá-la para verificar se um número é par ou ímpar:

```
let numero = 10;

if (numero % 2 === 0) {
    console.log("O número é par.");
} else {
    console.log("O número é ímpar.");
}
```

Aqui, o operador % calcula o resto da divisão por 2 para determinar se o número é par.

3 **Aplicando estruturas de controle for para iterações:** agora, queremos que nosso programa **imprima os número de 1 a 5**. Para isso, utilizamos a **estrutura for**, que irá nos ajudar a **repetir uma ou mais instruções** em um programa (iteração).

```
for (let i = 1; i <= 5; i++) {
    console.log(i);
}
```

O loop **for** nos permite **repetir uma ação por um número específico de vezes**. No caso do nosso programa, ele irá imprimir os números de 1 a 5.

4 **Resolvendo problemas com estruturas de repetição while:** agora, nós queremos que nosso programa **encontre o primeiro múltiplo de 3 maior que 20**, como você escreveria isso em um código JavaScript? Acompanhe na imagem:

```
let numero = 20;

while (numero % 3 !== 0) {
    numero++;
}

console.log("O primeiro múltiplo de 3 maior que 20 é:", numero);
```

Perceba que o loop **while** permite que continuemos incrementando **numero** até encontrarmos o múltiplo de 3.

5 **Integrando estruturas de controle para problemas complexos:** durante o desenvolvimento de um programa muitas vezes precisamos **combinar de várias estruturas de controle**. Vamos ver como isso acontece na prática, com base no nosso programa. Agora, queremos que ele **encontre os números primos até 20**. Como devemos fazer isso? Acompanhe o código abaixo:

```
for (let i = 2; i <= 20; i++) {
    let primo = true;

    for (let j = 2; j < i; j++) {
        if (i % j === 0) {
            primo = false;
            break;
        }
    }

    if (primo) {
        console.log(i);
    }
}
```

Perceba que houve a combinação dos loops **for** e **if** para identificar e imprimir números primos até 20.

Após estudar sobre as estruturas de controle em programação, é possível perceber como essas ferramentas são como os superpoderes que o desenvolvedor tem em mãos no mundo da programação. Com essas estruturas, é possível criar um código que toma decisões e faz as coisas se repetirem de um jeito otimizado e eficaz.

DESAFIO PRÁTICO

Otimizando Processos de Estoque em uma Loja Virtual

Descrição

Você é parte da equipe de desenvolvimento de uma loja virtual de produtos eletrônicos. Recentemente, a empresa identificou a necessidade de otimizar os processos relacionados ao controle de estoque. O desafio é lidar com um grande volume de produtos, gerenciar atualizações de inventário e identificar itens que precisam ser reabastecidos.

A loja possui uma ampla variedade de produtos, cada um com seu próprio nível de estoque. A gestão de inventário envolve monitorar as vendas, receber novos produtos e atualizar as quantidades disponíveis. A equipe deseja implementar melhorias nas estruturas de repetição para facilitar essas operações.

Objetivos

- Implementar um loop "for" para percorrer a lista de produtos e atualizar o estoque com base nas vendas diárias.
- Utilizar um loop "while" para monitorar continuamente as vendas e atualizar o estoque em tempo real.
- Aplicar um loop "do...while" para identificar produtos com estoque abaixo do limite mínimo e acionar automaticamente o processo de reabastecimento.

Orientações

- Certificar-se de que o loop "for" é eficiente para percorrer a lista de produtos e aplicar as atualizações necessárias.
- Implementar o loop "while" de forma a monitorar continuamente as vendas, garantindo que o estoque seja atualizado em tempo real.
- Utilizar o loop "do...while" para verificar regularmente se há produtos abaixo do limite mínimo de estoque e acionar o reabastecimento, assegurando uma gestão proativa do inventário.
- Proporcionar feedback claro ao usuário sobre as atualizações de estoque, garantindo transparência nas operações.

DESAFIO PRÁTICO II

Catálogo de produtos em uma loja virtual

Descrição

Você está colaborando com uma equipe de desenvolvimento em uma loja virtual que deseja aprimorar o catálogo de produtos. Os gestores deram a orientação que o catálogo precisa ser mais dinâmico, permitindo uma fácil adição e remoção de itens. A equipe identificou a necessidade de trabalhar com *arrays* e *loops* para otimizar essas operações.

A loja possui diversas categorias de produtos, como eletrônicos, moda e decoração. O catálogo atual é estático, tornando difícil a inclusão de novos produtos e a remoção de itens descontinuados. A ideia é criar um sistema mais flexível e eficiente usando *arrays* e *loops*.

Objetivos

- Criar um *array* que representa o catálogo de produtos, inicialmente com alguns itens pré-existentes;
- Implementar um *loop* **for** para adicionar facilmente novos produtos ao catálogo;
- Utilizar um *loop* **for** para identificar e remover produtos descontinuados do catálogo.

Orientações

- Inicie o *array* com alguns produtos já existentes na loja, representando diferentes categorias;
- Desenvolva um *loop* **for** que permita a adição de novos produtos ao catálogo de forma simples e eficiente;
- Implemente outro *loop* **for** que verifique os produtos existentes, removendo aqueles que estão descontinuados;
- Forneça *feedback* ao usuário após a adição ou remoção de produtos, garantindo uma experiência transparente.

Sistema de avaliação de desempenho escolar

Descrição

Você está trabalhando em um projeto para criar um sistema simples de avaliação de desempenho escolar. O objetivo é utilizar estruturas de controle para calcular médias e determinar o status de aprovação dos alunos.

O sistema será usado por professores para inserir as notas dos alunos em diferentes disciplinas. A ideia é calcular a média geral de cada aluno e indicar se eles foram aprovados ou reprovados com base em critérios simples.

Objetivos

- Desenvolver uma estrutura que permita a inserção das notas dos alunos em diferentes disciplinas;
- Utilizar estruturas de controle para calcular a média geral de cada aluno;
- Determinar se o aluno foi aprovado ou reprovado tendo como base a média calculada.

Orientações

- Crie um sistema que permita aos professores inserirem as notas de cada aluno em diferentes disciplinas;
- Utilize estruturas de controle, como *loops* e condicionais, para calcular a média geral de cada aluno;
- Estabeleça critérios simples para determinar se um aluno foi aprovado ou reprovado com base na média obtida;
- Forneça *feedback* claro aos professores sobre o desempenho de cada aluno, destacando aqueles que precisam de atenção adicional.

DESAFIO PRÁTICO IV

Otimização de rotas para entregas logísticas

Descrição

Você foi contratado por uma empresa de logística para otimizar as rotas de entrega. O desafio é resolver problemas complexos com estruturas de controle para garantir que as entregas sejam feitas de maneira eficiente, considerando diferentes variáveis.

A empresa realiza entregas em uma cidade com ruas complexas e diferentes condições de tráfego. O objetivo é encontrar a rota mais eficiente para cada entrega, minimizando o tempo e os custos associados.

Objetivos

- Desenvolver uma estrutura robusta para entrada de dados, considerando aspectos como endereços de entrega, condições de tráfego e restrições específicas;
- Utilizar estruturas de controle avançadas para calcular rotas otimizadas, levando em consideração variáveis dinâmicas como o tráfego em tempo real;
- Implementar uma lógica que priorize as entregas com base em critérios específicos, como prazos e prioridades dos clientes;
- Criar uma estrutura para lidar com situações excepcionais, como ruas bloqueadas, fornecendo rotas alternativas.

Orientações

- Desenvolva um sistema que permita a entrada flexível de dados, incluindo endereços de entrega, informações sobre o tráfego e prioridades dos clientes;
- Utilize algoritmos avançados para calcular as rotas mais eficientes, levando em consideração o tráfego em tempo real;

- Implemente uma lógica de priorização que considere prazos de entrega, prioridades dos clientes e outras variáveis relevantes;
- Antecipe possíveis problemas, como ruas bloqueadas, e forneça rotas alternativas em tempo real;
- Forneça relatórios detalhados sobre as rotas otimizadas, tempos de entrega estimados e possíveis problemas encontrados durante o processo.

Exploramos a eficiência em JavaScript através de três tópicos essenciais: **estruturas de repetição, *arrays* e estruturas de controle**. Começamos entendendo como usar as estruturas **for** e **while** para repetir ações de forma eficiente, permitindo-nos executar blocos de código várias vezes sem repetir linhas desnecessárias.

Em seguida, mergulhamos no mundo dos ***arrays***, aprendendo como essas estruturas de dados nos ajudam a organizar e manipular conjuntos de informações. Também estudamos o poder das estruturas de controle, como o **if** e o **else**, para a tomada de decisões inteligentes em nosso código. Ao final, percebemos como esses conceitos se entrelaçam, permitindo a criação de códigos mais robustos e eficientes. Com essa bagagem, você está pronto para enfrentar desafios mais complexos e escrever códigos JavaScript.

ATIVIDADE DE FIXAÇÃO

1. Explique a diferença entre as estruturas de repetição **for**, **while** e **do...while** em JavaScript, e dê um exemplo prático de situação em que cada uma seria mais apropriada.

2. Como você percorreria todos os elementos de um *array* em JavaScript usando a estrutura **for**? Compare essa abordagem com o uso do método **forEach**.

3. Descreva duas maneiras de adicionar um elemento ao final de um *array* em JavaScript. Em que momentos você escolheria uma no lugar da outra?

4. Crie uma estrutura condicional que verifique se um número é positivo, negativo ou zero. Como você aplicaria isso para resolver problemas práticos?

5. Explique como você usaria um *loop* **while** para encontrar o menor múltiplo de 7 maior que 100.

6. Implemente um *loop* **for** que itera sobre os índices de um *array* e exibe os elementos pares. Como você adaptaria isso para exibir os elementos ímpares?

7. Em que situação você escolheria usar um *loop* **do...while** ao invés de um **while**? Dê um exemplo prático.

8. Como você usaria um *loop* para encontrar a média dos valores em um *array* numérico em JavaScript? Considere a variedade de métodos disponíveis para realizar essa tarefa.

9. Desenvolva um programa que, utilizando estruturas de controle, identifica e imprime todos os números primos entre 1 e 50.

10. Em um cenário real de desenvolvimento *web*, como você usaria estruturas de repetição e *arrays* para otimizar a exibição dinâmica de conteúdo em uma página? Explique a sua abordagem.

CAPÍTULO 05

MANIPULAÇÃO DO DOCUMENTO COM JAVASCRIPT

O que esperar deste capítulo:

- Manipular dinamicamente o conteúdo de elementos HTML no DOM;
- Manipular, de modo avançado, propriedades e atributos de elementos HTML;
- Trabalhar com eventos no DOM, como cliques, teclas pressionadas, alterações de formulário etc.

5.1 Acesso a elementos HTML usando o Document Object Model (DOM)

O *Document Object Model* (DOM), em português, Modelo de Objeto de Documento, é uma **interface de programação que representa a estrutura hierárquica dos elementos HTML** em uma página *web*. A capacidade de **acessar e manipular** esses elementos através do DOM é fundamental para a construção de aplicações *web* dinâmicas e interativas.

O DOM é como uma árvore de nós que representa a estrutura do documento HTML, na qual cada elemento, atributo e texto no HTML é representado como um nó. No início, temos o nó raiz representando todo o documento e, a partir dele, podemos navegar pelos nós descendentes para acessar elementos específicos. Para que você entenda como o DOM funciona na prática, observe o código HTML a seguir.

```html
<!DOCTYPE html>
<html>
<head>
  <title>Exemplo DOM</title>
</head>
<body>
  <div id="conteudo">
    <p class="destaque">Este é um parágrafo de destaque.</p>
  </div>
</body>
</html>
```

5.1.1 Entendendo um código HTML simples

A seguir, vamos analisar passo a passo o que esse código cria:

1. começa com a declaração do tipo de documento (**<!DOCTYPE html>**);
2. em seguida, nós temos a abertura da tag **<html>**, que encapsula todos os outros elementos;
3. dentro da tag **<html>**, há uma seção **<head>** contendo uma tag **<title>** com o texto "Exemplo DOM";
4. a seção **<body>** contém um elemento ** ** com um atributo **id** definido como "**conteudo**";
5. dentro dessa ** **, nós encontramos um parágrafo (**<p>**) com a classe **destaque** e o texto "Este é um parágrafo de destaque".

Resultado visual:

- esse código cria uma página HTML simples com um parágrafo destacado, como podemos ver na imagem a seguir:

Este é um parágrafo de destaque.

- a classe **destaque** pode ser estilizada utilizando CSS para alterar a sua aparência na página.

A partir desse exemplo, conseguimos visualizar perfeitamente a estrutura de um código HTML. Cada elemento, seção ou atributo é determinado pelo desenvolvedor que escreve o código. A partir do **<html>** (nó raiz), é possível navegar por todos os outros elementos (nós) na estrutura (árvore) do código.

Disponível em: https://encurtador.com.br/eioNO. Acesso em: 19 fev. 2024.

5.1.2 Seleção de elementos no DOM

Existem várias maneiras de selecionar elementos no DOM. Porém, os métodos clássicos incluem **getElementById**, **getElementsByClassName** e **getElementsByTagName**. Além desses, o moderno **querySelector** permite que sejam feitas seleções mais flexíveis usando seletores CSS. A seleção de elementos permite que os desenvolvedores acessem e manipulem esses elementos de forma dinâmica por meio de linguagens de programação como JavaScript. Ao escolher o método de seleção adequado, é possível obter referências precisas aos elementos que desejamos acessar. Veja um exemplo em que o **querySelector** é utilizado para selecionar elementos:

```
let paragrafoDestaque = document.querySelector('.destaque');
console.log(paragrafoDestaque.textContent);  // saída do parágrafo
```

- **Comentário inicial:** o comentário **// Selecionando por classe** indica que o código está selecionando um elemento HTML com a classe "destaque";
- **Variável paragrafoDestaque:** uma variável chamada **paragrafoDestaque** é declarada e recebe o elemento HTML com a classe "destaque" usando **document.querySelector('.destaque')**;
- **Console Log:** o conteúdo interno (**innerHTML**) desse elemento é exibido no console usando **console.log(paragrafoDestaque.innerHTML)**. A saída, então, será "Este é um parágrafo".

Basicamente, esse código busca um elemento HTML com a classe "destaque" e imprime o conteúdo desse elemento no console. Essa é uma operação bastante comum em JavaScript para manipular elementos de uma página.

Assista ao vídeo a seguir, pois ele vai te ajudar a entender melhor como funciona a manipulação de elementos no DOM.

Vídeo: Curso de JS Aula 08 DOM Manipular HTML com o JavaScript.

Fonte: https://www.youtube.com/watch?v=0dBY09OJm04. Acesso em: 21 fev. 2024.

Pausa para refletir

O que me anima?	O que me preocupa?	O que eu preciso explorar?
O que te deixou animado(a) sobre este assunto?	Eita! Qual parte te preocupa e por que?	O que você sente que precisa explorar sobre esse assunto?

5.1.3 Utilizando métodos de seleção de forma eficiente

Ao selecionar elementos, é importante considerar a eficiência desta ação. Métodos como o **querySelector** podem lidar com **seleções mais complexas**, enquanto o **getElementById** é extremamente eficiente para **identificadores únicos**. A escolha do método depende do contexto e da complexidade da seleção desejada. Veja um exemplo:

```
// Selecionando por ID
let conteudoDiv = document.getElementById('conteudo');

// Exibindo o conteúdo interno no console
console.log(conteudoDiv.innerHTML); // Saída: <p class="destaque">Este</p>
```

Este trecho de código JavaScript seleciona um elemento HTML com o ID **“conteudo”** e exibe o seu conteúdo interno. Nesse caso, a saída será ** Este **. Isso significa que o elemento com o ID **“conteudo”** contém um parágrafo destacado com o texto **“Este”**. O método **getElementById** é usado para **encontrar o elemento com o ID especificado na página HTML**. Quando você executa esse código em uma página que possui um elemento com o ID “conteudo”, ele mostrará o conteúdo desse elemento no console. Viu como é fácil? Agora, que tal um desafio?

Hora do desafio!

Crie uma **lista de tarefas interativa** em que os usuários podem adicionar, marcar como concluída e excluir tarefas.

Requisitos necessários para cumprir o desafio

- Crie uma página HTML com um **campo de entrada** para a tarefa, um **botão de adição** e uma **lista para exibir as tarefas**.
- Utilize **getElementById**, **getElementsByClassName** e **getElementsByTagName** para interagir com os elementos do DOM.
- Ao clicar no botão de adição, a **tarefa digitada deve ser adicionada à lista**.
- Cada tarefa na lista deve ter uma opção para ser **marcada como concluída**.
- Adicione a **funcionalidade de exclusão** para remover uma tarefa da lista.

Utilize um ambiente de desenvolvimento de sua preferência, como o CodePen (https://codepen.io/) ou o Visual Studio Code (https://code.visualstudio.com/).

É hora de decolar!
Esse desafio prático permitirá que você pratique a seleção de elementos no DOM para criar uma aplicação básica de lista de tarefas, proporcionando uma experiência interativa ao adicionar tarefas, marcá-las como concluídas e excluí-las.

5.1.4 Navegando pela estrutura DOM

Ao lidar com o DOM, nós navegamos entre elementos, atributos e texto no HTML. Esses elementos, assim como discutimos anteriormente, se assemelham a nós em uma estrutura de árvore. Além dos métodos de seleção, utilizamos propriedades para percorrer essa árvore DOM de maneira eficaz. Algumas dessas propriedades incluem:

1. **parentNode**: permite acessar o elemento pai de um nó específico;
2. **childNodes**: retorna uma lista de todos os filhos de um nó;
3. **nextSibling**: acessa o próximo irmão de um nó;
4. **previousSibling**: acessa o irmão anterior de um nó.

Essas propriedades nos oferecem um leque de possibilidades para interagir com os elementos da página. Vamos acompanhar o exemplo a seguir:

```
let paragrafoDestaque = document.querySelector('.destaque');
let paiDoParagrafo = paragrafoDestaque.parentNode;
console.log(paiDoParagrafo.tagName);  // Saída: DIV
```

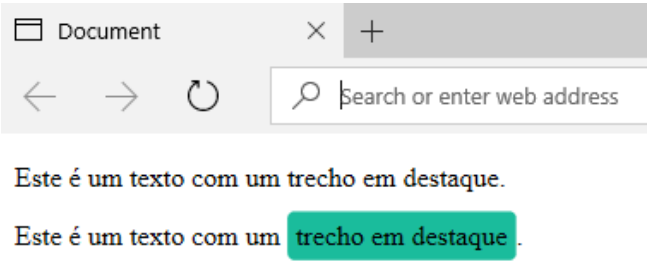
Este código JavaScript está realizando as operações a seguir.

1. **Seleção do elemento**
   - Primeiro, ele seleciona um elemento HTML com a classe **destaque** usando o método **document.querySelector('.destaque')**;
   - Esse elemento pode ser um parágrafo, uma imagem ou qualquer outro elemento com a classe **destaque**.
2. **Acesso ao nó pai (*parent node*)**
   - Em seguida, ele obtém o nó pai desse elemento usando a propriedade **parentNode**;
   - O nó pai é o elemento HTML que contém o elemento com a classe **destaque**.
3. **Impressão no console**

○ Por fim, ele imprime no console a etiqueta (*tag*) do nó pai usando **console.log(paiDoParagrafo.tagName)**;

○ A saída esperada é **DIV**, indicando que o nó pai é um elemento ** **. Ela é frequentemente usada para criar divisões ou contêineres genéricos em uma página *web*, agrupando e estruturando outros elementos HTML.

○ Imagine que temos um parágrafo destacado em nosso HTML e queremos saber qual é o elemento pai dele. A linguagem JavaScript nos permite acessar o DOM e manipular os elementos da página, como no caso do exemplo que acabamos de ver.

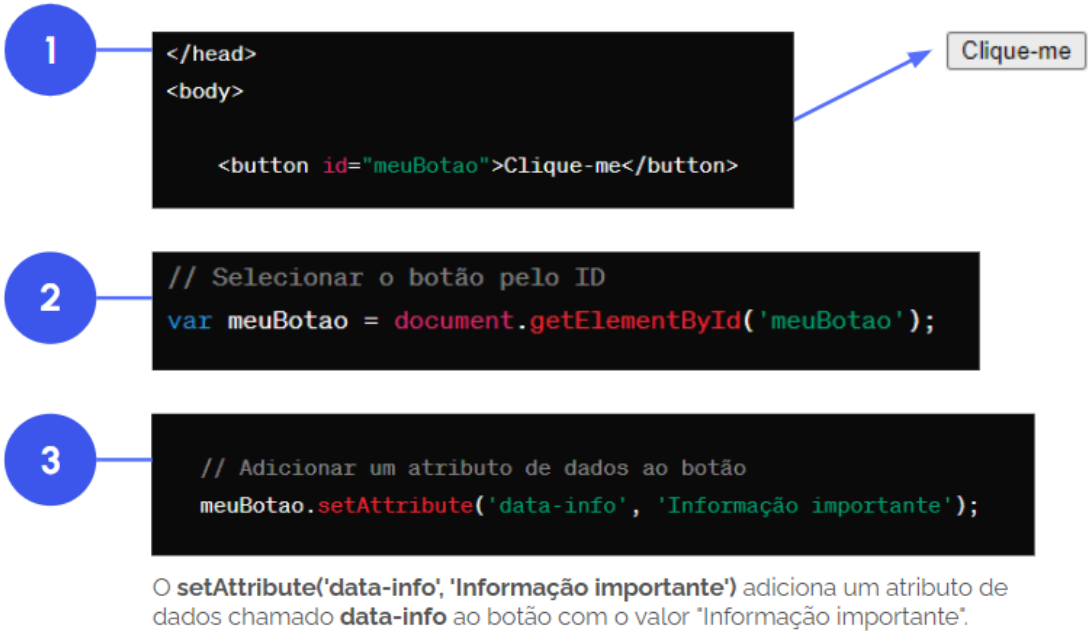


Fonte: https://www.devmedia.com.br/html-basico-codigos-html/16596. Acesso em: 21 fev. 2024.

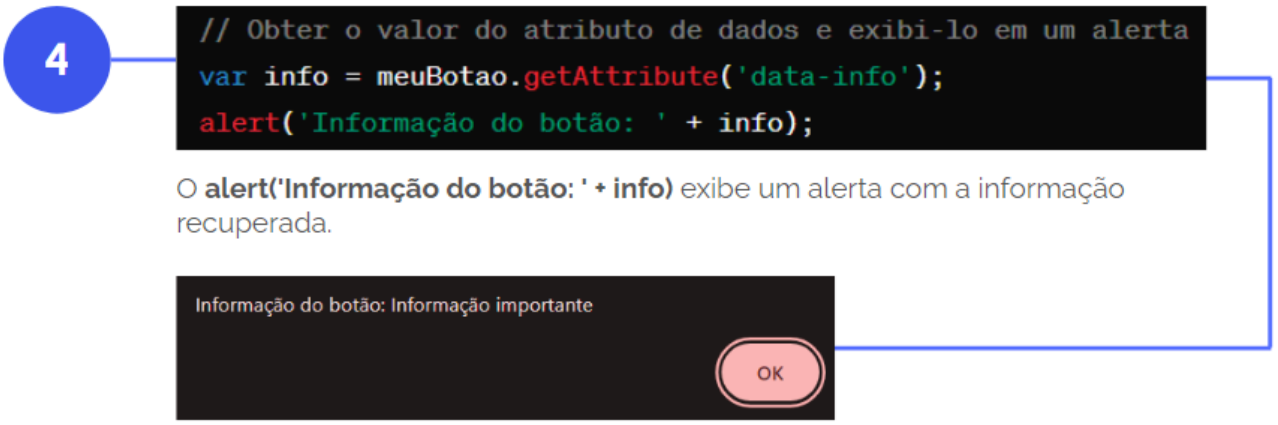
5.1.5 Manipulação de conteúdo e atributos

Uma vez que temos acesso aos elementos no DOM, é possível manipular o seu conteúdo e os seus atributos. Para isso, utilizamos a propriedade **innerHTML**, que permite **modificar o conteúdo interno do DOM**, enquanto **setAttribute** e **getAttribute** possibilitam a **manipulação de atributos**. Essas operações dinâmicas são cruciais para atualizar a interface do usuário em tempo real.

Suponha que você queira criar um botão em uma página HTML e deseje adicionar um **atributo de dados** a ele para armazenar informações adicionais. Depois, você quer recuperar esse valor e exibi-lo na tela quando esse botão for clicado. Ao utilizar a propriedade **setAttribute**, você consegue adicionar um atributo de dados ao botão. Veja como isso ficaria em um código HTML:



Para recuperar o valor do atributo de dados quando o botão é clicado, o **getAttribute** deve ser usado. Vamos ver como isso ficaria no nosso código:



Esse exemplo demonstra, de forma simples, como podemos usar as propriedades **setAttribute** e **getAttribute** para manipular atributos de elementos HTML em situações práticas.

Além da manipulação dos atributos, também é possível **modificar o conteúdo interno do código**. Para isso, utilizamos o **innerHTML**, uma propriedade do DOM em JavaScript que **representa o conteúdo HTML interno de um elemento**. Essa propriedade permite ler ou

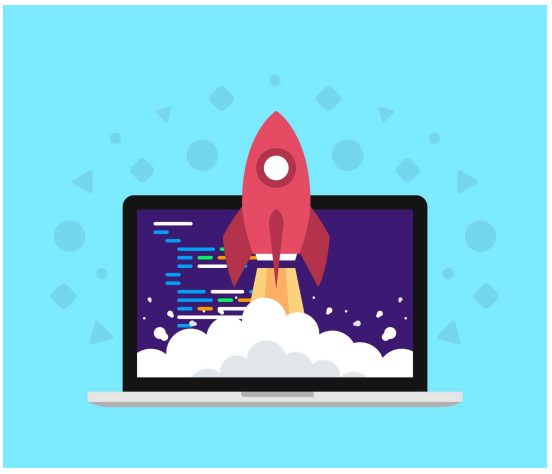
modificar o conteúdo HTML de um elemento diretamente por meio de um código JavaScript. Veja um exemplo:

```
paragrafoDestaque.innerHTML = 'Texto modificado dinamicamente.';
console.log(paragrafoDestaque.innerHTML); // Saída: Texto modificado
```

A seguir, vamos analisar esse código juntos:

- é atribuído um novo conteúdo à propriedade **innerHTML** de um elemento HTML com o ID ou classe "paragrafoDestaque";
- a linha de código é **paragrafoDestaque.innerHTML = 'Texto modificado dinamicamente.';**, o que significa que o texto dentro desse elemento será substituído por "Texto modificado dinamicamente.";
- em seguida, o código utiliza **console.log()** para exibir o novo conteúdo da propriedade **innerHTML** do elemento "paragrafoDestaque";
- dessa forma, a saída esperada é **"Texto modificado"**.

A propriedade **innerHTML** é muito útil quando se quer construir interfaces de usuário dinâmicas, já que ela é usada para atualizar o conteúdo de elementos, inserir novos elementos HTML ou renderizar dados dinâmicos.



Disponível em: https://encurtador.com.br/apPR2. Acesso em: 21 fev. 2024.

5.1.6 Práticas recomendadas e considerações de desempenho

Ao acessar elementos no DOM, é vital seguir práticas recomendadas. Veja quais são algumas delas no quadro a seguir.

1. **Evitar manipulações excessivas:** realizar muitas manipulações, especialmente em elementos visíveis na página, pode levar a uma baixa performance. Em vez disso, agrupe várias manipulações em uma única operação sempre que possível.

2. **Utilizar métodos eficientes de seleção:** utilize métodos nativos de seleção, como **getElementById**, **getElementsByClassName**, **querySelector**, pois, geralmente, eles são mais eficientes do que bibliotecas ou *frameworks* personalizados.

3. **Considerar a reutilização de referências a elementos:** armazene referências a elementos em variáveis e reutilize essas variáveis conforme necessário. Isso evita a necessidade de pesquisas repetidas no DOM, melhorando a eficiência e a legibilidade do código.

Importante!

Além dessas, **existem outras boas práticas** que garantem um desempenho otimizado. Pesquise para saber quais são elas e sempre mantenha seu código JavaScript eficiente e responsivo.

Parabéns!
Você acaba de dar os primeiros passos para dominar a manipulação dinâmica de páginas *web*. Continue nesse caminho, pois há muito mais para explorar. Vamos codificar juntos e descobrir as infinitas possibilidades que o DOM nos oferece!

5.2 Manipulação de elementos, propriedades e atributos

Assim como já vimos, a manipulação de elementos, propriedades e atributos no DOM permite a criação de páginas dinâmicas e interativas. Sendo assim, essa é uma habilidade crucial para que você se torne um(a) desenvolvedor(a) *web*. Para dar continuidade à nossa jornada, agora nós vamos abordar detalhadamente cada passo da manipulação no *Document Object Model* utilizando exemplos práticos.

Level up!

Se você quiser alcançar um nível ainda mais além na manipulação do DOM (*Document Object Model*), abra um ambiente de desenvolvimento de sua preferência, como o **CodePen** (https://codepen.io/) ou o **Visual Studio Code** (https://code.visualstudio.com/) e reproduza o passo a passo que veremos a seguir.

Passo 1 – Seleção de elementos no DOM

Para manipular elementos, é necessário selecioná-los primeiro. Os métodos de seleção mais comuns são:

- **getElementById:** seleciona um elemento pelo ID;
- **getElementsByClassName:** seleciona elementos por classe;
- **getElementsByTagName:** seleciona elementos por *tag*;
- **querySelector:** seleciona o primeiro elemento que corresponde ao seletor CSS.

Veja um exemplo:

```
let meuElemento = document.getElementById('meuId');
```

Nesse caso, o **getElementById** está sendo utilizado para selecionar um elemento pelo ID no código HTML.

Passo 2 – Manipulação de conteúdo e propriedades

Após a seleção do elemento, é viável manipular tanto seu conteúdo interno quanto suas propriedades. A propriedade **innerHTML** é empregada para modificar o conteúdo interno do elemento, ao passo que propriedades como **className** podem ser manipuladas para alterar as classes.

Observe o exemplo a seguir e não se esqueça de reproduzi-lo no seu ambiente de desenvolvimento!

```
meuElemento.innerHTML = 'Novo conteúdo!';
meuElemento.className = 'novaClasse';
```

Após a execução desse código, o **elemento** referenciado por **meuElemento** terá o seu conteúdo interno alterado para **'Novo conteúdo!'** e a sua **classe** será definida como **'novaClasse'**. Isso é útil para modificar dinamicamente a aparência e o conteúdo de elementos HTML em resposta a eventos, interações do usuário ou outras condições em uma aplicação *web*.

Passo 3 – Manipulação de atributos

A manipulação de atributos é feita usando os métodos **setAttribute** e **getAttribute**. Dessa forma, podemos adicionar ou modificar atributos, assim como recuperar os seus valores. Acompanhe o exemplo a seguir:

```
meuElemento.setAttribute('data-novo-atributo', 'valorAtributo');
let valorAtributo = meuElemento.getAttribute('data-novo-atributo');
```

Este código adiciona um novo atributo chamado **'data-novo-atributo'** ao elemento **meuElemento** com o valor **'valorAtributo'** e, em seguida, recupera esse valor usando **getAttribute** e o armazena na variável **valorAtributo**. Normalmente, isto é usado para associar dados personalizados a elementos HTML, especialmente quando os dados não são visíveis ou não devem ser apresentados diretamente no conteúdo da página.

Pausa para refletir

Você está conseguindo entender como usamos o JavaScript para mexer no conteúdo e no estilo de elementos em páginas *web*? É como dar uma cara nova para as coisas na tela! Além disso, lembre-se: ao fazer a manipulação do DOM, tenha sempre em mente as boas práticas para manter o desempenho do seu código otimizado.

Passo 4 – Estilo dinâmico com CSS

Estilos podem ser manipulados dinamicamente usando a propriedade **style**. Com ela, podemos alterar qualquer propriedade de estilo, como a **cor de fundo** ou o **tamanho da fonte**. Aplique o exemplo a seguir no seu ambiente de desenvolvimento:

```
meuElemento.style.backgroundColor = 'blue';
```

Esse código faz com que o **fundo** do elemento **meuElemento** seja alterado para a **cor azul**. Essa é uma maneira simples e direta de aplicar estilos diretamente no JavaScript, pois a propriedade **style** permite acessar e manipular as propriedades de estilo CSS de um elemento HTML.

Passo 5 – Remoção e adição de elementos

Podemos **adicionar novos elementos ao DOM** com os métodos **createElement** e **appendChild**. Já a **remoção de elementos** é feita com o método **removeChild**. Veja como esses métodos são aplicados em um código:

```
let novoParagrafo = document.createElement('p');
novoParagrafo.innerHTML = 'Novo parágrafo adicionado!';
meuElemento.appendChild(novoParagrafo);

// Removendo o parágrafo
meuElemento.removeChild(novoParagrafo);
```

Percebeu o que este código faz? Ele **cria**, **adiciona** e, em seguida, **remove** um novo parágrafo de um elemento HTML referenciado pela variável **meuElemento**.

- **let novoParagrafo = document.createElement('p');:** cria um novo elemento **'p'** (parágrafo) no DOM usando o método **createElement**. Esse novo elemento é armazenado na variável **novoParagrafo**;

- **novoParagrafo.innerHTML = 'Novo parágrafo adicionado!';:** define o conteúdo interno do novo parágrafo criado anteriormente. Neste caso, o texto 'Novo parágrafo adicionado!' será o conteúdo do parágrafo;

- **meuElemento.appendChild(novoParagrafo);:** adiciona o novo parágrafo como filho do elemento referenciado por **meuElemento**. O método **appendChild** é usado para anexar um nó (no caso, o **novoParagrafo**) ao final da lista de filhos do nó pai (**meuElemento**);

- **meuElemento.removeChild(novoParagrafo);:** remove o parágrafo que foi adicionado anteriormente. O método **removeChild** é usado para remover um nó filho do nó pai.

Passo 6 – Práticas recomendadas e desempenho

Para criar páginas *web* rápidas e eficientes, é essencial seguir boas práticas ao manipular o DOM. **Evitar muitas manipulações**, **escolher métodos eficientes para selecionar elementos** e **reutilizar referências** tornam nosso código mais ágil. Isso não só melhora a experiência do usuário, mas também contribui para a construção de aplicações *web* mais robustas e eficientes. Ao adotar essas práticas, garantimos um código mais limpo e um desempenho otimizado nos nossos projetos!

RESUMO

Estudamos que a manipulação do documento com JavaScript é uma habilidade central no desenvolvimento *web*, permitindo que desenvolvedores modifiquem dinamicamente elementos HTML, estilos e atributos em uma página. Vimos que, utilizando o DOM, o JavaScript possibilita a **seleção** de elementos específicos, a **atualização** do conteúdo interno e a **manipulação** de atributos, proporcionando uma experiência interativa aos usuários.

