





Roadmap JavaScript



Diego Nascimento



● Fase 1: Fundamentos


Objetivo: Entender a linguagem em si, sem frameworks ou bibliotecas.





-  **Introdução ao JavaScript**
 - O que é JavaScript e para que serve
 - Como o JS funciona no navegador (Engine V8, por exemplo)
 - Onde escrever (console, `<script>`, editores como VS Code)
-  **Sintaxe Básica**
 - Variáveis (`let`, `const`, `var`)
 - Tipos de dados (string, number, boolean, null, undefined, symbol, object)
 - Operadores (aritméticos, lógicos, comparação)
-  **Controle de Fluxo**
 - Condicionais (`if`, `else`, `switch`)
 - Loops (`for`, `while`, `do...while`, `for...of`, `for...in`)
-  **Funções**
 - Declaração e expressão
 - Arrow functions

- Parâmetros padrão, rest/spread
 - Escopo e closures
 -  **Objetos e Arrays**
 - Criação, acesso e manipulação
 - Métodos importantes: `.map()`, `.filter()`, `.redu()`, `.forEach()`
 -  **DOM (Document Object Model)**
 - Seleção de elementos (`getElementById`, `querySelector`)
 - Manipulação de conteúdo e atributos
 - Eventos (`addEventListener`)
-

Fase 2: Intermediário

Objetivo: Começar a entender o JavaScript mais moderno e seu uso em aplicações reais.

-  **ES6+ Features**
 - Desestruturação
 - Template literals
 - Classes e herança

- Módulos (`import/export`)
 - Promises e `async/await`
 -  **Manipulação de Formulários e Eventos**
 - Validar dados
 - Enviar formulários
 - Eventos de teclado, mouse e submit
 -  **Fetch API**
 - Requisições HTTP com `fetch()`
 - Trabalhar com JSON
 - Tratamento de erros
 -  **Armazenamento Local**
 - `localStorage`, `sessionStorage`, `cookies`
 -  **Debugging e Ferramentas**
 - DevTools do navegador
 - `console.log`, `debugger`
 - Linters (ESLint)
-





Fase 3: Avançado

Objetivo: Tornar-se apto a construir aplicações robustas e compreender melhor a linguagem.

-  **Programação Assíncrona**
 - Event Loop
 - Microtasks vs macrotasks
 - `Promise.all`, `Promi.race`
-  **Design Patterns em JS**
 - Module pattern
 - Singleton, Factory, Observer
-  **Testes**
 - Testes unitários com Jest
 - Testes de integração
-  **Segurança**
 - XSS, CSRF
 - Sanitização de inputs
-  **Webpack, Babel e ferramentas de build**


Fase 4: Aplicações com JavaScript

Objetivo: Usar o conhecimento para criar projetos reais e conectar com tecnologias modernas.

-  **DOM Dinâmico e SPA** - Criar interfaces interativas
 - Navegação sem reload
 -  **Frameworks e Bibliotecas (opcional)**
 - React.js (ou Vue.js, Svelte)
 - Uso de componentes, props, estado
 -  **TypeScript (JS com tipagem)**
 - Tipagem estática
 - Interfaces, tipos, enums
 -  **Back-end com JavaScript**
 - Node.js e Express
 - APIs REST
 - MongoDB (com Mongoose)
-

Fase 5: Prática e Projetos

Objetivo: Consolidar o conhecimento com experiência prática.

-  **Projetos sugeridos**
 - To-Do List
 - Conversor de moedas

- Jogo da velha
 - Cronômetro / Timer
 - Consumo de API (ex: clima, filmes)
 - CRUD com armazenamento local
 - Aplicação completa com front-end e back-end
-

Fase 01 – Fundamentos de JavaScript

Objetivo da aula

- Entender o que é JavaScript e como ele funciona no navegador.
 - Aprender a sintaxe básica da linguagem.
 - Criar suas primeiras variáveis, estruturas de controle e funções.
-

1. O que é JavaScript?

JavaScript é uma **linguagem de programação de alto nível, interpretada**, usada principalmente para tornar páginas da web interativas.

Onde o JS roda?

- Navegadores (Chrome, Firefox, etc.)
- Servidores (com Node.js)

Exemplo:

```
<script>  
  alert('Olá, mundo!');  
</script>
```



2. Variáveis e Tipos de Dados



Como declarar variáveis:

```
let nome = 'Ana';  
const idade = 25;  
var cidade = 'São Paulo'; // forma antiga, evite
```



Tipos básicos:

- **String:** `'texto'`, `"texto"`
- **Number:** `10`, `3.14`
- **Boolean:** `true`, `false`
- **Null:** `null` (ausência proposital)
- **Undefined:** `undefined` (não definido)
- **Object** e **Array** (estruturados – veremos depois)



Exemplo:

```
let nome = "Carlos";  
let idade = 30;  
let online = true;
```

3. Operadores

Aritméticos:

+ - * / %

Comparação:

== === != !== > < >= <=

Lógicos:

&& || !

4. Controle de Fluxo (Condicionais)

if, else e else if:

```
let idade = 18;
```

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



5. Loops (Repetição)

for

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

while

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



6. Funções

Declarando funções:

```
function saudacao(nome) {  
  return "Olá, " + nome + "!";  
}
```

Chamando a função:

```
console.log(saudacao("Maria"));
```

Arrow Function (moderna):

```
const soma = (a, b) => a + b;
```



7. Exercícios Práticos

1. Crie uma variável com seu nome e exiba no console.
 2. Crie uma função que receba 2 números e retorne o maior.
 3. Faça um loop que imprima os números de 1 a 10.
 4. Crie um programa que diga se um número é par ou ímpar.
-



8. Projeto Mini: Validador de Idade

Objetivo:

Pedir a idade do usuário com `prompt()` e responder com `alert()` se ele pode dirigir.

```
<script>
let idade = prompt("Qual sua idade?");
if (idade >= 18) {
  alert("Você pode dirigir!");
} else {
  alert("Você ainda não pode dirigir.");
}
</script>
```



Tarefa para casa

- Refazer todos os exemplos no console do navegador.
- Criar uma calculadora simples (adição e subtração) usando `prompt()` e `alert()`.



Recapitulando

- ✓ O que é JavaScript
 - ✓ Como declarar variáveis e tipos
 - ✓ Operadores, condicionais, loops
 - ✓ Como criar e usar funções
 - ✓ Projeto simples com `prompt` e `alert`
-

Introdução ao JavaScript







O que é JavaScript?

JavaScript (JS) é uma linguagem de programação **interpretada**, **dinâmica**, e **baseada em protótipos**, usada principalmente para criar **interatividade em páginas web**.

 **JS não é Java!** São linguagens diferentes.

Para que serve o JavaScript?

JavaScript é usado para:

-  Interações com o usuário (cliques, teclado, etc)
 -  Validação de formulários
 -  Atualizar o conteúdo da página sem recarregar (AJAX)
 -  Criar animações e jogos simples
 -  Conectar com servidores e APIs
 -  Criar aplicações completas (frontend e backend)
-

Onde o JavaScript é usado?

Ambiente	Exemplos
----------	----------

Navegadores	Chrome, Firefox, Safari, Edge
-------------	-------------------------------

Servidores Node.js (backend com JS)

Apps híbridos React Native, Electron

Como o JavaScript funciona?

JavaScript roda em **motores (engines)** embutidos nos navegadores.

- V8 (Chrome, Node.js)
- SpiderMonkey (Firefox)
- JavaScriptCore (Safari)

 O motor lê seu código JS, interpreta e executa no navegador.

Como escrever JavaScript?

1. Dentro do HTML:

```
<script>  
  alert("Olá, mundo!");  
</script>
```

2. Arquivo externo:

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

No `script.js`:

```
console.log("Olá do arquivo externo!");
```

Testando JavaScript no Navegador

Você pode testar código diretamente no navegador:

1. Clique com o botão direito na página
2. Vá em **"Inspecionar"** → **Aba "Console"**
3. Digite:

```
console.log("Testando JavaScript!");
```

Primeiros comandos

// Exibe uma mensagem no console

```
console.log("Olá, JavaScript!");
```

// Exibe um alerta no navegador

```
alert("Bem-vindo ao JS!");
```

// Pede um dado ao usuário

```
let nome = prompt("Qual seu nome?");
```

```
alert("Olá, " + nome + "!");
```

Resumo da Introdução

Conceito	Descrição
Linguagem	Interpretada, dinâmica, usada na web
Finalidade	Interatividade, lógica, comunicação web
Execução	No navegador ou servidor (Node.js)

Onde escrever No HTML, arquivo externo ou console

Prática sugerida

1. Crie uma página HTML simples.
 2. Use `<script>` para exibir uma mensagem com `alert()`.
 3. Teste o console do navegador com `console.log()`.
-



Aula – Sintaxe Básica em JavaScript



Objetivo

- Aprender a **estrutura básica** da linguagem JavaScript.
 - Compreender como **declarações, variáveis, tipos, operadores e comentários** funcionam.
-



1. Estrutura Básica de um Código JS

// Isso é um comentário

let nome = "Maria";

console.log(nome);

- O código JS é lido **linha por linha**.
 - Cada instrução termina com **;** (opcional, mas recomendado).
 - **Case sensitive**: **nome** ≠ **Nome**
-



2. Comentários

- Comentário de uma linha: **//**
- Comentário de múltiplas linhas: **/* ... */**

```
// Comentário simples
/* Comentário
   em várias linhas */
```



3. Variáveis

Tipos de declaração:

```
let nome = "João"; // pode mudar
const idade = 25; // constante (não muda)
var cidade = "SP"; // (forma antiga, evite)
```

Regras de nomes:

- Não começar com número
 - Sem espaços ou caracteres especiais (exceto `_`, `$`)
 - Evite palavras reservadas (como `function`, `var`, etc.)
-



4. Tipos de Dados

```
let texto = "Olá"; // String
let numero = 42; // Number
let ativo = true; // Boolean
let indefinido; // Undefined
let vazio = null; // Null
```

Você pode usar `typeof` para verificar o tipo:

```
console.log(typeof texto); // string
```



5. Operadores

Aritméticos:

+ - * / % **

Comparação:

== // igual (valor)

=== // igual (valor e tipo)

!= // diferente

!== // diferente (valor e tipo)

> < >= <=

Lógicos:

&& // E

|| // OU

! // NÃO



6. Regras de Bloco (Escopo)

As chaves `{ }` definem blocos de código, por exemplo em funções, loops e condicionais.

```
if (true) {  
  let x = 10;  
  console.log(x);  
}
```

7. Exercícios Práticos

1. Declare uma variável com seu nome e exiba no console.
 2. Crie duas variáveis numéricas e exiba a soma.
 3. Crie uma constante com um valor booleano.
 4. Verifique o tipo de uma variável usando `typeof`.
-

Exemplo Completo:

```
// Declarações
let nome = "Carla";
const idade = 22;
let ativa = true;

// Operações
let anoNascimento = 2025 - idade;

// Saída
console.log("Nome:", nome);
console.log("Idade:", idade);
console.log("Nasceu em:", anoNascimento);
```

Desafio rápido

Crie um programa que peça dois números ao usuário com `prompt()` e exiba a soma usando `alert()`.

```
let n1 = Number(prompt("Digite o primeiro número:"));
```

```
let n2 = Number(prompt("Digite o segundo número:"));  
let resultado = n1 + n2;  
alert("A soma é: " + resultado);
```

Aula – Controle de Fluxo em JavaScript

Objetivo

Aprender como o JavaScript **toma decisões** e **repete ações** com estruturas de controle como **if**, **else**, **switch**, **for**, **while**, etc.

1. Condicionais: **if**, **else if**, **else**

Permite executar diferentes blocos de código dependendo de uma **condição booleana**.

Exemplo:

```
let idade = 18;
```

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

```
} else {  
  console.log("Você é menor de idade.");  
}
```

Com **else if**:

```
let nota = 7;
```

```
if (nota >= 9) {  
  console.log("Excelente!");  
} else if (nota >= 6) {  
  console.log("Aprovado");  
} else {  
  console.log("Reprovado");  
}
```

2. Switch Case

Útil quando há **múltiplas condições baseadas em um mesmo valor**.

Exemplo:

```
let cor = "vermelho";
```

```
switch (cor) {  
  case "azul":  
    console.log("Você escolheu azul.");  
    break;  
  case "vermelho":  
    console.log("Você escolheu vermelho.");  
    break;  
  default:  
    console.log("Cor não reconhecida.");  
}
```



3. Laços de Repetição (Loops)

for – repetição com contador:

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

while – repete enquanto a condição for verdadeira:

```
let i = 1;  
while (i <= 3) {  
  console.log("Contando:", i);  
  i++;  
}
```

do...while – executa pelo menos uma vez:

```
let senha;  
do {  
  senha = prompt("Digite a senha:");  
} while (senha !== "1234");  
  
alert("Acesso liberado!");
```



4. Controle com **break** e **continue**

break → interrompe o loop

continue → pula para a próxima iteração

```
for (let i = 1; i <= 5; i++) {  
  if (i === 3) continue; // pula o 3  
  if (i === 5) break;    // para no 5  
  console.log(i);  
}
```

}

Exercícios práticos

1. Crie um programa que receba um número e diga se ele é **par** ou **ímpar**.
 2. Use **switch** para responder a dias da semana (segunda, terça, etc.).
 3. Escreva um **for** que imprima os números de 10 a 1 (de trás pra frente).
 4. Crie um **while** que peça um nome até que o usuário digite "sair".
-

Projeto mini: Classificador de idade

Objetivo: Receber idade do usuário e classificar em:

- Criança (0-12)
- Adolescente (13-17)
- Adulto (18-59)
- Idoso (60+)

Código:

```
let idade = Number(prompt("Digite sua idade:"));
```

```
if (idade <= 12) {
```

```
    alert("Você é uma criança.");
} else if (idade <= 17) {
    alert("Você é adolescente.");
} else if (idade <= 59) {
    alert("Você é adulto.");
} else {
    alert("Você é idoso.");
}
```

Recapitulando

Estrutura	Descrição
<code>if/else</code>	Avalia condições booleanas
<code>switch</code>	Verifica múltiplos valores
<code>for</code>	Repetição com contador
<code>while</code>	Repetição com condição
<code>do...while</code>	Garante ao menos uma execução
<code>break</code> <code>continue</code>	/ Controle de fluxo nos loops



Aula – Funções em JavaScript



Objetivo

- Entender o que são funções
 - Aprender a criar, chamar e reutilizar funções
 - Explorar tipos de funções: **declaradas**, **anônimas**, **arrow functions**
 - Praticar com exemplos e mini-desafios
-



1. O que é uma função?

Uma **função** é um bloco de código que executa uma **tarefa específica** e pode ser **reutilizado**.



