

```
File Edit Options Buffers Tools Im-JavaScript
1 var sumOld = function(a, b) {
2   return a + b;
3 };
4
5 // Arrow functions
6
7 var sum = a => ({ });
8
9 console.log(sum(5, 15));
```

ARROW FUNCTIONS IA:

Arrow Functions (Funções de Seta) em JavaScript

Arrow Functions são uma forma **mais curta e moderna** de escrever funções em JavaScript, introduzidas no **ES6 (ES2015)**. Elas simplificam a sintaxe e mudam a forma como o **this** funciona.

O que são Arrow Functions

São funções anônimas escritas com `=>`, ideais para funções pequenas e callbacks.

Sintaxe básica

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

Sintaxe reduzida

- **Um único parâmetro:** parênteses opcionais
- **Uma única expressão:** `return` implícito

```
const dobro = x => x * 2;
```

Arrow Functions e o `this`

Arrow functions **não criam seu próprio `this`**. Elas herdam o `this` do contexto onde foram criadas.

```
function Pessoa() {
  this.idade = 0;

  setInterval(() => {
    this.idade++;
    console.log(this.idade);
  }, 1000);
}
```

📌 Isso evita erros comuns ao usar funções tradicionais em callbacks.

Quando usar

- ✓ Callbacks (`map`, `filter`, `reduce`)
- ✓ Funções curtas e simples
- ✓ Código mais limpo

```
const numeros = [1, 2, 3];
const dobrados = numeros.map(n => n * 2);
```

Quando não usar

- ✗ Métodos de objetos que dependem de `this` próprio
- ✗ Construtores (`new`)
- ✗ Quando precisar de `arguments`

```
const obj = {
  valor: 10,
```

```
mostrar: () => console.log(this.valor) // this incorreto
};
```

Vantagens

- Código mais curto
- Melhor legibilidade
- Evita problemas com `this`

📌 **Resumo:** Arrow functions tornam o JavaScript mais simples e moderno, mas devem ser usadas com cuidado quando o `this` for importante.

```
File Edit Options Buffers Tools IM-Javascript-IDE
1 var obj = {
2   sum(a, b) {
3     return a + b;
4   }
5 }
6
7 console.log(obj);
```



```
File Edit Options Buffers Tools IM-Javascript-IDE
1 var propName = 'test';
2
3 var obj = {};
4
5 obj[propName] = 'prop value';
6
7 console.log(obj);
```

```
File Edit Options Buffers Tools IM-Javascript-IDE JavaScript JavaScript Help
1 var obj = {
2   showContext: function showContext() {
3     console.log(this);
4   },
5   log: function log(value) {
6     console.log(value);
7   }
8 };
9
10console.log(obj.showContext());

```

```
File Edit Options Buffers Tools IM-Javascript-IDE JavaScript JavaScript Help
1 var obj = {
2   showContext: function showContext() {
3     // this = obj
4
5     setTimeout(() => {
6       this.log('after 1000ms');
7     }, 1000);
8   },
9   log: function log(value) {
10    console.log(value);
11  }
12};
13
14obj.showContext();

```

DEFAULT FUNCTION ARGUMENTS

*No javascript, ao multiplicar um número por um parametro indefinido ele retorna nada

*Era comum, quando um desenvolvedor esquecece de dar o valor para o parametro, usar o operador || 1, dizendo que o valor do parâmetro é 1

Também seria possível recorrer a validações de tipos, de identificar o tipo do 2 parametro no código abaxo

```
1 function multiply(a, b) {
2   b = typeof b === 'undefined' ? 1 : b;
3 
4   return a * b;
5 }
6
7 console.log(multiply(5, ));
```

Após o 6, dá para usar o caractere = após o parametro, dá para definir um valor fixo ao parametro

```
function multiply(b = 1, a) {
  return a * b;
}

console.log(multiply(5));
```

A ordem é importante, não pode referenciar antes da variável ser criada

```
1 // lazy evaluation
2 function randomNumber() {
3   console.log('Generating a random number...');
4 
5   return Math.random() * 10;
6 }
7
8 function multiply(a, b = randomNumber()) {
9   return a * b;
10}
11
12console.log(multiply(5));
13console.log(multiply(5));
```

Nesse exemplo, a função foi invocada duas vezes, toda vez que deixa de passar um parâmetro, ela invoca outra função

RESUMO DA IA

Default Function Arguments (Parâmetros Padrão) em JavaScript

Parâmetros padrão permitem definir **valores iniciais** para os parâmetros de uma função caso nenhum valor seja passado (ou seja passado `undefined`). Esse recurso foi introduzido no **ES6 (ES2015)** e ajuda a evitar verificações manuais dentro da função.

Exemplo básico

```
function saudacao(nome = "Visitante") {  
    console.log(`Olá, ${nome}!`);  
}  
  
saudacao();          // Olá, Visitante!  
saudacao("Renan"); // Olá, Renan!
```

Como funciona

- Se o argumento **não for informado** ou for `undefined`, o valor padrão é usado.
- Se for passado `null`, o valor padrão **não é aplicado**.

```
function teste(valor = 10) {  
    console.log(valor);  
}  
  
teste();          // 10  
teste(undefined); // 10  
teste(null);     // null
```

Expressões como valor padrão

Os valores padrão podem ser **expressões** ou chamadas de função:

```
function gerarId(id = Math.random()) {  
    return id;  
}
```

Dependência entre parâmetros

Um parâmetro pode usar o valor de outro:

```
function calcular(a, b = a * 2) {  
    return a + b;  
}  
  
calcular(5); // 15
```

Vantagens

- Código mais limpo e legível
- Menos verificações com `if` ou `||`
- Evita erros com parâmetros ausentes

💡 **Resumo:** Default function arguments tornam funções mais seguras e organizadas, garantindo valores padrão quando os argumentos não são fornecidos.

ENHANCED OBJECT LITERALS

Enhanced Object Literals (Objetos Literais Aprimorados) em JavaScript

Enhanced Object Literals são recursos introduzidos no **ES6 (ES2015)** que tornam a criação de objetos **mais curta, legível e expressiva**, reduzindo código repetitivo.

Principais recursos

1. Property Shorthand

Quando o nome da propriedade é igual ao nome da variável, pode-se omitir a repetição:

```
const nome = "Renan";  
const idade = 17;  
  
const pessoa = { nome, idade };
```

2. Métodos com sintaxe simplificada

Não é mais necessário usar a palavra `function`:

```
const usuario = {
  login() {
    return "Usuário logado";
  }
};
```

3. Propriedades computadas

Permite usar expressões como nome de propriedades:

```
const campo = "email";

const cliente = {
  [campo]: "teste@email.com"
};
```

4. Combinação com funções e variáveis

Facilita retornar objetos em funções:

```
function criarProduto(nome, preco) {
  return { nome, preco };
}
```

Vantagens

- Código mais curto e organizado
- Melhor legibilidade
- Menos repetição
- Facilita manutenção

 **Resumo:** Enhanced Object Literals simplificam a criação de objetos em JavaScript, tornando o código mais moderno, limpo e eficiente.

REST E SPREAD OPERATOR

```
1 function sum(a, b) {  
2   return a + b;  
3 }  
4  
5 console.log(sum(5, 5));
```

Essa função recebe dois parâmetros, soma seus argumentos e retorna a soma deles

*Arguments é uma variável reservada que refere-se a todos os argumentos que possui uma função

```
1 function sum(a, b) {  
2   var value = 0;  
3  
4   for (var i = 0; i < arguments.length; i++) {  
5     value += arguments[i];  
6   }  
7  
8   return value;  
9 }  
10  
11console.log(sum(5, 5, 5, 2, 3));
```

O Rest operator escreve-se com ... pontos, dentro da lista de argumentos, na sua frente, defini-se um valor para sua variável

O Rest operator pode ser usado para pegar parâmetros restantes

O Spread operator, nessa função abaixo, ele pega todos os itens do array e transforma em parâmetro, ele não se limita a listas, ele pode ser usado em strings, arrays e objetos para construir outros objetos literais e objetos iterável

```
File Edit Options Buffers Tools IM-Javascript-IDE JavaScript Help  
1 // rest operator, spread operator ...  
2  
3 const multiply = (...args) => args.reduce((acc, value) => acc * value, 1);  
4  
5 const sum = (...rest) => {  
6   return multiply(...rest);  
7 }  
8  
9 console.log(sum(5, 5, 5, 2, 3));
```

Resumindo, tem a função de quebrar os itens e os repassar para algum lugar.