Além disso, nós exploramos que a capacidade de criar e remover elementos, juntamente com a **gestão de eventos**, amplia as possibilidades de construção de interfaces *web* dinâmicas e responsivas, enquanto as **boas práticas recomendadas** garantem eficiência e desempenho otimizados. Com isso, conseguimos aprender que a manipulação do documento com JavaScript é essencial para o desenvolvimento de aplicações *web* interativas e adaptáveis.

Disponível em: https://encurtador.com.br/cqtUZ. 21 fev. 2024.

ATIVIDADE DE FIXAÇÃO

1. O que é o DOM (*Document Object Model*) e qual é o seu papel na manipulação do documento com JavaScript?
2. Quais são os principais métodos de seleção de elementos no DOM e como eles diferem nas suas abordagens?
3. Explique como a linguagem JavaScript permite a manipulação dinâmica do conteúdo interno de elementos HTML. Dê exemplos práticos.
4. Como os atributos de elementos HTML podem ser manipulados usando JavaScript? Forneça exemplos de adição, modificação e remoção de atributos.
5. Explique o processo de criação de novos elementos HTML no DOM com JavaScript. Por que isso é útil na construção de interfaces dinâmicas?
6. Qual é a importância da manipulação de estilos com JavaScript no contexto da construção de páginas *web* interativas?
7. Como os eventos são tratados em JavaScript no contexto da manipulação do documento? Dê exemplos de como os manipuladores de eventos podem ser utilizados.
8. Quais são algumas das práticas recomendadas para otimizar o desempenho ao manipular o documento com JavaScript?
9. Como o JavaScript permite a remoção eficiente de elementos do DOM? Explique a importância dessa capacidade no desenvolvimento *web*.
10. Por que a manipulação do documento com JavaScript é considerada essencial para criar experiências *web* mais dinâmicas e interativas? Explique.

CAPÍTULO 06

DESENVOLVENDO EXPERIÊNCIAS INTERATIVAS: EVENTOS E RESPOSTAS EM JAVASCRIPT

O que esperar deste capítulo:

- Compreender os conceitos fundamentais de eventos em JavaScript, incluindo tipos de eventos, propagação e manipulação;
- Desenvolver habilidades avançadas na manipulação de eventos, englobando a atribuição de manipuladores de eventos a elementos HTML;
- Aplicar as habilidades adquiridas na criação de interatividade em páginas *web* de forma prática.

6.1 Eventos JavaScript e como responder a ações do usuário

A programação de eventos em JavaScript desempenha um papel crucial na criação de experiências *web* interativas e responsivas. Os eventos são utilizados para **detectar ações do usuário** e **desencadear respostas específicas**. Por exemplo, quando os usuários interagem com uma página, seja **clicando em botões**, **digitando no teclado** ou **movendo o *mouse***, os eventos são acionados. Sendo assim, compreender como capturar e responder a eles é fundamental para quem trabalha com o desenvolvimento *web*.

Neste capítulo, nós vamos explorar em detalhes como trabalhar com eventos, desde a sua identificação até a implementação de respostas dinâmicas e eficazes.

6.1.1 Identificação de eventos e atribuição de manipuladores de eventos

Como vimos, os eventos em JavaScript são ações ou ocorrências que acontecem durante a execução de um programa ou a partir de uma interação do usuário com uma página *web*. Esses eventos podem ser desde interações do mouse, como cliques e movimentos, até eventos de teclado e manipulação de formulários. Dessa forma, a identificação desses eventos é essencial para personalizar a experiência do usuário de acordo com as suas ações em uma página *web*.

A atribuição de manipuladores de eventos é feita usando o método **addEventListener**. Este método permite que você especifique o **tipo de evento** que deseja observar e a **função** que deve ser executada quando esse evento ocorre. Isso permite uma abordagem mais modular e escalável do código que você estiver desenvolvendo. A seguir, veja um exemplo de código JavaScript que mostra a identificação e a resposta a um clique em um botão HTML:

```
// Exemplo de identificação e resposta a um clique em um botão
const meuBotao = document.getElementById('meuBotao');

meuBotao.addEventListener('click', function() {
    alert('Botão clicado!');
});
```

Observe a seguinte estrutura: **meuBotao.addEventListener('click', function() { alert('Botão clicado!'); });**. Nela, um **evento de clique ('click')** é adicionado ao botão identificado no início do código usando o método **addEventListener**. Este método recebe dois argumentos:

- **'click':** que especifica que a função a seguir será executada quando ocorrer um clique no botão;

- **function() { alert('Botão clicado!'); }:** é a função de retorno que será executada quando o evento de clique ocorrer. Neste caso, a função exibe um alerta na tela com a mensagem 'Botão clicado!'.

Depois da análise deste código, acho que ficou mais fácil de entender como funciona a manipulação de eventos em JavaScript, não é mesmo? Mas vamos continuar a nossa jornada, pois há muito mais para aprender pela frente!

6.1.2 Respostas dinâmicas e funcionalidades avançadas

As respostas dinâmicas estão relacionadas à capacidade de um sistema ou página *web* alterar o seu conteúdo ou comportamento em tempo real, geralmente em **resposta a eventos específicos**. Essa resposta pode ir além de simples alertas. Vamos conhecer algumas no quadro a seguir:

✓ Atualização dinâmica de conteúdo na página

Permite modificar o conteúdo de uma página *web* em tempo real, sem a necessidade de recarregar a página inteira. Um exemplo disso é o **Instagram**, que, ao receber uma nova postagem, atualiza automaticamente a linha do tempo do usuário sem recarregar a página.

✓ Execução de animações suaves

Proporciona a criação de transições visuais suaves e interativas, como movimento de elementos, alterações de cor ou de tamanho. É possível observar isso em *sites* de compras, como a **Amazon**, que utiliza animações para suavizar a transição entre diferentes páginas ou para destacar produtos ao passar o *mouse* sobre eles.

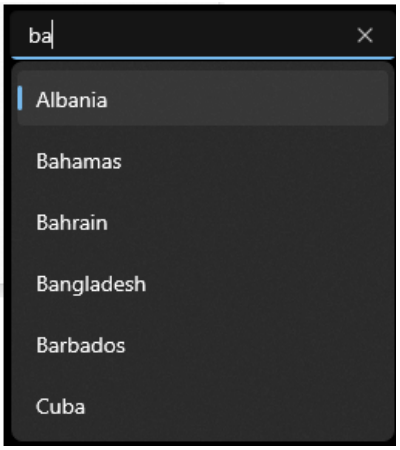
✓ Validação e envio assíncrono de formulários

Permite a validação de dados inseridos em um formulário sem a necessidade de recarregar a página. Além disso, a submissão assíncrona do formulário permite que o envio de dados ocorra em segundo plano, sem interromper a interação do usuário. Por exemplo, um formulário de registro de *login* em um *site*, como a **Netflix**, que valida os campos antes de enviar as informações ao servidor, fornecendo *feedback* instantâneo ao usuário sem recarregar a página.

As respostas dinâmicas estão intrinsecamente ligadas aos eventos em JavaScript, pois é por meio deles que os desenvolvedores podem criar interações e experiências dinâmicas, adaptando o conteúdo e o comportamento da página *web* conforme for necessário.

Além das respostas dinâmicas, também podemos configurar **funcionalidades** nos eventos em JavaScript. Essas funcionalidades são técnicas e **abordagens mais sofisticadas** para lidar com eventos e criar interações mais complexas e dinâmicas em páginas *web*. A capacidade de associar funcionalidades específicas a eventos específicos é crucial para uma experiência de usuário mais envolvente.

Um exemplo de funcionalidade avançada em JavaScript pode ser a implementação de uma busca assíncrona em tempo real enquanto o usuário digita em um campo de pesquisa. Isso é comumente conhecido como *autocomplete* ou **sugestão de pesquisa**. Você já deve ter visto isso em diversos *sites*.



Disponível em: https://www.syncfusion.com/winui-controls/autocomplete. Acesso em: 22 fev. 2024.

6.1.3 Delegação de eventos para eficiência

Quando lidamos com muitos elementos dinâmicos, a delegação de eventos se torna uma técnica eficiente. Delegar eventos significa designar funções específicas para lidar com essas ações. Ou seja, em vez de atribuir manipuladores de eventos a cada elemento de forma individual, nós delegamos o evento a um ancestral comum, melhorando o desempenho e simplificando o código. Mas, vamos entender melhor o que isso significa a seguir.

Imagine que você está organizando uma festa e precisa atribuir tarefas para diferentes pessoas. Em JavaScript, delegar eventos é um pouco como atribuir responsabilidades a diferentes convidados para lidar com certas situações durante a festa. Você pode pensar em um mestre de cerimônias na festa, que delega a responsabilidade de cuidar da música para um DJ, por exemplo, ou a responsabilidade de supervisionar os jogos no dia da festa para um organizador de jogos e assim por diante.

Disponível em: https://www.shutterstock.com/pt/image-vector/set-happy-cartoon-people-having-fun-1630448623.

Acesso em: 23 fev. 2024.

Na programação, nós utilizamos técnicas como a delegação de eventos para **atribuir funções específicas a elementos HTML**. Por exemplo, em vez de colocar um manipulador de

eventos em cada botão, você pode delegar a responsabilidade a um elemento pai que irá lidar com os eventos dos filhos. Considere um exemplo simples de uma lista de tarefas (*to-do list*) em que queremos lidar com **cliques nos itens da lista**, mas, em vez de adicionar um manipulador de evento para cada item, vamos usar a delegação de eventos.

```
// Adiciona um único manipulador de evento ao elemento pai
    document.getElementById('lista-tarefas').addEventListener('click',
function(e) {
      // Verifica se o clique foi em um elemento li
      if (e.target.tagName === 'LI') {
        // Exibe o texto do item clicado no console (poderia ser
qualquer outra ação desejada)
        console.log('Item clicado:', e.target.textContent);
```

Neste exemplo, um único manipulador de evento é adicionado ao elemento pai **<ul>** (lista de tarefas). Quando ocorre um clique em qualquer lugar dentro da lista, o manipulador verifica se o alvo (elemento clicado) é uma **<li>** (item da lista). Se for o caso, ele realiza a ação desejada, que, neste cenário, é exibir o texto do item clicado no console. Essa abordagem é mais eficiente e escalável do que adicionar manipuladores individuais para cada item da lista. Isso **melhora a organização do código**, assim como delegar tarefas na festa ajuda a manter as coisas organizadas!

6.1.4 Evitando o *callback hell*

Callback hell, também conhecido como "*Pyramid of Doom*", refere-se a um padrão de código em JavaScript que ocorre quando **várias chamadas de função assíncronas são aninhadas de uma maneira que resulta em um código difícil de ler e manter**. Geralmente, isso acontece quando existem tem muitos *callbacks* dentro de *callbacks*, formando uma pirâmide visualmente confusa.

Mas...

... o que é um *callback*?

Callback

Função que é passada como argumento para outra função e é executada após a conclusão de uma operação assíncrona ou de um evento específico. Essa técnica **permite a execução de um código em resposta a ações ou eventos**, tornando possível o controle de fluxo assíncrono e a implementação de funcionalidades, como o tratamento de eventos.

Agora, veja um **exemplo de *callback hell*** e como essa estrutura ficaria confusa e visualmente caótica em um código:

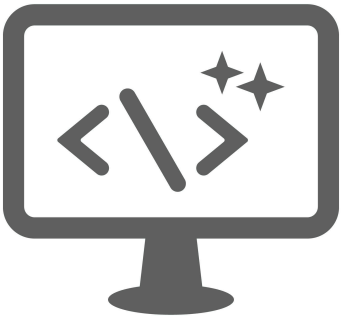
```
funcaoAssincrona1(function(resultado1) {
    funcaoAssincrona2(function(resultado2) {
        funcaoAssincrona3(function(resultado3) {
            // ... e assim por diante
        });
    });
});
```

Ao lidar com várias respostas de eventos, é essencial evitar que o *callback hell* aconteça, ou seja, é necessário manter um código assíncrono limpo e legível. Neste caso, a utilização de **Promises** ou **async/await** pode ser benéfica. Entenda melhor como fazer isso no quadro a seguir:

Essas duas abordagens tornam o código mais linear, fácil de entender e de ser mantido, melhorando a legibilidade e a organização. Além disso, elas facilitam o tratamento de erros em operações assíncronas.

6.1.5 Acessibilidade e boas práticas ao lidar com eventos

Outro aspecto que também ajuda a manter o código legível e organizado é seguir as boas práticas do desenvolvimento *web*, realizando a nomenclatura descritiva de funções e prezando sempre pela organização durante o seu trabalho. Tudo isso em conjunto aprimora a manutenção e a compreensão do código.



Disponível em: https://encurtador.com.br/dntY6. Acesso em: 23 fev. 2024.

Lembre-se que, ao desenvolver para a *web*, a **acessibilidade** também deve ser priorizada. Garanta que as respostas aos eventos sejam amigáveis aos leitores de tela e dispositivos assistivos. Além disso, monitore o desempenho para evitar manipulações excessivas do DOM, pois isso pode impactar a experiência do usuário.

Ao combinar acessibilidade e boas práticas de codificação, é possível criar aplicações *web* mais acessíveis, usáveis e sustentáveis em JavaScript. Isso não apenas beneficia usuários com necessidades diversas, mas também contribui para a manutenção e escalabilidade do código ao longo do tempo.

Disponível em: http://tinyurl.com/4r8v393e. Acesso em: 29 fev. 2024.

6.2 Criando interatividade em uma página *web*

A construção de interatividade em uma página *web* é uma jornada fascinante que envolve habilidades em HTML, CSS e JavaScript. Essas linguagens permitem criar experiências envolventes para os usuários, tornando a navegação mais dinâmica e interessante. Agora, nós vamos mergulhar em um guia que possui o passo a passo para criar a sua própria página *web*,

aplicando todos os conhecimentos que já adquirimos até aqui. Prepare-se para explorar o mundo da programação *web*!

Mão na massa!

Seguindo o passo a passo a seguir, **crie uma página *web*** que não apenas exiba conteúdo, mas também responda dinamicamente às ações do usuário, proporcionando uma experiência interativa e moderna para o usuário.

Passo 1 – Estruturação HTML para interatividade

Comece pela estrutura HTML, na qual a semântica é crucial. Utilize *tags* apropriadas para diferentes seções da página. Incorporar elementos interativos, como botões, é essencial. Mantenha uma estrutura limpa e bem organizada, como você pode ver na imagem a seguir:

```
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.
    <title>Página Interativa</title>
    <link rel="stylesheet" href="styles.css">
</head>
<body>
    <header>
        <h1>Minha Página Interativa</h1>
    </header>
    <main>
        <button id="meuBotao">Clique-me</button>
    </main>
    <script src="script.js"></script>
</body>
</html>
```

E aí, conseguiu acompanhar a leitura desse código? Qual seria o resultado dele em uma página? Faça o teste!

Passo 2 – Estilizando com CSS para aprimorar a interatividade

Agora, utilize o CSS para estilizar os elementos interativos, tornando-os visualmente atraentes. Adicione transições para criar efeitos suaves durante as interações. Veja um exemplo disso no código a seguir:

```css
body {
    font-family: 'Arial', sans-serif;
}

button {
    padding: 10px;
    background-color: #3498db;
    color: #fff;
    border: none;
    cursor: pointer;
    transition: background-color 0.3s ease;
}

button:hover {
    background-color: #2980b9;
}
```

Este trecho de código CSS estiliza os elementos **body** e **button** em uma página da *web*, tornando os botões mais atraentes visualmente e interativos para os usuários que visitam a página. Experimente colocar isso em prática no seu código!

Passo 3 – Adicionando interatividade com JavaScript

Introduza o JavaScript para tornar os elementos interativos do seu código verdadeiramente funcionais. Adicione manipuladores de eventos para criar respostas dinâmicas, como alertas simples em um primeiro momento. "Se liga" no exemplo a seguir:

```javascript
// script.js
const meuBotao = document.getElementById('meuBotao');

meuBotao.addEventListener('click', function() {
    alert('Você clicou no botão!');
});
```

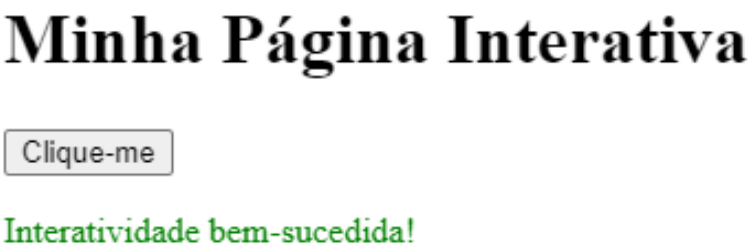
Faça o teste aí no seu código e compartilhe com seus colegas o que você conseguiu desenvolver nessa etapa!

Passo 4 – Respostas dinâmicas e animações avançadas

Nesta etapa, você irá expandir as respostas dinâmicas do seu código usando o JavaScript. Além de simples alertas, atualize o conteúdo da página de forma dinâmica, execute animações ou, até mesmo, adicione e remova elementos do DOM para uma experiência mais envolvente. Observe isso sendo colocado em prática no código a seguir:

```
// script.js
const meuBotao = document.getElementById('meuBotao');
const meuParagrafo = document.createElement('p');
meuParagrafo.textContent = 'Clique no botão para ver a magia acontecer!

meuBotao.addEventListener('click', function() {
    meuParagrafo.style.color = 'green';
    meuParagrafo.textContent = 'Interatividade bem-sucedida!';
    document.body.appendChild(meuParagrafo);
});
```

Este código cria um botão e um parágrafo em uma página da *web*. Quando o botão é clicado, o texto do parágrafo muda para "Interatividade bem-sucedida!" e a cor do texto é alterada para verde. Veja o resultado:



Level up!

Você pode copiar e usar esse código em seu próprio projeto! Experimente fazer mais modificações na sua página, editando o código da forma que você quiser. O céu é o limite!

Passo 5 – Validação de formulários e interatividade avançada

Em páginas com formulários, vá além de simples resposta e cliques. Implemente no seu projetoa validação de formulários e forneça *feedback* imediato ao usuário, elevando o nível de interatividade da página.

```
// script.js
const meuFormulario = document.getElementById('meuFormulario');

meuFormulario.addEventListener('submit', function(event) {
    event.preventDefault(); // Evita o envio tradicional do formulário
    // Lógica de validação e envio assíncrono dos dados
});
```

Por exemplo, o código que você acabou de ver está configurando um ouvinte de eventos (**addEventListener**) para o evento de envio do formulário. Quando o formulário for enviado pelo usuário, a função *callback* (**function(event) {event.preventDefault();**) será chamada, impedindo o comportamento padrão do formulário e permitindo que o desenvolvedor manipule a validação e o envio dos dados de maneira personalizada.

Passo 6 – Teste e otimização contínua

Para finalizar, **teste a sua página em vários navegadores** para garantir que a interatividade esteja consistente. Além disso, **otimize continuamente o código** para que ele sempre tenha um desempenho ideal, **evitando manipulações excessivas do DOM**.

Parabéns!

É fantástico ver nossa própria página *web* em ação, não é mesmo!? Parabéns por aplicar todos os conhecimentos que estudamos até agora. Se precisar de mais dicas, *feedbacks* ou quiser compartilhar suas experiências, fale com seu(a) professor(a), troque ideias com seus colegas e continue aprofundando seus estudos sobre o tema.

DESAFIO PRÁTICO

Criando interatividade em uma página *web*.

Descrição

Você foi designado para aprimorar a experiência do usuário em uma página web, tornando-a mais interativa e dinâmica. O desafio é implementar funcionalidades avançadas que proporcionem uma experiência envolvente aos usuários.

A página web em questão é um portal de notícias que deseja se destacar pela interatividade. Os usuários devem ter a capacidade não apenas de consumir informações, mas também de interagir ativamente com o conteúdo.

Objetivos

- Desenvolver um sistema de comentários que permita aos usuários responderem uns aos outros, votarem em comentários e destacarem os mais relevantes;
- Implementar uma navegação sem recarregamento de página, possibilitando a troca de seções do site sem perder o contexto;
- Criar um sistema de notificações *push* para alertar os usuários sobre novas notícias, interações em seus comentários e atualizações relevantes;
- Desenvolver um mecanismo que aprenda as preferências do usuário ao longo do tempo e forneça um conteúdo personalizado com base em seu histórico de navegação;
- Integrar visualizações de dados interativas, como gráficos dinâmicos e mapas interativos, para tornar a exploração de dados mais envolvente.

Orientações

- Implemente um sistema robusto de comentários com funcionalidades avançadas, como votação, respostas e destaque para comentários relevantes;
- Utilize tecnologias como AJAX ou WebSockets para criar uma navegação suave entre diferentes seções da página sem recarregar a página inteira;
- Integre um serviço de notificações *push*, considerando a permissão do usuário para receber alertas relevantes;
- Desenvolva um algoritmo que analisa o comportamento do usuário e personaliza a experiência de navegação com base em suas preferências;
- Incorpore bibliotecas ou ferramentas que permitam a criação de visualizações de dados interativas para enriquecer o conteúdo do portal de notícias.

RESUMO

Estudamos como a manipulação de eventos em JavaScript é essencial para criar páginas *web* interativas e dinâmicas. Nesse contexto, conhecemos o conceito de evento, que são ações do usuário ao interagir com a página, como cliques, teclas pressionadas ou movimentos do *mouse*, e que desencadeiam respostas específicas no navegador. Ao compreender os tipos de eventos e como manipulá-los, os desenvolvedores podem criar experiências mais envolventes, implementando desde animações e atualizações de conteúdo até a validação de formulários.

Dessa forma, a capacidade de atribuir manipuladores de eventos, delegar eventos eficientemente e seguir boas práticas durante o desenvolvimento do código garantem uma resposta ágil e acessível às ações do usuário, enriquecendo a interatividade em uma página *web* de forma significativa.

ATIVIDADE DE FIXAÇÃO

1. O que são eventos em JavaScript e como eles são acionados em uma página *web*?

2. Explique a diferença entre eventos de captura (*capturing*) e borbulhamento (*bubbling*) em JavaScript. Qual é a importância desses dois conceitos na manipulação de eventos?

3. Como podemos atribuir um manipulador de evento a um elemento HTML usando JavaScript? Forneça um exemplo prático.

4. Quais são alguns tipos comuns de eventos do usuário e como podemos responder a eles usando JavaScript? Dê exemplos específicos.

5. O que é a delegação de eventos e por que isso é considerado uma prática eficiente na manipulação de eventos em páginas *web*?

6. Explique como evitar o problema de *callback hell* ao lidar com múltiplos eventos em JavaScript.

7. Quais são algumas boas práticas ao trabalhar com eventos em termos de acessibilidade para diferentes dispositivos?

8. Como podemos interromper a propagação de eventos em JavaScript? Em que situações isso pode ser útil?

9. Descreva como criar uma animação simples em resposta a um evento do usuário usando JavaScript.

10. Qual é a importância da manipulação adequada de eventos no contexto de criar experiências *web* interativas e envolventes?

CAPÍTULO 07

EXPLORANDO O JAVASCRIPT: FUNÇÕES DE ORDEM SUPERIOR E ESCOPO DE VARIÁVEIS

O que esperar deste capítulo:

- Compreender as funções de ordem superior, como **map**, **filter** e **reduce**;
- Explorar, de forma detalhada, o escopo de variáveis em JavaScript, incluindo o escopo global e o escopo de função;
- Conhecer o conceito de *closure* e *hoisting* e as suas aplicações práticas.

7.1 Funções de ordem superior

Em JavaScript, as funções de ordem superior são **aquelas que podem receber outras funções como argumentos ou retornar funções como resultados**. Essa capacidade de manipular funções como valores é uma característica fundamental das linguagens de programação. Elas são conceitos fundamentais no arsenal de conhecimento de um programador JavaScript, proporcionando maneiras poderosas de se manipular *arrays*.

Existem três principais funções de ordem superior em JavaScript, que são: **map**, **filter** e **reduce**. Neste capítulo, vamos explorá-las para entender não apenas como utilizá-las, mas também o motivo que tornam elas tão valiosas.

7.1.1 Função map

A função **map** é uma ferramenta versátil para **transformar cada elemento de um *array***. A magia por trás dela reside na capacidade de aplicar uma função a cada item, gerando um novo *array*. Considere o exemplo a seguir:

```
const numeros = [1, 2, 3, 4, 5];

const numerosDobrados = numeros.map(function(numero) {
    return numero * 2;
});

console.log(numerosDobrados); // Saída: [2, 4, 6, 8, 10]
```

Aqui, a função passada para **map** dobra cada número do *array* original (**return numero * 2**), criando um novo *array* de números dobrados. Viu como é fácil de entender como essa função funciona? Agora, acompanhe no quadro abaixo o motivo dessa função ser útil e entenda quando ela deve ser utilizada.

Transformação de dados

A principal finalidade da função **map** é transformar os elementos de um *array*, aplicando uma função a cada elemento e gerando um novo *array* com os resultados. Isso é útil quando você precisa realizar operações em cada item do *array* original.

Legibilidade

O uso da função **map** muitas vezes resulta em um código mais conciso e legível, pois a lógica de transformação é encapsulada na própria função. Isso torna o código mais declarativo, focando no que deve ser feito em vez de como deve ser feito.

Imutabilidade

A função **map não modifica o** array **original**. Ela retorna um novo *array* com os resultados da aplicação da função a cada elemento. Isso é benéfico, pois evita efeitos colaterais e torna o código mais previsível.

Manutenção simples

O uso da função **map** facilita a manutenção do código, pois a transformação aplicada aos elementos é expressa de forma clara. Isso é especialmente útil ao lidar com operações complexas ou quando é necessário ajustar a lógica de transformação no futuro.

Vamos continuar e conhecer melhor a próxima função de ordem superior: a função **filter**.

7.1.2 Função filter

Esta função permite **filtrar elementos de um *array*** com base em uma condição fornecida. Vamos exemplificar filtrando apenas os números pares. Observe o código a seguir:

```
const numerosPares = numeros.filter(function(numero) {
    return numero % 2 === 0;
});

console.log(numerosPares); // Saída: [2, 4]
```

E aí, conseguiu entender como essa função funciona? Perceba que a função dentro do **filter** retorna **true** para números pares, resultando em um novo *array* contendo apenas esses números. Essa função é útil em diversas situações e oferece benefícios significativos, tais como:

Facilidade de filtragem

A função **filter** simplifica o processo de filtragem de elementos com base em critérios específicos. Isso torna o código mais legível e fácil de entender, pois a lógica de filtragem é encapsulada na própria função.

Código mais conciso

Muitas vezes, utilizar **filter** resulta em um código mais conciso em comparação com o uso de *loops* tradicionais. O código é mais declarativo, focando no que deve ser feito em vez de como deve ser feito.

Maior legibilidade e manutenção

A função **filter** torna o código mais legível ao expressar a intenção do filtro de forma clara e direta. Isso facilita a manutenção do código, pois o propósito da operação de filtragem é evidente.

Imutabilidade

A função **filter não modifica o *array* original**. Em vez disso, ela retorna um novo *array* contendo apenas os elementos que atendem à condição. Isso contribui para práticas de programação funcional e ajuda a evitar efeitos colaterais indesejados.

7.1.3 Função reduce

Frequentemente, a função **reduce** é utilizada para processar e acumular valores de um *array* para obter um resultado único. Por exemplo, é possível usá-la para calcular a soma de todos os elementos, assim como no código a seguir:

```
const somaTotal = numeros.reduce(function(acumulador, numero) {
    return acumulador + numero;
}, 0);

console.log(somaTotal); // Saída: 15
```

Perceba que o **reduce** agrega os números somando-os ao acumulador, começando com um valor inicial de 0, ou seja, essa função **reduziu** o *array* a um único valor.

Veja no quadro a seguir algumas situações em que você pode utilizar essa função:

Agregar dados

A função **reduce** é útil quando você precisa agregar ou combinar os elementos de um *array* em um único valor. Isso pode incluir a soma de todos os elementos, a concatenação de *strings*, a contagem de ocorrências, entre outras operações.

Transformação complexa de dados

Quando a transformação dos elementos envolve uma lógica mais complexa e que não pode ser facilmente realizada com **map** ou **filter**, a função **reduce** permite maior uma flexibilidade. Isso ocorre porque você tem acesso ao acumulador, que pode armazenar um estado intermediário durante o processo de redução.

Construir estruturas de dados

A função **reduce** pode ser usada para construir estruturas de dados mais complexas, como objetos ou *arrays*, a partir dos elementos de um *array* original.

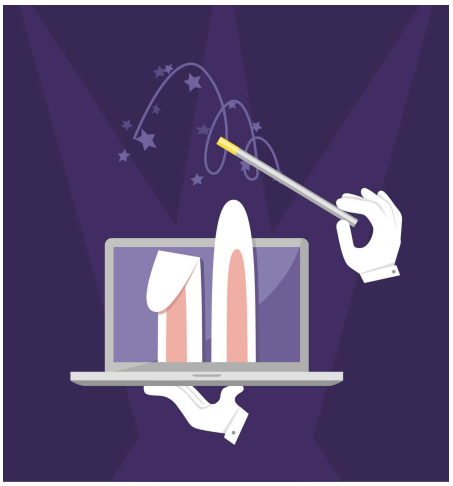
Lidar com muitos dados

A função **reduce** pode ser mais eficiente do que **map** ou **filter** em termos de desempenho ao lidar com grandes volumes de dados, pois **reduz o *array* a um único valor**, evitando a criação de novos *arrays* intermediários.

É importante notar que o uso da função **reduce** pode exigir uma compreensão mais profunda do problema em comparação com as funções **map** e **filter**, pois você está gerenciando um estado acumulado ao longo do processo de redução. No entanto, a sua versatilidade e capacidade de lidar com uma variedade de casos tornam ela uma ferramenta poderosa em JavaScript.

7.1.4 Combinando as funções de ordem superior para realizar tarefas complexas

Ao trabalharmos com as funções de ordem superior, a verdadeira magia acontece quando combinamos todas elas para realizar tarefas complexas em uma única linha de código.



Disponível em: https://www.shutterstock.com/pt/image-vector/magic-wand-web-tricks-263619371. Acesso em: 23 fev. 2024.

A combinação de funções de ordem superior em JavaScript refere-se à prática de utilizar múltiplas funções de ordem superior em conjunto para realizar tarefas mais complexas e sofisticadas. Isso é possível porque essas funções são poderosas e podem ser combinadas de formas criativas para manipular e transformar dados de maneira eficiente. A combinação delas permite que os desenvolvedores escrevam códigos mais expressivos, concisos e modulares.

Para que você entenda melhor como isso pode ser feito, vamos analisar um código escrito para calcular a soma dos quadrados dos números pares:

```
const resultado = numeros
    .filter(numero => numero % 2 === 0) // Filtra números pares
    .map(numero => numero * numero) // Calcula o quadrado de cada número
    .reduce((acumulador, numero) => acumulador + numero, 0); // Soma os 

console.log(resultado); // Saída: 20
```

Neste exemplo, podemos observar como o código realiza as seguintes operações em um *array*:

1. filtragem: a função **filter** é usada para criar um novo *array* com todos os elementos que passam no teste implementado pela função fornecida. Nesse caso em específico, a função é (**numero => numero % 2 === 0**), que verifica se um número é par;

2. mapeamento: a função **map** cria um novo *array* com os resultados da chamada de uma função para cada elemento do *array* original. Aqui, a função é (**numero => numero * numero**), que calcula o quadrado de um número;

3. redução: a função **reduce** aplica uma função contra um acumulador e cada elemento do *array* (da esquerda para a direita), reduzindo-o a um único valor. Neste caso, a função é **(acumulador, numero) => acumulador + numero**, que soma os números.

Com esse exemplo, fica evidente como as funções de ordem superior podem ser combinadas para realizar tarefas complexas de maneira modular e expressiva. Ao compor essas funções, você pode criar *pipelines* de transformação de dados mais sofisticados e aproveitar as funcionalidades existentes de forma eficaz em seu código!

7.2 Escopo de variáveis em JavaScript

Em JavaScript, o escopo de variáveis refere-se à **região do código em que uma determinada variável é visível, acessível e modificável**. Ele determina as regras de alcance das variáveis, ou seja, onde elas podem ser utilizadas ou modificadas. Existem dois tipos predominantes de escopo em JavaScript: o escopo **global** e o escopo **local**. No entanto, além desses, há outros tipos a serem explorados. Neste capítulo, nós vamos aprofundar nosso entendimento sobre este tema, os seus diversos tipos, as suas regras e as suas implicações. Está preparado?

7.2.1 Escopo global e local: entendendo as fronteiras

Assim como vimos anteriormente, em JavaScript, as variáveis podem existir em dois principais escopos: o global e o local. As variáveis declaradas fora de qualquer função ou

bloco de código têm escopo global, enquanto aquelas declaradas dentro de funções têm escopo local. Vamos visualizar isso no código a seguir:

```
// Exemplo de escopo global
let variavelGlobal = "Eu sou global";

function exemploEscopo() {
    // Exemplo de escopo local
    let variavelLocal = "Eu sou local";

    console.log(variavelGlobal); // Acesso a variável global
    console.log(variavelLocal); // Acesso a variável local
}

exemploEscopo();

console.log(variavelGlobal); // Acesso a variável global fora da função
// console.log(variavelLocal); // Erro: variavelLocal is not defined
```

No exemplo acima, **variavelGlobal** é acessível em todo o código, enquanto **variavelLocal** é acessível apenas dentro da função **exemploEscopo**.

7.2.2 Escopo de bloco e let: introduzindo *block-scoped variables*

Antes do ECMAScript 6 (ES6), uma versão do JavaScript lançada no mercado em 2015, essa linguagem não possuía escopo de bloco. No ES6, foi feita a introdução da palavra-chave **let** e, com isso, se tornou possível criar variáveis com escopo de bloco.

A palavra-chave **let** é usada para declarar variáveis em JavaScript. A sua principal característica é que as variáveis por ele declaradas têm um escopo de bloco. Isso significa que elas são visíveis apenas dentro do bloco de código em que são definidas. Vamos ver como isso funciona? Então, "se liga" no exemplo a seguir:

```
if (true) {
    let variavelBloco = "Eu sou de um bloco";
    console.log(variavelBloco); // Acesso a variável de bloco
}

// console.log(variavelBloco); // Erro: variavelBloco is not defined
```

Aqui, é possível perceber que a **variavelBloco** só é acessível dentro do bloco **if**. Caso alguém tentasse **acessá-la fora do bloco, isso resultaria em um erro**. O uso de **let** e o escopo de bloco contribuem para um código mais seguro, evitando conflitos de variáveis e permitindo uma melhor organização e encapsulamento de variáveis em blocos específicos.

7.2.3 Closure: o poder do acesso lexical

Closure é um fenômeno em JavaScript no qual **uma função interna tem acesso às variáveis de sua função externa**, mesmo após a função externa ter retornado. É como se uma função lembrasse do escopo em que foi criada, mesmo quando ela é executada fora dele. Isso só é possível devido ao **acesso lexical**.

