



JavaScript

APOSTILA



Créditos

Organização e Produção

Davi Domingues

Douglas de Oliveira

Fernando Esquírio Torres

Rafael Lopes dos Santos



É proibida a duplicação ou reprodução deste volume, no todo ou em parte, em quaisquer formas ou por quaisquer meios (eletrônicos, mecânico, gravação, fotocópia, distribuição pela internet ou outros), sem permissão prévia do Instituto da Oportunidade Social.

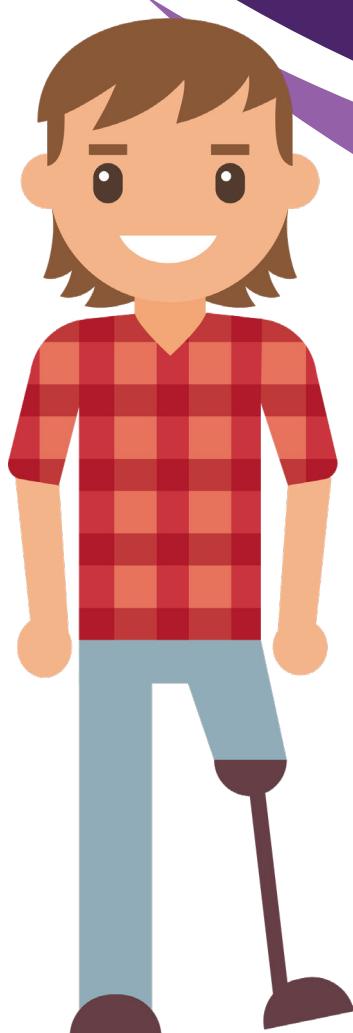
São Paulo
2022

Atenção

Em caso de dúvidas, sugestões ou reclamações. Entre em contato com a equipe de conteúdo.

educacional@ios.org.br

ESSA É SUA APOSTILA!



Ela vai te acompanhar durante todo o período do curso.

Cuide dela com carinho e responsabilidade.

Para que ela chegassem toda cheia de estilo do jeito que você está vendo, muita gente quebrou a cabeça para te entregar a melhor experiência de aprendizagem.

Ótimos estudos!

SUMÁRIO

Introdução ao JavaScript	6
Console, variáveis e operadores	20
Strings.....	37
Estruturas condicionais	50
Laços de repetição	65
Array.....	80
Array de Objetos	93
Objetos, Funções e Eventos.....	104
Métodos de alto nível para manipular arrays.....	116
Classes e Funções.....	127
JavaScript HTML DOM – Parte 01	138
JavaScript HTML DOM – Parte 02	149
JavaScript HTML DOM – Parte 03	159
JavaScript HTML DOM – Parte 04	172
JavaScript HTML DOM – Parte 05	183
Projeto Book List	192
Filtro em uma Lista	202
Image Slider.....	210
Modal com o JavaScript.....	222
Classificação com estrelas no JavaScript	234
API com JavaScript - Parte 01	243

API com JavaScript - Parte 02	256
Busca CEP API.....	262
Rick and Morty API	272
Busca de Estados Brasileiros	281
Criando uma API – Parte 01	293
Criando uma API – Parte 02	305



Introdução ao JavaScript

Os objetivos desta aula são:

- Conhecer a linguagem de programação JavaScript;
- Familiarizar-se com as extensões VsCode voltadas ao Javascript;
- Iniciar a compreensão da sintaxe da linguagem;
- Criar sua primeira página utilizando JavaScript.

Bons estudos!

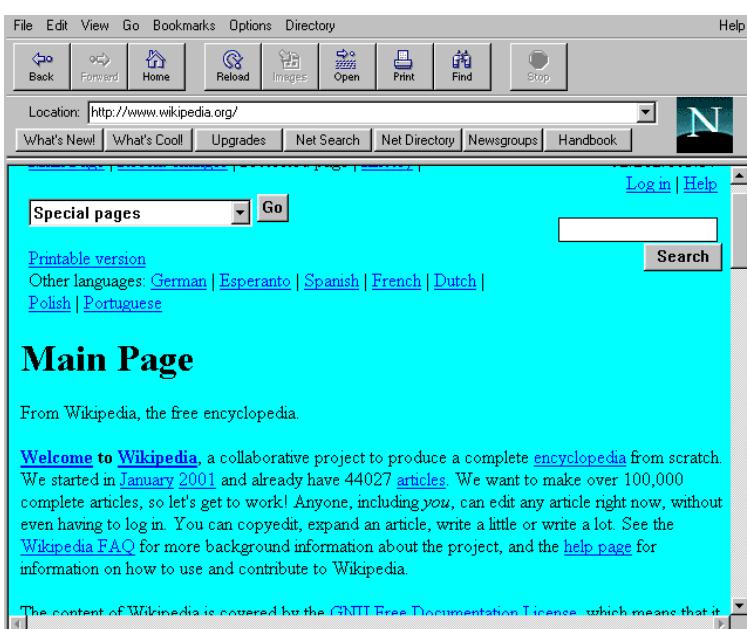
Introdução ao JavaScript



Brendan Eich

JavaScript (ou apenas **JS**) é uma linguagem de programação, que está de acordo com a especificação **ECMAScript**. Ela foi criada na década de 90 por **Brendan Eich** a serviço da **Netscape**.

Essa década foi um período de revolução, pois os browsers ainda eram estáticos. O navegador mais popular dessa época era o **Mosaic**, da **NCSA**.



Netscape Navigator

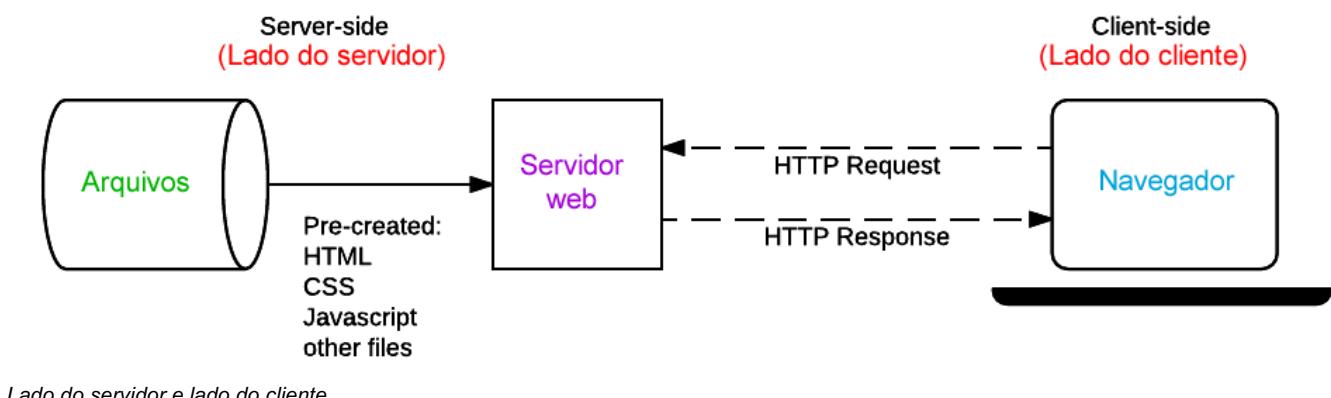
A Netscape chegou à conclusão que a web teria que se tornar mais dinâmica, pois o Navigator tinha sempre que fazer uma requisição ao servidor para obter uma resposta no navegador. Em 1995, a Netscape contratou Brendan Eich para criar uma linguagem que proporcionasse isso.

O JS é uma **linguagem de alto nível interpretada com tipagem dinâmica fraca e mutiparadigma** (protótipos, orientado a objeto, imperativo e, funcional).

O **JS**, além do **HTML** e do **CSS**, é uma das **tecnologias bases da World Wide Web**. E, atualmente é uma das linguagens **mais utilizadas do lado do cliente**, mas também pode ser utilizada do lado do **servidor** por meio de ambiente como o **NodeJS**.

Lado do cliente e lado do servidor

Programação baseada na internet possui **dois principais lados: programação do lado do servidor e programação do lado do cliente**. Na programação do lado do **servidor**, o **código executa em um servidor web**. E nesse caso, os navegadores **comunicam-se** com web servers utilizando **requisições HTTP (HyperText Transfer Protocol)**. Por exemplo, quando você clica em um link em uma página da web, seja para enviar um formulário ou para fazer uma pesquisa, uma **HTTP request** (solicitação HTTP) é enviada do seu navegador para o servidor de destino. Na programação do lado do cliente, os programas são executados no computador do usuário utilizando scripts, que são carregados juntos com os arquivos **HTML** e **CSS**.



Lado do servidor e lado do cliente

Características

O JS é uma linguagem **imperativa** e **estruturada**, que suporta os elementos de sintaxe de programação estruturada da linguagem C como, por exemplo, **if**, **while**, **switch**, etc. O JS também possui **tipagem dinâmica** e **baseada em objetos**. Objetos JavaScript são **arrays associativos**, potencializados com um **protótipo** e cada chave **fornecida o nome para uma propriedade de objeto**. Propriedades e seus valores podem ser adicionadas, mudadas, ou deletadas em tempo de execução, veremos mais sobre objetos posteriormente.

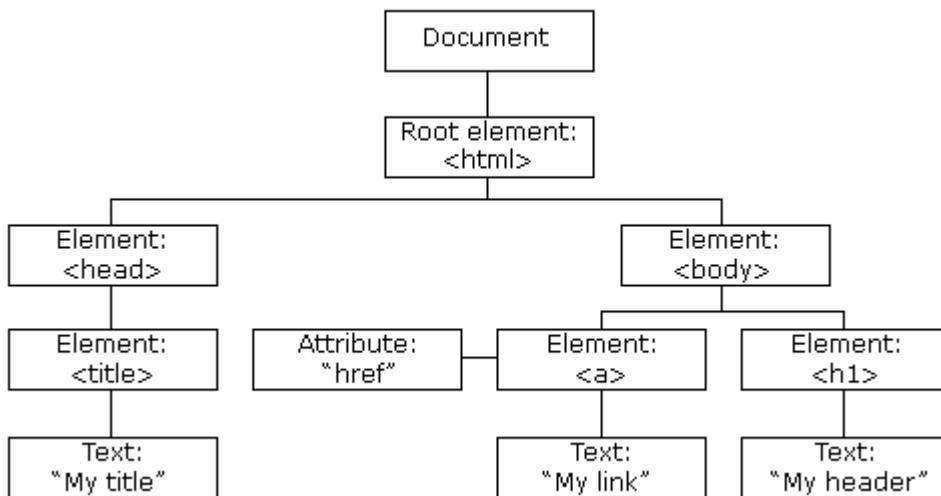
Vanilla JavaScript

Vanilla JavaScript ou **Vanilla JS** se referem ao **JavaScript desenvolvido puramente, sem o suporte de qualquer estrutura ou biblioteca adicional**. Scripts escritos em **Vanilla JS** são códigos **JavaScript simples**. O que vamos aprender primeiramente aqui é **programar códigos em Vanilla JS** e depois aprenderemos como utilizar uma **biblioteca/framework** para auxiliar no desenvolvimento das aplicações web.

JavaScript e HTML DOM

O **HTML DOM (Document Object Model)** permite o JavaScript **acessar** e **modificar** todos os **elementos HTML em um documento** (página web). Quando uma página web é carregada, o

navegador cria o DOM da página com a **estrutura de elementos e a árvore de objetos dessa página**. O JS pode adicionar, **alterar** ou **remover os elementos**, os **atributos** e **estilos CSS** da página, bem criar ou **reagir a eventos HTML** da página.



Exemplo de uma árvore de objetos do DOM.

Extensões do VS Code

Seguem algumas **sugestões** de **extensão** para você ter no seu VS Code, que podem te ajudar no desenvolvimento de códigos em JavaScript.

A extensão **ES Lint** analisa estaticamente seu código para encontrar problemas rapidamente e muitos problemas encontrados pelo ESLint podem ser corrigidos automaticamente.



A extensão **JavaScript (ES6) code snippets** contém **snippets** para a **sintaxe ES6** do **JavaScript**. Por exemplo, o snippet **imp** importa módulos inteiros (ex.: **import fs from 'fs'**);



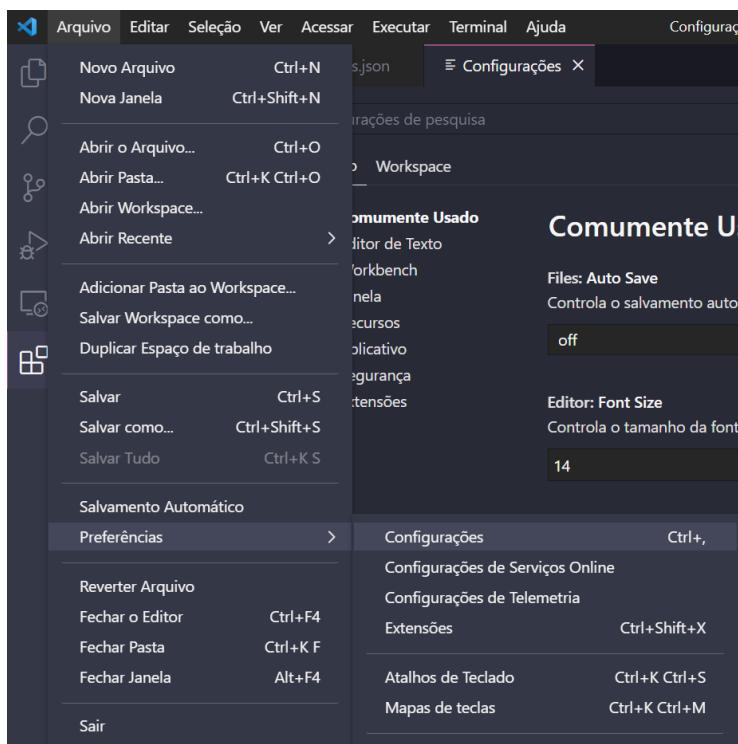
A extensão **Bracket Pair Colorizer 2** colore pares de parênteses, colchetes ou chaves com a mesma cor, facilitando a visualização de blocos de comando.



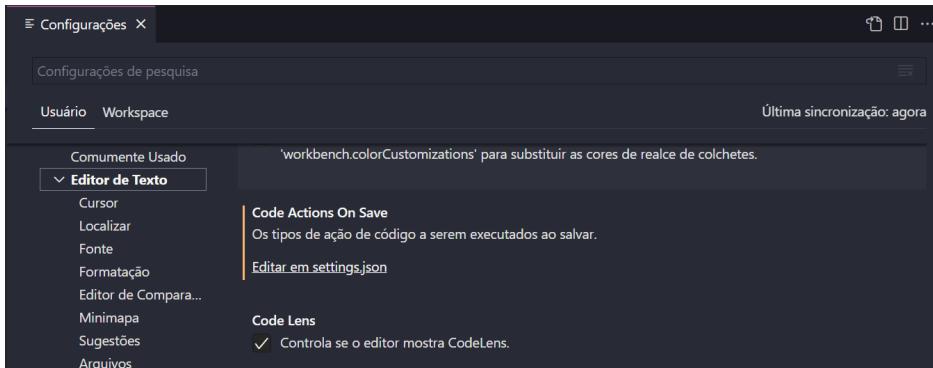
Ao instalar a extensão **ESLint**, você precisa configurá-las no seu VS Code para que a extensão possa realizar as organizações automaticamente.

Vamos aos passos para fazer as configurações:

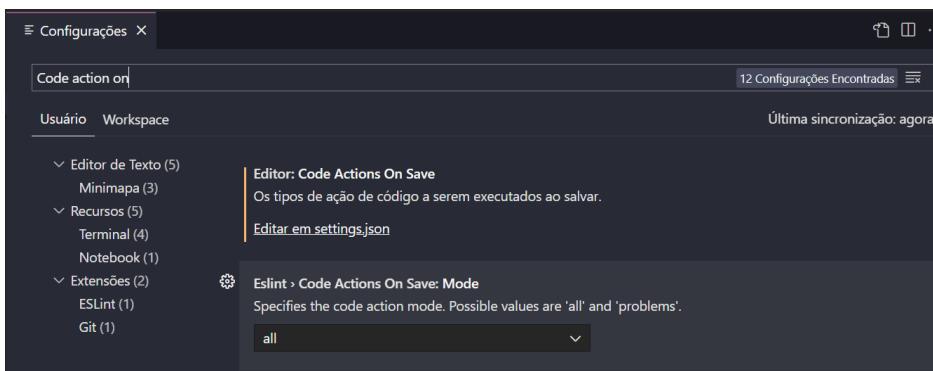
Você deve abrir o arquivo **settings.json** da sua IDE VS Code. Acesse o **Menu Arquivo → Preferências → Configurações** ou utilize o atalho do teclado **Ctrl+,**



Na aba configurações que será aberta, role até a opção **Code Actions On Save** e clique em **Editar em settings.json**.



Você pode também buscar por **Code Actions On Save** na barra de **Configurações de pesquisa**.



O arquivo **settings.json** irá ser aberto. **Insira o trecho de configuração das extensões Prettier – Code formatter e ESLint.**

Observação: Não se esqueça de incluir `,` no final da linha anterior.

```
"eslint.validate": ["javascript", "typescript"],  
"prettier.tabWidth": 4,  
"prettier.singleQuote": true,  
"prettier.bracketSpacing": true
```

Agora o seu código em **JavaScript** será **formatado** e **verificado** sempre que você o salvar.

Comentários no JavaScript

Comentários são **anotações inseridas no código fonte com o objetivo de descrever alguma lógica**, instrução ou um lembrete TO-DO. Por exemplo, você pode usar comentários para: **lemburar algo importante quando o código foi desenvolvido, criar seções para organização e ainda para criar um cabeçalho do código**. **Comentários são ignorados pelo compilador na verificação da sintaxe do código**. Os comentários mais comuns em JavaScript são:

- Comentário de linha, que é iniciado por //

```
// Texto do comentário
```
- Comentário de bloco, que é iniciado por /* e finalizado por */

```
/*
Esse é um comentário tradicional. Ele
pode ser dividido em várias linhas
*/
```



Importante! Comentários em uma linguagem de programação é muito importante e deve ser usado com consciência. Quando estamos ensinando uma nova linguagem de programação inserimos muitos comentários para que o aluno possa entender o comando ou instrução que está executada, mas no ambiente profissional, você deve primeiro ver a política de documentação da empresa e códigos exemplos do sistema que você irá trabalhar.

Espaços em brancos

O espaço em branco **geralmente é insignificante**, mas ocasionalmente é **necessário usar o espaço em branco para separar sequências de caracteres** que, de outra forma, seriam combinadas em um único token. Por exemplo:

```
let num = 3;
```

O espaço em branco entre a palavra reservada **let** e o nome da variável **num** é **necessário e não deve ser removido**, mas ou outros espaços em branco são opcionais.

```
let num=3;
```

Nomes

Os nomes de variáveis ou funções em JavaScript podem conter letras, dígitos ou underline e não pode coincidir com uma das palavras reservadas da linguagem.

Palavras reservadas em JavaScript

abstract	arguments	boolean	break	byte
case	catch	char	class*	const
continue	debugger	default	delete	do
double	else	enum*	eval	export*
extends*	false	final	finally	float
for	function	goto	if	implements
import*	in	instanceof	int	interface
let	long	native	new	null
package	private	protected	public	return
short	static	super*	switch	synchronized
this	throw	throws	transient	true
try	typeof	var	void	volatile
while	with	yield		

Palavras reservadas em JavaScript.

Números

O **JavaScript** tem um **único tipo de número**. Internamente, é representado como **ponto flutuante de 64 bits**, o mesmo que o **double do Java**. Portanto, **em JS, não há diferença entre 1 e 1.0, esses números são interpretados como mesmo valor**. Tudo que você precisa saber sobre um número é que ele é um número. Uma grande classe de erros de tipo numérico é evitada.

Strings

Strings são **sequência de caracteres**, que no JavaScript, devem ser envolvidas utilizando **aspas simples ou aspas duplas**, mesmo se elas contêm zero ou mais caracteres. Todos os exemplos mostrados abaixo são válidos para serem usados na linguagem JavaScript.

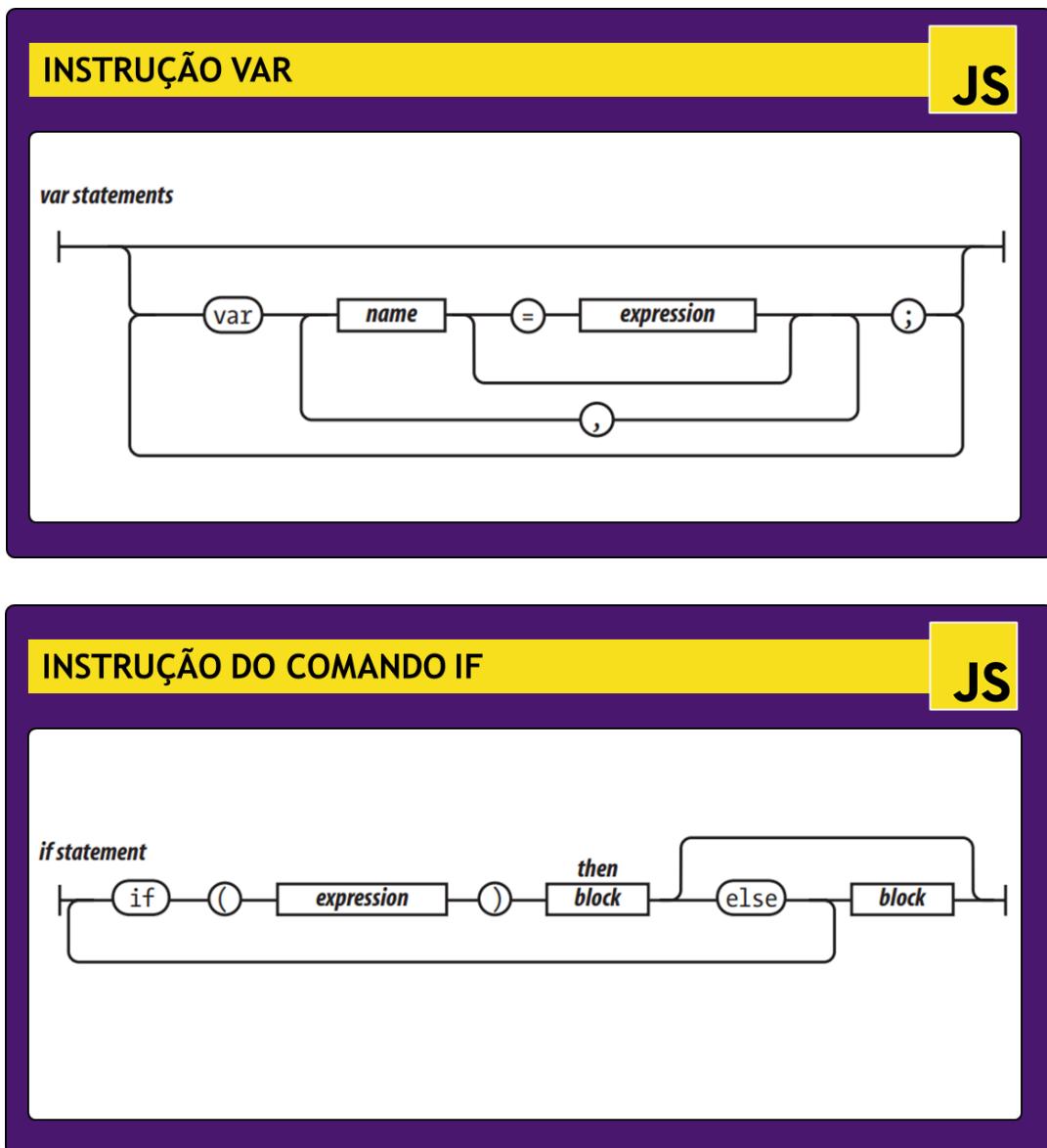
```
let nome = "Irmão do Jorel";
let dados = "";
const frutas = ["maçã", "manga", "pêra"];
```

Se você fez a configuração da extensão Preittier como mostrada anteriormente, a configuração "**prettier.singleQuote": true**" irá padronizar o uso de aspas simples em toda string no seu código JavaScript.

```
let nome = 'Irmão do Jorel';
let dados = '';
const frutas = ['maçã', 'manga', 'pêra'];
```

Instruções

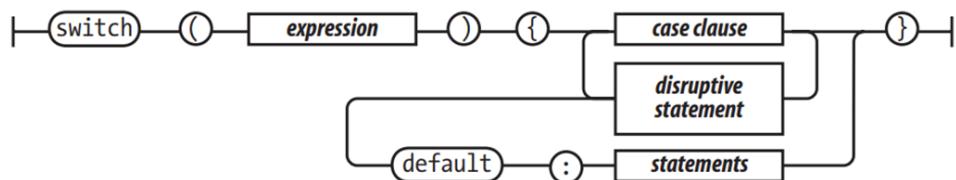
Uma **unidade de compilação** contém um **conjunto de instruções executáveis**. Em navegadores web, cada marcação `<script>` fornece uma unidade de compilação que é **compilada e executada imediatamente**. Desse modo o JS possui uma **sintaxe específica** para **interpretar corretamente as suas instruções**. **Você deve ficar atento a essas sintaxes para não inserir códigos errados no seu script**. Abaixo seguem algumas sintaxes usada pelo JavaScript:



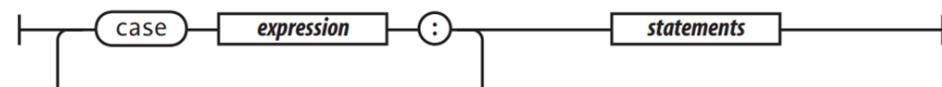
INSTRUÇÃO DO COMANDO SWITCH-CASE

JS

switch statement



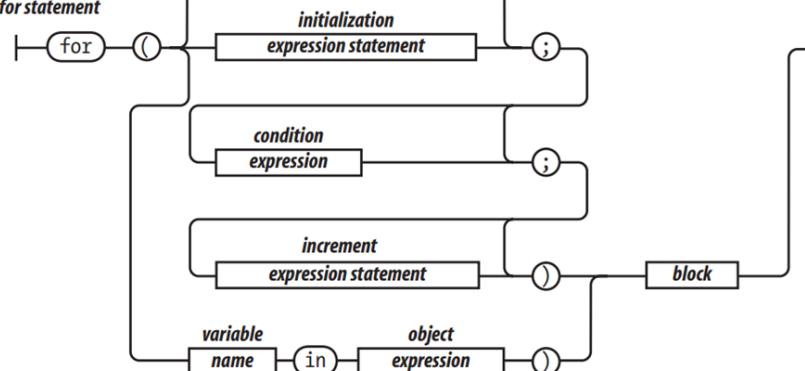
case clause



INSTRUÇÃO DO COMANDO FOR

JS

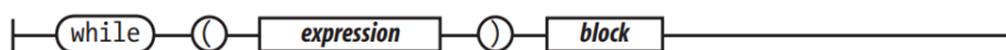
for statement



INSTRUÇÃO DO COMANDO WHILE

JS

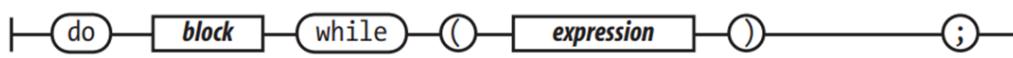
while statement



INSTRUÇÃO DO COMANDO DO-WHILE

JS

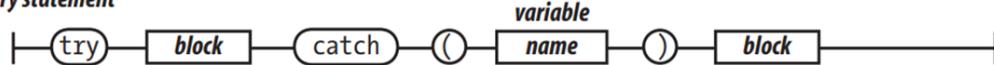
do statement



INSTRUÇÃO DO COMANDO TRY-CATCH

JS

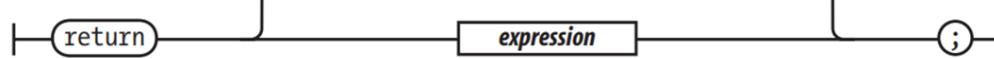
try statement



INSTRUÇÃO DO COMANDO RETURN

JS

return statement



Trabalhar com o elemento <script>

Um código JavaScript é anexado ao arquivo HTML através da marcação <script>. O elemento <script> exige marcações de **abertura** e **fechamento**, sendo assim, você deve inserir a marcação de abertura <script>, que indica o **início do código JavaScript**, e, também, deve inserir a marcação de fechamento </script>, que indica o **final do código JavaScript**. Você pode inserir o código JavaScript no documento HTML de **duas formas**:

Embutir o código **JavaScript** no documento HTML através do trecho de código:

```
<script>
    // Aqui virá o código JavaScript
</script>
```

Vincular um **arquivo de script externo** com o código **JavaScript** através do atributo src:

```
<script src="arquivo-javascript.js"></script>
```

Daremos alguns exemplos de como usar o **código JavaScript embutido no documento HTML**, mas **é interessante você já se acostumar a criar um arquivo externo para deixar separado as duas partes, como é feito com o estilo CSS**.

Mensagem de alerta

O método **alert()** mostra uma caixa de alerta com uma mensagem específica e um botão de OK. Por exemplo:

```
alert('Vamos aprender JavaScript?');
```



Vamos praticar!

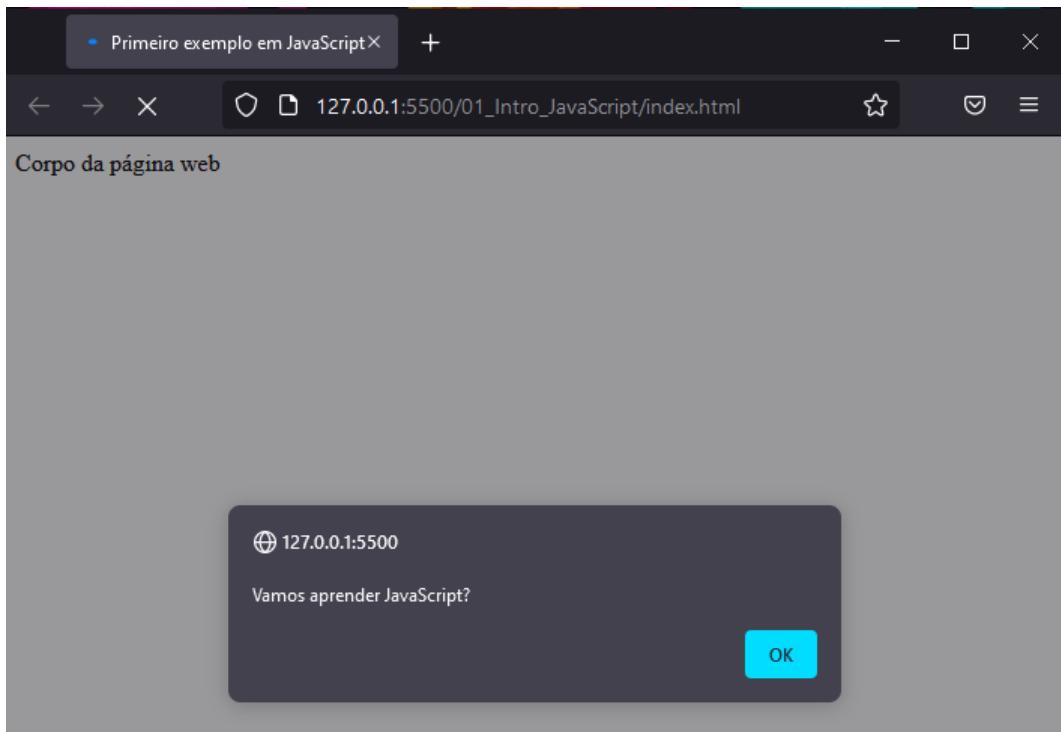
Vamos criar nossa primeira aplicação web com o comando mais simples em JavaScript utilizando o método **alert()**. Esse método é o equivalente ao “**Olá Mundo**” em outras linguagens de programação. Siga os passos para criar a nossa primeira aplicação:

1. Abra o **VS Code** e **escolha um diretório de trabalho para o seu projeto**.
2. **Crie um diretório para seu projeto com o nome representativo**, por exemplo, **Intro_JavaScript**.
3. Crie um arquivo dentro do diretório do projeto com o nome **index.html**.
4. Insira o seguinte código no seu arquivo index.html.

```
<!DOCTYPE html>
<html lang="pt-br">
  <head>
    <meta charset="UTF-8" />
    <title>Primeiro exemplo em JavaScript</title>
  </head>
  <body>
    Corpo da página web
    <script>
      alert('Vamos aprender JavaScript?');
    </script>
  </body>
</html>
```

Esse código mostra a marcação <**script**>, que contém uma instrução em JavaScript. A instrução é a chamada do método **alert()** com a string “**Vamos aprender JavaScript?**”.

Salve (**Ctrl+S**) o arquivo e clique no botão  **Go Live** da extensão Live Server. Assim, você poderá visualizar a página index.html exibindo a mensagem de alerta.



Glossário

Aqui estão algumas definições e conceitos que você precisa conhecer para entender melhor (Fonte: Wikipedia e outras fontes da internet).

Document Object Model (DOM): é uma convenção multiplataforma e independente de linguagem de programação, fiscalizada pelo entidade World Wide Web Consortium (W3C), para representação e interação com objetos em documentos HTML, XHTML e, XML.[1][2][3] Onde os elementos/nós de cada documento são organizados em uma estrutura de árvore, chamada de Árvore DOM, que endereça e manipula via uso de funções/métodos (interface pública) sobre os objetos, especificada de acordo com a interface de programação de aplicações (API) utilizada, que oferece uma maneira padrão de se acessar cada elemento de um documento, criando páginas altamente dinâmicas.

Linguagem de script ou scripting: é uma linguagem de programação que suporta scripts, programas escritos para um sistema de tempo de execução especial que automatiza a execução de tarefas que seriam executadas, uma de cada vez, por um operador humano.

Snippets: em programação é um termo usado para indicar uma pequena região de código reutilizável. Esses trechos fornecem uma maneira fácil inserir automaticamente códigos ou funções comumente usados em um código.

Paradigma de programação: é a forma de se classificar determinada linguagem de programação com base em seu funcionamento e sua estruturação. Alguns exemplos de paradigmas de programação são a Programação orientada a objetos, Programação Estruturada e a Programação Imperativa.

Tipagem: um sistema de tipos é um conjunto de regras que atribuem uma propriedade chamada de tipo para as várias construções - tais como variáveis, expressões, funções ou módulos - que um programa de computador é composto.

Tipagem dinâmica: é uma característica de determinadas linguagens de programação, que não exigem declarações de tipos de dados, pois são capazes de escolher que tipo utilizar dinamicamente para cada variável, podendo alterá-lo durante a compilação ou a execução do programa.

Tipagem fraca: Em linguagens fracamente tipadas, as variáveis declaradas, poderão intercambiar seus tipos a qualquer momento, ou seja, poderão receber valores de tipos de dados diferentes e fazer operações entre eles sem a necessidade de uma conversão (casting) explícita do tipo.



Console, variáveis e operadores

Os objetivos desta aula são:

- Compreender o uso e a concatenação de strings ;
- Conhecer os diferentes tipos de operadores utilizados em JavaScript;
- Reconhecer diferentes tipos de variáveis e aplicar o objeto console.

Valores e tipos de dados

Dentro do mundo do computador, **existem apenas dados**. Você pode **ler dados, modificar dados, criar dados**, mas o que **não são dados não pode ser mencionado**. Todos esses dados são **armazenados como longas sequências de bits** e, portanto, são fundamentalmente semelhantes.

Valores

Os valores em um computador são quantidade de bits que representam uma informação.

```
1           // Valor do tipo numérico  
"Homer"    // Valor do tipo string (sequência de caracteres)
```

Números

Valores podem ser do **tipo numérico**, que representam **valores constantes**. Como foi dito, **o JavaScript tem um único tipo de número**. Internamente, é representado como **ponto flutuante de 64 bits**. Você deve separar a parte fracionária de um número utilizando o ponto:

```
9.81
```

Números especiais

Existem **três valores especiais** em **JavaScript** que **são considerados números**, mas não se comportam **como números normais**. Os dois primeiros são **Infinity** e **-Infinity**, que representam os **infinitos positivos e negativos**. A expressão **Infinity - 1** ainda é infinito e assim por diante. O terceiro é o **NaN** significa “**Not a Number**” (“**não é um número**”), embora seja um valor do tipo numérico. Você obterá este resultado quando, por exemplo, tentar calcular **0/0 (zero dividido por zero)**.

Strings

String é um outro tipo de dado, que é usado para representar texto, e devem ser envolvidas utilizando aspas simples ou aspas duplas.

```
"Presentemente eu posso me considerar um sujeito de sorte"  
"Porque apesar de muito moço, me sinto são e salvo e forte"  
"E tenho comigo pensado: Deus é brasileiro e anda do meu lado"  
"E assim já não posso sofrer no ano passado"
```

Strings não podem ser divididas, multiplicadas ou subtraídas, mas o operador **+** pode ser usado para concatenar (juntar, unir) duas ou mais strings. Por exemplo:

Concatenando strings

```
"Instituto" + ' ' + 'da' + " " + "Oportunidade" + ' ' + 'Social'
```

Equivalente

```
Instituto da Oportunidade Social
```

Observe, que você pode utilizar aspas simples e duplas quando quiser dentro da string. Você pode inserir caracteres especiais de escape na string, por exemplo:

```
"Essa é a primeira linha\nE essa é a segunda linha"
```

A sua string será interpretada como:

```
Essa é a primeira linha
E essa é a segunda linha
```

O **caractere de escape** (**\n**) indica um **nova linha e retorno do cursor para o início do parágrafo**. A barra invertida (****) é chamada de **caractere de escape**. Quando uma barra invertida é encontrada em uma string de caracteres, o interpretador examina o próximo caractere e o combina com a barra invertida para formar uma sequência de escape.

Código escape	Resultado
\n	Nova linha e posiciona o cursor no início da nova linha do console.
\t	Tabulação horizontal. Move o cursor do console horizontalmente um espaço de tabulação.
\v	Tabulação vertical. Move o cursor do console verticalmente um espaço de tabulação.
\	Barra invertida. Insere uma barra invertida na string.
\'	Aspa simples. Insere uma aspa simples na string.
\"	Aspas duplas. Insere uma aspas dupla na string.
\b	Backspace. Retorna o cursor uma posição para trás.

Código de escape comuns no JavaScript.

Valores booleanos

Muitas vezes é **útil** ter um **valor que distingue apenas duas possibilidades**, como **verdadeiro** e **falso**. Para isso, o JavaScript possui um tipo **booleano**, que possui apenas **dois valores**, **true**

(**verdadeiro**) e **false (falso)**). Existem dois tipos de operadores que retornam um resultado do tipo **booleano**: operadores de **comparação (relacionais)** e **operadores lógicos**.

Valores vazios

Existem **dois valores especiais**, escritos como **null (nulos)** e **undefined (indefinidos)**, que são usados para denotar a **ausência de um valor significativo**. Eles próprios **são valores**, mas **não contêm nenhuma informação**. Muitas operações na linguagem que não produzem um **valor significativo** resultam em **indefinidas simplesmente porque têm que produzir algum valor**.

A diferença de significado entre **undefined** e **null** é um acidente do design do JavaScript, e isso não importa na maioria das vezes. Nos casos em que você realmente precisa se preocupar com esses valores, recomendo tratá-los como geralmente intercambiáveis.

Conversão automática

JavaScript faz de tudo para **aceitar quase qualquer programa que você forneça**, até mesmo programas que fazem coisas estranhas. Por exemplo, você pode fazer **operações com tipos diferentes de dados**:

Operação	Resultado
<code>8 * null</code>	0
<code>"5" - 1</code>	4
<code>"5" + 1</code>	51
<code>"five" * 2</code>	NaN

Quando um operador é aplicado ao tipo "**errado**" de valor, o JavaScript **silenciosamente converte esse valor para o tipo de que precisa**, usando um conjunto de regras que muitas vezes não são o que você deseja ou espera. Isso é chamado de **coerção de tipo**.

Operadores

Os operadores especificam uma avaliação/ação a ser executada em um ou mais operandos e podem ser **aritméticos, relacionais** ou **lógicos**.

Operadores aritméticos

A principal coisa a fazer com os **números** é realizar **operações aritméticas**. Operações aritméticas como **adição** ou **multiplicação** tomam **dois valores numéricos** e produzem um **novo número a partir deles**. Por exemplo:

```
100 + 4 * 11
```

Os símbolos (+) e (*) são chamados de **operadores aritméticos**. A tabela abaixo mostra os **operadores aritméticos do JavaScript**.

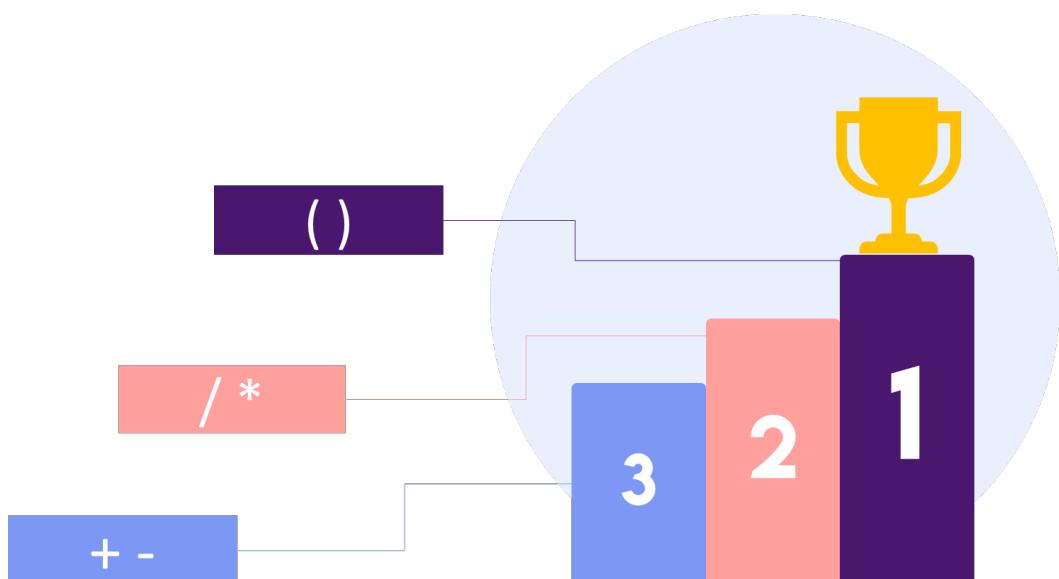
Categoria	Operador	Descrição
Operadores aritméticos	+	Adição
	-	Subtração
	*	Multiplicação
	**	Exponenciação
	/	Divisão
	%	Módulo (Resto da divisão inteira)
	++	Incremento
	--	Decremento

Operadores aritméticos do JavaScript.

Nesse exemplo, primeiro é realizada a multiplicação e não a adição por causa da precedência dos operadores.

```
100 + 4 * 11
```

A ordem de precedência dos operandos aritméticos é:



Desse modo a operação:

```
3 * 5 + 2
```

O programa primeiro faz a multiplicação e com o resultado dessa multiplicação realiza a soma. Os parênteses têm a maior precedência, portanto a operação:

```
3 * (5 + 2)
```

Primeiro é realizado a soma dentro dos parênteses e com o resultado dessa soma realiza a multiplicação.

Operadores de comparação (relacionais)

Os **operadores relacionais** são utilizados na realização de **comparação entre valores**. Por exemplo:

Operação	Resultado
$3 > 2$	true
$3 < 2$	false

Os principais operadores de comparação são mostrados na tabela abaixo.

Categoria	Operador	Descrição
Operadores de comparação	$==$	Valores iguais
	$====$	Valores e tipos iguais
	$!=$	Diferente
	$<$	Menor que
	\leq	Menor ou igual
	$>$	Maior que
	\geq	Maior ou igual

Operadores de comparação do JavaScript.

O JavaScript utiliza o operador **triplo de igualdade**, para garantir a **comparação dos valores executando a conversão de tipos**. Sendo assim no JavaScript:

Operação	Resultado
$2 == "2"$	True
$2 === "2"$	False

Operadores lógicos

Operadores lógicos são usados em programação para concatenar expressões que estabelecem uma relação de comparação entre valores. Os principais operadores lógicos são mostrados abaixo:

Categoria	Operador	Descrição
Operadores Lógicos	&&	Lógica “and” ou “e”, que retorna verdadeiro se todos os operandos forem verdadeiros.
		Lógica “or” ou “ou”, que retorna verdadeiro se pelo menos um operando for verdadeiro.
	!	Lógica “not” ou “não”, que inverte o valor lógico se é verdadeiro, retorna falso e se é falso retorna verdadeiro.

Operadores lógicos do JavaScript.

Operador de atribuição

O operador de atribuição (=) permite atribuir um **valor** a **uma variável**, por exemplo. O JavaScript permite utilizar uma forma contraída do operador de atribuição. A tabela a seguir, mostra diversos exemplos do **operador de atribuição** com os **operadores**, por exemplo: a instrução `x += 2` é equivalente a `x = x + 2`, ou seja, não precisamos repetir o `x` novamente.

Operador	Exemplo	Equivalente
=	<code>x = y</code>	<code>x = y</code>
+=	<code>x += y</code>	<code>x = x + y</code>
-=	<code>x -= y</code>	<code>x = x - y</code>
*=	<code>x *= y</code>	<code>x = x * y</code>
/=	<code>x /= y</code>	<code>x = x / y</code>
%=	<code>x %= y</code>	<code>x = x % y</code>
**=	<code>x **= y</code>	<code>x = x ** y</code>
<=>	<code>x <=> y</code>	<code>x = x <> y</code>
>=>	<code>x >=> y</code>	<code>x = x >> y</code>
>>>=	<code>x >>>= y</code>	<code>x = x >>> y</code>
&=	<code>x &= y</code>	<code>x = x & y</code>
^=	<code>x ^= y</code>	<code>x = x ^ y</code>
=	<code>x = y</code>	<code>x = x y</code>
**=	<code>x **= y</code>	<code>x = x ** y</code>

Operador de atribuição.

Operadores unários

Nem todos os operadores são símbolos. Alguns são **escritos como palavras**. Um exemplo é o operador **typeof**, que produz um **valor de string nomeando o tipo do valor que você forneceu**. Por exemplo:

```
console.log(typeof 4.5)
// → number
console.log(typeof "x")
// → string
```

O **typeof** de **4.5** retornou que ele é um tipo **numérico** e de “**x**” retornou uma **string**.

Outro operador **unário** é o símbolo de **-** (menos), que quando usado em um número transforma o valor de **positivo** para **negativo**. Por exemplo:

-8

Variáveis

Os **dados de um programa** são **armazenados em variáveis**. As variáveis são ferramentas indispensáveis na programação, são nelas que colocamos valores para podermos trabalhar com esses dados posteriormente.



Importante! *O nome de variáveis ou constantes deve começar por letras ou underline (_) e pode conter números e não pode conter caracteres especiais. O nome também não pode ser uma palavra-chave ou palavra reservada.*

Nomes válidos:

- x, y, ola_01, _teste

Nomes inválidos:

- 12_teste, nome pessoa, 1xx3

Programadores geralmente usam somente letras minúsculas para os nomes de variáveis, mas isso é uma convenção utilizada e não uma regra.

No JavaScript, podemos declarar variáveis de três maneiras:

- Usando a palavra reservada **var**
- Usando a palavra reservada **let**
- Usando a palavra reservada **const**

Var

A instrução **var** declara uma variável no escopo de uma função ou no escopo global e é opcional inicializar o seu valor.

```
var x = 1; // Variável numerica
var nome = 'Homer'; // Variável string
var teste; // Variável não inicializada
```

Let

A palavra-chave **let** foi introduzida na ES6 em 2015. Variáveis definidas com **let não podem ser redeclaradas**, ou seja, **você não pode declarar novamente uma variável com o mesmo nome**. Por isso, nos programas mais recentes declarar variáveis utilizando **let** é cada vez mais comum.

O let não permite isso

```
let x = 'John Doe';
let x = 0;
```

SyntaxError: Identifier 'a' has already been declared

```
var x = "John Doe";
var x = 0;
```

Sem problemas, 😊!

A instrução **let** também permite que você **declare uma variável no escopo de bloco**, ou seja, em uma **região delimitada pela abertura e fechamento de chaves {}**. Por exemplo:

```
{
  let z = 10;
  // Comandos
}
```

Const

A palavra-chave **const** foi também introduzida na ES6 em 2015. **Variáveis definidas com const não podem ser redeclaradas e não podem ter seu valor alterado**, ou seja, **você não pode declarar novamente uma variável com o mesmo nome e uma vez inicializada o seu valor será o mesmo até o fim do programa**.

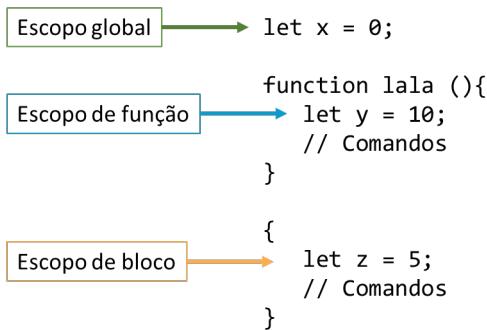
```
const PI = 3.141592653589793;
PI = 3.14; // Isso produzirá um erro
PI = PI + 10; // Isso também produzirá um erro
```



Dica!

Escopo de Variáveis: O escopo é o conjunto de regras que determinam o uso e a validade de variáveis nas diversas partes do programa. Um escopo define uma região do programa definida pela abertura e fechamento de chaves {}. JavaScript permite criar variáveis em três escopos: *global*, *de função* e *de bloco*. O escopo de função e de bloco são também chamados de escopo local. No caso, o escopo de uma variável define uma região do código onde a variável é visível, ou seja, ela pode ser acessada.

Por exemplo: a variável `x` foi declarada globalmente e pode ser acessada ou modificada em qualquer lugar do código, pois não há um escopo a delimitando. Então ela pode ser acessada ou modificada pela função ou pelo bloco normalmente. Já a variável `y` está definida localmente no escopo da função `lala` e só vai existir dentro do escopo dessa função, não podendo ser acessada fora da função. O mesmo acontece com a variável `z` só pode ser acessada no escopo do bloco.



Strict Mode

Você pode “**falar**” para o **interpretador do JavaScript** que você quer usar o **strict mode**. O **strict mode** indica que **você não pode usar nenhuma variável sem a devida declaração**. Para ativar esse modo, você deve colocar a string “**use strict**” no início do arquivo que você irá colocar seu código JavaScript. Vejamos o exemplo:

```
'use strict';
x = 3.14; // Isso gerará erro, pois a variável x não foi declarada
```

Objeto console

O objeto `console` fornece acesso ao `console` (terminal) de *debugging* do navegador. O `console` possui diversos métodos, mas vamos aprender com calma quatro métodos do objeto `console`:

Método	Descrição
<code>clear()</code>	Limpa o <code>console</code> .
<code>error()</code>	Envia uma mensagem de erro no <code>console</code> .
<code>log()</code>	Envia uma mensagem no <code>console</code> .
<code>warn()</code>	Envia uma mensagem de aviso no <code>console</code> .

Utilizar o **objeto console** para fazer **debug no seu código** e **visualizar as coisas acontecendo é melhor e mais prático do que usar a mensagem de alerta**. Por isso, usaremos esse objeto constantemente nos nossos códigos.

Importante! Não se preocupe no momento em saber o que é um objeto em JavaScript. Esse conceito será abordado mais adiante com mais detalhes.



Vamos praticar!

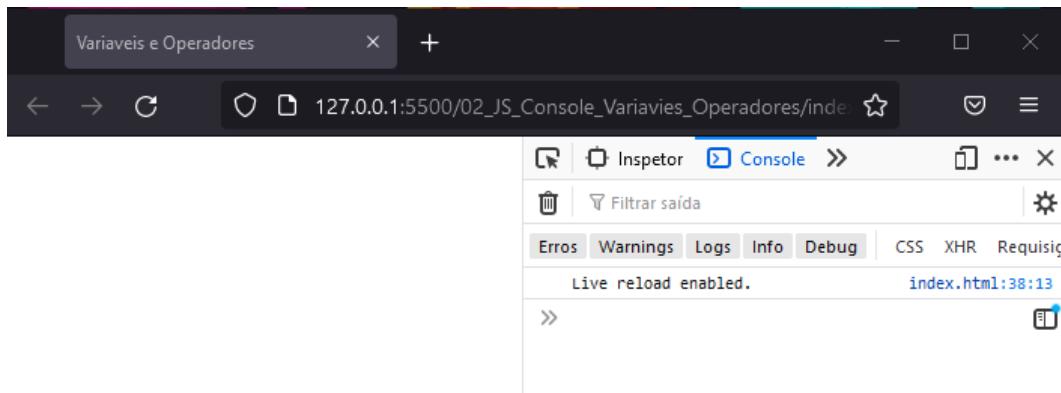
Vamos criar um projeto para testar toda teoria que vimos nesse tema. e. Siga os passos para criar o projeto:

1. Abra o **VS Code** e escolha um **diretório de trabalho** para o seu projeto.
2. Crie um diretório para seu projeto com o nome representativo, por exemplo, **JS_Console_Variavies_Operadores**.
3. Crie um arquivo dentro do diretório do projeto com o nome **index.html**.
4. Insira o seguinte código no seu arquivo **index.html**.

```
<!DOCTYPE html>
<html lang="pt-br">
  <head>
    <meta charset="UTF-8" />
    <link rel="shortcut icon" href="#" />
    <title>Variaveis e Operadores</title>
  </head>
  <body>
    <script src="main.js"></script>
  </body>
</html>
```

Esse código mostra a marcação **<script>** sem nenhum código JavaScript entre a abertura e o fechamento da tag, apenas o atributo **src** com o valor **main.js**. Isso significa que **o código JavaScript está em um arquivo externo**. Portanto, temos que criar esse novo arquivo **main.js** dentro do diretório do projeto.

Abra o arquivo **index.html** e clique no botão **Go Live** da extensão **Live Server** e abra a **Ferramenta de Desenvolvedor** do seu navegador.



Importante! O Navegador web abre arquivos com a extensão .html, por isso devemos sempre voltar ao arquivo HTML da nossa aplicação antes de clicar para abrir a extensão **Live Server**. Se você tentar abrir a extensão com um o arquivo de extensão diferente, o navegador irá apresentar a hierarquia de arquivos e diretórios do projeto.

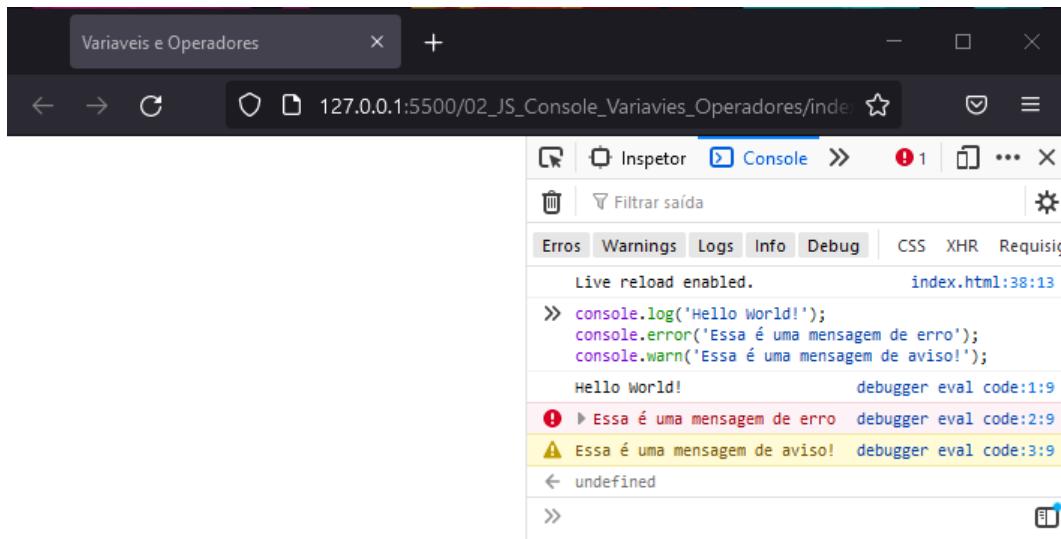
Primeiros comandos em JS

Você pode digitar os comandos do JavaScript diretamente no console ou inserir no arquivo **main.js** como iremos fazer nessa prática. Como já estamos com a aplicação aberta no navegador web (já iniciamos o **Live Server**). Então sempre que colocarmos uma instrução em JavaScript utilizando o objeto console, a mensagem será exibida no console do navegador. Vamos começar a programar, siga os passos para ira atualizando o seu arquivo **main.js**.

Os primeiros comandos serão mensagens utilizando os métodos **log()**, **error()** e **warn()** do objeto console. Insira os comandos no seu arquivo **main.js**.

```
console.log('Hello World!');  
console.error('Essa é uma mensagem de erro');  
console.warn('Essa é uma mensagem de aviso!');
```

Ao salvar o arquivo, você poderá visualizar as mensagens impressas no console.



```

Variáveis e Operadores      +
← → C  127.0.0.1:5500/02_JS_Console_Variavies_Operadores/index.html
Inspecto Console  ↗  1 | ...
Delete Filter saída
Erros Warnings Logs Info Debug CSS XHR Requisições
Live reload enabled. index.html:38:13
>> console.log('Hello World!');
console.error('Essa é uma mensagem de erro');
console.warn('Essa é uma mensagem de aviso!');
Hello World! debugger eval code:1:9
! Essa é uma mensagem de erro debugger eval code:2:9
⚠ Essa é uma mensagem de aviso! debugger eval code:3:9
← undefined
>>

```

Observe que cada tipo de mensagem é apresentada com **uma cor diferente** e a mensagem de **erro** ativa uma **badge na barra do console**, que indica que algo está errado na sua aplicação.

Agora vamos trabalhar com **variáveis**. O código abaixo declara **duas variáveis idade e nome**.

Vamos usar o método **clear()** sempre que mudarmos de assuntos, para que nosso console não fique cheio de mensagens.

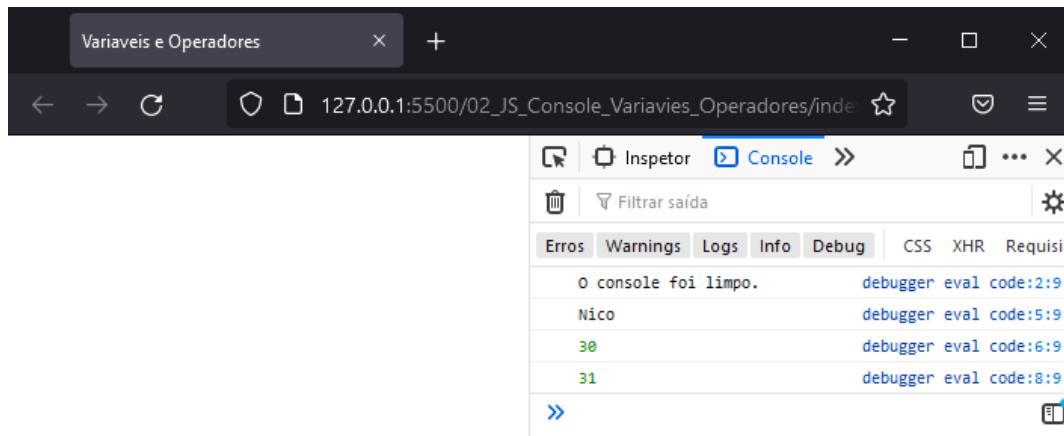
```
// Let, const
console.clear();
let idade = 30;
const nome = 'Nico';
console.log(nome);
console.log(idade);
idade = 31;
console.log(idade);
```

Declaramos uma **variável numérica** e, declaramos uma **string**. Imprimimos o valor das **variáveis separadamente**. Atualizamos o valor da variável **idade**, lembre-se não é possível alterar o valor da variável **nome**, pois ela foi declarada como **const**. O valor atualizado da variável **idade**.



Atenção: Observe que estamos colocando os números das linhas nos códigos, isso uma sugestão de onde você deve inserir o código mostrado e, também, para não precisarmos colocar o código completo. Você pode fazer como quiser, apagar o código anterior e substituir pelo novo código mostrado ou manter o código completo inserindo a instrução na linha sugerida.

Ao salvar o arquivo **main.js**, podemos ver o resultado impresso no console.



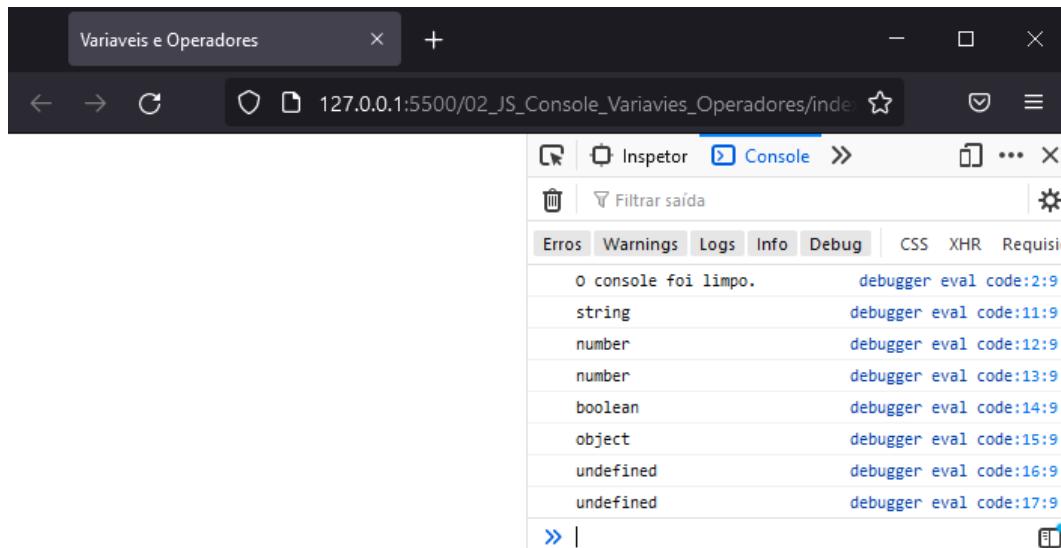
Agora vamos trabalhar com os **outros tipos de dados** e utilizar o operador **unário typeof** para verificar o tipo de cada uma das variáveis declaradas. Para isso insira o seguinte código no arquivo **main.js**

```
// String, Numbers, Boolean, null, undefined
console.clear();
const nome_pessoa = 'John';
const idade1 = 30;
const rating = 4.5;
const isCool = true;
const x = null;
const y = undefined;
let z;

console.log(typeof nome_pessoa); // Type of const
console.log(typeof idade1);
console.log(typeof rating);
console.log(typeof isCool);
console.log(typeof x);
console.log(typeof y);
console.log(typeof z);
```

Observe que tivemos que declarar **nome_pessoa**, pois uma variável **nome** já foi declarada anteriormente como **const** e não podemos usar variáveis com nomes iguais. O mesmo acontece com a variável **idade**.

Ao salvar o arquivo **main.js**, você pode ver no console do navegador as mensagens impressas com a cada tipo de dados das variáveis declaradas.



Na ordem impressa no console temos:

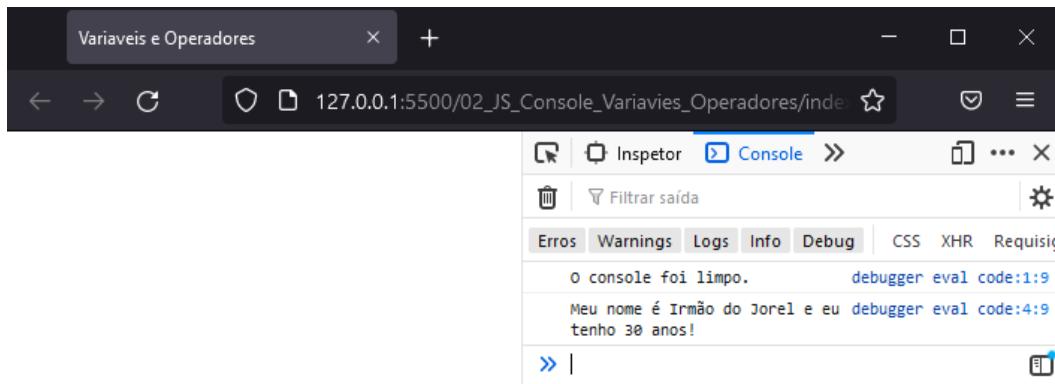
- Tipo de dados da variável nome_pessoa: **string**
- Tipo de dados da variável idade1: **number**
- Tipo de dados da variável rating: **number**
- Tipo de dados da variável isCool: **boolean**
- Tipo de dados da variável x: **object**
- Tipo de dados da variável y: **undefined**
- Tipo de dados da variável z: **undefined**

Concatenar strings é muito comum em JavaScript. Você irá fazer isso muitas vezes, portanto preste bastante atenção será que esse assunto for mostrado no material. Vamos inserir o seguinte código para concatenar strings no arquivo **main.js**

```
// Concatenar strings
console.clear();
const pessoa = 'Irmão do Jorel';
const idade2 = 30;
console.log('Meu nome é ' + pessoa + ' e eu tenho ' + idade2 + ' anos!');
```

O comando mostra uma string sendo concatenado com o operador +, nessa maneira de concatenar o texto deve colocado entre aspas simples ou duplas e o nome da variável sem aspas.

Ao salvar o arquivo **main.js**, é possível visualizar o resultado no console.

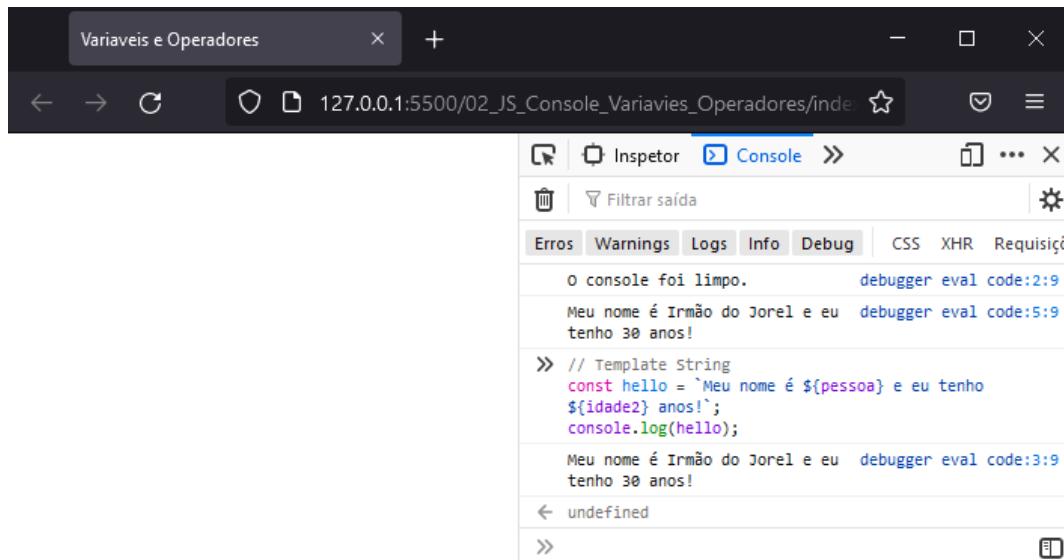


Outra forma mais prática de concatenar strings é usar Template Strings. Atualize o arquivo **main.js** com o exemplo abaixo.

```
// Template String
const hello = `Meu nome é ${pessoa} e eu tenho ${idade2} anos!`;
console.log(hello);
```

Veja que utilizamos um **Template String**. Falaremos mais sobre essa forma prática de **concatenar strings**, mas uma rápida explicação: **você deve colocar a string entre acentos de crase, os nomes das variáveis começando por cifrão (\$)** e entre chaves { } e não é necessário utilizar o operador +

O resultado apresentado no console é o mesmo do anterior.

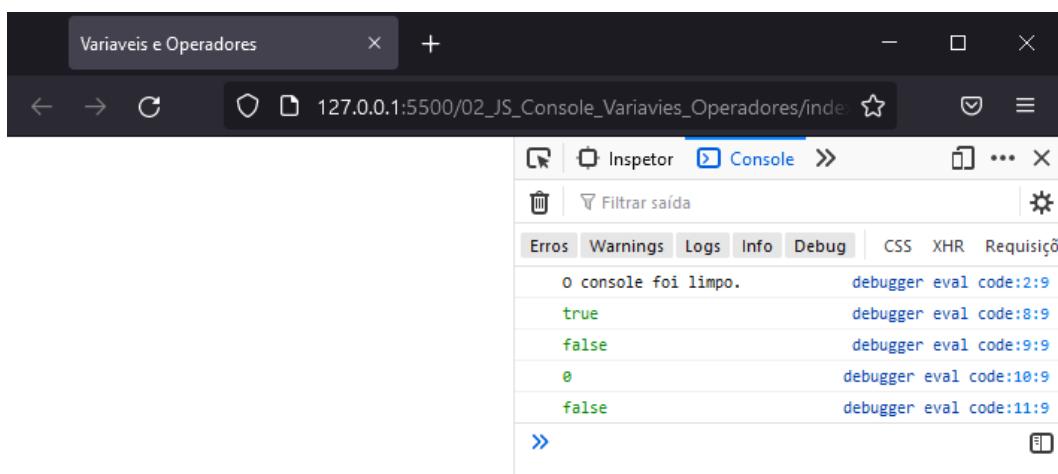


Por fim, vamos ver alguns resultados de uso de operadores de comparação e lógicos. Atualize o arquivo **main.js** com o código abaixo

```
// Operadores de comparação e Lógico
console.clear();
let teste1 = 1;
let teste2 = 0;
let valor1 = true;
let valor2 = false;

console.log(teste1 > teste2); // Operador de comparação
console.log(teste1 < teste2); // Operador de comparação
console.log(teste1 && teste2); // Operador Lógico
console.log(valor1 && valor2); // Operador Lógico
```

Observe o resultado no console do navegador.



Importante: Falaremos mais sobre esses testes lógicos na parte de estruturas condicionais e estruturas de repetição. Mas se você quiser se adiantar pode procurar por operações lógicas: **AND**, **OR**, **NOT** e por tabelas verdades.



Strings

Os objetivos desta aula são:

- Compreender o uso de strings e sua sintaxe;
- Conhecer o conceito de concatenar strings;
- aprender os diferentes métodos de strings.

Mais sobre strings

String é um tipo especial em toda linguagem de programação e no JavaScript não seria diferente. O tipo de dados String tem propriedades e métodos, que permitem manipulá-las de diversas formas. Vamos aprender alguns desses métodos e dessas propriedades.

Strings é um conjunto de caracteres muito utilizado em diferentes linguagens de programação de diversas formas possíveis. Uma string pode conter letras, números e caracteres especiais e, dessa forma, possibilita enviar bloco de dados e não somente palavras ou frases. No contexto de aplicações reais, strings são a forma de enviar informações de uma aplicação para outra. Podemos citar o formato JSON (JavaScript Object Notation), que é um padrão de formatação de dados, que é facilmente “parseado” (dividido) em informações úteis. Outro padrão muito utilizado em aplicações é o XML (Extensible Markup Language). Como foi dito anteriormente, no JS, uma string é criada colocando as informações entre aspas simples ou duplas:

```
const string1 = "Uma string";  
  
const string2 = 'Também 123 [] é uma strings 231 -1';
```

Ela também pode ser colocada entre crases:

```
const string3 = `Essa também é uma string entre crases`;
```



Importante! Esse modo de utilizar strings entre crases é muito útil quando queremos concatenar (juntar) strings como outras strings ou dados de variáveis etc.

Concatenando strings

A concatenação de strings pode ser realizada com o operador + de forma semelhante a muitas linguagens de programação, tais como: Java, C++, C#, etc.

Concatenando strings

```
"Instituto" + ' ' + 'da' + " " + "Oportunidade" + ' ' + 'Social'
```

Resultado

Instituto da Oportunidade Social

Mas quando estamos trabalhando com **dados dinâmicos (variáveis, resultados de funções, etc)**, é mais **eficiente utilizar o formato de concatenar utilizando crases**. Vamos criar um projeto para ver isso na prática.



Vamos praticar

Abra o **VS Code** e escolha um **diretório de trabalho para o seu projeto**.

Crie um diretório para seu projeto com o nome representativo, por exemplo, **JS_Strings**

Crie um arquivo dentro do diretório do projeto com o nome **index.html**.

Insira o seguinte código no seu arquivo **index.html**.

```
<!DOCTYPE html>
<html lang="pt-br">
  <head>
    <meta charset="UTF-8" />
    <link rel="shortcut icon" href="#" />
    <title>Strings</title>
  </head>
  <body>
    <script src=".js/main.js"></script>
  </body>
</html>
```

Esse código mostra a marcação **<script>** sem nenhum código **JavaScript** entre a **abertura** e o **fechamento** da tag, apenas o atributo **src** com o valor **main.js**. Isso significa que o código JavaScript está em um arquivo externo. Portanto, temos que criar esse novo arquivo **main.js** dentro do diretório do projeto.

Observe que **mudamos a organização dos arquivos no projeto**, agora o arquivo **main.js** está em uma pasta chamada **js**, enquanto o arquivo **index.html** está no **diretório raiz do projeto**. Isso é importante para você ser uma pessoa organizada com os seus projetos. Além disso, o caminho do script agora está com apontando para o novo diretório (**<script src=".js/main.js"></script>**).

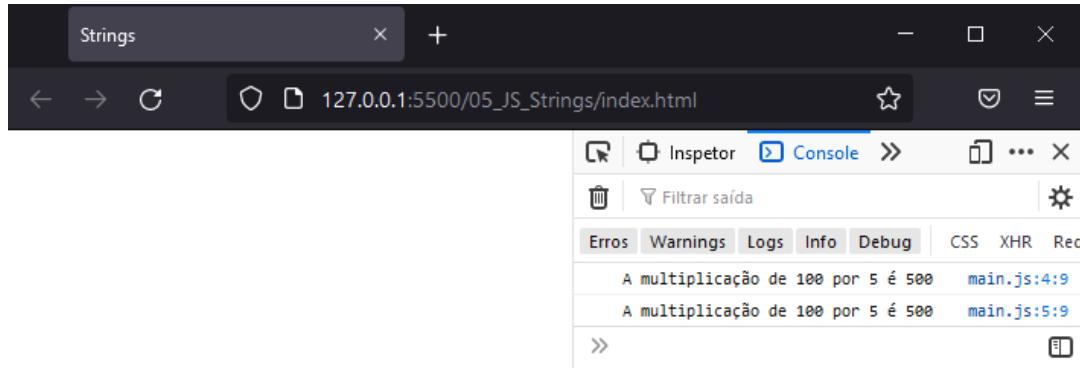
Vamos deixar o arquivo **main.js vazio** e à medida que vamos aprendendo coisas novas vamos completando a implementação do código **JavaScript**.

Abra o arquivo **index.html**, clique no botão **Go Live** da extensão **Live Server** e abra as Ferramentas de desenvolvimento do navegador web (Atalho **F12**).

Agora, vamos implementar todo o nosso código desse tema no arquivo **main.js** e ver as mensagens impressas no console.

```
// Concatenar strings
let valor01 = 100;
const valor02 = 5;
console.log('A multiplicação de ' + valor01 + ' por ' + valor02 + ' é ' + valor01 *
valor02 );
console.log(`A multiplicação de ${valor01} por ${valor02} é ${valor01 * valor02}`);
```

Antes de explicar o que foi programado vamos **ver o resultado mostrado no console**.



As duas instruções **produzem o mesmo resultado**. A primeira instrução de soma mostra a forma **tradicional para concatenar strings**. Isso, **pode ser uma vantagem para pessoas, que vêm de outras linguagens de programação**, se sentirem mais habituadas com esse formato de concatenação. A **desvantagem principal é a quantidade de aberturas e fechamentos de aspas**, além dos **muitos operadores de soma**, que podem fazer a pessoa programadora se perder e acabar esquecendo de **abrir ou fechar um par de aspas corretamente ou de colocar um sinal de +**.

Desse modo, a maneira de concatenar mostrada na **segunda instrução de soma**, conhecida como **template strings**, pois elas **permitem embutir expressões de JavaScript dentro da string através do símbolo de \$ e a abertura e fechamento das chaves {}**

```
`${expressão_do_JavaScript}`
```

Essa expressão pode ser **qualquer instrução válida no JavaScript**. Utilizando a segunda forma de **concatenar strings**, a **programação fica mais prática e rápida**. Lembre-se disso, quando estivermos criando **uma string enorme com muitas linhas**.

Métodos de strings

Em **JavaScript** praticamente qualquer tipo de dado **é um objeto**. Cada item dessa "**coleção de valores**", é chamado de **propriedade**. Cada propriedade é composta por um **par de "nome: valor"**. Quando uma propriedade **armazena uma função**, ela se torna **o que chamamos de método**.

Vamos aprender sobre os **métodos de strings** continuando a nossa programação no arquivo **main.js**.

Acesso a caracteres

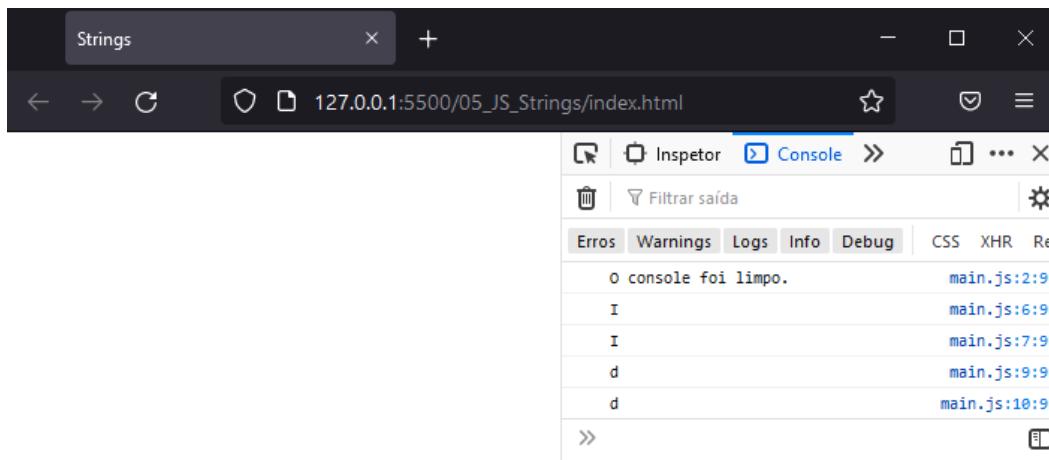
Vamos começar pelo **acesso à caracteres em uma string**. Podemos fazer o acesso de **duas formas**, a **primeiro** é pelo **método charAt(n)**, onde **n** é o **valor da posição do caractere na string**, e a **segunda** é através de **colchetes [n]**, onde **n** é o **valor da posição do caractere na string**. Vamos ver isso na prática, siga os passos para continuar a implementação:

Continuando a implementação do projeto, insira o seguinte código no arquivo **main.js**:

```
// Métodos de strings
console.clear();
const string01 = 'Instituto da Oportunidade Social';

// Acessando o primeiro caractere
console.log(string01.charAt(0));
console.log(string01[0]);
// Acessando o décimo primeiro caractere
console.log(string01.charAt(10));
console.log(string01[10]);
```

O resultado é mostrado no console do navegador:



Observe que a **primeira posição do caractere na string** é a **zero (0)**, isso porque **string** são **vetores (arrays)** e em **linguagens de programação** a **primeira posição sempre começa do índice zero**. Por isso, as **primeiras instruções retornaram I**, pois é o caractere na **primeira posição (índice 0)**. E o caractere na **décima primeira posição (índice 10)** é o **d**.

Observe também que para acessar **um método em JavaScript** colocamos o **nome_do_objeto** depois **ponto final** e depois o **nome do método**:

string01	.	charAt(10)
nome_do_objeto	ponto final	nome do método

Tamanho da string

A propriedade **length** retorna o **tamanho da string**, ou seja, o **número de caracteres que a string possui**. Esse método será muito útil em laços de repetição, quando iremos implementar um código para percorrer os caracteres de uma string.

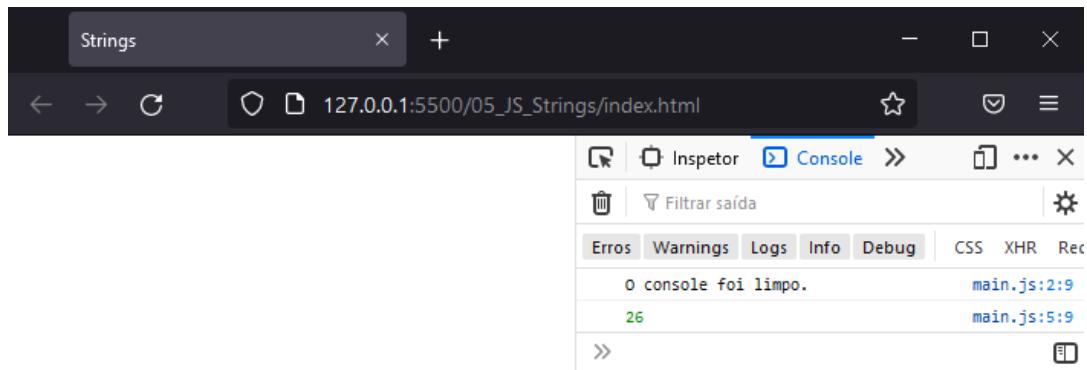
Vamos ver isso na prática, siga os passos para continuar a implementação:

Continuando a implementação do projeto, insira o seguinte código no arquivo **main.js**:

```
// Tamanho de strings
console.clear();
let texto = 'abcdefghijklmnoprstuvwxyz';

console.log(texto.length);
```

O resultado é mostrado no console do navegador:



Observe que o comando **texto.length** retorna a quantidade de **caracteres da string texto**, que são **26 caracteres**.

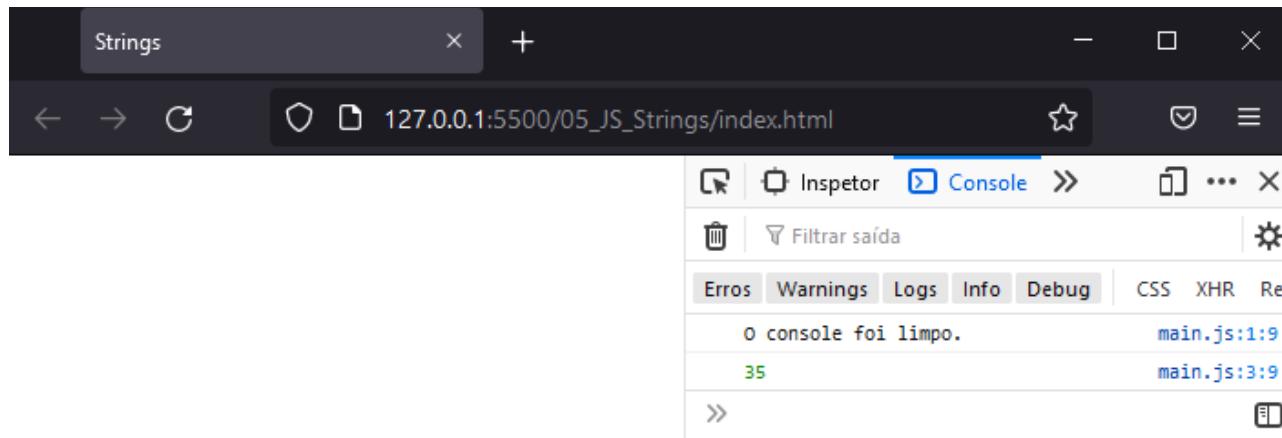
Esse método conta **todos os caracteres da string**, seja **letras** ou **números, espaços em branco**.

Vejamos o exemplo:

Insira o seguinte código no arquivo **main.js**:

```
console.clear();
const texto2 = 'Também 123 [] é uma strings 231 -1\n';
console.log(texto2.length);
```

O resultado é mostrado no console do navegador:



Olha que nesse exemplo ele contou **todos os caracteres da string separadamente**, mas ele conta o caractere de escape \n como apenas um caractere, que é o certo. **Pode conferir** 😊.

Maiúsculas e minúsculas

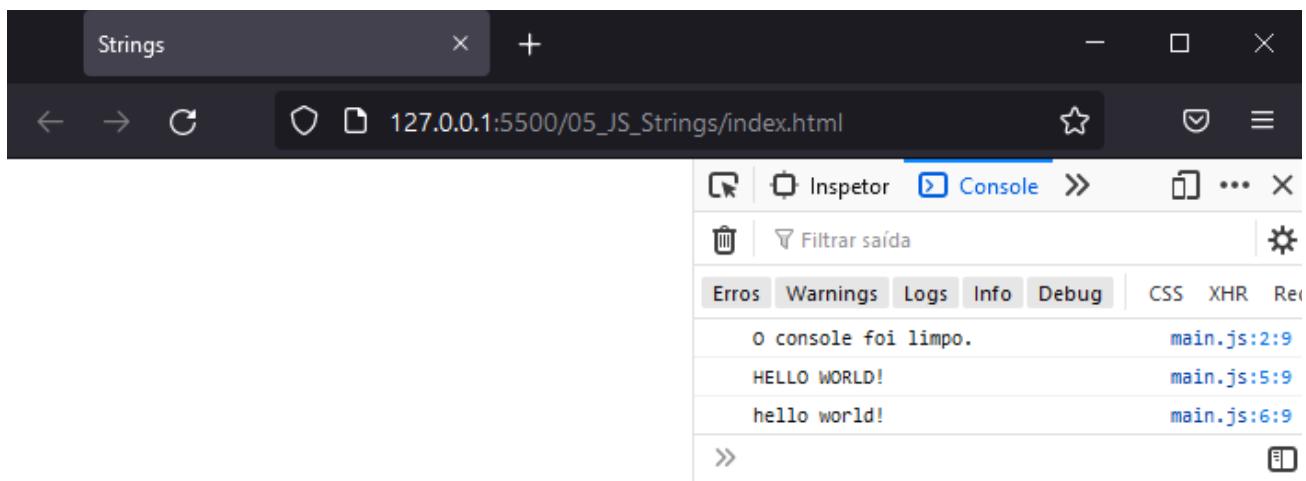
Você pode transformar **todos os caracteres de uma string para maiúsculo ou para minúsculo**, isso facilitam, por exemplo, **comparar strings para verificar se são iguais**. O método **toUpperCase** transforma **todos os caracteres de uma string para maiúsculos** e o método **toLowerCase** transforma para **minúsculo**. Vamos ver isso na prática, siga os passos para continuar a implementação:

Continuando a implementação do projeto, insira o seguinte código no arquivo **main.js**:

```
// Maiusculo e minúsculo
console.clear();
const s = 'Hello World!';

console.log(s.toUpperCase());
console.log(s.toLowerCase());
```

O resultado é mostrado no console do navegador:



Observe que o comando `s.toUpperCase()` transformou **todos os caracteres da string para maiúsculo**, enquanto o comando `s.toLowerCase()` transformou **todos os caracteres da string para minúsculo**.

