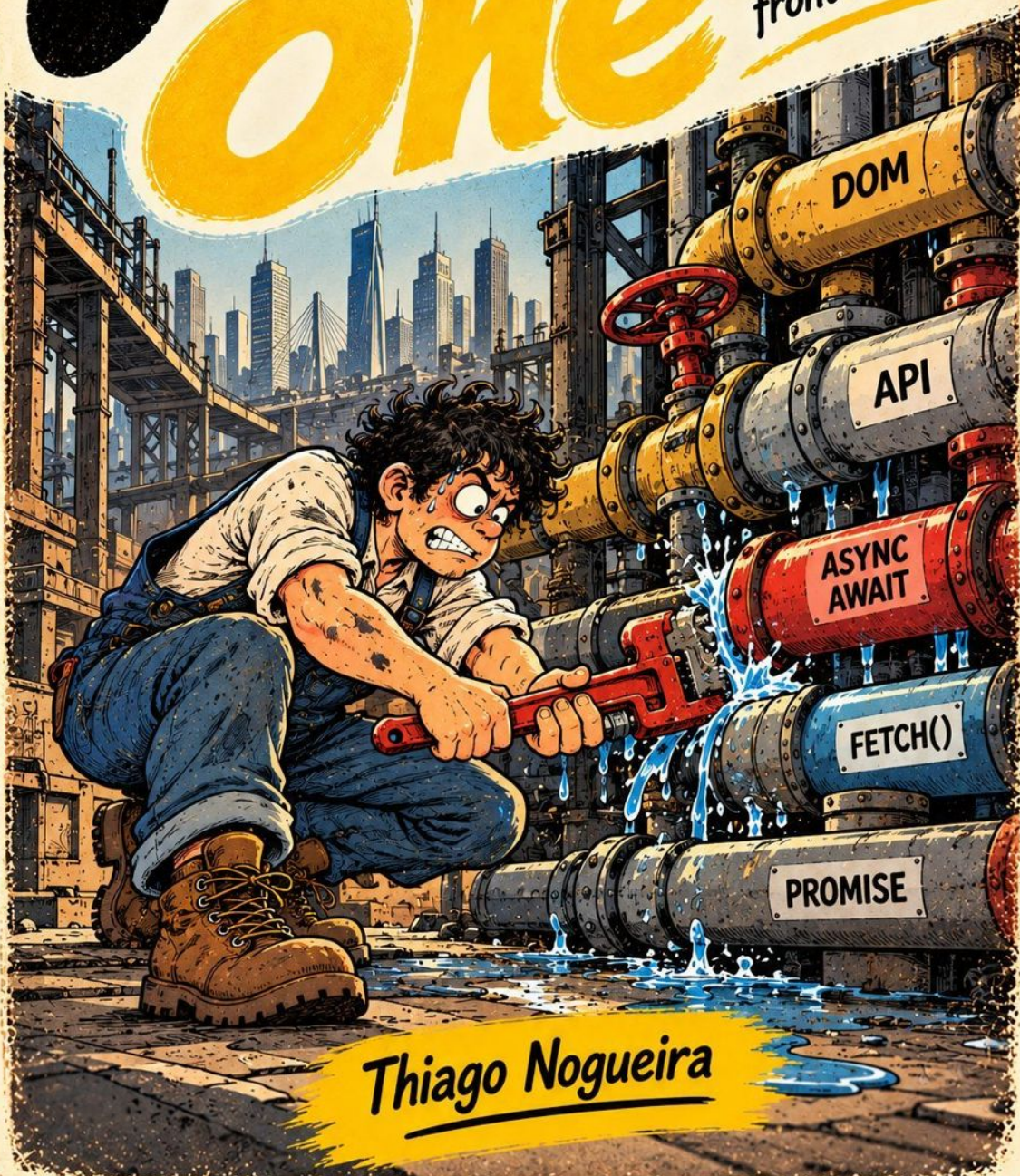


# JavaScript One

Do zero ao  
front moderno.



Thiago Nogueira

# JavaScript One — Pré-via

*Do primeiro console.log ao painel em produção*

---

**Thiago Nogueira**

PENSARBEM

## **JavaScript One — Prévia**

© 2026 Thiago Nogueira

Pensarbem

Prévia · 1ª edição

Composto em IBM Plex Serif, IBM Plex Sans e IBM Plex Mono. Diagramado por uma pipeline de tipografia programática.

# Sumário

---

|                                                                                        |    |
|----------------------------------------------------------------------------------------|----|
| <b>Antes de começar</b>                                                                | 1  |
| A Borba, a Calha e a data                                                              | 3  |
| Quem aparece                                                                           | 4  |
| Como o código aparece                                                                  | 5  |
| <b>1 Do Mocha ao Node</b>                                                              | 9  |
| 1.1 Maio de 1995                                                                       | 12 |
| 1.2 O nome, em dezembro                                                                | 13 |
| 1.3 Dois implementadores, um padrão                                                    | 13 |
| 1.4 A edição 4, que não foi publicada                                                  | 14 |
| 1.5 O que a página fez enquanto o padrão brigava                                       | 15 |
| 1.6 Fora do navegador, de novo                                                         | 16 |
| 1.7 A sexta edição, e o calendário                                                     | 17 |
| <b>2 O que vamos construir</b>                                                         | 21 |
| 2.1 O que o <code>package.json</code> está contando                                    | 24 |
| 2.2 O histórico que ele apontou                                                        | 25 |
| 2.3 O que o dia 6 de maio está cobrando                                                | 26 |
| 2.4 Instalando                                                                         | 27 |
| <b>3 O primeiro programa</b>                                                           | 30 |
| 3.1 O mesmo texto, no navegador                                                        | 34 |
| 3.2 Um nome para a situação                                                            | 35 |
| 3.3 Três falhas da primeira hora                                                       | 36 |
| 3.4 Comentário                                                                         | 37 |
| <b>4 Variáveis e tipos</b>                                                             | 41 |
| 4.1 Os tipos do dia a dia                                                              | 43 |
| 4.2 <code>undefined</code> e <code>null</code>                                         | 43 |
| 4.3 O <code>+</code> que junta e o <code>-</code> que converte                         | 43 |
| 4.4 <code>==</code> converte; <code>===</code> não                                     | 43 |
| 4.5 Dinheiro não mora em <code>number</code>                                           | 43 |
| <b>5 Operadores</b>                                                                    | 44 |
| 5.1 Aritmética                                                                         | 47 |
| 5.2 O <code>+</code> tem dois empregos                                                 | 47 |
| 5.3 Atribuição composta                                                                | 47 |
| 5.4 Comparar                                                                           | 47 |
| 5.5 <code>&amp;&amp;</code> , <code>  </code> , <code>!</code> — e o que eles devolvem | 47 |
| 5.6 Precedência                                                                        | 47 |

|           |                                                            |    |
|-----------|------------------------------------------------------------|----|
| <b>6</b>  | <b>Condicionais</b>                                        | 48 |
| 6.1       | <code>if</code> , <code>else if</code> , <code>else</code> | 51 |
| 6.2       | Dois <code>if</code> não são uma escolha                   | 51 |
| 6.3       | Uma função, um estado                                      | 51 |
| 6.4       | <code>switch</code> , para conjunto fechado                | 51 |
| 6.5       | A forma curta                                              | 51 |
| <b>7</b>  | <b>Repetições</b>                                          | 52 |
| 7.1       | <code>for...of</code>                                      | 55 |
| 7.2       | O <code>for</code> clássico                                | 55 |
| 7.3       | <code>while</code>                                         | 55 |
| 7.4       | O acumulador                                               | 55 |
| 7.5       | <code>continue</code> e <code>break</code>                 | 55 |
| 7.6       | A E-5033, antes e depois                                   | 55 |
| <b>8</b>  | <b>Funções</b>                                             | 59 |
| 8.1       | O que uma função é                                         | 61 |
| 8.2       | O frete, na linha 94                                       | 61 |
| 8.3       | Tirar uma responsabilidade                                 | 61 |
| 8.4       | Conferir contra o antigo                                   | 61 |
| 8.5       | Valor padrão e argumento que falta                         | 61 |
| 8.6       | A outra forma: arrow function                              | 61 |
| <b>9</b>  | <b>Escopo</b>                                              | 62 |
| 9.1       | Onde um nome existe                                        | 65 |
| 9.2       | O laço também é um bloco                                   | 65 |
| 9.3       | <code>var</code> : o escopo é a função                     | 65 |
| 9.4       | Existe antes de ser criado                                 | 65 |
| 9.5       | O nome sem declaração                                      | 65 |
| 9.6       | Sombra                                                     | 65 |
| <b>10</b> | <b>Arrays e objetos</b>                                    | 66 |
| 10.1      | Objeto: a entrega                                          | 69 |
| 10.2      | O nome que não existe                                      | 69 |
| 10.3      | Array: a manhã                                             | 69 |
| 10.4      | Os métodos que substituem o laço                           | 69 |
| 10.5      | Dois formatos, uma entrega                                 | 69 |
| <b>11</b> | <b>Desestruturação e spread</b>                            | 70 |
| 11.1      | Um objeto, dois nomes                                      | 73 |
| 11.2      | Spread: uma cópia de verdade                               | 73 |
| 11.3      | A cópia é rasa                                             | 73 |
| 11.4      | Desestruturação                                            | 73 |
| 11.5      | Parâmetros com nome                                        | 73 |
| 11.6      | O detalhe da E-5188                                        | 73 |

|           |                                                       |     |
|-----------|-------------------------------------------------------|-----|
| <b>12</b> | <b>Callbacks</b>                                      | 77  |
| 12.1      | Uma função como argumento                             | 79  |
| 12.2      | O que não espera                                      | 79  |
| 12.3      | Três continuações                                     | 79  |
| 12.4      | O erro no meio                                        | 79  |
| 12.5      | A convenção do erro primeiro                          | 79  |
| <b>13</b> | <b>Promises</b>                                       | 80  |
| 13.1      | Uma promessa de valor                                 | 83  |
| 13.2      | Rejeitar e <code>catch</code>                         | 83  |
| 13.3      | Encadear                                              | 83  |
| 13.4      | Fora de ordem                                         | 83  |
| 13.5      | <code>Promise.all</code> : o que precisa chegar junto | 83  |
| <b>14</b> | <b>async/await</b>                                    | 84  |
| 14.1      | <code>async</code> e <code>await</code>               | 87  |
| 14.2      | <code>try</code> e <code>catch</code>                 | 88  |
| 14.3      | O mesmo fluxo do capítulo anterior                    | 89  |
| 14.4      | Esperar em sequência sem precisar                     | 90  |
| 14.5      | O <code>await</code> que faltou                       | 90  |
| <b>15</b> | <b>Closures</b>                                       | 93  |
| 15.1      | Uma função lembra onde nasceu                         | 95  |
| 15.2      | O laço com <code>var</code>                           | 95  |
| 15.3      | O laço com <code>let</code>                           | 95  |
| 15.4      | Melhor ainda: não depender da posição                 | 95  |
| 15.5      | Closure de propósito                                  | 95  |
| <b>16</b> | <b>DOM e eventos</b>                                  | 99  |
| 16.1      | A página como objeto                                  | 101 |
| 16.2      | Achar e mudar                                         | 101 |
| 16.3      | Escutar um clique                                     | 101 |
| 16.4      | Dois ouvintes                                         | 101 |
| 16.5      | Um ouvinte para a lista inteira                       | 101 |
| <b>17</b> | <b>Debugging</b>                                      | 102 |
| 17.1      | Reproduzir antes de explicar                          | 106 |
| 17.2      | As ferramentas do navegador                           | 106 |
| 17.3      | O estilo calculado                                    | 107 |
| 17.4      | Causa e sintoma                                       | 108 |
| 17.5      | O console, e o <code>debugger</code>                  | 108 |
| 17.6      | O teste da receita                                    | 109 |
| <b>18</b> | <b>Estado</b>                                         | 111 |
| 18.1      | Onde o dado mora                                      | 113 |
| 18.2      | A reprodução                                          | 113 |

|           |                                        |            |
|-----------|----------------------------------------|------------|
| 18.3      | Um lugar só                            | 113        |
| 18.4      | Mudar e redesenhar                     | 113        |
| 18.5      | No navegador                           | 113        |
| <b>19</b> | <b>Módulos</b>                         | <b>117</b> |
| 19.1      | Um arquivo, um módulo                  | 119        |
| 19.2      | O formato de 2022                      | 119        |
| 19.3      | A ordem em que os módulos rodam        | 119        |
| 19.4      | A dependência circular                 | 119        |
| 19.5      | O terceiro arquivo                     | 119        |
| 19.6      | Exportação padrão                      | 119        |
| <b>20</b> | <b>npm</b>                             | <b>120</b> |
| 20.1      | O <code>package.json</code> , de novo  | 123        |
| 20.2      | O lockfile                             | 123        |
| 20.3      | Onze linhas, em 2016                   | 123        |
| 20.4      | Atualizar vendo o que quebra           | 123        |
| 20.5      | De três clientes para um, numa chamada | 123        |
| <b>21</b> | <b>Testes</b>                          | <b>124</b> |
| 21.1      | Um teste é uma afirmação que roda      | 127        |
| 21.2      | O teste do caso do Paulo               | 127        |
| 21.3      | A correção                             | 127        |
| 21.4      | O que testar                           | 127        |
| 21.5      | O que o verde não diz                  | 127        |
| <b>22</b> | <b>APIs</b>                            | <b>131</b> |
| 22.1      | Um pedido e uma resposta               | 133        |
| 22.2      | O número diz o que aconteceu           | 133        |
| 22.3      | O servidor pequeno                     | 133        |
| 22.4      | <code>fetch</code> não falha num 422   | 133        |
| 22.5      | Os dois verdes                         | 133        |
| <b>23</b> | <b>Performance</b>                     | <b>134</b> |
| 23.1      | Medir primeiro                         | 137        |
| 23.2      | O que o redesenho fazia                | 137        |
| 23.3      | Redesenhar só o que mudou              | 137        |
| 23.4      | Eventos que disparam demais            | 137        |
| <b>24</b> | <b>Git</b>                             | <b>141</b> |
| 24.1      | O que o histórico guarda               | 143        |
| 24.2      | Uma mensagem que responde              | 143        |
| 24.3      | O que não entra                        | 143        |
| 24.4      | Ramo e merge                           | 143        |
| <b>25</b> | <b>Deploy</b>                          | <b>144</b> |

|           |                                 |            |
|-----------|---------------------------------|------------|
| 25.1      | O que funciona na minha máquina | 147        |
| 25.2      | O que tem de ser igual          | 147        |
| 25.3      | O horário                       | 147        |
| 25.4      | Segunda de manhã                | 147        |
| <b>26</b> | <b>Projeto final</b>            | <b>148</b> |
| 26.1      | O que a pasta tem               | 151        |
| 26.2      | Subir e testar                  | 151        |
| 26.3      | Três entregas, o papel e a tela | 151        |
| 26.4      | Os trinta segundos              | 151        |
|           | <b>Respostas</b>                | <b>152</b> |
|           | <b>Glossário</b>                | <b>159</b> |