O *closure* é útil para criar funções que têm acesso a variáveis fora de seu próprio escopo e podem "lembrar" do estado em que foram criadas. Vejamos um exemplo:

```
function criarFuncao() {
    let mensagem = "Olá, ";

    function saudacao(nome) {
        console.log(mensagem + nome);
    }

    return saudacao;
}

const minhaFuncao = criarFuncao();
minhaFuncao("João"); // Saída: Olá, João
```

Neste código, **saudacao** tem acesso à variável **mensagem** mesmo após **criarFuncao** ter sido executada. Isso, nada mais é, que um *closure* em ação.

Closures são uma parte importante da flexibilidade e do poder da linguagem JavaScript. Dessa forma, o seu entendimento é crucial para desenvolvedores que desejam tirar o máximo proveito das capacidades da linguagem. Para que você fique mais capacitado ainda neste assunto, vamos conhecer alguns pontos importantes sobre *closures*.

1. Um *closure* "captura" as variáveis do escopo em que foi criado, permitindo que a função interna as utilize posteriormente.

2. *Closures* podem ser usados para criar encapsulamento em JavaScript, ocultando variáveis do escopo global e permitindo um controle mais preciso sobre o acesso a essas variáveis.

3. *Closures* permitem que funções "lembrem-se" do estado em que foram criadas. Isso é útil em situações onde é necessário manter um estado específico entre chamadas de função.

4. *Closures* são frequentemente encontrados quando uma função interna é retornada por uma função externa.

7.2.4 *Hoisting*: entendendo a elevação de variáveis

Hoisting é o comportamento em JavaScript em que as **declarações de variáveis são movidas para o topo de seus escopos durante a fase de compilação**. Isso significa que podemos usar uma variável antes de ela ser declarada. No entanto, é importante destacar que **apenas a declaração é elevada, não a sua inicialização**. Em outras palavras, mesmo que você escreva seu código de cima para baixo, o JavaScript, nos bastidores, faz algumas "mágicas" e move algumas partes do código para o topo antes de realmente executá-lo.

O código a seguir ilustra bem um exemplo de *hoisting*. Vamos analisá-lo:

```
console.log(variavelElevada); // Saída: undefined
var variavelElevada = "Eu fui elevada";
console.log(variavelElevada); // Saída: Eu fui elevada
```

- Antes de executar o código, a declaração da variável **variavelElevada** é movida para o topo do escopo. No entanto, como a inicialização ainda não ocorreu, o valor inicial é **undefined**. Portanto, o primeiro **console.log** imprime **undefined**.
- Agora, a variável **variavelElevada** é declarada e inicializada com o valor "Eu fui elevada".
- O segundo **console.log** imprime o valor atual da variável **variavelElevada**, que foi inicializada anteriormente. Neste ponto, a variável já foi elevada (movida para o topo) e possui o valor "Eu fui elevada".

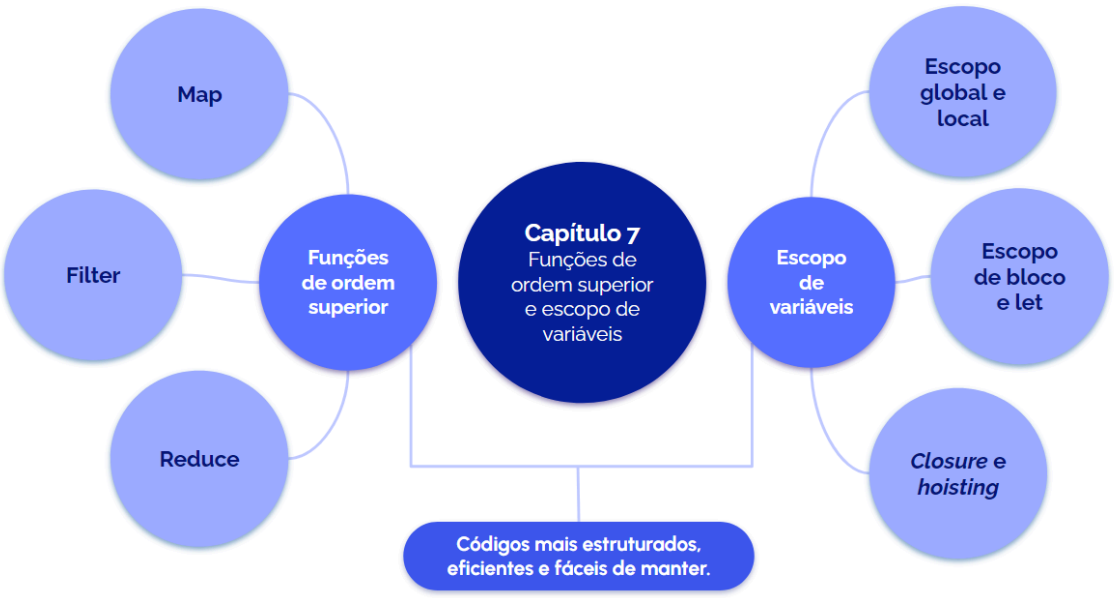
Podemos dizer que, ao utilizar o *hoisting*, é como se a linguagem JavaScript organizasse algumas coisas antes de começar a executar o código, permitindo que você use variáveis mesmo antes de declará-las, mas com um **valor inicial de *undefined***. Saber aplicar este recurso é uma habilidade que você deve ter, pois essa é uma característica peculiar dessa linguagem e requer cuidados na hora de aplicá-la, pois pode levar a comportamentos inesperados caso não seja usada corretamente.

RESUMO

Abordamos as funções de ordem superior e o escopo de variáveis, conceitos que são cruciais para desenvolvedores que buscam eficiência e elegância em seus códigos. Também vimos que as funções de ordem superior, como **map**, **filter** e **reduce**, oferecem uma abordagem funcional para manipular *arrays*, permitindo transformações e filtragens de maneira concisa e expressiva. Aprendemos que, em conjunto com o entendimento sobre o escopo de variáveis e sobre os conceitos de escopo global, local e *closure*, os desenvolvedores conseguem ter total controle sobre a visibilidade e acessibilidade das variáveis, o que contribui para a criação de códigos mais estruturados e eficientes.

Por fim, trabalhamos o conceito de *hoisting*, tema essencial para desenvolvedores JavaScript, pois influencia diretamente o comportamento das variáveis e funções no momento da execução do código. Assim como estudamos, o *hoisting* move as declarações para o topo do escopo durante a fase de compilação, permitindo o uso de variáveis e funções antes mesmo de serem declaradas no código. Ao entender o funcionamento do *hoisting*, os desenvolvedores podem evitar surpresas indesejadas e escrever código mais previsível.

Com todos esses conhecimentos em mãos, você poderá criar aplicações mais poderosas e compreender como a linguagem JavaScript lida com as funções de ordem superior e o escopo de variáveis, recursos essenciais para construir sistemas robustos e expressivos.



ATIVIDADE DE FIXAÇÃO

1. Explique o que são funções de ordem superior em JavaScript. Dê exemplos práticos de como **map**, **filter** e **reduce** podem ser utilizadas como funções de ordens superior (HOF), destacando suas aplicações com *arrays*.

2. Discorra sobre as diferenças entre escopo global e escopo local em JavaScript. Em seguida, explique como o uso de **let** impacta o escopo de variáveis em comparação com **var**.

3. Explique o conceito de *hoisting* no contexto de variáveis em JavaScript e fale o motivo de ser importante estar ciente do *hoisting* ao escrever um código JavaScript.

4. O que é escopo de bloco e como ele difere do escopo de função em JavaScript? Apresente um exemplo prático de código que demonstra o escopo de bloco.

5. Descreva como ocorre um *closure* em JavaScript e, em seguida, explique o motivo dos closures serem úteis e em quais situações eles são comumente aplicados. Dê um exemplo de código que envolva um closure.

6. Como as funções de ordem superior facilitam a manipulação de *arrays* em comparação com abordagens tradicionais? Explique a diferença entre **map** e **filter** no contexto de manipulação de *arrays*.

7. Como você combinaria o uso de funções de ordem superior e escopo de variáveis para resolver um problema complexo? Dê um exemplo prático que envolva tanto o uso de funções de ordem superior (HOF) quanto o gerenciamento adequado do escopo de variáveis.

8. Quais são os problemas em potencial associados ao uso excessivo de variáveis globais e como o escopo de variáveis globais pode afetar a modularidade e a manutenção do código?

9. Liste outras funções de ordem superior além de **map**, **filter** e **reduce**, em JavaScript e os propósitos de cada uma delas.

10. Quais são algumas boas práticas ao usar funções de ordem superior?

CAPÍTULO 08

EXPLORANDO O UNIVERSO ORIENTADO A OBJETOS EM JAVASCRIPT: FUNDAMENTOS, MÉTODOS PERSONALIZADOS E FUNÇÕES AVANÇADAS

O que esperar deste capítulo:

- Desenvolver a habilidade de aplicar métodos e acessar propriedades de objetos predefinidos;
- Aplicar os conhecimentos adquiridos na criação e manipulação de objetos em projetos práticos;
- Utilizar funções avançadas em conjunto com objetos em JavaScript.

8.1 Introdução a objetos em JavaScript

JavaScript, uma linguagem amplamente utilizada para desenvolvimento *web*, tem sua estrutura centrada em objetos, que são estruturas de dados fundamentais que permitem **armazenar e organizar informações** de diversas formas.

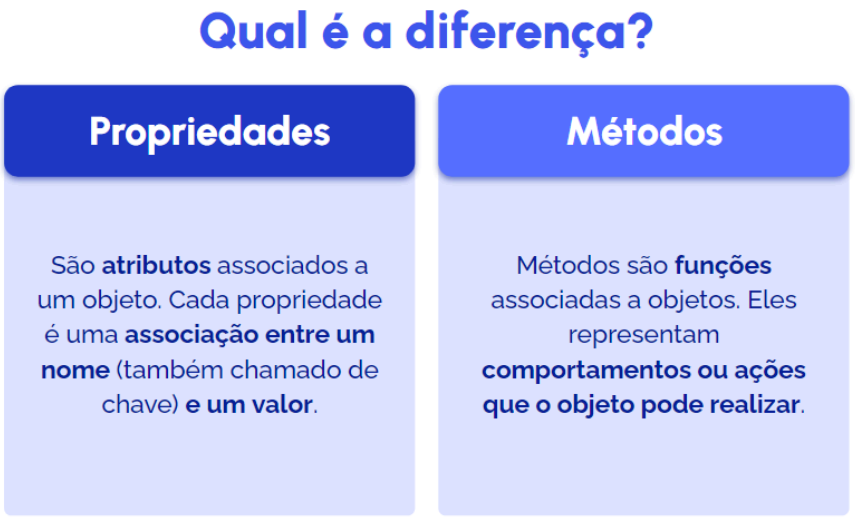
Um objeto é uma **coleção de propriedades**, em que cada propriedade é um par chave-valor e que podem conter qualquer tipo de dado, incluindo números, *strings*, funções e, até mesmo, outros objetos. Para criar um objeto, utilizamos a notação de chaves **{}**. Vamos ver como isso funciona no código a seguir:

```
let carro = {
  modelo: "Civic",
  ano: 2020,
  ligar: function() {
    console.log("O carro está ligado.");
  },
  desligar: function() {
    console.log("O carro está desligado.");
  }
};
```

E aí, conseguiu identificar qual é o objeto? Neste exemplo, o objeto é **carro**. Ele é definido como um objeto literal e possui várias propriedades, como modelo e ano, além de métodos (funções associadas a ele), como ligar e desligar. Viu como é fácil? Mas, vamos conhecer mais detalhadamente o que são as propriedades e os métodos relacionados aos objetos.

8.1.1 Propriedades e métodos associados aos objetos JavaScript

Em JavaScript, os objetos têm propriedades e métodos que definem as suas características e os seus comportamentos. Vamos entender esses dois conceitos no esquema a seguir.



Adicionando e removendo propriedades e métodos

Podemos adicionar novas propriedades a um objeto a qualquer momento. Veja como isso é feito no código à seguir:

Métodos

Ainda analisando o exemplo do carro, os métodos são **ligar** e **desligar**, funções que podem ser chamadas no contexto do objeto carro para **realizar ações específicas**.

```
let carro = {
  modelo: "Civic",
  ano: 2020,
  ligar: function() {
    console.log("O carro está ligado.");
  },
  desligar: function() {
    console.log("O carro está desligado.");
  }
};
```

Essas adições mostram como você pode estender dinamicamente um objeto em JavaScript, adicionando propriedades e métodos conforme necessário. Legal, não é?

Agora, caso você queira remover as propriedades e os métodos, é só utilizar o operador **delete**, como na imagem do código a seguir:

```
// Adicionando nova propriedade
carro.pneuNovo = true;

// Removendo propriedade
delete carro.ano;
```

Acessando propriedades e métodos

O acesso às propriedades e aos métodos refere-se à forma como interagimos com os objetos em JavaScript, seja para recuperar valores de suas propriedades ou para chamar as suas funções (métodos).

Para acessar as propriedades de um objeto, recorremos à notação de ponto, como em: **objeto.propriedade**. Da mesma maneira, para invocar um método, empregamos a notação de ponto seguida por parênteses vazios, como em: **objeto.metodo()**.Observe:

```
// Acessando propriedades
console.log(carro.modelo); // Saída: Civic

// Invocando método
carro.ligar(); // Saída: O carro está ligado!
```

O acesso às propriedades e aos métodos de objetos em JavaScript é fundamental para manipular e interagir com os dados e comportamentos associados a esses objetos. Sendo assim, para que isso seja feito, utilizamos a notação de ponto, proporcionando uma maneira simples e clara de acessar e fazer a manipulação dos dados e dos comportamentos associados aos objetos.

Construtores e objetos personalizados

Construtores são funções especiais utilizadas para **criar e inicializar objetos com propriedades e métodos específicos**. Eles são chamados usando a palavra-chave **new** e são frequentemente usados quando precisamos criar vários objetos semelhantes, seguindo um mesmo padrão.

Já os **objetos personalizados** são instâncias criadas a partir de construtores, proporcionando uma maneira de criar objetos com propriedades e métodos específicos. Veja como isso se aplica em um código:

```
// Construtor de objetos Carro
function Carro(modelo, ano, cor) {
    this.modelo = modelo;
    this.ano = ano;
    this.cor = cor;
    this.ligar = function() {
        console.log("O carro está ligado!");
    };
}

// Criando um novo objeto Carro
let meuCarro = new Carro("Fusca", 1980, "Azul");
```

Esse código JavaScript define um construtor chamado **Carro** e cria uma instância desse objeto utilizando a palavra-chave **new**. Vamos explicar o que está acontecendo passo a passo para que você entenda melhor:

- **definição do construtor –** o construtor **Carro** é uma função que aceita três parâmetros (modelo, ano, e cor). Dentro dele, são definidas três propriedades (**modelo**, **ano**, e **cor**) e um método (**ligar**) associado ao objeto que será criado;

- **criação de uma instância usando o construtor –** a palavra-chave **new** é usada para criar uma nova instância do objeto **Carro**. A instância é armazenada na variável **meuCarro** e os argumentos passados são "**Fusca**" para modelo, **1980** para ano e "**Azul**" para cor.

Pausa para refletir

De que maneira esses conceitos facilitam a organização, reutilização de código e implementação de comportamentos específicos em objetos personalizados?

Ao incorporar propriedades e métodos em objetos personalizados, os desenvolvedores aproveitam a orientação a objetos para criar códigos mais organizados, reutilizáveis e flexíveis, facilitando a modelagem de entidades complexas e a implementação de comportamentos específicos em uma aplicação JavaScript.

Herança e protótipos

Herança e protótipos estão relacionados à forma como os objetos compartilham propriedades e métodos, o que significa que os **objetos podem herdar propriedades e métodos** de outros objetos, simplificando a criação de estruturas complexas. Vamos analisar como isso deve ser feito em um código JavaScript.

```javascript
// Objeto protótipo
let animal = {
    comer: function() {
        console.log("O animal está comendo.");
    }
};

// Objeto que herda do protótipo
let leao = Object.create(animal);
leao.rugir = function() {
    console.log("O leão está rugindo!");
};

leao.comer(); // Saída: O animal está comendo.
leao.rugir(); // Saída: O leão está rugindo!
```

Esse código envolve o uso de protótipos para estabelecer uma relação de herança entre dois objetos: **animal** e **leao**. Vamos entender melhor o que está acontecendo no passo a passo a seguir:

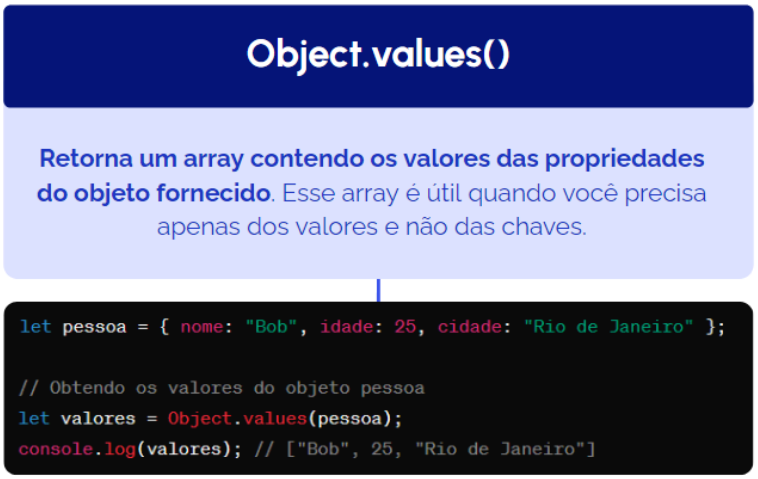
1. criação do objeto "animal":
    - Um objeto chamado **animal** é criado com uma propriedade chamada **comer**;
    - Essa propriedade é uma função anônima que imprime a mensagem "O animal está comendo." no console.
2. criação do objeto "leao" com herança de protótipo:
    - Usando **Object.create(animal)**, é criado um novo objeto chamado **leao**;
    - O objeto **leao** herda as propriedades e métodos do objeto **animal**.
3. adição de um método "rugir" ao objeto "leao":
    - O objeto **leao** recebe uma nova propriedade chamada **rugir**;
    - Essa propriedade é uma função anônima que imprime a mensagem "O leão está rugindo!" no console.
4. chamada dos métodos "comer" e "rugir" no objeto "leao":
    - Chamamos o método **comer** no objeto **leao**, que exibe a mensagem "O animal está comendo.";

- Em seguida, chamamos o método **rugir** no objeto **leao**, que exibe a mensagem "O leão está rugindo!".

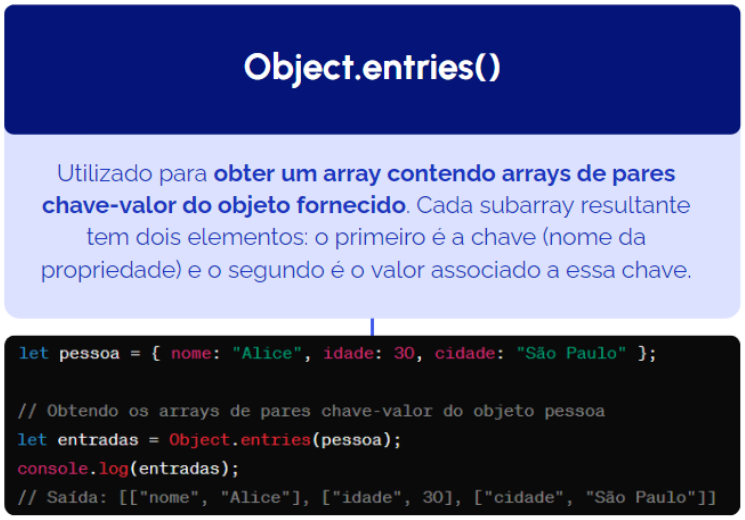
Esse exemplo de código ilustra bem como ocorre o processo de herança prototípica em JavaScript. Através dele, podemos perceber como o objeto **leao** é criado com **animal** como seu **protótipo**, o que permite **herdar** o método **comer**. Depois dessa análise explicando o passo a passo ficou mais fácil entender os conceitos de herança e protótipo, não é?

8.2 Métodos e propriedades de objetos predefinidos e personalizados

JavaScript, como uma linguagem orientada a objetos, oferece uma ampla gama de métodos e propriedades, assim como a que acabamos de estudar. Esses **métodos e propriedades podem ser predefinidos e personalizados** para que os desenvolvedores consigam fazer uma manipulação eficiente de objetos, facilitando tarefas comuns e fornecendo uma variedade de recursos úteis para manipular e interagir com objetos. E aí, vamos conhecer alguns dos métodos e propriedades predefinidos mais comuns em objetos JavaScript?



Esses dois métodos, **Object.keys(objeto)** e **Object.values(objeto)**, são utilizados para **extrair informações específicas de um objeto**. Eles são úteis quando você precisa trabalhar com as chaves (nomes de propriedades) e os valores do objeto, respectivamente.

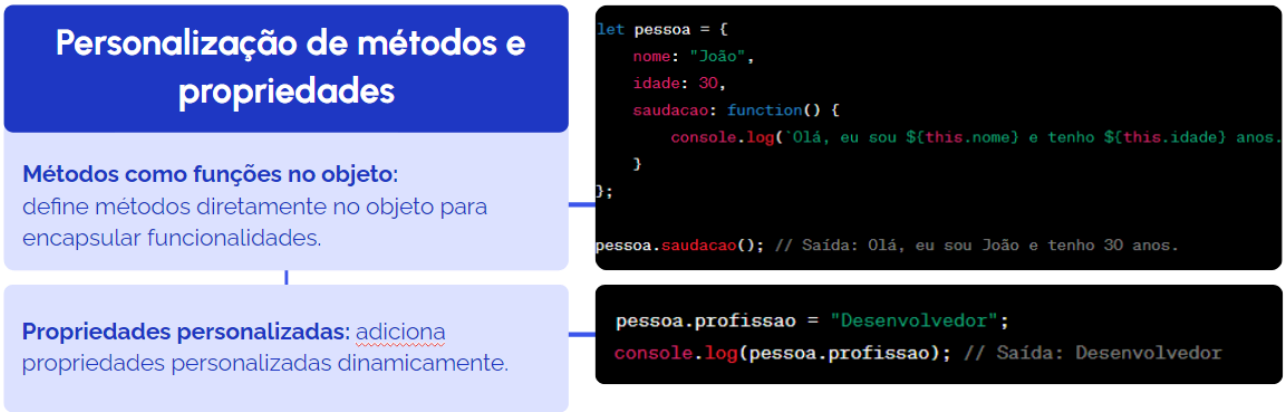


Neste exemplo, **Object.entries(pessoa)** retorna um *array* contendo *subarrays* com pares chave-valor correspondentes às propriedades do objeto pessoa, que são **nome**, **idade** e **cidade**. Este método predefinido é particularmente útil quando você precisa de uma representação mais flexível e manipulável das propriedades de um objeto, pois ele proporciona uma maneira concisa de acessar e trabalhar com pares chave-valor.

8.2.1 Personalizando métodos e propriedades de objetos em JavaScript

A personalização dos métodos e das propriedades de objetos é uma prática comum que permite que os desenvolvedores **criem funcionalidades específicas adaptadas às**

necessidades de suas aplicações. Isso pode ser feito de várias maneiras, a depender do contexto e dos requisitos do projeto. Veja duas formas de se fazer isso:



Personalizar métodos e propriedades de objetos em JavaScript envolve a capacidade de modificar ou estender o comportamento padrão de objetos existentes. Essa personalização pode ser útil para adaptar objetos de acordo com requisitos específicos, criar funcionalidades adicionais ou modificar comportamentos existentes.

8.2.2 Manipulando métodos personalizados dinamicamente

A manipulação dinâmica com métodos personalizados em JavaScript refere-se à capacidade de **adicionar, modificar ou remover métodos de objetos em tempo de execução**. Essa dinamicidade é proporcionada pela natureza flexível e dinâmica da linguagem. Veja como podemos fazer isso no código:

A manipulação dinâmica com métodos personalizados é uma prática valiosa quando você precisa de **flexibilidade e adaptabilidade** no seu código, permitindo a criação de sistemas mais modulares, extensíveis e fáceis de manter. No entanto, deve-se usá-la com cuidado para garantir que as alterações dinâmicas não introduzam comportamentos inesperados ou dificuldades de manutenção.

8.2.3 Construindo objetos personalizados com métodos

Construir objetos personalizados com métodos em JavaScript é uma prática comum que envolve a **definição de funções construtoras** e a **adição de métodos a esses objetos** para fornecer funcionalidades específicas. Essa abordagem permite criar instâncias de objetos que compartilham características comuns, ao mesmo tempo que oferece a flexibilidade de personalizar o comportamento dessas instâncias. Veja o exemplo a seguir:

```
function Livro(titulo, autor, ano) {
    this.titulo = titulo;
    this.autor = autor;
    this.ano = ano;
    this.info = function() {
        console.log(`${this.titulo}, escrito por ${this.autor}, publicado
    };
}

let meuLivro = new Livro("JavaScript Explorado", "John Developer", 2022);
meuLivro.info(); // Saída: JavaScript Explorado, escrito por John Develop
```

Assim como vimos, personalizar métodos e propriedades de objetos é um aspecto fundamental da programação em JavaScript, pois permite a criação de objetos ajustados de forma detalhada às necessidades específicas das aplicações. A escolha do método de personalização vai depender das necessidades específicas do projeto, da preferência pessoal e da experiência do desenvolvedor. A flexibilidade do JavaScript em suportar diferentes estilos de programação torna essa linguagem poderosa para o desenvolvimento de aplicações robustas e personalizadas.

Disponível em: https://encurtador.com.br/fryU5. Acesso em: 03 mar. 2024.

8.3 Trabalhando com funções avançadas e objetos complexos

Além das operações básicas de criação e manipulação de objetos, existem várias funções avançadas que facilitam o trabalho com objetos em JavaScript, permitindo que os desenvolvedores realizem tarefas complexas de forma mais eficiente. Vamos conhecer algumas dessas funções e entender como trabalhar com elas?

8.3.1 Funções como propriedades de objetos

Em JavaScript, funções podem ser atribuídas como propriedades de objetos. Esse recurso é uma parte fundamental da linguagem e permite uma abordagem orientada a objetos. Quando uma função é armazenada como uma propriedade de um objeto, ela é frequentemente chamada de **método do objeto**. Métodos são, essencialmente, funções que têm acesso ao contexto do objeto através de uma palavra-chave, assim como vimos anteriormente, permitindo que eles operem nos dados do objeto e encapsulem lógicas específicas relacionadas ao objeto. Veja um exemplo de como isso é feito em um código:

```
let carro = {
    velocidade: 0,
    acelerar: function() {
        this.velocidade += 10;
    },
    frear: function() {
        this.velocidade -= 5;
    }
};

carro.acelerar();
console.log(carro.velocidade); // Saída: 10

carro.frear();
console.log(carro.velocidade); // Saída: 5
```

8.3.2 Funções como métodos dinâmicos

Em JavaScript, é possível criar funções diretamente no objeto para que ele interaja dinamicamente com outras propriedades. Veja um exemplo:

```
let pessoa = {
    nome: "Ana",
    idade: 25,
    saudacao() {
        console.log(`Olá, eu sou ${this.nome} e tenho ${this.idade} anos
    }
};

pessoa.saudacao(); // Saída: Olá, eu sou Ana e tenho 25 anos.
```

E aí, conseguiu acompanhar a leitura desses códigos?
A habilidade de ler e compreender um código é fundamental para que você desenvolva suas habilidades em programação.

8.3.3 Funções como construtores de objetos

Funções como construtores de objetos em JavaScript referem-se ao uso de funções para criar instâncias de objetos. Assim como já estudamos neste capítulo, quando uma função é utilizada com a palavra-chave **new**, ela se comporta como um construtor, **criando e**

retornando um novo objeto. Dessa forma, esse objeto pode ter as propriedades e os métodos associados à função construtora. Veja um exemplo:

```
function Animal(nome, tipo) {
    this.nome = nome;
    this.tipo = tipo;
}

Animal.prototype.emitirSom = function() {
    console.log("Som genérico de animal.");
};

let gato = new Animal("Whiskers", "Felino");
gato.emitirSom(); // Saída: Som genérico de animal.
```

8.3.4 Trabalhando com objetos complexos

Trabalhar com objetos complexos em JavaScript envolve a manipulação de objetos que contêm propriedades aninhadas, *arrays*, funções e outros objetos. Vamos explorar algumas técnicas e conceitos que podem ser úteis nesse contexto:

Objetos aninhados

Isso acontece quando **um objeto contém propriedades que também são objetos**. Ou seja, há uma estrutura hierárquica, onde um objeto é incorporado como valor de uma propriedade de outro objeto. Veja no exemplo ao lado como isso deve ser escrito em um código JavaScript.

```
let pessoa = {
  nome: "Alice",
  endereco: {
    rua: "Rua Principal",
    cidade: "Cidade A"
  },
  contato: {
    telefone: "123-456-789",
    email: "alice@example.com"
  }
};
```

Neste exemplo:

- **pessoa** é um objeto que contém as propriedades **nome**, **endereco**, e **contato**;
- **endereco** e **contato** são objetos aninhados dentro do objeto **pessoa**;
- cada objeto aninhado possui as suas próprias propriedades (rua e cidade em **endereco** e telefone e *e-mail* em **contato**).

Frequentemente, os objetos aninhados são utilizados para representar dados complexos em JavaScript, como informações de usuários, configurações de aplicativos, ou qualquer situação em que haja uma hierarquia natural nas informações a serem modeladas. Isso ajuda a organizar e estruturar o código de forma mais compreensível e modular.

Métodos dinâmicos em objetos aninhados

Referem-se à **utilização de métodos dinâmicos para interagir com objetos aninhados**. Isso permite adicionar funcionalidades dinâmicas a objetos complexos e ajustar o comportamento do código com base em condições ou requisitos específicos, como no exemplo ao lado.

```
let pessoa = {
  nome: "João",
  endereco: {
    rua: "Rua Principal",
    cidade: "Cidade A"
  },
  adicionarPrefixo: function(prefixo) {
    this.nome = `${prefixo} ${this.nome}`;
  }
};

// Chamando o método existente
console.log(pessoa.nome); // João

// Chamando o método dinâmico adicionado
pessoa.adicionarPrefixo("Sr.");
console.log(pessoa.nome); // Sr. João
```

Neste exemplo, o objeto pessoa possui um método chamado **adicionarPrefixo**, que adiciona um prefixo ao nome da pessoa. Esse método é definido quando o objeto é criado.

Vamos analisar juntos?

```
let loja = {
    nome: "SuperTech",
    produtos: [
        { nome: "Smartphone", preco: 899 },
        { nome: "Laptop", preco: 1499 }
    ],
    calcularTotal() {
        let total = this.produtos.reduce((acc, produto) => acc + produto.
        console.log(`Total: $${total}`);
    }
};

loja.calcularTotal(); // Saída: Total: $2398
```

Após analisar este código, você consegue identificar onde o método dinâmico foi usado?

Os métodos dinâmicos podem ser úteis em situações em que é preciso adaptar o comportamento do seu código com base em condições ou requisitos específicos, evitando a necessidade de criar métodos estáticos para cada situação. Isso contribui para um código mais flexível e fácil de manter.

DESAFIO PRÁTICO

Explorando métodos e propriedades de objetos em JavaScript

Descrição

Você foi contratado como desenvolvedor(a) para aprimorar uma aplicação web robusta que gerencia informações de uma biblioteca digital. A aplicação já possui uma base sólida, mas é necessário aprofundar-se nos conceitos de métodos e propriedades de objetos em JavaScript para otimizar a manipulação e organização de dados.

A biblioteca digital contém uma vasta coleção de livros e documentos multimídia. Os usuários da plataforma desejam ter uma experiência mais rica ao interagir com esses recursos, incluindo a capacidade de marcar favoritos, obter recomendações personalizadas e explorar conteúdo de maneira mais eficiente.

Objetivos

- Implementar um sistema de marcação de favoritos que permita aos usuários salvar e organizar os seus itens preferidos na biblioteca;
- Desenvolver um algoritmo que analise o histórico de interações do usuário e forneça recomendações personalizadas com base em suas preferências;
- Criar funcionalidades de filtragem e ordenação que permitam aos usuários explorar rapidamente diferentes categorias de conteúdo na biblioteca;
- Incorporar métodos de edição e exclusão para que os usuários possam gerenciar eficientemente os conteúdos que contribuem para a biblioteca;
- Integrar dados de fontes externas, como informações de avaliação de livros ou dados de autores, usando APIs para enriquecer a experiência do usuário.

Orientações

- Implemente um sistema de favoritos que permita aos usuários adicionar e remover itens da biblioteca, bem como visualizar facilmente seus favoritos salvos;
- Desenvolva um algoritmo de recomendação que analise as interações passadas do usuário e forneça sugestões personalizadas com base em padrões identificados;
- Crie opções de filtragem e ordenação para diferentes tipos de conteúdo, como livros, documentos ou multimídia, facilitando a navegação;
- Incorpore métodos de edição e exclusão para que os usuários possam gerenciar suas contribuições para a biblioteca, mantendo-a organizada;
- Utilize APIs externas para obter informações adicionais sobre livros, autores ou qualquer outra categoria relevante presente na biblioteca.

O JavaScript, sendo uma linguagem orientada a objetos, proporciona um vasto universo de recursos para modelagem de dados. Neste capítulo, aprendemos a criar e manipular objetos, entendendo as suas propriedades e os seus métodos. Ao estudar sobre os métodos personalizados, descobrimos como atribuir funções específicas a objetos, tornando-os dinâmicos e interativos. Além disso, exploramos funções avançadas em objetos, aprendemos a utilizar construtores, protótipos e herança para criar estruturas mais complexas.

Neste capítulo, também exploramos como modelar objetos aninhados, interagir dinamicamente com métodos e utilizar funções construtoras, conhecimentos que ajudam os desenvolvedores a ganharem flexibilidade na representação de dados complexos. A combinação desses elementos proporciona uma compreensão profunda do universo orientado a objetos, permitindo o desenvolvimento de código mais modular, reutilizável e escalável. Esses conceitos são fundamentais e irão capacitar você, futuro(a) desenvolvedor(a), a construir aplicações robustas e eficientes, aproveitando ao máximo os objetos em JavaScript. Por isso, continue explorando e praticando tudo o que você viu aqui, pois esses conceitos são essenciais para que você aprimore suas habilidades e crie aplicações mais sofisticadas em JavaScript.

ATIVIDADE DE FIXAÇÃO

1. Explique o que é um objeto em JavaScript e como podemos criar e manipular as suas propriedades.

2. Diferencie os métodos personalizados dos métodos predefinidos em objetos, dando exemplos práticos de ambos.

3. Como a utilização de funções, como propriedades de objetos, pode tornar os objetos mais dinâmicos? Dê um exemplo prático.

4. Explique a importância dos objetos aninhados em JavaScript e forneça um exemplo de como você criaria e manipularia um objeto aninhado.

5. Qual é a diferença entre uma função construtora e uma função normal em JavaScript? Como você utilizaria uma função construtora para criar objetos?

