

ZERO ONE 1

Java One — Prévia

Do primeiro println à API REST em produção

Thiago Nogueira

PENSARBEM

Java 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

Antes de começar	1
Como este livro funciona	4
O que você precisa	4
O elenco	5
Convenções	6
1 O que vamos construir	10
1.1 Um programa que atende, em vez de um que roda	12
1.2 CRUD: quatro verbos e nada mais	14
1.3 Java, Spring e Spring Boot não são a mesma coisa	14
1.4 Trinta anos em uma página	15
1.5 Instalando o ambiente	16
1.6 O caminho inteiro, em uma imagem	17
2 Primeiro programa em Java	19
2.1 O arquivo	22
2.2 A assinatura mais copiada da história	23
2.3 Argumentos: o programa recebendo o mundo	24
2.4 Quando o compilador reclama	25
2.5 Duas palavras que você vai ouvir muito	25
2.6 E a IDE?	25
3 Variáveis e tipos	27
3.1 A forma da declaração	30
3.2 Os tipos que resolvem quase tudo	30
3.3 O compilador confere antes de rodar	30
3.4 Conversões: quando o tipo muda de ideia	30
3.5 Dinheiro não é <code>double</code>	30
3.6 Constantes	30
3.7 <code>var</code> : deixar o compilador escrever o tipo	30
4 Operadores e expressões	31
4.1 Aritméticos	34
4.2 Atribuição e a forma curta	34
4.3 Incremento: o <code>++</code> e a posição dele	34
4.4 Comparação	34
4.5 <code>==</code> compara identidade, <code>.equals()</code> compara conteúdo	34
4.6 Lógicos	34
4.7 O ternário	34
5 Condicionais	35

5.1	<code>if</code> exige um <code>boolean</code>	38
5.2	Chaves: o caso em que economizar custa caro	38
5.3	A escada de <code>else if</code>	38
5.4	<code>switch</code> : quando a pergunta é “qual destes?”	38
5.5	Qual usar	38
5.6	Um <code>if</code> que você vai escrever muito	38
6	Repetições	39
6.1	Uma lista de valores	42
6.2	O laço que você vai usar quase sempre	42
6.3	O laço com contador	42
6.4	<code>while</code> e <code>do while</code>	42
6.5	Acumular um resultado	42
6.6	Sair antes: <code>break</code> e <code>continue</code>	42
6.7	Laço infinito: como acontece e como sair	42
7	Métodos	46
7.1	Da repetição ao método	48
7.2	<code>void</code> e <code>return</code>	48
7.3	Escopo: onde um nome existe	48
7.4	Sobrecarga: mesmo nome, assinaturas diferentes	48
7.5	Métodos que documentam regras	48
8	Arrays e coleções	49
8.1	As três perguntas	52
8.2	<code>List</code> : ordem e repetição	52
8.3	<code>Set</code> : pertence ou não	52
8.4	<code>Map</code> : chave e valor	52
8.5	Listas imutáveis: o atalho moderno	52
8.6	Ordenar	52
8.7	Percorrer: as três formas e quando usar cada uma	52
8.8	O que o projeto vai usar	52
9	Orientação a objetos	53
9.1	Do dado solto ao objeto	56
9.2	A primeira classe	56
9.3	Classe e instância não são a mesma coisa	56
9.4	<code>static</code> finalmente explicado	56
9.5	<code>toString</code> : o objeto se apresentando	56
9.6	<code>equals</code> e <code>hashCode</code> : a lição do capítulo 4, agora do seu lado	56
10	Encapsulamento	57
10.1	<code>private</code> : fechar a porta	60
10.2	Validar onde o objeto nasce	60
10.3	Getters e setters, com critério	60
10.4	Os quatro níveis de acesso	60
10.5	Imutabilidade: a versão radical	60

10.6	Onde isso vai aparecer no projeto	60
11	Herança e interfaces	61
11.1	Herança: <code>extends</code>	64
11.2	Polimorfismo: o mesmo comando, respostas diferentes	64
11.3	Interface: o contrato sem implementação	64
11.4	Por que interface é a base do Spring	64
11.5	Métodos <code>default</code> : interface com implementação	64
11.6	Classe abstrata: o meio do caminho	64
11.7	Composição: quando “tem um” vence “é um”	64
11.8	<code>final</code> : proibir a herança	64
12	Enums e records	65
12.1	O problema do <code>String</code> como categoria	68
12.2	<code>enum</code> : o conjunto fechado	68
12.3	Enum com dados e comportamento	68
12.4	<code>record</code> : o dado puro	68
12.5	Quando usar record e quando usar classe	68
12.6	Os dois juntos, como o projeto vai usar	68
13	Exceções	69
13.1	O erro mais famoso da plataforma	72
13.2	<code>try</code> , <code>catch</code> , <code>finally</code>	72
13.3	Dois famílias: <i>checked</i> e <i>unchecked</i>	72
13.4	<code>throw</code> : avisar em vez de mentir	72
13.5	Exceção própria: o nome que carrega o significado	72
13.6	<code>Optional</code> : a alternativa ao <code>null</code>	72
13.7	<code>try-with-resources</code>	72
14	Lambdas e streams	73
14.1	Lambda: uma função sem nome	76
14.2	Referência de método: a lambda que já existe	76
14.3	Stream: a sequência de operações	76
14.4	As cinco operações que resolvem quase tudo	76
14.5	Reduzir a um número	76
14.6	Agrupar: o <code>Map</code> do capítulo 8 em uma linha	76
14.7	Quando o laço ainda é melhor	76
14.8	O pipeline que o projeto vai usar	76
15	O que é Spring	80
15.1	O problema que o Spring resolve	82
15.2	Inversão de controle	83
15.3	O contêiner e os <i>beans</i>	83
15.4	Quatro anotações que dizem a mesma coisa	85
15.5	Escopo: quantos objetos existem	85
15.6	O que exatamente o Spring Boot acrescenta	85
15.7	Onde a mágica cobra o preço	86

16	O primeiro projeto Spring Boot	88
16.1	Gerando o projeto	90
16.2	A estrutura, pasta por pasta	90
16.3	A classe principal	90
16.4	pom.xml : as dependências	90
16.5	Rodando	90
16.6	application.properties	90
16.7	Um endpoint de teste, só para provar que está vivo	90
17	A primeira API	91
17.1	HTTP em uma página	94
17.2	O controlador	94
17.3	Ler dados: três origens, três anotações	94
17.4	O CRUD completo	94
17.5	Os cinco diálogos, em ordem	94
17.6	Testando pela linha de comando	94
17.7	O que está errado nesta versão	94
18	Modelando a aplicação	98
18.1	Entidade: o substantivo que tem identidade	100
18.2	O modelo do livro	100
18.3	Os atributos, um por um	100
18.4	A chave primária	100
18.5	Relações: quem conhece quem	100
18.6	Três perguntas que evitam refatoração	100
18.7	Estrutura de pacotes	100
19	Banco de dados	101
19.1	Subindo um PostgreSQL em dois minutos	104
19.2	SQL: quatro instruções e nada mais (por enquanto)	104
19.3	As quatro operações	104
19.4	Chave estrangeira: a integridade que o banco cobra	104
19.5	Índice: por que uma busca é rápida	104
19.6	Conectando a aplicação	104
19.7	O SQL não vai embora	104
20	Spring Data JPA	105
20.1	Três nomes, três papéis	108
20.2	A entidade	108
20.3	Por que a entidade precisa de um construtor vazio	110
20.4	O ciclo de vida de uma entidade	111
20.5	Conferindo o mapeamento	112
20.6	Quando a tradução automática atrapalha	112
21	Repository	114
21.1	O que acabou de acontecer	116

21.2	Os métodos que vêm de graça	116
21.3	Query methods: a consulta que nasce do nome	116
21.4	@Query: quando o nome não basta	116
21.5	Testando o repositório de verdade	116
22	Service	117
22.1	As três responsabilidades, separadas	120
22.2	O serviço	120
22.3	@Transactional: o que essa anotação realmente faz	120
22.4	As exceções de negócio	120
22.5	A regra de negócio de verdade	120
22.6	O controlador, agora magro	120
23	Controller	121
23.1	O CRUD inteiro	124
23.2	Duas formas de declarar o status	124
23.3	A operação que não é CRUD	124
23.4	A API completa, em uma tabela	124
23.5	Provando que funciona	124
23.6	O que ainda está feito	124
24	DTOs	125
24.1	Três problemas concretos	128
24.2	Dois records, dois contratos	128
24.3	A conversão	128
24.4	O controlador final da Parte 4	128
24.5	Ajustando o JSON sem tocar na entidade	128
24.6	Quando o DTO não vale a pena	128
25	Validação	132
25.1	O problema, em uma requisição	134
25.2	Bean Validation	134
25.3	As anotações que você vai usar	134
25.4	O que o cliente recebe	134
25.5	Validação própria	134
25.6	Onde validar: as três camadas	134
25.7	Validando objetos aninhados e listas	134
26	Tratamento de erros	135
26.1	Um lugar para traduzir tudo	138
26.2	A resposta de erro padronizada	138
26.3	Erro de validação com o campo que falhou	138
26.4	Escolhendo o status	138
26.5	A rede de segurança	138
26.6	O problema do erro genérico bem-intencionado	138

27	Paginação e ordenação	139
27.1	Pageable : o Spring já sabe fazer	142
27.2	O SQL que sai disso	142
27.3	Defina o padrão e o teto	142
27.4	O custo escondido do OFFSET	142
27.5	Ordenação e o que ela pode expor	142
27.6	O contrato final da listagem	142
28	Filtros e buscas	143
28.1	O jeito errado, que funciona por seis meses	146
28.2	Filtro simples: o nome do método resolve	146
28.3	A explosão combinatória dos filtros opcionais	146
28.4	Specification: o filtro como objeto	146
28.5	Por que LIKE '%termo%' é lento	146
28.6	Acento: o detalhe que ninguém lembra	146
28.7	Qual das três formas usar	146
29	Relacionamentos JPA	150
29.1	A relação mais comum: @ManyToOne	152
29.2	O outro lado: @OneToMany	152
29.3	Bidirecional cobra disciplina	152
29.4	@OneToMany com dono próprio: os itens do pedido	152
29.5	@ManyToMany : quando os dois lados são muitos	152
29.6	O modelo do projeto, completo	152
30	Problemas reais do JPA	153
30.1	O problema N+1	156
30.2	As três soluções	156
30.3	LazyInitializationException	156
30.4	Transação: onde ela começa e onde termina	156
30.5	O checklist do JPA em produção	156
31	Spring Security	160
31.1	Duas palavras que não são sinônimas	162
31.2	Adicionando a dependência (e o susto)	162
31.3	A cadeia de filtros	162
31.4	Senha nunca é guardada	162
31.5	O usuário como entidade	162
31.6	Ensinando o Spring a encontrar o usuário	162
32	JWT	163
32.1	O fluxo inteiro	166
32.2	O que é um JWT, exatamente	166
32.3	Gerando e validando	166
32.4	O filtro que lê o cabeçalho	166
32.5	O endpoint de login	166