# Antes de começar

*Três jeitos de pedir a mesma entrega, um comentário pedindo para não mexer, e uma lista impressa às 5h35 que ainda manda mais que o painel.*

---

Esta é a pasta do painel da Borba, no ar desde março de 2022:

```
1  painel/  
2    package.json  
3    index.html  
4    lista.js  
5    lista.css  
6    entrega.js  
7    http/  
8      client.js
```

O `package.json` anota uma biblioteca, o `Axios`, na versão 0.21.1. O `lista.js` pede dados com `fetch`, que já vem com a linguagem. A pasta `http/` tem um terceiro jeito, escrito à mão. Os três perguntam a outro computador como está uma entrega.

No fim de `entrega.js` tem um comentário de duas linhas de idade e quatro anos de obediência:

```
// NÃO MEXER. FUNCIONA.
```

E funciona.

Cento e sessenta motoristas, três cidades, o primeiro caminhão saindo às 5h40. A Livia imprime a lista às 5h35. Quando o papel e o painel discordam, o caminhão sai pelo papel. O Seu Aldo, no cais, acredita no papel. Nesses quatro anos ele acertou o bastante para a sala inteira ter aprendido a não discutir.

Você vai escrever o programa que merece discordar dele. A pasta de produção fica onde está. Mexer nela na primeira semana é o jeito mais curto de parar a manhã.

## A Borba, a Calha e a data

A **Borba Entregas** faz entrega no mesmo dia em Ribeirão Preto, Franca e São Carlos. A **Estúdio Calha** — dezenove pessoas, o tipo de casa de software que herda o sistema quando quem escreveu já foi embora — assinou para o painel aguentar uma cláusula nova.

E existe uma data que ninguém pode empurrar.

A **Mercado Leste**, onze lojas, renova o contrato com a Borba na quarta-feira, **6 de maio**. O fornecimento vale R\$ 2,1 milhões por ano. A cláusula nova pede que cada status apareça no painel em até trinta segundos depois da marcação do motorista. Cada atraso custa R\$ 3,50.

A Livia contou o buraco de hoje em cerca de 2.400 por dia: R\$ 8.400 por dia, mais de R\$ 180 mil num mês, se nada mudar.

O contrato da Calha é R\$ 96 mil, preço fechado. Um terço disso já virou reunião. Se a cláusula estiver de pé em 6 de maio, começa uma mensalidade de R\$ 14 mil. Se não estiver, a Borba paga a multa e a mensalidade não começa.

O servidor que serve o painel de manhã roda **Node 20**. Essa linha deixa de receber correção em **30 de abril**, seis dias antes da renovação.

## Quem aparece

**Bia** entrou na Calha em 2 de março. Recebeu acesso ao repositório na sexta, dia 13, e passou o fim de semana lendo a pasta. Aprende rápido. A pergunta que ela faz é, em geral, a que a sala estava evitando.

**Rafael** é sênior. Já viu decisão temporária o bastante para saber quanto tempo ela dura. Responde curto e, quando pode, aponta para o arquivo em vez de explicar.

**Marcos** é o tech lead. Gosta de arquitetura e de desenhar coisa que ainda não existe. A tentativa de trocar um cliente HTTP por outro, pela metade, é dele.

**Cláudia** é a PO. Traduz a operação para o time. A tradução às vezes perde a restrição, porque ela não estava no cais às 5h40. O ofício dela é esse: transformar a frase da Livia em alguma coisa que vira tarefa.

**Paulo** é o QA. Tem um prazer quase científico pelo caso que o desenvolvedor jurou que não acontecia.

**Neto** é o DevOps. Aparece quando alguém diz que funciona na própria máquina.

**Livia Borba** toca a operação. Imprime a lista, abre o painel, e é a única pessoa do contrato capaz de descrever o que o sistema faz.

**Seu Aldo** fundou a Borba com uma van, em 1998. Continua no cais às 5h40. Quase não entra em reunião. Quando o papel e a tela discordam, o caminhão obedece a ele.

**Diego Pacheco** escreveu o painel em março de 2022 e saiu da Calha em novembro de 2024. O comentário é dele. Não é vilão. O arquivo despachou caminhão por quatro anos.

**Renato** fundou a Calha. Nunca escreveu JavaScript e não finge que escreveu. Volta de palestra com uma caixa nova no slide.

E o **painel**, com o comentário no fim da função, é o sistema em produção. Cada coisa nova que aparecer aqui vai ser medida contra ele.

## Como o código aparece

Código aparece assim, às vezes com o nome do arquivo:

**STATUS.JS**

javascript

```
const situacao = "a caminho";  
console.log(situacao);
```

`const` cria um nome e o prende a um valor. Aqui o valor é o texto entre aspas. `console.log` escreve esse valor no terminal. O ponto e vírgula encerra a instrução.

O que o terminal responde aparece sem nome de arquivo e sem realce:

```
a caminho
```

Quando o programa quebra, o que interessa vem primeiro: o arquivo, a linha, um cursor em cima do ponto e o tipo da falha. As linhas de baixo, quando aparecerem, são o motor atravessando o próprio código até chegar na sua. A primeira é a sua.

### ● O ESSENCIAL

Comando de terminal aparece com \$ na frente. O \$ representa o prompt e não faz parte do comando: não digite.

Você não precisa instalar nada para começar a ler. Quando o primeiro programa precisar rodar, a instalação vem junto, com o teste que confirma que deu certo.

A data que ninguém pode empurrar é 6 de maio. Até lá, a Livia imprime às 5h35, e o caminhão sai pelo papel quando o painel discorda.





PARTE 1

# O que já está no ar

*De onde a linguagem veio, por que o painel é JavaScript, e o primeiro programa rodando na máquina de quem acabou de receber o acesso.*

## NESTA PARTE

---

- 1 Do Mocha ao Node
- 2 O que vamos construir
- 3 O primeiro programa



## CAPÍTULO

# 1

## Do Mocha ao Node

*Dez dias em maio de 1995, um acordo com a Sun em dezembro, e um padrão que não pôde herdar o nome comercial.*

---

*“JavaScript is an easy-to-use object scripting language designed for creating live online applications that link together objects and resources on both clients and servers.”*

COMUNICADO CONJUNTO DA NETSCAPE E DA SUN, 4 DE DEZEMBRO DE 1995

### AO FIM DESTES CAPÍTULO

Contar a história documentada do JavaScript: a encomenda de 1995, o padrão que recebeu outro nome, a edição 4 que não saiu, o V8, o Node e o calendário de versões que o servidor da Borba ainda obedece.

**ENQUANTO ISSO, NA EMPRESA** Por que JavaScript?

O Renato tinha visto uma palestra na quarta e voltou com três frases na quinta.

— Esse painel da Borba. Por que JavaScript?

— Porque é o que está no ar — disse a Bia. — A página é JavaScript. O servidor é JavaScript. A Livia usa isso desde 2022.

— Mas a gente não ia modernizar?

— Vamos. De um arquivo que só o Diego sabia mexer para um programa que a Mercado Leste consiga cobrar em trinta segundos.

O Renato fez o gesto de quem tira uma coisa da frente do rosto.

— Não dava pra fazer um aplicativo de verdade?

— Dava.

— Então.

— Aí a gente joga fora quatro anos de regra que não estão escritas em lugar nenhum, e entrega em 6 de maio.

Pausa.

— É que JavaScript é meio... — ele procurou a palavra — ...de botão.

O Rafael não levantou os olhos da tela.

— Temporário dura três anos. Esse está com quatro.

A Cláudia falou sem tirar o olho da planilha da semana.

— O painel da manhã roda em quê?

— Node — disse o Marcos.

— Qual versão?

O Rafael respondeu antes.

— Vinte. No servidor. Para de receber correção em 30 de abril.

---

A reunião foi na quinta, 12 de março, véspera do acesso da Bia ao repositório. Ela repetiu o que a sala já dizia. O Rafael falou a versão que estava instalada no servidor.

Node é o programa que executa essa linguagem fora do navegador. A versão 20 é a da Borba. A data de 30 de abril faz parte do calendário público dessa linha, e o calendário tem uma origem: em 2015 o projeto do Node passou a publicar até quando cada versão recebe correção.

A linguagem em si começa antes, num navegador, com outro nome.

## 1.1 Maio de 1995

Brendan Eich chegou à Netscape em abril de 1995. A empresa tinha pouco mais de um ano. O Netscape Navigator era o navegador mais usado, e uma página, naquele ano, era um documento: texto, link, figura. A Netscape queria uma linguagem dentro do navegador para o documento reagir a um clique sem pedir a página inteira de novo.

Havia duas pressões em cima da mesa. Uma era Java, da Sun, que a Netscape ia licenciar: linguagem da moda, com compilação, classe e tipo declarado, feita para programa grande. A outra era o que tinham oferecido ao Eich na contratação: algo leve, no espírito de Scheme, para quem escrevia página. Scheme é uma linguagem em que função é um valor, pequena o bastante para uma pesquisa de laboratório. O acordo que saiu dali não foi nenhuma das duas propostas inteiras.

O protótipo, escrito em maio, levou o nome interno de Mocha. Eich depois descreveu o prazo como cerca de dez dias. Nesses dias entrou o núcleo que a linguagem ainda tem.

Uma função era um valor. Dava para guardar esse valor num nome e entregar a outra função. Isso vinha de Scheme. O navegador, dali em diante, pôde tratar um clique assim: o botão recebe uma função e a chama quando o clique acontece.

Um objeto era um conjunto de propriedades, e podia apontar para outro objeto. Se a propriedade não está nele, a busca continua no outro. Esse outro se chama protótipo. O modelo veio de Self, uma linguagem de objetos sem classe. Java, ao contrário, começa pela classe: o molde existe antes do objeto. No Mocha, o objeto existe primeiro. O `new` e as funções construtoras entraram para a sintaxe lembrar Java de longe — chave, ponto e vírgula, nome com inicial maiúscula — sem adotar o molde.

**Protótipo** O objeto para o qual outro objeto aponta quando não acha uma propriedade em si mesmo. A busca segue esse ponteiro. É o modelo de objeto do JavaScript desde o protótipo de maio de 1995. Classe, no sentido de Java, não fazia parte dessa encomenda.

**Java** Linguagem da Sun, anunciada em 1995, com classe, tipo declarado e compilação. A Netscape ia embarcá-la no navegador para aplicação grande. O Mocha recebeu ordem de parecer com ela na superfície. O objeto, a função e o momento em que o programa roda ficaram de outro desenho.

### 🕒 OS DEZ DIAS

O número que circula — dez dias — é o prazo do primeiro protótipo, em maio de 1995. Entre maio e dezembro o código foi reescrito para caber no Navigator, ganhou dois nomes públicos e foi anunciado num contrato com outra empresa.

O núcleo de função-como-valor e de protótipo é que atravessou esse semestre. O prazo explica a pressa do primeiro arquivo. O arquivo que a Livia abre saiu de trinta anos de edição em cima desse núcleo.

## 1.2 O nome, em dezembro

Em setembro de 1995 o beta do Netscape Navigator 2.0 levava a linguagem com o nome LiveScript. Em 4 de dezembro a Netscape e a Sun publicaram o comunicado que abre este capítulo. LiveScript passou a se chamar JavaScript. Marc Andreessen, da Netscape, queria o calor da palavra Java, que naquele mês estava em toda palestra. O comunicado vendia os dois como par: Java para os programadores que criam componentes, JavaScript para quem escreve página e aplicação de empresa.

A marca ficou com a Sun. Hoje está com a Oracle, que comprou a Sun. Um padrão internacional não podia ser publicado com a marca de um fabricante. Esse detalhe de contrato decide o nome que o comitê usa até hoje.