Substring e Slipt

O método **substring corta a string de acordo com os índices indicados entre os parênteses**, por exemplo:

```
const str = 'Mozilla';
console.log(str.substring(1, 3));
// Saída esperada: "oz"

console.log(str.substring(2));
// Saída esperada: "zilla"
```

O comando `str.substring(1, 3)` **corta a string a partir do índice 1 até o índice 3** e o comando `str.substring(2)` **corta a string a partir do índice 2 até o seu final**.

O método **split divide a string em substrings** de acordo com a **regra colocada entre parênteses**. O objeto retornado por esse **método** é um array. Por exemplo:

```
const str1 = 'A raposa é um animal esperto';

const palavras = str1.split(' ');
console.log(palavras[3]);
// Saída esperada: "um"

const chars = str1.split('');
console.log(chars[7]);
// Saída esperada: "a"

const strCopy = str1.split();
console.log(strCopy);
// Saída esperada: Array ["A raposa é um animal esperto"]
```

O comando `str1.split(' ')` [atente-se que existe um espaço em branco dentro das aspas simples] **separou a string em substring** de acordo com o espaço entre elas. Desse modo, o comando retorna um **array com seis palavras separadas**. Então, o comando `palavras[3]` acessa a **palavra** está no índice 3 que é ‘um’.

O comando `str1.split("")` [atente-se que não existe mais espaço em branco entre as aspas simples] **separou a string em substring de caracteres**, ou seja, **colocou cada caractere em**

um índice do vetor. Desse modo, o comando **retorna um array com vinte oito posições**, que são os **caracteres da string**. Então, o comando **char[7]** acessa o caractere está no **índice 7** que é **'a'**.

Por fim, o comando **str1.split()**, sem nenhuma regra faz uma cópia idêntica da **string**.

Vamos ver isso na prática, siga os passos para continuar a implementação:
Continuando a implementação do projeto, insira o seguinte código no arquivo **main.js**:

```
// Substring e split
console.clear();
const str = 'Mozilla';

console.log(str.substring(1, 3));
console.log(str.substring(2));

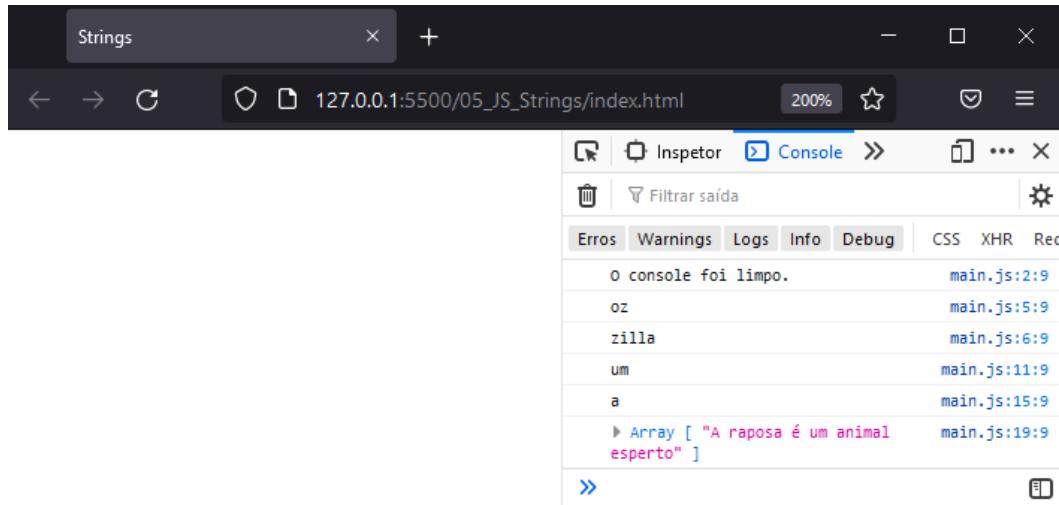
const str1 = 'A raposa é um animal esperto';

const palavras = str1.split(' ');
console.log(palavras[3]);
// Saída esperada: "um"

const chars = str1.split('');
console.log(chars[7]);
// Saída esperada: "a"

const strCopy = str1.split();
console.log(strCopy);
// Saída esperada: Array [ "A raposa é um animal esperto" ]
```

O resultado é mostrado no console do navegador:



Substituir uma string dentro da string

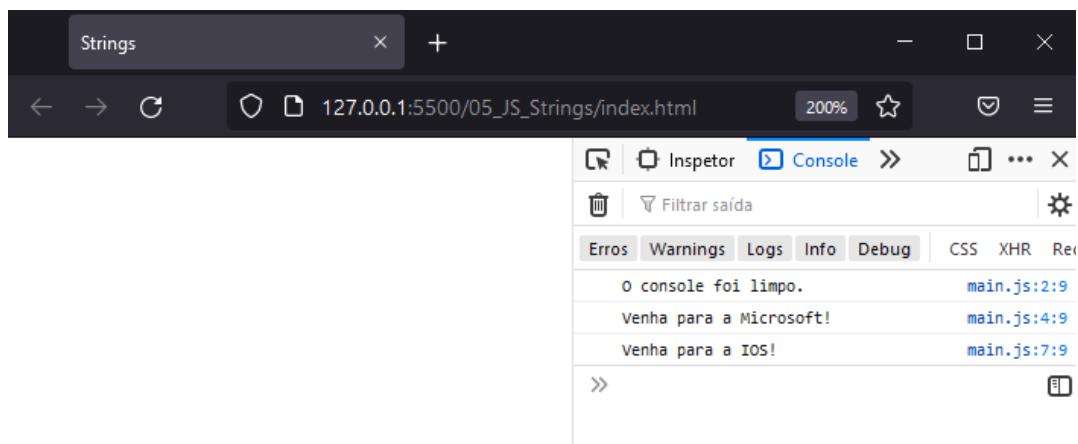
O método **replace** substitui um pedaço específico da string por outra coisa que você desejar. Vamos ver isso na prática, siga os passos para continuar a implementação:

Continuando a implementação do projeto, insira o seguinte código no arquivo **main.js**:

```
// Substituir string
console.clear();
let mensagem = 'Venha para a Microsoft!';
console.log(mensagem);

let novaMensagem = mensagem.replace('Microsoft', 'IOS');
console.log(novaMensagem);
```

O resultado é mostrado no console do navegador:



Podemos ver, nesse exemplo, que a palavra **Microsoft** foi substituída por **IOS** com o comando **mensagem.replace('Microsoft', 'IOS')**.

Remover espaços da string

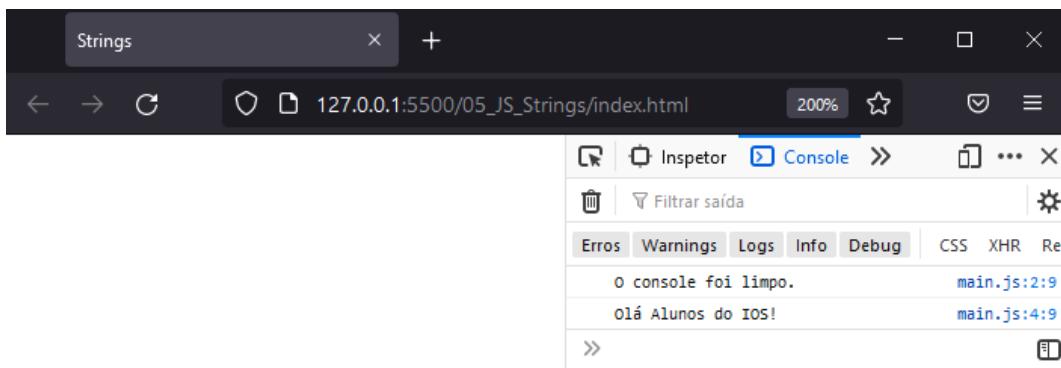
O método **trim** remove os espaços em branco do início e o final de uma string. Isso é útil por exemplo, quando estamos lendo um texto digitado pelo usuário em um formulário e esse texto vem com espaços no início e/ou no final. Vamos ver isso na prática, siga os passos para continuar a implementação:

Continuando a implementação do projeto, insira o seguinte código no arquivo **main.js**:

```
//Remover Espaços
console.clear();
let text = '      Olá Alunos do IOS!      ';
```

```
console.log(text.trim());
```

O resultado é mostrado no console do navegador:



Podemos ver, nesse exemplo, os **espaços no início e no final da string foram retirados com o comando `text.trim()`**.

Métodos de buscar em strings

Existem alguns métodos usados para **buscar algo dentro da string**, eles são úteis para programas que precisem **encontrar algum valor ou padrão na sequência de caracteres enviada**. Os métodos são:

- **`String.indexOf()`**: esse método retorna o **índice (posição)** da primeira vez que um texto especificado entre os parênteses foi encontrado na string.
- **`String.lastIndexOf()`**: esse método retorna o índice da última ocorrência de um texto especificado entre os parênteses foi encontrado na string.
- **`String.search()`**: esse método busca um valor específico e retorna a posição inicial desse valor na primeira vez que ele é encontrado. Semelhante ao `indexOf()`.
- **`String.startsWith()`**: esse método verifica se a string inicia com um valor específico.
- **`String.endsWith()`**: esse método verifica se a string termina com um valor específico.

Vamos ver isso na prática, siga os passos para continuar a implementação:

Continuando a implementação do projeto, insira o seguinte código no arquivo **main.js**:

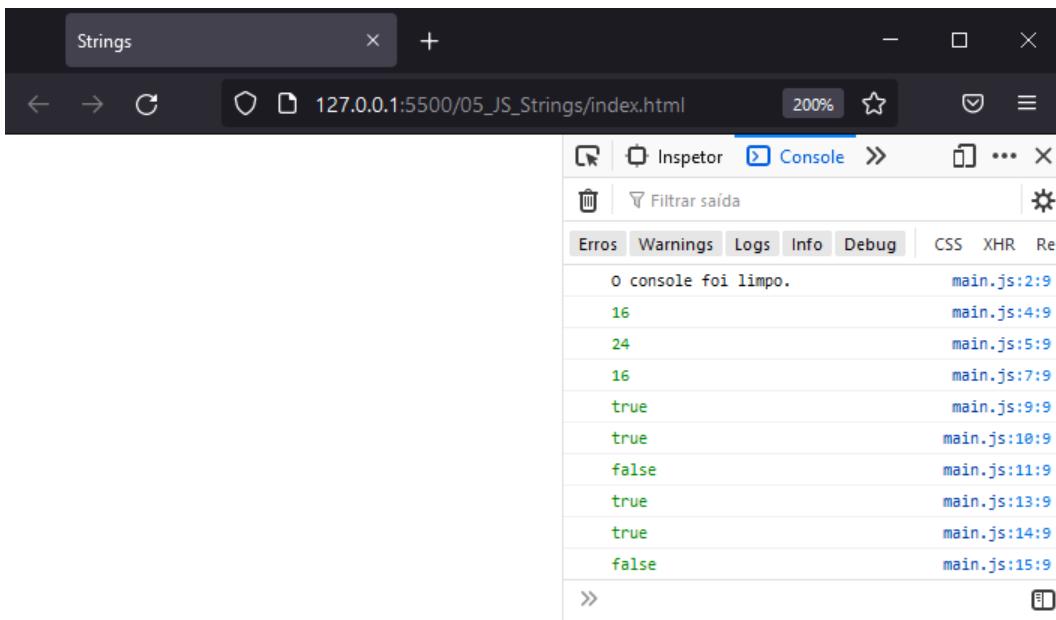
```
// Métodos de busca en strings
console.clear();
let frase = 'Sou um aluno do IOS e o IOS é muito bom!';
console.log(frase.indexOf('IOS')) // Returns 16
console.log(frase.lastIndexOf('IOS')) // Returns 24

console.log(frase.search('IOS')) // Returns 16
```

```
console.log(frase.startsWith('S')) // Verdadeiro - retorna true
console.log(frase.startsWith('Sou')) // Verdadeiro - retorna true
console.log(frase.startsWith('a')) // Falso - retorna false

console.log(frase.endsWith('!')) // Verdadeiro - retorna true
console.log(frase.endsWith('bom!')) // Verdadeiro - retorna true
console.log(frase.endsWith('x')) // Falso - retorna false
```

O resultado é mostrado no console do navegador:



Podemos ver, o comando `frase.indexOf('IOS')` retorna o **índice do início da string**, onde ela aparece pela primeira vez. O comando `frase.lastIndexOf('IOS')` retorna o **índice do início da string, onde ela aparece pela última vez**. O comando `frase.search('IOS')` é semelhante ao `indexOf` e retorna 16 também. O comando `frase.startsWith('S')` verifica se a **string começa** com **S** e retorna **verdadeiro**. O comando `frase.startsWith('Sou')` verifica se a **string começa** com **Sou** e retorna **verdadeiro**. O comando `frase.startsWith('a')` verifica se a **string começa** com **a** e retorna **falso**. O comando `frase.endsWith('!')` verifica se a **string termina** com **!** e retorna **verdadeiro**. O comando `frase.endsWith('bom!')` verifica se a **string termina** com **bom!** e retorna **verdadeiro**. O comando `frase.endsWith('x')` verifica se a **string termina** com **x** e retorna **falso**.

Conclusão

Existem muitos outros métodos para manipular strings e gastaríamos muito tempo se quisessem abordar todos esses métodos. À medida que outros métodos forem aparecendo, vamos explicando cada uma desses. O importante é que você sempre deve usar a internet para procurar soluções em programação, mas não apenas copie e cole uma solução, um bom programador procura entender a solução proposta e adaptá-la no seu projeto.

Para saber mais: Você pode consultar a lista de métodos de strings nos links:

https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/String

https://www.w3schools.com/js/js_string_methods.asp

https://www.w3schools.com/js/js_string_search.asp

Como desafio, tente entender e implementar alguns desses métodos.



Estruturas condicionais

Os objetivos desta aula são:

- Compreender o uso de estruturas e desvios condicionais;
- Conhecer o operador ternário;
- Aprender a utilização do switch-case.

Desvios condicionais

Os **desvios condicionais decidem o fluxo de execução de programa**. Esses desvios são construídos com **estruturas condicionais simples (if)**, **composta (if-else)** e **switch**. Uma das tarefas fundamentais de qualquer programa é **decidir o que deve ser executado a seguir**. Os comandos de decisão permitem determinar qual é a ação a ser tomada com base no resultado de uma expressão condicional.

Seja em estruturas condicionais ou laços de repetições, sempre será necessário testar uma condição que irá resultar em um valor booleano (**true** ou **false**). Por isso, quase sempre, usamos de operadores de **comparação e/ou lógicos**:

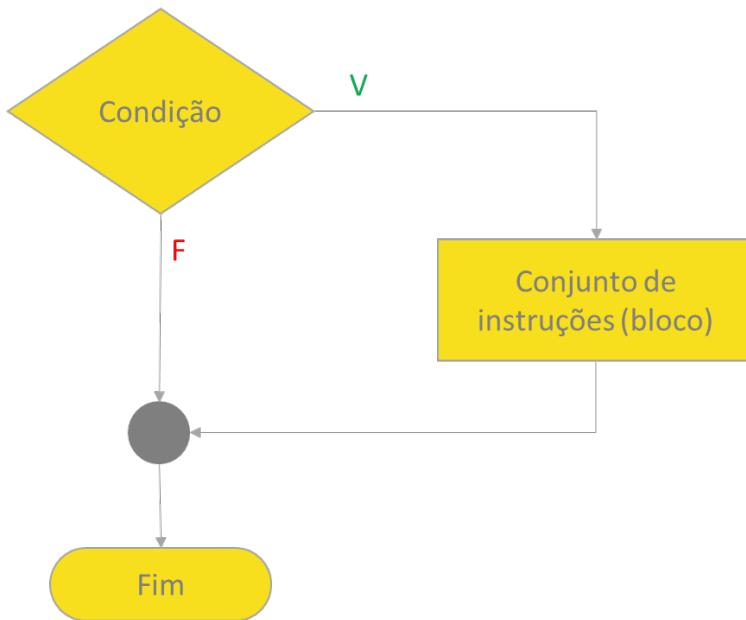
Categoria	Operador	Descrição
Operadores de comparação	==	Igual
	!=	Diferente
	<	Menor que
	<=	Menor ou igual
	>	Maior que
	>=	Maior ou igual
	==	Triplo igual

Categoria	Operador	Descrição
Operadores lógicos	&&	Lógica “and” ou “e”, que retorna verdadeiro se todos os operandos forem verdadeiros.
		Lógica “or” ou “ou”, que retorna verdadeiro se pelo menos um operando for verdadeiro.
	!	Lógica “not” ou “não”, que inverte o valor lógico se é verdadeiro, retorna falso e se é falso retorna verdadeiro.

Estrutura condicional simples (if)

A estrutura condicional simples **if** é usada para verificar se **dada condição é atendida**:

- **Se for**, um **conjunto de instruções deverá ser executado**;
- **Se não**, o fluxo de execução do algoritmo **seguirá após o fim do bloco de decisão**;



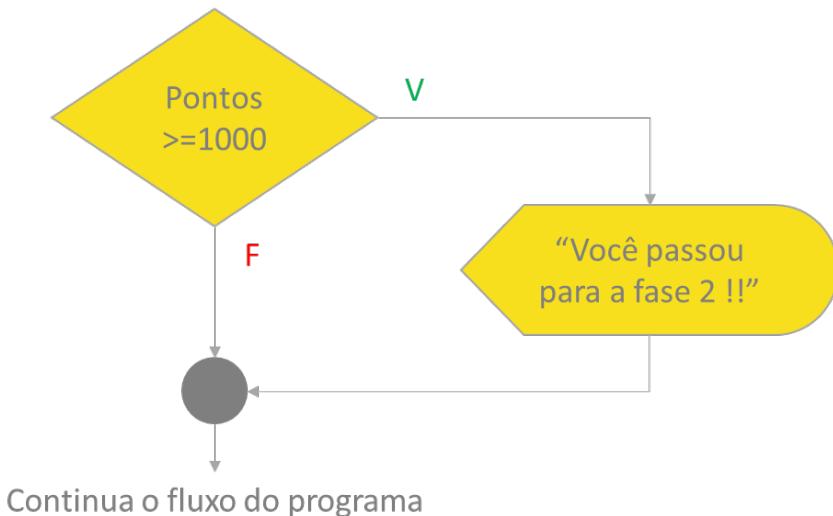
A sintaxe da estrutura **if** é:

```

if (condição) {
    instrucao1;
    instrucao2;
    ...
    instrucaoN;
}
proximaInstrucao;
  
```

Se a condição entre parênteses for verdadeira, o conjunto de instruções entre chaves (dentro do bloco da estrutura condicional if) será executado, caso contrário esse conjunto é saltado e o programa irá executar a próxima instrução fora das chaves da estrutura if.

Exemplo: Um jogador somente irá **passar para a fase 2 se atingir 1000 pontos**.





Vamos praticar

Siga os passos para criar o projeto:

Abra o VS Code e escolha um diretório de trabalho para o seu projeto.

Abra o **VS Code** e escolha um **diretório de trabalho para o seu projeto**.

Crie um diretório para seu projeto com o nome representativo, por exemplo, **JS_Estruturas**

Crie um arquivo dentro do diretório do projeto com o nome **index.html**.

Insira o seguinte código no seu arquivo **index.html**.

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

<head>
    <meta charset="UTF-8" />
    <link rel="shortcut icon" href="#" />
    <title>Estruturas Condicionais</title>
</head>

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

</html>
```

Esse código mostra a marcação **<script>** sem nenhum código **JavaScript** entre a **abertura** e o **fechamento** da tag, apenas o atributo **src** com o valor **main.js**. Isso significa que o código JavaScript está em um arquivo externo. Portanto, temos que criar esse novo arquivo **main.js** dentro do diretório do projeto.

Observe que **mudamos a organização dos arquivos no projeto**, agora o arquivo **main.js** está em uma pasta chamada **js**, enquanto o arquivo **index.html** está no **diretório raiz do projeto**. Isso é importante para você ser uma pessoa organizada com os seus projetos. Além disso, o caminho do script agora está com apontando para o novo diretório (**<script src=".js/main.js"></script>**).

Vamos deixar o arquivo **main.js vazio** e à medida que vamos aprendendo coisas novas vamos completando a implementação do código **JavaScript**.

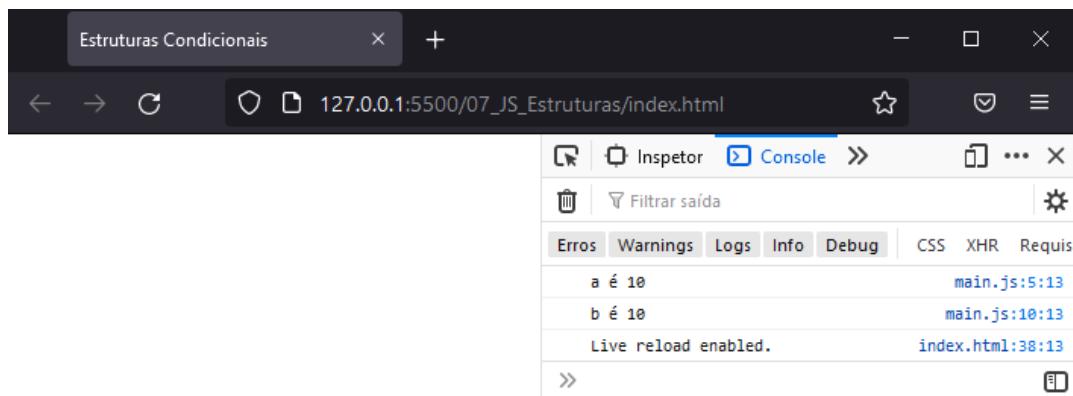
Abra o arquivo **index.html**, clique no botão  **Go Live** da extensão **Live Server** e abra as Ferramentas de desenvolvimento do navegador web (Atalho **F12**).

Agora, vamos implementar todo o nosso código desse tema no arquivo **main.js** e ver as mensagens impressas no console.

```
// Estrutura condicional simples
// Igual duplo
const a = 10;
if (a == 10) {
    console.log('a é 10');
}

const b = '10';
if (b == 10) {
    console.log('b é 10');
}
```

Antes de explicar o que foi programado vamos ver o resultado mostrado no console.



Sempre que testamos se **um valor é igual ao outro** utilizando o **igual duplo**, a linguagem JavaScript **não verifica o tipo de dados do valor comparado**. Por isso, ao comparar a **variável a**, que contém o **valor numérico 10**, com o **valor numérico 10** resulta **verdadeiro**, então o comando para imprimir a mensagem no console é executado:

a é 10 main.js:5:13

O mesmo resultado **verdadeiro** acontece, ao **comparar a variável b**, que contém a **string 10**, com o **valor numérico 10**, então o comando para imprimir a mensagem no console é executado:

b é 10 main.js:10:13

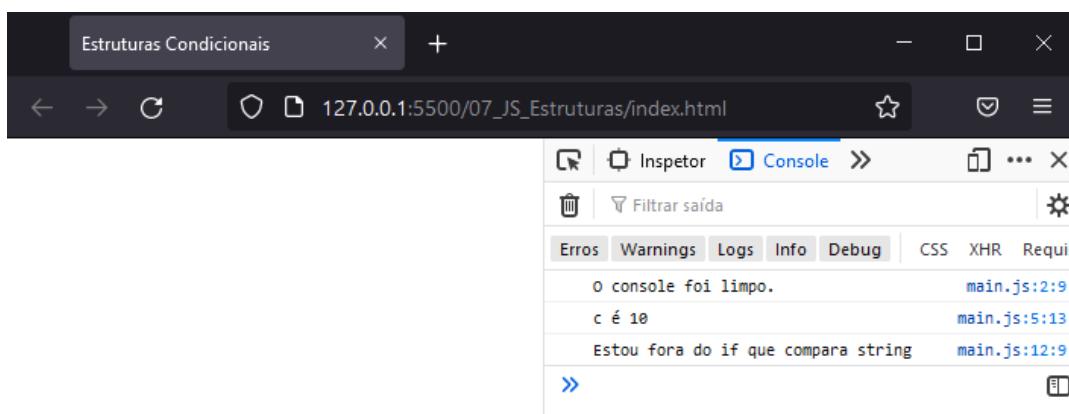
Portanto, cuidado, se você quiser **diferenciar o tipo de dados deve usar o triplo igual**.

Continuando a implementação do projeto, insira o seguinte código no arquivo **main.js**:

```
// Triplo igual - verifica o tipo
console.clear();
const c = 10;
if (c === 10) console.log('c é 10');

const d = '10';
if (d === 10) {
  console.log('d é 10');
}
console.log('Estou fora do if que compara string');
```

O resultado é mostrado no console do navegador:



Você deve notar a diferença nesse último exemplo, ao comparar **uma string com um valor numérico**, o resultado da comparação é **falso**, então o comando não é executado e o programa salva para a **próxima instrução**.

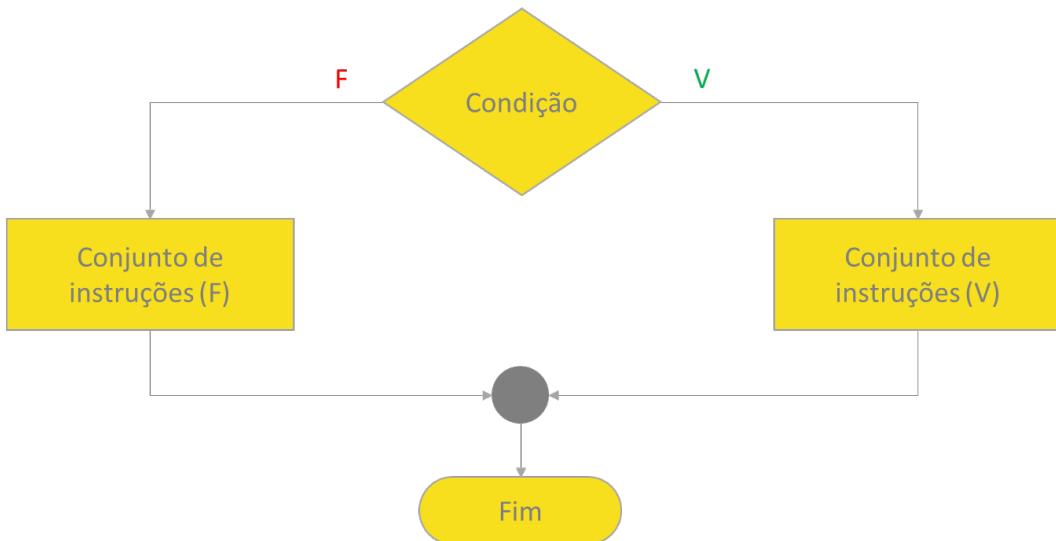
Estou fora do if que compara string main.js:22:9

Observe também que quando o bloco do comando **if** tem apenas uma única instrução, as chaves podem ser ocultadas (primeiro bloco **if**). Essa regra vai valer para todas as estruturas condicionais e laços de repetições ensinados nesse tema.

Estrutura condicional composta (if-else)

A **estrutura if-else** prevê **dois conjuntos de instruções para serem executadas** de acordo com a avaliação da condição:

- Um conjunto de instruções que será executado quando a condição resultar em **Verdadeiro**;
- Um conjunto de instruções quando a condição resultar em **Falso**;



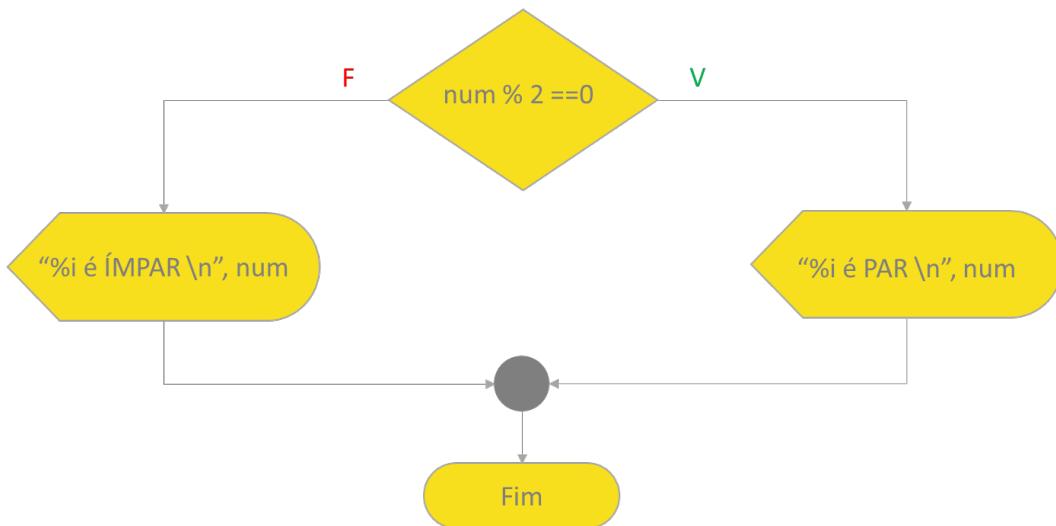
A sintaxe do if-else é:

```

if (condição) {
    instrucao1;
    instrucao2;
}
else {
    instrucao3;
    instrucao4;
}
proximaInstrucao;
  
```

Se a condição entre parênteses for **verdadeira**, o conjunto de instruções (**V**) será executado, caso a condição for **falsa**, o conjunto de instruções (**F**) será executado.

Exemplo: verificar se um número é par ou ímpar.



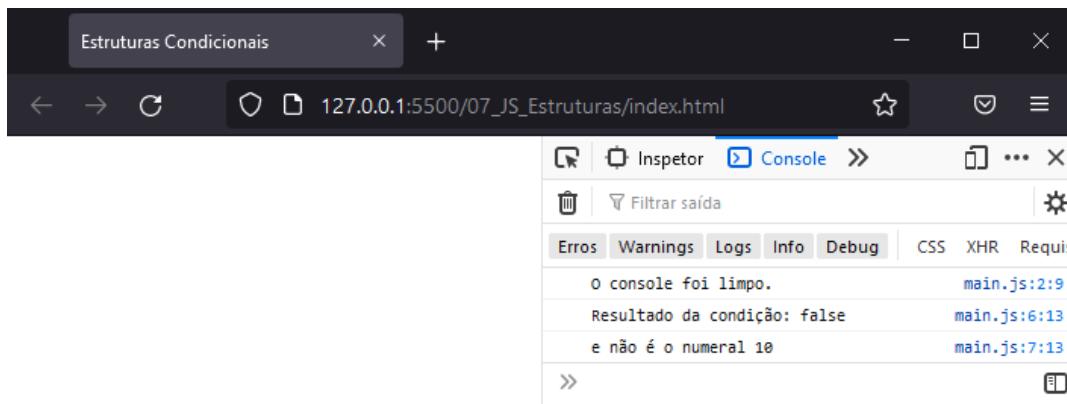


Vamos Praticar

No arquivo, **main.js** digite o seguinte código.

```
// Estrutura condicional composta
console.clear();
const e = '10';
if (e === 10) console.log('e é o numeral 10');
else {
  console.log(`Resultado da condição: ${e === 10}`);
  console.log('e não é o numeral 10');
}
```

O resultado é mostrado no console do navegador.



Observe que o **conteúdo da variável e é a string '10'**, portanto a condição resulta em um valor **falso**. Por isso o **bloco de comandos do else será executado**.

Resultado da condição: false	main.js:29:13
e não é o numeral 10	main.js:30:13

Podemos também ter **múltiplas comparações dentro dos parênteses**. Nesse caso, além dos operadores de comparação, devemos usar os **operadores lógicos**.

Continuando a implementação do projeto, insira o seguinte código no arquivo **main.js**:

```
// Múltiplas condições
console.clear();
const f = 4;
const g = 11;

if (f > 5 || g > 10) {
  console.log(`Condição é ${f > 5 || g > 10}`);
```

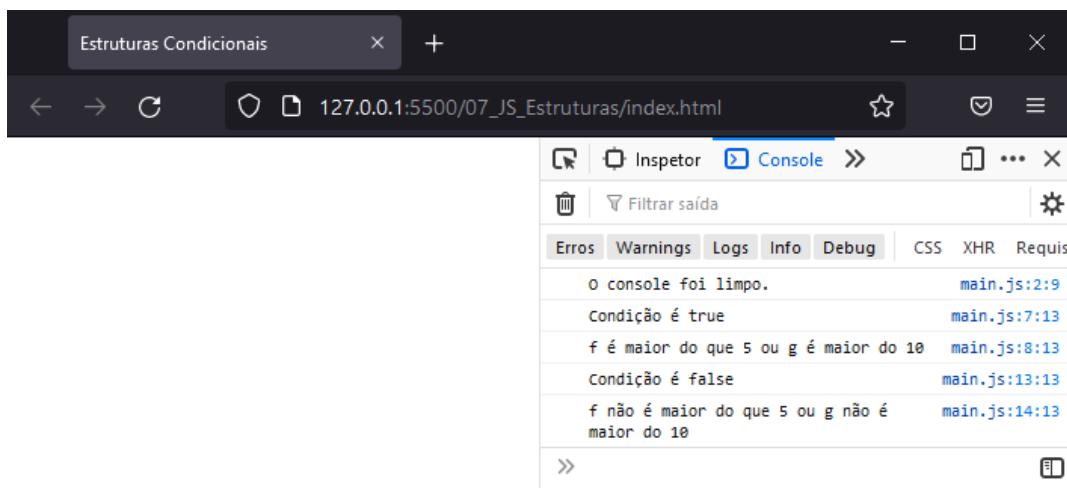
```

        console.log('f é maior do que 5 ou g é maior do 10');
} else console.log('f não é maior do que 5 e g não é maior do 10');

if (f > 5 && g > 10) console.log('f é maior do que 5 e g é maior do 10');
else {
    console.log(`Condição é ${f > 5 && g > 10}`);
    console.log('f não é maior do que 5 ou g não é maior do 10');
}

```

O resultado é mostrado no console do navegador:



Observe a múltipla comparação resulta em **true** (**Lembre-se da tabela verdade da lógica OR**), então o bloco de comando do **if é executado**:

Condição é true main.js:39:13

f é maior do que 5 ou g é maior do 10 main.js:40:13

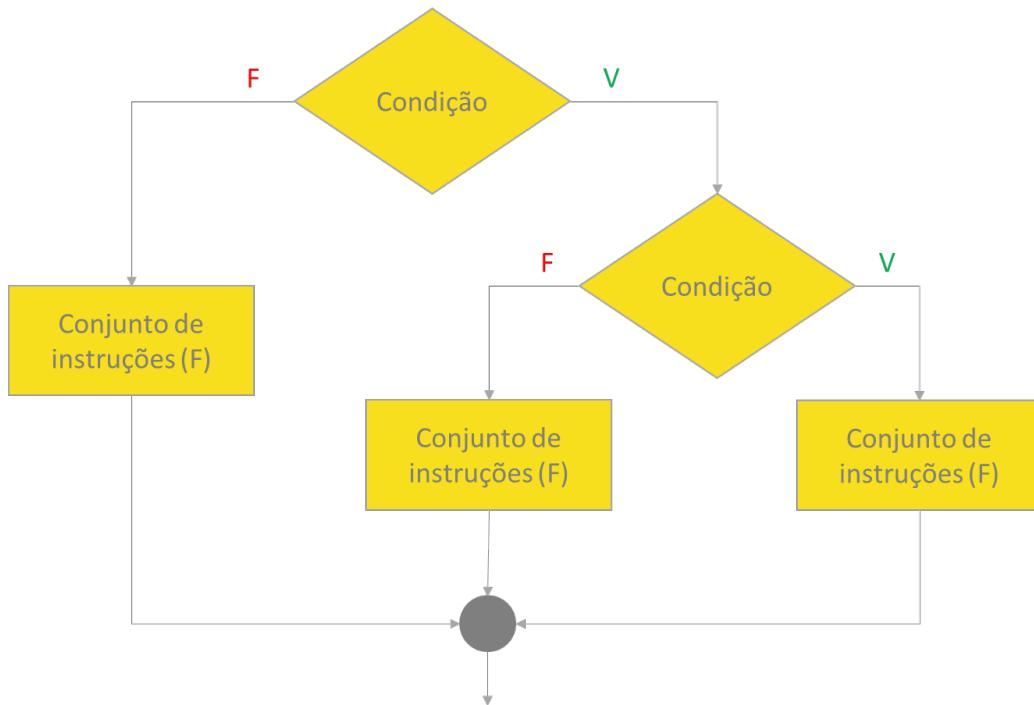
Também, é possível notar que a comparação seguinte resulta em **false** (**Lembre-se da tabela verdade da lógica AND**), então o bloco de comando do **else é executado**:

Condição é false main.js:45:13

f não é maior do que 5 ou g não é maior do 10 main.js:46:13

Desvios condicionais encadeados

Você pode encadear vários desvios condicionais quando for necessário verificar diversas condições. E cada condição **depende do resultado da condição anterior**. Basicamente, **if dentro de if (denominado ifs-elses aninhados)**:



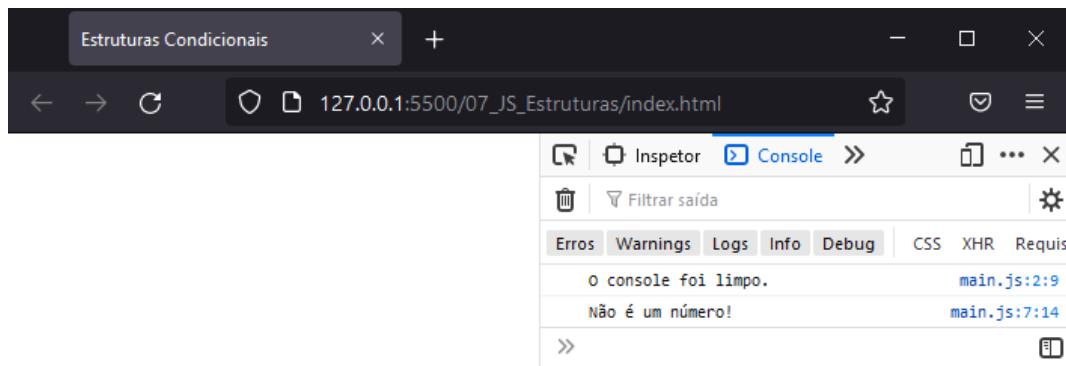
Vamos Praticar

No arquivo, **main.js** digite o seguinte código.

```
// Desvios encadeados
console.clear();
const num = 'Ola';

if (num % 2 == 0 && !isNaN(num)) console.log('Número par!');
else if (num % 2 != 0 && !isNaN(num)) console.log('Número ímpar');
else console.log('Não é um número!');
```

O resultado é mostrado no console do navegador.



```

Estruturas Condicionais
127.0.0.1:5500/07_JS_Estruturas/index.html
Console
O console foi limpo.
main.js:2:9
Não é um número!
main.js:7:14
  
```

Nesse exemplo, a **condição** faz **duas comparações**:

Primeiro, compara se o **número é par** (**resto da divisão por 2 for zero, então o número é par**), como **10 é par** resulta em **verdadeiro**.

Segunda, utiliza o método **isNaN** (**is Not a Number, não é um número**) para verificar **se é um não é um número**. Esse método retornará **true se não for um número** e **false se for um número**.

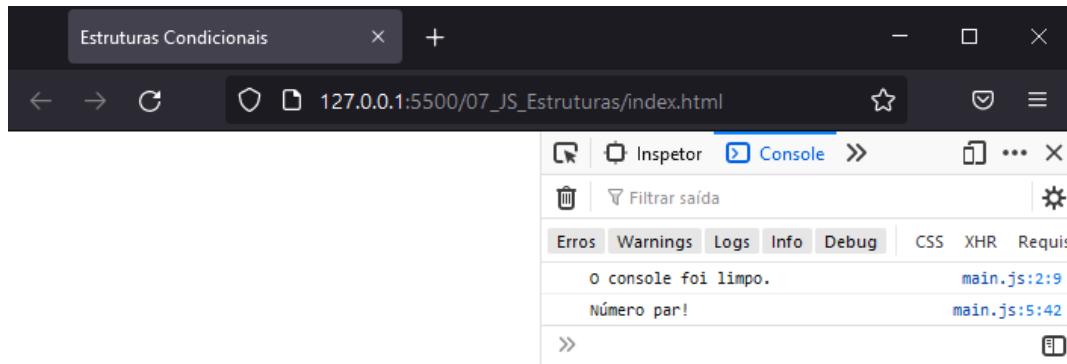
Como o conteúdo da variável **num** é '**Ola**', o conteúdo não é mais um número, então a primeira e a segunda comparações retornarão **false**. Portanto, o bloco de comandos do **else é executado**, imprimindo no console:

Vamos alterar o valor da variável **num** para **10**.

```
// Desvios encadeados
console.clear();
const num = 10;

if (num % 2 == 0 && !isNaN(num)) console.log('Número par!');
else if (num % 2 != 0 && !isNaN(num)) console.log('Número ímpar');
else console.log('Não é um número!');
```

O resultado é mostrado no console do navegador:



Como **10 é um número**, então o resultado de **isNaN(num)** é **false**, mas ao usar o operador de **negação exclamação (!)**, o valor lógico é **invertido tornando a expressão true**.

Como as **duas comparações são verdadeiras** de acordo com o operador lógico **AND (&&)**, o resultado final é **true**. Portanto, o bloco de comandos do **if é executado**, imprimindo no console:

Número par!	main.js:53:27
-------------	---------------

Se alteramos o valor da variável num para 11, o número **não é mais par**, então a primeira comparação **retornará false** e a **segunda (num % 2 != 0 && !isNaN(num))** será **true**. Portanto, o bloco de comandos do **else if é executado**, imprimindo no console:

Número ímpar	main.js:54:32
--------------	---------------

Operador ternário

O **operador ternário** é uma **instrução equivalente** a estrutura condicional composta **`if...else`** e a sua vantagem é tornar o código bem mais enxuto. A sua sintaxe é:

```
condição ? expr_1 : expr_2
```

Onde:

- **condição** é a condição que será testada.
- **expr_1** é o que fazer quando a condição for verdadeira.
- **expr_2** é o que fazer quando a condição for falsa.

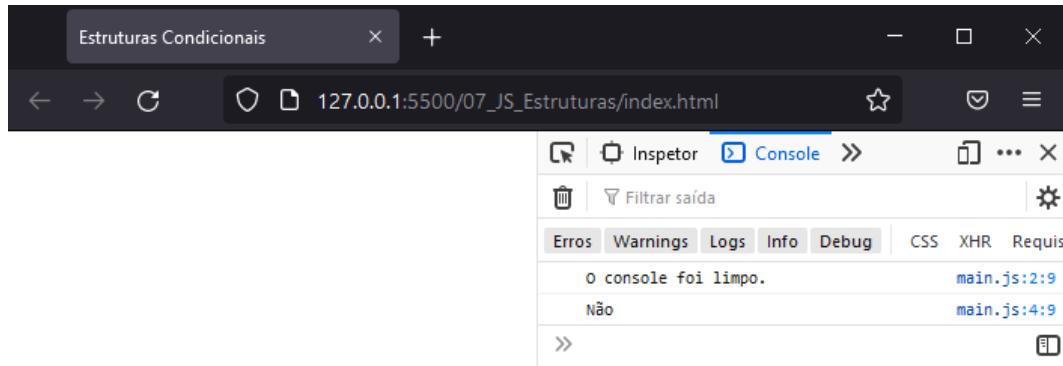


Vamos praticar

Continuando a implementação do projeto, insira o seguinte código no arquivo **main.js**:

```
// Operador ternário - exemplo 1
console.clear();
let resultado = 3 > 4 ? "Sim" : "Não";
console.log(resultado)
```

O resultado é mostrado no console do navegador:

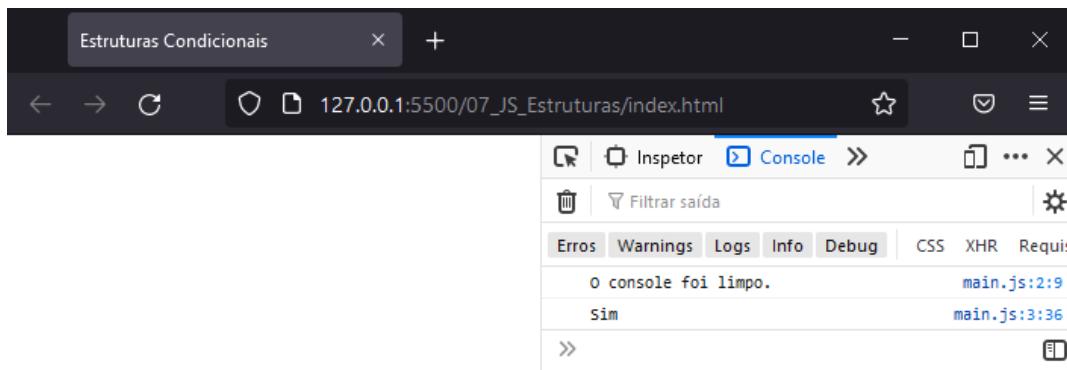


No exemplo, a condição é **comparar se `3 > 4`**. Como o resultado é **false**, o valor '**Não**' é **armazenado na variável resultado**. E ao imprimirmos essa variável no console, podemos ver que o conteúdo é '**Não**'.

Continuando a implementação do projeto, insira o seguinte código no arquivo **main.js**:

```
// Operador ternário - exemplo 2
console.clear();
let result = Math.PI < 4 ? console.log('Sim') : console.log('Não');
```

O resultado é mostrado no console do navegador:



No exemplo, usamos o objeto **Math** do **JavaScript**, que possui diversos valores matemáticas, por exemplo:

```
Math.PI;           // retorna 3.141592653589793
Math.E            // retorna o número de Euler
Math.SQRT2        // retorna a raiz quadrada de 2
Math.SQRT1_2      // retorna a raiz quadrada de 1/2
Math.LN2          // retorna o Logaritmo natural de 2
Math.LN10         // retorna o Logaritmo natural de 10
```

Então, **Math.PI** retorna **3,14...** que é um número **menor** do que **4**. Portanto, a **condição é verdadeira**, executando, assim, a instrução **console.log('Sim')** para imprimir a mensagem no console.

Switch

O **switch-case** trabalha com **situações mutuamente exclusivas**. A sua sintaxe é:

```
switch (expressão) {
    case valor1:
        //Instruções executadas quando o resultado da expressão for igual á valor1
        [break];
    case valor2:
        //Instruções executadas quando o resultado da expressão for igual á valor2
        [break];
    ...
    case valueN:
        //Instruções executadas quando o resultado da expressão for igual á valorN
        [break];
    default:
        //Instruções executadas quando o valor da expressão é diferente de todos os cases
        [break];
}
```

Se um caso do switch for verdadeiro, as instruções dele serão executadas, as demais não serão. A expressão dentro dos parênteses será comparada com o valor de cada case, quando "casar", executa a lista de comandos e finaliza o switch com o break. Se nenhum caso for verdadeiro, as instruções do caso default serão executadas.



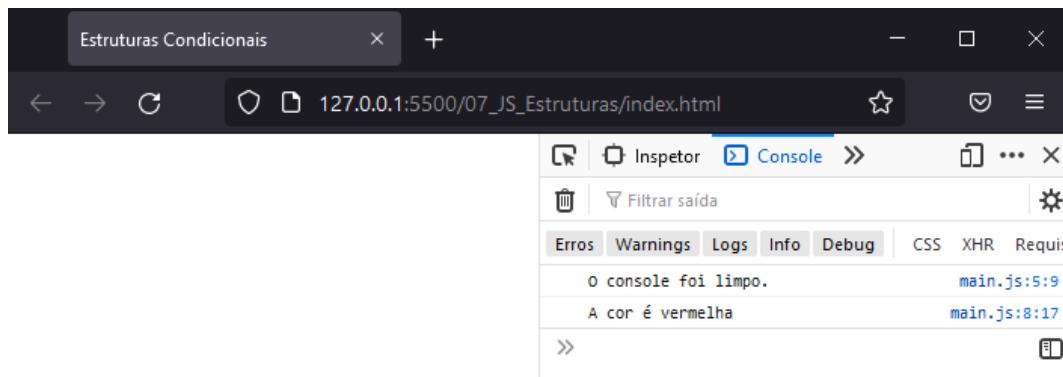
Vamos praticar

Continuando a implementação do projeto, insira o seguinte código no arquivo **main.js**:

```
// Switch
const h = 11;
const cor = h > 10 ? 'vermelha' : 'azul';

console.clear();
switch (cor) {
  case 'vermelha':
    console.log('A cor é vermelha');
    break;
  case 'azul':
    console.log('A cor é azul');
    break;
  default:
    console.log('A cor não é vermelha ou azul');
    break;
}
```

O resultado é mostrado no console do navegador:



No exemplo, **h** vale **11** e a condição **h > 10** é verdadeira. Portanto, a string '**vermelha**' é armazenada na variável **cor**.

Em seguida, a variável **cor** é comparada no **switch** e “**casa**” com o **primeiro case do switch**, isso faz com que a instrução seja **executada** e resultado na impressão do console:

A cor é vermelha

main.js:73:17

Se **alteramos o valor de há para 5**, o resultado será **azul**, pois a condição é **falsa** e o **segundo case do switch** é o que “casa” com **azul** agora:

A cor é azul

main.js:76:17

Conclusão

Procure sempre fazer mais do que é passado para você em sala de aula. Por exemplo, você pode alterar as condições, utilizar outras comparações múltiplas para ver o resultado que será gerado.

Seguem alguns links para você estudar e aprender mais:

Operador Ternário:

https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Operators/Conditional_Operator

https://www.w3schools.com/js/js_comparisons.asp

if-else:

<https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Statements/if...else>

Switch-case:

<https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Statements/switch>

Objeto Math:

https://www.w3schools.com/js/js_math.asp



Laços de repetição

Os objetivos desta aula são:

- Compreender o uso de laços de repetição;
- Diferenciar e entender quando aplicar os laços for, while e do-while;
- Combinar o uso de laços com arrays.

Laços de repetição

Os **laços de repetição** são **comandos utilizados para executar instruções mais de uma vez**, ou seja, **cria loops de repetição de uma ou mais instruções com um número limitado de vezes**. Eles possuem com variáveis de **controle/acumuladora**.

Para trabalhar com laços de repetições, **sempre será necessário testar uma condição que irá resultar em um valor booleano (true ou false)**. Por isso, quase sempre, fazemos usar de operadores de comparação e/ou lógicos:

Categoria	Operador	Descrição
Operadores de comparação	==	Igual
	!=	Diferente
	<	Menor que
	<=	Menor ou igual
	>	Maior que
	>=	Maior ou igual
	==>	Triplo igual

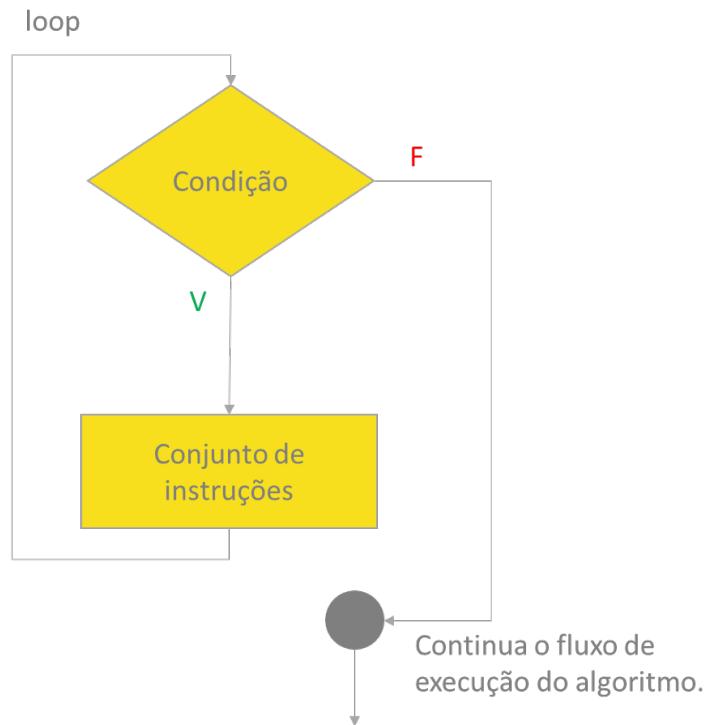
Categoria	Operador	Descrição
Operadores lógicos	&&	Lógica “and” ou “e”, que retorna verdadeiro se todos os operandos forem verdadeiros.
		Lógica “or” ou “ou”, que retorna verdadeiro se pelo menos um operando for verdadeiro.
	!	Lógica “not” ou “não”, que inverte o valor lógico se é verdadeiro, retorna falso e se é falso retorna verdadeiro.

Uma estrutura de repetição permite especificar que um **BLOCO ({ }) de instruções** ou o **programa todo deverá ser repetido** enquanto alguma condição permanecer **verdadeira**. O **número de repetições** pode ser **fixo** ou **atender alguma condição**.

Vantagens: O algoritmo passa a ter um **número menor de linhas** e de **instruções repetidas devido a estrutura de repetição** e é possível aumentar a amplitude de processamento sem alterar o tamanho do código.

Laço while

O laço **while** é uma **repetição com teste no início do comando**. Assim, o **bloco de instruções** **será repetidamente executado enquanto a condição for verdadeira**. Quando a condição for **falsa**, a execução do bloco de comandos **será interrompido**:



A sintaxe da estrutura **while** é:

```

while ( condição )
{
  instrucao1;
  instrucao2;
  instrucao3;
  instrucaoN;
}
proximaInstrucao;
  
```

O **bloco de comandos é executado** enquanto a condição for **verdadeira**. As chaves podem ser ocultadas se o comando contiver apenas uma instrução:

```

while ( condição ) instrucao1;
proximaInstrucao;
  
```

No **while**, a **condição é verificada antes de entrar no laço** e **enquanto** o resultado da condição for **verdadeiro** – executa o bloco de instruções. Quando a condição se tornar **falsa**, o laço é

encerrado e o fluxo vai para a próxima **instrução do algoritmo**. Portanto, a condição é testada a cada **iteração (volta)**.

Importante: Laço é um **bloco de instruções** ({ }) que será **executado repetidas vezes** e que está contido em uma estrutura de repetição.

Todo laço de repetição **pode ter um contador com variável de controle da condição**. Um contador é **uma variável do tipo inteiro usada para contar a quantidade de vezes que um bloco de instruções é repetido** e deve **ser inicializado antes de ser utilizado**.

```

Inicialização;

while(condição){
    Bloco de instruções;
    incremento ou decremento;
}

let contador = 1;

while(contador <= 10){
    console.log(`Valor = ${contador}`);
    contador++;
}

```

Observe, a **inicialização fora do comando while**, e o **incremente ou decremente do contador dentro do while**. Caso você não garantir que a condição em algum momento se torne **falsa**, o programa ficará em um **loop infinito**.



Vamos praticar

Siga os passos para criar o projeto:

Abra o VS Code e escolha um diretório de trabalho para o seu projeto.

Abra o **VS Code** e escolha um **diretório de trabalho para o seu projeto**.

Crie um diretório para seu projeto com o nome representativo, por exemplo, **JS_Lacos**

Crie um arquivo dentro do diretório do projeto com o nome **index.html**.

Insira o seguinte código no seu arquivo **index.html**.

```

<!DOCTYPE html>
<html lang="pt-br">

<head>
    <meta charset="UTF-8" />
    <link rel="shortcut icon" href="#" />
    <title>Laços de Repetição</title>

```

```
</head>

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

</html>
```

Esse código mostra a marcação **<script>** sem nenhum código **JavaScript** entre a **abertura** e o **fechamento** da tag, apenas o atributo **src** com o valor **main.js**. Isso significa que o código **JavaScript** está em um arquivo externo. Portanto, temos que criar esse novo arquivo **main.js** dentro do diretório do projeto.

Observe que **mudamos a organização dos arquivos no projeto**, agora o arquivo **main.js** está em uma pasta chamada **js**, enquanto o arquivo **index.html** está no **diretório raiz do projeto**. Isso é importante para você ser uma pessoa organizada com os seus projetos. Além disso, o caminho do script agora está com apontando para o novo diretório (**<script src=".js/main.js"></script>**).

Vamos deixar o arquivo **main.js vazio** e à medida que vamos aprendendo coisas novas vamos completando a implementação do código **JavaScript**.

Abra o arquivo **index.html**, clique no botão  **Go Live** da extensão **Live Server** e abra as Ferramentas de desenvolvimento do navegador web (Atalho **F12**).

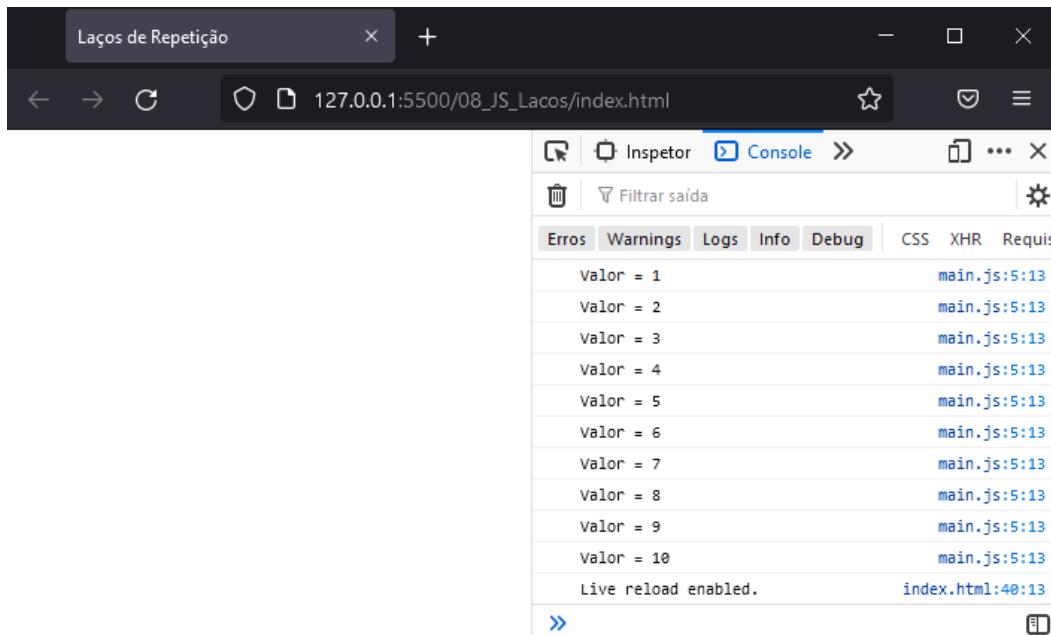
Agora, vamos implementar todo o nosso código desse tema no arquivo **main.js** e ver as mensagens impressas no console.

No arquivo, **main.js** digite o seguinte código.

```
// while
let contador = 1;

while (contador <= 10) {
  console.log(`Valor = ${contador}`);
  contador++;
}
```

Vamos ver o resultado mostrado no console.



Observe que a **variável** com o nome **contador** inicia com o **valor 1**, que é o **valor inicializado**, portanto a condição (**contador <= 10**) é **verdadeira** e será impresso o valor pela primeira vez no console:

Valor = 1 main.js:4:13

Em seguida, a variável é incrementada em uma unidade, a condição é testada novamente e o resultado **continua verdadeiro**. Portanto o segundo valor é impresso no console:

Valor = 2 main.js:4:13

Isso vai ser repetir até a variável assumir o valor 11, que então faz a condição retornar falso e encerrar a execução do laço.

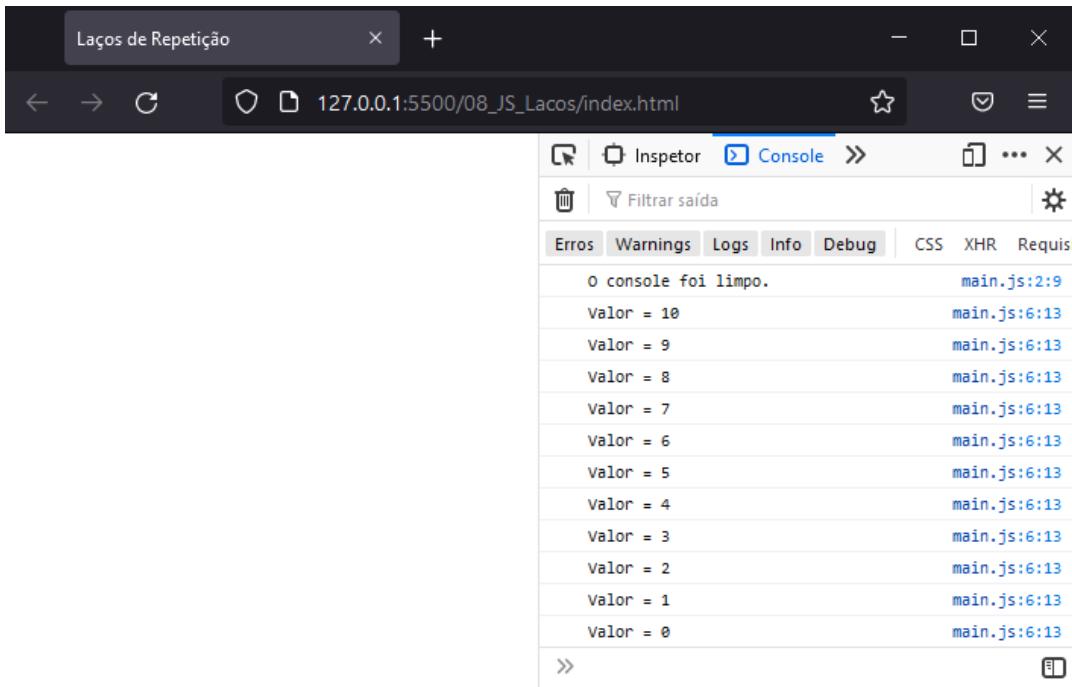
Podemos implementar um laço com **decremento** também.

Continuando a implementação do projeto, insira o seguinte código no arquivo **main.js**:

```
// while com decremento
console.clear();
let cont = 10;

while (cont >= 0) {
    console.log(`Valor = ${cont}`);
    cont--;
}
```

O resultado é mostrado no console do navegador:



The screenshot shows a browser window with the title "Laços de Repetição". Below the title bar is a toolbar with icons for back, forward, search, and refresh. The address bar displays the URL "127.0.0.1:5500/08_JS_Lacos/index.html". The main content area is a developer tools console tab labeled "Console". The console tab has several sub-tabs: Erros, Warnings, Logs, Info, Debug, CSS, XHR, and Requisições. The "Debug" tab is selected. The console output shows the following text:
 O console foi limpo.
 Valor = 10
 Valor = 9
 Valor = 8
 Valor = 7
 Valor = 6
 Valor = 5
 Valor = 4
 Valor = 3
 Valor = 2
 Valor = 1
 Valor = 0

Observe que a variável com o nome **contador** inicia com o **valor 10**, que é o valor **inicializado**, portanto a condição (**contador >= 0**) é **verdadeira** e será **impresso o valor pela primeira vez no console**:

Valor = 10 main.js:14:13

Em seguida, a **variável é decrementada em uma unidade**, a condição é **testada novamente** e o **resultado continua verdadeiro**. Portanto o segundo valor é impresso no console:

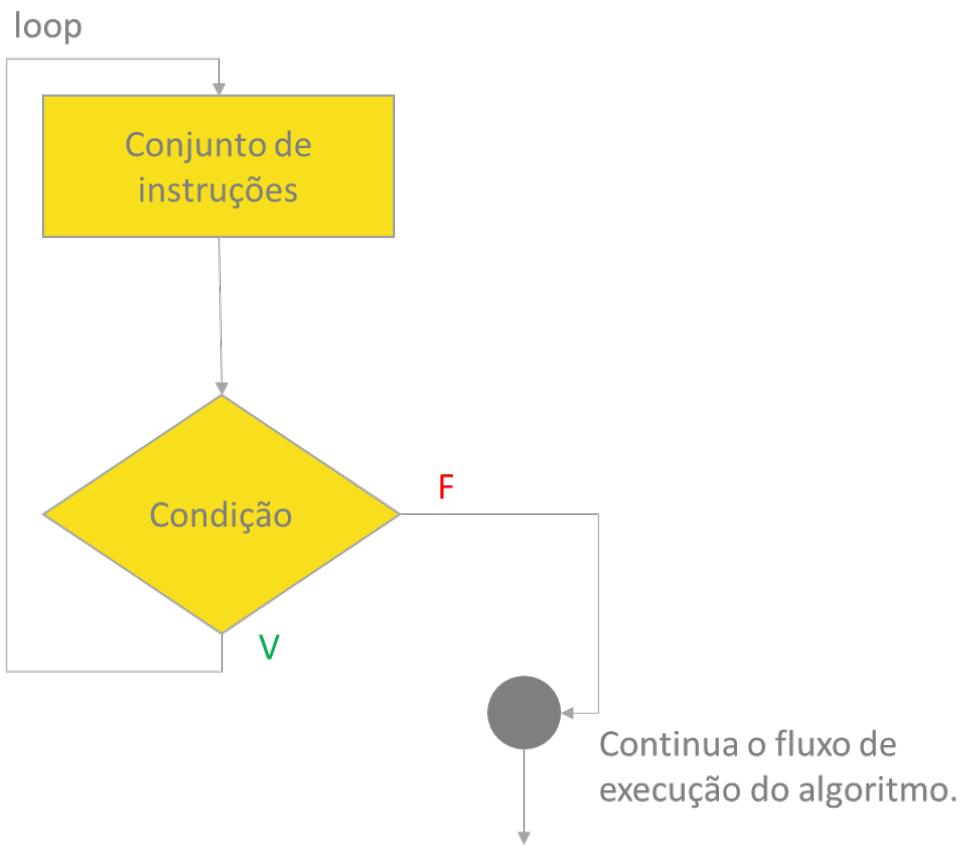
Valor = 9 main.js:14:13

Isso vai ser repetir até a variável assumir o valor -1, que então faz a condição retornar falso e encerrar a execução do laço.

Note, também, que a **instrução executada é sempre a mesma**, isso significa que a instrução **está sendo repetida por diversas vezes**.

Laço do-while

Esse tipo de estrutura de repetição é caracterizado por fazer o teste de controle no final do bloco de comandos. Os comandos repetidos são executados pelos menos uma vez antes da condição ser testada. A condição é sempre testada no final do laço e após a execução do bloco de comandos dele.



A diferença básica para o comando while é que o bloco a ser repetido sempre executa ao menos uma vez. **A sintaxe do do-while é:**

```
do{
  instrucao1;
  instrucao2;
  instrucao3;
  instrucaoN;
} while( condição );
proximaInstrucao;
```

O bloco de comandos é **executado** enquanto a condição for **verdadeira**.



Vamos Praticar

No arquivo, **main.js** digite o seguinte código.

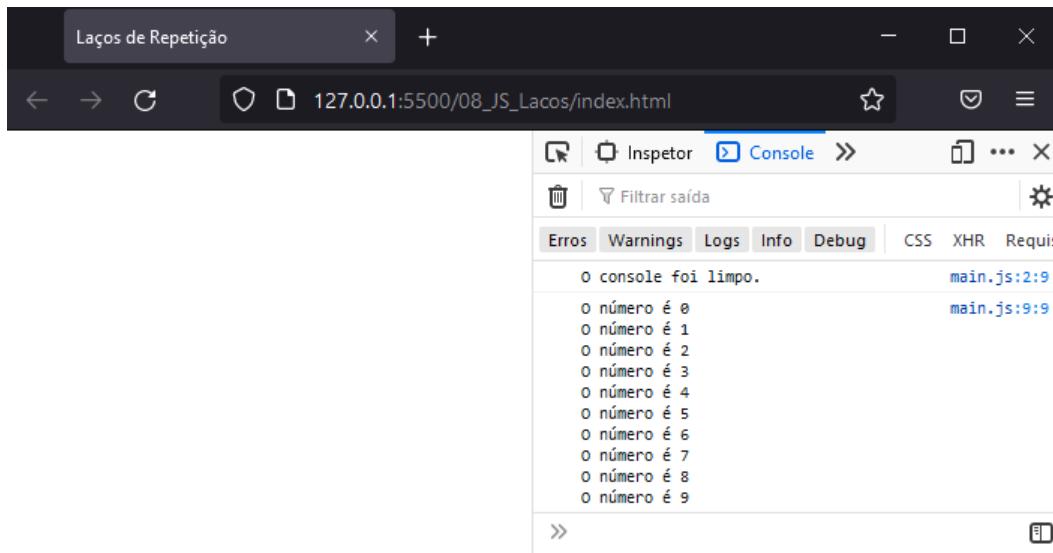
```
// do-while
console.clear();
let i = 0,
  text = '';
do {
```

```

text += `O número é ${i}\n`;
i++;
} while (i < 10);
console.log(text);

```

O resultado é mostrado no console do navegador.

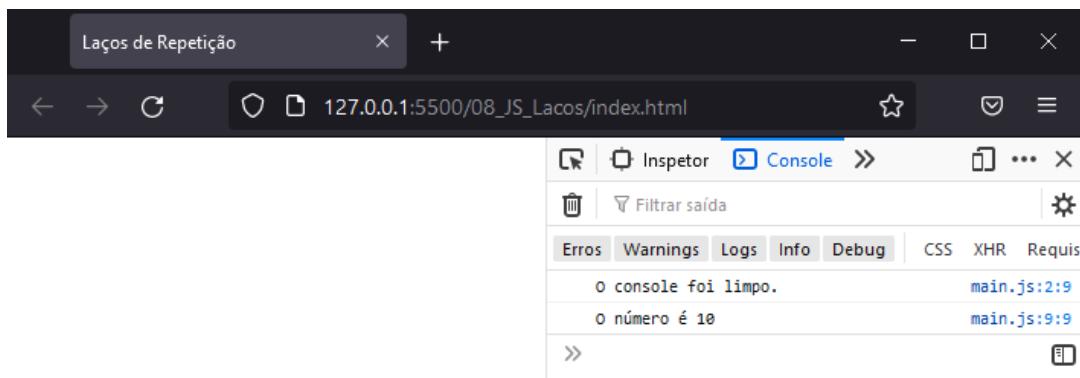


Observe que o **while** executa o bloco **pelo menos uma vez**, por exemplo se você alterar o valor de **i** para **10**. Ele irá **imprimir uma vez no console** e ao testar a **condição**, ela **retornará falso, encerando, assim, o laço**.

```

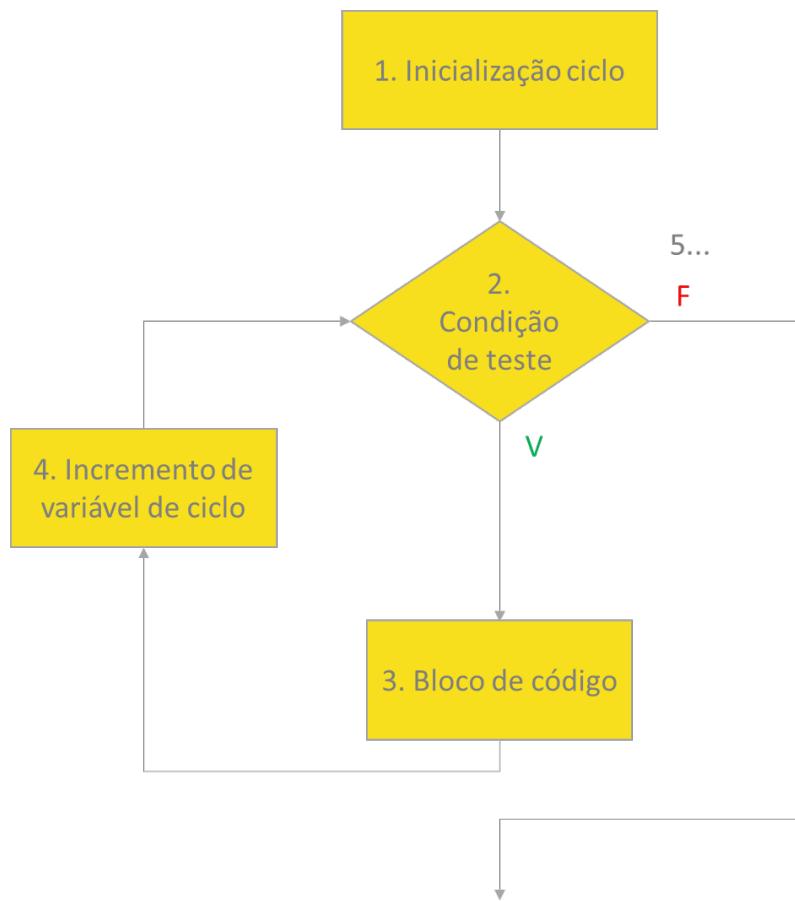
// do-while
console.clear();
let i = 10,
    text = '';
do {
    text += `O número é ${i}\n`;
    i++;
} while (i < 10);
console.log(text);

```



Laço For

O comando **for** também **realiza o teste lógico no início do laço**. Ele tem uma estrutura **um pouco diferente** do **while**, mas sua execução é similar. O programa **não executará** nenhuma repetição (**ações programadas**) sem antes **testar a condição**:



A sintaxe do comando **for** é:

```

1)           2)           3)
for(inicialização; condição; incremento){
    instrucao1;
    instrucao2;
    instrucaoN;
}
    proximaInstrucao;
  
```

- 1)** A **inicialização** é uma atribuição e é executada uma **única vez** antes do laço ser iniciado
- 2)** **Condição** que controla o laço. Repete o bloco (**{}**) enquanto a condição for verdadeira
- 3)** O **incremento** é sempre a última instrução do laço (ex: `cont++`)

As chaves podem ser ocultadas se o comando contiver apenas uma instrução:

```
for(inicializacao; condicao_testada; incremento ou decremento)
    instrucao;
```

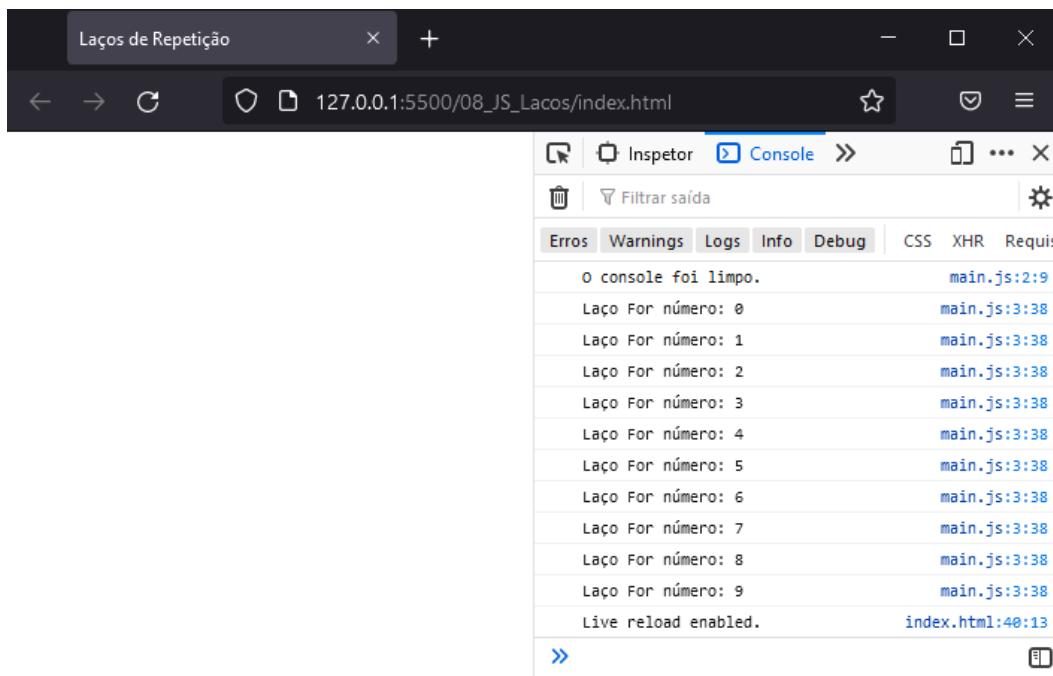


Vamos Praticar

No arquivo, **main.js** digite o seguinte código.

```
// For
console.clear();
for (let i = 0; i < 10; i++) console.log(`Laço For número: ${i}`);
```

O resultado é mostrado no console do navegador.



The screenshot shows a browser window with the address bar pointing to '127.0.0.1:5500/08_JS_Lacos/index.html'. Below the address bar is the developer tools interface with the 'Console' tab selected. The console output displays the following text:

```
O console foi limpo. main.js:2:9
Laço For número: 0 main.js:3:38
Laço For número: 1 main.js:3:38
Laço For número: 2 main.js:3:38
Laço For número: 3 main.js:3:38
Laço For número: 4 main.js:3:38
Laço For número: 5 main.js:3:38
Laço For número: 6 main.js:3:38
Laço For número: 7 main.js:3:38
Laço For número: 8 main.js:3:38
Laço For número: 9 main.js:3:38
Live reload enabled. index.html:40:13
```

No comando **for** temos:

Inicialização	Condição	Incremento
for (let i = 0;	i < 10;	i++)

O valor da **variável i** é **inicializada** com **0**, desse modo a **condição é verdadeira** e **mensagem é impressa pela primeira vez**:

Laço For número: 0 main.js:30:38

Em seguida, a **variável i** é **incrementada** de **uma unidade**, a condição é **testada** e **continua verdadeira, imprimindo para mensagem pela segunda vez**:

Laço For número: 1 main.js:30:38

Em seguida, a variável **i** é **incrementada** de uma unidade, a condição é **testada** e continua **verdadeira**, imprimindo para mensagem pela terceira vez:

Laço For número: 2

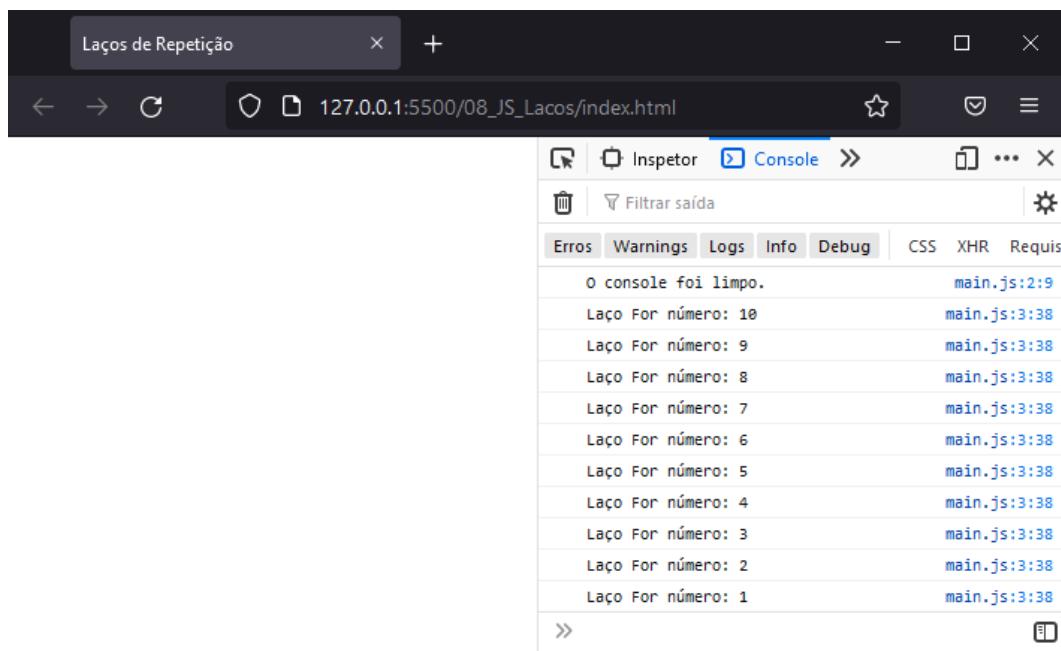
main.js:30:38

E isso vai se repetindo **até a variável i assumir o valor 10**, que torna a condição **falsa**, **encerrando assim o laço**.

Continuando a implementação do projeto, insira o seguinte código no arquivo **main.js**:

```
// For - Decremento
console.clear();
for (let i = 10; i > 0; i--) console.log(`Laço For número: ${i}`);
```

O resultado é mostrado no console do navegador:



Esse for faz a impressão **contrário** do anterior usado o **decremento da variável** de controle.

Inicialização	Condição	Decremento
for (let i = 10;	i > 0;	i--)

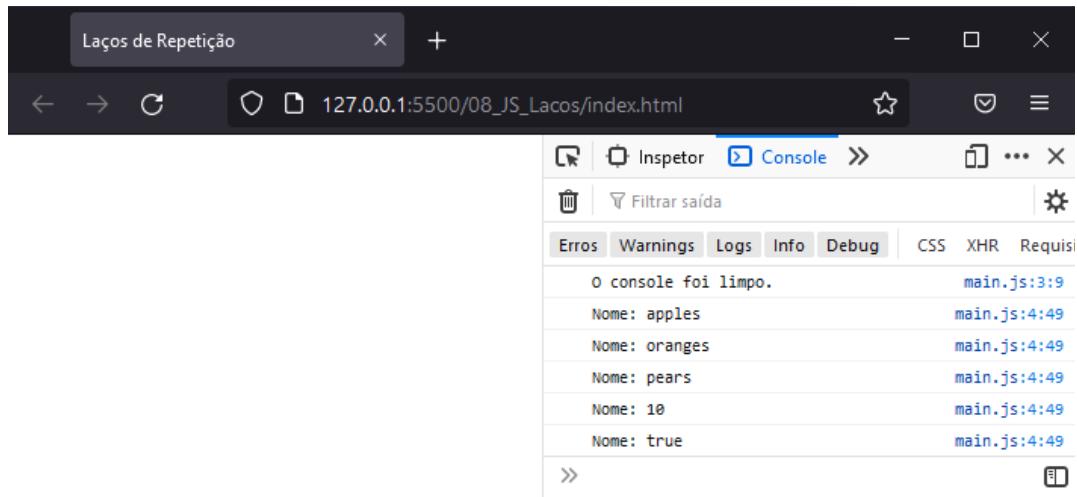
Laços de repetição e arrays

Utilizamos **muito laços de repetição** para **preencher** ou **percorrer array** ou **objetos de arrays** vamos ver os exemplos:

No arquivo, **main.js** digite o seguinte código.

```
// Laços de arrays
const frutas = ['apples', 'oranges', 'pears', 10, true];
console.clear();
for (let j = 0; j < frutas.length; j++) console.log(`Nome: ${frutas[j]}\n`);
```

O resultado é mostrado no console do navegador.



Como você pode ver, o **laço percorreu todos os elementos do array** para imprimir **cada um no console**.

Outro exemplo, no arquivo, **main.js** digite o seguinte código.

```
// Array de objetos e Laços
console.clear();
const todos = [
  {
    id: 1,
    text: 'Take out trash',
    isCompleted: true,
  },
  {
    id: 2,
    text: 'Meeting with boss',
    isCompleted: true,
  },
  {
```

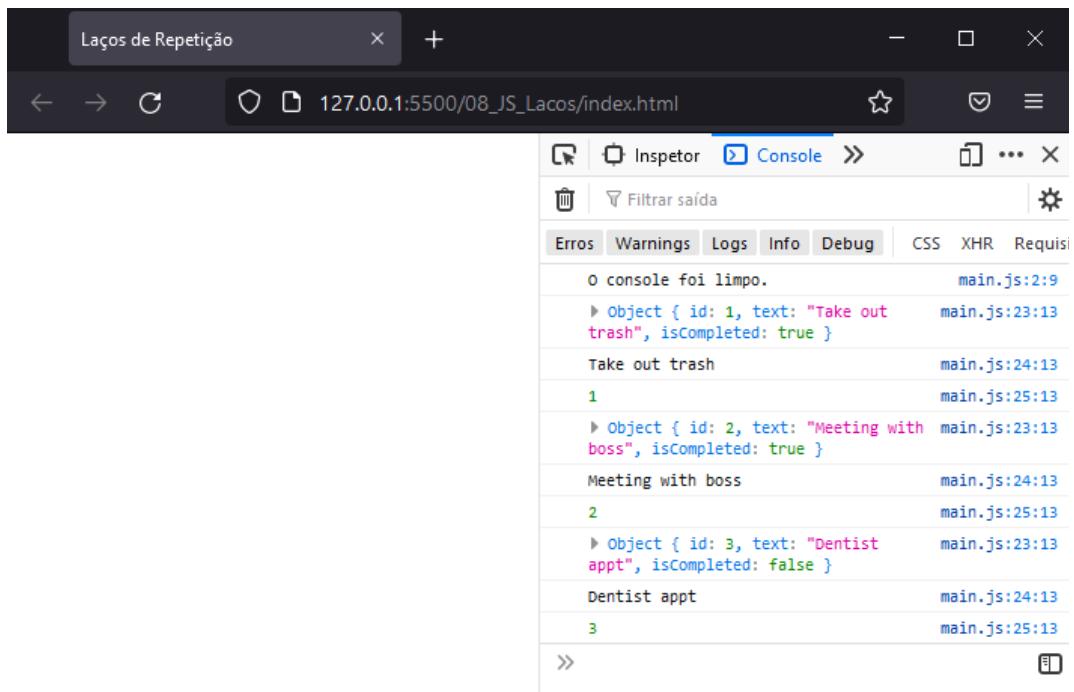
```

        id: 3,
        text: 'Dentist appt',
        isCompleted: false,
    },
];

// For overLoop
for (let t of todos) {
    console.log(t);
    console.log(t.text);
    console.log(t.id);
}

```

O resultado é mostrado no console do navegador.



