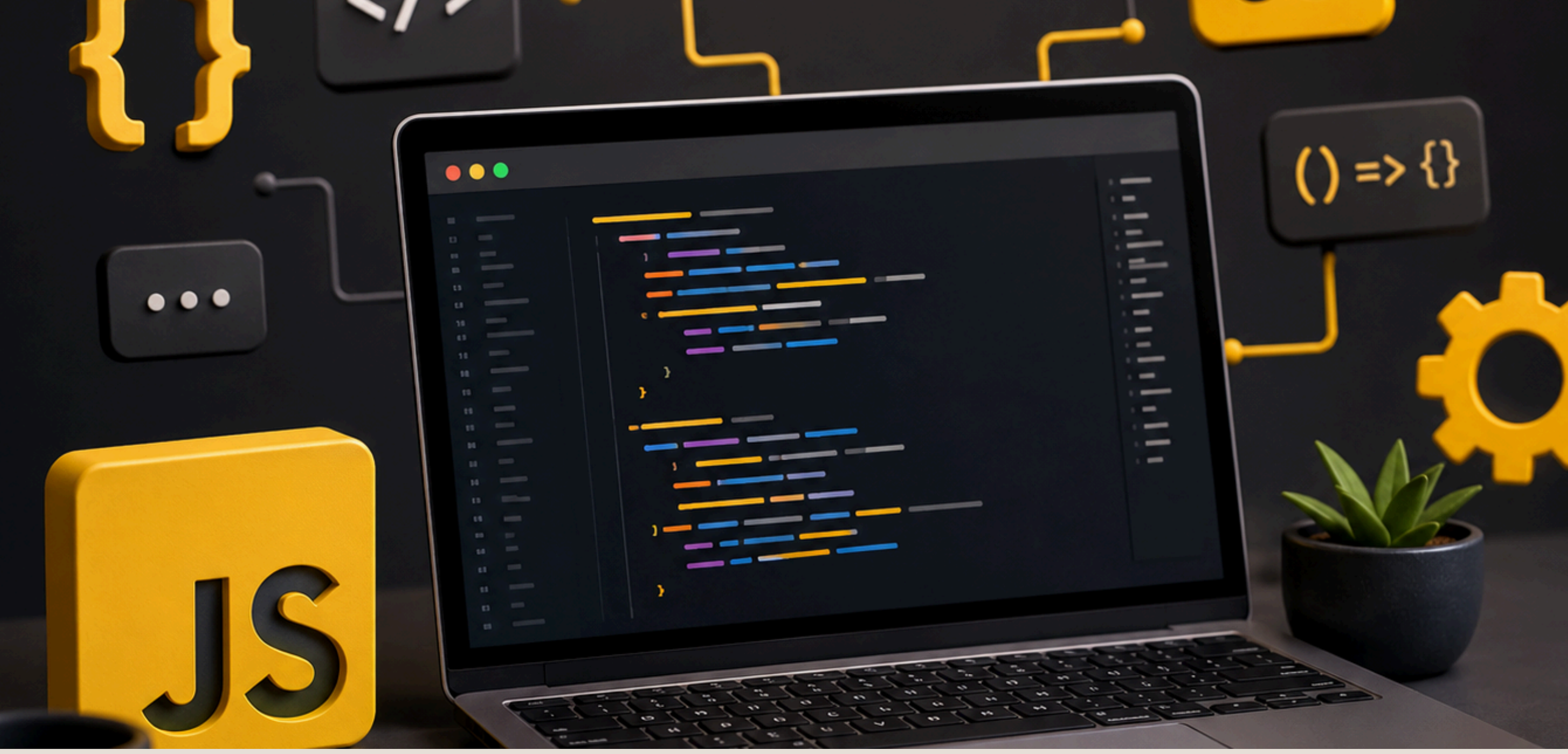
Introdução a JavaScript

Uma abordagem objetiva

Giseldo Neo e Alana Neo

2026-05-26



UMA ABORDAGEM OBJETIVA

INTRODUÇÃO A JAVASCRIPT

GISELDO NEO

ALANA NEO

Índice

Bem Vindo

- Baixe a versão epub
- Baixe a versão PDF

Este livro apresenta uma introdução prática e progressiva ao JavaScript, começando pelos conceitos essenciais para quem está dando os primeiros passos e avançando até temas usados em aplicações reais. A obra explica por que aprender JavaScript hoje, onde a linguagem é executada no navegador e no Node.js, e como configurar um ambiente de estudo produtivo. Em seguida, desenvolve os fundamentos da linguagem com variáveis, tipos primitivos, conversões, valores truthy e falsy, template strings e cuidados com números, sempre com exemplos próximos da rotina de estudo e de pequenos problemas do cotidiano. O conteúdo avança para operadores, estruturas de decisão e laços, mostrando como controlar o fluxo de execução e resolver tarefas com mais clareza. Depois, o livro aprofunda o uso de funções, parâmetros, retornos, escopo, callbacks e boas práticas, para que o código fique mais reutilizável e organizado. Na sequência, trabalha arrays, objetos, desestruturação, spread e métodos úteis de manipulação de coleções, além de estruturas lineares como fila e pilha, conectando teoria e aplicação prática. A obra também dedica espaço à programação modular, à organização de arquivos e à criação de módulos, e depois introduz conceitos de orientação a objetos e o uso combinado com módulos ES. Na parte de front-end, o livro aborda o DOM, eventos, formulários, alteração dinâmica da interface e criação de elementos, preparando o leitor para construir páginas interativas. Em seguida, explora programação assíncrona, promises, async/await, tratamento de erros e consumo de APIs com fetch, incluindo um exemplo com previsão do tempo. O fechamento trata de boas práticas, validações, testes unitários com Vitest e um projeto final que integra os conhecimentos em uma aplicação completa de tarefas, reforçando a ideia de aprender JavaScript construindo soluções reais. Por fim, os capítulos de Node.js expandem o horizonte para o ecossistema do servidor, cobrindo instalação, gerenciamento de pacotes com npm e Yarn, módulos CommonJS e ES Modules, e a criação de um servidor HTTP simples, mostrando como a linguagem pode ser usada

tanto no navegador quanto no backend.

Boa Leitura.

Giseldo Neo e Alana Neo

Parte I

Módulo 1 — Javascript básico

Capítulo 1

Introdução

Este livro foi preparado para estudantes do ensino médio técnico que estão começando em JavaScript e querem aprender de forma prática.

Ao longo dos capítulos, você vai sair do básico (variáveis e decisões) até temas mais avançados, como módulos, orientação a objetos, eventos de interface, APIs e testes.

1.1 Como estudar este livro

1. Estude um capítulo por vez, na ordem.
2. Digite os exemplos no seu computador, em vez de só ler.
3. Faça pequenas mudanças no código para observar o que acontece.
4. Resolva os exercícios antes de olhar soluções prontas.
5. Use o projeto final para consolidar os conceitos.

1.2 O que você vai construir como estudante

- programas simples de cálculo (média, desconto, orçamento);
- pequenos sistemas com listas (tarefas, alunos, produtos);
- páginas com interação (DOM e eventos);
- consumo de APIs para trazer dados reais;
- testes para verificar se funções estão corretas.

1.3 Pré-requisitos

- lógica de programação básica;

- editor de código (VS Code recomendado);
- navegador atualizado (Chrome, Edge ou Firefox);
- Node.js (recomendado a partir dos capítulos de módulos e testes).

1.4 Preparando o ambiente

1.4.1 1) Criar uma pasta de estudos

Crie uma pasta chamada `javascript-estudos` e, dentro dela, uma pasta por capítulo.

1.4.2 2) Teste no navegador

Crie um arquivo `index.html`:

```
<!DOCTYPE html>
<html lang="pt-BR">
<head>
  <meta charset="UTF-8" />
  <title>Teste JavaScript</title>
</head>
<body>
  <h1>Meu primeiro teste</h1>
  <script>
    console.log("JavaScript rodando no navegador")
  </script>
</body>
</html>
```

Abra esse arquivo no navegador e depois abra o console com `F12`.

1.4.3 3) Teste no Node.js

Crie um arquivo `app.js`:

```
console.log("JavaScript rodando com Node.js")
```

No terminal:

```
node app.js
```

1.5 Dica de rotina semanal

1. Segunda: leitura de teoria (30 min).
2. Terça: prática de exemplos (40 min).
3. Quarta: exercícios (40 min).
4. Quinta: revisão dos erros (20 min).
5. Sexta: mini desafio com código próprio (30 min).

1.6 Erros comuns de quem está começando

1. Copiar código sem testar.
2. Pular exercícios.
3. Tentar aprender tudo em um único dia.
4. Não ler mensagens de erro no console.
5. Desistir ao primeiro bug.

1.7 Objetivo final

No último capítulo, você vai montar um projeto completo conectando os conteúdos estudados. A ideia é terminar com segurança para continuar aprendendo frameworks e desenvolvimento web moderno.

Capítulo 2

O que é JavaScript?

JavaScript é uma linguagem de programação criada para dar interatividade às páginas web. Com o tempo, ela também passou a ser usada no servidor, em aplicativos mobile, desktop e automações.

2.1 Por que aprender JavaScript?

1. É a linguagem principal da web.
2. Tem grande mercado de trabalho.
3. Permite criar projetos completos (front-end e back-end).
4. É ótima para começar lógica e programação prática.

2.2 Onde o JavaScript roda?

- navegador (Chrome, Edge, Firefox);
- servidor com Node.js;
- aplicações desktop e mobile usando frameworks.

2.3 Seu primeiro programa

```
console.log("Olá, mundo!")
```

Esse comando imprime uma mensagem no console.

2.4 Rodando no navegador

Crie um arquivo `index.html`:

```
<!DOCTYPE html>
<html lang="pt-BR">
<head>
  <meta charset="UTF-8" />
  <title>Primeiro JS</title>
</head>
<body>
  <h1>Teste JavaScript</h1>
  <script>
    console.log("Executando no navegador")
  </script>
</body>
</html>
```

2.5 Rodando com Node.js

Crie `app.js`:

```
console.log("Executando com Node.js")
```

No terminal:

```
node app.js
```

2.6 JavaScript no dia a dia de um estudante

Exemplo: calcular tempo de deslocamento até a escola.

```
const distanciaKm = 4
const velocidadeKmH = 5
const tempoHoras = distanciaKm / velocidadeKmH

console.log(`Tempo estimado: ${tempoHoras} hora(s)`)
```

Exemplo: simular gasto diário com lanche.

```
const salgado = 8
const suco = 6
const total = salgado + suco

console.log(`Gasto de hoje: R$ ${total}`)
```

2.7 Comentários no código

```
// Comentário de uma linha

/*
  Comentário
  de múltiplas linhas
*/
```

Use comentários para explicar decisões importantes, não para repetir o óbvio.

2.8 Boas práticas iniciais

1. Nomeie variáveis com clareza.
2. Mantenha o código identado.
3. Teste em pequenos passos.
4. Leia mensagens de erro com atenção.

2.9 Exercícios

1. Mostre seu nome, turma e cidade no console.
2. Crie um programa que calcule o custo de 5 passagens de ônibus.
3. Crie um HTML com botão e, no `<script>`, mostre no console `"Página carregada"`.
4. Desafio: simule o custo semanal de transporte para a escola.

Capítulo 3

Variáveis e Tipos

Variável é um espaço na memória usado para armazenar dados durante a execução do programa.

Em JavaScript, variáveis servem para guardar valores que podem ser usados, combinados e transformados ao longo do programa. Entender bem como declarar e manipular variáveis evita muitos erros comuns em código iniciante.

3.1 `let`, `const` e `var`

- `let`: valor pode mudar.
- `const`: valor não pode ser reatribuído.
- `var`: forma antiga, evite em novos projetos.

Quando possível, prefira `const`. Isso ajuda a deixar claro o que nao deve mudar. Use `let` quando a variavel realmente precisar ser atualizada.

```
let idade = 16
const escola = "Escola Técnica"

idade = 17

console.log(idade)
console.log(escola)
```

3.1.1 Escopo (onde a variavel existe)

- `let` e `const` tem escopo de bloco: so existem dentro de chaves `{ }`.
- `var` tem escopo de funcao: pode vazar para fora de blocos e causar bugs.

```
if (true) {
    let dentro = "ok"
    var fora = "opa"
}

// console.log(dentro) // erro: dentro nao existe aqui
console.log(fora) // funciona, mas e perigoso
```

3.1.2 Reatribuicao vs. mutacao

`const` impede reatribuicao, mas nao impede mutar objetos e arrays.

```
const aluno = { nome: "Joao", nota: 8 }
aluno.nota = 9 // ok

// aluno = { nome: "Maria" } // erro
```

3.2 Regras de nome de variáveis

1. Comece com letra, _ ou $.
2. Não use espaços.
3. Evite nomes genéricos como `x`, `y`, `z` em projetos reais.
4. Use nomes claros: `notaFinal`, `valorTotal`, `nomeAluno`.

Boas praticas:

- Use `camelCase` em nomes de variaveis e funcoes.
- Evite abreviacoes confusas.
- Prefira nomes que indiquem o papel do dado: `quantidadeItens`, `estaAtivo`.

3.3 Tipos primitivos

- `string` (texto)
- `number` (número)
- `boolean` (`true` ou `false`)
- `null`
- `undefined`
- `bigint`
- `symbol`

Outros tipos importantes:

- `object`: estruturas como arrays, objetos e datas.
- `function`: funcoes tambem sao valores.

```
const nome = "Ana"
const nota = 8.5
const aprovado = true

console.log(typeof nome)     // string
console.log(typeof nota)     // number
console.log(typeof aprovado) // boolean
```

3.3.1 `null` vs `undefined`

- `undefined`: valor nao definido (variavel declarada mas sem valor).
- `null`: valor vazio definido pelo programador.

```
let resultado
const vazio = null

console.log(resultado) // undefined
console.log(vazio) // null
```

Curiosidade: `typeof null` retorna `object` por um detalhe historico.

3.4 Conversão de tipos

Quando um valor de texto precisa virar numero, use `Number`. Para conversoes mais tolerantes, use `parseInt` ou `parseFloat`.

```
const textoNumero = "42"
const numero = Number(textoNumero)

console.log(numero + 8) // 50
```

```
console.log(parseInt("12px", 10)) // 12
console.log(parseFloat("10.5em")) // 10.5
```

Outras conversões úteis:

```
console.log(String(150))      // "150"
console.log(Boolean(1))       // true
console.log(Boolean(0))       // false
```

3.4.1 Valores truthy e falsy

Em condicoes, alguns valores sao considerados `false`:

- `false`, `0`, `""`, `null`, `undefined`, `NaN`

Todo o resto e `true`.

3.5 `NaN` e validação de número

`NaN` significa “Not a Number”, quando uma operação numérica falha.

```
const entrada = "abc"
const valor = Number(entrada)

console.log(valor) // NaN
console.log(Number.isNaN(valor)) // true
```

Use `Number.isNaN` para validar numeros depois de conversoes. `isNaN` global pode confundir porque converte tipos antes de testar.

3.6 Template strings

Template string usa crase para montar frases com variáveis.

```
const aluno = "Carlos"
const media = 7.8
console.log(`Aluno: ${aluno} | Média: ${media}`)
```

Template strings tambem permitem quebrar linhas com facilidade:

```
const aviso = `Aluno: ${aluno}
Media: ${media}`
console.log(aviso)
```

3.7 Precisao de numeros

JavaScript usa ponto flutuante, entao algumas somas nao ficam exatas.

```
console.log(0.1 + 0.2) // 0.30000000000000004
```

Em valores de dinheiro, prefira trabalhar em centavos (inteiros) e formatar no final.

3.8 Exemplo prático: orçamento de passeio

```
const transporte = 50
const alimentacao = 45
const ingresso = 30

const total = transporte + alimentacao + ingresso
console.log(`Custo total do passeio: R$ ${total}`)
```

3.9 Exemplo prático: situação do estudante

```
const faltas = 12
const limite = 15
const podeFaltarMais = faltas < limite

console.log(`Pode faltar mais? ${podeFaltarMais}`)
```

3.10 Exercícios

1. Declare variáveis para nome, idade e cidade e exiba no console.
2. Converta a string `"150"` para número e some com 25.
3. Crie uma mensagem com template string informando produto e preço.
4. Crie um programa que receba preço e quantidade e exiba o total.
5. Desafio: leia uma string numérica e valide se a conversão resultou em número válido.