O comunicado também falava em cliente e servidor. Não era figura de linguagem vazia. Em 1996 a Netscape vendeu o LiveWire, um servidor em que JavaScript rodava fora do navegador, ao lado da página. Esse produto teve vida curta. Não é o ancestral do Node. O Node, treze anos depois, nasce de outro motor e de outro jeito de tratar entrada e saída. O que o comunicado de 1995 documenta é a intenção comercial da Netscape naquela semana, mais um servidor que não permaneceu.

O Navigator 2.0 final saiu em março de 1996, já com o nome novo. Mocha tinha durado a primavera. LiveScript, o outono. JavaScript é o nome que sobrou, e é o nome que o Renato usou na quinta.

## 1.3 Dois implementadores, um padrão

Em agosto de 1996 a Microsoft publicou o JScript no Internet Explorer 3. Era a linguagem refeita por engenharia reversa, com outro nome, porque JavaScript era marca. A página passou a ter dois implementa-

dores que não se comprometiam a aceitar o mesmo texto. Quem escrevia para os dois navegadores testava nos dois, e mesmo assim encontrava diferença na árvore de objetos que representa a página — o DOM, Document Object Model — mais do que na linguagem em si.

A Netscape precisava de um padrão que a Microsoft aceitasse implementar. Um padrão com o nome JavaScript esbarrava na marca. Em novembro de 1996 a linguagem foi entregue à Ecma International. A primeira edição do ECMA-262 saiu em junho de 1997, editada por Guy Steele, com cerca de noventa e cinco páginas. A linguagem descrita no documento se chama ECMAScript. O comitê que a mantém é o TC39, o comitê técnico 39 da Ecma.

**ECMAScript** O nome da linguagem dentro do padrão ECMA-262. JavaScript continua sendo o nome comercial, usado pelo navegador, pelo Node e por quem escreve. São o mesmo idioma. Quando a edição importar, ela vem com o ano, ou com o número antigo: ES3, ES5, ES2015.

A segunda edição, em 1998, foi trabalho editorial. A terceira, em dezembro de 1999, é a ES3: expressões regulares, `try/catch` — um bloco que tenta e outro que recebe a falha — e um conjunto de operações de texto que a página já estava pedindo. Durante anos, “JavaScript” na prática queria dizer “o que a ES3 descreve e o que cada navegador acrescentou por conta própria”.

## 1.4 A edição 4, que não foi publicada

Logo depois da ES3 o comitê começou uma quarta edição muito maior. O rascunho, ao longo dos anos 2000, tinha classe, módulo e um sistema de tipos opcional. Era quase uma segunda linguagem, pensada para aplicação grande, alinhada ao que o Eich e parte do grupo queriam desde o fim dos anos 1990.

A Adobe implementou esse rascunho. O ActionScript 3, em 2006, dentro do Flash Player, tinha classe, pacote e tipo opcional tirados da edição 4 que ainda não existia como padrão. Quem escrevia Flash escrevia um JavaScript que o navegador, fora do plugin, não falava.

Do outro lado, Microsoft e Yahoo consideravam o rascunho grande demais para a página já instalada. Douglas Crockford, na Yahoo, sustentava que a web não aguentava uma língua nova incompatível com a que já estava nos sites. Chris Wilson, na Microsoft, estava no mesmo lado. O Eich, por um bom tempo, esteve no lado da edição 4.

Em agosto de 2008 o impasse fechou. O grupo anunciou que a edição 4, naquela forma, não seria publicada. O acordo recebeu o nome de Harmony. Primeiro sairia uma edição pequena em cima da ES3, sem quebrar o que já rodava. Depois, cada acréscimo entraria sem transformar a língua numa segunda língua. O ActionScript 3 ficou onde estava, dentro do Flash. O JavaScript da página não herdou aquele rascunho.

O que saiu do acordo, em dezembro de 2009, foi a ES5. Entrou o modo estrito: uma string especial no topo de um arquivo, `"use strict"`, e o motor passa a recusar alguns silêncios. Atribuir a um nome que ninguém criou, por exemplo, vira erro, em vez de criar um nome visível no programa inteiro. Entraram também operações de lista — percorrer, filtrar, transformar — e o JSON como parte da linguagem.

**JSON** JavaScript Object Notation. Um texto para levar dado de um programa a outro: chaves, nomes entre aspas duplas, listas entre colchetes. É um recorte da forma como a linguagem já escrevia um objeto em 1995. Douglas Crockford isolou esse recorte por volta de 2001 e publicou as regras. Em 2009 a ES5 colocou na linguagem as funções que leem esse texto e que o escrevem. Sistemas que não são JavaScript passaram a usá-lo como formato neutro.

## 1.5 O que a página fez enquanto o padrão brigava

A ES3 ficou sozinha de 1999 até 2009. A página não ficou.

O Internet Explorer 5, ainda em 1999, tinha um objeto capaz de pedir um dado ao servidor sem recarregar a página. O nome que grudou nesse jeito de construir tela veio depois, num ensaio de Jesse James Garrett, em 18 de fevereiro de 2005: Ajax. A página deixou de ser só documento. Passou a ser um programa que conversa com um servidor e reescreve um pedaço de si.

Os objetos com que se mexe na página, porém, não eram os mesmos no Navigator e no Internet Explorer. Em 14 de janeiro de 2006, num BarCamp em Nova York, John Resig mostrou o jQuery: uma biblioteca que oferecia uma escrita só para essas diferenças. A versão 1.0 saiu em agosto daquele ano. Durante um tempo longo, aprender a fazer página dinâmica era aprender jQuery. A linguagem e a biblioteca dividiam o mesmo arquivo, e muita gente só via a biblioteca.

Em setembro de 2008, no mesmo ano em que o Harmony foi anunciado, o Google publicou o Chrome. Dentro dele vinha um motor novo,

o V8, escrito na Dinamarca por um time liderado por Lars Bak. Bak tinha trabalhado no HotSpot, o motor de Java da Sun. O V8 observa, enquanto o programa roda, que tipos de fato passam por cada função, e compila de novo as funções que mais rodam. JavaScript grande na página deixou de ser, por princípio, um programa lento.

**V8** Motor de JavaScript do Google, publicado em 2008 com o Chrome. É o motor que o Node usa. O Chrome e o Node podem carregar versões diferentes desse motor. Os dois executam a linguagem. Os dois não são obrigados a aceitar exatamente o mesmo texto, se as versões divergirem.

## 1.6 Fora do navegador, de novo

Em maio de 2009 Ryan Dahl publicou a primeira versão do Node. A apresentação que ficou conhecida é de novembro, na JSConf, em Berlim. Ele queria um servidor capaz de segurar muitas conexões abertas ao mesmo tempo. No modelo antigo, cada conexão ocupava uma linha de execução e ficava parada esperando a rede. Dahl olhou para o navegador: ali o JavaScript já vivia num laço de eventos. O programa entrega uma função, segue, e a função roda quando a resposta chega. Uma espera de rede devolve o controle ao laço. Uma conta longa, feita no meio, segura o clique até terminar.

Ele tirou o V8 do Chrome e pôs em volta as peças que um navegador não tem: arquivo, porta, processo. A escolha da linguagem teve um motivo técnico registrado por ele: JavaScript, fora do navegador, ainda não tinha uma biblioteca padrão de entrada e saída. Não havia um estoque de funções que parassem o programa à espera de disco ou de rede. Dava para construir esse estoque inteiro em cima do laço de eventos, sem misturar com outro estilo.

**Laço de eventos** A fila em que o motor registra “quando isto acontecer, chame esta função” e, no meio-tempo, executa o que já pode ser executado. É o desenho do navegador desde que função virou valor, e é o desenho que o Node adotou em 2009 para o servidor. O painel da Borba espera rede a maior parte do tempo. É esse mecanismo que segura a espera.

Em 2010 Isaac Schlueter publicou o npm, o comando e o registro com que um projeto Node declara as bibliotecas de que precisa e as instala. O registro cresceu até virar o lugar padrão de publicar JavaScript. Em 22 de março de 2016 um pacote de onze linhas, o `left-pad`, foi removido pelo autor, Azer Koçulu, no meio de uma disputa de nome com uma empresa. A cadeia de instalação de milhares de projetos passava por

aquelas onze linhas. Os builds quebraram no mesmo dia. O registro voltou a publicar o pacote com outro responsável.

## 1.7 A sexta edição, e o calendário

O Harmony tinha ficado a promessa de crescer sem quebrar. A sexta edição do ECMA-262 saiu em junho de 2015. O nome oficial é ES2015. Muita gente ainda diz ES6, pelo número da edição.

O que entrou ali vinha direto das brigas anteriores. `let` e `const` criam nome visível no bloco, não na função inteira: uma correção do alcance que `var` tinha desde 1995. Módulo entra no padrão porque já existiam dois sistemas caseiros. O Node usava `require`, função que puxa outro arquivo e devolve o que ele exporta. No navegador, bibliotecas como o RequireJS usavam outro formato, o AMD, para carregar arquivo pela rede. A edição de 2015 definiu `import` e `export`. Os arquivos antigos não foram reescritos no dia da publicação. Os dois jeitos continuaram no mundo.

Classe entrou como escrita. O mecanismo continuou sendo o protótipo de 1995. A palavra `class` organiza num bloco só o que antes se escrevia como função construtora mais um objeto de protótipo. É açúcar de sintaxe: uma grafia mais curta para uma estrutura que o motor já executava. A edição 4 queria classe de verdade, com tipo. A edição de 2015 publicou a grafia, e recusou a segunda língua.

Promessa entrou como objeto que representa um resultado que ainda não chegou. O texto entre crases, que interpola valor, também. Nenhuma dessas formas apagou a anterior. Edição nova, nesse padrão, acumula.

Os navegadores implementaram a edição aos poucos. Em setembro de 2014 Sebastian McKenzie tinha começado um tradutor, o 6to5, rebatizado de Babel em 2015. Ele lia o texto da ES2015 e escrevia texto que a ES5 já executava. Quem publicava página passou a ter um passo de tradução no meio do caminho. O passo existiu porque o padrão e os navegadores não andavam no mesmo mês. Muitos projetos mantiveram o passo depois que os navegadores alcançaram a edição, porque o caminho de build já estava montado em volta dele.

Enquanto a sexta edição saía, o Node vivia outra fusão. No fim de 2014, parte dos mantenedores discordou da Joyent, a empresa que guardava o projeto, sobre ritmo e governança. Em janeiro de 2015

publicaram um fork, o io.js, com versão própria e comitê aberto. Em setembro de 2015 os dois linhas se encontraram de novo no Node 4. A partir dessa reunião, as linhas pares ganharam prazo público de manutenção. O nome desse prazo é LTS.

**LTS** *Long Term Support*. Linha do Node que recebe correção de segurança por um período combinado, com data de término publicada antes. A linha chamada Current é a mais nova, ainda sem esse prazo, e muda seguido. Servidor de operação fica numa LTS.

Desde 2016 o TC39 publica uma edição por ano, em geral em junho. Uma proposta atravessa estágios públicos. O que chega ao último estágio entra na edição daquele ano. `async` e `await` entraram na de 2017: duas palavras para escrever, na ordem da leitura, uma função que espera uma promessa. O comitê deixou de guardar mudança durante uma década para soltar uma edição gigante. A edição gigante tinha sido a 4, e ela não foi publicada.

O Node 20, o do servidor da Borba, foi publicado em abril de 2023. A data de fim de correção, no calendário do projeto, é 30 de abril de 2026. O Node 24 foi publicado em maio de 2025, entrou em LTS em outubro desse ano, e recebe correção até 30 de abril de 2028. A Cláudia perguntou “qual versão?” na quinta. A resposta do servidor é essa linha, e essa data.



**FIGURA 1.1** A linguagem, do protótipo de maio de 1995 ao calendário que o servidor da Borba ainda obedece.

## RESUMO

- Maio de 1995: Eich escreve o Mocha na Netscape. Função é valor. Objeto delega a um protótipo. Classe, no sentido de Java, não está nesse desenho.
- 4 de dezembro de 1995: a Sun e a Netscape publicam o nome JavaScript. A marca fica com a Sun, hoje Oracle. Por isso o padrão, em 1997, se chama ECMAScript.
- A edição 4 não foi publicada. Em agosto de 2008 o acordo Harmony troca a segunda língua por edições que não quebram a página. A ES5 sai em dezembro de 2009.
- O Node, em maio de 2009, tira o V8 do Chrome e adota o laço de eventos do navegador. O prazo LTS nasce em 2015, na reunião com o io.js.
- A ES2015, em junho de 2015, acrescenta grafia — `let`, `const`, módulo, classe, promessa — em cima do núcleo de 1995. O Node 20 da Borba recebe correção até 30 de abril de 2026.

- 1** O comunicado de 4 de dezembro de 1995 anuncia JavaScript junto com Java. Escreva, em três linhas, quem publicou o texto, para que servia o nome novo, e qual modelo de objeto o Mocha já tinha que não é o modelo de classe do Java.
- 2** Por que o padrão da linguagem não se chama JavaScript? Reconstitua os fatos de agosto de 1996 e de junho de 1997.
- 3** A Cláudia anotou num ticket — o registro numerado de uma tarefa — o texto “atualizar o Node do servidor esta semana, para a versão mais nova que existir”. A semana é a de 16 de março de 2026. Você aceitaria o texto do ticket? Justifique em três linhas, com o calendário que o projeto do Node publica desde 2015.



## CAPÍTULO

# 2

## O que vamos construir

*Segunda-feira, 8h17. Três jeitos de pedir a mesma entrega, e um histórico que sabe o nome de quem tentou.*

---

### AO FIM DESTES CAPÍTULOS

Ler o package.json e um git log o bastante para saber o que há na pasta, descrever o que o dia 6 de maio cobra do painel, e terminar com o Node 24 instalado e conferido.