String e arrays são iteráveis

```
File Edit Options Buffers Tools IM-Javascript-IDE JavaScript
1 // rest operator, spread operator ...
2 // strings, arrays, literal objects e objetos iteráveis
3
4 const str = 'Digital Innovation One';
5 const arr = [1, 2, 3, 4];
6
7 function logArgs(a, b, c) {
8   console.log(a, b, c);
9 }
10
11const arr2 = arr.concat([5, 6, 7]);
12
13console.log(arr2);
```

REST e SPREAD Operator (...) em JavaScript — explicação completa e clara

Em JavaScript, o **REST** e o **SPREAD Operator** usam a **mesma sintaxe (...)**, mas o **comportamento muda conforme o contexto**. Ambos foram introduzidos no **ES6 (ES2015)** e são essenciais para escrever código moderno, limpo e flexível.

◆ REST Operator (...)

O que é

O **REST** é usado para **juntar vários valores em um único array**. Ele aparece principalmente em **parâmetros de funções ou desestruturação**.

💡 Pense no REST como: “*juntar o resto*”.

REST em funções

Permite receber uma quantidade indefinida de argumentos:

```
function soma(...numeros) {  
    return numeros.reduce((total, n) => total + n, 0);  
}  
  
soma(1, 2, 3, 4); // 10
```

→ Todos os valores passados viram um **array**.

REST na desestruturação

```
const aluno = {  
    nome: "Renan",  
    idade: 17,  
    curso: "Desenvolvimento de Sistemas"  
};  
  
const { nome, ...dados } = aluno;  
  
console.log(nome); // Renan  
console.log(dados); // { idade: 17, curso: "Desenvolvimento de Sistemas" }
```

✓ Vantagens do REST

- Aceita número variável de parâmetros
 - Substitui o uso antigo de **arguments**
 - Código mais limpo e organizado
-

◆ SPREAD Operator (...)

O que é

O **SPREAD** faz o oposto do REST: ele **espalha os valores** de um array ou objeto em outro contexto.

👉 Pense no SPREAD como: “espalhar”.

SPREAD em arrays

```
const a = [1, 2, 3];
const b = [...a, 4, 5];

console.log(b); // [1, 2, 3, 4, 5]
```

SPREAD em objetos

```
const usuario = {
  nome: "Renan",
  idade: 17
};

const usuarioCompleto = {
  ...usuario,
  curso: "TDS"
};

console.log(usuarioCompleto);
```

SPREAD em chamadas de função

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

const valores = [2, 4, 6];
soma(...valores); // 12
```

Vantagens do SPREAD

- Clona arrays e objetos
 - Evita mutações indesejadas
 - Facilita junção de dados
-

Diferença principal (muito importante)

Operador	O que faz
REST	Junta vários valores em um array
SPREAD	Espalha os valores de um array ou objeto

Resumo rápido:

- REST → recebe
 - SPREAD → envia
-

Conclusão

REST e SPREAD são fundamentais no JavaScript moderno. Eles tornam o código **mais flexível, legível e seguro**, especialmente ao trabalhar com funções, arrays, objetos e programação funcional.

Se quiser, posso te passar **exercícios, resumo para prova, ou comparação com exemplos reais de backend e frontend** 🧑‍💻✨

Outra forma de utilizar o spread é utilizar ele em objetos literais, porém só podem ser usados para construir novos objetos

```
3
4 const str = 'Digital Innovation One';
5 const arr = [1, 2, 3, 4];
6
7 function logArgs(a, b, c) {
8   console.log(a, b, c);
9 }
10
11const arrClone = [...arr];
12
13const obj = {
14   test: 123
15};
16
17const obj2 = {
18   ...obj
19   test2: 'hello'
20};
21
22console.log(obj2);
```

Só pode ser utilizado em objetos literais não iteráveis

Um objeto é iterável (iterable), se ele define seu comportamento de iteração, como no caso de quais valores percorridos em um laço do tipo for..of.

```
const obj = {  
  test: 123  
};  
  
const obj2 = {  
  test: 456  
};  
  
const objMerged = {  
  ...obj  
};  
  
console.log(objMerged);
```

Também serve para gerar clone de objetos como no exemplo abaixo

```
3  
4 const obj = {  
5   test: 123  
6 };  
7  
8 const obj2 = { ...obj };  
9  
10 obj2.test = 456;  
11 console.log(obj);
```

Ele é um clone raso, pois o objeto tem propriedades que são outros objetos

```
4 const obj = {  
5   test: 123,  
6   subObj: {  
7     test: 123  
8   }  
9 };  
10  
11const obj2 = { ...obj };  
12  
13obj2.subObj.test = 456;  
14  
15console.log(obj);  
■
```

DESTRUCTURING EM REACTJS

O Comum no código seria como no exemplo:

```
1 // Destructuring Assignment  
2  
3var arr = ['Apple', 'Banana', 'Orange'];  
4  
5var apple = arr[0];  
6var banana = arr[1];  
7var orange = arr[2];  
■
```

O problema é que essa fórmula é muito verbosa e inviável pela quantidade de itens

```
1 // Destructuring Assignment  
2  
3var [apple2, banana2, orange2] = ['Apple', 'Banana', 'Orange'];  
4  
5console.log(apple, apple2);  
■
```

Ele destruiu o array e atribuiu variáveis

 [Destructuring em ReactJS – explicação clara e prática](#)

Destructuring é um recurso do JavaScript (ES6) muito usado no **React** para **extrair valores de objetos e arrays de forma simples e legível**. No React, ele aparece o tempo todo em **props, state, hooks e imports**.

◆ O que é Destructuring

É uma forma de “desmontar” objetos ou arrays, pegando apenas os valores que você precisa.

Exemplo simples (JavaScript puro)

```
const pessoa = {  
    nome: "Renan",  
    idade: 17  
};  
  
const { nome, idade } = pessoa;
```

◆ Destructuring de props (uso mais comum no React)

Em vez de acessar `props.nome`, você pode extrair direto:

✗ Sem destructuring

```
function Card(props) {  
    return <h1>{props.titulo}</h1>;  
}
```

✓ Com destructuring

```
function Card({ titulo }) {  
    return <h1>{titulo}</h1>;  
}
```

📌 Isso deixa o código mais **limpo e fácil de ler**.

◆ Destructuring no state (useState)

O `useState` retorna um `array`, então usamos destructuring de array:

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

- `contador` → valor do estado
 - `setContador` → função para alterar o estado
-

◆ Destructuring em eventos

```
function handleChange({ target }) {
  console.log(target.value);
}
```

Ou ainda mais direto:

```
function handleChange({ target: { value } }) {
  console.log(value);
}
```

◆ Destructuring em props com valores padrão

```
function Botao({ texto = "Clique aqui", cor = "blue" }) {
  return <button style={{ color: cor }}>{texto}</button>;
}
```

◆ Destructuring em objetos dentro de props

```
function Perfil({ usuario: { nome, email } }) {
  return (
    <>
    <p>{nome}</p>
```

```
        <p>{email}</p>
      </>
    );
}
```

◆ Destructuring com rest operator

```
function Input({ label, ...props }) {
  return (
    <>
      <label>{label}</label>
      <input {...props} />
    </>
  );
}
```

👉 Muito usado para **componentes reutilizáveis**.

✓ Vantagens do destructuring no React

- Código mais curto
 - Melhor legibilidade
 - Menos repetição (**props.**)
 - Facilita manutenção
-

🧠 Resumo final

- Destructuring é essencial no React
- Muito usado em **props, hooks e eventos**

- Combina perfeitamente com REST/SPREAD
- Deixa o código mais profissional e moderno

```
var obj = {  
  name: 'Celso'  
};  
  
// Destructuring Assignment  
  
var { name } = obj;  
  
console.log(name);
```

Ele destroi o array, e procura um valor para a propriedade name

```
File Edit Options BUTTERS TOOLS LM-Javascript-IDE JavaScript JavaScri  
1 var arr = ['Apple', 'Orange'];  
2 var obj = {  
3   name: 'Celso'  
4 };  
5  
6 // Destructuring Assignment  
7  
8 var [apple2] = arr;  
9 var { name: name2 } = obj;  
10  
11name2 = 'Henrique';  
12  
13console.log(obj);
```

Mesmo atribuindo valor a name2, continua lendo celso como valor da variável name