Capítulo 4

Operadores e Estruturas de Controle

Neste capítulo, você vai aprender a tomar decisões no código e repetir tarefas com laços.

4.1 Operadores aritméticos

```
const a = 10
const b = 3

console.log(a + b) // soma
console.log(a - b) // subtração
console.log(a * b) // multiplicação
console.log(a / b) // divisão
console.log(a % b) // resto
```

4.2 Operadores relacionais

```
console.log(8 > 5)   // true
console.log(8 >= 8)  // true
console.log(8 < 5)   // false
console.log(8 === 8) // true
console.log(8 !== 7) // true
```

4.3 Operadores lógicos

- `&&` (E): tudo precisa ser verdadeiro.
- `||` (OU): ao menos uma condição verdadeira.
- `!` (NÃO): inverte.

```
const nota = 7
const frequencia = 80

const aprovado = nota >= 6 && frequencia >= 75
console.log(aprovado)
```

4.4 `if`, `else if`, `else`

```
const temperatura = 18

if (temperatura < 15) {
  console.log("Frio")
} else if (temperatura <= 25) {
  console.log("Agradável")
} else {
  console.log("Quente")
}
```

4.5 Exemplo escolar: status do aluno

```
const media = 6.2

if (media >= 7) {
  console.log("Aprovado")
} else if (media >= 5) {
  console.log("Recuperação")
} else {
  console.log("Reprovado")
}
```

4.6 `switch`

```
const dia = 3

switch (dia) {
  case 1:
    console.log("Domingo")
    break
  case 2:
    console.log("Segunda")
    break
  case 3:
    console.log("Terça")
    break
  default:
    console.log("Dia inválido")
}
```

4.7 Operador ternário

```
const idade = 17
const status = idade >= 18 ? "Maior de idade" : "Menor de idade"
console.log(status)
```

4.8 Laço `for`

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

4.9 Laço `while`

```
let contador = 1

while (contador <= 3) {
```

```
  console.log(contador)
  contador++
}
```

4.10 `break` e `continue`

```
for (let i = 1; i <= 10; i++) {
  if (i === 4) continue
  if (i === 8) break
  console.log(i)
}
```

4.11 Exemplo prático: orçamento até limite

```
const gastos = [20, 35, 18, 40, 12]
let total = 0

for (let i = 0; i < gastos.length; i++) {
  if (total + gastos[i] > 90) {
    console.log("Limite de orçamento atingido")
    break
  }

  total += gastos[i]
}

console.log(`Total acumulado: R$ ${total}`)
```

4.12 Exercícios

1. Classifique uma nota em `A`, `B`, `C` ou `D`.
2. Mostre os números pares de 2 a 20.
3. Calcule a soma de 1 até 100 com laço.
4. Use `switch` para exibir o nome do mês com base no número.
5. Desafio: simule uma compra que para quando atingir um limite de gasto.

Capítulo 5

Funções

Funções são blocos de código reutilizáveis.
Elas recebem dados de entrada, executam uma tarefa e podem devolver um resultado.

5.1 Por que funções são importantes?

1. Evitam repetição de código.
2. Organizam melhor o programa.
3. Facilitam testes e manutenção.
4. Permitem dividir problemas grandes em partes menores.

5.2 Declaração de função

```
function saudacao(nome) {
  return `Olá, ${nome}!`
}

console.log(saudacao("Marina"))
```

5.3 Parâmetros e retorno

```
function media(n1, n2, n3) {
  return (n1 + n2 + n3) / 3
}
```

```
const resultado = media(7, 8, 9)
console.log(resultado)
```

5.3.1 Exemplo do cotidiano estudantil

```
function mediaPresenca(aula1, aula2, aula3, aula4) {
  return (aula1 + aula2 + aula3 + aula4) / 4
}

const presenca = mediaPresenca(100, 100, 75, 100)
console.log(`Média de presença: ${presenca}%`)
```

5.4 Parâmetros com valor padrão

```
function calcularIngresso(valor, taxaServico = 5) {
  return valor + taxaServico
}

console.log(calcularIngresso(30))    // 35
console.log(calcularIngresso(30, 8)) // 38
```

5.5 Função anônima em variável

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

console.log(dobro(9))
```

5.6 Arrow functions

```
const quadrado = (n) => n * n
console.log(quadrado(6))
```

Exemplo com turismo:

```
const calcularGastoPasseio = (transporte, alimentacao, ingresso) =>
  transporte + alimentacao + ingresso

const gastoTotal = calcularGastoPasseio(80, 55, 30)
console.log(`Gasto total no passeio: R$ ${gastoTotal}`)
```

5.7 Escopo

- variáveis globais: acessíveis fora de funções;
- variáveis locais: só existem dentro da função.

```
let curso = "Informática"

function mostrarCurso() {
  let modulo = 2
  console.log(curso)
  console.log(modulo)
}

mostrarCurso()
```

5.8 Funções puras e efeitos colaterais

Função pura: para a mesma entrada, sempre devolve a mesma saída e não altera estado externo.

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

Função com efeito colateral:

```
let contador = 0

function incrementarContador() {
  contador++
}
```

5.9 Função que calcula x função que exibe

```
function calcularDesconto(valorCompra, porcentagem) {
  return valorCompra - (valorCompra * porcentagem) / 100
}

function mostrarPrecoFinal(valorOriginal, desconto) {
  const precoFinal = calcularDesconto(valorOriginal, desconto)
  console.log(`De R$ ${valorOriginal} por R$ ${precoFinal}`)
}

mostrarPrecoFinal(120, 10)
```

5.10 Callback (função passada para outra função)

```
function processarAluno(nome, callback) {
  const mensagem = `Aluno: ${nome}`
  callback(mensagem)
}

processarAluno("Bruna", (texto) => {
  console.log(texto.toUpperCase())
})
```

5.11 Exemplo integrado: roteiro de viagem

```
function somarDespesas(listaDeValores) {
  let total = 0

  for (const valor of listaDeValores) {
    total += valor
  }

  return total
}
```

```
function calcularCustoPorPessoa(total, quantidadePessoas) {
  return total / quantidadePessoas
}

const despesas = [90, 60, 40, 30]
const totalViagem = somarDespesas(despesas)
const custoIndividual = calcularCustoPorPessoa(totalViagem, 4)

console.log(`Total da viagem: R$ ${totalViagem}`)
console.log(`Custo por pessoa: R$ ${custoIndividual}`)
```

5.12 Boas práticas para funções

1. Use nomes claros (`calcularMedia`, `buscarAluno`).
2. Faça funções curtas e com um objetivo.
3. Prefira `return` para reaproveitamento.
4. Evite misturar cálculo com interface.
5. Teste cenários comuns e cenários de erro.

5.13 Exercícios

1. Crie `tempoCaminhada(distanciaKm, velocidadeKmH)`.
2. Crie `converterRealParaGuarani(real, cotacao)`.
3. Crie `custoViagemBonito(transporte, hospedagem, alimentacao, passeios)`.
4. Crie `mediaNotas(notas)` para um array de notas.
5. Desafio: `situacaoAluno(media, frequencia)` com regras definidas por você.

Capítulo 6

Arrays e Objetos

Arrays e objetos são estruturas fundamentais para organizar dados.

- `array`: lista ordenada de valores.
- `objeto`: coleção de pares `chave: valor`.

6.1 Arrays

```
const notas = [7.5, 8.0, 6.5, 9.0]

console.log(notas[0])      // primeiro item
console.log(notas.length)  // quantidade de itens
```

6.2 Métodos mais usados em arrays

```
const alunos = ["Ana", "Bruno"]

alunos.push("Carla")   // adiciona no fim
const removido = alunos.pop() // remove do fim

console.log(removido)
console.log(alunos)
```

Outros úteis:

```
const fila = ["A", "B", "C"]
fila.shift()      // remove do início
fila.unshift("X") // adiciona no início
console.log(fila)
```

6.3 Percorrendo arrays

6.3.1 `for` tradicional

```
const precos = [12, 30, 18]
let total = 0

for (let i = 0; i < precos.length; i++) {
  total += precos[i]
}

console.log(`Total: R$ ${total}`)
```

6.3.2 `for...of`

```
const cidades = ["Ponta Porã", "Dourados", "Bonito"]

for (const cidade of cidades) {
  console.log(cidade)
}
```

6.4 `map`, `filter`, `reduce`

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

const dobrados = numeros.map((n) => n * 2)
const pares = numeros.filter((n) => n % 2 === 0)
const soma = numeros.reduce((acc, n) => acc + n, 0)

console.log(dobrados)
```

```
console.log(pares)
console.log(soma)
```

6.5 Objetos

```
const aluno = {
  nome: "João",
  turma: "2A",
  ativo: true,
  exibir() {
    return `${this.nome} - ${this.turma}`
  }
}

console.log(aluno.exibir())
```

6.6 Acessando e alterando propriedades

```
const produto = {
  nome: "Caderno",
  preco: 25
}

console.log(produto.nome)
produto.preco = 22
produto.estoque = 15
console.log(produto)
```

6.7 Array de objetos

Muito comum em sistemas:

```
const passeios = [
  { local: "Bonito", preco: 120 },
  { local: "Pantanal", preco: 200 },
  { local: "Campo Grande", preco: 80 }
```

```
]

const baratos = passeios.filter((p) => p.preco <= 120)
console.log(baratos)
```

6.8 Desestruturação

```
const estudante = {
  nome: "Lívia",
  cidade: "Ponta Porã",
  curso: "Informática"
}

const { nome, cidade } = estudante
console.log(nome, cidade)
```

6.9 Spread em arrays e objetos

```
const base = [1, 2, 3]
const copia = [...base, 4]

const alunoBase = { nome: "Rafa", turma: "1B" }
const alunoAtualizado = { ...alunoBase, turma: "2B" }

console.log(copia)
console.log(alunoAtualizado)
```

6.10 Exercícios

1. Crie um array com 10 números e exiba apenas os ímpares.
2. Dado um array de preços, calcule o total com `reduce`.
3. Crie um objeto `produto` com nome, preço e estoque.
4. Crie um array de objetos `alunos` com `nome` e `nota`; exiba só os aprovados.
5. Desafio: monte uma lista de passeios com preço e exiba o mais barato.

Capítulo 7

Vetores, Matrizes, Fila e Pilha

Este capítulo aprofunda estruturas lineares muito usadas em programação.

7.1 Vetores (arrays unidimensionais)

Vetores armazenam elementos em sequência e cada posição tem um índice.

7.1.1 Exemplo 1: notas de uma turma

```
const notas = [7.0, 8.5, 6.0, 9.0]

console.log(`Primeira nota: ${notas[0]}`)
console.log(`Quantidade de notas: ${notas.length}`)
```

7.1.2 Exemplo 2: soma e média

```
const valores = [10, 20, 30, 40]
let soma = 0

for (let i = 0; i < valores.length; i++) {
  soma += valores[i]
}

const media = soma / valores.length
console.log(`Soma: ${soma} | Média: ${media}`)
```

7.1.3 Exemplo 3: filtro de valores

```
const numeros = [3, 8, 11, 14, 19, 20]
const pares = numeros.filter((n) => n % 2 === 0)

console.log(pares)
```

7.2 Matrizes (arrays bidimensionais)

Uma matriz é um array de arrays, com linhas e colunas.

7.2.1 Exemplo 1: acesso por linha e coluna

```
const matriz = [
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9]
]

console.log(matriz[1][2]) // 6
```

7.2.2 Exemplo 2: percorrer todos os elementos

```
const tabela = [
  [10, 20],
  [30, 40],
  [50, 60]
]

for (let linha = 0; linha < tabela.length; linha++) {
  for (let coluna = 0; coluna < tabela[linha].length; coluna++) {
    console.log(`Posição [${linha}][${coluna}] = ${tabela[linha][coluna]}`)
  }
}
```

7.2.3 Exemplo 3: diagonal principal

```
const m = [
  [2, 1, 0],
  [4, 3, 8],
  [9, 7, 6]
]

let diagonal = 0

for (let i = 0; i < m.length; i++) {
  diagonal += m[i][i]
}

console.log(`Soma diagonal principal: ${diagonal}`)
```

7.3 Fila (FIFO: First In, First Out)

Em fila, o primeiro a entrar é o primeiro a sair.

7.3.1 Exemplo 1: atendimento

```
const fila = []

fila.push("Aluno 1")
fila.push("Aluno 2")
fila.push("Aluno 3")

const atendido = fila.shift()
console.log(`Atendido: ${atendido}`)
console.log(fila)
```

7.3.2 Exemplo 2: próximo da fila

```
const filaSenhas = ["A01", "A02", "A03"]
console.log(`Próximo: ${filaSenhas[0]}`)
```

7.3.3 Exemplo 3: fila com validação

```
const filaLaboratorio = []

function chamarAluno() {
  if (filaLaboratorio.length === 0) {
    return "Fila vazia"
  }

  return filaLaboratorio.shift()
}

console.log(chamarAluno())
filaLaboratorio.push("Carlos")
console.log(chamarAluno())
```

7.4 Pilha (LIFO: Last In, First Out)

Em pilha, o último a entrar é o primeiro a sair.

7.4.1 Exemplo 1: empilhar e desempilhar

```
const pilha = []

pilha.push("Prato 1")
pilha.push("Prato 2")
pilha.push("Prato 3")

const removido = pilha.pop()
console.log(`Removido: ${removido}`)
console.log(pilha)
```

7.4.2 Exemplo 2: topo sem remover

```
const historico = ["home", "produtos", "checkout"]
const topo = historico[historico.length - 1]
```

```
console.log(`Página atual: ${topo}`)
```

7.4.3 Exemplo 3: desfazer ação

```
const acoes = []

function executar(acao) {
  acoes.push(acao)
}

function desfazer() {
  if (acoes.length === 0) return "Nada para desfazer"
  return `Desfez: ${acoes.pop()}`
}

executar("Digitou texto")
executar("Apagou linha")
console.log(desfazer())
console.log(desfazer())
console.log(desfazer())
```

7.5 Aplicação prática combinando as estruturas

```
const filaAtendimento = ["Aluno A", "Aluno B"]
const historicoChamadas = []
const temposEspera = [4, 6, 3, 5]

const chamado = filaAtendimento.shift()
historicoChamadas.push(chamado)

const mediaEspera =
  temposEspera.reduce((acc, t) => acc + t, 0) / temposEspera.length

console.log(`Chamado agora: ${chamado}`)
console.log(`Média de espera: ${mediaEspera} min`)
```

7.6 Exercícios

1. Crie um vetor de 6 números e calcule a média.
2. Crie uma matriz `3x2` e exiba só os valores maiores que `25`.
3. Simule uma fila com 4 pessoas e atenda 2.
4. Simule uma pilha de páginas e implemente “voltar”.
5. Desafio: combine fila e pilha em um sistema simples de atendimento escolar.

Capítulo 8

Funções, Blocos, Procedimentos, Módulos e Parâmetros

Este capítulo organiza conceitos importantes para escrever código legível, reutilizável e escalável.

8.1 Blocos

Bloco é o trecho entre chaves `{}`. Ele delimita escopo.

```
let turma = "2A"

{
  let turma = "3A"
  console.log(`Dentro do bloco: ${turma}`)
}

console.log(`Fora do bloco: ${turma}`)
```

8.2 Procedimentos

Procedimento executa uma ação e normalmente não depende de retorno.

```
function mostrarCabecalho() {
  console.log("=== Sistema Acadêmico ===")
}
```

```
mostrarCabecalho()
```

Com parâmetro:

```
function saudarAluno(nome) {
  console.log(`Bem-vindo(a), ${nome}`)
}

saudarAluno("Marina")
```

8.3 Funções com retorno

```
function calcularMedia(n1, n2, n3) {
  return (n1 + n2 + n3) / 3
}

console.log(calcularMedia(7, 8, 9))
```

8.4 Passagem de parâmetros

8.4.1 Primitivo: cópia por valor

```
let quantidade = 5

function alterarNumero(n) {
  n = 99
}

alterarNumero(quantidade)
console.log(quantidade) // 5
```

8.4.2 Objeto: referência compartilhada

```
const aluno = { nome: "Ana", nota: 8 }

function atualizarNota(dados) {
  dados.nota = 10
```

```
}

atualizarNota(aluno)
console.log(aluno.nota) // 10
```

8.4.3 Evitando mutação com spread

```
const tarefa = { titulo: "Estudar", concluida: false }

function concluirSemMutar(item) {
  return { ...item, concluida: true }
}

const tarefaConcluida = concluirSemMutar(tarefa)
console.log(tarefa)
console.log(tarefaConcluida)
```

8.5 Modularização

Quando o projeto cresce, separar por arquivos ajuda muito.