```

0 console foi limpo. main.js:2:9
▶ Object { id: 1, text: "Take out trash", isCompleted: true } main.js:23:13
Take out trash main.js:24:13
1 main.js:25:13
▶ Object { id: 2, text: "Meeting with boss", isCompleted: true } main.js:23:13
Meeting with boss main.js:24:13
2 main.js:25:13
▶ Object { id: 3, text: "Dentist appt", isCompleted: false } main.js:23:13
Dentist appt main.js:24:13
3 main.js:25:13
>>

```

Nesse exemplo, usamos o chamado **for overloop**, que na **inicialização, condição e incremento** temos apenas a instrução **let t of todos**. Essa instrução do JavaScript é útil para **percorrer os índices de arrays**. Ela cria uma variável (**let t**), que irá **armazenar um objeto por vez de execução do loop** e vai começar no **primeiro objeto do array**, ir **incrementando até chegar no último e encerrar o laço**. Tudo isso é feito pelo interpretador do JS de forma automática.

Importante: essa é uma forma de percorrer um array que está sendo cada vez mais substituída pelos métodos de alto nível. Mas é interessante você ver isso para saber as diferenças e semelhanças de usar cada uma das técnicas.

Conclusão

Procure sempre fazer mais do que é passado para você em sala de aula. Por exemplo, você pode alterar as condições, utilizar comparações múltiplas nos laços, criar novos laços para ver o resultado que será gerado.

Seguem alguns links para você estudar e aprender mais:

<https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Statements/while>

<https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Statements/do...while>

<https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Statements/for>



Os objetivos desta aula são:

- Compreender o uso de arrays e sua sintaxe;
- Conhecer o conceito de arrays unidimensional e bidimensional;
- Aprender os diferentes métodos de arrays.

Arrays

Arrays são usados para armazenar múltiplos valores em uma única variável. Arrays podem ter mais de **uma ou mais dimensões** e, na literatura em português, é normal você encontrar o nome **vetor** para **referenciar array de uma dimensão e matrizes para array de duas dimensões**. Resumindo, você pode chamar tudo de array que qualquer pessoa que programa irá entender.

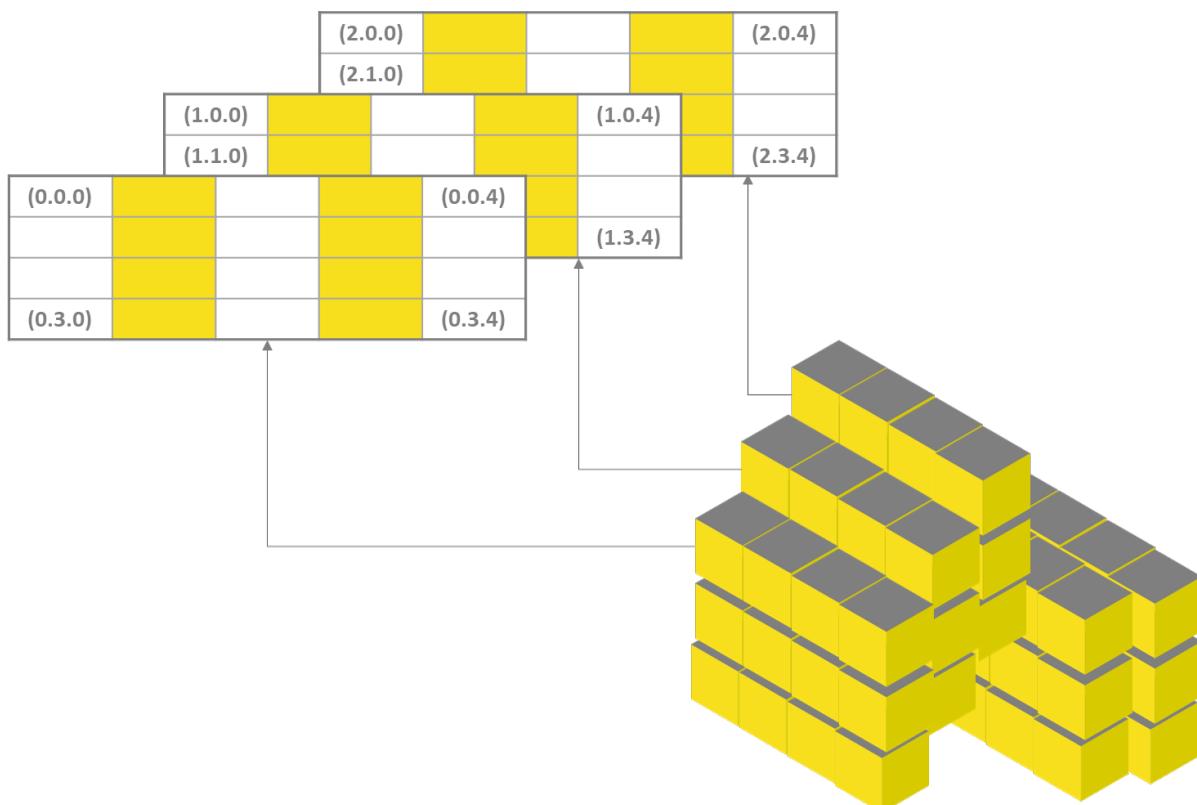
É importante saber que primeiro valor de um array possui índice zero (**0**), portanto para **array unidimensionais**, temos os seguintes índices:

(0)	(1)	(2)	(3)	(4)
-----	-----	-----	-----	-----

Para **arrays bidimensionais**, temos os seguintes índices:

(0.0)				(0.4)
(1.0)				
(3.0)				(3.4)

E para **arrays tridimensionais**, temos os seguintes índices:



Assim por diante. É muito importante que você saiba isso de cor, por não cometer erros na programação, quando for necessário preencher ou percorrer um array. Vamos ver isso na prática.



Vamos praticar

Siga os passos para criar o projeto:

Abra o **VS Code** e escolha um **diretório de trabalho para o seu projeto**.

Crie um diretório para seu projeto com o nome representativo, por exemplo, **JS_Arrays**.

Crie um arquivo dentro do diretório do projeto com o nome **index.html**.

Insira o seguinte código no seu arquivo **index.html**

```
<!DOCTYPE html>
<html lang="pt-br">
  <head>
    <meta charset="UTF-8" />
    <link rel="shortcut icon" href="#" />
    <title>Strings</title>
  </head>
  <body>
    <script src=".js/main.js"></script>
  </body>
</html>
```

Esse código mostra a marcação **<script>** sem nenhum código **JavaScript** entre a **abertura** e o **fechamento** da tag, apenas o atributo **src** com o valor **main.js**. Isso significa que o código JavaScript está em um arquivo externo. Portanto, temos que criar esse novo arquivo **main.js** dentro do diretório do projeto.

Observe que **mudamos a organização dos arquivos no projeto**, agora o arquivo **main.js** está em uma pasta chamada **js**, enquanto o arquivo **index.html** está no **diretório raiz do projeto**. Isso é importante para você ser uma pessoa organizada com os seus projetos. Além disso, o caminho do script agora está com apontando para o novo diretório (**<script src=".js/main.js"></script>**).

Vamos deixar o arquivo **main.js vazio** e à medida que vamos aprendendo coisas novas vamos completando a implementação do código **JavaScript**.

Abra o arquivo **index.html**, clique no botão  **Go Live** da extensão **Live Server** e abra as Ferramentas de desenvolvimento do navegador web (Atalho **F12**).

Agora, vamos implementar todo o nosso código desse tema no arquivo **main.js** e ver as mensagens impressas no console.

Array unidimensional

Vamos ver como acessar **elementos armazenados em um array unidimensional**. No arquivo, **main.js** digite o seguinte código.

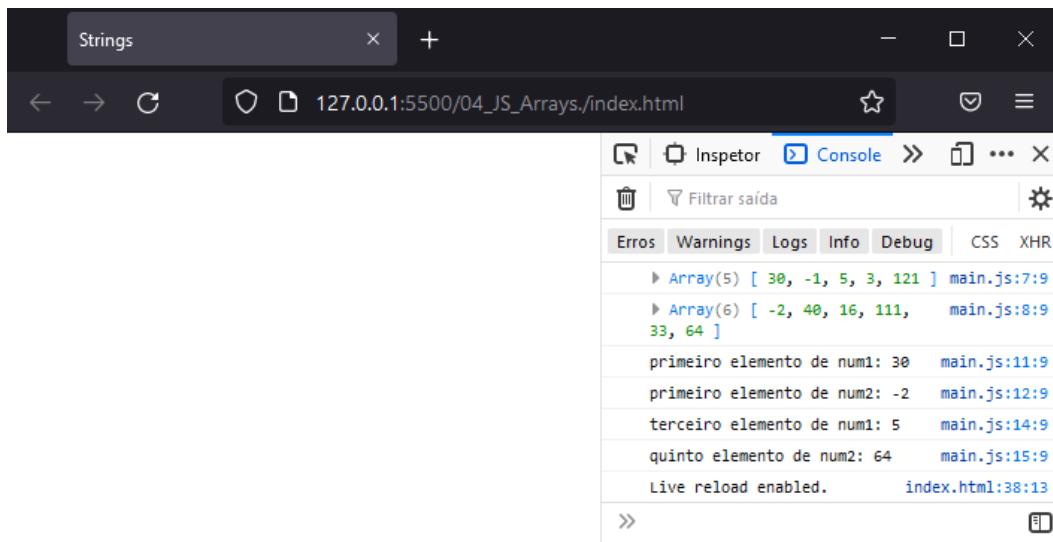
```
//Arrays unidimensionais
// Podemos criar array assim
const num1 = new Array(30, -1, 5, 3, 121);
// Ou podemos fazer
const num2 = [-2, 40, 16, 111, 33, 64];

console.log(num1);
console.log(num2);

// Acessando elementos diversos dos arrays
console.log(`primeiro elemento de num1: ${num1[0]}`);
console.log(`primeiro elemento de num2: ${num2[0]}`);

console.log(`terceiro elemento de num1: ${num1[2]}`);
console.log(`quinto elemento de num2: ${num2[5]}`);
```

Antes de explicar o que foi programado vamos ver o resultado mostrado no console.



As instruções usadas **criam o mesmo tipo de array**, a diferença é que a **primeira pode exigir mais recurso computacional**, portanto é **mais lenta**, e a **segunda é mais prática e mais utilizada na maioria dos casos**.

As instruções usadas **criam um array de uma linha e várias colunas**, a primeira `const num1 = new Array(30, -1, 5, 3, 121)` cria um array com **5 colunas**, enquanto a segunda `const num2 = [-2, 40, 16, 111, 33, 64]` cria um array com **6 colunas**. Observe que as **colunas são definidas pela vírgula** que está entre os elementos.

Olha que interessante, como foi falando no início do tema, para acessar o primeiro elemento devemos indicar o **índice zero**.

Arrays bidimensionais

Agora vamos ver como trabalhar com **arrays bidimensionais**. Nesse caso, o primeiro elemento vai estar na posição **(0, 0)**. Continuando a implementação do projeto, insira o seguinte código no arquivo main.js:

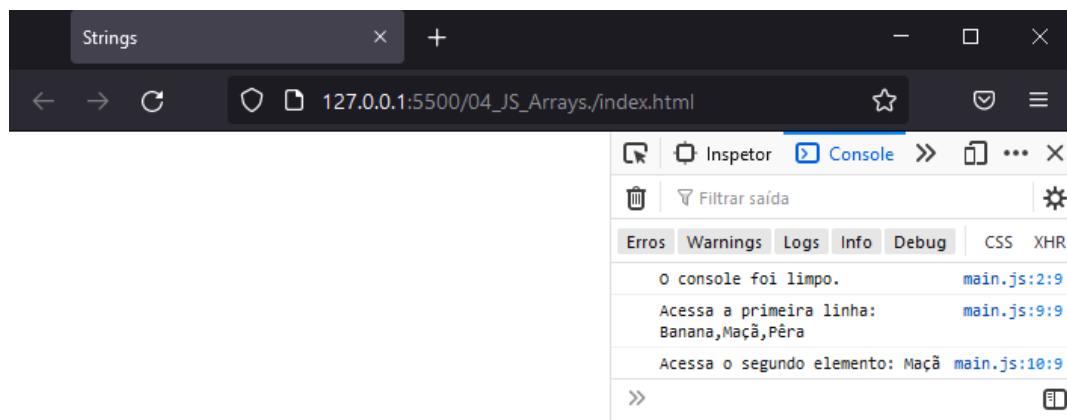
```
//Arrays bidimensionais
console.clear();
const matrix = [
  ['Banana', 'Maçã', 'Pêra'],
  ['Laranja', true, 1],
  [null, 'Uva', -350],
];
// Acessando elementos diversos dos arrays
console.log(`Acessa a primeira linha: ${matrix[0]}`);
console.log(`Acessa o segundo elemento: ${matrix[0][1]}`);
```

Você pode notar que o **array em JS não precisa armazenar somente o mesmo tipo**, no array **matrix**, que foi criado, **temos tipos literais, numéricos, booleanos ,etc.**

Você deve notar também, que **um array bidimensional** possui **diversos array dentro dele**, por isso **é necessário que os elementos de cada linha devam estar dentro de colchetes []**.

Outro ponto importante é a instrução **matrix[0]** acessa **toda a linha zero do array**, ou seja, quando especificamos **uma dimensão em um array bidimensional**, estamos buscando a **linha inteiro do array**. Portanto, **para acessar o elemento**, devemos **indicar as duas dimensões**: **matrix[0][1]**.

O resultado é mostrado no console do navegador:



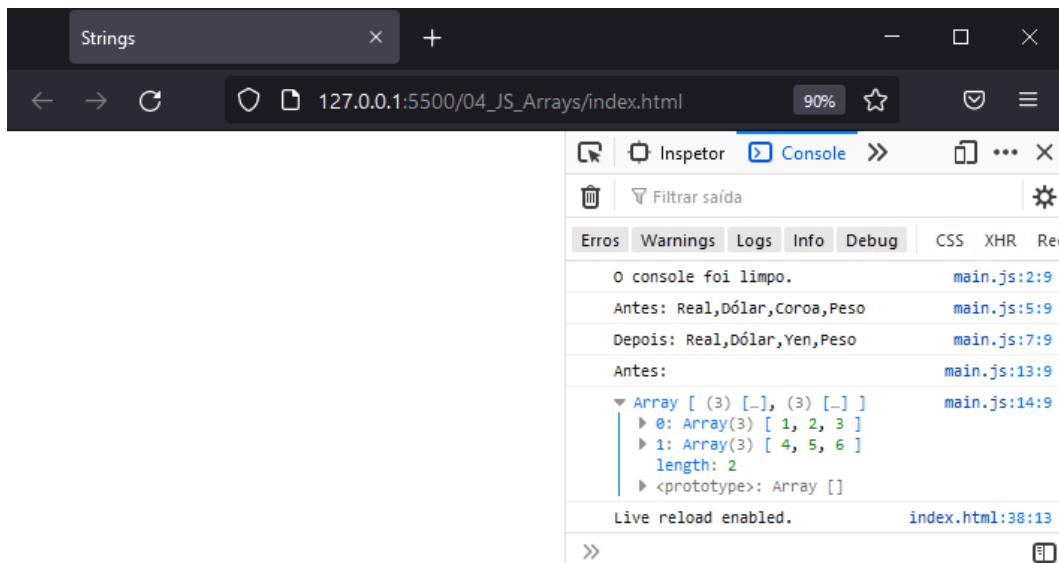
Para **alterar algum valor é bem simples**, basta **indicarmos o índice do elemento que queremos alterar e igualar ao valor desejado**. Continuando a implementação do projeto, insira o seguinte código no arquivo main.js:

```
// Alterando um valor do array
console.clear();
let moedas = ['Real', 'Dólar', 'Coroa', 'Peso'];

console.log(`Antes: ${moedas}`);
moedas[2] = 'Yen';
console.log(`Depois: ${moedas}`);

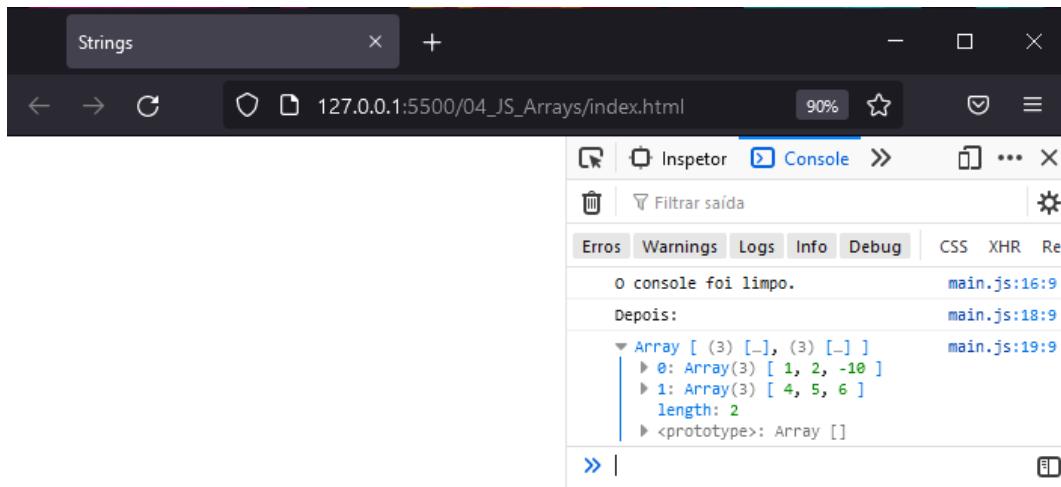
let matrix1 = [
  [1, 2, 3],
  [4, 5, 6],
];
console.log('Antes:');
console.log(matrix1);
```

A instrução `moedas[2] = 'Yen'` muda o valor terceiro elemento do array unidimensional **moeda**.



```
// Alterando um valor do array
console.clear();
matrix1[0][2] = -10;
console.log('Depois:');
console.log(matrix1);
```

Enquanto, a instrução `matrix1[0][2] = -10` altera elemento (0, 2) do array bidimensional **matrix1**.



Métodos de arrays

Vamos aprender sobre os métodos de arrays continuando a nossa programação no arquivo **main.js**.

Convertendo Array em String

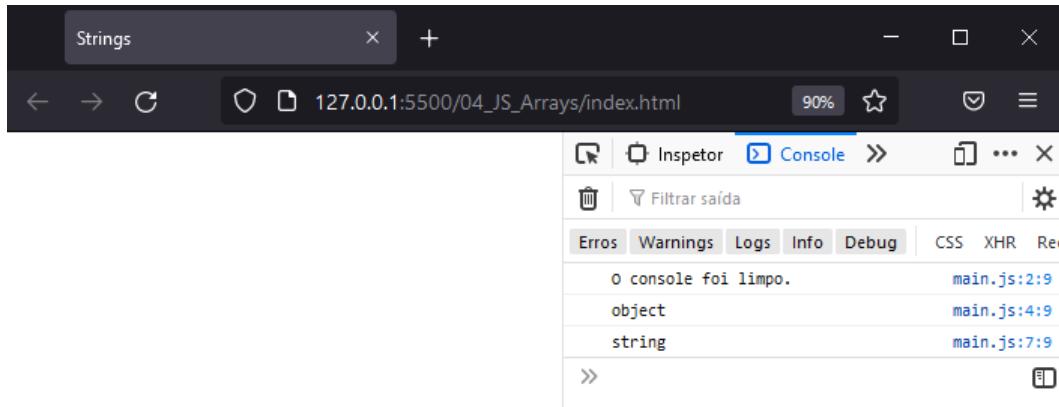
O método **toString()** converte **um array em uma string**, onde os elementos do **array são separados por vírgulas**. Vamos ver isso na prática, siga os passos para continuar a implementação:

Continuando a implementação do projeto, insira o seguinte código no arquivo **main.js**:

```
//Métodos em Arrays
console.clear();
let numArray = [1, 2, 3, 4];
console.log(typeof numArray);

let numArray2 = numArray.toString();
console.log(typeof numArray2);
```

O resultado é mostrado no console do navegador:



Observe que o **tipo de dados do primeiro array**, que **foi impresso no console**, é um **objeto**, enquanto o **tipo de dados do segundo é uma string**.

Observe também que para acessar um **método em JavaScript** colocamos o **nome_do_objeto** depois **ponto final** e depois o nome o **método**:

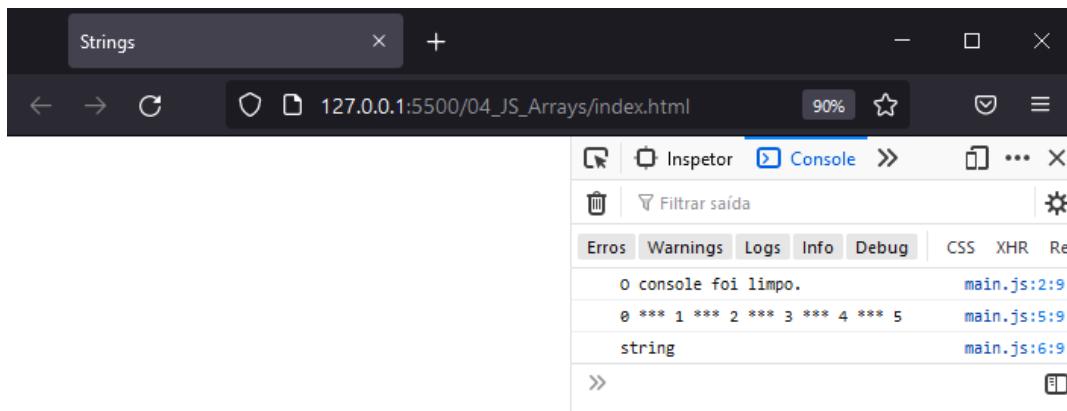
numArray	.	toString()
nome_do_objeto	ponto final	nome do método

Existe também o método **join()** que **converte o array em string**, mas você pode colocar qualquer caractere como separador dos elementos. **Vamos ver na prática:**

Continuando a implementação do projeto, insira o seguinte código no arquivo **main.js**:

```
// join()
console.clear();
let teste1 = [0, 1, 2, 3, 4, 5];
let teste2 = teste1.join(' *** ');
console.log(teste2);
console.log(typeof teste2);
```

O resultado é mostrado no console do navegador:



O resultado impresso no console mostra o separador personalizado inserido entre os elementos do array.

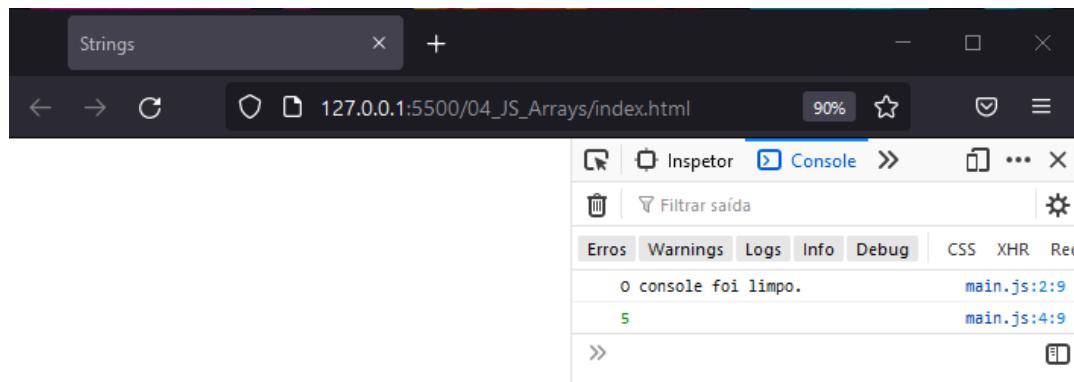
Propriedade: Tamanho do array

A propriedade **length** retorna **tamanho do array**, ou seja, **a quantidade de elementos que esse array possui**. Vamos ver isso na prática, siga os passos para continuar a implementação:

Continuando a implementação do projeto, insira o seguinte código no arquivo **main.js**:

```
//Tamanho de Arrays
console.clear();
let array = [0, 1, 2, 3, 4];
console.log(array.length);
```

O resultado é mostrado no console do navegador:



Observe que o comando **array.length** retorna a quantidade de elementos do array, que são 5.

Inserir e remover elemento no array

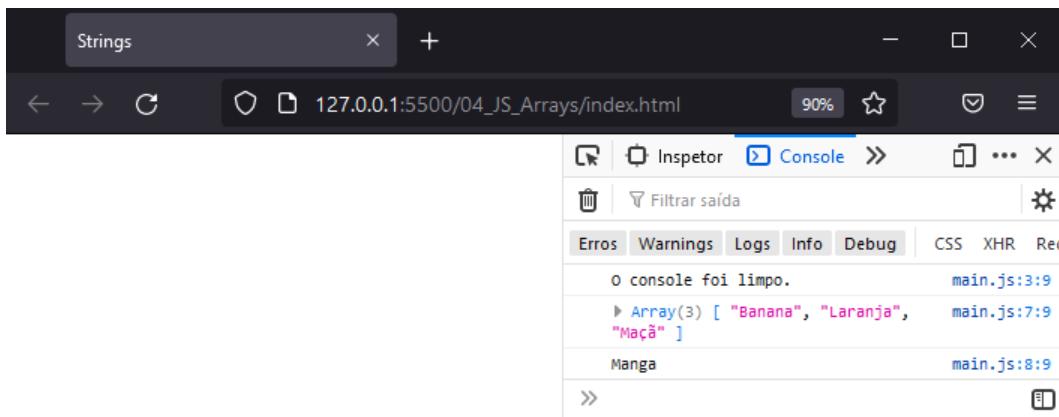
O método **pop()** remove o último elemento do array e retorna esse elemento retirado, que pode ser armazenado em uma variável. Vamos ver isso na prática, siga os passos para continuar a implementação:

Continuando a implementação do projeto, insira o seguinte código no arquivo **main.js**:

```
// Remover e inserir elementos
// pop()
console.clear();
const frutas1 = ['Banana', 'Laranja', 'Maçã', 'Manga'];
let x = frutas1.pop(); // x = "Manga"

console.log(frutas1);
console.log(x);
```

O resultado é mostrado no console do navegador:



O **array criado inicialmente tinha a fruta Manga**, após a execução da instrução `frutas1.pop()`, esse **último elemento foi retirado**. A variável `x` foi usada para vermos o elemento retirado sendo armazenado em algum lugar.

Já o método **push()** insere um elemento no final do array e retorna o tamanho do novo array, que **pode ser armazenado em uma variável**. Vamos ver isso na prática, siga os passos para continuar a implementação:

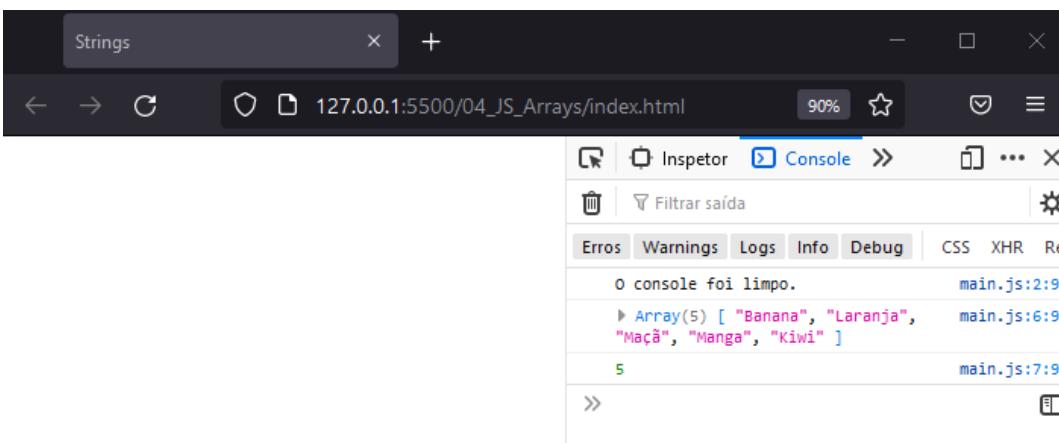
Continuando a implementação do projeto, insira o seguinte código no arquivo **main.js**:

```

// push()
console.clear();
const frutas2 = ['Banana', 'Laranja', 'Maçã', 'Manga'];
let y = frutas2.push('Kiwi'); // y = 5

console.log(frutas2);
console.log(y);
  
```

O resultado é mostrado no console do navegador:



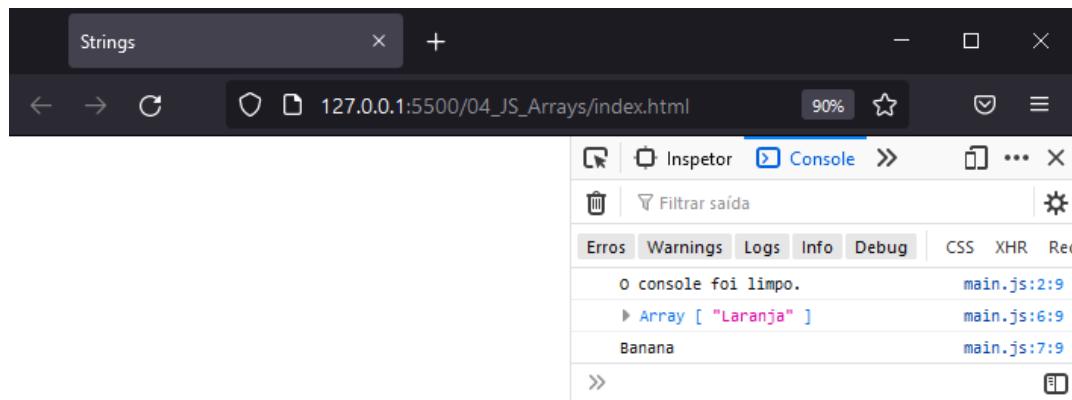
O **array criado inicialmente não continha a fruta Kiwi**, após a execução da instrução `frutas2.push('Kiwi')`, esse elemento foi **adicionado na última posição**. A variável **y** foi usada para vermos o **tamanho do novo array, agora com o elemento adicionado**.

Por sua vez, o método **shift()** remove o elemento no início do array deslocando todos os elementos em uma posição e retorna esse elemento retirado, que pode ser armazenado em uma variável. Vamos ver isso na prática, siga os passos para continuar a implementação:

Continuando a implementação do projeto, insira o seguinte código no arquivo **main.js**:

```
// shift()
console.clear();
const frutas3 = ['Banana', 'Laranja'];
let z = frutas3.shift(); // z = 'Banana'
console.log(frutas3);
console.log(z);
```

O resultado é mostrado no console do navegador:



O **array originalmente tinha dois elementos** e depois da execução da instrução `frutas3.shift()` **passou a ter um**. A variável **z** foi usada para **vermos o elemento retirado sendo armazenado em algum lugar**.

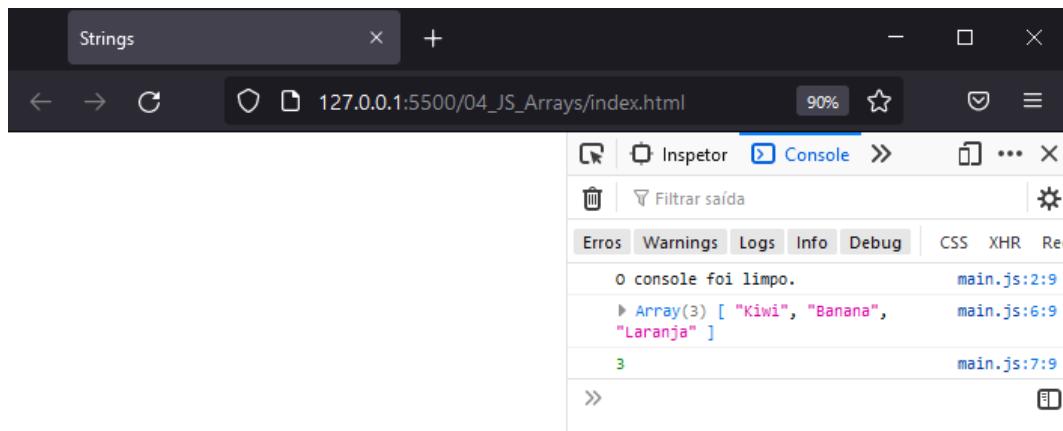
Por fim, o método **unshift()** insere um elemento no início do array e retorna o tamanho do novo array, que pode ser armazenado em uma variável. Vamos ver isso na prática, siga os passos para continuar a implementação:

Continuando a implementação do projeto, insira o seguinte código no arquivo **main.js**:

```
// unshift()
console.clear();
const frutas4 = ['Banana', 'Laranja'];
let w = frutas4.unshift('Kiwi'); // w = 3

console.log(frutas4);
console.log(w);
```

O resultado é mostrado no console do navegador:



The screenshot shows a browser window with the URL 127.0.0.1:5500/04_JS_Arrays/index.html. The developer tools console tab is active. The output in the console is:

```

O console foi limpo. main.js:2:9
▶ Array(3) [ "Kiwi", "Banana", "Laranja" ] main.js:6:9
3 main.js:7:9
»

```

Observe que os **elementos do array forma deslocados para inserir o elemento Kiwi.** após a execução da instrução `frutas4.unshift('Kiwi')`, esse elemento foi **adicionado na primeira posição.** A **variável** foi usada para **vermos o tamanho do novo array, agora com o elemento adicionado.**

Apagando um elemento em uma posição específica

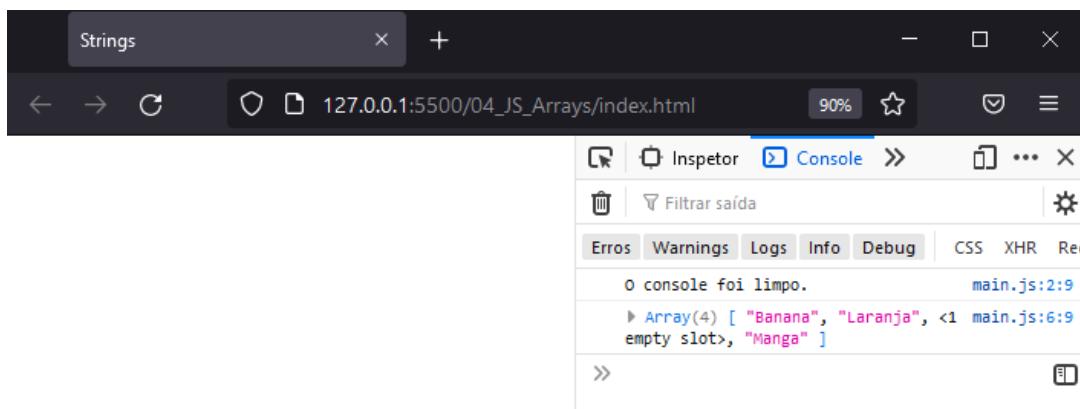
O comando **delete apaga um elemento na posição desejada e altera para slot vazio**, que é o mesmo de **`undefined`**. Vamos ver isso na prática, siga os passos para continuar a implementação:

Continuando a implementação do projeto, insira o seguinte código no arquivo **main.js**:

```
// delete
console.clear();
const frutas5 = ['Banana', 'Laranja', 'Maçã', 'Manga'];
delete frutas5[2];

console.log(frutas5);
```

O resultado é mostrado no console do navegador:



The screenshot shows a browser window with the URL 127.0.0.1:5500/04_JS_Arrays/index.html. The developer tools console tab is active. The output in the console is:

```

O console foi limpo. main.js:2:9
▶ Array(4) [ "Banana", "Laranja", <1 empty slot>, "Manga" ] main.js:6:9
»

```

Note que o **elemento Maçã**, que **existe quando o array foi criado**, foi **substituído por <1 empty slot>** após a execução da instrução `delete frutas5[2]`.

Conclusão

Existem muitos outros métodos para manipular arrays e gastaríamos muito tempo se quisessem abordar todos esses métodos. À medida que outros métodos forem aparecendo, vamos explicando cada uma desses. O importante é que você sempre deve usar a internet para procurar soluções em programação, mas não apenas copie e cole uma solução, um bom programador procura entender a solução proposta e adaptá-la no seu projeto.

Para saber mais: Você pode consultar a lista de métodos de strings nos links:

https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Array

https://www.w3schools.com/js/js_array_methods.asp

Como desafio, tente entender e implementar alguns desses métodos.



Array de Objetos

Os objetivos desta aula são:

- Compreender o uso de objetos e sua sintaxe;
- Conhecer o formato JSON.

Objetos

Vimos que **arrays** são um **conjunto de valores** que **não precisam ser do mesmo tipo** e que **objetos são uma coleção de dados e/ou funcionalidades relacionadas**. Objetos, geralmente, são construídos com **pares chave: valor**, onde a **chave** é um rótulo escolhido pela pessoa programadora e o **valor** é o conteúdo armazenado nesse rótulo. Por exemplo:

Sintaxe geral	Exemplo 01:	Exemplo 02:
<pre>const ou let nome_do_objeto = { chave01: valor01, chave02: valor02, chave03: valor03, ... chaveXX: valorXX, }</pre>	<pre>const carro = { tipo:"Fiat", modelo:"500", cor:"branco" };</pre>	<pre>let pessoa = { firstName: "Irmão do", lastName: "Jorel", idade: 10, corDosOlhos: "preto" };</pre>

Você deve observar que o **nome da chave** pode ser **qualquer um** desde que **não contenha espaços ou caracteres especiais**. Pode ser em qualquer língua: português, inglês, alemão, sueco, etc. Mas, lembre-se que, ser usar o **nome em português não colocar acentuação gráfica. Isso é uma regra obrigatória.**

Outra observação é a **convenção de camelCase**, que comumente usada por pessoas que programam para o nome da chave com **duas palavras ou mais palavras**, exemplo: **firstName**, **corDosOlhos**, **nomePesssoa**, etc. Isso é uma convenção não uma regra obrigatória.



Vamos praticar

Siga os passos para criar o projeto:

Abra o VS Code e escolha um diretório de trabalho para o seu projeto.

Abra o **VS Code** e escolha um **diretório de trabalho para o seu projeto**.

Crie um diretório para seu projeto com o nome representativo, por exemplo, **JS_Array_Objetos**.

Crie um arquivo dentro do diretório do projeto com o nome **index.html**.

Insira o seguinte código no seu arquivo **index.html**.

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

<head>
    <meta charset="UTF-8" />
    <link rel="shortcut icon" href="#" />
    <title>Objetos e Array de Objetos</title>
</head>

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

</html>
```

Esse código mostra a marcação **<script>** sem nenhum código **JavaScript** entre a **abertura** e o **fechamento** da tag, apenas o atributo **src** com o valor **main.js**. Isso significa que o código **JavaScript** está em um arquivo externo. Portanto, temos que criar esse novo arquivo **main.js** dentro do diretório do projeto.

Observe que **mudamos a organização dos arquivos no projeto**, agora o arquivo **main.js** está em uma pasta chamada **js**, enquanto o arquivo **index.html** está no **diretório raiz do projeto**. Isso é importante para você ser uma pessoa organizada com os seus projetos. Além disso, o caminho do script agora está com apontando para o novo diretório (**<script src=".js/main.js"></script>**).

Vamos deixar o arquivo **main.js vazio** e à medida que vamos aprendendo coisas novas vamos completando a implementação do código **JavaScript**.

Abra o arquivo **index.html**, clique no botão  **Go Live** da extensão **Live Server** e abra as Ferramentas de desenvolvimento do navegador web (Atalho **F12**).

Agora, vamos implementar todo o nosso código desse tema no arquivo **main.js** e ver as mensagens impressas no console.

Vamos como utilizar **objetos** em **JS**. No arquivo, **main.js** digite o seguinte código.

```
// Objetos
let pessoa = {
    firstName: 'Irmão do',
    lastName: 'Jorel',
    idade: 10,
    corDosOlhos: 'preto',
    hobbies: ['música', 'filmes', 'esportes'],
    endereco: {
        rua: 'Rua do bobos',
        numero: 0,
```

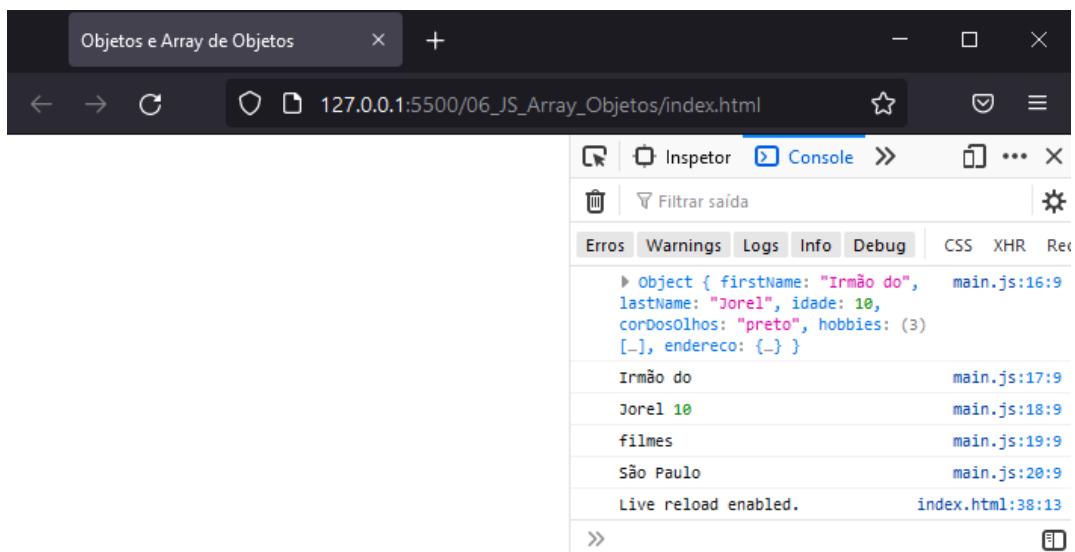
```

        cidade: 'São Paulo',
        estado: 'SP',
    },
};

console.log(pessoa);
console.log(pessoa.firstName);
console.log(pessoa.lastName, pessoa.idade);
console.log(pessoa.hobbies[1]);
console.log(pessoa.endereco.cidade);

```

Antes de explicar o que foi programado vamos ver o resultado mostrado no console.



O comando **console.log(pessoa);** imprime **todo o objeto**, podemos expandir e todos os pares chave: valor do objeto.

```

▼ Object { firstName: "Irmão do", lastName: "Jorel", idade: 10, corDosOlhos: "preto", hobbies: (3) [...], endereco: {...} }
  ↳ corDosOlhos: "preto"
  ↳ endereco: Object { rua: "Rua do bobos", numero: 0, cidade: "São Paulo", ... }
    ↳ firstName: "Irmão do"
  ↳ hobbies: Array(3) [ "música", "filmes", "esportes" ]
    ↳ idade: 10
    ↳ lastName: "Jorel"
  ↳ <prototype>: Object { ... }

```

Observe que a linha mostrada em **main.js 16:9**, o 16 é a linha da instrução que executou e resultou na impressão do log console.

Em seguida, acessamos apenas o **campo/chave firstName** do objeto e o resultado é:

Irmão do

main.js:17:9

Observe que a linha mostrada em **main.js 17:9**, o **17** é a linha da instrução que executou e resultou na impressão do log console.

Depois, acessamos apenas o **campo/chave lastName** do objeto e a **idade** do objeto e o resultado é:

Jorel 10

main.js:18:9

Observe que a linha mostrada em **main.js 18:9**, o **18** é a linha da instrução que executou e resultou na impressão do log console.

Após, acessamos apenas o **segundo elemento** do **campo/chave hobbies** do objeto e o resultado é:

filmes

main.js:19:9

Observe que a linha mostrada em **main.js 18:9**, o **18** é a linha da instrução que executou e resultou na impressão do log console.

E por último, acessamos apenas o **campo/chave cidade** do objeto **endereco** que está dentro do nosso objeto **pessoa** e o resultado é:

São Paulo

main.js:20:9

Observe que a linha mostrada em **main.js 20:9**, o **20** é a linha da instrução que executou e resultou na impressão do log console.

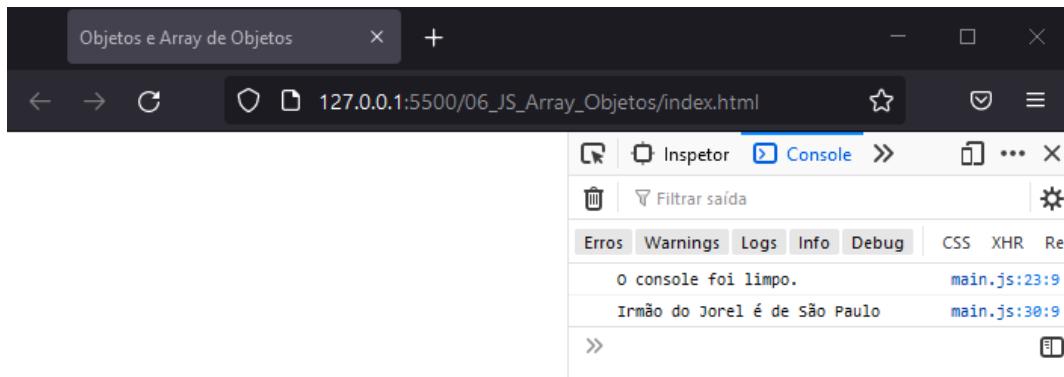
Podemos abordar, agora, um assunto bem legal, que é **atribuição via desestruturação (destructuring assignment)**. O JavaScript permite **extrair os dados de um array ou um objeto e armazenar em variáveis simples** através da **atribuição via desestruturação (destructuring assignment)**. Mas como isso funciona? Vamos ver na prática.

Continuando a implementação do projeto, insira o seguinte código no arquivo **main.js**:

```
// Atribuição via desestruturação
console.clear();

const {
  firstName,
  lastName,
  endereco: { cidade },
} = pessoa;
console.log(` ${firstName} ${lastName} é de ${cidade}`);
```

O resultado é mostrado no console do navegador:



Observe que foi possível usar apenas o **nome da variável**, que é mais curto, para acessar o conteúdo para ser impresso. Isso só foi possível por causa da **atribuição via desestruturação, que foi feita**:

```
const {
  firstName,
  lastName,
  endereco: { cidade },
} = pessoa;
```

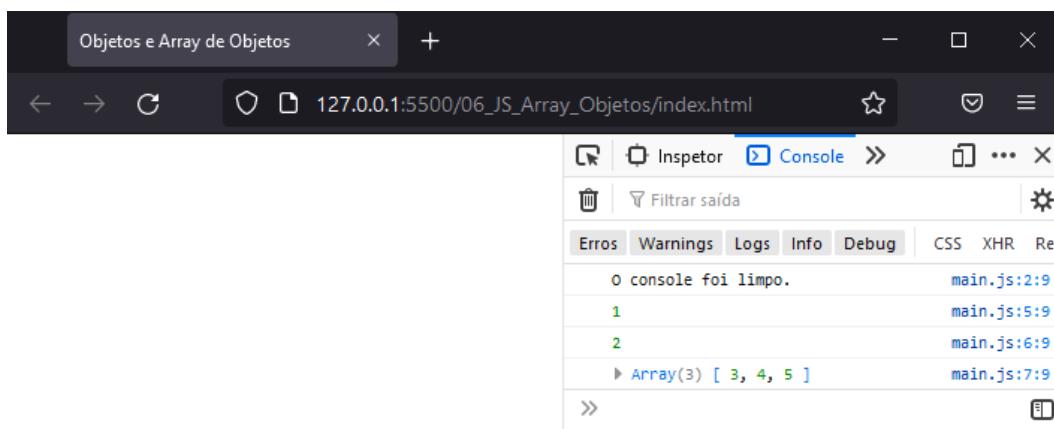
Importante: Para a atribuição via desestruturação ocorrer corretamente em objetos, você deve sempre usar os nomes das variáveis iguais as chaves que você deseja extrair. Caso contrário o conteúdo será undefined. Sabia bem esse assunto, isso é muito utilizado em JS.

Já em arrays a **atribuição via desestruturação (destructuring assignment)** funciona um pouco diferente. Basicamente, é só você igualar um vetor a outro.

Continuando a implementação do projeto, insira o seguinte código no arquivo **main.js**:

```
// Atribuição via desestruturação com arrays
console.clear();
const array = [1, 2, 3, 4, 5];
let [valor01, valor02, ...resto] = array;
console.log(valor01);
console.log(valor02);
console.log(resto);
```

O resultado é mostrado no console do navegador:



Observe que com **arrays** só foi necessário fazer uma **igualdade entre eles**:

```
let [valor01, valor02, ...resto] = array;
```

Esse é um array, observe os colchetes	Igualdade	Esse também é um array
[valor01, valor02, ...resto]	=	array

Você deve observar também o operador **rest** (`...`), que permite trabalhar com **múltiplos parâmetros de arrays e funções**. O **rest** é utilizado para extrair **múltiplos valores do array** ou para a função receber uma **quantidade indefinida de valores**.

No exemplo, o primeiro elemento do **array** é armazenado na variável **valor01**, o segundo elemento do **array** é armazenado na variável **valor02** e os **demais (a partir do terceiro elemento)** foram armazenados na variável **resto**, que é um **array também** como mostra no console:

▶ Array(3) [3, 4, 5] main.js:38:9

Você pode testar os seguintes comandos:

```
let [valorA, valorB, valorC ...restante] = array;
console.log(valorA);
console.log(valorB);
console.log(valorC);
console.log(restante);
```

E verá que a variável **restante** irá armazenar o conteúdo a partir do **quarto elemento até o final** do array.

Array de objetos

Do mesmo jeito temos **arrays** com **valores diversos**, podemos ter **arrays de objetos**. Vamos ver isso na prática.



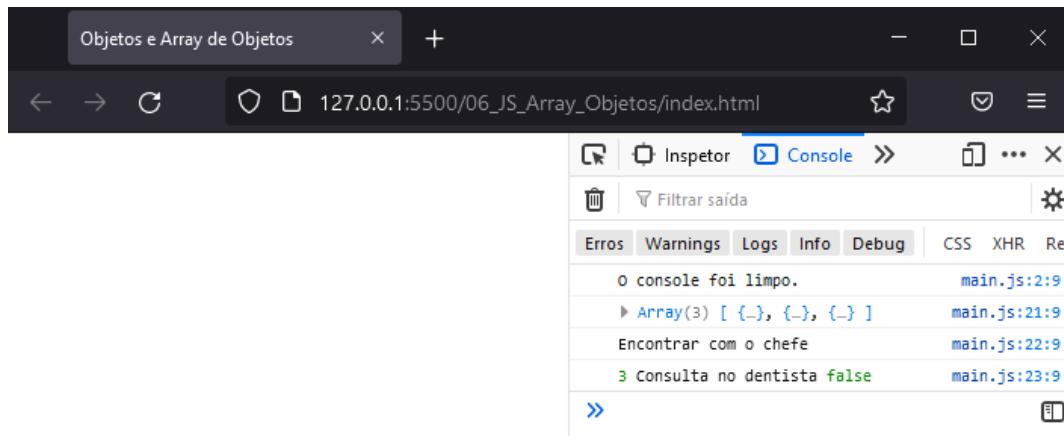
Vamos praticar

Siga os passos para continuar a implementação no arquivo **main.js**:

Continuando a implementação do projeto, insira o seguinte código no arquivo **main.js**:

```
// Array de objetos
console.clear();
const tarefas = [
  {
    id: 1,
    texto: 'Levar o lixo para fora',
    isCompleted: true,
  },
  {
    id: 2,
    texto: 'Encontrar com o chefe',
    isCompleted: true,
  },
  {
    id: 3,
    texto: 'Consulta no dentista',
    isCompleted: false,
  },
];
console.log(tarefas);
console.log(tarefas[1].texto);
console.log(tarefas[2].id, tarefas[2].texto, tarefas[2].isCompleted);
```

O resultado é mostrado no console do navegador:



Você pode ver que o array **tarefas** possui **diversos objetos dentro dele**. E para acessar a cada objeto você deve **referenciar o índice do mesmo**. Por exemplo, quando acessamos **todo o array** e o resultado é:

```

▼ Array(3) [ {…}, {…}, {…} ]           main.js:60:9
  ▶ 0: Object { id: 1, texto: "Levar o lixo para
    fora", isCompleted: true }
  ▶ 1: Object { id: 2, texto: "Encontrar com o
    chefe", isCompleted: true }
  ▶ 2: Object { id: 3, texto: "Consulta no
    dentista", isCompleted: false }
    length: 3
  ▶ <prototype>: Array []

```

Em seguida, acessamos o **campo/chave** texto do **segundo objeto do array** e o resultado é:

Encontrar com o chefe main.js:61:9

Depois, acessamos os **campos/chaves** do **terceiro objeto do array** e o resultado é:

3 Consulta no dentista false main.js:62:9

JSON

JSON (**JavaScript Object Notation**) é um formato **compacto**, de **padrão aberto** e **independente** usado para a **troca de dados entre sistemas de forma simples e rápida**. Ele utiliza **texto legível a humanos**, no formato **atributo-valor** (natureza auto-descritiva). Isto é, um **modelo de transmissão de informações no formato texto**, muito usado em serviços na internet.



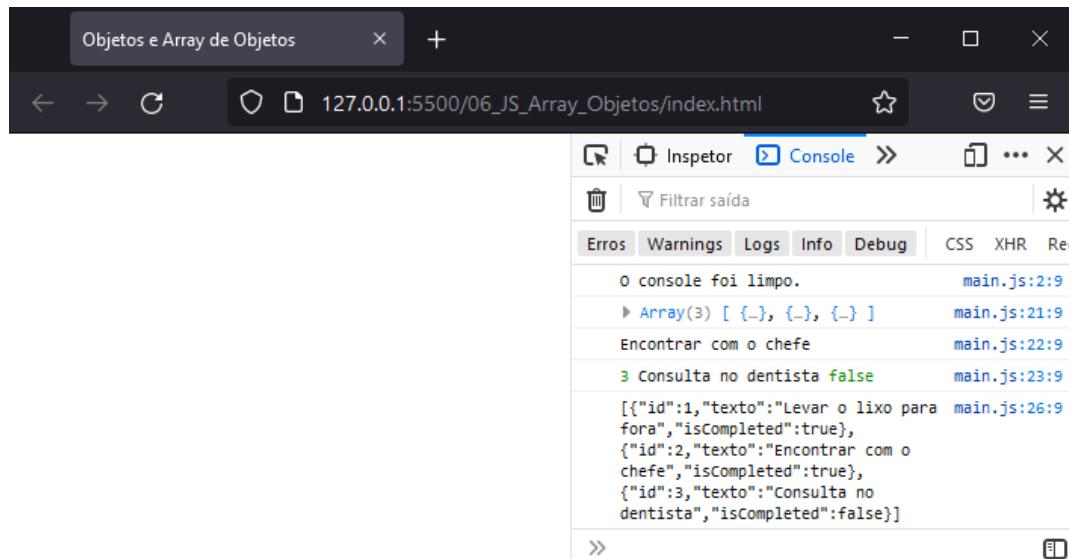
Vamos praticar

Siga os passos para continuar a implementação no arquivo **main.js**:

Continuando a implementação do projeto, insira o seguinte código no arquivo **main.js**:

```
// Transformar um array em JSON
console.log(JSON.stringify(tarefas));
```

O resultado é mostrado no console do navegador:



Observe que resultado foi a **impressão** de uma **string** na notação **JSON**, utilizando o **método stringify()**.

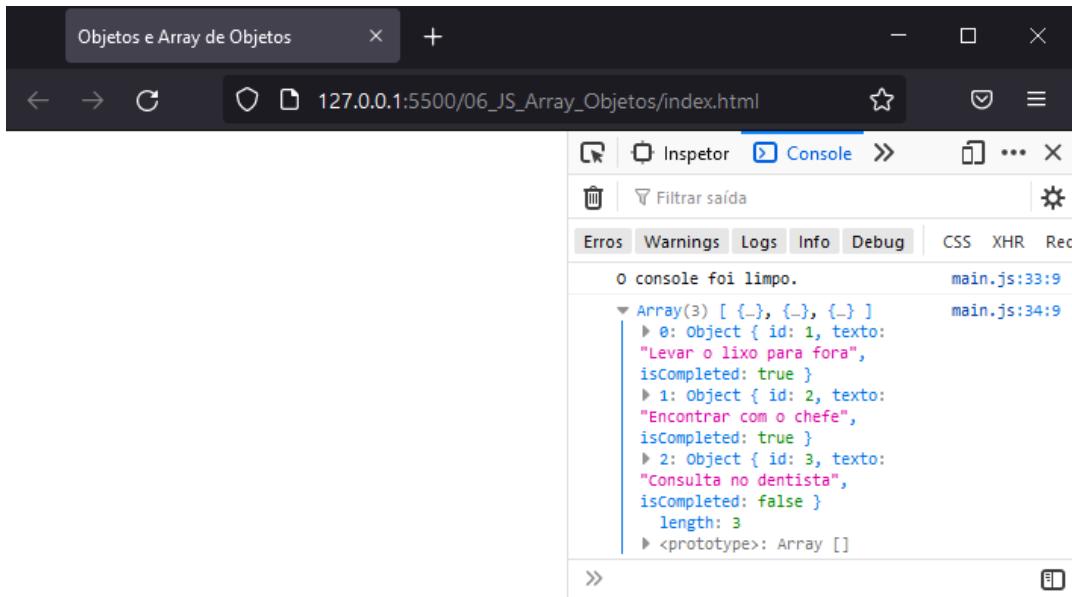
```
[{ "id": 1, "texto": "Levar o lixo para fora", "isCompleted": true }, { "id": 2, "texto": "Encontrar com o chefe", "isCompleted": true }, { "id": 3, "texto": "Consulta no dentista", "isCompleted": false }]
```

Da mesma forma que você consegue transformar um **objeto** em **JSON**, é possível transformar um **JSON** em um **objeto**.

Continuando a implementação do projeto, insira o seguinte código no arquivo **main.js**:

```
// JSON
const tarefa_JSON =
  '[{"id":1,"texto":"Levar o lixo para
fora","isCompleted":true},{ "id":2,"texto":"Encontrar com o
chefe","isCompleted":true}, {"id":3,"texto":"Consulta no
dentista","isCompleted":false}]';
const objeto = JSON.parse(tarefa_JSON);
console.clear();
console.log(objeto);
```

O resultado é mostrado no console do navegador:



```

Objetos e Array de Objetos  × + 
← → ⌂ 127.0.0.1:5500/06_JS_Array_Objetos/index.html ☆ ⌂ ⌂ ...
🔗 Inspetor Console ⌂ ...
🔗 Filtrar saída
Erros Warnings Logs Info Debug CSS XHR Rec
O console foi limpo. main.js:33:9
Array(3) [ {…}, {…}, {…} ] main.js:34:9
▶ 0: Object { id: 1, texto: "Levar o lixo para fora", isCompleted: true }
▶ 1: Object { id: 2, texto: "Encontrar com o chefe", isCompleted: true }
▶ 2: Object { id: 3, texto: "Consulta no dentista", isCompleted: false }
    length: 3
    <prototype>: Array []
»

```

Observe como retornamos o **mesmo array de objetos original**, convertendo o **JSON** em objeto pelo **método parse()**.

Conclusão

Existem muitas utilidades sobre arrays de objeto e o operador rest. Confira esses links e teste os códigos mostrados neles:

https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment

https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Operators/Spread_syntax

<https://www.devmedia.com.br/javascript-operadores-rest-e-spread/41200>

Para saber mais sobre JSON, vale a pena dar uma conferida nos links:

<https://pt.wikipedia.org/wiki/JSON>

<https://www.freecodecamp.org/news/javascript-array-of-objects-tutorial-how-to-create-update-and-loop-through-objects-using-js-array-methods/>

Esse segundo link está em inglês, mas você pode usar a extensão do Google Translate no Chrome para traduzir e ler o artigo.



Objetos, Funções e Eventos

Os objetivos desta aula são:

- Compreender o uso de funções e sua sintaxe;
- Conhecer o conceito de arrow function;
- Entender abstração de objetos do mundo real;
- Conhecer os diferentes tipos de eventos em JavaScript.

Funções

Funções são usadas em programação para executar ações que são rotineiramente executadas em um programa. Uma função é um bloco de código implementado para executar uma tarefa em particular. Para executar uma função devemos sempre invocá-la (chamá-la) dentro do código.

A sintaxe para criar funções em JS é:

```
function myFunc(valor1, valor2) {
    return valor1 * valor2;
}
```



Importante!: O nome de funções segue as mesmas regras de variáveis deve começar por letras ou underline (_) e pode conter números e não pode conter caracteres especiais. O nome também não pode ser uma palavra-chave ou palavra reservada. Programadores geralmente usam somente nomes de funções como camelCase ou CamelCase, isto é, a primeira letra de cada palavra deve ser escrita em maiúscula. Isso padroniza o código e facilita identificar o que é variável e o que é função.

No exemplo mostrado anteriormente, a função com o nome **myFunc** espera receber dois parâmetros (**valor1** e **valor2**) e retorna o resultado da multiplicação dos dois valores (**return valor1 * valor2;**). Quando o JavaScript encontra a instrução **return**, a função para de ser executada. Se a função foi chamada a partir de uma instrução, o JavaScript "retornará" para executar o código após a instrução de chamada. As funções geralmente **calculam um valor de retorno**. O valor de retorno é "retornado" de volta ao objeto que a chamou.



Vamos praticar

Vamos criar um projeto para testar toda teoria que vimos nesse tema. Siga os passos para criar o projeto:

Abra o **VS Code** e escolha um **diretório de trabalho para o seu projeto**.

Crie um diretório para seu projeto com o nome representativo, por exemplo, **JS_Funcoes_Objetos_Eventos**.

Crie um arquivo dentro do diretório do projeto com o nome **index.html** e insira o seguinte código:

```
<!DOCTYPE html>
<html lang="pt-br">
    <head>
        <meta charset="UTF-8" />
        <link rel="shortcut icon" href="#" />
```

```
<title>Funções, Objetos, Eventos e Strings</title>
</head>
<body>
    <script src="main.js"></script>
</body>
</html>
```

Esse código mostra a marcação **<script>** sem **nenhum código JavaScript** entre a **abertura** e o **fechamento** da tag, apenas o atributo **src** com o valor **main.js**. Isso significa que o código JavaScript está em um arquivo externo. Portanto, temos que criar esse novo arquivo **main.js** dentro do diretório do projeto.

Vamos deixar o arquivo **main.js** **vazio** e a **medida que vamos aprendendo coisas novas vamos completando a implementação do código JavaScript**.

Abra o arquivo **index.html**, clique no botão  **Go Live** da extensão **Live Server** e abra as Ferramentas de desenvolvimento do navegador web (Atalho **F12**).

Abra o arquivo **main.js** e insira o seguinte código:

```
// Funções
function addNums(num1 = 1, num2 = 1) {
    return num1 + num2;
}

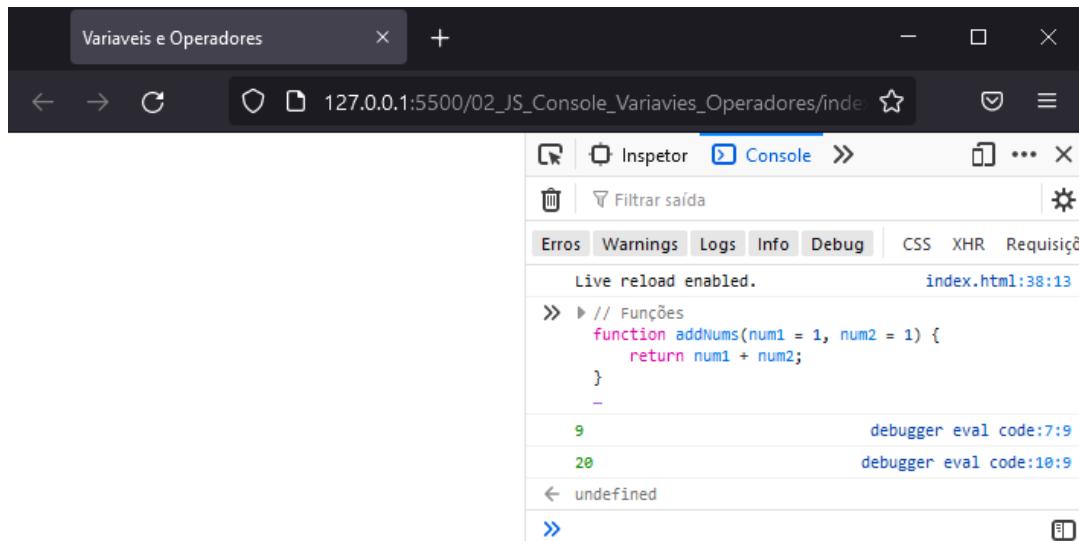
let x = addNums(4, 5); // Chamada da função addNums
console.log(x);

let z = myFunc(4, 5); // Chamada da função myFunc
console.log(z);

function myFunc(num1, num2) {
    return num1 * num2;
}
```

Podemos observar a **declaração de duas funções**: a **primeira** com o nome **addNums**, e a **segunda** com o nome **MyFunc**. Note, também, que no JS **você pode invocar a função antes da sua declaração**, por exemplo a **chamada da função myFunc** está antes da sua declaração. Note, também, que os **parâmetros da função addNums** possui valores padrão (**num1 = 1, num2 = 1**), ou seja, **se na chamada da função não passarmos nenhum valor**, os argumentos **num1 e num2 assumirão o valor 1**.

Com a arquivo **index.html** aberto no seu navegador, você pode salvar o arquivo **main.js** e ver o resultado da execução do código JavaScript no console.



Arrow Function

O conceito de **Arrow Function** foi introduzido em 2015 no ES6. Acostume-se bem com essa forma **mais prática de declarar funções**, pois muitos exemplos que você encontrará na internet usará esse tipo de declaração. A sintaxe de uma **arrow function** é:

```
const hello = () => {
  return 'Olá Arrow Function!';
};
```

Basicamente, você **deve atribuir a função a uma variável declarada com a palavra const** e utilizar o operador **=>** para indicar o **bloco da função**. A palavra **const** pode ser ocultada se você não estiver usando o **Strict Mode**, mas vamos sempre seguir as dicas de bons programadores e nunca deixar de declarar uma variável.

No exemplo mostrado anteriormente, a **função não tem nenhum parâmetro de entrada (dentro dos parênteses da função está vazio)**. Toda função pode ter **zero ou mais parâmetros de entrada**. Mas também podemos fazer uma versão da função **addNums** como **arrow function**, vejamos o exemplo:

```
const addNums2 = (num1 = 1, num2 = 1) => {
  return num1 + num2;
};
```



Vamos praticar

No arquivo **main.js**, insira o seguinte código:

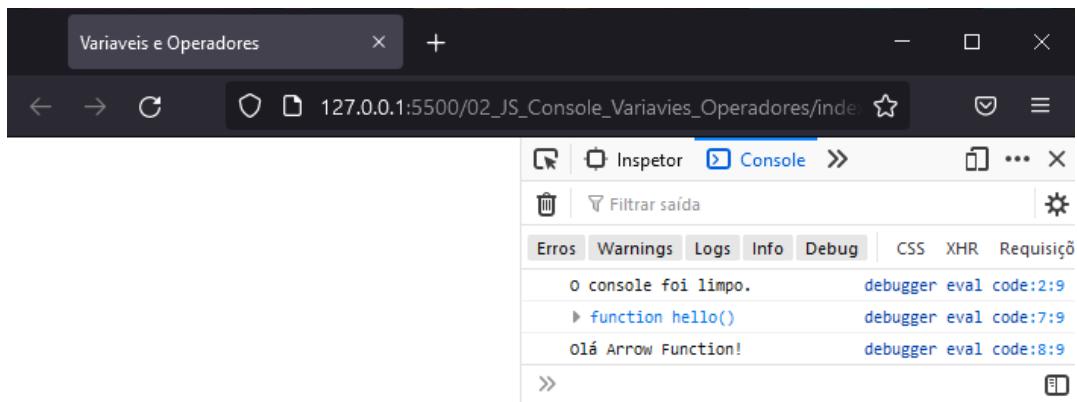
```
//Arrow functions
console.clear();
```

```
const hello = () => {
    return 'Olá Arrow Function!';
};

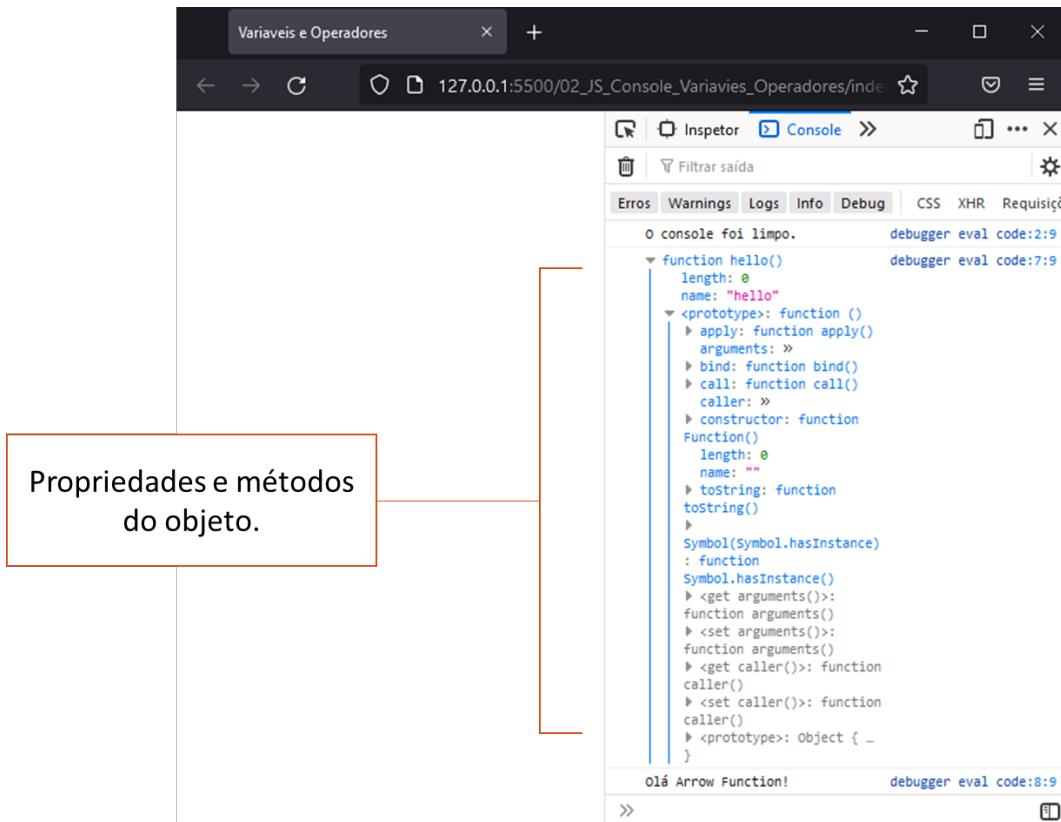
console.log(hello); // Retorna o objeto função
console.log(hello()); // Executa a função e imprime a string no return
```

Observe que quando usamos **apenas o nome da função sem os parênteses**, o console **exibe o objeto function**. Lembre-se disso, pois a medida que avançamos no estudo de JS, vamos fazer **debug** para saber o tipo de objeto que estamos manipulando e não estaremos interessados somente no valor do objeto.

Ao salvar o arquivo **main.js**, é possível ver as informações impressas no console.



Vamos dar uma olhada no objeto **function hello**. Clique na seta antes do **objeto para expandir** o objeto e também clique na seta do **prototype** para expandir completamente o objeto.



Propriedades e métodos do objeto.

Observe que o **objeto possui diversas, propriedade, métodos e protótipos**. Não se preocupe com isso agora, mas isso será bem útil mais adiante e, também, é bom você saber que **existe muita coisa legal para aprenderemos**. E lembre-se da fala de Issac Newton ,que foi usada na série **Dark**:

“What we know is a drop, what we don't know is an ocean.” – **Isaac Newton**.

“O que sabemos é uma gota, o que não sabemos é um oceano.” – **Issac Newton**.

Vamos inserir a versão da função **addNum** com **Arrow Function** no nosso código.

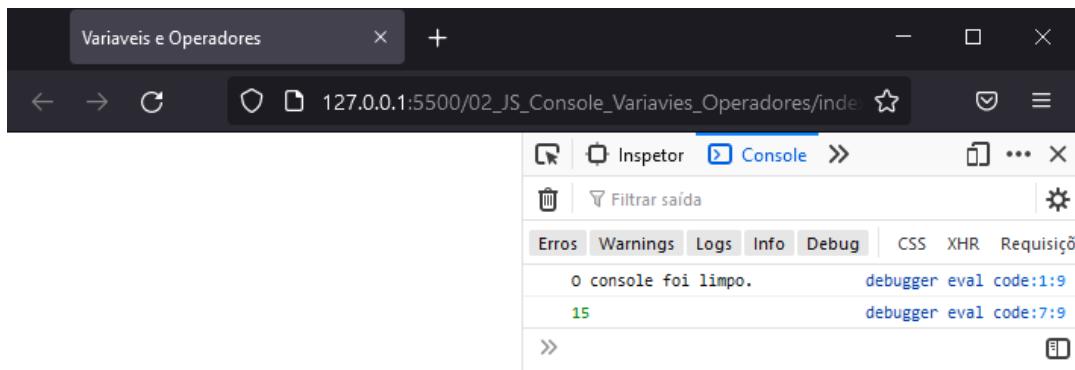
```

console.clear();
const addNums2 = (num1 = 1, num2 = 1) => {
  return num1 + num2;
};

let soma = addNums2(5, 10);
console.log(soma);

```

O resultado pode ser visto no console do navegador:



Objetos

Na vida real um **carro** é um **objeto** e esse objeto possui **propriedades** como, por exemplo, **nome, modelo e peso**, e **métodos** (ações a serem executadas) tais como: **ligar, dirigir, frear e parar**. As **propriedades de um carro podem ser as mesmas**, mas os **valores são diferentes de carro para carro**. Os **métodos de um carro podem também ser os mesmos**, mas a execução pode ser ligeiramente diferente de carro para carro, por exemplo, **um carro pode ligar utilizando a chave e outro pode ser com um botão**.

Objetos
JS



Propriedades	Métodos
car.name = Chevrolet	car.start()
car.model = Celta	car.drive()
car.weight = 890kg	car.brake()
car.color = grey	car.stop()

No JS, um **objeto** é uma coleção de dados e/ou funcionalidades relacionadas (que geralmente consistem em **diversas variáveis e funções** — que são chamadas **de propriedades e métodos quando estão dentro de objetos**). Por enquanto, pense que **toda declaração de variável ou função feita no JS** é um **objeto**. As variáveis mostradas abaixo são objetos:

```
let marca = 'Fiat';
```

```
const carro = {
```

```

type: 'Fiat',
model: '500',
color: 'white',
};

const pessoa = {
  firstName: 'John',
  lastName: 'Doe',
  age: 50,
  eyeColor: 'blue',
};

```

Os objetos **carro** e **pessoa** são bem interessantes, pois eles são **formados por um conjunto de valores** ou o que chamamos de **Array de objetos**. Falaremos mais sobre array de objetos mais adiante. Vamos abordar cada assunto com calma e aprofundar gradativamente.

Eventos

Eventos são **ações executadas quando algo acontece na página web**, ou seja, é a **reação algum estímulo ou interação em elemento HTML**. Esses eventos do HTML normalmente chamam funções do JavaScript. Eventos HTML são coisas que o navegador pode fazer ou algo que o usuário pode fazer. Os principais eventos do HTML são:

Event	Description
onchange	Um elemento HTML é alterado.
onclick	O usuário clica em um elemento HTML.
onmouseover	O usuário move o mouse sobre o elemento HTML.
onmouseout	O usuário tira o mouse de cima do elemento HTML.
onkeydown	O usuário pressiona um Tecla do teclado.
onload	O navegador termina de carregar a página.



Importante! Se você quiser saber mais sobre eventos HTML pode consultar o link do w3schools, que está disponível em:

https://www.w3schools.com/jsref/dom_obj_event.asp



Vamos praticar

Vamos implementar nesse momento os **eventos onmouseover, onload e onclick**, pois esse eventos são mais legais quando estivermos **acessando o DOM**. Vamos ter uma aula somente sobre **JavaScript e DOM**, então aguardem. Vamos precisar modificar o arquivo **index.html** e o **main.js**. Siga os passos para implementar alguns eventos utilizando HTML e JavaScript.

Atualize o arquivo **index.html** com o código mostrado a seguir:

```
<!DOCTYPE html>
<html lang="pt-br">
  <head>
    <meta charset="UTF-8" />
    <link rel="shortcut icon" href="#" />
    <title>Funções, Objetos, Eventos e Strings</title>
  </head>
  <body onLoad="boasVindas()">
    <button onclick="eventClique()">Clique aqui</button>
    <script src="main.js"></script>
  </body>
</html>
```

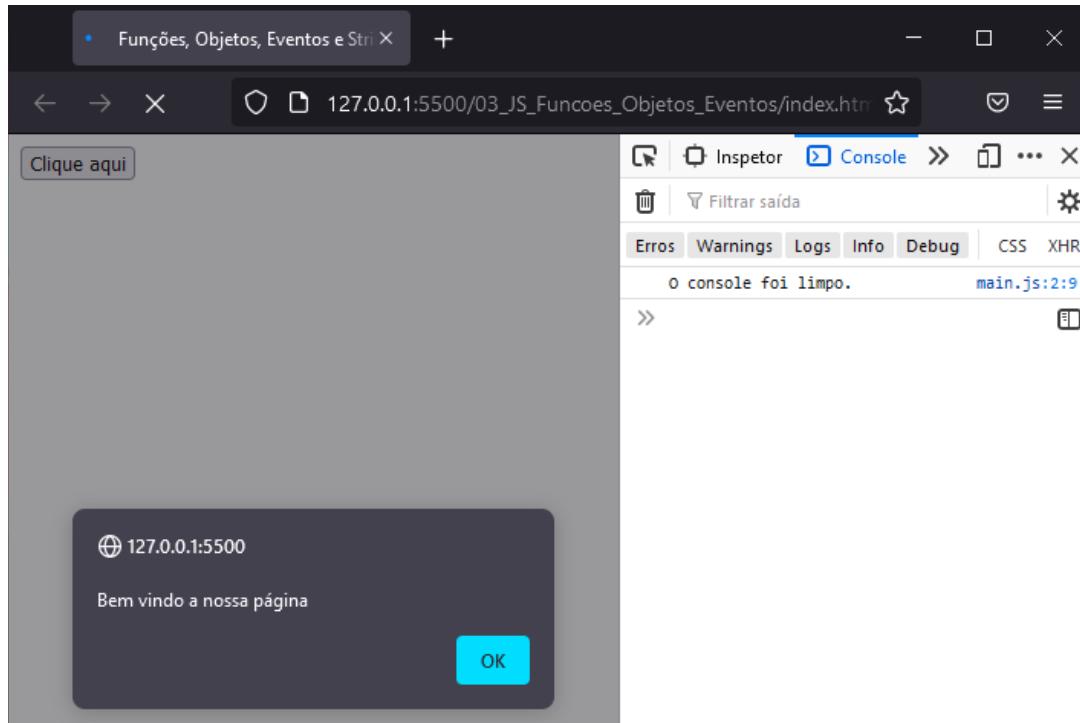
Nesse código fizemos **duas atualizações**, a **primeira é na marcação <body>** a inserção do evento **onload**, que chama a função **boasVindas()**, e a **segunda alteração é a inserção de um botão com o evento onclick**, que invoca a função **eventClique()**. Essas funções serão implementadas no arquivo **main.js** e você pode dar qualquer nome para elas desde que respeite as regras para nomes de funções.

Salve o arquivo **index.html** e vamos para o arquivo **main.js**. Insira o código abaixo com as declarações das funções.

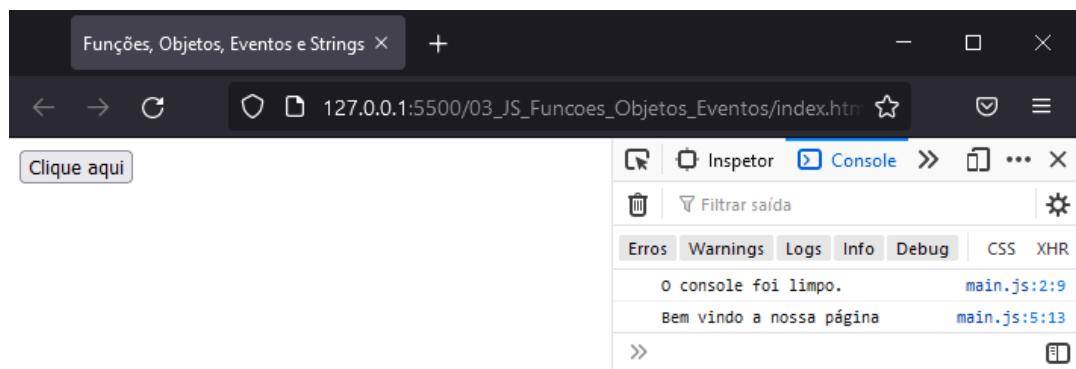
```
// Eventos
console.clear();
const boasVindas = () => {
  alert('Bem vindo a nossa página');
  console.log('Bem vindo a nossa página');
};

const eventClique = () => {
  console.log('Você clicou no botão');
};
```

Salve o arquivo **main.js** e abra a página no navegador (arquivo **index.html**).

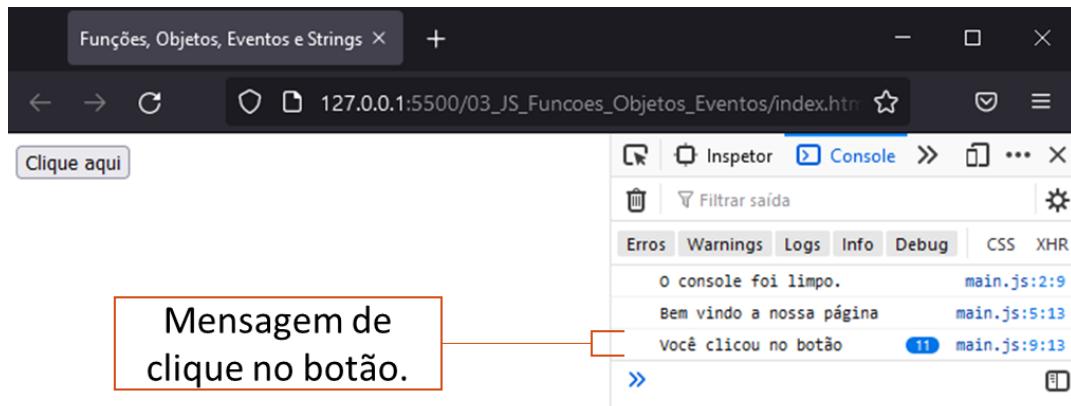


Ao carregar a página o evento **onload** é disparado e uma **mensagem de alerta é disparada**. Após você clicar no botão **OK**, a mesma mensagem irá aparecer no console (instrução **console.log()**).



Essas mensagens aparecem sempre que você recarregar a página.

Se você **clicar no botão** verá a mensagem no **console** com uma **badge** mostrando quantas vezes o botão foi clicado.



Agora, vamos fazer mais **uma atualização** no arquivo **index.html** como mostra o código a seguir:

```
<!DOCTYPE html>
<html lang="pt-br">
  <head>
    <meta charset="UTF-8" />
    <link rel="shortcut icon" href="#" />
    <title>Funções, Objetos, Eventos e Strings</title>
  </head>
  <body onLoad="boasVindas()">
    <h2 onmouseover="mouseEmCima()">Passe o mouse sobre esse título</h2>
    <button onclick="eventClique()">Clique aqui</button>
    <script src="main.js"></script>
  </body>
</html>
```

Nesse código, **inserimos um elemento heading <h2>** com o evento **onmouseover**, que chama a função **mouseEmCima()**. Vamos implementar a função **mouseEmCima()** no arquivo **main.js**.

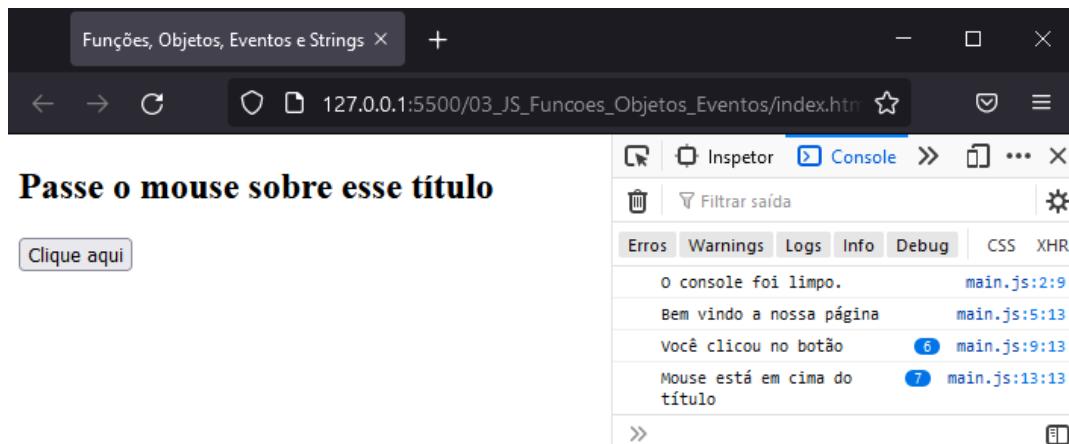
Volte ao arquivo **main.js** e insira o código abaixo.

```
// Eventos
console.clear();
const boasVindas = () => {
  alert('Bem vindo a nossa página');
  console.log('Bem vindo a nossa página');
};

const eventClique = () => {
  console.log('Você clicou no botão');
};

const mouseEmCima = () => {
  console.log('Mouse está em cima do título');
};
```

Passe o mouse por cima do título e veja a mensagem impressa no console.





Métodos de alto nível para manipular arrays

Os objetivos desta aula são:

- Compreender a manipulação de arrays;
- Aprender os métodos: `forEach()`, `map()`, `filter()` e `find()`.

Método forEach()

O método **forEach()** executa uma determina função para **cada um dos elementos de um array**. **Ele não é executado em elementos vazios do array**. A sintaxe do método é:

```
nome_do_array.forEach(callback(currentValue [, index [, array]])[, thisArg]);
```

Os parâmetros entre colchetes são opcionais e cada um significa:

- **callback**: é a função que será chamada para ser executada em cada elemento do array.
- **currentValue**: é o valor do elemento que está sendo processado no momento.
- **index** (opcional): O índice do elemento atual sendo processado no array.
- **array** (opcional): O array que forEach() está sendo aplicado.
- **thisArg** (opcional): Valor a ser usado como this quando executar callback.

Desse modo, o **forEach()** executa uma vez função que estará no lugar o argumento **callback** para **cada elemento de array** que possui **um valor atribuído**. Algumas das **vantagens** do método **forEach()** são:

- O método percorre automaticamente os elementos do array.
- O método permite executar uma função sobre cada elemento do array.



Vamos praticar

Siga os passos para criar o projeto:

Abra o VS Code e escolha um diretório de trabalho para o seu projeto.

Abra o **VS Code** e escolha um **diretório de trabalho para o seu projeto**.

Crie um diretório para seu projeto com o nome representativo, por exemplo, **JS_Array_High_Level**

Crie um arquivo dentro do diretório do projeto com o nome **index.html**.

Insira o seguinte código no seu arquivo **index.html**.

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

<head>
  <meta charset="UTF-8" />
  <link rel="shortcut icon" href="#" />
  <title>Métodos de alto nível para manipular arrays</title>
</head>
```

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

</html>
```

Esse código mostra a marcação **<script>** sem nenhum código **JavaScript** entre a **abertura** e o **fechamento** da tag, apenas o atributo **src** com o valor **main.js**. Isso significa que o código **JavaScript** está em um arquivo externo. Portanto, temos que criar esse novo arquivo **main.js** dentro do diretório do projeto.