6. Descreva como a herança de objetos é alcançada usando protótipos em JavaScript. Dê um exemplo prático.

7. Por que a utilização de protótipos é valiosa ao criar objetos em JavaScript? Explique como isso contribui para a eficiência do código.

8. Como você definiria e utilizaria um método dinâmico em um objeto? Forneça um exemplo prático.

9. Quais são os benefícios de utilizar métodos avançados como **Object.keys(), Object.values()** e **Object.entries()** ao manipular objetos em JavaScript?

10. Explique como a orientação a objetos em JavaScript pode contribuir para a modularidade e para a reutilização de código em projetos de desenvolvimento.

CAPÍTULO 09

INTERATIVIDADE AVANÇADA NA WEB COM JAVASCRIPT: MANIPULAÇÃO E GERENCIAMENTO DE EVENTOS E VALIDAÇÃO DE FORMULÁRIOS

O que esperar deste capítulo:

- Desenvolvimento de manipuladores de eventos eficientes;
- Gerenciamento de eventos JavaScript;
- Validação avançada de formulários.

9.1 Uso de manipuladores de eventos para responder a ações do usuário

Para entendermos o conceito de manipuladores de eventos, primeiro é preciso relembrar o conceito de evento e o que isso significa em JavaScript. Nessa linguagem, os **eventos são ocorrências que acontecem no navegador ou no ambiente em que o código JavaScript está sendo executado em resposta a ações do usuário**. Por exemplo, no contexto de uma página *web*, essas ações podem ser cliques, rolagens de tela, pressionamentos de teclas, eventos disparados programaticamente, como animações ou temporizadores, e muito mais.

Quando um evento ocorre, algum código é executado em resposta a essas ações e é aqui que entram os **manipuladores de eventos**. Eles são funções que são executadas em resposta a eventos específicos que ocorrem no navegador.

Neste capítulo, nós vamos conhecer mais sobre esses manipuladores e aprenderemos como usá-los ao lidar com os eventos. Em seguida, vamos entender como fazer o gerenciamento desses eventos e finalizaremos abordando a validação de formulários, elementos que permitem que os usuários interajam com uma página da *web*.

Ao final deste capítulo, você será um *expert* na criação de páginas interativas e que respondem de forma eficaz às ações do usuário, proporcionando uma experiência mais dinâmica e envolvente na *web*.

9.1.1 Manipuladores de eventos

Assim como vimos anteriormente, os manipuladores de eventos desempenham um papel crucial na criação de interatividade em páginas da *web*. Eles consistem em funções que são acionadas em resposta a eventos específicos ocorridos no navegador. Quando esses eventos ocorrem, as funções associadas são chamadas automaticamente, permitindo uma resposta dinâmica às ações do usuário. Em resumo, os manipuladores de eventos são essenciais para tornar as páginas *web* mais interativas e responsivas.

Há várias maneiras de se adicionar manipuladores de eventos a elementos HTML em JavaScript. Aqui, vamos explorar as mais comuns.

Atributos HTML

Você pode adicionar um manipulador de eventos diretamente como um atributo de um elemento HTML. Essa abordagem é simples, mas mistura JavaScript com o HTML, o que pode dificultar a manutenção do código.

```
<button onclick="alert('Você clicou em mim!')">Clique Aqui</button>
```

Propriedade do evento no DOM

Você pode atribuir uma função a uma propriedade de evento de um elemento. Essa técnica mantém o JavaScript separado do HTML, o que é uma **prática recomendada**.

```
<button id="meuBotao">Clique Aqui</button>
<script>
  document.getElementById("meuBotao").onclick = function() {
    alert("Você clicou no botão!");
  };
</script>
```

Método addEventListener()

A abordagem mais flexível e recomendada é usar o método **addEventListener()**, pois ele **permite adicionar múltiplos ouvintes de eventos ao mesmo elemento** e, também, usar eventos que não são suportados por todos os navegadores por meio de atributos de eventos HTML ou propriedades de eventos DOM.

```
<button id="meuBotao2">Clique Aqui</button>
<script>
  let botao = document.getElementById("meuBotao2");
  botao.addEventListener("click", function() {
    alert("Você clicou no botão!");
  });
</script>
```

Ao utilizar a abordagem **addEventListener()** para adicionar manipuladores de eventos, existem algumas opções de eventos mais comuns que são utilizados em aplicativos *web*, como:

click: ocorre quando um elemento é clicado.

mouseover: acionado quando o *mouse* passa sobre um elemento.

mouseout: acionado quando o *mouse* sai de um elemento.

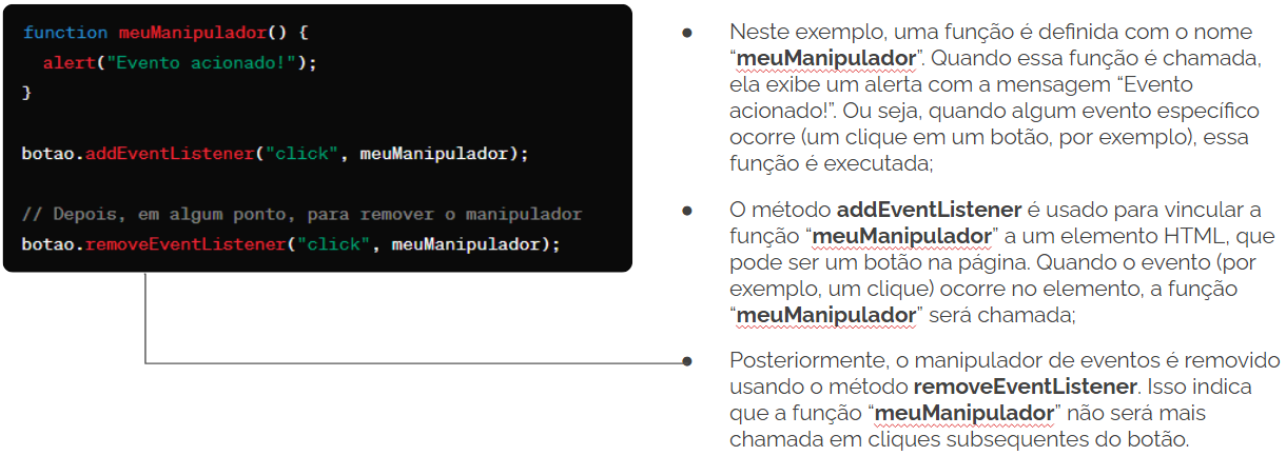
keydown: ocorre quando uma tecla é pressionada.

load: acionado quando o recurso terminou de carregar.

9.1.2 Removendo manipuladores de eventos

A remoção de um manipulador de eventos em JavaScript é realizada pelo método **removeEventListener()**. No entanto, para remover um manipulador de eventos específico,

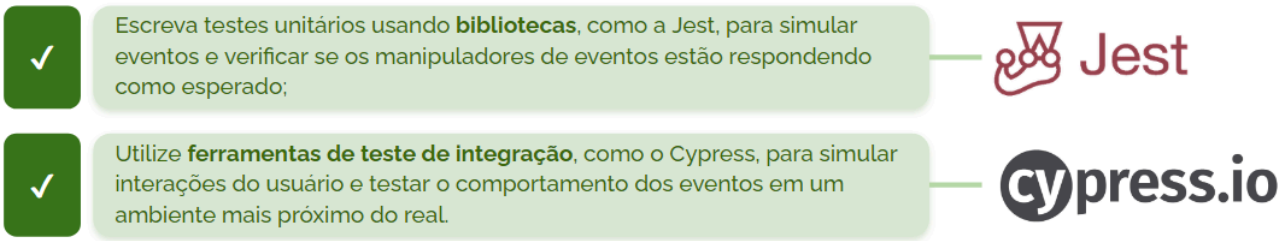
você precisa passar a mesma função de *callback* que foi usada para adicioná-lo. Isso ocorre porque o JavaScript associa a função de *callback* original ao manipulador de eventos e a utiliza como uma referência para removê-la posteriormente. Veja o exemplo a seguir:



Essa abordagem é fundamental porque **permite a identificação específica do manipulador de eventos que você deseja remover**, especialmente quando há vários manipuladores associados a um mesmo tipo de evento. Caso contrário, o JavaScript não saberia qual manipulador de eventos remover, podendo levar a comportamentos inesperados na sua aplicação.

9.1.3 Testando manipuladores de eventos

Testar os manipuladores de eventos em aplicações JavaScript é fundamental para garantir a funcionalidade e a confiabilidade da interação do usuário. Isso inclui verificar se os eventos disparam as funções corretas e se elas executam as tarefas pretendidas corretamente. Para fazer isso, você deve colocar em prática as seguintes orientações:



Realizar o teste dos manipuladores de eventos tem o objetivo de garantir uma interação robusta e sem falhas na aplicação. Com isso, você poderá desenvolver uma base de código confiável e oferecer uma experiência excelente ao usuário .

Vamos praticar?

Para praticar o que vimos até aqui, vamos **criar um exemplo em que um usuário pode interagir com elementos na página**. Utilize um ambiente de desenvolvimento de sua preferência, como o **CodePen** (https://codepen.io/) ou o **Visual Studio Code** (https://code.visualstudio.com/) e **reproduza o código ao lado**.

```
<div id="info">Passe o mouse sobre mim!</div>
<script>
  let infoDiv = document.getElementById("info");

  infoDiv.addEventListener("mouseover", function() {
    this.style.backgroundColor = "yellow";
    this.textContent = "Mouse está aqui!";
  });

  infoDiv.addEventListener("mouseout", function() {
    this.style.backgroundColor = "";
    this.textContent = "Passe o mouse sobre mim!";
  });
</script>
```

E aí, conseguiu completar a tarefa?

O que você notou que esse código faz?

Neste exemplo, quando o usuário passa o *mouse* sobre a **div**, a cor de fundo e o texto mudam e, quando o *mouse* sai, eles voltam ao normal.

Você conseguiu o mesmo resultado? Aproveite a oportunidade para colocar em prática todos os seus conhecimentos sobre eventos, sobre a linguagem JavaScript e sobre os códigos HTML.

9.2 Gerenciamento de eventos em JavaScript

Assim como já vimos, os eventos são ações ou ocorrências que acontecem no sistema que está sendo manipulado. Saber gerenciar esses eventos é fundamental para que a interatividade em aplicações *web* aconteça de forma eficiente. O gerenciamento de eventos em JavaScript segue um fluxo específico e, aqui, nós vamos conhecer cada etapa desse processo.

1 **Seleção do elemento DOM:** primeiro, é necessário selecionar o elemento DOM (*Document Object Model*) que você quer associar ao ouvinte de eventos (**EventListener**). Isso pode ser feito usando métodos como **document.getElementById()**, **document.querySelector()** ou similares.

2 **Registro do EventListener:** após selecionar o elemento, você deve registrar um ouvinte de evento (**EventListener**) usando o método **addEventListener()**. Este método recebe dois argumentos essenciais: o **nome do evento** (como "**click**" ou "**mouseover**") e a **função de *callback*** a ser executada quando o evento ocorrer. Veja um exemplo de como essa etapa deve ficar em um código:

```
const button = document.getElementById('myButton');
button.addEventListener('click', function() {
  console.log('Botão clicado!');
});
```

Este código cria uma referência para um botão HTML com o ID **'myButton'** e adiciona um ouvinte de eventos para o evento de clique nesse botão. Quando o botão é clicado, a mensagem 'Botão clicado!' é exibida no console.

3 **Propagação de eventos:** esta etapa pode ocorrer de duas maneiras: **borbulhamento (*bubbling*)** e **captura (*capturing*)**. Por padrão, os eventos em JavaScript propagam na fase de borbulhamento.

Bubbling	*Capturing*
O evento começa no elemento mais interno e se propaga até os elementos ancestrais.	O evento começa no elemento mais externo e se propaga até o elemento mais interno.

Você pode controlar esse comportamento no terceiro argumento do addEventListener(), definindo-o como *true* para captura ou *false* (padrão) para borbulhamento.

Fique atento!

Utilizando o **event.stopPropagation()**, podemos **parar a propagação de um evento**, evitando que ele continue para a próxima fase. Por sua vez, o método **event.preventDefault()**, **impede o comportamento padrão do evento**. Você pode fazer isso de acordo com o passo a passo abaixo:

- ✓ Adicione um ouvinte de eventos a múltiplos elementos aninhados no DOM;
- ✓ Use **event.target** para identificar o elemento que disparou o evento;
- ✓ Aplique **event.stopPropagation()** em um manipulador de evento para prevenir a propagação;
- ✓ Utilize **event.preventDefault()** em eventos de *link* (**<a>**) para demonstrar como prevenir a navegação padrão.

4

Remoção de ouvintes de eventos (EventListener): se necessário, é possível remover um ouvinte de evento usando o método **removeEventListener()**. Isso é útil para parar de ouvir eventos em certas condições ou para evitar vazamentos de memória.

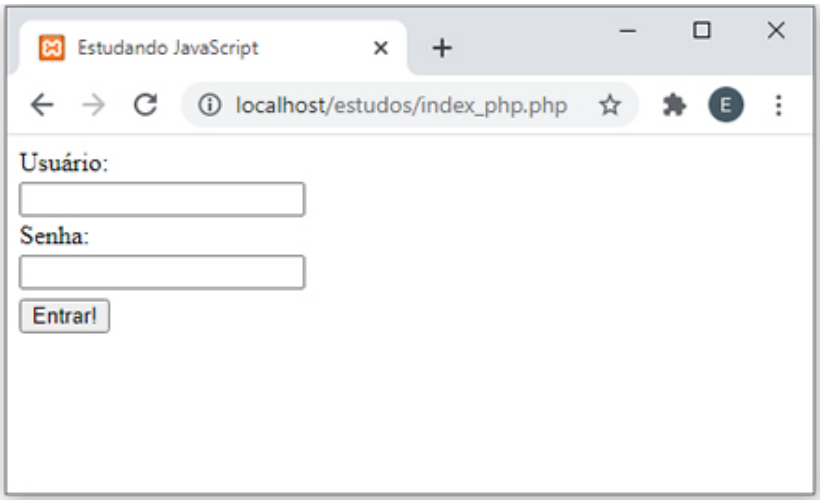
```
// Função para remover o botão e seu ouvinte de evento
function removerBotao(botao, handler) {
    // Removendo o ouvinte de evento
    botao.removeEventListener('click', handler);

    // Removendo o botão do documento
    document.body.removeChild(botao);
}

// Removendo o botão e seu ouvinte de evento
removerBotao(botao, cliqueHandler);
```

Neste exemplo, ao chamar a função **removerBotao**, tanto o botão quanto seu ouvinte de evento serão removidos, permitindo que o navegador libere a memória ocupada por eles.

Essas etapas compõem o processo geral de gerenciamento de eventos em JavaScript, permitindo que você responda às interações do usuário e outros eventos no navegador de forma dinâmica, criando uma página responsiva e interativa.

E aí, está conseguindo entender o assunto ou ficou com alguma dúvida?
Se necessário, fale com seu(a) professor(a) para esclarecer alguns pontos que você ainda não conseguiu entender. Estamos apenas no início da nossa jornada de aprendizado, então vamos continuar explorando a implementação de eventos em JavaScript juntos. Vamos lá!?

9.2.1 Eventos em formulários



Disponível em: https://encurtador.com.br/JUVY5. Acesso em: 05 mar. 2024.

No contexto dos formulários, o gerenciamento de eventos é crucial por diversos motivos, como:

✓ Validar dados de entrada antes de enviar um formulário

Antes de enviar um formulário, é comum validar os dados inseridos pelo usuário para garantir que estejam de acordo com os requisitos especificados. O evento mais comum usado para isso é o evento **submit**, que ocorre quando o usuário tenta enviar o formulário. O gerenciamento desse evento permite que você execute funções de validação e impeça o envio se os dados não estiverem corretos.

✓ Impedir o envio de formulários se necessário

Além da validação de dados, você pode querer impedir o envio do formulário em certas condições. Isso pode ser feito utilizando o evento **submit** e chamando **event.preventDefault()** quando necessário.

✓ Manipular eventos de campos de entrada

Campos de entrada, como caixas de texto, seleções e botões, têm diferentes eventos associados a eles. Por exemplo, o evento **change** é acionado quando o valor de um campo de entrada é alterado, enquanto o evento **input** é acionado continuamente durante a digitação. Manipular esses eventos permite que você reaja dinamicamente às mudanças nos campos de entrada.

Esses são apenas alguns exemplos de como o gerenciamento de eventos é utilizado em formulários para melhorar a experiência do usuário, garantindo, também, a integridade dos dados submetidos. Porém, vamos abordar melhor o conceito de formulários mais adiante.

Event delegation

A delegação de eventos em JavaScript é uma técnica que aproveita a propagação de eventos (*bubbling*) no DOM para **gerenciar eventos de uma maneira mais eficiente**. Em vez de atribuir um manipulador de eventos a cada elemento individual, **você adiciona um único ouvinte de eventos a um elemento pai para gerenciar todos os eventos que se propagam dos seus elementos filhos**. Ou seja, você atribui um único manipulador a um ancestral comum dos elementos e, em seguida, determina qual filho específico gerou o evento. Isso é útil para elementos adicionados dinamicamente ao DOM. Veja um passo a passo de como isso deve ser feito:

✓ Crie um elemento contêiner no HTML;

- ✓ Adicione um ouvinte de eventos ao elemento pai;
- ✓ No manipulador de eventos, use **event.target** para identificar o elemento filho que disparou o evento para que ele responda apropriadamente.

Eventos personalizados

Eventos personalizados são **eventos definidos pelo desenvolvedor**, sendo específicos para as necessidades da aplicação que está sendo desenvolvida. Essa personalização permite a criação de mecanismos específicos para lidar com a comunicação entre componentes, módulos ou partes diferentes de uma aplicação. Isso proporciona mais flexibilidade ao código, além dos eventos padrão do DOM. A seguir, veja um passo a passo de como isso deve ser feito:

- ✓ Use **new CustomEvent('nomeDoEvento', { detail: { /* dados */ } })** para criar um evento personalizado;
- ✓ Adicione um ouvinte para esse evento personalizado em um elemento relevante;
- ✓ Dispare o evento personalizado em um momento apropriado usando **element.dispatchEvent(eventoPersonalizado)**.

Suponha que você esteja desenvolvendo uma aplicação de carrinho de compras e queira notificar outras partes do código sempre que um item for adicionado ao carrinho. Nesse caso, você pode criar um evento personalizado chamado **itemAdicionadoAoCarrinho**. Vamos ver como ficaria o código dessa aplicação:

```
<button id="adicionarItemAoCarrinho">Adicionar Item ao Carrinho</button>

<script>
document.addEventListener('DOMContentLoaded', function() {
  // Função para adicionar item ao carrinho
  function adicionarItemAoCarrinho(nomeItem) {
    // Lógica para adicionar o item ao carrinho (simulado)
    console.log(`Item adicionado ao carrinho: ${nomeItem}`);

    // Disparando evento personalizado
    const eventoItemAdicionado = new CustomEvent('itemAdicionadoAoCarrinho', { detail: { itemName: nomeItem } });
    document.dispatchEvent(eventoItemAdicionado);
  }
```

```
  // Adicionando um ouvinte de eventos ao botão
  const botaoAdicionarItem = document.getElementById('adicionarItemAoCarrinho');
  botaoAdicionarItem.addEventListener('click', function() {
    // Simulando a adição de um item ao carrinho
    const nomeItem = prompt('Digite o nome do item:');
    adicionarItemAoCarrinho(nomeItem);
  });

  // Outras partes da aplicação que desejam saber sobre itens adicionados
  document.addEventListener('itemAdicionadoAoCarrinho', function(event) {
    const itemName = event.detail.itemName;
    console.log(`Notificação: Novo item adicionado ao carrinho - ${itemName}`);
    // Lógica adicional aqui (por exemplo, atualizar a exibição do carrinho)
  });
});
</script>
```

Nesse código, nós conseguimos ver todo o processo de gerenciamento de eventos. Vamos analisá-lo com mais detalhes:

- **botão "Adicionar Item ao Carrinho"**: um botão com o ID **adicionarItemAoCarrinho** é criado na interface do usuário;
- **bloco JavaScript:** o bloco JavaScript começa com um **ouvinte de eventos,** que é acionado quando o DOM é completamente carregado (**DOMContentLoaded**);
- **gunção adicionarItemAoCarrinho**: dentro do bloco, há uma função chamada **adicionarItemAoCarrinho**, que simula a adição de um item ao carrinho. Ela também **dispara um evento personalizado** chamado **itemAdicionadoAoCarrinho**, incluindo detalhes sobre o item adicionado;
- **ouvinte de evento no botão**: um ouvinte de eventos é adicionado ao botão com ID **adicionarItemAoCarrinho**. Quando o botão é clicado, a função **adicionarItemAoCarrinho** é chamada, solicitando ao usuário que insira o nome do item;
- **ouvinte de evento personalizado**: outra parte do código registra um ouvinte de eventos para o evento personalizado **itemAdicionadoAoCarrinho**. Quando esse evento é disparado (ou seja, quando um item é adicionado), a função associada a ele é executada, exibindo uma mensagem de notificação no console.

Esse exemplo mostra um cenário simples em que um evento personalizado é usado para notificar outras partes da aplicação quando ele é acionado. Isso ilustra como a personalização de eventos pode ser útil para a comunicação entre diferentes partes de uma aplicação.

Disponível em: https://www.shutterstock.com/pt/image-vector/shopping-girl-sale-discounts-concept-young-1521166121. Acesso em: 05 mar. 2024.

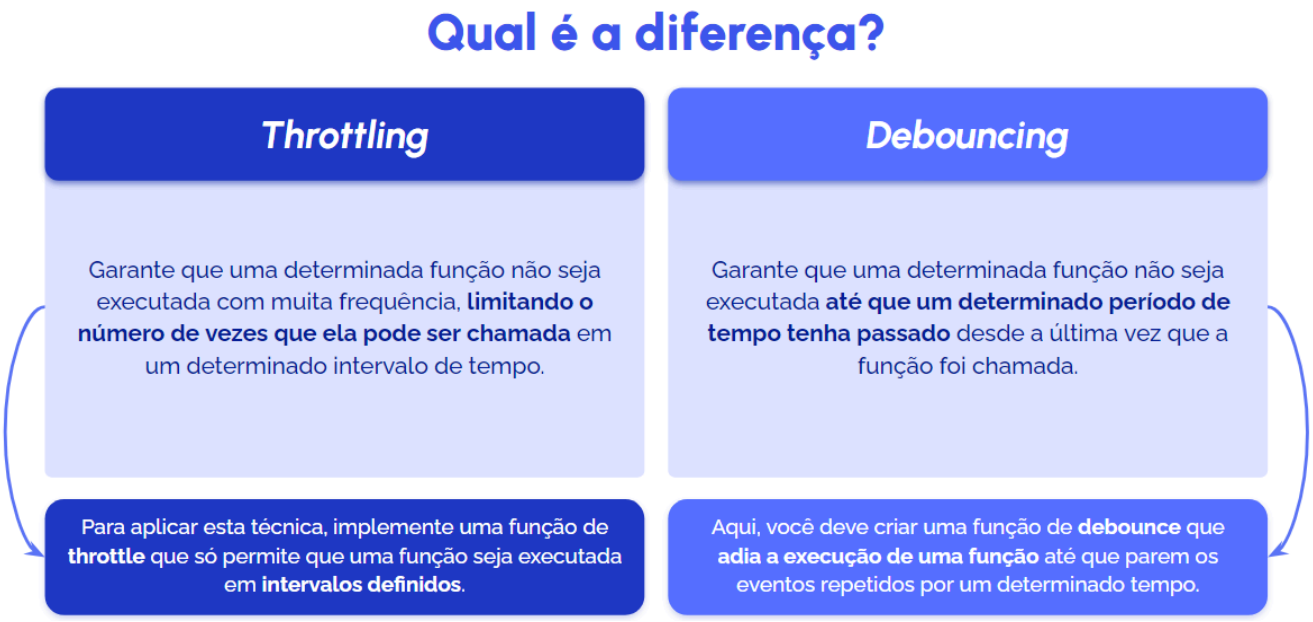
Throttling e debouncing

Throttling e *debouncing* são técnicas utilizadas para **otimizar a performance e melhorar a eficiência em situações em que eventos frequentes**, como o redimensionamento de janela, a digitação ou a rolagem de tela. Sabe quando você fica rolando o *feed* do seu aplicativo favorito? Pode ter certeza que o *throttling* e o *debouncing* estão em ação!

Disponível em: https://encurtador.com.br/ioCVY. Acesso em: 05 mar. 2024.

Essas ações do usuário podem levar a execuções excessivas de código, podendo impactar negativamente no desempenho da aplicação. As duas as técnicas controlam a frequência com que uma função é chamada em resposta a esses eventos.

Para que você entenda melhor como essas técnicas funcionam e consiga usá-las em suas aplicações, veja o esquema a seguir:



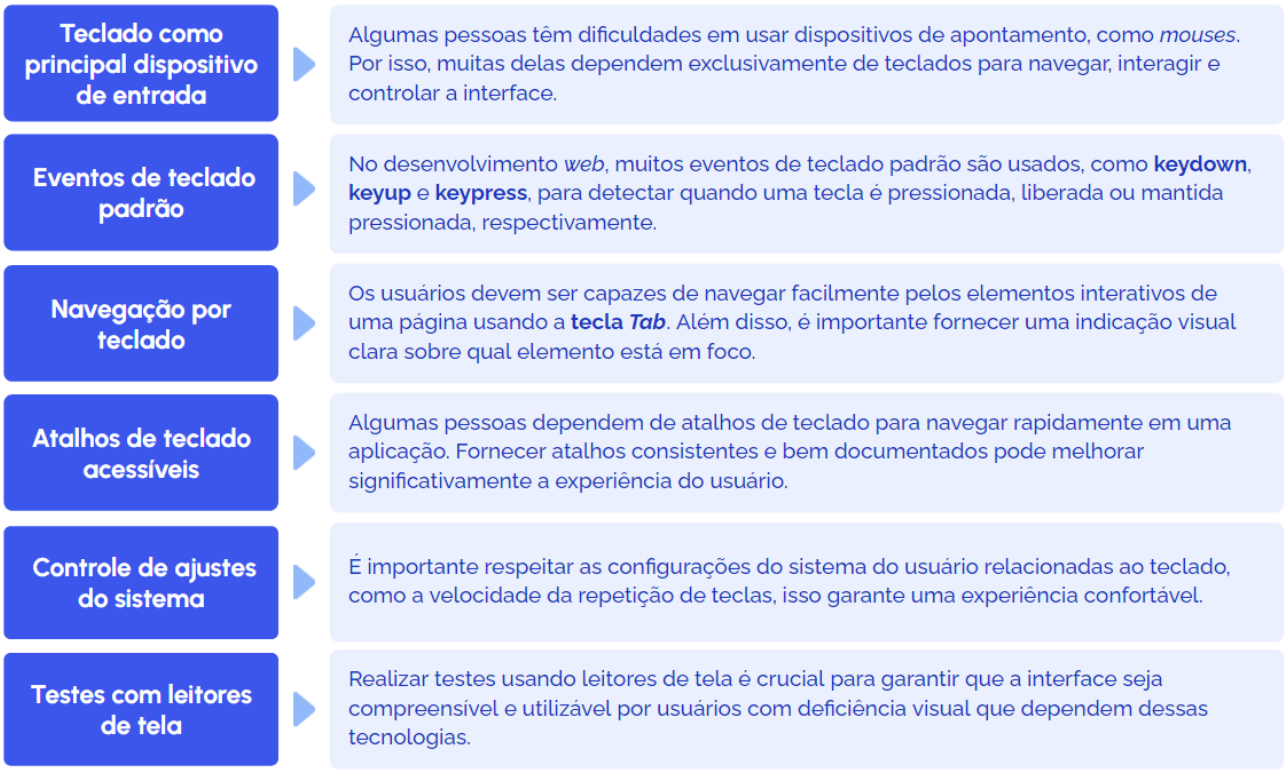
Mas... quando usar cada técnica? O *throttling* é útil quando você quer garantir que uma ação seja executada em intervalos regulares, mas não necessariamente na última chamada. Por exemplo, no redimensionamento de janela, na rolagem etc. Já o *debouncing* é usado quando você quer atrasar a execução de uma ação até que o usuário tenha cessado a atividade, garantindo que a última chamada seja efetuada após um intervalo de inatividade. Por exemplo, sugestões de pesquisa enquanto o usuário digita, validação de entrada etc.

As duas técnicas são ferramentas valiosas para otimizar a experiência do usuário e melhorar a eficiência em aplicações *web* e a sua escolha vai depender do contexto e dos requisitos específicos de cada situação.

Acessibilidade e eventos do teclado

A acessibilidade em desenvolvimento *web* refere-se à prática de criar interfaces digitais que sejam inclusivas e acessíveis a todas as pessoas, incluindo aquelas com deficiências ou limitações de qualquer outro tipo. Nesse contexto, eventos do teclado desempenham um papel fundamental na acessibilidade, já que muitos usuários dependem de dispositivos de

entrada baseados em teclado para conseguirem interagir com a *web*. A seguir, conheça alguns conceitos e práticas relacionados à acessibilidade e eventos do teclado:



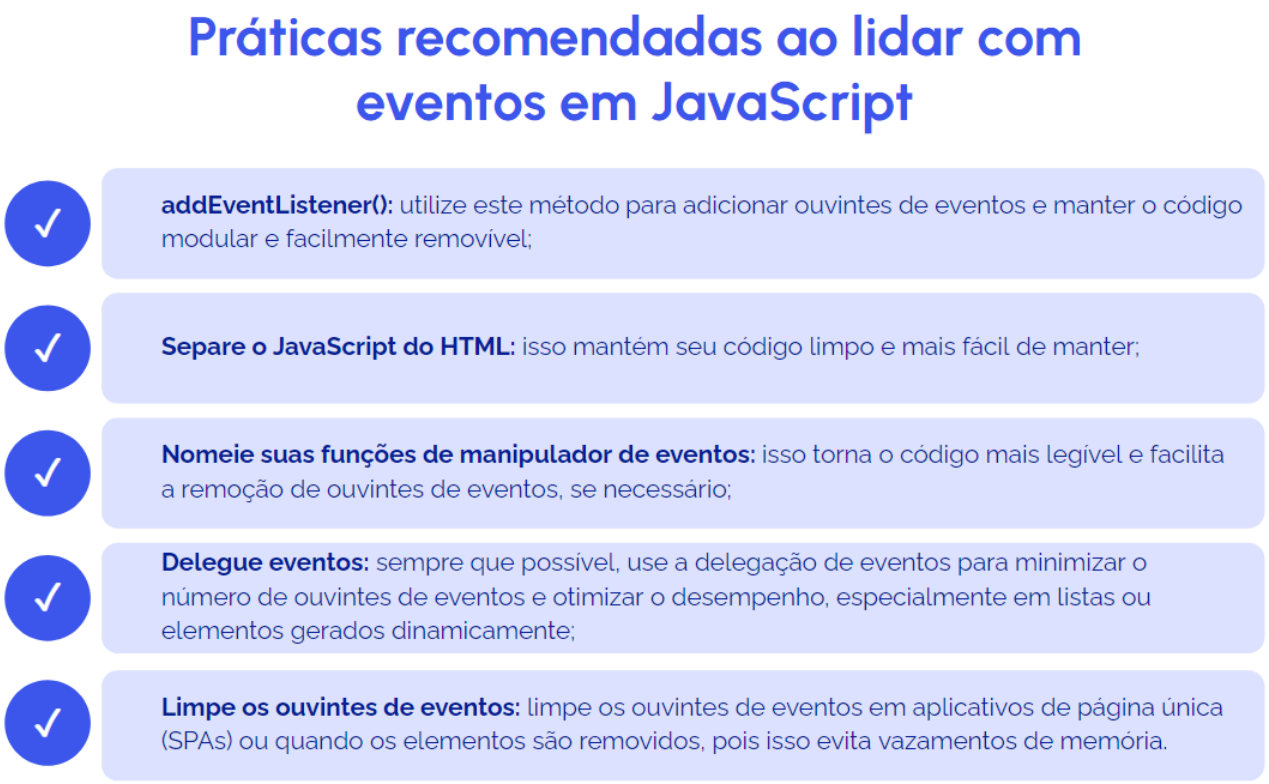
Para fazer o gerenciamento dos eventos de teclado, você deve:

- usar eventos de teclado como **keydown** para **adicionar interatividade acessível**;
- implementar **manipuladores de eventos** que respondam a teclas específicas (por exemplo, *Enter* ou Espaço) para **simular os cliques do *mouse***.

A acessibilidade é uma parte essencial do *design* e do desenvolvimento *web* e, ao considerar os eventos do teclado e implementar práticas que facilitam a interação de usuários que dependem do teclado como dispositivo de entrada, você contribui para uma experiência *on-line* mais inclusiva.

9.2.2 Práticas recomendadas

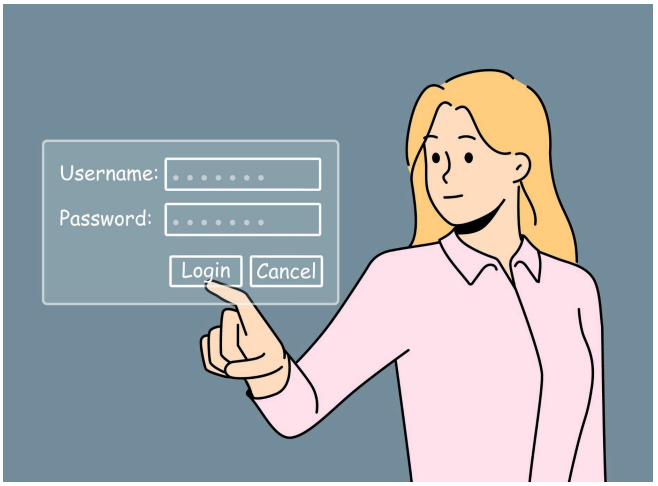
Ao realizar a manipulação e o gerenciamento de eventos em JavaScript, algumas práticas recomendadas podem melhorar a eficiência, a legibilidade e a manutenibilidade do código. Conheça algumas diretrizes no quadro a seguir:



Adotar essas práticas ajudam a criar um código JavaScript mais robusto, eficiente e fácil de manter, promovendo boas práticas de desenvolvimento *web*.

9.3 Validação de formulários com JavaScript

A validação de formulários é uma etapa crucial no desenvolvimento de aplicações *web*, garantindo que os dados inseridos pelos usuários sejam corretos e completos antes de serem enviados ao servidor. A linguagem JavaScript oferece uma maneira flexível e poderosa de validar formulários no lado do cliente, melhorando a experiência do usuário ao fornecer *feedbacks* imediatos sobre os erros de entrada. Agora, vamos estudar como você pode usar o JavaScript para validar formulários de maneira eficaz.



Disponível em: https://encurtador.com.br/uyBHO. Acesso em: 06 mar. 2024.