2. Função declarada (tradicional)

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

```
saudacao(); // Chamada da função
```



3. Funções com parâmetros

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

```
}
```

```
saudacao("Ana");  
saudacao("Pedro");
```

4. Funções com retorno (**return**)

```
function soma(a, b) {  
  return a + b;  
}
```

```
let resultado = soma(3, 4);  
console.log("Resultado:", resultado);
```

- O **return** encerra a função e devolve um valor.
-

5. Tipos de Funções

Declarada (já vimos acima)

```
function exemplo() {}
```

Anônima (sem nome, geralmente usada em variáveis)

```
const dobro = function (x) {  
  return x * 2;  
};
```

```
console.log(dobro(5)); // 10
```

➡ Arrow Function (forma moderna e mais curta)

```
const triplo = (x) => x * 3;
```

```
console.log(triplo(4)); // 12
```

Com mais de um parâmetro ou corpo maior:

```
const mensagem = (nome, idade) => {  
  return `Olá, ${nome}, você tem ${idade} anos.`;  
};
```

6. Funções dentro de outras funções

```
function calcularIMC(peso, altura) {  
  function arredondar(valor) {  
    return Math.round(valor * 100) / 100;  
  }  
}
```

```
  let imc = peso / (altura * altura);  
  return arredondar(imc);  
}
```

```
console.log(calcularIMC(70, 1.75));
```

7. Exercícios

1. Crie uma função que receba dois números e retorne a multiplicação.
2. Crie uma função `ehPar(numero)` que retorne `true` se for par.
3. Crie uma arrow function que diga "Bom dia, [nome]".



Mini Projeto: Calculadora Simples

Código:

```
function calcular(a, b, operacao) {  
  if (operacao === "+") return a + b;  
  if (operacao === "-") return a - b;  
  if (operacao === "*") return a * b;  
  if (operacao === "/") return a / b;  
  return "Operação inválida";  
}  
  
let n1 = Number(prompt("Digite o primeiro número:"));  
let n2 = Number(prompt("Digite o segundo número:"));  
let op = prompt("Digite a operação (+, -, *, /)");  
  
let resultado = calcular(n1, n2, op);  
alert("Resultado: " + resultado);
```



Recapitulando

Conceito

Exemplo

Declaração

```
function  
nome() {}
```

Parâmetros

```
function  
soma(a, b) {}
```

Retorno

```
return valor;
```

Função anônima

```
const    x    =  
function() {}
```

Arrow
function

```
const x = ()  
=> {}
```

Aula – Objetos e Arrays em JavaScript

Objetivo

- Entender o que são **objetos** e **arrays**
 - Aprender como criá-los, acessar, modificar e iterar
 - Praticar com exemplos e desafios
-

1. Objetos (Object)

Um **objeto** representa uma **coleção de pares chave: valor**. Ele modela entidades do mundo real, como usuários, produtos, etc.

Sintaxe:

```
let pessoa = {
```



```
nome: "Alice",  
idade: 30,  
ativa: true  
};
```



Acessando valores:

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



Modificando valores:

```
pessoa.nome = "Bianca";  
pessoa.idade += 1;
```



Adicionando propriedades:

```
pessoa.cidade = "São Paulo";
```



Removendo propriedades:

```
delete pessoa.ativa;
```



2. Arrays (Vetores)

Um **array** é uma lista **ordenada** de valores, identificados por **índices numéricos**, começando em 0.



Sintaxe:

```
let frutas = ["maçã", "banana", "laranja"];
```



Acessando itens:

```
console.log(frutas[0]); // "maçã"
```



Modificando:

```
frutas[1] = "abacaxi"; // ["maçã", "abacaxi", "laranja"]
```



Adicionando:

```
frutas.push("uva");    // adiciona no final  
frutas.unshift("kiwi"); // adiciona no início
```



Removendo:

```
frutas.pop();    // remove do final  
frutas.shift();  // remove do início
```



3. Iterando Objetos e Arrays



for...of (para arrays)

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



for...in (para objetos)

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



4. Array de Objetos

Muito comum em APIs e listas complexas:

```
let produtos = [  
  { nome: "Camiseta", preco: 30 },  
  { nome: "Calça", preco: 80 },  
  { nome: "Tênis", preco: 150 }  
];  
  
for (let item of produtos) {  
  console.log(item.nome + " custa R$" + item.preco);  
}
```



Exercícios

1. Crie um objeto **livro** com propriedades **titulo**, **autor**, e **ano**.
2. Crie um array com 4 cores.
3. Adicione e remova elementos do array com **push()** e **pop()**.
4. Crie um array de 3 objetos representando alunos com **nome** e **nota**.
5. Itere sobre os alunos e exiba uma mensagem personalizada para cada um.



Desafio Mini-projeto: Cadastro de Usuários

```
let usuarios = [];  
  
function cadastrar(nome, idade) {  
  usuarios.push({ nome, idade });  
}  
  
cadastrar("Ana", 25);  
cadastrar("João", 30);  
  
for (let u of usuarios) {  
  console.log(`Usuário: ${u.nome}, Idade: ${u.idade}`);  
}
```



Resumo

Tipo	Estrutura	Acesso
Objeto	<code>{ chave: valor }</code>	<code>obj.chave</code> ou <code>obj["chave"]</code>
Array	<code>[valor1, valor2]</code>	<code>arr[índice]</code>



Aula – DOM em JavaScript



Objetivo

- Entender o que é o **DOM**
 - Aprender a **acessar, modificar e responder a elementos HTML**
 - Manipular conteúdo, atributos, classes e eventos
-



1. O que é DOM?

DOM = Document Object Model

É uma **representação em árvore** de um documento HTML.
Cada **tag vira um objeto**, que pode ser **manipulado via JavaScript**.

```
<body>
```

```
<h1 id="titulo">Olá!</h1>
```

```
<button onclick="mudarTexto()">Clique aqui</button>
```

```
</body>
```

```
function mudarTexto() {
```

```
    document.getElementById("titulo").innerText = "Texto alterado!";
```

```
}
```

2. Seletores DOM

Por ID:

```
document.getElementById("meuld");
```

Por Classe:

```
document.getElementsByClassName("minhaClasse");
```

Por Tag:

```
document.getElementsByTagName("p");
```

Modernos (recomendado):

```
document.querySelector(".classe, #id, tag");
```

```
document.querySelectorAll("p"); // lista de elementos
```

3. Modificando Conteúdo

```
element.innerText = "Novo texto";
```

```
element.innerHTML = "<strong>HTML aqui</strong>";
```



4. Modificando Atributos e Estilos

```
element.setAttribute("href", "https://google.com");
```

```
element.style.color = "blue";
```

+ 5. Criando e Removendo Elementos

```
let novoItem = document.createElement("li");
```

```
novoItem.innerText = "Novo item";
```

```
document.body.appendChild(novoItem);
```

```
document.body.removeChild(novoItem); // remove
```



6. Eventos

Clique, mudança, etc.

```
<button id="botao">Clique</button>
```

```
document.getElementById("botao").addEventListener("click", () => {
```

```
  alert("Você clicou!");
```

```
});
```

Outros eventos comuns:

- "mouseover" (passar o mouse)
 - "keydown" (tecla pressionada)
 - "change" (mudança em input)
 - "submit" (formulário enviado)
-

7. Exemplo Completo

```
<h2 id="mensagem">Bem-vindo</h2>
```

```
<button onclick="alterarMensagem()">Mudar</button>
```

```
<script>
```

```
function alterarMensagem() {
```

```
  let el = document.getElementById("mensagem");
```

```
  el.innerText = "Olá, DOM!";
```

```
  el.style.color = "red";
```

```
}
```

```
</script>
```

Exercícios

1. Faça um botão que **altera o texto de um parágrafo**.

2. Faça um campo de texto e botão que **exibe o nome digitado**.
3. Crie uma lista (ul) e um botão para **adicionar itens dinamicamente**.
4. Mude a **cor de fundo** da página ao clicar em um botão.



Desafio prático: To-do List básica



Resumo

Tarefa	Código
Selecionar elemento	<code>document.querySelector("#id")</code>
Mudar texto	<code>element.innerText = "novo"</code>
Mudar estilo	<code>element.style.color = "red"</code>
Criar elemento	<code>document.createElement("div")</code>

Adicionar
evento

```
element.addEventListener(  
  "click", fn
```

✨ Aula: JavaScript Intermediário (Fase 2)

🌟 Objetivo

Explorar recursos modernos do JavaScript (ES6+), manipulando eventos, formulários, requisitando dados de APIs e utilizando armazenamento local. Também abordaremos boas práticas de debugging e ferramentas de desenvolvimento.

✅ ES6+ Features

🧰 1. Desestruturação

Permite extrair dados de arrays e objetos de forma mais clara:

```
const usuario = { nome: "Ana", idade: 28 };  
const { nome, idade } = usuario;  
console.log(nome); // "Ana"
```

📋 2. Template Literals

Facilitam a concatenação de strings com variáveis:

```
const produto = "Caneta";  
const preco = 2.5;  
console.log(`O produto ${produto} custa R${preco}`);
```

👛 3. Classes e Herança

Sintaxe moderna para criar objetos com herança:

```
class Animal {  
  constructor(nome) {  
    this.nome = nome;  
  }  
  falar() {  
    console.log(`${this.nome} fez um som.`);  
  }  
}
```

```
}
```

```
class Cachorro extends Animal {  
  falar() {  
    console.log(`${this.nome} latiu.`);  
  }  
}
```

```
const dog = new Cachorro("Rex");  
dog.falar();
```

4. Módulos (import/export)

Permite dividir o código em arquivos reutilizáveis:

```
// arquivo: soma.js  
export function soma(a, b) {  
  return a + b;  
}
```

```
// arquivo: main.js  
import { soma } from './soma.js';  
console.log(soma(2, 3));
```

5. Promises e async/await

```
// Promise  
fetch("https://api.exemplo.com/dados")  
  .then(res => res.json())  
  .then(data => console.log(data))  
  .catch(err => console.error(err));
```

```
// async/await  
async function buscarDados() {  
  try {  
    const res = await fetch("https://api.exemplo.com/dados");  
    const data = await res.json();
```

```
    console.log(data);
  } catch (err) {
    console.error(err);
  }
}
```

Manipulação de Formulários e Eventos

1. Validação de dados

```
const form = document.querySelector("form");
form.addEventListener("submit", (e) => {
  e.preventDefault();
  const nome = form.nome.value;
  if (nome.length < 3) {
    alert("Nome muito curto!");
    return;
  }
  console.log("Dados enviados!");
});
```

2. Enviar formulários com fetch

```
async function enviarDados(dados) {
  const res = await fetch("/api", {
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify(dados)
  });
  const resposta = await res.json();
  console.log(resposta);
}
```

3. Eventos comuns

```
document.addEventListener("keydown", e => console.log(e.key));
```

```
document.querySelector("button").addEventListener("click", () => alert("Clicado!"));
```

Fetch API

1. Requisições HTTP

```
fetch("https://jsonplaceholder.typicode.com/posts")
  .then(res => res.json())
  .then(data => console.log(data))
  .catch(err => console.error("Erro:", err));
```

2. Trabalhar com JSON

```
const jsonData = JSON.stringify({ nome: "João" });
const obj = JSON.parse(jsonData);
```

3. Tratamento de erros

```
try {
  // algum código
} catch (erro) {
  console.error("Erro encontrado:", erro);
}
```

Armazenamento Local

localStorage e sessionStorage

```
localStorage.setItem("tema", "escuro");
const tema = localStorage.getItem("tema");
localStorage.removeItem("tema");
```

Cookies (básico)

```
document.cookie = "usuario=Joana; expires=Fri, 31 Dec 2025 23:59:59 UTC";
```

Debugging e Ferramentas

1. DevTools

Use F12 no navegador para inspecionar, debugar, ver erros e logs.

2. `console.log()` e `debugger`

```
console.log("Estado atual:", variavel);  
debugger; // Pausa execução no DevTools
```

3. Linters (ESLint)

- Instale ESLint no VS Code
- Ajuda a encontrar erros e seguir boas práticas de estilo

```
npm install eslint --save-dev  
npx eslint script.js
```

Atividades Práticas

1. Crie uma página com um formulário de login que valide e exiba dados no console.
2. Consuma a API de "<https://jsonplaceholder.typicode.com/posts>" e exiba títulos.

3. Armazene a preferência de tema da página (claro/escuro) no `localStorage`.
 4. Adicione um botão que imprime dados via `console.log()` ao clicar.
-



Conclusão

Nesta fase intermediária você aprendeu a lidar com recursos modernos do JavaScript, manipular dados reais com `fetch`, armazenar preferências e usar ferramentas profissionais para escrever um código melhor.



Aula: ES6+ Features



Objetivo

- Compreender os principais recursos introduzidos nas versões modernas do JavaScript (ES6+)
 - Aplicar esses recursos na prática para tornar o código mais limpo, legível e moderno
-

Tópicos abordados

1. Desestruturação (Destructuring)

Permite extrair valores de objetos e arrays de forma prática:

// Objeto

```
const pessoa = { nome: "João", idade: 25 };  
const { nome, idade } = pessoa;
```

// Array

```
const numeros = [1, 2, 3];  
const [a, b, c] = numeros;
```

✓ *Facilita acesso e legibilidade*

2. Template Literals (Strings com crase)

Permite criar strings com variáveis de forma elegante:

```
const produto = "Caderno";  
const preco = 12.5;  
console.log(`O ${produto} custa R${preco}`);
```

✓ *Usa crase (`) e `${}` para inserir variáveis*

3. Arrow Functions

Funções mais curtas e com escopo de `this` preservado:

```
const soma = (a, b) => a + b;
```

```
const saudacao = nome => `Olá, ${nome}`;
```

✓ *Mais concisas e ideais para callbacks*

4. Classes e Herança

Abstração moderna baseada em protótipos:

```
class Pessoa {  
  constructor(nome) {  
    this.nome = nome;  
  }  
  falar() {  
    console.log(`${this.nome} está falando.`);  
  }  
}
```

```
class Aluno extends Pessoa {  
  estudar() {  
    console.log(`${this.nome} está estudando.`);  
  }  
}
```

✓ *Clara e reutilizável*

5. Módulos (import/export)

Permite dividir o código em múltiplos arquivos:

```
// arquivo: utils.js  
export function dobrar(x) {  
  return x * 2;  
}
```

```
// arquivo: main.js  
import { dobrar } from './utils.js';  
console.log(dobrar(4));
```

✓ *Modularização e reaproveitamento de código*

6. Promises e async/await

✓ **Promises:**

```
fetch("https://jsonplaceholder.typicode.com/posts")
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error("Erro:", error));
```

✓ **async/await:**

```
async function carregarDados() {
  try {
    const response = await
    fetch("https://jsonplaceholder.typicode.com/posts");
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error("Erro:", error);
  }
}
```

✓ *Facilita código assíncrono, principalmente requisições de APIs*



Atividades Práticas

1. Crie uma função que recebe um objeto e desestrutura **nome** e **idade**.

2. Use `template literals` para imprimir uma mensagem com essas variáveis.
3. Escreva uma `arrow function` que retorna o dobro de um número.
4. Implemente uma classe `Carro` com método `acelerar()`.
5. Crie dois arquivos: um com função `soma()` e outro que a importa e usa.
6. Faça uma requisição com `async/await` usando `fetch()` e exiba os dados no console.