```
1 var arr = ['Apple', 'Orange'];
2 var obj = {
3   name: 'Celso',
4   props: {
5     age: 26,
6     favoriteColors: ['black', 'blue']
7   }
8 };
9
10 var age = obj.props.age;
11 // Destructuring Assignment
12
13 var [apple2] = arr;
14 var {
15   props: {
16     age: age2,
17     favoriteColor: [color1, color2]
18   }
19 } = obj;
20
21 console.log(color1);
```

```
1 var arr = [{ name: 'Apple', type: 'fruit' }];
2 var obj = {
3   name: 'Celso',
4   props: {
5     age: 26,
6     favoriteColors: ['black', 'blue']
7   }
8 };
9
10 var fruit1 = arr[0].name;
11 // Destructuring Assignment
12
13 var { name: fruit } = arr;
14
15 console.log(name);
```

```
1 var arr = [{ name: 'Apple', type: 'fruit' }];
2 var obj = {
3   name: 'Celso',
4   props: {
5     age: 26,
6     favoriteColors: ['black', 'blue']
7   }
8 };
9
10var fruit1 = arr[0].name;
11
12// Destructuring Assignment
13
14let [{ name: fruitName }] = arr;
15
16// functions
17
18function sum([a, b] = []) {
19  return a + b;
20}
21
22console.log(sum([], 5));
```

```
var arr = [{ name: 'Apple', type: 'fruit' }];
var obj = {
  name: 'Celso',
  props: {
    age: 26,
    favoriteColors: ['black', 'blue']
  }
};

var fruit1 = arr[0].name;
// Destructuring Assignment
let [{ name: fruitName }] = arr;
// functions
function sum({ a, b }) {
  return a + b;
}
console.log(sum({ a: 5, b: 5 }));
```

GENERATORS

Symbols são maneiras de gerar um identificador único
Ele é único, um identificador único

```
// Symbols  
2const uniqueId = Symbol('Hello');  
4const uniqueId2 = Symbol('Hello');  
5  
6console.log(uniqueId === uniqueId2);
```

Pode ser utilizado para gerar propriedades privadas

```
// Symbols  
  
const uniqueId = Symbol('Hello');  
  
const obj = {  
  [uniqueId]: 'Hello',  
  _id  
};  
  
console.log(obj);
```

O Symbol tem uma série de propriedades chamadas de know symbols:

Exemplos:

- › Symbol.iterator
- › Symbol(Symbol.iterator)
- › Symbol.split
- › Symbol(Symbol.split)
- › Symbol.toPrimitive
- › Symbol(Symbol.toPrimitive)
- › Symbol.asyncIterator
- › Symbol(Symbol.asyncIterator)
- › |

```
const uniqueId = Symbol('Hello');

// Well known symbols

Symbol.iterator;
Symbol.split;
Symbol.toStringTag;
0
1const arr = [1, 2, 3, 4];
2
3const it = arr[Symbol.iterator]();
4
5console.log(it.next());
```

Essa interface é responsável pela iteração do array
Cada vez que invoca o n

```
const uniqueId = Symbol('Hello');

// Well known symbols

Symbol.iterator;           []
Symbol.split;
Symbol.toStringTag;
0
1const arr = [1, 2, 3, 4];
2
3const it = arr[Symbol.iterator]();
4
5console.log(it.next());
6console.log(it.next());
7console.log(it.next());
8console.log(it.next());
9console.log(it.next());
```

ext, ele vai trazendo os valores que tem o array

```
const uniqueId = Symbol('Hello');

// Well known symbols

Symbol.iterator;
Symbol.split;
Symbol.toStringTag;

const arr = [1, 2, 3, 4];
const it = arr[Symbol.iterator]();

console.log(it.next());
console.log(it.next());
console.log(it.next());
console.log(it.next());
console.log(it.next());
```

Porém na última, o valor vem indefinido

◆ Symbols no JavaScript (ES6)

Symbol é um **tipo primitivo** introduzido no ES6 que cria **identificadores únicos**.

📍 Características principais

- Cada **Symbol()** é **único**, mesmo com a mesma descrição
- Muito usado para **evitar conflitos de nomes** em objetos
- Não aparece em loops comuns (**for...in**, **Object.keys()**)

🧠 Por que usar?

- Para criar propriedades “privadas”
- Para evitar sobrescrita accidental de atributos

 **Exemplo**

```
const id = Symbol("id");

const usuario = {
  nome: "Renan",
  [id]: 123
};

console.log(usuario[id]); // 123
```

 **Symbol como chave interna**

O JavaScript usa **Symbols internos**, como:

- `Symbol.iterator`
 - `Symbol.toStringTag`
-

 **Iterators no JavaScript (ES6)**

Iterators permitem percorrer elementos de uma coleção (arrays, strings, maps, etc.).

 **O que é um Iterator?**

É um objeto que implementa o método:

```
next()
```

Que retorna:

```
{ value: ..., done: true/false }
```

 **Exemplo de Iterator manual**

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

```
const iterator = numeros[Symbol.iterator]();

console.log(iterator.next()); // { value: 1, done: false }
console.log(iterator.next()); // { value: 2, done: false }
```

◆ Relação entre Symbol e Iterator

O **Symbol Symbol.iterator** define se um objeto é **iterável**.

Se um objeto possui esse Symbol, ele pode ser usado em:

- `for...of`
- `spread (...)`
- `Array.from()`

Exemplo

```
const texto = "JS";

for (let letra of texto) {
  console.log(letra);
}
```

◆ Criando um Iterator personalizado

```
const contador = {
  inicio: 1,
  fim: 3,
  [Symbol.iterator]() {
    let atual = this.inicio;
    let fim = this.fim;

    return {
      next() {
```

```
        if (atual <= fim) {
            return { value: atual++, done: false };
        }
        return { done: true };
    }
};

for (let n of contador) {
    console.log(n);
}
```

✓ Resumindo em poucas palavras

- **Symbol** → cria chaves únicas e seguras
- **Iterator** → controla como os dados são percorridos
- **Symbol.iterator** → conecta os dois conceitos
- Base do funcionamento do `for...of`, `spread` e `Array.from()`

RESUMO IA ACIMA

```
1 // Symbols
2
3 const uniqueId = Symbol('Hello');
4
5 // Well known symbols
6
7 Symbol.iterator;
8
9 const arr = [1, 2, 3, 4];
10const str = 'Digital Innovation One';
11
12console.log(arr[Symbol.iterator]().next());
13
14const obj = {
15   values: [1, 2, 3, 4],
16   [Symbol.iterator]() {
17     return {
18       next() {
19         if (this.values.length === 0) {
20           return { done: true };
21         }
22         return { value: this.values.shift() };
23       }
24     };
25   }
26};
27
28const iterator = obj[Symbol.iterator]();
29
30iterator.next();
31
32// { value: 1, done: false }
33
34iterator.next();
35
36// { value: 2, done: false }
37
38iterator.next();
39
40// { value: 3, done: false }
41
42iterator.next();
43
44// { value: 4, done: false }
45
46iterator.next();
47
48// { done: true }
```

A Cada vez que a pessoa fizer um next, tem que dar um return a algum objeto

```
3 const uniqueId = Symbol('Hello');
4
5 // Well known symbols
6
7 Symbol.iterator;
8
9 const arr = [1, 2, 3, 4];
10 const str = 'Digital Innovation One';
11
12 const obj = {
13   values: [1, 2, 3, 4],
14   [Symbol.iterator]() {
15     let i = 0;
16
17     return {
18       next: () => {
19         i++;
20
21         return {
22           value: this.values[i - 1],
23           done: i > this.values.length
24         };
25       }
26     };
27   }
28 };
29
30 const it = obj[Symbol.iterator]();
31
32 console.log(it.next());
33   get value
```

GENERATORS

```
const uniqueId = Symbol('Hello');

// Well known symbols

Symbol.iterator;

const arr = [1, 2, 3, 4];
const str = 'Digital Innovation One';

const obj = {
  values: [1, 2, 3, 4],
  [Symbol.iterator]() {
    let i = 0;

    return {
      next: () => {
        i++;

        return {
          value: this.values[i - 1],
          done: i > this.values.length
        };
      }
    };
  }
};


```

Resumo ia

Claro! Aqui vai uma **explicação completa, clara e bem didática sobre Generators no JavaScript (ES6)**, mantendo o mesmo nível das anteriores ↪

◆ Generators no JavaScript (ES6)

Generators são funções especiais que podem pausar e retomar sua execução, produzindo valores sob demanda.

Eles são muito usados para:

- Criar **iterators personalizados**
- Trabalhar com **fluxos de dados**
- Simplificar lógicas complexas de iteração



Como reconhecer um Generator?

Um generator é declarado com:

- `function*`
- Usa a palavra-chave `yield`



Exemplo básico

```
function* contador() {  
    yield 1;  
    yield 2;  
    yield 3;  
}  
  
const gen = contador();  
  
console.log(gen.next()); // { value: 1, done: false }  
console.log(gen.next()); // { value: 2, done: false }
```

◆ `yield` vs `return`

- `yield` → pausa a função e devolve um valor
- `return` → encerra o generator definitivamente

```
function* exemplo() {  
    yield "A";  
    return "B";  
    yield "C"; // nunca será executado  
}
```

◆ Generators são Iterators

Todo **generator** é automaticamente um iterator
Ele já implementa o `Symbol.iterator`.

Uso com `for...of`

```
function* letras() {  
    yield "J";  
    yield "S";  
}  
  
for (let l of letras()) {  
    console.log(l);  
}
```

◆ Generator infinito

Generators podem produzir valores infinitamente, sem travar o programa.