Antes de enviar os dados de um formulário, é importante verificar se eles estão corretos. Isso inclui, por exemplo, garantir que o usuário preencheu todos os campos obrigatórios, que o endereço de *e-mail* esteja em um formato válido e que as senhas inseridas correspondem.

9.3.1 Estrutura básica de um formulário HTML

Para começar, vamos analisar um exemplo simples de formulário HTML que precisaremos validar:

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

  <label for="senha">Senha:</label>
  <input type="password" id="senha" required minlength="8">

  <label for="confirmaSenha">Confirme a Senha:</label>
  <input type="password" id="confirmaSenha" required>

  <button type="submit">Registrar</button>
</form>
<div id="mensagemErro"></div>
```

Veja o passo a passo da validação do formulário que está sendo criado a partir deste código a seguir:

1 Validação com JavaScript

Para validar este formulário com JavaScript, primeiro precisamos interceptar o evento de envio do formulário para então verificar os dados.

2 Interceptando o envio do formulário

Você pode adicionar um ouvinte de eventos ao formulário para escutar o evento **submit**. Isso permite que você execute uma função de validação quando o usuário tenta enviar o formulário:

```
const form = document.getElementById('meuFormulario');
form.addEventListener('submit', function(event) {
  event.preventDefault(); // Previne o envio do formulário
  validaFormulario();
});
```

3 Função de validação

A função **validaFormulario** pode então verificar cada campo, como podemos acompanhar nos códigos a seguir:

3.1

```
// Validação da senha
if (senha.length < 8) {
  mensagemErro += 'A senha deve ter pelo menos 8 caracteres.\n';
}

// Confirmação da senha
if (senha !== confirmaSenha) {
  mensagemErro += 'As senhas não coincidem.\n';
}
```

3.2

```
function validaFormulario() {
  const email = document.getElementById('email').value;
  const senha = document.getElementById('senha').value;
  const confirmaSenha = document.getElementById('confirmaSenha').value;
  let mensagemErro = '';

  // Validação do email
  if (!email) {
    mensagemErro += 'Email é obrigatório.\n';
  } else if (!/^\S+@\S+\.\S+$/.test(email)) {
    mensagemErro += 'Email inválido.\n';
  }
```

3.3

```
// Exibindo erros
if (mensagemErro) {
  document.getElementById('mensagemErro').textContent = mensagemErro;
  return false; // Impede o envio do formulário
} else {
  // Aqui, você pode prosseguir com o envio do formulário ou processamento adicional
  alert('Formulário validado com sucesso!');
}
```

E aí, conseguiu acompanhar a leitura desses códigos?
A habilidade de ler e compreender um código é fundamental para que você desenvolva as suas habilidades em programação.

9.3.2 HTML5 e a validação de formulários

O HTML5 introduziu várias técnicas de validação de formulários que podem ser usadas junto com a validação JavaScript para melhorar a experiência do usuário. Isso inclui atributos como **required**, **minlength**, **maxlength**, **pattern** e tipos de **input**, como *e-mail* e URL, que oferecem uma camada adicional de validação sem necessidade de código JavaScript.

Por exemplo, o navegador automaticamente verifica se um campo com o atributo **required** foi preenchido antes de permitir o envio do formulário. Da mesma forma, o tipo de **input email** verifica se o usuário inseriu um endereço de *e-mail* em um formato válido.

Com o HTML5, várias validações podem ser realizadas automaticamente pelo navegador, simplificando o processo de garantir que os dados de entrada sejam corretos e completos. Esses recursos de validação integrados na quinta versão do HTML simplificam significativamente a tarefa de validar formulários, melhorando a experiência do usuário e reduzindo a quantidade de código JavaScript necessário para validar entradas de formulário. No entanto, ainda é importante realizar validações no lado do servidor como uma camada adicional de segurança, já que a validação do lado do cliente pode ser contornada.

DESAFIO PRÁTICO

Aprimorando a interação do usuário com manipuladores de eventos em JavaScript

Descrição

Você é parte de uma equipe de desenvolvimento encarregada de revitalizar a interface de usuário de um aplicativo de produtividade *on-line*. O aplicativo permite que os usuários criem, gerenciem e compartilhem listas de tarefas, mas a interação atual é considerada rígida e pouco intuitiva. A sua missão é utilizar manipuladores de eventos em JavaScript para tornar a experiência do usuário mais dinâmica e responsiva.

O aplicativo de produtividade tem como alvo usuários que buscam uma ferramenta eficiente para o gerenciamento de tarefas diárias. Contudo, *feedbacks* dos usuários indicam dificuldades em interações básicas como adicionar, editar ou excluir tarefas, além de uma falta geral de *feedback* visual durante essas ações. A sua tarefa é melhorar essas interações implementando manipuladores de eventos que respondam de forma imediata às ações do usuário.

Objetivos

- Implementar um manipulador de eventos para o botão "Adicionar Tarefa" que permita a inserção instantânea de novas tarefas na lista;
- Desenvolver um sistema de edição *in-loco* para as tarefas, permitindo aos usuários modificar uma tarefa diretamente na interface da lista;
- Criar um manipulador de eventos para um ícone ou botão de exclusão em cada tarefa, facilitando a remoção de itens indesejados com confirmação instantânea;
- Implementar *feedbacks* visuais para ações do usuário, como animações suaves ou alterações de cor, para indicar a conclusão de uma ação (como adicionar, editar e excluir);
- Incorporar manipuladores de eventos para ações do teclado, permitindo aos usuários a utilização de atalhos para uma navegação mais eficiente no aplicativo.

Orientações

- Utilize o evento **click** para capturar a ação de adicionar e implemente uma função que atualize a lista de tarefas de forma reativa;
- Aplique manipuladores de eventos como **dblclick** para habilitar a edição *in-loco*, utilizando campos de texto temporários que salvam as alterações ao perder o foco (*blur*) ou pressionar *Enter*;
- Associe o evento **click** a ícones de exclusão em cada tarefa, incluindo uma confirmação de exclusão para evitar remoções acidentais;
- Implemente transições CSS ou animações JavaScript acionadas por eventos para realçar a interação do usuário com a aplicação, melhorando a experiência visual;
- Use o evento **keydown** para detectar ações de teclado específicas, permitindo atalhos como "Adicionar nova tarefa" ou "Salvar edição" sem necessidade de interação direta com o *mouse.*

Nesse capítulo, aprendemos que o uso de manipuladores de eventos em JavaScript desempenha um papel fundamental na criação de interatividade em páginas *web*, permitindo que o código responda de maneira dinâmica às ações do usuário. Vimos que esses manipuladores são responsáveis por capturar eventos como cliques, movimentos do *mouse*, pressionamentos de tecla etc., proporcionando uma experiência mais envolvente aos usuários. Ao adicionar ouvintes de eventos a elementos HTML, seja por meio de atributos, propriedades ou métodos como **addEventListener()**, os desenvolvedores podem criar interfaces dinâmicas e responsivas, melhorando significativamente a usabilidade das aplicações web.

Exploramos que a utilização de manipuladores de eventos permite a execução de funções específicas em resposta a interações do usuário. Por exemplo, ao clicar em um botão, é possível exibir uma mensagem, alterar o conteúdo de uma página ou realizar ações mais complexas. Por vim, vimos que a capacidade de resposta imediata, aliada à flexibilidade proporcionada pelo JavaScript, contribui para a criação de interfaces intuitivas e interativas, fundamentais para a qualidade e eficácia das experiências digitais na *web*.

ATIVIDADE DE FIXAÇÃO

1. Explique o que são manipuladores de eventos em JavaScript e qual é o propósito principal de sua utilização em páginas *web*.

2. Descreva o processo de adição de um manipulador de eventos a um elemento HTML usando atributos HTML. Quais são as limitações dessa abordagem?

3. Como você adicionaria dinamicamente um manipulador de eventos a um botão usando JavaScript, preferencialmente sem misturar o código HTML e o JavaScript?

4. Discuta a diferença entre a fase de captura e a fase de bolha na propagação de eventos no DOM. Por que isso é importante ao lidar com manipuladores de eventos?

5. Explique o conceito de delegação de eventos e forneça um exemplo prático de sua aplicação. Quais são as vantagens dessa abordagem?

6. Qual é a importância da função **event.preventDefault()** em manipuladores de eventos? Forneça um exemplo de situação em que essa função seria necessária.

7. Como você removeria um manipulador de eventos específico de um elemento usando JavaScript? Por que a remoção de eventos pode ser necessária em certos casos?

8. Aborde a necessidade de validação de formulários usando manipuladores de eventos. Como você garantiria que um formulário só seja enviado quando os dados inseridos são válidos?

9. Descreva a diferença entre os eventos de teclado **keydown** e **keypress**. Como você usaria esses eventos para melhorar a usabilidade de um formulário?

10. Por que é importante considerar a acessibilidade ao implementar manipuladores de eventos em páginas *web*? Quais práticas recomendadas você seguiria para garantir uma experiência acessível aos usuários?

Capítulo 10

AJAX E A COMUNICAÇÃO COM O SERVIDOR

O que esperar deste capítulo:

- Entender o conceito de Ajax e a sua importância;
- Realizar solicitações HTTP assíncronas com JavaScript;
- Entender a manipulação de dados JSON.

10.1 O que é Ajax e qual é a sua importância?

Em termos simples, **Ajax (*Asynchronous JavaScript and XML*)** é uma abordagem que permite que partes específicas de uma página *web* sejam atualizadas sem precisar recarregar a página inteira. Isso contribui para que a experiência do usuário seja mais suave e responsiva.

Como isso acontece na prática?

Imagine que você está preenchendo um formulário de inscrição para um curso *on-line*. Nele, você fornece informações como o seu nome completo, a sua idade e o curso que deseja realizar.

Quando você preenche seu nome e clica em "enviar", os dados do formulário são enviados para o servidor sem recarregar a página inteira. Em seguida, uma mensagem de confirmação é exibida na tela, sem interromper a sua experiência de navegação.

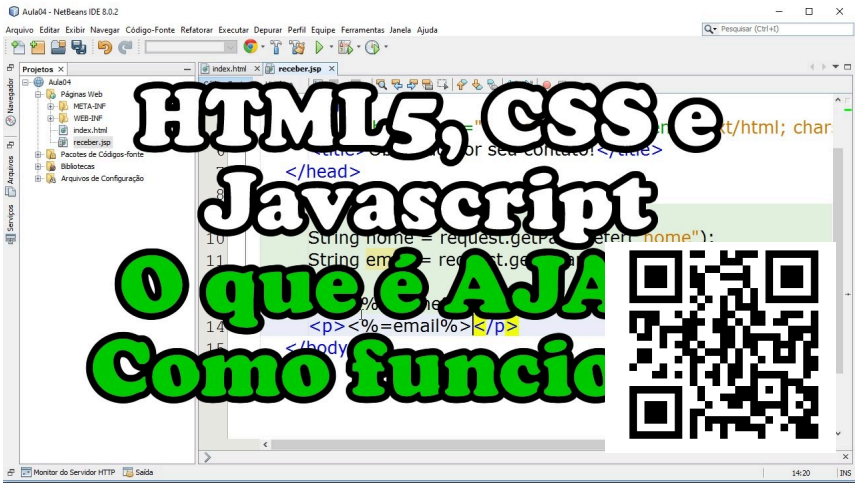
Todo esse processo só ocorre de maneira fluida e responsiva graças ao Ajax!

Importância do Ajax

O Ajax usa JavaScript para enviar e receber dados do servidor de forma assíncrona, o que significa que o usuário pode enviar e receber informações em segundo plano, sem interromper o que o usuário está fazendo. Ou seja, é como ter um assistente invisível trabalhando duro para tornar a sua experiência na *web* mais suave e rápida.

Além disso, o Ajax é uma técnica de programação que permite que os aplicativos *web* se comportem mais como aplicativos de *desktop*.

Para entender melhor essa técnica de programação, assista ao vídeo a seguir e, depois, responda às perguntas propostas:



Fonte: Vídeo *HTML, CSS, JS - O que é AJAX? Como funciona? Entenda com exemplos*, do canal do YouTube 2Guarinos. Disponível em: https://www.youtube.com/watch?app=desktop&v=AbzL8TrtSb8. Acesso em: 18 mar. 2024.

- Após assistir ao vídeo, como você definiria Ajax?
- De modo geral, como o Ajax funciona?
- Cite dois exemplos de uso dos recursos do Ajax.

Situações de uso do Ajax

O Ajax é uma tecnologia amplamente utilizada na *web* moderna para criar experiências mais interativas e responsivas para os usuários. A seguir, vamos conhecer algumas situações em que o seu uso faz toda diferença na experiência do usuário.

✓ Formulários de *login* e registro

Como já vimos, quando um usuário preenche um formulário de *login* ou registro em um *site*, o Ajax pode ser usado para enviar as credenciais do usuário para o servidor e verificar se estão corretas sem recarregar a página inteira. Dessa forma, se as credenciais estiverem corretas, o usuário pode ser redirecionado para a próxima página ou receber uma mensagem de confirmação sem que a página seja recarregada.

✓ Carregamento de conteúdo dinâmico

Em *sites* de comércio eletrônico, por exemplo, o Ajax pode ser usado para carregar detalhes de produtos, avaliações de clientes, ou, até mesmo, adicionar produtos ao carrinho de compras sem a necessidade de recarregar toda a página.

✓ Sistema de comentários em tempo real

Em um *blog* ou em uma plataforma de mídia social, o Ajax pode ser utilizado para permitir que os usuários enviem comentários e visualizem novos comentários sem atualizar a página, criando uma experiência mais interativa e dinâmica.

✓ Sistemas de sugestão e preenchimento automático

Ao preencher um formulário de pesquisa ou cadastro em um *site*, o Ajax pode ser usado para fornecer sugestões de palavras-chave ou opções de preenchimento automático conforme o usuário digita, sem a necessidade de recarregar a página.

✓ Atualização de conteúdo em tempo real

Imagine que você está construindo um *site* de previsão do tempo. Com ajuda do Ajax, você pode fazer com que as atualizações da previsão aconteçam automaticamente, sem que o usuário precise atualizar manualmente a página. Isso mantém as pessoas atualizadas sem qualquer esforço extra.

Quais são as tecnologias que o Ajax utiliza?

Você já entendeu que o Ajax é uma técnica de programação poderosa utilizada no desenvolvimento *web* para criar páginas mais dinâmicas e interativas, sem a necessidade de recarregar a página inteira, não é mesmo? Agora, é importante conhecer as várias tecnologias que ele usa para funcionar adequadamente. Veja o quadro a seguir:

JavaScript

Essa é a principal linguagem de programação utilizada no Ajax. Ela é usada para manipular o DOM (*Document Object Model*) e enviar solicitações assíncronas para o servidor.

XMLHttpRequest (XHR)

É uma API do navegador que permite enviar e receber dados do servidor sem recarregar a página e funciona como a espinha dorsal do Ajax. Um exemplo de uso seria:

```
var xhr = new XMLHttpRequest();
xhr.open('GET', 'dados.html', true);
xhr.onreadystatechange = function() {
  if (xhr.readyState == 4 && xhr.status == 200) {
    document.getElementById('conteudo').innerHTML = xhr.responseText;
  }
};
xhr.send();
```

JSON (JavaScript Object Notation)

Embora o Ajax possa usar XML para trocar dados com o servidor, o JSON é frequentemente preferido devido à sua simplicidade e facilidade de uso. Por exemplo:

```
var dados = { nome: 'João', idade: 25 };
var json = JSON.stringify(dados);
```

HTML e CSS

O CSS é utilizado para estilizar as páginas *web* e o HTML serve para estruturar essas páginas. Além disso, o Ajax pode ser usado para inserir dados recebidos do servidor na página.

Servidores e APIs

Para interagir com um servidor, o Ajax faz solicitações HTTP para *endpoints* específicos que, por sua vez, podem ser alimentados por APIs que retornam dados em formatos como JSON ou XML.

Entendendo o funcionamento do Ajax

Para que seja possível entender o funcionamento do Ajax, vamos trabalhar considerando algumas explicações mais simplificadas sobre essa técnica em funcionamento.

1 Evento do usuário

O processo se inicia quando o usuário interage com a página *web*, clicando em um botão ou preenchendo um formulário, por exemplo.

2 Requisição Ajax

Depois disso, o JavaScript cria, no navegador *web*, uma requisição Ajax para o servidor. Essa requisição pode ser feita utilizando:

- função **XMLHttpRequest**
- métodos mais modernos como **fetch()**

! A requisição Ajax pode incluir dados a serem enviados para o servidor, como informações de formulário ou parâmetros específicos.

3 Processamento no servidor

O servidor recebe a requisição Ajax, processa os dados recebidos e, em seguida, realiza a operação necessária (como acessar um banco de dados, executar cálculos etc.). Só depois que ele prepara uma resposta.

4 Resposta do servidor

Após processar a requisição, o servidor envia uma resposta de volta para o navegador. Essa resposta pode se apresentar nos seguintes formatos:

- JSON
- XML
- HTML

O formato depende do que é necessário para atualizar a página.

5 Manipulação da resposta

No navegador, o JavaScript recebe a resposta do servidor e manipula os dados conforme for necessário. Isso pode incluir atualizar partes da página, exibir mensagens de erro, ou realizar outras ações dinâmicas.

6 Atualização da página

Finalmente, o JavaScript atualiza a página *web*, exibindo os dados recebidos do servidor ou realizando qualquer outra ação necessária para completar o processo.

Todo esse processo parece longo, mas ocorre de maneira muito rápida.

10.2 Fazendo solicitações HTTP assíncronas com JavaScript

Assim como já vimos, um dos recursos tecnológicos que o Ajax utiliza para funcionar corretamente são as solicitações HTTP assíncronas, que desempenham um papel crucial na criação de experiências de usuário mais rápidas e interativas. Ao permitir a comunicação com um servidor em segundo plano, evitamos a espera por recargas de página completas, tornando as aplicações mais eficientes e responsivas.

Utilizando o objeto XMLHttpRequest

Ao fazer solicitações HTTP assíncronas utilizando JavaScript, um recurso muito importante entra em cena: o **XMLHttpRequest**, ou, simplesmente, **XHR**.

O que é o XMLHttpRequest?

- É uma ferramenta do JavaScript que ajuda a enviar e receber dados de um servidor *web* sem ter que recarregar toda a página;
- Basicamente, ele ajuda a tornar as páginas da *web* mais interativas, permitindo que elas se comuniquem com os servidores de forma rápida e eficiente, sem interromper a experiência do usuário. É uma ferramenta fundamental para criar aplicativos da *web* modernos e dinâmicos!

Exemplo de uso

Imagine que você está em uma loja *on-line* e quer adicionar um item ao seu carrinho de compras sem ter que recarregar toda a página. Nesse caso, você usa o **XMLHttpRequest** para enviar um pedido ao servidor dizendo qual item você quer adicionar.

Enquanto espera pela resposta do servidor, você pode continuar navegando pela loja.

Quando o servidor responde, o JavaScript adiciona o item ao seu carrinho de compras sem interromper o que você estava fazendo.

Na prática

O objeto XMLHttpRequest é a espinha dorsal do AJAX quando o assunto é fazer solicitações HTTP assíncronas em JavaScript. Ao criar uma instância desse objeto, abrimos a porta para uma série de funcionalidades assíncronas.

```
var xhr = new XMLHttpRequest();
```

Aspectos importantes sobre o uso do XMLHttpRequest

Para utilizar essa ferramenta de forma adequada, você precisa compreender alguns pontos cruciais. Eles são:

- como configurar uma solicitação;
- como lidar com a resposta assíncrona;
- como enviar uma solicitação.

1. Configurando a solicitação: definindo o método e a URL

Antes de enviar a solicitação, precisamos configurar detalhes como o método (**GET**, **POST** etc.) e a URL de destino. O exemplo abaixo ilustra como configurar uma solicitação **GET**:

```
xhr.open("GET", "https://api.exemplo.com/dados", true);
```

Você conseguiu entender o código?

Estamos chamando o método open do objeto **xhr**. Este método é utilizado para inicializar uma requisição. Neste caso:

- **"GET"**: indica que estamos fazendo uma requisição do tipo GET;
- **"https://api.exemplo.com/dados":** é a URL para a qual estamos fazendo a requisição. No exemplo, é uma URL fictícia, mas, na prática, você substituiria esse espaço pela URL real da API que deseja acessar;
- **true:** indica que a requisição é assíncrona. Isso significa que o código continuará executando outras tarefas enquanto aguarda a resposta do servidor.

2. Lidando com a resposta: assincronia em ação

A resposta do servidor é manipulada através do evento **onreadystatechange**. É aqui que podemos verificar se a solicitação foi concluída com sucesso e processar os dados recebidos.

```
xhr.onreadystatechange = function() {
  if (xhr.readyState == 4 && xhr.status == 200) {
    // Manipulando a resposta do servidor
    console.log(xhr.responseText);
  }
};
```

Você conseguiu entender o código?

- **xhr.onreadystatechange:** aqui, estamos atribuindo uma função anônima ao evento onreadystatechange do objeto **xhr**. Esse evento é acionado sempre que o estado da requisição muda;
- **if (xhr.readyState == 4 && xhr.status == 200):** dentro da função, verificamos se o **readyState** (estado da requisição) é igual a 4 (ou seja, a requisição está completa) e se o *status* (código de *status* HTTP) é igual a 200 (indicando sucesso). Se essas condições forem atendidas, o bloco de código dentro do **if** será executado;
- **console.log(xhr.responseText):**dentro do bloco **if**, estamos imprimindo a resposta do servidor no console. O **xhr.responseText** contém o texto da resposta recebida após a requisição ser concluída com sucesso.

3. Enviando a solicitação: ação assíncrona em curso

Após configurar a solicitação e definir como lidar com a resposta, enviamos efetivamente a solicitação ao servidor utilizando o seguinte código:

```
xhr.send();
```

Fique atento!

Embora o XMLHttpRequest seja amplamente utilizado, a introdução de **promessas** e a **Fetch API** proporcionam uma abordagem mais moderna para lidar com solicitações assíncronas. A Fetch API simplifica a sintaxe, enquanto as promessas oferecem um modelo mais claro para o código assíncrono.

```
fetch("https://api.exemplo.com/dados")
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error(error));
```

Explicando o código

O código utiliza o método **fetch** para fazer uma requisição assíncrona a uma API. Vamos entender cada parte?

- **fetch("https://api.exemplo.com/dados"):** aqui, estamos fazendo uma requisição **GET** para a URL "**https://api.exemplo.com/dados**". O método **fetch** retorna uma promessa que representa a resposta da requisição;
- **.then(response => response.json()):** quando a resposta da requisição estiver disponível, usamos o método **.json()** para extrair os dados da resposta. Isso também retorna uma promessa;
- **.then(data => console.log(data)):** assim que os dados estiverem prontos, imprimimos o conteúdo no console. O objeto **data** contém os dados da resposta no formato JSON;
- **.catch(error => console.error(error)):** se ocorrer algum erro durante a requisição ou o processamento dos dados, ele será capturado e impresso no console.

10.3 Manipulando dados JSON

O Ajax costuma utilizar a linguagem XML para trocar dados com o servidor. Porém, frequentemente, o JSON é preferível devido à sua simplicidade e facilidade de uso.

O que é o JSON e como ele funciona para manipulação de dados?

O que é JSON?

Definição	Estrutura básica
• JSON, ou JavaScript *Object Notation*, é um formato leve de troca de dados, fácil para humanos lerem e escreverem e para máquinas interpretarem e gerarem; • É amplamente utilizado na comunicação entre sistemas *web*, especialmente em serviços de API e na transmissão de dados estruturados entre um servidor e um cliente; • Originalmente, ele é derivado da sintaxe de objetos JavaScript. Por isso, é mais simples e flexível.	• Sua estrutura consiste em pares chave-valor, em que as chaves são *strings* e os valores podem ser de vários tipos de dados, incluindo ***strings***, **números**, **booleanos**, ***arrays*** e, até mesmo, outros **objetos** JSON; • Utiliza **chaves {}** para delimitar objetos, **colchetes []** para delimitar *arrays* e **vírgulas** para separar os diferentes elementos. Isso permite a aninhamento de dados e construção de estruturas complexas.

```
{
    "chave1": "valor1",
    "chave2": "valor2",
    "chave3": {
        "subchave1": "subvalor1",
        "subchave2": "subvalor2"
    },
    "chave4": ["item1", "item2", "item3"],
    "chave5": 123,
    "chave6": true,
    "chave7": null
}
```

Vantagens do JSON

- Uma das principais vantagens do JSON é sua facilidade de uso em várias linguagens de programação, não se limitando ao JavaScript;
- As bibliotecas para manipulação de JSON estão disponíveis em praticamente todas as linguagens de programação, facilitando a integração entre sistemas;
- Devido à sua natureza textual e ampla compatibilidade, o JSON é amplamente utilizado em serviços *web* para comunicação entre servidores e clientes, em APIs de várias plataformas, no armazenamento de configurações e em muitas outras aplicações em que a transferência de dados estruturados é necessária.

Explorando a manipulação de dados JSON em detalhes

No contexto do desenvolvimento *web*, a manipulação de dados JSON desempenha um papel crucial, especialmente ao lidar com solicitações HTTP assíncronas em JavaScript. Por isso, agora, vamos explorar todo o processo de manipulação de dados JSON nesse contexto.

1. Entendendo a estrutura JSON

O JSON é uma estrutura composta por pares chave-valor, semelhante à notação de objetos em JavaScript. Os dados podem ser representados como objetos, *arrays*, *strings*, números, booleanos e valores nulos. Compreender essa estrutura é essencial para uma manipulação eficaz.

Vamos compreender melhor todos esses termos?

String	É uma **sequência de caracteres,** sejam letras, frases, números ou símbolos, que representam um texto e são delimitadas por aspas duplas.
Número	Usados para representar **quantidades,** podendo ser número **inteiros** ou número de **ponto flutuante** (decimais).
Objeto	É uma coleção de **pares chave-valor,** em que a chave é sempre uma *string* e o valor pode ser qualquer tipo de dado JSON.
Booleano	É um tipo de dado que pode ter apenas dois valores: **verdadeiro (true)** ou **falso (false)**.
Array	Representa uma **coleção** ou **lista** ordenada de valores, delimitada por **colchetes []**. Os valores podem ser de qualquer tipo JSON, incluindo ***strings***, **números**, **objetos**, **booleanos** ou, até mesmo, **outro *array***.
Null	É escrito sem aspas e representa um **valor nulo,** indicando a **ausência** de valor ou a **inexistência de um válido** para ser representado.

Assim, nós temos, por exemplo:

```
{
  "nome": "John Doe",
  "idade": 30,
  "cidade": "Exemploville",
  "hobbies": ["Leitura", "Esportes"]
}
```

Nesse código, os dados estão sendo declarados em formatos diferentes, ou seja, como ***strings***, **números** e ***arrays***.

2. Analisando dados JSON em JavaScript: *parse* e *stringify*

O JavaScript fornece métodos nativos para converter dados JSON de (e para) objetos JavaScript. Nesse contexto, os métodos *parse* e *stringify* entram em cena.

Parse	Stringify
O método **JSON.parse()** converte uma string JSON em um objeto	O **JSON.stringify()** faz o inverso, convertendo um objeto em uma *string* JSON
Convertendo uma *string* JSON para objeto JavaScript **var dadosObjeto = JSON.parse('{"nome": "John Doe", "idade": 30, "cidade": "Exemploville"}');**	Convertendo um objeto JavaScript para *string* JSON **var dadosJSON = JSON.stringify({ nome: "John Doe", idade: 30, cidade: "Exemploville" });**

3. Integração com solicitações HTTP assíncronas: XMLHttpRequest e Fetch API

Ao realizar solicitações HTTP assíncronas, a manipulação de dados JSON é comum. Ao receber a resposta do servidor, nós utilizamos **JSON.parse()** para converter a *string* JSON em um objeto JavaScript utilizável. Essa é a hora em que o **XMLHttpRequest** e o **Fetch API** podem entrar em ação.

```
// Exemplo com XMLHttpRequest
var xhr = new XMLHttpRequest();
xhr.onreadystatechange = function() {
  if (xhr.readyState == 4 && xhr.status == 200) {
    var respostaJSON = JSON.parse(xhr.responseText);
    console.log(respostaJSON);
  }
};
xhr.open("GET", "exemplo.json", true);
xhr.send();
```

Explicando o código anterior

- Neste exemplo, estamos utilizando o objeto **XMLHttpRequest** para fazer uma requisição assíncrona a uma API;

- O método **.open("GET", "exemplo.json", true)** configura a requisição para buscar dados do arquivo **"exemplo.json"**;
- Quando a resposta da requisição estiver disponível, verificamos se o *status* é 200 (indicando sucesso) e, em seguida, convertemos a resposta em um objeto JSON usando **JSON.parse(xhr.responseText)**;
- Por fim, imprimimos os dados no console com **console.log(respostaJSON)**.

```
// Exemplo com Fetch API
fetch("exemplo.json")
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error(error));
```

Explicando o código anterior

- Neste exemplo, utilizamos a moderna Fetch API para fazer uma requisição **GET** à mesma URL **"exemplo.json"**;
- Em seguida, a resposta é processada encadeando promessas:
    - primeiro, usamos .**then(response => response.json())** para extrair os dados da resposta em formato JSON;
    - em seguida, usamos **.then(data => console.log(data))** para imprimir os dados no console;
    - se ocorrer algum erro durante a requisição ou o processamento dos dados, ele será capturado e impresso no console com **.catch(error => console.error(error))**.

4. Trabalhando com dados JSON: modificação e utilização

Uma vez convertidos para objetos JavaScript, os dados JSON podem ser manipulados conforme for necessário. Dessa forma, **adicionar**, **modificar** ou **excluir** propriedades torna-se um processo simples e eficiente.

```
// Modificando um valor em um objeto JavaScript
dadosObjeto.idade = 31;

// Adicionando uma nova propriedade
dadosObjeto.profissao = "Desenvolvedor";