Observe que **mudamos a organização dos arquivos no projeto**, agora o arquivo **main.js** está em uma pasta chamada **js**, enquanto o arquivo **index.html** está no **diretório raiz do projeto**. Isso é importante para você ser uma pessoa organizada com os seus projetos. Além disso, o caminho do script agora está com apontando para o novo diretório (**<script src=".js/main.js"></script>**).

Vamos deixar o arquivo **main.js vazio** e à medida que vamos aprendendo coisas novas vamos completando a implementação do código **JavaScript**.

Abra o arquivo **index.html**, clique no botão  **Go Live** da extensão **Live Server** e abra as Ferramentas de desenvolvimento do navegador web (Atalho **F12**).

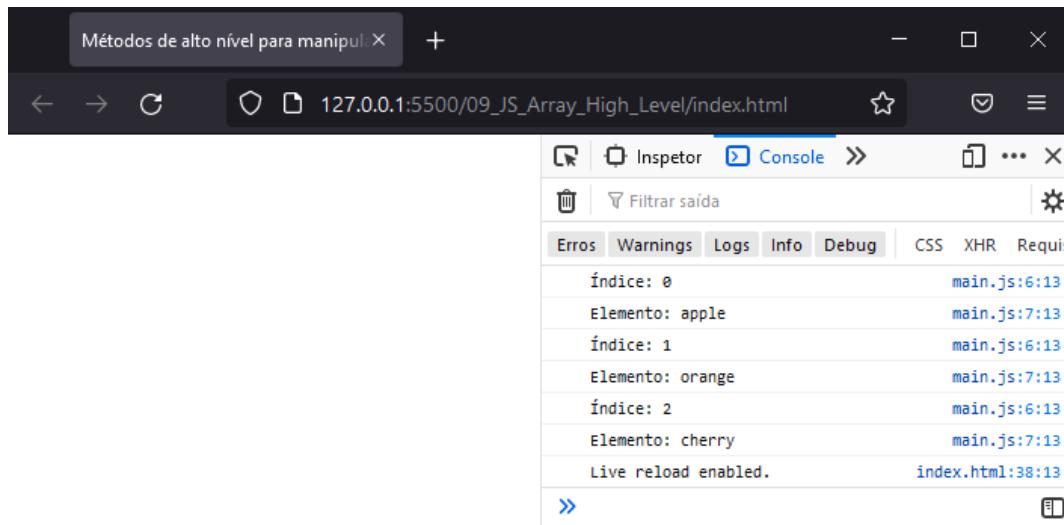
Agora, vamos implementar todo o nosso código desse tema no arquivo **main.js** e ver as mensagens impressas no console.

No arquivo, **main.js** digite o seguinte código.

```
// Método forEach()
const frutas = ['apple', 'orange', 'cherry'];
frutas.forEach(minhaFuncao);

function minhaFuncao(item, index) {
  console.log(`Índice: ${index}`);
  console.log(`Elemento: ${item}`);
}
```

Vamos ver o resultado mostrado no console.



A instrução mostra como chamar o método **forEach()**:

Nome do array	Ponto final	forEach	(função chamada)
frutas	.	forEach	(minhaFuncao)

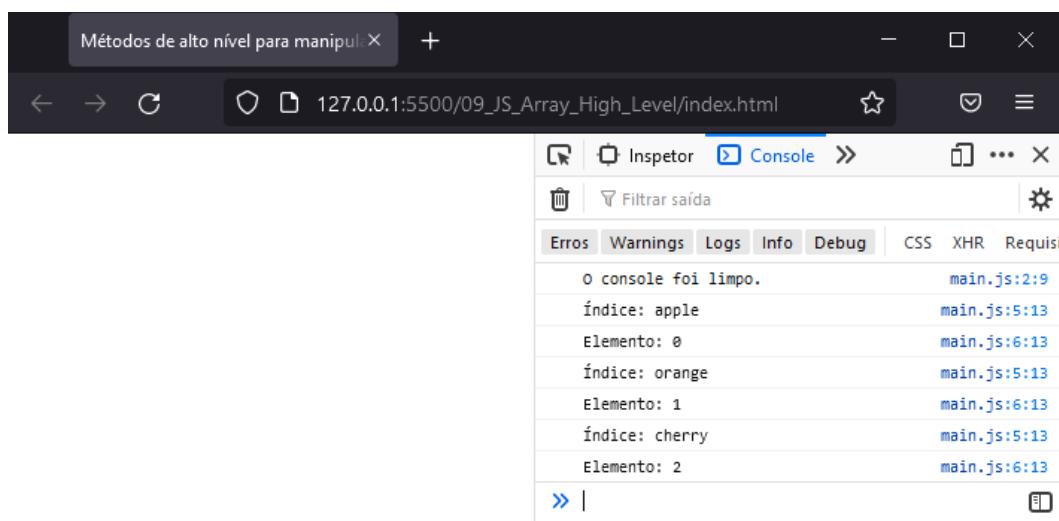
Observe que para cada elemento a função **minhaFuncao** foi **executada para cada elemento, imprimindo no console o índice e o valor do elemento.**

Importante: Dificilmente faremos um código como mostrado anteriormente, isso porque o JS tem **arrow function**, que é muito mais **prático** e torna o código mais **enxuto** e **elegante**. Por isso: o código anterior ficar assim com **arrow function**:

No arquivo, **main.js** digite o seguinte código.

```
// Método forEach() com Arrow Function
console.clear();
const frutas01 = ['apple', 'orange', 'cherry'];
frutas01.forEach((index, item) => {
  console.log(`Índice: ${index}`);
  console.log(`Elemento: ${item}`);
});
```

Vamos ver o resultado mostrado no console.



The screenshot shows a browser window with the URL 127.0.0.1:5500/09_JS_Array_High_Level/index.html. The developer tools console tab is selected. The output in the console is:

```
O console foi limpo.
Índice: apple
Elemento: 0
Índice: orange
Elemento: 1
Índice: cherry
Elemento: 2
```

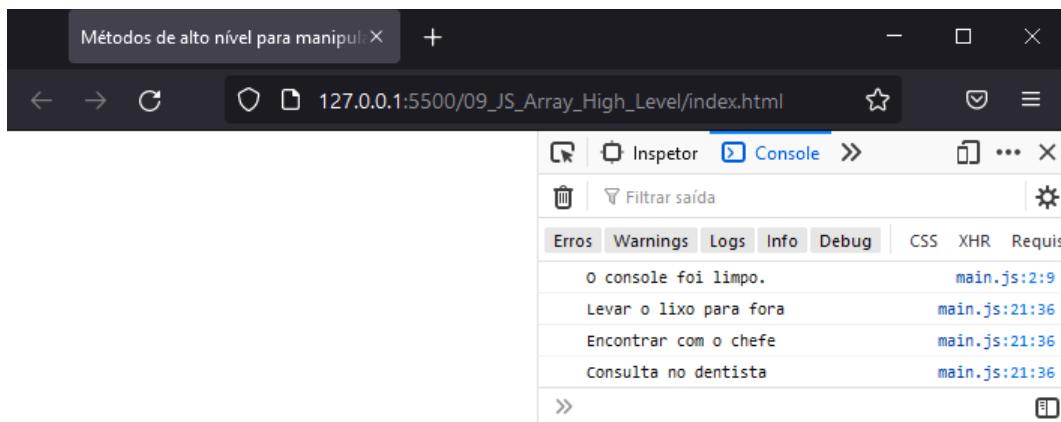
O resultado é o mesmo do anterior, mas o código é mais simples. Por isso, **entenda bem essa estrutura de chamada de arrow function**, pois ela é muito útil em JS.

Além da praticidade de percorrer array, também podemos aplicar esse método em arrays de objetos.

Continuando a implementação do projeto, insira o seguinte código no arquivo **main.js**:

```
// forEach() com array de objetos
console.clear();
const tarefas = [
  {
    id: 1,
    texto: 'Levar o lixo para fora',
    isCompleted: true,
  },
  {
    id: 2,
    texto: 'Encontrar com o chefe',
    isCompleted: true,
  },
  {
    id: 3,
    texto: 'Consulta no dentista',
    isCompleted: false,
  },
];
tarefas.forEach((teste) => console.log(teste.texto));
```

O resultado é mostrado no console do navegador:



A screenshot of a browser's developer tools showing the 'Console' tab selected. The output area displays the following log entries:

```

O console foi limpo.          main.js:2:9
Levar o lixo para fora       main.js:21:36
Encontrar com o chefe         main.js:21:36
Consulta no dentista          main.js:21:36

```

O interessante é que **não é necessário indicar o índice do array com nos laços de repetição.**

Importante: Não apague o código anterior, pois vamos utilizar o array de objetos **tarefas** nos métodos, que iremos aprender daqui para frente.

Método map()

O método **map()** cria um **novo array com o resultado gerado pela chamada de uma função para cada elemento do array**. Ele não é executado em elementos vazios do array. A sintaxe do método é:

```
let novo_array = nome_do_array.map(callback[, thisArg]);
```

Os parâmetros entre colchetes são opcionais e cada um significa:

- **callback**: é a função que o retorno produz o elemento do novo Array.

Desse modo, o **map()** executa uma vez função que estará no lugar o argumento **callback** para cada elemento de array que possui **um valor atribuído** e constrói **um novo array com base nos valores retornados pela execução da função**.



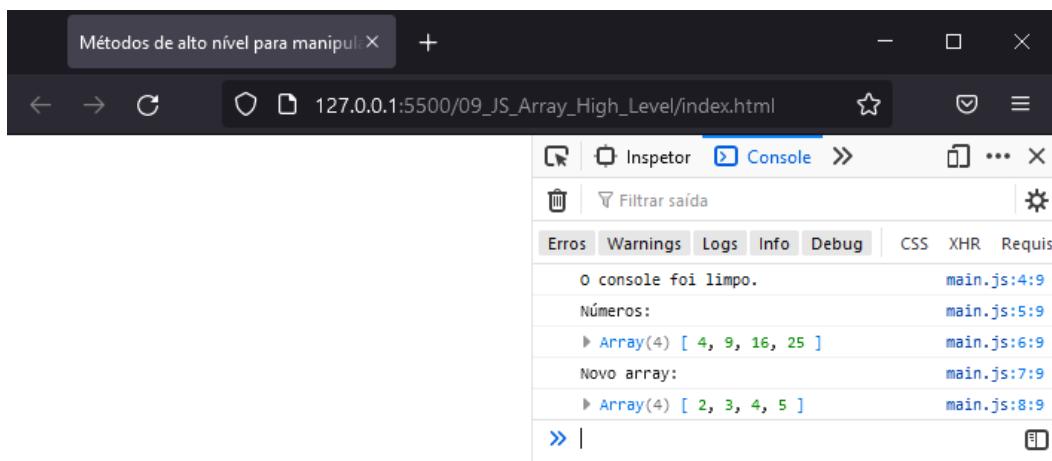
Vamos praticar

No arquivo, **main.js** digite o seguinte código:

```
// Método map()
const numeros = [4, 9, 16, 25];
const newArray = numeros.map(Math.sqrt);
console.clear();
console.log('Números:');
console.log(numeros);
```

```
console.log('Novo array:');
console.log(newArray);
```

O resultado é mostrado no console do navegador.



The screenshot shows a browser window with the URL "127.0.0.1:5500/09_JS_Array_High_Level/index.html". The browser's address bar also displays "Métodos de alto nível para manipulação de arrays". The developer tools are open, specifically the "Console" tab. The console output is as follows:

```

O console foi limpo.                                main.js:4:9
Números:                                            main.js:5:9
▶ Array(4) [ 4, 9, 16, 25 ]                      main.js:6:9
Novo array:                                         main.js:7:9
▶ Array(4) [ 2, 3, 4, 5 ]                          main.js:8:9

```

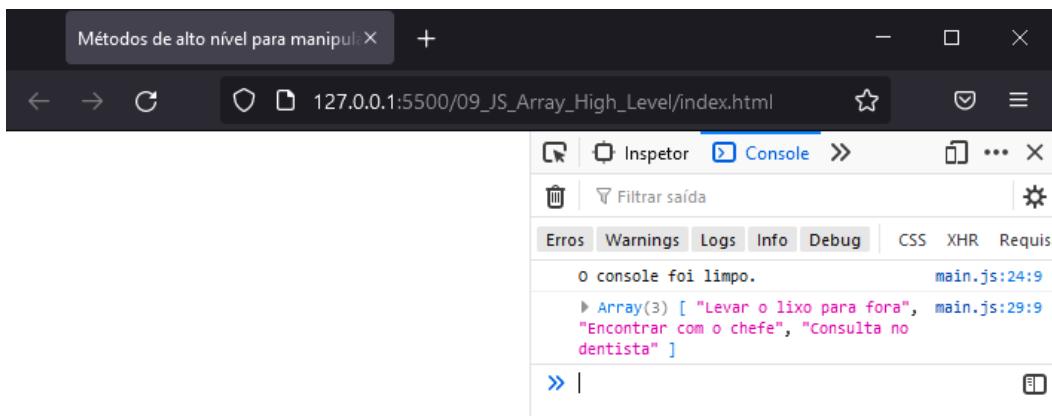
O método **map()** ou método **sqrt** do objeto **Math**, que retorna a raiz quadrada (**square root**) de cada elemento do **array** e **armazenado o resultado no novo array** com o nome **newArray**, que é impresso no console como array.

Vamos ver outro exemplo que utilizará o array de objetos **tarefas** nesse método, que foi usado anteriormente.

No arquivo, **main.js** digite o seguinte código.

```
// Outro exemplo do map
console.clear();
const mapText = tarefas.map((valor) => {
    return valor.texto;
});
//Imprime o novo array
console.log(mapText);
```

O resultado é mostrado no console do navegador.



The screenshot shows a browser window with the URL "127.0.0.1:5500/09_JS_Array_High_Level/index.html". The browser's address bar also displays "Métodos de alto nível para manipulação de arrays". The developer tools are open, specifically the "Console" tab. The console output is as follows:

```

O console foi limpo.                                main.js:24:9
▶ Array(3) [ "Levar o lixo para fora", main.js:29:9
  "Encontrar com o chefe", "Consulta no
  dentista" ]
```

O método **map()** executa **a função para armazenar o valor do campo texto de cada objeto do array tarefas**. Observe que é criado um **novo array** com o nome **mapText**, que é impresso no console como array.

Método filter()

O método **filter()** cria um novo array com o resultado do filtro aplicado pelo método, ou seja, os elementos que “passaram” na condição configurada no filtro. A sintaxe do método é:

```
let novo_array = nome_do_array.filter(callback[, thisArg]);
```

Os parâmetros entre colchetes são opcionais e cada um significa:

- **callback**: é a função com a condição para testar cada elemento do array.

Desse modo, o **filter()** executa uma vez função que estará no lugar o argumento **callback** para cada **elemento de array** que possui um **valor atribuído** e **constrói um novo array** com os elementos que **retornarem true** para a condição configurada no filtro.



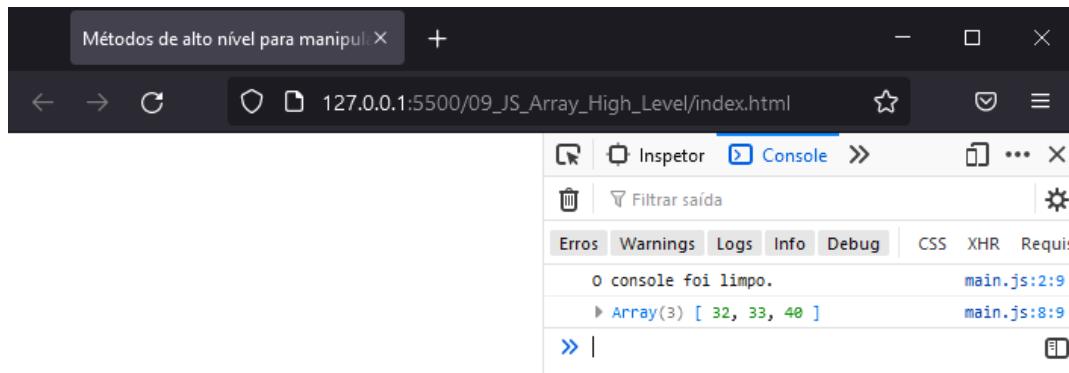
Vamos praticar

No arquivo, **main.js** digite o seguinte código.

```
// Método filter
console.clear();
const idades = [32, 33, 16, 40];

let filtroIdade = idades.filter((idade) => {
  return idade >= 18;
});
console.log(filtroIdade);
```

O resultado é mostrado no console do navegador.



O filtro diz para **retornar os valores maiores de 18**, ou seja, serão retornados os valores **32, 33 e 40**. Como podemos ver na mensagem impressa no console.

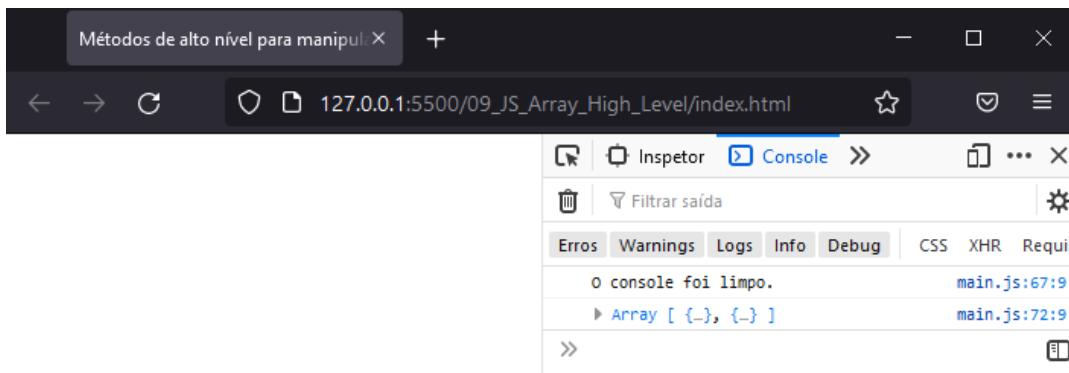
Vamos utilizar o **array de objetos tarefas** nesse método, que foi usado anteriormente.

No arquivo, **main.js** digite o seguinte código.

```
// Outro exemplo do filter
console.clear();
const filtroCompletas = tarefas.filter((item) => {
  return item.isCompleted === true;
});

console.log(filtroCompletas);
```

O resultado é mostrado no console do navegador.



A condição do filtro configurada era para retornar **todos os objetos do array que possuam o campo/chave com o valor true**. Desse modo o resultado é o retorno dos dois primeiros objetos, como pode ser visto no valor impresso. Caso você não visualize os valores do array:

▶ Array [{ ... }, { ... }] main.js:72:9

Basta você clicar na seta ao lado do nome Array para expandir os objetos.

Método find()

O método **find()** retorna o valor dos **elementos do array** que passarem na condição configurada. Semelhante ao método **filter()**, porém o **find() não cria um novo array**. A sintaxe do método é:

```
nome_do_array.find(callback(element[, index[, array]])[, thisArg]);
```

Os parâmetros entre colchetes são opcionais e cada um significa:

- **callback**: é a função com a iteração de cada elemento do array.

Desse modo, o **find()** executa uma vez função que estará no lugar o argumento **callback** para cada **elemento de array** e retorna os **valores que retornarem true** para a condição configurada no filtro.



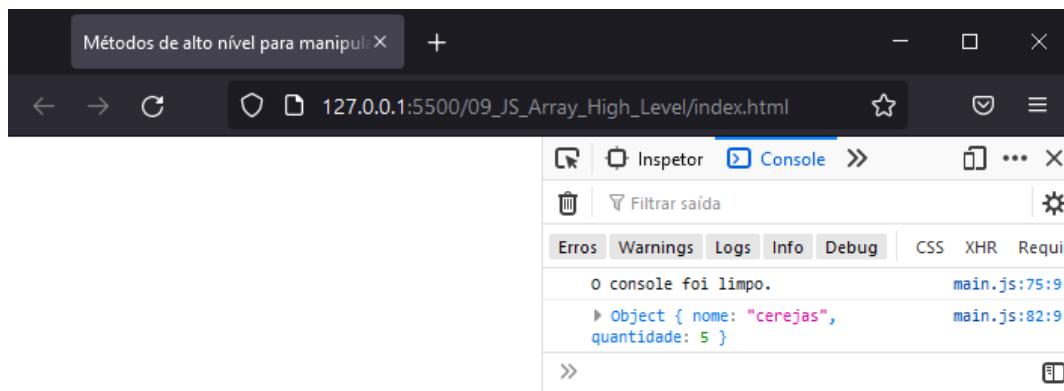
Vamos praticar

No arquivo, **main.js** digite o seguinte código.

```
// Método find()
console.clear();
const meuArray = [
    { nome: 'apples', quantidade: 2 },
    { nome: 'bananas', quantidade: 0 },
    { nome: 'cerejas', quantidade: 5 },
];

console.log(
    meuArray.find((fruta) => {
        return fruta.nome === 'cerejas';
    })
);
```

O resultado é mostrado no console do navegador.



Objeto que o método **find()** retornou, o **objeto cujo campo nome era igual a cerejas**, como foi configurada a condição.

Para Aprender mais

Procure sempre aprender e estudar mais. Seguem alguns links para você estudar e aprender mais:

forEach()

https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach

https://www.w3schools.com/jsref/jsref_foreach.asp

map()

https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Array/map

https://www.w3schools.com/jsref/jsref_map.asp

filter()

https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Array/filter

https://www.w3schools.com/jsref/jsref_filter.asp

find()

https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Array/find

https://www.w3schools.com/jsref/jsref_find.asp

Existem muito outros métodos para manipular arrays, vale a pena você conferir a lista nos links:

https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Array#

https://www.w3schools.com/jsref/jsref_obj_array.asp



Classes e Funções

Os objetivos desta aula são:

- Compreender o conceito de classes e sua sintaxe;
- Conhecer os métodos de uma classe;
- Aprender o uso do this no contexto de funções.

Classes

Classes foram introduzidas no JS no **ECMAScript 2015**, mais conhecido como **ES6**, e elas são simplificações da linguagem para utilizarmos **herança baseadas nos protótipos**. Uma **classe** JavaScript **não é um objeto** e sim um **template para objetos JavaScript**. A sintaxe para classes **não introduz um novo modelo de herança de orientação a objetos em JavaScript**.

Classes em JavaScript **provêm uma maneira mais simples e clara** de **criar objetos** e lidar com **herança**. O bloco de instruções (corpo) da declaração de classes é executados em modo estrito.

Para saber mais: Você pode saber mais sobre o modo estrito no link:

https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Strict_mode

A sintaxe de uma classe em JavaScript é:

```
class NomeClasse {  
    constructor() { ... }  
}
```

Por exemplo, a classe a seguir tem o **nome Carro** e **duas propriedades iniciais: nome e ano**. Algumas literaturas chamam essas propriedades de **atributos** e/ou **estados da classe**.

```
class Carro {  
    constructor(nome, ano) {  
        this.nome = nome;  
        this.ano = ano;  
    }  
}
```



Vamos praticar

Siga os passos para criar o projeto:

Abra o VS Code e escolha um diretório de trabalho para o seu projeto.

Abra o **VS Code** e escolha um **diretório de trabalho para o seu projeto**.

Crie um diretório para seu projeto com o nome representativo, por exemplo, **JS_Classe_Func**

Crie um arquivo dentro do diretório do projeto com o nome **index.html**.

Insira o seguinte código no seu arquivo **index.html**.

```
<!DOCTYPE html>  
<html lang="pt-br">  
    <head>  
        <meta charset="UTF-8" />
```

```

<link rel="shortcut icon" href="#" />
<title>Classes e Funções</title>
</head>
<body>
    <script src=".js/main.js"></script>
</body>
</html>

```

Esse código mostra a marcação **<script>** sem nenhum código **JavaScript** entre a **abertura** e o **fechamento** da tag, apenas o atributo **src** com o valor **main.js**. Isso significa que o código **JavaScript** está em um arquivo externo. Portanto, temos que criar esse novo arquivo **main.js** dentro do diretório do projeto.

Observe que **mudamos a organização dos arquivos no projeto**, agora o arquivo **main.js** está em uma pasta chamada **js**, enquanto o arquivo **index.html** está no **diretório raiz do projeto**. Isso é importante para você ser uma pessoa organizada com os seus projetos. Além disso, o caminho do script agora está com apontando para o novo diretório (**<script src=".js/main.js"></script>**).

Vamos deixar o arquivo **main.js vazio** e à medida que vamos aprendendo coisas novas vamos completando a implementação do código **JavaScript**.

Abra o arquivo **index.html**, clique no botão  **Go Live** da extensão **Live Server** e abra as Ferramentas de desenvolvimento do navegador web (Atalho **F12**).

Agora, vamos implementar todo o nosso código desse tema no arquivo **main.js** e ver as mensagens impressas no console.

No arquivo, **main.js** digite o seguinte código.

```

// Classes
class Carro {
    constructor(nome, ano) {
        this.nome = nome;
        this.ano = ano;
    }
}

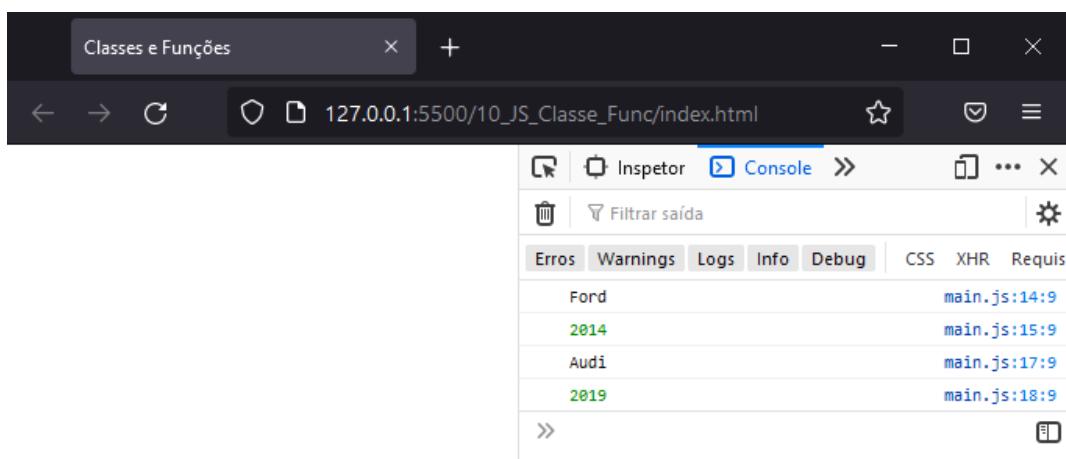
// Intanciando objetos à classe
let meuCarro1 = new Carro('Ford', 2014);
let meuCarro2 = new Carro('Audi', 2019);

// Imprimindo valores
console.log(meuCarro1.nome);
console.log(meuCarro1.ano);

console.log(meuCarro2.nome);
console.log(meuCarro2.ano);

```

Vamos ver o resultado mostrado no console.



The screenshot shows a browser window with the address bar at 127.0.0.1:5500/10_JS_Classe_Func/index.html. Below the address bar is a toolbar with icons for back, forward, search, and refresh. The main area is a developer tools console tab labeled 'Console'. The console has tabs for Errors, Warnings, Logs, Info, Debug, CSS, XHR, and Requisições. The 'Logs' tab is selected. It displays four entries: 'Ford' at main.js:14:9, '2014' at main.js:15:9, 'Audi' at main.js:17:9, and '2019' at main.js:18:9. There is also a 'Filtrar saída' (Filter output) button and a gear icon for settings.

A classe é **criada** e usada para **instanciar objetos à classe**, ou seja, **para criar objetos utilizando o template da classe**. Nesse exemplo, o template da classe possui duas propriedades **nome** e **ano**. Desse modo, o **primeiro valor** colocado entre parênteses na instrução `new Carro('Ford', 2014)` é associado a propriedade **nome** e o **segundo valor** é associado à propriedade **ano**. Se você inverter a ordem de passagem dos dados, verá que os valores trocarão de propriedade.

Ao imprimirmos cada uma das propriedades dos objetos, podemos visualizar os conteúdos **armazenados nas propriedades dos objetos criados**.

O método **constructor** é um tipo especial de método usado para **criar e iniciar um objeto criado pela classe**. Esse método é chamado automaticamente, quando um **novo objeto é criado utilizando o template da classe**.

A palavra-chave **this** é usada de dentro da classe para referenciar a **instância atual**. Porém, em JS, essa palavra-chave pode ter outras funções. Como veremos na próxima seção, quando utilizarmos em funções.

Importante: Diferente de funções, que podem ser declaradas antes ou depois de serem invocadas, as classes devem ser declaradas antes de serem utilizadas. Caso contrário, o programa gerará erro de referência da classe.

Métodos de uma classe

Os **métodos de uma classe** são criados com a mesma sintaxe de um **método de objetos**. Além do **constructor**, você pode criar **diversos protótipos de métodos na classe**:

```
class NomeClasse {  
    constructor() { ... }  
    metodo_1() { ... }  
    metodo_2() { ... }
```

```
metodo_3() { ... }
```



Vamos praticar

Continuando a implementação do projeto, no arquivo, **main.js** digite o seguinte código:

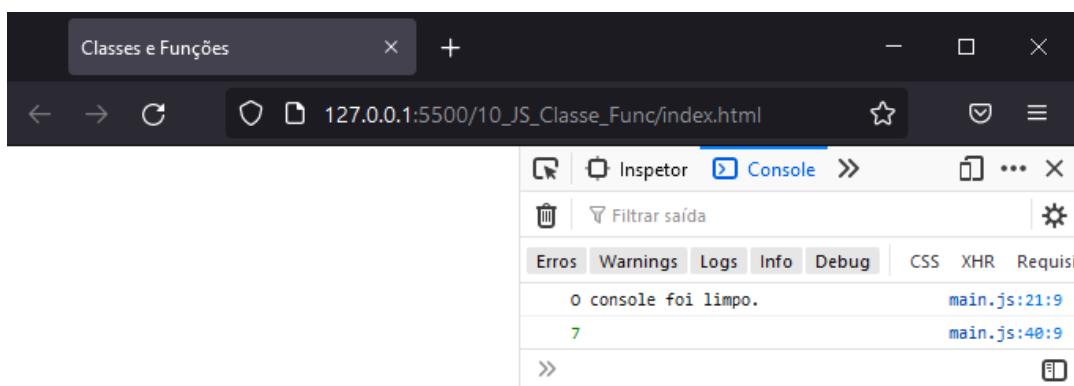
```
// Declarar nova classe com métodos
console.clear();
class NovoCarro {
    constructor(nome, ano) {
        this.nome = nome;
        this.ano = ano;
    }
    idadeCarro(ano) {
        return ano - this.ano;
    }
}

//Buscando o ano atual automaticamente
let dataHoje = new Date();
let ano = dataHoje.getFullYear();
// console.log(dataHoje);

// Intanciando o objeto à classe
let meuNovoCarro = new NovoCarro('Ford', 2014);

console.log(meuNovoCarro.idadeCarro(ano));
```

O resultado é mostrado no console do navegador.



O método **idadeCarro**, que foi declarado, calcula a **quantidade de anos do carro** de acordo com o **ano atual**. Para executar esse método, basta colocar o **nome do objeto, ponto final e o nome do método com o parâmetro** (quando houver) **passado entre os parênteses**.

Outra coisa interessante no código é como buscamos o ano atual. Quando criamos um **objeto** com a classe **Date do JavaScript**. Essa classe já está **embutida na linguagem e retorna o data atual como dia, mês, ano, hora, minutos, segundos e outras informações**. Se imprimir o valor da variável **dataHoje**, descomentando o comando `console.log(dataHoje)`, poderá ver a data dia atual impressa no console. Por exemplo, no momento da produção desse conteúdo a data de hoje é:

▶ Date Sun Oct 17 2021 12:33:44 GMT+0200 (Horário de main.js:35:9
Verão da Europa Central)

O ano atual é obtido pelo método **getFullYear** que existe na declaração dessa classe.

Para saber mais: Você pode ver mais sobre essa classe na documentação disponibilizada pela Mozilla no link:

https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Date

Mais um exemplo de classe

Para fixar mais o conceito de classe, vamos para mais um exemplo.

Continuando a implementação do projeto, no arquivo, **main.js** digite o seguinte código:

```
// Mais um exemplo de classe
console.clear();
class ClassePessoa {
    constructor(firstName, lastName, dob) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.dob = new Date(dob);
    }

    getBirthYear() {
        return this.dob.getFullYear();
    }

    getFullName() {
        return `${this.firstName} ${this.lastName}`;
    }
}

// Instantiate object
```

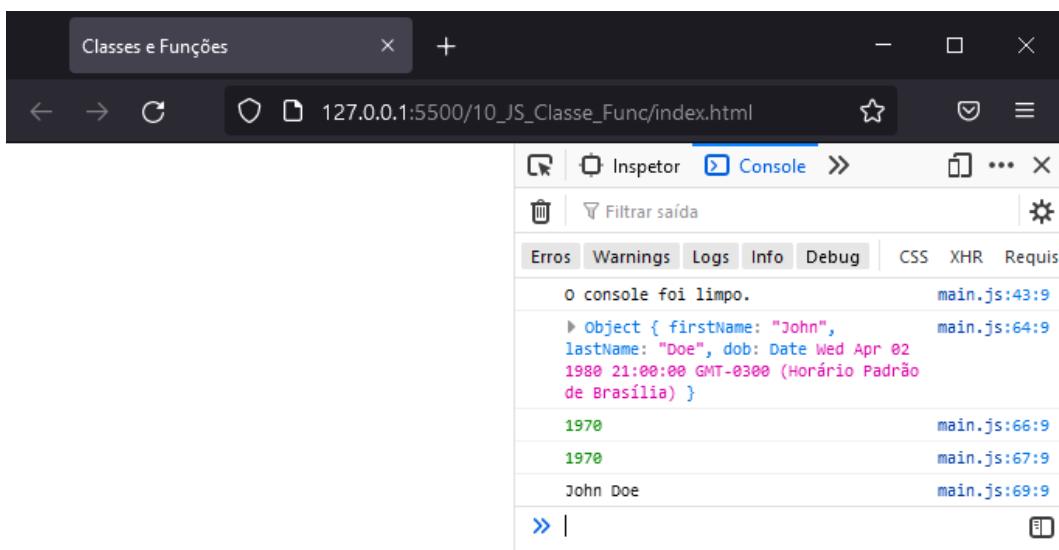
```
const pessoa1 = new ClassePessoa('John', 'Doe', '1980-04-03');
const pessoa2 = new ClassePessoa('Marry', 'Smith', '1970-06-13');

console.log(pessoa1);

console.log(pessoa2.dob.getFullYear());
console.log(pessoa2.getBirthYear());

console.log(pessoa1.getFullName());
```

O resultado é mostrado no console do navegador.



Nesse exemplo, **criamos uma classe** com os atributos **firstName** (**primeiro nome**), **lastName** (**último nome**) e **dob** (**Data of Birth, data de nascimento**) e com os **métodos** **getBirthYear** (**busca ano do nascimento**) e **getFullName** (**buscar nome completo [primeiro e último nome]**).

Criamos **dois objetos instanciados à classe** e passamos os valores: **primeiro nome, último nome** e **data de nascimento** (no padrão **AAAA-MM-DD**, **AAAA – ano** com quatro dígitos, **MM-mês** com dois dígitos e **DD-dia** com dois dígitos).

Imprimimos o objeto **pessoa1** completo e o resultado é:

```
▶ Object { firstName: "John", lastName: "Doe", dob: main.js:64:9
  Date Thu Apr 03 1980 01:00:00 GMT+0100 (Horário
  Padrão da Europa Central) }
```

Buscamos o **ano de nascimento** do objeto **pessoa2** através do método **getFullYear**, que existe na classe **Date** (veja que o atributo **dob** é um **objeto criado com essa classe**) e o resultado é:
1970 main.js:66:9

Também, podemos **buscar o ano de nascimento** pelo método criado na classe **getBirthYear** e o resultado será o mesmo:

1970

main.js:67:9

Por fim, podemos usar o método **getFullName** da classe para busca do **nome completo do objeto pessoa1**.

John Doe

main.js:69:9

Lexical this no contexto de Funções

Podemos utilizar a palavra-chave **this** no **contexto de funções**. Em muitos casos, o valor **this** é determinado pela **forma como a função é chamada**. Por exemplo:

- **Em um método**, o **this** faz referência ao próprio objeto.
- **Sozinho**, o **this** faz referência ao objeto global.
- **Em uma função**, o **this** faz referência ao objeto global.
- **Em uma função**, no modo estrito, o **this** é **undefined**.
- **Em um evento**, o **this** faz referência ao elemento que disparou o evento.
- **Em método** como **call()** e **apply()** podem fazer referência do **this** para qualquer objeto.



Vamos praticar

No arquivo, **main.js** digite o seguinte código.

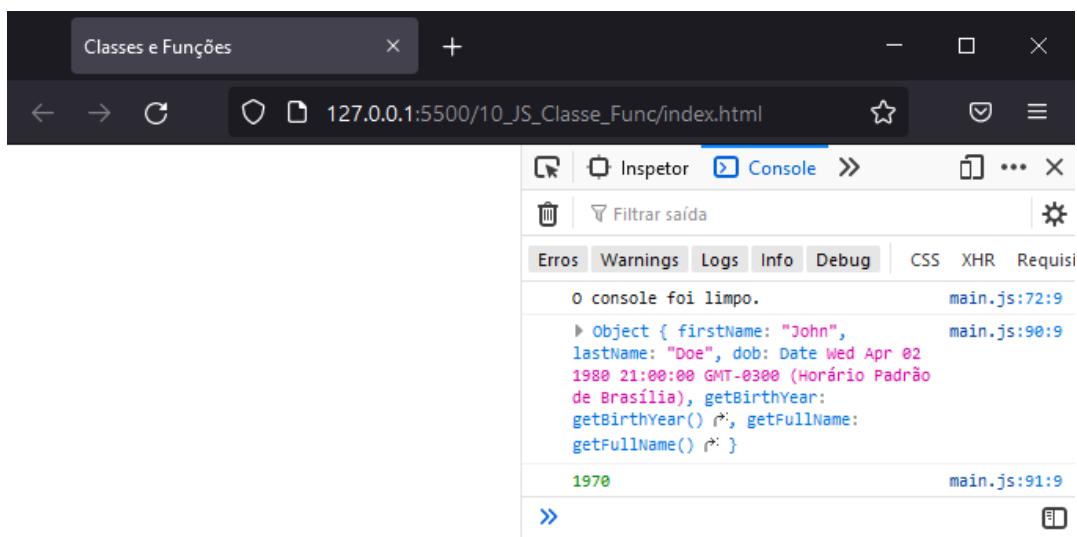
```
// Lexical this em uma função
console.clear();
// Constructor de função - Lexical this
function PessoaFunc(firstName, lastName, dob) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.dob = new Date(dob);
    this.getBirthYear = function () {
        return this.dob.getFullYear();
    };
    this.getFullName = function () {
        return `${this.firstName} ${this.lastName}`;
    };
}

// Instanciando os objetos
const pessoa3 = new PessoaFunc('John', 'Doe', '1980-04-03');
```

```
const pessoa4 = new PessoaFunc('Marry', 'Smith', '1970-06-13');

console.log(pessoa3);
console.log(pessoa4.dob.getFullYear());
```

O resultado é mostrado no console do navegador.



Nesse exemplo, criamos uma função com a mesma funcionalidade da **ClassePessoa**, criada anteriormente. Nele temos, os atributos:

```
this.firstName = firstName;
this.lastName = lastName;
this.dob = new Date(dob);
```

E os métodos:

```
this.getBirthYear = function () {
  return this.dob.getFullYear();
};
const getFirstName = () => {
  return `${this.firstName} ${this.lastName}`;
};
```

Observe, que nesse exemplo, o método **getBirthYear** foi criado com o **lexical this** e o método **getFirstName** foi criado com **Arrow Function**. Isso foi para mostrar que você tem várias maneiras de criar métodos em JS.

Em seguida, criamos as **instâncias de objetos com os valores iniciais** e **acessamos** esses valores para **imprimir no console**. Observe que essa função se comporta como a classe criada anteriormente. Portanto é importante você saber as **duas abordagens**, pois nos sistemas em

produção de uma empresa, você encontrará trechos de código com um outro tipo de implementação.

Prototype de objetos

O **JS** permite criar **protótipos de objetos**, que basicamente é permitir **criar atributos ou métodos em objetos já criados anteriormente**. E esses novos protótipos são **herdados automaticamente**.



Vamos praticar

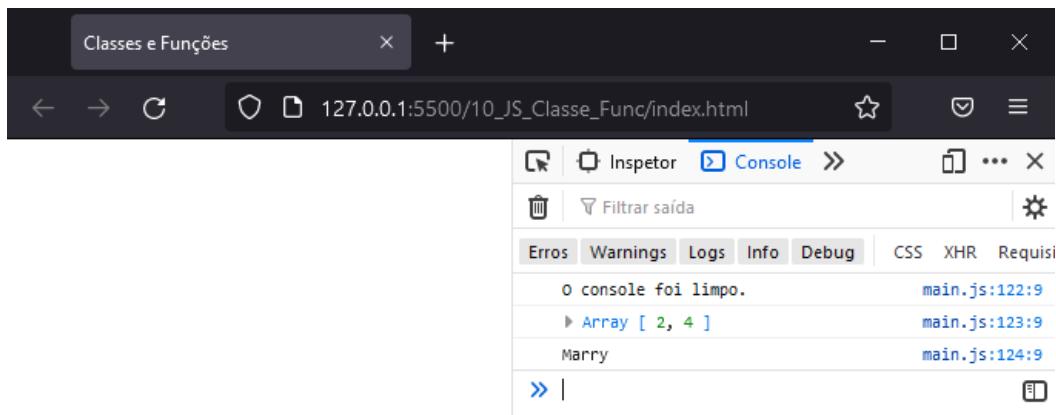
No arquivo, **main.js** digite o seguinte código.

```
// Adicionando protótipos
PessoaFunc.prototype.getBirthDayMonth = function () {
    let dados = [this.dob.getDate(), this.dob.getMonth() + 1];
    return dados;
};

PessoaFunc.prototype.getFirstName = function () {
    return `${this.firstName}`;
};

console.clear();
console.log(pessoa3.getBirthDayMonth());
console.log(pessoa4.getFirstName());
```

O resultado é mostrado no console do navegador.



Nesse exemplo, **dois protótipos novos** na função **PessoaFunc()**:

```
PessoaFunc.prototype.getBirthDayMonth = function () {
    let dados = [this.dob.getDate(), this.dob.getMonth() + 1];
```

```
    return dados;
};

PessoaFunc.prototype.getFirstName = function () {
    return `${this.firstName}`;
};
```

Esses métodos são **automaticamente herdados** pelos objetos **pessoa3** e **pessoa4**, que foram instanciados anteriormente. Por isso, podemos usar esses métodos normalmente.

Para aprender mais

Procure sempre aprender e estudar mais. Seguem alguns links para você estudar e aprender mais:

Classes:

<https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Classes>

https://www.w3schools.com/js/js_classes.asp

This

<https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Operators/this>

https://www.w3schools.com/js/js_this.asp

Modo estrito:

https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Strict_mode

https://www.w3schools.com/js/js_strict.asp

Prototype:

https://developer.mozilla.org/pt-BR/docs/Learn/JavaScript/Objects/Object_prototypes

https://www.w3schools.com/js/js_object_prototypes.asp



JavaScript HTML DOM - Parte 01

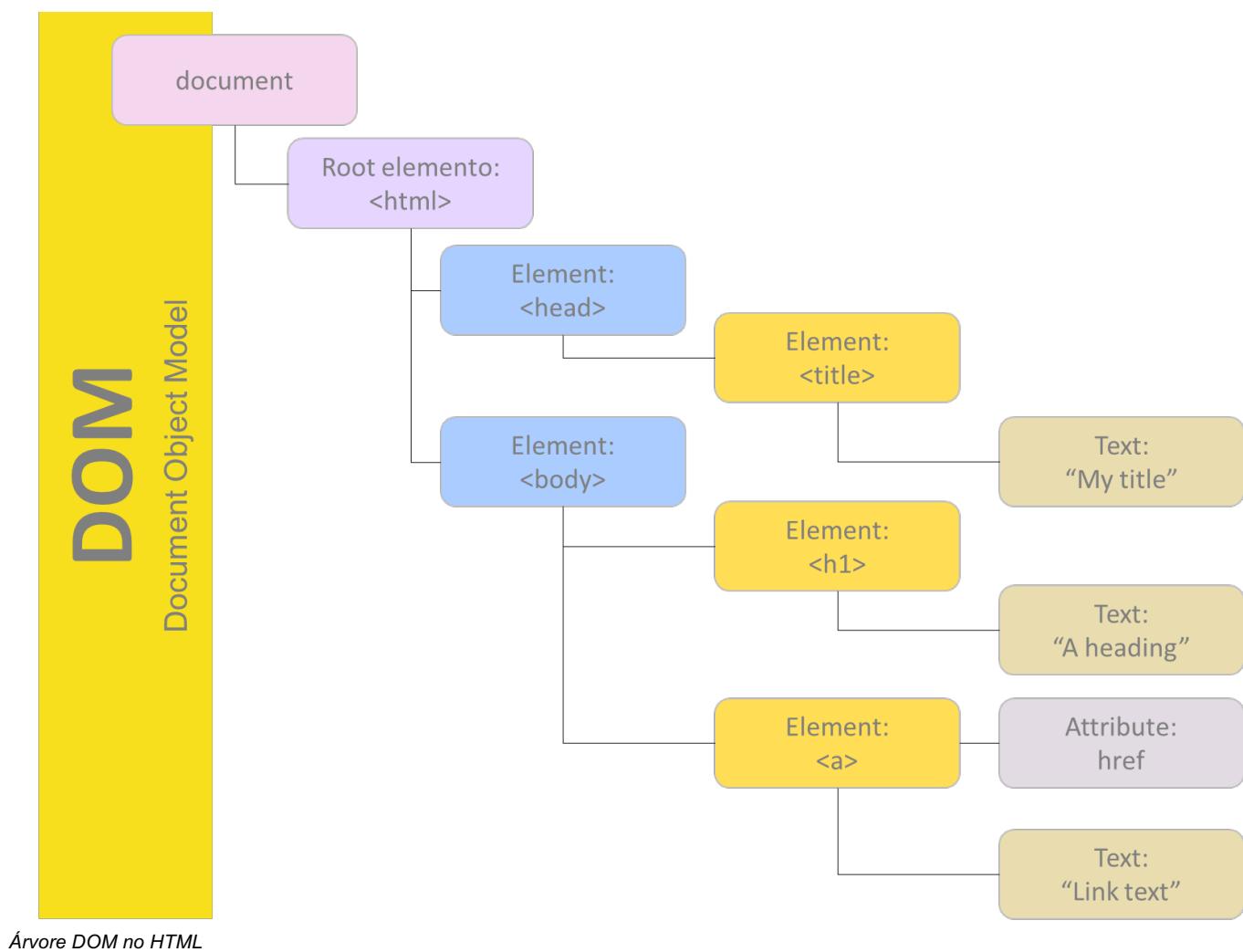
Os objetivos desta aula são:

- Compreender o uso do DOM (Document Object Model);
- Manipular elementos e estilos com JavaScript.

DOM

O **DOM** (**D**ocument **O**bject **M**odel) é uma interface de programação multiplataforma e independente de linguagem, que é usada em documentos **HTML**, **XHTML**, **XML** e **SVG**. Essa interface foi criada e é mantida pela entidade **World Wide Web Consortium (W3C)**.

Basicamente, o DOM **fornecer uma representação estruturada do documento como uma árvore**, onde os **elementos do documento** são como **nós das ramificações da árvore**. Para um documento HTML, a **imagem abaixo mostra a árvore DOM** com alguns elementos (**tag**). É importante você ver essa árvore na **língua inglesa mesmo**, pois o **HTML** e o **DOM** são baseados **no inglês** e você vai usar essas palavras na **língua original desses recursos**.



O **DOM** define **métodos/funções** que permitem **acesso a essa árvore**, de modo que permita **alterar a estrutura, estilo e conteúdo do mostrado no documento**. Não somente o documento pode ser manipulado como também os **nós (elementos)** podem ser **modificados através de eventos**. Desse modo, a **essência do DOM** é **conectar páginas web a scripts ou a linguagens de programação**.

Toda página carregada no navegador possui o objeto **document**, que está no topo da árvore e é o **único objeto nesse nível**. Esse objeto serve como **ponto de entrada para o conteúdo da página**, ou seja, ele permite utilizarmos um **script** ou uma **determinada linguagem de programação para manipular os elementos** existentes nessa página.

JavaScript HTML DOM

Até o momento, **utilizamos o console para exibir as mensagens com o resultado do processamento de alguma informação no JavaScript**. O console é importante para você fazer o **debug** de um código, aprender lógica de programação de uma determinada linguagem e ser um recurso de visualização do estado de uma aplicação, mas o Javascript é muito mais do isso. A principal aplicação do JavaScript é realizar a manipulação dos elementos do DOM, pois o **JavaScript foi criado para incorporar uma lógica computacional dentro de um ambiente de script**, que é o ambiente encontrado em um navegador web. Pois o navegador web consegue executar um script, mas não consegue executar um programa.

Com o **DOM**, Javascript tem o poder para criar conteúdo dinâmico em um documento HTML, permitindo o JS:

- alterar todos os elementos HTML na página,
- alterar todos os atributos HTML na página,
- mudar todos os estilos CSS na página,
- remover elementos e atributos HTML existentes,
- adicionar novos elementos e atributos HTML,
- reagir a todos os eventos HTML existentes na página e
- pode criar novos eventos HTML na página.

Objeto Window

O **objeto window** representa a **janela do navegador** que contém o elemento **DOM**. Foi esse objeto que permitiu utilizarmos o método **alert()** para exibir mensagens no navegador. Vamos fazer um pequeno projeto para ver esse objeto funcionando. Vamos criar o script no arquivo HTML mesmo, pois é somente uma demonstração didática.



Vamos praticar

Abra o **VS Code** e escolha um **diretório de trabalho para o seu projeto**.

Crie um diretório para seu projeto com o nome representativo, por exemplo, **JS_DOM_Parte_01**

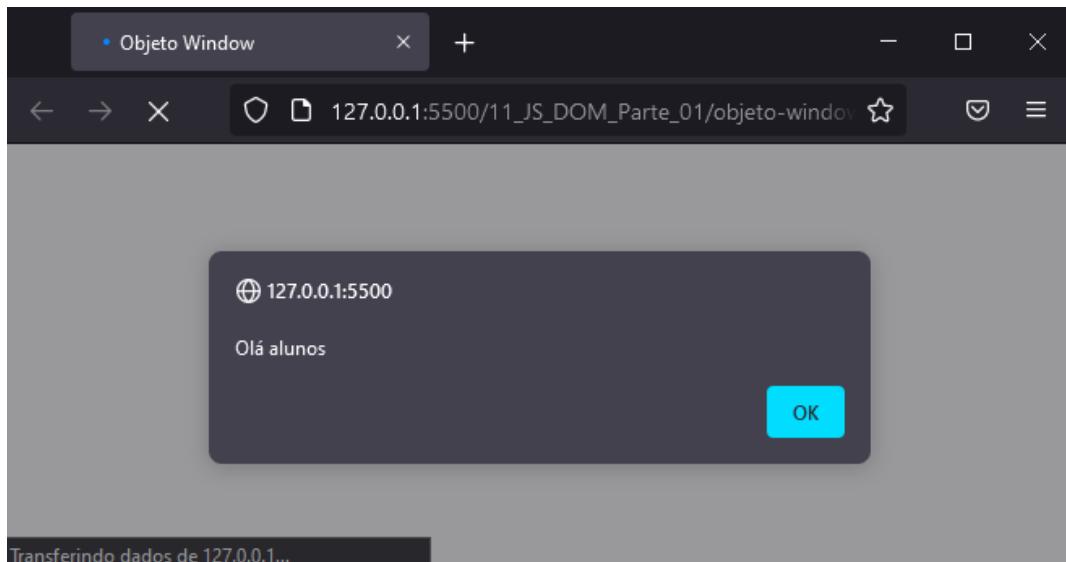
Crie um arquivo dentro do diretório do projeto com o nome **objeto-window.html**.

Insira o seguinte código no seu arquivo **objeto-window.html**.

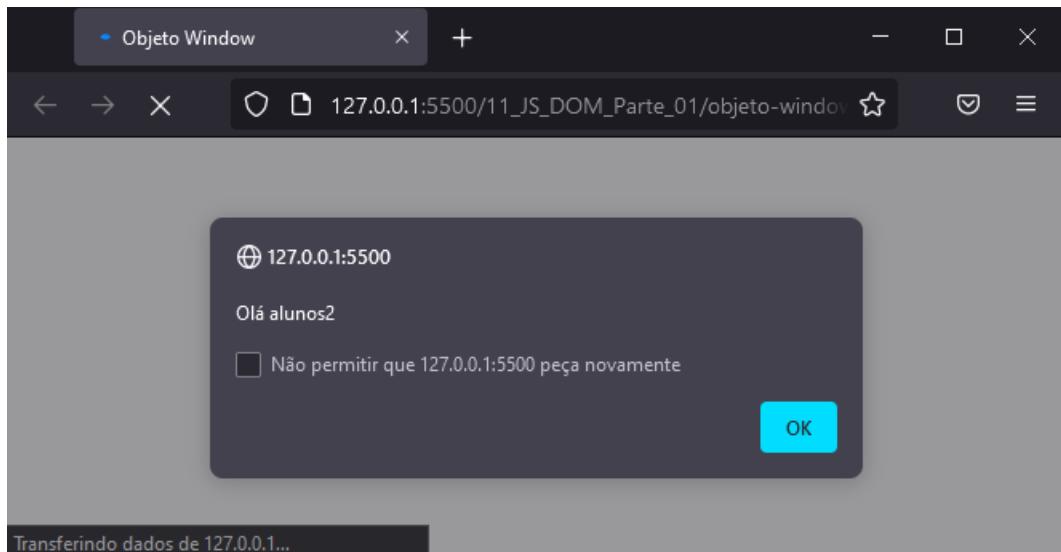
```
<!DOCTYPE html>
<html lang="pt-br">
  <head>
    <meta charset="UTF-8" />
    <link rel="shortcut icon" href="#" />
    <title>Objeto Window</title>
  </head>
  <body>
    <script>
      // objeto window
      console.log(window);

      window.alert('Olá alunos');
      alert('Olá alunos2');
    </script>
  </body>
</html>
```

Clique no botão  **Go Live** da extensão **Live Server** e abra as Ferramentas de desenvolvimento do navegador web (Atalho **F12**). O resultado é mostrado nas imagens abaixo.



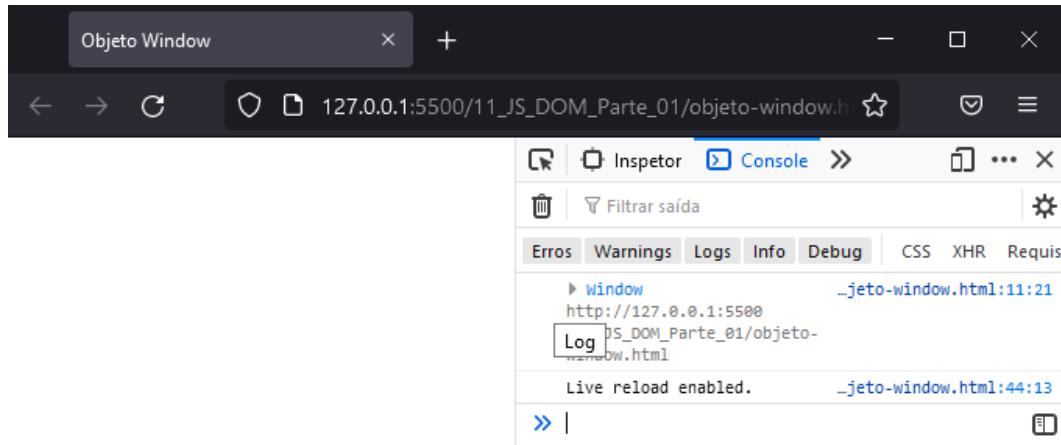
Primeiro o navegador mostra a primeira mensagem.



Após **clicar em OK** na primeira mensagem, o navegador mostra a segunda mensagem. Isso mostra que o objeto window não precisa estar explícito na chamada dos seus métodos.

```
window.alert('Olá alunos');
alert('Olá alunos2');
```

Se abrirmos o console para ver o que foi impresso, podemos ver o objeto **window** impresso no console.



Se expandirmos esse objeto, podemos ver mais detalhes dele com suas propriedades de métodos para manipularmos a janela de exibição do documento. Não se preocupe se as informações desse objeto parecerem diferentes para você, pois essas informações mudam de navegador para navegador.

```

▼ Window http://127.0.0.1:5500/03-      teste.html:11:21
JavaScript/11_JS_DOM/teste.html
  ► GetParams: function r(t) ↵
    __REACT_DEVTOOLS_APPEND_COMPONENT_
    STACK__: true
    __REACT_DEVTOOLS_BREAK_ON_CONSOLE_
    ERRORS__: false
    __REACT_DEVTOOLS_BROWSER_THEME__:
    "light"
  ►
    __REACT_DEVTOOLS_COMPONENT_FILTERS__
    : Array [ ... ]
    __REACT_DEVTOOLS_GLOBAL_HOOK__: »
    __REACT_DEVTOOLS_HIDE_CONSOLE_LOGS
    __IN_STRICT_MODE__: false
    __REACT_DEVTOOLS_SHOW_INLINE_WARNINGS_AND_ERRORS__: true
  ►
    __REDUX_DEVTOOLS_EXTENSION_COMPOSE__
    : function
    __REDUX_DEVTOOLS_EXTENSION_COMPOSE__
    () ↵
    ► __REDUX_DEVTOOLS_EXTENSION__:
    function x(t, e, n) ↵
    ► devToolsExtension: function
    devToolsExtension() ↵
    ► <default properties>
    ► <get
    __REACT_DEVTOOLS_GLOBAL_HOOK__():
    function get() ↵
    ► <prototype>: WindowPrototype { ... }
  
```

Alguns dos métodos do objeto **window** são mostrados abaixo:

Método	Descrição
alert()	Mostra uma caixa de alerta contendo uma mensagem e um botão de OK.
confirm()	Mostra uma caixa de diálogo contendo uma mensagem com um botão de OK e um de CANCEL.
prompt()	Mostra uma caixa de diálogo para o usuário entrar com algum dado.
open()	Abre uma nova janela no navegador.
close()	Fechá a janela atual do navegador.
setTimeout()	Executa uma ação após um determinado tempo. Essa ação pode ser uma chamada de função, uma expressão de avaliação, etc.

Métodos do DOM

Como foi dito toda página possui um objeto **document** e ele é a **porta de acesso para todos os elementos da página web**. Desse modo, é possível pelo JavaScript utilizar os diversos métodos do DOM.

Vamos praticar

Vamos ver no console, algumas das informações que esse objeto carrega. Siga os passos para criar o projeto:

No diretório **JS_DOM_Parte_01**, crie um arquivo dentro do diretório do projeto com o nome **index.html** e insira o seguinte código no seu arquivo **index.html**.

```
<!DOCTYPE html>
<html lang="pt-br">
  <head>
    <meta charset="UTF-8" />
    <link rel="shortcut icon" href="#" />
    <title> JavaScript com HTML DOM – Parte 01</title>
  </head>
  <body>
    <script src=".js/main.js"></script>
  </body>
</html>
```

Observe que **mudamos a organização dos arquivos no projeto**, agora o arquivo **main.js** está em uma pasta chamada **js**, enquanto o arquivo **index.html** está no **diretório raiz do projeto**. Isso é importante para você ser uma pessoa organizada com os seus projetos. Além disso, o caminho do script agora está com apontando para o novo diretório (**<script src=".js/main.js"></script>**).

Vamos deixar o arquivo **main.js vazio** e à medida que vamos aprendendo coisas novas vamos completando a implementação do código **JavaScript**.

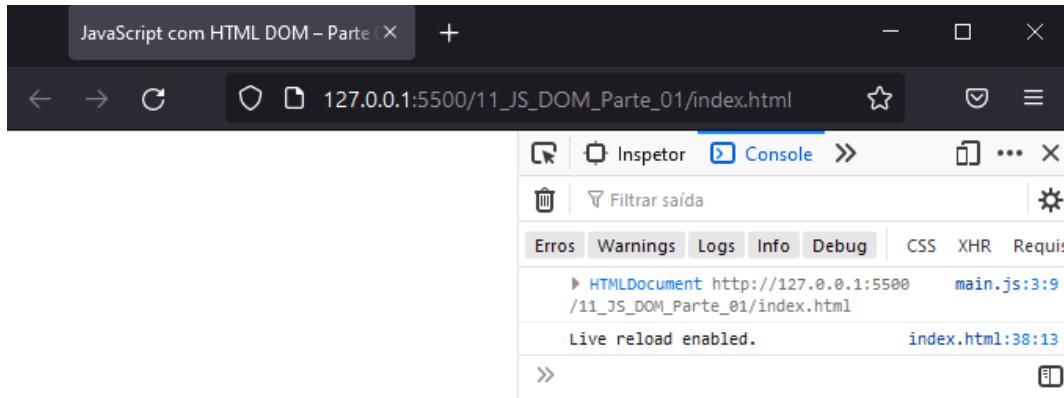
Abra o arquivo **index.html**, clique no botão  **Go Live** da extensão **Live Server** e abra as Ferramentas de desenvolvimento do navegador web (Atalho **F12**).

Agora, vamos implementar todo o nosso código desse tema no arquivo **main.js** e ver as mensagens impressas no console.

No arquivo, **main.js** digite o seguinte código.

```
// JavaScript e DOM
// objeto document
console.log(document);
```

Abra o arquivo **index.html**, clique no botão  **Go Live** da extensão **Live Server** e abra as Ferramentas de desenvolvimento do navegador web (Atalho **F12**). Veja o objeto impresso no console.



Se expandirmos o objeto **HTMLDocument**, podemos ver algumas das informações carregados por ele.

```
▼ HTMLDocument http://127.0.0.1:5500/03-                main.js:4:9
  JavaScript/11_JS_DOM/index.html
    URL: "http://127.0.0.1:5500/03-
  JavaScript/11_JS_DOM/index.html"
    ▶ activeElement: <body> ⚡
      alinkColor: ""
    ▶ all: HTMLAllCollection { 0: html ⚡ ,
      1: head ⚡ , length: 8, ... }
    ▶ anchors: HTMLCollection { length: 0 }
    ▶ applets: HTMLCollection { length: 0 }
    baseURI: "http://127.0.0.1:5500/03-
  JavaScript/11_JS_DOM/index.html"
    bgColor: ""
    ▶ body: <body> ⚡
      characterSet: "UTF-8"
      charset: "UTF-8"
      childElementCount: 1
    ▶ childNodes: NodeList [ <!DOCTYPE html>,
      html ⚡ ]
    ▶ children: HTMLCollection { 0: html ⚡ ,
      length: 1 }
      compatMode: "CSS1Compat"
      contentType: "text/html"
      cookie: ""
      currentScript: null
    ▶ defaultView: Window
    http://127.0.0.1:5500/03-JavaScript
    /11_JS_DOM/index.html
      designMode: "off"
      dir: ""
    ▶ doctype: <!DOCTYPE html>
    ▶ documentElement: <html lang="en"> ⚡
      documentURI: "http://127.0.0.1:5500/03-
  JavaScript/11_JS_DOM/index.html"
      domain: "127.0.0.1"
```

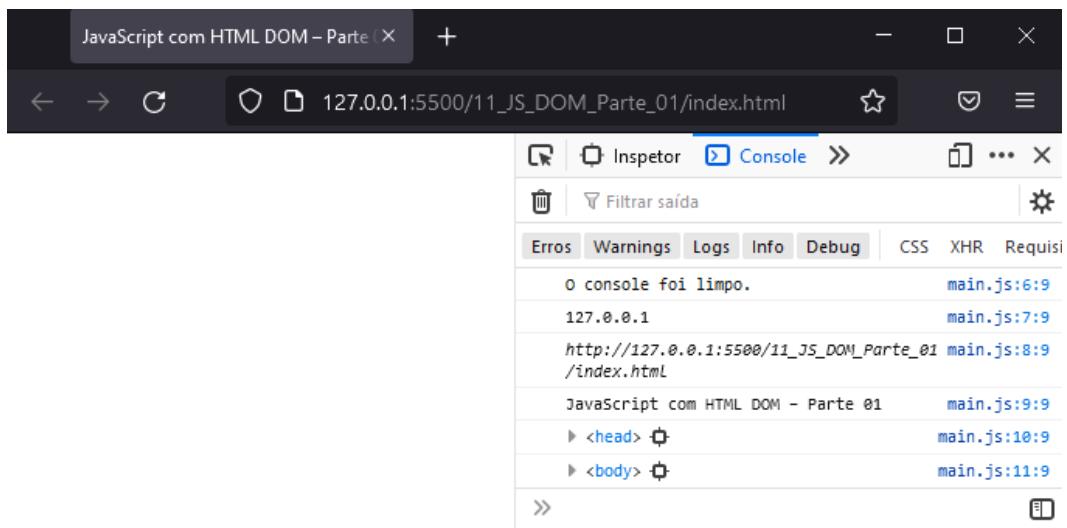
Só esse pedaço mostrado no podemos ver diversas propriedades e elementos como: **URL**, **head**, **body**, **doctype**, etc. Vamos testar acessar algumas dessas informações.

No arquivo **main.js**, insira o seguinte código.

```
// Acessar informações
console.clear();
console.log(document.domain);
console.log(document.URL);
```

```
console.log(document.title);
console.log(document.head);
console.log(document.body);
```

Veja as informações impressas no console.



A primeira instrução **imprime o domínio que o documento está aberto**:

127.0.0.1 main.js:7:9

A segunda instrução **imprime URL que o documento está aberto**:

http://127.0.0.1:5500/03-JavaScript/11_JS_DOM main.js:8:9
/index.html

A terceira instrução **imprime o título do documento**:

JavaScript com HTML DOM main.js:9:9

A quarta instrução **imprime o elemento <head> do documento**:

▶ <head> main.js:10:9

A ultima instrução **imprime o elemento <body> do documento**:

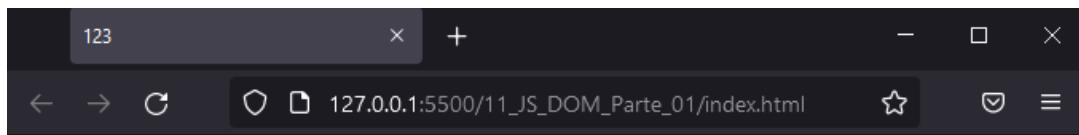
▶ <body> main.js:11:9

Podemos alterar qualquer informação se quisermos. Por exemplo, vamos **alterar o título da página e criar um elemento <h1>** no **body** da página.

No arquivo, **main.js** digite o seguinte código.

```
// Alterar o título
document.title = 123;
// Criar um elemento <h1>
let heading = document.createElement('H1');
heading.innerHTML = 'Olá alunos!';
document.body.appendChild(heading);
cabecalho.style.borderBottom = 'solid 3px #000';
```

Vamos ver o resultado mostrado no console.



Olá alunos!

Observe que a instrução, **alterar o título da página** para **123**. E em seguida, é **criada** a tag **<h1>** e **inserido um conteúdo** dentro desse elemento para então ser exibido dentro do **<body>** da página.

O método **createElement()** cria um novo nó/elemento com um nome específico. Esse nome deve pode ser com letras maiúsculas ou minúsculas.

- Criar um elemento botão
- Criar um elemento parágrafo
- Criar um elemento imagem

```
document.createElement('BUTTON');
document.createElement('p');
document.createElement('img');
```

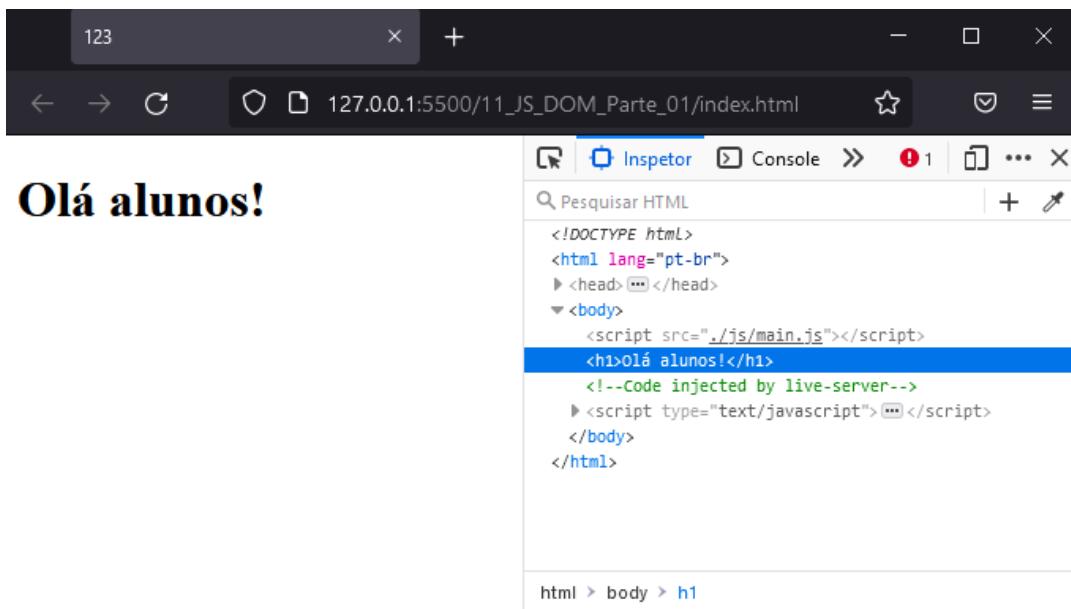
A propriedade **innerHTML** define ou retorna o conteúdo de um elemento HTML. E o método **appendChild()** adiciona o nó/elemento criado em um elemento pai específico.

```
document.body.appendChild(heading);
```

No exemplo, o elemento **heading** foi adiciona dentro do elemento **<body>**.

A propriedade **style** permite configurar o **estilo CSS desse novo elemento**.

Se usarmos o recurso de **Elements** (no navegador **Chrome**) ou **Inspetor** (no navegador **Firefox**), podemos ver novo elemento inserido no código HTML da página.



The screenshot shows a browser window with the URL `127.0.0.1:5500/11_JS_DOM_Parte_01/index.html`. The page content is "Olá alunos!". Below the content, the browser's developer tools are open, specifically the "Inspector" tab. The DOM tree is visible, showing the structure of the HTML document. A blue highlight is placed over the `<hi>Olá alunos!</hi>` node. The status bar at the bottom of the developer tools indicates the path: `html > body > h1`.

Para aprender mais

Procure sempre aprender e estudar mais. Seguem alguns links para você estudar e aprender mais:

DOM:

https://pt.wikipedia.org/wiki/Modelo_de_Objetos_de_Documentos

<https://developer.mozilla.org/pt-BR/docs/Glossary/DOM>

https://developer.mozilla.org/pt-BR/docs/Web/API/Document_Object_Model

https://www.w3schools.com/js/js_htmldom.asp

Objeto window:

<https://developer.mozilla.org/pt-BR/docs/Web/API/Window>

Objeto document:

<https://developer.mozilla.org/pt-BR/docs/Web/API/Document>

Método createElement():

<https://developer.mozilla.org/pt-BR/docs/Web/API/Document/createElement>

Propriedade innerHTML:

<https://developer.mozilla.org/pt-BR/docs/Web/API/Element/innerHTML>

Método appendChild():

<https://developer.mozilla.org/pt-BR/docs/Web/API/Node/appendChild>



JavaScript HTML DOM - Parte 02

Os objetivos desta aula são:

- Compreender o uso do DOM (Document Object Model);
- Conhecer os métodos getElementById, getElementsByClassName, getElementByTagName e getElementsByTagName.

Método getElementById

O método **getElementById** é o modo comum de **acessar um elemento HTML** na página web. Esse método retorna a **referência do elemento** através do atributo **ID** ou **null** se a **ID não for encontrada**. Esse é um dos motivos do atributo **id** em uma página web ser **único**.



Vamos praticar!

Abra o **VS Code** e escolha um **diretório de trabalho para o seu projeto**.

Crie um diretório para seu projeto com o nome representativo, por exemplo, **JS_DOM_Parte_02**

Crie um arquivo dentro do diretório do projeto com o nome **index.html**.

Insira o seguinte código no seu arquivo **index.html**.

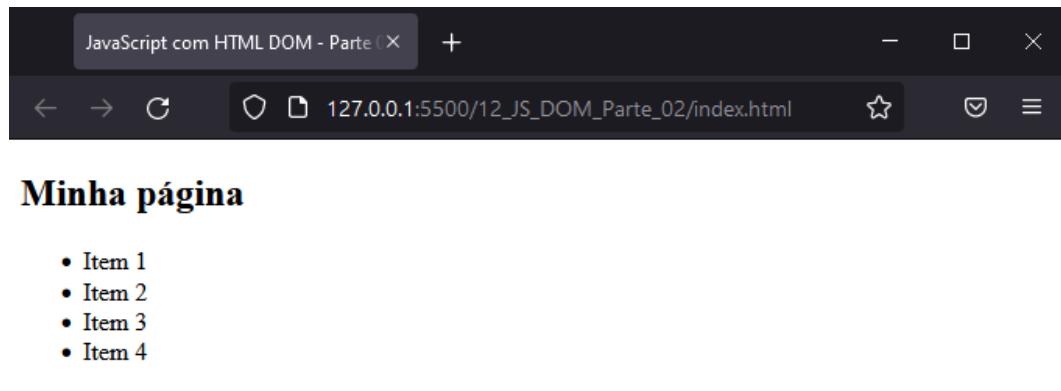
```
<!DOCTYPE html>
<html lang="pt-br">
  <head>
    <meta charset="UTF-8" />
    <link rel="shortcut icon" href="#" />
    <title>JavaScript com HTML DOM - Parte 02</title>
  </head>
  <body>
    <h2 id="titulo">Minha página</h2>

    <ul id="items" class="list">
      <li class="item">Item 1</li>
      <li class="item">Item 2</li>
      <li class="item" name="fitem">Item 3</li>
      <li class="item" name="fitem">Item 4</li>
    </ul>

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

Observe no arquivo **index.html**, que colocamos o atributo **id** na marcação **<h2>**.

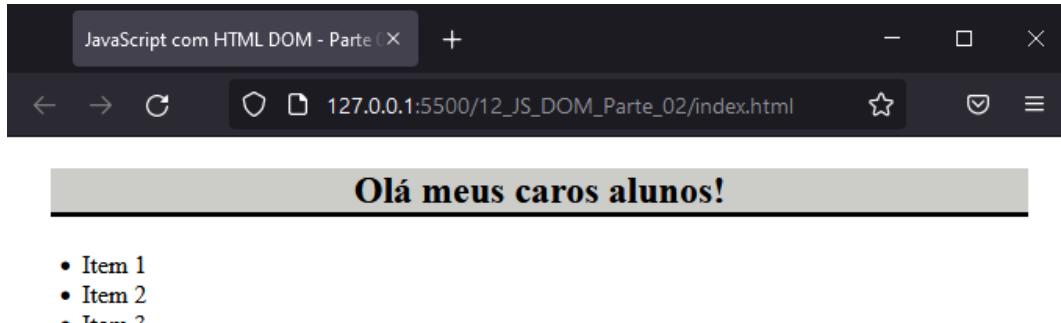
Clique no botão **Go Live** da extensão **Live Server** e abra as Ferramentas de desenvolvimento do navegador web (Atalho **F12**). O resultado é mostrado nas imagens abaixo.



No arquivo **main.js**, insira o seguinte código.

```
// Método getElementById
let titulo = document.getElementById('titulo');
// Alterando o conteúdo do elemento
titulo.innerHTML = 'Olá meus caros alunos!';
// Configurando o estilo CSS do elemento
titulo.style.textAlign = 'center';
titulo.style.backgroundColor = '#CCCCCC9';
titulo.style.borderBottom = 'solid 3px #000';
titulo.style.margin = '20px';
```

Veja o resultado apresentado no navegador.



Observe que a instrução **criou uma referência** do elemento **<h2>** da página e, então foi possível **manipular o conteúdo** e as **configurações de estilo** desse elemento.

Método getElementsByClassName

O método **getElementsByClassName** retorna a coleção de todos os elementos do documento com a classe específica. Isso significa que esse método pode retornar mais de um elemento.



Vamos praticar

Continuando a implementação do projeto, no arquivo **main.js**, insira o seguinte código.

```
// Método getElementsByClassName
let items = document.getElementsByClassName('item');
console.log(items);
console.log(items[1]);
items[1].textContent = 'Hello 2';
items[1].style.fontWeight = 'bold';
items[1].style.backgroundColor = 'yellow';
```

Veja o resultado apresentado no navegador.



Olá meus caros alunos!

- Item 1
- Hello 2
- Item 3
- Item 4

Buscamos todos os itens da lista, pois todos eles têm a classe **item** associada a eles. A instrução seguinte **imprime no console** os **itens retornado pelo método** e armazenados no objeto **items**:

```
▼ HTMLCollection { 0: li.item , 1: li.item , 2: li.item , 3: li.item , length: 4 }
  |   ▶ 0: <li class="item">
  |   ▶ 1: <li class="item" style="font-weight: bold; background-color: yellow;">
  |   |   ▶ 2: <li class="item">
  |   |   ▶ 3: <li class="item">
  |   |   length: 4
  |   |   ▶ <prototype>: HTMLCollectionPrototype {
  |   |   |   item(), namedItem(), length: Getter, ... }
```

Como podemos ver o objeto **items** possui vários itens dentro dele e podemos acessar individualmente como mostra a instrução, que resulta na impressão no console do segundo item do objeto:

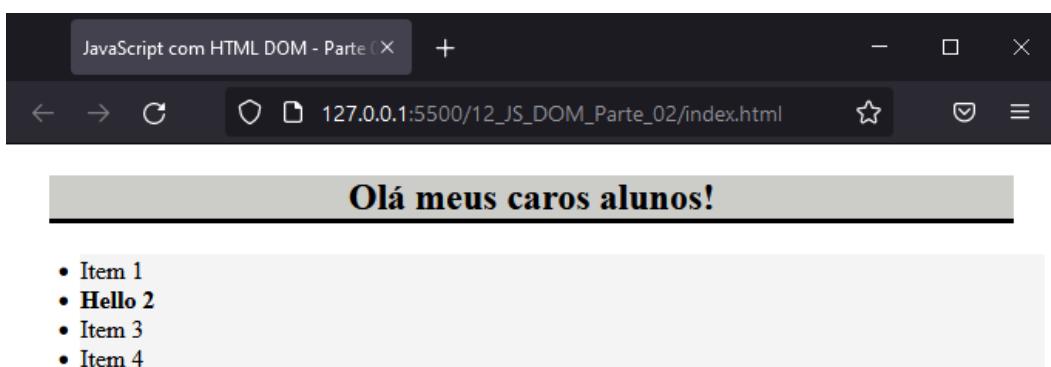
```
▶ <li class="item" style="font-weight: bold; background-color: yellow;"> main.js:14:9
```

As próximas instruções alteram o conteúdo do segundo item da lista, a espessura da fonte e a cor de fundo, respectivamente.

Podemos utilizar um **laço de repetição** para **alterar a cor de fundo de todos os itens da lista**, no arquivo **main.js**, insira o seguinte código.

```
for (let i = 0; i < items.length; i++) {
    items[i].style.backgroundColor = '#f4f4f4';
}
```

Veja o resultado apresentado no navegador.



O laço de repetição **percorre os itens dentro do objeto** e modifica a **cor de fundo** de cada item referenciando-o pelo **índice do array**.

Método getElementsByTagName

O método **getElementsByTagName** retorna a coleção de todos os elementos do documento com a tag, ou seja, com a mesma marcação HTML. Isso significa que esse método pode retornar mais de um elemento.



Vamos praticar

Continuando a implementação do projeto, no arquivo **main.js**, insira o seguinte código.

```
// Método getElementsByTagName
let li = document.getElementsByTagName('li');
console.log(li);

for (let i = 0; i < li.length; i++) {
    if (i % 2) li[i].style.backgroundColor = '#f4f4f4';
    else li[i].style.backgroundColor = '#fff';
}
```

Veja o resultado apresentado no navegador.



No nosso exemplo, o método `getElementsByClassName` e `getElementsByTagName` permitiram manipular os **mesmos elementos**. Isso foi intencional para mostrar que existem **diversas maneiras de acessar um elemento no DOM**.

Depois, alteramos as cores de fundo dos elementos ``, os elementos com índices **pares** são configurados com **fundo branco (#fff)** e os elementos com índices **ímpares** são configurados com **fundo gelo (#f4f4f4)**. Lembre-se, o primeiro elemento do **array** possui **índice zero**, portanto com quarto elemento na lista temos: **índice 0, índice 1, índice 2 e índice 3**.

Método `getElementsByName`

O método `getElementsByName` retorna a coleção de **todos os elementos do documento** com o **nome específico**, ou seja, esse método verifica o atributo **HTML name**. Isso significa que esse método pode retornar mais de um elemento.



Vamos praticar

Continuando a implementação do projeto, no arquivo `main.js`, insira o seguinte código.

```
// Método getElementsByName
let nome = document.getElementsByName('fitem');
console.clear();
console.log(nome);
nome[0].textContent = 'Olá pessoas';
nome[0].style.backgroundColor = 'yellow';
nome[1].textContent = 'Tudo bem?';
nome[1].style.backgroundColor = '#BAC1FB';
```

Veja o resultado apresentado no navegador.



Olá meus caros alunos!

- Item 1
- Hello 2
- Olá pessoas
- Tudo bem?

O método **getElementsByName** retornou os dois últimos itens da lista, pois eles têm o nome igual a item:

```
<ul id="items" class="list">
  <li class="item" id="item1">Item 1</li>
  <li class="item" id="item2">Item 2</li>
  <li class="item" name="fitem">Item 3</li>
  <li class="item" name="fitem">Item 4</li>
</ul>
```

E então foi possível acessar as propriedades **textContent** e **style** para alterar o conteúdo e o estilo desses elementos. A instrução `nome[0].textContent = 'Olá pessoas'` altera o conteúdo do terceiro item da lista e `nome[0].style.backgroundColor = 'yellow'` altera a cor de fundo para amarelo. A instrução `nome[1].textContent = 'Tudo bem?'` altera o conteúdo do quarto item da lista e `nome[1].style.backgroundColor = '#BAC1FB'` altera a cor de fundo para azul claro.

Remover um elemento no DOM

O método **remove()** permite remover um elemento do DOM.



Vamos praticar

Continuando a implementação do projeto, no arquivo **main.js**, insira o seguinte código.

```
// Remove elemento do DOM
let item2 = document.getElementById('item2');
item2.remove();
```

Veja o resultado apresentado no navegador.



Olá meus caros alunos!

- Item 1
- Olá pessoas
- Tudo bem?

Observe que o **item2** não aparece mais na página, pois ele foi removido com o método **remove()**. Vamos retornar esse elemento novamente.

Continuando a implementação do projeto, no arquivo **main.js**, insira o seguinte código.

```
// Retornando o elemento
let lista = document.getElementById('items');
let item1 = document.getElementById('item1');
lista.insertBefore(item2, item1.nextSibling);
```

Veja o resultado apresentado no navegador.



Olá meus caros alunos!

- Item 1
- Hello 2
- Olá pessoas
- Tudo bem?

Observe que o **item2** foi **devolvido para o seu lugar**. A instrução busca o **elemento pai** dos **itens da lista** e a instrução seguinte busca o **elemento filho**, que queremos **ter como referência para inserir o item2**. Nesse caso, queremos inserir o **item2**, após o **item1**.

A próxima instrução **insere o item2** dentro da lista pelo método **insertBefore** e a propriedade **nextSibling** retorna o **próximo elemento irmão (que tem o mesmo pai)**. Desse modo, o **item2** pôde ser inserido após o seu irmão **item1**.

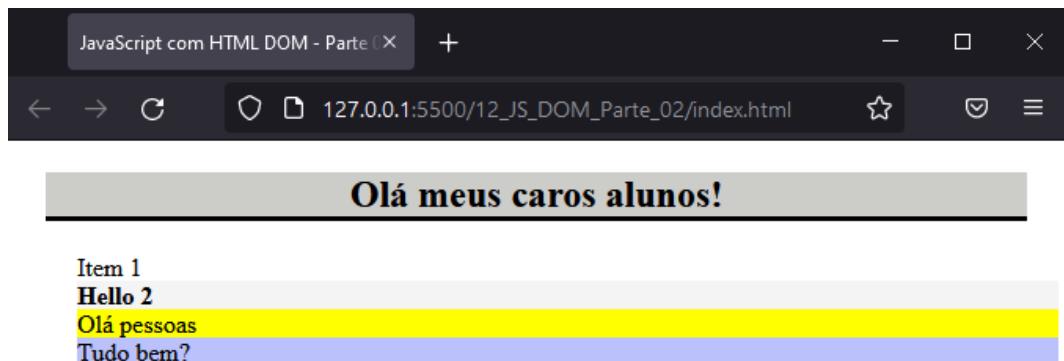
Alterar estilo da lista

Vamos alterar o **estilo da lista não ordenada**. Siga os passos para fazer isso:

Continuando a implementação do projeto, no arquivo **main.js**, insira o seguinte código.

```
// Alterando o estilo da lista
let ul = document.getElementById('items');
ul.style.listStyle = 'none';
/
```

Veja o resultado apresentado no navegador.



A propriedade **listStyle** configura o **estilo da lista** e no exemplo para não mostrar o marcador da lista (**o bullet point**).

No arquivo **main.js**, insira o seguinte código.