```
function* infinito() {  
    let i = 0;  
    while (true) {  
        yield i++;  
    }  
}  
  
const gen = infinito();  
console.log(gen.next().value); // 0  
console.log(gen.next().value); // 1
```

◆ Passando valores para o Generator

É possível enviar valores para dentro do generator usando `next(valor)`.

```
function* soma() {  
    let valor = yield 10;  
    yield valor + 5;  
}  
  
const gen = soma();  
gen.next();      // inicia  
console.log(gen.next(20).value); // 25
```

◆ **yield*** (delegação)

O **yield*** delega a execução para outro iterator ou generator.

```
function* gen1() {  
    yield 1;  
    yield 2;  
}
```

```
function* gen2() {  
    yield* gen1();  
    yield 3;  
}
```

◆ Diferença entre Iterator manual e Generator

Iterator manual	Generator
Código longo	Código simples
Controle manual de next()	Controle automático
Mais verboso	Mais legível



Resumo rápido (estilo prova / slide)

- **Generator** = função pausável
- Usa **function*** e **yield**
- Implementa **Symbol.iterator**
- Ideal para iteradores complexos ou infinitos
- Base para controle avançado de fluxo

Se quiser, posso:

- Comparar **generator vs async/await**
- Criar **exemplos práticos de prova**
- Relacionar com **React, Node.js ou Streams**

Pode ser usado para construir iteradores de maneira mais simplificada

```
22         value: this.values[i - 1],
23         done: i > this.values.length
24     );
25 }
26 };
27 }
28};
29
30// Generators
31
32function* hello() {
33     console.log('Hello');
34     yield 1;
35
36     console.log('From');
37     const value = yield 2;
38
39     console.log(value);
40}
41
42const it = hello();
43
44console.log(it.next());
45console.log(it.next());
46console.log(it.next('Outside!'));
```

```
function* naturalNumbers() {
    let number = 0;
    while (true) {
        yield number;
        number++;
    }
}

const it = naturalNumbers();

console.log(it.next());
console.log(it.next());
console.log(it.next());
console.log(it.next());
```

Os generators também são usados para construir a interface dos objetos iteráveis

```
3 const uniqueId = Symbol('Hello');
4
5 // Well known symbols
6
7 Symbol.iterator;
8
9 const arr = [1, 2, 3, 4];
10const str = 'Digital Innovation One';
11
12// Generators
13const obj = {
14  values: [1, 2, 3, 4],
15  *[Symbol.iterator]() {
16    for (var i = 0; i < this.values.length; i++) {
17      yield this.values[i];
18    }
19  }
20};
21
22for (let value of obj) {
23  console.log(value);
24}
```

Além de funções com pausa, na função acima foi usado para construir uma meta propriedade em const obj

CALLBACKS E PROMISES:

```
function doSomething(callback) {
  setTimeout(function() {
    // did something
    callback('First data');
  }, 1000);
}

function doOtherThing(callback) {
  setTimeout(function() {
    // did other thing
    callback('Second data');
  }, 1000);
}
```

A Função abaixo é um exemplo para demonstrar processamento de dados

O que são Callbacks?

Callback é uma função passada como argumento para outra função, para ser executada depois que uma tarefa assíncrona termina.

👉 Muito usado antes do ES6 para lidar com:

- `setTimeout`
- Requisições
- Leitura de arquivos

📌 Ideia central

“Quando terminar, chama essa função.”

◆ Analisando o código (CALLBACKS)

1 Função `doSomething`

```
function doSomething(callback) {  
  setTimeout(function() {  
    callback('First data');  
  }, 1000);  
}
```

- Simula uma operação assíncrona
 - Após 1 segundo, executa o `callback`
 - Retorna "First data"
-

2 Função `doOtherThing`

```
function doOtherThing(callback) {
```

```
setTimeout(function() {
  callback('Second data');
}, 1000);
}
```

- Mesmo comportamento
 - Retorna "Second data"
-

3 Função doAll

```
function doAll() {
  try {
    doSomething(function(data) {
      var processedData = data.split('');
```

- Chama `doSomething`
 - Recebe o resultado no callback (`data`)
 - Processa os dados (`split`)
-

4 Callback dentro de callback (problema)

```
doOtherThing(function(data2) {
  var processedData2 = data2.split('');
```

⚠ Aqui surge o **Callback Hell**:

- Função dentro de função
- Código difícil de ler
- Difícil de tratar erros

5 setTimeout final

```
setTimeout(function() {  
    console.log(processedData, processedData2);  
, 1000);
```

- Apenas imprime os resultados depois de outro delay
-

✗ Problema do código atual

- **try/catch NÃO captura erro assíncrono**
 - Código difícil de manter
 - Escalabilidade ruim
-

🔥 O que são Promises?

Promise representa um valor que:

- Pode estar disponível no futuro
- Pode **resolver (resolve)** ou **falhar (reject)**

Estados:

- **pending**
 - **fulfilled**
 - **rejected**
-

♦ Refatorando o código para PROMISES (completo)

✓ Versão com Promises

```
function doSomething() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve('First data');
    }, 1000);
  });
}

function doOtherThing() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve('Second data');
    }, 1000);
  });
}
```

♦ Encadeamento com .then()

```
function doAll() {
  doSomething()
    .then(data => {
      const processedData = data.split('');
      return doOtherThing()
        .then(data2 => {
          const processedData2 = data2.split('');
          return { processedData, processedData2 };
        });
    })
    .then(result => {
      console.log(result.processedData, result.processedData2);
    })
    .catch(err => {
      console.error('Erro:', err);
    });
}
```

```
}

doAll();
```

★ Versão MODERNA (async / await) — a melhor prática

```
async function doAll() {
  try {
    const data = await doSomething();
    const processedData = data.split('');

    const data2 = await doOtherThing();
    const processedData2 = data2.split('');

    console.log(processedData, processedData2);
  } catch (err) {
    console.error('Erro:', err);
  }
}

doAll();
```

🧠 Comparação rápida

Callbacks	Promises / async
-----------	---------------------

Código aninhado	Código linear
-----------------	---------------

Difícil de manter	Fácil leitura
-------------------	---------------

Erro difícil	<code>catch</code> funciona
--------------	-----------------------------

Callback Hell	Código limpo
---------------	--------------

✓ Conclusão (nível prova / explicação oral)

- **Callbacks** foram a primeira solução para async no JS
- Geram **callback hell**
- **Promises** resolvem o problema
- `async/await` é apenas sintaxe mais limpa sobre Promises
- Seu código é um exemplo clássico de **evolução do JS**

◆ O que é **async** no JavaScript?

`async` é uma **palavra-chave** do JavaScript que transforma uma função em **assíncrona**, permitindo escrever código assíncrono de forma **simples, linear e legível**, como se fosse síncrono.

👉 Ela foi introduzida no **ES2017** e é baseada em **Promises**.

📌 O que uma função **async** faz?

Quando você declara uma função como `async`:

1. Ela **sempre retorna uma Promise**
2. Permite usar a palavra-chave `await` dentro dela
3. Facilita o tratamento de erros com `try/catch`

```
async function exemplo() {  
  return 10;  
}
```

Equivale a:

```
function exemplo() {  
  return Promise.resolve(10);
```

```
}
```

◆ O que é **await**?

await pausa a execução da função **async** até que a Promise seja resolvida.

⚠ Importante:

- **await** só funciona dentro de funções **async**
- Ele não trava o **JavaScript**, apenas a função

```
const resultado = await algumaPromise();
```

◆ Exemplo simples

✗ Com Promise (.then)

```
function buscarDados() {  
  return fetch('/api').then(res => res.json());  
}
```

✓ Com **async/await**