8.5.1 Export nomeado

matematica.js

```
export function somar(a, b) {
  return a + b
}
```

main.js

```
import { somar } from "./matematica.js"

console.log(somar(5, 7))
```

8.5.2 Múltiplos exports

boletim.js

```
export function media(a, b) {
  return (a + b) / 2
}

export function aprovado(mediaFinal) {
  return mediaFinal >= 6
}
```

app.js

```
import { media, aprovado } from "./boletim.js"

const m = media(7, 5)
console.log(m)
console.log(aprovado(m))
```

8.5.3 Export default

mensagem.js

```
export default function gerarMensagem(nome) {
  return `Olá, ${nome}!`
}
```

principal.js

```
import gerarMensagem from "./mensagem.js"

console.log(gerarMensagem("Carlos"))
```

8.6 Estrutura sugerida para projetos pequenos

```
projeto/
  src/
    calculos.js
    validacoes.js
    app.js
```

8.7 Exemplo integrador

`calculos.js`

```
export function totalDespesas(lista) {
  return lista.reduce((acc, item) => acc + item, 0)
}
```

`app.js`

```
import { totalDespesas } from "./calculos.js"

const gastos = [45, 30, 20, 10]
console.log(`Total: R$ ${totalDespesas(gastos)}`)
```

8.8 Exercícios

1. Crie um módulo `conversoes.js` com metro para centímetro.
2. Crie um módulo `boletim.js` com funções de média e status.
3. Crie um `export default` para formatar preço em reais.
4. Desafio: monte um mini projeto com `app.js`, `dados.js` e `calculos.js`.

Capítulo 9

POO e Módulos

Programação Orientada a Objetos (POO) ajuda a modelar o mundo real em código usando classes e objetos.

9.1 Conceitos básicos de POO

- classe: molde para criar objetos;
- objeto: instância da classe;
- atributo: característica do objeto;
- método: comportamento do objeto.

9.2 Criando uma classe

```
class Pessoa {
  constructor(nome, idade) {
    this.nome = nome
    this.idade = idade
  }

  apresentar() {
    return `Meu nome é ${this.nome} e tenho ${this.idade} anos.`
  }
}

const p1 = new Pessoa("Lia", 17)
```

```
console.log(p1.apresentar())
```

9.3 Encapsulando regras com métodos

```
class ContaEstudante {
  constructor(saldoInicial = 0) {
    this.saldo = saldoInicial
  }

  depositar(valor) {
    if (valor <= 0) return
    this.saldo += valor
  }

  sacar(valor) {
    if (valor > this.saldo) return "Saldo insuficiente"
    this.saldo -= valor
    return "Saque realizado"
  }
}

const conta = new ContaEstudante(100)
conta.depositar(50)
console.log(conta.sacar(30))
console.log(conta.saldo)
```

9.4 Herança

Herança permite reutilizar código de uma classe base.

```
class Pessoa {
  constructor(nome, idade) {
    this.nome = nome
    this.idade = idade
  }
}
```

```
class Aluno extends Pessoa {
  constructor(nome, idade, curso) {
    super(nome, idade)
    this.curso = curso
  }

  dados() {
    return `${this.nome} - ${this.curso}`
  }
}

const aluno = new Aluno("Rafa", 16, "Informática")
console.log(aluno.dados())
```

9.5 Polimorfismo (ideia prática)

Diferentes classes podem ter métodos com o mesmo nome e comportamentos específicos.

```
class Transporte {
  calcularTempo() {
    return "Tempo padrão"
  }
}

class Onibus extends Transporte {
  calcularTempo() {
    return "Tempo estimado: 25 min"
  }
}

class Bicicleta extends Transporte {
  calcularTempo() {
    return "Tempo estimado: 18 min"
  }
}

const opcoes = [new Onibus(), new Bicicleta()]
```

```
opcoes.forEach((item) => console.log(item.calcularTempo()))
```

9.6 Relembrando módulos ES

9.6.1 Export nomeado

matematica.js

```
export function somar(a, b) {
  return a + b
}
```

main.js

```
import { somar } from "./matematica.js"
console.log(somar(5, 7))
```

9.6.2 Export default

formatador.js

```
export default function formatarMoeda(valor) {
  return `R$ ${valor.toFixed(2)}`
}
```

app.js

```
import formatarMoeda from "./formatador.js"
console.log(formatarMoeda(35.5))
```

9.7 Exemplo integrado: classe + módulo

produto.js

```
export class Produto {
  constructor(nome, preco) {
    this.nome = nome
    this.preco = preco
  }
```

```
  aplicarDesconto(percentual) {
    this.preco -= this.preco * (percentual / 100)
  }
}
```

index.js

```
import { Produto } from "./produto.js"

const caderno = new Produto("Caderno", 30)
caderno.aplicarDesconto(10)
console.log(caderno)
```

9.8 Exercícios

1. Crie uma classe `Produto` com nome, preço e método de desconto.
2. Crie `Funcionario` e `Professor` usando herança.
3. Crie uma classe `RoteiroTuristico` com método para calcular custo total.
4. Separe funções em módulos e importe no arquivo principal.
5. Desafio: combine uma classe com persistência em array de objetos.

Capítulo 10

DOM e Eventos

DOM (Document Object Model) é a estrutura da página HTML que o JavaScript consegue manipular.

Com DOM e eventos, você cria interfaces interativas.

10.1 Selecionando elementos

```
<h2 id="titulo">Loja Técnica</h2>
<button id="botao">Clique</button>
```

```
const titulo = document.getElementById("titulo")
const botao = document.getElementById("botao")

titulo.textContent = "Sistema Escolar"
```

Outros seletores:

```
const primeiroCard = document.querySelector(".card")
const todosOsCards = document.querySelectorAll(".card")
```

10.2 Alterando estilos e classes

```
const card = document.querySelector(".card")

card.classList.add("ativo")
```

```
card.classList.toggle("destaque")
card.style.border = "2px solid #1f6feb"
```

10.3 Eventos

```
botao.addEventListener("click", () => {
  alert("Botão clicado!")
})
```

Eventos comuns:

- click
- input
- change
- submit
- keydown

10.4 Formulários

```
<form id="formAluno">
  <input id="nome" placeholder="Nome" />
  <button type="submit">Salvar</button>
</form>
```

```
const form = document.getElementById("formAluno")

form.addEventListener("submit", (event) => {
  event.preventDefault()

  const nome = document.getElementById("nome").value.trim()

  if (!nome) {
    alert("Informe o nome")
    return
  }

  console.log(`Aluno cadastrado: ${nome}`)
```

```
})
```

10.5 Criando elementos dinamicamente

```
<ul id="lista"></ul>
```

```
const lista = document.getElementById("lista")
const itens = ["Caderno", "Caneta", "Lápis"]

for (const item of itens) {
  const li = document.createElement("li")
  li.textContent = item
  lista.appendChild(li)
}
```

10.6 Delegação de eventos

Útil quando itens são criados dinamicamente.

```
lista.addEventListener("click", (event) => {
  if (event.target.tagName === "LI") {
    event.target.classList.toggle("feito")
  }
})
```

10.7 Exemplo integrado: contador

```
<p id="valor">0</p>
<button id="menos">-</button>
<button id="mais">+</button>
```

```
const campo = document.getElementById("valor")
const botaoMenos = document.getElementById("menos")
const botaoMais = document.getElementById("mais")

let contador = 0
```

```
function renderizar() {
  campo.textContent = contador
}

botaoMenos.addEventListener("click", () => {
  contador--
  renderizar()
})

botaoMais.addEventListener("click", () => {
  contador++
  renderizar()
})
```

10.8 Exercícios

1. Crie uma página com botão que altera a cor de fundo.
2. Faça um contador com botões de aumentar e diminuir.
3. Valide um formulário para impedir envio com campo vazio.
4. Crie uma lista de tarefas com botão de remover item.
5. Desafio: implemente filtro de tarefas (todas, pendentes, concluídas).

Capítulo 11

Assincronismo e APIs

Nem toda tarefa no JavaScript termina imediatamente.
Chamamos isso de comportamento assíncrono.

Exemplos:

- requisições de rede;
- leitura de arquivos;
- timers (`setTimeout`, `setInterval`).

11.1 Entendendo a ordem de execução

```
console.log("Início")

setTimeout(() => {
  console.log("Executou depois de 2 segundos")
}, 2000)

console.log("Fim")
```

Saída esperada:

1. `Início`
2. `Fim`
3. `Executou depois de 2 segundos`

11.2 Promises

Promise representa um resultado futuro: pendente, resolvido ou rejeitado.

```
function buscarDados() {
  return new Promise((resolve) => {
    setTimeout(() => resolve("Dados carregados"), 1000)
  })
}

buscarDados().then((mensagem) => console.log(mensagem))
```

Com erro:

```
function validarIdade(idade) {
  return new Promise((resolve, reject) => {
    if (idade >= 18) resolve("Acesso liberado")
    else reject(new Error("Acesso negado"))
  })
}
```

11.3 `async/await`

`async` e `await` deixam o código assíncrono mais legível.

```
async function executar() {
  const resposta = await buscarDados()
  console.log(resposta)
}

executar()
```

11.4 Tratamento de erros com `try/catch`

```
async function testarIdade() {
  try {
    const msg = await validarIdade(16)
    console.log(msg)
```

```
  } catch (erro) {
    console.error("Erro:", erro.message)
  }
}

testarIdade()
```

11.5 Consumindo API com `fetch`

```
async function carregarUsuarios() {
  const response = await fetch("https://jsonplaceholder.typicode.com/users")
  const usuarios = await response.json()
  console.log(usuarios)
}

carregarUsuarios()
```

11.6 Verificando status HTTP

```
async function buscarPost(id) {
  const response = await fetch(`https://jsonplaceholder.typicode.com/posts/${id}`)

  if (!response.ok) {
    throw new Error(`Falha HTTP: ${response.status}`)
  }

  return response.json()
}
```

11.7 Exemplo prático: previsão do tempo

Fluxo comum:

1. usuário escolhe cidade;
2. aplicação chama API;
3. mostra temperatura e condição do clima;

4. em erro, exibe mensagem amigável.

```
async function carregarClima(url) {
  try {
    const response = await fetch(url)

    if (!response.ok) {
      throw new Error("Não foi possível buscar o clima")
    }

    const dados = await response.json()
    console.log(dados)
  } catch (erro) {
    console.error("Erro ao carregar clima:", erro.message)
  }
}
```

11.8 Exercícios

1. Faça uma função assíncrona que aguarde 3 segundos e retorne mensagem.
2. Consuma uma API pública e mostre apenas os nomes retornados.
3. Trate erros com `try/catch` no `fetch`.
4. Crie função que busque um post por ID e valide status da resposta.
5. Desafio: montar tela com busca e carregamento (`loading`) para uma API.

Capítulo 12

Testes e Boas Práticas

Escrever código que funciona é importante.
Escrever código confiável e fácil de manter é essencial.

12.1 Boas práticas essenciais

1. Use nomes claros para variáveis e funções.
2. Crie funções pequenas com uma responsabilidade.
3. Evite duplicação de código.
4. Trate erros de forma explícita.
5. Comente apenas quando necessário.

12.2 Exemplo: função clara e validada

```
function calcularMedia(notas) {
  if (!Array.isArray(notas) || notas.length === 0) {
    throw new Error("Informe um array de notas")
  }

  const soma = notas.reduce((acc, n) => acc + n, 0)
  return soma / notas.length
}
```

12.3 Tratamento de erros

```
function dividir(a, b) {
  if (b === 0) {
    throw new Error("Divisão por zero não permitida")
  }

  return a / b
}

try {
  console.log(dividir(10, 0))
} catch (erro) {
  console.error("Erro:", erro.message)
}
```

12.4 Validações úteis

```
function criarAluno(nome, idade) {
  if (!nome || typeof nome !== "string") {
    throw new Error("Nome inválido")
  }

  if (typeof idade !== "number" || idade < 0) {
    throw new Error("Idade inválida")
  }

  return { nome, idade }
}
```

12.5 Introdução a testes unitários com Vitest

sum.js

```
export function sum(a, b) {
  return a + b
```

```
}
```

sum.test.js

```
import { describe, it, expect } from "vitest"
import { sum } from "./sum.js"

describe("sum", () => {
  it("deve somar dois números", () => {
    expect(sum(2, 3)).toBe(5)
  })
})
```

12.6 Testando casos de erro

dividir.test.js

```
import { describe, it, expect } from "vitest"
import { dividir } from "./dividir.js"

describe("dividir", () => {
  it("deve lançar erro quando divisor é zero", () => {
    expect(() => dividir(10, 0)).toThrow("Divisão por zero")
  })
})
```

12.7 Cobertura mínima de testes

Para funções críticas, tente cobrir:

1. caso comum;
2. caso de borda;
3. caso inválido.

12.8 Checklist de revisão antes de entregar

1. O código está legível?
2. Há validações mínimas?
3. Erros são tratados?

4. Existem testes para as funções principais?
5. Os nomes estão claros?

12.9 Exercícios

1. Refatore um código longo em funções menores.
2. Implemente validações com `throw` e `try/catch`.
3. Escreva 3 testes para funções matemáticas simples.
4. Escreva teste para um cenário de erro.
5. Desafio: testar uma função que recebe array e retorna média.

Capítulo 13

Projeto Final Integrador

Neste capítulo, você vai aplicar os principais conceitos do livro em um projeto completo.

13.1 Objetivo do projeto

Desenvolver um sistema de cadastro e organização de tarefas escolares.

13.2 Funcionalidades obrigatórias

1. Cadastrar tarefa com título e prazo.
2. Marcar tarefa como concluída.
3. Excluir tarefa.
4. Salvar dados em `localStorage`.
5. Organizar código em funções e, se possível, módulos.

13.3 Funcionalidades recomendadas

1. Filtro por status (todas, pendentes, concluídas).
2. Busca por texto.
3. Contador de tarefas concluídas e pendentes.
4. Validação de formulário.

13.4 Estrutura sugerida

```
projeto/
  index.html
  style.css
  app.js
```

Versão modular:

```
projeto/
  index.html
  style.css
  src/
    app.js
    storage.js
    tarefas.js
    ui.js
```

13.5 Modelo da tarefa

```
const tarefa = {
  id: Date.now(),
  titulo: "Estudar JavaScript",
  prazo: "2026-03-01",
  concluida: false
}
```

13.6 Etapa 1: interface base

Campos mínimos:

- input de título;
- input de prazo;
- botão de salvar;
- lista de tarefas.

13.7 Etapa 2: estado da aplicação

```
let tarefas = []
```

13.8 Etapa 3: criar tarefas

```
function criarTarefa(titulo, prazo) {
  return {
    id: Date.now(),
    titulo,
    prazo,
    concluida: false
  }
}
```

13.9 Etapa 4: renderizar lista

```
function renderizarTarefas(lista) {
  console.log("Renderizar", lista)
}
```

Depois substitua o `console.log` pela renderização real no DOM.