**ENQUANTO ISSO, NA EMPRESA** Não mexe nisso

Bia chegou mais cedo.

O repositório já estava aberto. Ela tinha recebido acesso na sexta e passado o fim de semana tentando entender o projeto.

Abriu o `package.json`.

Leu as dependências.

Rolou.

Continuou rolando.

Parou.

— Rafael?

— Hm?

— Por que temos três bibliotecas para fazer requisição HTTP?

Rafael continuou olhando para o monitor.

— Temos?

— Axios, uma chamada direta de `fetch` e uma pasta chamada `http`.

— Ah.

— “Ah” o quê?

— Não mexe nisso.

— Por quê?

— Porque ninguém sabe.

Bia ficou olhando para ele.

— Mas alguém colocou.

— Sim.

— Quem?

Rafael apontou para o histórico do Git.

— Descubra.

---



**FIGURA 2.1** Por que temos três bibliotecas para fazer requisição HTTP?

O repositório é a pasta do painel com a história de cada mudança guardada pelo Git. O histórico é a lista dessas mudanças. A Bia acabou de descobrir que a pasta pede a mesma coisa de três jeitos, e que a sala não tem uma frase pronta para dizer qual dos três vale.

Dá para descobrir quem colocou. Não dá, só com essa conversa, para descobrir qual chamada é segura. As duas perguntas não são a mesma.

## 2.1 O que o `package.json` está contando

O arquivo é JSON: chaves, e nomes entre aspas duplas. O nome do projeto e a lista do que ele precisa para funcionar.

### PACKAGE.JSON

json

```

1  {
2    "name": "painel",
3    "dependencies": {
4      "axios": "0.21.1"
5    }
6  }
```

`name` é o nome da pasta como projeto. `dependencies` é o mapa das bibliotecas que alguém instalou e quis registrar. Cada nome aponta para um número de versão. `"axios": "0.21.1"` quer dizer: esta pasta foi montada com o Axios na versão 0.21.1.

O Axios não vem com a linguagem. É uma biblioteca de pedido HTTP publicada no registro npm. A versão anotada é a de um tutorial que já era antigo quando o arquivo foi criado, e o arquivo não se atualizou sozinho.

`fetch` não aparece nesse mapa. Não é biblioteca: é uma função que o navegador e o Node, a partir da linha 18, já trazem. O `lista.js` chama essa função. A pasta `http/` também não é dependência de ninguém. É código da própria Borba, um cliente escrito à mão.

**HTTP** O protocolo com que um programa pede um dado a outro computador. Uma requisição sai, uma resposta volta. Axios, `fetch` e a pasta `http/` fazem essa requisição. Três textos, um protocolo.

**npm** O registro público de bibliotecas JavaScript, e também o comando que instala uma delas a partir do que o `package.json` anotou. Vem junto com o Node. Não há nada para instalar agora. A lista acima é a da produção, e a instrução da manhã foi não mexer nela.

Três clientes não são três opiniões iguais. São três épocas encostadas:

| ONDE ESTÁ                                                                | O QUE É                       | QUANDO ENTROU  |
|--------------------------------------------------------------------------|-------------------------------|----------------|
| <code>http/client.js</code>                                              | código da casa, escrito à mão | março de 2022  |
| Axios em <code>entrega.js</code>                                         | biblioteca, versão 0.21.1     | março de 2024  |
| <code>fetch</code> em <code>lista.js</code> e em <code>entrega.js</code> | função da própria linguagem   | agosto de 2025 |

**TABELA 2.1** A pasta guarda as três. A manhã da Lívia usa o que estiver sendo chamado no caminho que a tela percorre hoje — e isso não está escrito no `package.json`.

## 2.2 O histórico que ele apontou

Git guarda cada mudança registrada do repositório. Cada registro é um **commit**: um conjunto de alterações com autor, data e uma mensagem que o autor escreveu na hora. `git log` imprime esses registros, do mais novo para o mais antigo.

O comando que lista esse histórico, só no arquivo da função e só nos três registros mais recentes que o tocaram, produz isto:

```

1  $ git log -3 -- entrega.js
2  commit 91ab220
3  Author: Marcos Teixeira
4  Date:   Tue Aug 12 18:03:44 2025 -0300
5
6      troca axios por fetch
7
8  commit 4b21877
9  Author: Diego Pacheco
10 Date:   Mon Mar 18 19:41:12 2024 -0300
11
12     ajustes
13
14 commit c77e014
15 Author: Diego Pacheco
16 Date:   Mon Mar 14 22:16:05 2022 -0300
17
18     cliente http

```

-3 pede só três registros. -- entrega.js pede só os que mexeram nesse arquivo. O -- separa as opções do nome do arquivo, para o Git não tentar ler entrega.js como se fosse mais uma opção.

Cada bloco tem quatro partes. commit e o código curto identificam o registro: aquele código é um apelido de um número grande que o Git calculou a partir do conteúdo. Author é quem registrou. Date é quando. O texto recuado embaixo é a mensagem.

A mensagem do Marcos diz que ele trocou Axios por fetch. O commit mexeu em entrega.js e em lista.js: O fetch ficou nos dois. A pasta ainda tem Axios, e ainda tem http/. A mensagem descreve a intenção. A pasta descreve o que sobrou. Ninguém voltou num registro posterior para escrever que a troca tinha ficado pela metade.

O commit de março de 2024, mensagem ajustes, é o que deixou no fim de entrega.js o comentário que pede para não mexer. A mensagem não diz isso. O arquivo diz.

**Git** Programa que guarda o histórico de uma pasta: o que mudou, quem mudou, quando, e a mensagem escrita na hora. O git log lê esse histórico. Não altera o arquivo. Quem colocou está no Author. O que a pessoa achou que estava fazendo está na mensagem. O que ela fez está no arquivo.

## 2.3 O que o dia 6 de maio está cobrando

Não é uma plataforma. Não são três caixas no quadro. São duas frases que a Livia consegue conferir sem ajuda.

A primeira: cada status que o motorista marca aparece no painel em até trinta segundos. A segunda: a lista que está na tela às 5h35 é a mesma lista que sai na impressora. Se as duas forem verdade na quarta-feira, 6 de maio, a cláusula está de pé. O resto é caminho.

---

#### ENQUANTO ISSO, NA EMPRESA    Uns dois mil e quatrocentos

A Cláudia chegou às 11h40 com a folha da Livia.

— Ela contou. Uns dois mil e quatrocentos por dia, fora da janela.

O Marcos já estava no quadro. Três caixas: frete, status, notificação.

— A gente separa. Cada caixa vira um serviço.

O Rafael olhou para a folha.

— Hoje é um arquivo. A cláusula é quarta, 6 de maio.

— E se a gente não separar?

— Aí a tela diz a verdade de manhã. É isso que os R\$ 3,50 estão cobrando.

---

Serviço, na frase do Marcos, seria outro programa, rodando à parte, só para aquela caixa. Dá para chegar lá. Não é o que a folha pede para esta quarta-feira. A folha pede que o número que a Livia contou caia, e que o papel pare de ganhar do painel às 5h40.

O programa que você vai escrever cabe numa pasta sua, chamada `borba`. O painel de produção continua no servidor, intocado. O que cresce na sua máquina é a versão que dá para rodar, quebrar e mostrar para alguém sem pedir licença ao caminhão.

## 2.4 Instalando

Baixe o Node em [<https://nodejs.org>](https://nodejs.org). Escolha a linha **LTS**. Nesta edição, essa linha é a 24, com correção publicada até 30 de abril de 2028. A 22 também roda o que está neste livro, e recebe correção até 30 de abril de 2027. A linha Current — em 2026, a 26 — muda toda semana. Não é a linha da sua pasta.

No Windows, o instalador pergunta se inclui o Node no PATH. Deixe marcado. PATH é a lista de pastas em que o terminal procura um comando. Sem essa marca, `node` não é encontrado, mesmo com o programa instalado.

Feche o terminal se ele já estava aberto, abra de novo, e confira:

```
$ node --version
v24.9.0
```

O trecho depois do segundo ponto muda conforme o mês da instalação. Qualquer resposta que comece com `v24` está na linha. Se o comando não existir, o terminal não viu a instalação: feche, abra, tente outra vez antes de instalar por cima.

### ▲ O TERMINAL QUE JÁ ESTAVA ABERTO

Um terminal guarda o PATH do momento em que foi aberto. Instalar o Node não atualiza essa cópia. O sintoma é `node` não reconhecido numa janela, e reconhecido na janela ao lado.

### 🔗 ERRO COMUM

`node` pode existir duas vezes na mesma máquina, com versões diferentes. O servidor da Borba responde 20. A sua máquina, se você seguiu a LTS, responde 24. Os dois se chamam Node.

No Windows, `where.exe node` lista os caminhos. No macOS e no Linux, `which node` mostra qual arquivo o terminal vai chamar. Descubra o seu agora e use esse. “Eu instalei” com o terminal chamando outro arquivo é a manhã inteira perdida num erro que não está no seu código.

### ▶ NA PRÁTICA

Rode `node --version`. Se a resposta começar com `v24`, crie a pasta do trabalho e entre nela:

```
$ mkdir borba
$ cd borba
```

Não rode `npm install`. Não copie o `package.json` da Borba. A pasta `borba` começa vazia de propósito: ainda não existe um programa.

### RESUMO

- A pasta de produção tem três clientes HTTP: Axios 0.21.1, `fetch` e `http/client.js`.
- `git log` mostra autor, data e mensagem. A mensagem do Marcos não é a pasta que sobrou.
- O dia 6 de maio cobra trinta segundos e uma lista igual à do papel.
- O Node da sua máquina é o 24, conferido com `node --version`. O do servidor é o 20, até 30 de abril.

**■ O PROJETO AGORA**

Você tem o Node 24 instalado, a pasta `borba` criada, e as duas frases que a Lívia vai conferir em 6 de maio. Ainda não há programa. É daqui que o arquivo começa.

- 1** A Lívia liga às 5h50: “sumiu a entrega do mercado da Vila. No papel está.” Escreva, em duas linhas, o que ela precisa que o painel responda, e o que ela não precisa saber para receber essa resposta.
- 2** Sem olhar de novo o bloco do `git log`, diga quem registrou mudança em `entrega.js` em agosto de 2025 e o que a mensagem afirma. Em seguida diga uma coisa que a mensagem não prova.
- 3** O Marcos quer, esta semana, três programas separados: um só para o frete, um só para o status, um só para a notificação. A folha da Lívia está em cima da mesa. Você aceitaria? Justifique em três linhas.





## CAPÍTULO

# 3

## O primeiro programa

*Dois Nodes na mesma manhã, e só um deles  
pode atender o cais.*

---

### AO FIM DESTE CAPÍTULO

Escrever, salvar e rodar um programa JavaScript no Node; escrever texto com crase; prender um nome com `const` e reatribuir com `let`; ler as três falhas que aparecem na primeira hora.

**ENQUANTO ISSO, NA EMPRESA Qual máquina?**

A Bia abriu o terminal na pasta dela.

```
$ node --version  
v24.9.0
```

No servidor, a mesma pergunta respondia 20.

— Aqui está rodando — disse ela.

O Neto apareceu na porta. Ninguém tinha chamado.

— Qual máquina?

— A minha.

— E a do servidor?

O Rafael, sem levantar os olhos:

— Vinte. Até 30 de abril.

O Neto olhou para o número na tela dela.

— Não mexe no vinte. O da manhã depende dele. O seu fica do lado, e antes de rodar qualquer coisa você descobre qual dos dois o terminal está chamando.

Rodar JavaScript parece a parte mais simples do trabalho, e é onde se perde a manhã: o `node` que responde no terminal quase nunca é o único da máquina. O programa desta página roda no seu, o 24. O painel de produção continua no 20, e ninguém desta página vai publicá-lo hoje.

O menor programa útil tem uma linha. Salve isto num arquivo chamado `status.js`, dentro da pasta `borba`:

```
STATUS.JS javascript  
1 console.log("Entrega E-4821: a caminho");
```

Abra o terminal nessa pasta e rode:

```
$ node status.js  
Entrega E-4821: a caminho
```

`console.log` escreve o valor no terminal e pula para a linha seguinte. O que está entre aspas é texto, e o texto sai do jeito que foi escrito. O ponto e vírgula encerra a instrução. Você vai ver arquivo da Borba sem ele: os dois costumam rodar. Este livro escreve o ponto e vírgula para o fim da instrução ficar visível.

O Node leu o arquivo, executou a linha e terminou. Não há um segundo comando para “compilar” antes. A compilação existe, dentro do motor, e você não a dispara.

**V8** O motor, escrito pelo Google, que executa o JavaScript dentro do Chrome e dentro do Node. `node status.js` entrega o seu arquivo a esse motor, mais às peças de arquivo e terminal que o Node acrescenta. A versão do motor no seu Node 24 não é a versão do motor no Node 20 do servidor. O arquivo desta página cabe nas duas.