// Excluindo uma propriedade
delete dadosObjeto.hobbies;
```

Conseguiu entender todos os códigos?

Parabéns! Você está se tornando um verdadeiro detetive digital desvendando todos esses códigos.

DESAFIO PRÁTICO

Compreensão das funções e da importância do Ajax

Descrição

Contexto:

Você é o líder técnico de uma equipe de desenvolvimento encarregada de otimizar a experiência do usuário em um **aplicativo de notícias *on-line***. Atualmente, o aplicativo recarrega a página inteira a cada nova notícia, resultando em uma experiência pouco fluida e bastante demorada.

Cenário:

O objetivo da sua equipe é fazer com que o aplicativo de notícias se destaque pela entrega rápida e eficiente de informações aos usuários. No entanto, o método atual em que a página recarrega para exibir notícias mais recentes não está alinhado com as expectativas modernas de navegação na *web*. Por isso, você pretende introduzir o carregamento dinâmico de notícias através de solicitações assíncronas, proporcionando uma experiência mais suave e interativa. No entanto, a equipe ainda é relutante a algumas tecnologias e a sua função é explicar o Ajax e mostrar as suas vantagens para implementação na empresa.

Objetivos

- Demonstrar a importância e o uso do Ajax.

Orientações

- Crie uma apresentação explicativa sobre seu uso e funcionalidade considerando todo o aprendizado sobre o Ajax;
- Descreva a importância do Ajax em páginas *web*.

DESAFIO PRÁTICO II

Análise das vantagens de uso do Ajax

Descrição

Contexto:

Você é o líder técnico de uma equipe de desenvolvimento encarregada de otimizar a experiência do usuário em um aplicativo de notícias on-line. Atualmente, o aplicativo recarrega a página inteira a cada nova notícia, resultando em uma experiência pouco fluida e bastante demorada.

Cenário:

O objetivo da sua equipe é fazer com que o aplicativo de notícias se destaque pela entrega rápida e eficiente de informações aos usuários. No entanto, o método atual em que a página recarrega para exibir notícias mais recentes não está alinhado com as expectativas modernas de navegação na web. Por isso, você pretende introduzir o carregamento dinâmico de notícias através de solicitações assíncronas, proporcionando uma experiência mais suave e interativa. No entanto, a equipe ainda é relutante a algumas tecnologias e a sua função é explicar o Ajax e mostrar as suas vantagens para implementação na empresa.

Objetivos

- Apresentar modelos já existentes de *sites* que utilizam Ajax;
- Apresentar informações relevantes sobre os exemplos citados, a fim de demonstrar o uso e as vantagens do Ajax.

Orientações

- Acesse sites ou páginas que utilizam a tecnologia Ajax;
- Crie um relatório demonstrativo das vantagens dessa tecnologia com base nas páginas acessadas anteriormente.

DESAFIO PRÁTICO III

Comparação de *sites* que utilizam Ajax e os que não utilizam.

Descrição

Contexto:

Você é o líder técnico de uma equipe de desenvolvimento encarregada de otimizar a experiência do usuário em um **aplicativo de notícias *on-line***. Atualmente, o aplicativo recarrega a página inteira a cada nova notícia, resultando em uma experiência pouco fluida e bastante demorada.

Cenário:

O objetivo da sua equipe é fazer com que o aplicativo de notícias se destaque pela entrega rápida e eficiente de informações aos usuários. No entanto, o método atual em que a página recarrega para exibir notícias mais recentes não está alinhado com as expectativas modernas de navegação na *web*. Por isso, você pretende introduzir o carregamento dinâmico de notícias através de solicitações assíncronas, proporcionando uma experiência mais suave e interativa.

No entanto, a equipe ainda é relutante a algumas tecnologias e a sua função é explicar o Ajax e mostrar as suas vantagens para implementação na empresa.

Objetivos

- Identificar as diferenças entre *sites* que utilizam o Ajax e os que não o possuem.

Orientações

- Pesquise ao menos três *sites* que não possuam as abordagens do Ajax e três que utilizam essa tecnologia;
- Analise os *sites* de acordo com os seus aprendizados;
- Crie um relatório fazendo uma comparação entre os *sites*.

Neste capítulo, nós exploramos como a comunicação assíncrona facilitada pelo Ajax é fundamental para a construção de aplicações modernas e interativas. Com ele, os desenvolvedores podem realizar solicitações para o servidor, receber dados em formato JSON e, em seguida, manipular dinamicamente o conteúdo da página, sem a necessidade de recarregá-la completamente.

Nesse contexto, também trabalhamos com o objeto **XMLHttpRequest** e a **Fetch API** em JavaScript, ferramentas que permitem enviar solicitações HTTP, receber respostas e atualizar o conteúdo da página de forma assíncrona, resultando em interfaces mais eficientes e envolventes.

ATIVIDADE DE FIXAÇÃO

1. Explique o que significa a sigla Ajax e como ela revolucionou a comunicação entre o navegador e o servidor em aplicações *web*.
2. Diferencie a abordagem assíncrona do Ajax em comparação com métodos tradicionais de comunicação entre o navegador e o servidor.
3. Descreva o papel do objeto **XMLHttpRequest** no contexto do Ajax. Como ele é utilizado para realizar solicitações assíncronas?
4. Comente sobre a Fetch API como uma alternativa ao **XMLHttpRequest**. Quais são as vantagens e desvantagens de cada abordagem?
5. Explique como o Ajax utiliza o formato JSON para a troca de dados entre o navegador e o servidor. Dê exemplos práticos de como esses dados podem ser manipulados no JavaScript.
6. Detalhe os passos necessários para realizar uma solicitação HTTP assíncrona usando o objeto **XMLHttpRequest**.
7. Como o Ajax contribui para a eficiência no desenvolvimento *web* ao permitir a atualização dinâmica de conteúdo? Dê exemplos de situações práticas em que essa eficiência é evidente.
8. Descreva a importância da validação de dados JSON ao manipular informações recebidas do servidor. Por que é crucial garantir a integridade desses dados?
9. Explique como a comunicação assíncrona facilitada pelo Ajax impacta positivamente a experiência do usuário. Quais são os benefícios específicos que os usuários podem esperar?
10. Diante das evoluções tecnológicas, discuta se o Ajax ainda é uma tecnologia relevante para o desenvolvimento *web* ou se existem alternativas mais modernas e eficazes.

Capítulo 11

INOVAÇÃO DINÂMICA NA *WEB*: ATUALIZAÇÃO E INTERAÇÃO COM SERVIDORES

O que esperar deste capítulo:

- Compreender as vantagens e o impacto da atualização dinâmica na experiência do usuário;
- Conhecer as tecnologias essenciais, incluindo Ajax, WebSocket e Fetch API;
- Implementar atualizações dinâmicas usando JavaScript.

11.1 Atualização dinâmica de conteúdo

A atualização dinâmica de conteúdo é uma técnica fundamental no desenvolvimento *web*, pois permite a modificação de partes específicas de uma página sem a necessidade de que ela seja recarregada por completo. Isso torna a navegação muito mais otimizada para o usuário.

Este processo é conduzido por meio da manipulação do DOM (Modelo de Objeto de Documento) com JavaScript, permitindo que os desenvolvedores alterem elementos HTML, estilos e até adicionem ou removam elementos da página. Essas modificações podem ser baseadas em interações do usuário, eventos ou dados recebidos de uma API (Interface de Programação de Aplicações).

A essência da atualização dinâmica reside na sua capacidade de proporcionar uma experiência contínua e ágil ao usuário. Ao contrário da abordagem tradicional, em que a página inteira é recarregada, a atualização dinâmica utiliza **tecnologias assíncronas**, como o Ajax, para realizar a comunicação entre o navegador e o servidor. Isso resulta em uma interação mais rápida e eficiente, pois apenas partes específicas da página são atualizadas conforme necessário.

Tecnologias-chave para aplicar a atualização dinâmica

Conhecer e ter o domínio das tecnologias usadas para aplicar a atualização dinâmica é muito importante, especialmente no contexto do desenvolvimento de *software* e aplicativos

web. Assim como já vimos, as atualizações dinâmicas permitem que os aplicativos sejam atualizados de forma assíncrona, ou seja, os processos não ocorrem, ou não são executados, ao mesmo tempo, evitando a necessidade de intervenção manual do usuário ou *downloads* extensivos. Isso não apenas melhora a experiência do usuário, mas também facilita a gestão e a distribuição de novas funcionalidades, correções de segurança e melhorias de desempenho.

Para aplicar a atualização dinâmica em projetos de desenvolvimento *web*, as tecnologias mais utilizadas são o AJAX, o Fetch API e o WebSocket. O **AJAX** permite a realização de **solicitações assíncronas** para buscar dados do servidor sem interromper a experiência do usuário. Já o **Fetch API**, uma evolução do AJAX, simplifica ainda mais esse processo, oferecendo uma **interface mais moderna e amigável**. O **WebSocket**, por sua vez, se destaca ao proporcionar uma **comunicação bidirecional**, ideal para atualizações em tempo real em aplicações que exigem respostas instantâneas. Conheça mais características dessas três tecnologias no quadro a seguir e saiba como aplicá-las ao implementar a atualização dinâmica nos seus projetos.

AJAX

- Normalmente é utilizado em conjunto com JavaScript para criar interfaces mais interativas e responsivas;
- O principal objetivo do AJAX é realizar requisições assíncronas ao servidor, buscando ou enviando dados sem a necessidade de recarregar a página;
- Utiliza o objeto **XMLHttpRequest** ou **fetch** para criar requisições HTTP assíncronas.

Fetch API

- Interface moderna para fazer requisições HTTP assíncronas em JavaScript;
- Simplifica o processo de realizar solicitações de rede e, normalmente, é utilizada para atualizações dinâmicas em páginas da *web*;
- Realiza requisições HTTP assíncronas para buscar ou enviar dados sem recarregar toda a página.

WebSocket

- Protocolo de comunicação bidirecional que fornece uma conexão persistente entre o cliente e o servidor, permitindo a troca de mensagens em tempo real;
- É frequentemente utilizado para atualizações dinâmicas em tempo real em aplicações *web*;
- Solução eficiente e escalável para atualizações dinâmicas em tempo real, sendo especialmente útil em aplicações que exigem interação instantânea entre o cliente e o servidor.

Implementação prática da atualização dinâmica com JavaScript

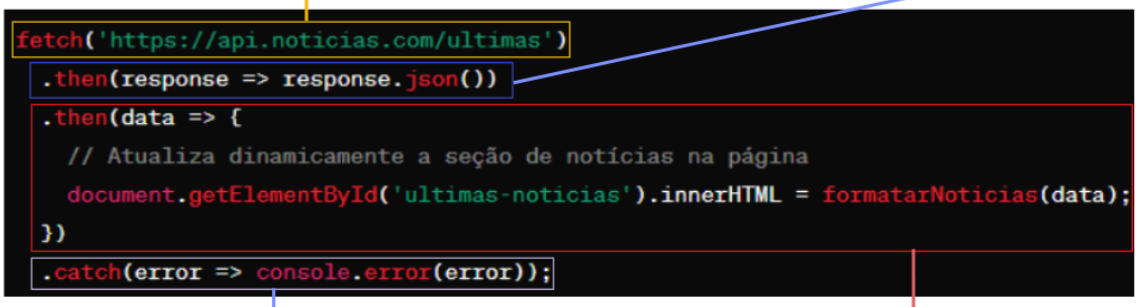
A implementação da atualização dinâmica envolve o uso de JavaScript para criar uma experiência de usuário dinâmica e responsiva. Vamos considerar um exemplo específico: a atualização dinâmica de um *feed* de notícias em uma página. Utilizando a **Fetch API**, podemos realizar uma solicitação assíncrona para buscar as últimas notícias do servidor e, em seguida, integrar esses dados na página de forma dinâmica, sem a necessidade de recarregar. O código abaixo ilustra esse processo:

```
fetch('https://api.noticias.com/ultimas')
  .then(response => response.json())
  .then(data => {
    // Atualiza dinamicamente a seção de notícias na página
    document.getElementById('ultimas-noticias').innerHTML = formatarNoticias(data);
  })
  .catch(error => console.error(error));
```

Esse código nos mostra como buscar e exibir dados de uma API usando JavaScript. Vamos entender melhor analisando cada linha:

Esta linha faz uma requisição para a URL **'https://api.noticias.com/ultimas'** para obter as últimas notícias.

Uma vez que a resposta é recebida, ela é convertida em JSON (*JavaScript Object Notation*), formato de intercâmbio de dados leve e independente de linguagem de programação.

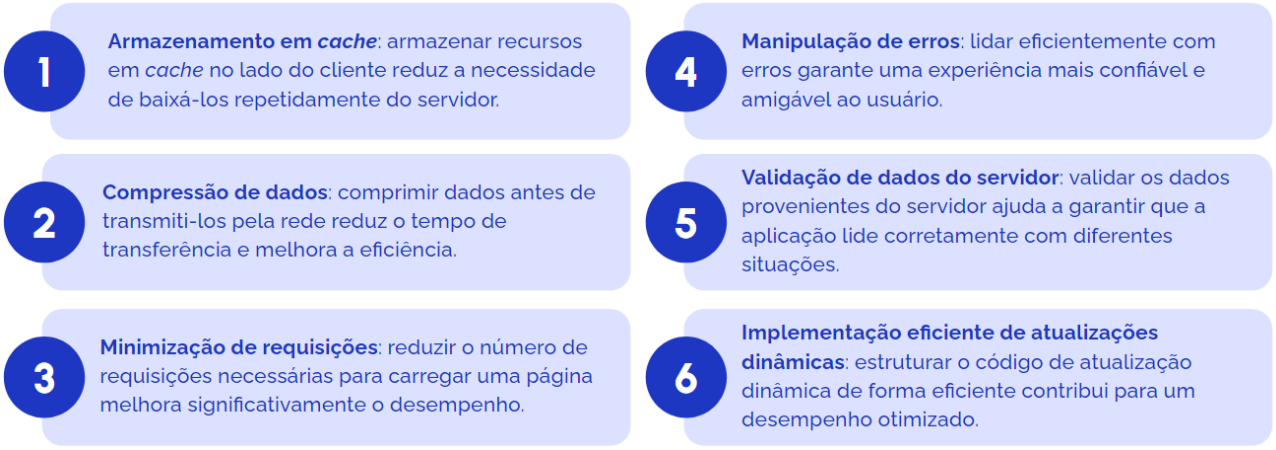


O método **.catch()** é usado para capturar e tratar qualquer erro que possa ocorrer durante a solicitação ou no processo de manipulação da resposta.

Em seguida, os dados JSON são usados para atualizar dinamicamente o conteúdo HTML da seção de notícias na página *web*, especificamente o elemento com o ID **'ultimas-noticias'**.

Otimização e boas práticas

Para garantir uma experiência eficiente, é crucial otimizar a atualização dinâmica. Estratégias como o **armazenamento em cache**, a **compressão de dados** e a **minimização de requisições** são fundamentais. Além disso, é essencial implementar práticas robustas de **manipulação de erros** e **validação de dados** provenientes do servidor, assegurando uma experiência confiável para o usuário.



Ao combinar essas práticas, os desenvolvedores podem criar aplicações *web* mais eficientes, proporcionando aos usuários uma experiência rápida, confiável e agradável. Essas otimizações são especialmente cruciais em ambientes nos quais a interação dinâmica é essencial, como em aplicações *web* modernas.

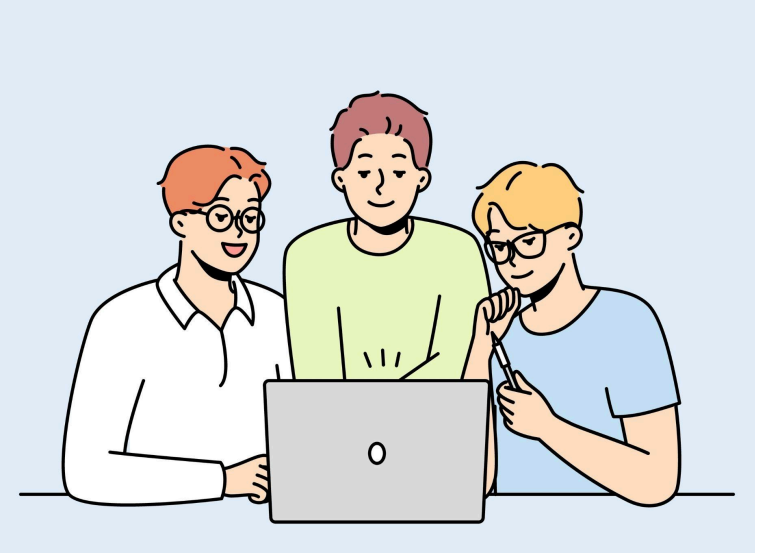
11.2 Criando uma aplicação que busca e exibe dados de um servidor

A criação de uma aplicação capaz de buscar e exibir dados de um servidor é uma habilidade essencial para os desenvolvedores *web* modernos. Neste guia passo a passo, você irá desenvolver uma aplicação que busca e exibe uma lista de usuários, colocando em prática desde a configuração do ambiente de uma aplicação até a interação com o servidor. Isso fará com que você tenha uma compreensão abrangente desse processo, que é crucial na sua formação como desenvolvedor.

Passo 1 - Configuração do ambiente

Antes de começar, certifique-se de ter um ambiente de desenvolvimento configurado. Isso inclui um editor de código, um navegador *web* e um servidor para hospedar a sua

aplicação. Ferramentas populares como o Visual Studio Code, o Google Chrome e o Node.js facilitam esse processo.



Disponível em: https://www.shutterstock.com/pt/image-vector/smiling-male-colleagues-look-computer-screen-2220852125. Acesso em: 12 mar. 2024.

Passo 2 - Estrutura inicial da aplicação

Inicie criando a estrutura básica da sua aplicação, o que envolve a criação dos arquivos HTML, CSS e JavaScript necessários. Defina a estrutura do HTML para a página e adicione estilos básicos com CSS para melhorar a experiência visual. Reproduza o código a seguir no seu ambiente de desenvolvimento e veja o que acontece.

```
<!DOCTYPE html> <!-- Define o tipo de documento como HTML -->

<html lang="pt"> <!-- Define a linguagem do documento como Português -->



<head> <!-- Início da seção de cabeçalho do documento -->

  <meta charset="UTF-8"> <!-- Define a codificação de caracteres do documento como UTF-8 -->

   <meta name="viewport" content="width=device-width, initial-scale=1.0"> <!-- Define a viewport para tornar o site responsivo -->

  <title>Minha Aplicação</title> <!-- Define o título do documento que aparece na aba do navegador -->
```

```
  <link rel="stylesheet" href="styles.css"> <!-- Vincula a folha de estilos CSS externa ao documento HTML -->
</head> <!-- Fim da seção de cabeçalho do documento -->

<body> <!-- Início do corpo do documento -->
  <div id="app"> <!-- Cria uma div com o id "app" -->
    <!-- O conteúdo da aplicação será exibido aqui -->
  </div> <!-- Fim da div com o id "app" -->
  <script src="app.js"></script> <!-- Vincula o arquivo JavaScript externo ao documento HTML -->
</body> <!-- Fim do corpo do documento -->

</html> <!-- Fim do documento HTML -->
```

Passo 3 - Implementação da lógica JavaScript

Desenvolva a lógica JavaScript para interagir com o servidor. Utilize a Fetch API para fazer solicitações HTTP assíncronas. Vamos considerar um exemplo em que a aplicação que você está desenvolvendo busca e exibe uma lista de usuários a partir de um servidor, assim como no código a seguir:

```
# Importa a biblioteca requests para fazer requisições HTTP
import requests

# Define a URL da API que será buscada
url = "https://api.exemplo.com/usuarios"
```

```
try:
    # Realiza uma requisição GET para obter dados da API
    response = requests.get(url)

    # Verifica se a resposta foi bem-sucedida (código de status 200)
    if response.status_code == 200:
        # Converte a resposta em JSON
        data = response.json()

        # Itera sobre cada usuário obtido dos dados da API
        for usuario in data:
            # Obtém o nome do usuário
            nome_usuario = usuario.get("nome")

            # Exibe o nome do usuário
            print(f"Nome do usuário: {nome_usuario}")
    else:
        print(f"Erro na requisição. Código de status: {response.status_code}")
except requests.RequestException as e:
    print(f"Erro na requisição: {e}")
```

```
# Comentários:

# - O código faz uma requisição GET para a URL da API.

# - Se a resposta for bem-sucedida (código 200), converte os dados em JSON.

# - Itera sobre cada usuário e exibe o nome.

# - Trata erros de requisição usando try-except.
```

Passo 4 - Testando e iterando

Nesse momento, você deve testar a sua aplicação localmente para garantir que a interação com o servidor esteja funcionando corretamente. Utilize ferramentas de desenvolvedor do navegador que você escolheu para depurar e aprimorar a implementação conforme for necessário.

Para melhorar a robustez do seu código, você pode adicionar tratamento de erros, que ajudará a lidar com situações inesperadas, como falhas na conexão com a API ou respostas inválidas. Você pode usar blocos **try** e **except** para capturar exceções e lidar com elas de maneira adequada. Veja o exemplo do código a seguir:

```
        try:

            response = requests.get(url)

            if response.status_code == 200:

                data = response.json()

                # Processar os dados aqui

            else:
        print(f"Erro na requisição. Código de status:    {response.status_code}")

         except requests.RequestException as e:

            print(f"Erro na requisição: {e}")
```

Disponível em: https://encurtador.com.br/cuyC7. Acesso em: 12 mar. 2024.

Passo 5 - *Deploy* e manutenção

Após testar e iterar localmente, considere a possibilidade de implantar a sua aplicação em um servidor para torná-la acessível ao público. Além disso, lembre-se que você deve estar preparado para realizar a manutenção contínua da sua aplicação, corrigindo *bugs* e incorporando *feedbacks* dos usuários para aprimorar a experiência deles.

Implantação em um servidor

Imagine que você criou um bolo delicioso e quer compartilhá-lo com outras pessoas. Para fazer isso, você precisa colocar o bolo em um prato e deixá-lo acessível para que todos possam pegar um pedaço.

Da mesma forma, ao implantar a sua aplicação em um servidor, você está colocando o seu código (ou seja, o "bolo") em um lugar no qual outras pessoas podem acessá-lo pela internet.

Para isso, escolha um serviço de hospedagem (como alugar um espaço na cozinha de um restaurante) e coloque sua aplicação lá. Nessa situação, o servidor é como o prato no qual o bolo fica disponível para todos.

Manutenção contínua

Imagine que o seu bolo está sendo servido em uma festa. À medida que as pessoas comem, você precisa ficar de olho para garantir que haja sempre bolo disponível. Da mesma forma, após implantar sua aplicação, é preciso cuidar dela. Para isso, siga os seguintes passos:

- verifique se a aplicação está funcionando corretamente;

- corrija possíveis problemas que possam surgir (como se alguém reclamasse que o bolo está muito doce ou queimado);
- mantenha cópias de segurança (como tirar fotos do bolo antes de cortá-lo);
- esteja pronto para fazer ajustes e melhorias com base no *feedback* dos usuários.

Disponível em: https://encurtador.com.br/ko357. Acesso em: 12 mar. 2024.

***Level up*!**

Depois de seguir todos os passos desse exercício, você alcançou um nível acima na sua jornada *dev*! Continue praticando as suas habilidades de desenvolvimento e siga aprofundando os seus conhecimentos sobre atualização dinâmica em JavaScript.

DESAFIO PRÁTICO

Atualização dinâmica de conteúdo da página

Descrição

Contexto:

Você faz parte de uma equipe de desenvolvimento encarregada de melhorar a experiência do usuário em um aplicativo de gerenciamento de tarefas *on-line*. A demanda principal é criar uma funcionalidade que permita a atualização dinâmica do conteúdo da página, sem a necessidade de recarregar a página inteira. Ou seja, os usuários devem poder adicionar, editar e excluir tarefas de maneira rápida e eficiente, sem interrupções na navegação.

Cenário:

Atualmente, o aplicativo de gerenciamento de tarefas requer que os usuários recarreguem a página para verem as alterações feitas por eles mesmos ou por outros membros da equipe. Isso cria uma experiência descontínua e impacta a eficiência do trabalho colaborativo. A equipe busca uma solução que permita a atualização em tempo real do conteúdo da página.

Objetivos

- Desenvolver um sistema que permita a atualização automática e em tempo real do conteúdo da página, refletindo as mudanças feitas por qualquer usuário imediatamente;
- Implementar funcionalidades para adicionar, editar e excluir tarefas de forma dinâmica, sem a necessidade de recarregar toda a página;
- Integrar notificações instantâneas para informar os usuários sobre alterações recentes, garantindo que eles estejam sempre atualizados sobre as atualizações na lista de tarefas;
- Permitir uma experiência de sincronização colaborativa, em que várias pessoas possam trabalhar simultaneamente na mesma lista de tarefas sem conflitos;

- Implementar um registro de histórico que rastreie e exiba as atualizações recentes, proporcionando transparência sobre quem fez quais alterações.

Orientações

- Utilize tecnologias como o WebSocket ou *Server-Sent Events* (SSE) para estabelecer uma comunicação em tempo real entre o servidor e os clientes;
- Desenvolva métodos para adicionar, editar e excluir tarefas de forma dinâmica, atualizando a interface do usuário sem a necessidade de recarregar toda a página;
- Integre um sistema de notificações em tempo real para informar os usuários sobre mudanças nas tarefas, garantindo que todos estejam cientes das atualizações;
- Implemente técnicas para lidar com a colaboração simultânea, evitando conflitos e garantindo a coesão dos dados compartilhados entre os usuários;
- Crie um mecanismo para registrar e exibir um histórico claro de todas as atualizações recentes, permitindo uma visão completa das mudanças realizadas.

DESAFIO PRÁTICO II

Desenvolvimento de aplicação de busca de dados em servidor

Descrição

Contexto:

Você faz parte de uma equipe de desenvolvimento que está encarregada de criar uma aplicação *web* altamente eficiente que busca e exibe dados de um servidor remoto. A aplicação é destinada a fornecer informações detalhadas sobre produtos em um grande catálogo *on-line*. A principal exigência é criar uma solução que otimize a velocidade de carregamento, a usabilidade e a apresentação dos dados.

Cenário:

A empresa possui um vasto catálogo de produtos e, frequentemente, os clientes desejam procurar e visualizar detalhes específicos sobre determinados produtos. O objetivo é criar uma aplicação que ofereça uma experiência de usuário fluida, apresentando os resultados da pesquisa de maneira rápida e eficiente.

Objetivos

- Implementar uma busca em tempo real que forneça resultados instantâneos à medida que os usuários digitam sua consulta;
- Projetar uma interface de usuário intuitiva e responsiva que seja fácil de navegar, proporcionando uma experiência agradável ao usuário;
- Utilizar técnicas de carregamento dinâmico para apresentar os detalhes do produto, garantindo que apenas as informações necessárias sejam recuperadas do servidor;
- Aplicar técnicas avançadas de otimização de imagens para garantir que as fotos dos produtos sejam carregadas rapidamente, sem comprometer a qualidade visual;
- Implementar um sistema de *cache* eficiente para armazenar localmente os dados que são frequentemente acessados, reduzindo a dependência do servidor para operações repetitivas.

Orientações

- Utilize tecnologias como AJAX para implementar uma busca em tempo real que minimize o tempo de resposta e forneça sugestões instantâneas;
- Adote princípios de *design* responsivo e intuitivo, garantindo uma experiência de usuário agradável em dispositivos variados;
- Implemente um sistema de carregamento dinâmico que busque apenas os dados essenciais relacionados aos produtos em foco;
- Utilize técnicas como compressão de imagem, carregamento progressivo e *lazy loading* para otimizar o desempenho das imagens;
- Desenvolva um mecanismo de *cache* inteligente que armazene localmente dados frequentemente acessados, reduzindo a carga no servidor e melhorando a velocidade de resposta.

RESUMO

Neste capítulo, vimos que a atualização dinâmica de conteúdo é um conceito central para criar páginas *web* interativas e responsivas. Dessa forma, aprendemos que, ao utilizar tecnologias assíncronas como Ajax, Fetch API e WebSocket, os desenvolvedores podem modificar partes específicas da página sem a necessidade de recarregar toda a estrutura. Estudamos que, ao compreender os fundamentos, utilizar as ferramentas certas e implementar boas práticas, é possível criar aplicações que oferecem uma experiência de usuário mais ágil e contínua. Sendo assim, essa abordagem não apenas mantém os usuários envolvidos, mas também abre portas para atualizações em tempo real e interatividade dinâmica, definindo o padrão para aplicações *web* modernas.

Para consolidar os conhecimentos sobre os assuntos estudados no decorrer do capítulo, seguimos um passo a passo de como desenvolver uma aplicação que busca e exibe dados de um servidor, uma habilidade essencial no universo da programação *web* moderna. Abordamos que esse processo envolve configurar o ambiente de desenvolvimento, criar uma estrutura inicial da aplicação com HTML, CSS e JavaScript e implementar a lógica para interagir com o servidor utilizando a Fetch API. Essa habilidade é valiosa para desenvolvedores que buscam oferecer experiências de usuário superiores e que querem se manter atualizados com as tendências e demandas do desenvolvimento *web* moderno.

ATIVIDADE DE FIXAÇÃO

1. Explique a importância da Fetch API ao criar uma aplicação que busca dados de um servidor. Como ela facilita o processo de comunicação assíncrona?

2. Descreva os passos essenciais para configurar o ambiente de desenvolvimento ao criar uma aplicação que busca dados de um servidor. Quais ferramentas são, geralmente, recomendadas para esse propósito?

3. Na implementação JavaScript apresentada para exibir uma lista de usuários, explique o papel da função **exibirUsuarios** e como ela contribui para a dinâmica da página.

4. Como a atualização dinâmica de conteúdo difere da abordagem tradicional de recarregar páginas inteiras? Destaque as vantagens proporcionadas por essa técnica.

5. Por que é crucial testar localmente antes de implantar em um ambiente de produção ao criar uma aplicação que busca dados de um servidor? Quais ferramentas podem ser úteis nesse processo?

6. Qual é a função do evento **'DOMContentLoaded'** no contexto da implementação JavaScript para buscar e exibir dados de um servidor? Explique o motivo desse evento ser utilizado.

7. Descreva como você incorporaria a manipulação de erros na aplicação, garantindo uma experiência robusta para o usuário, especialmente ao lidar com falhas na comunicação com o servidor.

8. Ao considerar a atualização dinâmica de conteúdo, explique como estratégias como o armazenamento em *cache* e a minimização de requisições podem otimizar a experiência do usuário.

9. Como a implementação prática de uma aplicação poderia ser aprimorada para lidar com dados mais complexos do servidor, como imagens ou informações adicionais sobre os usuários?

10. Em relação à atualização dinâmica de conteúdo, explique como a tecnologia WebSocket difere da Fetch API em termos de comunicação com o servidor. Quando seria apropriado utilizar cada uma dessas abordagens?

Capítulo 12

DESENVOLVIMENTO *FRONT-END* AVANÇADO: *FRAMEWORKS*, BIBLIOTECAS E ESCOLHA DE TECNOLOGIAS

O que esperar deste capítulo:

- Compreender e aplicar os conceitos fundamentais dos *frameworks* e das bibliotecas populares de *front-end*, como React, Angular e Vue.js;
- Entender como e quando usar bibliotecas em projetos, maximizando a eficiência e a manutenibilidade do código;
- Desenvolver competências para atualizar e modernizar projetos existentes com novas tecnologias, melhorando a performance, a escalabilidade e a manutenção.

12.1 Aplicação e estudo de bibliotecas e *frameworks*

A utilização de bibliotecas e *frameworks* desempenha um papel crucial no desenvolvimento em JavaScript, pois eles contribuem para a eficiência, produtividade e manutenibilidade dos projetos. Cada uma dessas ferramentas ajuda a otimizar o processo de desenvolvimento, promovendo a consistência e a reutilização do código.

Neste capítulo, vamos abordar os *frameworks* e as bibliotecas relacionados ao *front-end*, como **React**, **Angular**, **Vue.js** e **jQuery**, que são projetados para facilitar o desenvolvimento da interface do usuário (ou UI, que significa *User Interface*) de uma aplicação *web*. Mas, antes de iniciarmos, vamos entender melhor a diferença entre esses dois conceitos.

Bibliotecas

Uma biblioteca é um **conjunto de funções ou rotinas reutilizáveis** que podem ser chamadas por um programa para realizar **tarefas específicas**, como a manipulação de *strings*, operações matemáticas etc., e o desenvolvedor decide quando e como chamar essas funções. As bibliotecas mais populares são:

jQuery	Biblioteca JavaScript rápida e rica em recursos para simplificar a manipulação do DOM e interações com o servidor.
React	Biblioteca JavaScript de código aberto utilizada para construir interfaces de usuário interativas e dinâmicas.

Para que você entenda melhor como uma biblioteca funciona, imagine que uma pessoa quer preparar uma macarronada. Ela tem duas opções: preparar a receita do zero – ou seja, fazer seu próprio molho de tomate, a massa do macarrão e, até mesmo, cultivar os seus próprios vegetais – ou ela pode ir ao mercado e escolher os ingredientes que achar melhor diante de todas as opções que estão nas prateleiras. Depois, é só cozinhar e colocar a comida no prato. A segunda opção certamente será mais prática e rápida, não é mesmo?

É assim que uma biblioteca JavaScript funciona. Ela já dispõe de um conjunto de "ingredientes" que você pode escolher e usar para construir o seu projeto de *software* ou aplicação *web*, sem precisar escrever o código do zero. Você pode simplesmente usar uma biblioteca que já tenha as funcionalidades que deseja que sejam implementadas, tornando o processo de desenvolvimento bem mais rápido e eficiente.

Disponível em: https://encurtador.com.br/ikm29. Acesso em: 13 mar. 2024.

Frameworks

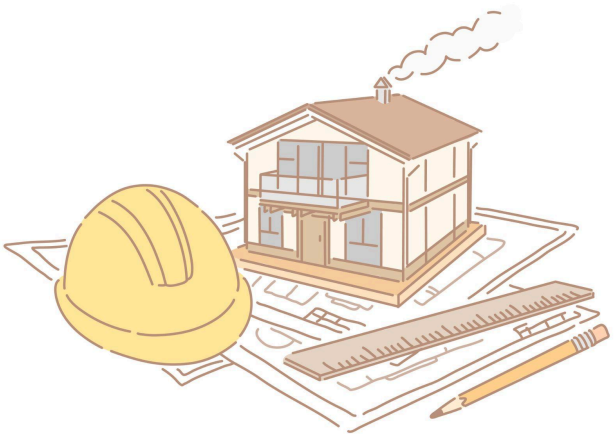
Assim como as bibliotecas, os *frameworks* também ajudam a otimizar o desenvolvimento de projetos de desenvolvimento *web*. No entanto, essa é uma estrutura mais abrangente,

fornecendo uma arquitetura predefinida e um conjunto de convenções que os desenvolvedores podem seguir para acelerar o processo de desenvolvimento, mantendo a qualidade, a eficiência e a escalabilidade. Aqui estão dois exemplos de *frameworks* populares:

Angular	*Framework* utilizado para o desenvolvimento de aplicações *web front-end* robustas e escaláveis.
Vue.js	*Framework* utilizado para construir interfaces de usuário interativas. É uma opção leve e fácil de integrar em projetos existentes.

Para que esse conceito fique mais claro, imagine que, para construir uma casa, você pudesse usar um *framework* JavaScript. Esse *framework* já oferece um plano de *design* predefinido, com todos os materiais e componentes que você precisa para a construção da sua casa. Com ele, você ainda pode personalizar a casa, escolhendo as cores das paredes e alterando a disposição dos cômodos, mas toda a estrutura básica já está predeterminada pelo *framework*, garantindo que a sua construção será bem-sucedida.

Usar *frameworks* pode ser uma ótima escolha, pois eles oferecem um caminho estruturado, permitindo o desenvolvimento de aplicações complexas com menos esforço e maior confiança na estabilidade e na manutenção dos projetos.



Disponível em: https://encurtador.com.br/firLQ Acesso em: 13 mar. 2024.

Se liga!

Biblioteca

Conjunto de funções ou rotinas reutilizáveis que podem ser chamadas por um programa para realizar **tarefas específicas em um projeto**.

VS

Framework

Estrutura mais abrangente que **fornece uma arquitetura** para o desenvolvimento de *software* de um **projeto completo**. Ele define a estrutura da aplicação e controla o fluxo do programa.

12.2 Uso de bibliotecas para simplificar tarefas comuns

No processo de desenvolvimento de *software*, a utilização de bibliotecas tem se tornado uma prática comum e valiosa. A principal vantagem do uso das bibliotecas é a **economia de tempo e esforço**. Ao utilizar uma biblioteca bem estabelecida, os desenvolvedores podem confiar que as funcionalidades fornecidas são robustas e confiáveis, uma vez que elas são desenvolvidas e testadas pela comunidade de desenvolvedores. Além disso, muitas bibliotecas seguem **padrões e melhores práticas de programação**, o que facilita a manutenção de um código limpo, organizado e de fácil compreensão.

Além disso, as bibliotecas são projetadas para serem **facilmente integradas a outros sistemas ou *frameworks***. Isso significa que os desenvolvedores podem aproveitar as funcionalidades da biblioteca em suas próprias aplicações, independentemente do ambiente ou da tecnologia que estão usando.

Agora, vamos nos aprofundar em duas das principais bibliotecas utilizadas em JavaScript: React e jQuery.

React

Disponível em: https://encurtador.com.br/dmCW8. Acesso em: 14 mar. 2024.

Desenvolvido pelo Facebook, o React é uma biblioteca mantida pela comunidade de *software* livre e, geralmente, é utilizada em conjunto com o JavaScript e o HTML para criar componentes reutilizáveis e construir interfaces de usuário (UI) interativas.

Uma das características do React é o seu **modelo de programação declarativo**, em que o desenvolvedor especifica as características desejadas do resultado final, sem se preocupar com os passos exatos para atingir esse objetivo. Isso torna mais fácil o desenvolvimento e a manutenção de aplicações *web*.

Para que você entenda melhor como utilizar essa biblioteca, nada melhor do que colocar a mão na massa e criar um projeto utilizando o React.

Mão na massa!

Siga o passo a passo a seguir e **crie uma aplicação de lista de tarefas** que inclui a criação de componentes, manipulação de estado e interação com o usuário.

Passo 1 - configuração do projeto

Antes de começar, certifique-se de ter o **Node.js** instalado. Para isso, acompanhe o tutorial do vídeo e execute os comandos no seu terminal:

Fonte: https://www.youtube.com/watch?v=-cLzUDoTQY0. Acesso em: 14 mar. 2024.

Passo 2 - criação de componentes

Depois de fazer as configurações iniciais, você precisa criar três componentes: **App**, **TodoList**, e **TodoItem**. Acompanhe os códigos a seguir:

App.js

```
// Importando React e useState do React.
import React, { useState } from 'react';
// Importando o componente TodoList do arquivo TodoList.js.
import TodoList from './TodoList';


// Definindo o componente funcional App.
function App() {
 // Utilizando o hook de estado (useState) para criar um estado 'tasks' inicializado como um array vazio,
 // e 'setTasks' como uma função para atualizar esse estado.
 const [tasks, setTasks] = useState([]);


 // Criando a função 'addTask', que recebe uma nova tarefa como parâmetro e a adiciona ao estado 'tasks'.
 const addTask = (task) => setTasks([...tasks, task]);


 // Retornando à estrutura JSX do componente App.
 return (
  <div>
   {/* Um título simples na página. */}
```

```
    <h1>Lista de Tarefas</h1>
    {/* Renderizando o componente TodoList e passando as propriedades 'tasks' e 'addTask' para ele. */}
    <TodoList tasks={tasks} addTask={addTask} />
  </div>
 );
}

// Exportando o componente App para ser utilizado em outros arquivos.
export default App;
```

ToDoList.js

```
// Importando React e useState do React.
import React, { useState } from 'react';
// Importando o componente TodoList do arquivo TodoList.js.
import TodoList from './TodoList';