32.6	O preço de não guardar nada	166
33	Usuários e permissões	167
33.1	O mapa de permissões da Aurora	170
33.2	Dois lugares para declarar a regra	170
33.3	A regra que o filtro não consegue expressar	170
33.4	Quem está logado, dentro do código	170
33.5	Autorização também é regra de negócio	170
33.6	O que o cliente recebe	170
34	Testando Java	174
34.1	O que um teste compra	176
34.2	O primeiro teste	176
34.3	AssertJ: a afirmação que se lê	176
34.4	O nome do teste é a especificação	176
34.5	Organizando: <code>@Nested</code> e <code>@DisplayName</code>	176
34.6	Casos repetidos: <code>@ParameterizedTest</code>	176
34.7	O que não testar	176
34.8	Cobertura: um número que engana	176
35	Testando services	177
35.1	O mock	180
35.2	Verificando o que foi chamado	180
35.3	O vocabulário mínimo do Mockito	180
35.4	Quando o mock passa a atrapalhar	180
35.5	A alternativa: o duplê escrito à mão	180
36	Testando controllers	181
36.1	MockMvc: HTTP sem rede	184
36.2	Status e cabeçalho	184
36.3	Validação: o teste que prova que o 400 é 400	184
36.4	Testando com usuário autenticado	184
36.5	<code>jsonPath</code> : navegando na resposta	184
36.6	Quando subir a aplicação inteira	184
37	Testando o banco	185
37.1	<code>@DataJpaTest</code>	188
37.2	O problema do H2	188
37.3	Testcontainers: o banco de verdade, descartável	188
37.4	Migração versionada: o teste que vale para produção	188
38	Swagger e OpenAPI	192
38.1	Uma dependência, e a documentação existe	194
38.2	OpenAPI, Swagger e a confusão de nomes	194
38.3	Enriquecendo o que o código não diz	194
38.4	Documentando o DTO, que é onde o cliente olha	194

38.5	O botão de autorizar	194
38.6	O que a documentação gerada não resolve	194
39	Clientes HTTP	195
39.1	<code>curl</code> : o denominador comum	198
39.2	Arquivos <code>.http</code> : a requisição versionada	198
39.3	Postman e Insomnia	198
39.4	Um passo além: gerar o cliente	198
40	Git	199
40.1	Os quatro comandos do dia	202
40.2	O commit é uma mensagem para o futuro	202
40.3	Ramos	203
40.4	Conflito: o que é e como sair	204
40.5	Desfazer, sem drama	205
40.6	O histórico como ferramenta	205
41	Projeto final	210
41.1	Os requisitos	212
41.2	O modelo	212
41.3	O contrato da API	212
41.4	O pedido, que é a parte nova	212
41.5	A ordem sugerida	212
41.6	O que conta como pronto	212
41.7	Três armadilhas que este projeto tem de propósito	212
42	Docker e deploy	213
42.1	Por que Docker	216
42.2	O Dockerfile em dois estágios	216
42.3	Docker Compose: a aplicação e o banco juntos	216
42.4	Configuração por ambiente, nunca por arquivo	216
42.5	O que olhar quando está no ar	216
42.6	Integração contínua, em vinte linhas	216
42.7	O que vem depois deste livro	216
	Respostas	217
	Glossário	223

Antes de começar

*Um livro curto sobre uma linguagem que
costuma ser ensinada longa demais.*

Existem duas maneiras de ensinar Java. A primeira apresenta a linguagem inteira — trinta anos de recursos, quatro tipos de laço, oito modificadores, a hierarquia de coleções em um diagrama do tamanho de uma parede — e promete que um dia aquilo vira um programa. A segunda escolhe um programa e ensina só o Java que ele exige, quando ele exige.

Este livro é do segundo tipo. Ele tem um projeto: uma API REST de catálogo de produtos, com banco de dados, validação, autenticação, testes e documentação. Você vai construí-la. Cada conceito da linguagem aparece no capítulo em que o projeto precisa dele, e não antes.

● O ESSENCIAL

O livro não é uma referência de Java. É um caminho até uma API funcionando. Se você quiser a referência completa depois, ela cabe em um site; o caminho é que é difícil de achar.

Como este livro funciona

O projeto cresce em linha reta. No capítulo 2 você imprime uma linha no terminal; no capítulo 17 responde uma requisição HTTP; no 23 tem um CRUD completo; no 32 protege esse CRUD com um token; no 42 sobe tudo em um contêiner. Cada capítulo encerra com o estado do projeto naquele ponto:

■ O PROJETO AGORA

Você tem: nada ainda. Um terminal aberto, um JDK instalado e uma pasta vazia. É exatamente daqui que todo software começa.

O caminho não é limpo de propósito. Você vai escrever código que quebra, ler a mensagem de erro inteira e consertar. É assim que se aprende a programar de verdade — e é a parte que os tutoriais editam fora.

O que você precisa

Um JDK 21 ou mais novo, um editor de texto e um terminal. Um banco PostgreSQL aparece na Parte 4, e o capítulo 19 explica como subir um em dois minutos. Nada mais.

CONFIRME QUE ESTÁ TUDO NO LUGAR

bash

```
java -version
javac -version
```

Se os dois responderem com um número de versão, você está pronto.

⚡ ERRO COMUM

`java -version` respondendo e `javac -version` não respondendo é o sintoma clássico de ter instalado um JRE em vez de um JDK. O JRE só executa; o JDK compila. Você precisa do segundo.

O elenco

Entre uma explicação e outra, você vai encontrar cenas de uma empresa. Ela se chama Aurora Comércio, vende de tudo pela internet e acabou de decidir que precisa de uma API — de preferência para ontem.

As cenas não são enfeite: elas existem porque conceito abstrato gruda melhor quando vem colado a uma situação que você reconhece. Quando o livro explicar `NullPointerException`, você vai lembrar do Seu Antônio.

ENQUANTO ISSO, NA EMPRESA Os cinco que você vai encontrar

Carlos entrou há três semanas. Sabe lógica, não sabe Java, e foi escalado para “aquele projetinho da API”. Ele é você.

Marina é a desenvolvedora sênior. Já viu esse filme, sabe como termina e mesmo assim insiste em revisar cada *pull request* linha por linha.

Roberto é o gerente. Não escreve código, escreve prazos. Para ele, qualquer pedido cabe na frase “é só uma alteraçãozinha”.

Cláudia cuida do produto. Traz requisitos novos com a naturalidade de quem traz café — inclusive na véspera da entrega.

Seu Antônio, 68 anos, é o primeiro cliente da loja. Ele encontra, em dois minutos de uso, bugs que a equipe não reproduz em duas semanas.



FIGURA 0.1 O time da Aurora Comércio no primeiro dia do projeto.

Convenções

Trechos como `System.out.println` aparecem na fonte de código quando citados no meio da frase. Blocos maiores vêm com o nome do arquivo no alto — em Java esse nome importa mais do que em quase qualquer outra linguagem, e o capítulo 2 explica por quê.

Cinco marcas aparecem nas margens do texto e vale reconhecê-las de antemão:

★ VOCÊ SABIA

Você sabia traz a curiosidade, a história, a decisão de projeto que ficou. Pode ser saltada sem prejuízo — mas é o que faz você contar a história em uma mesa de bar.

⚠ ERRO COMUM

Erro comum mostra a armadilha antes de você cair nela. Quase sempre é um erro que eu cometi.

🕒 NOS BASTIDORES

Nos bastidores conta por que a linguagem é assim. Java tem muitas decisões estranhas, e quase todas têm um motivo datado.

✓ PONTO DE CONTROLE

Ponto de controle fecha o capítulo listando o que você já sabe fazer — não o que leu, o que *sabe fazer*.

E **O projeto agora** é o marco do capítulo: uma frase sobre onde a API está. Se você abrir o livro no meio, é a primeira coisa que deve procurar.

PARTE 1

Começando do zero

Você escreve Java antes de entender Java. A linguagem entra pela porta da prática: compilar, errar, ler o erro, consertar.

NESTA PARTE

- 1 O que vamos construir
- 2 Primeiro programa em Java
- 3 Variáveis e tipos
- 4 Operadores e expressões
- 5 Condicionais
- 6 Repetições

CAPÍTULO

1

O que vamos construir

*Antes da primeira linha de código, o destino.
Ninguém aprende bem uma viagem sem saber
onde ela termina.*

*“Eu não tinha ideia de que estava escrevendo uma
linguagem de programação para a internet. Eu
estava escrevendo uma para torradeiras.”*

JAMES GOSLING, CRIADOR DO JAVA

AO FIM DESTE CAPÍTULO

Explicar, com as palavras certas, o que é uma API REST, o que é um CRUD e qual é a diferença entre Java, Spring e Spring Boot — e ter o ambiente instalado e testado.

No fim deste livro existe um programa rodando. Ele não tem tela, não tem botão e ninguém vai elogiar o visual dele. Ele fica esperando, e quando alguém pergunta a coisa certa, ele responde:

REQUISIÇÃO **GET** /products/7
Accept: application/json

RESPOSTA **200 OK**
Content-Type: application/json

```
{
  "id": 7,
  "name": "Teclado mecânico",
  "price": 349.90,
  "quantity": 12
}
```

Isso é uma **API**. Não tem mistério na sigla: *Application Programming Interface*, uma interface para programas em vez de para pessoas. O navegador, o aplicativo de celular e o site do cliente falam com ela pelo mesmo protocolo que você usa para ler notícias — HTTP.

1.1 Um programa que atende, em vez de um que roda

Um programa comum começa, faz o trabalho e termina. Um servidor não termina: ele sobe, abre uma porta e fica aguardando. Essa diferença muda como você pensa o código.

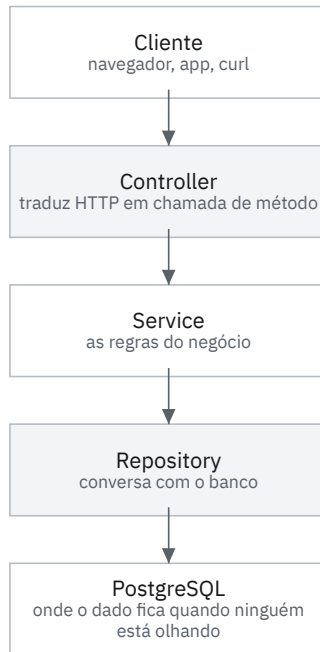


FIGURA 1.1 O caminho de uma requisição: cada camada resolve um problema e passa adiante.

Essas quatro camadas são o esqueleto do projeto. Elas aparecem no capítulo 21 e ficam até o fim. Guarde a ordem: a requisição desce, a resposta sobe.

ENQUANTO ISSO, NA EMPRESA A reunião de vinte minutos

A reunião estava marcada para vinte minutos e durava uma hora e dez.

— A gente precisa de uma API — disse Roberto, no quadro branco, escrevendo a palavra API e sublinhando duas vezes, como se sublinhar resolvesse.

— Uma API para quê? — perguntou Marina.

— Para o aplicativo conversar com o sistema.

— E o que o aplicativo precisa perguntar?

