

PHP One — Prévia

Volume 2 · Acelerando com Laravel

Thiago Nogueira

PENSARBEM

PHP 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
Segunda-feira, nove e dez	3
O fim de semana do Nonato	3
O meio-termo que ninguém pediu	4
O que você traz do volume 1	5
Como este volume é organizado	6
1 O que é HTTP	9
1.1 A troca inteira, em texto	11
1.2 Os sete status que você vai usar	12
1.3 O que o PHP enxerga	13
1.4 Por que o \$_POST estava vazio	14
1.5 Cabeçalhos que mudam o comportamento	15
1.6 O servidor não lembra de você	15
1.7 Um endpoint cru	16
2 O que é uma API REST	19
2.1 O endereço nomeia uma coisa, não uma ação	21
2.2 Seguro, idempotente, e nem um nem outro	21
2.3 O relatório que corrigia	21
2.4 A repetição que você não controla	21
2.5 Então o que é “renovar tudo”	21
2.6 O status já responde metade	21
2.7 O contrato com quem você não controla	21
2.8 REST não é lei	21
2.9 O desenho da Casa Amarela	21
3 Um framework de quarenta linhas	22
3.1 Um arquivo por página	25
3.2 Front controller: uma porta só	25
3.3 Um roteador em vinte linhas	26
3.4 O contêiner ingênuo	29
3.5 Middleware: a cebola antes de ter nome	31
3.6 O que ainda falta	32
3.7 O que você acabou de entender	32
4 O que é o Laravel	35
4.1 Framework é código que chama o seu	37
4.2 As peças	37
4.3 O ciclo de uma requisição	37
4.4 Convenção sobre configuração, e o que ela custa	37
4.5 O que ele decide por você, e como discordar	37

4.6	Laravel, Symfony ou nenhum dos dois	37
5	O primeiro projeto	41
5.1	<code>composer create-project</code>	43
5.2	A rota que responde em dois minutos	44
5.3	As pastas que importam	44
5.4	O primeiro erro de permissão	45
5.5	<code>.env</code> : a configuração desta máquina	46
5.6	Servir o projeto: escolha uma	47
5.7	O que o <code>index.php</code> faz	47
6	Configuração, ambiente e Artisan	50
6.1	<code>config()</code> e <code>env()</code> : a diferença que derruba produção	53
6.2	O arquivo de configuração do projeto	53
6.3	Ambientes	53
6.4	Artisan: três ferramentas com um nome só	53
6.5	<code>tinker</code> : o console que conhece o projeto	53
6.6	Um comando próprio	53
7	Rotas e controllers	54
7.1	O arquivo de rotas é o índice	57
7.2	<code>web.php</code> e <code>api.php</code> : dois mundos	57
7.3	As rotas da Casa Amarela	57
7.4	Parâmetros e restrições	57
7.5	O parâmetro que já vem virado registro	57
7.6	Nome de rota: a URL que muda sem quebrar nada	57
7.7	Três formatos de controller, e o que cada um comunica	57
7.8	O controller não precisa saber de tudo	57
8	Requests e responses	58
8.1	<code>Request</code> é um objeto	61
8.2	O dado certo, já no tipo certo	61
8.3	Devolver: array, model ou resposta	61
8.4	A mesma rota, duas respostas	61
8.5	Upload: não confie em nada que veio junto	61
9	Blade, quando a tela ainda é a resposta	62
9.1	Nem tudo é API	65
9.2	Blade é PHP com menos cerimônia	65
9.3	<code>{{ }}</code> escapa, e isso é o recurso	65
9.4	Layout e componente	65
9.5	O formulário e o token que você não vê	65
9.6	Quando parar	65
10	Migrations, seeders e factories	69
10.1	Migration não é backup	71