```
// Marcadores numéricos
ul.style.listStyle = 'decimal inside';
```

Veja o resultado apresentado no navegador.



Agora é exibido os marcadores numéricos na lista.

Para aprender mais

Procure sempre aprender e estudar mais. Seguem alguns links para você estudar e aprender mais:

`getElementById`:

<https://developer.mozilla.org/pt-BR/docs/Web/API/Document/getElementById>

https://www.w3schools.com/js/js_htmldom_methods.asp

getElementsByClassName:

<https://developer.mozilla.org/pt-BR/docs/Web/API/Document/getElementsByClassName>

https://www.w3schools.com/Jsref/met_document_getelementsbyclassname.asp

getElementsByTagName:

<https://developer.mozilla.org/pt-BR/docs/Web/API/Document/getElementsByTagName>

https://www.w3schools.com/Jsref/met_document_getelementsbytagname.asp

getElementsByName:

<https://developer.mozilla.org/pt-BR/docs/Web/API/Document/getElementsByName>

https://www.w3schools.com/Jsref/met_doc_getelementsbyname.asp

Element.remove():

<https://developer.mozilla.org/en-US/docs/Web/API/Element/remove>



JavaScript HTML DOM - Parte 03

Os objetivos desta aula são:

- Compreender o uso do DOM (Document Object Model);
- Conhecer os eventos em JavaScript;
- Aprender os métodos querySelector e querySelectorAll.

Método querySelector

O método **querySelector** retorna o primeiro elemento dentro do documento que corresponde ao **seletor desejado**. Caso **nenhum elemento seja encontrado**, o método retornará **null**. A sintaxe do método é:

```
elemento = document.querySelector(seletores);
```

Onde:

- **elemento**: é o objeto do JavaScript para armazenar o elemento encontrado.
- **seletores**: é uma string com um ou mais seletores desejados. Para mais de um seletor, deve-se separá-los por vírgula.

Por exemplo:

```
let paragrafo = document.querySelector("p");

let parClasse = document.querySelector("p.exemplo");

let elemento = document.querySelector(".myclass");

const myForm = document.querySelector('#my-span');
```

A primeira instrução busca o **primeiro elemento parágrafo no documento**. A **tag entre aspas** em ponto final ou **# indica que queremos buscar um elemento/tag/marcação HTML**.

A segunda instrução busca o **primeiro elemento parágrafo** que possui a classe **.exemplo** no documento.

A terceira instrução busca o **primeiro elemento que possui a classe chamada myclass**. O ponto final no seletor (**.myclass**) indica que queremos buscar o elemento pela classe.

A quarta instrução busca o primeiro elemento que possui o atributo **id** chamado **my-span**. A **hashtag** (ou trilha, ou jogo da velha, diferentes nomes para esse símbolo #) indica que desejamos procurar o **elemento pela id**.



Vamos praticar!

Abra o **VS Code** e escolha um **diretório de trabalho para o seu projeto**.

Crie um diretório para seu projeto com o nome representativo, por exemplo, **JS_DOM_Parte_03**

Crie um arquivo dentro do diretório do projeto com o nome **index.html**.

Insira o seguinte código no seu arquivo **index.html**.

```

<!DOCTYPE html>
<html lang="pt-br">

<head>
    <meta charset="UTF-8" />
    <link rel="shortcut icon" href="#" />
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.0.1/dist/css/bootstrap.min.css" rel="stylesheet"
    integrity="sha384-fXqyf8fT4ZVJzWQ9j4oWVQDfHdPbWQcLqWZL1tAjYBQDgkCw=" crossorigin="anonymous" />
        <title>JavaScript com HTML DOM - Parte 03</title>
</head>

<body>
    <header>
        <h1 class="container text-center p-3 bg-light">
            Mais métodos do JS
        </h1>
    </header>
    <main class="container border">
        <p>Meu parágrafo sem classe.</p>
        <hr />
        <p class="exemplo">Meu parágrafo com a classe .exemplo</p>
        <hr />
        <div class="myclass">Meu elemento div com a classe .myclass</div>
        <hr />
        <span id="my-span">Meu elemento span com a id #my-span.</span>
        <hr />
        <br />
        <button class="btn btn-primary" onclick="Clique()">
            Clique aqui
        </button>
        <hr />
        <ul class="items">
            <li id="my-li" class="item">Item 1</li>
            <li class="item teste">Item 2</li>
            <li class="item teste">Item 3</li>
            <li id="my-li" class="item">Item 4</li>
        </ul>
    </main>
    <script src="./js/main.js"></script>
</body>

</html>

```

Observe que **mudamos a organização dos arquivos no projeto**, agora o arquivo **main.js** está em uma pasta chamada **js**, enquanto o arquivo **index.html** está no **diretório raiz do projeto**.

Isso é importante para você ser uma pessoa organizada com os seus projetos. Além disso, o caminho do script agora está com apontando para o novo diretório (`<script src=".js/main.js"></script>`).

Vamos deixar o arquivo **main.js vazio** e à medida que vamos aprendendo coisas novas vamos completando a implementação do código **JavaScript**.

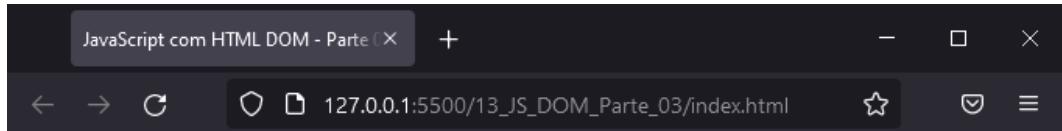
Abra o arquivo index.html, clique no botão  Go Live da extensão **Live Server** e abra as Ferramentas de desenvolvimento do navegador web (Atalho **F12**).

Agora, vamos implementar todo o nosso código desse tema no arquivo **main.js** e ver as mensagens impressas no console.

No arquivo, **main.js** digite o seguinte código.

```
// Método querySelector
let paragrafo = document.querySelector('p');
paragrafo.style.background = '#FBEBEA';
```

Veja o resultado apresentado no navegador.



The screenshot shows a browser window titled "JavaScript com HTML DOM - Parte 03". The address bar indicates the page is at `127.0.0.1:5500/13_JS_DOM_Parte_03/index.html`. The main content area displays the heading "Mais métodos do JS" and four paragraphs with different styles applied:

- "Meu parágrafo sem classe." (background-color: #FBEBEA)
- "Meu parágrafo com a classe .exemplo"
- "Meu elemento div com a classe .myclass"
- "Meu elemento span com a id #my-span."

Below the paragraphs is a blue button labeled "Clique aqui". At the bottom of the page is a list of four items:

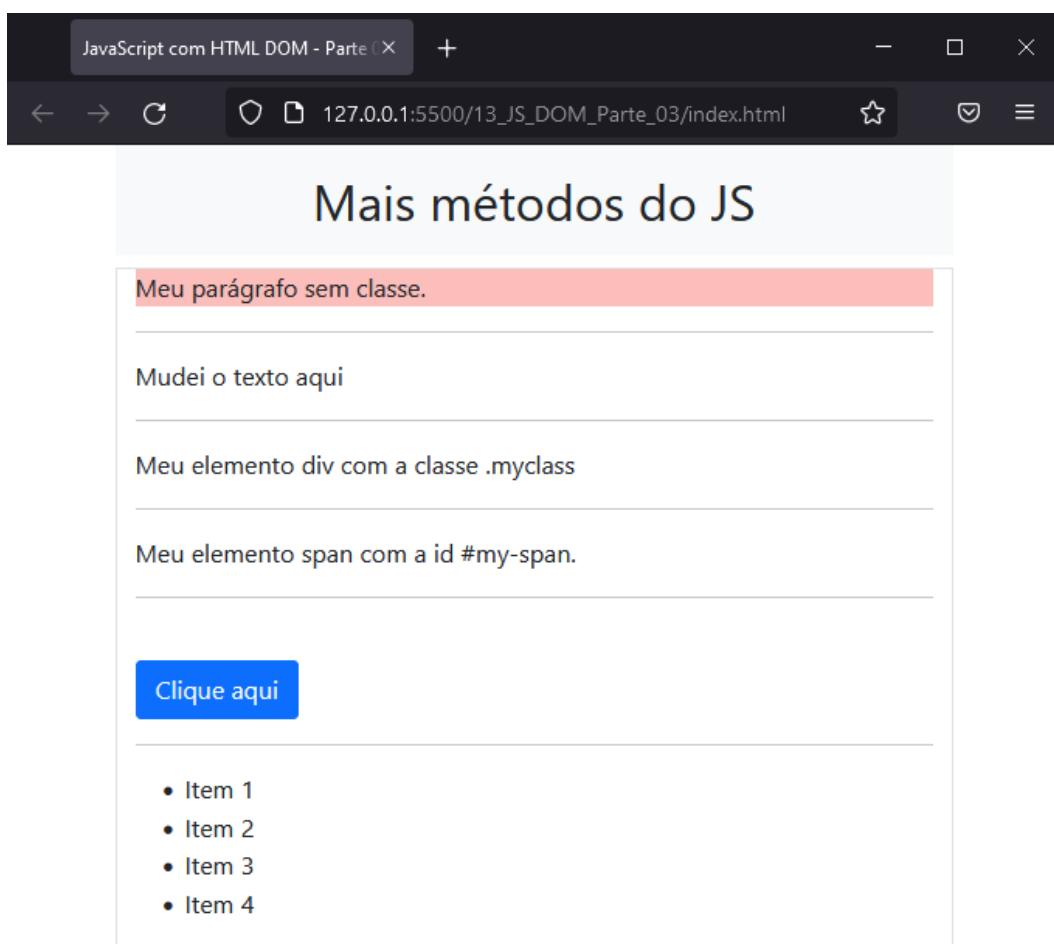
- Item 1
- Item 2
- Item 3
- Item 4

Observe que o método busca pelo **elemento com a parágrafo** do documento. Então, esse elemento é **retornado** e, assim, é possível alterarmos a **cor de fundo do parágrafo**. Note que o outro elemento parágrafo não teve sua cor de fundo alterado.

Continuando a implementação do projeto, no arquivo **main.js**, insira o seguinte código.

```
// Busca a tag e a classe
let parClasse = (document.querySelector('p.exemplo').innerHTML =
    'Mudei o texto aqui');
```

Veja o resultado apresentado no navegador.



Observe que, o texto do segundo parágrafo que possui a classe **.exemplo**. O texto do primeiro parágrafo ficou intacto. O interessante dessa instrução é que podemos fazer dois comandos de uma vez só:

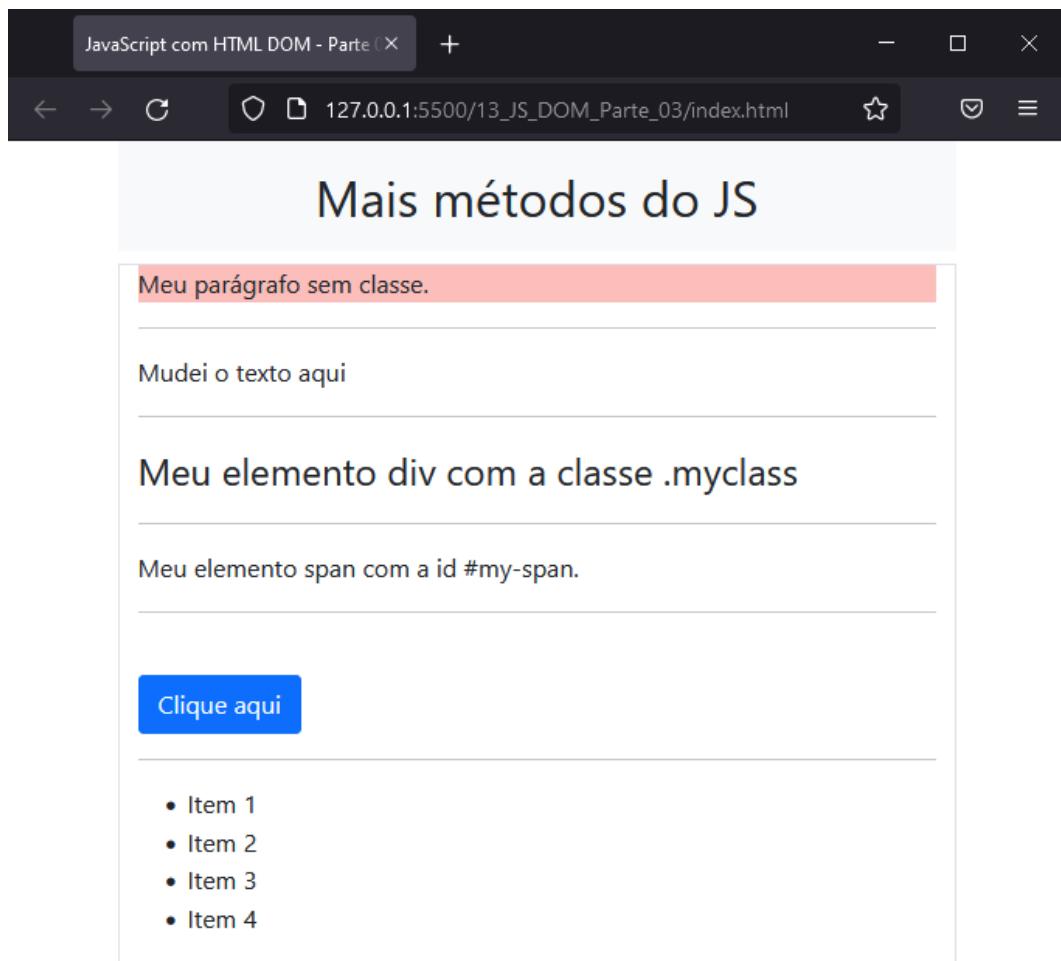
- A busca do elemento com o método: **querySelector('p.exemplo')**
- A modificação do conteúdo com a propriedade: **innerHTML**.

O que foi necessário apenas é colocar os comandos na ordem certa (primeiro o método de busca e depois a propriedade) com os devidos pontos finais no ligar correto.

Continuando a implementação do projeto, no arquivo **main.js**, insira o seguinte código.

```
// Busca somente a classe
let elemento = (document.querySelector('.myclass')).style.fontSize = 'x-large');
```

Veja o resultado apresentado no navegador.



Note que, o tamanho do texto no elemento <div> ficou maior, pois ele é o elemento que contém a classe **.myclass**. Desse modo, a instrução **busca e modifica o tamanho do texto do elemento**.

Eventos

No JavaScript, **eventos são ações ou ocorrências** que acontecem na página web que estamos implementando. Esses eventos geralmente **invocam funções que iram executar as operações desejadas**. Alguns exemplos de eventos são: o usuário **clica em um botão na página**, você

pode querer responder a esta ação mostrando na tela uma caixa de informações ou alterando o estilo de um elemento da página ou submetendo as informações de um formulário.



Vamos praticar

O nosso elemento <button> no código HTML, possui o evento **onclick** e esse evento chama a função **Clique()**, que será implementada com o JavaScript.

Continuando a implementação do projeto, no arquivo **main.js**, insira o seguinte código.

```
// Função chamada pelo evento onclick
const Clique = () => {
    document.querySelector('#my-span').style.backgroundColor = '#CEFBBA';
};
```

Você pode ver o resultado do evento no navegador ao **clicar no botão**.

Mais métodos do JS

Meu parágrafo sem classe.

Mudei o texto aqui

Meu elemento div com a classe .myclass

Meu elemento span com a id #my-span.

Clique aqui

- Item 1
- Item 2
- Item 3
- Item 4

Note que, ao clicarmos no botão, a função **Clique()** é chamada e então acontece a busca pelo elemento com a **ID** igual a **#my-span** (`querySelector('#my-span')`). Quando o elemento é encontrado a cor de fundo é alterada (`style.backgroundColor = '#CEFBBA'`).

Método querySelectorAll

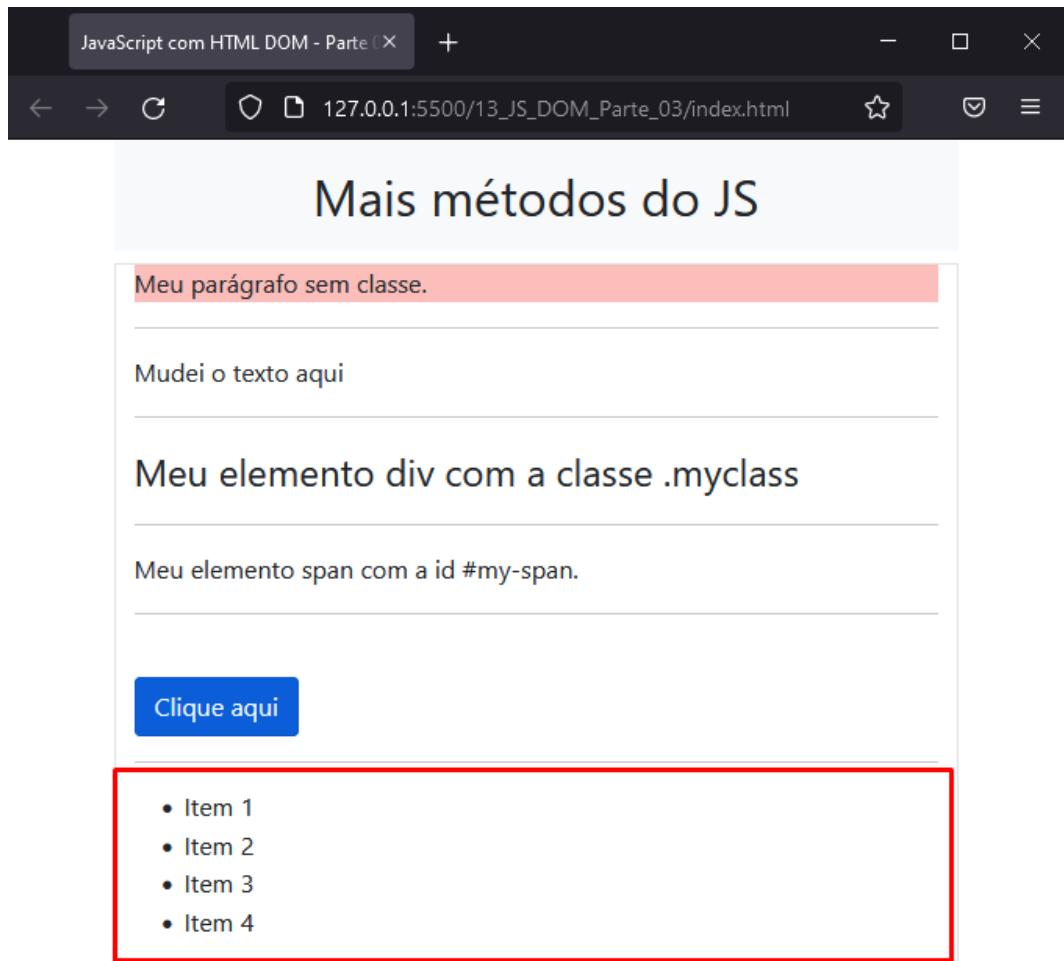
O método **querySelectorAll** retorna **todos os elementos presentes no documento que possuam o seletor desejado**, ou seja, esse método retorna uma lista dos elementos presentes no documento que coincidam com o seletor ou grupo de seletores especificados. A sintaxe do método é:

```
listaElementos = document.querySelectorAll(seletores);
```

Onde:

- **listaElementos**: é o objeto do JavaScript para armazenar os elementos encontrados.
- **seletores**: é uma string com um ou mais seletores desejados. Para mais de um seletor, deve-se separá-los por vírgula.

Nessa parte, vamos trabalhar com a lista que está abaixo do botão na página web.



The screenshot shows a browser window titled "JavaScript com HTML DOM - Parte 03". The address bar shows the URL "127.0.0.1:5500/13_JS_DOM_Parte_03/index.html". The main content area displays the following text:

Mais métodos do JS

Meu parágrafo sem classe.

Mudei o texto aqui

Meu elemento div com a classe .myclass

Meu elemento span com a id #my-span.

Clique aqui

A list of items is displayed in a red-bordered box:

- Item 1
- Item 2
- Item 3
- Item 4



Vamos praticar

Continuando a implementação do projeto, no arquivo **main.js**, insira o seguinte código.

```
// Método querySelectorAll
let listaLi = document.querySelectorAll('li');
for (i = 0; i < listaLi.length; i++) {
    listaLi[i].style.border = '1px solid #0000FF';
    listaLi[i].style.padding = '3px';
}
console.log(listaLi);
```

Veja o resultado apresentado no navegador.

JavaScript com HTML DOM - Parte 03

127.0.0.1:5500/13_JS_DOM_Parte_03/index.html

Mais métodos do JS

Meu parágrafo sem classe.

Mudei o texto aqui

Meu elemento div com a classe .myclass

Meu elemento span com a id #my-span.

Clique aqui

- Item 1
- Item 2
- Item 3
- Item 4

No nosso exemplo, o método **querySelectorAll** seleciona todos os elementos **** do documento. Se você colocar o comando **console.log(listaLi)**, poderá ver a **NodeList** com os quatro elementos **** armazenados no objeto **listaLi** impressos no console:

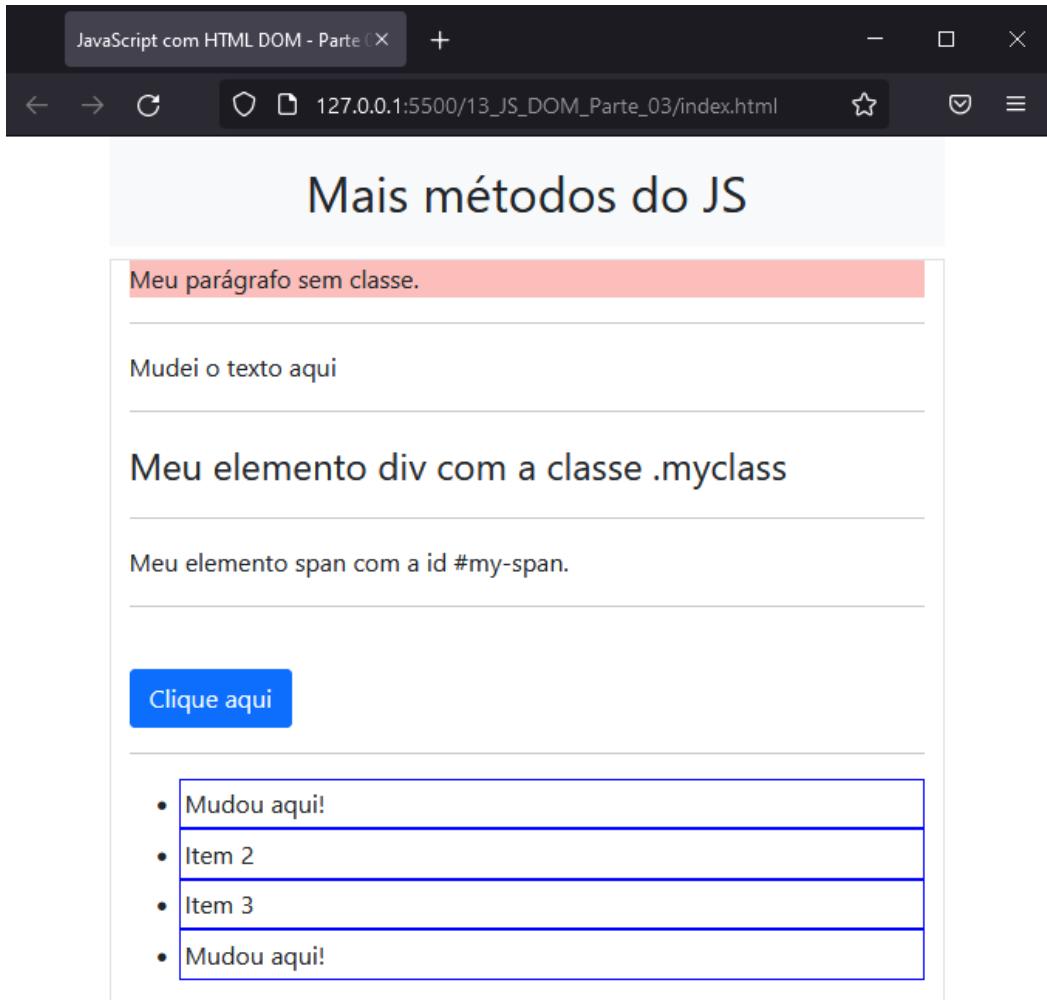
```
▼ NodeList(4) [ li#my-li.item , main.js:23:9
  li.item.teste , li.item.teste ,
  li#my-li.item ]
  ▶ 0: <li id="my-li" class="item"
    style="border: 1px solid rgb(0, 0, 255);
    padding: 3px;">
    ▶ 1: <li class="item teste"
      style="border: 1px solid rgb(0, 0, 255);
      padding: 3px;">
    ▶ 2: <li class="item teste"
      style="border: 1px solid rgb(0, 0, 255);
      padding: 3px;">
    ▶ 3: <li id="my-li" class="item"
      style="border: 1px solid rgb(0, 0, 255);
      padding: 3px;">
      length: 4
  ▶ <prototype>: NodeListPrototype { item:
    item(), keys: keys(), values: values(), ...
  }
```

O laço de **repetição é necessário** para que possamos **percorrer o array** e modificar a **borda** e **o espaçamento**.

Continuando a implementação do projeto, no arquivo **main.js**, insira o seguinte código.

```
// Busca por id
let myLi = document.querySelectorAll('#my-li');
myLi.forEach((li) => (li.innerHTML = 'Mudou aqui!'));
```

Veja o resultado apresentado no navegador:



JavaScript com HTML DOM - Parte 03

127.0.0.1:5500/13_JS_DOM_Parte_03/index.html

Mais métodos do JS

Meu parágrafo sem classe.

Mudei o texto aqui

Meu elemento div com a classe .myclass

Meu elemento span com a id #my-span.

Clique aqui

- Mudou aqui!
- Item 2
- Item 3
- Mudou aqui!

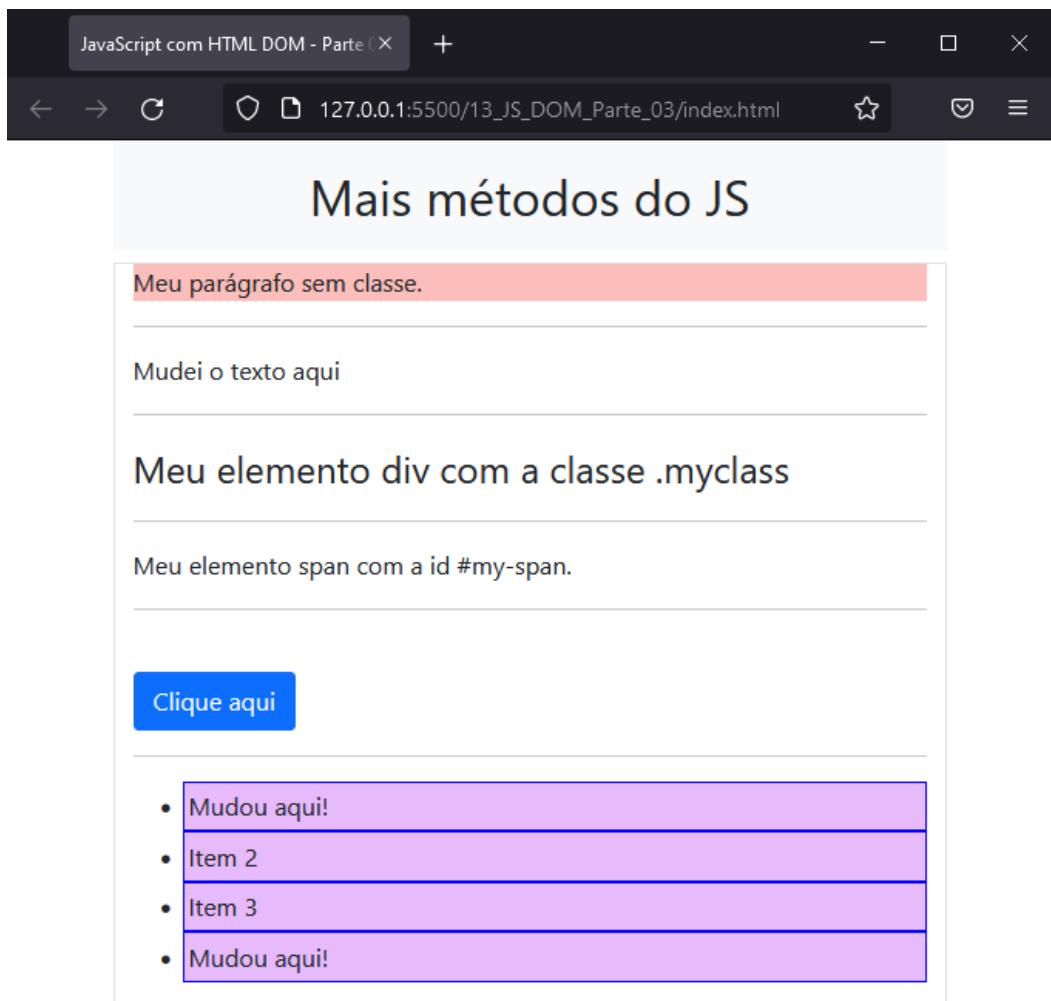
Observe que, o método **querySelectorAll** buscar os elementos com a **id** igual a **#my-li** e encontra o **primeiro** e o **último elemento da lista**. Então, alteramos o conteúdo do texto desses elementos.

Note que, ao invés de usarmos um laço de repetição, utilizamos o método de alto nível **forEach()** para **percorrer o array** e **alterar o conteúdo** através da propriedade **innerHTML**.

Continuando a implementação do projeto, no arquivo **main.js**, insira o seguinte código.

```
// Busca por classe
let myItem = document.querySelectorAll('.item');
myItem.forEach((item) => (item.style.backgroundColor = '#E7BAFB'));
```

Veja o resultado apresentado no navegador.



JavaScript com HTML DOM - Parte 03

Mais métodos do JS

Meu parágrafo sem classe.

Mudei o texto aqui

Meu elemento div com a classe .myclass

Meu elemento span com a id #my-span.

Clique aqui

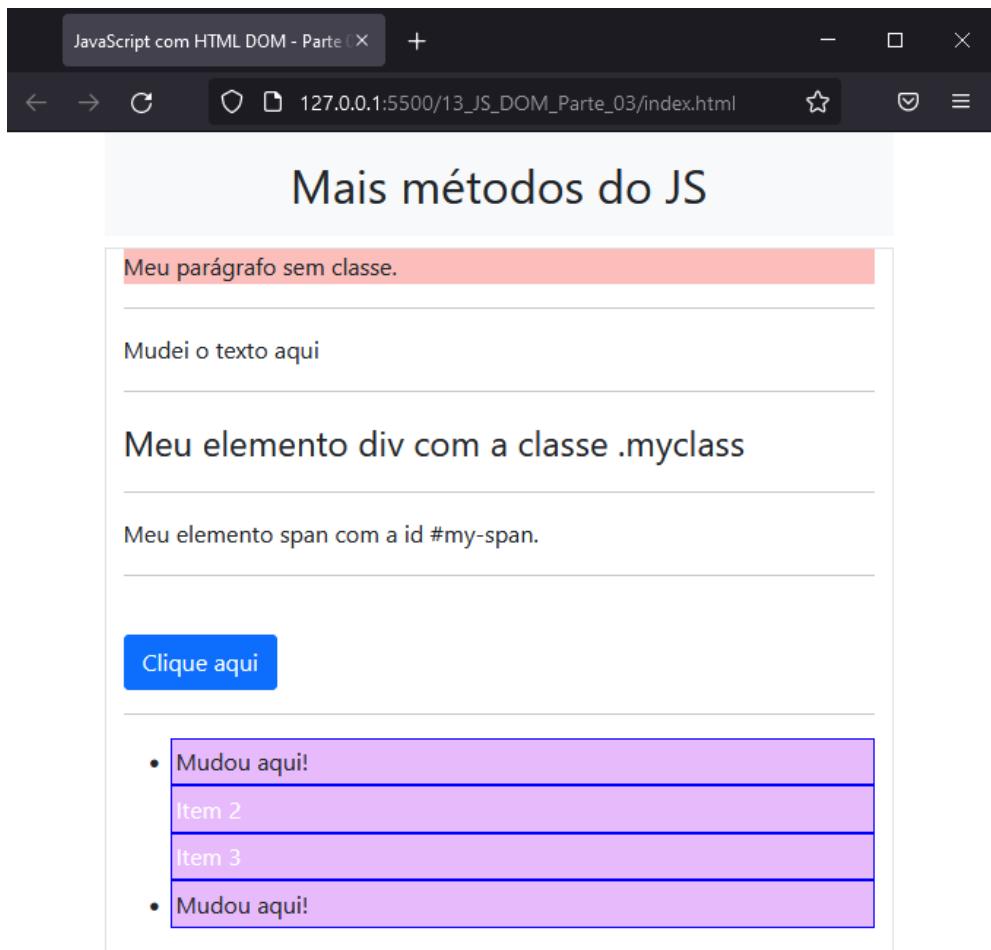
- Mudou aqui!
- Item 2
- Item 3
- Mudou aqui!

Agora, o nosso exemplo mostra o método **querySelectorAll** buscando os elementos com a **classe igual** a **.item** e encontra **todos os elementos da lista**, pois **todos tem essa classe**. Então, podemos alterar a cor de fundo desses elementos.

Continuando a implementação do projeto, no arquivo **main.js**, insira o seguinte código.

```
// Busca por elemento e classe
let myTeste = document.querySelectorAll('li.item');
myTeste.forEach((elemento) => (elemento.style.backgroundColor = '#FFF'));
```

Veja o resultado apresentado no navegador



Mais métodos do JS

Meu parágrafo sem classe.

Mudei o texto aqui

Meu elemento div com a classe .myclass

Meu elemento span com a id #my-span.

Clique aqui

- Mudou aqui!
- Item 2
- Item 3
- Mudou aqui!

Agora, o nosso exemplo mostra o método **querySelectorAll** buscando os elementos , que possuem com a **classe igual** a **.teste** e encontra o segundo e o terceiro elementos da lista.

Então, podemos alterar a **cor de fonte desses elementos**.

Para aprender mais

Procure sempre aprender e estudar mais. Seguem alguns links para você estudar e aprender mais:

querySelector():

<https://developer.mozilla.org/pt-BR/docs/Web/API/Element/querySelector>

https://www.w3schools.com/Jsref/met_document_queryselector.asp

Eventos:

https://developer.mozilla.org/pt-BR/docs/Learn/JavaScript/Building_blocks/Events

https://www.w3schools.com/jsref/dom_obj_event.asp

querySelectorAll():

<https://developer.mozilla.org/pt-BR/docs/Web/API/Document/querySelectorAll>

https://www.w3schools.com/jsref/met_document_queryselectorall.asp



JavaScript HTML DOM - Parte 04

Os objetivos desta aula são:

- Compreender o uso do DOM (Document Object Model);
- Aplicar código JavaScript para trabalhar com formulários;
- Conhecer os métodos addEventListener;
- Validações com RegEx.



Vamos praticar

Abra o **VS Code** e escolha um **diretório de trabalho para o seu projeto**.

Crie um diretório para seu projeto com o nome representativo, por exemplo, **JS_DOM_Parte_04**

Crie um arquivo dentro do diretório do projeto com o nome **index.html**.

Insira o seguinte código no seu arquivo **index.html**.

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

<head>
    <meta charset="UTF-8" />
    <link rel="shortcut icon" href="#" />
    <link rel="stylesheet" href="./css/style.css" />
    <title>JavaScript com HTML DOM - Parte 04</title>
</head>

<body>
    <header>
        <h1>Agendamento de Clientes</h1>
    </header>
    <section class="container">
        <form id="my-form">
            <h1>Adicionar Usuário</h1>
            <div class="msg"></div>
            <div>
                <label for="name">Nome:</label>
                <input type="text" id="name" required />
            </div>
            <div>
                <label for="email">E-mail:</label>
                <input type="text" id="email" required />
            </div>
            <div class="msg_email"></div>
            <div>
                <label for="horario">Horário:</label>
                <select name="hours" id="horario">
                    <option value="-->">-->
                    <option value="9:00h">9:00</option>
                    <option value="10:00h">10:00</option>
                    <option value="11:00h">11:00</option>
                    <option value="12:00h">12:00</option>
                </select>
            </div>
            <input class="btn" type="submit" value="Enviar" />
        </form>
    </section>
</body>
```

```
</form>
<ul id="users"></ul>
</section>
<script src=".js/main.js"></script>
</body>

</html>
```

Esse código mostra a marcação `<link>` com o arquivo **CSS** e no atributo **href** mostra que ele está em uma pasta chamada **css**. Portanto, vamos criar o **diretório css** e o **arquivo style.css** dentro desse projeto com o seguinte código.

```
* {
    margin: 0;
    padding: 0;
    box-sizing: border-box;
}

body {
    font-family: Arial, Helvetica, sans-serif;
    line-height: 1.6;
}

ul {
    list-style: none;
}

ul li {
    padding: 5px;
    background: #f4f4f4;
    margin: 5px 0;
}

header {
    background: #f4f4f4;
    padding: 1rem;
    text-align: center;
}

.container {
    margin: auto;
    width: 500px;
    overflow: auto;
    padding: 3rem 2rem;
}

#my-form {
    padding: 2rem;
    background: #f4f4f4;
```

```
}

#my-form label {
    display: block;
}

#my-form input[type='text'] {
    width: 100%;
    padding: 8px;
    margin-bottom: 10px;
    border-radius: 5px;
    border: 1px solid #ccc;
}

#my-form select {
    width: 20%;
    padding: 8px;
    margin-bottom: 10px;
    border-radius: 5px;
    border: 1px solid #ccc;
}

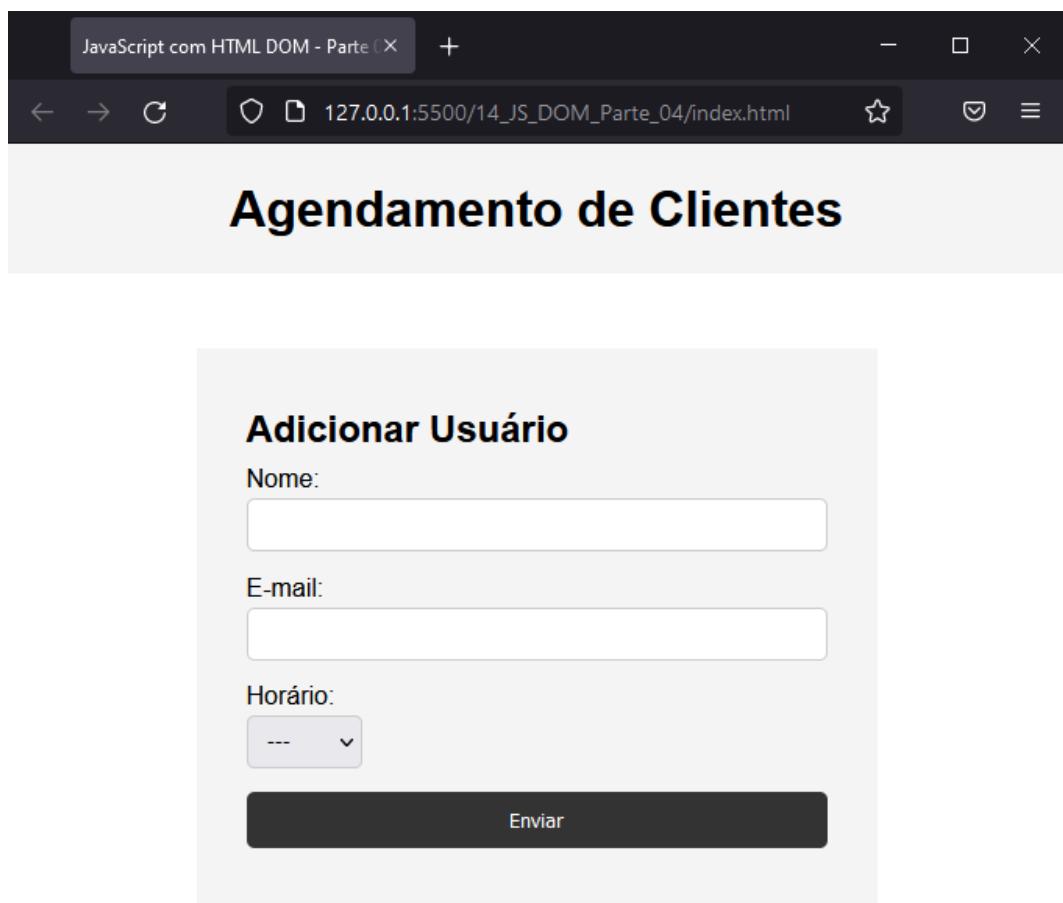
.btn {
    display: block;
    width: 100%;
    padding: 10px 15px;
    border: 0;
    background: #333;
    color: #fff;
    border-radius: 5px;
    margin: 5px 0;
}

.btn:hover {
    background: #444;
}

.bg-dark {
    background: #333;
    color: #fff;
}

.error {
    font-size: 0.8rem;
    background: #fff;
    color: red;
}
```

Abra o arquivo index.html, clique no botão **Go Live** da extensão **Live Server** e abra as Ferramentas de desenvolvimento do navegador web (Atalho **F12**).



Observe que **mudamos a organização dos arquivos no projeto**, agora o arquivo **main.js** está em uma pasta chamada **js**, o arquivo **style.css** está em uma pasta chamada **css**, enquanto o arquivo **index.html** está no **diretório raiz do projeto**. Isso é importante para você ser uma pessoa organizada com os seus projetos. Além disso, o caminho do script agora está com apontando para o novo diretório (`<script src=".js/main.js"></script>`).

Vamos deixar o arquivo **main.js vazio** e à medida que vamos aprendendo coisas novas vamos completando a implementação do código **JavaScript**.

Colocando código em JS para trabalhar com o formulário

A aplicação que desejamos fazer é a seguinte: Esse é um formulário de **agendamento de clientes**. O agendamento será mostrado **apenas na interface do usuário**, não vamos **salvar em nenhum servidor de backend**, portanto se a sua **recarregar** os **dados serão perdidos**. Então, a nossa agenda de clientes funcionará da seguinte forma:

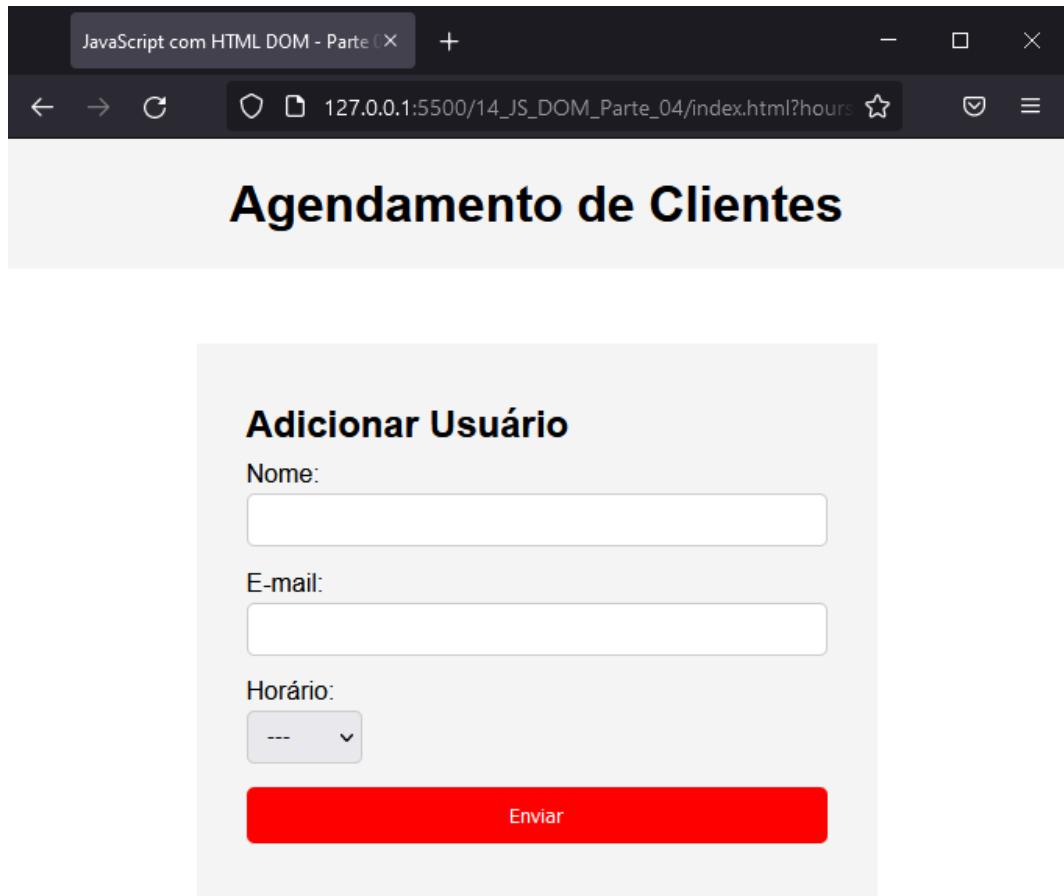
- Você irá preencher o **Nome** e o **E-mail**, observe que o campo está com o atributo do HTML **required**.
- Assim que os campos estiverem corretamente preenchidos e você clicar no botão submete, as informações serão exibidas na lista disponível na linha 27 do código HTML.

Vamos implementar nosso código em JS para manipular as informações e os elementos do formulário. Siga os passos para completar o código do JavaScript

Primeiro, vamos brincar de alterar a cor do botão para enviar os dados no formulário. No arquivo **main.js**, insira o seguinte código.

```
// Método querySelector
const btn = document.querySelector('.btn');
btn.style.background = 'red';
```

Veja o resultado apresentado no navegador.



The screenshot shows a browser window with the title "JavaScript com HTML DOM - Parte 04". The address bar displays the URL "127.0.0.1:5500/14_JS_DOM_Parte_04/index.html?hours". The main content area has a large title "Agendamento de Clientes". Below it is a form titled "Adicionar Usuário". The form contains three input fields: "Nome" (with a placeholder), "E-mail" (with a placeholder), and "Horário" (a dropdown menu with options "..."). A large red button at the bottom is labeled "Enviar".

Observe que o método **querySelector** busca pelo elemento com a **classe .btn**, que é encontrada no botão do código HTML. Então, esse elemento é retornado e, assim, é possível alterarmos a **cor de fundo do elemento**.

Agora vamos buscar os elementos do formulário e salvá-los em variáveis para trabalharmos com eles.

```
// Busca pelos elementos do Formulário
const myForm = document.querySelector('#my-form');
const nameInput = document.querySelector('#name');
const emailInput = document.querySelector('#email');
const horario = document.querySelector('#horario');
const msg = document.querySelector('.msg');
const msg_email = document.querySelector('.msg_email');
const userList = document.querySelector('#users');
```

A primeira instrução busca pelo elemento com a **ID igual** a **#my-form** através do método **querySelector**, que retorna o elemento **<form>** do formulário.

A instrução seguinte, busca pelo elemento com a **ID igual** a **#name** também através do método **querySelector**, que retorna o **primeiro elemento <input>** dentro do formulário.

A instrução seguinte, busca pelo elemento com a **ID igual** a **#email** também através do método **querySelector**, que retorna o **segundo elemento <input>** dentro do formulário.

A instrução seguinte, busca pelo elemento com a **ID igual** a **#horario** também através do método **querySelector**, que retorna o elemento **<selector>** dentro do formulário.

A instrução seguinte, busca pelo elemento com a **classe igual** a **.msg** também através do método **querySelector**, que retorna o elemento **<div>** logo abaixo do **<input> name**. Esse elemento vai ser usado para exibir uma mensagem caso o valor do campo name esteja vazio.

A instrução seguinte, busca pelo elemento com a **classe igual** a **.msg_email** também através do método **querySelector**, que retorna o elemento **<div>** logo abaixo do **<input> email**. Esse elemento vai ser usado para exibir uma mensagem caso o usuário insira um email inválido.

Por último, a instrução busca pelo elemento com a **ID igual** a **#users** também através do método **querySelector**, que retorna o elemento **** fora do formulário e dentro do documento HTML. Esse elemento será usado para inserirmos as novas informações do agendamento

Método addEventListener

O método **addEventListener** “vigia” um **evento de um elemento específico**. Esse evento pode ser o clique de um botão, a submissão de um formulário, etc. No nosso exemplo, vamos utilizar esse método para “vigar” o evento de **submissão do formulário**.



Vamos praticar

Continuando a implementação do projeto, no arquivo **main.js**, insira o seguinte código.

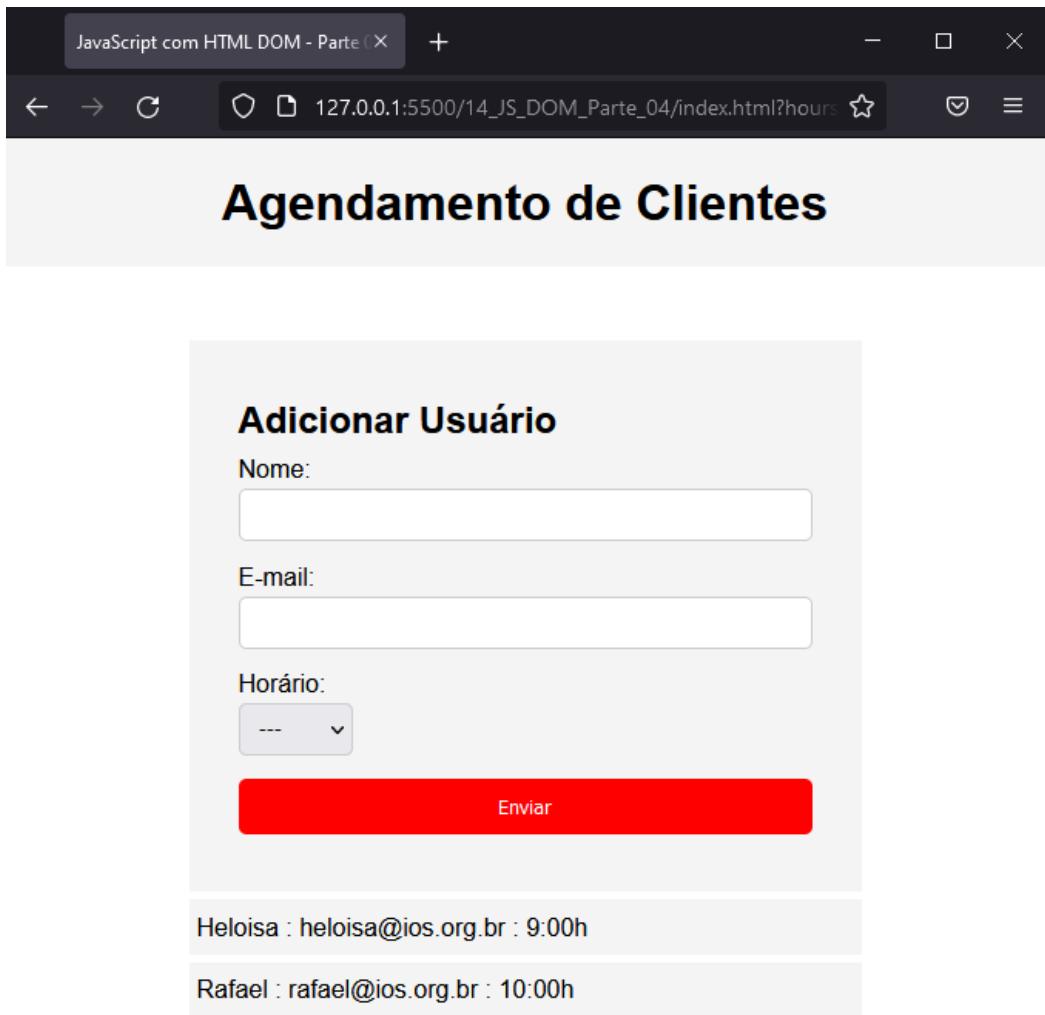
```
// Método addEventListener
myForm.addEventListener('submit', onSubmit);
```

Nesse código mostrado acima, o método está registrado para o evento **submit** do elemento **myForm**, que é o nosso formulário. Observe que esse elemento chama a função **onSubmit**, que vamos implementar a seguir.

No arquivo **main.js**, insira o seguinte código.

```
// Função onSubmit
function onSubmit(e) {
    e.preventDefault();
    // console.log(nameInput.value);
    if (nameInput.value === '' || emailInput.value === '') {
        // alert('Por favor, preencha os dados.');
        msg.classList.add('error');
        msg.innerHTML = 'Por favor, preencha os dados.';
        setTimeout(() => msg.remove(), 3000);
    } else {
        // console.log('sucess');
        const li = document.createElement('li');
        li.appendChild(
            document.createTextNode(
                `${nameInput.value} : ${emailInput.value} : ${horario.value}`
            )
        );
        userList.appendChild(li);
        // Limpa o formulário
        nameInput.value = '';
        emailInput.value = '';
        horario.getElementsByTagName('option')[0].selected = 'selected';
        nameInput.focus(); //Coloca o foco no elmento
    }
}
```

Veja o resultado apresentado no navegador.



JavaScript com HTML DOM - Parte 04

Agendamento de Clientes

Adicionar Usuário

Nome:

E-mail:

Horário:

Heloisa : heloisa@ios.org.br : 9:00h

Rafael : rafael@ios.org.br : 10:00h

A instrução **e.preventDefault()** é para evitar que a página seja **recarregada por causa da ação de submeter o formulário**, pois essa é uma ação padrão do HTML quando estamos trabalhando com formulários.

Testamos se os campos do **nome** e do **e-mail estão vazios**, caso **um deles esteja vazio** a mensagem no **<div class="msg"></div>** irá aparecer por 3 segundos (**setTimeout(() => msg.remove(), 3000)**).

A propriedade **classList.add** adiciona uma classe no elemento específico, que no caso é o elemento **<div>** com a classe **.msg**.

Caso os campos estejam preenchidos o **else** será executado. A instrução cria um elemento **** para inserir um **item na lista**.

O método **appendChild** adiciona um **nó ao final da lista de elementos filhos** de um **elemento pai especificado**. No nosso caso, esse método vai inserido itens ao final da lista.

Por fim, as instruções limpam o conteúdo do formulário e coloca o cursor no campo do nome (**nameInput.focus()**)

Validando e-mail com RegEx

O **Regex (regular expression)** é uma **sequência de caracteres que especifica um padrão de busca**. Normalmente, esses padrões são usados por algoritmos de busca de strings para realizar operações de "localizar" ou de "localizar e substituir" e também para validação de entrada de dados. Você pode testar qualquer tipo de expressão regular no site <https://regex101.com/>.



Vamos praticar

Continuando a implementação do projeto, no arquivo **main.js**, insira o seguinte código.

```
// Validando e-mail
emailInput.addEventListener('change', (e) => {
  let padrao = new RegExp(/.*@.*\..*/i);
  if (!padrao.test(emailInput.value)) {
    // alert('Por favor, insira um e-mail válido.');
    msg_email.classList.add('error');
    msg_email.innerHTML = 'Por favor, insira um e-mail válido.';
    setTimeout(() => msg.remove(), 3000);
  }
});
```

Veja o resultado apresentado no navegador:



Adicionar Usuário

Nome:

E-mail:

Por favor, insira um e-mail válido.

Horário:

A instrução vincula a execução de uma **Arrow Function** ao evento **change** do **campo de e-mail**. Quando você **terminar de digitar e sair do campo**, a função de verificação do e-mail é **executada**.

A expressão testada é ***/.*@.*\./i***, que verifica se tem um símbolo de arroba (**@**) e um **ponto final no e-mail**. Não é a melhor das validações de e-mail, mas funciona para o nosso caso.

Caso o e-mail esteja inválido de acordo com o teste lógico ***!padrao.test(emailInput.value)***, a mensagem de erro irá aparecer.

Para aprender mais

Procure sempre aprender e estudar mais. Seguem alguns links para você estudar e aprender mais:

addEventListener:

<https://developer.mozilla.org/pt-BR/docs/Web/API/EventTarget/addEventListener>

https://www.w3schools.com/jsref/met_element_addeventlistener.asp

appendChild:

<https://developer.mozilla.org/pt-BR/docs/Web/API/Node/appendChild>

https://www.w3schools.com/jsref/met_node_appendchild.asp

Regex:

<https://regex101.com/>

https://en.wikipedia.org/wiki/Regular_expression

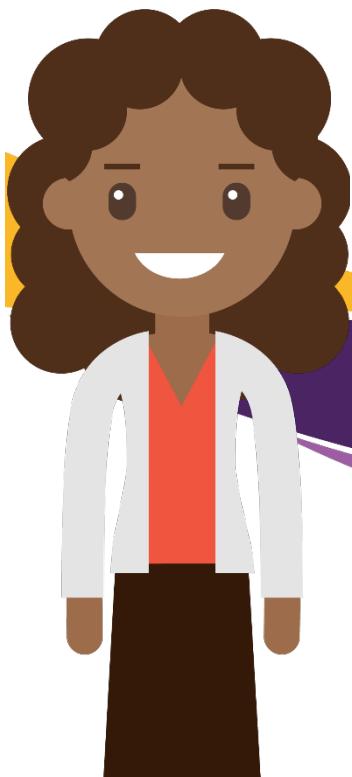
Javascript e Formulários:

<https://www.youtube.com/watch?v=OR8ySydmqLQ>

<https://www.youtube.com/watch?v=evS7nVlbsw4>

<https://www.youtube.com/watch?v=qMxhzUh2gUc>

<https://www.youtube.com/watch?v=NoZOqtK6ecl>



JavaScript HTML DOM - Parte 05

Os objetivos desta aula são:

- Compreender o uso do DOM (Document Object Model);
- Aplicar código JavaScript para trabalhar com formulários;



Vamos praticar

Abra o **VS Code** e escolha um **diretório de trabalho para o seu projeto**.

Crie um diretório para seu projeto com o nome representativo, por exemplo, **JS_DOM_Parte_05**

Crie um arquivo dentro do diretório do projeto com o nome **index.html**.

Insira o seguinte código no seu arquivo **index.html**.

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

<head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <link rel="shortcut icon" href="#" />
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.0.1/dist/css/bootstrap.min.css" rel="stylesheet"
        integrity="sha384-+0n0xVW2eSR5OomGNYDnhzAbDsOXxcvSN1TPprVMTNdbiYZCxYb00l7+AMvyTG2x"
        crossorigin="anonymous" />
    <title>DOM - Lista de Itens</title>
</head>

<body>
    <header id="main-header" class="bg-success text-white p-4 mb-3">
        <div class="container">
            <div class="row">
                <div class="col-md-6">
                    <h1 id="header-title">Lista de Itens</h1>
                </div>
                <div class="col-md-6 align-self-center">
                    <input type="text" class="form-control" id="filter"
                        placeholder="Busca item..." />
                </div>
            </div>
        </div>
    </header>
    <div class="container">
        <div id="main" class="card card-body">
            <h2 class="title">Adicionar Item</h2>
            <form id="addForm" class="form-control mb-3">
                <div class="row">
                    <div class="col-4">
                        <input type="text" class="form-control mr-2" id="item" />
                    </div>
                    <div class="col-2">
```

```

        <input type="submit" class="btn btn-dark" value="Add" />
    </div>
</div>
</form>
<h2 class="title">Minha lista</h2>
<ul id="items" class="list-group">
    <li class="list-group-item">
        Arroz
        <button class="btn btn-danger btn-sm float-end delete">
            X
        </button>
    </li>
    <li class="list-group-item">
        Feijão
        <button class="btn btn-danger btn-sm float-end delete">
            X
        </button>
    </li>
</ul>
</div>
</div>

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

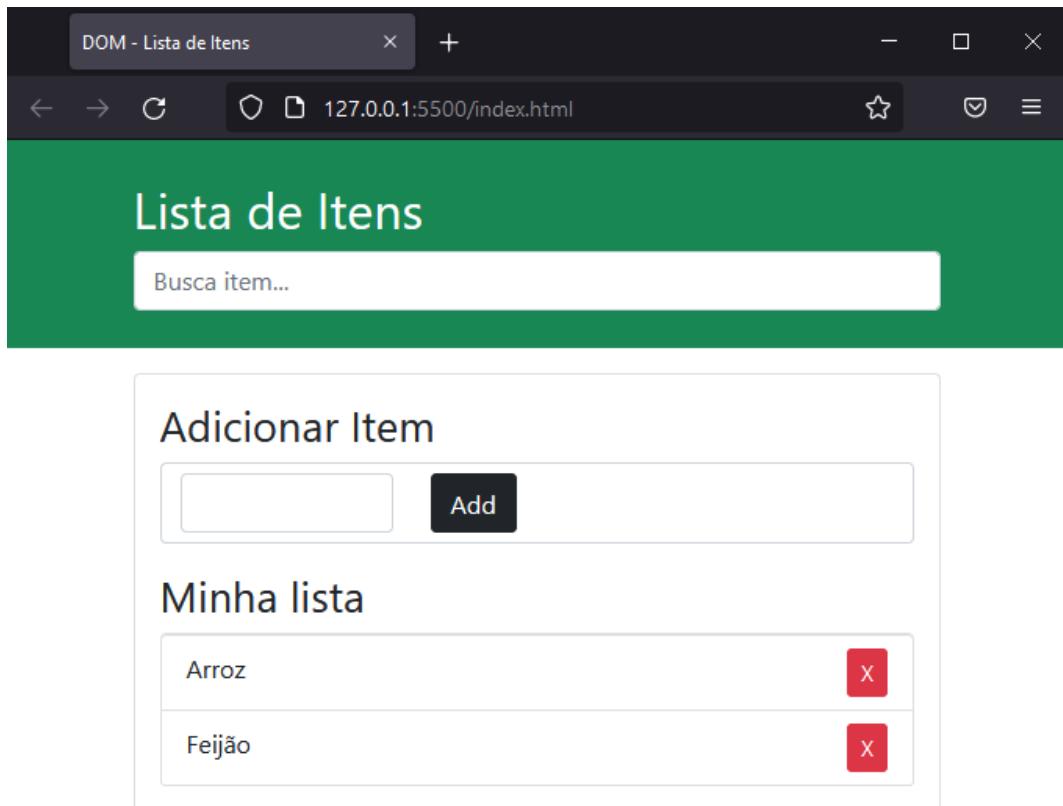
</html>
```

Nesse código, estamos usando na marcação `<link>` o **CDN do Bootstrap**. Isso vai permitir utilizarmos as **classes predefinidas do Bootstrap** para **estilizar a nossa página**.

Esse código mostra também a marcação `<script>` sem nenhum código **JavaScript** entre a **abertura** e o **fechamento** da tag, apenas o atributo `src` com o valor **main.js**. Isso significa que o código **JavaScript** está em um **arquivo externo**. Portanto, temos que criar esse novo arquivo **main.js** dentro do diretório do projeto.

Vamos deixar o arquivo **main.js vazio** e à medida que vamos aprendendo coisas novas iremos **completar a implementação do código JavaScript**.

Abra o arquivo `index.html`, clique no botão  **Go Live** da extensão **Live Server**. A página inicial mostrada será:



Colocando código em JS para manipular a nossa lista de itens

A aplicação que desejamos fazer é a seguinte: uma **lista de itens** como se fosse o **Keep do Google**, onde vamos poder fazer inserir os produtos que desejamos comprar. Então, a nossa lista de compras funcionará da seguinte forma:

Você irá preencher o nome do item e clicar no botão **Add** para **adicionar-lo na lista**.

Você poderá usar o **botão X** em vermelho para **tirar o item da lista**.

E por fim, você poderá **procurar um item** na sua lista com o campo **Busca item**

Vamos implementar nosso código em **JS** para **manipular as informações** e os **elementos do formulário**. Siga os passos para completar o código do JavaScript.

Primeiro, vamos **buscar os elementos** da página pela **ID**. No arquivo **main.js**, insira o seguinte código.

```
let form = document.getElementById('addForm');
let itemList = document.getElementById('items');
let filter = document.getElementById('filter');
```

Nesse caso, estamos criando um objeto chamado **form** que por meio do método **getElementById** nos permite **acessar os elementos do nosso formulário** para adicionar os itens.

Adicionar Item



Um formulário com uma barra de pesquisa vazia e um botão "Add" em destaque.

O objeto **itemList** que por meio do método **getElementById** nos permite acessar a lista de itens.

Minha lista

Arroz	X
Feijão	X

E por fim, o objeto **filter** que nos permitirá **acessar o conteúdo** do <input> para buscar um elemento na lista.



Agora vamos vincular um evento a cada um desse objetos criados anteriormente.

```
// Form submit event
form.addEventListener('submit', addItem);
// Delete event
itemList.addEventListener('click', removeItem);
// Filter event
filter.addEventListener('keyup', buscarItems);
```

A primeira instrução **vincula** o nosso objeto **form** à função **addItem**, que é disparado sempre que o botão **Add** é clicado. Lembrado que o nosso botão **Add** é responsável por **submeter o nosso formulário** (atributo **type** do botão tem o valor **submit**).

```
<input type="submit" class="btn btn-dark" value="Add" />
```

A segunda instrução **vincula** o nosso objeto **itemList** à função **removeltem**, que é disparado sempre que o botão **X** é **clicado**. Ou seja, quando você **clicar no botão o item será apagado da lista**.

A terceira instrução **vincula** o nosso objeto **filter** à função **buscarItems**, que é disparada sempre quando uma tecla do teclado é solta sempre que você começar a **digitar algo** no **<input> Busca item...**

Agora vamos começar a implementar nossas **funções** que estão **vinculadas aos objetos** e são **invocadas** quando um determinado **evento acontece**. No arquivo **main.js**, insira o seguinte código.

```
// Adiciona item
function addItem(e) {
  e.preventDefault();

  // Pega o valor do <input>
  let newItem = document.getElementById('item').value;

  // Cria novo elemento <li>
  let li = document.createElement('li');
  // Adiciona classe
  li.className = 'list-group-item';
  // Adiciona o texto no novo elemento com o valor armazenado no newItem
  li.appendChild(document.createTextNode(newItem));

  // Cria o elemento botão para deletar um item
  let deleteBtn = document.createElement('button');

  // Adiciona classes para o botão de deletar
  deleteBtn.className = 'btn btn-danger btn- float-end delete';

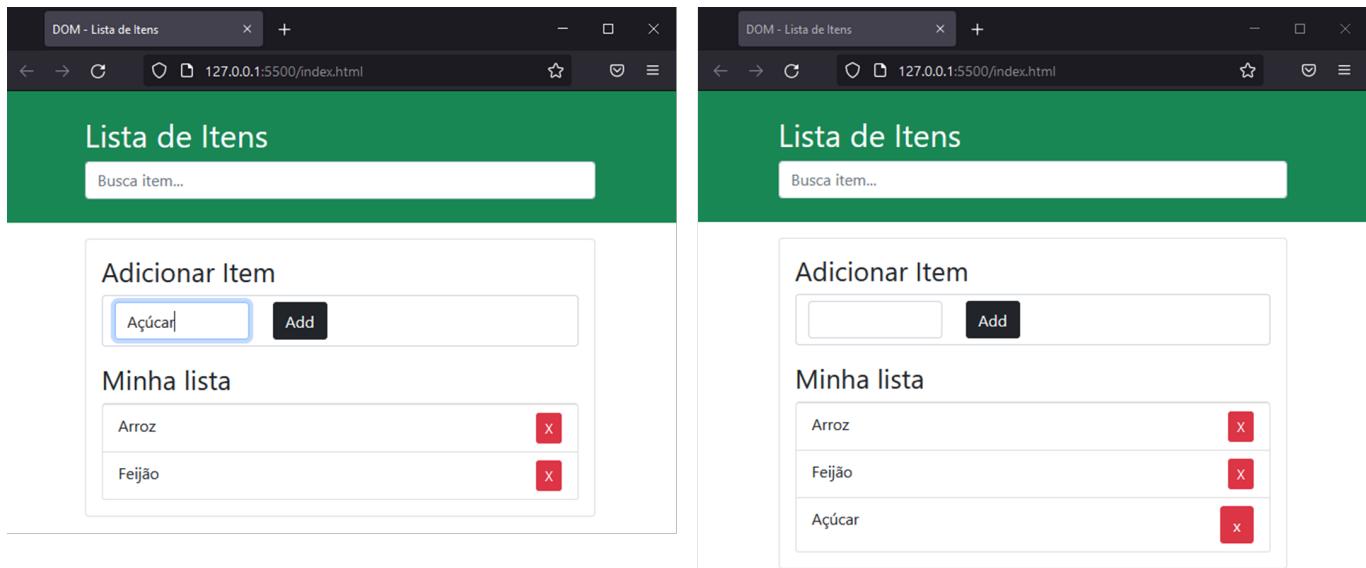
  // Acrescenta o texto no novo elemento
  deleteBtn.appendChild(document.createTextNode('x'));

  // Acrescenta o botão no elemento Li
  li.appendChild(deleteBtn);

  itemList.appendChild(li); // Adiciona o novo item na lista

  form.reset(); // Limpa o formulário
}
```

Com esse código podemos inserir um novo item na nossa lista de compras. Por exemplo, podemos digitar no **<input> Adicionar Item** o produto **Açucar** e clicar no botão **Add**.



The screenshots show a user interface for managing a shopping list. On the left, a text input field contains the text 'Açúcar' and a black 'Add' button is visible. On the right, the input field is empty, and the 'Add' button is now greyed out. Below the input field, there is a list titled 'Minha lista' containing three items: 'Arroz', 'Feijão', and 'Açúcar'. Each item has a small red square with a white 'X' icon to its right, which likely serves as a delete button.

Vamos entender o código

A instrução `e.preventDefault()` é para **evitar** que a página **seja recarregada por causa da ação de submeter o formulário**, pois essa é uma **ação padrão do HTML** quando estamos trabalhando com formulários.

Criamos uma variável **newItem** e **armazenamos o valor digitado** no campo `<input>` para **Adicionar Item**.

Em seguida, criamos um elemento **HTML **, e adicionamos a classe **list-group-item** a esse **novo elemento**, depois **adicionamos o texto armazenado** na variável **newItem** nesse **novo elemento**.

Criamos um **botão** e **adicionamos algumas classes** nesse **botão** e acrescentamos o texto do botão.

Adicionamos o **botão de deletar** no elemento ``.

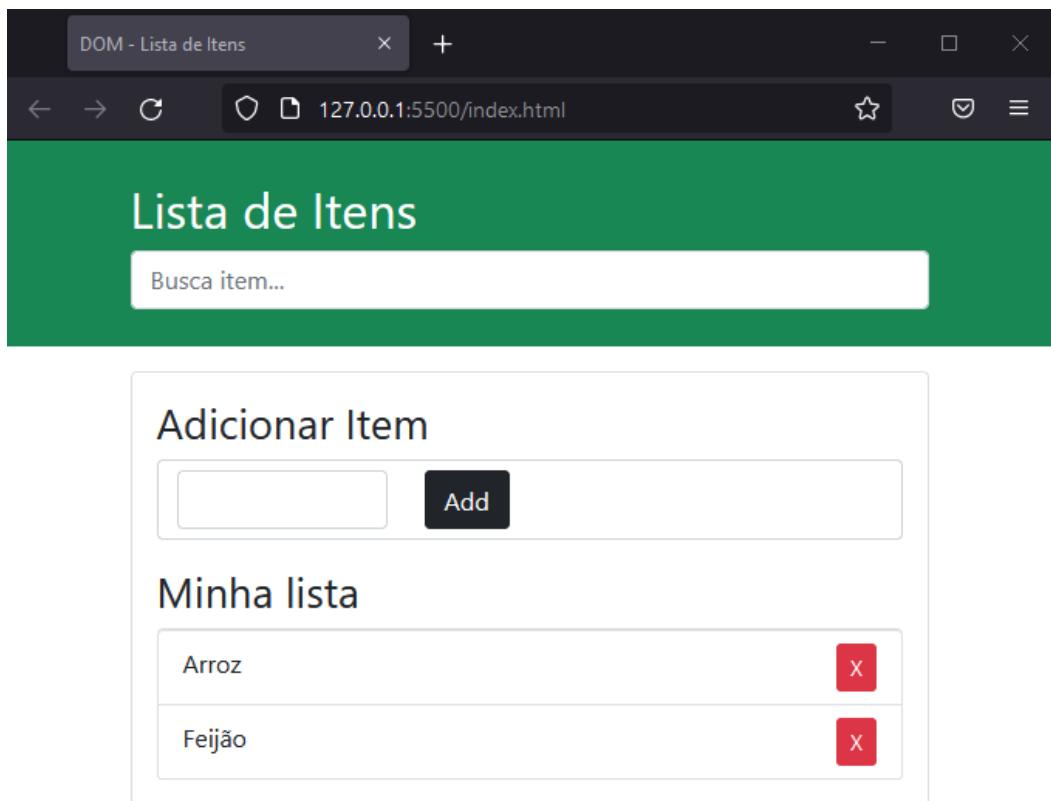
Adicionamos o **novo item** criado na **lista de itens**.

E por fim, **limpamos o formulário**.

Agora vamos implementar a **função removeItem**, no arquivo **main.js**, insira o seguinte código:

```
// Função Remove Item
function removeItem(e) {
    if (e.target.classList.contains('delete')) {
        let li = e.target.parentElement;
        itemList.removeChild(li);
    }
}
```

Você pode clicar em qualquer item e **removê-lo da lista**, por exemplo, o item **Açúcar**.



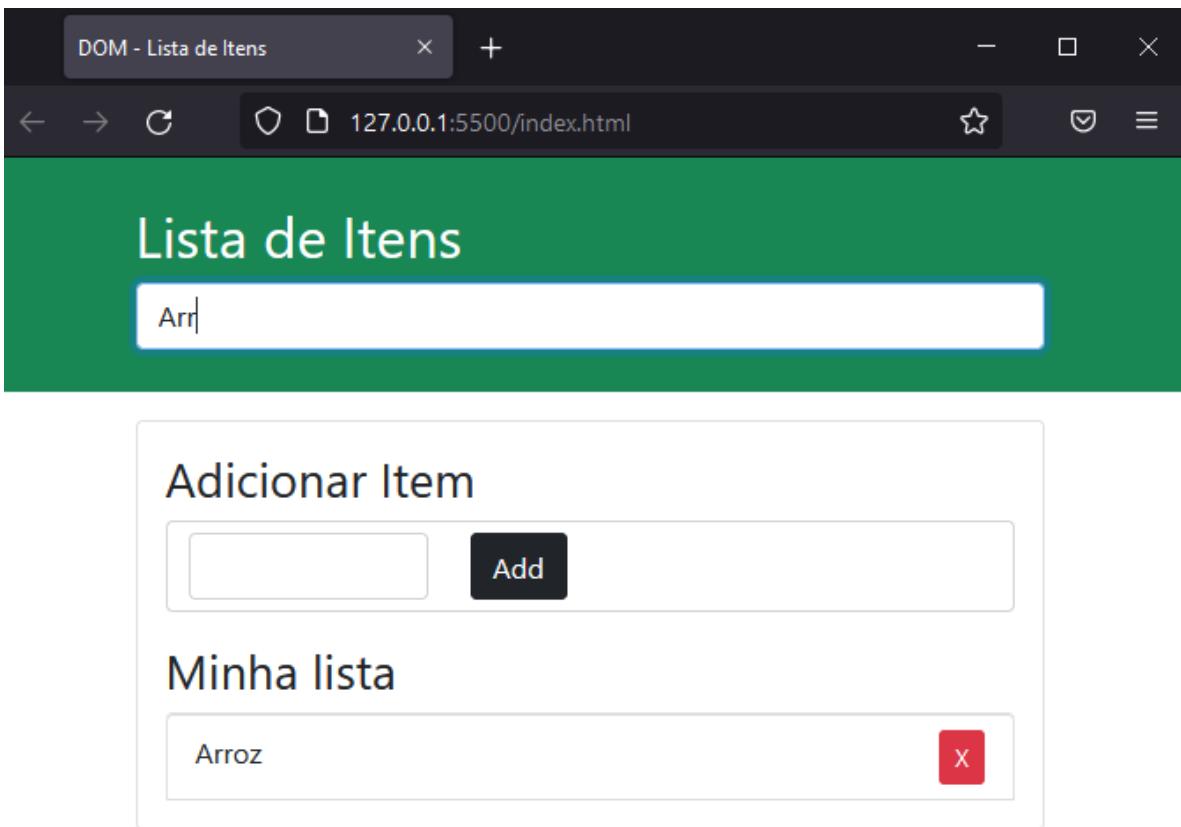
The screenshot shows a web application interface. At the top, a header bar displays 'DOM - Lista de Itens' and the URL '127.0.0.1:5500/index.html'. Below the header is a green navigation bar with the title 'Lista de Itens' and a search input field labeled 'Busca item...'. The main content area contains a form titled 'Adicionar Item' with two input fields and a 'Add' button. Below this is a section titled 'Minha lista' listing items 'Arroz' and 'Feijão', each accompanied by a red square button with a white 'X' symbol.

Verificamos se o **item clicado** contém a classe **delete**, se **verdadeiro** o conjunto de comandos **dentro do if será executado**. A instrução seguinte busca o elemento **** que teve seu **botão X clicado**. E por fim, removemos esse elemento da lista e então serão exibidos **apenas os itens restantes**.

Por fim, vamos implementar a função **buscarItem**, no arquivo **main.js**, insira o seguinte código.

```
// Função buscarItems
function buscarItems(e) {
    // Converte o texto digitado para minúsculo
    let text = e.target.value.toLowerCase();
    // Busca todos os itens
    let items = itemList.getElementsByTagName('li');
    // Converte os itens para array
    Array.from(items).forEach(function (item) {
        // Busca o primeiro item da lista
        let itemName = item.firstChild.textContent;
        // Busca o item na lista que começa com o mesmo texto digitado
        if (itemName.toLowerCase().indexOf(text) != -1) {
            item.style.display = 'block'; // exibe o item
        } else {
            item.style.display = 'none'; // oculta o item
        }
    });
}
```

Você buscar qualquer item na lista, por exemplo, Arroz.



Adicionar Item

Add

Minha lista

Arroz X

A primeira instrução converte o texto digitado no <input> **Busca item...** e o transforma para **minúsculo**.

A instrução busca **todos os elementos disponíveis na lista**.

Depois, convertemos os **itens para um array** e utiliza o método **forEach()** para percorrer os **itens na lista**.

A instrução seguinte, busca o **primeiro elemento da lista**.

Por fim, testamos se o **item atual da lista** começar o **valor digitado** no <input> **Busca item...**

Caso **verdadeiro**, altera o atributo **display** para **block** (**exibindo o item**), caso falso, **altera o atributo display para none** (**ocultando o item**).

Você pode colocar muitos itens na lista e testar buscar um ou mais itens que começam com valores iguais.



Projeto Book List

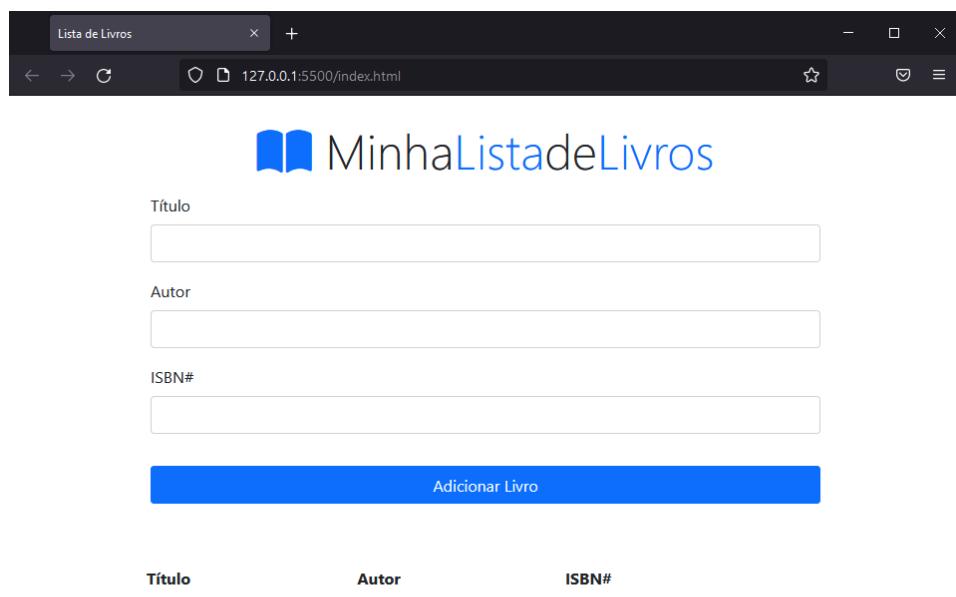
Os objetivos desta aula são:

- Exercitar os conhecimentos aprendidos por meio da criação de projetos;
- Compreender a aplicação da linguagem Javascript em diferentes contextos.

Sobre a lista de livros

Vamos criar uma **lista de livros** com as algumas informações sobre a obra, como por exemplo: **Título, Autor e ISBN**. Se você não conhece a sigla **ISBN**, ela significa **International Standard Book Number**, que é um sistema internacional de **identificação de livros e softwares** que utiliza números para classificá-los por **título, autor, país, editora e edição**.

Nossa lista de livros permitirá você **inserir os dados do livro**, clicar no botão **adicionar Livro** e o livro será na listado **abaixo do formulário de inserção de livros**. A nossa interface vai ser igual a mostrada a seguir:



The screenshot shows a browser window titled "Lista de Livros". The main content area has a title "MinhaLista de Livros" with a blue book icon. Below it is a form with three input fields: "Título", "Autor", and "ISBN#". Each field has a placeholder text and a corresponding empty input box. Below the form is a blue button labeled "Adicionar Livro". At the bottom of the page, there is a table header with columns "Título", "Autor", and "ISBN#".



Vamos praticar

Abra o **VS Code** e escolha um **diretório de trabalho para o seu projeto**.

Crie um diretório para seu projeto com o nome representativo, por exemplo, **JS_BookList**

Crie um arquivo dentro do diretório do projeto com o nome **index.html**.

Insira o seguinte código no seu arquivo **index.html**.

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

<head>
```

```
<meta charset="UTF-8" />
<link rel="shortcut icon" href="#" />

<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap.min.css"
rel="stylesheet"
    integrity="sha384-
1BmE4kWBq78iYhFldvKuhfTAU6auU8tT94WrHftjDbrCEXSU1oBoqyl2QvZ6jIW3"
crossorigin="anonymous">

<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-
awesome/6.0.0-beta3/css/all.min.css"
    integrity="sha512-
Fo3rlrZj/k7ujTnHg4CGR2D7kSs0v4LLanw2qksYuR1Ez0+tcEPQogQ0KaoGN26/zrn20ImR1DfuLWnOo7aBA==
"
    crossorigin="anonymous" referrerPolicy="no-referrer" />

<title>Lista de Livros</title>
</head>

<body>
    <div class="container mt-4">
        <h1 class="display-4 text-center">
            <i class="fas fa-book-open text-primary"></i> Minha<span class="text-
primary">Lista</span>de<span
                class="text-primary">Livros</span>
        </h1>
        <form id="book-form" class="form-control border-0">
            <div class="mb-3">
                <label class="form-label" for="title">Título</label>
                <input type="text" id="title" class="form-control">
            </div>
            <div class="mb-3">
                <label class="form-label" for="author">Autor</label>
                <input type="text" id="author" class="form-control">
            </div>
            <div class="mb-3">
                <label class="form-label" for="isbn">ISBN#</label>
                <input type="text" id="isbn" class="form-control">
            </div>
            <div class="d-grid">
                <input type="submit" value="Adicionar Livro" class="btn btn-primary mt-
3">
            </div>
        </form>
        <table class="table table-striped mt-5">
            <thead>
                <tr>
                    <th>Título</th>
                    <th>Autor</th>
                
```

```

<th>ISBN#</th>
<th></th>
</tr>
</thead>
<tbody id="book-list"></tbody>
</table>
</div>

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

</html>

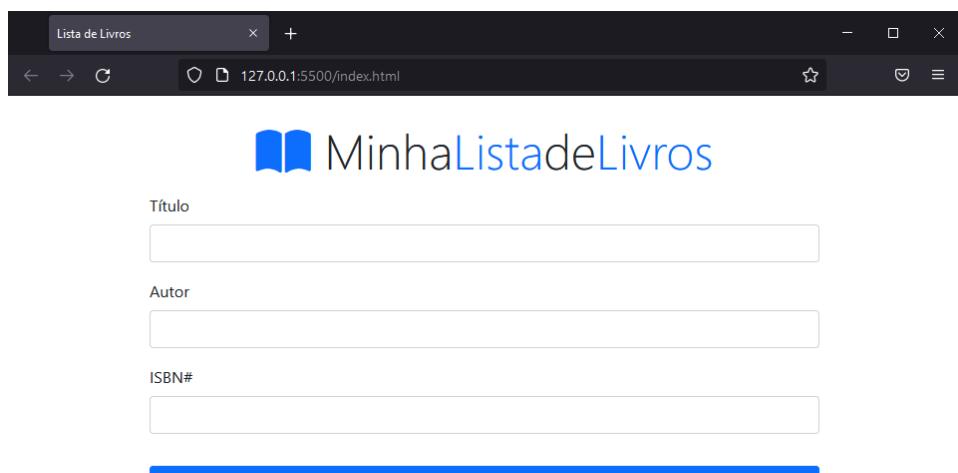
```

Nesse código, estamos usando na marcação `<link>` o **CDN do Bootstrap**. Isso vai permitir utilizarmos as **classes predefinidas do Bootstrap** para **estilizar a nossa página**. E também o **CDN do FontAwesome** para podermos colocar os ícones na página.

Esse código mostra também a marcação `<script>` sem nenhum código **JavaScript** entre a **abertura** e o **fechamento** da tag, apenas o atributo `src` com o valor **main.js**. Isso significa que o código **JavaScript** está em um **arquivo externo**. Portanto, temos que criar esse novo arquivo **main.js** dentro do diretório do projeto.

Vamos deixar o arquivo **main.js vazio** e à medida que vamos aprendendo coisas novas iremos **completar a implementação do código JavaScript**.

Abra o arquivo `index.html`, clique no botão  **Go Live** da extensão **Live Server**. A página inicial mostrada será:



Interface da Lista de Livros.

Colocando código em JS para manipular a nossa lista de itens

Vamos implementar nosso código em **JS** para ser possível inserir dados na nossa lista de livros. Siga os passos para completar o código do **JavaScript**

Primeiro, vamos criar uma **classe** chamada **Book**, que representará um livro. No arquivo **main.js**, insira o seguinte código:

```
// Classe Book: Representa um Livro
class Book {
    constructor(title, author, isbn) {
        this.title = title;
        this.author = author;
        this.isbn = isbn;
    }
}
```

Nesse caso, nossa **classe possui um construtor** que com os seguintes atributos: **title**, **author** e **isbn**. O campo **title** é para armazenar o **nome do livro**, o campo **author** para armazenar o **nome do autor** e o **isbn** para armazenar o **ISBN da obra**.