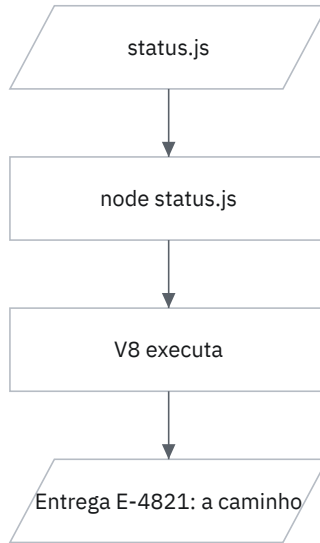


**FIGURA 3.1** Do arquivo à linha no terminal. Você dispara um comando. A compilação fica dentro do motor.

### 3.1 O mesmo texto, no navegador

O painel da Lívia não roda com `node status.js`. Roda no navegador, que tem o próprio motor. O Chrome usa o V8. O Firefox usa outro, o SpiderMonkey. A linguagem é a mesma; o programa ao redor, não.

Abra o navegador, aperte F12 e escolha o painel **Console**. F12 abre as ferramentas de desenvolvedor: o lugar em que o navegador mostra o que a página está fazendo. No console, digite a mesma chamada e confirme com Enter:

```
console.log("Entrega E-4821: a caminho");
```

A linha aparece de novo. Você não salvou arquivo. O console executa o que você digita e esquece quando a aba fecha. Serve para perguntar. Não serve para ser o programa.

**Console do navegador** Uma linha de comando dentro das ferramentas de desenvolvedor. Executa JavaScript na página que está aberta. Uma página em branco não é o painel da Borba. O que você digitou aqui é seu. Não consultou entrega nenhuma no servidor.

Quando o programa precisa ficar, o caminho é arquivo salvo e `node .` O console volta quando a pergunta for sobre uma página.

## 3.2 Um nome para a situação

O código da entrega e a situação estão cravados no texto. No dia em que a situação mudar, você edita a frase inteira e reza para não errar o trecho que era fixo. Um nome separa as duas coisas.

**STATUS.JS**

javascript

```
1  const codigo = "E-4821";
2  const situacao = "a caminho";
3
4  console.log(`Entrega ${codigo}: ${situacao}`);
```

```
$ node status.js
Entrega E-4821: a caminho
```

A saída é a mesma. O arquivo, não. `codigo` e `situacao` são nomes presos a textos. A linha do `console.log` usa crase, não aspas. Entre crases, o que está em `{ }` é lido como código, e o resultado entra no texto.

### AS PARTES DE UM TEXTO COM CRASE

```
console.log(`Entrega ${codigo}: ${situacao}`);
```

1 2 3

- 1 A crase liga a interpolação. Com aspas, `${codigo}` sairia escrito assim, com o cifrão.
- 2 `${codigo}` troca o nome pelo valor. Dentro das chaves vale uma expressão.
- 3 O que está fora das chaves é texto fixo, inclusive os dois-pontos e o espaço.

**const** Cria um nome e o prende a um valor. Prender quer dizer que o nome não aceita outra atribuição em seguida. `situacao = "entregue"` na linha de baixo falha. O valor `"a caminho"` continua sendo o valor daquele nome até o programa terminar. A falha, gravada no Node v24.9.0, começa assim. O número das linhas internas muda de uma versão para outra; o tipo da falha, não.

```
troca.js:2
situacao = "entregue";
          ^
```

TypeError: Assignment to constant variable.

`TypeError` é o tipo. `Assignment to constant variable` é o motivo: atribuição a variável constante. O cursor `^` está no `=`, que é o ponto em que o nome recusou o valor novo. A linha `troca.js:2` diz arquivo e número da linha.

Quando o nome precisa mudar de valor, o jeito é `let`.

**ANDA.JS**

javascript

```
1 let situacao = "a caminho";
2 situacao = "entregue";
3 console.log(situacao);
```

```
$ node anda.js
entregue
```

**let** Cria um nome que aceita outra atribuição. A segunda linha aponta `situacao` para outro texto. A primeira ligação deixa de valer. Use `let` quando o valor muda. No resto, `const`. Um nome que não muda é mais fácil de ler às 5h40 do que um nome que talvez mude.

### 3.3 Três falhas da primeira hora

As mensagens abaixo foram gravadas no Node v24.9.0. Está só o começo de cada uma: arquivo, cursor e tipo. Debaixo disso o Node imprime o caminho interno até a sua linha. Esse caminho repete de uma falha para outra. A parte que muda, e a que se lê primeiro, é a de cima.

**O arquivo não está onde você chamou.**

```
Error: Cannot find module 'D:\borba\status.js'
code: 'MODULE_NOT_FOUND'
```

`MODULE_NOT_FOUND` quer dizer que o Node procurou um arquivo e não achou. O caminho é o lugar em que ele procurou. Quase sempre o terminal está em outra pasta, ou o arquivo se chama `status.js.txt` porque o editor acrescentou a extensão por conta própria. Confira com o nome que está na pasta, não com o nome que você lembra de ter digitado.

A palavra `module`, na mensagem, é o nome interno: o Node procura o seu arquivo como se procura um módulo. O caminho ao lado é o lugar em que ele procurou. A linha de cima basta para achar o problema.

**Faltou fechar.**

**QUEBRA.JS**

javascript

```
1 console.log("Entrega E-4821: a caminho")
```

```
quebra.js:1  
console.log("Entrega E-4821: a caminho"  
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^  
  
SyntaxError: missing ) after argument list
```

`SyntaxError` é texto que nem chegou a rodar: a gramática não fecha. `missing ) after argument list` aponta o parêntese que falta. O cursor cobre o argumento porque foi ali que o Node desistiu de esperar o `)`. Falta também o ponto e vírgula, e o erro não fala nele. Ele para no primeiro furo.

O nome não existe.

```
NOME.JS javascript
1 console.log(situacao);
```

```
nome.js:1
console.log(situacao)
           ^
ReferenceError: situacao is not defined
```

`ReferenceError` é um nome usado sem ter sido criado. `situacao` não foi escrito com `const` nem com `let` neste arquivo. O console do navegador às vezes ainda tem um nome deixado por uma digitação anterior. O `node nome.js` começa do zero toda vez. Se o nome não está no arquivo, ele não existe.

## ERRO COMUM

Aspas e crase não são intercambiáveis. "Entrega `{codigo}`" imprime o cifrão e as chaves. A interpolação só liga com crase. E o nome dentro de `{ }` precisa existir. `{situacao}` com `situacao` nunca criado é o mesmo `ReferenceError`, só que no meio da frase.

### 3.4 Comentário

Tem linha que o programa não deve executar, e que uma pessoa precisa ler.

```
STATUS.JS javascript
```

```
1 // situação como o motorista marcou, não como a tela gostaria
2 const situacao = "a caminho";
3 console.log(situacao);
```

Tudo depois de `//`, na mesma linha, é comentário. O Node ignora. O comentário não explica o que a linha já diz. Explica o que a linha não consegue carregar: de onde vem o valor, ou por que ele não é outro.

O comentário no fim de `entrega.js`, `// NÃO MEXER. FUNCIONA.`, é desse gênero. Não descreve a função. Descreve uma decisão de março de 2024: não mexer. O medo que justificava a decisão não está na linha.

### RESUMO

- `node status.js` lê o arquivo, executa e termina. A compilação fica dentro do V8.
- Texto entre crases interpola o que estiver em `${ }`. Aspas não interpolam.
- `const` prende o nome. `let` deixa reatribuir. Reatribuir `const` é `TypeError`.
- Arquivo ausente, parêntese faltando e nome inexistente são três falhas diferentes: `MODULE_NOT_FOUND`, `SyntaxError`, `ReferenceError`.

**1** Na pasta `borba`, altere `status.js` para a saída sair exatamente assim:

```
Entrega E-4821: a caminho (Ribeirão Preto)
```

A cidade deve ser um `const`, não um pedaço cravado na frase.

**2** O arquivo abaixo vai falhar. Diga o tipo da falha e a linha, sem rodar.  
 .. Depois rode e confira.

| STATUS.JS                                                                           | javascript |
|-------------------------------------------------------------------------------------|------------|
| <pre>const codigo = "E-4821" console.log(`Entrega \${codigo}: \${situacao}`);</pre> |            |

**3** A Bia precisa que `situacao` comece em `"a caminho"` e passe a `"entregue"` antes do `console.log`. Ela escreveu as duas atribuições com `const` e recebeu `TypeError`. Mostre o arquivo que imprime `entregue`, e diga por que a primeira linha não pode continuar sendo `const`.



PARTE 2

# A linguagem que o painel já fala

*Tipos, contas, decisões e listas. O lugar em que um valor trocado muda o caminhão que sai.*

## NESTA PARTE

---

- 4 Variáveis e tipos
- 5 Operadores
- 6 Condicionais
- 7 Repetições



## CAPÍTULO

# 4

# Variáveis e tipos

*O campo de pendências chegou vazio. O painel comparou vazio com zero, achou igual, e liberou a entrega para sair sem a assinatura.*

---

### AO FIM DESTES CAPÍTULOS

Distinguir os tipos que o painel usa todo dia, perguntar o tipo com `typeof`, parar uma conversão automática antes que ela vire regra de negócio, e guardar dinheiro em centavos inteiros.

—

**Capítulo disponível na  
versão completa**

—





## CAPÍTULO

# 5

# Operadores

*Frete de 10 mais adicional de 20. A tela da Livia mostrava R\$ 1020.*

---

### AO FIM DESTES CAPÍTULO

Fazer conta sem colar texto sem querer, comparar sem que a ordem alfabética decida por você, combinar condições com `&&` e `||` sabendo o que eles devolvem, e ler a precedência no trecho em que ela muda o resultado.

—

**Capítulo disponível na  
versão completa**

—





## CAPÍTULO

# 6

## Condicionais

*A entrega E-5012 aparecia como entregue e como cancelada. Dois ifs rodaram, e nenhum deles sabia do outro.*

---

### AO FIM DESTES CAPÍTULO

Escrever decisões com `if`, `else if` e `else`; usar `switch` só quando o status é de fato um conjunto fechado; sair cedo de uma função; e recusar um par de estados que não pode existir.

—

**Capítulo disponível na  
versão completa**

—





## CAPÍTULO

# 7

## Repetições

*O papel tinha quatro entregas para Franca. O painel mostrava três. O laço pulava a que tinha a cidade em branco.*

---

### AO FIM DESTES CAPÍTULOS

Percorrer a lista com for...of, saber o que o for clássico acrescenta, somar e contar com um acumulador, e ver um continue pular a entrega que não devia ser pulada.

—

**Capítulo disponível na  
versão completa**

—





PARTE 3

# Funções, nomes e dados tortos

*A função de 187 linhas, o nome que vaza do bloco, e a entrega que cada sistema descreve de um jeito.*

## NESTA PARTE

---

- 8 Funções
- 9 Escopo
- 10 Arrays e objetos
- 11 Desestruturação e spread



## CAPÍTULO

# 8

## Funções

*Cento e oitenta e sete linhas, oito responsabilidades, e um comentário na última pedindo para ninguém mexer. O Rafael deixou tirar uma.*

---

### AO FIM DESTES CAPÍTULOS

Declarar uma função, passar argumentos, devolver um valor, e tirar uma responsabilidade de processarEntrega com nome e saída visível — sem limpar as outras cento e oitenta linhas no mesmo dia.

—

**Capítulo disponível na  
versão completa**

—





## CAPÍTULO

# 9

## Escopo

*A mensagem ‘sua entrega chegou’ foi para o cliente errado. O nome da entrega do laço continuava existindo depois do laço, com a última entrega dentro.*

---

### AO FIM DESTES CAPÍTULOS

Prever onde um nome existe — no bloco, na função, no arquivo —, entender o que var faz de diferente de let e const, e reproduzir o nome que vaza do laço e leva a notificação para o cliente errado.

—

**Capítulo disponível na  
versão completa**

—





## CAPÍTULO

# 10

## Arrays e objetos

*O aplicativo do motorista manda status. O sistema antigo manda situacao. As duas estão certas, cada uma no sistema de onde veio.*

---

### AO FIM DESTE CAPÍTULO

Modelar uma entrega como objeto e a manhã como array, ler e escrever propriedades sem ser enganado por um nome que não existe, percorrer e transformar a lista com os métodos de array, e levar dois formatos da mesma entrega para um só.

—

**Capítulo disponível na  
versão completa**

—





## CAPÍTULO

# 11

## Desestruturação e spread

*A tela de edição mudou o endereço só para mostrar. A lista da manhã, que guardava o mesmo objeto, mudou junto.*

---

### AO FIM DESTES CAPÍTULOS

Ler um objeto por desestruturação, receber parâmetros com nome, copiar com spread sem achar que copiou o que está aninhado, e reproduzir a mutação compartilhada que mudou a lista sem ninguém mandar.

—

**Capítulo disponível na  
versão completa**

—





PARTE 4

# O tempo, que não espera

*Callback, promessa, async/await e closure. O programa deixa de ser uma lista de passos e passa a ser várias coisas no meio do caminho.*

## NESTA PARTE

---

- 12 Callbacks
- 13 Promises
- 14 async/await
- 15 Closures



## CAPÍTULO

# 12

# Callbacks

*Gravar o status chama quem notifica, que chama quem atualiza a tela. Quando a notificação falhou, a tela ficou esperando. E ninguém soube.*

---

### AO FIM DESTES CAPÍTULOS

Passar uma função para outra, entender que o código que espera não para o programa, ler uma cadeia de três continuações na ordem em que elas de fato rodam, e não deixar o erro do meio sumir.

—

**Capítulo disponível na  
versão completa**

—





## CAPÍTULO

# 13

## Promises

*A tela mostrava o frete de antes do ajuste ao lado do status de depois. As duas respostas chegaram, cada uma na sua hora, e cada uma escreveu por cima da outra.*

---

### AO FIM DESTE CAPÍTULO

Criar e consumir uma Promise, encadear then e catch, explicar por que três pedidos terminam fora da ordem em que foram feitos, e juntar respostas que precisam aparecer juntas com Promise.all.