10.2	O esquema vira arquivo	71
10.3	O estado do banco mora numa tabela	71
10.4	<code>down()</code> honesto	71
10.5	Alterar coluna sem derrubar a aplicação	71
10.6	Seeder: o dado que todo ambiente precisa	71
10.7	Factory: o dado de mentira que parece de verdade	71
10.8	Quem roda migration em produção	71
11	Eloquent	72
11.1	Eloquent não é SQL mágico	75
11.2	O model e as convenções	76
11.3	Active Record: o objeto que sabe se salvar	77
11.4	Buscar, e o <code>null</code> que escapa	78
11.5	Mass assignment	79
11.6	Cast: o tipo certo dos dois lados	80
11.7	Accessor: valor calculado com cara de coluna	81
11.8	Scope: a consulta que ganha nome	82
11.9	Ver o SQL antes de confiar	82
11.10	Model gigante: os sinais	83
12	Relacionamentos	86
12.1	A chave estrangeira mora do lado que tem muitos	89
12.2	Muitos para muitos	89
12.3	N+1: cem consultas atrás de um ponto	89
12.4	<code>with</code> , <code>load</code> e <code>withCount</code>	89
12.5	A proteção que transforma o N+1 em erro	89
12.6	Um pulo a mais, e o polimórfico	89
12.7	Apagar em cascata	89
13	O CRUD completo	93
13.1	As cinco rotas e o que cada uma promete	95
13.2	Criar	95
13.3	Ler um: <code>404</code> é uma resposta	95
13.4	<code>PUT</code> e <code>PATCH</code> não são a mesma coisa	95
13.5	Apagar	95
13.6	Uma operação de negócio, um <code>commit</code>	95
13.7	“É só um CRUD”	95
14	Validation e Form Requests	96
14.1	Validação não é regra de negócio	99
14.2	Regras onde o dado entra	99
14.3	Form Request: o controller que volta a ter quatro linhas	99
14.4	Normalizar antes de conferir	99
14.5	<code>sometimes</code> , <code>nullable</code> e o <code>PATCH</code> que apaga campo	99
14.6	Regra própria, e quando ela já existe	99
14.7	<code>422</code> , <code>errors</code> e o campo que o cliente destaca	99
14.8	O empréstimo, e o que não entra no Form Request	99

14.9	Validar também o que sai	99
15	API Resources	100
15.1	O model não é o JSON	103
15.2	JsonResource	103
15.3	Coleções	103
15.4	whenLoaded: o relacionamento que só aparece se foi carregado	103
15.5	O campo que não pode sair, nunca	103
15.6	Um formato, a API inteira	103
15.7	Mudar a resposta sem quebrar quem já usa	103
16	Paginação, filtros e buscas	104
16.1	Listagem sem limite é uma negação de serviço que você publicou	107
16.2	Três paginadores	107
16.3	Ordenação estável: o desempate que ninguém lembra	107
16.4	Filtro opcional sem escada de if	107
16.5	Busca por texto: LIKE, acento e o índice que não é usado	107
16.6	Ordenar por campo do cliente sem abrir o banco para ele	107
16.7	O teto é do servidor	107
17	Erros padronizados	108
17.1	Três formatos de erro na mesma API	111
17.2	O formato	112
17.3	O handler, onde toda exceção termina	113
17.4	Exceção de domínio vira status numa linha	114
17.5	O 404 do binding: útil e genérico demais	116
17.6	500 não vaza nada	117
17.7	Código de incidente: o número que o suporte pede	118
17.8	401 e 403 não são a mesma coisa	119
18	Service Container e injeção de dependência	125
18.1	O new espalhado pelo código é o problema	127
18.2	Pedir em vez de montar	127
18.3	O contêiner monta o grafo	127
18.4	bind, singleton e scoped	127
18.5	Service Provider: onde as instruções moram	127
18.6	Interface no construtor, implementação no provider	127
18.7	Trocar a implementação no teste	127
18.8	As duas formas de usar o contêiner errado	127
19	Services: onde a regra de negócio mora	128
19.1	Controller com oitocentas linhas começa com vinte	131
19.2	A regra que envolve duas entidades não cabe no model	131
19.3	O service	131
19.4	A regra corrigida, e a regra que não era regra	131
19.5	O service não conhece HTTP	131