13.10 Etapa 5: concluir e remover

```
function alternarConclusao(id) {
  tarefas = tarefas.map((t) =>
    t.id === id ? { ...t, concluida: !t.concluida } : t
  )
}

function removerTarefa(id) {
  tarefas = tarefas.filter((t) => t.id !== id)
}
```

13.11 Etapa 6: persistência

```
function salvarNoStorage() {
  localStorage.setItem("tarefas", JSON.stringify(tarefas))
}

function carregarDoStorage() {
  const dados = localStorage.getItem("tarefas")
  tarefas = dados ? JSON.parse(dados) : []
}
```

13.12 Etapa 7: validação

```
function validarFormulario(titulo, prazo) {
  if (!titulo.trim()) return "Informe o título da tarefa"
  if (!prazo) return "Informe o prazo"
  return null
}
```

13.13 Roteiro de implementação

1. Montar HTML e CSS.
2. Implementar criação de tarefa.
3. Renderizar no DOM.
4. Implementar concluir/remover.
5. Integrar `localStorage`.
6. Adicionar filtros e busca.
7. Revisar e testar.

13.14 Critérios de avaliação

- funcionamento correto (40%);
- organização e legibilidade (25%);
- uso de boas práticas (15%);
- interface e experiência do usuário (10%);
- documentação do projeto (10%).

13.15 Desafios extras

1. Ordenar por prazo.
2. Destacar tarefas atrasadas automaticamente.
3. Exportar tarefas em JSON.
4. Criar modo escuro com botão de alternância.

13.16 Entrega sugerida

1. Repositório com código fonte.
2. `README.md` com instruções de uso.
3. Capturas de tela da aplicação.
4. Lista de melhorias futuras.

Parte II

Módulo 2 — Programação Orientada a Objetos

Capítulo 14

Introdução ao Node.js

O objetivo deste capítulo é que você entenda o que torna o Node.js único, como sua arquitetura não-bloqueante e o Event Loop funcionam, e como configurar um ambiente de desenvolvimento funcional. Além disso, você criará seu primeiro servidor "Hello World" usando o módulo nativo `http` do Node.js. O material é estruturado para ser claro, abrangente e prático, com exemplos comentados e explicações detalhadas.

14.1 Teoria: O que é Node.js?

14.1.1 Introdução ao Node.js

Node.js é uma **plataforma de desenvolvimento** que permite executar JavaScript fora do navegador. Ele foi criado em 2009 por Ryan Dahl e é baseado no motor **V8** do Google Chrome, que compila e executa código JavaScript com alta eficiência. Diferentemente do JavaScript tradicional, que roda no lado do cliente (navegadores), o Node.js possibilita o uso de JavaScript no lado do servidor, permitindo a construção de aplicações backend, como APIs, servidores web, ferramentas de linha de comando (CLI) e sistemas em tempo real.

Node.js é amplamente adotado por empresas como **Netflix**, **Uber**, **LinkedIn** e **PayPal** devido à sua alta performance, escalabilidade e capacidade de lidar com operações assíncronas. Ele é particularmente adequado para aplicações que requerem baixa latência e alta concorrência, como chats em tempo real, streaming de vídeo e APIs RESTful.

14.1.1.1 Por que Node.js é diferente?

- **JavaScript no servidor**: Antes do Node.js, JavaScript era limitado a navegadores. O Node.js trouxe a linguagem para o backend, permitindo que desenvolvedores usem uma única linguagem para o frontend e o backend, simplificando o desenvolvimento full-stack.
- **Leve e rápido**: O motor V8 é altamente otimizado, garantindo execução rápida de código JavaScript.
- **Ecossistema npm**: O Node.js vem com o **npm** (Node Package Manager), que abriga milhões de pacotes open-source, permitindo que desenvolvedores adicionem funcionalidades rapidamente.
- **Comunidade ativa**: A comunidade Node.js é uma das maiores do mundo, com constante atualização de bibliotecas, ferramentas e suporte em plataformas como X (#NodeJS) e Stack Overflow.

14.1.1.2 Histórico e Contexto

O Node.js surgiu em um momento em que o desenvolvimento web exigia soluções mais escaláveis para lidar com o crescimento de aplicações em tempo real. Antes de 2009, servidores web tradicionais (como Apache com PHP) usavam modelos síncronos baseados em threads, onde cada conexão de cliente consumia uma thread, limitando a escalabilidade em cenários de alta concorrência. Ryan Dahl percebeu que o JavaScript, com sua natureza assíncrona (ex.: callbacks em eventos DOM), poderia ser aproveitado para criar um servidor mais eficiente. Assim, o Node.js foi projetado para maximizar a concorrência usando um modelo não-bloqueante.

14.1.2 Arquitetura do Node.js Event Loop e Não Bloqueante

O diferencial do Node.js está em sua **arquitetura assíncrona e não-bloqueante**, impulsionada pelo **Event Loop**. Para entender como isso funciona, vamos explorar os conceitos fundamentais.

14.1.2.1 Modelo Não-Bloqueante

Em linguagens tradicionais como PHP ou Ruby (em configurações padrão), as operações de entrada/saída (I/O), como leitura de arquivos, consultas a bancos de dados ou chamadas de rede, são síncronas por padrão. Isso significa que o programa "para" enquanto espera a conclusão de uma operação. Por exemplo, ao ler um arquivo, o servidor aguarda até que o arquivo esteja completamente carregado antes de processar a próxima tarefa.

No Node.js, as operações de I/O são **assíncronas** e **não-bloqueantes**. O Node.js não espera a conclusão de uma tarefa de I/O para prosseguir. Em vez disso, ele delega a tarefa ao sistema operacional e continua executando outras partes do código. Quando a tarefa de I/O é concluída,

o Node.js é notificado por meio de **callbacks**, **promises** ou **async/await**.

Exemplo ilustrativo:

Imagine um restaurante onde o garçom (o Node.js) recebe pedidos de várias mesas (tarefas). Em um modelo síncrono, o garçom ficaria parado na cozinha esperando cada prato ficar pronto antes de atender outra mesa. No modelo não-bloqueante do Node.js, o garçom entrega o pedido à cozinha e imediatamente atende outras mesas, voltando apenas quando o prato está pronto.

Essa abordagem permite que o Node.js lide com **milhares de conexões simultâneas** com eficiência, tornando-o ideal para aplicações que requerem alta concorrência, como servidores de chat ou streaming.

14.1.2.2 O Event Loop

O **Event Loop** é o mecanismo central do Node.js, responsável por gerenciar todas as operações assíncronas. Ele opera em uma **única thread**, mas delega operações de I/O (como leitura de arquivos ou chamadas HTTP) ao sistema operacional, que as executa em threads separadas no background (via **libuv**, a biblioteca de I/O do Node.js). Quando uma tarefa de I/O é concluída, o Event Loop é notificado e executa a callback associada.

Funcionamento event loop

O Event Loop é um loop contínuo que verifica se há tarefas pendentes em diferentes **fases**. Cada fase é responsável por um tipo específico de tarefa. As principais fases são:

1. **Timers**: Executa callbacks de `setTimeout` e `setInterval` que atingiram seu tempo limite.
2. **Pending Callbacks**: Executa callbacks de operações de I/O que foram concluídas (ex.: leitura de arquivos).
3. **Idle, Prepare**: Fases internas usadas pelo Node.js para manutenção.
4. **Poll**: Recupera novos eventos de I/O (ex.: conexões de rede ou leitura de arquivos). Se não houver eventos, o Event Loop pode pausar aqui.
5. **Check**: Executa callbacks de `setImmediate`.
6. **Close Callbacks**: Executa callbacks de eventos de fechamento (ex.: fechar uma conexão de socket).

Exemplo visual:

Imagine uma roda-giratória em um parque de diversões. A roda (Event Loop) gira continuamente, verificando cada "assento" (fase) para ver se há uma tarefa pronta para ser executada. Se não houver tarefas, a roda continua girando. Se uma tarefa de I/O (como ler um arquivo) é iniciada, ela é delegada ao sistema operacional, e o Event Loop continua girando, verificando outras

tarefas. Quando o arquivo está pronto, a callback associada é colocada na fila para ser executada na próxima iteração.

luxo Simplificado do Event Loop

1. O Node.js inicia e entra no Event Loop.
2. Uma requisição HTTP chega ao servidor.
3. O Node.js delega a tarefa (ex.: buscar dados de um banco) ao sistema operacional via libuv.
4. Enquanto espera, o Event Loop processa outras requisições ou tarefas.
5. Quando os dados do banco estão prontos, a callback é enfileirada.
6. O Event Loop executa a callback na fase apropriada, enviando a resposta ao cliente.

14.1.2.3 Benefícios da Arquitetura

- **Alta concorrência**: O Node.js pode lidar com milhares de conexões simultâneas com uma única thread, ao contrário de servidores tradicionais que criam uma thread por conexão.
- **Baixa latência**: Como as operações não bloqueiam, o Node.js responde rapidamente, mesmo sob alta carga.
- **Escalabilidade**: Ideal para aplicações que precisam escalar horizontalmente, como APIs e sistemas em tempo real.
- **Eficiência de recursos**: Consome menos memória e CPU em comparação com modelos baseados em threads.

14.1.2.4 Limitações

Embora o Node.js seja poderoso, ele não é ideal para todas as situações:

- **Tarefas intensivas de CPU**: Como o Node.js opera em uma única thread, tarefas que exigem muito processamento (ex.: cálculos complexos ou compressão de vídeo) podem bloquear o Event Loop, reduzindo a performance. Para isso, o Node.js oferece o módulo `worker_threads`, que será abordado em módulos avançados.
- **Debugging complexo**: A natureza assíncrona pode dificultar a depuração em projetos mal estruturados, especialmente para iniciantes.
- **Curva de aprendizado**: Entender o Event Loop e gerenciar callbacks ou promises requer prática.

14.1.3 Casos de Uso do Node.js

Node.js é versátil e usado em uma ampla gama de aplicações. Aqui estão os principais casos de uso, com exemplos reais:

14.1.3.1 APIs RESTful

Node.js é amplamente utilizado para criar **APIs RESTful** devido à sua capacidade de lidar com muitas requisições simultâneas. Frameworks como **Express** simplificam a criação de rotas, middlewares e integração com bancos de dados.

Exemplo real: A **Netflix** usa Node.js para construir APIs que entregam conteúdo personalizado aos usuários, lidando com milhões de requisições por segundo.

14.1.3.2 Aplicações em Tempo Real

Aplicações que exigem comunicação em tempo real, como chats, jogos online ou notificações ao vivo, se beneficiam da arquitetura não-bloqueante do Node.js. Bibliotecas como **Socket.IO** facilitam a implementação de WebSockets.

Exemplo real: O **Slack** usa Node.js para suportar mensagens em tempo real e notificações push.

14.1.3.3 Streaming de Dados

Node.js é ideal para **streaming de dados**, como vídeos ou arquivos grandes, pois pode processar dados em pedaços (chunks) sem carregar tudo na memória.

Exemplo real: A **Netflix** utiliza Node.js para streaming de vídeo, otimizando a entrega de conteúdo em alta escala.

14.1.3.4 Ferramentas CLI

Node.js é amplamente usado para criar **ferramentas de linha de comando**, como o `create-react-app` ou o `vue-cli`, devido à sua facilidade de manipulação de arquivos e processos.

Exemplo real: O **npm**, o gerenciador de pacotes do Node.js, é ele próprio uma ferramenta CLI escrita em Node.js.

14.1.3.5 Microserviços

A arquitetura leve do Node.js o torna ideal para **microserviços**, onde diferentes serviços podem ser escritos e escalados independentemente.

Exemplo real: A **Uber** usa Node.js em sua arquitetura de microserviços para gerenciar diferentes partes de sua plataforma, como localização de motoristas e pagamentos.

14.1.3.6 Internet das Coisas (IoT)

Node.js é usado em dispositivos IoT devido à sua leveza e capacidade de lidar com muitas conexões simultâneas.

Exemplo real: Empresas de automação residencial usam Node.js para gerenciar dispositivos conectados, como lâmpadas inteligentes e sensores.

14.1.4 Por que Aprender Node.js em 2025?

- **Demanda de mercado**: Node.js é uma das tecnologias mais requisitadas em vagas de desenvolvimento backend e full-stack, segundo plataformas como LinkedIn e Indeed.
- **Ecossistema maduro**: Com frameworks como Express, NestJS e Fastify, o Node.js é robusto e moderno.
- **Integração com tecnologias modernas**: Node.js se integra facilmente com **TypeScript**, bancos NoSQL (MongoDB), bancos relacionais (PostgreSQL) e ferramentas de CI/CD.
- **Comunidade e recursos**: Milhares de pacotes no npm, tutoriais, e suporte ativo em comunidades como X (#NodeJS), Reddit (r/node) e GitHub.
- **Versatilidade**: Node.js é usado em startups, grandes empresas e projetos open-source, oferecendo oportunidades em diversos setores.

14.2 Ferramentas: Configurando o Ambiente

Antes de mergulharmos no exemplo prático, vamos configurar o ambiente de desenvolvimento. Esta seção cobre a instalação do Node.js, npm e a configuração do Visual Studio Code (VS Code) para garantir que você tenha tudo pronto para começar.

14.2.1 Instalando o Node.js e npm

O Node.js inclui o **npm** (Node Package Manager) por padrão, que será usado para gerenciar dependências e scripts.

14.2.1.1 Passos para Instalação

1. **Baixe o Node.js**:
   - Acesse o site oficial: nodejs.org.
   - Baixe a versão **LTS** (Long Term Support), que é a mais estável. Em 2025, a versão LTS mais recente provavelmente será a v20.x ou superior.

- Para **Windows** e **macOS**, use o instalador oficial. Para **Linux**, siga as instruções específicas para sua distribuição:

```
# Exemplo para Ubuntu
sudo apt update
sudo apt install nodejs npm
```

2. **Verifique a Instalação**:
   - Abra o terminal (Prompt de Comando, PowerShell ou Bash) e execute:

```
node -v
npm -v
```

   - Isso deve retornar as versões instaladas (ex.: `v20.12.2` e `10.5.0`).
3. **Atualize o npm (opcional)**:
   - Para garantir a versão mais recente do npm, execute:

```
npm install -g npm@latest
```

14.2.1.2 Dica Avançada

Considere usar o **nvm** (Node Version Manager) para gerenciar múltiplas versões do Node.js. Isso é útil para projetos que exigem versões específicas:

```
# Instalação do nvm (Linux/macOS)
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.5/install.sh | bash
# Instalar uma versão específica
nvm install 20
```

14.2.2 Configurando o Visual Studio Code

O **VS Code** é um editor de código leve e poderoso, amplamente usado para desenvolvimento Node.js.

14.2.2.1 Passos para Configuração

1. **Instale o VS Code**:
   - Baixe e instale em code.visualstudio.com.
2. **Instale Extensões Úteis**:
   - **ESLint**: Para linting e formatação de código JavaScript.
   - **Prettier**: Para formatação automática de código.
   - **Node.js Extension Pack**: Inclui suporte para depuração e snippets.
   - **REST Client**: Para testar APIs diretamente no VS Code.

- Para instalar, abra o VS Code, vá para a aba de extensões (`Ctrl+Shift+X`) e pesquise pelos nomes.