Conclusão

Esses recursos do ES6+ são hoje **essenciais** para qualquer desenvolvedor moderno. Eles **aumentam a produtividade, melhoram a legibilidade e encorajam boas práticas.**

Aula: Manipulação de Formulários e Eventos

Objetivo

- Aprender a **capturar e manipular eventos** em formulários
- Validar dados de entrada com JavaScript
- Enviar formulários e lidar com respostas
- Trabalhar com **eventos de teclado, mouse e submit**

Estrutura da Aula

1. Capturando o evento **submit**

```
<form id="meuFormulario">  
  <input type="text" name="nome" placeholder="Seu nome" />  
  <button type="submit">Enviar</button>  
</form>
```

```
const form = document.getElementById("meuFormulario");
```

```
form.addEventListener("submit", function (event) {  
  event.preventDefault(); // impede recarregamento  
  const nome = form.nome.value;  
  alert(`Olá, ${nome}`);  
});
```

2. Validação de dados

```
form.addEventListener("submit", function (event) {  
  event.preventDefault();  
  const nome = form.nome.value.trim();  
  
  if (nome === "") {  
    alert("O nome não pode estar vazio.");  
    return;  
  }  
  
  if (nome.length < 3) {  
    alert("O nome deve ter ao menos 3 caracteres.");  
    return;  
  }  
  
  alert(`Olá, ${nome}!`);  
});
```

3. Enviando formulário com **fetch**

```
form.addEventListener("submit", async function (event) {  
  event.preventDefault();  
  
  const dados = {  
    nome: form.nome.value,  
  };  
  
  try {  
    const response = await fetch("/api/enviar", {  
      method: "POST",  
      headers: { "Content-Type": "application/json" },  
      body: JSON.stringify(dados),  
    });  
  });
```

```
const resultado = await response.json();
alert("Dados enviados com sucesso!");
} catch (erro) {
  console.error("Erro ao enviar:", erro);
}
});
```

4. Eventos de mouse e teclado

Clique em botão

```
document.querySelector("button").addEventListener("click", () => {
  console.log("Botão clicado!");
});
```

Pressionar tecla

```
document.addEventListener("keydown", (e) => {
  console.log(`Tecla pressionada: ${e.key}`);
});
```

Detectar mudanças no input

```
form.nome.addEventListener("input", () => {
  console.log("Valor atual:", form.nome.value);
});
```

Dicas úteis

- Use `.trim()` para limpar espaços desnecessários
- Use `.toLowerCase()` ou `.toUpperCase()` para normalizar entradas

- Combine `classList.add/remove/toggle` para feedback visual
-



Exercícios Práticos

1. Faça um formulário com campo de e-mail e senha que valida os dados.
 2. Adicione um evento de teclado que exibe a tecla digitada em tempo real.
 3. Adicione validação que impede envio se os campos estiverem vazios.
 4. Envie os dados via `fetch()` e mostre a resposta na tela.
 5. Crie feedback visual: campo inválido fica com borda vermelha.
-



Conclusão

Manipular formulários e eventos permite criar experiências de usuário mais dinâmicas e confiáveis. Essa habilidade é indispensável para qualquer aplicação web real.



Aula: Fetch API

Objetivo

- Entender como fazer **requisições HTTP** com `fetch()`
- Trabalhar com **JSON**
- Tratar **erros** corretamente
- Integrar o `fetch()` com formulários e exibição de dados

1. O que é `fetch()`?

`fetch()` é uma função nativa do JavaScript moderno para fazer **requisições assíncronas** a servidores (APIs).

Ele retorna uma **Promise**, permitindo usar `.then()` ou `async/await`.

Exemplo básico com `.then()`

```
fetch("https://jsonplaceholder.typicode.com/posts")  
  .then(response => response.json())  
  .then(data => console.log(data))  
  .catch(error => console.error("Erro:", error));
```

Usando `async/await`

```
async function carregarPosts() {  
  try {  
    const response = await  
    fetch("https://jsonplaceholder.typicode.com/posts");  
    const posts = await response.json();  
    console.log(posts);  
  } catch (erro) {  
    console.error("Erro ao carregar posts:", erro);  
  }  
}
```

2. Trabalhando com JSON

Enviando dados com **POST**

```
async function enviarDados() {  
  const dados = {  
    nome: "Ana",  
    idade: 30  
  };  
  
  try {  
    const resposta = await  
    fetch("https://jsonplaceholder.typicode.com/posts", {
```

```
method: "POST",  
headers: {  
  "Content-Type": "application/json"  
},  
body: JSON.stringify(dados)  
});
```

```
const resultado = await resposta.json();  
console.log("Enviado:", resultado);  
} catch (erro) {  
  console.error("Erro no envio:", erro);  
}  
}
```



3. Tratamento de Erros

```
async function buscar() {  
  try {  
    const res = await fetch("https://exemplo.com/api");  
    if (!res.ok) throw new Error(`HTTP error! status: ${res.status}`);  
    const data = await res.json();  
    console.log(data);  
  }  
}
```

```
} catch (err) {  
  console.error("Erro ao buscar:", err.message);  
}  
}
```



Atividades Práticas

1. Crie uma página que carrega e exibe os títulos dos posts da API <https://jsonplaceholder.typicode.com/posts>
 2. Adicione um formulário com nome e email, e envie esses dados via **POST**
 3. Exiba mensagens de erro se a requisição falhar (ex: erro de rede, status 404)
 4. Bonus: mostre um loader (“Carregando...”) enquanto aguarda resposta
-



Conclusão







A Fetch API é essencial no desenvolvimento web moderno, pois permite integrar **APIs externas** e criar aplicações **dinâmicas e interativas**. Combinada com **async/await** e tratamento de erros, oferece uma experiência robusta e elegante para desenvolvedores.

Aula: Armazenamento Local no Navegador

Objetivo

- Aprender a armazenar, recuperar e remover dados usando `localStorage`, `sessionStorage` e cookies
 - Entender a diferença entre os tipos de armazenamento
 - Aplicar esses conhecimentos em exemplos práticos
-

Tipos de Armazenamento

Tipo	Persiste após fechar o navegador?	Acessível por JS?	Expira automaticamente?
<code>localStorage</code>	 Sim	 Sim	 Não
<code>sessionStorage</code>	 Não (só durante a aba atual)	 Sim	 Sim (fecha aba)

Cookies

✓ Sim (até data de
expiração)

✓ Sim

✓ Sim
(configurável)



1. localStorage



Salvar dado

```
localStorage.setItem("usuario", "Ana");
```



Ler dado

```
const nome = localStorage.getItem("usuario");  
console.log(nome); // "Ana"
```



Remover dado

```
localStorage.removeItem("usuario");
```



Limpar tudo

```
localStorage.clear();
```



2. sessionStorage

Tem os mesmos métodos do `localStorage`, mas os dados só duram enquanto a aba estiver aberta.

```
sessionStorage.setItem("sessao", "ativo");  
console.log(sessionStorage.getItem("sessao")); // "ativo"
```



3. Cookies

✓ Criar cookie

```
document.cookie = "tema=escuro; expires=Fri, 31 Dec 2025 23:59:59 GMT; path=/";
```

✓ Ler cookies

```
console.log(document.cookie); // "tema=escuro"
```

⚠ Cookies têm limite de tamanho (~4KB) e são enviados ao servidor a cada requisição.



Projeto Prático: Tema Escuro com localStorage

HTML

```
<button id="alternarTema">Alternar Tema</button>
```

JavaScript

```
const botao = document.getElementById("alternarTema");
```

```
botao.addEventListener("click", () => {  
  const temaAtual = localStorage.getItem("tema") || "claro";  
  const novoTema = temaAtual === "claro" ? "escuro" : "claro";  
  document.body.className = novoTema;  
  localStorage.setItem("tema", novoTema);  
});
```

```
// Aplicar o tema salvo ao carregar
```

```
document.body.className = localStorage.getItem("tema") || "claro";
```



Conclusão

Usar o armazenamento local te permite:

- **Persistir preferências do usuário**
- **Evitar chamadas desnecessárias à API**
- **Melhorar a experiência do usuário**

Para projetos reais, o `localStorage` é a melhor opção para dados que não precisam ser sigilosos. Cookies são ideais para autenticação e integração com servidores.



Aula: Debugging e Ferramentas de Desenvolvimento



Objetivo

- Aprender a usar as ferramentas de desenvolvedor do navegador

- Identificar e corrigir erros no código
 - Usar o `console`, `debugger` e linters como o ESLint para escrever código mais limpo e confiável
-

Conteúdo da Aula

1. Ferramentas do Desenvolvedor (DevTools)

♦ Acessando:

- Chrome / Edge: clique com botão direito → **"Inspecionar"**
- Atalho: `F12` ou `Ctrl + Shift + I`

♦ Principais abas:

- **Console**: erros, logs e comandos JS em tempo real
 - **Elements**: HTML e CSS da página
 - **Network**: monitorar requisições
 - **Sources**: visualizar e depurar JS
 - **Application**: gerenciar localStorage, cookies etc.
-

2. `console.log()` e variações

```
console.log("Mensagem comum");  
console.error("Erro!");
```

```
console.warn("Aviso!");  
console.info("Informação");  
console.table([ { nome: "Ana" }, { nome: "João" } ]);
```

✅ Útil para inspecionar variáveis e acompanhar a execução

3. 🐛 Usando **debugger**

Insere um ponto de parada manual no código:

```
function soma(a, b) {  
  debugger; // a execução para aqui  
  return a + b;  
}
```

```
soma(5, 3);
```

No DevTools (aba *Sources*), você pode:

- Adicionar **breakpoints**
 - Ver o **call stack**
 - Observar **valores em tempo real**
-

4. 🧹 **ESLint (Linting)**

O ESLint analisa seu código e avisa sobre:

- Erros de sintaxe
- Má formatação

- Problemas de estilo ou más práticas

- ◆ **Exemplo de regra:**

```
// Erro com ESLint: variável não usada
const nome = "Carlos";
```

- ◆ **Configuração básica (`.eslintrc.json`)**

```
{
  "env": {
    "browser": true,
    "es2021": true
  },
  "extends": "eslint:recommended",
  "parserOptions": {
    "ecmaVersion": 12
  },
  "rules": {
    "no-unused-vars": "warn",
    "semi": ["error", "always"]
  }
}
```

✓ Pode ser integrado com editores como VS Code



Atividades Práticas

1. Abra o DevTools, vá até a aba **Console** e use `console.log()` para mostrar uma mensagem.
2. Crie uma função com `debugger` e execute passo a passo no navegador.

3. Instale o ESLint e veja sugestões no seu código.
 4. Ative breakpoints na aba *Sources* e observe valores de variáveis.
 5. Explore o `localStorage` e cookies pela aba *Application*.
-




Conclusão

Saber **depurar e testar seu código** com as ferramentas certas é tão importante quanto escrevê-lo. Quanto mais cedo você dominar o uso de `console`, `debugger`, DevTools e linters, mais rápido vai se tornar um desenvolvedor eficaz.



Fase 3: JavaScript Avançado

 **Objetivo:** Tornar-se apto a construir aplicações robustas, seguras e testáveis com JavaScript moderno.



1. Programação Assíncrona Avançada



Event Loop

- O mecanismo que gerencia a execução do código, filas de eventos e mensagens assíncronas.
- Exemplo:

```
console.log("Início");
```

```
setTimeout(() => {  
  console.log("setTimeout");  
}, 0);
```

```
Promise.resolve().then(() => {  
  console.log("Promise");  
});
```

```
console.log("Fim");
```

Saída: Início → Fim → Promise → setTimeout
Explica **microtasks** vs **macrotasks**.

Microtasks vs Macrotasks

Tipo	Exemplo	Prioridade
Microtask	<code>.then()</code> , <code>await</code>	Alta
Macrotask	<code>setTimeout()</code> , <code>setInterval()</code>	Baixa

Promise.all, Promise.race

```
const p1 = fetch("/api/dado1");  
const p2 = fetch("/api/dado2");
```

```
Promise.all([p1, p2]).then(([res1, res2]) => {  
  console.log("Todos prontos");  
});
```

```
Promise.race([p1, p2]).then((primeiro) => {  
  console.log("Primeira resposta chegou");  
});
```

2. Design Patterns em JavaScript

Module Pattern

```
const contador = (function () {  
  let valor = 0;  
  return {  
    incrementar: () => ++valor,  
    obter: () => valor  
  };  
})();
```

Singleton Pattern

```
class Configuracao {  
  constructor() {  
    if (Configuracao.instance) {  
      return Configuracao.instance;  
    }  
    this.tema = "claro";  
    Configuracao.instance = this;  
  }  
}
```

Factory Pattern

```
function criarUsuario(nome, admin = false) {
```

```
return {
  nome,
  admin,
  saudacao() {
    return `Olá, ${this.nome}`;
  }
};
}
```

👁️ Observer Pattern

```
class Evento {
  constructor() {
    this.ouvintes = [];
  }

  ouvir(fn) {
    this.ouvintes.push(fn);
  }

  disparar(dado) {
    this.ouvintes.forEach(fn => fn(dado));
  }
}
```

✅ 3. Testes com Jest

🧪 Teste Unitário

```
// soma.js
function soma(a, b) {
  return a + b;
}
module.exports = soma;
```

```
// soma.test.js
```

```
const soma = require("./soma");
```

```
test("soma 1 + 2 = 3", () => {  
  expect(soma(1, 2)).toBe(3);  
});
```

Teste de Integração

- Testar múltiplos módulos juntos (ex: login + API + sessão).
 - Usa **supertest**, **jest**, e mocks.
-

4. Segurança em JavaScript

XSS (Cross-site Scripting)

```
<!-- Perigoso -->  
<div id="mensagem"></div>  
<script>  
  const mensagem = "<img src=x onerror=alert('XSS')>";  
  document.getElementById("mensagem").innerHTML = mensagem;  
</script>
```

Solução: sanitize ou use **.textContent**

```
document.getElementById("mensagem").textContent = mensagem;
```

CSRF (Cross-Site Request Forgery)

- Enviar requisições usando sessões autenticadas do usuário sem consentimento.

- Proteções: tokens CSRF, CORS, **SameSite** cookies.