// Definindo o componente funcional App.
function App() {
 // Utilizando o hook de estado (useState) para criar um estado 'tasks' inicializado como um array vazio,
 // e 'setTasks' como uma função para atualizar esse estado.
 const [tasks, setTasks] = useState([]);

 // Criando a função 'addTask', que recebe uma nova tarefa como parâmetro e a adiciona ao estado 'tasks'.
 const addTask = (task) => setTasks([...tasks, task]);

 // Retornando a estrutura JSX do componente App.
 return (
  <div>
   {/* Um título simples na página. */}
```

```
    <h1>Lista de Tarefas</h1>
    {/* Renderizando o componente TodoList e passando as propriedades 'tasks' e 'addTask' para ele. */}
    <TodoList tasks={tasks} addTask={addTask} />
  </div>
 );
}
// Exportando o componente App para ser utilizado em outros arquivos.
export default App;
```

TodoItem.js

```
// Importando React e useState do pacote 'react'.
import React, { useState } from 'react';


// Definindo o componente funcional TodoItem, que recebe a propriedade 'task'.
function TodoItem({ task }) {
 // Criando um estado local 'completed' e uma função 'setCompleted' para modificar o estado.
 const [completed, setCompleted] = useState(false);


 Retornando a estrutura JSX do componente TodoItem.
 return (
  <li style={{ textDecoration: completed ? 'line-through' : 'none' }}>
   {/* Caixa de seleção (checkbox) para marcar a tarefa como concluída ou não. */}
   <input
    type="checkbox"
    checked={completed}
    // Quando o estado da caixa de seleção muda, chama a função para atualizar 'completed'.
    onChange={() => setCompleted(!completed)}
   />
```

```
    {/* Exibindo o nome da tarefa, aplicando estilo de texto riscado se a tarefa estiver concluída. */}
    {task}
  </li>
 );
}


// Exportando o componente TodoItem para ser utilizado em outros arquivos.
export default TodoItem;
```

Passo 3 - personalize a sua aplicação

Agora que a sua aplicação React de lista de tarefas está rodando, é hora de fazer dela algo único e com a sua cara! Aqui estão algumas ideias para personalizar e deixar a sua lista de tarefas com o estilo que você quiser:

Características	Descrição
Dê vida às tarefas	Acrescente *emojis* ou ícones às tarefas para expressar sentimentos ou temas. Faça com que tarefas importantes tenham destaque especial. Para isso, você pode acessar o *site* Flaticon (https://www.flaticon.com/br/), escolher um ícone ou **emoji** e copiar e colá-lo onde você quiser na sua lista de tarefas.
Temas e cores vibrantes	Usando o CSS, você pode deixar tudo com a sua cara. Explore os diferentes temas e cores para tornar a aplicação mais divertida.
Organização inteligente	Experimente formas inovadoras de organizar as tarefas, como por cores ou *tags*. Adicione uma função de pesquisa para facilitar encontrar tarefas específicas.
Desafios e conquistas	Implemente um sistema de recompensas ou conquistas para motivar a conclusão de tarefas. Crie desafios diários ou semanais para tornar a lista mais dinâmica. Veja um exemplo:

```
function pesquisarTarefas(palavraChave) {
  // Filtra as tarefas com base na palavra-chave
  const tarefasFiltradas = tarefas.filter(tarefa =>
    tarefa.texto.includes(palavraChave)
  );
  // Exibe os resultados relevantes
  renderizarTarefas(tarefasFiltradas);
 }
```

Interação divertida

Adicione animações e transições para tornar a interação mais divertida. Faça as tarefas "pularem" quando marcadas como concluídas. Veja como você pode fazer isso:

```
  @keyframes beat {
  0% { transform: scale(1); }
  50% { transform: scale(1.3); }
  100% { transform: scale(1); }
}


#heart {
  animation: beat 1s infinite;
  cursor: pointer;
}
```

Personalização do usuário

Permita que os usuários personalizem a aparência da lista conforme as suas preferências. Adicione avatares ou imagens aos perfis dos usuários. Veja um exemplo:

```
<script>
function criaImagem() {
  const div = document.getElementById("imagens");
  const image = document.createElement("img");
```

```
    image.src = prompt("Digite o nome ou link da imagem:"); // Solicita o caminho da imagem
    div.appendChild(image); // Adiciona a imagem à div
  }
</script>
```

Lembre-se da persistência

Faça as tarefas persistirem para que os usuários nunca percam o seu progresso. Use o armazenamento local para salvar as tarefas, mesmo se fecharem o navegador. Veja como isso pode ser feito:

```
<script>
// Função para adicionar uma nova tarefa à lista
function adicionarTarefa() {
  // Captura o valor da nova tarefa inserida pelo usuário
  const novaTarefa = document.getElementById("nova-tarefa").value;
  // Verifica se a nova tarefa não está vazia
  if (novaTarefa) {
    // Obtém a referência à lista de tarefas no HTML
    const lista = document.getElementById("lista-tarefas");
    // Cria um novo elemento de lista (li) para representar a nova tarefa
    const li = document.createElement("li");
    // Define o texto do novo elemento li como a nova tarefa
    li.textContent = novaTarefa;
    // Adiciona o novo elemento li à lista de tarefas
    lista.appendChild(li);

    // Salva a nova tarefa no localStorage
    localStorage.setItem("tarefa_" + Date.now(), novaTarefa);

    // Limpa o campo de entrada para que o usuário possa inserir uma nova tarefa
```

```
    document.getElementById("nova-tarefa").value = "";
  }
}


// Função para carregar as tarefas do localStorage ao carregar a página
window.onload = function () {
  // Loop através de todas as chaves no localStorage
  for (let i = 0; i < localStorage.length; i++) {
    // Obtém a chave atual
    const chave = localStorage.key(i);
    // Verifica se a chave começa com "tarefa_"
    if (chave.startsWith("tarefa_")) {
        // Se a chave corresponder ao padrão, obtém o valor (tarefa) associado a essa chave
      const tarefa = localStorage.getItem(chave);
      // Cria um novo elemento li para representar a tarefa
      const li = document.createElement("li");
      // Define o texto do elemento li como a tarefa recuperada do localStorage
      li.textContent = tarefa;
      // Adiciona o elemento li à lista de tarefas no HTML
      document.getElementById("lista-tarefas").appendChild(li);
    }
  }
};
</script>
```

Explore essas sugestões e sinta-se à vontade para criar as suas próprias ideias! A personalização é a chave para fazer algo realmente único. Divirta-se codificando!

jQuery

O jQuery é uma biblioteca de JavaScript amplamente utilizada no desenvolvimento *web*. Pense nela como um conjunto de ferramentas prontas para simplificar tarefas comuns, tornando a criação de *sites* interativos mais fácil e eficiente. Criada para **simplificar o uso do JavaScript**, essa biblioteca fornece uma sintaxe mais amigável e concisa, permitindo que você faça mais coisas com menos código. Por exemplo, em vez de escrever várias linhas de código para realizar tarefas, é possível usar métodos simples e intuitivos fornecidos pelo jQuery, economizando tempo e esforço no desenvolvimento da sua aplicação.

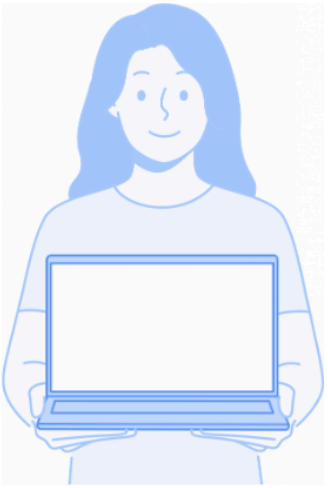
Além disso, uma das principais funções do jQuery é **facilitar a manipulação de elementos em uma página *web***. Com ele, você pode selecionar facilmente elementos HTML, modificar o seu conteúdo, o seu estilo e, até mesmo, responder a eventos, como cliques do usuário. Conheça mais algumas das vantagens de se utilizar o jQuery:

Cross-browser compatibility	O jQuery cuida automaticamente de diferenças entre navegadores, o que significa que você pode escrever um código que funciona de maneira consistente em diferentes navegadores, evitando problemas comuns no desenvolvimento *web*.
Seleção de elementos	O jQuery permite selecionar elementos HTML de forma mais simples. Por exemplo, você pode selecionar todos os botões da página usando o **$("button")**.
Manipulação do DOM	O jQuery facilita a manipulação do DOM (*Document Object Model*), permitindo adicionar, remover ou modificar elementos na página.
Eventos	O jQuery simplifica o tratamento de eventos, como cliques, teclas pressionadas etc. Por exemplo, você pode usar **$("button").click(function() { ... })** para lidar com cliques em botões.
Efeitos e animações	O jQuery oferece métodos para criar animações e efeitos visuais com a utilização das funções. **.fadeIn()** e **.slideUp()**.
Ajax	O jQuery facilita as chamadas assíncronas ao servidor (Ajax), permitindo que os dados sejam carregados dinamicamente, sem precisar recarregar a página.

Agora que você já conhece as características do jQuery e entendeu como ele é utilizado no desenvolvimento *web*, é hora de praticar!

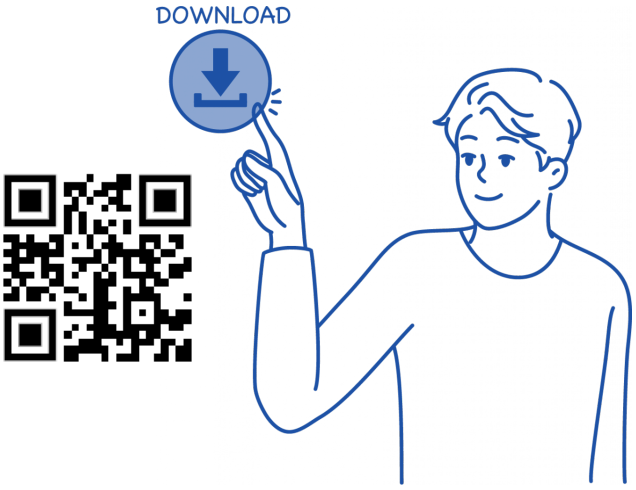
Mão na massa!

Siga o passo a passo a seguir e **use o jQuery para ocultar elementos no seu projeto da lista de tarefas**.



Passo 1 - instalação e incorporação do jQuery ao projeto

Comece adicionando o jQuery ao seu projeto. Você pode utilizar um CDN (*Content Delivery Network*, em português, Rede de Distribuição de Conteúdo) ou baixar o arquivo diretamente do *site* oficial, que você pode acessar através do QR Code a seguir.



Feito isso, insira o seguinte *script* no seu HTML para incluir a biblioteca jQuery:

```
<script src="https://code.jquery.com/jquery-3.6.0.min.js"></script>
```

Passo 2 - seleção e manipulação do DOM

O jQuery simplifica a manipulação do DOM através de seletores intuitivos. Por exemplo, para alterar o texto de todos os parágrafos, utilize:

```
$('p').text('Novo texto para todos os parágrafos');
```

Você também pode selecionar um elemento HTML (por exemplo, **$("button")**) e usar métodos jQuery para manipulá-lo usando **.hide()** ou **.show()**.

Passo 3 - tratamento eficiente de eventos

Vincular eventos torna-se mais conciso com o jQuery. Nessa etapa, vincule uma função ao clique de todos os botões. Para isso, utilize o código a seguir:

```
$('button').on('click', function() {
  alert('Botão clicado!');
});
```

Passo 4 - incorporando animações

As animações podem ser facilmente incorporadas usando os métodos de animação jQuery. Por exemplo, para realizar um efeito de *fade-out* em elementos com a classe **'elemento'**, utilize:

```
$('.elemento').fadeOut(1000);
```

Passo 5 -: chamadas AJAX simplificadas

Como já estudamos, o JQuery simplifica chamadas Ajax para carregamento assíncrono de dados. Nessa etapa, realize uma requisição usando a função **ajax** do jQuery. Veja como isso deve pode ser feito no código a seguir:

```
// Este código é um exemplo de uma chamada AJAX usando jQuery para solicitar dados de uma API.

// Aqui está o que cada parte do código faz:



// Inicia uma chamada AJAX usando a função ajax do jQuery.

$.ajax({

  // Define a URL da API de onde os dados serão solicitados.
```

```
  url: 'http://api.exemplo.com/dados',

  // Especifica que será utilizado o método HTTP GET para solicitar os dados.

  method: 'GET',

  // Uma função callback que é executada se a requisição for bem-sucedida.

  success: function(data) {

    // Imprime uma mensagem no console junto com os dados recebidos da API.

    console.log('Dados recebidos:', data);

  },

  // Uma função callback que é executada se ocorrer um erro na requisição.

  error: function(err) {

    // Imprime uma mensagem de erro no console junto com detalhes sobre o erro ocorrido.

    console.error('Erro na requisição:', err);

  }

});
```

Passo 6 - explorando *plugins* para funcionalidades adicionais

Amplie as funcionalidades do jQuery integrando *plugins* específicos ao seu projeto. Por exemplo, adicione um calendário à sua lista de tarefas incluindo o *plugin* correspondente:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/jqueryui/1.12.1/jquery-ui.min.js"
```

E aí, como ficou a sua lista de tarefas depois de utilizar o jQuery?
Compartilhe com seus colegas o que você conseguiu desenvolver no seu projeto!

12.3 Trabalhando com frameworks no desenvolvimento front-end

Trabalhar com *frameworks* no desenvolvimento *front-end* é uma prática que transformou significativamente a maneira como os *sites* e as aplicações *web* são construídos. Como já vimos, os *frameworks* visam simplificar e acelerar o desenvolvimento de projetos, proporcionando uma estrutura base sobre a qual os desenvolvedores podem construir as suas aplicações.

Ao iniciar um projeto do zero, os desenvolvedores enfrentam diversos desafios técnicos que vão desde a configuração do ambiente de desenvolvimento até a implementação de funcionalidades complexas e responsivas que funcionem bem em diferentes dispositivos e navegadores. Aqui, os *frameworks front-end* entram como soluções poderosas, oferecendo um caminho guiado que facilita esse processo.

Um dos grandes benefícios de utilizar *frameworks* é a eficiência em termos de tempo e recursos. Ao invés de dedicar horas codificando soluções padrão para problemas comuns, os desenvolvedores podem aproveitar as soluções já existentes dentro do *framework*, o que acelera o desenvolvimento e leva a um produto final mais aprimorado em menos tempo.

Disponível em: https://encurtador.com.br/dgrNY. Acesso em: 14 mar. 2024.

Os *frameworks* mais utilizados em JavaScript são **Angular** e **Vue.js**. Ambos são utilizados para construir aplicações *web* modernas e interativas, mas seguem abordagens ligeiramente diferentes e têm pontos fortes únicos.

Angular

Mantido pelo Google, o Angular é um framework robusto e completo para o desenvolvimento de aplicações web de página única (SPA - Single Page Applications), um tipo de aplicação web ou site que interage com o usuário reescrevendo a página atual em vez de carregar páginas inteiras do servidor. O Angular utiliza TypeScript, uma linguagem de programação de código aberto desenvolvida pela Microsoft, que adiciona tipos estáticos e objetos baseados em classes, entre outras funcionalidades. Isso ajuda a melhorar a qualidade do código e tornar as aplicações mais fáceis de manter. Ele é ideal para projetos complexos e de grande escala, em que a estrutura e as práticas padronizadas podem ajudar a manter o código organizado e escalável.

A seguir, vamos ver um exemplo usando o Tour of Heroes, que oferece uma visão completa do desenvolvimento de aplicações web com o Angular. Essa aplicação abrange desde a criação de componentes até a integração com serviços externos.

```
// heroes.component.ts

// Importamos o decorator 'Component' do módulo '@angular/core'.
import { Component } from '@angular/core';

// Definimos o componente 'HeroesComponent'.
@Component(
 // O seletor 'app-heroes' será usado no HTML para renderizar este    componente.
  selector: 'app-heroes',
```

```
  // O arquivo 'heroes.component.html' será usado como template para este componente.
  templateUrl: './heroes.component.html',
  // O arquivo 'heroes.component.css' contém estilos específicos para este componente.
  styleUrls: ['./heroes.component.css']
})
export class HeroesComponent {
  // Criamos uma lista de heróis como exemplo.
  heroes = ['Superman', 'Batman', 'Wonder Woman', 'Spider-Man'];
}
```

Vue.js

Lançado em 2014, o Vue.js é um *framework* utilizado para a construção de interfaces de usuário (UIs). Ao contrário do Angular, ele é projetado para ser adotado de forma incremental. A sua biblioteca principal foca apenas na camada de visualização, facilitando a integração com outras bibliotecas ou projetos existentes.

Esse *framework* também é conhecido pela sua simplicidade e flexibilidade, permitindo que desenvolvedores construam rapidamente interfaces interativas e SPAs de maneira eficiente, o que faz dele uma escolha popular entre desenvolvedores *front-end*.

A seguir, vamos ver um exemplo bem simples de Vue.js:

```
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Vue.js To-Do List Example</title>
    <!-- Inclua o Vue.js via CDN -->
    <script src="https://cdn.jsdelivr.net/npm/vue@2.6.14/dist/vue.js"></script>
</head>
<body>
    <div id="app">
        <!-- Exibe o título da lista -->
        <h1>Minha Lista de Tarefas</h1>
```

```
    <!-- Input para adicionar novas tarefas -->
    <input v-model="newTask" @keyup.enter="addTask" placeholder="Digite uma tarefa e pressione Enter">

    <!-- Lista de tarefas -->
    <ul>
      <!-- Usamos a diretiva v-for para iterar sobre as tarefas -->
      <li v-for="(task, index) in tasks" :key="index">
        {{ task }}
        <!-- Botão para remover a tarefa -->
        <button @click="removeTask(index)">Remover</button>
      </li>
    </ul>
  </div>

  <script>
    new Vue({
      el: '#app',
      data: {
        newTask: '', // Armazena a nova tarefa digitada
        tasks: [] // Array para armazenar as tarefas
      },
      methods: {
        addTask() {
          if (this.newTask.trim() !== '') {
            this.tasks.push(this.newTask); // Adiciona a nova tarefa à lista
            this.newTask = ''; // Limpa o campo de entrada
          }
        },
        removeTask(index) {
          this.tasks.splice(index, 1); // Remove a tarefa pelo indice
        }
      }
    });
  </script>
</body>
</html>
```

Entendendo o código

- Criamos uma instância **Vue** com um campo de entrada para adicionar tarefas;
- Usamos a diretiva **v-for** para iterar sobre as tarefas existentes e exibi-las na lista;
- Cada tarefa tem um botão "**Remover**", que chama o método **removeTask** para excluí-la.

Disponível em: https://encurtador.com.br/diBEZ. Acesso em: 14 mar. 2024.

12.4 Escolhendo a tecnologia certa para projetos específicos

A escolha da tecnologia certa, seja um *framework* ou uma biblioteca, para um projeto específico é uma decisão crucial que pode afetar a eficiência do desenvolvimento, a qualidade do produto final e, até mesmo, o sucesso do seu projeto a longo prazo.

Aqui, vamos apresentar um guia detalhado para te ajudar na escolha da tecnologia mais adequada para as necessidades do seu projeto.

1 Compreenda os requisitos do projeto

Antes de avaliar opções tecnológicas, é fundamental realizar uma análise completa dos requisitos do projeto. Isso inclui:

- identificar claramente os objetivos do projeto;
- definir o escopo do trabalho;
- listar as funcionalidades necessárias;
- considerar eventuais restrições (orçamento, prazos etc.).

A compreensão completa desses requisitos proporcionará uma base sólida para a escolha da tecnologia mais adequada.

2 Avalie as necessidades do *front-end*

Avalie a capacidade da tecnologia de criar interfaces de usuário interativas e responsivas no desenvolvimento *front-end*.

3 Conheça as características dos *frameworks* e bibliotecas

Explore minuciosamente os *frameworks* e as bibliotecas disponíveis para o desenvolvimento *front-end*. Considere detalhadamente o React, o jQuery, o Angular e o Vue.js, examine as suas características, os seus ecossistemas e a sua facilidade de integração;

4 Considere a escalabilidade e manutenção

Avalie a escalabilidade da tecnologia escolhida:

- certifique-se de que ela pode crescer conforme as demandas do projeto aumentam;
- analise as implicações de manutenção a longo prazo, incluindo:
  - ativa participação da comunidade;
  - atualizações frequentes;
  - recursos de segurança.

5 Analise o desempenho e segurança

Realize testes abrangentes de desempenho para garantir que a tecnologia escolhida atenda às expectativas de resposta rápida e eficiência. Além disso, considere a segurança do sistema.

6 Verifique se há integração com sistemas existentes

Se o projeto envolve a integração com sistemas já existentes, é fundamental conduzir uma análise minuciosa. O objetivo é garantir a compatibilidade e uma integração suave. A interoperabilidade é crucial para evitar complicações durante o desenvolvimento.

7 Faça uma análise detalhada de custos e orçamento

Considere todos os aspectos financeiros associados à adoção da tecnologia escolhida. Avalie:

- os custos de licença;
- o investimento em treinamento da equipe;
- a disponibilidade de suporte técnico;
- quaisquer outros custos indiretos.

Como você pode ver, escolher entre um *framework* e uma biblioteca depende de uma análise cuidadosa de diversos aspectos, como as necessidades específicas do projeto, os recursos da equipe e os objetivos de longo prazo. Ou seja, não existe uma resposta única para todos os projetos, a decisão deve ser baseada em uma combinação de fatores técnicos, de equipe e de negócios. Portanto, saber fazer uma avaliação cuidadosa é essencial para definir a tecnologia mais adequada para o seu projeto.

DESAFIO PRÁTICO

Selecionar uma tecnologia para um projeto específico

Descrição

Contexto:

Você é o líder técnico de uma equipe de desenvolvimento responsável por escolher a tecnologia adequada para um projeto inovador e complexo. O projeto envolve o desenvolvimento de uma plataforma de *e-commerce* altamente escalável, com recursos avançados de personalização, recomendação de produtos em tempo real e integração com sistemas de pagamento complexos.

Cenário:

A empresa está entrando em um novo mercado e busca criar uma presença *on-line* significativa. A plataforma de *e-commerce* é crucial para o sucesso da expansão e a escolha da tecnologia certa vai desempenhar um papel fundamental no desempenho, na escalabilidade e na capacidade de oferecer uma experiência de usuário excepcional.

Objetivos

- Escolher uma tecnologia que suporte a expansão significativa do número de usuários, garantindo um desempenho consistente mesmo em períodos de tráfego intenso;

- Identificar ferramentas e *frameworks* que permitam a implementação eficiente de recursos avançados de personalização, incluindo recomendações de produtos em tempo real com base no comportamento do usuário;
- Selecionar tecnologias que facilitem a integração suave com sistemas de pagamento complexos, garantindo segurança e conformidade com padrões do setor;
- Avaliar a viabilidade da implementação de uma arquitetura de microsserviços para facilitar o desenvolvimento, a implantação e a manutenção de componentes independentes;
- Priorizar tecnologias que ofereçam medidas de segurança para proteger dados sensíveis dos clientes, garantindo conformidade com regulamentações de privacidade.

Orientações

- Avalie *frameworks* e plataformas que tenham um histórico comprovado de escalabilidade, como serviços em nuvem, *cache* distribuído e balanceamento de carga eficiente;
- Considere *frameworks* de aprendizado de máquina e processamento de eventos em tempo real para implementar recursos avançados de personalização e recomendação;
- Pesquise tecnologias que ofereçam integração fácil com *gateways* de pagamento populares e forneçam APIs robustas para comunicação com sistemas financeiros.
- Explore *frameworks* e plataformas que suportem a arquitetura de microsserviços, facilitando o desenvolvimento, a implantação e a manutenção de componentes independentes;
- Escolha tecnologias que ofereçam recursos avançados de segurança, como criptografia forte, gerenciamento de identidade e controle de acesso granular.

RESUMO

Neste capítulo, estudamos a importância das bibliotecas e *frameworks* no desenvolvimento em JavaScript, destacando a sua contribuição para a eficiência, produtividade e manutenibilidade dos projetos. Vimos que as bibliotecas oferecem conjuntos de funções reutilizáveis para realizar tarefas específicas, enquanto os *frameworks* fornecem uma estrutura mais abrangente, com arquiteturas predefinidas e convenções para acelerar o desenvolvimento.

Aprendemos que as bibliotecas mais utilizadas em JavaScript são React e jQuery, que dispõem de diversos benefícios e vantagens para o desenvolvimento *front-end*. Vimos que o React é muito utilizado para criar componentes reutilizáveis e construir interfaces de usuário (UIs) interativas e que uma de suas características é seu modelo de programação declarativo. Também aprendemos que, assim como o React, o jQuery também é uma biblioteca amplamente utilizada, criada para simplificar o uso do JavaScript, sendo uma ótima opção para simplificar tarefas comuns de desenvolvimento *web*, como a manipulação de elementos DOM e chamadas Ajax.

Em seguida, abordamos de forma mais aprofundada os *frameworks* Angular e Vue.js, destacando as suas características, os seus usos e as suas diferenças. Aprendemos que o Angular é ideal para projetos complexos e de grande escala, enquanto o Vue.js tem como vantagens a sua simplicidade e flexibilidade, permitindo que os desenvolvedores construam interfaces interativas e SPAs de forma rápida e eficiente.

Por fim, apresentamos orientações importantes que auxiliam na hora de escolher a tecnologia certa para projetos específicos. Estudamos que este processo envolve uma análise aprofundada dos *frameworks* e bibliotecas, além dos requisitos do projeto e uma avaliação cuidadosa das necessidades do *front-end*. Ao seguir essas orientações, os desenvolvedores podem tomar decisões informadas, garantindo que a tecnologia escolhida seja adequada, eficiente e alinhada com os objetivos específicos de cada projeto, resultando em soluções bem-sucedidas.

ATIVIDADE DE FIXAÇÃO

1. Explique a diferença fundamental entre um *framework* e uma biblioteca em JavaScript, fornecendo exemplos específicos para ilustrar cada conceito.

2. Ao escolher um *framework* para um projeto *web*, quais fatores você consideraria primordiais? Destaque a importância de características como comunidade, documentação e facilidade de aprendizado.

3. Descreva o padrão de *design* MVC (*Model-View-Controller*) e explique como *frameworks* como Angular e Vue.js implementam ou seguem esse padrão.

4. Ao desenvolver uma aplicação *web* com uma biblioteca *front-end*, como o React, quais são as vantagens e desvantagens da abordagem de componentes?

5. Como as bibliotecas, como o jQuery, facilitam o desenvolvimento *front-end* em comparação com a manipulação direta do DOM? Dê exemplos práticos de situações em que o uso de jQuery pode ser vantajoso.

6. Qual é o objetivo principal da aplicação Tour of Heroes no Angular?

7. Qual é a importância da escalabilidade ao escolher um *framework* para um projeto? Discuta as considerações que os desenvolvedores devem levar em conta em relação à escalabilidade.

8. Em que situações você escolheria uma biblioteca específica em detrimento de um *framework* ao desenvolver uma aplicação *web*? Dê exemplos de casos em que essa escolha pode ser mais apropriada.

9. Ao considerar a segurança em *frameworks* e bibliotecas JavaScript, discuta práticas recomendadas e recursos que ajudam a garantir a segurança de uma aplicação *web*.

10. Como as bibliotecas modernas, como o React, têm abordado o desafio da gerência de estado em aplicações complexas? Explique conceitos como "*lifting state up*" e "*context*" no contexto dessas bibliotecas.

Capítulo 13

DESENVOLVIMENTO JAVASCRIPT: PADRONIZAÇÃO, DEPURAÇÃO E TRATAMENTO DE ERROS

O que esperar deste capítulo:

- Utilizar ferramentas como ESLint para garantir a consistência e qualidade do código;
- Utilizar recursos de inspeção de elementos, monitoramento de rede e console para depurar aplicações JavaScript;
- Identificar e resolver gargalos de desempenho em código JavaScript.

13.1 Padronização de código e convenções de nomenclatura

Ao desenvolver um *software,* é importante usar os códigos de maneira padronizada, respeitando as convenções de nomenclatura, pois, dessa forma, você estará criando códigos legíveis, consistentes e de fácil manutenção. Para te ajudar nesse processo, vamos explorar um conjunto de práticas que vão garantir a qualidade do seu código JavaScript.

Boas práticas de codificação

Antes de explorarmos as convenções específicas de nomenclatura, é preciso ter em mente os benefícios que as boas práticas de codificação podem trazer:

Se liga!

Existem ferramentas, como o **ESLint,** que ajudam a automatizar a aplicação de boas práticas de codificação. Essas ferramentas verificam os códigos em busca de erros, inconsistências e violações das convenções.

Daqui a pouco nós vamos conhecer mais sobre o ESLint.

Convenções de nomenclatura

A consistência nas nomenclaturas é crucial para facilitar a leitura do código. Para isso, devemos utilizar convenções de formatação de texto como o **CamelCase** e o **PascalCase.** As suas aplicações são:

- **CamelCase:** utilizado para variáveis e funções;
- **PascalCase:** utilizado para classes e nomes descritivos.

CamelCase

Neste estilo, as palavras são agrupadas sem espaços e cada palavra subsequente após a primeira começa com letra maiúscula, **exceto a primeira palavra.**

É frequentemente usado em linguagens como JavaScript, Python e C#, principalmente para nomear variáveis e funções.

```
let nomeDoUsuario = "John";
function calcularSoma(numero1, numero2) {
    return numero1 + numero2;
}
```

PascalCase

Semelhante ao CamelCase, mas a primeira letra de cada palavra é maiúscula, **incluindo a primeira palavra.**

É amplamente usado em linguagens como C#, Java e C++, especialmente para nomear classes e métodos públicos.

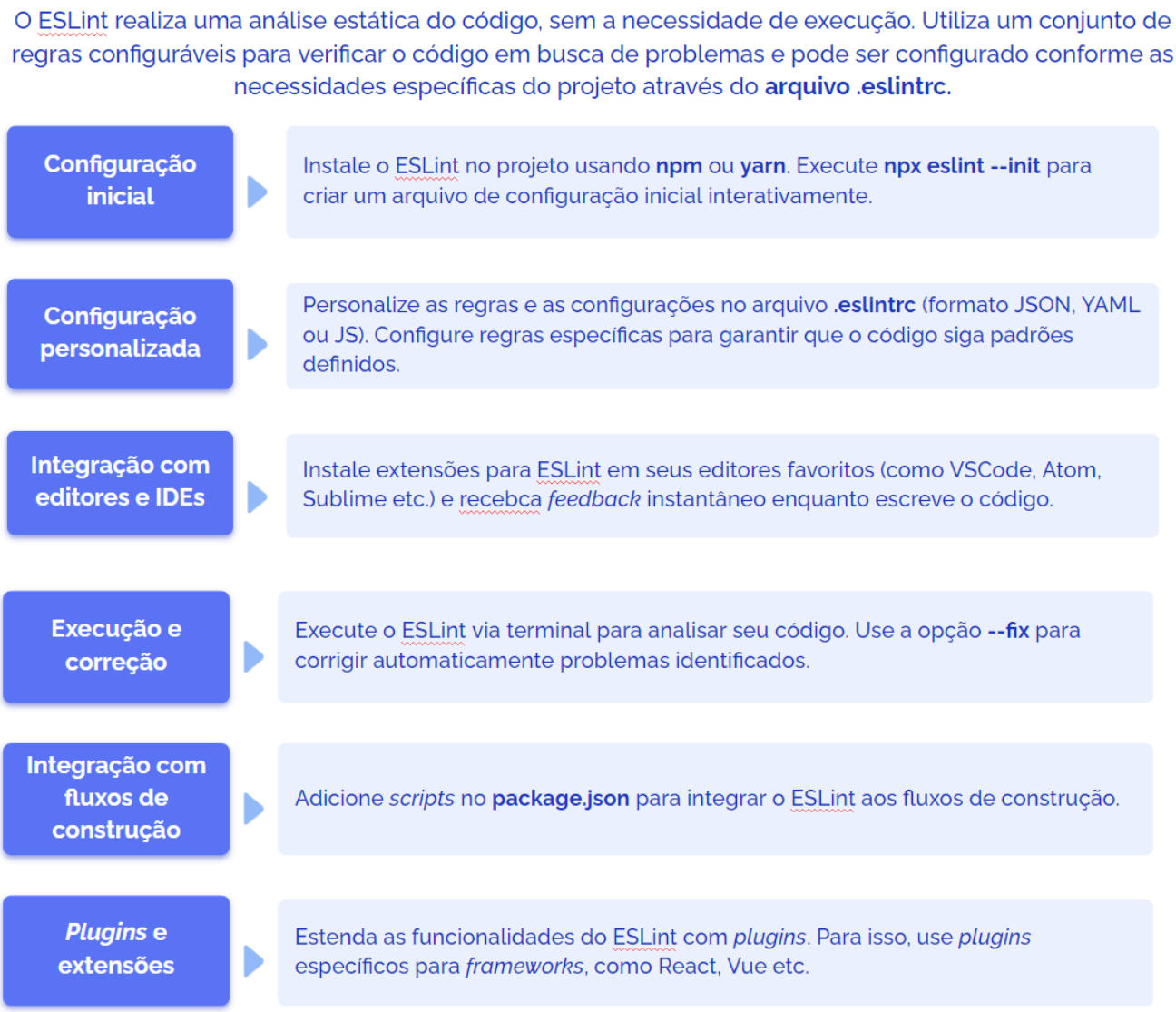
```
class Produto {
    constructor(nome, preco) {
        this.nome = nome;
        this.preco = preco;
    }
}
```

ESLint: uma ferramenta para boas práticas de codificação

O ESLint é uma ferramenta de *linting* para JavaScript, criada para ajudar desenvolvedores a manterem um código consistente, identificando e corrigindo erros, padrões de codificação inconsistentes e más práticas.

Ele garante que toda a equipe siga as mesmas convenções e práticas de codificação. Além disso, ele também **ajuda a identificar problemas antes mesmo de executar o código**, reduzindo *bugs* e melhorando a qualidade do código e enfatiza a legibilidade e manutenibilidade do código, tornando-o mais fácil de entender.

Certamente, você já conseguiu entender a importância de utilizar o ESLint. Mas, **como ele funciona?**



Uso com Git Hooks

Integre o ESLint com Git Hooks para garantir que o código esteja em conformidade antes de ser comprometido.

Exemplo da configuração

Instalação das dependências

- No terminal, execute o seguinte comando para instalar as dependências do ESLint e do Prettier no projeto:

```
npm install eslint eslint-config-prettier eslint-plugin-prettier prettier --save-dev
```

Configuração do ESLint

- Crie um arquivo **.eslintrc.js** na raiz do projeto;
- Defina as regras de formatação e estilo no arquivo **.eslintrc.js**. Por exemplo:

```
module.exports = {
  extends: ['eslint:recommended', 'plugin:prettier/recommended'],
  rules: {
   // Defina suas regras personalizadas aqui
  },
 };
```

Configuração do Prettier

- Crie um arquivo **.prettierrc.js** na raiz do projeto;
- Configure as opções do Prettier no arquivo **.prettierrc.js**. Por exemplo:

```
module.exports = {
  semi: true,
  singleQuote: true,
  trailingComma: 'all',
 };
```

Integração com o editor (VSCode)

- Instale as extensões do ESLint e do Prettier no VSCode;
- Abra as configurações do VSCode e habilite a formatação automática ao salvar o arquivo.