```
async function buscarDados() {  
  const resposta = await fetch('/api');  
  const dados = await resposta.json();  
  return dados;  
}
```

✓ Mesmo resultado

✓ Código mais limpo

◆ Async na prática (exemplo didático)

```
function atraso() {  
    return new Promise(resolve => {  
        setTimeout(() => resolve("Pronto"), 1000);  
    });  
}  
  
async function executar() {  
    console.log("Início");  
    const msg = await atraso();  
    console.log(msg);  
    console.log("Fim");  
}  
  
executar();
```

⬆️ Saída:

```
Início  
(1 segundo)  
Pronto  
Fim
```

◆ Tratamento de erros com async

```
async function executar() {  
    try {  
        const dados = await promessaComErro();  
        console.log(dados);  
    } catch (erro) {  
        console.error("Erro:", erro);  
    }  
}
```

✓ Substitui .catch()

◆ Async NÃO é...

- ✗ Não é paralelismo
- ✗ Não cria threads
- ✗ Não deixa o JS multithread

👉 Ele apenas **organiza Promises de forma mais clara**

As promises permitem que encadeiam uma nas outras

```
2 const doSomethingPromise = () =>
3   new Promise((resolve, reject) => {
4     setTimeout(function() {
5       // did something
6       resolve('First data');
7     }, 1500);
8   });
9
10
11const doOtherThingPromise = () =>
12  new Promise((resolve, reject) => {
13    setTimeout(function() {
14      // did something
15      resolve('Second data');
16    }, 1000);
17  });
18
19Promise.race([doSomethingPromise(), doOtherThingPromise()]).then(data => {
20  console.log(data);
21});
```

Fetch, Async/Await e EventEmitter

A Fetch faz requisições, mas trabalha utilizando promise

```
File Edit Options Buttons Tools Help
1 fetch('/data.json').then(responseStream => {
2   responseStream.json().then(data => {
3     console.log(data);
4   });
5 });
```

O then retorna os dados

*Se tiver um erro de rede, ele será identificado no catch

```
1 fetch('http://localhost:8080/dataXPTO.json')
2   .then(responseStream => responseStream.json())
3   .then(data => {
4     console.log(data);
5   })
6   .catch(err => {
7     console.log('Erro: ', err);
8   });

```

Só acontece um erro de fato no fetch ao tentar transformar em .json

O **.json** no contexto de JavaScript (JS) se refere principalmente a dois conceitos inter-relacionados:

1. **O formato de dados JSON (JavaScript Object Notation):** É um formato leve de troca de dados, muito utilizado para enviar dados do servidor para a página web (e vice-versa).
2. **Métodos e Operações do JS que lidam com JSON:** São as funções nativas do JavaScript que permitem trabalhar com esse formato de dados, como parsear (converter) strings JSON em objetos JS e vice-versa.

Aqui está um detalhamento de cada um:

1. O que é JSON?

JSON (JavaScript Object Notation) é um formato de texto que armazena dados de uma maneira estruturada e de **fácil leitura para humanos e fácil processamento para máquinas**.

Características-chave:

- **Derivado do JavaScript:** Embora o nome sugira uma ligação forte, o JSON é um formato de dados independente da linguagem. Quase todas as linguagens de programação (Python, Java, PHP, etc.) têm bibliotecas para ler e escrever JSON.
- **Representação de Dados:** Ele representa dados como **pares de nome/valor** (como propriedades de um objeto JS) e **listas ordenadas de valores** (como arrays JS).
- **Extensão de Arquivo:** Arquivos que armazenam dados neste formato geralmente usam a extensão **.json**.

Exemplo de Estrutura JSON:

```
JSON
{
  "nome": "João da Silva",
  "idade": 30,
  "cidades_visitadas": ["São Paulo", "Rio de Janeiro", "Brasília"],
  "ativo": true
}
```

2. Como o JavaScript Usa JSON?

A principal função do JavaScript em relação ao JSON é a conversão. O objeto global **JSON** nativo do JavaScript fornece dois métodos essenciais:

A. **JSON.parse()**

- **Para que serve:** Converte uma **string JSON** (texto) em um **Objeto JavaScript** que você pode manipular.
- **Cenário de Uso:** Quando você recebe dados do servidor (via uma requisição HTTP, como AJAX ou **fetch**), eles quase sempre vêm como uma string. Você precisa do **JSON.parse()** para transformá-los em um objeto que o seu código JS possa usar.

Exemplo:

```
JavaScript
const jsonString = '{"produto": "Notebook", "preco": 3500};

// Converte a string JSON em um Objeto JS
const objetoJs = JSON.parse(jsonString);

console.log(objetoJs.produto); // Saída: Notebook
console.log(typeof objetoJs); // Saída: object
```

B. **JSON.stringify()**

- **Para que serve:** Converte um **Objeto JavaScript** em uma **string JSON** (texto).
- **Cenário de Uso:** Quando você precisa enviar dados do seu código JS de volta para o servidor. Antes de enviar, você precisa converter o objeto em uma string para que o servidor possa recebê-lo e processá-lo.

Exemplo:

```
JavaScript
```

```
const objetoJs = {  
    usuario: "maria.souza",  
    status: "online"  
};  
  
// Converte o Objeto JS em uma string JSON  
const stringJson = JSON.stringify(objetoJs);  
  
console.log(stringJson); // Saída: '{"usuario":"maria.souza","status":"online"}'  
console.log(typeof stringJson); // Saída: string
```

Resumo da Função no JS:

O JSON é o **formato padrão para a comunicação** na web moderna, e o JavaScript usa o objeto **JSON** nativo (com seus métodos **parse** e **stringify**) para **traduzir** entre o formato de dados web (JSON string) e o formato de dados que o código JS pode manipular (Objeto JS).

O Fetch só vai dar um erro na promise caso aconteca um erro de rede, o status não faz acontecer o erro

```
1 fetch('http://localhost:8080/dataXPTO.json')  
2   .then(responseStream => {  
3     console.log(responseStream);  
4     if (responseStream.status === 200) {  
5       responseStream.json();  
6     }  
7   })  
8   .then(data => {  
9     console.log(data);  
10  })  
11  .catch(err => {  
12    console.log('Erro: ', err);  
13  });
```

```
1 fetch('http://localhost:8080/dataXPTO.json')
2   .then(responseStream => {
3     if (responseStream.status === 200) {
4       return responseStream.json();
5     } else {
6       throw new Error('Request error');
7     }
8   })
9   .then(data => {
10    console.log(data);
11  })
12  .catch(err => {
13    console.log('Error: ', err);
14  });

```

dessa forma, o erro só acontece pois o status é maior que 200

Async/Await

É uma forma mais simples de criar promise e ao mesmo tempo criar uma promise dentro da outra

Resumo sobre fetch no JavaScript (ES6)

O **fetch** é uma API moderna do JavaScript usada para **fazer requisições HTTP** (como GET, POST, PUT, DELETE) e consumir dados de servidores, geralmente em **JSON**.

Ela substitui o uso antigo do **XMLHttpRequest**, sendo **mais simples, legível e baseada em Promises**.

◆ Características principais

- Retorna uma **Promise**
- Funciona de forma **assíncrona**
- Usa métodos HTTP padrão
- Muito usada para consumir **APIs REST**
- Integra bem com **Promises e async/await**

◆ Sintaxe básica

```
fetch(url)
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error(error));
```

📌 Explicação:

- `fetch(url)` → faz a requisição
- `response` → resposta do servidor
- `response.json()` → converte a resposta para objeto JS
- `catch()` → trata erros

◆ Exemplo com método POST

```
fetch("https://api.exemplo.com/usuarios", {
  method: "POST",
  headers: {
    "Content-Type": "application/json"
  },
  body: JSON.stringify({
    nome: "Renan",
    idade: 17
  })
})
.then(res => res.json())
.then(data => console.log(data));
```

◆ Usando `async/await` (mais moderno)

```
async function buscarDados() {
```

```
try {
  const response = await fetch(url);
  const data = await response.json();
  console.log(data);
} catch (erro) {
  console.error(erro);
}
}
```

✓ Vantagens do fetch

- Código mais limpo
- Fácil de ler e manter
- Baseado em Promises
- Padrão moderno do JS

Resumo IA Fetch Acima

Resumo IA Async/Await:

```
// ES7 - Async / Await

const simpleFunc = async () => {
  return 12345;
};

simpleFunc().then(data => {
  console.log(data);
});
```

Resumo sobre `async / await` no JavaScript

`async` e `await` são recursos do JavaScript moderno que facilitam o trabalho com **operações assíncronas**, deixando o código mais **legível** e parecido com código síncrono. Eles são baseados em **Promises** e não substituem, mas simplificam o uso delas.

♦ **async**

- Usado antes de uma função
- Faz com que a função **sempre retorne uma Promise**
- Permite o uso do `await` dentro dela

```
async function exemplo() {  
    return "Olá";  
}  
// retorna uma Promise
```

♦ **await**

- Usado apenas **dentro de funções async**
- Pausa a execução da função até a Promise ser resolvida
- Evita o uso excessivo de `.then()`

```
const resposta = await fetch(url);
```

♦ **Exemplo prático**

```
async function buscarUsuarios() {  
    try {  
        const response = await fetch("https://api.exemplo.com/usuarios");  
        const dados = await response.json();  
        console.log(dados);  
    } catch (erro) {  
        console.error("Erro:", erro);  
    }  
}
```

📌 O que acontece aqui:

1. `fetch()` faz a requisição
 2. `await` espera a resposta
 3. `response.json()` converte os dados
 4. `try/catch` trata erros
-

♦ Comparação com Promises

Com Promise

```
fetch(url)
  .then(res => res.json())
  .then(data => console.log(data))
  .catch(err => console.error(err));
```

Com `async/await`

```
const res = await fetch(url);
const data = await res.json();
```

→ Código mais limpo e fácil de entender.

✓ Vantagens do `async/await`

- Melhora a legibilidade
- Facilita o tratamento de erros
- Código mais organizado
- Muito usado com `fetch` e APIs

O `Await` espera que outras promises sejam resolvidas

```

const asyncTimer = () =>
  new Promise((resolve, reject) =>
    setTimeout(() => {
      resolve(12345);
    }, 1000);
  );

const simpleFunc = async () => {
  const data = await asyncTimer();

  return data;
};

simpleFunc();

```

Pelo fato de ter colocado a palavra await, ele aguardou resolver a new Promise, para depois retornar data

```

// Função que simula uma operação assíncrona com Promise
const asyncTimer = () =>
  new Promise((resolve, reject) => {
    setTimeout(() => { // Executa após 1 segundo
      resolve(12345); // Resolve a Promise com o valor 12345
    }, 1000);
  });

// Função assíncrona (sempre retorna uma Promise)
const simpleFunc = async () => {

  // Aguarda a Promise do asyncTimer ser resolvida
  const data = await asyncTimer();
  console.log(data); // Exibe: 12345

  // Faz uma requisição fetch e aguarda o JSON da resposta
  const dataJSON = await fetch('/data.json')
    .then(resStream => resStream.json()); // Converte resposta para JSON

  // Retorna os dados JSON
  return dataJSON;
};

```

EVENTEMITTER

É exclusivo do [Node.js](#)

Para utilizá-la, basta instanciar ela ou estender

```
const EventEmitter = require('events');

class Users extends EventEmitter {
  userLogged(data) {
    this.emit('User logged', data);
  }
}

const users = new Users();
users.on('User logged', data => {
  console.log(data);
});

Users.userLogged({ user: 'Celso Henrique' });


```

```
const EventEmitter = require('events');

class Users extends EventEmitter {
  userLogged(data) {
    setTimeout(() => {
      this.emit('User logged', data);
    }, 2000);
  }
}

const users = new Users();
users.on('User logged', data => {
  console.log(data);
});

users.userLogged({ user: 'Celso Henrique' });
users.userLogged({ user: 'Vicente Rodrigues' });


```

TESTES, TDD E BDD

```
1 # Testes
2
3 ## Testes automatizados
4
5 * Testes unitários;
6 * Testes integrados;
7 * Testes funcionais.
8
9 ## Testes manuais e automatizados
10
11* Testes usabilidade;
12* Testes de aceitação do usuário;
13* Protótipos;
14* Testes funcionais;
15* Exemplos;
16* Alpha e beta;
17* Mais...
18
19## Ferramentas de teste
20
21* Teste de carga e performance;
22* Testes de segurança;
23* Mais...
24
25# TDD (Test Driven Development)
26
27É um dos pilares do Extreme Programming, consiste em testar e refatorar em pequenos ciclos,
28onde você escreve o teste antes do código, faz o mesmo passar e refatora o código.
29
30### Vantagens
31
32* Feedback rápido;
33* Maior segurança em alterações e novas funcionalidades;
34* Código mais limpo;
35* Produtividade.
```

Testes unitários servem para testar a maior parte do seu código

Testes integrados testam componentes juntos, como funções se comportam juntas, etc

Testes funcionais testam integração do seu sistema com outros sistemas, testam uma funcionalidade de ponta a ponta

No front-end, testa como um usuário faria

TDD:

É um dos pilares do Extreme programming, consiste em criar um teste antes mesmo de criar seu código, para definir o comportamento que aquela unidade terá, o primeiro código que você escrever tem como único objetivo fazer ele passar e voltar a refatoração, as etapas são:

Escrita do teste

Escrita do código

Refatoração

- O Ciclo se repete ao longo do percurso

O Código se torna mais limpo, aumenta a produtividade e o código se torna mais enxuto

BDD:

É uma técnica de desenvolvimento ágil que visa integrar regras de negócio com linguagem de programação, com ele descreve o comportamento dos componentes e gera a documentação dinâmica

```
39# BDD (Behavior Driven Development)
40Técnica de desenvolvimento ágil que visa integrar regras de negócio com linguagens de programação.
41
42## Pilares
43* Testes;
44* Documentação;
45* Exemplos;
46
47### Vantagens
48
49* Compartilhamento de conhecimento;
50* Documentação dinâmica;
51* Visão do todo.
```

Ele visa você a pensar o comportamento que o software deve ter antes mesmo de começar o código.

MOCHA, CHAI E SINON

MOCHA: É uma ferramenta usada para executar seus testes, no javascript, ele pode ser utilizado tanto em browser quanto em outras plataformas

Mocha é um **framework de testes para JavaScript**, muito usado em **Node.js** (e também no navegador), que serve para **criar, organizar e executar testes automatizados**.

Para que o Mocha serve?

Ele ajuda a garantir que seu código funcione corretamente, testando **funções, classes, APIs e sistemas** de forma automática.

Principais características

- Executa testes síncronos e assíncronos
 - Funciona muito bem com **Promises** e **async/await**
 - Permite organizar testes em blocos (**describe** e **it**)
 - Pode ser usado com bibliotecas de asserção como **Chai**
 - Muito usado em **back-end com Node.js**
-

Estrutura básica de um teste com Mocha

```
describe("Soma", () => {
  it("deve somar dois números corretamente", () => {
    const resultado = 2 + 3;
    if (resultado !== 5) {
      throw new Error("Erro na soma");
    }
  });
});
```

- ◆ **describe** → agrupa testes
 - ◆ **it** → define um teste específico
-

Testes assíncronos com `async/await`

```
describe("Busca de dados", () => {
  it("deve retornar dados da API", async () => {
    const dados = await buscarDados();
    if (!dados) {
      throw new Error("Dados não retornados");
    }
  });
});
```

Mocha ≠ Jest

- **Mocha** → só executa testes (mais flexível)
 - **Jest** → já vem com tudo integrado (assertions, mocks)
-

Quando usar Mocha?

-  Projetos Node.js

-  APIs REST
-  Quando você quer **mais controle** sobre o ambiente de testes
-  Muito comum em projetos profissionais

Resumo mocha IA

◆ Chai

O **Chai** é uma **biblioteca de asserções** para JavaScript.
Ela é usada junto com frameworks de teste como **Mocha** para **verificar se o resultado de um teste está correto**.

👉 Em resumo: **Chai responde “o resultado está certo ou errado?”**

Estilos de asserção do Chai

O Chai permite escrever testes de várias formas:

1 expect (mais usado)

```
const { expect } = require("chai");
```

```
expect(2 + 2).to.equal(4);
```

2 assert

```
const assert = require("chai").assert;
```

```
assert.equal(2 + 2, 4);
```

3 should

```
(2 + 2).should.equal(4);
```

Exemplo real com função

```
function soma(a, b) {  
  return a + b;  
  
}  
  
expect(soma(2, 3)).to.equal(5);
```

👉 Se o valor for diferente, o teste **falha automaticamente**.

◆ Sinon

O **Sinon** é uma biblioteca usada para **simular comportamentos** em testes.

👉 Em resumo: **Sinon “finge” funções, APIs ou serviços externos**.

Ele é muito usado para:

- ♦ **Mocks**
 - ♦ **Stubs**
 - ♦ **Spies**
-

1 Spy (espionar função)

Serve para verificar **se uma função foi chamada**.