Sanitização de Inputs

```
function sanitizar(texto) {  
  return texto.replace(/</g, "&lt;").replace(/>/g, "&gt;");  
}
```

5. Ferramentas de Build: Webpack e Babel

Webpack

- Empacotador de módulos que transforma JS, CSS e imagens em bundles otimizados.

```
// webpack.config.js  
module.exports = {  
  entry: "./src/index.js",  
  output: {  
    filename: "bundle.js",  
    path: __dirname + "/dist"  
  },  
  module: {  
    rules: [  
      { test: /\.js$/, exclude: /node_modules/, use: "babel-loader" }  
    ]  
  }  
};
```

Babel

- Compila JavaScript moderno para versões mais compatíveis (ES6 → ES5)

```
npm install @babel/core @babel/preset-env babel-loader
```

```
// .babelrc
{
  "presets": ["@babel/preset-env"]
}
```

Atividades Práticas

1. Crie um app que carrega vários dados via `Promise.all`.
2. Refatore seu código usando o padrão **Factory**.
3. Escreva testes unitários com Jest para funções principais.
4. Identifique e corrija falhas XSS num input com `innerHTML`.
5. Crie uma configuração básica do Webpack para um projeto real.

Conclusão

Nesta fase, você ganha **maturidade técnica** com JavaScript: entende como a linguagem funciona internamente, aplica **padrões de projeto**, testa e protege seus sistemas, e configura pipelines modernas com Webpack e Babel.

Aula: Programação Assíncrona em JavaScript

Objetivo

- Entender como o JavaScript lida com operações assíncronas
 - Aprender o funcionamento do **Event Loop**
 - Usar `Promises`, `async/await`, `Promise.all()` e `Promise.race()`
 - Diferenciar **microtasks** e **macrotasks**
-

1. O que é Programação Assíncrona?

JavaScript é **single-threaded**, mas não bloqueante. Operações como requisições HTTP, timers, ou leitura de arquivos são executadas fora da thread principal e retornam seus resultados mais tarde.

2. Event Loop

O **Event Loop** é o mecanismo responsável por:

1. Executar o código da pilha principal
2. Verificar a **fila de microtasks** (ex: `Promise.then`)
3. Depois, verificar a **fila de macrotasks** (ex: `setTimeout`)



Exemplo prático:

```
console.log("Início");
```

```
setTimeout(() => console.log("setTimeout"), 0);
```

```
Promise.resolve().then(() => console.log("Promise"));
```

```
console.log("Fim");
```

Saída:

Início

Fim

Promise

setTimeout



3. Promises

Uma **Promise** representa uma operação assíncrona que pode **resolver** (sucesso) ou **rejeitar** (erro).



Exemplo:

```
const promessa = new Promise((resolve, reject) => {  
  setTimeout(() => resolve("Sucesso!"), 1000);  
});
```

```
promessa.then(resultado => console.log(resultado));
```

4. Async / Await

Azucar sintático sobre Promises. Permite escrever código assíncrono como se fosse síncrono.

```
async function carregarDados() {  
  try {  
    const resposta = await  
    fetch("https://jsonplaceholder.typicode.com/posts/1");  
    const dados = await resposta.json();  
    console.log(dados);  
  } catch (erro) {  
    console.error("Erro:", erro);  
  }  
}
```

5. Microtasks vs Macrotasks

Tipo	Exemplos	Executado após
Microtask	<code>Promise.then</code> , <code>await</code>	código síncrono
Macrotask	<code>setTimeout</code> , <code>setInterval</code> , <code>events</code>	microtasks DOM

6. Promise.all / Promise.race

 **Promise.all**

Espera **todas** as Promises finalizarem:

```
const p1 = fetch("/api1");  
const p2 = fetch("/api2");
```

```
Promise.all([p1, p2]).then(([res1, res2]) => {  
  console.log("Ambas completas");  
});
```

Promise.race

Retorna o primeiro resultado (sucesso ou erro):

```
Promise.race([p1, p2]).then((respostaMaisRápida) => {  
  console.log("Chegou primeiro:", respostaMaisRápida);  
});
```

Atividade Prática

1. Crie uma **Promise** que resolve após 2 segundos com uma mensagem.
2. Use **async/await** para buscar dados de uma API pública (ex: <https://jsonplaceholder.typicode.com/posts>).
3. Teste o comportamento de **Promise.all** e **Promise.race** com **setTimeout**.
4. Inspecione a ordem de execução entre **setTimeout** e **Promise** usando DevTools.

Conclusão

A programação assíncrona é o **coração do JavaScript moderno**, permitindo interfaces responsivas e integração com APIs externas. Dominar **Promises**, **async/await** e o Event Loop vai te dar mais controle e previsibilidade ao lidar com operações complexas.

Aula: Design Patterns em JavaScript

Objetivo

- Entender e aplicar **padrões de projeto** em JavaScript
 - Melhorar a estrutura e manutenção do código
 - Compreender padrões como **Module**, **Singleton**, **Factory** e **Observer**
-

1. O que são Design Patterns?

Design Patterns são soluções reutilizáveis para problemas comuns de design em software. Eles ajudam a tornar o código mais **modular**, **flexível** e **manutenível**.

Tipos comuns de Design Patterns:

- **Criação**: Como objetos são criados (ex: Factory, Singleton)
- **Estrutural**: Como as classes e objetos estão organizados (ex: Module)

- **Comportamentais:** Como as interações entre objetos acontecem (ex: Observer)
-



2. Module Pattern

O **Module Pattern** permite organizar o código em módulos que **encapsulam** funcionalidades e dados privados. Ele ajuda a evitar o uso excessivo do escopo global.

Exemplo:

```
const contador = (function () {  
  let valor = 0; // variável privada  
  
  return {  
    incrementar: function () {  
      valor++;  
      return valor;  
    },  
    obterValor: function () {  
      return valor;  
    }  
  };  
})();  
  
console.log(contador.incrementar()); // 1  
console.log(contador.obterValor()); // 1
```

Vantagens:

- **Encapsulamento** de dados
- **Evita poluição do escopo global**

- Organiza o código de forma modular
-

3. Singleton Pattern

O **Singleton** assegura que uma classe tenha **somente uma instância** e fornece um ponto de acesso global a essa instância.

Exemplo:

```
class Configuracao {  
  constructor() {  
    if (Configuracao.instance) {  
      return Configuracao.instance;  
    }  
    this.tema = "claro";  
    Configuracao.instance = this;  
  }  
  
  alterarTema(novoTema) {  
    this.tema = novoTema;  
  }  
}
```

```
const config1 = new Configuracao();  
const config2 = new Configuracao();  
  
console.log(config1 === config2); // true
```

Vantagens:

- Garantia de **uma única instância** global.
- Útil para gerenciar **configurações** ou **recursos compartilhados**.



4. Factory Pattern

O **Factory Pattern** fornece uma maneira de **criar objetos** sem precisar especificar a classe exata do objeto. Ele usa uma função que retorna diferentes tipos de objetos com base em parâmetros.

Exemplo:

```
function criarCarro(tipo) {  
  if (tipo === "esportivo") {  
    return { tipo: "esportivo", acelerar() { console.log("Acelerando rápido!"); } };  
  } else if (tipo === "sedan") {  
    return { tipo: "sedan", acelerar() { console.log("Acelerando com conforto!"); } };  
  }  
}
```

```
const carro1 = criarCarro("esportivo");  
const carro2 = criarCarro("sedan");
```

```
carro1.acelerar(); // Acelerando rápido!  
carro2.acelerar(); // Acelerando com conforto!
```

Vantagens:

- **Flexibilidade** para criar diferentes tipos de objetos.
- Centraliza a **lógica de criação** de objetos.



5. Observer Pattern

O **Observer Pattern** define uma dependência de um-para-muitos, onde **objetos observadores** são notificados de mudanças no objeto observado.

Exemplo:

```
class Evento {  
  constructor() {  
    this.ouvintes = [];  
  }  
  
  ouvir(fn) {  
    this.ouvintes.push(fn);  
  }  
  
  disparar(dado) {  
    this.ouvintes.forEach(fn => fn(dado));  
  }  
}  
  
const evento = new Evento();  
  
// Observadores  
evento.ouvir((dados) => console.log("Ouvinte 1:", dados));  
evento.ouvir((dados) => console.log("Ouvinte 2:", dados));  
  
// Disparando o evento  
evento.disparar("Evento disparado!");
```

Saída:

Ouvinte 1: Evento disparado!
Ouvinte 2: Evento disparado!

Vantagens:

- **Desacoplamento** entre os objetos.
 - Notifica múltiplos observadores de mudanças em um objeto.
-



6. Atividades Práticas

1. **Module Pattern**: Crie um módulo para gerenciar o estado de um contador (com incremento, decremento e reset).
 2. **Singleton Pattern**: Implemente um sistema de **configuração de aplicativo** usando o Singleton.
 3. **Factory Pattern**: Crie uma fábrica para gerar diferentes tipos de **usuários** (admin, normal, visitante) com funcionalidades específicas.
 4. **Observer Pattern**: Crie um **sistema de notificações** onde vários observadores (usuários) são notificados sobre um evento importante (ex: novo post).
-



Conclusão

Design Patterns são essenciais para criar um código mais **estruturado**, **modular** e **manutenível**. Ao aplicar padrões como **Module**, **Singleton**, **Factory** e **Observer**, você organiza melhor seu projeto e melhora a colaboração entre os componentes.

Esses padrões são amplamente usados em aplicações de grande escala, como frameworks e bibliotecas populares, para garantir **flexibilidade** e **manutenibilidade** do código.

Aula: Testes em JavaScript

Objetivo

- Compreender a importância dos **testes** em JavaScript
 - Aprender a escrever **testes unitários** e **testes de integração**
 - Usar o framework **Jest** para facilitar o processo de testes
-

1. O que são Testes?

Testes são uma maneira de garantir que seu código está **funcionando corretamente**, detectando falhas e garantindo a qualidade do software.

- **Testes Unitários:** Testam unidades específicas do código (funções, métodos, classes).
 - **Testes de Integração:** Testam como várias partes do sistema funcionam juntas.
-



2. Por que Testar?

- **Reduzir bugs:** Ao detectar problemas logo no início.
 - **Facilitar mudanças:** Alterações no código são seguras quando você tem uma boa cobertura de testes.
 - **Documentação:** Testes podem servir como **documentação viva** de como o código deve funcionar.
-



3. Testes Unitários com Jest



O que é o Jest?

Jest é um framework de testes para JavaScript, criado pelo Facebook. Ele facilita a criação de testes unitários e de integração, oferecendo funcionalidades como:

- **Assersções:** Métodos para comparar resultados esperados e obtidos.
- **Mocks:** Substituir funções externas por versões controladas em testes.
- **Snapshots:** Comparar saídas de funções ou componentes com uma versão armazenada.



Instalando o Jest

```
npm init -y  
npm install --save-dev jest
```

No arquivo `package.json`, adicione o seguinte script:

```
"scripts": {  
  "test": "jest"  
}
```

Agora você pode rodar seus testes com o comando `npm test`.



Exemplo de Teste Unitário

```
// soma.js  
function soma(a, b) {  
  return a + b;  
}  
  
module.exports = soma;
```

Agora, crie o arquivo de teste:

```
// soma.test.js  
const soma = require('./soma');  
  
test('soma 1 + 2 deve ser igual a 3', () => {  
  expect(soma(1, 2)).toBe(3);  
});
```

Rodando o comando `npm test`, o Jest executa o teste e verifica se a soma está correta.



Algumas Funções do Jest:

- `test()` – Define um caso de teste.
- `expect()` – Faz uma **assertiva** sobre o valor.
- `.toBe()` – Verifica se o valor é igual ao esperado.

- `.toBeNull()`, `.toBeTruthy()`, `.toBeGreaterThan()` – Vários tipos de asserções.
-

4. Testes de Integração

Testes de integração garantem que diferentes partes do sistema funcionem corretamente quando combinadas.

Exemplo de Teste de Integração

Imaginemos um sistema simples onde temos um módulo para buscar dados de um banco de dados fictício.

```
// banco.js
const fetchDados = () => {
  return [
    { id: 1, nome: 'João' },
    { id: 2, nome: 'Maria' },
  ];
};

module.exports = { fetchDados };
```

Agora, testamos a integração entre a função que busca dados e a função que usa esses dados:

```
// usuario.test.js
const { fetchDados } = require('./banco');

test('deve retornar um usuário com id 1', () => {
  const dados = fetchDados();
  const usuario = dados.find(u => u.id === 1);
  expect(usuario.nome).toBe('João');
});
```

5. Mocking de Funções

Às vezes, você precisa **simular** funções externas para testar apenas o seu código. O Jest oferece uma maneira fácil de fazer isso.

Exemplo de Mocking:

```
// usuario.js
const fetchDados = require('./banco');

function buscarUsuario(id) {
  const usuarios = fetchDados();
  return usuarios.find(u => u.id === id);
}

module.exports = buscarUsuario;
```

No teste, vamos mockar **fetchDados**:

```
// usuario.test.js
const buscarUsuario = require('./usuario');
jest.mock('./banco', () => ({
  fetchDados: jest.fn(() => [
    { id: 1, nome: 'João' },
    { id: 2, nome: 'Maria' }
  ])
}));

test('deve retornar o usuário com id 1', () => {
  const usuario = buscarUsuario(1);
  expect(usuario.nome).toBe('João');
});
```

6. Testes de Snapshot

Jest pode tirar **snapshots** do retorno de funções ou componentes e compará-los com versões anteriores.

Exemplo de Snapshot:

```
// função que retorna um objeto
function gerarPessoa(nome) {
  return { nome };
}

test('snapshot da pessoa', () => {
  const pessoa = gerarPessoa('Carlos');
  expect(pessoa).toMatchSnapshot();
});
```

Jest irá criar um arquivo de snapshot e, em execuções futuras, comparará o valor retornado com o valor armazenado.

7. Cobertura de Testes

Jest oferece a funcionalidade de gerar relatórios sobre a **cobertura de testes** do seu código.

Rodando com cobertura:

```
npm test -- --coverage
```

Jest gera um relatório sobre quais partes do seu código foram testadas.

8. Atividades Práticas

1. Crie um módulo simples com funções que fazem cálculos (soma, subtração, multiplicação).
 2. Escreva **testes unitários** para cada função utilizando o Jest.
 3. Implemente uma função que faz requisição para uma API fictícia (pode mockar com `jest.fn()`).
 4. Realize **testes de integração** verificando a interação entre a API e os componentes do sistema.
 5. Explore o uso de **snapshots** em componentes ou objetos complexos.
-



Conclusão

Testar seu código é uma prática essencial para garantir a **qualidade** e **confiabilidade** do seu software. O Jest facilita a criação de testes unitários e de integração, oferecendo uma ampla gama de funcionalidades, como mocks, snapshots e cobertura de código.

Com o domínio de testes, você pode escrever aplicações mais robustas e seguras, além de facilitar a manutenção ao longo do tempo.