Agora vamos implementar a classe **UI (User Interface)**.

```
// Classe UI (User Interface): Lida com as tarefas da UI
class UI {
    static displayBooks() {
        const books = Store.getBooks();
        books.forEach((book) => UI.addBookToList(book));
    }

    static addBookToList(book) {
        const list = document.querySelector('#book-list');
        const row = document.createElement('tr');
        row.innerHTML =
            ` ${book.title} | ${book.author} | ${book.isbn} | X |`;
        list.appendChild(row);
    }

    static deleteBook(el) {
        if (el.classList.contains('delete')) {
            el.parentElement.parentElement.remove();
        }
    }
}
```

```

static showAlert(message, className, fontSize, alignText) {
    const div = document.createElement('div');
    div.className = `alert alert-${className} ${fontSize} text-${alignText}`;
    div.appendChild(document.createTextNode(message));
    const container = document.querySelector('.container');
    const form = document.querySelector('#book-form');
    container.insertBefore(div, form);

    // Apaga o alerta após 3 segundos
    setTimeout(() => document.querySelector('.alert').remove(), 3000);
}

static clearFields() {
    document.querySelector('#title').value = '';
    document.querySelector('#author').value = '';
    document.querySelector('#isbn').value = '';
}
}

```

A classe **UI** foi criado para lidar com as **tarefas da interface do usuário (UI)**. Por isso, ela possui diversos métodos, tais como:

displayBooks: esse método busca o livro no fomulário através da função **getBooks** da classe **Store** (**que ainda vai ser implementada**) e insere o livro na tabela abaixo do formulário através da função **addBookToList** da class **UI**.

addBookToList: esse método cria a **linha da nossa tabela** e a **acrescenta** na nossa **lista**.

deleteBook: remove o livro da lista.

showAlert: mostra uma **mensagem** de **alerta por 3 segundos** (função **setTimeout**) com uma **mensagem passada como parâmetro**. Observe que nessa função podemos passar **além da mensagem**, o **tipo do alerta** (parâmetro **className**), o **tamanho da fonte** (**fontSize**) e o **alinhamento do texto** (**alignText**).

clearFields: esse método **limpa os campos do formulário**, após clicarmos no botão **Adicionar Livro**.

Agora vamos implementar a classe **Store**. No arquivo **main.js**, insira o seguinte código.

```

// Classe Store: Lida com o Storage Local do navegador web
class Store {
    static getBooks() {
        let books;
        if (localStorage.getItem('books') === null) {
            books = [];
        } else {
            books = JSON.parse(localStorage.getItem('books'));
        }
        return books;
    }
}

```

```

    }

    static addBook(book) {
        const books = Store.getBooks();
        books.push(book);
        localStorage.setItem('books', JSON.stringify(books));
    }

    static removeBook(isbn) {
        const books = Store.getBooks();
        books.forEach((book, index) => {
            if (book.isbn === isbn) {
                books.splice(index, 1);
            }
        });
        localStorage.setItem('books', JSON.stringify(books));
    }
}

```

A classe **Store** lida com o **Storage local** do navegador web. A propriedade **localStorage** permite acessar um **objeto Storage local**. Todo navegador web tem um **local para armazenar dados temporários da página/aplicação** que ele está **exibindo/executando**. Sendo assim, a classe **Store** possui diversos métodos para manipular dados no **Storage local**:

getBooks: busca (lê) um livro (**dado**) armazenado no **Storage local**. Esse método testa se o **Storage** está vazio. Caso positivo, a função retorna um **array vazio**. Caso contrário, ela retorna utiliza o método **JSON.parse()** para analisar uma string **JSON** e construir um objeto **JavaScript** (**que no nosso caso é um array**).

addBook: adiciona um livro no **Storage** local do navegador web.

removeBook: apaga um livro no **Storage** local do navegador web de acordo com o índice (**index**) selecionado.

Agora no nosso código vamos adicionar os eventos para invocarem os métodos das classes implementados anteriormente. Primeiro vamos implementar o evento para mostrar os livros na interface do usuário.

```
// Evento: Mostrar os Livros
document.addEventListener('DOMContentLoaded', UI.displayBooks);
```

O evento para mostrar os livros invoca a função **displayBooks** da classe **UI**. Dessa forma os livros disponíveis no **Storage local** do **navegador web** serão exibidos na **UI**.

Em seguida, vamos implementar o adicionar um livro no **Storage local** do navegador e, assim, exibi-lo na nossa lista.

```
// Evento: Adicionar um Livro
document.querySelector('#book-form').addEventListener('submit', (e) => {
    // Evita a ação de submeter formulário
    e.preventDefault();

    // Busca valores do formulário
    const title = document.querySelector('#title').value;
    const author = document.querySelector('#author').value;
    const isbn = document.querySelector('#isbn').value;

    // Validação, verifica se os campos então preenchidos
    if (title === '' || author === '' || isbn === '') {
        UI.showAlert(
            'Favor preencha todos os campos!',
            'danger',
            'fs-4 fw-bold',
            'center'
        );
    } else {
        // Instância da classe Book
        const book = new Book(title, author, isbn);

        // Adiciona o Livro na Lista
        UI.addBookToList(book);

        // Adiciona o Livro no Localstorage do navegador web
        Store.addBook(book);

        // Mostra mensagem de sucesso
        UI.showAlert('Livro adicionado', 'success', 'fs-4 fw-bold', 'center');

        // Limpa os campos do formulário
        UI.clearFields();
    }
});
```

Esse evento é disparado quando **submetemos o formulário**, ou seja, quando **clicamos no botão Adicionar Livro**. Nesse evento podemos observar o método **addBookList**, usado para exibir o **livro na UI**, o método **addBook**, usado para adicionar o livro no **Storage local**, o método **showAlert**, para exibir uma mensagem de que o **livro foi adicionado com sucesso**, e o método **clearFields**, usado para limpar os campos do formulário.

Por fim, vamos adicionar o evento para **remover um livro da lista** e do **Storage local**.

```
// Evento: Remover um Livro
document.querySelector('#book-list').addEventListener('click', (e) => {
    // Remove o Livro da UI
    UI.deleteBook(e.target);
```

```
// Remove o Livro do Localstore do navegador web
Store.removeBook(e.target.parentElement.previousElementSibling.textContent);

// Mostra mensagem de sucesso
UI.showAlert('Livro removido', 'success', 'fs-4 fw-bold', 'center');
});
```

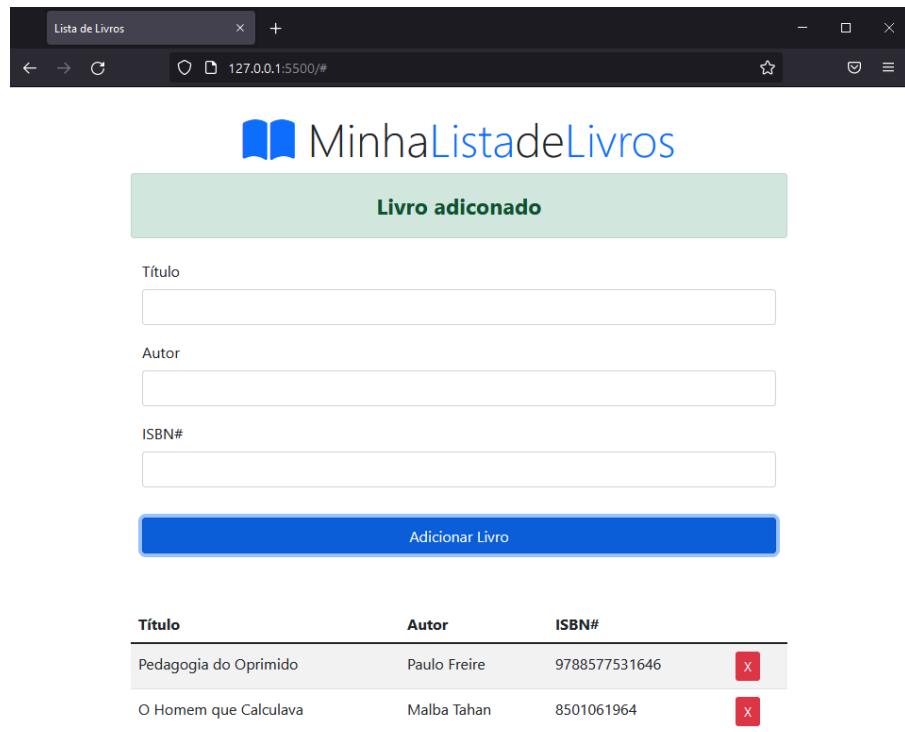
Esse evento utiliza o método **deleteBook** para remover o **livro na lista da interface do usuário**, o método **removeBook** para remover o livro do Storage local e o método **showAlert** para mostra uma **mensagem de alerta** quando o **livro for removido**.

Agora podemos brincar com nossa interface gráfica. Por exemplo, vamos adicionar dois livros:

Realize o cadastro:

Título: Pedagogia do Oprimido
 Autor: Paulo Freire
 ISBN: 9788577531646

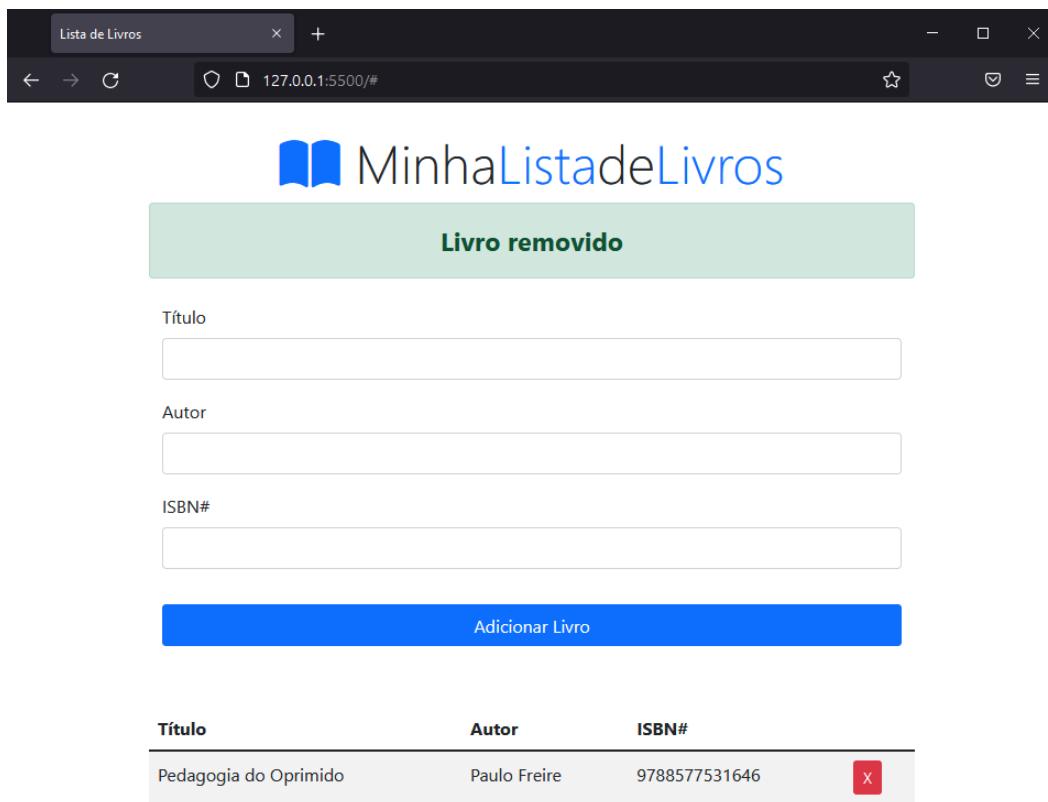
Título: O Homem que Calculava
 Autor: Malba Tahan
 ISBN: 8501061964



Título	Autor	ISBN#
Pedagogia do Oprimido	Paulo Freire	9788577531646
O Homem que Calculava	Malba Tahan	8501061964

Observe que a mensagem de **Livro adicionado** é apresentada por **3 segundos** e depois ela desaparece da tela.

Agora podemos retirar um livro da lista. Por exemplo, vamos clicar no X do livro **O Homem que calculava**.



Título	Autor	ISBN#
Pedagogia do Oprimido	Paulo Freire	9788577531646

Observe que a mensagem de livro removido será exibida por 3 segundos e desaparece da tela.

LocalStorage

A funcionalidade do **localStorage** consiste em salvar, adicionar, recuperar ou **excluir dados localmente em um navegador Web**. Dessa forma, o localStorage nos permite armazenar dados de forma simples e sem expiração, ou seja, ficam lá **enquanto não apagarmos por código ou pelo próprio navegador**.

O fato do dado ficar armazenado sem expiração é uma das vantagens de se usar o **localStorage**, assim podemos atualizar a nossa página quantas vezes quisermos que os livros nunca sumirão. Os dados só serão apagados, quando fecharmos o navegador ou apagarmos manualmente da interface do usuário.

É muito importante procurar ler mais sobre **localStorage**, você pode começar por esses artigos:

- <https://tableless.com.br/guia-f%C3%A1cil-sobre-usar-localstorage-com-javascript/>
- <https://medium.com/jaquarebetech/dlskaddaldkslkdlskdlk-333dae8ef9b8>
- <https://warcontent.com/localstorage-javascript/>
- <https://www.treinaweb.com.br/blog/quando-usar-sessionstorage-e-localstorage>



Filtro em uma Lista

Os objetivos desta aula são:

- Exercitar os conhecimentos aprendidos por meio da criação de projetos;
- Compreender a aplicação da linguagem Javascript em diferentes contextos.

Sobre o filtro em uma lista

Nessa aula, você aprenderá como fazer um **filtro de busca** dentro de uma **lista de contatos** fictícios.



Vamos praticar

Abra o **VS Code** e escolha um **diretório de trabalho para o seu projeto**.

Crie um diretório para seu projeto com o nome representativo, por exemplo, **JS_Filtro_Lista**

Crie um arquivo dentro do diretório do projeto com o nome **index.html**.

Insira o seguinte código no seu arquivo **index.html**.

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

<head>
    <meta charset="UTF-8" />
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap.min.css" rel="stylesheet"
        integrity="sha384-E4kWBq78iYhFldvKuhfTAU6auU8tT94WrHftjDbrCEXSU1oBoqyl2QvZ6jIW3"
        crossorigin="anonymous" />
    <link rel="stylesheet" href="./css/estilo.css" />
    <link rel="shortcut icon" href="#" />
    <title>Meus Contatos</title>
</head>

<body>
    <div class="container">
        <h1 class="text-center mt-3 mb-3">Meus Contatos</h1>
        <input type="text" id="filterInput" placeholder="Buscar nomes..." class="form-control mb-2" />
        <ul id="names" class="list-group">
            <li class="mb-1">
                <h5>A</h5>
            </li>
            <li class="list-group-item">
                <a href="#">Abe</a>
            </li>
            <li class="list-group-item">
                <a href="#">Adam</a>
            </li>
            <li class="list-group-item">
                <a href="#">Alan</a>
            </li>
        </ul>
    </div>
</body>
```

```

        </li>
        <li class="list-group-item">
            <a href="#">Anna</a>
        </li>
        <li class="mb-1">
            <h5>B</h5>
        </li>
        <li class="list-group-item">
            <a href="#">Beth</a>
        </li>
        <li class="list-group-item">
            <a href="#">Bill</a>
        </li>
        <li class="list-group-item">
            <a href="#">Bob</a>
        </li>
        <li class="list-group-item">
            <a href="#">Brad</a>
        </li>
        <li class="mb-1">
            <h5>C</h5>
        </li>
        <li class="list-group-item">
            <a href="#">Carrie</a>
        </li>
        <li class="list-group-item">
            <a href="#">Cathy</a>
        </li>
        <li class="list-group-item">
            <a href="#">Courtney</a>
        </li>
    </ul>

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

</html>

```

Nesse código, estamos usando na marcação `<link>` o **CDN do Bootstrap**. Isso vai permitir utilizarmos as **classes predefinidas do Bootstrap** para **estilizar a nossa página**.

Esse código possui uma marcação `<link>` com o arquivo **CSS** e no atributo **href** mostra que ele está em uma **pasta chamada css**. Portanto, vamos criar o diretório **css** e o arquivo **estilo.css** dentro desse projeto com o seguinte código.

```

ul {
    list-style-type: none;
}

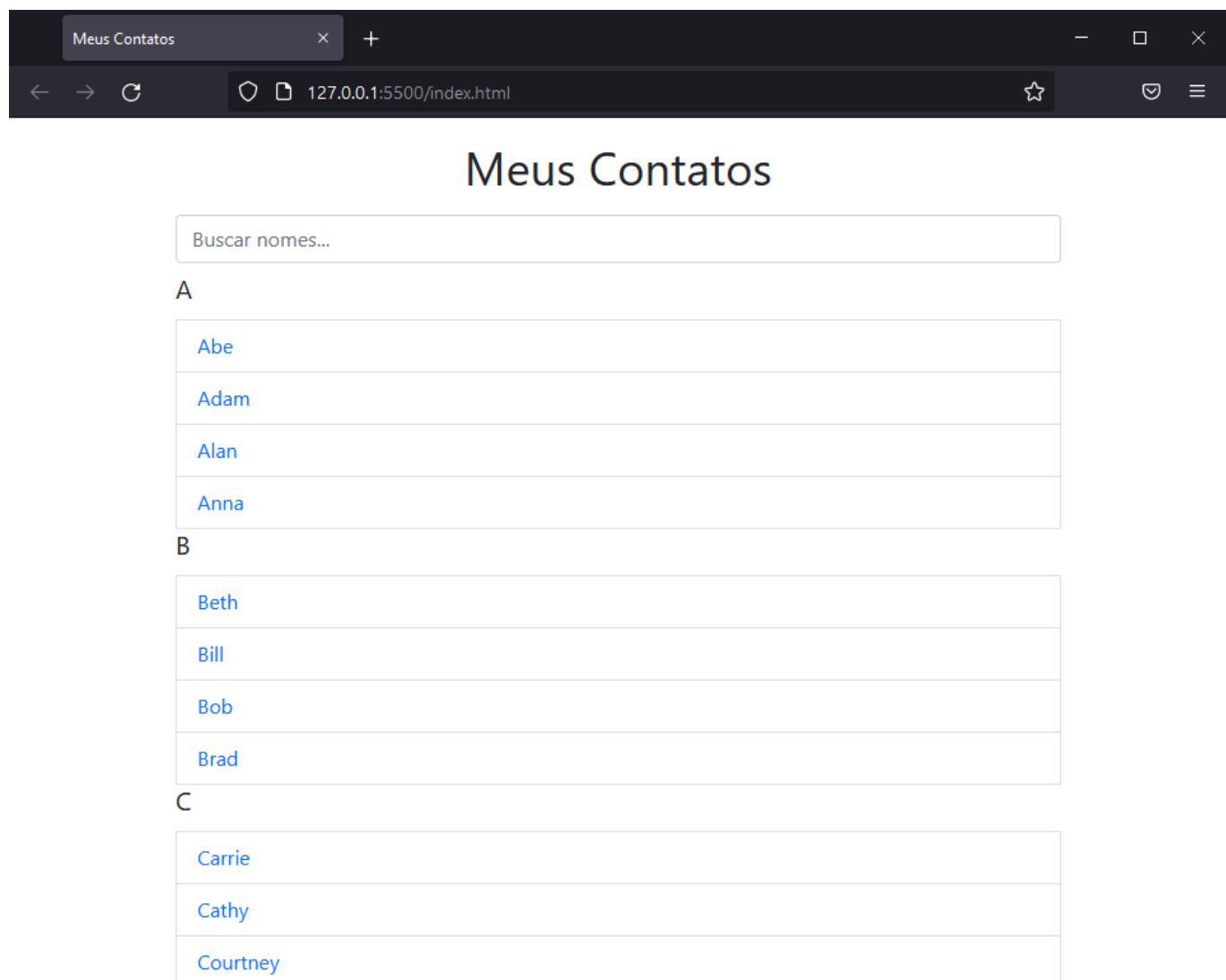
```

```
a {
    text-decoration: none;
}
```

Esse código mostra também a marcação <script> sem nenhum código **JavaScript** entre a **abertura** e o **fechamento** da tag, apenas o atributo **src** com o valor **main.js**. Isso significa que o código **JavaScript** está em um **arquivo externo**. Portanto, temos que criar esse novo arquivo **main.js** dentro do diretório do projeto.

Vamos deixar o arquivo **main.js vazio** e à medida que vamos aprendendo coisas novas iremos **completar a implementação do código JavaScript**.

Abra o arquivo `index.html`, clique no botão  **Go Live** da extensão **Live Server**. A página inicial mostrada será:



The screenshot shows a browser window with the title bar "Meus Contatos". The address bar displays the URL "127.0.0.1:5500/index.html". The main content area is titled "Meus Contatos" and contains a search bar with the placeholder "Buscar nomes...". Below the search bar, the contacts are organized into three sections: "A", "B", and "C", each containing a list of names. The names listed are:

- A**: Abe, Adam, Alan, Anna
- B**: Beth, Bill, Bob, Brad
- C**: Carrie, Cathy, Courtney

Interface da Lista de Contatos.

Colocando código em JS para manipular a nossa lista de itens

Vamos implementar nosso **código** em **JS** para ser possível buscar nomes na nossa lista de contatos. Siga os passos para completar o código do **JavaScript**

No arquivo **main.js**, insira o seguinte código.

```
// Buscar o elemento input da página
let filterInput = document.getElementById('filterInput');
// Adicionar um evento a esse elemento
filterInput.addEventListener('keyup', filterNames);
```

Utilizamos o método **getElementById()** para retornar o elemento **<input>** da página HTML e **referenciá-lo** na variável **filterInput**.

Em seguida, adicionamos um **evento** a esse **elemento do HTML**. Nesse caso, o **evento** é o **keyup**, que invoca a função **filterNames** todo vez que soltamos uma **tecla digitada** no **<input>** **Buscar nomes...**

Agora vamos implementar a **função** para **filtrar os nomes de acordo com o valor digitado** no elemento **<input>** do HTML.

```
function filterNames() {
    // Buscar o valor digitado no elemento input
    let filterValue = document.getElementById('filterInput').value.toUpperCase();

    // Buscar as lista de nomes
    let ul = document.getElementById('names');
    // Buscar os elementos <li> da lista não ordenada <ul>
    let li = ul.querySelectorAll('li.list-group-item');

    // Laços para percorrer os elementos da lista
    for (let i = 0; i < li.length; i++) {
        let a = li[i].getElementsByTagName('a')[0];
        console.log(a);
        // Verifica se o item da lista começa com o valor
        if (a.innerHTML.toUpperCase().indexOf(filterValue) > -1) {
            li[i].style.display = '';
        } else {
            li[i].style.display = 'none';
        }
    }
}
```

Vamos analisar essa função por partes:

Com o comando:

```
let filterValue = document.getElementById('filterInput').value.toUpperCase();
```

Transformamos o conteúdo digitado no elemento `<input>` para **letras maiúsculas**. Isso é interessante para ficar mais fácil de **comparar valores de strings** sem se preocupar se o usuário digitou **maiúsculo** ou **minúsculo**.

Em seguida, buscamos a nossa lista **não ordenada ** através da função `getElementById()` que pega o elemento com a `id="names"`. Nessa instrução estamos buscando a lista não ordenada completa com todos os elementos dentro dela, tanto os **itens ** quanto os **heading <h5>** e os **link <a>**.

Com o comando:

```
let li = ul.querySelectorAll('li.list-group-item');
```

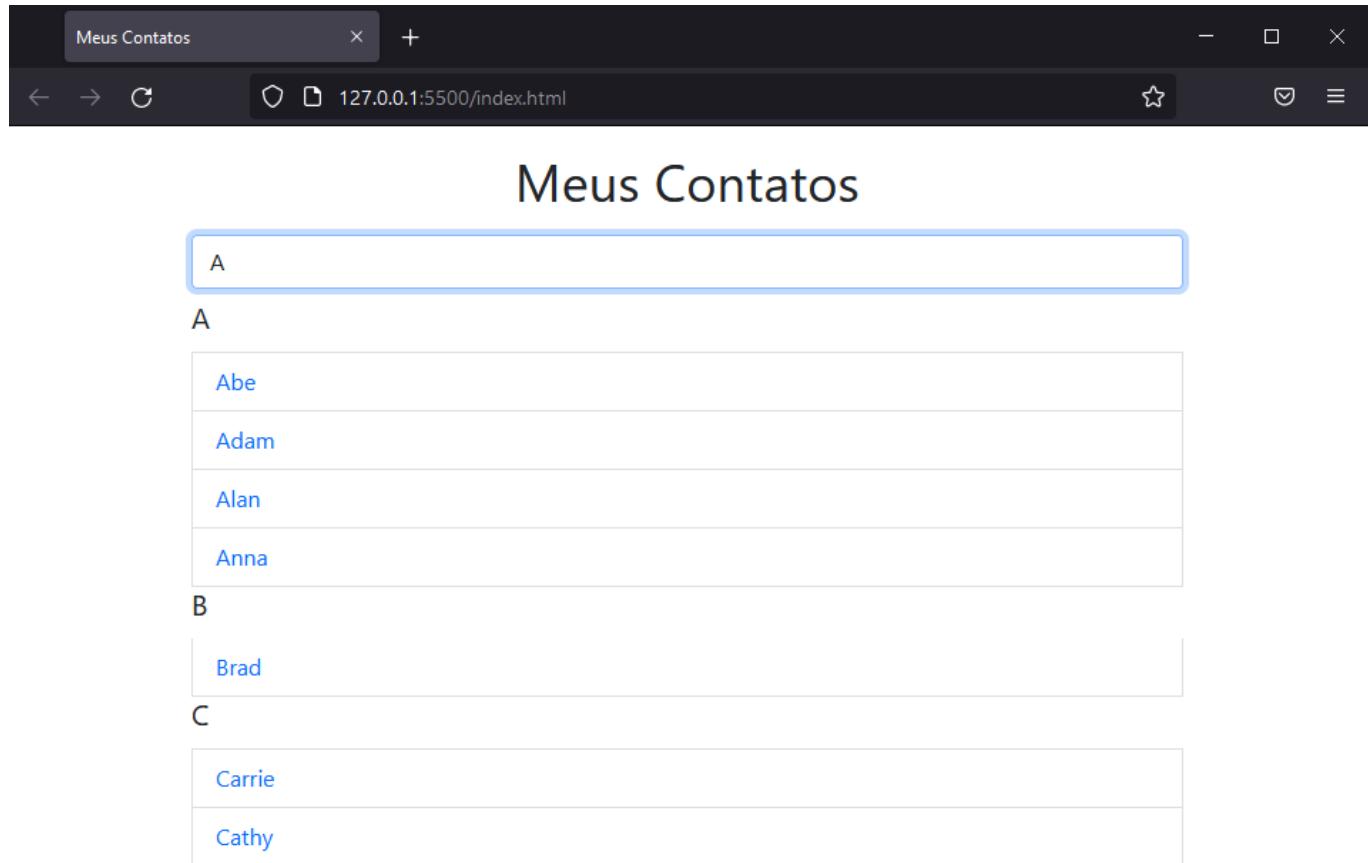
Separamos apenas os elementos `` que possuem a classe `.list-group-item` usando o método `querySelectorAll()`. Desse modo, agora no **array li**, encontramos apenas os **itens da lista com os nomes da lista de contatos**.

Depois, utilizamos um **laço de repetição** para percorrer o **array li** e buscar **item por item** para poder alterar a propriedade `style.display`. Essa propriedade permite **exibir ou não um elemento HTML** na página web.

Separamos o elemento `<a>` do **array li** com a função `getElementsByTagName()`.

Em seguida, testamos **se o item da lista** (elemento `<a>`) possui a string digitado no pelo usuário `<input>` para **buscar nome**. Caso a string for encontrada, o método `indexOf()` retornará um valor maior do **-1** e a propriedade `style.display` desse elemento fica vazia, exibindo assim o elemento. Caso contrário, o método `indexOf()` não encontrar a string, ele retornará o valor **-1** e a propriedade `style.display` recebe o valor **none** fazendo com que o elemento deixe de ser **exibido na página**.

Você pode testar a sua busca na lista. Por exemplo, digite **A**.



Meus Contatos

A

Abe
Adam
Alan
Anna

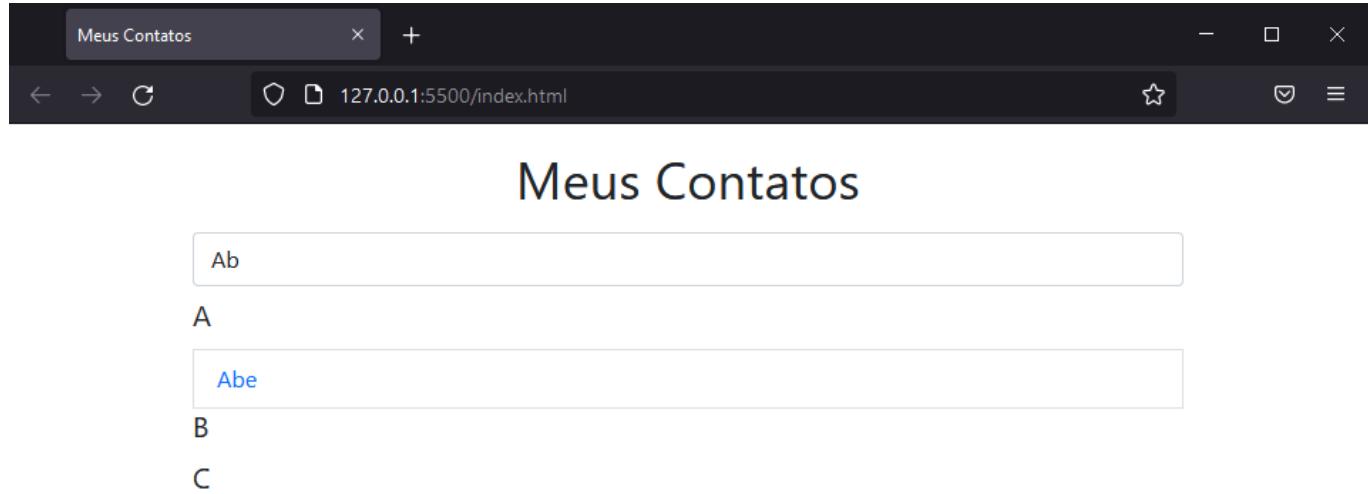
B

Brad

C

Carrie
Cathy

Por exemplo, digite **Ab**.



Meus Contatos

Ab

A

Abe

B

C

Por exemplo, digite **C**.

Meus Contatos x +

127.0.0.1:5500/index.html

Meus Contatos

C

A

B

C

Carrie
Cathy
Courtney



Image Slider

Os objetivos desta aula são:

- Exercitar os conhecimentos aprendidos por meio da criação de projetos;
- Compreender a aplicação da linguagem Javascript em diferentes contextos.

Sobre o filtro em uma lista

Nessa aula, você aprenderá como fazer um **filtro de busca** dentro de um apresentador de imagens com informações diversas.

Criação do projeto inicial

Abra o **VS Code** e escolha um **diretório de trabalho para o seu projeto**.

Crie um diretório para seu projeto com o nome representativo, por exemplo, **JS_ Image_Slider**

Crie um arquivo dentro do diretório do projeto com o nome **index.html**.

Insira o seguinte código no seu arquivo **index.html**.

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

<head>
    <meta charset="UTF-8">
    <link rel="shortcut icon" href="#">

    <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/6.0.0-beta3/css/all.min.css"
        integrity="sha512-
Fo3rlrZj/k7ujTnHg4CGR2D7kSs0v4LLanw2qksYuR1Ez0+tcaEPQogQ0KaoGN26/zrn20ImR1DfuLWh0o7aBA==
"crossorigin="anonymous" referrerpolicy="no-referrer" />

    <link rel="stylesheet" href=".//css/estilo.css">

    <title>Apresentador de Imagens</title>

</head>

<body>
    <div class="slider">
        <div class="slide atual">
            <div class="content">
                <h1>Slide Um</h1>
                <p>Lorem ipsum, dolor sit amet consectetur adipisicing elit. Necessitatibus quaerat quidem beatae alias ducimus hic officia velit illum dolores omnis.</p>
            </div>
        </div>

        <div class="slide">
            <div class="content">
                <h1>Slide Dois</h1>
                <p>Lorem ipsum dolor sit amet consectetur adipisicing elit. Adipisci officiis repellendus facilis odio quos quas amet labore voluptas nihil ipsam.</p>
            </div>
        </div>
    </div>
</body>
```

```
        </div>
    </div>

    <div class="slide">
        <div class="content">
            <h1>Slide Três</h1>
            <p>Lorem ipsum dolor, sit amet consectetur adipisicing elit. Corporis alias aut excepturi id
                reprehenderit vero a odit dolor quia laudantium.</p>
        </div>
    </div>

    <div class="slide">
        <div class="content">
            <h1>Slide Quatro</h1>
            <p>Lorem ipsum dolor sit amet consectetur adipisicing elit. Libero expedita nulla eius dolores amet
                recusandae ad molestias accusantium veniam iste.</p>
        </div>
    </div>

    <div class="slide">
        <div class="content">
            <h1>Slide Cinco</h1>
            <p>Lorem ipsum dolor sit amet consectetur adipisicing elit. Libero expedita nulla eius dolores amet
                recusandae ad molestias accusantium veniam iste.</p>
        </div>
    </div>

    <div class="slide">
        <div class="content">
            <h1>Slide Seis</h1>
            <p>Lorem ipsum dolor sit amet consectetur adipisicing elit. Libero expedita nulla eius dolores amet
                recusandae ad molestias accusantium veniam iste.</p>
        </div>
    </div>
</div>

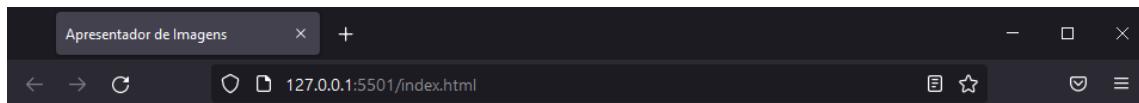
<div class="buttons">
    <button id="prev"><i class="fas fa-arrow-left"></i></button>
    <button id="next"><i class="fas fa-arrow-right"></i></button>
</div>

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

</html>
```

Nesse código, estamos usando na marcação <link> o **CDN** do **FontAwesome** para poder usar os ícones das **setas de avançar** e de **retroceder** na nossa página.

Se clicarmos no botão  Go Live da extensão **Live Server**, veremos a nossa página sem nenhuma formatação ainda.



Slide Um

Lore ipsum, dolor sit amet consectetur adipisicing elit. Necessitatibus quaerat quidem beatae alias ducimus hic officia velit illum dolores omnis.

Slide Dois

Lore ipsum dolor sit amet consectetur adipisicing elit. Adipisci officiis repellendus facilis odio quo quas amet labore voluptas nihil ipsam.

Slide Três

Lore ipsum dolor, sit amet consectetur adipisicing elit. Corporis alias aut excepturi id reprehenderit vero a odit dolor quia laudantium.

Slide Quatro

Lore ipsum dolor sit amet consectetur adipisicing elit. Libero expedita nulla eius dolores amet recusandae ad molestias accusantium veniam iste.

Slide Cinco

Lore ipsum dolor sit amet consectetur adipisicing elit. Libero expedita nulla eius dolores amet recusandae ad molestias accusantium veniam iste.

Slide Seis

Lore ipsum dolor sit amet consectetur adipisicing elit. Libero expedita nulla eius dolores amet recusandae ad molestias accusantium veniam iste.



Nesse código podemos ver também a **marcação <link>** para utilizarmos o arquivo **estilo.css**, onde vamos inserir o seguinte código para estilizar a nossa página.

```
@import url('https://fonts.googleapis.com/css?family=Roboto');

* {
  box-sizing: border-box;
  margin: 0;
  padding: 0;
}

body {
  font-family: 'Roboto', sans-serif;
  background: #333;
  color: #fff;
  line-height: 1.6;
```

```
}

.slider {
    position: relative;
    overflow: hidden;
    height: 100vh;
    width: 100vw;
}

.slide {
    position: absolute;
    top: 0;
    left: 0;
    width: 100%;
    height: 100%;
    opacity: 0;
    transition: opacity 0.4s ease-in-out;
}

.slide.atual {
    opacity: 1;
}

.slide .content {
    position: absolute;
    bottom: 70px;
    left: -600px;
    opacity: 0;
    width: 600px;
    background-color: rgba(255, 255, 255, 0.8);
    color: #333;
    padding: 35px;
}

.slide .content h1 {
    margin-bottom: 10px;
}

.slide.atual .content {
    opacity: 1;
    transform: translateX(600px);
    transition: all 0.7s ease-in-out 0.3s;
}

.buttons button#next {
    position: absolute;
    top: 40%;
    right: 15px;
}
```

```

.buttons button#prev {
  position: absolute;
  top: 40%;
  left: 15px;
}

.buttons button {
  font-size: 2rem;
  border: 2px solid #fff;
  background-color: transparent;
  color: #fff;
  cursor: pointer;
  padding: 13px 15px;
  border-radius: 50%;
  outline: none;
}

.buttons button:hover {
  background-color: #fff;
  color: #333;
}

@media (max-width: 500px) {
  .slide .content {
    bottom: -300px;
    left: 0;
    width: 100%;
  }

  .slide.atual .content {
    transform: translateY(-300px);
  }
}

/* Backgorund Images */

.slide:first-child {
  background: url('https://source.unsplash.com/RyRpq9SUwAU/1600x900') no-repeat
    center top/cover;
}

.slide:nth-child(2) {
  background: url('https://source.unsplash.com/Nyvq2juw4_o/1600x900') no-repeat
    center top/cover;
}

.slide:nth-child(3) {
  background: url('https://source.unsplash.com/TMOeGZw9NY4/1600x900') no-repeat
    center top/cover;
}

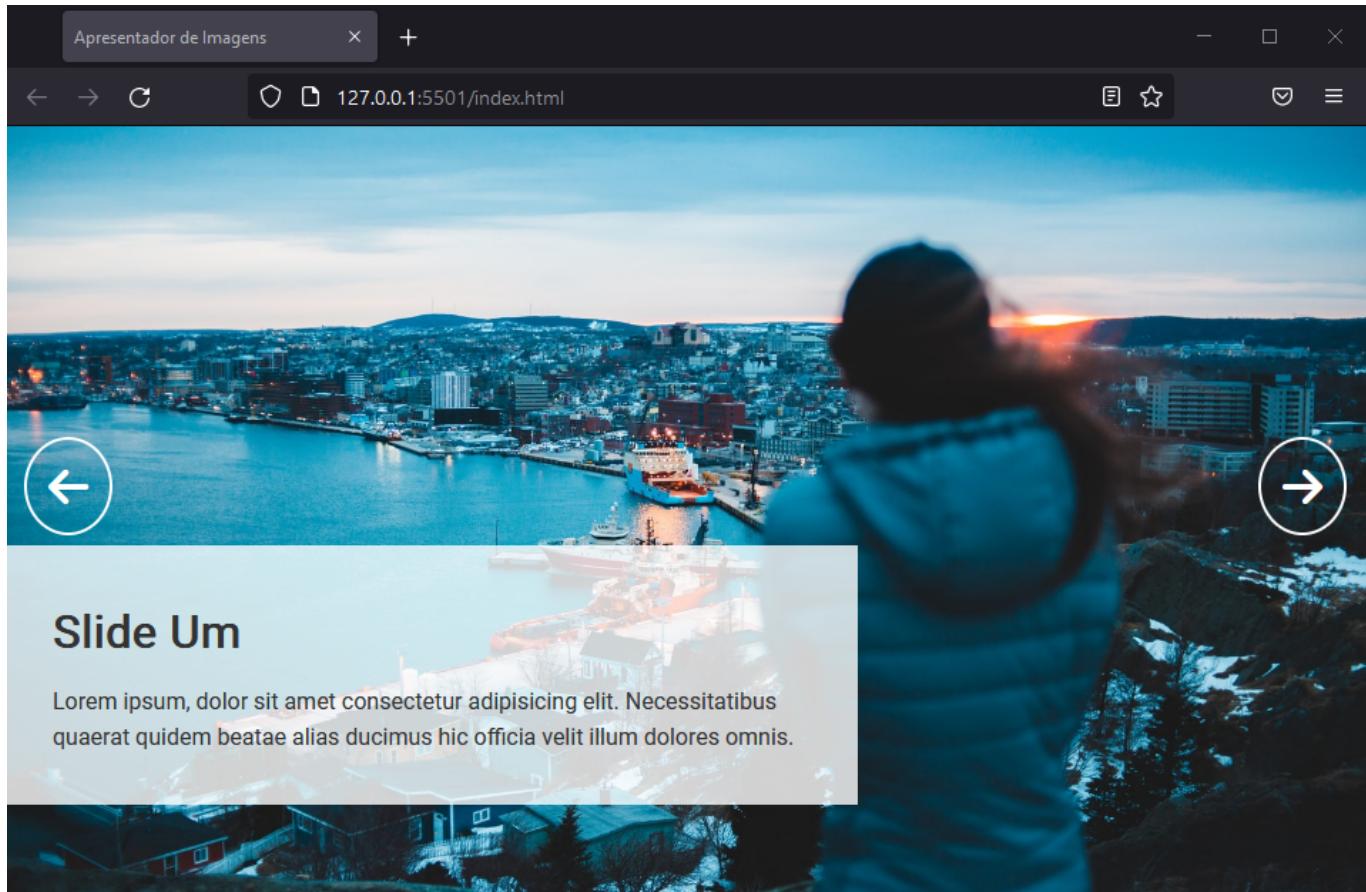
```

```
.slide:nth-child(4) {  
  background: url('https://source.unsplash.com/yXpA_eCbtzI/1600x900') no-repeat  
  center top/cover;  
}  
.slide:nth-child(5) {  
  background: url('https://source.unsplash.com/ULmaQh9Gvbg/1600x900') no-repeat  
  center top/cover;  
}  
.slide:nth-child(6) {  
  background: url('https://source.unsplash.com/ggZuL3BTSJU/1600x900') no-repeat  
  center center/cover;  
}  
  
@media (max-width: 500px) {  
  .slide .content {  
    bottom: -300px;  
    left: 0;  
    width: 100%;  
  }  
  
  .slide.atual .content {  
    transform: translateY(-300px);  
  }  
}
```

Esse código mostra também a marcação <script> sem nenhum código **JavaScript** entre a **abertura** e o **fechamento** da tag, apenas o atributo **src** com o valor **main.js**. Isso significa que o código **JavaScript** está em um **arquivo externo**. Portanto, temos que criar esse novo arquivo **main.js** dentro do diretório do projeto.

Vamos deixar o arquivo **main.js vazio** e à medida que vamos aprendendo coisas novas iremos **completar a implementação do código JavaScript**.

Se visualizarmos o nosso site agora, ele vai aparecer todo estilizado, mas os botões de avançar e retroceder ainda não irão funcionar.



Note que o site se adapta e redimensiona de acordo com o tamanho da janela do navegador.

Colocando código em JS para implementar nosso Image Slider

Vamos implementar nosso código em **JS** para criar o nosso **Image Slider**. Siga os passos para completar o código do **JavaSc0ript**

No arquivo **main.js**, insira o seguinte código.

```
// Busca todos os elementos com a classe slide
const slides = document.querySelectorAll('.slide');
// Busca o botão de avançar
const next = document.querySelector('#next');
// Busca o botão de retroceder
const prev = document.querySelector('#prev');
// Habilita ou desabilita a transição automática entre slides
const auto = false; // Auto scroll
// Intervalo entre transições
const intervalTime = 2000;
// Tempo para exibir o slide na transição automática
let slideInterval;
```

Utilizamos o método **querySelectorAll()** para retornar todos os elementos com a classe **.slide** que estão na **página HTML** e referenciá-lo na **variável slides**.

Depois, usamos o método **querySelector()** para retornar o botão de **avançar** da nossa aplicação, pois esse botão é que tem a **id="next"**.

Em seguida, usamos o método **querySelector()** para retornar o botão de **retroceder** da nossa aplicação, pois esse botão é que tem a **id="prev"**.

Temos a variável **auto** usada para **habilitar** e **desabilitar** a **apresentação de slides automática**. No momento vamos deixar com o valor **false**, para passarmos os slides **manualmente**.

Depois, temos a variável **intervalTime** para determinar o **intervalo** entre as **transições de slides** no modo automático.

Temos a variável **slideInterval** que é usada para determinar o **tempo** que o **slide ficará na tela** na transição **automática**.

Agora vamos implementar a função para **avançar para o próximo slide**.

```
// Função para avançar para o próximo slide
const nextSlide = () => {
    // Busca o slide com a classe atual
    const atual = document.querySelector('.atual');
    // Remove a classe atual do slide
    atual.classList.remove('atual');
    // Verifica se existe o próximo slide
    if (atual.nextElementSibling) {
        // Adiciona a classe atual no próximo slide
        atual.nextElementSibling.classList.add('atual');
    } else {
        // Adiciona a classe atual no primeiro slide
        slides[0].classList.add('atual');
    }

    setTimeout(() => atual.classList.remove('atual'));
};


```

Vamos analisar essa função por partes:

Utilizamos o método **querySelector()** para retornar o elemento com a classe **.atual**, que é o slide que está **sendo apresentado na tela**.

Removemos a classe **atual do slide** buscado anteriormente.

Verificamos se **existe um próximo slide**. Se houver um próximo slide adicionamos a classe **.atual** nesse próximo slide. Caso **não houver** um próximo slide temos que **voltar para o início da lista**, ou seja para o **slide zero**.

Por fim, definimos o **tempo entre as transições dos slides**.

Em seguida vamos implementar a função para **retroceder para o slide anterior**.

```
// Função para voltar para o slide anterior
const prevSlide = () => {
    // Busca o slide com a classe atual
    const atual = document.querySelector('.atual');
    // Remove a classe atual do slide
    atual.classList.remove('atual');
    // Verifica se existe o slide anterior
    if (atual.previousElementSibling) {
        // Adiciona a classe atual no slide anterior
        atual.previousElementSibling.classList.add('atual');
    } else {
        // Adiciona a classe atual no último slide
        slides[slides.length - 1].classList.add('atual');
    }
    setTimeout(() => atual.classList.remove('atual'));
};
```

Vamos analisar essa função por partes:

Utilizamos o método **querySelector()** para retornar o elemento com a classe **.atual**, que é o slide que está **sendo apresentado na tela**.

Removemos a classe **atual do slide** buscado anteriormente.

Verificamos se **existe um slide anterior**. Se houver um slide anterior adicionamos a classe **.atual** nesse slide anterior. Caso **não houver** um slide anterior temos que **ir para o final da lista**, ou seja para o **último slide**.

Por fim, definimos o **tempo entre as transições dos slides**.

Vamos, então, implementar os eventos para os botões de avançar e retroceder para que possamos navegar pelos slides.

```
// Eventos para os botões
next.addEventListener('click', (e) => {
    nextSlide(); // Invoca a função próximo slide
    // Executa a transição entre slides
    if (auto) {
        clearInterval(slideInterval);
        slideInterval = setInterval(nextSlide, intervalTime);
    }
});

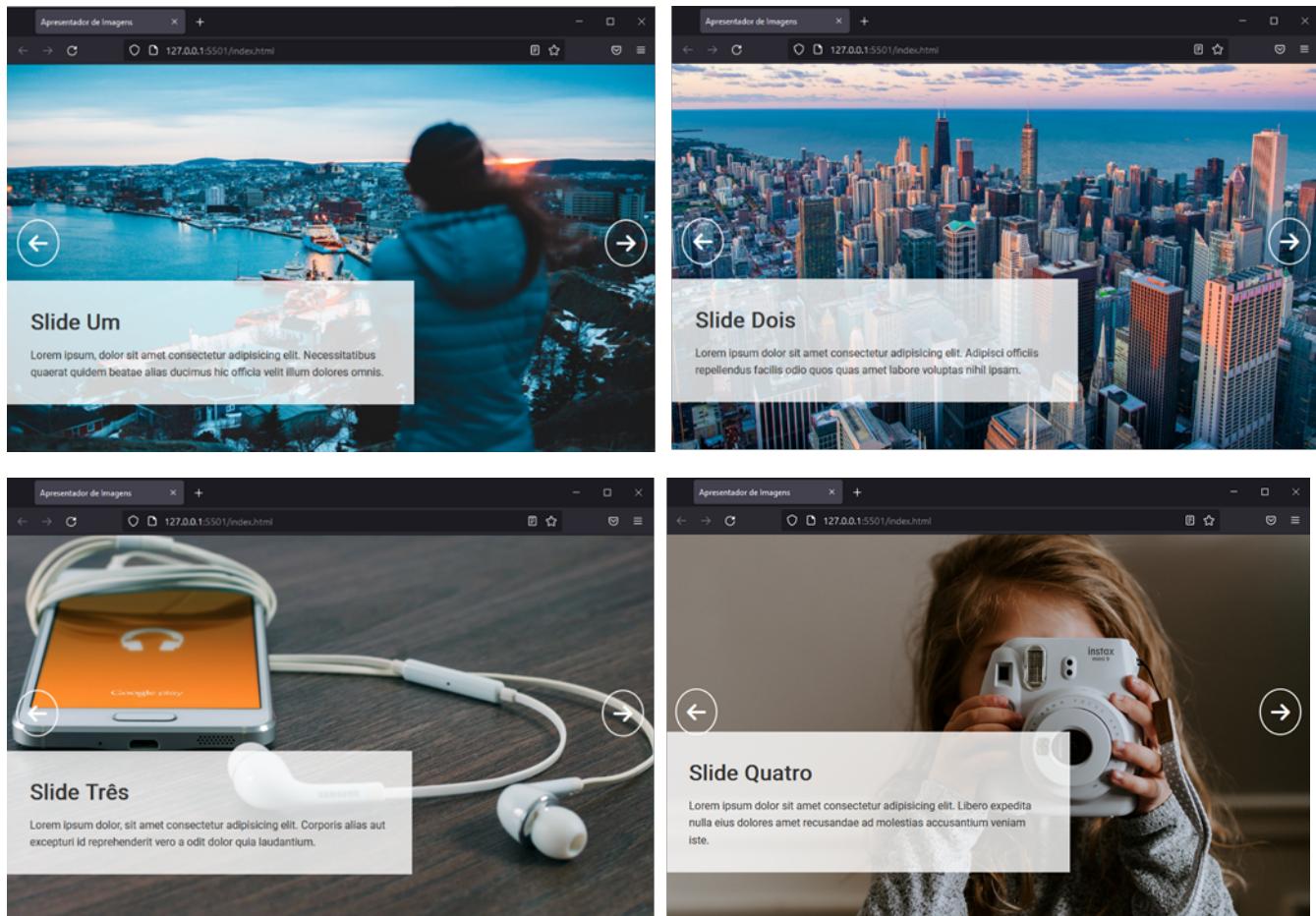
prev.addEventListener('click', (e) => {
    prevSlide(); // Invoca a função slide anterior
    // Executa a transição entre slides
    if (auto) {
        clearInterval(slideInterval);
        slideInterval = setInterval(nextSlide, intervalTime);
    }
});
```

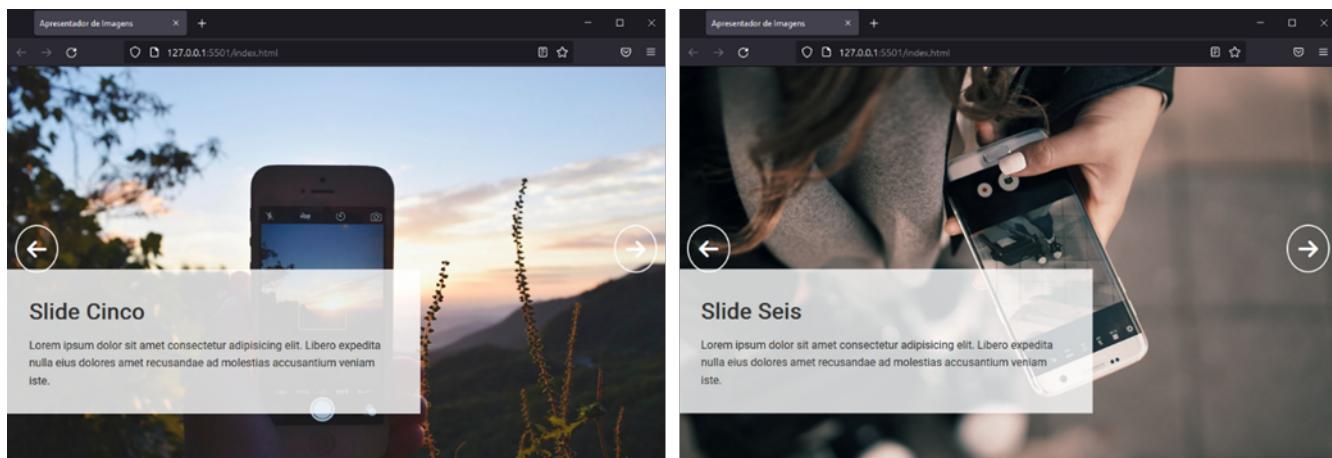
Esse trecho de código **adiciona os eventos** que são **disparados** quando clicamos os botões para **avançar e retroceder** os slides. No caso, a função **nextSlide()** é invocada quando clicamos no botão **avançar** (id="next") e a função **prevSlide()** é invocada quando clicamos no botão **retroceder** (id="prev").

Por fim, podemos configurar a **apresentação automática**, caso a variável **auto** esteja com o valor **true**.

```
// Caso a transição automática entre slides esteja habilitada
if (auto) {
    // Mostra o próximo slide automaticamente depois de um tempo
    slideInterval = setInterval(nextSlide, intervalTime);
}
```

Agora, você pode testar a aplicação brincar de passar por **todos os slides** clicando nos botões de **avançar e retroceder**.







Modal com o JavaScript

Os objetivos desta aula são:

- Exercitar os conhecimentos aprendidos por meio da criação de projetos;
- Compreender a aplicação da linguagem Javascript em diferentes contextos.

O que é um modal?

Modal é uma **janela** que abre **sobre uma página web** para **exibir ou coletar informações** e depois **ser fechada**. Muitas vezes associamos um modal a um **pop-up**. Vamos aprender a fazer um **modal** sem usar nenhum **framework** ou **biblioteca**.



Vamos praticar

Abra o **VS Code** e escolha um **diretório de trabalho para o seu projeto**.

Crie um diretório para seu projeto com o nome representativo, por exemplo, **JS_ Modal**

Crie um arquivo dentro do diretório do projeto com o nome **index.html**.

Insira o seguinte código no seu arquivo **index.html**.

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

<head>
    <meta charset="UTF-8">
    <link rel="shortcut icon" href="#">
    <link rel="stylesheet" href="./css/estilo.css">
    <title>Modal com HTML, CSS e JS</title>
</head>

<body>
    <button id="modal-btn" class="button">Abrir Modal</button>

    <div id="my-modal" class="modal">
        <div class="modal-content">
            <div class="modal-header">
                <span class="close">&times;</span>
                <h2>Cabeçalho do Modal Header</h2>
            </div>
            <div class="modal-body">
                <p>Esse é meu modal</p>
                <p>Lorem ipsum dolor sit amet consectetur adipisicing elit. Incidunt debitis dignissimos corrupti veniam deserunt repellendus tempora esse. Nulla corrupti modi cumque dolorem nam, ipsam eum, perspiciatis sint voluptatum possimus quisquam.</p>
            </div>
            <div class="modal-footer">
                <h3>Rodapé Modal</h3>
            </div>
        </div>
    </div>
```

```
</div>

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

</html>
```

Se clicarmos no botão  Go Live da extensão Live Server, veremos a nossa página sem nenhuma formatação ainda.



Cabeçalho do Modal Header

Esse é meu modal

 Lorem ipsum dolor sit amet consectetur dipisicing elit. Incidunt debitis dignissimos corrupti veniam deserunt repellendus tempora esse. Nulla corrupti modi cumque dolorem nam, ipsam eum, perspiciatis sint voluptatum possimus quisquam.

Rodapé Modal

Nesse código podemos ver também a marcação <link> para utilizarmos o arquivo **estilo.css**, onde vamos inserir o seguinte código para estilizar a nossa página.

```
body {
    font-family: Arial, Helvetica, sans-serif;
    background: #f4f4f4;
    font-size: 17px;
    line-height: 1.6;
    display: flex;
    height: 100vh;
    align-items: center;
    justify-content: center
}

.button {
    background: #428bca;
    padding: 1em 2em;
    color: #fff;
    border: 0;
    border-radius: 5px;
    cursor: pointer
}

.button:hover {
    background: #3876ac
}
```

```
.modal {  
    display: none;  
    position: fixed;  
    z-index: 1;  
    left: 0;  
    top: 0;  
    height: 100%;  
    width: 100%;  
    overflow: auto;  
    background-color: rgba(0, 0, 0, .5)  
}  
  
.modal-content {  
    margin: 10% auto;  
    width: 60%;  
    box-shadow: 0 5px 8px 0 rgba(0, 0, 0, .2), 0 7px 20px 0 rgba(0, 0, 0, .17);  
    animation: modalopen 1s  
}  
  
.modal-header {  
    background: #428bca;  
    padding: 15px;  
    color: #fff;  
    border-top-left-radius: 5px;  
    border-top-right-radius: 5px  
}  
  
.modal-header h2 {  
    margin: 0  
}  
  
.modal-footer {  
    background: #428bca;  
    padding: 10px;  
    color: #fff;  
    text-align: center;  
    border-bottom-left-radius: 5px;  
    border-bottom-right-radius: 5px  
}  
  
.modal-footer h3 {  
    margin: 0  
}  
  
.modal-body {  
    padding: 10px 20px;  
    background: #fff  
}
```

```
.close {
  color: #ccc;
  float: right;
  font-size: 30px
}

.close:hover, .close:focus {
  color: #000;
  text-decoration: none;
  cursor: pointer
}

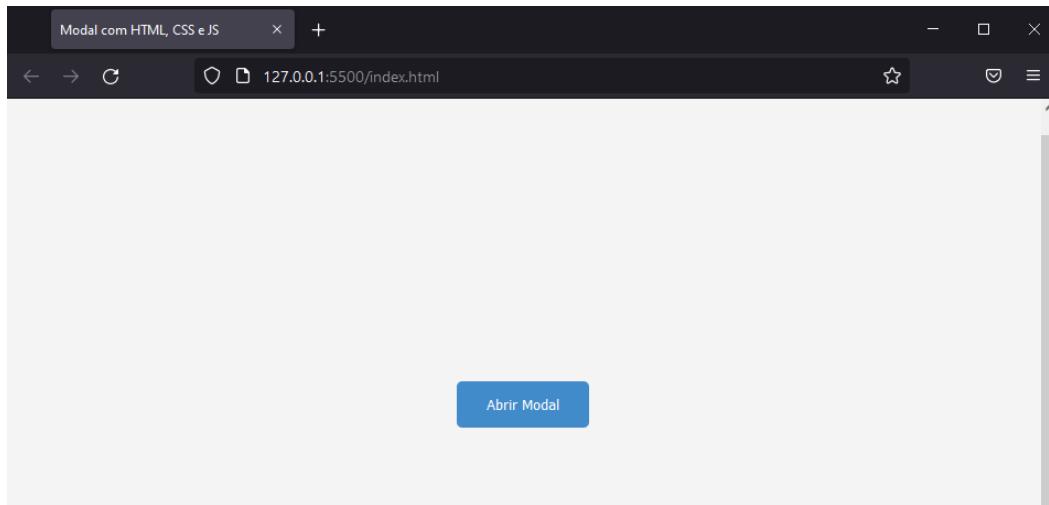
@keyframes modalopen {
  from {
    opacity: 0
  }

  to {
    opacity: 1
  }
}
```

Esse código mostra também a marcação <script> sem nenhum código **JavaScript** entre a **abertura** e o **fechamento** da tag, apenas o atributo **src** com o valor **main.js**. Isso significa que o código **JavaScript** está em um **arquivo externo**. Portanto, temos que criar esse novo arquivo **main.js** dentro do diretório do projeto.

Vamos deixar o arquivo **main.js vazio** e à medida que vamos aprendendo coisas novas iremos **completar a implementação do código JavaScript**.

Se visualizarmos o nosso site agora, ele vai aparecer com o botão estilizada, mas ainda não funcionando e nosso modal oculto na página.



Colocando código em JS para criar nosso modal

Vamos implementar nosso código em **JS** para criar o nosso **Modal**. Siga os passos para completar o código do **JavaScript**

No arquivo **main.js**, insira o seguinte código.

```
// Busca os elementos com a ID my-modal no DOM
const modal = document.querySelector('#my-modal');
// Busca os elementos com a ID modal-btn no DOM
const modalBtn = document.querySelector('#modal-btn');
// Busca os elementos com a classe .close no DOM
const closeBtn = document.querySelector('.close');
```

Utilizamos o método **querySelector()** para retornar o elemento com a **id="my-modal"** no DOM, que no nosso exemplo é o elemento **<div>** do código **HTML** com as informações do nosso modal.

Usamos o método **querySelector()** para retornar o botão com a **id="modal-btn"** que é usado para **abrir** o nosso **modal**.

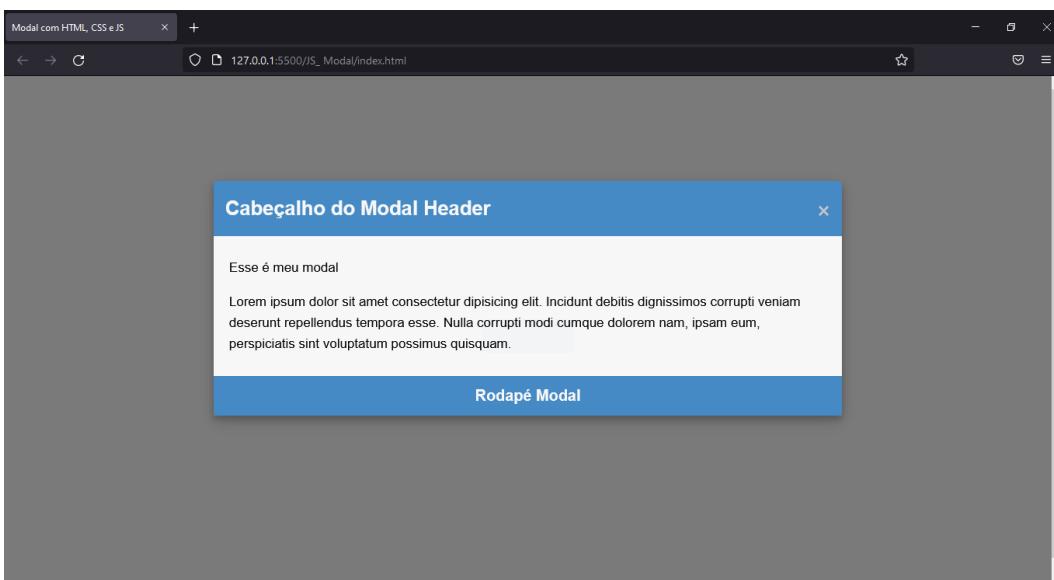
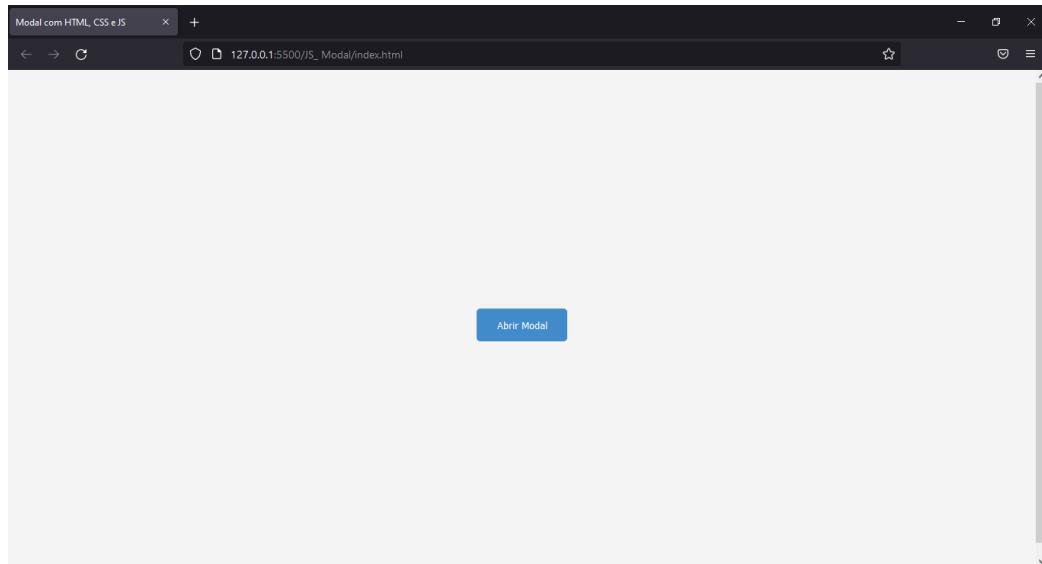
Em seguida, usamos o método **querySelector()** para **retornar o botão** com a classe **.close** dentro do **código do modal** (HTML) que é usado para fechar o nosso modal.

Vamos, então, implementar os **eventos** para os **botões de abrir e fechar** o nosso **modal**.

```
// Adiciona eventos de clique para invocar as funções para abrir e fechar o modal
modalBtn.addEventListener('click', () => (modal.style.display = 'block'));
closeBtn.addEventListener('click', () => (modal.style.display = 'none'));
```

Esse trecho de código adiciona os eventos que são disparados quando clicamos os botões para **abrir o modal** ou no **botão X** para **fechar o modal**. Nesse caso, estamos modificando a propriedade **display** para **block**, quando clicamos no **botão abrir** e para **none**, quando clicamos no **botão fechar**.

Agora, você pode testar a aplicação brincar de **abrir e fechar o modal**.



Agora vamos **implementar** a **funcionalidade** de **fechar o modal** quando clicarmos em qualquer região da janela do navegador que esteja **fora do modal**.

```
// Fecha o modal ser clicarmos fora do modal
window.addEventListener('click', (e) => {
  if (e.target == modal) {
    modal.style.display = 'none';
  }
});
```

Adicionamos um **evento** de clique no objeto **window** do **JS**, que representa a janela do **navegador web**. Depois, testamos se o local clicado está **fora do nosso modal**. Caso **verdadeiro**, mudamos a propriedade **display** para **none** e paramos **exibir o modal**.

Portanto, agora você pode **fechar** clicando no **botão X** ou **fora do modal** na janela do navegador.



Vamos praticar

Implementando dois modais

Podemos colocar em uma página web, **quantos modais acharmos necessários**. Basta identificá-los corretamente **dentro do seu código**. No nosso projeto, vamos criar um outro arquivo HTML chamado **index2.html** e inserir o seguinte código:

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

<head>
    <meta charset="UTF-8">
    <link rel="shortcut icon" href="#">
    <link rel="stylesheet" href="./css/estilo2.css">
    <title>Múltiplos Modais</title>

</head>

<body>
    <div class="container">
        <h2>Primeiro Modal</h2>
        <!-- Abrir o Modal -->
        <button class="modal-button" href="#myModal1">Abrir Modal</button>
        <hr>
        <h2>Segundo Modal</h2>
        <!-- Abrir o Modal -->
        <button class="modal-button" href="#myModal2">Abrir Modal</button>
    </div>

    <!-- Modal 1 -->
    <div id="myModal1" class="modal">
        <!-- Modal content -->
        <div class="modal-content">
            <div class="modal-header">
                <span class="close">&times;</span>
                <h2>Cabeçalho do Modal 1</h2>
            </div>
            <div class="modal-body">
                <p>Algum texto para ser exibido no corpo do Modal</p>
                <p>Algum outro texto...</p>
            </div>
        </div>
    </div>
</body>
```

```

        </div>
        <div class="modal-footer">
            <h3>Rodapé do Modal</h3>
        </div>
    </div>

</div>

<!-- Modal 2 -->
<div id="myModal2" class="modal">
    <!-- Modal content -->
    <div class="modal-content">
        <div class="modal-header">
            <span class="close">&times;</span>
            <h2>Cabeçalho do Modal 2</h2>
        </div>
        <div class="modal-body">
            <p>Algum texto para ser exibido no corpo do Modal</p>
            <p>Algum outro texto...</p>
        </div>
        <div class="modal-footer">
            <h3>Rodapé do Modal</h3>
        </div>
    </div>
</div>

<script src=".//js/main2.js"></script>
</body>

</html>

```

Note que os modais possuem **ID diferentes**, temos o modal com a **id="myModal1"** e, temos o modal com a **id="myModal2"**.

Agora vamos criar o arquivo **estilo2.css** dentro do diretório **css** para colocar o código de estilização do nosso segundo exemplo.

```

<!DOCTYPE html>
<html lang="pt-br">

<head>
    <meta charset="UTF-8">
    <link rel="shortcut icon" href="#">
    <link rel="stylesheet" href=".//css/estilo2.css">
    <title>Múltiplos Modais</title>

</head>

```

```

<body>
  <div class="container">
    <h2>Primeiro Modal</h2>
    <!-- Abrir o Modal -->
    <button class="modal-button" href="#myModal1">Abrir Modal</button>
    <hr>
    <h2>Segundo Modal</h2>
    <!-- Abrir o Modal -->
    <button class="modal-button" href="#myModal2">Abrir Modal</button>
  </div>

  <!-- Modal 1 -->
  <div id="myModal1" class="modal">
    <!-- Modal content -->
    <div class="modal-content">
      <div class="modal-header">
        <span class="close">&times;</span>
        <h2>Cabeçalho do Modal 1</h2>
      </div>
      <div class="modal-body">
        <p>Algum texto para ser exibido no corpo do Modal</p>
        <p>Algum outro texto...</p>
      </div>
      <div class="modal-footer">
        <h3>Rodapé do Modal</h3>
      </div>
    </div>
  </div>

  <!-- Modal 2 -->
  <div id="myModal2" class="modal">
    <!-- Modal content -->
    <div class="modal-content">
      <div class="modal-header">
        <span class="close">&times;</span>
        <h2>Cabeçalho do Modal 2</h2>
      </div>
      <div class="modal-body">
        <p>Algum texto para ser exibido no corpo do Modal</p>
        <p>Algum outro texto...</p>
      </div>
      <div class="modal-footer">
        <h3>Rodapé do Modal</h3>
      </div>
    </div>
  </div>
</div>

```

```
<script src="./js/main2.js"></script>
</body>

</html>
```

Por fim, vamos criar o arquivo **main2.js** para inserir o nosso código em **JS**:

```
// Busca os botões da DOM na página web
var btn = document.querySelectorAll('button.modal-button');

// Busca todos os modais na página
var modals = document.querySelectorAll('.modal');

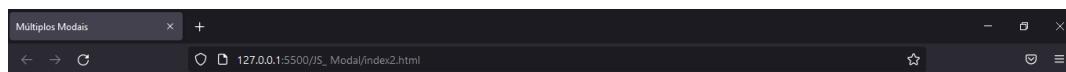
// Busca o elemento <span> que fecha o nosso modal
var spans = document.getElementsByClassName('close');

// Quando o usuário clica no botão, abre o modal
for (var i = 0; i < btn.length; i++) {
    btn[i].onclick = function (e) {
        e.preventDefault();
        modal = document.querySelector(e.target.getAttribute('href'));
        modal.style.display = 'block';
    };
}

// Quando o usuário clica no <spam> (x), fecha o modal
for (var i = 0; i < spans.length; i++) {
    spans[i].onclick = function () {
        for (var index in modals) {
            if (typeof modals[index].style !== 'undefined')
                modals[index].style.display = 'none';
        }
    };
}

// Quando o usuário clioca em qualquer Lugar fora do modal,
// feche-o
window.onclick = function (event) {
    if (event.target.classList.contains('modal')) {
        for (var index in modals) {
            if (typeof modals[index].style !== 'undefined')
                modals[index].style.display = 'none';
        }
    }
};
```

Agora você pode testar o seu código, **abrindo e fechando** um **modal** de **cada vez**.

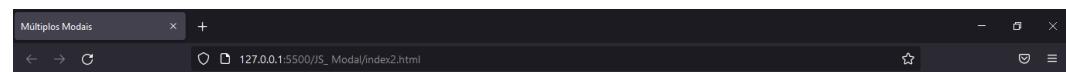


Primeiro Modal

Abrir Modal

Segundo Modal

Abrir Modal



Cabeçalho do Modal 1

Algun texto para ser exibido no corpo do Modal
Algun outro texto...

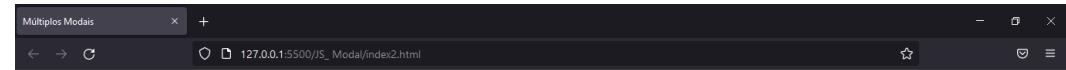
Rodapé do Modal

Primeiro Modal

Abrir Modal

Segundo Modal

Abrir Modal



Cabeçalho do Modal 2

Algun texto para ser exibido no corpo do Modal
Algun outro texto...

Rodapé do Modal

Primeiro Modal

Abrir Modal

Segundo Modal

Abrir Modal



Classificação com estrelas no JavaScript

Os objetivos desta aula são:

- Exercitar os conhecimentos aprendidos por meio da criação de projetos;
- Compreender a aplicação da linguagem Javascript em diferentes contextos.

Bons estudos!

Classificação com estrelas no JavaScript

Nessa aula, você aprenderá como fazer um sistema para **classificar** um **produto** por **estrelas** como existe em diversos sites de compras na internet.



Vamos praticar

Abra o **VS Code** e escolha um **diretório de trabalho para o seu projeto**.

Crie um diretório para seu projeto com o nome representativo, por exemplo, **JS_Star_Rating**

Crie um arquivo dentro do diretório do projeto com o nome **index.html**.

Insira o seguinte código no seu arquivo **index.html**.

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

<head>
    <meta charset="UTF-8" />
    <link rel="shortcut icon" href="#" />
    <link rel="stylesheet" href="./css/estilo.css" />

    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap.min.css"
    rel="stylesheet"
        integrity="sha384-
1BmE4kWBq78iYhFldvKuhfTAU6auU8tT94WrHftjDbrCEXSU1oBoqyl2QvZ6jIW3"
crossorigin="anonymous" />

    <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-
awesome/6.0.0-beta3/css/all.min.css"
        integrity="sha512-
Fo3rlrZj/k7ujTnHg4CGR2D7kSs0v4LLanw2qksYuR1Ez0+tcaEPQogQ0KaoGN26/zrn20ImR1DfuLWn0o7aBA=="
        crossorigin="anonymous" referrerPolicy="no-referrer" />

    <title>Sistema de Classificação</title>
</head>

<body>
    <div class="container mt-5">
        <form>
            <select id="product-select" class="form-control custom-select">
                <option value="0" disabled selected>Selecione um Produto</option>
                <option value="sony">Sony 4K TV</option>
                <option value="samsung">Samsung 4K TV</option>
                <option value="tcl">TCL 4K TV</option>
                <option value="panasonic">LG 4K TV</option>
                <option value="phillips">Phillips 4K TV</option>
            </select>
        </form>
    </div>
</body>
```

```
</select>

    <input type="number" id="rating-control" class="form-control" step="0.1"
max="5" placeholder="Rate 1 - 5"
disabled />
</form>

<table class="table table-striped">
    <thead>
        <tr>
            <th>4K Television</th>
            <th>Rating</th>
        </tr>
    </thead>
    <tbody>
        <tr class="sony">
            <td>Sony 4k</td>
            <td>
                <div class="stars-outer">
                    <div class="stars-inner"></div>
                </div>
                <span class="number-rating"></span>
            </td>
        </tr>

        <tr class="samsung">
            <td>Samsung 4K TV</td>
            <td>
                <div class="stars-outer">
                    <div class="stars-inner"></div>
                </div>
                <span class="number-rating"></span>
            </td>
        </tr>
        <tr class="tcl">
            <td>TCL 4K TV</td>
            <td>
                <div class="stars-outer">
                    <div class="stars-inner"></div>
                </div>
                <span class="number-rating"></span>
            </td>
        </tr>
        <tr class="panasonic">
            <td>Panasonic 4K TV</td>
            <td>
                <div class="stars-outer">
                    <div class="stars-inner"></div>
                </div>
            </td>
        </tr>
    </tbody>
</table>
```

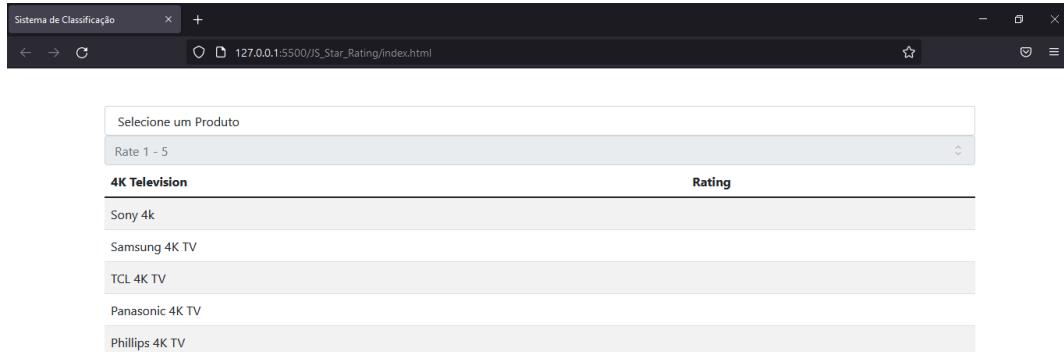
```

        <span class="number-rating"></span>
    </td>
</tr>
<tr class="phillips">
    <td>Phillips 4K TV</td>
    <td>
        <div class="stars-outer">
            <div class="stars-inner"></div>
        </div>
        <span class="number-rating"></span>
    </td>
</tr>
</tbody>
</table>
</div>

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

</html>
```

Se clicarmos no botão  Go Live da extensão Live Server, veremos a nossa página sem nenhuma formatação ainda.



Nesse código podemos ver também a marcação `<link>` para utilizarmos o arquivo **estilo.css**, onde vamos inserir o seguinte código para estilizar a nossa página.

```

.stars-outer {
    position: relative;
    display: inline-block;
}

.stars-outer::before {
    content: '\f005 \f005 \f005 \f005 \f005';
    font-family: 'Font Awesome 5 Free';
    font-weight: 900;
    color: #ccc;
```

```

}

.stars-inner {
    position: absolute;
    top: 0;
    left: 0;
    white-space: nowrap;
    overflow: hidden;
    width: 0;
}

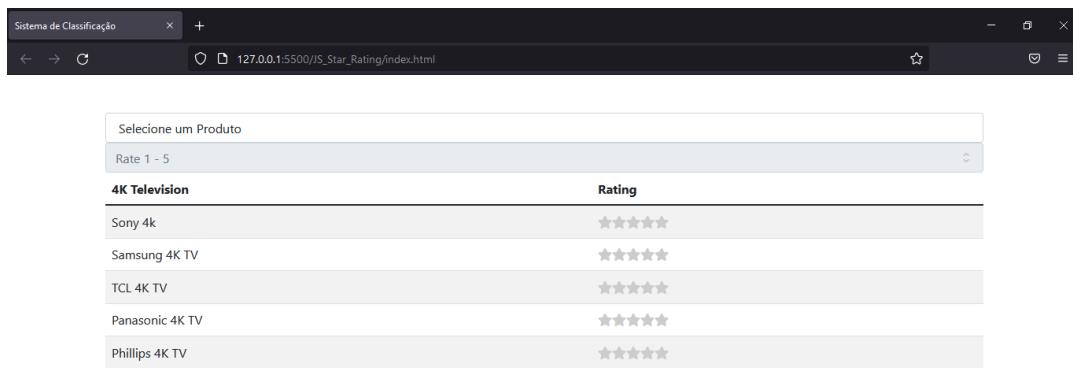
.stars-inner::before {
    content: '\f005 \f005 \f005 \f005';
    font-family: 'Font Awesome 5 Free';
    font-weight: 900;
    color: #f8ce0b;
}

```

Esse código mostra também a marcação <script> sem nenhum código **JavaScript** entre a **abertura** e o **fechamento** da tag, apenas o atributo **src** com o valor **main.js**. Isso significa que o código **JavaScript** está em um **arquivo externo**. Portanto, temos que criar esse novo arquivo **main.js** dentro do diretório do projeto.

Vamos deixar o arquivo **main.js vazio** e à medida que vamos aprendendo coisas novas iremos **completar a implementação do código JavaScript**.

Se visualizarmos o nosso site agora, ele vai aparecer com o botão estilizada, mas ainda sem interatividade.



Colocando código JS para fazer a classificação de estrelas

Vamos implementar nosso código em **JS** para ser possível **converter** um **valor em real para outras moedas**. Siga os passos para completar o código do **JavaScript**.

No arquivo **main.js**, insira o seguinte código.