```
const sinon = require("sinon");

const callback = sinon.spy();

callback();

sinon.assert.calledOnce(callback);
```

💡 Você não altera a função, só observa o comportamento.

2 Stub (simular retorno)

Substitui uma função real por uma falsa.

```
const sinon = require("sinon");
```

```
const user = {
  getName: () => "Renan"
};

const stub = sinon.stub(user, "getName").returns("Teste");

console.log(user.getName()); // Teste
```

💡 Muito usado para simular **banco de dados ou API**.

3 Mock (controle total)

Define **como a função deve ser chamada**.

```
const sinon = require("sinon");
```

```
const api = {  
  salvar: () => true  
};
```

```
const mock = sinon.mock(api);  
mock.expects("salvar").once();  
  
api.salvar();  
mock.verify();
```



Chai + Sinon + Mocha (combo perfeito)

- **Mocha** → executa os testes
 - **Chai** → faz as verificações
 - **Sinon** → simula comportamentos externos
-

Exemplo completo

```
describe("Serviço de usuário", () => {  
  it("deve chamar salvar uma vez", () => {  
    const repo = { salvar: () => true };  
    const spy = sinon.spy(repo, "salvar");  
  
    repo.salvar();  
  
    expect(spy.calledOnce).to.be.true;  
  });  
});
```

Resumo rápido (para prova

- **Mocha** → framework de testes
- **Chai** → biblioteca de asserções
- **Sinon** → mocks, stubs e spies

Resumo IA

Qualquer erro que acontecer, no mocha, ele vai falhar o teste, o assert serve para descrever o comportamento e evitar que o erro dispare

```
1 const assert = require('assert');
2 const Math = require('../src/math.js');
3
4 describe('Math class', function() {
5   it('Sum two numbers', function(done) {
6     const math = new Math();
7
8     math.sum(5, 5, value => {
9       assert.equal(value, 10);
10      done();
11    });
12  });
13});
```

O done garante que o teste vai aguardar a promise ser chamada, que o teste só seja concluído quando você invoca-lo

O Timeout altera o tempo máximo do mocha

```
File Edit Options Buffers Tools IM-SCRIPT-EDITOR CWD: /home/luiz/Documentos/Node.js/03-testes-unitarios
1 const assert = require('assert');
2 const Math = require('../src/math.js');
3
4 describe('Math class', function() {
5   it('Sum two numbers', done => {
6     const math = new Math();
7     this.timeout(3000);
8
9     math.sum(5, 5, value => {
10       assert.equal(value, 10);
11       done();
12     });
13   });
14});
```

O this corresponde a describe, o mocha recomenda que utilize functions para ter o controle de escopo, o mocha permite escrever testes e comportamentos que não existem ainda, sem que exista uma função para representá-los

O Mocha tem a capacidade de executar apenas um teste

O Only faz executar apenas um teste

```
File Edit Options Buffers Tools Help
1 const assert = require('assert');
2 const Math = require('../src/math.js');
3
4 describe('Math class', function() {
5   it('Sum two numbers', function(done) {
6     const math = new Math();
7     this.timeout(3000);
8
9     math.sum(5, 5, value => {
10       assert.equal(value, 10);
11       done();
12     });
13   });
14
15 it.skip('Multiply two numbers', function() {
16   const math = new Math();
17
18   assert.equal(math.multiply(5, 5), 25);
19 });
20});
```

O Mocha disponibiliza formas de executar o código evitando repetições, os chamados hooks

```
File Edit Options Buffers Tools IM-Javascript-IDE JavaScript JavaScript
1 const assert = require('assert');
2 const Math = require('../src/math.js');
3
4 let value = 0;
5
6 describe('Math class', function() {
7     //hooks
8     beforeEach(function() {
9         value = 0;
10    });
11
12     it('Sum two numbers', function(done) {const math = new Math();
13         this.timeout(3000);
14
15         value = 5;
16
17         math.sum(value, 5, value => {
18             assert.equal(value, 10);
19             done();
20         });
21     });
22
23     it('Multiply two numbers', function() {
24         const math = new Math();
25
26         assert.equal(math.multiply(value, 5), 25);
27     });
28});
```

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

```
// Importa o módulo nativo do Node.js para fazer as verificações (assertions)
```

```
const Math = require('../src/math.js');
```

```
// Importa a classe Math que será testada (arquivo do projeto)
```

```
let value = 0;
```

```
// Variável auxiliar usada nos testes
```

```
describe('Math class', function () {  
  // describe agrupa os testes relacionados à classe Math  
  
  // hooks  
  
  beforeEach(function () {  
    value = 0;  
  
    // beforeEach é executado ANTES de cada teste  
  
    // garante que a variável value sempre comece em 0  
  });  
  
  
  it('Sum two numbers', function (done) {  
    // Define um teste: "Somar dois números"  
  
    // done indica que o teste é assíncrono  
  
  
    const math = new Math();  
  
    // Cria uma nova instância da classe Math  
  
  
    this.timeout(3000);  
  
    // Define o tempo máximo do teste (3 segundos)  
  
    // útil para testes assíncronos  
  
  
    value = 5;  
  
    // Define o valor inicial
```

math.sum

O que esse código testa?

- **Soma de dois números (assíncrona)** usando callback
 - **Multiplicação de dois números (síncrona)**
 - Uso de **hooks (beforeEach)**
 - Uso de **assert** para validação
 - Uso de **done()** para testes assíncronos no Mocha
-

📌 Resumo rápido (estilo prova)

- **describe** → agrupa testes
- **it** → define um teste
- **beforeEach** → roda antes de cada teste
- **assert.equal** → valida resultado
- **done()** → finaliza teste assíncrono

O Chai é uma ferramenta que faz algo parecido com o assert

math.sum(value, 5, value => {

→ Chama o método sum da classe Math.

→ É um método assíncrono, que retorna o resultado via callback.

js

Copiar código

```
expect(value).to.equal(10);
```

- ◆ Aqui entra o Chai.

expect(...) → estilo de asserção do Chai

.to.equal(10) → verifica se o valor recebido é exatamente 10

📌 Em Chai:

equal → comparação simples (valores primitivos)

Se o valor não for 10, o teste falha automaticamente.

js

Copiar código

```
done();
```

→ Indica ao Mocha que o teste assíncrono terminou com sucesso.

js

Copiar código

```
});
```

→ Fecha o callback da soma.

- ◆ it.only

js

Copiar código

```
it.only('Multiply two numbers', function () {
```

→ it.only é um recurso do Mocha, não do Chai.

→ Ele faz com que somente esse teste seja executado, ignorando os outros.

📌 Muito usado para:

depuração

focar em um único teste

js

Copiar código

```
const math = new Math();
```

→ Cria uma instância da classe Math (não é usada diretamente aqui, mas mantém o padrão do teste).

- ◆ Objetos para teste

js

Copiar código

```
const obj = {
```

```
  name: 'Celso Henrique'
```

```
};
```

js

Copiar código

```
const obj2 = {  
    name: 'Celso Henrique'  
};
```

→ Dois objetos diferentes na memória, mas com o mesmo conteúdo.

- ◆ Chai e comparação profunda

js

Copiar código

```
expect(obj).to.deep.equal(obj2)
```

- ◆ Essa linha é o ponto mais importante sobre o Chai aqui.

O que está acontecendo?

expect(obj) → valor a ser testado

.to.deep.equal(obj2) → compara estrutura e valores internos

📌 Em Chai:

equal ✗ não funciona para objetos

deep.equal ✓ compara propriedades internas

Exemplo:

js

Copiar código

```
obj === obj2      // false  
deep.equal(obj2) // true
```

 Relação direta com Chai

Neste código você usa o estilo expect, que é o mais comum:

expect(valor).to.equal(x) → valores simples

expect(obj).to.deep.equal(outroObj) → objetos

Mensagens de erro claras e legíveis

 Resumo rápido (estilo prova)

Chai → faz as verificações (expect)

equal → valores primitivos

deep.equal → objetos e arrays

it.only → executa apenas um teste

done() → finaliza teste assíncrono