—

**Capítulo disponível na  
versão completa**

—





## CAPÍTULO

# 14

## async/await

*A atualização ficou legível. E o frete da E-5302 apareceu na tela como [object Promise]1000.*

---

### AO FIM DESTES CAPÍTULOS

Reescrever o fluxo do capítulo anterior com `async` e `await`, tratar erro com `try` e `catch`, apontar a linha em que o programa espera, e reconhecer pela saída o `await` que faltou.

**ENQUANTO ISSO, NA EMPRESA [object Promise]**

Quarta, 1º de abril. A Bia reescreveu a atualização da linha com `async` e `await` na terça à noite. De manhã, o Rafael leu sem comentar e aprovou.

Às 10h, o Paulo mandou um print:

```
E-5302 | a caminho | R$ [object Promise]1000
```

— É 1º de abril — disse a Bia.

— Não é — disse o Paulo. — É a E-5302. Urgente.

A Bia abriu a função. A linha era esta:

```
const frete = buscarFrete(e.codigo) + (e.urgente ? 1000 : 0);
```

— Falta o `await` — disse o Rafael.

— Mas o resto da função tem.

— O resto da função não é esta linha.

**14.1** `async` e `await`

O capítulo 13 escreveu passos em sequência com `then`. `async` e `await` escrevem a mesma coisa com cara de código de cima para baixo:

**LINHA.JS**

javascript

```
1 function buscar(valor, ms) {
2   return new Promise((r) => setTimeout(() => r(valor), ms));
3 }
4
5 async function atualizarLinha(codigo) {
6   console.log(`pedindo ${codigo}`);
7   const status = await buscar("entregue", 120);
8   const frete = await buscar(2340, 60);
9   console.log(`${codigo} | ${status} | ${frete}`);
10  return { status, frete };
11 }
12
13 atualizarLinha("E-5261");
14 console.log("a chamada devolveu na hora");
```

```
$ node linha.js
pedindo E-5261
a chamada devolveu na hora
E-5261 | entregue | 2340
```

`async` antes de `function` faz duas coisas: permite usar `await` dentro dela, e faz a função **sempre devolver uma promessa**. O `return` do fim cumpre essa promessa.

`await` antes de uma promessa **pausa esta função** até a promessa terminar, e entrega o valor dela. `status` recebe "entregue", não uma promessa.

A palavra que importa é *esta*. O `await` não para o programa. A função `atualizarLinha` fica suspensa na linha do `await` quem a chamou recebe uma promessa e segue — e é por isso que “a chamada devolveu na hora” sai antes da linha da entrega.

**await** Pausa a função `async` em que está até a promessa à direita terminar, e devolve o valor dela. Se a promessa for rejeitada, o `await` lança o erro, como um `throw` naquela linha. Fora de uma função `async` — num arquivo comum do Node —, `await` é `SyntaxError`.

## 14.2 try e catch

Com `await`, a promessa rejeitada vira um erro lançado na linha do `await`. E erro lançado se trata com o `try` que o capítulo 12 disse que não alcançava callbacks — porque agora a espera acontece **dentro** do bloco:

LINHA.JS

javascript

```

1  function falhar(mensagem, ms) {
2    return new Promise( (_, rejeitar) =>
3      setTimeout(() => rejeitar(new Error(mensagem)), ms),
4    );
5  }
6
7  async function marcarEntregue(codigo) {
8    try {
9      await buscar("gravado", 100);
10     console.log(`${codigo}: gravado`);
11     await falhar("WhatsApp: 503", 100);
12     console.log(`${codigo}: notificado`);
13   } catch (erro) {
14     console.log(`${codigo}: ${erro.message}`);
15   }
16   console.log(`${codigo}: tela atualizada`);
17 }
18
19 marcarEntregue("E-5230");

```

```

$ node linha.js
E-5230: gravado
E-5230: WhatsApp: 503
E-5230: tela atualizada

```

A linha depois do `await` que falhou não roda; a execução pula para o `catch`, e depois segue. A estrutura é a de qualquer código com erro — sem a convenção do erro primeiro, sem `catch` pendurado no fim de uma cadeia.

## 14.3 O mesmo fluxo do capítulo anterior

A atualização com rodadas, que o capítulo 13 escreveu com `Promise.all` e `then`:

TELA.JS

javascript

```
1  const inicio = Date.now();
2  const t = () => `${String(Date.now() - inicio).padStart(4)} ms`;
3
4  function buscar(valor, ms) {
5    return new Promise((r) => setTimeout(() => r(valor), ms));
6  }
7
8  let rodadaAtual = 0;
9
10 async function atualizar(status, frete, msStatus, msFrete) {
11   rodadaAtual++;
12   const minha = rodadaAtual;
13
14   const [s, f] = await Promise.all([
15     buscar(status, msStatus),
16     buscar(frete, msFrete),
17   ]);
18
19   if (minha !== rodadaAtual) {
20     console.log(`${t()} rodada ${minha} descartada`);
21     return;
22   }
23   console.log(`${t()} rodada ${minha}: ${s} | ${f}`);
24 }
25
26 atualizar("a caminho", 1890, 120, 450);
27 setTimeout(() => atualizar("entregue", 2340, 120, 60), 200);
```

```
$ node tela.js
321 ms rodada 2: entregue | 2340
451 ms rodada 1 descartada
```

A saída é a mesma do capítulo anterior. O código diz a mesma coisa em outra ordem de leitura: o `await Promise.all(...)` é a linha em que a função espera, e tudo abaixo dela acontece depois das duas respostas.

## 14.4 Esperar em sequência sem precisar

`await` torna fácil escrever uma espera atrás da outra, mesmo quando as duas não dependem entre si:

```
const status = await buscar("entregue", 120);
const frete = await buscar(2340, 450);
```

Isso leva 570 ms: o pedido do frete só **sai** depois que o status voltou. Com `Promise.all`, os dois saem juntos e o tempo é o do mais lento, 450 ms. Na tela da manhã, com 160 entregas, a diferença entre esperar uma de cada vez e esperar todas juntas é a diferença entre um minuto e meio e um segundo.

A regra: `await` em sequência quando o segundo pedido **precisa** do resultado do primeiro. Quando não precisa, `Promise.all`.

## 14.5 O `await` que faltou

A linha da E-5302:

```
URGENTE.JS                                     javascript
1  async function buscarFrete(codigo) {
2      return 2340;
3  }
4
5  async function freteDaLinha(e) {
6      const frete = buscarFrete(e.codigo) + (e.urgente ? 1000 : 0);
7      return frete;
8  }
9
10 freteDaLinha({ codigo: "E-5302", urgente: true })
11   .then((f) => console.log(`R$ ${f}`));
```

```
$ node urgente.js
R$ [object Promise]1000
```

`buscarFrete` é `async`, e por isso devolve uma promessa — mesmo com um `return 2340` direto. Sem `await`, `frete` recebe a promessa. O `+` com um objeto de um lado transforma o objeto em texto — `"[object Promise]"` — e junta o `1000`. É o `+` que junta, do capítulo 5, com uma promessa no lugar do formulário.

Com o `await`:

```
const frete = (await buscarFrete(e.codigo)) + (e.urgente ? 1000 : 0);
```

R\$ 3340

Os parênteses em volta do `await` deixam claro o que é esperado antes da soma.

### ⚡ ERRO COMUM

O `await` que falta não dá erro. Dá um valor errado com cara de valor: um `[object Promise]` num texto, um `NaN` numa conta, um `if` (promessa) que é sempre verdadeiro — porque uma promessa é um objeto, e objeto é verdadeiro.

Quando uma saída mostra `[object Promise]`, ou uma condição com uma chamada `async` sempre dá o mesmo resultado, procure o `await` que falta na linha em que o valor nasceu.

### ● O ESSENCIAL

`async` faz a função devolver uma promessa, sempre. `await` pausa **só a função em que está**. Todo valor que vem de uma função `async` precisa de um `await` — ou de um `then` — antes de ser usado como valor.

### RESUMO

- `async` permite `await` e faz a função devolver uma promessa.
- `await` pausa a função até a promessa terminar e entrega o valor. O programa, fora dela, segue.
- Promessa rejeitada lança o erro na linha do `await` `try` e `catch` tratam como qualquer erro.
- `await` em sequência só quando o segundo precisa do primeiro. Senão, `await Promise.all`.
- `await` esquecido não quebra: produz `[object Promise]`, `NaN` ou uma condição sempre verdadeira.

## 1 Diga o que cada `console.log` imprime:

```
1  async function frete() {
2    return 1890;
3  }
4
5  console.log(frete());
6  frete().then((v) => console.log(v));
```

## 2 Reescreva com `async` e `await`, mantendo o comportamento:

```
..
```

```
1  gravarStatus("E-5230")
2    .then(() => atualizarLinha("E-5230"))
3    .then(() =>
4      notificarCliente("E-5230").catch((erro) => {
5        console.log(`aviso pendente: ${erro.message}`);
6      }),
7    )
8    .catch((erro) => console.log(`gravação falhou:
${erro.message}`));
```

### 3 A tela da manhã faz isto para 160 entregas:

...

```
for (const e of entregas) {
  e.status = await buscarStatus(e.codigo);
}
```

Cada pedido leva cerca de 300 ms. Diga quanto a lista demora, reescreva, e diga o que a versão nova precisa tratar que a antiga não precisava.



## CAPÍTULO

# 15

## Closures

*Cada botão de cancelar da lista cancelava a última entrega. Quem lia o código não via o número errado escrito em lugar nenhum.*

---

### AO FIM DESTES CAPÍTULO

Dizer qual variável uma função criada dentro de outra enxerga, usar isso de propósito para guardar estado, e reproduzir o laço que prende o último valor em todos os botões da lista.

—

**Capítulo disponível na  
versão completa**

—





## PARTE 5

# A tela

*O DOM, o bug que só aparece com a janela estreita, e as duas partes da interface que acreditam em coisas diferentes.*

### NESTA PARTE

---

- 16 DOM e eventos
- 17 Debugging
- 18 Estado



## CAPÍTULO

# 16

## DOM e eventos

*Um clique em cancelar, dois pedidos de cancelamento. O segundo chegava ao servidor depois que a entrega já tinha mudado.*

---

### AO FIM DESTES CAPÍTULOS

Achar um elemento da página, ler e escrever o texto dele, escutar um clique, e perceber quando o mesmo botão ganhou dois ouvintes — com a página aberta no navegador e o clique contado.

—

**Capítulo disponível na  
versão completa**

—





## CAPÍTULO

# 17

## Debugging

*Ticket 1847: a entrega some da lista quando a Livia cancela. Três pessoas tentaram reproduzir e não conseguiram. O Paulo diminuiu a janela.*

---

### AO FIM DESTE CAPÍTULO

Reproduzir um defeito antes de explicá-lo, usar as ferramentas de desenvolvedor do navegador — elementos, estilo calculado, console, ponto de parada —, e separar a causa do sintoma quando o sintoma aponta para o lugar errado.

**ENQUANTO ISSO, NA EMPRESA O bug impossível**

Quarta, 8 de abril. O ticket 1847 estava aberto havia seis dias.

*1847. Ao cancelar uma entrega, ela some da lista. Deveria continuar aparecendo, riscada, até o fim do dia. (Lívia, pelo telefone)*

A Bia tinha tentado na segunda: cancelou três entregas em homologação. As três ficaram na lista, riscadas. O Marcos tentou na terça, no computador dele: riscadas. A Cláudia pediu para a Lívia gravar a tela, e a Lívia respondeu que estava no cais, com o tablet, e que não sabia gravar.

— Não reproduz — disse o Marcos, na reunião das 10h. — Deve ser cache.

O Paulo, que não tinha falado a reunião inteira, pegou o notebook, abriu o painel de homologação e arrastou a borda da janela do navegador para a esquerda, até ela ficar da largura de um tablet. Cancelou a E-5460.

A linha sumiu.

Ele arrastou a borda de volta. A linha reapareceu, riscada.

— Não é o cancelamento — disse ele. — É a janela.



**FIGURA 17.1** Não é o cancelamento. É a janela.

## 17.1 Reproduzir antes de explicar

O ticket 1847 descrevia um **sintoma** — “ao cancelar, some” — e três pessoas foram procurar a causa no cancelamento. Todas abriram o código de `cancelar`, todas o acharam certo, e todas estavam certas. O código de cancelar não fazia a entrega sumir.

A pergunta que o Paulo fez não foi “por que some?”. Foi “o que é diferente entre a Livia e nós?”. A Livia usa um tablet, no cais, na horizontal. As três tentativas foram em monitores de escritório.

### ● O ESSENCIAL

Antes de explicar um defeito, reproduza-o. Se não reproduz, a pergunta não é “onde está o erro no código”, é “o que a pessoa que viu tem de diferente”: o aparelho, o tamanho da tela, o navegador, a conta, a hora, os dados. Explicação sem reprodução é um palpite com cara de diagnóstico.

Reproduzido, o defeito tem uma receita:

1. janela com menos de 900 pixels de largura;
2. uma entrega na lista;
3. cancelar.

A linha some. Com a janela larga, não some. É a receita que vai no ticket, antes de qualquer hipótese.