Documentação e comunidade

Consulte a documentação oficial do ESLint para detalhes sobre regras, configuração avançada e atualizações. Essa documentação pode ser encontrada no *site* oficial da ferramenta, que pode ser acessado lendo o QR code a seguir.

Além disso, faça parte da comunidade do ESLint para obter suporte, compartilhar experiências e contribuir para o desenvolvimento contínuo.

Code review

Quando um desenvolvedor cria um código-fonte, é importante que ele passe pelo processo de *code review*, ou, em português, revisão de código. Nesse processo, o código-fonte criado é **examinado** por outros membros da equipe de desenvolvimento antes de ser integrado ao repositório principal do projeto.

O objetivo principal da revisão de código é melhorar a qualidade do código, identificando:

- possíveis erros;
- problemas de *design*;
- oportunidades de otimização;
- violações de padrões de codificação.

Durante uma revisão, os revisores analisam o código para garantir que ele seja legível, mantível, seguro e eficiente. Eles podem procurar por questões como:

Erros de lógica	Qualidade do código	Boas práticas de programação	Segurança	Desempenho
Identificar *bugs* ou comportamentos inesperados no código.	Avaliar a clareza, a legibilidade e a coesão do código.	Verificar se o código segue as diretrizes de codificação e padrões estabelecidos pela equipe.	Identificar possíveis vulnerabilidades de segurança ou práticas inseguras no código.	Identificar oportunidades de otimização para melhorar o desempenho do código.

Integração Contínua (CI): contribuindo para a qualidade do código

Outro recurso importante para garantir a qualidade do código é a Integração Contínua (CI, do inglês *Continuous Integration*). Ela faz uma **compilação do código** sempre que uma alteração é feita no repositório central do projeto e realiza **testes automatizados** regularmente, ajudando a garantir que o código introduzido no projeto seja funcional e atenda aos requisitos definidos. Isso tudo contribui para a manutenção da qualidade do *software*.

Ao integrar o código de diferentes membros da equipe de forma contínua, a CI facilita a colaboração entre desenvolvedores, garantindo que todos trabalhem em uma base comum e tenham visibilidade sobre as alterações realizadas.

Pipelines CI/CD

Pipelines são sequências automatizadas de processos que ocorrem durante o desenvolvimento, teste e entrega de um *software*. Esses processos são organizados em etapas e executados em ordem, formando uma "linha de produção" para o código, desde a sua integração até a sua implantação.

Esse processo pode ser dividido em duas etapas principais: a **Integração Contínua,** ou CI (*Continuous Integration*), e a **Entrega Contínua** ou CD (*Continuous Delivery*).

Integração Contínua (CI)

É a prática de mesclar todas as alterações de código de desenvolvedores em um repositório central de forma contínua. As alterações de código são, então, automaticamente construídas, testadas e validadas para detectar problemas de integração o mais cedo possível.

1. Um desenvolvedor faz uma alteração no código-fonte e envia-a para o repositório compartilhado (por exemplo, Git);
2. O servidor de integração contínua detecta a mudança e inicia automaticamente o processo de integração da parte alterada;

3. O código é compilado, testado e verificado quanto à sua qualidade;
4. Se os testes passarem e a qualidade for aceitável, o código é considerado integrado.

Entrega Contínua (CD)

É a prática de entregar automaticamente o código validado para o ambiente de produção. Com a CD, você pode decidir liberar seu aplicativo para produção a qualquer momento, garantindo que o seu *software* esteja sempre pronto para ser lançado.

1. Após a integração, o próximo passo é a entrega contínua;
2. O código aprovado é implantado em um ambiente de teste ou pré-produção;
3. Testes adicionais, como testes de aceitação, são executados;
4. Se tudo estiver bem, o código é promovido para o ambiente de produção.

Embora seja possível executar **manualmente** cada uma das etapas de um *pipeline* de CI/CD, o verdadeiro valor dos *pipelines* é realizado através da **automação**. Ela ajuda a minimizar erros humanos e a manter um processo consistente de como o *software* é lançado.

As ferramentas que estão incluídas no pipeline podem incluir:

- compilação de código;
- testes de unidade;
- análise de código;
- segurança;
- criação de binários.

Para ambientes em contêineres, este *pipeline* também pode incluir o empacotamento do código em uma imagem de contêiner para ser implantada em uma nuvem híbrida.

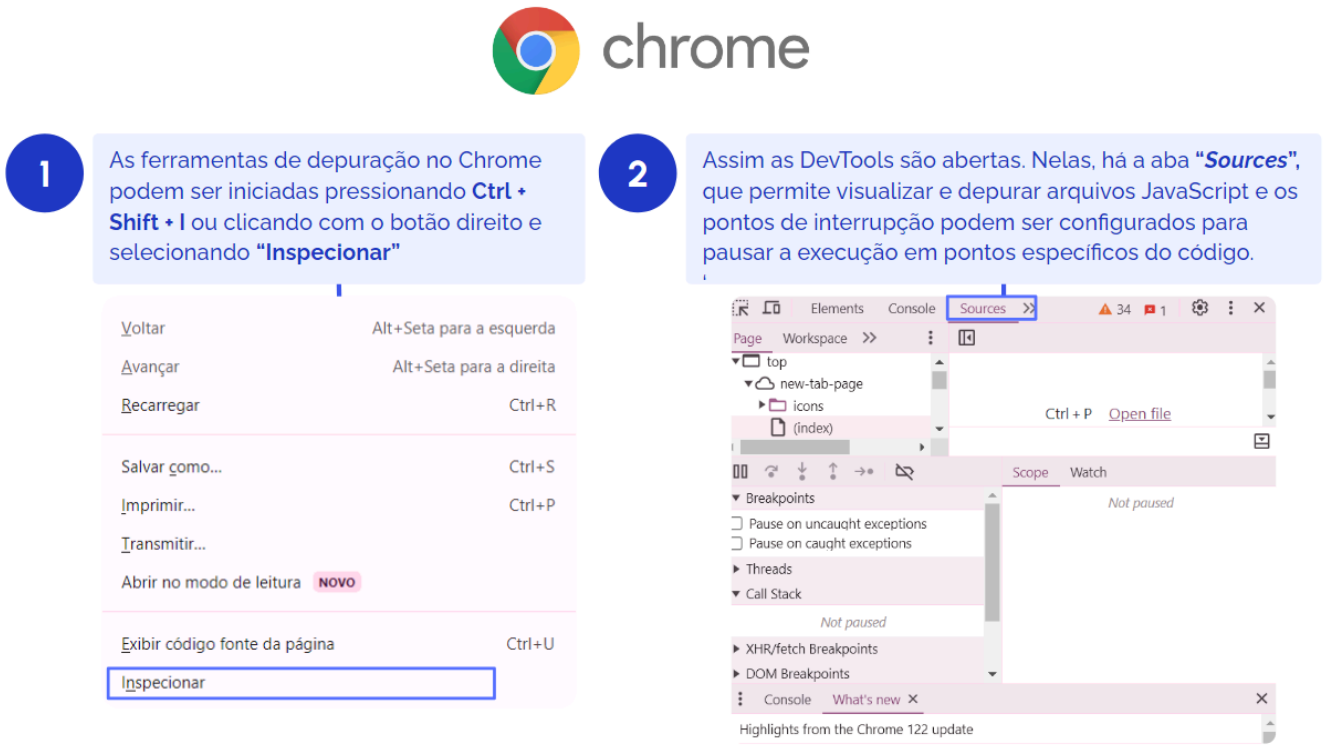
13.2 Utilizando ferramentas de depuração do navegador

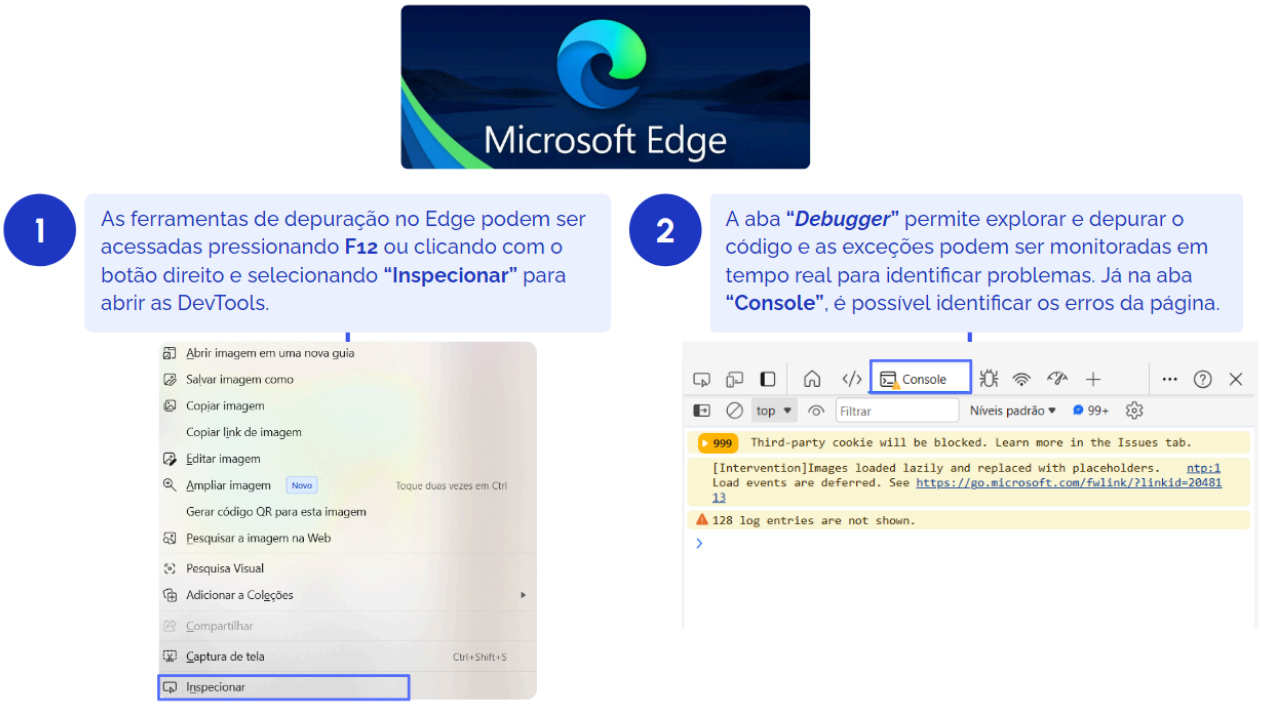
A depuração do navegador é o processo de **identificar**, **analisar** e **corrigir problemas em aplicações *web*** através das ferramentas de desenvolvimento integradas nos navegadores.

Os navegadores modernos, como o Google Chrome, Mozilla Firefox e Microsoft Edge, oferecem conjuntos de ferramentas de desenvolvimento que permitem aos desenvolvedores inspecionar elementos da página, monitorar o desempenho, depurar JavaScript e muito mais.

Essas ferramentas se localizam no **DevTools,** uma abreviação para "*Developer Tools*" (em português, Ferramentas de Desenvolvimento), que se refere às ferramentas integradas em **todos os navegadores modernos** para ajudar os desenvolvedores a criar, testar e depurar aplicações *web*.

Cada navegador possui suas próprias DevTools, que, por sua vez, oferecem uma ampla variedade de recursos para facilitar o desenvolvimento *web*.





Como desenvolvedores, é essencial compreender as ferramentas à nossa disposição para otimizar o processo de desenvolvimento. Assim como vimos, as DevTools, presentes em **todos os**

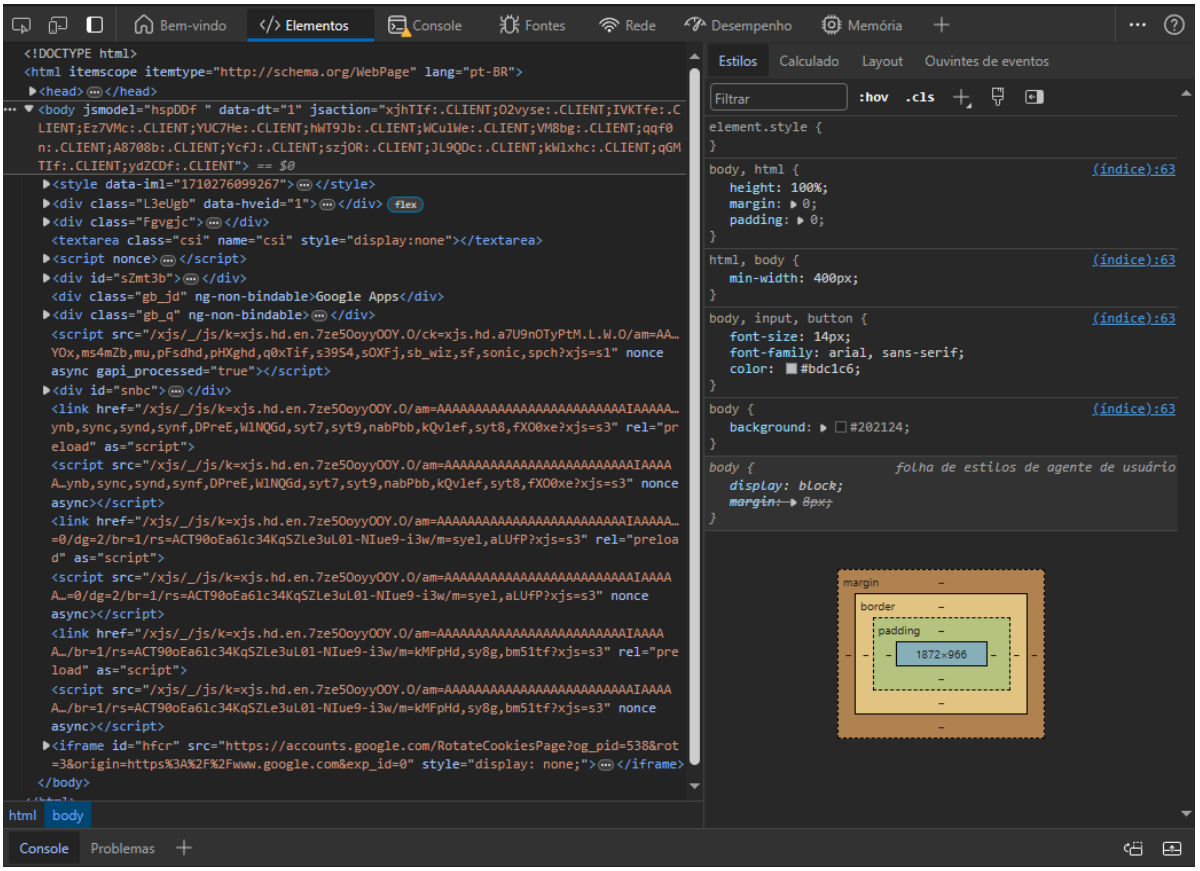
navegadores, têm diversas ferramentas poderosas com essa finalidade. Vamos dar uma olhada nas principais abas disponíveis, que são:

- **Elementos </>;**
- **Console;**
- **Fontes (*Sources*);**
- **Redes (*Network*).**

Elementos </>

Nas DevTools, a aba "Elementos", ou *Elements*, permite verificar todos os elementos da página.

Nesse espaço, você pode fazer uma **análise detalhada** e em tempo real do HTML e do CSS de uma página da *web* e **modificar elementos** como o conteúdo de texto, os estilos, adicionar classes etc.

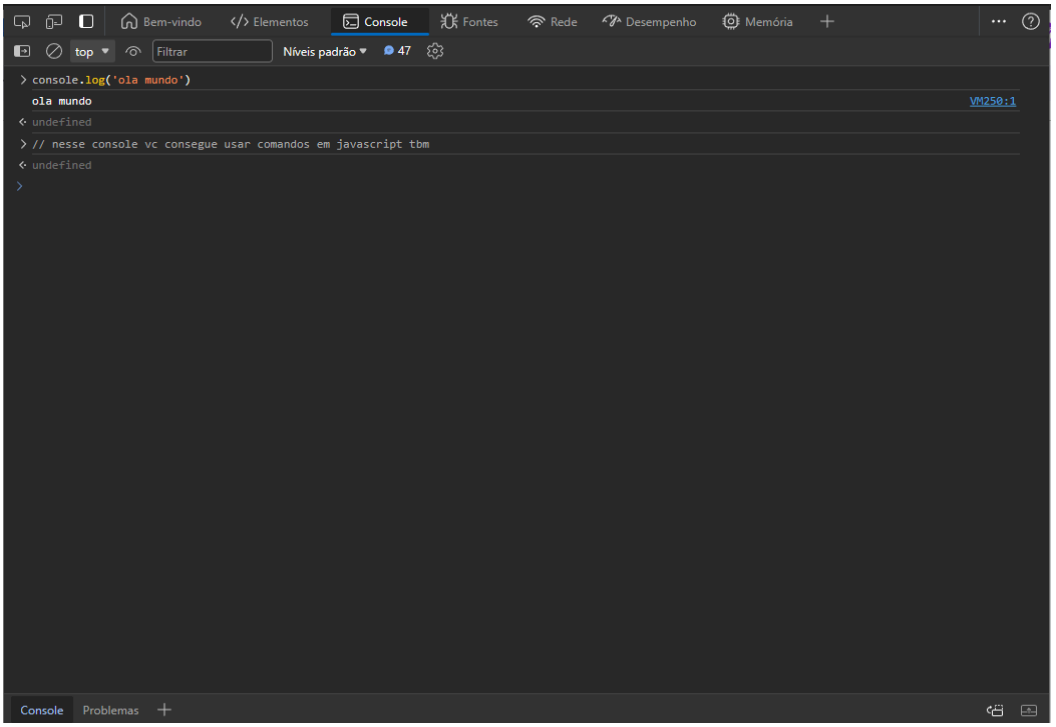


Fonte: Imagem desenvolvida pelo próprio autor (2024).

Console </>

A aba "Console" oferece uma interface de linha de comando repleta de recursos, como:

- **visualização de mensagens logadas:** aqui, os desenvolvedores podem deixar mensagens importantes. Usando **console.log()**, **console.warn()**, **console.error()**, entre outros, é possível registrar mensagens úteis enquanto o JavaScript roda;
- **depuração:** o console oferece comandos especiais para indicar *bugs* e problemas, como **console.error()** e **console.warn()**;
- **execução de JavaScript:** é possível executar JavaScript diretamente no console, mudando o título da página ou mexendo com o DOM;
- **acesso ao DOM:** é possível explorar e manipular elementos HTML diretamente do console. Isso é ideal para testar pequenas alterações ou entender a estrutura da página.

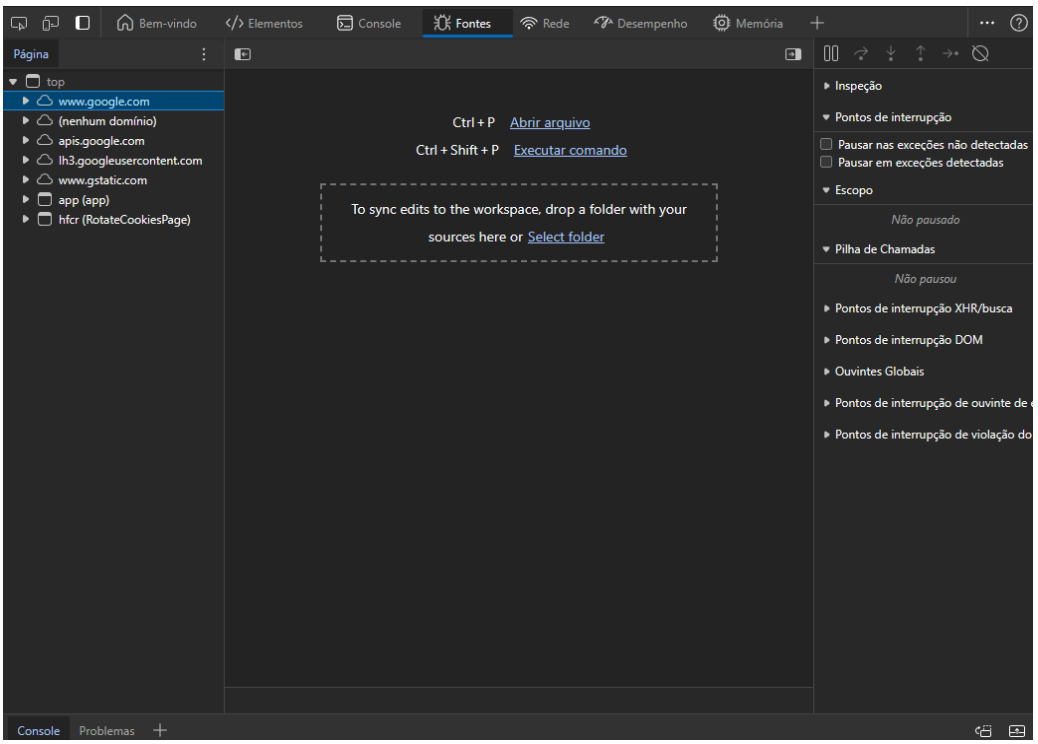


Fonte: Imagem desenvolvida pelo próprio autor (2024).

Fontes (sources)

Nas DevTools, a aba "Fontes" é o lugar em que você pode analisar os arquivos de origem da sua página *web* de maneira mais profunda. Algumas possibilidades são:

- **visualização de arquivos:** nessa aba, é possível explorar todos os arquivos de origem carregados pela página, seja JavaScript, CSS ou HTML;
- **edição de arquivos:** aqui, você pode modificar os arquivos de origem diretamente nas DevTools e ver as mudanças acontecerem em tempo real na página;
- **mapeamento de origem:** mesmo que o código esteja minificado, os mapas de origem vão te ajudar a depurar o código original.

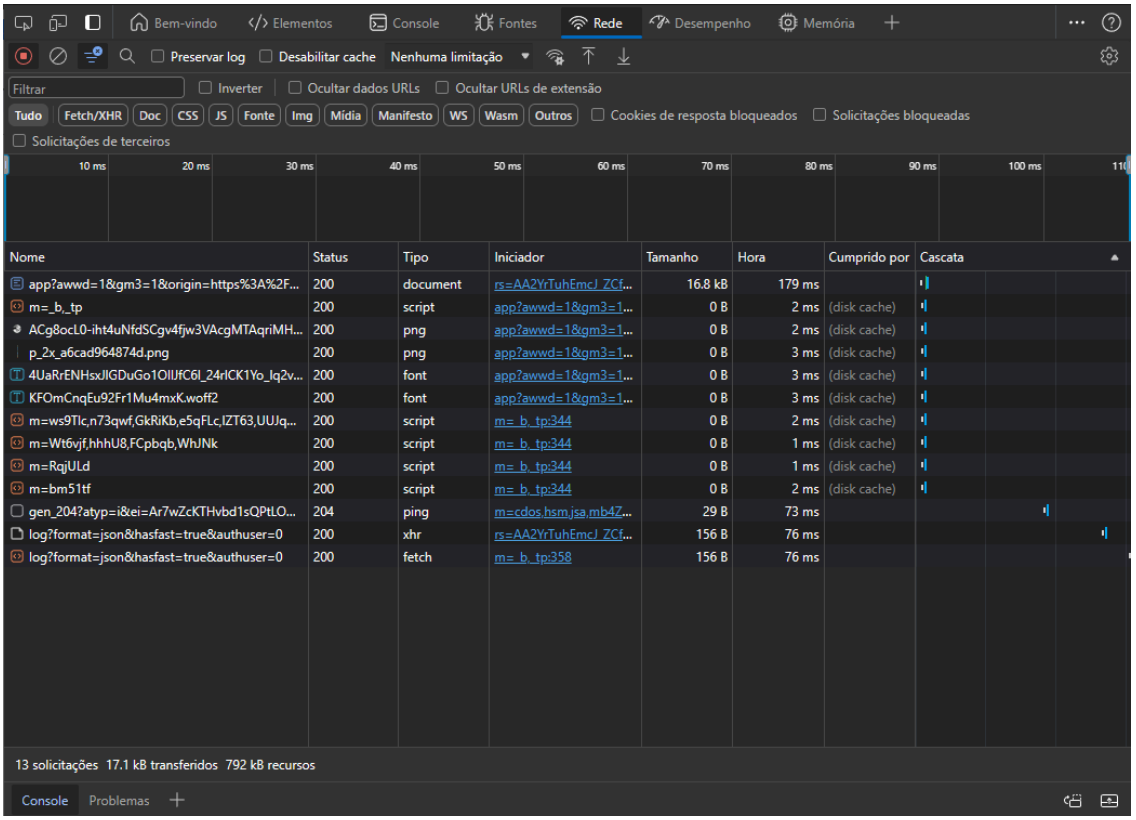


Fonte: Imagem desenvolvida pelo próprio autor (2024).

Redes (*Networks*)

A aba "Redes" é usada para monitorar e analisar as solicitações de rede feitas pelo navegador ao carregar uma página da *web*. Nessa aba, é possível:

- ver todas as solicitações de rede com detalhes, como a URL, o método, o *status* e o tempo de carregamento;
- analisar cada solicitação ou resposta, incluindo cabeçalhos HTTP, parâmetros e corpo da resposta;
- filtrar as solicitações por tipo de recurso ou *status* da resposta.



Fonte: Imagem desenvolvida pelo próprio autor (2024).

Hora do desafio

Escolha um site da *web*, abra as DevTools no navegador e use a aba **"Console"** para verificar se existem erros na página ou alguma mensagem de aviso. Ao final, lembre-se que documentar todas as alterações que você observou. Vamos lá?

13.3 Lidando com erros comuns em JavaScript

Muitas vezes, o desenvolvimento em JavaScript envolve a identificação e correção de erros comuns. Por isso, para ser um bom desenvolvedor, é importante saber como resolver esses problemas de maneira eficiente.

Entendendo os tipos de erros

- Familiarize-se com os tipos comuns de erros em JavaScript, como **ReferenceError**, **TypeError** e **SyntaxError**. Cada um fornece pistas sobre a natureza do problema.

ReferenceError: erro que ocorre quando você tenta acessar uma variável que não foi declarada ou está fora do escopo atual;

TypeError: erro que ocorre quando você tenta executar uma operação em um valor de um tipo que não suporta essa operação;

SyntaxError: erro ocorre quando o interpretador encontra um erro de sintaxe no código JavaScript. Isso pode acontecer por uma variedade de razões, como uma palavra-chave mal escrita, uma expressão mal formada ou um erro de digitação.

- No console do navegador, observe as mensagens de erro fornecidas, incluindo a linha do código em que o erro ocorreu.

Utilizando try-catch para capturar exceções

- Identifique o bloco de código suscetível a erros e envolva-o em um bloco **try**.
- Adicione um bloco **catch** para lidar com a exceção, exibindo uma mensagem útil ou executando ações específicas.

```
try {
    // Bloco de código propenso a erros
    const resultado = dividirNumeros(10, 0);
    console.log(resultado);
} catch (erro) {
    // Trata a exceção
    console.error("Erro ao dividir:", erro.message);
}
```

Utilizando finally para código de limpeza

- Adicione um bloco **finally** após o bloco **try-catch**.
- Coloque código que deve ser executado, independentemente de ocorrer uma exceção ou não.

```
try {
    // Bloco de código propenso a erros
    const resultado = dividirNumeros(10, 0);
    console.log(resultado);
} catch (erro) {
    // Trata a exceção
    console.error("Erro ao dividir:", erro.message);
} finally {
    // Código de limpeza
    console.log("Finalizando a execução.");
}
```

Lançando exceções personalizadas

- Crie uma classe de erro personalizada estendendo **Error**.
- Utilize a palavra-chave **throw** para lançar a exceção personalizada.

```
class MeuErro extends Error {
    constructor(mensagem) {
        super(mensagem);
        this.name = "MeuErro";
    }
}

function exemploErroPersonalizado() {
    throw new MeuErro("Isso é um erro personalizado!");
}
```

Logging detalhado para depuração

- Utilize a função **console.error** para mensagens de erro detalhadas.
- Inclua informações relevantes, como os valores de variáveis, para facilitar a identificação do problema.

```
function exemploErroDetalhado() {
    const numero = "Não é um número";
    if (isNaN(numero)) {
        console.error(`Erro: "${numero}" não é um número válido.`);
    }
}
```

Vulnerabilidades comuns em JavaScript

Injeção de código (*Injection*):

- a injeção de código ocorre quando um invasor consegue inserir um código malicioso em um sistema, geralmente através de entradas não validadas que são interpretadas como comandos legítimos pelo sistema;
- pode permitir ao invasor obter acesso não autorizado a informações sensíveis, manipular dados, executar comandos maliciosos e até mesmo assumir o controle do sistema afetado.

XSS (*Cross-Site Scripting*):

- o XSS ocorre quando um invasor consegue inserir *scripts* maliciosos em páginas da *web*, que, posteriormente, são executados nos navegadores dos usuários. Isso é possível quando o aplicativo *web* não sanitiza corretamente a entrada do usuário antes de exibi-la em suas páginas;
- pode ser usado para roubar informações confidenciais, como *cookies* de autenticação, sequestrar sessões de usuário, redirecionar para *sites* maliciosos ou realizar outras ações maliciosas em nome do usuário afetado.

CSRF (*Cross-Site Request Forgery*):

- a CSRF ocorre quando um invasor engana um usuário autenticado a executar ações indesejadas em um aplicativo *web* sem o seu conhecimento ou consentimento. Isso é possível quando o aplicativo não verifica a origem das solicitações que recebe;
- Se um usuário autenticado visitar esse *site* malicioso enquanto estiver logado no aplicativo legítimo, a solicitação pode ser executada em nome do usuário sem que ele saiba;
- pode ser usado para executar ações indesejadas em nome de usuários autenticados, como transferências de fundos, alterações de configuração ou qualquer outra ação que o aplicativo permita.

Para combater essas e outras vulnerabilidades, você pode adotar medida como:

Combatendo vulnerabilidades comuns

- **Validação de entrada:** verifique e valide todas as entradas de dados do usuário para garantir que estejam em um formato esperado e seguro. Isso ajuda a prevenir ataques de injeção de código, como SQL *Injection* e XSS.
- **Sanitização de dados:** remova qualquer conteúdo malicioso ou indesejado dos dados antes de processá-los ou exibi-los. Isso ajuda a prevenir ataques de XSS e outros tipos de injeção de código.
- **Controle de acesso:** implemente mecanismos de autenticação e autorização para garantir que apenas usuários autorizados tenham acesso a recursos ou funcionalidades específicas do aplicativo.
- **Uso de bibliotecas de segurança de terceiros:** utilize bibliotecas de segurança de terceiros que tenham sido amplamente testadas e comprovadas para lidar com aspectos críticos de segurança, como criptografia, prevenção de CSRF e XSS.

DESAFIO PRÁTICO

Lidando com erros comuns em JavaScript

Descrição

A sua equipe está desenvolvendo uma aplicação *web* para uma grande empresa de *e-commerce*. Essa aplicação permite aos usuários pesquisar, visualizar e comprar produtos, além de interagir com outros usuários por meio de recursos de *chat* e comentários.
Durante o desenvolvimento, vários erros comuns em JavaScript surgiram, incluindo erros de sintaxe, manipulação inadequada de exceções e problemas de segurança.

Objetivos

- Identificar e corrigir erros de sintaxe no código JavaScript para garantir que ele seja executado corretamente pelo navegador;
- Implementar tratamento adequado de exceções para lidar com erros inesperados de forma elegante e evitar que a aplicação quebre;
- Implementar práticas de segurança adequadas para proteger a aplicação contra vulnerabilidades comuns, como a injeção de código, o XSS e a CSRF.

Orientações

- Utilize ferramentas de desenvolvimento integradas dos navegadores para identificar erros de sintaxe no código JavaScript e corrigi-los conforme for necessário;
- Utilize blocos **try-catch** para envolver um código suscetível a erros e implemente tratamentos adequados para diferentes tipos de exceções;
- Implemente práticas de segurança recomendadas, como validação de entrada, sanitização de dados, controle de acesso e uso de bibliotecas de segurança de terceiros quando apropriado.

RESUMO

Neste capítulo, abordamos a importância de implementar boas práticas de codificação em JavaScript para garantir, não apenas a legibilidade do código, mas também a manutenção eficiente do *software*. Também exploramos as DevTools e as ferramentas de depuração que os navegadores dispõem, além de entender as funções das diversas guias do DevTools e como elas podem ser utilizadas pelos desenvolvedores.

Por fim, aprendemos as melhores maneiras de identificar e lidar com erros comuns em JavaScript, mantendo um padrão de qualidade elevado, detectando erros e *bugs* antes que eles causem estragos.

ATIVIDADE DE FIXAÇÃO

1. Qual é a importância das boas práticas de codificação em JavaScript? Como elas podem impactar na legibilidade e manutenção do código?

2. O que a barra de elementos faz nas DevTools?

3. Como você abriria a aba “Fontes” diretamente nas DevTools?

4. Qual é a função da aba “Console” nas DevTools?

5. Por que a manipulação de exceções é importante em JavaScript? Como o bloco **try-catch** pode ser empregado para lidar com erros?

6. Explique o uso do bloco **finally** e diga em que cenários ele pode ser útil durante a manipulação de exceções.

7. Como criar e utilizar uma exceção personalizada em JavaScript? Forneça um exemplo prático dessa abordagem.

8. Discuta a importância da depuração detalhada, incluindo mensagens de erro específicas e informações relevantes, para facilitar a identificação de problemas.

9. Como as ferramentas de depuração do navegador, como o Console e as guias de Origens (*Sources*), podem ser eficientemente utilizadas para depurar código JavaScript?

10. Por que a padronização de código e a adoção de convenções de nomenclatura são consideradas boas práticas? Como isso contribui para um código mais consistente e fácil de entender?

Referências

FORBELLONE, André Luiz Villar; EBERSPÄCHER, Henri Frederico. *Lógica de Programação*: A Construção de Algoritmos e Estruturas de Dados. 4. ed. São Paulo: Pearson Prentice Hall, 2019.

LUTZ, Mark. *Aprendendo Python*: Do Iniciante ao Profissional. São Paulo: Novatec, 2014.

MANZANO, José Augusto N. G.; OLIVEIRA, Jayr Figueiredo de. *Algoritmos*: Lógica para Desenvolvimento de Programação de Computadores. 28. ed. São Paulo: Érica, 2018.

PUGA, Sandra; RISSETTI, Gerson. *Lógica de Programação e Estrutura de Dados*: Com Aplicações em Java. Rio de Janeiro: LTC, 2015.

RAMALHO, Luciano. *Python Fluente*: Programação Clara, Concisa e Eficaz. São Paulo: Novatec, 2016.

ZIVIANI, Nivio. *Projeto de Algoritmos com Implementações em Pascal e C*. 4. ed. São Paulo: Cengage Learning, 2015.