Roberto ficou quieto por três segundos, o que na escala dele é uma eternidade. Depois olhou para Cláudia, que olhou para o caderno, que tinha onze post-its e nenhuma resposta.

— Produtos — arriscou Cláudia. — Ele precisa listar produtos. E cadastrar. E editar. E... apagar, acho.

— Isso tem nome — disse Marina. — Chama-se CRUD.

— Ótimo! — Roberto escreveu CRUD no quadro, sublinhou duas vezes. — Em três semanas dá, né?

Carlos, que tinha três semanas de empresa e zero linha de Java escrita, levantou a mão devagar. Ninguém viu.

1.2 CRUD: quatro verbos e nada mais

A maior parte de qualquer sistema é guardar coisa, mostrar coisa, mudar coisa e apagar coisa. Alguém batizou isso de **CRUD** — *create, read, update, delete* — e o HTTP já tinha um verbo para cada:

OPERAÇÃO	VERBO HTTP	CAMINHO	O QUE DEVOLVE
Criar	POST	/products	201 e o recurso criado
Listar	GET	/products	200 e uma página de itens
Buscar um	GET	/products/{id}	200 ou 404
Atualizar	PUT	/products/{id}	200 ou 404
Apagar	DELETE	/products/{id}	204 ou 404

TABELA 1.1 O CRUD que você vai construir na Parte 4. Decore a tabela e você decorou metade do trabalho de um backend.

★ VOCÊ SABIA

O termo CRUD apareceu em 1983, no livro *Managing the Data Base Environment*, de James Martin — quinze anos antes de existir uma API REST para chamar de sua. A ideia é mais velha que a web e vai sobreviver a ela.

1.3 Java, Spring e Spring Boot não são a mesma coisa

Essa confusão custa semanas a quem começa. Os três nomes aparecem juntos em todo tutorial e resolvem problemas diferentes:

- **Java** é a linguagem. As palavras, a gramática, os tipos. É o que você aprende nas Partes 1 e 2.
- **Spring** é um conjunto de bibliotecas que resolve o encanamento de uma aplicação: quem cria os objetos, quem liga um no outro, quem abre a transação do banco.
- **Spring Boot** é o Spring com as decisões já tomadas. Ele traz um servidor embutido, configuração por convenção e um único comando para rodar tudo.



FIGURA 1.2 Cada camada acrescenta decisões já tomadas — e cobra por elas em abstração.

A ordem importa: você não pode entender Spring sem entender objetos, e não pode entender Spring Boot sem entender Spring. É por isso que este livro gasta quatorze capítulos em Java antes de instalar o framework.

🕒 NOS BASTIDORES

O Spring nasceu em 2003 como uma reação. O padrão da época, Enterprise JavaBeans, exigia três arquivos e dois descritores XML para escrever uma classe que somasse dois números. Rod Johnson publicou um livro de mil páginas mostrando um jeito mais simples — e o código de exemplo do livro virou o framework mais usado da plataforma.

ENQUANTO ISSO, NA EMPRESA

Carlos anotou no caderno: “API = programa que responde a outro programa”.

Depois riscou e escreveu: “API = o momento em que dois departamentos da empresa finalmente tentam conversar — e descobrem que falam idiomas diferentes”.

Marina, passando por trás, leu por cima do ombro e disse que a segunda definição era melhor.

1.4 Trinta anos em uma página

Java não foi desenhado para servidores. Ele foi desenhado para eletrodomésticos, e quase todas as suas esquisitices vêm daí.



FIGURA 1.3 Por que a linguagem é assim: os marcos que ainda afetam o código que você vai escrever.

Duas datas valem uma frase a mais. **1995** é a origem da JVM, a máquina virtual que executa o bytecode — a razão pela qual o mesmo arquivo compilado roda no seu computador e no servidor. **2014** é a data em que Java deixou de ser uma linguagem só de objetos e ganhou funções como valor, que você vai usar no capítulo 14 sem nem perceber.

★ VOCÊ SABIA

O nome era **Oak**, por causa de um carvalho na frente da janela de James Gosling. O departamento jurídico achou uma marca registrada com o mesmo nome, a equipe foi tomar café e voltou com *Java* — a ilha de onde vinha o grão. É por isso que o logotipo é uma caneca fumegando.

1.5 Instalando o ambiente

Você precisa de três coisas, e a ordem é esta.

1. O JDK. Baixe o Temurin 21 (em adoptium.net) ou use o gerenciador do seu sistema. Confirme:

```
java -version
javac -version
```

2. Um editor. Qualquer um com realce de sintaxe serve para a Parte 1. A partir do capítulo 16, uma IDE ajuda de verdade — IntelliJ IDEA Community ou VS Code com a extensão de Java. Você não precisa dela ainda.

3. Uma pasta. Crie uma pasta para o livro e entre nela. Todo comando deste livro supõe que você está na raiz do projeto.

```
java-one/
└─ App.java  o próximo capítulo cria este arquivo
```

⚠ ERRO COMUM

Não instale o JDK 8 porque um vídeo antigo disse que “é o mais estável”. Metade da sintaxe deste livro — `var`, `record`, `switch` com seta — não existe lá. Se a sua empresa usa Java 8, você vai saber lidar depois de aprender a versão moderna; o contrário é bem mais difícil.

1.6 O caminho inteiro, em uma imagem

Dez partes, quarenta e dois capítulos, um projeto:

A ESPINHA DORSAL DO LIVRO

text

```
Java → OOP → HTTP → Spring Boot → REST → JPA
      → PostgreSQL → CRUD → Validação → Segurança
      → Testes → Projeto final
```

Se em algum momento você se perder, volte a esta linha e localize onde está. Cada flecha é uma decisão nova sobre o mesmo programa.

RESUMO

- Uma API é um programa que responde a outros programas, por HTTP.
- CRUD são quatro operações; o HTTP já tinha um verbo para cada uma.
- Java é a linguagem, Spring é o encanamento, Spring Boot é o Spring com as decisões prontas.
- A arquitetura do projeto tem quatro camadas: controller, service, repository, banco.

✓ PONTO DE CONTROLE

Você sabe explicar o que é uma API REST e o que é um CRUD, sabe a diferença entre Java, Spring e Spring Boot, e tem um JDK instalado que compila e executa.

■ O PROJETO AGORA

Ambiente pronto e destino conhecido. Nenhuma linha de código escrita — e isso está certo: o capítulo 1 de um projeto é sempre sobre decidir o que construir.

- 1 Escreva, com suas palavras e em três frases, o que a sua API vai fazer. Guarde o papel. No capítulo 41 você vai comparar com o que construiu.
- 2 Abra o terminal e rode `java -version`. Depois procure o número da versão na página de notas de lançamento do OpenJDK e descubra uma funcionalidade que a sua versão tem e a anterior não tinha.

CAPÍTULO

2

Primeiro programa em Java

Dois comandos, um arquivo e a linha mais famosa da programação.

AO FIM DESTE CAPÍTULO

Escrever, compilar e executar um programa Java pelo terminal, explicar cada palavra da assinatura do `main` e ler uma mensagem de erro do compilador sem entrar em pânico.

Em Python você escreve e roda. Em Java você escreve, **compila** e roda. Essa etapa extra é a primeira coisa que assusta quem vem de outra linguagem, e é também a que devolve mais valor: a compilação é onde alguém lê o seu código antes de você.

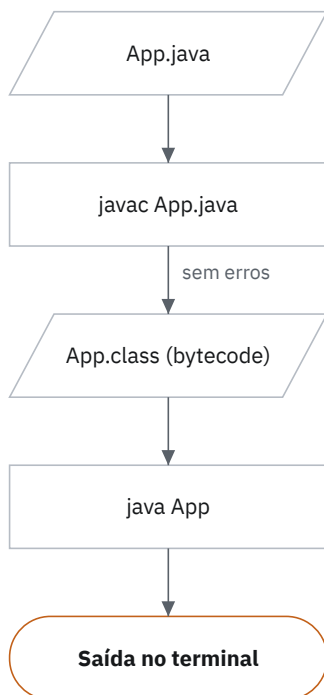


FIGURA 2.1 Do texto ao programa: o compilador entra no meio de propósito.

O arquivo `.class` não é código de máquina: é **bytecode**, um formato intermediário que qualquer JVM sabe executar. É por isso que o mesmo arquivo compilado roda no seu Windows e no servidor Linux sem recompilar — a promessa de 1995 que ainda se cumpre.

2.1 O arquivo

Crie um arquivo chamado `App.java`. O nome não é decoração: uma classe pública precisa morar em um arquivo com o mesmo nome dela.

APP.JAVA

java

```

1 public class App {
2     public static void main(String[] args) {
3         System.out.println("O primeiro programa");
4     }
5 }

```

Compile e execute:

```

javac App.java
java App

```

O primeiro comando cria `App.class` na mesma pasta. O segundo entrega esse arquivo à máquina virtual, que procura um método chamado `main` e começa por ele. Repare que `java App` não tem extensão: você não está executando um arquivo, está pedindo a uma classe que comece.

2.2 A assinatura mais copiada da história

Cinco palavras antes do nome do método, e cada uma resolve um problema concreto. Vale gastar uma página nisso porque você vai escrever essa linha até o fim da vida.

A ASSINATURA DO MAIN, PALAVRA POR PALAVRA

```

public class App {
    public static void main(String[] args) {
        System.out.println("O primeiro programa");
    }
}

```

- 1 `public class App` — a classe é o invólucro; em Java nada vive fora de uma.
- 2 `public` deixa a JVM enxergar o método de fora do arquivo.
- 3 `static` permite chamá-lo sem criar um objeto — no início do programa não existe nenhum.
- 4 `void` porque ele não devolve nada a quem chamou.
- 5 `String[] args` recebe o que você digitou depois do nome do programa.
- 6 `System.out` é a saída padrão; `println` escreve e pula a linha.

★ VOCÊ SABIA

`System.out.println` tem 18 caracteres para fazer o que Python faz com 5. A razão é consistência: `System` é uma classe, `out` é um campo dela e `println` é um método desse campo. Java preferiu não abrir exceção na gramática nem para a linha mais escrita do mundo. Em 2023, trinta anos depois, a linguagem finalmente ganhou `println` solto em classes implícitas — e a comunidade ainda discute se foi uma boa ideia.

ENQUANTO ISSO, NA EMPRESA Já está em produção?

Carlos rodou `javac App.java`. Esperou. Nada aconteceu.

Ficou olhando para o terminal por uns dez segundos, convencido de que tinha travado. Digitou `java App`. E então, na tela preta, apareceu:

```
O primeiro programa
```

Ele girou na cadeira e anunciou, alto demais para um escritório aberto:

— COMPILOU!

Roberto, que passava com um café na mão, parou.

— Compilou? Então já está em produção?

— Não. Ele imprime uma frase.

— Uma frase para o cliente?

— Uma frase para mim.

Roberto assentiu devagar, do jeito de quem não entendeu mas vai repetir a informação na próxima reunião como se tivesse entendido.