```
3   });
4 });
5
6 it('Multiply two numbers', function() {
7   const math = new Math();
8   const obj = {
9     name: 'Celso Henrique'
10    };
11
12   const obj2 = {
13     name: 'Celso Henrique'
14   };
15
16   expect(obj).to.deep.equal(obj2);
17 });
18
19 it.only('Calls res with sum and index values', function() {
20   const req = {};
21   const res = {
22     load: sinon.spy()
23   };
24   const math = new Math();
25
26   math.printSum(req, res, 5, 5);
27
28   expect(res.load.calledOnce).to.be.true;
29 });
30});
```

No código abaixo, ele criou uma função espiã

Math chama um método

```

0  const req = {};
1  const res = {
2      load: function load() {
3          console.log('Called!');
4      }
5  };
6
7  sinon.stub(res, 'load').returns('xpto');
8
9  const math = new Math();
0
1  math.printSum(req, res, 5, 5);
2
3  expect(res.load.args[0][0]).to.equal('index');
4 });
5});

```

res.restore();

Tornaria o método, um método normal

TRATAMENTO DE ERROS

O Javascript, assim que um erro é estoado, ele interrompe toda a execução do código
 seguinte, o catch é uma maneira no js de capturar um erro e evitar a perda de uma parte do
 código

```

file edit options buffers tools im-Javascript-
1try {
2  console.log(name);
3  const name = 'Celso Henrique';
4} catch (err) {
5  console.log('Error: ', err);
6}
7
8console.log('Keep going...');


```

O Erro pode ser uma classe também, basta instanciar ele, pode passar strings para a classe de erros

```
class CustomError extends Error {
  constructor({ message, data }) {
    super(message);
    this.data = data;
  }
}

try {
  const name = 'Celso Henrique';
0
1  const myError = new CustomError({
2    message: 'Custom message on custom error',
3    data: {
4      type: 'Server error'
5    }
6  });
7
8  throw myError;
9} catch (err) {
0  console.log(err);
1  console.log(err.data);
2} finally {
3  console.log('Keep going...');

4}
```

Temos um controle maior e tornar os erros mais dinâmicos e erros

```
1 class CustomError extends Error {
2   constructor({ message, data }) {
3     super(message);
4     this.data = data;
5   }
6 }
7
8 try {
9   const name = 'Celso Henrique';
10
11 const myError = new CustomError({
12   message: 'Custom message on custom error',
13   data: {
14     type: 'Server error'
15   }
16 });
17
18 throw myError;
19} catch (err) {
20   if (err.data.type === 'Server error') {
21   } else {
22   }
23} finally {
24   console.log('Keep going...');

25}
```

Debugging

Claro! Aqui vai um **resumo direto e bem organizado sobre debugging (depuração)**, no nível ideal para estudo e prova  

O que é Debugging?

Debugging (depuração) é o processo de **identificar, analisar e corrigir erros (bugs)** em um programa.

Esses erros podem ser:

-  erros de lógica

- ✗ erros de execução
 - ✗ comportamento inesperado do sistema
-



O que é um bug?

É qualquer falha que faz o software:

- produzir um resultado errado
 - travar
 - se comportar de forma diferente do esperado
-



Ferramentas de Debugging

Em JavaScript / Node.js:

- `console.log()` → inspeção rápida
- `debugger` → pausa a execução do código
- DevTools do navegador (F12)
- Debug do VS Code

Em testes:

- `it.only()` → isola um teste
 - `describe.only()` → isola um grupo
 - logs dentro do teste
-



Exemplo com debugger

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

→ O código para e permite inspecionar variáveis passo a passo.

Técnicas comuns de debugging

-  **Step by step** → executar linha por linha
 -  **Inspeção de variáveis** → ver valores em tempo real
 -  **Isolamento do problema** → testar pequenas partes
 -  **Reprodução do erro** → repetir o bug até entender a causa
-

Debugging em testes (Mocha / Chai)

- Rodar um teste por vez (`it.only`)
 - Verificar valores esperados vs retornados
 - Conferir se funções assíncronas chamam `done()`
-

Boas práticas

- Entender o erro antes de tentar corrigir
 - Ler mensagens de erro com atenção
 - Testar hipóteses pequenas
 - Corrigir **a causa**, não só o sintoma
-

Resumo final (bem curto)

- **Debugging** → processo de encontrar e corrigir erros
- Usa ferramentas para pausar, inspecionar e analisar o código
- Essencial para escrever código confiável e testável

Se quiser, posso montar um **exemplo prático com erro + depuração**, ou um **resumo ainda menor para slide** 

Aqui estão as **principais maneiras de solucionar problemas usando o depurador do navegador em JavaScript**, focando no **Chrome DevTools** (vale para Edge/Firefox também). É um guia prático para debugging 

Acessando o depurador

- Abra a página
 - Pressione **F12** ou **Ctrl + Shift + I**
 - Vá na aba **Sources**
-

II 1. Breakpoints (pontos de parada)

Permitem pausar a execução do código.

Tipos:

-  **Breakpoint de linha** → clique no número da linha
-  **Conditional breakpoint** → pausa só se a condição for verdadeira

```
if (total > 100) { // breakpoint condicional
```

► 2. Step by step (passo a passo)

Depois que o código pausa:

- **Step over (F10)** → executa a linha atual
 - **Step into (F11)** → entra na função
 - **Step out (Shift + F11)** → sai da função
 - **Resume (F8)** → continua o código
-

⌚ 3. Inspecionar variáveis

Enquanto o código está pausado:

- Veja variáveis no painel **Scope**
 - Passe o mouse sobre a variável
 - Digite o nome no **Console**
-



4. Watch (monitorar expressões)

Acompanha valores em tempo real.

```
total  
usuario.nome
```



5. Call Stack (pilha de chamadas)

Mostra **qual função chamou quem**.



Ajuda a entender:

- fluxo de execução
 - origem do erro
-



6. debugger (no código)

```
debugger;
```



Pausa automaticamente quando o DevTools está aberto.



7. Network (debug de requisições)

Ideal para **fetch / axios**:

- status da requisição
 - payload enviado
 - resposta da API
 - tempo de execução
-



8. Console (log inteligente)

```
console.log(valor);
console.table(array);
console.error(erro);
console.trace();
```



Melhor que usar só `alert()`.



9. Recarregar com cache limpo

- Clique com botão direito no refresh
 - Empty Cache and Hard Reload
-



10. Breakpoints automáticos

Em **Sources** → **Event Listener Breakpoints**:

- cliques
- submits
- timers

- promises

→ Ótimo para bugs “fantasma”.

Resumo rápido (para prova)

- Breakpoints → pausar execução
- Step by step → analisar fluxo
- Watch → monitorar valores
- Call Stack → rastrear chamadas
- Network → depurar requisições

Aqui vai um **resumo curto com erros de debugging fora do código E as soluções:**

Erros fora do código + soluções

1 ReferenceError / TypeError

Causa: variável inexistente ou uso incorreto

Solução:

- conferir nome da variável
 - verificar se o valor não é `undefined` ou `null`
 - usar `console.log()` antes
-

2 Erros de Network (404 / 500)

Causa: arquivo ou API não encontrada

Solução:

- conferir URL e caminhos
 - verificar se o servidor está rodando
 - olhar a aba **Network**
-

3 Erro de CORS

Causa: bloqueio de segurança do navegador

Solução:

- liberar CORS no back-end
 - usar proxy
 - testar em ambiente local corretamente
-

4 Erros de carregamento de script

Causa: ordem ou caminho errado

Solução:

- colocar `<script>` no final do body
 - usar `defer`
 - verificar `src`
-

5 Erros de cache

Causa: versão antiga do arquivo

Solução:

- hard reload (Ctrl + F5)

- limpar cache do navegador
-

6 Erros de ambiente

Causa: navegador ou extensão

Solução:

- testar em outro navegador
 - desativar extensões
 - atualizar o browser
-



Resumo final

- Nem todo erro está no código
- Use **Console + Network**
- Verifique caminho, servidor e cache
- Debugging é **analisar causa + aplicar solução**

O Console.timeEnd define o tempo de execução do código, em quanto tempo ele vai terminar

```
// Console  
  
// console.log('Black text');  
// console.warn('Yellow text with alert');  
// console.error('Red error text');  
  
// console.trace();  
  
/*  
0console.group('My group');  
1console.log('Info inside group');  
2console.log('More info inside group');  
3console.groupEnd('My group');  
4*/  
5  
6/*  
7console.time('Log time');  
8setTimeout(() => {  
9  console.timeEnd('Log time');  
0}, 2000);  
1*/  
2  
3//console.table(['Celso Henrique', 'Digital Innovation One']);  
4  
5console.log('%c styled log', 'color: blue; font-size: 16px');
```

O console.log tem a capacidade de ser estilizado