3. **Configurar o Terminal Integrado**:
   - Abra o terminal integrado (`Ctrl+``) e verifique se o `Node.js` está acessível comnode -v‘.
4. **Configurar Depuração**:
   - Crie um arquivo `launch.json` no diretório `.vscode`:

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "type": "node",
      "request": "launch",
      "name": "Launch Program",
      "program": "${workspaceFolder}/index.js"
    }
  ]
}
```

14.2.3 Estrutura do Projeto

Crie uma pasta para seus projetos Node.js:

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

Isso cria um arquivo `package.json` com configurações padrão:

```
{
  "name": "meu-primeiro-projeto",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
```

```
  "license": "ISC"
}
```

O `package.json` é o coração de qualquer projeto Node.js, definindo dependências, scripts e metadados.

14.3 Exemplo Prático: Criando um "Hello World" com o Módulo `http`

Agora que o ambiente está configurado, vamos criar nosso primeiro servidor Node.js usando o módulo nativo `http`. Este exemplo é simples, mas demonstra os conceitos fundamentais de criação de um servidor web com Node.js.

14.3.1 Objetivo do Exemplo

Criar um servidor HTTP que escuta na porta 3000 e responde com a mensagem "Hello World" quando acessado via navegador ou ferramenta como Postman.

14.3.2 Passo 1: Criar o Arquivo Principal

1. Na pasta `meu-primeiro-projeto`, crie um arquivo chamado `index.js`.
2. Adicione o seguinte código:

```
// Importa o módulo http nativo do Node.js
const http = require('http');

// Define o hostname e a porta onde o servidor vai rodar
const hostname = '127.0.0.1'; // localhost
const port = 3000;

// Cria o servidor HTTP
const server = http.createServer((req, res) => {
  // Define o status da resposta como 200 (OK) e o tipo de conteúdo como texto simple
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');

  // Envia a mensagem "Hello World" como resposta
```

```
  res.end('Hello World\n');
});

// Inicia o servidor e escuta na porta especificada
server.listen(port, hostname, () => {
  console.log(`Servidor rodando em http://${hostname}:${port}/`);
});
```

14.3.3 Explicação do Código

- **`const http = require('http')`**: Importa o módulo `http`, que permite criar servidores e clientes HTTP. O `require` é o sistema de módulos CommonJS, usado por padrão no Node.js.
- **`http.createServer`**: Cria um servidor HTTP. A função `(req, res)` é chamada toda vez que uma requisição HTTP é recebida:
  - `req` (request): Contém informações sobre a requisição, como método (GET, POST), URL e headers.
  - `res` (response): Usado para enviar a resposta ao cliente, como status, headers e corpo.
- **`res.statusCode = 200`**: Define o código de status HTTP (200 significa "OK").
- **`res.setHeader`**: Define o tipo de conteúdo da resposta como `text/plain`.
- **`res.end`**: Envia a resposta ao cliente e finaliza a conexão.
- **`server.listen`**: Inicia o servidor na porta e hostname especificados, exibindo uma mensagem no console quando o servidor está ativo.

14.3.4 Passo 2: Executar o Servidor

1. No terminal, na pasta do projeto, execute:

   ```
   node index.js
   ```

2. Você verá a mensagem: `Servidor rodando em http://127.0.0.1:3000/`.
3. Abra um navegador e acesse `http://localhost:3000`. Você verá a mensagem "Hello World".
4. Para parar o servidor, pressione `Ctrl+C` no terminal.

14.3.5 Passo 3: Testar com Ferramentas

- Use o **Postman** ou **curl** para testar a requisição:

```
curl http://localhost:3000
```

Isso deve retornar `Hello World`.

14.3.6 Passo 4: Explorando Mais

Vamos expandir o exemplo para lidar com diferentes rotas. Modifique o `index.js`:

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

const hostname = '127.0.0.1';
const port = 3000;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');

  // Verifica a URL da requisição
  if (req.url === '/') {
    res.end('Bem-vindo à página inicial!\n');
  } else if (req.url === '/about') {
    res.end('Esta é a página Sobre!\n');
  } else if (req.url === '/contact') {
    res.end('Entre em contato conosco!\n');
  } else {
    res.statusCode = 404;
    res.end('Página não encontrada!\n');
  }
});

server.listen(port, hostname, () => {
  console.log(`Servidor rodando em http://${hostname}:${port}/`);
});
```

14.3.6.1 Explicação do Código Expandido

- **`req.url`**: Contém o caminho da URL solicitada (ex.: `/`, `/about`).
- **Condicional `if/else`**: Verifica a URL e retorna uma resposta específica.
- **Status 404**: Retornado quando a URL não corresponde a nenhuma rota conhecida.

Teste as rotas acessando:

- `http://localhost:3000/` → "Bem-vindo à página inicial!"
- `http://localhost:3000/about` → "Esta é a página Sobre!"
- `http://localhost:3000/contact` → "Entre em contato conosco!"
- `http://localhost:3000/qualquercoisa` → "Página não encontrada!"

14.3.7 Passo 5: Depuração

Se algo der errado (ex.: porta já em uso), você verá um erro no terminal. Para depurar:

- Verifique se a porta 3000 está livre:

```
# Windows
netstat -a -n -o | find "3000"
# Linux/macOS
lsof -i :3000
```

- Use o VS Code para depuração:
  - Coloque um breakpoint clicando à esquerda de uma linha no `index.js`.
  - Pressione `F5` para iniciar a depuração e inspecionar variáveis como `req.url`.

14.4 Boas Práticas e Dicas

1. **Organize seu Código**:
   - Mantenha o código claro e comentado, mesmo em exemplos simples.
   - Use constantes para valores fixos, como `hostname` e `port`.
2. **Evite Bloqueios**:
   - O módulo `http` é assíncrono por natureza, mas evite operações síncronas (ex.: `fs.readFileSync`) em servidores reais, pois elas bloqueiam o Event Loop.
3. **Versionamento**:
   - Inicialize um repositório Git:

```
git init
git add .
git commit -m "Primeiro servidor Node.js"
```

4. **Teste Incrementalmente**:
   - Faça pequenas alterações e teste frequentemente para evitar erros acumulados.
5. **Explore o npm**:

- Instale o `nodemon` para reiniciar o servidor automaticamente ao salvar alterações:

```
npm install --save-dev nodemon
```

Adicione ao `package.json`:

```
"scripts": {
  "start": "node index.js",
  "dev": "nodemon index.js"
}
```

Execute com `npm run dev`.

14.5 Conclusão

Você aprendeu:

- **O que é Node.js**: Uma plataforma para executar JavaScript no servidor, baseada no motor V8.
- **Event Loop e Arquitetura Não-Bloqueante**: Como o Node.js lida com operações assíncronas para alta concorrência.
- **Casos de Uso**: APIs, streaming, aplicações em tempo real, CLIs e microserviços.
- **Configuração do Ambiente**: Instalação do Node.js, npm e VS Code.
- **Exemplo Prático**: Criação de um servidor HTTP simples com o módulo `http`.

O exemplo "Hello World" é apenas o começo, mas demonstra como o Node.js pode criar servidores web rapidamente e introduz conceitos fundamentais como requisições, respostas e o Event Loop. Nos próximos módulos, você expandirá esse conhecimento para criar APIs robustas, manipular arquivos e integrar bancos de dados.

14.5.1 Próximos Passos

- Experimente adicionar mais rotas ao servidor e testar diferentes métodos HTTP (ex.: POST).
- Explore a documentação do módulo `http` em nodejs.org.
- Prepare-se para o próximo capítulo, onde abordaremos **Módulos e CommonJS vs. ES Modules**.

Capítulo 15

Gerenciamento de Pacotes com npm/Yarn

O objetivo é que você compreenda como gerenciar pacotes de forma eficiente, organize dependências e automatize tarefas com scripts, preparando-se para projetos escaláveis. O material é dividido em duas partes: uma **teoria** detalhada sobre npm, Yarn e `package.json`, seguida de um **exemplo prático** que demonstra a integração da biblioteca `lodash` em um projeto Node.js.

15.1 Teoria: Estrutura do `package.json`, Dependências e Scripts npm/Yarn

15.1.1 Introdução ao Gerenciamento de Pacotes

No desenvolvimento Node.js, **pacotes** são bibliotecas ou módulos reutilizáveis que adicionam funcionalidades ao seu projeto, como manipulação de dados, criação de APIs ou automação de tarefas. O **npm** (Node Package Manager) é a ferramenta padrão do Node.js para gerenciar esses pacotes, enquanto o **Yarn** é uma alternativa popular que oferece melhor performance e funcionalidades adicionais. Ambos permitem instalar, atualizar e remover pacotes, além de gerenciar dependências e executar scripts personalizados.

O **npm** é instalado automaticamente com o Node.js e dá acesso ao maior repositório de pacotes open-source do mundo, o **npm registry** (npmjs.com), com milhões de pacotes disponíveis, como `lodash`, `express` e `jest`. O **Yarn**, desenvolvido pelo Facebook, é compatível com o npm registry, mas introduz recursos como instalação offline e maior determinismo.

O coração do gerenciamento de pacotes é o arquivo **`package.json`**, que define as configurações do projeto, dependências e scripts. Vamos explorar cada aspecto em detalhes.

15.1.2 O Arquivo `package.json`

O `package.json` é um arquivo JSON que serve como o manifesto do seu projeto Node.js. Ele contém metadados, dependências e scripts, garantindo que o projeto seja portátil e reproduzível em diferentes ambientes. O arquivo é criado automaticamente ao executar `npm init` ou `yarn init`.

15.1.2.1 Estrutura do `package.json`

Aqui está um exemplo típico de `package.json`:

```
{
  "name": "meu-projeto",
  "version": "1.0.0",
  "description": "Um projeto Node.js de exemplo",
  "main": "index.js",
  "type": "module",
  "scripts": {
    "start": "node index.js",
    "dev": "nodemon index.js",
    "test": "jest"
  },
  "keywords": ["node", "javascript"],
  "author": "Seu Nome",
  "license": "MIT",
  "dependencies": {
    "lodash": "^4.17.21",
    "express": "^4.18.2"
  },
  "devDependencies": {
    "nodemon": "^3.0.1",
    "jest": "^29.7.0"
  }
}
```

Campos principais:

- **`name`**: Nome do projeto ou pacote. Deve ser único se publicado no npm registry (ex.: `"meu-projeto"`).
- **`version`**: Versão do projeto, seguindo o padrão **SemVer** (Semantic Versioning, ex.:

`1.0.0`).

- **`description`**: Breve descrição do projeto, útil para documentação e publicação.
- **`main`**: Arquivo de entrada do projeto (ex.: `index.js`).
- **`type`**: Define o sistema de módulos (`"module"` para ES Modules ou `"commonjs"` para CommonJS).
- **`scripts`**: Comandos personalizados executados com `npm run <script>` ou `yarn <script>`.
- **`keywords`**: Palavras-chave para facilitar a busca no npm registry.
- **`author`**: Nome do autor (ex.: `"Seu Nome <seu.email@example.com>"`).
- **`license`**: Licença do projeto (ex.: `"MIT"`, `"ISC"`).
- **`dependencies`**: Pacotes necessários para a execução do projeto.
- **`devDependencies`**: Pacotes usados apenas em desenvolvimento ou testes.

15.1.2.2 Semantic Versioning (SemVer)

As versões de pacotes no `package.json` seguem o padrão **SemVer**: `MAJOR.MINOR.PATCH`. Exemplos:

- `1.0.0`: Versão inicial.
- `1.1.0`: Adiciona novas funcionalidades (MINOR), mas mantém compatibilidade.
- `2.0.0`: Introduz mudanças incompatíveis (MAJOR).
- `1.0.1`: Corrige bugs (PATCH) sem alterar funcionalidades.

Símbolos de versão: - `^4.17.21`: Permite atualizações de MINOR e PATCH (ex.: `4.x.x`). - `~4.17.21`: Permite apenas atualizações de PATCH (ex.: `4.17.x`). - `4.17.21`: Versão exata, sem atualizações automáticas.

15.1.2.3 Criando o `package.json`

- **Com npm**:

```
npm init
```

Responda às perguntas interativas (nome, versão, etc.) ou use:

```
npm init -y
```

para criar com valores padrão.

- **Com Yarn**:

```
yarn init
```

ou:

```
yarn init -y
```

15.1.3 Gerenciamento de Dependências

Dependências são pacotes externos listados no `package.json`. Elas são divididas em dois tipos:

- **`dependencies`**: Pacotes necessários para a execução do projeto em produção (ex.: `express`, `lodash`).
- **`devDependencies`**: Pacotes usados apenas em desenvolvimento ou testes (ex.: `nodemon`, `jest`).

15.1.3.1 Instalando Dependências

- **Com npm**:

```
npm install lodash
# Ou, abreviado:
npm i lodash
```

Para dependências de desenvolvimento:

```
npm install --save-dev nodemon
# Ou:
npm i -D nodemon
```

- **Com Yarn**:

```
yarn add lodash
```

Para dependências de desenvolvimento:

```
yarn add --dev nodemon
```

Ao instalar, o pacote é baixado para a pasta `node_modules`, e a versão é registrada no `package.json`.

15.1.3.2 O Arquivo `package-lock.json` (npm) ou `yarn.lock` (Yarn)

- **`package-lock.json`**: Gerado pelo npm, ele trava as versões exatas de todas as dependências (incluindo subdependências), garantindo consistência entre ambientes.
- **`yarn.lock`**: Equivalente no Yarn, com o mesmo propósito.

Boas práticas: - Sempre versione esses arquivos no Git para garantir builds reproduzíveis. - Evite editar manualmente; use comandos como `npm install` ou `yarn add`.

15.1.3.3 Atualizando Dependências

- **Com npm**:

```
npm update
```

Atualiza dependências dentro das faixas permitidas no `package.json`. Para verificar pacotes desatualizados:

```
npm outdated
```

- **Com Yarn**:

```
yarn upgrade
```

Para verificar pacotes desatualizados:

```
yarn outdated
```

15.1.3.4 Removendo Dependências

- **Com npm**:

```
npm uninstall lodash
```

- **Com Yarn**:

```
yarn remove lodash
```

15.1.3.5 Dependências Globais

Pacotes globais são instalados no sistema e podem ser usados em qualquer projeto:

- **npm**:

```
npm install -g nodemon
```

- **Yarn**:

```
yarn global add nodemon
```

Nota: Evite dependências globais em projetos reproduzíveis, pois elas podem causar inconsistências. Prefira dependências locais.

15.1.4 Scripts no `package.json`

O campo `scripts` no `package.json` permite definir comandos personalizados para automatizar tarefas, como iniciar o servidor, rodar testes ou formatar código.

15.1.4.1 Exemplo de Scripts

```
"scripts": {
  "start": "node index.js",
  "dev": "nodemon index.js",
  "test": "jest",
  "lint": "eslint .",
  "build": "tsc"
}
```

- **`start`**: Executado com `npm start` ou `yarn start`. Usado para iniciar a aplicação em produção.
- **`dev`**: Executado com `npm run dev` ou `yarn dev`. Usado para desenvolvimento com ferramentas como `nodemon`.
- **`test`**: Executado com `npm test` ou `yarn test`. Usado para rodar testes.
- **`lint`**: Verifica a qualidade do código com ESLint.
- **`build`**: Compila TypeScript (se aplicável).

15.1.4.2 4.2. Executando Scripts

- **npm**:

```
npm start
npm run dev
```

- **Yarn**:

```
yarn start
yarn dev
```

Nota: `start` e `test` não exigem `run`, mas outros scripts sim.

15.1.4.3 4.3. Scripts Pré e Pós

Você pode definir scripts que rodam automaticamente antes (`pre`) ou depois (`post`) de outro script:

```
"scripts": {
  "prestart": "npm run lint",
  "start": "node index.js",
  "poststart": "echo 'Servidor iniciado!'"
}
```

Executar `npm start` rodará `prestart`, `start` e `poststart` na ordem.

15.1.5 5. npm vs. Yarn: Diferenças e Quando Usar

Embora npm e Yarn sejam semelhantes, eles têm diferenças importantes:

Característica	npm	Yarn
Performance	Melhorou nas versões recentes (v10+)	Geralmente mais rápido
Lockfile	`package-lock.json`	`yarn.lock`
Instalação Offline	Suporte parcial	Cache offline robusto
Determinismo	Bom, mas Yarn é mais consistente	Altamente determinístico
Comandos	`npm install`, `npm run`	`yarn add`, `yarn`
Workspaces	Suporte básico	Suporte avançado para monorepos
Popularidade em 2025	Padrão, amplamente usado	Preferido em projetos grandes

15.1.5.1 5.1. Quando Usar npm?

- Projetos simples ou iniciantes, onde a simplicidade é prioridade.
- Quando você já usa o npm registry e não precisa de recursos avançados.
- Projetos que não requerem monorepos ou instalação offline.

15.1.5.2 5.2. Quando Usar Yarn?

- Projetos grandes ou monorepos (ex.: múltiplos pacotes em um repositório).
- Quando você precisa de instalação rápida e determinística.
- Projetos que exigem cache offline ou suporte avançado a workspaces.
- Equipes que preferem uma interface de comando mais amigável.

Em 2025, ambos são excelentes escolhas, mas **npm** é suficiente para a maioria dos projetos devido às melhorias recentes, enquanto **Yarn** é preferido em projetos complexos.