2.3 Argumentos: o programa recebendo o mundo

`String[] args` não é enfeite. Troque o corpo do `main`:

SAUDACAO.JAVA

java

```
1 public class Saudacao {
2     public static void main(String[] args) {
3         System.out.println("Olá, " + args[0]);
4     }
5 }
```

```
javac Saudacao.java
java Saudacao Ana
```

SAÍDA

text

```
Olá, Ana
```

Você acabou de escrever um programa com entrada. É pouco, mas é a diferença entre um exercício e uma ferramenta.

ERRO COMUM

Rode `java Saudacao` sem o nome. A resposta é

`ArrayIndexOutOfBoundsException: Index 0 out of bounds for length 0` — você pediu o primeiro item de uma lista vazia. Guarde essa mensagem: ela vai voltar no capítulo 6, e sempre diz exatamente qual índice você tentou usar.

2.4 Quando o compilador reclama

Apague o ponto e vírgula da linha 3 e compile de novo:

TERMINAL

text

```
Saudacao.java:3: error: ';' expected
    System.out.println("Olá, " + args[0])
                                ^
1 error
```

Quatro informações em três linhas: o arquivo, a linha, o que faltava e uma seta apontando o lugar exato. Um erro de compilação custa dez segundos. O mesmo defeito, em uma linguagem sem essa etapa, custaria uma execução quebrada na frente de um usuário.

O ESSENCIAL

Leia a **primeira** mensagem, não a última. Um erro de sintaxe confunde o compilador, que passa a reclamar de coisas certas logo abaixo. Conserte o primeiro, compile de novo, e metade da lista desaparece.

2.5 Duas palavras que você vai ouvir muito

JDK *Java Development Kit*: o pacote que compila e executa. Traz o `javac`, o `java`, o depurador e as bibliotecas.

JRE *Java Runtime Environment*: só executa. Não tem `javac`. Foi distribuído separadamente por anos e é a causa da metade das instalações frustradas de quem começa.

NOS BASTIDORES

Até o Java 8, instalar Java no desktop instalava um JRE com um plugin de navegador — a origem daquelas atualizações eternas do “Java Update”. Os applets morreram, o plugin foi removido no Java 11 e hoje a distribuição padrão é um JDK. Se algum tutorial mandar você baixar “o JRE”, ele é de outra década.

2.6 E a IDE?

Nada até aqui exigiu uma. Isso é deliberado: o botão verde de uma IDE esconde exatamente as duas etapas que você precisava ver. A partir do

capítulo 16, quando o projeto passa a ter dezenas de arquivos e dependências, uma IDE deixa de ser conforto e passa a ser ferramenta — e o livro vai usar uma.

```
java-one/
├── App.java    imprime uma linha
├── App.class   gerado pelo javac
└── Saudacao.java recebe um argumento
```

RESUMO

- Java compila antes de rodar: `javac` produz bytecode, `java` executa.
- Uma classe pública mora em um arquivo com o mesmo nome dela.
- `main` é onde a JVM começa, e cada palavra da assinatura resolve um problema concreto.
- O erro de compilação é a revisão mais barata que existe. Leia a primeira mensagem.

✓ PONTO DE CONTROLE

Você escreve, compila e executa um programa Java pelo terminal, sabe explicar `public static void main(String[] args)` e sabe o que fazer quando o `javac` reclama.

■ O PROJETO AGORA

O projeto tem uma classe que imprime e uma que recebe argumento. Nenhuma das duas sobrevive ao capítulo 16 — mas o hábito de compilar antes de rodar sobrevive ao livro.

- 1 Escreva um `Soma.java` que receba dois números como argumento e imprima a soma. Dica: `Integer.parseInt(args[0])` transforma texto em número.
- 2 Estrague o programa de propósito três vezes: troque `String[]` por `String`, apague uma chave `}` e escreva `System.out.println`. Leia as três mensagens antes de consertar.

CAPÍTULO

3

Variáveis e tipos

Declarar um tipo é contratar um revisor que trabalha de graça.

AO FIM DESTA CAPÍTULO

Declarar variáveis dos cinco tipos do dia a dia, prever o resultado de uma conversão e saber quando usar `var` sem perder a proteção do compilador.

**Capítulo disponível na
versão completa**

4

Operadores e expressões

O sinal de igual não significa igual, e o sinal de igual duplo engana com texto.

AO FIM DESTA CAPÍTULO

Prever o resultado de qualquer expressão aritmética, lógica ou de comparação, e explicar por que `==` não serve para comparar texto em Java.

—

**Capítulo disponível na
versão completa**

—

CAPÍTULO

5

Condicionais

Escolher um caminho é assumir o outro — inclusive quando ele não está escrito.

AO FIM DESTA CAPÍTULO

Escrever decisões com `if`, `else if` e `switch`, escolher entre eles com critério e reconhecer o caminho que você deixou implícito.

—

**Capítulo disponível na
versão completa**

—

6

Repetições

Escrever a mesma linha dez vezes é um erro de projeto, não de digitação.

AO FIM DESTA CAPÍTULO

Escolher entre `for`, `for-each` e `while` conforme o problema, escrever um acumulador correto e sair de um laço sem quebrar a lógica.

—

**Capítulo disponível na
versão completa**

—

PARTE 2

O Java que você realmente precisa

Não a linguagem toda — a parte dela que sustenta uma API. Métodos, coleções, objetos, exceções e streams, na dose exata.

NESTA PARTE

- 7 Métodos
- 8 Arrays e coleções
- 9 Orientação a objetos
- 10 Encapsulamento
- 11 Herança e interfaces
- 12 Enums e records
- 13 Exceções
- 14 Lambdas e streams

CAPÍTULO

7

Métodos

Dar nome a um trecho de código é a forma mais barata de documentação que existe.

AO FIM DESTES CAPÍTULOS

Extraír um método com parâmetros e retorno, entender o que é escopo e reconhecer quando uma sobrecarga ajuda e quando ela confunde.

—

**Capítulo disponível na
versão completa**

—

CAPÍTULO

8

Arrays e coleções

Array tem tamanho. Lista tem vida. Mapa tem chave. Escolher errado custa caro.

AO FIM DESTES CAPÍTULO

Escolher entre `List`, `Set` e `Map` pelo problema, percorrer cada um deles e explicar por que se declara a interface e não a implementação.

**Capítulo disponível na
versão completa**

CAPÍTULO

9

Orientação a objetos

Até aqui o código tinha verbos soltos. Agora ele ganha substantivos.

“Eu inventei o termo orientação a objetos, e posso dizer que não tinha C++ em mente.”

ALAN KAY, CRIADOR DO SMALLTALK

AO FIM DESTE CAPÍTULO

Escrever uma classe com atributos, construtor e métodos, criar objetos a partir dela e explicar a diferença entre classe e instância.

—

**Capítulo disponível na
versão completa**

—

CAPÍTULO

10

Encapsulamento

Esconder o atributo não é cerimônia: é o que impede o objeto de existir em estado impossível.

AO FIM DESTES CAPÍTULOS

Proteger o estado de um objeto com `private`, expor o que for necessário com métodos, e validar no construtor para que o objeto nunca nasça inválido.

**Capítulo disponível na
versão completa**

11

Herança e interfaces

Herança é um parentesco que você não pode desfazer. Interface é um contrato que você pode assinar com quem quiser.

AO FIM DESTA CAPÍTULO

Escrever uma hierarquia com `extends`, declarar e implementar uma interface, usar polimorfismo e decidir entre herança e composição.

—

**Capítulo disponível na
versão completa**

—

CAPÍTULO

12

Enums e records

Dois recursos que apagam código: um para o conjunto fechado, outro para o dado puro.

AO FIM DESTE CAPÍTULO

Substituir constantes de texto por `enum`, escrever um `record` em uma linha e explicar por que os dois deixam o compilador trabalhar mais.

—

**Capítulo disponível na
versão completa**

—

CAPÍTULO

13

Exceções

Um erro tratado é um requisito. Um erro engolido é uma dívida com juros.

“Eu chamo isso de erro de um bilhão de dólares. A invenção da referência nula, em 1965.”

TONY HOARE, CRIADOR DO QUICKSORT

AO FIM DESTES CAPÍTULOS

Lançar e capturar exceções, escolher entre *checked* e *unchecked*, criar uma exceção própria e nunca mais escrever um `catch` vazio.

—

**Capítulo disponível na
versão completa**

—

CAPÍTULO

14

Lambdas e streams

O laço diz como percorrer. O stream diz o que você quer. A segunda frase é mais curta.

AO FIM DESTE CAPÍTULO

Escrever uma lambda, encadear `filter`, `map`, `sorted` e `collect`, e reconhecer quando um stream ajuda e quando ele atrapalha.

—

**Capítulo disponível na
versão completa**

—

PARTE 3

Entrando no Spring Boot

*O framework que escreve o encanamento por você.
E o preço disso: entender quem está chamando o
seu código.*

NESTA PARTE

- 15 O que é Spring
- 16 O primeiro projeto Spring Boot
- 17 A primeira API

CAPÍTULO

15

O que é Spring

Você para de criar objetos com new. Em troca, alguém cria por você — e é preciso entender quem.

“Um framework é uma biblioteca que chama o seu código, em vez de ser chamada por ele.”

MARTIN FOWLER, SOBRE INVERSÃO DE CONTROLE

AO FIM DESTES CAPÍTULOS

Explicar inversão de controle e injeção de dependência com um exemplo de código, e dizer o que é um *bean* e o que é o contêiner.

Até o capítulo 14 você criou todos os objetos do programa. A partir daqui, quem cria é o framework. Essa inversão tem nome, tem motivo e tem um preço — e este capítulo é sobre os três. Não vamos instalar nada ainda: primeiro a ideia.

15.1 O problema que o Spring resolve

Considere a estrutura que você montou no capítulo 11:

QUEM MONTA ESSAS PEÇAS?	java
<pre> 1 public class ProductService { 2 private final ProductRepository repository; 3 4 public ProductService(ProductRepository repository) { 5 this.repository = repository; 6 } 7 } </pre>	

Alguém precisa escrever:

A MONTAGEM MANUAL	java
<pre> 1 var dataSource = new PostgresDataSource(url, user, pass); 2 var repository = new JdbcProductRepository(dataSource); 3 var service = new ProductService(repository); 4 var controller = new ProductController(service); </pre>	

Com quatro classes é aceitável. Uma API real tem quarenta: serviço de e-mail, cliente HTTP, cache, transação, autenticação, e cada um depende de outros três. Essa montagem passa a ser um arquivo de trezentas linhas que ninguém quer abrir — e que precisa ser feito em cada teste.

● O ESSENCIAL

O Spring não faz nada que você não poderia fazer à mão. Ele faz a **montagem** por você, sempre do mesmo jeito, e cuida da ordem. É um construtor de grafos de objetos, não um poder mágico.

ENQUANTO ISSO, NA EMPRESA A palavra framework

Roberto voltou de um evento com uma palavra nova e uma certeza.

— A gente precisa de um framework.

— Para quê? — perguntou Marina.

— Para acelerar. O cara do painel disse que reduz em 70% o tempo de desenvolvimento.

— Reduz o tempo de escrever o encanamento. A gente ainda não escreveu nem o encanamento nem o resto.