19.6	Um método público, uma transação	131
19.7	Repository: quando ajuda e quando é burocracia	131
19.8	Nem todo projeto precisa de todas as camadas	131
19.9	Service sem propósito: como reconhecer	131
20	Middleware	132
20.1	A cebola	135
20.2	Global, de grupo e de rota	135
20.3	Ordem importa mais do que parece	135
20.4	ForcaJson: a API que sempre responde JSON	135
20.5	RegistraRequisicao: o incidente que nasce no começo	135
20.6	Throttle: o limite que protege de você mesmo	135
20.7	O que não colocar ali dentro	135
21	Autenticação com Sanctum	136
21.1	Senha nunca é guardada	139
21.2	bcrypt, argon2 e o custo que é proposital	139
21.3	Sessão e token: dois problemas diferentes	139
21.4	Sanctum	139
21.5	Login, logout e GET /eu	139
21.6	Revogar de verdade	139
21.7	A resposta que não diz se o e-mail existe	139
22	Autorização: Gates e Policies	140
22.1	Autenticação diz quem; autorização diz o quê	143
22.2	Gate para a regra solta, Policy para o recurso	143
22.3	authorize, can e o 403 que sai sozinho	143
22.4	O recurso é dele?	143
22.5	Listagem com escopo	143
22.6	before: o admin, e o cuidado com ele	143
22.7	Papel não é permissão	143
22.8	Testar permissão é obrigatório	143
23	Events, jobs e filas	147
23.1	O que não precisa acontecer antes da resposta	149
23.2	Event e listener: desacoplar sem esconder	149
23.3	Job: a unidade de trabalho que pode falhar e voltar	149
23.4	Driver de fila: sync, database, redis	149
23.5	Worker, supervisor e o processo que precisa reiniciar	149
23.6	Retentativa, backoff e failed_jobs	149
23.7	Job precisa ser idempotente	149
23.8	Quando o evento vira espaguete invisível	149
24	Cache, logs e o que se mede	150
24.1	Uma fila existe porque esperar custa caro; um cache também	153
24.2	Medir antes de otimizar	153
24.3	Cache::remember e a pergunta difícil	153

24.4	Cache de configuração, rota e view	153
24.5	Log não é <code>dd()</code>	153
24.6	Níveis, canais e o log que dá para procurar	153
24.7	O que nunca entra no log	153
25	Testes: o que estamos tentando provar	157
25.1	Teste não prova que está certo	159
25.2	Pest ou PHPUnit: escolha uma e siga	159
25.3	Arrumar, agir, afirmar	159
25.4	Testar a regra sem banco e sem HTTP	159
25.5	As onze condições em menos de um segundo	159
25.6	Dublês: fake, stub, spy, mock	159
25.7	Factory como fixture	159
25.8	O teste que reproduz o defeito de ontem	159
26	Testes de feature, HTTP e banco	160
26.1	<code>getJson</code> , <code>postJson</code> e o contrato verificado	163
26.2	<code>RefreshDatabase</code> : isolamento sem <code>TRUNCATE</code>	163
26.3	<code>actingAs</code> e o teste de rota protegida	163
26.4	Testar <code>403</code> é testar o que ninguém testa à mão	163
26.5	O campo que nunca pode aparecer	163
26.6	<code>assertDatabaseHas</code> e o que ele prova	163
26.7	Fake de fila, e-mail e evento	163
26.8	Suíte lenta: diagnosticar antes de culpar o banco	163
26.9	A correção da esteira vermelha	163
27	Documentação da API	164
27.1	Documentação que não nasce do código começa a mentir	167
27.2	OpenAPI: o esquema que gera tudo o mais	167
27.3	Gerar a partir das rotas, requests e resources	167
27.4	Documentar o erro é documentar metade do contrato	167
27.5	Exemplos que a pessoa clica e executa	167
27.6	Versionar e marcar o que vai sair	167
27.7	Expor ou não a documentação em produção	167
28	Git, CI e o dia do deploy	168
28.1	<code>.gitignore</code> antes do primeiro <code>git add</code>	171
28.2	Segredo no histórico se resolve trocando o segredo	171
28.3	Commit que conta uma história	171
28.4	A esteira: Pint, PHPStan, testes	171
28.5	Migration em produção é um passo separado	171
28.6	<code>down</code> , cache e o worker que precisa reiniciar	171
28.7	Health check: vivo não é o mesmo que pronto	171
28.8	A lista antes de publicar	171
28.9	O primeiro pedido depois do ar	171
29	Upload de arquivos	175