Aula: Segurança em JavaScript

Objetivo

- Compreender as vulnerabilidades mais comuns em aplicações web
- Aprender como prevenir **XSS**, **CSRF** e outras ameaças
- Aplicar boas práticas de segurança ao lidar com dados do usuário

1. O que é Segurança Web?

Segurança web refere-se à proteção de aplicativos web contra ataques que possam comprometer a **integridade**, **confidencialidade** ou **disponibilidade** dos dados ou sistemas.

As vulnerabilidades mais comuns incluem **XSS**, **CSRF**, **injeção de SQL**, **falsificação de requisições** e **exposição de dados sensíveis**.

2. Cross-Site Scripting (XSS)

XSS ocorre quando um atacante consegue injetar **código JavaScript malicioso** em páginas web visualizadas por outros usuários. Isso pode permitir o **roubo de cookies**, **modificação do conteúdo da página** ou **execução de ações em nome do usuário**.

Tipos de XSS:

- **Reflexivo**: O código malicioso é enviado como parte de uma requisição HTTP (ex: URL).

- **Armazenado:** O código malicioso é armazenado no servidor e executado quando a página é carregada.
- **DOM-based:** O código é injetado e executado diretamente pelo JavaScript no navegador do usuário.

Como prevenir:

1. **Escape de caracteres especiais:** Escapar caracteres como `<`, `>`, `"`, `'`, etc., antes de exibí-los na página.
2. **Content Security Policy (CSP):** Uma medida de segurança que ajuda a mitigar XSS ao restringir fontes confiáveis para scripts.
3. **Evitar `eval()`:** Nunca use `eval()` em seu código, pois ele pode executar código malicioso.

Exemplo de prevenção de XSS:

// Função para escapar caracteres especiais

```
function escaparHTML(str) {  
  return str.replace(/[&<>'"`]/g, (match) => {  
    const map = {  
      '&': '&amp;',  
      '<': '&lt;',  
      '>': '&gt;',  
      '"': '&quot;',  
      "'": '&#x27;',  
      '`': '&#x60;'  
    };  
    return map[match];  
  });  
}
```

```
    return map[match];  
  });  
}  
  
const usuarioInput = "<script>alert('XSS!');</script>";  
const seguro = escaparHTML(usuarioInput);  
console.log(seguro); // Saída segura para exibir no HTML
```

3. Cross-Site Request Forgery (CSRF)

CSRF é um ataque onde um usuário autenticado em um site é induzido a realizar uma ação **não desejada** em outro site. Por exemplo, um atacante pode forjar uma requisição HTTP de um usuário autenticado para transferir dinheiro ou mudar a senha de sua conta.

Como prevenir:

1. **Tokens CSRF:** Enviar um token único e não previsível junto com cada requisição de modificação de dados. Esse token é validado no servidor.
2. **Verificação de Origem:** Verificar o cabeçalho **Origin** ou **Referer** para garantir que a requisição venha de uma origem confiável.
3. **SameSite Cookies:** Usar a propriedade **SameSite** nos cookies para evitar que sejam enviados em requisições de sites de terceiros.

Exemplo de prevenção com token CSRF:

// Em um formulário, envie um token CSRF no cabeçalho ou no corpo da requisição:

```
const tokenCSRF = document.getElementById('csrf-token').value;

fetch('/transferir', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
    'CSRF-Token': tokenCSRF
  },
  body: JSON.stringify({ valor: 100, contaDestino: '123456' })
});
```

4. Sanitização de Inputs

Ao lidar com dados de usuários (especialmente os que vêm de formulários), é fundamental **sanitizar** e **validar** esses dados para evitar a execução de código malicioso ou a injeção de dados inesperados.

Como sanitizar:

- **Remover tags HTML:** Usar bibliotecas como **DOMPurify** para limpar entradas de dados que possam conter HTML malicioso.

- **Validar tipos de dados:** Verificar se os dados estão no formato esperado antes de processá-los.

Exemplo de sanitização usando DOMPurify:

// Instale a biblioteca DOMPurify: npm install dompurify

```
import DOMPurify from 'dompurify';
```

```
const inputUsuario = '';
```

```
const inputSeguro = DOMPurify.sanitize(inputUsuario);
```

```
console.log(inputSeguro); // Saída: 
```

Boa prática:

- **Nunca confiar cegamente em dados do usuário.** Sempre valide e sanitize entradas antes de usá-las.



5. Outras Boas Práticas de Segurança

1. **Uso de HTTPS:** Sempre use **HTTPS** para garantir que a comunicação entre o cliente e o servidor seja **criptografada** e segura.
2. **Armazenamento seguro de senhas:** Use algoritmos de hash seguros, como **bcrypt**, para armazenar senhas. Nunca armazene senhas em texto claro.
3. **Headers de segurança HTTP:** Utilize cabeçalhos como **X-Content-Type-Options**, **X-Frame-Options** e **Strict-Transport-Security** para melhorar a segurança da

aplicação.

4. **Controle de sessões:** Defina um tempo de expiração para sessões e use cookies seguros (com flag `HttpOnly` e `Secure`).



6. Atividades Práticas

1. **Evite XSS:** Crie um formulário que recebe entrada do usuário e exibe em uma página. Aplique medidas de segurança para evitar XSS.
2. **Proteja contra CSRF:** Implemente um sistema de tokens CSRF em seu backend e crie um formulário que envie esse token junto com as requisições de modificação.
3. **Sanitize Input:** Use a biblioteca DOMPurify para limpar os dados de entrada de um formulário antes de usá-los.
4. **Implementação de segurança HTTP:** Simule um servidor e configure cabeçalhos de segurança como `Strict-Transport-Security` e `X-Content-Type-Options`.



Conclusão

A segurança em JavaScript é um aspecto fundamental no desenvolvimento web. Proteção contra vulnerabilidades como **XSS**, **CSRF** e a sanitização de entradas são passos cruciais para garantir a segurança de suas aplicações e a integridade dos dados dos usuários.

Adotar boas práticas de segurança, como o uso de **tokens CSRF**, **HTTPS**, e **validação e sanitização de dados**, são essenciais para proteger seus aplicativos contra ataques maliciosos.

Aula: Webpack, Babel e Ferramentas de Build

Objetivo

- Compreender o que são **Webpack** e **Babel** e como usá-los em um projeto JavaScript.
 - Aprender a configurar um ambiente de desenvolvimento moderno com ferramentas de build.
 - Explorar como otimizar o fluxo de trabalho para produção e desenvolvimento.
-

1. O que é Webpack?

Webpack é um **bundler** de módulos para aplicações JavaScript. Ele permite que você **empacote** (bundle) todos os arquivos de um projeto (JS, CSS, imagens, etc.) em um ou mais pacotes, que podem ser carregados pelo navegador. Webpack facilita o processo de otimização

e organização do código, criando uma versão mais rápida e eficiente do seu aplicativo.

Principais características do Webpack:

- **Bundling:** Agrupa módulos (arquivos JavaScript, CSS, imagens, etc.) em um ou mais arquivos de saída.
- **Code Splitting:** Divide seu código em partes menores (chunks) para carregar apenas o necessário, melhorando o desempenho.
- **Loaders:** Permite que você processe arquivos não JavaScript, como Sass, imagens, ou JSX, e os converta em algo que o navegador pode entender.
- **Plugins:** Ferramentas poderosas para otimizar, minificar e manipular os arquivos finais.
- **Hot Module Replacement (HMR):** Permite atualizar módulos no navegador em tempo real durante o desenvolvimento, sem precisar recarregar a página.

Como instalar o Webpack:

```
npm install --save-dev webpack webpack-cli
```

A partir daí, você pode configurar o Webpack no arquivo `webpack.config.js`.



2. Configuração Básica do Webpack

A configuração básica do Webpack envolve a definição de três partes principais:

1. **Entry:** O ponto de entrada, geralmente o arquivo principal da aplicação (como `index.js`).
2. **Output:** O arquivo de saída que o Webpack gerará (geralmente `bundle.js`).
3. **Loaders:** Configurações que permitem processar arquivos de tipos diferentes de JS (como JSX ou Sass).
4. **Plugins:** Ferramentas para otimizar o build, como minificação e gerenciamento de ambientes.

Exemplo de um arquivo `webpack.config.js` básico:

```
const path = require('path');

module.exports = {
  entry: './src/index.js', // Arquivo principal
  output: {
    path: path.resolve(__dirname, 'dist'), // Pasta de saída
    filename: 'bundle.js' // Nome do arquivo final
  },
  module: {
    rules: [
      {
        test: /\.js$/, // Aplica ao arquivos .js
        exclude: /node_modules/,
        use: 'babel-loader' // Usar o Babel para transpilação
      },
      {
        test: /\.scss$/, // Aplica a arquivos .scss
        use: ['style-loader', 'css-loader', 'sass-loader']
      }
    ]
  },
  plugins: [
    // Plugins de otimização, como a minificação do código
  ]
}
```

```
]
};
```



3. O que é Babel?

Babel é um compilador JavaScript que permite escrever código moderno usando as últimas versões da linguagem (ES6+), e transpila esse código para uma versão compatível com navegadores mais antigos. Ele é particularmente útil para garantir que seu código seja executado em qualquer ambiente, independentemente da versão do JavaScript suportada.

Principais recursos do Babel:

- **Transpilação de ES6+:** Babel converte código ES6+ (como `let`, `const`, `arrow functions`, `async/await`) para ES5, garantindo compatibilidade com navegadores antigos.
- **Plugins e Presets:** Babel possui presets como `@babel/preset-env` para transpilar JavaScript, e plugins para funcionalidades específicas, como JSX (React).
- **Compatibilidade com JSX:** Para React, Babel converte JSX para código JavaScript que o navegador pode executar.

Como instalar o Babel:

```
npm install --save-dev @babel/core @babel/cli @babel/preset-env
babel-loader
```

Adicione a configuração do Babel em um arquivo `.babelrc`:

```
{
  "presets": ["@babel/preset-env"]
}
```

Isso configura o Babel para transpilar o código ES6+ para uma versão compatível com navegadores antigos.

4. Integração de Webpack com Babel

O Webpack pode ser configurado para usar o Babel como **loader** para processar arquivos JavaScript.

No arquivo `webpack.config.js`, inclua a configuração do `babel-loader`:

```
module: {  
  rules: [  
    {  
      test: /\.js$/,  
      exclude: /node_modules/,  
      use: 'babel-loader' // Transpila o código JavaScript com Babel  
    }  
  ]  
}
```

Isso garante que todos os arquivos `.js` sejam processados pelo Babel durante o processo de build.

5. Plugins e Otimização

O Webpack possui uma série de **plugins** que podem ser usados para otimizar seu build. Alguns dos mais populares incluem:

- **HtmlWebpackPlugin**: Gera automaticamente o arquivo HTML que referencia o bundle gerado.

- **MiniCssExtractPlugin:** Extrai CSS em arquivos separados, útil para produção.
- **TerserPlugin:** Minifica o código JavaScript para reduzir o tamanho final.

Exemplo de configuração com plugins:

```
npm install --save-dev html-webpack-plugin mini-css-extract-plugin terser-webpack-plugin
```

No arquivo `webpack.config.js`, adicione os plugins:

```
const HtmlWebpackPlugin = require('html-webpack-plugin');
const MiniCssExtractPlugin = require('mini-css-extract-plugin');
const TerserPlugin = require('terser-webpack-plugin');

module.exports = {
  // Configuração básica...
  plugins: [
    new HtmlWebpackPlugin({
      template: './src/index.html' // Gera o HTML a partir de um template
    }),
    new MiniCssExtractPlugin({
      filename: '[name].css'
    })
  ],
  optimization: {
    minimize: true, // Ativa a minificação
    minimizer: [new TerserPlugin()] // Minifica o JS com o Terser
  }
};
```

6. Outras Ferramentas de Build

Além do Webpack e Babel, há outras ferramentas úteis para otimizar o fluxo de desenvolvimento:

- **NPM Scripts:** Usar scripts no `package.json` para executar tarefas como build, lint, teste, etc.
- **Prettier:** Uma ferramenta de formatação de código para garantir que seu código esteja sempre consistente.
- **ESLint:** Ferramenta para identificar e corrigir problemas de estilo e erros no seu código JavaScript.

Exemplo de configuração de scripts no `package.json`:

```
{
  "scripts": {
    "start": "webpack serve --open", // Inicia o servidor de
    desenvolvimento
    "build": "webpack --mode production", // Cria uma versão otimizada
    para produção
    "lint": "eslint .", // Roda o linter no código
    "format": "prettier --write ." // Formata o código com Prettier
  }
}
```

7. Atividades Práticas

1. **Configuração do Webpack:** Crie um projeto com Webpack e configure a transpilação de código JavaScript moderno (ES6+) usando Babel.
2. **Plugins de Produção:** Adicione plugins como `HtmlWebpackPlugin` e `TerserPlugin` para otimizar o seu bundle de produção.

3. **Divisão de Código:** Implemente o **code splitting** no Webpack para dividir seu código em arquivos menores, melhorando o desempenho.
4. **CSS e Sass:** Adicione suporte para importar e processar arquivos CSS e Sass no seu projeto com Webpack.



Conclusão

O **Webpack** e o **Babel** são ferramentas essenciais no desenvolvimento moderno de JavaScript. O **Webpack** permite que você organize e otimize seu código, enquanto o **Babel** garante que seu código seja compatível com todos os navegadores. Com o uso de plugins e boas práticas de configuração, é possível criar aplicações rápidas e escaláveis.

Aula: Fase 4 - Aplicações com JavaScript

Objetivo

- Criar interfaces dinâmicas com **DOM** e **SPA**.
- Utilizar **frameworks** como **React.js** para criar interfaces de usuário reativas.
- Integrar **TypeScript** ao JavaScript para melhorar a manutenção e escalabilidade do código.
- Construir um back-end básico utilizando **Node.js**, **Express** e **MongoDB**.

1. DOM Dinâmico e SPA - Criar Interfaces Interativas

O que é um SPA (Single-Page Application)?

Um **SPA** é uma aplicação web que carrega uma única página HTML e atualiza dinamicamente o conteúdo conforme o usuário interage com a aplicação, sem a necessidade de recarregar a página inteira.

Como Criar uma SPA com JavaScript:

A principal característica de um SPA é a **navegação sem reload** da página. Para fazer isso, usamos **JavaScript** para alterar o conteúdo da página dinamicamente e gerenciar a navegação entre diferentes "páginas" ou estados da aplicação.

Passos principais para criar uma SPA:

1. **Manipulação de URL com `history.pushState` e `history.replaceState`:** Usamos essas funções para manipular o histórico de navegação e mudar a URL sem recarregar a página.
2. **Mudança de conteúdo com o DOM:** Usamos o JavaScript para alterar o conteúdo da página sem recarregar.