15.1.6 6. Boas Práticas para Gerenciamento de Pacotes

1. **Mantenha o `package.json` Limpo**:
   - Remova dependências não utilizadas com `npm prune` ou `yarn autoclean`.
   - Use nomes descritivos e atualize metadados (ex.: `description`, `author`).
2. **Versione o Lockfile**:
   - Inclua `package-lock.json` ou `yarn.lock` no Git para builds consistentes.
3. **Evite Dependências Desnecessárias**:
   - Verifique se um pacote é realmente necessário antes de instalá-lo.
   - Use ferramentas como `depcheck` para identificar dependências não usadas.
4. **Atualize Regularmente**:
   - Use `npm outdated` ou `yarn outdated` para identificar pacotes desatualizados.
   - Teste atualizações em um ambiente de desenvolvimento antes de aplicar em produção.
5. **Use Scripts para Automação**:
   - Crie scripts para tarefas comuns (ex.: linting, testes, build).
   - Combine ferramentas como `nodemon`, `eslint` e `jest` para produtividade.
6. **Segurança**:
   - Use `npm audit` ou `yarn audit` para identificar vulnerabilidades:

     ```
     npm audit
     npm audit fix
     ```

   - Monitore pacotes com ferramentas como Dependabot (GitHub).

15.2 Exemplo Prático: Instalar e Usar a Biblioteca `lodash`

Neste exemplo prático, vamos criar um projeto Node.js, instalar a biblioteca **lodash** (uma biblioteca utilitária popular para manipulação de arrays, objetos e strings), e usá-la para realizar operações comuns, como agrupamento e filtragem de dados. Implementaremos o exemplo com **npm** e mostraremos como replicar com **Yarn**.

15.2.1 Objetivo do Exemplo

- Configurar um projeto Node.js com `package.json`.
- Instalar a biblioteca `lodash` como dependência.
- Criar um script que usa funções do `lodash` para manipular uma lista de objetos.
- Executar o projeto com scripts personalizados.

15.2.2 Passo 1: Configurar o Projeto

1. Crie uma nova pasta para o projeto:

```
mkdir lodash-exemplo
cd lodash-exemplo
npm init -y
```

2. O `package.json` será criado:

```
{
  "name": "lodash-exemplo",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

3. (Opcional) Para usar ES Modules, adicione:

```
"type": "module"
```

4. Instale o `nodemon` como dependência de desenvolvimento:

```
npm install --save-dev nodemon
```

Atualize os scripts no `package.json`:

```
"scripts": {
  "start": "node index.js",
  "dev": "nodemon index.js"
}
```

15.2.3 Passo 2: Instalar a Biblioteca `lodash`

1. Instale o `lodash` como dependência de produção:

```
npm install lodash
```

2. Verifique o `package.json` atualizado:

```
{
  "name": "lodash-exemplo",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "type": "module",
  "scripts": {
    "start": "node index.js",
    "dev": "nodemon index.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "lodash": "^4.17.21"
  },
  "devDependencies": {
    "nodemon": "^3.0.1"
  }
}
```

3. O `package-lock.json` também será gerado, travando as versões exatas.

15.2.4 Passo 3: Criar o Script com `lodash`

1. Crie um arquivo `index.js` (ou `index.mjs` para ES Modules):

```
// index.mjs (para ES Modules)
import _ from 'lodash';

// Lista de exemplo: dados de usuários
const usuarios = [
  { id: 1, nome: 'Alice', idade: 25, cidade: 'São Paulo' },
  { id: 2, nome: 'Bob', idade: 30, cidade: 'Rio de Janeiro' },
  { id: 3, nome: 'Charlie', idade: 25, cidade: 'São Paulo' },
```

```
  { id: 4, nome: 'David', idade: 35, cidade: 'Belo Horizonte' }
];

// Usando lodash para manipular dados
// 1. Agrupar usuários por cidade
const porCidade = _.groupBy(usuarios, 'cidade');
console.log('Usuários por cidade:', porCidade);

// 2. Filtrar usuários com idade 25
const idade25 = _.filter(usuarios, { idade: 25 });
console.log('Usuários com 25 anos:', idade25);

// 3. Mapear apenas os nomes
const nomes = _.map(usuarios, 'nome');
console.log('Nomes dos usuários:', nomes);

// 4. Encontrar usuário por ID
const usuarioId2 = _.find(usuarios, { id: 2 });
console.log('Usuário com ID 2:', usuarioId2);
```

Para CommonJS (se não usar `"type": "module"`):

```
// index.js
const _ = require('lodash');

// Mesmo código acima, apenas com require em vez de import
const usuarios = [
  { id: 1, nome: 'Alice', idade: 25, cidade: 'São Paulo' },
  { id: 2, nome: 'Bob', idade: 30, cidade: 'Rio de Janeiro' },
  { id: 3, nome: 'Charlie', idade: 25, cidade: 'São Paulo' },
  { id: 4, nome: 'David', idade: 35, cidade: 'Belo Horizonte' }
];

const porCidade = _.groupBy(usuarios, 'cidade');
console.log('Usuários por cidade:', porCidade);

const idade25 = _.filter(usuarios, { idade: 25 });
console.log('Usuários com 25 anos:', idade25);
```

```
const nomes = _.map(usuarios, 'nome');
console.log('Nomes dos usuários:', nomes);

const usuarioId2 = _.find(usuarios, { id: 2 });
console.log('Usuário com ID 2:', usuarioId2);
```

2. **Explicação do Código**:
   - Importamos o `lodash` (usando _ por convenção).
   - Criamos uma lista de objetos `usuarios` para simular dados reais.
   - Usamos funções do `lodash`:
     - `_.groupBy`: Agrupa objetos por uma propriedade (ex.: `cidade`).
     - `_.filter`: Filtra objetos com base em critérios.
     - `_.map`: Extrai uma propriedade de cada objeto.
     - `_.find`: Busca o primeiro objeto que corresponde ao critério.

15.2.5 Passo 4: Executar o Projeto

1. Execute o script:

```
npm start
```

Ou, para desenvolvimento com reinício automático:

```
npm run dev
```

2. **Saída esperada**:

```
Usuários por cidade: {
  'São Paulo': [
    { id: 1, nome: 'Alice', idade: 25, cidade: 'São Paulo' },
    { id: 3, nome: 'Charlie', idade: 25, cidade: 'São Paulo' }
  ],
  'Rio de Janeiro': [
    { id: 2, nome: 'Bob', idade: 30, cidade: 'Rio de Janeiro' }
  ],
  'Belo Horizonte': [
    { id: 4, nome: 'David', idade: 35, cidade: 'Belo Horizonte' }
  ]
}
```

```
Usuários com 25 anos: [
  { id: 1, nome: 'Alice', idade: 25, cidade: 'São Paulo' },
  { id: 3, nome: 'Charlie', idade: 25, cidade: 'São Paulo' }
]
Nomes dos usuários: ['Alice', 'Bob', 'Charlie', 'David']
Usuário com ID 2: { id: 2, nome: 'Bob', idade: 30, cidade: 'Rio de Janeiro' }
```

15.2.6 Passo 5: Replicando com Yarn

1. Remova o `node_modules` e `package-lock.json`:

```
rm -rf node_modules package-lock.json
```

2. Instale o Yarn (se necessário):

```
npm install -g yarn
```

3. Instale as dependências com Yarn:

```
yarn add lodash
yarn add --dev nodemon
```

4. Verifique o `yarn.lock` gerado e execute:

```
yarn start
```

   Ou:

```
yarn dev
```

5. A saída será idêntica, mas o Yarn cria um `yarn.lock` em vez de `package-lock.json`.

15.2.7 Passo 6: Depuração e Boas Práticas

1. **Depuração**:
   - Use o VS Code para depurar:
     - Crie um `launch.json`:

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "type": "node",
```

```
      "request": "launch",
      "name": "Launch Program",
      "program": "${workspaceFolder}/index.js"
    }
  ]
}
```

   - Adicione breakpoints e execute com `F5`.

2. **Verifique Dependências**:
   - Use `npm list` ou `yarn list` para ver a árvore de dependências.
   - Verifique vulnerabilidades:

```
npm audit
yarn audit
```

3. **Versionamento**:
   - Adicione os arquivos ao Git:

```
git init
git add .
git commit -m "Projeto com lodash usando npm/yarn"
```

4. **Evite `node_modules` no Git**:
   - Crie um `.gitignore`:
     `node_modules/`

15.3 Conclusão

Nesta capítulo, você aprendeu: - **Estrutura do `package.json`**: Metadados, dependências, scripts e SemVer. - **Gerenciamento de Dependências**: Como instalar, atualizar e remover pacotes com npm e Yarn. - **Scripts Personalizados**: Automatização de tarefas com o campo `scripts`. - **npm vs. Yarn**: Diferenças, vantagens e casos de uso. - **Exemplo Prático**: Instalou e usou a biblioteca `lodash` para manipular dados em um projeto Node.js.

Esses conceitos são fundamentais para gerenciar projetos Node.js de forma eficiente e escalável. Nos próximos módulos, você aplicará esse conhecimento para construir APIs com Express e manipular arquivos com o módulo `fs`.

15.3.1 Próximos Passos

- Experimente instalar outras bibliotecas (ex.: `moment`, `axios`) e criar scripts para usá-las.
- Explore a documentação do npm: npmjs.com e Yarn: yarnpkg.com.
- Prepare-se para o próximo módulo, onde abordaremos o **Projeto Prático: Servidor HTTP Simples**.

Capítulo 16

Projeto Prático - Servidor HTTP Simples

O objetivo é consolidar sua compreensão sobre como o Node.js lida com requisições HTTP, como estruturar um servidor básico e como integrar manipulação de arquivos com o módulo `fs`. O material é dividido em duas partes: o **Projeto Principal**, que implementa o servidor com as rotas especificadas, e o **Desafio Avançado**, que estende a funcionalidade com leitura de arquivos JSON. O conteúdo é prático, com exemplos comentados, boas práticas e explicações detalhadas para garantir um aprendizado sólido.

16.1 Introdução ao Projeto

O módulo `http` do Node.js permite criar servidores web que respondem a requisições HTTP, como GET, POST, entre outros. Neste projeto, você criará um servidor que:

- Escuta na porta 3000.
- Responde a requisições GET nas rotas `/`, `/about` e `/contact` com mensagens de texto.
- Retorna um erro 404 para rotas desconhecidas.
- (No Desafio Avançado) Lê um arquivo JSON e retorna seu conteúdo em uma rota `/data`.

Este projeto é uma introdução prática à construção de servidores web com Node.js, preparando você para tópicos mais avançados, como APIs RESTful com Express. Vamos usar o módulo `http` para manter a simplicidade e focar nos fundamentos, e o módulo `fs` (File System) para o Desafio Avançado.

16.1.1 Objetivos do Projeto

- Criar um servidor HTTP funcional com o módulo `http`.

- Implementar rotas GET básicas (`/`, `/about`, `/contact`).
- Lidar com erros (ex.: 404 para rotas inválidas).
- (Desafio Avançado) Integrar leitura de arquivos JSON com o módulo `fs`.
- Aplicar boas práticas, como organização de código, tratamento de erros e versionamento com Git.

16.1.2 Pré-requisitos

- Node.js (versão LTS, v20.x ou superior em 2025) instalado.
- Visual Studio Code (VS Code) configurado com extensões recomendadas (ESLint, Prettier).
- Conhecimento básico de JavaScript e do módulo `http` (conforme a Introdução ao Node.js).
- Um projeto Node.js inicializado com `npm init -y`.

16.2 Projeto Principal: Criando o Servidor HTTP Simples

16.2.1 Passo 1: Configurar o Projeto

1. **Criar a Pasta do Projeto**: Crie uma nova pasta para o projeto e inicialize o `package.json`:

```
mkdir servidor-http-simples
cd servidor-http-simples
npm init -y
```

2. **Verificar o `package.json`**: O comando `npm init -y` cria um `package.json` com valores padrão:

```
{
  "name": "servidor-http-simples",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

3. **Adicionar Scripts e Nodemon**: Instale o `nodemon` para reiniciar o servidor automaticamente durante o desenvolvimento:

```
npm install --save-dev nodemon
```

Atualize o `package.json` com scripts úteis:

```
"scripts": {
  "start": "node index.js",
  "dev": "nodemon index.js"
}
```

4. **Configurar ES Modules (Opcional)**: Para usar ES Modules (recomendado em 2025), adicione ao `package.json`:

```
"type": "module"
```

Caso prefira CommonJS, pule esta etapa. O exemplo será apresentado em ambos os formatos.

16.2.2 Passo 2: Criar o Servidor HTTP

1. **Criar o Arquivo Principal**: Crie um arquivo `index.js` (ou `index.mjs` para ES Modules) na pasta do projeto.
2. **Implementar o Servidor**:
   - **CommonJS**:

```
// index.js
const http = require('http');

const hostname = '127.0.0.1';
const port = 3000;

const server = http.createServer((req, res) => {
  res.setHeader('Content-Type', 'text/plain; charset=utf-8');

  if (req.method === 'GET') {
    if (req.url === '/') {
      res.statusCode = 200;
      res.end('Bem-vindo à página inicial!\n');
    } else if (req.url === '/about') {
```

```
      res.statusCode = 200;
      res.end('Sobre nós: Somos uma equipe apaixonada por tecnologia!\n');
    } else if (req.url === '/contact') {
      res.statusCode = 200;
      res.end('Contato: Envie um e-mail para contato@exemplo.com\n');
    } else {
      res.statusCode = 404;
      res.end('Página não encontrada!\n');
    }
  } else {
    res.statusCode = 405;
    res.end('Método não permitido. Use GET.\n');
  }
});

server.listen(port, hostname, () => {
  console.log(`Servidor rodando em http://${hostname}:${port}/`);
});
```

- **ES Modules**:

```
// index.mjs
import http from 'http';

const hostname = '127.0.0.1';
const port = 3000;

const server = http.createServer((req, res) => {
  res.setHeader('Content-Type', 'text/plain; charset=utf-8');

  if (req.method === 'GET') {
    if (req.url === '/') {
      res.statusCode = 200;
      res.end('Bem-vindo à página inicial!\n');
    } else if (req.url === '/about') {
      res.statusCode = 200;
      res.end('Sobre nós: Somos uma equipe apaixonada por tecnologia!\n');
    } else if (req.url === '/contact') {
```

```
      res.statusCode = 200;
      res.end('Contato: Envie um e-mail para contato@exemplo.com\n');
    } else {
      res.statusCode = 404;
      res.end('Página não encontrada!\n');
    }
  } else {
    res.statusCode = 405;
    res.end('Método não permitido. Use GET.\n');
  }
});

server.listen(port, hostname, () => {
  console.log(`Servidor rodando em http://${hostname}:${port}/`);
});
```

3. **Explicação do Código**:
   - **Importação**: Usamos `require('http')` (CommonJS) ou `import http from 'http'` (ES Modules) para importar o módulo `http`.
   - **Configuração**: Definimos `hostname` (`127.0.0.1`) e `port` (3000) para o servidor.
   - **Criação do Servidor**: `http.createServer` cria o servidor, recebendo uma função de callback que lida com requisições (`req`) e respostas (`res`).
   - **Rotas**:
     - `req.method === 'GET'`: Verifica se a requisição é do tipo GET.
     - `req.url`: Verifica a URL solicitada (`/`, `/about`, `/contact`).
     - Respostas com `res.statusCode` e `res.end` para cada rota.
     - Status 404 para rotas desconhecidas.
     - Status 405 para métodos não suportados.
   - **Cabeçalho**: `Content-Type: text/plain; charset=utf-8` garante que o texto seja exibido corretamente, incluindo caracteres especiais.
   - **Inicialização**: `server.listen` inicia o servidor e exibe uma mensagem no console.

16.2.3 Passo 3: Executar o Servidor

1. **Executar o Projeto**:

```
npm start
```

Ou, para desenvolvimento com reinício automático:

```
npm run dev
```

2. **Testar as Rotas**:
   - Abra um navegador e acesse:
     - `http://localhost:3000/` → "Bem-vindo à página inicial!"
     - `http://localhost:3000/about` → "Sobre nós: Somos uma equipe apaixonada por tecnologia!"
     - `http://localhost:3000/contact` → "Contato: Envie um e-mail para contato@exemplo.com"
     - `http://localhost:3000/qualquercoisa` → "Página não encontrada!"
   - Use o **Postman** ou **curl** para testar:

```
curl http://localhost:3000/
curl http://localhost:3000/about
curl http://localhost:3000/contact
curl -X POST http://localhost:3000/ # Deve retornar "Método não permitido"
```

3. **Saída Esperada**:
   - No terminal: `Servidor rodando em http://127.0.0.1:3000/`.
   - No navegador ou curl: As mensagens correspondentes a cada rota.

16.2.4 Passo 4: Boas Práticas

1. **Organização do Código**:
   - Use constantes para valores fixos (`hostname`, `port`).
   - Adicione comentários claros para facilitar a manutenção.
   - Mantenha a lógica de rotas simples e modular (futuras aulas usarão Express para isso).
2. **Tratamento de Erros**:
   - Incluímos status 404 e 405 para lidar com rotas inválidas e métodos não suportados.
   - Considere adicionar logs para erros (ex.: `console.error`).
3. **Versionamento**:
   - Inicialize um repositório Git:

```
git init
git add .
git commit -m "Servidor HTTP simples com rotas GET"
```

   - Crie um `.gitignore`:

```
node_modules/
```

4. **Depuração**:
   - Configure o VS Code para depuração:

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "type": "node",
      "request": "launch",
      "name": "Launch Program",
      "program": "${workspaceFolder}/index.js"
    }
  ]
}
```

   - Adicione breakpoints para inspecionar `req.url` e `req.method`.
5. **Performance**:
   - Evite operações síncronas (ex.: `fs.readFileSync`) no servidor, pois bloqueiam o Event Loop.
   - Teste o servidor com múltiplas requisições simultâneas (ex.: usando ferramentas como `ab` ou `wrk`).