29.1	Três limites antes do seu código	177
29.2	Validar a imagem pelo que ela é	177
29.3	Disco privado, disco público	177
29.4	Processar depois de responder	177
29.5	Trocar sem deixar órfão	177
29.6	Testar sem disco e sem fila	177
30	Mail e notificações	178
30.1	Documento ou aviso	181
30.2	Quem assina o que sai	181
30.3	A notificação	181
30.4	Um canal que o Laravel não tem	181
30.5	Um job por canal	181
30.6	Em português	181
30.7	Chegar à caixa de entrada	181
30.8	O documento, como Mailable	181
30.9	Testar sem mandar	181
31	Queues na prática	182
31.1	A métrica é a espera	185
31.2	Uma fila por urgência	185
31.3	Importar em lote, com progresso	185
31.4	Encadear o que depende	185
31.5	O ritmo do provedor	185
31.6	Não fazer duas vezes	185
31.7	<code>timeout</code> e <code>retry_after</code>	185
31.8	O worker é um processo que envelhece	185
31.9	Quando falha em produção	185
	Encerramento	186
31.10	O caminho dos dois volumes	189
31.11	O que este livro não cobre, e onde procurar	189
31.12	Uma última coisa	189
	Respostas	190
	Glossário	204

Antes de começar

Vinte e cinco fins de semana até 31 de março. O Nonato contou no calendário da parede, e a conta não fechou por um.

O volume 1 terminou numa sexta-feira, com a pasta arrumada e uma pergunta no caderno da Tainá. Este começa na segunda.

Segunda-feira, nove e dez

A primeira tarefa da semana não teve nada de web. O Dedé trocou a senha do banco — a que tinha passado por onze arquivos e pelo histórico do Git — e colou a nova no `.env`, que não vai para lugar nenhum.

— Qual era a antiga? — perguntou a Tainá.

— senha.

— E a nova?

— Não é senha.

O Nonato, da mesa ao lado, levantou a mão sem tirar os olhos da tela.

— Protesto. Em 2009 a senha era `casaamarela2009`. Alguém trocou para `senha` depois de mim.

Às nove e quarenta, o Seu Juvenal entrou na sala de reunião da Vertexo com um pacote de pão de queijo e um anúncio.

— O meu neto, o Kauã, fez um aplicativo. Num fim de semana. Para a biblioteca. — Ele virou o celular para a mesa: uma tela azul, o logotipo da Casa Amarela esticado na horizontal e um botão *Renovar*. — Só falta ligar no sistema.

— Ligar como? — perguntou a Márcia.

— Isso eu deixo com vocês. Ele disse que é só uma API.

A Tainá abriu o caderno na última página escrita. Embaixo de *o que chega do navegador até o PHP?* apareceu uma segunda linha: *e do aplicativo do Kauã?*

O fim de semana do Nonato

A discussão começou do jeito que discussão técnica começa na Vertexo: com alguém dizendo que não precisava discutir.

— A gente faz em PHP puro — disse o Nonato. — Vocês acabaram de passar um livro inteiro aprendendo PHP. Eu fiz o Sistema num fim de semana, e ele está no ar há quinze anos.

— Com `mysql_query` e senha em MD5 — disse o Dedé.

— Com `mysql_query` e senha em MD5, e nenhum acervo perdido.

O Dedé foi até o quadro branco.

— Tudo bem. O que o aplicativo do Kauã precisa fazer?

— Ver os livros. Emprestar. Renovar. Devolver.

O Dedé escreveu quatro palavras — **livros, exemplares, leitores, empréstimos** — e, ao lado de cada uma, as mesmas quatro letras.

— CRUD — disse a Tainá. — *Create, read, update, delete.*

— Quatro tabelas, quatro operações. Dezesesseis endpoints. É a parte fácil. — Ele fez um traço embaixo. — Agora o resto.

E escreveu, uma por linha: *rotas. ler JSON. responder JSON. validar cada campo. senha do leitor. token do aplicativo. quem pode o quê. erro num formato só. paginação. busca. e-mail de aviso. fila, para o e-mail não travar a tela. cache. log. migração de banco. testes. documentação para o Kauã. deploy.*

— Cada uma dessas — disse o Nonato — é um fim de semana.