Exemplo de SPA com navegação dinâmica:

```
document.querySelector('#home').addEventListener('click', () => {  
  history.pushState({}, 'Home', '/home');  
  renderHomePage();  
});
```

```
document.querySelector('#about').addEventListener('click', () => {  
  history.pushState({}, 'About', '/about');  
  renderAboutPage();  
});
```

```
function renderHomePage() {  
  document.getElementById('content').innerHTML = "<h1>Home  
Page</h1>";  
}
```

```
function renderAboutPage() {  
  document.getElementById('content').innerHTML = "<h1>About  
Page</h1>";  
}
```

```
// Detectando mudanças de URL para recarregar o conteúdo
```

```
window.addEventListener('popstate', () => {  
  if (window.location.pathname === '/home') {  
    renderHomePage();  
  } else if (window.location.pathname === '/about') {  
    renderAboutPage();  
  }  
});
```



2. Frameworks e Bibliotecas (opcional)

React.js (ou Vue.js, Svelte)

Os **frameworks JavaScript** modernos como **React.js**, **Vue.js** e **Svelte** permitem criar interfaces dinâmicas e reativas com componentes reutilizáveis e manutenção mais fácil.

Vamos usar **React.js** como exemplo, que é um dos frameworks mais populares para criar SPAs.

Principais conceitos do React:

- **Componentes:** São blocos reutilizáveis de código que representam uma parte da interface.
- **Props:** São propriedades que você passa para os componentes, permitindo que eles recebam dados e funções.
- **Estado (State):** Um mecanismo interno do React que mantém dados específicos de cada componente.

Exemplo básico de componente React:

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

```
function Counter() {  
  const [count, setCount] = useState(0);
```

```
const increment = () => setCount(count + 1);

return (
  <div>
    <p>Contagem: {count}</p>
    <button onClick={increment}>Incrementar</button>
  </div>
);
}

export default Counter;
```

- **useState**: Um hook do React que permite gerenciar o estado dentro de um componente.
- **onClick**: Um manipulador de eventos para executar ações ao interagir com o componente.

Com o React, podemos criar SPAs completas, com roteamento interno e gerenciamento de estado, tornando a interface muito mais interativa.

3. TypeScript (JS com Tipagem)

TypeScript é um **superset** do JavaScript que adiciona **tipagem estática** ao código. Ele ajuda a encontrar erros de tipo durante o desenvolvimento e melhora a manutenção do código, especialmente em projetos grandes.

Principais características do TypeScript:

- **Tipagem Estática**: Você pode especificar tipos para variáveis, parâmetros de funções e valores de retorno.

- **Interfaces:** Permitem definir contratos para objetos e classes.
- **Enums:** Definem um conjunto de valores nomeados, facilitando a leitura do código.

Exemplo de tipagem estática com TypeScript:

```
// Tipagem de variável
```

```
let nome: string = 'João';
```

```
// Tipagem de função
```

```
function soma(a: number, b: number): number {  
    return a + b;  
}
```

```
console.log(soma(2, 3)); // Saída: 5
```

Exemplo de Interface e Enum:

```
// Interface para definir a estrutura de um objeto
```

```
interface Pessoa {  
    nome: string;  
    idade: number;  
}
```

```
const pessoa: Pessoa = {  
    nome: 'Maria',  
    idade: 30  
};
```

```
// Enum para valores fixos
```

```
enum Status {  
    Ativo,  
    Inativo  
}
```

```
const status: Status = Status.Ativo;
```

Como integrar TypeScript em seu projeto:

1. Instale o **TypeScript**:

```
npm install --save-dev typescript
```

2. Crie um arquivo `tsconfig.json` com a configuração do TypeScript:

```
{
  "compilerOptions": {
    "target": "es6",
    "module": "commonjs",
    "strict": true
  }
}
```

3. Compile seus arquivos `.ts` para `.js` com o comando:

```
npx tsc
```

4. Back-end com JavaScript - Node.js e Express

Node.js é uma plataforma para executar JavaScript no lado do servidor, permitindo construir back-ends completos usando apenas JavaScript.

O que é o Express.js?

Express.js é um framework minimalista para construir aplicações web e APIs REST de forma rápida e fácil com o Node.js.

Criando um servidor simples com Node.js e Express:

1. Instale as dependências:

```
npm install express
```

2. Crie um servidor básico com Express:

```
// server.js
const express = require('express');
const app = express();

app.get('/', (req, res) => {
  res.send('Olá, mundo!');
});

app.listen(3000, () => {
  console.log('Servidor rodando na porta 3000');
});
```

Criando uma API REST simples:

Você pode usar o Express para criar APIs RESTful, que permitem que seu front-end se comunique com o back-end.

```
// API simples com Express
app.get('/api/users', (req, res) => {
  const users = [{ nome: 'João' }, { nome: 'Maria' }];
  res.json(users);
});
```



5. MongoDB com Mongoose

MongoDB é um banco de dados NoSQL muito usado em aplicações JavaScript modernas. O **Mongoose** é uma biblioteca que facilita a interação com o MongoDB usando JavaScript.

Como usar MongoDB com Mongoose:

1. Instale o **Mongoose**:

```
npm install mongoose
```

2. Conecte ao MongoDB e defina um modelo com Mongoose:

```
const mongoose = require('mongoose');

// Conectando ao MongoDB
mongoose.connect('mongodb://localhost:27017/meuapp', {
  useNewUrlParser: true, useUnifiedTopology: true });

// Definindo um modelo de dados
const Usuario = mongoose.model('Usuario', {
  nome: String,
  idade: Number
});

// Criando um novo usuário
const usuario = new Usuario({ nome: 'João', idade: 30 });
usuario.save().then(() => console.log('Usuário salvo'));
```

3. Criando uma API para gerenciar dados do MongoDB:

```
app.get('/api/usuarios', async (req, res) => {
  const usuarios = await Usuario.find();
  res.json(usuarios);
});
```



6. Atividades Práticas

1. **SPA com JavaScript:** Crie uma SPA simples com navegação dinâmica e manipulação de URL utilizando **history.pushState**.
2. **React Component:** Crie um componente React básico que exibe uma lista de itens e permite adicionar novos itens ao clicar em um botão.
3. **TypeScript:** Converta um projeto JavaScript simples para TypeScript, implementando tipos e interfaces.
4. **Back-end com Express:** Crie uma API REST que manipula dados de usuários e armazena em um banco de dados MongoDB utilizando Mongoose.



Conclusão

Nesta fase, você aprendeu a integrar **JavaScript** com tecnologias modernas para criar **aplicações reais**. Criamos SPAs dinâmicas, trabalhamos com **React** e **TypeScript**, e configuramos um back-end básico com **Node.js**, **Express** e **MongoDB**. Essas são as bases para construir aplicações complexas e escaláveis no futuro!

Aula: DOM Dinâmico e SPA (Single Page Application)

Objetivo

- Criar interfaces dinâmicas e interativas com manipulação do **DOM**.
- Desenvolver uma **Single Page Application (SPA)**, onde as interações do usuário não causam recarregamento da página, melhorando a experiência do usuário.

O que é DOM Dinâmico?

O **DOM Dinâmico** refere-se à manipulação da árvore DOM de uma página web com JavaScript, permitindo adicionar, remover ou modificar os elementos da página em tempo real. Com o DOM Dinâmico, é possível alterar a estrutura do conteúdo da página sem recarregar a página inteira.

O que é uma SPA (Single Page Application)?

Uma **SPA** é uma aplicação web que carrega uma única página HTML e, após o carregamento inicial, não recarrega mais a página ao navegar para diferentes partes da aplicação. Em vez disso, o conteúdo da página é atualizado dinamicamente usando JavaScript, proporcionando uma experiência mais fluida e interativa para o usuário.

Como Criar uma SPA

A criação de uma SPA envolve manipulação do **DOM** para alterar o conteúdo da página sem a necessidade de recarregar a página inteira. O JavaScript é usado para fazer essas mudanças de conteúdo e, ao mesmo tempo, manipular a URL para refletir a navegação sem perder o estado da página.

Passos principais:

1. **Manipulação da URL** com o `history.pushState()` e `history.replaceState()`, que permitem alterar a URL da página sem fazer o recarregamento.
2. **Renderização de conteúdo dinâmico** com JavaScript, manipulando o DOM para substituir o conteúdo da página.
3. **Monitoramento de alterações na URL** com `window.addEventListener('popstate', callback)`, para permitir que a navegação seja rastreada corretamente.

Estrutura Básica de uma SPA

Vamos construir uma SPA simples com duas "páginas" (Home e About), e navegação entre elas sem recarregar a página.

Exemplo de código para uma SPA básica:

1. Estrutura HTML (index.html):

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

```

<title>SPA - Navegação Sem Recarregar</title>
<style>
  /* Estilos básicos */
  #home, #about {
    display: none;
  }
</style>
</head>
<body>
  <nav>
    <ul>
      <li><a href="#" id="home-link">Home</a></li>
      <li><a href="#" id="about-link">About</a></li>
    </ul>
  </nav>

  <div id="content">
    <!-- Conteúdo da página será renderizado aqui -->
  </div>

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

```

2. JavaScript para SPA (app.js):

// Funções de renderização para as páginas

```

function renderHomePage() {
  document.getElementById('content').innerHTML = "<h1>Home
Page</h1><p>Bem-vindo à nossa SPA!</p>";
  document.getElementById('home').style.display = 'block';
  document.getElementById('about').style.display = 'none';
}

function renderAboutPage() {
  document.getElementById('content').innerHTML = "<h1>About
Page</h1><p>Esta é a página sobre nós.</p>";
  document.getElementById('home').style.display = 'none';
}

```

```

    document.getElementById('about').style.display = 'block';
}

// Configurando navegação com o histórico
document.getElementById('home-link').addEventListener('click', (e) => {
    e.preventDefault();
    history.pushState({}, 'Home', '/home');
    renderHomePage();
});

document.getElementById('about-link').addEventListener('click', (e) => {
    e.preventDefault();
    history.pushState({}, 'About', '/about');
    renderAboutPage();
});

// Monitorando alterações na URL
window.addEventListener('popstate', (e) => {
    if (window.location.pathname === '/home') {
        renderHomePage();
    } else if (window.location.pathname === '/about') {
        renderAboutPage();
    }
});

// Carregando a página inicial
if (window.location.pathname === '/home') {
    renderHomePage();
} else {
    renderAboutPage();
}

```

Explicação do código:

1. Manipulação de Navegação:

- Quando um link é clicado, o evento `click` é interceptado e o `history.pushState()` é chamado para alterar a URL sem recarregar a página.
- A função `renderHomePage()` ou `renderAboutPage()` é chamada para atualizar o conteúdo da página.

2. Monitoramento de Alterações na URL:

- O `popstate` é disparado sempre que a URL é alterada (como quando o usuário navega para trás ou para frente no histórico). Usamos esse evento para verificar a URL e renderizar o conteúdo adequado.

3. Controle de Exibição:

- O conteúdo da página é alterado no `div#content` com o conteúdo de cada página (Home ou About), e os outros conteúdos são ocultados, permitindo apenas um conteúdo por vez.



Benefícios de uma SPA

- **Experiência de usuário mais fluida:** Sem recarregar a página, o conteúdo é atualizado instantaneamente, o que torna a navegação muito mais rápida e suave.
- **Maior controle sobre o conteúdo da página:** Como você manipula o DOM diretamente, você tem total controle sobre o conteúdo exibido e a forma como ele é apresentado.
- **Menor carga no servidor:** Como apenas os dados são enviados entre o cliente e o servidor, em vez de re-renderizar a página

inteira, o tráfego de rede pode ser mais eficiente.

Desafios Comuns

- **Gerenciamento de estado:** À medida que a aplicação cresce, o gerenciamento de estado pode se tornar complexo. Para resolver isso, você pode usar bibliotecas como **Redux** (no React) ou **Vuex** (no Vue.js).
 - **SEO:** SPAs podem ser difíceis de otimizar para mecanismos de busca, já que o conteúdo é carregado dinamicamente. Uma solução é usar o **SSR (Server-Side Rendering)** ou **Prerendering** para gerar HTML estático antes da página ser carregada.
-

Atividades Práticas

1. **SPA com JavaScript puro:** Crie uma aplicação simples de várias páginas (Home, About, Contact) usando a manipulação do DOM e navegação sem recarregar a página.
 2. **Desafio com componentes:** Crie um sistema de navegação com abas (tab) usando o DOM, onde o conteúdo das abas é carregado dinamicamente sem recarregar a página.
 3. **Implementação de animações:** Use CSS e JavaScript para adicionar animações de transição entre as páginas ao navegar entre os diferentes conteúdos.
-

Conclusão

Nesta aula, aprendemos a criar uma **SPA (Single Page Application)** básica utilizando JavaScript e manipulação do DOM. A SPA permite navegar entre diferentes seções do conteúdo sem recarregar a página, proporcionando uma experiência mais fluida e interativa. Esta técnica é fundamental para o desenvolvimento de aplicações modernas e é amplamente usada em frameworks como **React**, **Vue.js** e **Angular**.

Aula: Frameworks e Bibliotecas

Nesta aula, vamos explorar os principais **frameworks e bibliotecas** JavaScript modernos que permitem criar aplicações dinâmicas e escaláveis com mais facilidade e eficiência. Vamos nos concentrar em **React.js**, um dos frameworks mais populares, mas também abordaremos outras opções como **Vue.js** e **Svelte**.

Objetivo

- Compreender o conceito de **frameworks e bibliotecas** JavaScript.

- Aprender os principais conceitos do **React.js**.
 - Explorar as principais funcionalidades e como elas ajudam a criar **aplicações web** dinâmicas e interativas.
-

O que são Frameworks e Bibliotecas?

- **Biblioteca:** Uma biblioteca é um conjunto de funcionalidades que você pode usar para ajudar no desenvolvimento do seu aplicativo. Ela fornece funções e métodos prontos que você pode utilizar quando precisar, sem a necessidade de reescrever tudo do zero. Você chama as funções da biblioteca conforme necessário.
- **Framework:** Um framework é mais completo. Ele define a estrutura do seu aplicativo e estabelece como as diferentes partes devem interagir. Com um framework, você geralmente segue a sua "arquitetura" e "padrões" predefinidos. O framework geralmente "chama" o seu código em vez de você chamar o código do framework, ou seja, ele impõe mais regras.

React.js é tecnicamente considerado uma biblioteca, mas muitas vezes é chamado de **framework** devido à sua robustez e à forma como ele facilita a criação de interfaces de usuário dinâmicas.

React.js

O que é React?

React.js (ou simplesmente **React**) é uma biblioteca JavaScript para a construção de interfaces de usuário, desenvolvida pelo Facebook. O React permite que você crie componentes reutilizáveis que são combinados para formar a interface completa. Ele se destaca pela

renderização reativa, ou seja, quando o estado de um componente muda, o React automaticamente re-renderiza esse componente (e seus filhos) de forma eficiente.

Por que usar React?