## 17.2 As ferramentas do navegador

F12, ou botão direito e “Inspecionar”, abre as ferramentas de desenvolvedor. O capítulo 3 usou o **Console**. Para um elemento que some, a aba certa é outra.

**Elementos** mostra o DOM do capítulo 16, vivo, como está agora — não o HTML que veio do servidor. Com a janela estreita e a E-5460 cancelada, a Bia procurou a linha:

```
<li class="entrega cancelada" data-codigo="E-5460">
  E-5460 · Padaria Jardim · cancelada
</li>
```

O elemento **existe**. O cancelamento não o apagou. Ele está na página e não aparece. Isso muda a pergunta de novo: não é “quem removeu?”, é “o que o esconde?”.

## 17.3 O estilo calculado

Com o `<li>` selecionado, o painel ao lado mostra os estilos. A aba **Calculado** (*Computed*) mostra o valor **final** de cada propriedade — depois de todas as regras de CSS que se aplicam ao elemento terem brigado entre si:

```
display: none
```

E, ao lado da propriedade, o arquivo e a linha da regra que venceu:

```
lista.css:212
```

### LISTA.CSS (LINHAS 205–215)

CSS

```
1  @media (max-width: 900px) {
2    .entrega {
3      padding: 4px 8px;
4      font-size: 14px;
5    }
6
7    .entrega.cancelada {
8      display: none;
9    }
10 }
```

`@media (max-width: 900px)` aplica as regras de dentro só quando a janela tem até 900 pixels. Dentro dela, `.entrega.cancelada` — um elemento com as duas classes — recebe `display: none`, que o tira da página.

O `git log` de `lista.css` tinha a mensagem do commit: `versão mobile: esconde canceladas para caber na tela`, de outubro de 2023. Alguém pensou num celular. Ninguém pensou que o tablet do cais, na horizontal, tinha 890 pixels.

**Estilo calculado** O valor final que o navegador aplica a cada propriedade de CSS de um elemento, depois de resolver todas as regras que o atingem. As ferramentas de desenvolvedor mostram esse valor e a regra de onde ele veio. Quando um elemento “some”, o estilo calculado responde se ele está lá e escondido, ou se não está lá.

## 17.4 Causa e sintoma

|                            | O QUE O TICKET DIZIA | O QUE ERA                                |
|----------------------------|----------------------|------------------------------------------|
| sintoma                    | “ao cancelar, some”  | a linha some com a janela estreita       |
| o que o cancelamento fazia | parecia a causa      | só punha a classe <code>cancelada</code> |
| causa                      | —                    | a regra de 2023 para celular             |

**TABELA 17.1** O cancelamento fornecia o estado. A regra de CSS fazia o resto. As três pessoas que leram o código de `cancelar` estavam lendo o código certo para a pergunta errada.

A correção é uma linha. A Livia confirmou que quer ver as canceladas em qualquer tela, riscadas:

| LISTA.CSS (DEPOIS)                                                                                                      | CSS |
|-------------------------------------------------------------------------------------------------------------------------|-----|
| <pre> 1  @media (max-width: 900px) { 2    .entrega { 3      padding: 4px 8px; 4      font-size: 14px; 5    } 6  }</pre> |     |

A regra que escondia saiu. A regra geral, fora do `@media`, já riscava a entrega cancelada, e passou a valer também no tablet.

## 17.5 O console, e o `debugger`

Nem todo defeito está no CSS. Para o que está no JavaScript, as mesmas ferramentas têm duas formas de olhar por dentro.

**O console, de propósito.** `console.log` no meio do código, com o nome do que se imprime:

```
console.log("cancelar", { codigo, statusAntes: e.status });
```

Com um objeto entre chaves, o console mostra os nomes junto com os valores. `console.log(codigo, status)` imprime dois textos soltos, e às 5h40 ninguém lembra qual era qual.

**O ponto de parada.** A palavra `debugger` numa linha, com as ferramentas abertas, **para** a execução ali:

```

1  function cancelar(codigo) {
2      const e = estado.entregas.find((x) => x.codigo === codigo);
3      debugger;
4      e.status = "cancelada";
5      desenhar();
6  }

```

A página congela na linha. A aba **Fontes** (*Sources*) mostra o arquivo, a linha destacada, e o valor de cada nome visível ali — `codigo`, `e`, `estado`. Os botões de passo avançam uma linha por vez. É o `console.log` de todas as variáveis ao mesmo tempo, sem ter de adivinhar qual imprimir.

O mesmo efeito, sem mudar o arquivo: clicar no número da linha na aba Fontes marca um ponto de parada.

#### ✂ ERRO COMUM

`debugger` e `console.log` de investigação esquecidos vão para produção. O `debugger` não faz nada na tela da Livia — as ferramentas dela estão fechadas —, e o `console.log` escreve para ninguém. O custo não é o defeito; é o arquivo que mente sobre o que é código de verdade. Antes do commit, a busca por `debugger` e por `console.log` no que mudou.

## 17.6 O teste da receita

A receita do 1847 virou uma conferência que qualquer um roda antes de fechar uma mudança na lista: abrir o painel, estreitar a janela até 800 pixels, cancelar uma entrega, e ver a linha riscada. As ferramentas de desenvolvedor têm um modo de dispositivo — o ícone do celular e do tablet no canto — que fixa a largura sem arrastar borda.

O ticket foi fechado com a receita, a causa, a linha removida, e o nome do Paulo no campo “reproduzido por”.

#### RESUMO

- Reproduza antes de explicar. Se não reproduz, procure o que a pessoa que viu tem de diferente.
- A receita do defeito vai no ticket antes da hipótese.
- A aba Elementos mostra o DOM vivo: se o elemento existe, algo o esconde; se não existe, algo o removeu.
- O estilo calculado mostra o valor final e a regra de onde ele veio.
- O sintoma aponta para onde o defeito aparece, não para onde ele está.
- `console.log` com nomes; `debugger` para parar e olhar tudo. Os dois saem antes do commit.

- 1 Um item da lista não aparece. Na aba Elementos, ele está lá. Diga três propriedades de CSS que você procuraria no estilo calculado.
- 2 A Cláudia abre o ticket 1851: “o total de fretes do rodapé está errado às vezes”. Escreva as três primeiras perguntas que você faria antes de abrir o código.
- 3 O Marcos propõe, para evitar o próximo 1847, proibir `display: none` em todo o `lista.css`. Responda.



## CAPÍTULO

# 18

## Estado

*A lista dizia cancelada. O painel de detalhe, ao lado, dizia a caminho. Cada pedaço da tela tinha guardado a sua cópia da entrega.*

---

### AO FIM DESTES CAPÍTULOS

Ter um lugar só de onde a tela lê a lista, mudar o dado ali e redesenhar a partir dele, e ver a diferença entre a tela que atualiza as duas regiões e a que atualiza uma.

—

**Capítulo disponível na  
versão completa**

—





PARTE 6

# O repositório

*Módulos, a terceira biblioteca de HTTP, e o caso que o Paulo achou porque jurou que ia procurar.*

## NESTA PARTE

---

- 19 Módulos
- 20 npm
- 21 Testes



## CAPÍTULO

# 19

# Módulos

*entrega.js passou a importar lista.js, que já importava entrega.js. O painel parou de carregar com uma mensagem sobre acessar um nome antes de ele existir.*

---

### AO FIM DESTES CAPÍTULO

Separar o painel em arquivos com export e import, entender em que ordem o Node carrega módulos que se importam, reproduzir a dependência circular, e desfazê-la com um terceiro arquivo que guarda o dado.

—

**Capítulo disponível na  
versão completa**

—





## CAPÍTULO

# 20

## npm

*Atualizar o Axios de 0.21 para a versão atual levou um minuto. A lista de canceladas de Franca parou de carregar, e só ela.*

---

### AO FIM DESTES CAPÍTULOS

Ler o package.json e o package-lock.json, instalar de propósito, atualizar uma dependência vendo exatamente o que quebra, e sair de três clientes HTTP para um no caminho de uma chamada com nome — não numa faxina geral.

—

**Capítulo disponível na  
versão completa**

—





## CAPÍTULO

# 21

## Testes

*O Paulo cancelou a entrega no intervalo entre o clique em iniciar rota e a resposta do servidor. A entrega cancelada voltou a aparecer a caminho.*

---

### AO FIM DESTES CAPÍTULO

Escrever um teste com `node:test` que falha pelo motivo certo e passa quando o código muda; testar o intervalo que ninguém imaginou, e não um espelho do caminho feliz; e saber o que um teste verde afirma.

—

**Capítulo disponível na  
versão completa**

—





PARTE 7

# O outro lado e o custo

*O contrato com o servidor, e a decisão inocente que começa a custar caro quando a lista cresce.*

## NESTA PARTE

---

22 APIs

23 Performance



## CAPÍTULO

# 22

## APIs

*A página mandava status cancelado. O servidor aceitava CANCELADA. Os dois tinham teste verde, e a entrega nunca foi cancelada no servidor.*

---

### AO FIM DESTES CAPÍTULOS

Fazer um pedido com fetch e ler a resposta inteira, status incluído; entender por que fetch não falha num 422; e escrever, numa tabela curta, o contrato que a página e o servidor vão obedecer.

—

**Capítulo disponível na  
versão completa**

—





## CAPÍTULO

# 23

## Performance

*O relógio do canto da tela mudava a cada segundo. A cada segundo, cento e sessenta cartões eram jogados fora e feitos de novo. O tablet do cais esquentava.*

---

### AO FIM DESTES CAPÍTULOS

Medir antes de reescrever, contar quantas vezes a tela é refeita e por quê, e parar de reconstruir a lista inteira a cada evento que não muda a lista — com números antes e depois.

—

**Capítulo disponível na  
versão completa**

—





## PARTE 8

# Sair da máquina

*O commit que alguém precisa ler daqui a seis meses, a sexta às 16h52, e a manhã em que o papel e o painel dizem a mesma coisa.*

### NESTA PARTE

---

- 24 Git
- 25 Deploy
- 26 Projeto final



## CAPÍTULO

# 24

## Git

*Quinta, 17h43. A Bia escreveu ajustes na mensagem do commit. O Rafael leu por cima do ombro e puxou uma cadeira.*

---

### AO FIM DESTES CAPÍTULOS

Escrever uma mensagem de commit que responde o que mudou e por quê seis meses depois, dizer ao Git o que não entra no repositório, e concluir um merge em que os dois lados mexeram na mesma função sem apagar o lado do outro.

—

**Capítulo disponível na  
versão completa**

—





## CAPÍTULO

# 25

## Deploy

*Sexta, 16h52. Na máquina da Bia, Node 24, tudo verde. No servidor, Node 20, o rodapé nem carregava. O Neto apareceu na porta antes de alguém chamar.*

---

### AO FIM DESTES CAPÍTULOS

Dizer o que tem de ser igual entre a máquina local e o servidor — versão do Node, variáveis de ambiente, o comando que sobe o processo —, achar a diferença concreta que quebra o Node 20, e recusar um horário.

—

**Capítulo disponível na  
versão completa**

—





## CAPÍTULO

# 26

## Projeto final

*Quarta, 6 de maio, 5h35. A Livia imprimiu a lista, pôs o papel ao lado do tablet e foi descendo com o dedo.*

---

### AO FIM DESTE CAPÍTULO

Juntar o que o livro construiu na pasta borba — lista, status, frete, o teste do caso do Paulo, o contrato com o servidor, um commit legível e um jeito de subir que o Neto aceita numa segunda — e fazer, sozinho, o próximo pedido que a operação vai trazer.

—

**Capítulo disponível na  
versão completa**

—





# Respostas

---

## Do Mocha ao Node

- 1 A Netscape e a Sun publicaram o comunicado. O nome JavaScript foi escolhido para acompanhar o Java, que era a linguagem da moda, num par comercial: uma para componente, outra para página.

O Mocha, desde maio, usava protótipo. O objeto existe e aponta para outro objeto quando lhe falta uma propriedade. Classe, o molde do Java, não era o mecanismo. O `class` de 2015 é uma grafia posterior em cima desse ponteiro.

- 2 Em agosto de 1996 a Microsoft publicou o JScript no Internet Explorer 3, uma implementação com outro nome, porque JavaScript era marca da Sun.

Um padrão que a Microsoft aceitasse implementar não podia carregar essa marca. A Netscape entregou a linguagem à Ecma. Em junho de 1997 saiu a primeira edição do ECMA-262. No documento, o nome é ECMAScript.

- 3 Não com esse texto. Desde a reunião com o `io.js`, em 2015, linha par tem prazo LTS, e a linha Current muda sem esse prazo. “A mais nova que existir” não diz qual das duas a Cláudia está pedindo.

O servidor está no 20, cuja correção termina em 30 de abril de 2026. Trocar é escolher uma LTS pelo número — a 22 ou a 24 — e publicar isso num horário em que a manhã da Livia não dependa do resultado. O ticket não nomeia a linha nem o horário.

## O que vamos construir

- 1 Ela precisa saber se a entrega está na lista da manhã e em que status. A resposta é uma linha, ou “não está”.

Ela não precisa saber se quem buscou o dado foi o Axios, o `fetch` ou a pasta `http/`, nem em qual arquivo a função mora. Isso é o caminho. A conferência dela é o papel na mão.

- 2 Marcos Teixeira, em 12 de agosto de 2025. A mensagem afirma: “troca axios por fetch”.

Não prova que o Axios tenha saído da pasta, nem que o `fetch` esteja dentro de `entrega.js`. Prova o que ele escreveu na hora de registrar. O que sobrou se vê no arquivo.