— Então conta.

O Nonato contou as linhas em voz alta, contando junto as que o Dedé ia lembrando no caminho. Deram vinte e seis. Depois ele se levantou, foi até o calendário da parede — um calendário de farmácia, com os meses em fileira — e contou os sábados até 31 de março, com o dedo.

— Vinte e cinco.

— Faltou um — disse a Tainá.

— Faltou o fim de semana em que dá errado — disse a Márcia. — Sempre tem esse.

O Nonato sentou de novo, devagar.

— Em 2009 não tinha aplicativo.

O meio-termo que ninguém pediu

— Laravel — disse o Dedé. — Tudo o que está no quadro vem pronto, e testado por mais gente do que a gente vai conhecer na vida.

— E aí ninguém aqui sabe o que roda por baixo — respondeu o Nonato. — Vira mágica. E mágica quebra na sexta à noite.

Foi a Tainá quem desempatou, sem perceber que estava desempatando.

— E se a gente fizer à mão primeiro? Pequeno. Só para ver o que ele faz. Depois usa o de verdade.

O Dedé olhou para o quadro. O Nonato olhou para o calendário.

— Quanto é pequeno? — perguntou o Nonato.

— Quarenta linhas.

— Quarenta linhas eu leio.

A Vera, que tinha vindo só entregar a lista das onze regras e ficado pelo pão de queijo, pegou a bolsa.

— Façam como quiserem. Na segunda às nove eu abro. Com ou sem aplicativo.

O plano ficou no quadro até o fim do projeto, na letra do Dedé, com uma seta da Tainá do lado:

- | | |
|--|--------------|
| 1. entender o que o navegador (e o Kauã) manda | -> HTTP |
| 2. desenhar a conversa antes do código | -> REST |
| 3. escrever um framework pequeno, à mão | -> 40 linhas |
| 4. usar o de verdade, sabendo o que ele faz | -> Laravel |

É a ordem deste volume.

O que você traz do volume 1

Tudo o que o Laravel vai supor que você sabe: tipos, arrays, funções, closures, SQL escrito à mão, PDO, Composer, classes, interfaces, exceções, tipagem estrita, enums, datas com fuso e erros que avisam. E um projeto organizado do jeito que o framework vai organizar:

acervo/	
bin/	
config/app.php	importar-doacoes.php, atrasados.php
public/	fuso, prazo, banco — lidos do .env
meça aqui	vazia: é a porta da web, e a web co-
src/	
Acervo/	Livro, Exemplar, StatusExemplar
Emprestimos/	Dinheiro, PrazoDeEmprestimo, Statu-
sEmprestimo	
Circulacao/	exceções de domínio, CalendarioDaBi-
biblioteca	
Importacao/	Csv, Importador
Tempo/	Relogio, RelogioDoSistema, Relogio-
Parado	
Registro.php	log com contexto
Servicos.php	contêiner de fábricas, escrito à mão
var/log/	
bootstrap.php	config, erros viram exceção, serviços
.env, .env.example	
composer.json	psr-4 CasaAmarela\, phpstan nível 5

E o banco `casa_amarela`, com livros, exemplares, leitores e empréstimos, escrito em SQL puro. Cada peça do Laravel vai ser comparada com uma coisa desta árvore — e, em algumas comparações, a árvore ganha.

Se você chegou direto a este volume: a **Biblioteca Comunitária Casa Amarela** tem quatro mil títulos e um sistema em PHP de 2009, o **Sistema**, que atende o balcão enquanto a **Vertexo Sistemas** constrói o substituto. A verba vem de um edital cultural, com prestação de contas em **31 de março**: se atrasar, a verba volta. **Dedé** explica, **Tainá** pergunta e anota, **Vera** sabe as regras, **Márcia** guarda o prazo, **Nonato** escreveu o

Sistema, **Cléber** responde “ele processa” e **Seu Juvenal** traz o próximo pedido.

Como este volume é organizado

Os capítulos têm numeração própria, a partir do 1. Quando o texto citar um capítulo do primeiro livro, a citação diz isso: “o capítulo 21 do volume 1”. Sem a indicação, o capítulo é deste volume.