16.3 Desafio Avançado: Adicionar uma Rota que Lê um Arquivo JSON

Neste desafio, você adicionará uma rota `/data` que lê um arquivo JSON do sistema de arquivos e retorna seu conteúdo como resposta. Isso introduz o uso do módulo `fs` (File System) do Node.js e demonstra como integrar dados persistentes em um servidor HTTP.

16.3.1 Passo 1: Criar o Arquivo JSON

1. Crie um arquivo `data.json` na raiz do projeto com o seguinte conteúdo:

```
{
  "site": {
    "name": "Meu Site",
    "version": "1.0.0",
```

```
    "features": [
      "Página inicial dinâmica",
      "Seção sobre a equipe",
      "Formulário de contato"
    ]
  }
}
```

2. Verifique se o arquivo está na mesma pasta que `index.js` (ou `index.mjs`).

16.3.2 Passo 2: Atualizar o Servidor

Modifique o arquivo `index.js` (ou `index.mjs`) para incluir a rota `/data` que lê o arquivo JSON usando o módulo `fs`.

- **CommonJS**:

```
// index.js
const http = require('http');
const fs = require('fs').promises; // Usar versão assíncrona do fs

const hostname = '127.0.0.1';
const port = 3000;

const server = http.createServer(async (req, res) => {
  // Definir cabeçalho padrão
  res.setHeader('Content-Type', 'text/plain; charset=utf-8');

  if (req.method === 'GET') {
    if (req.url === '/') {
      res.statusCode = 200;
      res.end('Bem-vindo à página inicial!\n');
    } else if (req.url === '/about') {
      res.statusCode = 200;
      res.end('Sobre nós: Somos uma equipe apaixonada por tecnologia!\n');
    } else if (req.url === '/contact') {
      res.statusCode = 200;
      res.end('Contato: Envie um e-mail para contato@exemplo.com\n');
    } else if (req.url === '/data') {
```

```
      try {
        // Ler o arquivo JSON de forma assíncrona
        const data = await fs.readFile('./data.json', 'utf-8');
        // Alterar o Content-Type para JSON
        res.setHeader('Content-Type', 'application/json; charset=utf-8');
        res.statusCode = 200;
        res.end(data);
      } catch (error) {
        res.statusCode = 500;
        res.end('Erro ao ler o arquivo JSON: ' + error.message + '\n');
      }
    } else {
      res.statusCode = 404;
      res.end('Página não encontrada!\n');
    }
  } else {
    res.statusCode = 405;
    res.end('Método não permitido. Use GET.\n');
  }
});

server.listen(port, hostname, () => {
  console.log(`Servidor rodando em http://${hostname}:${port}/`);
});
```

- **ES Modules**:

```
// index.mjs
import http from 'http';
import { promises as fs } from 'fs';

const hostname = '127.0.0.1';
const port = 3000;

const server = http.createServer(async (req, res) => {
  res.setHeader('Content-Type', 'text/plain; charset=utf-8');

  if (req.method === 'GET') {
```

```
    if (req.url === '/') {
      res.statusCode = 200;
      res.end('Bem-vindo à página inicial!\n');
    } else if (req.url === '/about') {
      res.statusCode = 200;
      res.end('Sobre nós: Somos uma equipe apaixonada por tecnologia!\n');
    } else if (req.url === '/contact') {
      res.statusCode = 200;
      res.end('Contato: Envie um e-mail para contato@exemplo.com\n');
    } else if (req.url === '/data') {
      try {
        const data = await fs.readFile('./data.json', 'utf-8');
        res.setHeader('Content-Type', 'application/json; charset=utf-8');
        res.statusCode = 200;
        res.end(data);
      } catch (error) {
        res.statusCode = 500;
        res.end('Erro ao ler o arquivo JSON: ' + error.message + '\n');
      }
    } else {
      res.statusCode = 404;
      res.end('Página não encontrada!\n');
    }
  } else {
    res.statusCode = 405;
    res.end('Método não permitido. Use GET.\n');
  }
});

server.listen(port, hostname, () => {
  console.log(`Servidor rodando em http://${hostname}:${port}/`);
});
```

16.3.3 Passo 3: Explicação do Código do Desafio

- **Importação do Módulo `fs`**:
  - Usamos `fs.promises` (CommonJS: `require('fs').promises`, ES Modules:

`import { promises as fs } from 'fs'`) para operações assíncronas, evitando bloqueios no Event Loop.
  - O módulo `fs` permite interagir com o sistema de arquivos (leitura, escrita, etc.).
- **Rota `/data`**:
  - Verifica se `req.url === '/data'`.
  - Usa `fs.readFile` para ler o arquivo `data.json` de forma assíncrona.
  - Define o `Content-Type` como `application/json` para indicar que a resposta é JSON.
  - Retorna o conteúdo do arquivo com `res.end(data)`.
- **Tratamento de Erros**:
  - Usa `try/catch` para capturar erros (ex.: arquivo não encontrado).
  - Retorna status 500 com uma mensagem de erro se a leitura falhar.
- **Boas Práticas**:
  - Usa operações assíncronas (`fs.promises`) para manter o servidor não-bloqueante.
  - Define o charset (`utf-8`) para suportar caracteres especiais.
  - Mantém o código modular e comentado.

16.3.4 Passo 4: Testar o Desafio

1. **Executar o Servidor**:

```
npm run dev
```

2. **Testar a Rota `/data`**:

   - No navegador, acesse `http://localhost:3000/data`. Você verá o conteúdo do `data.json`:

```
{
  "site": {
    "name": "Meu Site",
    "version": "1.0.0",
    "features": [
      "Página inicial dinâmica",
      "Seção sobre a equipe",
      "Formulário de contato"
    ]
  }
}
```

- Com curl:

```
curl http://localhost:3000/data
```

3. **Testar Erros**:

- Renomeie ou remova o arquivo `data.json` e acesse `http://localhost:3000/data`. Você verá:

```
Erro ao ler o arquivo JSON: ENOENT: no such file or directory...
```

16.3.5 Passo 5: Depuração e Melhorias

1. **Depuração**:
   - Adicione breakpoints no VS Code na linha do `fs.readFile` para inspecionar o conteúdo de `data`.
   - Use `console.log` para depurar erros:

```
catch (error) {
  console.error('Erro ao ler JSON:', error);
  res.statusCode = 500;
  res.end('Erro ao ler o arquivo JSON: ' + error.message + '\n');
}
```

2. **Melhorias Possíveis**:
   - **Validação do JSON**: Parse o conteúdo com `JSON.parse` para garantir que é um JSON válido:

```
const data = await fs.readFile('./data.json', 'utf-8');
JSON.parse(data); // Lança erro se o JSON for inválido
```

   - **Cache**: Armazene o conteúdo do JSON em memória para evitar leituras repetidas:

```
let cache = null;
if (cache) {
  res.setHeader('Content-Type', 'application/json; charset=utf-8');
  res.statusCode = 200;
  res.end(cache);
} else {
  const data = await fs.readFile('./data.json', 'utf-8');
  cache = data;
  res.setHeader('Content-Type', 'application/json; charset=utf-8');
  res.statusCode = 200;
```

```
  res.end(data);
}
```

- **Modularização**: Mova a lógica de rotas para um arquivo separado (explorado em módulos futuros com Express).

3. **Versionamento**:
   - Atualize o repositório Git:

```
git add .
git commit -m "Adicionada rota /data para ler arquivo JSON"
```

16.4 Conclusão

Neste projeto prático, você: - Criou um **servidor HTTP simples** com o módulo `http`, implementando rotas GET para `/`, `/about` e `/contact`. - Lidaram com erros (404 para rotas inválidas, 405 para métodos não suportados). - Completou o **Desafio Avançado**, adicionando uma rota `/data` que lê um arquivo JSON com o módulo `fs.promises`. - Aplicou boas práticas, como tratamento de erros, uso de operações assíncronas e versionamento com Git.

Este projeto consolida os fundamentos do Node.js, como o Event Loop, requisições HTTP e manipulação de arquivos, preparando você para tópicos mais avançados, como APIs com Express e integração com bancos de dados.

16.4.1 Próximos Passos

- Experimente adicionar mais rotas (ex.: `/users`, `/products`) ou suportar outros métodos HTTP (ex.: POST).
- Explore o módulo `fs` para outras operações (ex.: escrita de arquivos).
- Prepare-se para o **Módulo 2**, onde abordaremos o **Sistema de Arquivos (fs module)** e operações assíncronas em profundidade.

Capítulo 17

Módulos e CommonJS vs. ES Modules

O objetivo é que você compreenda as características, vantagens e desvantagens de cada sistema de módulos, além de dominar a criação e importação de módulos personalizados. O material é estruturado em duas partes: uma **teoria** detalhada sobre CommonJS e ES Modules, seguida de um **exemplo prático** que demonstra a implementação em ambos os formatos.

17.1 Teoria: Diferenças entre CommonJS e ES Modules

17.1.1 Introdução aos Módulos no Node.js

Módulos são blocos de código reutilizáveis que permitem organizar aplicações Node.js de forma modular. Eles encapsulam funcionalidades específicas, como funções, objetos ou classes, e podem ser importados em outros arquivos para evitar repetição de código e melhorar a manutenibilidade. No Node.js, os módulos são essenciais para estruturar projetos escaláveis, desde pequenos scripts até APIs complexas.

O Node.js suporta dois sistemas de módulos principais: - **CommonJS**: O sistema original do Node.js, introduzido em 2009. - **ES Modules (ESM)**: O padrão oficial do JavaScript (ECMAScript), introduzido no ES6 (2015) e totalmente suportado no Node.js a partir da versão 12.

Ambos os sistemas têm sintaxes e comportamentos distintos, e entender suas diferenças é crucial para escolher o mais adequado ao seu projeto e escrever código compatível com as práticas modernas.

17.1.2 CommonJS: O Sistema de Módulos Original do Node.js

CommonJS é o sistema de módulos nativo do Node.js desde sua criação. Ele foi projetado para permitir modularidade em ambientes JavaScript fora do navegador, onde o ES6 ainda não existia. CommonJS é amplamente usado em projetos legados e em muitos pacotes npm, embora esteja sendo gradualmente substituído pelos ES Modules em projetos modernos.

17.1.2.1 Características do CommonJS

- **Sintaxe**: Usa `require()` para importar módulos e `module.exports` ou `exports` para exportá-los.
- **Carregamento Síncrono**: Módulos CommonJS são carregados de forma síncrona, ou seja, o Node.js lê e executa o módulo no momento da importação.
- **Escopo do Módulo**: Cada arquivo é tratado como um módulo independente, com seu próprio escopo. Variáveis locais não são acessíveis fora do módulo, a menos que sejam explicitamente exportadas.
- **Cópia de Valores**: Quando um módulo é importado, o Node.js cria uma cópia dos valores exportados. Alterações no módulo original após a importação não afetam a cópia importada (exceto para objetos, que são passados por referência).
- **Suporte Nativo**: CommonJS é suportado em todas as versões do Node.js, sem necessidade de configuração adicional.

17.1.2.2 Sintaxe do CommonJS

- **Exportação**:

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

Ou, alternativamente:

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

- **Importação**:

```
// arquivo: index.js
const math = require('./math');
console.log(math.soma(2, 3)); // 5
```

- **Notas**:
  - O caminho `./math` indica um módulo local no mesmo diretório. O sufixo `.js` é opcional.
  - `require()` pode importar módulos nativos (ex.: `fs`, `http`) ou pacotes npm (ex.: `lodash`).

17.1.2.3 Vantagens do CommonJS

- **Simplicidade**: A sintaxe é direta e fácil de entender para iniciantes.
- **Compatibilidade**: Funciona em todas as versões do Node.js e é amplamente usado em pacotes npm legados.
- **Carregamento Dinâmico**: Permite importar módulos condicionalmente ou dinamicamente (ex.: `require(algumaVariavel)`).
- **Suporte em Ferramentas**: Muitas ferramentas de build e bundlers (ex.: Webpack) suportam CommonJS nativamente.

17.1.2.4 Desvantagens do CommonJS

- **Carregamento Síncrono**: Pode causar atrasos em aplicações grandes, especialmente ao carregar muitos módulos no início.
- **Falta de Tree Shaking**: Não suporta tree shaking (remoção de código morto), o que pode aumentar o tamanho do bundle em aplicações frontend.
- **Incompatibilidade com ES Modules**: Não pode ser usado diretamente com ES Modules sem ferramentas de conversão (ex.: Babel).
- **Sintaxe Menos Moderna**: Comparada aos ES Modules, a sintaxe é considerada menos elegante e menos alinhada com o JavaScript moderno.

17.1.2.5 Como o CommonJS Funciona Internamente

Quando você usa `require('./math')`, o Node.js segue este processo: 1. **Resolução do Módulo**: O Node.js localiza o arquivo ou pacote com base no caminho fornecido. 2. **Carregamento**: O arquivo é lido e executado. 3. **Wrapping**: O Node.js envolve o código do módulo em uma função para criar um escopo isolado: `javascript (function (exports, require, module, __filename, __dirname) { // Código do módulo });` 4. **Caching**: O módulo é armazenado em cache, garantindo que múltiplas chamadas a `require()` retornem

a mesma instância. 5. **Exportação**: O objeto `module.exports` é retornado para o código que chamou `require()`.

17.1.3 ES Modules: O Padrão Moderno do JavaScript

ES Modules (ESM) é o sistema de módulos oficial do ECMAScript, introduzido no ES6 (2015). Ele foi projetado para ser um padrão unificado para JavaScript, funcionando tanto no navegador quanto no servidor (Node.js). Desde a versão 12 do Node.js (2019), os ES Modules são suportados nativamente, e em 2025, eles são o padrão recomendado para novos projetos devido à sua integração com o ecossistema JavaScript moderno.

17.1.3.1 Características dos ES Modules

- **Sintaxe**: Usa `import` para importar e `export` para exportar.
- **Carregamento Assíncrono**: Módulos ESM são carregados de forma assíncrona, permitindo melhor performance em aplicações modernas.
- **Escopo Estrito**: Funcionam automaticamente em modo estrito (`"use strict"`), garantindo maior segurança.
- **Suporte a Tree Shaking**: Permite que ferramentas de build (ex.: Rollup, Vite) removam código não utilizado, otimizando o tamanho do bundle.
- **Interoperabilidade**: Compatível com navegadores e ferramentas modernas, facilitando o desenvolvimento full-stack.