- 3 Não esta semana. Separar em três programas é uma mudança de estrutura. A folha pede que o status caiba em trinta segundos e que a lista da tela coincida com a impressa.

Dá para fazer as duas coisas dentro do arquivo que já existe, medindo a diferença. Três programas novos, com a cláusula em 6 de maio, trocam um problema que a Livia sabe contar por três problemas que ninguém rodou ainda.

## O primeiro programa

1

```
STATUS.JS javascript

const codigo = "E-4821";
const situacao = "a caminho";
const cidade = "Ribeirão Preto";

console.log(`Entrega ${codigo}: ${situacao} (${cidade})`);

$ node status.js
Entrega E-4821: a caminho (Ribeirão Preto)
```

Os parênteses em volta de `${cidade}` estão fora das chaves. Por isso saem na impressão como texto, em volta do nome da cidade.

2

ReferenceError na linha 2. `codigo` existe. `situacao` não foi criado.

..

A falta do ponto e vírgula na linha 1 não é o que quebra. O Node aceita a instrução mesmo assim e segue. O nome ausente é o primeiro furo de verdade.

3

```
ANDA.JS javascript

let situacao = "a caminho";
situacao = "entregue";
console.log(situacao);
```

`const` recusa a segunda atribuição. O nome precisa nascer com `let` para a linha seguinte poder apontá-lo a outro texto. Se a situação não mudasse dentro do arquivo, `const` seria o certo — e o `TypeError` estaria protegendo essa decisão.

## async/await

1

Primeiro `Promise { 1890 }`, depois 1890.

A função `async` devolve uma promessa, mesmo retornando um número direto. O Node mostra a promessa já cumprida com o valor entre chaves. O `then` recebe o número.

**2**

..

```

1  async function marcar(codigo) {
2    try {
3      await gravarStatus(codigo);
4      await atualizarLinha(codigo);
5    } catch (erro) {
6      console.log(`gravação falhou: ${erro.message}`);
7      return;
8    }
9
10   try {
11     await notificarCliente(codigo);
12   } catch (erro) {
13     console.log(`aviso pendente: ${erro.message}`);
14   }
15 }

```

Dois `try`: um para o que é essencial, que interrompe; outro para a notificação, que só registra. O `return` no primeiro `catch` impede de notificar uma entrega que não foi gravada — o que a cadeia original também impedia, por o `catch` estar no fim.

**3**

...

Demora  $160 \times 300$  ms, uns 48 segundos: cada pedido só sai quando o anterior voltou. A Livia imprime às 5h35 e a lista ainda está carregando.

```

1  const resultados = await Promise.allSettled(
2    entregas.map((e) => buscarStatus(e.codigo)),
3  );
4
5  resultados.forEach((r, i) => {
6    entregas[i].status =
7      r.status === "fulfilled" ? r.value : "sem resposta";
8  });

```

Os 160 pedidos saem juntos, e a lista demora o tempo do mais lento.

O que a versão nova precisa tratar: 160 pedidos ao mesmo tempo contra um servidor que antes recebia um por vez. Se o servidor da Borba não aguentar, alguns vão falhar — por isso `allSettled`, e não `all` — e vale limitar quantos saem de uma vez. E a ordem de chegada deixa de ser a da lista; o índice `i` é o que devolve cada resposta à sua entrega.

## Debugging

- 1** `display: none`, que o tira da página; `visibility: hidden`, que o deixa ocupando espaço mas invisível; e `opacity: 0`, que o deixa transparente.

E, fora dessas três, a posição: um elemento com `height: 0` e `overflow: hidden`, ou posicionado fora da área visível. O estilo calculado mostra todas, com a regra de origem.

- 2** Errado quanto, e comparado com o quê? Um print com o total da tela e o número que ela esperava, e de onde veio esse número.

Quando? Em que horário, depois de qual ação — cancelar, ajustar frete, atualizar —, e se some sozinho depois.

Onde? Em qual aparelho, com qual tela, com a lista de qual cidade.

As três perguntas buscam a receita. “Às vezes” quase sempre esconde uma condição que ainda não foi descoberta, como a largura da janela no 1847.

- 3** A proibição ataca a ferramenta, e não o defeito. `display: none` é o jeito certo de esconder o menu fechado, o aviso já lido, o botão que não se aplica. Proibido, ele volta como `height: 0` OU `opacity: 0`, que escondem do mesmo jeito e são mais difíceis de achar no estilo calculado.

O defeito do 1847 foi uma regra de **negócio** — o que a operação vê — escrita no CSS, sem ninguém da operação saber. A proposta que eu levaria: regra de CSS que esconde **dado** — entrega, frete, status — passa por quem decide o que a Lívia vê, e a conferência da janela estreita entra na lista do que se testa antes de subir.



# Glossário

---

**await** Pausa a função `async` em que está até a promessa à direita terminar, e devolve o valor dela. Se a promessa for rejeitada, o `await` lança o erro, como um `throw` naquela linha. Fora de uma função `async` — num arquivo comum do Node —, `await` é `SyntaxError`.

**Console do navegador** Uma linha de comando dentro das ferramentas de desenvolvedor. Executa JavaScript na página que está aberta. Uma página em branco não é o painel da Borba. O que você digitou aqui é seu. Não consultou entrega nenhuma no servidor.

**const** Cria um nome e o prende a um valor. Prender quer dizer que o nome não aceita outra atribuição em seguida. `situacao = "entregue"` na linha de baixo falha. O valor `"a caminho"` continua sendo o valor daquele nome até o programa terminar.

**ECMAScript** O nome da linguagem dentro do padrão ECMA-262. JavaScript continua sendo o nome comercial, usado pelo navegador, pelo Node e por quem escreve. São o mesmo idioma. Quando a edição importar, ela vem com o ano, ou com o número antigo: ES3, ES5, ES2015.

**Estilo calculado** O valor final que o navegador aplica a cada propriedade de CSS de um elemento, depois de resolver todas as regras que o atingem. As ferramentas de desenvolvedor mostram esse valor e a regra de onde ele veio. Quando um elemento “some”, o estilo calculado responde se ele está lá e escondido, ou se não está lá.

**Git** Programa que guarda o histórico de uma pasta: o que mudou, quem mudou, quando, e a mensagem escrita na hora. O `git log` lê esse histórico. Não altera o arquivo. Quem colocou está no `Author`. O que a pessoa achou que estava fazendo está na mensagem. O que ela fez está no arquivo.

**HTTP** O protocolo com que um programa pede um dado a outro computador. Uma requisição sai, uma resposta volta. `Axios`, `fetch` e a pasta `http/` fazem essa requisição. Três textos, um protocolo.

**Java** Linguagem da Sun, anunciada em 1995, com classe, tipo declarado e compilação. A Netscape ia embarcá-la no navegador para aplicação grande. O Mocha recebeu ordem de parecer com ela na superfície. O objeto, a função e o momento em que o programa roda ficaram de outro desenho.

**JSON** JavaScript Object Notation. Um texto para levar dado de um programa a outro: chaves, nomes entre aspas duplas, listas entre colchetes. É um recorte da forma como a linguagem já escrevia um objeto em 1995. Douglas Crockford isolou esse recorte por volta de 2001 e publicou as regras. Em 2009 a ES5 colocou na linguagem as funções que leem esse texto e que o escrevem. Sistemas que não são JavaScript passaram a usá-lo como formato neutro.

**Lazo de eventos** A fila em que o motor registra “quando isto acontecer, chame esta função” e, no meio-tempo, executa o que já pode ser executado. É o desenho do navegador desde que função virou valor, e é o desenho que o Node adotou em

2009 para o servidor. O painel da Borba espera rede a maior parte do tempo. É esse mecanismo que segura a espera.

**let** Cria um nome que aceita outra atribuição. A segunda linha aponta `situacao` para outro texto. A primeira ligação deixa de valer. Use **let** quando o valor muda. No resto, `const`. Um nome que não muda é mais fácil de ler às 5h40 do que um nome que talvez mude.

**LTS** *Long Term Support*. Linha do Node que recebe correção de segurança por um período combinado, com data de término publicada antes. A linha chamada Current é a mais nova, ainda sem esse prazo, e muda seguido. Servidor de operação fica numa LTS.

**npm** O registro público de bibliotecas JavaScript, e também o comando que instala uma delas a partir do que o `package.json` anotou. Vem junto com o Node. Não há nada para instalar agora. A lista acima é a da produção, e a instrução da manhã foi não mexer nela.

**Protótipo** O objeto para o qual outro objeto aponta quando não acha uma propriedade em si mesmo. A busca segue esse ponteiro. É o modelo de objeto do JavaScript desde o protótipo de maio de 1995. Classe, no sentido de Java, não fazia parte dessa encomenda.

**V8** Motor de JavaScript do Google, publicado em 2008 com o Chrome. É o motor que o Node usa. O Chrome e o Node podem carregar versões diferentes desse motor. Os dois executam a linguagem. Os dois não são obrigados a aceitar exatamente o mesmo texto, se as versões divergirem.

**V8** O motor, escrito pelo Google, que executa o JavaScript dentro do Chrome e dentro do Node. `node status.js` entrega o seu arquivo a esse motor, mais às peças de arquivo e terminal que o Node acrescenta. A versão do motor no seu Node 24 não é a versão do motor no Node 20 do servidor. O arquivo desta página cabe nas duas.

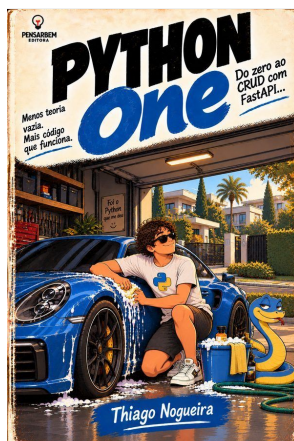
# Conheça as demais obras dessa coleção

---



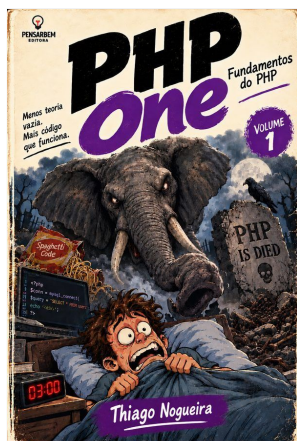
## Java One

*Do primeiro println à API REST em produção*



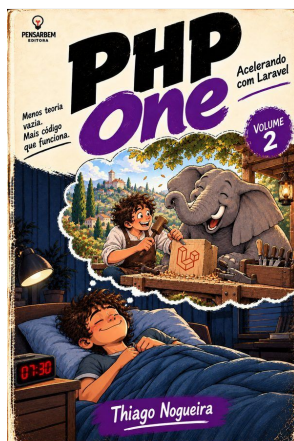
## Python One

*Do primeiro print à API REST em produção*



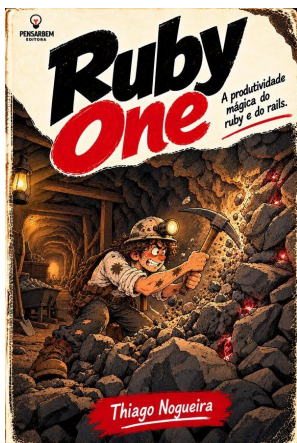
## PHP One

*Volume 1 · Fundamentos do PHP*



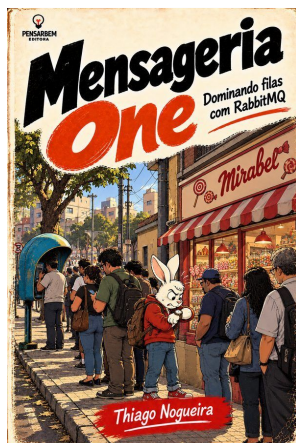
## PHP One

*Volume 2 · Acelerando com Laravel*



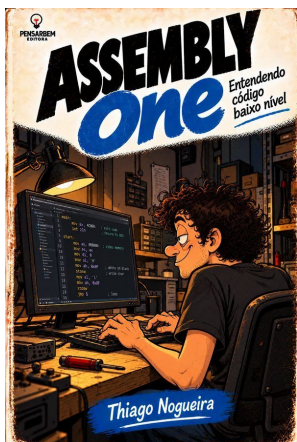
## Ruby One

*Do primeiro puts ao contrato em produção*



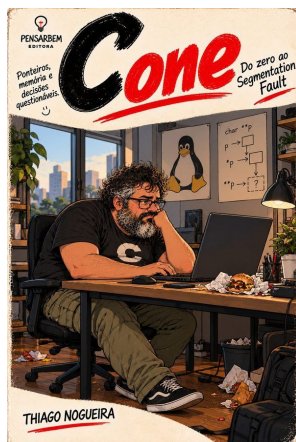
## Mensageria One

*RabbitMQ do primeiro publish à Black Friday sem perder pedido*



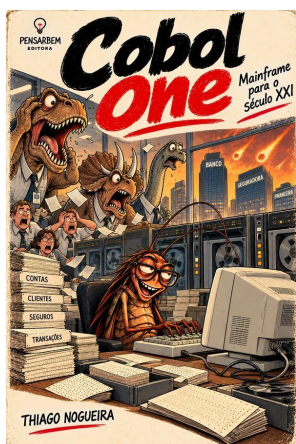
## Assembly One

*Do zero ao RISC-V*



## C One

*Do zero ao básico em sistemas embarcados e IoT*



## COBOL One

*Mainframe para o Século XXI*

---

**Visite a editora Pensar Bem →**

[pensarbem.com.br](http://pensarbem.com.br)