```
// Classificação inicial
const ratings = {
    sony: 4.7,
    samsung: 3.4,
    vizio: 2.3,
    panasonic: 3.6,
    phillips: 4.1,
};

// Total de estrelas
const starsTotal = 5;

// Produto
let product;

// Elementos do formulário
const productSelect = document.getElementById('product-select');
const ratingControl = document.getElementById('rating-control');
```

Criamos a variável **ratings** com os **valores iniciais** da **classificação** de cada uma das **televisões** que vamos mostrar na nossa aplicação.

Em seguida, criamos a nossa variável **starsTotal**, que **limita do valor máximo** de estrelas na classificação.

Criamos a variável **product** para **receber o produto selecionado** na lista suspensa da página web.

Buscamos o elemento **<select>** do formulário, que contém os nomes dos produtos.

E, buscamos o elemento **<input>** do formulário, que contém a faixa de **1 a 5** usada para **classificar os produtos**.

Vamos, então, implementar os eventos para serem disparados quando manipulamos o **<select>** e o **<input>** do formulário. No arquivo **script.js**, insira o seguinte código:

```
// Executa o método getRatings quando o DOM carregado no navegador
document.addEventListener('DOMContentLoaded', getRatings);

// Evento change para selecionar um produto
productSelect.addEventListener('change', (e) => {
    product = e.target.value;
    // Habilita do controle de classificação
    ratingControl.disabled = false;
    ratingControl.value = ratings[product];
});
```

```
// Evento blur, quando o elemento perde o foco
ratingControl.addEventListener('blur', (e) => {
    const rating = e.target.value;

    // Garante o valor máximo da classificação 5 estrelas
    if (rating > 5) {
        alert('Por favor, classifique entre 1 - 5');
        return;
    }

    // Alteração da classificação
    ratings[product] = rating;

    getRatings();
});
```

Esse trecho de código **adiciona** os **eventos** que são disparados quando **manipulamos** os **elementos** do **formulário**. Primeiro, temos o evento **DOMContentLoaded**, que disparado quando o **DOM** é **carregado** no **navegador**. Esse evento chama a **função getRatings()**, que **atualiza** a **classificação das estrelas** para cada um dos produtos. Essa função será **implementada** no **próximo passo**.

Em seguida, temos o **evento change**, que é disparado quando **selecionamos** um **produto** no elemento **<select>** do formulário. Esse evento habilita o **<input>** para podermos **alterar** a **classificação do produto**.

Por fim, temos o evento **blur**, que **limita a faixa da classificação até 5 estrelas, atualiza o valor** da **classificação**, então, invoca a **função getRatings()** para **atualizar** a informação exibida na página web.

Agora, vamos inserir o código da função **getRatings()**:

```
// Busca classificação
function getRatings() {
    for (let rating in ratings) {
        // console.log(ratings[rating]);

        // Transforma a classificação em porcentagem
        const starPercentage = (ratings[rating] / starsTotal) * 100;
        // console.log(starPercentage);

        // Arredonda o resultado para valores inteiros
        const starPercentageRounded = `${
            Math.round(starPercentage / 10) * 10
        }%`;
        // console.log(starPercentageRounded);

        // Define a Largura da do preenchimento de acordo com a porcentagem
```

```

document.querySelector(`.${rating} .stars-inner`).style.width =
    starPercentageRounded;

// Adiciona o valor da classificação para ser exibido na tela
document.querySelector(`.${rating} .number-rating`).innerHTML =
    ratings[rating];
}

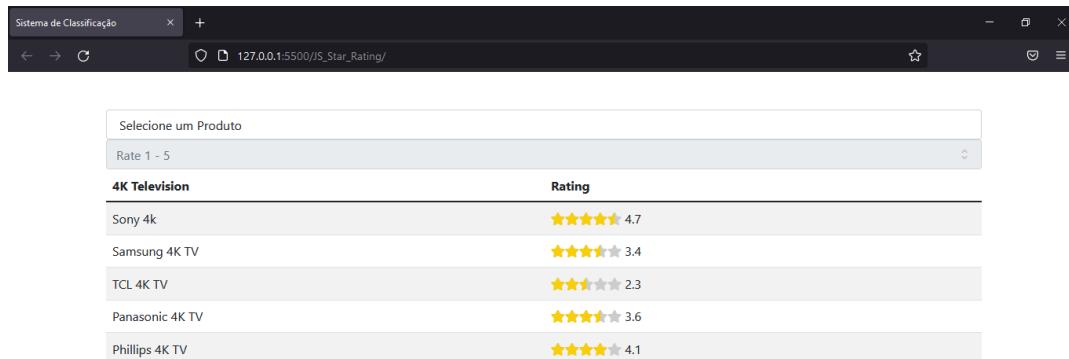
}

```

A função **getRatings()** **preenche/colore** as **estrelas** usadas para a **classificação de um produto**. Primeiro, calculamos o **valor percentual**, que um produto **foi classificado**. Em seguida, **arredondamos e convertemos** o valor **calculado anteriormente** para um **valor inteiro**, assim temos a classificação entre **0 a 100**. Então, definimos a **largura** que será **colorida** das estrelas de acordo com o **percentual calculado**.

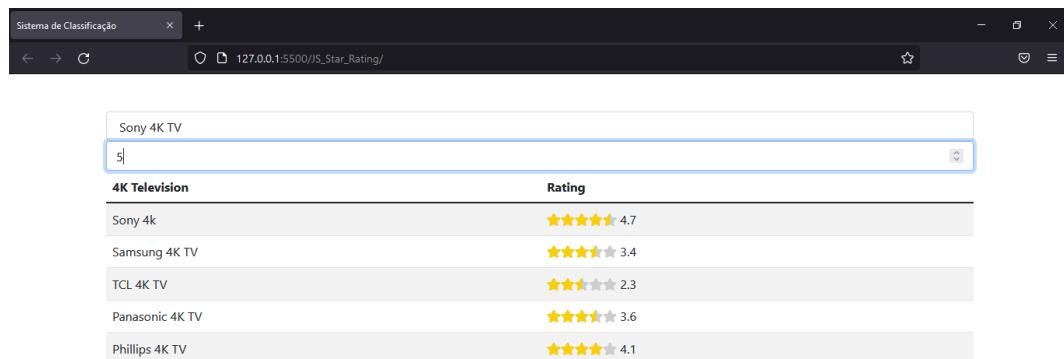
Por fim, mostramos o valor de **0 a 5** da classificação do produto na página web.

Agora, você pode testar a aplicação brincar de alterar a classificação de um produto. Inicialmente, a sua página aparecerá assim:



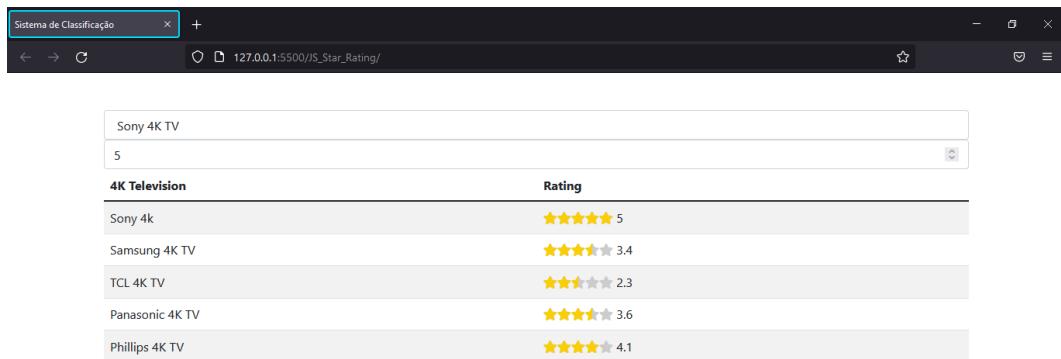
4K Television	Rating
Sony 4k	★★★★★ 4.7
Samsung 4K TV	★★★★☆ 3.4
TCL 4K TV	★★★☆☆ 2.3
Panasonic 4K TV	★★★★☆ 3.6
Phillips 4K TV	★★★★★ 4.1

Podemos então selecionar o produto **TV Sony 4k** e alterar a classificação para **5**.



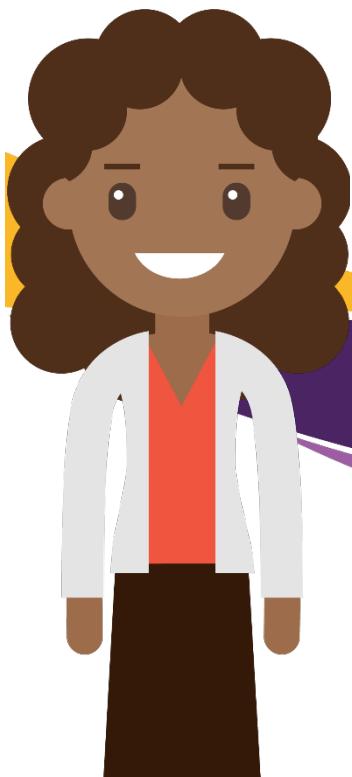
4K Television	Rating
Sony 4k	★★★★★ 4.7
Samsung 4K TV	★★★★☆ 3.4
TCL 4K TV	★★★☆☆ 2.3
Panasonic 4K TV	★★★★☆ 3.6
Phillips 4K TV	★★★★★ 4.1

Quando o foco sair do elemento **<input>** do formulário (evento **blur**), a classificação será alterada na página.



4K Television	Rating
Sony 4k	★★★★★ 5
Samsung 4K TV	★★★★☆ 3.4
TCL 4K TV	★★★★☆ 2.3
Panasonic 4K TV	★★★★☆ 3.6
Phillips 4K TV	★★★★☆ 4.1

Pronto, agora você pode brincar com as outras classificações de **TVs**.



API com JavaScript - Parte 01

Os objetivos desta aula são:

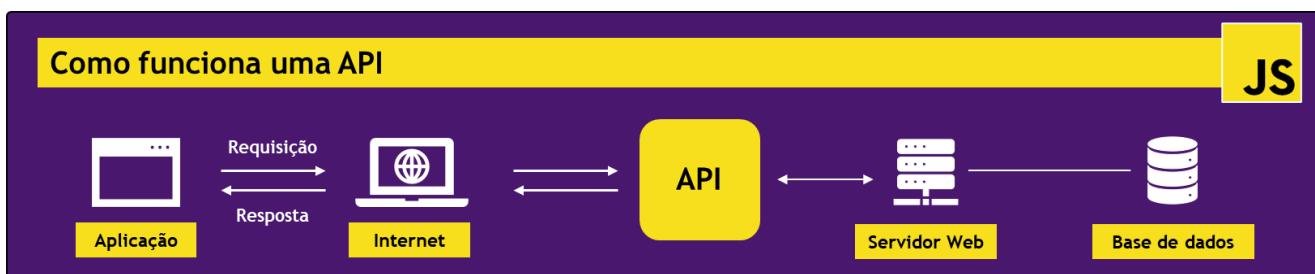
- Exercitar os conhecimentos aprendidos por meio da criação de projetos;
- Compreender a utilização de APIs utilizando Javascript.

Bons estudos!

O que é API?

API é uma abreviação de **Application Programming Interface** (**Interface de Programação de Aplicações**), que é um tipo de interface de software que oferece **serviços** para **outros softwares**. Em outras palavras, podemos dizer que uma **API** é um **software intermediário** que permite **duas aplicações “conversarem” entre si**. Por exemplo, podemos usar uma **API** para **buscar ou gravar informações** em um **banco de dados**, **consultar um número de CEP** para **autocompletar informações** em um formulário, etc.

Vamos imaginar que **abrimos** uma **aplicação** no **navegador web** do nosso **celular** ou **computador** e queremos saber o **endereço** de um **determinado CEP** (**Código de Endereçamento Postal**). Ao **buscar** as **informações** de um **CEP**, a aplicação fará uma **requisição (Request)** de **busca** dos **dados** desse **CEP** através de um **API**. Por sua vez, a **API** **envia a requisição** pra um **servidor web** que **busca** no **banco de dados** as **informações** do **CEP** e **envia a resposta (Response)** para a **API**. Por fim, a **API** retorna com os **dados solicitados** e a aplicação web os **exibe** no **navegador web**.



Funcionamento de uma API.

Importante!

 Você deve se acostumar com os termos em inglês, pois assim fica mais fácil de associar o comando em JavaScript a ação que será executada. Portanto, lembre-se que quando falarmos de **request** é uma **requisição** feita para buscar, alterar ou apagar dados e **response** é a **resposta** recebida após uma **requisição**.

Podemos encontrar **três** tipos de **APIs**: **públicas (public ou open)**, **privadas (private)** ou entre **parceiros (partner)**.

- **APIs públicas** são as mais comuns e disponibilizadas gratuitamente para qualquer empresa e desenvolvedor. Podem existir APIs públicas que limitam o acesso às informações ou a quantidade de buscas de dados, como por exemplo API do **Facebook** (Meta) ou API do **Google Maps**.
- **APIs privadas** são aquelas de **uso exclusivo de uma organização**, geralmente quem tem **acesso** a esse tipo de **API** são os **desenvolvedores de softwares** de uma empresa. Esse tipo de API dá acesso a **sistemas e dados internos de uma empresa**.

- APIs entre parceiros são usadas para **facilitar a integração e comunicação** entre **empresas parceiras**. Por exemplo, uma API entre parceiros pode ser usada para fazer a comunicação entre o **setor de compras de uma empresa** com seus **fornecedores** de **materiais e produtos**.

Métodos HTTP

As **requisições** de uma **API** são geralmente realizadas utilizando os **métodos HTTP**. Os métodos **HTTP** mais **populares** são: **POST**, **GET**, **PUT**, **PATCH** e **DELETE**.

Esses métodos correspondem às operações **CRUD**: **CREATE**, **READ**, **UPDATE**, **UPDATE** e **DELETE**, respectivamente. De modo, temos:

- O método **POST cria (CREATE)** algo na aplicação web. Por exemplo uma instância **registro de dados** em um **banco de dados**.
- O método **GET busca/lê (READ)** uma informação de uma aplicação. Por exemplo, **ler/buscar** um ou mais **registros** (instâncias) no **banco de dados**.
- O método **PUT atualiza (UPDATE)** as informações de uma aplicação. Por exemplo, **atualizar** todas as informações de uma **instância** no **banco de dados**.
- O método **PATCH** atualiza (**UPDATE**) as informações de uma aplicação.

Importante!

 A **diferença** entre o **PUT** e o **PATCH** é que o **primeiro** substitui **todas as informações de um registro** e o **segundo** modifica **uma ou mais informações** que **desejarmos**. Por exemplo, um registro no banco de dados tem três campos: **Nome**, **Telefone** e **data**. Utilizando o método **PUT** devemos enviar os dados dos três campos para **atualizar o registro**, mesmo se quisermos alterar apenas **uma informação**. Já utilizando o método **PUT**, podemos enviar apenas o **valor do campo** a ser alterado em um determinado registro.

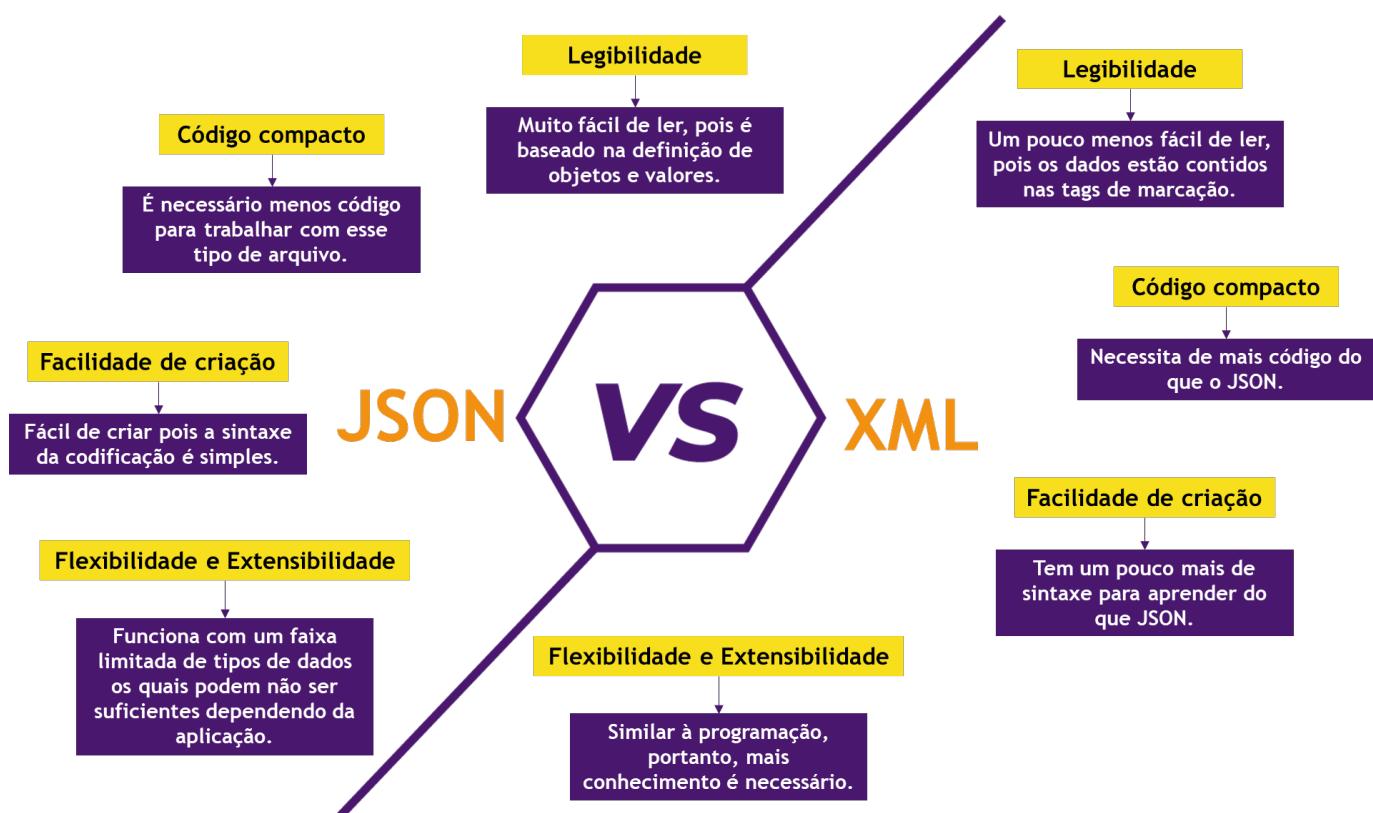
- O método **DELETE apaga (DELETE)** uma informação na aplicação. Por exemplo, **apagar** um **registro** no **banco de dados**.

Importante!

 Aqui demos exemplos de **buscar**, **criar**, **atualizar** e **apagar** registros em um **banco de dados**, mas podemos usar esses métodos para realizar essas operações **CRUD** em arquivos **JSON** ou **TXT** e em qualquer outro tipo de armazenamento de dados.

Response

Sempre que **enviamos** uma **requisição de dados** via **API** para uma aplicação utilizando o **método GET**, a aplicação geralmente envia um **response (resposta)** em formato **JSON** ou **XML**. A tabela abaixo podemos ver algumas das diferenças entre esses dois formatos.



JSON versus XML.

Abaixo, podemos ver as mesmas informações no formato JSON e no formato XML.

JSON

```
{
  "employees": [
    { "firstName": "John", "lastName": "Doe" },
    { "firstName": "Anna", "lastName": "Smith" },
    { "firstName": "Peter", "lastName": "Jones" }
  ]
}
```

XML

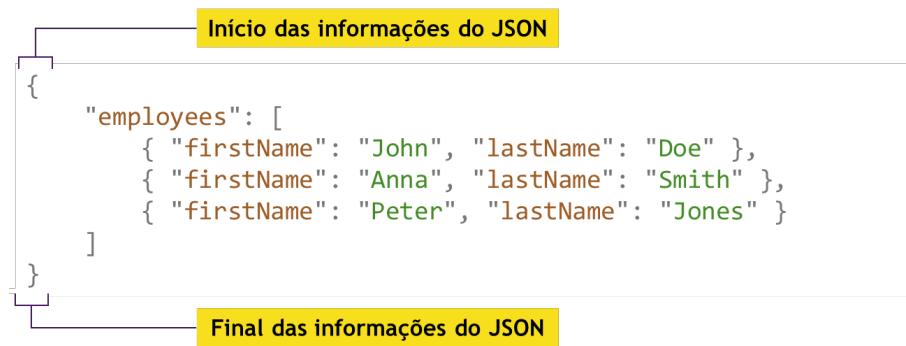
```
<employees>
  <employee>
    <firstName>John</firstName>
    <lastName>Doe</lastName>
```

```

</employee>
<employee>
    <firstName>Anna</firstName>
    <lastName>Smith</lastName>
</employee>
<employee>
    <firstName>Peter</firstName>
    <lastName>Jones</lastName>
</employee>
</employees>

```

As informações retornadas em formato **JSON (JavaScript Object Notation)** são **strings (sequência de caracteres)** escritas entre **aspas ("string")** e essas sempre estão dentro **chaves { }**, como podemos ver abaixo:

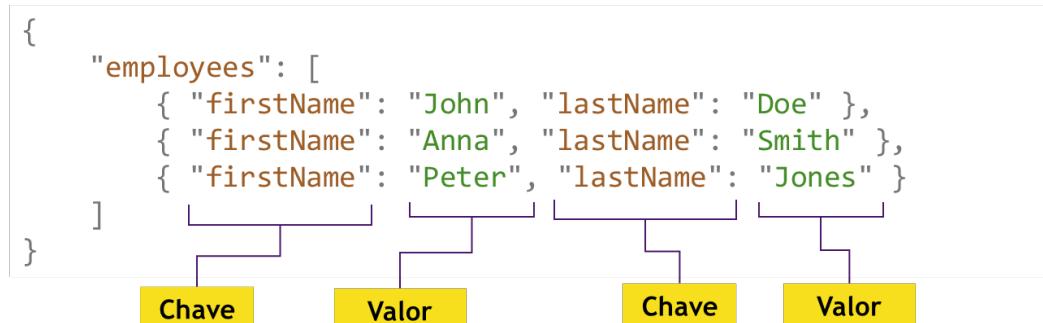


Início e final do JSON.

E são dispostas em **pares de chave-valor**:



Pares chave-valor – Imagem 1.



Pares chave-valor – Imagem 2.



Importante!

O par **chave-valor** são separados por **dois pontos** (**:**) e podemos ter **valores simples** como mostrada na **Imagen 2** ou valor com um **array** com mostrado na **Imagen 1**. Lembre-se: os elementos do **array** são separados por **vírgula** e devem estar **entre chaves** **{ }.**



Exemplos de API

Na visão da pessoa programadora que irá utilizar uma determinada API em sua aplicação, uma **API** é uma simples **URL** com alguns **campos para indicar a informação que você deseja buscar**. Por exemplo, podemos citar a **API** gratuita de **raças de cachorros** do site **Dog.ceo** (<https://dog.ceo/dog-api/>). Podemos utilizar essa **API** buscar diversas informações, por exemplo a **URL**:

<https://dog.ceo/api/breeds/image/random>

Retorna as informações em **JSON** de uma **imagem aleatória** de uma **raça de cachorro armazenadas** em um **banco de dados gratuito**. Você pode **copiar o endereço, colar no seu navegador web** e pressionar **ENTER** que verá uma **imagem aleatória** a cada vez que você **chamar/executar a URL**. No momento que esse material foi preparado os valores retornados foram:

Primeira vez que a URL foi chamada

```
{"message":"https://images.dog.ceo/breeds/spaniel-irish/n02102973\_4328.jpg","status":"success"}
```

Segunda vez que a URL foi chamada

```
{"message":"https://images.dog.ceo/breeds/sharpei/noel.jpg","status":"success"}
```

Observe que a **Response** em **JSON** com **dois pares** de chave-valor: **message** e **status**. O primeiro par tem a chave **message** com o valor sendo a **URL da imagem do cachorro**. O segundo par tem a chave **status** com o valor **success** indicando que a busca foi **realizada com sucesso**.

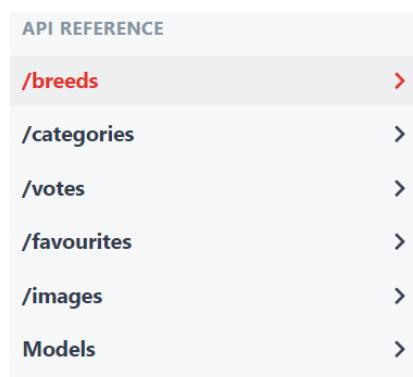
Outra API, **The Cat API** pode ser usada para **buscar imagens aleatórias de gatos no banco de dados**. A **URL** da **API** é:

<https://api.thecatapi.com/v1/images/search>

Ao colocarmos essa **URL** na barra de endereços do navegador web temos uma **response** em **JSON**, como por exemplo:

```
[{"breeds":[],"id":"PMFGf9kGn","url":"https://cdn2.thecatapi.com/images/PMFGf9kGn.jpg","width":700,"height":465}]
```

Note que, essa API **retorna mais informações** do que a anterior. Se visitarmos o site com a documentação da API (<https://docs.thecatapi.com/>), podemos ver que a **API** é mais **completa** e possibilita fazer buscas por **raças (breeds)**, **categorias (categories)**, etc.



API REFERENCE

- /breeds >
- /categories >
- /votes >
- /favourites >
- /images >
- Models >

Outras buscas na *The Cat API*.



Vamos praticar

Vamos brincar um pouco com essas **APIs e implementar** o método **HTTP GET** para **exibir a response** recebida quando enviamos um **request** na **Dog API**. Vamos também aprender a como mostrar esses resultados no nosso **Frontend** e transformar as informações em **JSON** em algo **visual**. Siga os passos para criar o projeto:

Criação do projeto inicial

Abra o **VS Code** e escolha um **diretório de trabalho para o seu projeto**.

Crie um diretório para seu projeto com o nome representativo, por exemplo, **JS_API_Parte_01**

Crie um arquivo dentro do diretório do projeto com o nome **index.html**.

Insira o seguinte código no seu arquivo **index.html**.

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

<head>
    <meta charset="UTF-8" />
```

```
<meta name="viewport" content="width=device-width, initial-scale=1.0" />
<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.2.0-
beta1/dist/css/bootstrap.min.css" rel="stylesheet"
      integrity="sha384-
0evHe/X+R7YkIZDRvuzKMRqM+OrBnVFBL6DOitfPri4tjfHxaWutUpFmBp4vmVor"
      crossorigin="anonymous" />
<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-
awesome/6.1.1/css/all.min.css"
      integrity="sha512-
KfkfwYDsLkIlwQp6LFnl8zNdlLGxu9YAA1QvwINks4PhcElQSvqcyVLLD9aMhXd13uQjoXtEKNosOWaZqXgel0g=="
      crossorigin="anonymous" referrerPolicy="no-referrer" />
<link rel="shortcut icon" href="#" />
<title>API em JavaScript</title>
</head>

<body>
    <section class="container mt-3 p-2 border border-2 rounded-4">
        <h1 class="p-3 mb-2 bg-dark text-white text-center p-1 m-0">Busca de imagens de
        cachorros <i
            class="fa-solid fa-dog pe-3 text-primary"></i></h1>

        <div class="mt-3 mb-3">
            <span id="url_api" class="p-3 mb-2 bg-
light">https://dog.ceo/api/breeds/image/random</span>
            <button type="button" class="btn btn-primary"
            onclick="getImage()">Busca!</button>
        </div>

        <div class="container row">
            <div class="col card">
                <h4>JSON</h4>
                <div id="json_aqui" class="container card-body" style="font-family:
courier;"></div>
            </div>
            <div class="col card text-center">
                <h4>Imagen</h4>
                <div id="imagem_aqui" class="container card-body"></div>
            </div>
        </div>
    </section>

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

</html>
```

No código **HTML** acima, criamos o **layout** para exibir as informações **buscadas na API** em nosso **Frontend**. Devemos ter atenção nos seguintes trechos de código:

```
<div class="mt-3 mb-3">
  <span id="url_api" class="p-3 mb-2 bg-
light">https://dog.ceo/api/breeds/image/random</span>
  <button type="button" class="btn btn-primary" onclick="getImage()">Busca!</button>
</div>
```

Esse primeiro trecho, nós criamos um elemento **** com a **URL** da **API** que iremos chamar e um **botão** para executar a função **getImage()** toda vez que clicarmos nele. Observe que estamos utilizando o evento **onclick** para invocar a função **getImage()**. Essa função será implementada mais adiante no arquivo com o código em **JavaScript**.

O próximo trecho é:

```
<div class="col card">
  <h4>JSON</h4>
  <div id="json_aqui" class="container card-body" style="font-family: courier;"></div>
</div>
```

Que será o local que vamos inserir o conteúdo do **JSON** retornado como **response** da requisição realizada na **API**. Note a **id="json_aqui"** no elemento **<div>**, que será o local usado pelo código **JavaScript** para inserir a **mensagem** de resposta em **JSON** na nossa página web.

O último trecho é:

```
<div class="col card text-center">
  <h4>Imagem</h4>
  <div id="imagem_aqui" class="container card-body"></div>
</div>
```

Esse trecho será o local de **exibição** da **imagem** na página web. Note o elemento **<div>** com a **id="image_aqui"**, esse elemento que será o local que colocaremos a **imagem** no **HTML** pelo código **JavaScript**.

Se clicarmos no botão  **Go Live** da extensão **Live Server**, veremos a nossa página conforme abaixo.



Esse código mostra também a marcação <script> sem nenhum código **JavaScript** entre a **abertura** e o **fechamento** da tag, apenas o atributo **src** com o valor **js/script.js**. Isso significa que o código **JavaScript** está em um **arquivo externo** e está dentro de um diretório chamado **js**. Portanto, temos que criar o diretório **js** dentro da pasta do projeto e o arquivo **script.js** dentro do diretório **js**. Vamos deixar o arquivo **script.js** vazio e à medida que vamos aprendendo coisas novas iremos completar a implementação do código **JavaScript**.

Vamos implementar nosso código em **JS** e aprender como usar uma **API**. Siga os passos para completar o código do JavaScript:

No arquivo **script.js**, insira o seguinte código.

```
// Função para fazer o request na API
function getImage(e) {
    // Busca a url da API no HTML
    let url_api = document.getElementById('url_api').innerText;
    // console.log(url_api);

    // Função fetch para buscar dados na API
    fetch(url_api, {
        method: 'GET',
    })
        .then((response) => {
            return response.json();
        })
        .then((data) => {
            // console.log(data);
            document.getElementById('json_aqui').innerText =
                JSON.stringify(data);
            // console.log(data.message)

            let imagem = `
                
            `;

            // Insere a imagem no elemento HTML com id imagem_aqui
            document.querySelector('#imagem_aqui').innerHTML = imagem;
        });
}
```

Vamos analisar o código por partes:

O código mostrado é a função **getImage()** que é invocada quando clicamos no botão **Busca!** Na nossa página web. A primeira ação da nossa função é **buscar** o **conteúdo** do elemento **** que contém o endereço da **API** que queremos **consultar**.

```
let url_api = document.getElementById('url_api').innerText;
```

Se **retiramos** o **comentário**, podemos ver impresso no **console** o **conteúdo** da variável **url_api** exatamente o endereço da nossa **API** que queremos fazer a **requisição**.



Importante!

Podemos usar a tecla de atalho F12 para abrir o console no navegador web.:

The screenshot shows a Firefox browser window titled "API em JavaScript". The address bar shows the URL "https://dog.ceo/api/breeds/image/random". Below the address bar is a search bar with the placeholder "Busca!". To the right of the search bar is a blue button labeled "Busca!". The main content area has two sections: "JSON" on the left containing the API response object, and "Imagen" on the right showing a picture of a dog. The JSON section contains the following code:

```
{"message": "https://images.dog.ceo/breeds/terrier-welsh/lucy.jpg", "status": "success"}
```

To the right of the browser is the developer tools interface. The "Console" tab is selected. It shows the command "script.js:5:13" and the output "https://dog.ceo/api/breeds/image/random". A yellow callout box points to the output text with the text "Conteúdo da variável url_api".

Podemos ver o método **fetch()**, que é responsável por **iniciar o processo de busca** de recurso na internet. No nosso exemplo, o método **fetch()** utiliza o método **HTTP GET** para requisitar informações da **API indicada** na variável **url_api**.

```
fetch(url_api, {
  method: 'GET',
})
```

O **método GET** pode ser ocultado ao usarmos a função **fetch()** e nesse caso o comando seria apenas:

```
fetch(url_api)
```



Dica!

se você quiser saber mais sobre o método `fetch()` pode consultar a seguinte página do Developer Mozilla disponível em: <https://developer.mozilla.org/pt-BR/docs/Web/API/fetch>

O método **fetch()** é uma maneira mais alto nível de acessar **APIs**, que é um pouco mais alto nível se comparamos com o **HTTPRequest**, que é uma maneira mais baixo nível para brincar com APIs.

Sempre que fazemos uma **requisição**, esperemos uma **resposta** e essa resposta é tratada no método **then()**.

```
.then((response) => {
  return response.json();
})
```

Esse primeiro **método then()** pega a **resposta (response)** retornada pelo **método fetch()**, armazena na variável **response** e retorna esse conteúdo em formato **JSON** através da conversão feita pelo **método json()** na instrução:

```
response.json();
```



Importante!

O nome da variável é personalizado, ou seja, você pode colocar o nome que quiser desde que **respeite as regras de nomes de variáveis**. No nosso exemplo, colocamos o nome **response** para seguir a **convenção ou padrão**.

Por fim, o segundo método **then()**, pega a **resposta** (que é a **response** do **método fetch()** em formato **JSON**) do anterior e **separa as informações** para serem mostradas na página web.

```
.then((data) => {
  // console.log(data);
  document.getElementById('json_aqui').innerText =
    JSON.stringify(data);
  // console.log(data.message)

  let imagem = `
    
  `;

  // Insere a imagem no elemento HTML com id imagem_aqui
  document.querySelector('#imagem_aqui').innerHTML = imagem;
});
```

Utilizamos a função **JSON.stringify()** para **converter os dados** em formato **JSON** em uma **string** que possa ser colocados no texto do elemento HTML (**innerText**) com a **id="json_aqui"**.

Construímos a marcação **HTML** para **exibir uma imagem** utilizando o valor da chave **message** no atributo **scr** do elemento **** (**src="\${data.message}"**).

Inserirmos esse novo elemento dentro do **<div>** com a **id="imagem_aqui"**.

Pronto, nossa aplicação está pronta. Você pode brincar de clicar no botão **Busca!** e ver o JSON e a imagem mudando randomicamente.

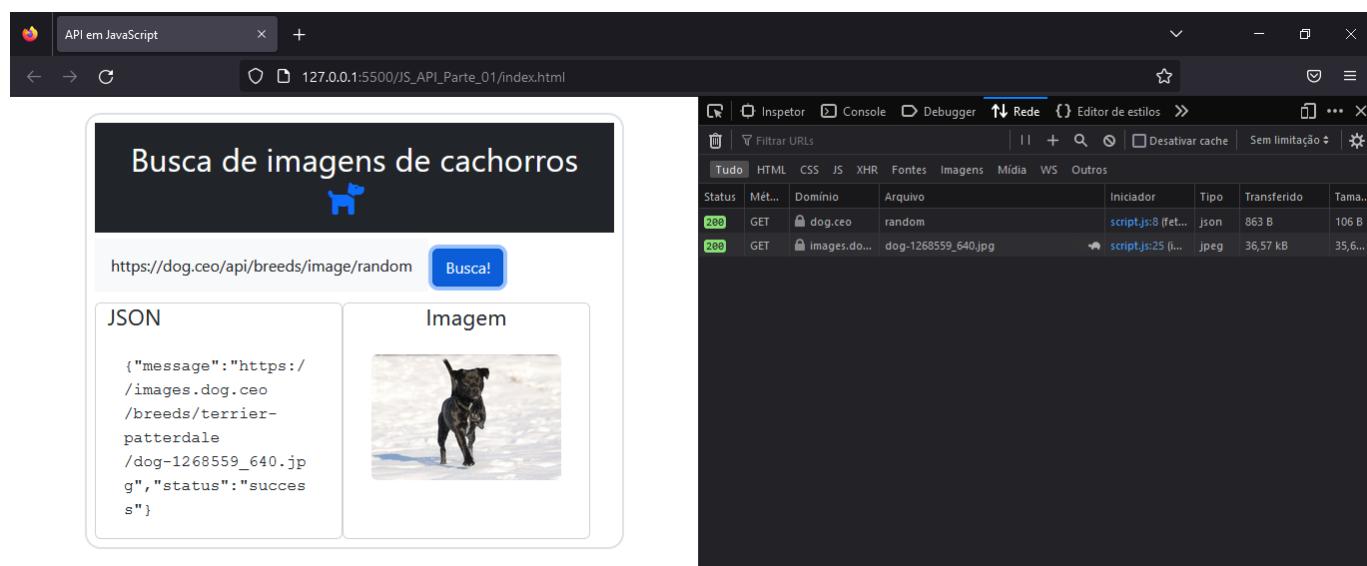
Importante!



Como sabemos que API vai funcionar em toda requisição, não colocamos nenhuma verificação de erro, mas iremos fazer isso no futuro. É importante prever quando erros de requisição para não quebrar a nossa aplicação.

Visualizando os requests no navegador

Podemos visualizar as **requisições** no **navegador web**, para isso basta acessar as ferramentas de desenvolvimento do navegador e selecionar a opção **Rede** (ou **Network**, dependendo do **idioma** do seu navegador).



Status	Mét...	Dominio	Arquivo	Iniciador	Tipo	Transferido	Tama...
200	GET	dog.ceo	random	script.js:8 [fet...	json	863 B	106 B
200	GET	images.d...	dog-1268559_640.jpg	script.js:25 (i...	jpeg	36,57 kB	35,6...

Importante!



Sobre a coluna **Status**, no momento precisamos saber apenas que o **Status 200** significa que a requisição retornou com sucesso. Mais adiante, iremos detalhar os principais códigos de Status em requisições de API.



API com JavaScript - Parte 02

Os objetivos desta aula são:

- Exercitar os conhecimentos aprendidos por meio da criação de projetos;
- Compreender a utilização de APIs utilizando Javascript.

Bons estudos!

The Cat API

No tema passado falamos sobre a **The Cat API** que retorna **informações** sobre **gatos**. Vamos implementar uma aplicação para **exibir** essas **informações dos gatinhos**. Mas antes vamos só relembrar a **URL** da **The Cat API**:

<https://api.thecatapi.com/v1/images/search>

E a **response** em **JSON** com os pares **chave-valor** devemos esperar:

```
[{"breeds":[],"id":"PMFGf9kGn","url":"https://cdn2.thecatapi.com/images/PMFGf9kGn.jpg","width":700,"height":465}]
```



Vamos praticar

Vamos brincar um pouco com essas **APIs** e implementar o método **HTTP GET** para exibir a **response** recebida quando enviamos um **request** na **The Cat API**. Vamos também aprender a como mostrar esses resultados no nosso **Frontend** e transformar as informações em **JSON** em **algo visual**. Siga os passos para criar o projeto:

Criação do projeto inicial

Abra o **VS Code** e escolha um **diretório de trabalho para o seu projeto**.

Crie um diretório para seu projeto com o nome representativo, por exemplo, **JS_API_Parte_02**

Crie um arquivo dentro do diretório do projeto com o nome **index.html**.

Insira o seguinte código no seu arquivo **index.html**.

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

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.2.0-beta1/dist/css/bootstrap.min.css" rel="stylesheet"
          integrity="sha384-0evHe/X+R7YkIZDRvuzKMRqM+OrBnVFBL6D0itfPri4tjfHxaWutUpFmBp4vmVor"
          crossorigin="anonymous">
        <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/6.1.1/css/all.min.css"
              integrity="sha512-KfkfwYDsLkIlwQp6LFn18zNdLGxu9YAA1QvwINks4PhcElQSvqcyVLLD9aMhXd13uQjoXtEKNosOWaZqXgel0g=="
        </head>
```

```

        crossorigin="anonymous" referrerPolicy="no-referrer" />
<link rel="shortcut icon" href="#">
<title>API em JavaScript - Parte II</title>
</head>

<body>
<section class="container mt-3 p-2 border border-2 rounded-4">
    <h1 class="p-3 mb-2 bg-dark text-white text-center p-1 m-0">Busca de imagens de
gatos <i
        class="fa-solid fa-cat pe-3 text-primary"></i></h1>

    <div class="mt-3 mb-3">
        <span id="url_api" class="p-3 mb-2 bg-
light">https://api.thecatapi.com/v1/images/search</span>
        <button type="button" class='btn btn-primary'
onclick="getImage()">Busca!</button>
    </div>

    <div class="container row">
        <div class="col card">
            <h4>Informações</h4>
            <p>ID: <span id="id"></span> </p>
        </div>
        <div class="col card text-center">
            <h4>Imagen</h4>
            <div id="imagem_aqui" class="container card-body"></div>
        </div>
    </div>
</section>

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

</body>

</html>
```

No código **HTML** acima, criamos o layout para **exibir** as **informações** buscadas na **API** em nosso **Frontend**. Devemos ter atenção nos seguintes trechos de código:

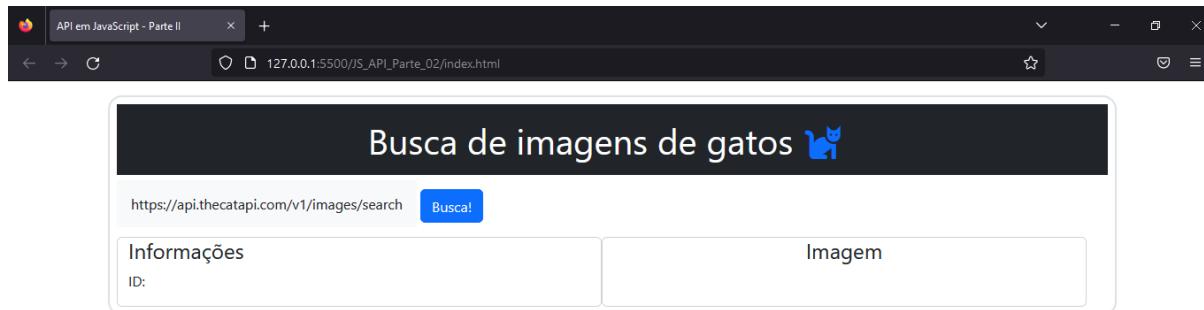
```
<div class="col card">
    <h4>Informações</h4>
    <p>ID: <span id="id"></span> </p>
</div>
```

Esse primeiro trecho, nós criamos um elemento **** com a **id="id"** para colocarmos a **ID** da imagem recebida no **response** da **requisição**.

E o trecho para exibir a imagem na página web. Note o elemento `<div>` com a `id="image_aqui"`, esse elemento que será o local que colocaremos a imagem no **HTML** pelo código **JavaScript**:

```
<div class="col card text-center">
  <h4>Imagen</h4>
  <div id="image_aqui" class="container card-body"></div>
</div>
```

Se clicarmos no botão  **Go Live** da extensão **Live Server**, veremos a nossa página sem nenhuma formatação ainda.



Esse código mostra também a marcação `<script>` sem nenhum código **JavaScript** entre a **abertura** e o **fechamento** da **tag**, apenas o atributo `src` com o valor `js/script.js`. Isso significa que o código **JavaScript** está em um **arquivo externo** e está dentro de um diretório chamado **js**. Portanto, temos que criar o diretório **js** dentro da pasta do projeto e o arquivo **script.js** dentro do diretório **js**.

Vamos deixar o arquivo **script.js** vazio e à medida que vamos aprendendo coisas novas iremos completar a implementação do código **JavaScript**.

No arquivo **script.js**, insira o seguinte código.

```
// Função para fazer o request na API
function getImage(e) {
  // Busca a url da API no HTML
  let url_api = document.getElementById('url_api').innerText;
  console.log(url_api);

  // Função fetch para buscar dados na API
  fetch(url_api, {
    method: 'GET',
  })
    .then((response) => {
      return response.json();
    })
}
```

```

.then((data) => {
  console.log(data);
  document.getElementById('id').innerText = data[0].id;

  let imagem =
    
  ;

  // Insere a imagem no elemento HTML com id imagem_aqui
  document.querySelector('#imagem_aqui').innerHTML = imagem;
});

}

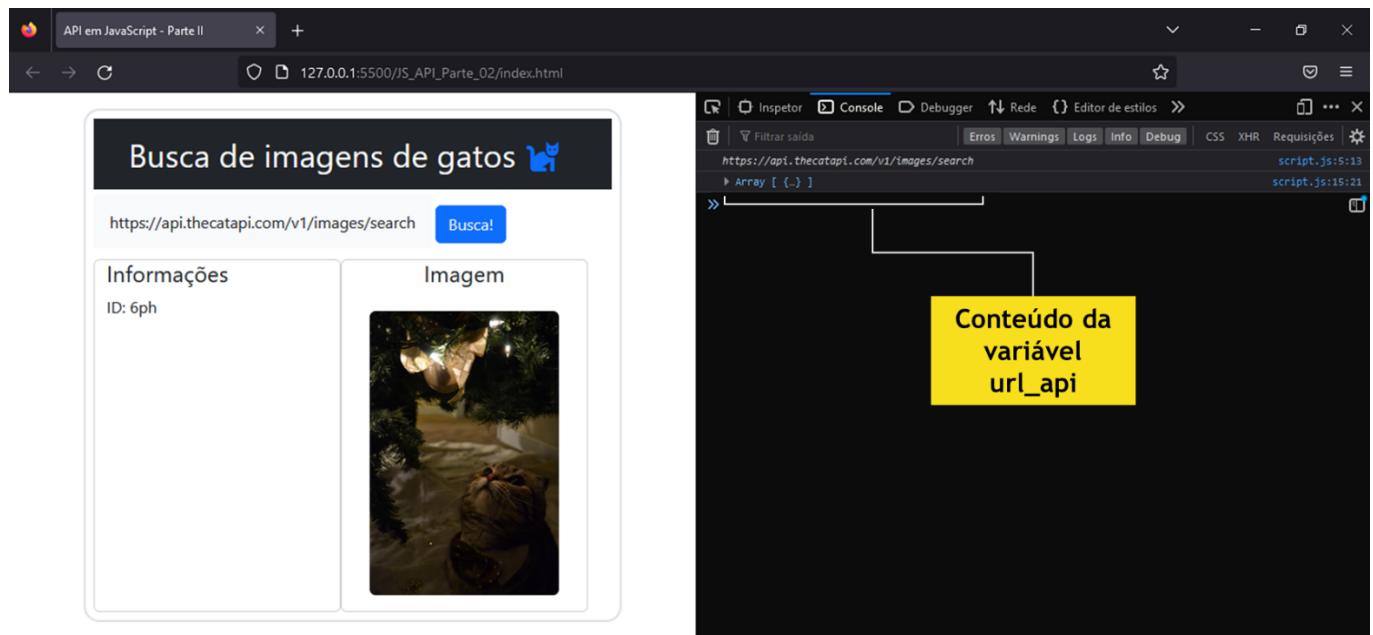
```

Vamos analisar o código por partes:

Da mesma forma que implementamos no código da busca pelos gatos, a primeira ação da função **getImage()** é buscar o conteúdo do elemento **** que contém o **endereço** da **API** que queremos consultar.

```
let url_api = document.getElementById('url_api').innerText;
```

O comando **console.log(url_api);** imprime no console do navegador web o **endereço** da nossa **API**.



The screenshot shows a Firefox browser window titled "API em JavaScript - Parte II". The address bar shows the URL "127.0.0.1:5500/Js_API_Parte_02/index.html". The main content area displays a search interface for "Busca de imagens de gatos" (Cat image search). It shows a list item with "Informações" (ID: 6ph) and an "Imagen" (image of a cat lying under a Christmas tree). The developer tools are open, specifically the "Console" tab. The console output shows the URL "https://api.thecatapi.com/v1/images/search" and an array response: "Array [{ }]". A yellow callout box points to the first element of the array with the text "Conteúdo da variável url_api".

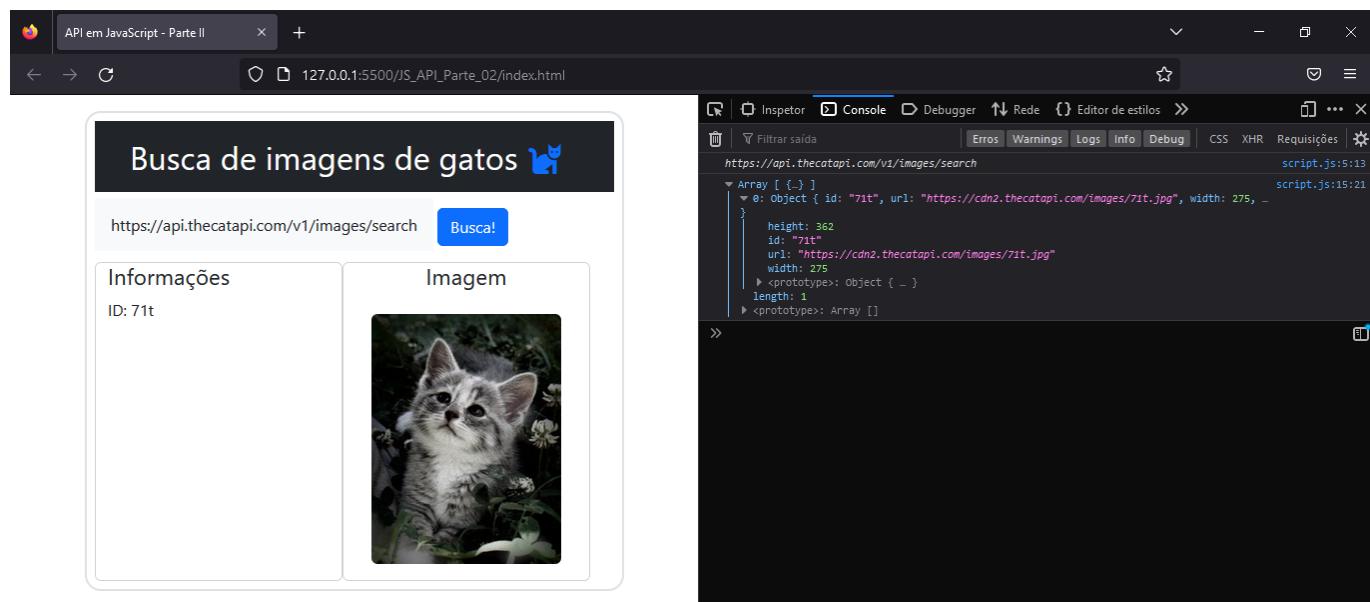
A principal diferença nesse código e no anterior é o **formato** do **dado retornado** após convertermos a **response** do método **fetch()** em **JSON**. No segundo método **then()** dessa função **getImage()**, temos o comando:

```
console.log(data);
```

Podemos ver no **console do navegador** que recebemos como **dados** um **array** com apenas o **primeiro elemento (índice 0)** e as informações **dentro desse elemento**. Por isso, para acessarmos qualquer informação precisamos utilizar o **índice** do **elemento dentro do array** por exemplo:

```
document.getElementById('id').innerText = data[0].id;
```

Vamos ver a diferença: A **API** dos **gatos** retorna uma **response** ao convertêmos, obtemos um **array**.



The screenshot shows a Firefox browser window titled "API em JavaScript - Parte II". The address bar shows the URL "127.0.0.1:5500/JS_API_Parte_02/index.html". The main content area displays a search interface for cat images, with a title "Busca de imagens de gatos" and a search button. Below it, there are two boxes: "Informações" containing the ID "71t" and "Imagen" showing a small thumbnail image of a kitten. To the right of the browser is the Firefox developer tools' "Console" tab, which is active. It shows the URL "https://api.thecatapi.com/v1/images/search" and the script.js:15:13 line of code. The console output is an array with one element, which is an object representing a cat image with properties like id, url, width, height, and a nested object for the image itself.

```
Array [ { } ]
  ↘ 0: Object { id: "71t", url: "https://cdn2.thecatapi.com/images/71t.jpg", width: 275, ... }
    ↘ height: 362
    ↘ id: "71t"
    ↘ url: "https://cdn2.thecatapi.com/images/71t.jpg"
    ↘ width: 275
    ↘ <prototype>: Object { ... }
    ↘ length: 1
    ↘ <prototype>: Array []
```

```

A **API** dos **cachorros** retorna uma **response** que ao convertêmos em obtemos um **objeto**.

Em resumo, essa **diferença** é porque quem desenvolveu a API quis usar **formatações de dados diferentes**. Por isso, é importante **ler a documentação da API**, quando houver, ou usar o comando **console.log** para debugar os valores recebidos e convertidos e, assim, saberemos o tipo de dado recebido e como extrair as informações que desejarmos.



**Busca CEP API**

**Os objetivos desta aula são:**

- Exercitar os conhecimentos aprendidos por meio da criação de projetos;
- Compreender a utilização de APIs utilizando Javascript.

**Bons estudos!**

## API ViaCEP

A **API ViaCEP** permite você consultar as informações de qualquer **Código de Endereçamento Postal (CEP)** válido no Brasil. Esse serviço é **gratuito e sem limitações**. Você pode consultar um **CEP** no formato de **8 dígitos** e as informações da API podem ser retornado nos seguintes formatos: "**json**", "**xml**", "**piped**" ou "**query**". Por exemplo: a URL

[viacep.com.br/ws/01001000/json/](http://viacep.com.br/ws/01001000/json/)

Retorna as seguintes informações no formato **JSON**:

```
{
 "cep": "01001-000",
 "logradouro": "Praça da Sé",
 "complemento": "lado ímpar",
 "bairro": "Sé",
 "localidade": "São Paulo",
 "uf": "SP",
 "ibge": "3550308",
 "gia": "1004",
 "ddd": "11",
 "siafi": "7107"
}
```

No total, temos **10 pares** de **chave-valor** na **response** dessa **API**.

### Importante!



Vamos trabalhar apenas com o formato **JSON**, mas se você quiser conferir os outros formatos pode acessar a documento no site: <https://viacep.com.br/>

Vamos brincar um pouco com essas **APIs e implementar** o método **HTTP GET** para **exibir** a **response** recebida quando **enviamos** um **request** da **ViaCEP**. Vamos também aprender a como mostrar esses resultados no nosso **Frontend** e transformar as informações em **JSON** em algo visual. Siga os passos para criar o projeto:



Vamos praticar

Abra o **VS Code** e escolha um **diretório de trabalho para o seu projeto**.

Crie um diretório para seu projeto com o nome representativo, por exemplo, **JS\_Busca\_CEP**

Crie um arquivo dentro do diretório do projeto com o nome **index.html**.

Insira o seguinte código no seu arquivo **index.html**.

```

<!DOCTYPE html>
<html lang="pt-br">

<head>
 <meta charset="UTF-8">
 <meta name="viewport" content="width=device-width, initial-scale=1.0">
 <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.2.0-
beta1/dist/css/bootstrap.min.css" rel="stylesheet"
 integrity="sha384-
0evHe/X+R7YkIZDRvuzKMRqM+OrBnVFBL6DOitfPri4tjfHxaWutUpFmBp4vmVor"
crossorigin="anonymous">
 <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-
awesome/6.1.1/css/all.min.css"
 integrity="sha512-
KfkfwYDsLkIlwQp6LFnl8zNdLGxu9YAA1QvwINks4PhcElQSvqcyVLLD9aMhXd13uQjoXtEKNosOWaZqXge10g=="
 crossorigin="anonymous" referrerPolicy="no-referrer" />
 <link rel="shortcut icon" href="#">
 <title>Busca CEP</title>
</head>

<body>
 <section class="container mt-3 p-2 border border-2 border-success rounded-4">
 <h1 class="bg-light text-center p-1 m-0"><i class="text-primary fa-solid fa-
globe pe-3"></i>Busca CEP</h1>
 <h2 class="text-center text-secondary p-2">Entre com um número válido de
CEP</h2>
 <form method="get" action=".">
 <div class="container row mb-3">
 <div class="col-2">
 <label class="form-label fw-bold">Cep:</label>
 </div>
 <div class="col-3">
 <input name="cep" type="text" id="cep" value="" size="10"
maxLength="9"
 onblur="pesquisaCEP(this.value);"/>
 </div>
 </div>
 <div class="container row mb-1">
 <div class="col-2">
 <label class="form-label col fw-bold">Rua:</label>
 </div>
 <div class="col-3">
 <input class="col" name="rua" type="text" id="rua" size="45" />
 </div>
 </div>
 <div class="container row">

```

```

<div class="col-2">
 <label class="form-label col fw-bold">Complemento:</label>
</div>
<div class="col-3">
 <input name="complemento" type="text" id="complemento" size="20" />

</div>
</div>

<div class="container row mb-1">
 <div class="col-2">
 <label class="form-label col fw-bold">Bairro:</label>
 </div>
 <div class="col-3">
 <input name="bairro" type="text" id="bairro" size="40" />
 </div>
</div>

<div class="container row mb-1">
 <div class="col-2">
 <label class="form-label col fw-bold">Cidade:</label>
 </div>
 <div class="col-3">
 <input name="cidade" type="text" id="cidade" size="40" />
 </div>
</div>

<div class="container row mb-1">
 <div class="col-2">
 <label class="form-label col fw-bold">Estado:</label>
 </div>
 <div class="col-3">
 <input name="uf" type="text" id="uf" size="2" />
 </div>
</div>

</form>
</section>

<div class="container">
 <div id="output"></div>
</div>

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

</html>

```

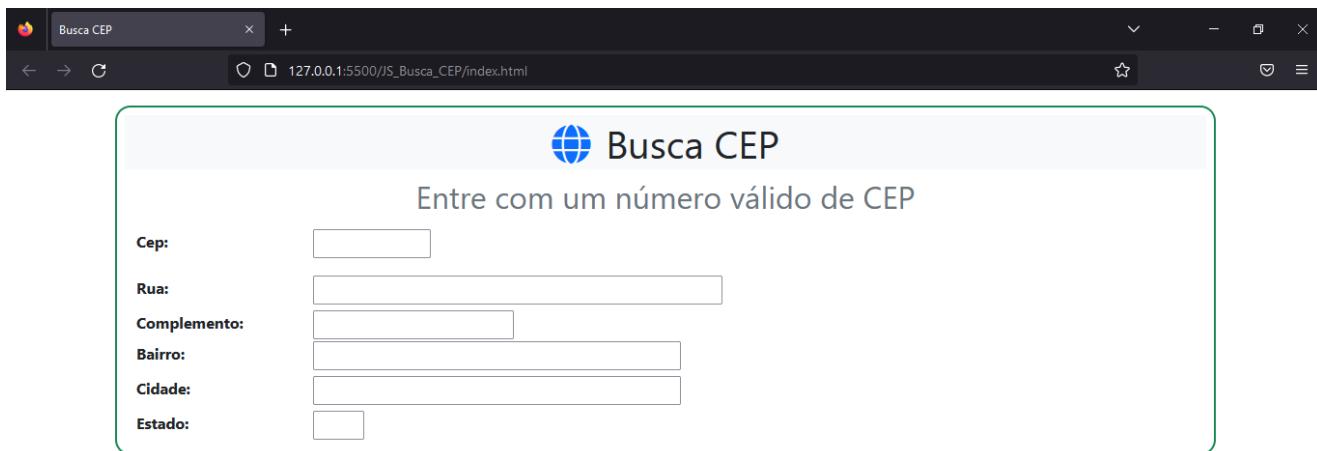
No código **HTML** acima, criamos o **layout** para **exibir as informações buscadas na API** em nosso **Frontend**. Basicamente, o que temos é um **formulário** com diversos **campos** que iremos preencher com nosso código **JavaScript**.

Um trecho que vamos abordar é o código abaixo:

```
<div class="col-3">
 <input name="cep" type="text" id="cep" value="" size="10" maxlength="9"
 onblur="pesquisaCEP(this.value);"/>
</div>
```

Esse primeiro trecho, temos um campo **<input>** do **formulário**, onde vamos inserir o **CEP** a ser **buscado**. O importante a se destacar aqui é a função **pesquisaCEP**, que é invocada quando o evento **onblur** acontece. Esse **evento é disparado** quando o **objeto perde o foco**, ou seja, depois que **digitarmos** o **CEP** e **pressionarmos** a tecla **TAB** ou **clicamos** em outro lugar da página, esse evento **invocará** a função **pesquisaCEP**. Essa função será implementada no nosso arquivo **script.js**.

Se clicarmos no botão  **Go Live** da extensão **Live Server**, veremos a nossa página sem nenhuma formatação ainda.



Esse código mostra também a marcação **<script>** sem nenhum código **JavaScript** entre a **abertura** e o **fechamento** da **tag**, apenas o atributo **src** com o valor **js/script.js**. Isso significa que o código **JavaScript** está em um **arquivo externo** e está dentro de um diretório chamado **js**. Portanto, temos que criar o diretório **js** dentro da pasta do projeto e o arquivo **script.js** dentro do diretório **js**. Vamos deixar o arquivo **script.js** vazio e à medida que vamos aprendendo coisas novas iremos completar a implementação do código **JavaScript**.

Vamos implementar nosso código em **JS** e aprender como usar uma **API**. Siga os passos para completar o código do JavaScript:

No arquivo **script.js**, insira o seguinte código.

```
function limpa_formulário_cep() {
 //Limpa valores do formulário de cep.
 document.getElementById('rua').value = '';
 document.getElementById('bairro').value = '';
 document.getElementById('complemento').value = '';
 document.getElementById('cidade').value = '';
 document.getElementById('uf').value = '';
}
```

A primeira **função** é responsável por **limpar os campos do formulário** e será usada quando quisermos que **novas informações** sejam exibidas ou quando **digitarmos um CEP inválido**. Continuando a implementação do código **JS**. Podemos agora implementar a função **meu\_callback()**.

```
const meu_callback = (conteudo) => {
 console.log(conteudo);
 if (!('erro' in conteudo)) {
 //Atualiza os campos com os valores.
 document.getElementById('rua').value = conteudo.logradouro;
 document.getElementById('bairro').value = conteudo.bairro;
 document.getElementById('complemento').value = conteudo.complemento;
 document.getElementById('cidade').value = conteudo.localidade;
 document.getElementById('uf').value = conteudo.uf;
 } //end if.
 else {
 //CEP não Encontrado.
 limpa_formulário_cep();
 alert('CEP não encontrado.');
 }
};
```

Essa **função** que é **chamada**, logo que **recebemos** uma **resposta da requisição solicitada**. Ela é responsável por **verificar** se a **resposta** (variável **conteudo**) veio **com erro** ou **não**.

```
if (!('erro' in conteudo))
```

Caso a **resposta** seja **válida**, ou seja, a **condição do comando if** foi verdadeira a função irá **preencher** os **campos do formulário** com os **dados** recebido na **resposta** da **requisição**. Por exemplo, a instrução:

```
document.getElementById('rua').value = conteudo.logradouro;
```

Coloca o valor da **chave logradouro** no **campo rua**.

A instrução:

```
document.getElementById('bairro').value = conteudo.bairro;
```

Coloca o valor da **chave bairro** no **campo bairro** e assim até preencher todos os campos do formulário.

Caso a resposta seja **inválida**, as **informações no formulário são limpas** e uma **mensagem de alerta irá aparecer na sua tela**.

```
else {
 //CEP não Encontrado.
 limpa_formulário_cep();
 alert('CEP não encontrado.');//
}
```

Por fim, devemos implementar a função **pesquisaCEP()**.

```
const pesquisaCEP = (valor) => {
 //Nova variável "cep" somente com dígitos.
 let cep = valor.replace(/\D/g, '');

 //Verifica se campo cep possui valor informado.
 if (cep != '') {
 //Expressão regular para validar o CEP.
 let validacep = /^[0-9]{8}$/;

 //Valida o formato do CEP.
 if (validacep.test(cep)) {
 //Preenche os campos com "..." enquanto consulta webservice.
 document.getElementById('rua').value = '...';
 document.getElementById('complemento').value = '...';
 document.getElementById('bairro').value = '...';
 document.getElementById('cidade').value = '...';
 document.getElementById('uf').value = '...';

 //Cria um elemento javascript.
 let script = document.createElement('script');

 //Sincroniza com o callback.
 script.src =
 'https://viacep.com.br/ws/' +
 cep +
 '/json/?callback=meu_callback';

 //Insere script no documento e carrega o conteúdo.
 document.body.appendChild(script);
 }
 }
}
```

```

} //end if.
else {
 //cep é inválido.
 limpa_formulário_cep();
 alert('Formato de CEP inválido.');
}
} //end if.
else {
 //cep sem valor, limpa formulário.
 limpa_formulário_cep();
}
};

}

```

A função **pesquisaCEP()** tem a variável **valor** que contém o **texto digitado** no campo **CEP** do **formulário**. A primeira instrução dessa função:

```
let cep = valor.replace(/\D/g, ''');
```

Ela **substitui** os **valores digitados** que são **texto** por **dígitos**, ou seja, quando digitamos o **CEP 01312001** no campo **CEP** do formulário, esse valor é interpretado como uma **string**. Portanto, precisamos usar o **RegEx \D/g** para converter esse **texto** em **números**.

Em seguida, temos uma declaração **if** para verificar se o **cep** é **diferente de vazio**. Caso esse teste resulte **falso** o código **limpa os campos do formulário**. Caso o teste resulte **verdadeiro**, o código dentro do **if** será executado.

Em seguida, usamos o **RegEx /^[0-9]{8}\$/** para validar o nosso **CEP**. Nesse caso, verificamos se temos **8 dígitos entre 0 e 9**.

Temos o teste para **verificar** se o **CEP** digitado é **válido**. Caso o **CEP** não seja válido o código limpa os **campos do formulário** e exibe a mensagem de alerta do **CEP inválido**. Caso o **CEP seja válido** as instruções seguintes serão executadas.

Logo após, preenchemos os campos do formulário com **três pontos**, exceto o campo do **CEP** que continua com o **valor digitado**.

A instrução:

```
let script = document.createElement('script');
```

Cria um elemento HTML <**script**> e a instrução:

```
script.src =
'https://viacep.com.br/ws/' +
cep +
'/json/?callback=meu_callback'
```

Altera o atributo **src** do elemento `<script>` criado anteriormente com a **URL** da **API**. Essa **URL** é um **pouco diferente**, pois além do **valor** do **CEP** passado pela variável **cep**, ela possui o nome da **função de callback**. A **função de callback** é função **invocada quando recebemos a resposta da requisição feita na API**. No nosso caso, a função de **callback** é a **meu\_callback**, que foi implementada anteriormente.

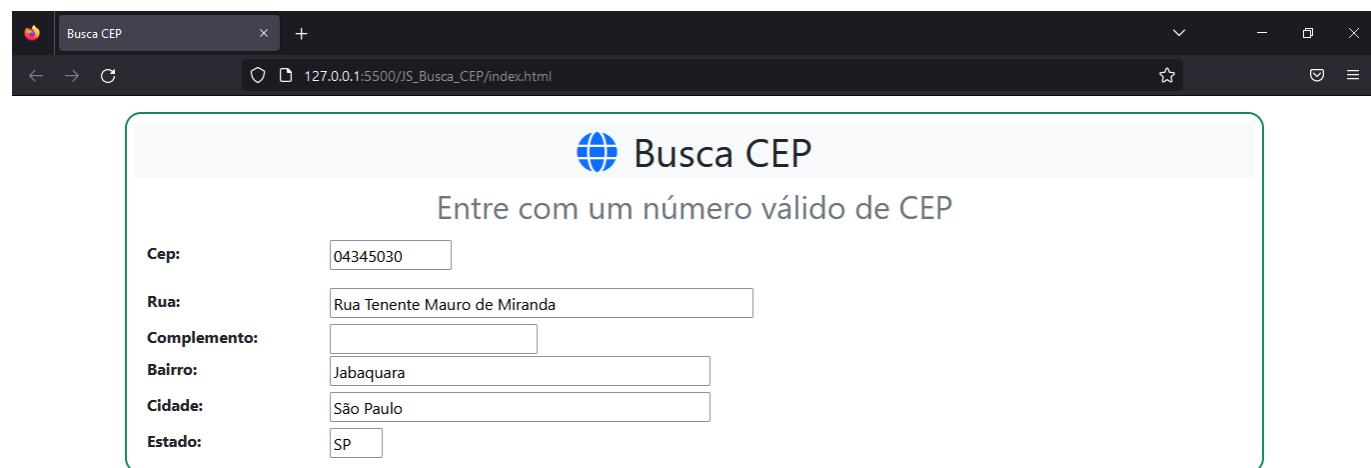
Concluímos com a instrução:

```
document.body.appendChild(script);
```

Que vai inserir o elemento `<script>` criado como **elemento filho** do `<body>` no documento **HTML**, ou seja, **dentro da marcação <body>** no arquivo **index.html** e assim executando o script fazendo a **requisição** para **consultar um determinado CEP digitado** pelo usuário e exibir as informações nos campos dos formulários.

## Testando o nosso projeto

E para finalizar, podemos testar o nosso projeto, você pode digitar um **CEP** a sua escolha e ver os campos do formulário sendo preenchidos como mostrado na imagem a baixa:



Se quiser testar um **CEP inválido**, você verá a **mensagem de alerta** aparecendo na tela:

Busca CEP

Entre com um número válido de CEP

Cep:

Rua:

Complemento:

Bairro:

Cidade:

Estado:

127.0.0.1:5500

Formato de CEP inválido.

Não permitir que 127.0.0.1:5500 peça novamente

OK



## Rick and Morty API

**Os objetivos desta aula são:**

- Exercitar os conhecimentos aprendidos por meio da criação de projetos;
- Compreender a utilização de APIs utilizando Javascript.

**Bons estudos!**

## Rick and Morty API

Nos dois exemplos anteriores, trabalhamos com **APIs simples**, que não tinham muitas opções de **requests**, isto é, a **requisição era sempre a mesma**, pois a **URL da API nunca mudava**. Utilizamos esses exemplos simples para você se **familiarizar** com o **método GET** antes de explorar mais sobre a **busca de dados** em uma **API**.

Agora, vamos ver como funciona a **API do Rick and Morty** que nos permite buscar **diversos tipos de informações e receber diferentes tipos de respostas**. Essa API está disponível no site: <https://rickandmortyapi.com/> e a URL base dela é:

<https://rickandmortyapi.com/api/>

Podemos fazer **diferentes tipos de requisições** nessa **API**. Por exemplo, a **URL** abaixo **requisita a primeira página** com as **informações** dos **personagens** da série:

<https://rickandmortyapi.com/api/character>

Se quisermos **requisitar** as **outras páginas** devemos especificar **explicitamente** a **página desejada**, como mostrado nas seguintes **URLs**:

<https://rickandmortyapi.com/api/character/?page=2> requisita a **segunda página**

<https://rickandmortyapi.com/api/character/?page=18> requisita a **décima oitava página**

Temos no total **42 páginas** de personagens. Existem outras **requisições de personagens** disponíveis que você pode conferir no link:

<https://rickandmortyapi.com/documentation/#character>.

Outro tipo de **requisição disponível** é por **locais** que **apareceram** na **série**. Basicamente, usamos a **URL** mostrada abaixo para **requisitar a primeira página** dos locais da série:

<https://rickandmortyapi.com/api/location>

Também, existem diversas **requisições de locais disponíveis** que você pode conferir no link:

<https://rickandmortyapi.com/documentation/#location>

E o terceiro tipo de **requisição** que podemos fazer é por **episódio**, através da **URL**:

<https://rickandmortyapi.com/documentation/#episode>

Também, existem diversas **requisições de episódios disponíveis** que você pode conferir no link: <https://rickandmortyapi.com/documentation/#episode>.

## Vamos praticar

Abra o **VS Code** e escolha um **diretório de trabalho para o seu projeto**.

Crie um diretório para seu projeto com o nome representativo, por exemplo, **JS\_API\_Rick\_Morty**

Crie um arquivo dentro do diretório do projeto com o nome **index.html**.

Insira o seguinte código no seu arquivo **index.html**.

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

<head>
 <meta charset="UTF-8">
 <meta name="viewport" content="width=device-width, initial-scale=1.0">
 <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.2.0-beta1/dist/css/bootstrap.min.css" rel="stylesheet"
 integrity="sha384-0evHe/X+R7YkIZDRvuzKMRqM+OrBnVFBL6D0itfPri4tjfHxaWutUpFmBp4vmVor"
 crossorigin="anonymous">
 <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/6.1.1/css/all.min.css"
 integrity="sha512-
KfkfwYDsLkIlwQp6LFnl8zNdLGxu9YAA1QvwINks4PhcElQSvqcyVLLD9aMhXd13uQjoXtEKNosOWaZqXgel0g=="
 crossorigin="anonymous" referrerPolicy="no-referrer" />
 <link rel="shortcut icon" href="#">
 <title>API em JavaScript - Ricky and Morty</title>
</head>

<body>
 <section class="container mt-3 p-2 border border-2 rounded-4">
 <h1 class="p-3 mb-2 bg-dark text-white text-center p-1 m-0">Ricky and Morty</h1>

 <div class="mt-3 mb-3">
 <label for="info">Informação:</label>
 <select name="info" id="info">
 <option value="" class="selected">...</option>
 <option value="character">Personagens</option>
 <option value="location">Localização</option>
 <option value="episode">Episódios</option>
 </select>
 <div class="mt-3">
 <label for="opt" id="label" style="display:none;"></label>
 <select name="opt" id="opt" style="display:none;"></select>
 </div>
 </div>
 </section>
</body>
```

```

<button type="button" id="button" class='btn btn-primary mt-3 mb-3' onclick="getData()" disabled>Busca!</button>

<div class="container row">
 <div class="col card">
 <h4>Informações</h4>
 <p>ID: </p>
 <p>Nome: </p>
 <div id="outros"></div>
 </div>
 <div class="col card text-center">
 <h4 id="title_card"></h4>
 <div id="imagem_aqui" class="container card-body"></div>
 </div>
</div>
</section>

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

</body>

</html>

```

No código **HTML** acima, podemos ver um **dropdown menu** onde poderemos escolher os **três tipos de requisições possíveis** para as **requisições na API Ricky and Morty**:

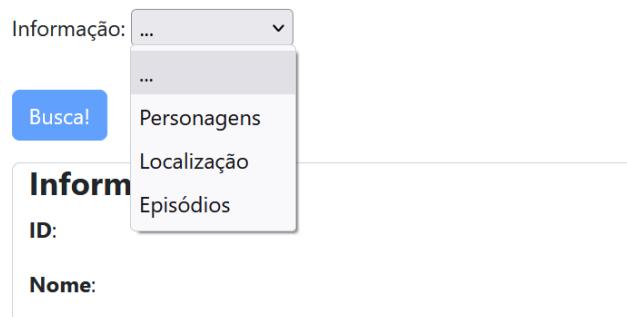
```

<select name="info" id="info">
 <option value="" class="selected">...</option>
 <option value="character">Personagens</option>
 <option value="location">Localização</option>
 <option value="episode">Episódios</option>
</select>

```

As opções são:

- Busca por **Personagens**.
- Busca por **Localização**.
- Busca por **Episódios**.

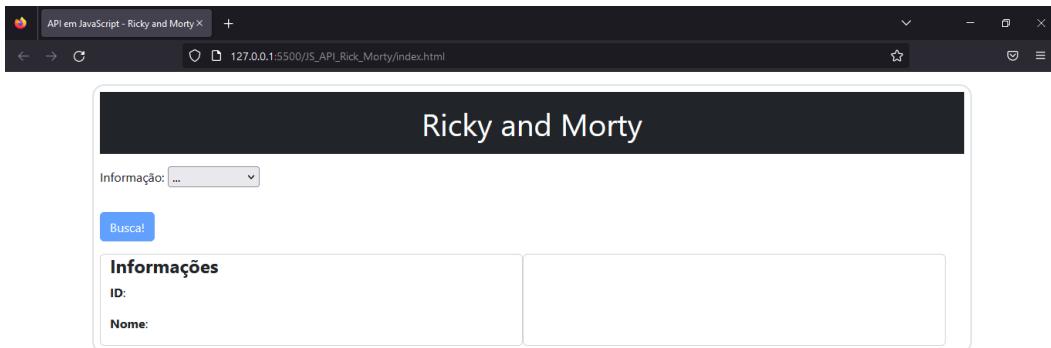


Logo abaixo, do menu **dropdown**, temos o uma outra tag **<select>** que inicialmente **não será exibida**, mas que **aparecerá** assim que selecionarmos **um dos três tipos de requisições disponíveis**. Essa segunda tag **<select>** irá **criar dinamicamente** um segundo menu **dropdown** preenchidos com valores de acordo com a **primeira opção que for selecionada**. Podemos ver o botão com o nome **Buscar**, esse botão será responsável por disparar a requisição na **API** invocando a função **getData()** que implementaremos em **JavaScript**.

```
<button type="button" id="button" class='btn btn-primary mt-3 mb-3' onclick="getData()"
disabled>Busca!</button>
```

Temos a região que **exibiremos** as **informações recebidas** quando fizermos alguma **requisição**.

Se clicarmos no botão  **Go Live** da extensão **Live Server**, veremos a nossa página sem nenhuma formatação ainda.



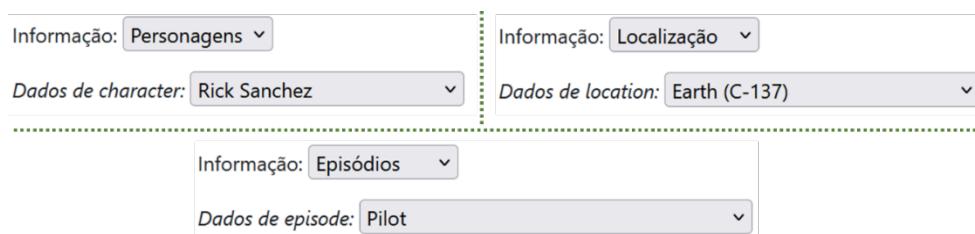
Esse código mostra também a marcação **<script>** sem nenhum código **JavaScript** entre a **abertura** e o **fechamento** da tag, apenas o atributo **src** com o valor **main.js**. Isso significa que o código **JavaScript** está em um **arquivo externo**. Portanto, temos que criar esse novo arquivo **main.js** dentro do diretório do projeto.

Vamos deixar o arquivo **main.js vazio** e à medida que vamos aprendendo coisas novas iremos **completar a implementação do código JavaScript**.

```
let select_opt = document.getElementById('opt');
let label = document.getElementById('label');
let button = document.getElementById('button');
let info = document.getElementById('info');
let outros = document.getElementById('outros');
let title_card = document.getElementById('title_card');
const api = 'https://rickandmortyapi.com/api/';
```

## Vamos analisar o código por partes:

Temos as variáveis **select\_opt** e **label**. A variável **select\_opt** será usada para criar **dinamicamente** o nosso segundo **dropdown** menu de acordo com a **opção de selecionada** na **requisição** e a variável **label** será usada para mostrar a informação contida nesse **segundo dropdown menu**. Se selecionar a opção **Personagens** aparecerá um **label (Dados do character)** e o **dropdown menu** com as **opções de caracteres da série**. Se selecionar a opção **Localização** aparecerá um **label (Dados do location)** e o **dropdown menu** com as opções dos **locais da série**. Se selecionar a opção **Episódios** aparecerá um **label (Dados do episódio)** e o **dropdown menu** com as opções de **episódios da série**.



As outras variáveis são:

- **button** – usada para habilitarmos o botão de busca.
- **info** – usada para atualizar as informações (ID e Nome) do card no HTML
- **outros** – usada para atualizar o elemento <div> no HTML com a ID outros.
- **title\_card** – usada para atualizar o elemento <h4> no HTML com a ID title\_card.
- **api** – usada para armazenada a URL base para API.

Continuando a nossa implementação do código **JavaScript**, vamos inserir o trecho de código mostrado abaixo:

```
document.getElementById('info').addEventListener('change', function (e) {
 let valor = info.value;
 let url_api = api;
 let str_opt = '';

 url_api += ` ${valor}`;

 if (valor != '') {
 fetch(url_api, {
 method: 'GET',
 })
 .then((res) => {
 return res.json();
 })
 .then((dados) => {
 dados.results.forEach((elemento) => {
```

```
 str_opt += `<option value=${elemento.id}>${elemento.name}</option>
 `;
 });
 // Insere as informações no elemento HTML com id dados
 select_opt.innerHTML = str_opt;
 label.innerHTML = `Dados de ${valor}: `;
 label.style.display = 'inline';
 select_opt.style.display = 'inline';
 button.disabled = false;
 });
}
});
```

Esse trecho de código contém a **função** que é **invocada** toda vez que o nosso **primeiro menu dropdown** tem sua **opção selecionada alterada**. O método **addEventListener()** com a opção **change** executa a nossa **função** toda vez que selecionamos uma opção no **menu dropdown**. A variável **valor** busca o conteúdo do atributo **value** da opção selecionada para então completar a URL da requisição:

```
url_api += `${valor}`;
```

Caso o valor **não seja vazio**, ou seja, **alguma opção já foi selecionada**, utilizamos o método **fetch()** para fazer a **primeira request** na API utilizando o **método GET** para preenchermos as informações do **segundo dropdown menu**. Uma vez que a **response da request** é retornada com **sucesso**, **convertemos** essas **informações recebida** em **JSON** e **construímos** o nosso **segundo menu dropdown** com as **informações recebidas**.

Agora, vamos implementar a função `getData()` que fará a requisição completa com as duas opções selecionadas.

```
const getData = () => {
 cleanData();
 let valor = info.value;
 let url_api = api;

 url_api += `#${valor}/${select_opt.value}`;

 fetch(url_api, {
 method: 'GET',
 })
 .then((res) => {
 return res.json();
 })
 .then((dados) => {
```

```

 console.log(dados);
 document.getElementById('id').innerText = dados.id;
 document.getElementById('name').innerText = dados.name;
 chooseData(dados);
 });
}

```

A **getData()** invoca a função **cleanData()** que será responsável por **limpar** algum dado que já esteja sendo **exibido na nossa página**. A função **cleanData()** será **implementada mais adiante**.

Montamos a **URL completa** da **API** de acordo com as **opções selecionadas** nos dois **dropdown menu**.

Com a **URL completa**, fazemos a **request** na **API**. A **response** recebida é **convertida para JSON**. E inserimos as **informações** nos **elementos HTML** da nossa página para exibi-los para o usuário.

Observe que completamos apenas as **informações ID e Nome** e depois invocamos a função **chooseData()**, que implementaremos a seguir:

```

const chooseData = (dados) => {
 if (info.value == 'character') {
 outros.innerHTML = `<p>Espécie: ${dados.species} </p>
 <p>Gênero: ${dados.gender} </p>
 <p>Origem: ${dados.origin.name} </p>
 `;
 title_card.innerHTML = 'Imagen';
 let imagem = `

 `;
 // Insere a imagem no elemento HTML com id imagem_aqui
 document.querySelector('#imagem_aqui').innerHTML = imagem;
 }

 if (info.value == 'location') {
 outros.innerHTML = `<p>Tipo: ${dados.type} </p>
 <p>Dimensão: ${dados.dimension} </p>
 `;
 }

 if (info.value == 'episode') {
 outros.innerHTML = `<p>Data de lançamento: ${dados.air_date} </p>
 <p>Temporada e episódio: ${dados.episode} </p>
 `;
 }
}

```

```
 `;
}
};
```

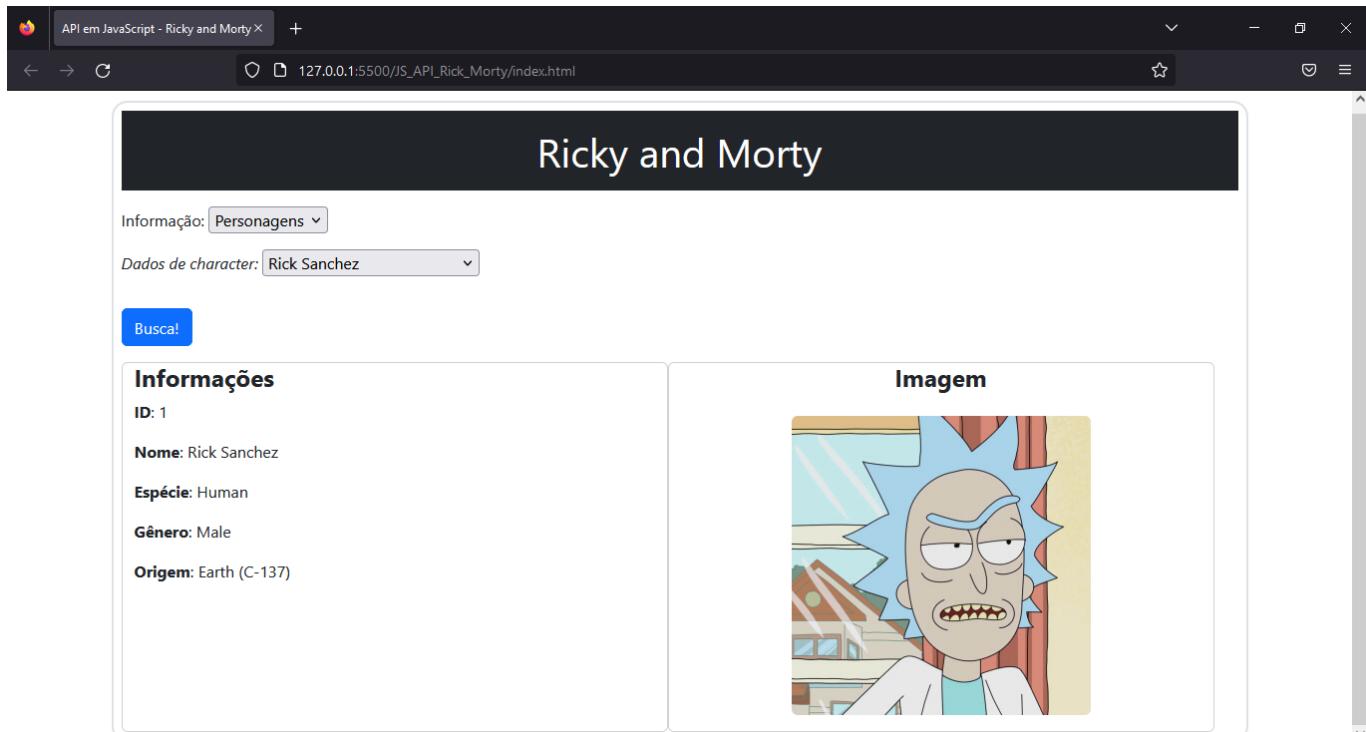
Essa função vai verificar qual foi a **requisição realizada** (**character**, **location** ou **episode**) e completar as informações que serão exibidas no nosso **Frontend** de acordo que a **response recebida**. Isso é necessário, porque cada uma opção de **request** retorna uma **quantidade de informações diferentes**. Por exemplo, a **request** por **character** (**Personagem**) retorna uma **imagem** com a **foto do Personagem** e essa informação **não existe nas outras requisições**.

Por fim, devemos finalizar o nosso código **JavaScript** com a função **cleanData()**.

```
const cleanData = () => {
 outros.innerHTML = '';
 title_card.innerHTML = '';
 document.querySelector('#imagem_aqui').innerHTML = '';
};

};
```

Agora podemos testar o nosso projeto, por exemplo selecionando o personagem **Rick Sanchez**.





## Busca de Estados Brasileiros

**Os objetivos desta aula são:**

- Exercitar os conhecimentos aprendidos por meio da criação de projetos;
- Compreender a utilização de APIs utilizando Javascript.

**Bons estudos!**

## Buscador de Estados

Nessa aula, você aprenderá como fazer um **buscador de estados**, que estarão disponibilizados em um arquivo **JSON**.



Vamos praticar

Abra o **VS Code** e escolha um **diretório de trabalho para o seu projeto**.

Crie um diretório para seu projeto com o nome representativo, por exemplo, **JS\_Buscador\_Estados**.

Crie um arquivo dentro do diretório do projeto com o nome **index.html**.

Insira o seguinte código no seu arquivo **index.html**.

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

<head>
 <meta charset="UTF-8">

 <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap.min.css" rel="stylesheet"
 integrity="sha384-1BmE4kBq78iYHfLdvKuhfTAU6auU8tT94WrHftjDbrCEXSU1oBoqyl2QvZ6jIW3"
 crossorigin="anonymous">

 <title>Busca de Estados Do Brasil</title>
</head>

<body>
 <div class="container mt-5">
 <div class="row">
 <div class="col-md-8 m-auto">
 <h3 class="text-center mb-3">
 Busca de Estados do Brasil
 </h3>

 <form>
 <input class="form-control form-control-lg" id="search" type="text"
 placeholder="Entre com o estado ou abreviação...">
 </form>
 <div id="match-list"></div>
 </div>
 </div>
 </div>
</body>
```

```

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

</html>
```

Nesse código, estamos usando na marcação <link> o **CDN do Bootstrap**. Isso vai permitir utilizarmos as **classes predefinidas** do **Bootstrap** para **estilizar** a nossa **página**.

Esse código mostra também a marcação <script> sem nenhum código **JavaScript** entre a **abertura** e o **fechamento** da tag, apenas o atributo **src** com o valor **main.js**. Isso significa que o código **JavaScript** está em um **arquivo externo**. Portanto, temos que criar esse novo arquivo **main.js** dentro do diretório do projeto.

Vamos deixar o arquivo **main.js vazio** e à medida que vamos aprendendo coisas novas iremos **completar a implementação do código JavaScript**.

Vamos também **criar um diretório** com o nome **dados** e dentro dessa pasta criar o arquivo **estados.json** com o seguinte conteúdo.

```
[
 {
 "abrev": "AC",
 "nome": "Acre",
 "capital": "Rio Branco",
 "area": "152.581",
 "pop": "790.101"
 },
 {
 "abrev": "AL",
 "nome": "Alagoas",
 "capital": "Maceió",
 "area": "27.767,7",
 "pop": "3.120.494"
 },
 {
 "abrev": "AP",
 "nome": "Amapá",
 "capital": "Macapá",
 "area": "142.814,585",
 "pop": "669.526"
 },
 {
 "abrev": "AM",
 "nome": "Amazonas",
 "capital": "Manaus",
 "area": "1.570.745,7",
 "pop": "3.483.985"
 },
]
```

```
{
 "abrev": "BA",
 "nome": "Bahia",
 "capital": "Salvador",
 "area": "565.733",
 "pop": "14.016.906"
},
{
 "abrev": "CE",
 "nome": "Ceará",
 "capital": "Fortaleza",
 "area": "146.348,3",
 "pop": "8.452.381"
},
{
 "abrev": "DF",
 "nome": "Distrito Federal",
 "capital": "",
 "area": "5.802",
 "pop": "2.570.160"
},
{
 "abrev": "ES",
 "nome": "Espírito Santo",
 "capital": "Vitória",
 "area": "46.077,5",
 "pop": "3.514.952"
},
{
 "abrev": "GO",
 "nome": "Goiás",
 "capital": "Goiania",
 "area": "340.086",
 "pop": "6.003.788"
},
{
 "abrev": "MA",
 "nome": "Maranhão",
 "capital": "São Luís",
 "area": "331.983,293",
 "pop": "6.574.789"
},
{
 "abrev": "MT",
 "nome": "Mato Grosso",
 "capital": "Cuiabá",
 "area": "903.357",
 "pop": "3.035.122"
},
```

```
{
 "abrev": "MS",
 "nome": "Mato Grosso do Sul",
 "capital": "Campo Grande",
 "area": "357.124,962",
 "pop": "2.449.024"
},
{
 "abrev": "MG",
 "nome": "Minas Gerais",
 "capital": "Belo Horizonte",
 "area": "586.528,29",
 "pop": "19.597.330"
},
{
 "abrev": "PA",
 "nome": "Pará",
 "capital": "Belém",
 "area": "1.247.689,5",
 "pop": "7.581.051"
},
{
 "abrev": "PB",
 "nome": "Paraíba",
 "capital": "João Pessoa",
 "area": "56.584,6",
 "pop": "3.766.528"
},
{
 "abrev": "PR",
 "nome": "Paraná",
 "capital": "Curitiba",
 "area": "199.298,982",
 "pop": "11.516.840"
},
{
 "abrev": "PE",
 "nome": "Pernambuco",
 "capital": "Recife",
 "area": "98.311,616",
 "pop": "8.796.448"
},
{
 "abrev": "PI",
 "nome": "Piauí",
 "capital": "Teresina",
 "area": "251.529,186",
 "pop": "3.118.360"
},
```

```
{
 "abrev": "RJ",
 "nome": "Rio de Janeiro",
 "capital": "Rio de Janeiro",
 "area": "43.696,1",
 "pop": "16.589.780"
},
{
 "abrev": "RN",
 "nome": "Rio Grande do Norte",
 "capital": "Natal",
 "area": "52.796,791",
 "pop": "3.168.027"
},
{
 "abrev": "RS",
 "nome": "Rio Grande do Sul",
 "capital": "Porto Alegre",
 "area": "281.707,149",
 "pop": "11.422.973"
},
{
 "abrev": "RO",
 "nome": "Rondônia",
 "capital": "Porto Velho",
 "area": "237.576,16",
 "pop": "1.562.409"
},
{
 "abrev": "RR",
 "nome": "Roraima",
 "capital": "Boa Vista",
 "area": "223.644,527",
 "pop": "450.479"
},
{
 "abrev": "SC",
 "nome": "Santa Catarina",
 "capital": "Florianópolis",
 "area": "95.730,684",
 "pop": "7.252.502"
},
{
 "abrev": "SP",
 "nome": "São Paulo",
 "capital": "São Paulo",
 "area": "248.222,8",
 "pop": "41.262.199"
},
```

```
{
 "abrev": "SE",
 "nome": "Sergipe",
 "capital": "Aracaju",
 "area": "21.910,4",
 "pop": "2.068.017"
},
{
 "abrev": "TO",
 "nome": "Tocantins",
 "capital": "Palmas",
 "area": "277.620,91",
 "pop": "1.383.445"
}
]
```

Abra o arquivo index.html, clique no botão  Go Live da extensão Live Server. A página inicial mostrada será:



## Colocando código em JS para realizar a busca de estados

Vamos implementar nosso código em **JS** para ser possível **buscar nomes** ou **siglas dos estados**. Siga os passos para completar o código do **JavaScript**

No arquivo **main.js**, insira o seguinte código.

```
// Busca o elemento com a ID search
const search = document.getElementById('search');
// Busca o elemento com a ID match-list
const matchList = document.getElementById('match-list');
```

Utilizamos o **método getElementById()** para retornar o elemento com **id="search"** na página **HTML** e **referenciá-lo** na variável **search**. Se voltarmos na página web, podemos ver que o elemento **<input>** do **formulário**, que é o **elemento retornado** nessa instrução.