Roberto anotou “70%” no caderno e circulou. Duas semanas depois, aquele número apareceu em um slide, sem contexto, ao lado do prazo de entrega.

Marina fez o que fazia sempre: abriu o projeto e mostrou a Carlos as quatro linhas de montagem manual que existiam no `main`.

— Isso aqui é o que o framework vai fazer por nós. Não é pouco. Mas também não é 70% de nada.

15.2 Inversão de controle

A expressão descreve exatamente a troca:

```
CONTROLE NO SEU CÓDIGO
class ProductService {
    private Repo repo =
        new JdbcRepo();
}
```

```
CONTROLE INVERTIDO
class ProductService {
    private final Repo repo;

    ProductService(Repo repo) {
        this.repo = repo;
    }
}
```

À esquerda, a classe decide qual implementação usar — e fica presa a ela para sempre. À direita, a classe **declara o que precisa** e recebe de fora. Quem decide passa a ser quem monta.

Essa segunda forma tem nome próprio: **injeção de dependência por construtor**. Ela é a única que este livro usa, e a recomendada oficialmente pelo Spring, por três razões: o objeto nasce completo, o atributo pode ser `final`, e a classe continua testável sem framework nenhum.

15.3 O contêiner e os *beans*

O QUE O SPRING LÊ

java

```
1  @Service
2  public class ProductService {
3      private final ProductRepository repository;
4
5      public ProductService(ProductRepository repository) {
6          this.repository = repository;
7      }
8  }
```

Uma anotação, e o acordo muda. `@Service` diz ao Spring: “*esta classe é minha, crie uma instância dela e guarde*”. Essa instância guardada é um

bean, e o lugar onde ela fica é o **contêiner** (formalmente, *application context*).



FIGURA 15.1 O contêiner monta o grafo na partida, olhando as anotações e os construtores.

Na partida, o Spring varre o pacote, encontra as classes anotadas, olha os construtores, descobre em que ordem criar cada uma e monta tudo. Se faltar uma peça, ele falha **na partida** — não na requisição do cliente.

COMO O SPRING DECIDE O QUE INJETAR

```

@Service
public class ProductService {
    private final ProductRepository repo;

    public ProductService(ProductRepository repo) {
        this.repo = repo;
    }
}
  
```

- ❶ A anotação marca a classe como candidata a bean.
- ❷ `final` garante que a dependência não muda depois de montada.
- ❸ Um único construtor: o Spring nem precisa de `@Autowired` desde a versão 4.3.
- ❹ O **tipo** do parâmetro é a chave da busca: ele procura um bean que sirva como `ProductRepository`.

15.4 Quatro anotações que dizem a mesma coisa

ANOTAÇÃO	CAMADA	DIFERENÇA REAL
@Component	qualquer	a forma genérica
@Service	regra de negócio	nenhuma técnica; comunica intenção
@Repository	acesso a dados	traduz exceções do banco
@Controller	entrada HTTP	habilita o mapeamento de rotas

TABELA 15.1 Todas criam um bean. Escolher a certa é documentação — e, em dois casos, também comportamento.

⚠ ERRO COMUM

@Autowired em atributo (*field injection*) ainda aparece em muito tutorial:

```
@Autowired
private ProductRepository repository; // evite
```

Funciona e cobra caro: o atributo não pode ser `final`, a classe mente sobre suas dependências (o construtor não as declara) e o teste unitário precisa de reflexão para preencher o campo. Use o construtor. Sempre.

15.5 Escopo: quantos objetos existem

Por padrão, um bean é **singleton**: existe **um** para a aplicação inteira, e todas as requisições usam o mesmo objeto. Isso tem uma consequência que assusta quem descobre tarde:

⚠ ERRO COMUM

Um bean singleton **não pode guardar estado de requisição**. Se o

`ProductService` tiver um atributo `private Product atual`, duas requisições simultâneas vão sobrescrever uma à outra — e o cliente A receberá o dado do cliente B. A regra é simples: **os atributos de um bean são só as suas dependências**. Dado de requisição vive em parâmetro e variável local.

15.6 O que exatamente o Spring Boot acrescenta

O Spring Framework resolve a montagem. O Spring Boot resolve a configuração:

- **servidor embutido:** o Tomcat vem dentro do JAR; não há servidor para instalar;
- **autoconfiguração:** viu o PostgreSQL no classpath? configura a conexão; viu o Jackson? configura JSON;
- `application.properties`: um arquivo, chaves com nome previsível;
- **arranque único:** `java -jar app.jar` e está no ar.

☹ **NOS BASTIDORES**

Em 2003, escrever a mesma aplicação exigia um arquivo `web.xml`, dois descritores `ejb-jar.xml`, um servidor de aplicação instalado à parte e um ciclo de *build-deploy* de vários minutos. O Spring Boot é de 2014 e a proposta era cortar isso: convenção em vez de configuração. A comunidade chamou de “opinionated” — o framework tem opinião, e você só discute quando precisa.

15.7 Onde a mágica cobra o preço

Vale dizer isto agora, antes de o encantamento começar:

- o erro vira longo. Uma pilha de exceção do Spring tem sessenta linhas, e as três que importam estão no meio;
- a ordem de criação é implícita. Quando duas classes dependem uma da outra, o erro é `circular reference` e a solução é repensar o desenho;
- a anotação esconde comportamento. `@Transactional` muda o que acontece em volta do método sem aparecer no corpo dele — e o capítulo 30 mostra o que isso significa na prática.

Nada disso é motivo para não usar. É motivo para saber o que está acontecendo, que é a diferença entre usar o framework e ser usado por ele.

Bean Um objeto cujo ciclo de vida é controlado pelo Spring: ele cria, injeta as dependências, guarda e destrói.

Contêiner (ApplicationContext) O registro de todos os beans da aplicação. Monta o grafo na partida e entrega a instância a quem pedir pelo tipo.

RESUMO

- Inversão de controle: a classe declara o que precisa; outro decide o que entregar.
- Injeção por construtor é a única forma recomendada — atributo `final`, objeto completo, teste sem framework.
- `@Service`, `@Repository`, `@Controller` e `@Component` criam beans.
- Bean é singleton por padrão: não guarde estado de requisição nele.

- Spring monta; Spring Boot configura.

✓ PONTO DE CONTROLE

Você explica inversão de controle com um exemplo, sabe por que injeção por construtor vence, reconhece as quatro anotações e sabe por que um bean não pode guardar estado de requisição.

■ O PROJETO AGORA

Nenhuma linha nova de projeto, mas a estrutura do capítulo 11 acabou de ganhar sentido: `ProductService` recebendo `ProductRepository` é exatamente o formato que o Spring espera encontrar. O capítulo 16 instala o framework.

- 1 Reescreva a montagem manual de quatro linhas deste capítulo explicando, em uma frase por linha, qual dependência cada objeto recebe.
- 2 Escreva duas classes, `EmailSender` (interface), `SmtEmailSender` e `FakeEmailSender`, e uma `OrderService` que receba a interface no construtor. Monte as duas combinações à mão, sem framework.
- 3 Explique, sem usar a palavra “mágica”, o que acontece entre `java -jar app.jar` e a primeira requisição atendida.

CAPÍTULO

16

O primeiro projeto Spring Boot

Três minutos entre um formulário na web e uma aplicação rodando na porta 8080.

AO FIM DESTES CAPÍTULOS

Gerar um projeto no Spring Initializr, entender cada pasta e cada linha do `pom.xml`, e subir a aplicação pela primeira vez.

—

**Capítulo disponível na
versão completa**

—

CAPÍTULO

17

A primeira API

Quatro verbos, um caminho e uma decisão sobre o que devolver em cada caso.

AO FIM DESTE CAPÍTULO

Escrever endpoints para os cinco verbos do CRUD, receber dados pelo caminho, pela query e pelo corpo, e escolher o código de status correto.

—

**Capítulo disponível na
versão completa**

—

PARTE 4

Construindo o CRUD

A espinha dorsal do livro. Entidade, banco, repositório, serviço, controlador e DTO — nesta ordem, uma camada por vez.

NESTA PARTE

- 18 Modelando a aplicação
- 19 Banco de dados
- 20 Spring Data JPA
- 21 Repository
- 22 Service
- 23 Controller
- 24 DTOs

CAPÍTULO

18

Modelando a aplicação

Antes de escolher a tecnologia, decidir os substantivos. Um modelo errado sobrevive a todas as refatorações.

AO FIM DESTES CAPÍTULOS

Identificar entidades, atributos e relações de um domínio, escolher a chave primária e justificar cada tipo de campo.

**Capítulo disponível na
versão completa**

CAPÍTULO

19

Banco de dados

Um lugar onde o dado sobrevive ao desligamento da máquina — e uma linguagem de cinquenta anos que ninguém conseguiu substituir.

AO FIM DESTES CAPÍTULOS

Subir um PostgreSQL, criar tabelas, escrever as quatro instruções do CRUD em SQL e explicar chave primária, chave estrangeira e índice.

—

**Capítulo disponível na
versão completa**

—

CAPÍTULO

20

Spring Data JPA

Uma camada que traduz objeto em linha de tabela. Poderosa, conveniente e cheia de armadilhas que este livro vai nomear.

AO FIM DESTES CAPÍTULOS

Anotar uma classe como entidade, mapear campos e enum, entender o que é o Hibernate e o que é o JPA, e saber quando a tradução automática atrapalha.

Você tem uma classe Java e uma tabela SQL descrevendo a mesma coisa. Escrever à mão a conversão entre as duas — `SELECT` em `Product`, `Product` em `INSERT` — é trabalho repetitivo e mecânico. O JPA faz isso.

20.1 Três nomes, três papéis

NOME	O QUE É
JPA	a especificação: as anotações e o contrato
Hibernate	a implementação que executa (a padrão do Boot)
Spring Data JPA	a camada que dispensa escrever o repositório

TABELA 20.1 Você programa contra o JPA, o Hibernate faz o trabalho, e o Spring Data poupa o código repetitivo. Confundir os três é comum e atrapalha na hora de procurar uma solução na internet.

★ VOCÊ SABIA

Antes do JPA (2006), cada projeto escrevia sua própria camada de acesso a dados, ou usava a API crua do JDBC — em que uma consulta de três colunas ocupava vinte linhas com `ResultSet`, `try/finally` e conversão manual de tipo. O Hibernate apareceu em 2001 como projeto independente, ficou tão dominante que a especificação oficial foi escrita **depois**, em cima dele. É um dos poucos casos em que o padrão seguiu a prática.