- **Componentes reutilizáveis:** No React, a interface é dividida em componentes independentes que podem ser reutilizados.
 - **Virtual DOM:** O React utiliza uma versão "virtual" da árvore DOM, o que torna as atualizações da interface mais rápidas e eficientes.
 - **Unidirecional:** A comunicação de dados entre componentes é feita de forma unidirecional, o que facilita o controle do fluxo de dados.
-

Principais conceitos do React.js

1. Componentes

No React, tudo é um **componente**. Os componentes podem ser **funcionais** ou **baseados em classe**. No entanto, com a introdução dos **Hooks**, a maioria dos desenvolvedores prefere usar componentes funcionais.

Exemplo de Componente Funcional:

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

```
function Counter() {  
  const [count, setCount] = useState(0); // useState é um Hook  
  
  return (  
    <div>  
      <p>Contagem: {count}</p>  
      <button onClick={() => setCount(count + 1)}>Incrementar</button>  
    </div>  
  );  
}
```

```
);  
}
```

```
export default Counter;
```

- **useState**: Hook que permite adicionar estado local aos componentes funcionais.
- **onClick**: Evento de click para disparar a função que altera o estado.

2. Props

As **props** (propriedades) são usadas para passar dados de um componente para outro. Elas são imutáveis, ou seja, o componente que recebe as props não pode alterá-las diretamente.

```
function Saudacao(props) {  
  return <h1>Olá, {props.nome}!</h1>;  
}
```

```
function App() {  
  return <Saudacao nome="Maria" />;  
}
```

3. Estado (State)

O **estado** é usado para armazenar dados que podem mudar durante a execução de um componente. Quando o estado de um componente muda, ele é re-renderizado automaticamente.

```
const [contador, setContador] = useState(0);
```

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

4. JSX (JavaScript XML)

O **JSX** é uma sintaxe de extensão para o JavaScript que permite escrever HTML dentro do código JavaScript. O JSX torna o código React mais legível e fácil de entender.

```
const element = <h1>Olá, Mundo!</h1>;
```

- O JSX é convertido para **JavaScript puro** pelo React.

5. Hooks

Hooks são funções que permitem usar o **estado** e outras funcionalidades do React em componentes funcionais.

- **useState**: Adiciona estado ao componente funcional.
- **useEffect**: Permite executar efeitos colaterais (como chamadas a APIs) em componentes funcionais.

```
useEffect(() => {  
  // Efeito a ser executado após o componente ser montado  
}, []); // O array vazio [] indica que o efeito deve ser executado apenas  
uma vez, ao montar o componente
```

6. Roteamento com React Router

Para criar uma SPA com React, você precisa de um sistema de navegação entre páginas. **React Router** é a biblioteca oficial para isso.

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

Exemplo de uso do **React Router** para navegação:

```
import { BrowserRouter as Router, Route, Switch, Link } from  
'react-router-dom';
```

```
function App() {  
  return (  
    <Router>  
      <nav>  
        <ul>  
          <li><Link to="/">Home</Link></li>  
          <li><Link to="/sobre">Sobre</Link></li>  
        </ul>  
      </nav>  
  
      <Switch>  
        <Route path="/" exact>  
          <Home />  
        </Route>  
        <Route path="/sobre">  
          <Sobre />  
        </Route>  
      </Switch>  
    </Router>  
  );  
}
```

```
function Home() {  
  return <h1>Bem-vindo à Home!</h1>;  
}
```

```
function Sobre() {  
  return <h1>Sobre a nossa aplicação</h1>;  
}
```



Outras Bibliotecas e Frameworks

1. Vue.js

Vue.js é um framework progressivo para construir interfaces de usuário. Ele é mais leve que o React, e seus conceitos são mais simples para quem está começando.

- **Componentes** são como em React.
- **Directives** como `v-if`, `v-for` e `v-bind` permitem manipular o DOM de maneira simples.
- É fácil integrar o Vue com outras bibliotecas e projetos existentes.

2. Svelte

Svelte é um framework que se diferencia dos outros por **não usar o Virtual DOM**. Em vez disso, ele compila seus componentes em código JavaScript eficiente durante a construção, o que pode resultar em um desempenho melhor.

- Não há necessidade de uma biblioteca extra no navegador.
- Mais simples de configurar e usar para quem está começando.

3. Angular

Angular é um framework completo e robusto desenvolvido pelo Google. Ele oferece uma solução de front-end para criar aplicações de grande escala com suporte a **roteamento**, **serviços**, **injeção de dependências** e muito mais.

Configuração de Projeto React com Create React App

1. **Criando um projeto React:** O jeito mais fácil de começar com React é usar o **Create React App**, uma ferramenta oficial para configurar rapidamente um projeto React com todas as

dependências necessárias.

```
npx create-react-app meu-app  
cd meu-app  
npm start
```

2. **Estrutura do projeto:** O **Create React App** cria automaticamente a estrutura básica de um projeto com React.

- **src/:** Pasta onde você coloca os arquivos de código JavaScript.
- **public/:** Contém o arquivo **index.html** e outros arquivos estáticos.
- **App.js:** Componente principal que será renderizado.



Atividades Práticas

1. **Crie um contador interativo** usando React com um botão para incrementar e exibir o número de cliques.
 2. **Desenvolva uma aplicação de to-do list** em React, onde você pode adicionar e remover tarefas.
 3. **Integre uma API externa** (por exemplo, uma API de filmes) usando **useEffect** para buscar e exibir dados dinamicamente.
 4. **Explore Vue.js ou Svelte** criando uma pequena SPA para comparar com o React.
-



Conclusão

Nesta aula, exploramos os principais conceitos de **React.js**, como componentes, props, estado e hooks. Vimos como configurar um projeto e criar uma SPA simples. Também discutimos outras opções como **Vue.js** e **Svelte**, que são alternativas populares ao React.

Agora, você está pronto para usar React para criar **aplicações web interativas** e aprender ainda mais com outras bibliotecas e frameworks para expandir seus conhecimentos.



Aula: Introdução ao TypeScript

O **TypeScript** é uma linguagem de programação que se baseia no **JavaScript**, mas adiciona um sistema de tipagem estática. Embora o JavaScript seja dinâmico e permita uma flexibilidade maior no tipo de dados, o TypeScript ajuda a melhorar a **manutenibilidade**, **segurança** e **clareza** do código ao forçar a declaração explícita de tipos.



Objetivo

- **Entender os benefícios do TypeScript** em relação ao JavaScript.
- **Aprender como adicionar tipagem estática** ao JavaScript usando TypeScript.
- **Compreender os conceitos principais do TypeScript**, como tipos, interfaces, classes e enums.

O que é TypeScript?

TypeScript é uma linguagem de código aberto desenvolvida pela Microsoft. Ele é **superconjunto** de JavaScript, o que significa que qualquer código JavaScript é também um código válido em TypeScript. O que o TypeScript adiciona é um sistema de tipagem estática que ajuda a identificar erros durante o desenvolvimento, antes mesmo da execução do código.

TypeScript compila para **JavaScript** padrão, então você pode usá-lo em qualquer ambiente que suporte JavaScript, como navegadores e servidores Node.js.

Principais Características do TypeScript

1. Tipagem Estática

A principal característica do TypeScript é a **tipagem estática**, o que significa que você define o tipo das variáveis, parâmetros e valores de retorno de funções antes da execução. Isso ajuda a prevenir muitos erros comuns que só seriam detectados em tempo de execução com JavaScript.

Exemplo de Tipagem Estática:

```
let nome: string = "João"; // Nome é uma string
let idade: number = 30;    // Idade é um número
```

2. Interfaces

As **interfaces** no TypeScript são uma maneira de definir a estrutura de um objeto. Elas garantem que o objeto tenha as propriedades e métodos especificados, sem que você precise usar uma classe.

Exemplo de Interface:

```
interface Pessoa {  
  nome: string;  
  idade: number;  
}
```

```
const pessoa: Pessoa = {  
  nome: "Maria",  
  idade: 25  
};
```

Interfaces são muito úteis quando você quer garantir que um objeto siga uma estrutura específica sem definir toda uma classe.

3. Classes e Herança

Embora o JavaScript tenha suporte a classes desde o ECMAScript 6, o TypeScript permite que você defina tipos mais precisos em suas classes, garantindo que as instâncias sigam a estrutura esperada.

Exemplo de Classe com Tipagem:

```
class Animal {  
  nome: string;  
  idade: number;  
  
  constructor(nome: string, idade: number) {  
    this.nome = nome;  
    this.idade = idade;  
  }  
  
  falar(): void {  
    console.log(` ${this.nome} está falando!` );  
  }  
}  
  
const cachorro = new Animal("Rex", 5);  
cachorro.falar();
```

Além disso, o TypeScript permite que você defina **herança**, **modificadores de acesso** (como `private`, `public` e `protected`), e outros conceitos avançados de programação orientada a objetos.

4. Enums

Os **enums** permitem criar um conjunto de valores nomeados que podem ser usados para representar uma variável com valores fixos.

Exemplo de Enum:

```
enum Cor {  
  Vermelho = "vermelho",  
  Azul = "azul",  
  Verde = "verde"  
}
```

```
let corFavorita: Cor = Cor.Azul;  
console.log(corFavorita); // Azul
```

Os enums ajudam a evitar "números mágicos" ou **strings** espalhadas pelo código, tornando-o mais legível e fácil de entender.

Instalando o TypeScript

1. Instalar TypeScript globalmente

Primeiro, você precisa instalar o TypeScript em sua máquina. Isso pode ser feito através do **npm** (Node Package Manager).

```
npm install -g typescript
```

2. Criando um projeto TypeScript

Depois de instalar o TypeScript, crie uma pasta para seu projeto e inicialize um projeto Node.js:

```
mkdir meu-projeto-ts  
cd meu-projeto-ts  
npm init -y
```

Em seguida, instale o TypeScript como dependência de desenvolvimento:

```
npm install --save-dev typescript
```

Agora, crie um arquivo `tsconfig.json`, que é onde você pode definir as configurações do compilador TypeScript:

```
npx tsc --init
```

3. Compilando TypeScript

Para compilar seu código TypeScript para JavaScript, use o seguinte comando:

```
npx tsc
```

Isso criará arquivos `.js` a partir dos arquivos `.ts` em seu projeto.



Converter JavaScript para TypeScript

Uma das maiores vantagens do TypeScript é que você pode começar a usar **gradualmente** em seu código JavaScript. Você pode escrever código JavaScript normal e mudar os arquivos `.js` para `.ts` à medida que for se acostumando.

Exemplo de Conversão de JavaScript para TypeScript:

Arquivo JavaScript (`app.js`):

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

Arquivo **TypeScript** (`app.ts`):

```
let idade: number = 30;  
console.log(idade);
```

Benefícios do TypeScript

1. **Detecção de erros em tempo de desenvolvimento:** O TypeScript oferece detecção de erros antes mesmo da execução, o que ajuda a evitar muitos erros comuns de codificação.
 2. **Suporte a grandes projetos:** O TypeScript ajuda a manter a base de código organizada e segura, sendo especialmente útil em projetos grandes.
 3. **Melhor IntelliSense:** O TypeScript melhora as sugestões do editor de código (como o VS Code), tornando a programação mais rápida e eficiente.
 4. **Integração com bibliotecas:** O TypeScript tem excelente suporte a bibliotecas de terceiros, como React, Angular e Node.js, além de fornecer **tipos** para muitas delas.
-

Exemplo Prático

Vamos criar uma pequena função de soma e tipar seus parâmetros e retorno.

Função em TypeScript:

```
function somar(a: number, b: number): number {  
  return a + b;  
}
```

```
const resultado = somar(10, 20);  
console.log(resultado); // 30
```

Aqui, a função **somar** recebe dois parâmetros do tipo **number** e retorna um **number**. Se tentarmos passar outro tipo de dado, o TypeScript gerará um erro de compilação.

Dicas para começar com TypeScript

- **Inicie com um pequeno projeto:** Não precisa migrar tudo de uma vez. Comece com arquivos pequenos ou um módulo e vá aos poucos.
 - **Use o `tsc --watch`:** Isso faz com que o TypeScript compile automaticamente os arquivos à medida que você os altera.
 - **Explore o TypeScript com React:** Se você estiver familiarizado com React, tente usar TypeScript em um projeto React. Isso ajuda a melhorar a experiência de desenvolvimento, especialmente em projetos grandes.
-



Conclusão

Agora você tem uma visão geral do **TypeScript** e como ele pode melhorar o desenvolvimento em JavaScript, proporcionando **tipagem estática**, **segurança** e **clareza** no código. Ao utilizar TypeScript, você pode evitar muitos erros comuns de codificação e melhorar a

manutenibilidade do seu código, o que é especialmente útil em projetos grandes e complexos.

Aprofunde seus conhecimentos explorando mais características do TypeScript, como **Generics**, **Decorators**, **Namespaces** e outros conceitos avançados.

Aula: Back-end com JavaScript

No desenvolvimento web, **JavaScript** não é apenas usado no front-end (navegadores), mas também pode ser usado no **back-end** (servidores). Com a introdução do **Node.js**, o JavaScript se tornou uma das principais linguagens para desenvolvimento de servidores, APIs e serviços em tempo real.

Nesta aula, vamos aprender sobre o uso de JavaScript no back-end, explorando as ferramentas e bibliotecas mais comuns para criar servidores web, APIs REST e como interagir com bancos de dados.

Objetivo

- **Entender o que é o Node.js** e como ele permite que o JavaScript seja usado no back-end.
- **Criar um servidor web básico** utilizando Node.js e o framework **Express.js**.
- **Construir uma API RESTful** para manipulação de dados.

- **Conectar-se a um banco de dados MongoDB** e realizar operações CRUD (Create, Read, Update, Delete).
-

O que é Node.js?

Node.js é um ambiente de execução JavaScript do lado do servidor que permite rodar código JavaScript fora do navegador. Ele é baseado no **V8 JavaScript Engine** do Google Chrome e usa um modelo de **I/O não-bloqueante e orientado a eventos**, tornando-o ideal para aplicativos escaláveis e de alto desempenho.

Vantagens do Node.js:

- **Desempenho rápido** devido ao motor V8 do Google.
 - **Escalabilidade** para lidar com muitos usuários simultaneamente.
 - **Unificação** do front-end e back-end com a mesma linguagem (JavaScript).
 - **Grande ecossistema** de pacotes via **npm (Node Package Manager)**.
-

Configuração do Projeto com Node.js e Express.js

1. Instalando Node.js

Primeiro, você precisa instalar o **Node.js**. Você pode baixar a versão mais recente do Node.js [aqui](#).

Verifique a instalação usando o comando:

```
node -v
```

```
npm -v
```

- **node**: Para verificar a versão do Node.js.
- **npm**: Para verificar a versão do **Node Package Manager**.

2. Inicializando o projeto

Crie uma pasta para o seu projeto e inicie um projeto **Node.js** com o seguinte comando:

```
mkdir meu-backend
```

```
cd meu-backend
```

```
npm init -y
```

Isso cria um arquivo **package.json** para o seu projeto, onde as dependências serão listadas.