17.1.3.2 Sintaxe dos ES Modules

- **Exportação**:

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

Ou, exportando múltiplos itens:

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

- **Importação**:

```
// arquivo: index.mjs
import { soma } from './math.mjs';
console.log(soma(2, 3)); // 5
```

- **Exportação Padrão (Default)**:

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

```
// arquivo: index.mjs
import soma from './math.mjs';
console.log(soma(2, 3)); // 5
```

- **Notas**:
  - Arquivos ESM usam a extensão `.mjs` ou exigem `"type": "module"` no `package.json`.
  - O caminho deve incluir a extensão `.mjs` (ou `.js` com configuração apropriada).

17.1.3.3 Vantagens dos ES Modules

- **Padrão Oficial**: Alinhado com o ECMAScript, garantindo compatibilidade com navegadores e Node.js.
- **Carregamento Assíncrono**: Melhora a performance ao carregar módulos sob demanda.
- **Tree Shaking**: Reduz o tamanho do código em aplicações frontend.
- **Sintaxe Moderna**: Mais clara e consistente com o JavaScript moderno (ex.: `import` é mais intuitivo que `require`).
- **Suporte a Top-Level Await**: Permite usar `await` diretamente no nível superior do módulo (ex.: `const data = await fetch(...)`).

17.1.3.4 Desvantagens dos ES Modules

- **Configuração Adicional**: Em projetos Node.js, é necessário configurar `"type": "module"` ou usar a extensão `.mjs`.
- **Compatibilidade com CommonJS**: Nem todos os pacotes npm são compatíveis com ESM, exigindo soluções como `import()` dinâmico ou ferramentas de conversão.
- **Curva de Aprendizado**: A sintaxe e as regras (ex.: necessidade de extensões nos caminhos) podem confundir iniciantes.
- **Suporte Parcial em Versões Antigas**: Embora irrelevante em 2025, projetos legados em versões antigas do Node.js (pré-v12) não suportam ESM nativamente.

17.1.3.5 Como os ES Modules Funcionam Internamente

Quando você usa `import`, o Node.js segue este processo: 1. **Resolução do Módulo**: O Node.js localiza o arquivo com base no caminho, exigindo a extensão (ex.: `.mjs`). 2. **Parsing**: O módulo é analisado para identificar todas as dependências (`import` e `export`). 3. **Carregamento Assíncrono**: As dependências são carregadas em paralelo, usando promises. 4. **Linkagem**: Os exports são vinculados aos imports, criando referências ao vivo (live bindings). 5. **Execução**: O código do módulo é executado, e os valores exportados ficam disponíveis.

Live Bindings: Diferentemente do CommonJS, os ES Modules usam vinculação ao vivo. Se o valor exportado mudar no módulo original, a mudança é refletida no módulo que importou:

```
// arquivo: counter.mjs
export let count = 0;
export function increment() {
  count++;
}
```

```
// arquivo: index.mjs
import { count, increment } from './counter.mjs';
console.log(count); // 0
increment();
console.log(count); // 1
```

17.1.4 Diferenças Chave entre CommonJS e ES Modules

Característica	CommonJS	ES Modules
Sintaxe	`require` / `module.exports`	`import` / `export`
Carregamento	Síncrono	Assíncrono
Extensão do Arquivo	`.js` (padrão)	`.mjs` ou `.js` com `"type": "module"`
Escopo	Escopo isolado, mas não estrito	Modo estrito por padrão
Tree Shaking	Não suportado	Suportado
Live Bindings	Cópia de valores	Vinculação ao vivo
Top-Level Await	Não suportado	Suportado
Compatibilidade	Ampla (todos os Node.js, pacotes npm)	Nativo desde Node.js 12, mas requer configuração
Uso Dinâmico	`require(variavel)`	`import()` (importação dinâmica)

17.1.4.1 Exemplo de Diferenças Práticas

CommonJS:

```
// math.js
module.exports = {
  soma: (a, b) => a + b
};
```

```
// index.js
const { soma } = require('./math');
console.log(soma(2, 3)); // 5
```

ES Modules:

```
// math.mjs
export const soma = (a, b) => a + b;
```

```
// index.mjs
import { soma } from './math.mjs';
console.log(soma(2, 3)); // 5
```

17.1.4.2 Configuração no Node.js

Para usar ES Modules no Node.js, você deve: - Usar a extensão `.mjs` para arquivos ESM. - Ou adicionar `"type": "module"` ao `package.json`: `json { "name": "meu-projeto", "type": "module" }` - Para CommonJS, o padrão é `"type": "commonjs"` (ou ausência do campo).

17.1.4.3 Interoperabilidade

Misturar CommonJS e ES Modules pode ser desafiador: - **Importar CommonJS em ESM**: `javascript import { createRequire } from 'module'; const require = createRequire(import.meta.url); const modulo = require('./modulo.cjs');` Ou usar importação dinâmica: `javascript const modulo = await import('./modulo.cjs');`

- **Importar ESM em CommonJS**:

```
const { soma } = await import('./math.mjs');
```

Para projetos grandes, ferramentas como **Babel** ou **esbuild** podem converter entre formatos.

17.1.5 Quando Usar CommonJS vs. ES Modules?

- **Use CommonJS**:
  - Em projetos legados ou pacotes npm que não suportam ESM.
  - Quando a simplicidade e a compatibilidade são prioridades.
  - Em scripts rápidos onde a configuração extra do ESM não é necessária.
- **Use ES Modules**:
  - Em projetos novos ou modernos (recomendado em 2025).
  - Quando você precisa de tree shaking ou integração com navegadores.
  - Para aproveitar recursos como top-level await e sintaxe moderna.
  - Em projetos full-stack onde a consistência entre frontend e backend é importante.

Em 2025, **ES Modules** é o padrão recomendado para novos projetos Node.js devido à sua integração com o ecossistema JavaScript moderno e suporte a ferramentas como Vite, Rollup e TypeScript.

17.2 Exemplo Prático: Criar e Importar um Módulo Personalizado

Neste exemplo prático, vamos criar um módulo personalizado para realizar operações matemáticas (soma, subtração e multiplicação) e importá-lo em um arquivo principal. Implementaremos o exemplo em **CommonJS** e **ES Modules** para demonstrar as diferenças na prática.

17.2.1 Objetivo do Exemplo

- Criar um módulo personalizado chamado `math` que exporta funções matemáticas.
- Importar o módulo em um arquivo principal (`index.js` ou `index.mjs`) e usar suas funções.
- Testar ambos os formatos (CommonJS e ES Modules) em um projeto Node.js.

17.2.2 Passo 1: Configurar o Projeto

1. Crie uma nova pasta para o projeto:

```
mkdir modulo-personalizado
cd modulo-personalizado
npm init -y
```

2. O `package.json` será criado com configurações padrão:

```
{
  "name": "modulo-personalizado",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

3. Para ES Modules, modifique o `package.json` (opcional, usado na segunda parte):

```
{
  "name": "modulo-personalizado",
  "version": "1.0.0",
  "type": "module",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

4. Instale o `nodemon` para reiniciar o servidor automaticamente (opcional):

```
npm install --save-dev nodemon
```

Adicione ao `package.json`:

```
"scripts": {
  "start": "node index.js",
  "dev": "nodemon index.js"
}
```

17.2.3 Passo 2: Implementação com CommonJS

1. **Criar o Módulo Personalizado**: Crie um arquivo `math.js`:

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

function subtracao(a, b) {
  return a - b;
}

function multiplicacao(a, b) {
  return a * b;
}

module.exports = {
  soma,
  subtracao,
  multiplicacao
};
```

2. **Importar e Usar o Módulo**: Crie um arquivo `index.js`:

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

console.log('Soma: ', math.soma(5, 3)); // Soma: 8
console.log('Subtração: ', math.subtracao(5, 3)); // Subtração: 2
console.log('Multiplicação: ', math.multiplicacao(5, 3)); // Multiplicação: 15
```

3. **Executar o Projeto**:

```
node index.js
```

Ou, com nodemon:

```
npm run dev
```

Saída esperada:

```
Soma: 8
Subtração: 2
Multiplicação: 15
```

4. **Explicação**:
   - O módulo `math.js` exporta um objeto contendo três funções usando `module.exports`.
   - O `index.js` importa o módulo com `require` e acessa as funções via `math.soma`, `math.subtracao`, etc.
   - O carregamento é síncrono, e o Node.js armazena o módulo em cache após a primeira importação.

17.2.4 Passo 3: Implementação com ES Modules

1. **Atualizar o package.json**: Certifique-se de que o `package.json` inclui:

```
"type": "module"
```

2. **Criar o Módulo Personalizado**: Crie um arquivo `math.mjs`:

```
// math.mjs
export function soma(a, b) {
  return a + b;
}

export function subtracao(a, b) {
  return a - b;
}

export function multiplicacao(a, b) {
  return a * b;
}

// Exportação padrão (opcional)
export default {
  soma,
  subtracao,
  multiplicacao
};
```

3. **Importar e Usar o Módulo**: Crie um arquivo `index.mjs`:

```
// index.mjs
import { soma, subtracao, multiplicacao } from './math.mjs';
// Ou, usando importação padrão:
// import math from './math.mjs';

console.log('Soma: ', soma(5, 3)); // Soma: 8
console.log('Subtração: ', subtracao(5, 3)); // Subtração: 2
console.log('Multiplicação: ', multiplicacao(5, 3)); // Multiplicação: 15
```

4. **Executar o Projeto**:

```
node index.mjs
```

Ou, com nodemon:

```
npm run dev
```

Saída esperada:

```
Soma: 8
Subtração: 2
Multiplicação: 15
```

5. **Explicação**:
   - O módulo `math.mjs` usa `export` para exportar funções individualmente e, opcionalmente, um objeto padrão com `export default`.
   - O `index.mjs` importa as funções com `import`, especificando o caminho com a extensão `.mjs`.
   - O carregamento é assíncrono, e os ES Modules suportam live bindings, permitindo que alterações no módulo original sejam refletidas.

17.2.5 Passo 4: Testando Diferenças com Live Bindings

Para demonstrar os **live bindings** dos ES Modules, vamos criar um exemplo que mostra a diferença em relação ao CommonJS.

1. **CommonJS (Cópia de Valores)**:

```
// counter.js
let count = 0;

function increment() {
```

```
  count++;
  console.log('Count no módulo:', count);
}

module.exports = { count, increment };
```

```
// index.js
const { count, increment } = require('./counter');
console.log('Count inicial:', count); // 0
increment();
console.log('Count após increment:', count); // 0 (não reflete a mudança)
```

2. **ES Modules (Live Bindings)**:

```
// counter.mjs
export let count = 0;

export function increment() {
  count++;
  console.log('Count no módulo:', count);
}
```

```
// index.mjs
import { count, increment } from './counter.mjs';
console.log('Count inicial:', count); // 0
increment();
console.log('Count após increment:', count); // 1 (reflete a mudança)
```

3. **Executar**:

   - Para CommonJS: `node index.js`
   - Para ES Modules: `node index.mjs`

   Saída do CommonJS:

```
Count inicial: 0
Count no módulo: 1
Count após increment: 0
```

   Saída do ES Modules:

```
Count inicial: 0
```

```
Count no módulo: 1
Count após increment: 1
```

4. **Explicação**:
   - No CommonJS, `count` é uma cópia do valor exportado, então mudanças no módulo original não afetam o valor importado.
   - Nos ES Modules, `count` é uma referência ao vivo, refletindo mudanças no módulo original.

17.2.6 Passo 5: Boas Práticas

1. **Organize Módulos**:
   - Crie uma pasta `lib` ou `utils` para módulos reutilizáveis.
   - Nomeie arquivos de forma descritiva (ex.: `mathOperations.js`).
2. **Use ES Modules em Projetos Novos**:
   - Configure `"type": "module"` no `package.json` para consistência.
   - Prefira `.js` com ESM em vez de `.mjs` para evitar extensões específicas.
3. **Evite Misturar Formatos**:
   - Escolha um sistema (CommonJS ou ESM) para todo o projeto para evitar problemas de interoperabilidade.
   - Use `import()` dinâmico para casos excepcionais.
4. **Versionamento**:
   - Adicione os arquivos ao Git:

```
git init
git add .
git commit -m "Módulo personalizado com CommonJS e ES Modules"
```

5. **Depuração**:
   - Use o VS Code para depurar:
     - Adicione breakpoints nos arquivos `index.js` ou `index.mjs`.
     - Configure o `launch.json` para ESM, se necessário:

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "type": "node",
      "request": "launch",
      "name": "Launch ESM",
```

```
      "program": "${workspaceFolder}/index.mjs"
    }
  ]
}
```

17.3 Conclusão

Neste capítulo, você aprendeu: - **CommonJS**: Sistema de módulos síncrono, usando `require` e `module.exports`, ideal para projetos legados. - **ES Modules**: Padrão moderno, assíncrono, com `import` e `export`, recomendado para projetos novos em 2025. - **Diferenças**: Carregamento, sintaxe, live bindings, tree shaking e interoperabilidade. - **Exemplo Prático**: Criou um módulo personalizado (`math`) e o importou em ambos os formatos, testando diferenças como live bindings.

Este conhecimento é fundamental para organizar projetos Node.js de forma modular e escalável. Nos próximos módulos, você aplicará esses conceitos para construir APIs, manipular arquivos e integrar bancos de dados.

17.3.1 Próximos Passos

- Experimente criar novos módulos personalizados (ex.: um módulo para strings ou cálculos avançados).
- Explore a documentação de módulos no Node.js: nodejs.org/api/modules.html.
- Prepare-se para o próximo módulo, onde abordaremos o **Sistema de Arquivos (fs module)**.

Capítulo 18

Sobre os autores

Giseldo da Silva Neo

GISELDO DA SILVA NEO é Professor de Informática no Instituto Federal de Alagoas (IFAL) e desenvolve pesquisas na área de IA. Doutorado em Ciência da Computação na Universidade Federal de Campina Grande (UFCG). Possui Mestrado em Modelagem Computacional do Conhecimento (UFAL) e Mestrado em Contabilidade (FUCAPE). Possui MBA em Gestão e Estratégia Empresarial , Especialização em Arquitetura e Engenharia de Software, MBA em Gestão de Projetos. Graduação em Análise e Desenvolvimento de Sistemas e Graduação em Processos Gerenciais e possui nível Técnico em Informática (ETFSE - Escola Técnica Federal de Sergipe).

Alana Viana Borges da Silva Neo

ALANA VIANA BORGES DA SILVA NEO é Professora de Informática no Instituto Federal do Mato Grosso do Sul (IFMS) e desenvolve pesquisas na área de Informática na Educação. Doutoranda em Ciência da Computação na Universidade Federal de Campina Grande (UFCG), Mestra em Modelagem Computacional do Conhecimento na Universidade Federal de Alagoas (UFAL), Especialista em Estratégias Didáticas para a Educação Básica com Uso de TIC na Universidade Federal de Alagoas (UFAL), Especialista em Desenvolvimento de Software, Especialista em Segurança da Informação, Graduada em Análise e Desenvolvimento de Sistemas e Bacharel em Sistemas de Informação pela Universidade Estácio de Sá (ESTÁCIO) e Licenciatura em Computação pelo Claretiano Centro Universitário.

Contato

Caso deseje entrar em contato com os autores para reportar algum erro, crítica ou sugestão, envie e-mail para giseldo@gmail.com ou acesse o site https://giseldo.github.io/

Aviso Legal

As informações fornecidas neste trabalho são apenas para fins educacionais e informativos. Embora todos os esforços tenham sido feitos para garantir a precisão e a integridade, o autor e o editor não fazem declarações ou garantias de qualquer tipo, expressas ou implícitas, em relação à precisão, confiabilidade ou completude do conteúdo.

O autor e o editor não poderão ser responsabilizados por quaisquer danos ou perdas decorrentes do uso deste material. As opiniões expressas são de responsabilidade exclusiva do autor e não refletem necessariamente as opiniões de qualquer instituição ou organização com a qual o autor possa estar vinculado.

Uso da IA Generativa

Algumas partes textuais e algumas imagens foram criadas ou alteradas com várias das IAs Generativas disponíveis no momento da escrita. Porém, todo o texto foi revisado pelos autores e revisores.