20.2 A entidade

PRODUCT.JAVA

java

```
1 package com.loja.catalog.product;
2
3 import jakarta.persistence.*;
4 import java.math.BigDecimal;
5 import java.time.Instant;
6
7 @Entity
8 @Table(name = "product")
9 public class Product {
10
11     @Id
12     @GeneratedValue(strategy = GenerationType.IDENTITY)
13     private Long id;
14
15     @Column(nullable = false, length = 120)
16     private String name;
17
18     @Column(columnDefinition = "TEXT")
19     private String description;
```

```

20
21     @Column(nullable = false, precision = 10, scale = 2)
22     private BigDecimal price;
23
24     @Column(nullable = false)
25     private Integer quantity = 0;
26
27     @Enumerated(EnumType.STRING)
28     @Column(nullable = false, length = 20)
29     private Status status = Status.ATIVO;
30
31     @Column(name = "created_at", nullable = false, updatable =
false)
32     private Instant createdAt = Instant.now();
33
34     protected Product() { }    // exigido pelo JPA
35
36     public Product(String name, BigDecimal price, Integer
quantity) {
37         this.name = name;
38         this.price = price;
39         this.quantity = quantity;
40     }
41
42     // getters e setters omitidos
43 }
    
```

AS ANOTAÇÕES QUE FAZEM A TRADUÇÃO

```

@Entity
@Table(name = "product")
public class Product {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Enumerated(EnumType.STRING)
    private Status status;
}
    
```

- ❶ @Entity diz ao Hibernate: esta classe vira linha de tabela.
- ❷ @Table nomeia a tabela. Sem ela, o nome da classe é usado.
- ❸ @Id marca a chave primária — obrigatória em toda entidade.
- ❹ IDENTITY delega a geração ao BIGSERIAL do banco.
- ❺ EnumType.STRING grava 'ATIVO'. Nunca use ORDINAL.

⚡ ERRO COMUM

`@Enumerated(EnumType.ORDINAL)` — que é o **padrão** quando você não declara nada — grava a **posição** do enum: 0, 1, 2. Se alguém inserir um valor novo no meio da lista, todos os registros do banco passam a significar outra coisa, silenciosamente. Declare sempre `EnumType.STRING`. Este é, sem concorrência, o defeito mais caro deste capítulo.

20.3 Por que a entidade precisa de um construtor vazio

Aquele `protected Product() { }` parece inútil e é obrigatório: o Hibernate cria o objeto por reflexão, sem saber nada dos seus construtores, e depois preenche os campos. `protected` é o suficiente — não precisa ser público, e assim ninguém cria um produto vazio por acidente.

É também o motivo pelo qual **entidade não pode ser um** `record` (capítulo 12): `record` não tem construtor vazio nem setters.

ENQUANTO ISSO, NA EMPRESA O dia em que INATIVO virou PROMOCAO

O enum tinha três valores e estava gravado como número no banco — a configuração padrão, que ninguém tinha escolhido conscientemente.

O zero era `ATIVO`, o um era `INATIVO`, o dois era `ESGOTADO`.

Em março, Cláudia pediu um status novo: `PROMOCAO`. Carlos abriu o enum e acrescentou o valor onde fazia sentido semanticamente — entre `ATIVO` e `INATIVO`, porque produto em promoção está mais perto de ativo do que de inativo.

O deploy foi na quarta. Na quinta, quatro mil produtos inativos apareceram na vitrine com etiqueta de promoção.

Ninguém tinha alterado dado nenhum. Os números no banco continuavam exatamente os mesmos. O que mudou foi o significado deles.

20.4 O ciclo de vida de uma entidade

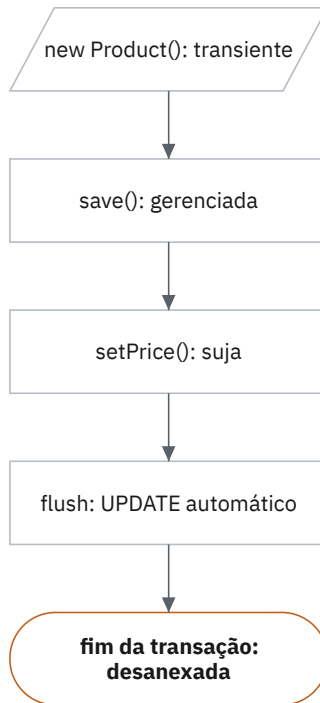


FIGURA 20.1 Os quatro estados de uma entidade — e por que uma alteração pode ser gravada sem você mandar.

Este é o comportamento que mais surpreende quem chega: dentro de uma transação, alterar um objeto **gerenciado** gera um `UPDATE` ao final, mesmo sem você chamar `save`. O Hibernate compara o estado atual com o que leu do banco (*dirty checking*) e grava a diferença.

● O ESSENCIAL

Entidade gerenciada não é um objeto comum: é um objeto que o Hibernate está vigiando. É por isso que o capítulo 24 insiste em não deixar a entidade sair da camada de serviço — fora dela, ninguém sabe se uma alteração vai virar `UPDATE` ou não.

20.5 Conferindo o mapeamento

APPLICATION.PROPERTIES	properties
<pre>spring.jpa.hibernate.ddl-auto=validate spring.jpa.show-sql=true spring.jpa.properties.hibernate.format_sql=true</pre>	

`validate` compara a entidade com a tabela na partida e **falha se não casarem**. É a configuração que você quer: se você acrescentou um campo na classe e esqueceu a coluna, descobre em dois segundos, não em produção.

SAÍDA COM SHOW-SQL, AO SALVAR	text
<pre>Hibernate: insert into product (created_at, description, name, price, quantity, status) values (?, ?, ?, ?, ?, ?)</pre>	

◆ DICA

Deixe `show-sql=true` durante todo o desenvolvimento. Ver o SQL que o Hibernate gera é o hábito que evita o problema N+1 do capítulo 30 — e é a única forma de perceber que uma listagem simples disparou 200 consultas.

20.6 Quando a tradução automática atrapalha

O JPA é ótimo para operações por entidade — carregar um produto, salvar, apagar. Ele é ruim para:

- **relatórios** com agregação e junção de cinco tabelas;
- **atualização em massa** (`UPDATE product SET status = 'INATIVO'` em um milhão de linhas);
- consultas em que a **forma do resultado** não é uma entidade.

Para esses casos, Spring oferece o `JdbcTemplate` e consultas nativas — e usar SQL direto não é derrota, é escolher a ferramenta certa. Uma aplicação madura tem as duas coisas.

☹ NOS BASTIDORES

A crítica organizada a esse tipo de ferramenta tem nome: *object-relational impedance mismatch*. Objetos têm herança e referências; tabelas têm chaves e junções. A tradução funciona em 90% dos casos e cobra os 10% restantes com juros — o famoso “o ORM está lento” que, quase sempre, é o ORM fazendo exatamente o que foi mandado.

Entidade gerenciada Objeto que o Hibernate está acompanhando dentro de uma transação. Alterações nele são gravadas automaticamente ao final.

RESUMO

- JPA é a especificação; Hibernate executa; Spring Data poupa código.
- `@Entity`, `@Id`, `@GeneratedValue` e `@Column` fazem o mapeamento.
- Use sempre `@Enumerated(EnumType.STRING)`.
- Entidade precisa de construtor vazio — e por isso não pode ser `record`.
- Dentro da transação, alterar entidade gerenciada gera `UPDATE` automático.
- `ddl-auto=validate` e `show-sql=true` são os padrões saudáveis.

✓ PONTO DE CONTROLE

Você anota uma entidade completa, sabe qual anotação faz o quê, explica o ciclo de vida de uma entidade e reconhece os casos em que o JPA não é a ferramenta certa.

■ O PROJETO AGORA

`Product` agora é uma entidade JPA mapeada para a tabela `product`. Falta quem execute as consultas — e o próximo capítulo mostra que isso é uma interface vazia.

- 1 Anote a classe `Category` como entidade, com `name` único e obrigatório. Confirme com `ddl-auto=validate` que a tabela casa com a classe.
- 2 Acrescente um campo `updatedAt` que seja preenchido automaticamente a cada alteração. Descubra `@PreUpdate`.
- 3 Mude `@Enumerated` para `ORDINAL`, salve dois produtos, acrescente um valor no **meio** do enum e leia os dados de novo. Descreva o que aconteceu.

CAPÍTULO

21

Repository

*Uma interface vazia que ganha vinte métodos.
Vale entender de onde eles vêm antes de confiar
neles.*

AO FIM DESTE CAPÍTULO

Criar um repositório com `JpaRepository`, usar os métodos prontos, escrever *query methods* pelo nome e explicar quem implementa a interface.

**Capítulo disponível na
versão completa**

CAPÍTULO

22

Service

A camada onde mora a regra. Se ela estiver vazia, a regra está espalhada em outro lugar — e você vai encontrá-la do jeito difícil.

AO FIM DESTE CAPÍTULO

Mover a regra de negócio do controlador para um serviço, injetar o repositório por construtor e usar `@Transactional` sabendo o que ele faz.

**Capítulo disponível na
versão completa**

CAPÍTULO

23

Controller

A camada mais fina do sistema — e a única que o cliente vê.

AO FIM DESTES CAPÍTULOS

Escrever o CRUD completo com as três camadas, devolver o status correto em cada caso e testar a API inteira pela linha de comando.

**Capítulo disponível na
versão completa**

CAPÍTULO

24

DTOs

O cliente não precisa saber como a sua tabela é desenhada. E você não quer que ele dependa disso.

AO FIM DESTES CAPÍTULOS

Separar entidade de contrato com records de entrada e saída, converter entre os dois e listar três problemas concretos que a exposição direta causa.

**Capítulo disponível na
versão completa**

PARTE 5

API de verdade

O que separa um CRUD de exercício de uma API que outra pessoa usa: validação, erro honesto, paginação e busca.

NESTA PARTE

- 25 Validação
- 26 Tratamento de erros
- 27 Paginação e ordenação
- 28 Filtros e buscas

CAPÍTULO

25

Validação

*Toda entrada é hostil até prova em contrário —
e a prova é uma anotação.*

AO FIM DESTES CAPÍTULOS

Validar entrada com Bean Validation, ligar a validação com `@Valid`, escrever uma restrição própria e decidir o que validar em cada camada.

—

**Capítulo disponível na
versão completa**

—

CAPÍTULO

26

Tratamento de erros

O erro é parte do contrato. Uma API que responde 500 para tudo é uma API que não conta o que aconteceu.

AO FIM DESTES CAPÍTULOS

Traduzir exceções de negócio em status HTTP com `@ControllerAdvice`, montar uma resposta de erro padronizada e escolher entre 400, 404, 409 e 500 com critério.

**Capítulo disponível na
versão completa**

CAPÍTULO

27

Paginação e ordenação

`GET /products` com um milhão de itens é um ataque de negação de serviço que você escreveu sozinho.

AO FIM DESTA CAPÍTULO

Paginar e ordenar uma listagem com `Pageable`, entender o custo de `OFFSET` e devolver metadados de paginação ao cliente.

**Capítulo disponível na
versão completa**

CAPÍTULO

28

Filtros e buscas

Filtrar no banco é uma consulta. Filtrar em memória é trazer tudo e jogar quase tudo fora.

AO FIM DESTES CAPÍTULOS

Escrever filtros combináveis com `Specification`, entender por que `LIKE '%termo%'` ignora o índice e decidir entre *query method*, `@Query` e busca dinâmica.

**Capítulo disponível na
versão completa**

PARTE 6

Relacionamentos

Quando as tabelas se conhecem, o JPA fica poderoso e perigoso na mesma medida. Aqui está o mapa dos buracos.