3. Instalando Express.js

O **Express.js** é um framework minimalista para o Node.js que facilita a criação de servidores web e APIs. Instale o Express:

```
npm install express
```

4. Criando um Servidor Básico com Express

Crie um arquivo chamado **index.js** e adicione o seguinte código para configurar um servidor básico:

```
const express = require('express');
```

```
const app = express();

const port = 3000;


// Rota básica

app.get('/', (req, res) => {

  res.send('Olá, Mundo!');

});


// Inicia o servidor

app.listen(port, () => {

  console.log(`Servidor rodando em http://localhost:${port}`);

});
```

Aqui, estamos criando um servidor que escuta na porta **3000**. Quando acessamos a rota raiz (**/**), ele responde com "Olá, Mundo!".

5. Rodando o servidor

Execute o servidor com o seguinte comando:

```
node index.js
```

Agora, ao acessar **http://localhost:3000/** no seu navegador, você verá a resposta "Olá, Mundo!".



Construindo uma API RESTful com Express

Agora, vamos criar uma API simples para manipulação de recursos. Para esta aula, vamos usar um **array em memória** para armazenar os dados. Em uma aplicação real, esses dados estariam em um banco de dados.

1. Estrutura Básica de uma API

Vamos construir um CRUD (Create, Read, Update, Delete) para gerenciar "tarefas".

Definindo a estrutura do servidor:

```
const express = require('express');
```

```
const app = express();
```

```
const port = 3000;
```

```
app.use(express.json()); // Middleware para analisar o corpo das  
requisições como JSON
```

```
let tarefas = [
```

```
  { id: 1, descricao: "Estudar JavaScript" },
```

```
  { id: 2, descricao: "Fazer exercícios de TypeScript" }
```

```
];
```

```
// Rota para obter todas as tarefas
```

```
app.get('/tarefas', (req, res) => {
```

```
  res.json(tarefas);
```

```
});
```

```
// Rota para criar uma nova tarefa
```

```
app.post('/tarefas', (req, res) => {  
  const { descricao } = req.body;  
  const novaTarefa = { id: tarefas.length + 1, descricao };  
  tarefas.push(novaTarefa);  
  res.status(201).json(novaTarefa);  
});
```

```
// Rota para atualizar uma tarefa
```

```
app.put('/tarefas/:id', (req, res) => {  
  const { id } = req.params;  
  const { descricao } = req.body;  
  const tarefa = tarefas.find(t => t.id === parseInt(id));  
  
  if (!tarefa) {  
    return res.status(404).json({ message: "Tarefa não encontrada" });  
  }
```

```
  tarefa.descricao = descricao;  
  res.json(tarefa);  
});
```

```
// Rota para deletar uma tarefa
```

```
app.delete('/tarefas/:id', (req, res) => {  
  const { id } = req.params;  
  tarefas = tarefas.filter(t => t.id !== parseInt(id));  
  res.status(204).end();  
});  
  
// Inicia o servidor  
app.listen(port, () => {  
  console.log(`API rodando em http://localhost:${port}`);  
});
```

2. Testando a API

- **GET /tarefas:** Para obter todas as tarefas.
- **POST /tarefas:** Para criar uma nova tarefa. Envie um corpo com a chave **descricao** para criar uma nova tarefa.
- **PUT /tarefas/:id:** Para atualizar a descrição de uma tarefa existente.
- **DELETE /tarefas/:id:** Para deletar uma tarefa.

Você pode testar essas rotas com ferramentas como **Postman** ou **Insomnia**.



Conectando com o MongoDB

Agora, vamos explorar como conectar a aplicação ao **MongoDB**, um banco de dados NoSQL popular.

1. Instalando o MongoDB e Mongoose

Primeiro, instale o **Mongoose**, uma biblioteca que facilita a interação com o MongoDB em Node.js.

```
npm install mongoose
```

2. Conectando ao MongoDB

Crie um arquivo de configuração para a conexão com o MongoDB:

```
const mongoose = require('mongoose');

mongoose.connect('mongodb://localhost:27017/meu-app', {
  useNewUrlParser: true,
  useUnifiedTopology: true
})

.then(() => console.log("Conectado ao MongoDB"))

.catch((err) => console.log("Erro na conexão com o MongoDB: ", err));
```

Aqui, estamos conectando ao banco de dados **meu-app** no MongoDB local.

3. Definindo um Modelo de Tarefa

No Mongoose, você define **modelos** para interagir com os dados. Vamos criar um modelo de tarefa.

```
const mongoose = require('mongoose');
```

```
const tarefaSchema = new mongoose.Schema({
  descricao: { type: String, required: true },
});

const Tarefa = mongoose.model('Tarefa', tarefaSchema);

module.exports = Tarefa;
```

4. Integrando a API com MongoDB

Agora, vamos substituir o array em memória pelo banco de dados MongoDB para armazenar as tarefas.

Atualizando o código do servidor:

```
const express = require('express');
const mongoose = require('mongoose');
const Tarefa = require('./tarefaModel');
const app = express();
const port = 3000;

app.use(express.json());

// Rota para obter todas as tarefas
app.get('/tarefas', async (req, res) => {
```



```
try {  
  const tarefas = await Tarefa.find();  
  res.json(tarefas);  
} catch (err) {  
  res.status(500).json({ message: "Erro ao buscar tarefas" });  
}  
});
```

// Rota para criar uma nova tarefa

```
app.post('/tarefas', async (req, res) => {  
  const { descricao } = req.body;  
  try {  
    const novaTarefa = new Tarefa({ descricao });  
    await novaTarefa.save();  
    res.status(201).json(novaTarefa);  
  } catch (err) {  
    res.status(400).json({ message: "Erro ao criar tarefa" });  
  }  
});
```

// Rota para atualizar uma tarefa

```
app.put('/tarefas/:id', async (req, res) => {  
  const { id } = req.params;
```

```
const { descricao } = req.body;

try {

  const tarefa = await Tarefa.findByIdAndUpdate(id, { descricao }, { new:
true });

  if (!tarefa) return res.status(404).json({ message: "Tarefa não
encontrada" });

  res.json(tarefa);

} catch (err) {

  res.status(400).json({ message: "Erro ao atualizar tarefa" });

}

});
```

// Rota para deletar uma tarefa

```
app.delete('/tarefas/:id', async (req, res) => {

  const { id } = req.params;

  try {

    await Tarefa.findByIdAndDelete(id);

    res.status(204).end();

  } catch (err) {

    res.status(400).json({ message: "Erro ao deletar tarefa" });

  }

});
```

// Conectar ao MongoDB e iniciar o servidor

```
mongoose.connect('mongodb://localhost:27017/meu-app', {  
  useNewUrlParser: true,  
  useUnifiedTopology: true  
})  
  
.then(() => {  
  app.listen(port, () => {  
    console.log(`API rodando em http://localhost:${port}`);  
  });  
})  
  
.catch((err) => console.log("Erro na conexão com o MongoDB: ", err));
```



Conclusão

Agora, você sabe como usar **JavaScript no back-end** com **Node.js** e **Express.js** para criar servidores e APIs. Além disso, você aprendeu a **conectar sua aplicação a um banco de dados MongoDB** usando Mongoose e como realizar operações CRUD. Com essas ferramentas, você está pronto para construir APIs escaláveis e dinâmicas no back-end com JavaScript.

Aula: Fase 5 - Prática e Projetos

Agora que você já tem uma boa compreensão dos conceitos fundamentais, intermediários e avançados de JavaScript, chegou o momento de consolidar esses conhecimentos através de **projetos práticos**. A prática é a melhor maneira de reforçar o que você aprendeu e desenvolver a confiança necessária para criar suas próprias aplicações reais.

Neste módulo, vamos sugerir uma série de **projetos práticos** que irão ajudá-lo a aplicar as habilidades que você adquiriu em **JavaScript**, **Node.js**, **TypeScript**, e outras tecnologias. Esses projetos variam em complexidade e abrangem diversos aspectos do desenvolvimento de software, desde **front-end** até **back-end**.

Objetivo

- Consolidar os conhecimentos adquiridos por meio da criação de **aplicações reais**.
 - Desenvolver **habilidades práticas** para construir aplicações completas e interativas.
 - Aplicar conceitos como **manipulação de DOM**, **consumo de APIs**, **armazenamento local**, **back-end com Node.js**, **interatividade** e **integridade dos dados**.
-

Projetos Sugeridos

Aqui estão alguns projetos sugeridos para você aplicar os conceitos aprendidos e ganhar experiência prática com JavaScript.

1. To-Do List (Lista de Tarefas)

Objetivo: Criar uma aplicação de lista de tarefas onde o usuário pode adicionar, editar e excluir tarefas.

Tecnologias envolvidas:

- **HTML** e **CSS** para a interface do usuário.
- **JavaScript** para lógica de adicionar e excluir tarefas.
- **Armazenamento Local (localStorage)** para persistência das tarefas.

Desafio adicional:

- Adicionar funcionalidades de **marcação de tarefa como concluída**.
- Implementar **filtro de tarefas** (por exemplo, todas, ativas, concluídas).

Requisitos principais:

- Exibir a lista de tarefas na página.
- Permitir adicionar, editar e excluir tarefas.
- Salvar as tarefas no **localStorage** para que persistam após recarregar a página.

2. Conversor de Moedas

Objetivo: Criar uma aplicação que converta valores entre diferentes moedas utilizando uma API pública de câmbio.

Tecnologias envolvidas:

- **HTML** e **CSS** para interface.
- **JavaScript** para lógica do conversor.
- **API de câmbio** para obter a taxa de câmbio e converter os valores.

Desafio adicional:

- Criar uma interface dinâmica onde o usuário pode selecionar a moeda de origem e destino.
- Implementar um **botão de atualização** para buscar as taxas de câmbio mais recentes.

Requisitos principais:

- Exibir os campos para inserir o valor e selecionar as moedas.
 - Utilizar uma API pública (como exchangerate-api.com) para obter as taxas de câmbio.
 - Mostrar o valor convertido de forma precisa e dinâmica.
-

3. Jogo da Velha (Tic-Tac-Toe)

Objetivo: Criar o clássico jogo da velha, onde dois jogadores podem jogar um contra o outro, e o jogo verifica automaticamente quando alguém vence ou empata.

Tecnologias envolvidas:

- **HTML** e **CSS** para a interface gráfica.

- **JavaScript** para a lógica do jogo.

Desafio adicional:

- Adicionar uma funcionalidade para jogar contra a máquina utilizando um algoritmo simples de inteligência artificial (como o algoritmo minimax).
- Implementar um **relógio de contagem de tempo** para cada jogador.

Requisitos principais:

- Exibir a grade do jogo e permitir que os jogadores façam suas jogadas.
- Verificar condições de vitória ou empate.
- Permitir reiniciar o jogo após um vencedor ou empate.

4. Cronômetro / Timer

Objetivo: Criar um cronômetro com funcionalidades de iniciar, pausar e resetar, e também um timer regressivo.

Tecnologias envolvidas:

- **HTML** e **CSS** para a interface.
- **JavaScript** para controlar o cronômetro e o timer.

Desafio adicional:

- Adicionar um botão para salvar o tempo em que o cronômetro foi parado.
- Permitir que o cronômetro seja configurado para contar regressivamente até um valor especificado pelo usuário.

Requisitos principais:

- Exibir o tempo no formato "00:00:00" (Horas:Minutos:Segundos).
 - Implementar a funcionalidade de **iniciar**, **pausar** e **resetar** o cronômetro.
 - Criar um **timer regressivo** que conta de um valor configurado até zero.
-

5. Consumo de API (ex: Clima, Filmes)

Objetivo: Criar uma aplicação que consome uma API pública e exibe informações como o clima ou filmes populares.

Tecnologias envolvidas:

- **HTML** e **CSS** para a interface.
- **JavaScript** para fazer requisições para a API.
- **Fetch API** ou **Axios** para consumir dados de APIs.

Desafio adicional:

- Adicionar funcionalidades de **pesquisa** para permitir que o usuário insira o nome de uma cidade (para clima) ou filme (para filmes).

- Mostrar informações detalhadas, como temperatura, previsão, ou classificação do filme.

Requisitos principais:

- Utilizar a API do **OpenWeather** (para clima) ou **OMDb API** (para filmes).
 - Exibir informações relevantes como clima atual, temperatura e previsão, ou detalhes sobre o filme (como título, ano, etc.).
 - Implementar uma **pesquisa interativa** para que o usuário possa obter informações sobre diferentes cidades ou filmes.
-

6. CRUD com Armazenamento Local

Objetivo: Criar uma aplicação CRUD (Criar, Ler, Atualizar, Deletar) simples para gerenciar um recurso (por exemplo, contatos ou itens de uma lista) com **localStorage**.

Tecnologias envolvidas:

- **HTML** e **CSS** para a interface.
- **JavaScript** para a lógica de manipulação de dados.
- **localStorage** para armazenar os dados de forma persistente.

Desafio adicional:

- Adicionar a funcionalidade de **editar** um item e salvar as alterações.

- Implementar uma funcionalidade de **ordenação** dos itens ou **pesquisa** de itens específicos.

Requisitos principais:

- Permitir que o usuário adicione, edite e exclua itens.
 - Armazenar os itens no **localStorage** para que persistam após o recarregamento da página.
 - Exibir uma lista dos itens armazenados.
-

7. Aplicação Completa com Front-end e Back-end

Objetivo: Criar uma aplicação web completa com **front-end** e **back-end**, onde o front-end consome dados do back-end, que, por sua vez, interage com um banco de dados.

Tecnologias envolvidas:

- **Front-end:** HTML, CSS, JavaScript, React ou Vue.js.
- **Back-end:** Node.js, Express.js, MongoDB (ou qualquer outro banco de dados).
- **Autenticação e Autorização:** JWT (JSON Web Tokens), OAuth.

Desafio adicional:

- Implementar **autenticação** de usuários com **login** e **registro**.
- Adicionar um sistema de **roles** e **permissões** (administrador, usuário, etc.).

- Implementar funcionalidades de **upload de arquivos**.

Requisitos principais:

- Criar um servidor back-end com **Node.js** e **Express.js**.
 - Criar um front-end interativo utilizando **React** ou **Vue.js**.
 - Conectar o front-end ao back-end para consumir dados de uma API RESTful.
 - Implementar uma base de dados (usando MongoDB ou qualquer outro banco de dados).
-

Conclusão

Agora que você tem uma lista de projetos práticos, o próximo passo é escolher um deles e começar a construir! Lembre-se de que a **prática é essencial** para solidificar o que você aprendeu. Tente implementar os projetos de forma incremental e, à medida que for se sentindo confortável, adicione novos recursos e funcionalidades.

Esses projetos ajudarão a desenvolver habilidades valiosas e a criar um portfólio que pode ser compartilhado com futuros empregadores ou clientes.