Usamos o método **getElementById()** para retornar o elemento com **id="match-list"** na página **HTML** e se olharmos a página web, o elemento retornado será o elemento **<div>**. Esse é o local que **usaremos** para **exibir o resultado na busca**.

Agora vamos implementar a **função** para **buscar os nomes do estado** de acordo com o **valor digitado** no elemento `<input>` do HTML.

```
/* Função para procurar os estados no arquivo estados.json
e filtra-los para exibir apenas os que correspondem a busca */
const buscarEstados = async (searchText) => {
 const res = await fetch('./dados/estados.json');
 const states = await res.json();
 // console.log(states);

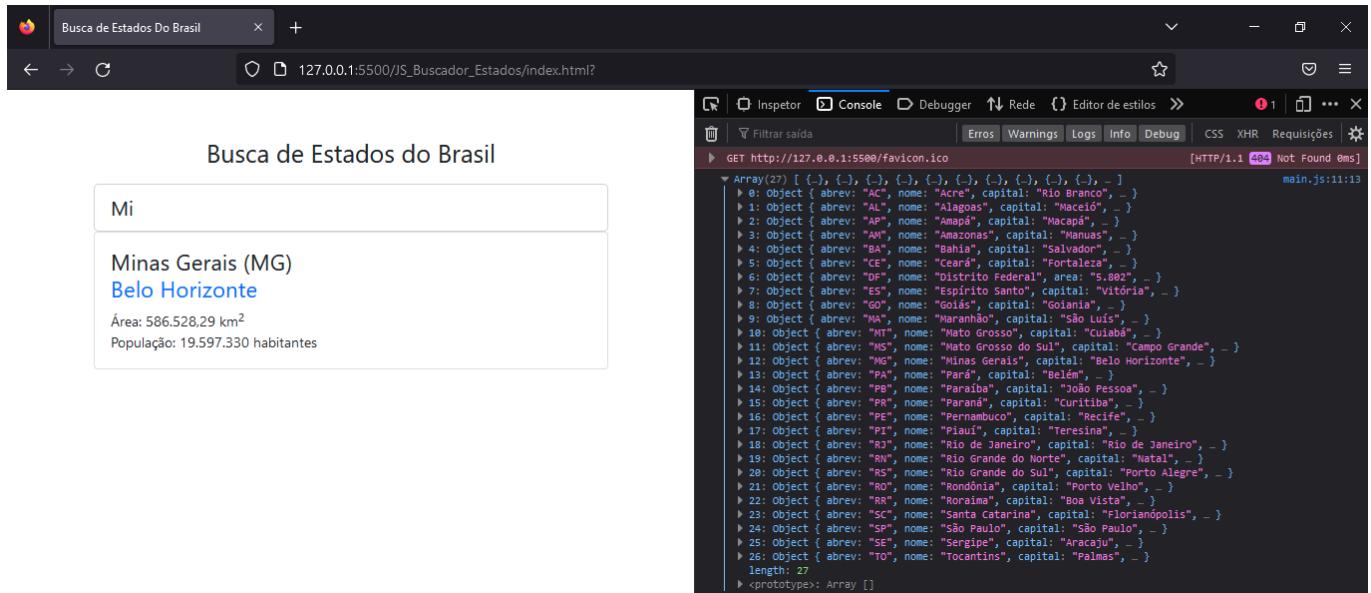
 // Pega os valores correspondentes com o texto digitado
 let matches = states.filter((state) => {
 const regex = new RegExp(`^${searchText}`, 'gi'); // Regular expression
 // Retorna o(s) estado(s) que corresponde ao texto digitado
 return state.nome.match(regex) || state.abrev.match(regex);
 });

 // Limpa a lista de exibição dos estados se o <input> estiver vazio
 if (searchText.length === 0) {
 matches = [];
 matchList.innerHTML = '';
 }
 // console.log(matches);
 saidaHTML(matches); // Invoca a função saidaHTML
};
```

Criamos nossa **função assíncrona** chamada `searchText` para **buscar os estados** que **correspondem ao texto digitado** no elemento `<input>` da página web. A declaração `async nome_da_funcao` define uma **função assíncrona**, que retorna um **objeto AsyncFunction**. A **função assíncrona** retorna uma **Promise** quando ela é chamada. Nesse contexto, quando a **função assíncrona retorna** um **valor**, a **Promise** é resolvida com o **valor retornado** ou a **Promise** pode ser **rejeitada** quando a **função assíncrona levanta alguma exceção**.

A palavra-chave `await` é usada para esperar a **Promise** ser resolvida. Nesse caso, esperamos a **busca do conteúdo** dentro do arquivo `estados.json` que é realizado pela função `fetch()`. E o **valor retornado** é armazenado na **variável res**.

Utilizamos o **método json()**, que retorna uma **Promise** para resolver a criação de um **objeto JavaScript** a partir de um **JSON**. Assim, se você quiser **visualizar o resultado retornado**, basta **descomentar** a instrução, para ver o **valor retornado** impresso no **console do navegador web** em forma de um **objeto do JavaScript** do tipo **array**. O conteúdo exibido será:



```

Array(27) [{} , {}]
 0: Object { abrev: "AC", nome: "Acre", capital: "Rio Branco", ... }
 1: Object { abrev: "AL", nome: "Alagoas", capital: "Maceió", ... }
 2: Object { abrev: "AP", nome: "Amapá", capital: "Macapá", ... }
 3: Object { abrev: "AM", nome: "Amazonas", capital: "Manaus", ... }
 4: Object { abrev: "BA", nome: "Bahia", capital: "Salvador", ... }
 5: Object { abrev: "CE", nome: "Ceará", capital: "Fortaleza", ... }
 6: Object { abrev: "DF", nome: "Distrito Federal", area: "5.802", ... }
 7: Object { abrev: "ES", nome: "Espírito Santo", capital: "Vitória", ... }
 8: Object { abrev: "GO", nome: "Goiás", capital: "Goiânia", ... }
 9: Object { abrev: "MA", nome: "Maranhão", capital: "São Luís", ... }
 10: Object { abrev: "MT", nome: "Mato Grosso", capital: "Cuiabá", ... }
 11: Object { abrev: "MS", nome: "Mato Grosso do Sul", capital: "Campo Grande", ... }
 12: Object { abrev: "NG", nome: "Minas Gerais", capital: "Belo Horizonte", ... }
 13: Object { abrev: "PA", nome: "Pará", capital: "Belém", ... }
 14: Object { abrev: "PB", nome: "Paraíba", capital: "João Pessoa", ... }
 15: Object { abrev: "PR", nome: "Paraná", capital: "Curitiba", ... }
 16: Object { abrev: "PE", nome: "Pernambuco", capital: "Recife", ... }
 17: Object { abrev: "PI", nome: "Piauí", capital: "Teresina", ... }
 18: Object { abrev: "RJ", nome: "Rio de Janeiro", capital: "Rio de Janeiro", ... }
 19: Object { abrev: "RN", nome: "Rio Grande do Norte", capital: "Natal", ... }
 20: Object { abrev: "RS", nome: "Rio Grande do Sul", capital: "Porto Alegre", ... }
 21: Object { abrev: "RO", nome: "Rondônia", capital: "Porto Velho", ... }
 22: Object { abrev: "RR", nome: "Roraima", capital: "Boa Vista", ... }
 23: Object { abrev: "SC", nome: "Santa Catarina", capital: "Florianópolis", ... }
 24: Object { abrev: "SP", nome: "São Paulo", capital: "São Paulo", ... }
 25: Object { abrev: "SE", nome: "Sergipe", capital: "Aracaju", ... }
 26: Object { abrev: "TO", nome: "Tocantins", capital: "Palmas", ... }
length: 27
<prototype>: Array []

```

Utilizamos o método **filter()** para **buscar os estados** que o **nome** ou a **sigla correspondem ao valor digitado** no **elemento <input>** da página web. Criamos a regra de busca utilizando expressões regulares **RegExp**. E, retornamos os valores que **correspondem ao valor buscado** de acordo com a regra do **Regexp**.

Limpamos o conteúdo de elemento **<div>** com **id="matchList"** para ficar vazio o local de exibição do resultado da busca, caso o usuário deixe **vazio** o elemento **<input>** da página web. Por fim, chamamos a função **saídaHTML()**, que será implementada a seguir. Essa função será responsável por **organizar os dados retornados** na busca dos nomes dos estados nos elementos **HTML** para exibi-los corretamente na página web.

Agora vamos implementar a função para **exibir o(s) valor(es) retornados** na nossa página web.

```

// Exibe os resultados no HTML
const saídaHTML = (matches) => {
 if (matches.length > 0) {
 // Constrói o HTML com os valores correspondentes a busca
 const html = matches
 .map(
 (match) =>
 `

${match.nome} (${match.abrev})
 Capital: ${match.capital}</h4> <small>Área: ${match.area} km2</small>
 População: ${match.pop} habitantes</small>

)
 .join('');

 // console.log(html);
}

```

```

 matchList.innerHTML = html; // Insere o conteúdo na página web
 }
};

```

Vamos analisar essa função por partes:

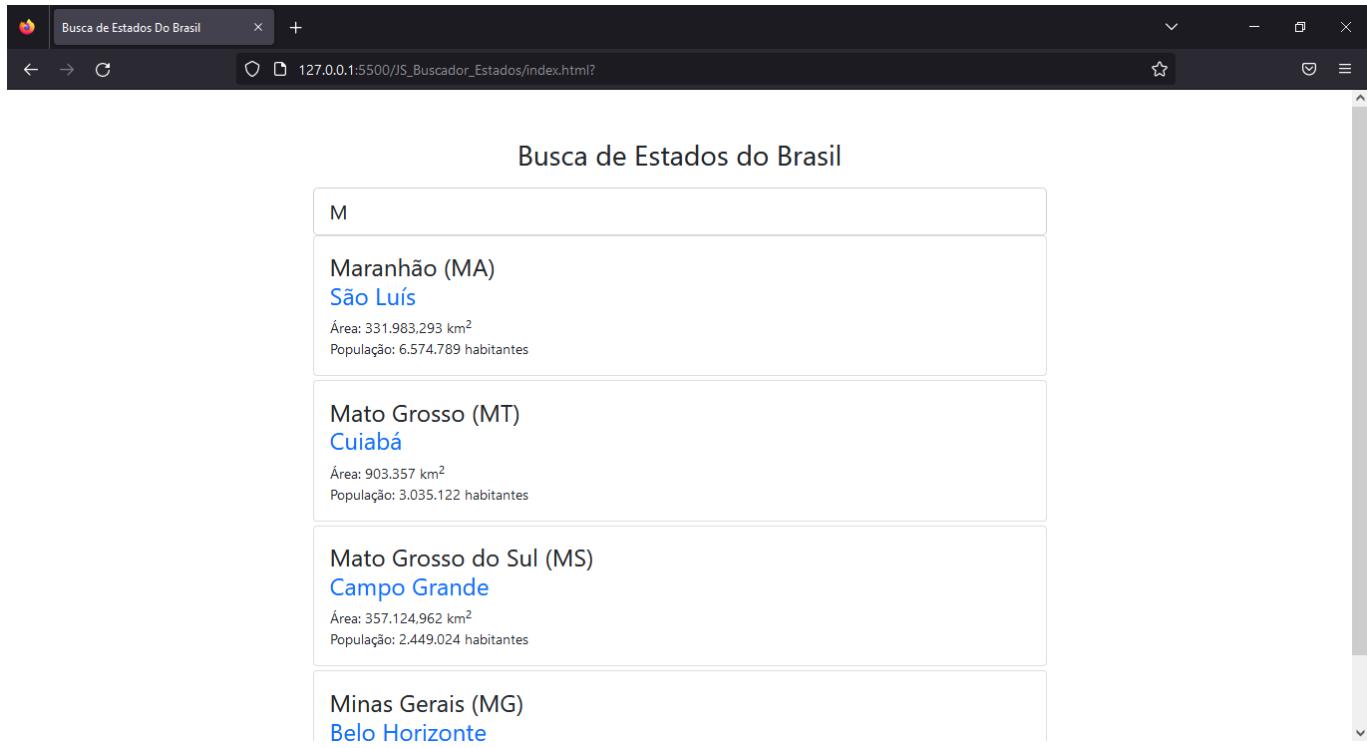
Construímos os elementos **HTML** com os valores corretos **retornados pela busca dos estados**. Inserimos o **conteúdo** em **HTML** na página web, exibindo assim a nossa busca no elemento **<div>** com a **id="match-list"**.

Por fim, devemos adicionar o evento no elemento **<input>** que está vinculado na variável **search**.

```
// Adiciona o evento no elemento <input> da página web
search.addEventListener('input', () => buscarEstados(search.value));
```

Vinculamos a função **buscarEstados()** para ser invocada toda vez que o valor do elemento **<input>** na página web muda.

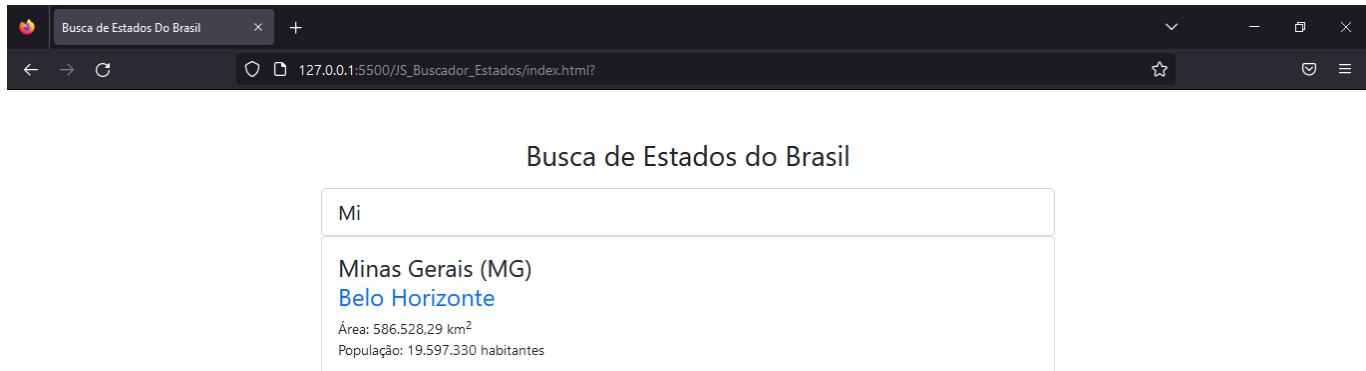
Você pode testar começando a digitar **M** e encontrará **vários estados** com começam com a letra **M**.



The screenshot shows a Firefox browser window with the title "Busca de Estados Do Brasil". The address bar displays "127.0.0.1:5500/JS\_Buscador\_Estados/index.html?". The main content area is titled "Busca de Estados do Brasil" and contains a list of states starting with the letter "M".

- Maranhão (MA)**  
São Luís  
Área: 331.983,293 km<sup>2</sup>  
População: 6.574.789 habitantes
- Mato Grosso (MT)**  
Cuiabá  
Área: 903.357 km<sup>2</sup>  
População: 3.035.122 habitantes
- Mato Grosso do Sul (MS)**  
Campo Grande  
Área: 357.124,962 km<sup>2</sup>  
População: 2.449.024 habitantes
- Minas Gerais (MG)**  
Belo Horizonte

Continuando a digitar **Mi** e encontrará **Minas Gerais**.



Busca de Estados do Brasil

Mi

Minas Gerais (MG)  
[Belo Horizonte](#)  
Área: 586.528,29 km<sup>2</sup>  
População: 19.597.330 habitantes

## Aprenda mais

Se você quer saber mais sobre funções assíncronas, pode consultar os seguintes links e implementar os diversos exemplos disponibilizados nesses sites.

### Funções assíncronas:

[https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Statements/async\\_function](https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Statements/async_function)

### AsyncFunction:

[https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global\\_Objects/AsyncFunction](https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/AsyncFunction)

### Tornando mais fácil a programação assíncrona com async e await:

[https://developer.mozilla.org/pt-BR/docs/Learn/JavaScript/Asynchronous/Async\\_await](https://developer.mozilla.org/pt-BR/docs/Learn/JavaScript/Asynchronous/Async_await)

### Desvendando DEFINITIVAMENTE as Promises em JavaScript:

<https://www.youtube.com/watch?v=nRJhc6vXyK4>

### Async / Await SIMPLES e DESCOMPLICADO no JavaScript:

<https://www.youtube.com/watch?v=h0sNAXE1ozo>

### JavaScript assíncrono: callbacks, promises e async functions:

<https://medium.com/@alcidesqueiroz/javascript-ass%C3%ADncrono-callbacks-promises-e-async-functions-9191b8272298>

**Programação assíncrona em JavaScript com Promises:**

<https://www.devmedia.com.br/programacao-assincrona-em-javascript-com-promises/33184>

**Asynchronous JavaScript:**

[https://www.w3schools.com/js/js\\_asynchronous.asp](https://www.w3schools.com/js/js_asynchronous.asp)



## Criando uma API - Parte 01

**Os objetivos desta aula são:**

- Exercitar os conhecimentos aprendidos por meio da criação de projetos;
- Criar uma API utilizando Javascript.

**Bons estudos!**

## Métodos HTTP

Hoje aprenderemos como **criar** uma **API completa** com os principais **métodos HTTP**.

### **Importante!**



Os métodos *HTTP* mais populares são: *POST*, *GET*, *PUT*, *PATCH* e *DELETE*. Esses métodos correspondem às operações *CRUD*: *CREATE*, *READ*, *UPDATE*, *UPDATE* e *DELETE*, respectivamente. De modo, temos:

- O método **POST cria (CREATE)** algo na aplicação web. Por exemplo uma instância (registro) de dados em um banco de dados.
- O método **GET busca/lê (READ)** uma informação de uma aplicação. Por exemplo, ler/buscar um ou mais registros (instâncias) no banco de dados.
- O método **PUT atualiza (UPDATE)** as informações de uma aplicação. Por exemplo, atualizar todas as informações de uma instância no banco de dados.
- O método **PATCH atualiza (UPDATE)** as informações de uma aplicação.
- O método **DELETE apaga (DELETE)** uma informação na aplicação. Por exemplo, apagar um registro no banco de dados.

## Códigos de Status

Toda **requisição** realizada pelo **cliente** em um **servidor** via **API** retorna um **código de Status** indicando o **tipo de resposta recebida**. Esses códigos são separados pelas seguintes faixas:

- **Respostas de informação (100-199),**
- **Respostas de sucesso (200-299),**
- **Redirecionamentos (300-399)**
- **Erros do cliente (400-499)**
- **Erros do servidor (500-599).**

## Vamos detalhes alguns desses códigos:

### Código: 100

**Continue:** Essa resposta provisória indica que tudo ocorreu bem até agora e que o cliente deve continuar com a requisição ou ignorar se já concluiu o que gostaria.

### Código: 101

**Switching Protocol:** Esse código é enviado em resposta a um cabeçalho de solicitação Upgrade pelo cliente, e indica o protocolo a que o servidor está alternando.

### Código: 200

**OK:** a requisição foi bem-sucedida. O significado do sucesso varia de acordo com o método HTTP.

### Código: 201

**Created:** A requisição foi bem-sucedida e um novo recurso ou nova instância foi criado na base de dados. Esta é uma típica resposta enviada após uma requisição POST.

### Código: 204

**No Content:** Não há conteúdo para enviar para esta solicitação, mas os cabeçalhos podem ser úteis. O user-agent pode atualizar seus cabeçalhos em cache para este recurso com os novos.

### Código: 302

**Found:** Esse código de resposta significa que a URI (Uniform Resource Identifier) do recurso requerido foi mudada temporariamente. Novas mudanças na URI poderão ser feitas no futuro. Portanto, a mesma URI deve ser usada pelo cliente em requisições futuras.

### Código: 304

**Not Modified:** Essa resposta é usada para questões de cache. Diz ao cliente que a resposta não foi modificada. Portanto, o cliente pode usar a mesma versão em cache da resposta.

### Código: 400

**Bad Request:** Essa resposta significa que o servidor não entendeu a requisição pois está com uma sintaxe inválida.

### Código: 401

**Unauthorized:** O cliente deve se autenticar para obter a resposta solicitada.

### Código: 403

**Forbidden:** O cliente não tem direitos de acesso ao conteúdo portanto o servidor está rejeitando dar a resposta. Diferente do código 401, aqui a identidade do cliente é conhecida.

### Código: 404

**Not Found:** O servidor não pode encontrar o recurso solicitado. Este código de resposta talvez seja o mais famoso devido à frequência com que acontece na web.

### Código: 500

**Internal Server Error:** O servidor encontrou uma situação com a qual não sabe lidar.

### Código: 501

**Not Implemented:** O método da requisição não é suportado pelo servidor e não pode ser manipulado. Os únicos métodos exigidos que servidores suportem (e portanto não devem retornar este código) são GET e HEAD.

### Código: 502

**Bad Gateway:** Esta resposta de erro significa que o servidor, ao trabalhar como um gateway a fim de obter uma resposta necessária para manipular a requisição, obteve uma resposta inválida.



#### Dica!

Esse são somente alguns dos códigos de Status que podem ser retornados ao realizarmos uma requisição. Se você quiser pode visitar uma lista mais completa com os códigos de status no link:

<https://developer.mozilla.org/pt-BR/docs/Web/HTTP>Status>



Vamos praticar

Abra o **VS Code** e escolha um **diretório de trabalho para o seu projeto**.

Crie um diretório para seu projeto com o nome representativo, por exemplo, **JS\_API\_Completa**

Crie um arquivo dentro do diretório do projeto com o nome **index.html**.

Insira o seguinte código no seu arquivo **index.html**.

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

<head>
 <meta charset="UTF-8">
 <link rel="shortcut icon" href="#">
 <!-- CSS only -->
 <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.2.0-beta1/dist/css/bootstrap.min.css" rel="stylesheet"
```

```

 integrity="sha384-
0evHe/X+R7YkIZDRvuzKMRqM+OrBnVFBL6D0itfPri4tjfHxaWutUpFmBp4vmVor"
crossorigin="anonymous">

 <title>Lista de usuários do IOS</title>
</head>

<body>
 <div class="container mt-5">
 <div class="row mt-3">
 <div class="col-md-6">
 <h3 class="text-center mb-3">
 Cadastrar usuários
 </h3>
 <form id="add-user-form" class="form-control">
 <div class="mb-3">
 <label for="name" class="form-label">Nome</label>
 <input type="text" class="form-control" id="name"
placeholder="Nome do usuário">
 </div>
 <div class="mb-3">
 <label for="e-mail" class="form-label">E-mail</label>
 <input type="email" class="form-control" id="e-mail"
placeholder="email@example.com.br">
 </div>
 <button type="submit" class="btn btn-primary mb-3">Adicionar</button>
 </form>
 </div>
 </div>
 </div>

 <div id="users-list" class="row mt-5">
 <!-- Aqui é o local que os dados dos usuários serão inseridos -->
 </div>

</div>

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

</html>
</pre>

```

Nesse código, estamos usando na marcação `<link>` o **CDN** do **Bootstrap**. Isso vai permitir utilizarmos as **classes predefinidas** do **Bootstrap** para estilizar a nossa página.

Esse código mostra também a marcação `<script>` sem nenhum código **JavaScript** entre a **abertura** e o **fechamento** da tag, apenas o atributo `src` com o valor **main.js**. Isso significa que

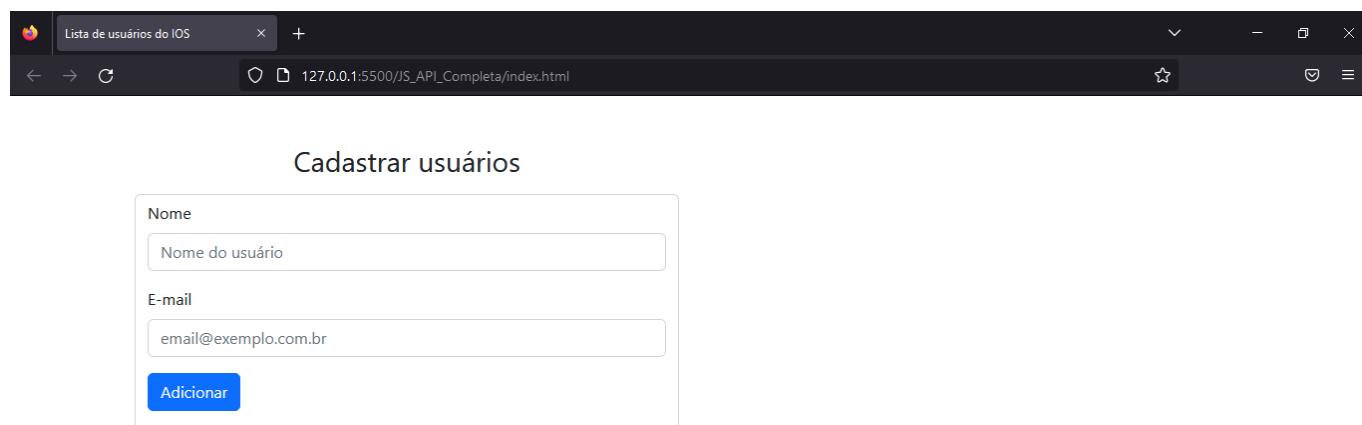
o código **JavaScript** está em um **arquivo externo**. Portanto, temos que criar esse novo arquivo **main.js** dentro do diretório do projeto.

Vamos deixar o arquivo **main.js vazio** e à medida que vamos aprendendo coisas novas iremos **completar a implementação do código JavaScript**.

Vamos também **criar um diretório** com o nome **dados** e dentro dessa pasta **criar** o arquivo **db.json** com as informações de **três usuários fictícios**.

```
{
 "users": [
 {
 "id": 1,
 "name": "Lucas Graham",
 "email": "l.graham@hotmail.com"
 },
 {
 "id": 2,
 "name": "Everaldo Silva",
 "email": "e.silva@gmail.com"
 },
 {
 "id": 3,
 "name": "Claudio Borges",
 "email": "c.borges@gmail.com"
 }
]
}
```

Abra o arquivo `index.html`, clique no botão  **Go Live** da extensão **Live Server**. A página inicial mostrada será:



Cadastrar usuários

Nome  
Nome do usuário

E-mail  
email@example.com.br

Adicionar

## Criando o servidor para acessar os dados

Usar arquivos comuns no nosso computador para acessar é **interessante** mas possui **diversas limitações**. Por exemplo, nas aulas anteriores usamos um arquivo **JSON** para **buscar** os **dados** e **exibi-los** no nosso **Frontend** utilizando o **método HTTP GET**. Tudo funcionou bem, mas **não podemos fazer nada** além de **ler dados desse arquivo**, se quisermos **criar**, **atualizar** ou **apagar** qualquer informação nesse arquivo precisamos de **criar** ou **simular** um **servidor web** para realizar as outras **operações CRUD** com os métodos **HTTP POST**, **PUT/PATCH** ou **DELETE**.

Por isso, vamos usar o **pacote/módulo json-server** que é uma ferramenta muito útil para **simularmos um banco de dados** utilizando um **arquivo JSON** e assim poderemos **ler**, **criar**, **atualizar** e **apagar** dados **desse arquivo realizando as operações CRUD** através dos métodos **HTTP**.

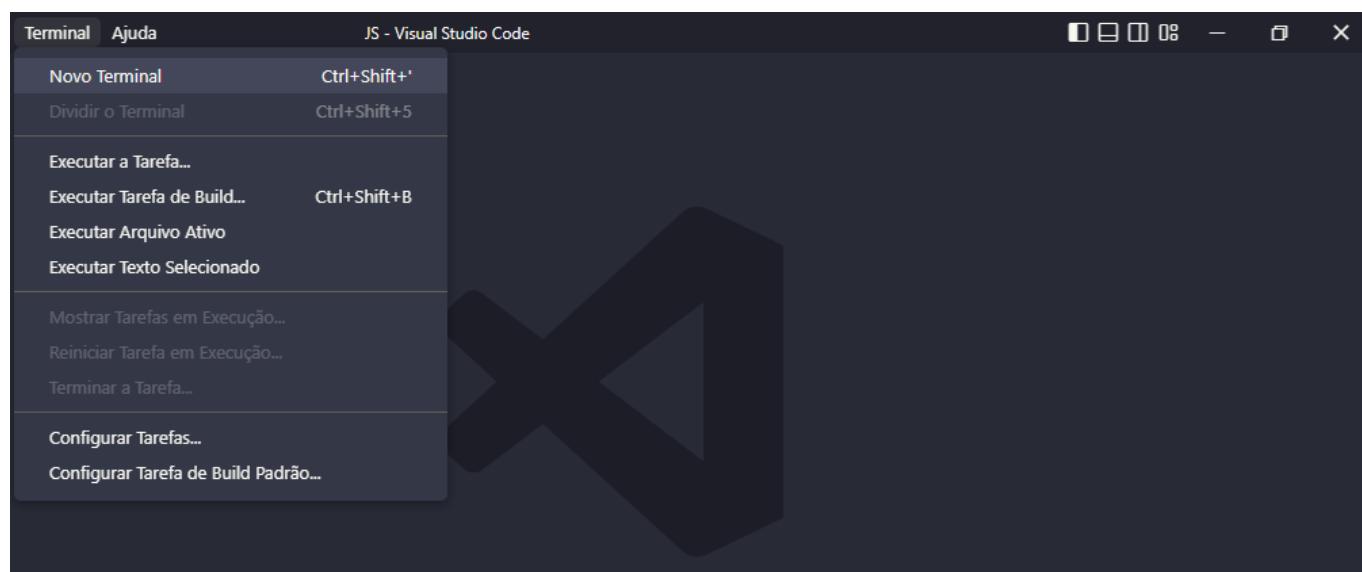
### **Importante!**



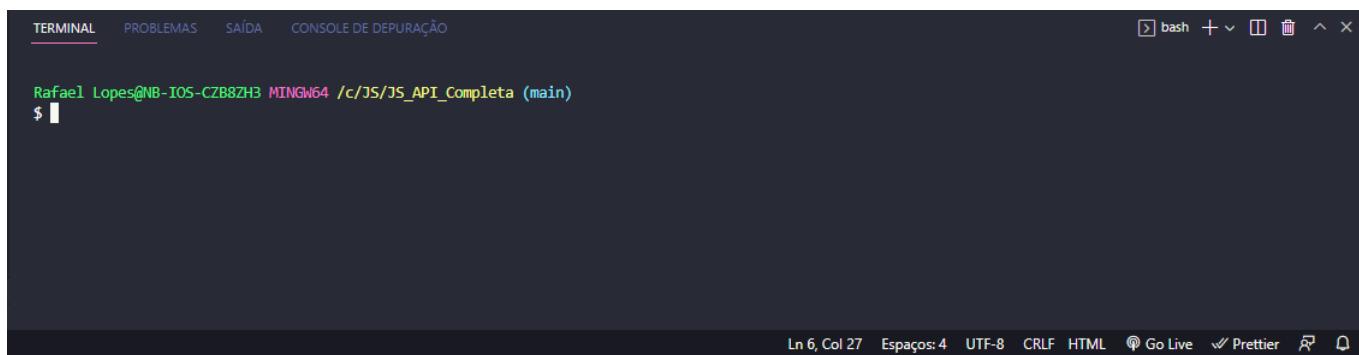
Se você quiser saber mais sobre o módulo **json-server** você pode acessar o **repositório oficial no GitHub** disponível no link: <https://github.com/typicode/json-server>

Antes de instalar o módulo **json-server**, você precisa ter o software **Node.js** instalado no seu computador. Precisamos do **Node.js** para utilizarmos os gerenciadores pacotes (**bibliotecas**) **NPM (Node Package Manager)** e **NPX (Node Package Execute)** e, também, para executarmos os códigos **JavaScript** necessários para **rodar o nosso servidor**. O software **Node.js** está disponível para download em: <https://nodejs.org/pt-br/>.

Abra o **terminal no VS Code** utilizando as teclas de atalho **Ctrl + `** ou **Ctrl + Shift + `** ou acesso o menu **Terminal** e escolhendo a opção **New Terminal**.



Com o **terminal aberto**, tenha certeza de que você está **dentro do diretório do seu projeto**. Por exemplo, eu estou desenvolvendo essa aplicação no repositório local do meu GitHub pessoal chamado **JS** e o projeto está dentro do diretório **JS\_API\_Completa**. Esse passo é importante, porque vamos **instalar o módulo json-server localmente no diretório atual de trabalho**.



A screenshot of a terminal window. At the top, there are tabs labeled 'TERMINAL', 'PROBLEMAS', 'SAÍDA', and 'CONSOLE DE DEPURAÇÃO'. On the right side of the window, there are icons for bash, a plus sign, a minus sign, a square, a trash can, an upward arrow, and a downward arrow. The main area shows the command line: 'Rafael Lopes@NB-IOS-CZB8ZH3 MINGW64 /c/JS/JS\_API\_Completa (main)' followed by a '\$' prompt. Below the command line, there is a large empty space. At the bottom of the terminal window, there are status indicators: 'Ln 6, Col 27', 'Espaços: 4', 'UTF-8', 'CRLF', 'HTML', 'Go Live', 'Prettier', and a refresh icon.

Após ter certeza de que você está no **diretório local correto**, o primeiro passo é **digitar o comando**:

```
npm init -y
```

Esse comando irá **criar um arquivo** chamado **package.json** com as **informações do projeto**. Se você abrir esse arquivo verá as **informações criadas automaticamente** nele.



A screenshot of a code editor showing the 'package.json' file. The file contains the following JSON code:

```
1 {
2 "name": "js_api_completa",
3 "version": "1.0.0",
4 "description": "",
5 "main": "index.js",
6 "scripts": {
7 "test": "echo \\\"Error: no test specified\\\" && exit 1"
8 },
9 "keywords": [],
10 "author": "",
11 "license": "ISC"
12 }
```

Vamos continuar a nossa **instalação** do **módulo json-server** e depois voltaremos no arquivo **package.json**. Portanto, ainda no **terminal de comando**, vamos **digitar o comando** abaixo para instalar o pacote **json-server**:

```
npm install json-server
```

TERMINAL PROBLEMAS SAÍDA CONSOLE DE DEPURAÇÃO

```
Rafael Lopes@NB-IOS-CZB8ZH3 MINGW64 /c/JS/JS_API_Completa (main)
$ npm install json-server

added 109 packages, and audited 110 packages in 11s

10 packages are looking for funding
 run `npm fund` for details

found 0 vulnerabilities
```

### **Importante!**

 Você também pode instalar uma versão específica do módulo utilizando o comando:  
**npm install json-server@version\_number**

Por exemplo, estamos utilizando no momento a versão 0.17.0, então se quisermos instalar essa versão devemos colocar a seguinte instrução.

**npm install json-server@0.17.0**

Esse comando irá **instalar localmente o módulo** e criar uma **pasta** com o nome **node\_modules** no nosso diretório.



Agora podemos **voltar** no nosso arquivo **package.json** e **inserir** a **instrução de configuração** do nosso **servidor** dentro da **secção scripts**. A instrução é:

```
"server": "json-server --watch ./dados/db.json --port 5000"
```

Veja que ao inserir o comando, surge um erro:



```

1 {
2 "name": "js_api_completa",
3 "version": "1.0.0",
4 "description": "",
5 "main": "index.js",
6 > Depurar
7 "scripts": {
8 "test": "echo \"Error: no test specified\" && exit 1"
9 "server": "json-server --watch ./dados/db.json --port 5000"
10 },
11 "keywords": [],
12 "author": "",
13 "license": "ISC",
14 "dependencies": {
15 "json-server": "^0.17.1"
16 }
17 }

```

Isso ocorre, pois **não separamos as instruções**, insira uma **virgula (,)** no **final do comando anterior** para corrigir o problema:



```

1 {
2 "name": "js_api_completa",
3 "version": "1.0.0",
4 "description": "",
5 "main": "index.js",
6 > Depurar
7 "scripts": {
8 "test": "echo \"Error: no test specified\" && exit 1",
9 "server": "json-server --watch ./dados/db.json --port 5000"
10 },
11 "keywords": [],
12 "author": "",
13 "license": "ISC",
14 "dependencies": {
15 "json-server": "^0.17.1"
16 }
17 }

```

Voltando ao comando anterior:

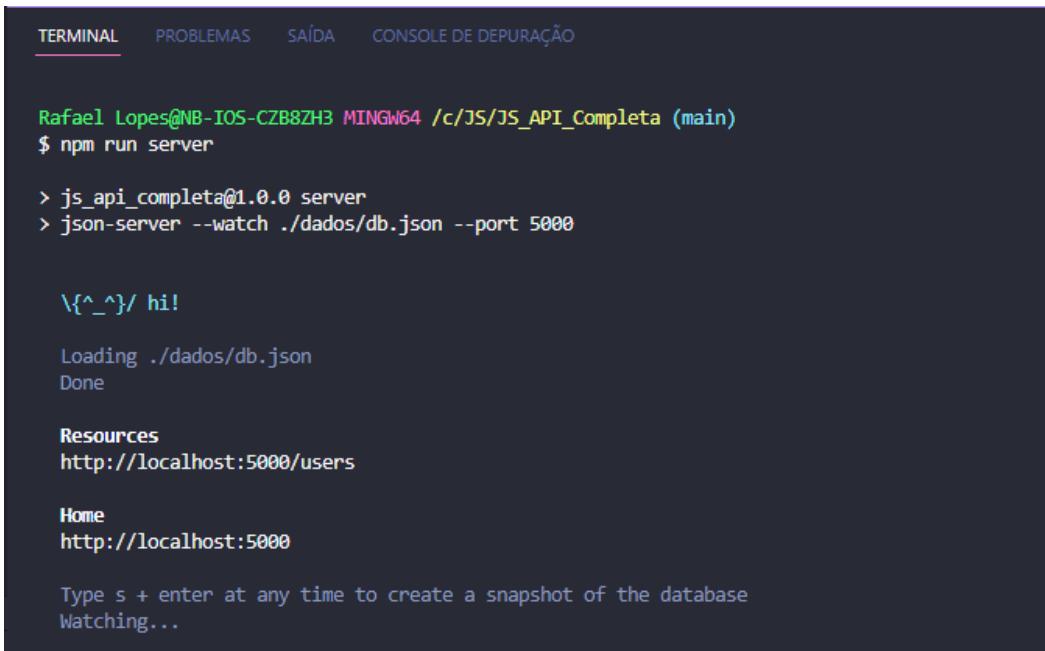
```
"server": "json-server --watch ./dados/db.json --port 5000"
```

Onde:

- **json-server** é nome do **módulo** que estamos utilizando
- **--watch ./dados/jd/json** indica o **caminho** do nosso **arquivo JSON** no nosso projeto
- **-- port 5000** configura a **porta local do nosso servidor**. Por **padrão**, a **porta local** é a **3000**, mas modicamos para **5000**, para não ter **conflito com outros módulos** do **npm** como por exemplo do **REACT**.

Agora podemos inicializar o nosso servidor com o comando:

```
npm run server
```



```

TERMINAL PROBLEMAS SAÍDA CONSOLE DE DEPURAÇÃO

Rafael Lopes@NB-IOS-CZB8ZH3 MINGW64 /c/JS/JS_API_Completa (main)
$ npm run server

> js_api_completa@1.0.0 server
> json-server --watch ./dados/db.json --port 5000

\{^_}\ hi!

Loading ./dados/db.json
Done

Resources
http://localhost:5000/users

Home
http://localhost:5000

Type s + enter at any time to create a snapshot of the database
Watching...

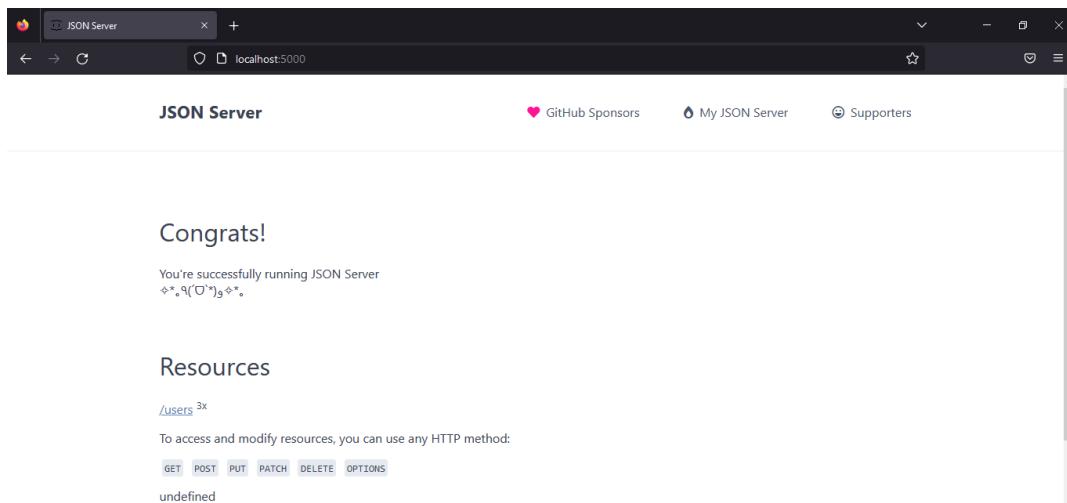
```

### **Importante!**

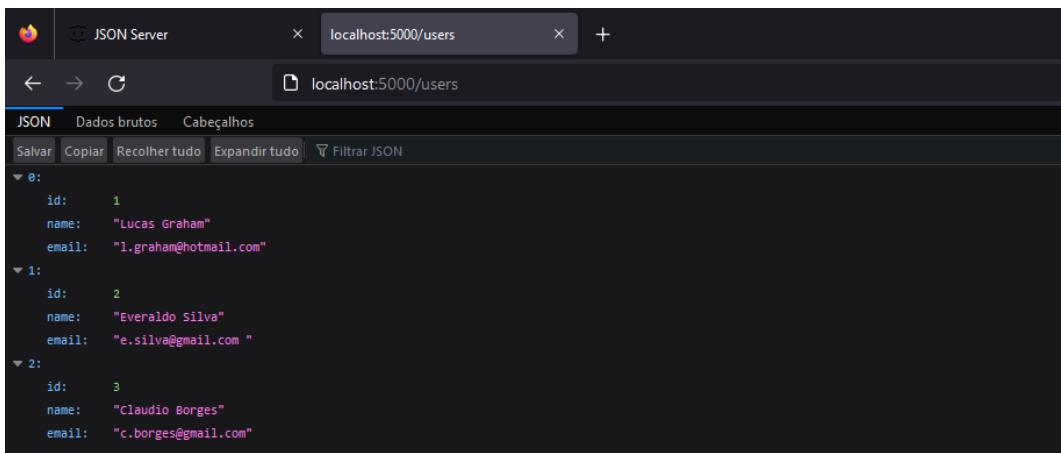


**Não fechar ou finalize o servidor** durante a **execução da nossa aplicação**, pois se não ela não irá funcionar.

Se tudo estiver configurado corretamente o seu servidor será **inicializado corretamente** e você pode abrir o seu **navegador web** e digitar a **URL** (<http://localhost:5000>) para confirmar que o **servidor está funcionando**.



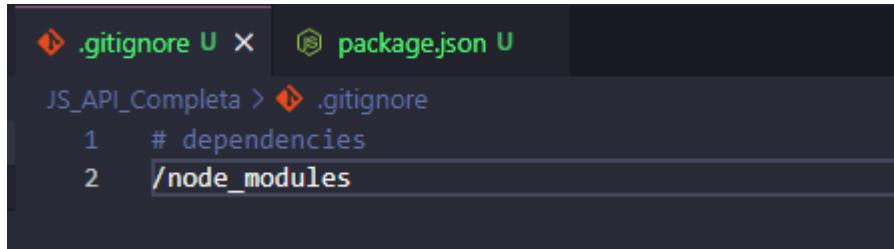
E também a **URL** (<http://localhost:5000/users>) para **visualizar os dados** os dados disponíveis no **arquivo JSON** no seu navegador.



```
[{"id": 1, "name": "Lucas Graham", "email": "l.graham@hotmail.com"}, {"id": 2, "name": "Everaldo Silva", "email": "e.silva@gmail.com"}, {"id": 3, "name": "Claudio Borges", "email": "c.borges@gmail.com"}]
```

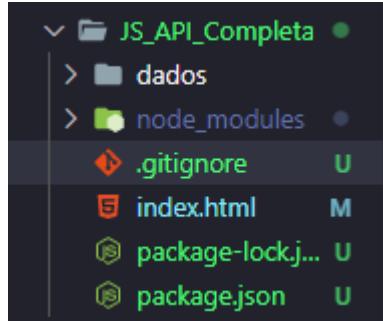
Por fim, como estamos utilizando um **repositório do GitHub**, não é interessante enviar os **todos os pacotes instalados pelo gerenciar de pacotes npm**. Então, vamos criar um **arquivo na raiz** do diretório do nosso projeto com o nome **.gitignore** (Importante que o nome seja exatamente esse). Esse arquivo contém a **lista de arquivos ou diretório para serem ignorados** pelo **GitHub** e que não queremos enviar para o **repositório remoto**. Nesse arquivo, vamos inserir a seguinte instrução:

```
dependencies
/node_modules
```



```
dependencies
/node_modules
```

Assim que **salvamos o esse arquivo** com essa **instrução** o diretório **node\_modules** fica com um **tom de cor diferente** dos **outros diretórios**, pois ele só existirá no **repositório local**.



Vamos **parar por aqui** e continuar a **implementação do nosso projeto** na **próxima aula**.



## Criando uma API - Parte 02

**Os objetivos desta aula são:**

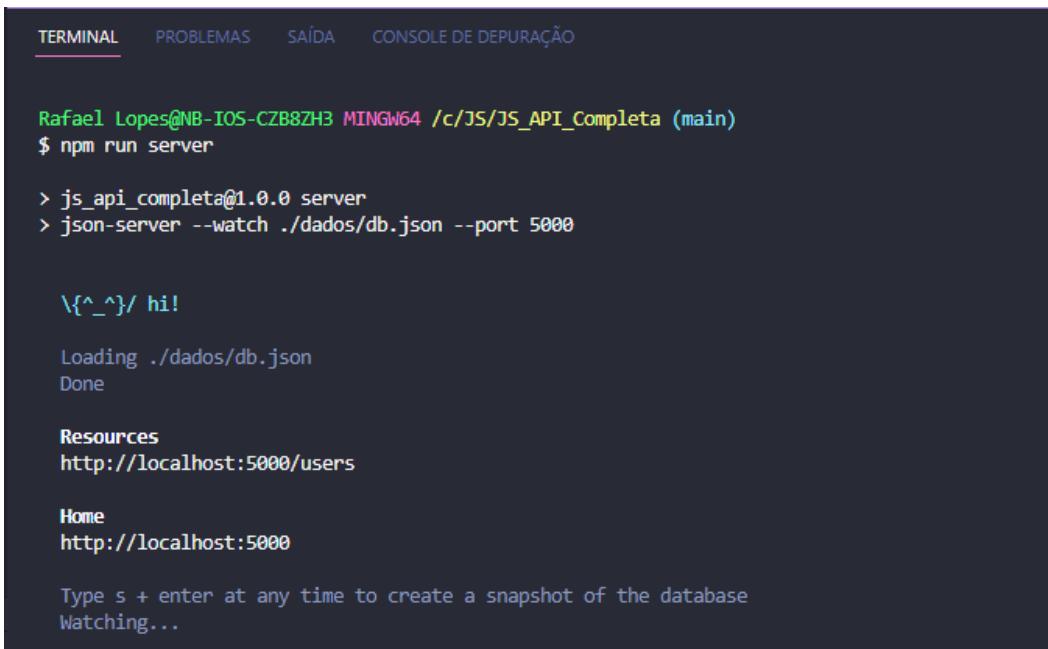
- Exercitar os conhecimentos aprendidos por meio da criação de projetos;
- Criar uma API utilizando Javascript.

**Bons estudos!**

## Vamos praticar

Antes de começar a **implementação** vamos abrir o **terminal do VS Code**, entrar no **diretório do projeto** e **inicializar o nosso servidor** digitando o comando no terminal:

```
npm run server
```



```
TERMINAL PROBLEMAS SAÍDA CONSOLE DE DEPURAÇÃO

Rafael Lopes@NB-IOS-CZB8ZH3 MINGW64 /c/JS/J5_API_Completa (main)
$ npm run server

> js_api_completa@1.0.0 server
> json-server --watch ./dados/db.json --port 5000

\{^_^\}/ hi!

Loading ./dados/db.json
Done

Resources
http://localhost:5000/users

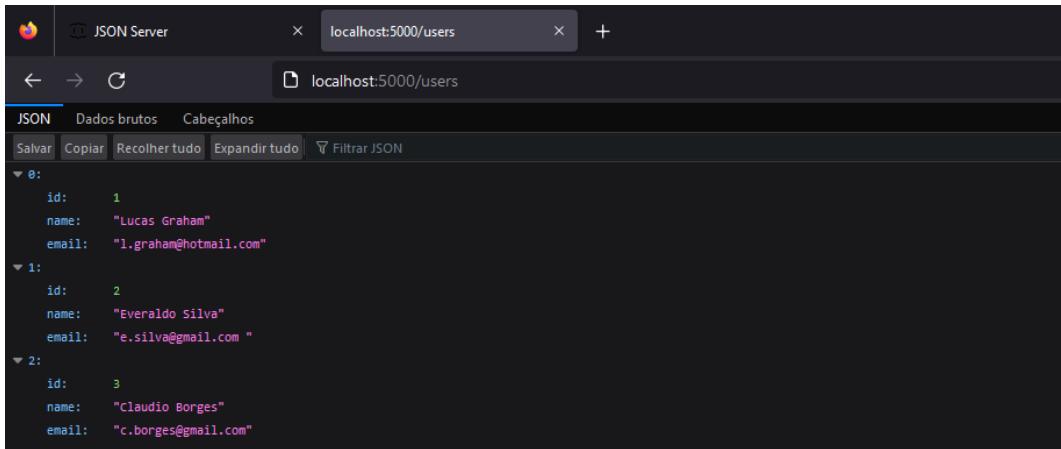
Home
http://localhost:5000

Type s + enter at any time to create a snapshot of the database
Watching...
```

### Importante!

 **Não fechar ou finalize o servidor** durante a **execução da nossa aplicação**, pois se não ela não irá funcionar.

Se tudo estiver configurado corretamente o seu servidor será **inicializado corretamente** e você pode abrir o seu **navegador web** e digitar a **URL** (<http://localhost:5000/users>) abaixo no seu navegador para **visualizar os dados disponíveis** no arquivo **JSON**.



```

{
 "0": {
 "id": 1,
 "name": "Lucas Graham",
 "email": "l.graham@hotmail.com"
 },
 "1": {
 "id": 2,
 "name": "Everaldo Silva",
 "email": "e.silva@gmail.com"
 },
 "2": {
 "id": 3,
 "name": "Claudio Borges",
 "email": "c.borges@gmail.com"
 }
}

```

Agora, vamos **implementar** nosso código em **JS** para **criamos nossa API**. Siga os passos para completar o **código do JavaScript**.

No arquivo **main.js**, começaremos pela criação das **variáveis** como mostra o seguinte código.

```

let usersList = document.getElementById('users-list');
let addUser = document.querySelector('#add-user-form');
let nameValue = document.getElementById('name');
let emailValue = document.getElementById('e-mail');
let btnAdd = document.querySelector('.btn');
let url_api = ' http://localhost:5000/users';
let output = '';

```

Utilizamos o método **getElementById()** para retornar o elemento com **id="users-list"** da página **HTML** e referenciá-lo na variável **usersList**. Esse será o **objeto** que usaremos para **atualizar** os **dados exibidos na lista de usuários** da nossa página **HTML**.

O método **querySelector()** **buscar** o elemento **HTML** com a **id igual a add-user-form** e referencia-o na variável **addUser**.

Podemos ver as **variáveis vinculadas** aos **dois campos do formulário**. Com elas podemos **executar ações** nesses **campos**, como por exemplo **pegar os valores digitados neles**.

Podemos ver a variável **btnAdd** referenciada ao botão de **adicionar do formulário**, que possui a classe **.btn**.

Temos a **URL** para nossa **API** com o caminho do nosso arquivo **JSON**, podemos ver a **variável output** que será onde **construiremos** nosso código **HTML** com a **lista de usuários** para serem **exibidos na página web**.



## Criação da operação READ (Método GET)

Vamos, então, implementar o método **HTTP GET** para **buscar os dados** dos **usuários cadastrados** no arquivo **JSON**.

```
// READ - Lê os usuários
// Método: GET

const renderData = (users) => {
 users.forEach((user) => {
 // console.log(user);
 output += `
 <div class="card col-md-6 bg-light">
 <div class="card-body" data-id=${user.id}>
 <h5 class="card-title">${user.id}</h5>
 <h6 class="card-subtitle mb-2">${user.name}</h6>
 <p class="card-text">${user.email}</p>
 Edit
 Delete
 </div>
 </div>
 `;
 });
 userslist.innerHTML = output;
};

const getData = () => {
 try {
 fetch(url_api, {
 method: 'GET',
 })
 .then((res) => res.json())
 .then((data) => renderData(data));
 } catch (err) {
 console.log(err);
 }
};

window.onload = function () {
 getData();
};
}
```

É importante **modularizar o nosso código**, portanto vamos criar uma **função para receber as informações dos usuários** e construir os **cards** em HTML para **exibir os dados** dos nossos usuários na página web. Nesse caso, a função **renderData()** é responsável por fazer a **renderização dos dados e incorporá-los no HTML**. Essa função **recebe como parâmetro** os **dados** e **armazena-os na variável users**. Então, ela utiliza o método **forEach()** para organizar cada item do objeto **JSON** no **layout do HTML** e armazena concatena todas as informações na variável **output**. Por fim, colocamos o **conteúdo** da variável **output** na nossa página HTML.

A função **getData()** possui uma declaração **try ... catch**, que é composto de um **bloco de comandos try** e outro **bloco catch**. O bloco de comandos **try** é executado **primeiro** e **se alguma exceção ocorrer** o bloco **catch** será **executado**. O bloco **catch** é onde devemos inserir

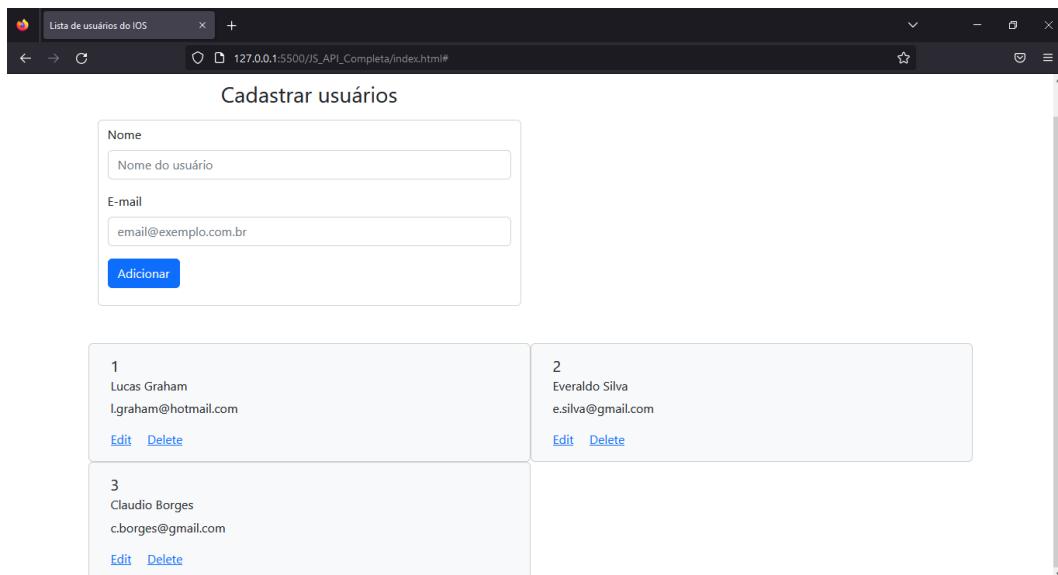
as **instruções** para lidar com os **erros que podem acontecer** no bloco **try**. No nosso exemplo, caso algum **erro aconteça** o bloco **catch** irá **imprimir no console** a mensagem do **erro que aconteceu**.

A sintaxe **async function** define uma função **assíncrona** que permite o uso da palavra-chave **await** dentro dela. Em resumo, as palavras-chaves **async** e **await** permitem o comportamento **assíncrono da função** e baseado em **promises (promissas)**. Essa é uma **sintaxe** comumente utilizada para **trabalhar com APIs**.

Dentro do bloco **try** podemos ver o **método fetch()** fazendo a **requisição na nossa API** utilizando o endereço da **URL** disponível na variável **url\_api** e o **método GET**. A primeira instrução **then** pega a **resposta da requisição** e retorna um **objeto JSON** como **resultado**. A segunda instrução **then**, invoca a função **renderData()** passando como parâmetro os **dados dos usuários**.

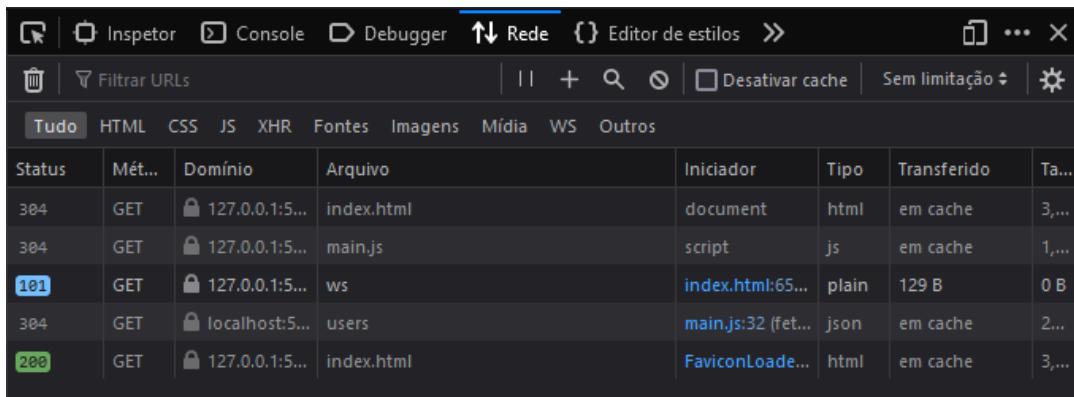
Por fim, a função **getData()** será invocada toda vez que a **página web for carregada** no navegador web.

Após salvar o nosso arquivo **main.js**, podemos **abrir a página** no **navegador web** para ver o **resultado no Frontend** exibindo os **três usuários já disponível no arquivo JSON**.



1	Lucas Graham l.graham@hotmail.com	<a href="#">Edit</a> <a href="#">Delete</a>
2	Everaldo Silva e.silva@gmail.com	<a href="#">Edit</a> <a href="#">Delete</a>
3	Claudio Borges c.borges@gmail.com	<a href="#">Edit</a> <a href="#">Delete</a>

Um ponto interessante que podemos comentar agora são os **códigos de status retornados quando realizamos a requisição**. Em destaque o **Status Code 200** de uma requisição feita pelo arquivo **main.js** que indica que a nossa requisição de busca dos dados foi **bem-sucedida**.



Status	Mét...	Domínio	Arquivo	Iniciador	Tipo	Transferido	Ta...
304	GET	127.0.0.1:5...	index.html	document	html	em cache	3,...
304	GET	127.0.0.1:5...	main.js	script	js	em cache	1,...
101	GET	127.0.0.1:5...	ws	index.html:65...	plain	129 B	0 B
304	GET	localhost:5...	users	main.js:32 (fet...	json	em cache	2,...
200	GET	127.0.0.1:5...	index.html	FaviconLoad...	html	em cache	3,...

### Importante!



Você pode visualizar os **códigos de status** na ferramenta de desenvolvimento do seu navegador (**tecla F12**) na aba **Rede**.



## Criação da operação CREATE (Método POST)

Agora vamos implementar a operação **CREATE** utilizando o método **HTTP POST**. Vamos inserir o seguinte código no arquivo **main.js**, após o código inserido anteriormente.

```
// CREATE - criar novo usuário
// Método HTTP: POST

addUser.addEventListener('submit', async (e) => {
 e.preventDefault();

 newUser = {
 name: nameValue.value,
 email: emailValue.value,
 };

 try {
 let res = await fetch(url_api, {
 method: 'POST',
 headers: {
 'Content-type': 'application/json',
 },
 body: JSON.stringify(newUser),
 })
 .then((res) => res.json())
 .then((data) => {
 let dataArr = [];
 dataArr.push(data);
 renderData(dataArr);
 addUser.reset();
 })
 } catch (err) {
 console.log(err);
 }
});
```

```

 });
} catch (err) {
 console.log(err);
}
});

```

Vinculamos o método `addEventListener()` à variável `addUser`, que permite **manipular** o **formulário** do nosso **Frontend**. Esse método está configurado para **vigiar qualquer evento** de **submissão** do **formulário (submit)** e caso o evento **aconteça** a função **assíncrona** será **executada**. Ou seja, sempre que o botão **Adicionar** for **pressionado** esse evento irá **executar** a **nossa função**.

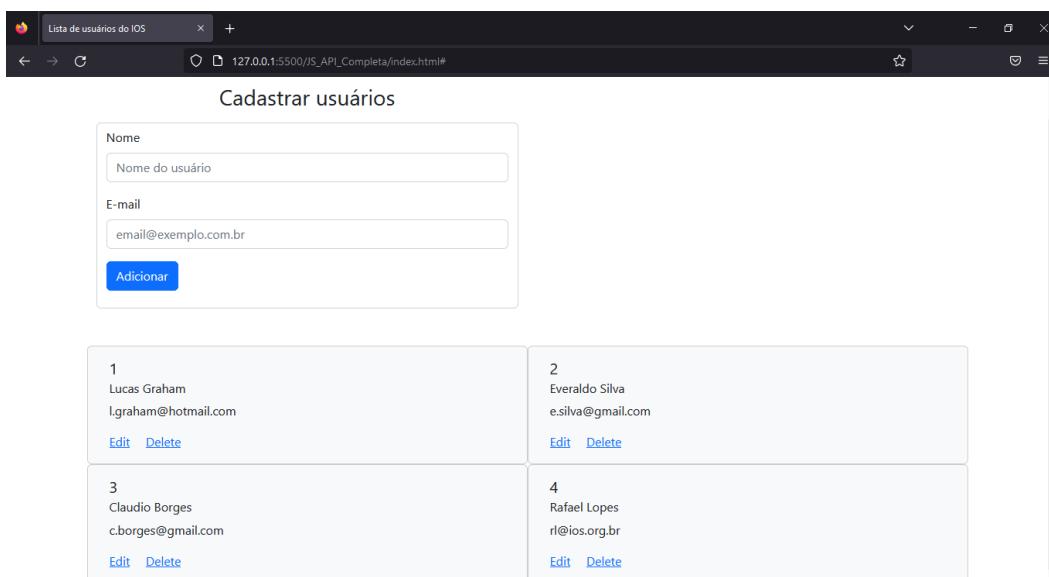
Podemos visualizar, a instrução usada para **criar um objeto** com os valores digitados nos **campos do formulário**.

Temos o método `fetch()` que utiliza o método **HTTP POST** para **criar um registro** no nosso **arquivo**. Podemos notar que estamos usando a chave **headers** com o cabeçalho **Content-Type** indicando o tipo de arquivo utilizado no recurso. No nosso caso, estamos utilizando um arquivo **JSON**. Por fim, a chave **body** indica o conteúdo que **queremos adicionar** no nosso arquivo de **dados**. A função `JSON.stringify()` converte um **objeto** ou um **valor** em uma **string JSON**.

A primeira instrução `then`, pega a resposta da **requisição** e retorna um **objeto JSON** como **resultado**. A segunda instrução `then`, cria uma **variável array** utiliza o método `push()` para inserir os **dados** do **novo registro** na variável `dataArr`, e, então, invoca a função `renderData()` passando como **parâmetro** a **variável** com os **novos dados para serem inseridos**.

Por fim, a instrução `addUser.reset()` limpa as **informações do formulário**.

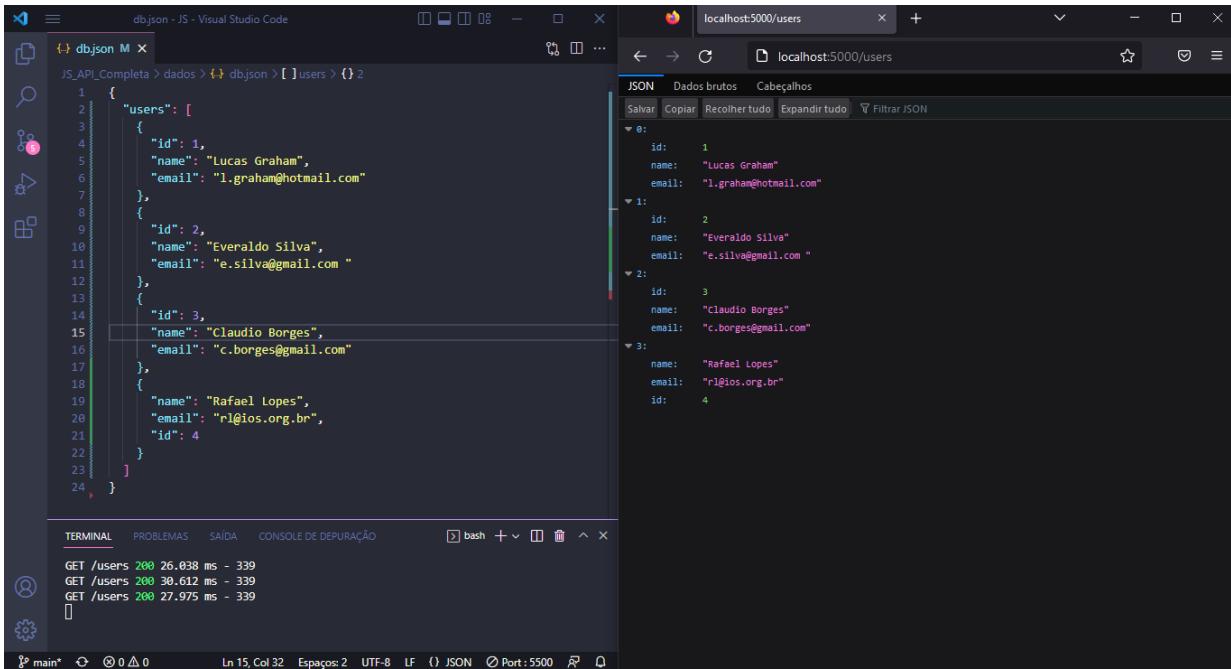
Você pode testar o método POST digitando um nome e e-mail no formulário e clicando botão adicionar.



The screenshot shows a web browser window titled "Lista de usuários do IOS". The address bar indicates the URL is `127.0.0.1:5500/JS_API_Completa/index.html#`. The main content area has a heading "Cadastrar usuários" and contains two input fields: "Nome" (Nome do usuário) and "E-mail" (email@example.com). Below these fields is a blue "Adicionar" button. To the right, there is a table displaying four user entries:

1	Lucas Graham l.graham@hotmail.com	<a href="#">Edit</a> <a href="#">Delete</a>
2	Everaldo Silva e.silva@gmail.com	<a href="#">Edit</a> <a href="#">Delete</a>
3	Claudio Borges c.borges@gmail.com	<a href="#">Edit</a> <a href="#">Delete</a>
4	Rafael Lopes rl@ios.org.br	<a href="#">Edit</a> <a href="#">Delete</a>

Se o novo usuário foi **adicionado** com **sucesso**, podemos o novo registro no **Frontend** e também no arquivo **db.json**, tanto no **servidor** rodando quanto no arquivo aberto no **VS Code**.



The screenshot shows a Visual Studio Code interface. On the left, the `db.json` file is open, displaying a JSON array of users:

```

1 {
 "users": [
 {
 "id": 1,
 "name": "Lucas Graham",
 "email": "l.graham@hotmail.com"
 },
 {
 "id": 2,
 "name": "Everaldo Silva",
 "email": "e.silva@gmail.com"
 },
 {
 "id": 3,
 "name": "Claudio Borges",
 "email": "c.borges@gmail.com"
 },
 {
 "name": "Rafael Lopes",
 "email": "rl@ios.org.br",
 "id": 4
 }
]
}

```

On the right, a browser window shows the JSON response from `localhost:5000/users`:

```

[{"id": 1, "name": "Lucas Graham", "email": "l.graham@hotmail.com"}, {"id": 2, "name": "Everaldo Silva", "email": "e.silva@gmail.com"}, {"id": 3, "name": "Claudio Borges", "email": "c.borges@gmail.com"}, {"name": "Rafael Lopes", "email": "rl@ios.org.br", "id": 4}]

```

The terminal at the bottom shows three successful GET requests to the /users endpoint.



## Criação da operação DELETE (Método DELETE)

Agora, vamos continuar a implementação do nosso código em **JS** para criamos a operação **DELETE** do **CRUD** para a nossa **API** utilizando o método **HTTP DELETE**. Siga os passos para completar o código do **JavaScript**.

Vamos implementar agora a operação **CRUD DELETE** utilizando os métodos **HTTP DELETE**. Digite o seguinte código no seu arquivo **main.js**.

```

// Outras operações CRUD - UPDATE e DELETE

usersList.addEventListener('click', async (e) => {
 e.preventDefault();

 let delButtonIsPressed = e.target.id == 'delete-post';
 let editButtonIsPressed = e.target.id == 'edit-post';
 let id = e.target.parentElement.dataset.id;

 // DELETE - apaga usuário existente
 // Método HTTP: DELETE

 if (delButtonIsPressed) {
 await fetch(`${url_api}/${id}`, {
 method: 'DELETE',
 })
 .then((res) => res.json())
 .then(() => location.reload());
 }
}

```

```

 }

 // UPDATE - atualiza usuário existente
 // Método HTTP: PUT
});

```

Como as operações **DELETE** e **UPDATE** modificam a **lista de usuários** vamos criar uma função para as **duas operações**.

Podemos ver no código acima, estamos usando o método **addEventListener()** à variável **usersList**, e esse evento é acionado sempre que clicamos (**click**) com o botão esquerdo do mouse em **qualquer parte da lista de usuários** e caso o clique aconteça a função **assíncrona** será **executada**.

Dentro da função **assíncrona**, podemos ver as **variáveis** que iremos utilizar para **identificar** os botões **Edit** e **Delete** da tag **<div>** com a classe **card** criado na função **renderData()** onde inserimos as informações dos usuários buscadas no arquivo **JSON**. A variável **delButtonIsPressed** busca dentro da área do **card** o elemento **HTML <a>** com a **id igual a delete-post**. Essa variável será utilizada no método **DELETE**. A variável **editButtonIsPressed** busca o elemento **HTML <a>** com a **id igual a edit-post** e será usado no método **UPDATE**.

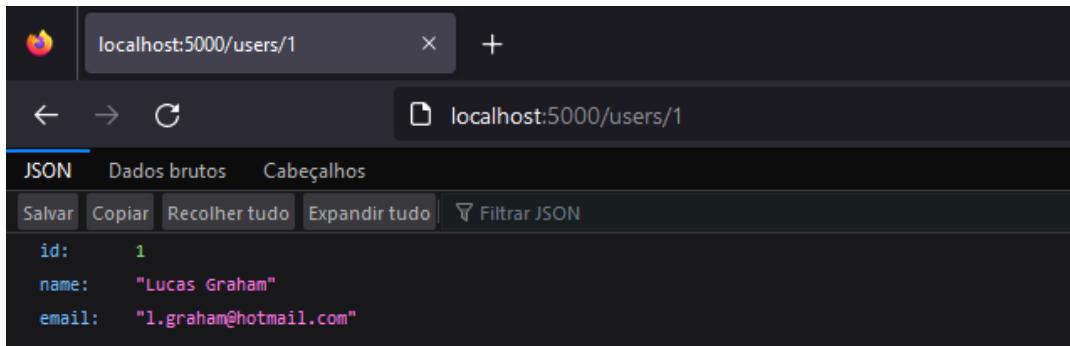
Temos a variável **id** que busca o atributo **data-id** do **card** com a **lista de usuários**. A instrução **e.target.parentElement.dataset.id** retorna o valor **desse atributo**. Se colocarmos uma instrução para imprimir o **valor retornado por essa instrução** como a mostrada abaixo, poderemos ver que ela retorna o número do **id** do **registro** no arquivo **JSON**, quando clicamos no **card exibido no Frontend**.

```
console.log(e.target.parentElement.dataset.id)
```

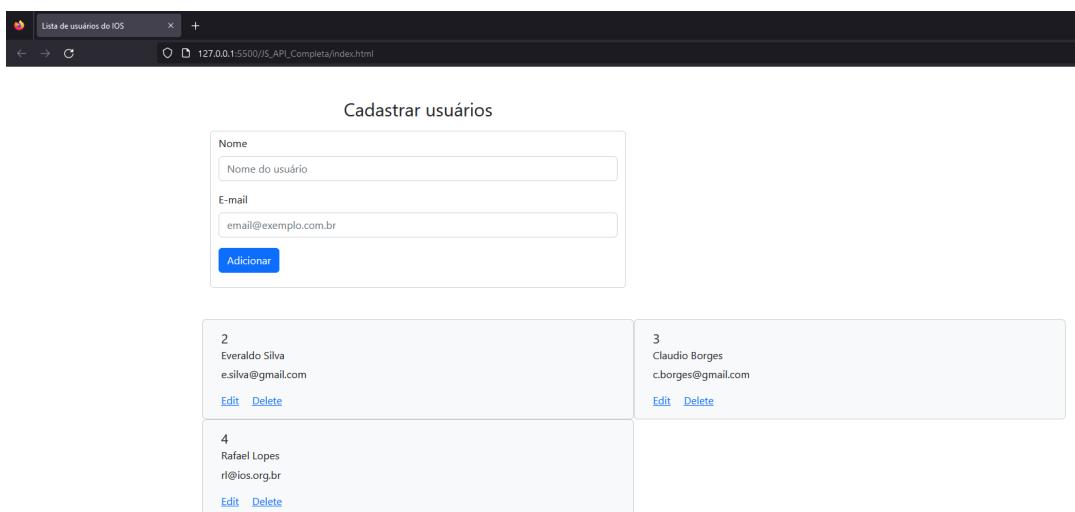
Então, agora podemos **analisar** o método **DELETE**. A função **fetch()** busca o registro de acordo com a **id**. Nesse caso, a **URL** de **busca do dado** é construída com a variável **url\_api** e a variável **id**:

```
`${url_api}/${id}`
```

Essa **URL** busca **um registro apenas**, podemos testar no nosso **navegador web**, por exemplo, se digitarmos a URL: <http://localhost:5000/users/1>, veremos que o **servidor** irá **exibir apenas o registro** com a **ID igual a 1**. Você pode testar com as outras **Ids** disponíveis no arquivo **JSON**.

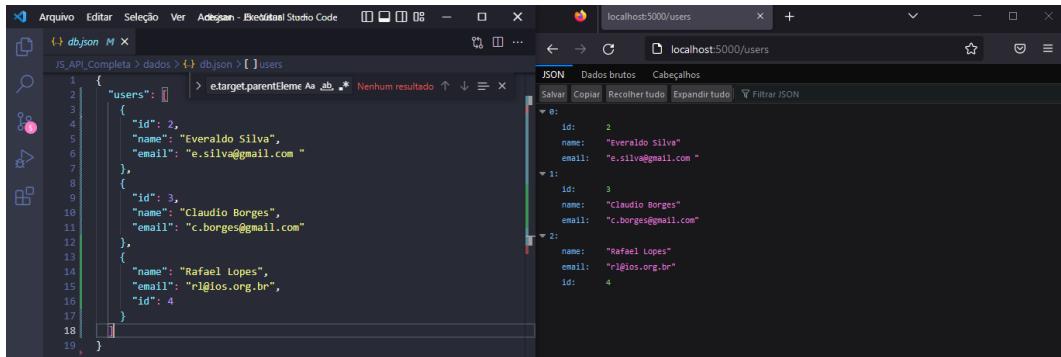


Também podemos ver o **método HTTP DELETE** indicando que queremos **apagar um registro** no arquivo **JSON**. A primeira instrução **then**, pega a **resposta da requisição** e retorna um **objeto JSON** como **resultado**. E a segunda instrução **then**, usa o método **location.reload()** que recarrega a **URL atual** e funciona como um botão de **atualizar a página** que está sendo exibida. Você pode testar o método **DELETE** e **apagar** qualquer um dos **usuários** de sua escolha, por exemplo clicar no botão **Delete** do **usuário 1**.



2	Everaldo Silva e.silva@gmail.com	<a href="#">Edit</a> <a href="#">Delete</a>
3	Claudio Borges c.borges@gmail.com	<a href="#">Edit</a> <a href="#">Delete</a>
4	Rafael Lopes rl@ios.org.br	<a href="#">Edit</a> <a href="#">Delete</a>

Se o usuário foi **apagado com sucesso**, podemos ver o **Frontend** atualizado com os **registros restantes** e, também, no arquivo **db.json**, tanto no **servidor** rodando quanto no arquivo aberto no **VS Code**.



```

db.json
[{"users": [{"id": 2, "name": "Everaldo Silva", "email": "e.silva@gmail.com"}, {"id": 3, "name": "Claudio Borges", "email": "c.borges@gmail.com"}, {"id": 4, "name": "Rafael Lopes", "email": "rl@ios.org.br"}]}

```

```

localhost:5000/users
[{"id": 2, "name": "Everaldo Silva", "email": "e.silva@gmail.com"}, {"id": 3, "name": "Claudio Borges", "email": "c.borges@gmail.com"}, {"id": 4, "name": "Rafael Lopes", "email": "rl@ios.org.br"}]

```

### Importante!

Você tem que recarregar a página do servidor (<http://localhost:5000/users>) no navegador web, se ela já estiver aberta, para ver que o registro foi apagado.



## Criação da operação UPDATE (Método PUT)

Agora, vamos continuar a implementação do nosso código em **JS** para criamos a operação **UPDATE** do **CRUD** para a nossa **API** utilizando o método **HTTP PUT**. Siga os passos para completar o código do **JavaScript**.

Vamos implementar agora a operação **CRUD UPDATE** utilizando os métodos **HTTP PATCH**. Vamos colocar o seguinte código junto do método **DELETE**, logo abaixo do comentário **//UPDATE** – atualiza usuário existente.

```
// UPDATE - atualiza usuário existente
// Método HTTP: PUT

if (editButtonIsPressed) {
 const parent = e.target.parentElement;
 let nameContent = parent.querySelector('.card-subtitle').textContent;
 let emailContent = parent.querySelector('.card-text').textContent;

 nameValue.value = nameContent;
 emailValue.value = emailContent;
}

btnAdd.addEventListener('click', async (e) => {
 e.preventDefault();
 await fetch(`${url_api}/${id}`, {
 method: 'PATCH',
 headers: {
 'Content-type': 'application/json',
 },
 body: JSON.stringify({
 name: nameValue.value,
 email: emailValue.value,
 }),
 })
 .then((res) => res.json())
 .then(() => {
 location.reload();
 addUser.reset();
 });
});
```

O primeiro bloco de instruções do código mostrado acima, mostra uma declaração **if** que testa se o **botão Edit** foi **pressionado**. Caso pressionarmos o **botão**, os valores do **nome** e **e-mail** serão carregados para os **respectivos campos do formulário** no **Frontend**. Desse modo poderemos **editar os dados** do **usuário existente**.

Guardamos o elemento pai que corresponde ao **card** que foi **clicado**. Buscamos o **conteúdo** do elemento **HTML** com a classe **.card-subtitle** que é o **nome do usuário**. Em seguida, buscamos o **conteúdo** do elemento **HTML** com a classe **.card-text** que é o **e-mail do usuário**. Por fim, adicionamos os valores de **nome** e **e-mail** nos **campos do formulário**.

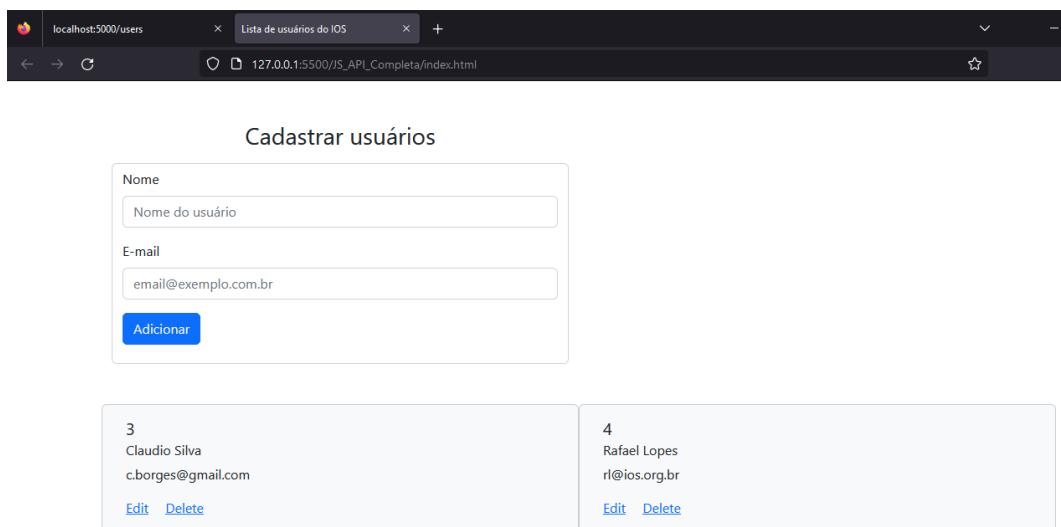
No segundo bloco de instruções, temos o método **addEventListener()** à variável **addUser**, que dispara a função **assíncrona** quando o botão **Adicionar** é **clicado**. Dentro da função

**assíncrona** temos, o método **fetch()** que utiliza o método **HTTP PATCH** para realizar a operação **UPDATE** de um registro no arquivo **JSON**.

Podemos ver o **body** da **requisição** que busca os valores **atualizados** do nos campos **nome** e **e-mail** do **formulário** de **atualização** no usuário que foi **editado**.

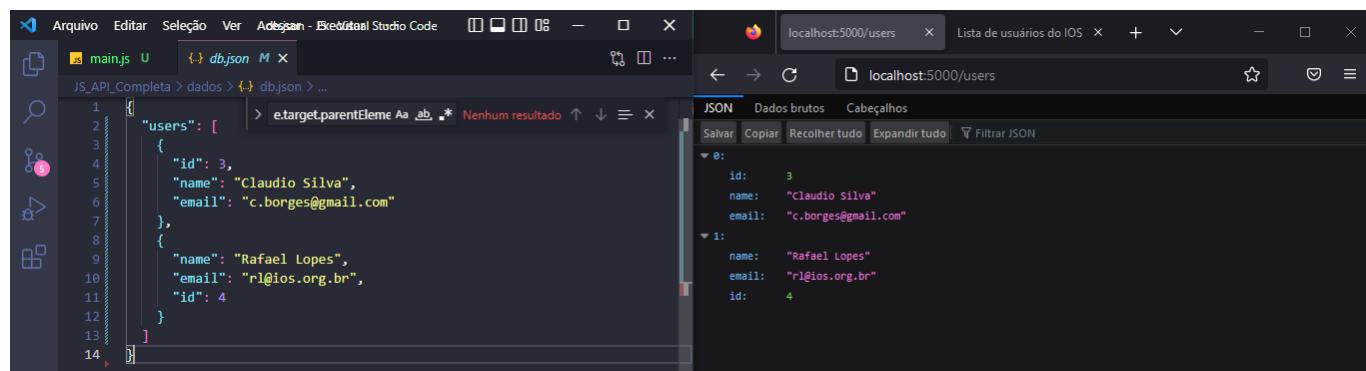
Por fim, temos a segunda instrução **then** com o método **location.reload()** para **atualizar** a página web e **addUser.reset()** para **limpar os campos do formulário**.

Podemos testar o método **PATCH**, clicando no botão **Edit** de **algum usuário**, como por exemplo, o **usuário 3**. Podemos **editar** o seu **nome** e **adicionar** o **sobrenome Silva**. Quando clicamos no botão **Adicionar** o **Frontend** será **atualizado** com as **informações do usuário 3 editadas**.



3	Cláudio Silva	c.borges@gmail.com	<a href="#">Edit</a>	<a href="#">Delete</a>
4	Rafael Lopes	rl@ios.org.br	<a href="#">Edit</a>	<a href="#">Delete</a>

Se o novo usuário foi **editado com sucesso**, podemos o novo registro no **Frontend** e também no arquivo **db.json**, tanto no **servidor** rodando quanto no **arquivo aberto** no **VS Code**.



```
1 [
2 "users": [
3 {
4 "id": 3,
5 "name": "Cláudio Silva",
6 "email": "c.borges@gmail.com"
7 },
8 {
9 "name": "Rafael Lopes",
10 "email": "rl@ios.org.br",
11 "id": 4
12 }
13]
14]
```

E isso é tudo pessoal.