NESTA PARTE

- 29 Relacionamentos JPA
- 30 Problemas reais do JPA

CAPÍTULO

29

Relacionamentos JPA

Duas tabelas que se conhecem viram duas classes que se apontam — e é aí que o JPA fica poderoso e perigoso na mesma medida.

AO FIM DESTES CAPÍTULO

Mapear @ManyToOne, @OneToMany e @ManyToMany, escolher o lado dono da relação e evitar as três armadilhas clássicas do mapeamento bidirecional.

**Capítulo disponível na
versão completa**

CAPÍTULO

30

Problemas reais do JPA

O ORM faz exatamente o que você mandou. O problema é que você mandou sem saber.

“Toda abstração não trivial vaza.”

JOEL SPOLSKY, LEI DAS ABSTRAÇÕES VAZADAS

AO FIM DESTE CAPÍTULO

Diagnosticar o problema N+1 pelo log, escolher entre JOIN FETCH e @EntityGraph, e explicar LazyInitializationException sem adivinhação.

—

**Capítulo disponível na
versão completa**

—

PARTE 7

Segurança

Autenticação, autorização e um token que diz quem você é sem que o servidor precise lembrar.

NESTA PARTE

- 31 Spring Security
- 32 JWT
- 33 Usuários e permissões

CAPÍTULO

31

Spring Security

A API está no ar há três capítulos e qualquer pessoa com a URL apaga qualquer produto.

AO FIM DESTE CAPÍTULO

Adicionar Spring Security ao projeto, entender a cadeia de filtros, configurar quais rotas são públicas e guardar senha do jeito certo.

**Capítulo disponível na
versão completa**

CAPÍTULO

32

JWT

Um crachá assinado que o servidor não precisa guardar — e que ele também não consegue cancelar.

AO FIM DESTES CAPÍTULOS

Emitir e validar um token JWT, entender o que é assinatura e o que é criptografia, e escolher o tempo de expiração com consciência do que isso significa.

**Capítulo disponível na
versão completa**

CAPÍTULO

33

Usuários e permissões

Saber quem é a pessoa é metade do trabalho. A outra metade é decidir o que ela pode fazer.

AO FIM DESTES CAPÍTULO

Proteger rotas por papel com `@PreAuthorize` e por regra de dono do recurso, e saber por que autorização também mora no serviço.

**Capítulo disponível na
versão completa**

PARTE 8

Testes

*A rede que permite mexer no código sem medo.
Unidade, integração e um banco de verdade
dentro do teste.*

NESTA PARTE

- 34 Testando Java
- 35 Testando services
- 36 Testando controllers
- 37 Testando o banco

CAPÍTULO

34

Testando Java

Teste não é sobre provar que funciona. É sobre poder mexer no código sem medo na sexta-feira.

AO FIM DESTES CAPÍTULOS

Escrever testes com JUnit 5 e AssertJ, nomear um teste de forma que ele documente a regra, e saber o que não vale a pena testar.

—

**Capítulo disponível na
versão completa**

—

CAPÍTULO

35

Testando services

Para testar a regra, o banco precisa sair do caminho — e alguém precisa fingir ser ele.

AO FIM DESTE CAPÍTULO

Isolar o serviço com Mockito, verificar chamadas com `verify`, e reconhecer quando um mock deixou de ajudar e passou a testar a si mesmo.

—

**Capítulo disponível na
versão completa**

—

CAPÍTULO

36

Testando controllers

O contrato HTTP é a única parte do sistema que outra pessoa depende. É a que mais merece um teste.

AO FIM DESTES CAPÍTULOS

Testar endpoints com MockMvc, verificar status, cabeçalho e JSON, e simular um usuário autenticado sem fazer login.

—

**Capítulo disponível na
versão completa**

—

CAPÍTULO

37

Testando o banco

Testar contra um banco diferente do de produção é ensaiar a peça em outro teatro.

AO FIM DESTES CAPÍTULOS

Testar repositórios com `@DataJpaTest`, subir um PostgreSQL de verdade no teste com Testcontainers e entender por que o H2 engana.

**Capítulo disponível na
versão completa**

PARTE 9

Documentação e ferramentas

Uma API que ninguém sabe usar não existe. E um projeto sem histórico não tem volta.

NESTA PARTE

- 38 Swagger e OpenAPI
- 39 Clientes HTTP
- 40 Git

CAPÍTULO

38

Swagger e OpenAPI

Uma API que ninguém sabe usar não existe. E documentação escrita à mão envelhece em uma semana.

AO FIM DESTES CAPÍTULOS

Gerar documentação OpenAPI a partir do código, enriquecê-la com anotações e entender por que documentação separada do código sempre mente.

**Capítulo disponível na
versão completa**

CAPÍTULO

39

Clientes HTTP

A ferramenta importa menos que o hábito de guardar a requisição junto do código que a atende.

AO FIM DESTES CAPÍTULOS

Exercitar a API com `curl`, arquivos `.http` e coleções do Postman, e organizar isso de um jeito que sobreviva à saída de quem escreveu.

—

**Capítulo disponível na
versão completa**

—

CAPÍTULO

40

Git

O código diz o que o sistema faz. O histórico diz por quê — e é a única parte que você não consegue reconstruir.

AO FIM DESTE CAPÍTULO

Versionar o projeto com commits que explicam a intenção, trabalhar com ramos e *pull request*, e resolver um conflito sem pânico.

Este é o capítulo que deveria ter sido o primeiro e é o penúltimo, de propósito: Git só faz sentido quando existe algo com história para contar. O projeto da Aurora tem quarenta capítulos de decisões.

40.1 Os quatro comandos do dia

O CICLO INTEIRO

bash

```
1 git status           # o que mudou
2 git add src/main/java/... # escolher o que vai
3 git commit -m "mensagem" # registrar
4 git push             # publicar
```

COMEÇANDO O PROJETO

bash

```
1 git init
2 echo "target/" > .gitignore
3 git add .
4 git commit -m "primeiro commit: esqueleto do catálogo"
5 git remote add origin git@github.com:aurora/catalog.git
6 git push -u origin main
```

⚠ ERRO COMUM

target/, .idea/, *.class e qualquer arquivo com senha **nunca** entram no repositório. Um .gitignore esquecido no primeiro dia significa, seis meses depois, um repositório de 400 MB com binários compilados. E um application.properties com a senha do banco versionado significa que a senha vazou — mesmo que você apague depois, ela continua no histórico para sempre.

40.2 O commit é uma mensagem para o futuro

O QUE NÃO AJUDA

ajustes

correções

fix

wip

update final

O QUE EXPLICA

corrige N+1 na
listagem de produtos

usa @EntityGraph para
carregar a categoria
junto: 4.032 consultas
viraram 1

O primeiro grupo aparece em todo repositório do mundo e é inútil no dia em que alguém precisa entender por que uma linha existe. O segundo responde à única pergunta que o código não responde sozinho: **por quê**.

O FORMATO QUE A MAIORIA DOS TIMES ADOTA

text

```

1  tipo: resumo no imperativo, até 50 caracteres
2
3  Corpo opcional, explicando o motivo e o contexto – não o que
4  o diff já mostra. Quebre em 72 colunas.
5
6  Closes #42

```

TIPO QUANDO

feat	funcionalidade nova
fix	correção de defeito
refactor	muda a forma, não o comportamento
test	acrescenta ou corrige teste
docs	documentação
chore	build, dependência, configuração

TABELA 40.1 A convenção *Conventional Commits*. Ela permite gerar o *changelog* automaticamente — e, mais importante, obriga quem escreve a classificar a própria mudança.

● O ESSENCIAL

Um bom commit é **pequeno e coerente**: uma mudança, um motivo. Se a mensagem precisa da palavra “e” duas vezes, são dois commits. O custo de separar é de trinta segundos; o benefício é poder reverter um sem desfazer o outro.

40.3 Ramos

O FLUXO DE UMA FUNCIONALIDADE

bash

```

1  git switch -c feat/busca-por-nome      # cria e entra no ramo
2
3  # ... trabalha, commita ...
4
5  git push -u origin feat/busca-por-nome
6  # abre o pull request na interface do GitHub
7
8  git switch main
9  git pull
10 git branch -d feat/busca-por-nome     # apaga o ramo local

```

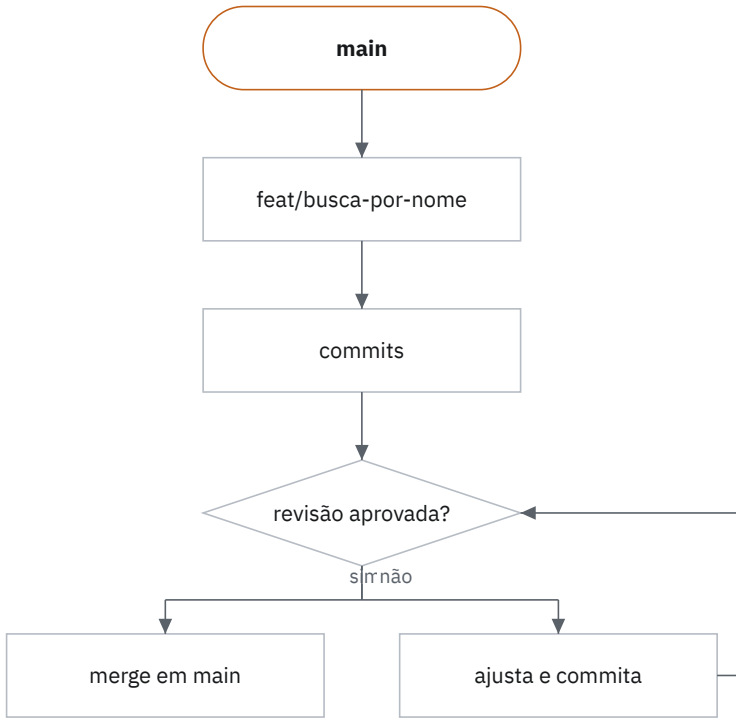


FIGURA 40.1 Um ramo por funcionalidade, revisão antes de entrar.

O ramo protegido — `main` que só aceita alteração por *pull request* aprovado, com os testes do capítulo 34 ao 37 passando — é a configuração mais barata de qualidade que existe. São dois cliques na interface do GitHub.

40.4 Conflito: o que é e como sair

A MENSAGEM QUE ASSUSTA E NÃO DEVERIA

text

```
Auto-merging ProductService.java
CONFLICT (content): Merge conflict in ProductService.java
Automatic merge failed; fix conflicts and commit the result.
```

O ARQUIVO FICA ASSIM

java

```
1 <<<<<< HEAD
2     return repository.findAll(pageable)
3         .map(ProductResponse::of);
4 =====
5     return repository.findAll(spec, pageable)
6         .map(ProductResponse::of);
7 >>>>>> feat/busca-por-nome
```

Entre <<<<<< e ===== está a sua versão; entre ===== e >>>>>>, a do outro ramo. Você apaga os marcadores e deixa o código correto — que pode ser um dos dois, ou uma terceira coisa que junta os dois.

```
git add ProductService.java
git commit          # a mensagem de merge já vem pronta
```

⚡ ERRO COMUM

Conflito não é erro: é o Git dizendo que duas pessoas mudaram a mesma linha e que ele não tem como adivinhar qual vale. Aceitar cegamente “a minha versão” para se livrar da tela é como apagar o trabalho do colega sem ler — e acontece o tempo todo. Leia os dois lados. Em caso de dúvida, chame a pessoa.

40.5 Desfazer, sem drama

OS QUATRO DESFAZERES ÚTEIS	bash
1 <code>git restore ProductService.java</code>	<i># descarta alteração não commitada</i>
2 <code>git restore --staged arquivo.java</code>	<i># tira do "add"</i>
3 <code>git commit --amend</code>	<i># corrige o último commit (local!)</i>
4 <code>git revert a1b2c3d</code>	<i># desfaz criando outro commit</i>

⚡ ERRO COMUM

`git reset --hard` apaga trabalho não commitado e não tem volta. E `git push --force` reescreve a história do servidor: se alguém já tinha puxado aquele ramo, o repositório dela passa a divergir do seu de um jeito confuso. Em ramo compartilhado, desfaça com `revert`, que cria um commit novo e preserva o histórico.

40.6 O histórico como ferramenta

O QUE VOCÊ VAI USAR QUANDO ALGO QUEBRAR	bash
1 <code>git log --oneline -20</code>	
2 <code>git log -p ProductService.java</code>	<i># a história de um arquivo</i>
3 <code>git blame ProductService.java</code>	<i># quem escreveu cada linha</i>
4 <code>git bisect start</code>	<i># acha o commit que quebrou</i>

`git blame` tem fama injusta de ferramenta de acusação. Na prática, ele é o caminho mais rápido para a **mensagem de commit** que explica por que aquela linha estranha existe — e é aí que a qualidade das mensagens deixa de ser etiqueta e passa a ser economia de tempo.

ENQUANTO ISSO, NA EMPRESA **Funciona na minha máquina**

O bug apareceu na quinta e só acontecia em homologação.

Carlos rodou local: funcionava. Rodou de novo: funcionava. Passou a manhã comparando configuração, versão de Java, versão do PostgreSQL.

Às onze e meia, Marina pediu para ver o `git status` dele.

```
On branch feat/filtros
Changes not staged for commit:
  modified:   src/main/resources/application.properties
  modified:   src/main/java/.../ProductSpecs.java
  modified:   src/main/java/.../SecurityConfig.java
```

Três arquivos alterados e não commitados. Funcionava na máquina do Carlos porque a correção existia só na máquina do Carlos — em arquivos que ele tinha mexido na terça e esquecido.

— Não é que funcione na sua máquina — disse Marina. — É que a sua máquina tem um software que não existe em lugar nenhum. Nem no repositório.

No commit daquela tarde, Carlos escreveu a mensagem mais longa da vida dele. Marina aprovou sem comentários.

RESUMO

- `status`, `add`, `commit`, `push` resolvem o dia; o resto é para quando algo dá errado.
- `.gitignore` no primeiro dia; senha nunca, em nenhum commit.
- A mensagem explica o **porquê**; o diff já mostra o quê.
- Um ramo por funcionalidade, `main` protegido por *pull request* e testes.
- Conflito é uma pergunta, não um erro. `revert` em ramo compartilhado, nunca `push --force`.

✓ PONTO DE CONTROLE

Você versiona o projeto, escreve commits que explicam a intenção, trabalha em ramos com revisão e resolve um conflito lendo os dois lados.

■ O PROJETO AGORA

Fim da Parte 9. O projeto tem histórico, documentação que se atualiza sozinha e requisições versionadas. Tudo que faltava para outra pessoa assumir o código — e é exatamente isso que a Parte 10 vai pedir de você.

- 1 Inicialize o repositório, crie o `.gitignore` e faça o primeiro commit com uma mensagem que explique o que o projeto é.

- 2 Crie um ramo, altere uma linha, volte para `main`, altere a mesma
.. linha de outro jeito e provoque um conflito de propósito. Resolva-o.
- 3 Use `git bisect` para encontrar o commit que introduziu um defeito.
... Crie o cenário: dez commits, um deles quebrando um teste.

PARTE 10

Projeto final

Agora sem mastigar. Você recebe os requisitos e constrói a loja inteira — e depois a coloca no ar.

NESTA PARTE

- 41 Projeto final
- 42 Docker e deploy

CAPÍTULO

41

Projeto final

Agora sem mastigar. Você recebe os requisitos e constrói a loja inteira.

“Eu não sabia o que estava fazendo, então fiz. E aí eu soube.”

DITADO DE PROGRAMADOR, ATRIBUÍDO A TODO MUNDO

AO FIM DESTES CAPÍTULOS

Construir sozinho, do zero, a API completa da loja — com as decisões de modelagem, arquitetura, segurança e teste tomadas por você.

**Capítulo disponível na
versão completa**

CAPÍTULO

42

Docker e deploy

Um programa que só roda na sua máquina é um hobby. O que roda em qualquer máquina é um sistema.

AO FIM DESTES CAPÍTULOS

Empacotar a aplicação em uma imagem Docker, subir tudo com Docker Compose, configurar por variável de ambiente e saber o que olhar depois que ela está no ar.

**Capítulo disponível na
versão completa**

Respostas

O que vamos construir

- 1 Não há resposta certa, mas há uma resposta boa: ela fala de *dados* e de *operações*, não de tecnologia. “Guardar produtos de uma loja, permitir buscar por nome e preço, e só deixar um administrador cadastrar” é uma descrição melhor do que “uma API em Spring Boot com PostgreSQL”.
- 2 O objetivo do exercício não é a resposta, é o hábito: saber em qual versão você está e onde se lê o que ela mudou. Um programador Java que não sabe a própria versão vai copiar código que não compila.

Primeiro programa em Java

1

```
1 public class Soma {
2     public static void main(String[] args) {
3         int a = Integer.parseInt(args[0]);
4         int b = Integer.parseInt(args[1]);
5         System.out.println(a + b);
6     }
7 }
```

Sem o `parseInt`, `args[0] + args[1]` concatena texto: 2 e 3 viram “23”. É o primeiro encontro com uma ideia do próximo capítulo — o tipo decide o que o `+` significa.

- 2 A primeira compila e falha ao rodar com `NoSuchMethodError: main` — a assinatura tem de ser exata. A segunda dá `reached end of file while parsing`. A terceira dá `cannot find symbol`, com uma seta apontando `Sistem`. Três mensagens, três categorias: assinatura, sintaxe e nome.

O que é Spring

- 1 O `DataSource` recebe a configuração de conexão; o repositório recebe o `DataSource` o serviço recebe o repositório; o controlador recebe o serviço. Cada camada conhece **apenas a de baixo** — e é essa cadeia que o Spring vai montar sozinho.

2

```
var real = new OrderService(new SmtEmailSender());
var teste = new OrderService(new FakeEmailSender());
```

Duas linhas, duas configurações do mesmo sistema. Você acabou de fazer, à mão, exatamente o que o `@MockBean` do capítulo 36 faz — e é por isso que injeção por construtor torna o teste trivial.

- 3** A JVM carrega a classe principal; o Spring Boot cria o `ApplicationContext` o contêiner varre os pacotes procurando anotações; para cada classe encontrada, resolve as dependências do construtor (criando-as antes, se necessário) e guarda o bean; a autoconfiguração sobe o Tomcat embutido e registra as rotas dos controladores; a porta abre. Nada disso é mágica: é um grafo sendo montado em ordem topológica.

Spring Data JPA

1

```

1  @Entity
2  @Table(name = "category")
3  public class Category {
4      @Id
5      @GeneratedValue(strategy = GenerationType.IDENTITY)
6      private Long id;
7
8      @Column(nullable = false, unique = true, length = 80)
9      private String name;
10
11     protected Category() { }
12 }

```

2

```

1  @Column(name = "updated_at")
2  private Instant updatedAt;
3
4  @PreUpdate
5  void aoAtualizar() {
6      this.updatedAt = Instant.now();
7  }

```

`@PreUpdate` e `@PrePersist` são ganchos do ciclo de vida. Úteis, e com um porém: eles rodam dentro do Hibernate, o que significa que não rodam quando você faz uma atualização em massa por SQL nativo.

- 3** Os produtos gravados como 1 (que era `INATIVO`) passam a ser lidos como o valor novo que ocupou a posição 1. Nenhum erro, nenhum aviso: só dados que mudaram de significado. É o tipo de defeito que só é descoberto por um cliente reclamando, meses depois — e a correção exige um script de migração feito à mão.

Git

- 1 Um detalhe que quase ninguém faz e vale muito: escreva no corpo do primeiro commit **por que** o projeto existe e qual problema ele resolve. É a mensagem que mais gente vai ler, e a única que ninguém consegue reconstruir depois.
- 2 Fazer isso de propósito, com calma, em um projeto que não importa, é a diferença entre ficar paralisado e resolver em dois minutos no dia em que acontecer de verdade — que será numa sexta, com pressa.
- 3 `git bisect start`, `git bisect bad`, `git bisect good <commit antigo>`, e O Git faz uma busca binária — dez commits viram três ou quatro verificações. Com um script de teste, `git bisect run ./mvnw test` acha o culpado sozinho. É a ferramenta mais subestimada do Git e a que mais impressiona quando você a usa na frente de alguém.

Glossário

Bean Um objeto cujo ciclo de vida é controlado pelo Spring: ele cria, injeta as dependências, guarda e destrói.

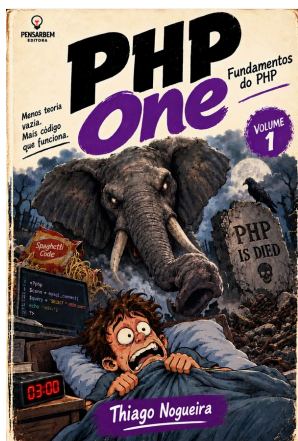
Contêiner (ApplicationContext) O registro de todos os beans da aplicação. Monta o grafo na partida e entrega a instância a quem pedir pelo tipo.

Entidade gerenciada Objeto que o Hibernate está acompanhando dentro de uma transação. Alterações nele são gravadas automaticamente ao final.

JDK *Java Development Kit*: o pacote que compila e executa. Traz o `javac`, o `java`, o depurador e as bibliotecas.

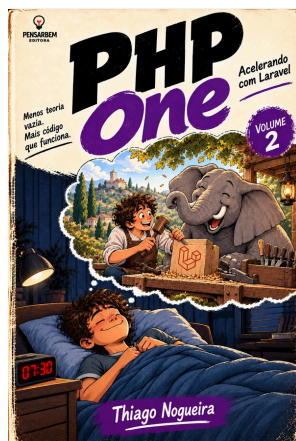
JRE *Java Runtime Environment*: só executa. Não tem `javac`. Foi distribuído separadamente por anos e é a causa da metade das instalações frustradas de quem começa.

Conheça as demais obras dessa coleção



PHP One

Volume 1 · Fundamentos do PHP



PHP One

Volume 2 · Acelerando com Laravel