PARTE	CAPÍTULOS	O QUE ACONTECE
1 · A web por baixo do framework	1–4	HTTP, REST, quarenta linhas e a escolha do framework
2 · Dentro do Laravel	5–9	projeto, configuração, rotas, requests, Blade
3 · Eloquent sobre o SQL	10–12	migrations, Eloquent, relacionamentos
4 · A API de verdade	13–17	CRUD, validação, resources, paginação, erros
5 · Arquitetura e segurança	18–22	contêiner, services, middleware, Sanctum, Policies
6 · Depois da resposta	23–24	filas, cache, logs e medição
7 · Provar e publicar	25–28	testes, documentação, Git, CI e deploy
8 · O sistema no ar	29–31	upload, e-mail e notificações, filas em produção

TABELA 0.1 Oito partes, na ordem em que o problema aparece. O framework só entra no capítulo 4, depois de você ter escrito à mão o que ele faz; a última parte começa no dia seguinte ao deploy.

► NA PRÁTICA

Tenha o PHP 8.3 e o Composer instalados, conferidos com `php -v` e `composer -V`. Se você chegou aqui pelo volume 1, já tem os dois. Se não, os capítulos 2 e 16 do volume 1 fazem essa instalação passo a passo.

Faltam vinte e cinco fins de semana. Na segunda às nove, a Vera abre.

PARTE 1

A web por baixo do framework

HTTP, REST e um framework de quarenta linhas escrito à mão. Depois deste trecho, nenhuma parte do Laravel vai parecer mágica — vai parecer trabalho que alguém já fez.

NESTA PARTE

- 1 O que é HTTP
- 2 O que é uma API REST
- 3 Um framework de quarenta linhas
- 4 O que é o Laravel

CAPÍTULO

1

O que é HTTP

O aplicativo enviava os dados. O servidor recebia os dados. E o \$_POST estava vazio.

AO FIM DESTES CAPÍTULO

Ler uma troca HTTP inteira com `curl -v`, saber exatamente o que o PHP enxerga de uma requisição, entender por que `$_POST` fica vazio com JSON e escrever um endpoint cru que recebe e devolve JSON sem framework.

ENQUANTO ISSO, NA EMPRESA Ele não manda nada

A Tainá tinha ligado uma cópia do aplicativo do Kauã no acervo, para testar. Ele cadastrava empréstimo e recebia sempre a mesma resposta: *“exemplar não informado”*.

— Ele não manda nada — disse Tainá. — Olha aqui, o `$_POST` chega vazio.

— Vazio como?

— Vazio. `array(0)`.

Dedé pediu para ela rodar o pedido de novo, mas pela linha de comando, com uma opção que ele soletrou letra por letra.

```
> POST /emprestimos HTTP/1.1
> Content-Type: application/json
> Content-Length: 34
>
{"exemplar_id":812,"leitor_id":47}
```

— Está mandando — disse Tainá.

— Está.

— Mas o `$_POST`...

— O `$_POST` não é o que o aplicativo mandou. É o que o PHP resolveu guardar lá.

1.1 A troca inteira, em texto

HTTP é texto que vai e texto que volta. A opção `-v` do `curl` mostra as duas metades: as linhas que começam com `>` saíram da sua máquina, as que começam com `<` voltaram do servidor.

```
$ curl -v -X POST http://localhost:8000/emprestimos \
-H "Content-Type: application/json" \
-d '{"exemplar_id":812}'
```

```

1  > POST /emprestimos HTTP/1.1
2  > Host: localhost:8000
3  > User-Agent: curl/8.18.0
4  > Accept: */*
5  > Content-Type: application/json
6  > Content-Length: 19
7  >
8  < HTTP/1.1 201 Created
9  < Date: Mon, 21 Sep 2026 20:29:51 GMT
10 < X-Powered-By: PHP/8.4.2
11 < Content-Type: application/json
12 <

```

As duas metades têm a mesma forma: uma primeira linha diferente, uma lista de cabeçalhos, uma linha em branco e um corpo opcional.

Na ida, a primeira linha traz verbo, caminho e versão. Na volta, traz a versão, o número e o nome do status.

● O ESSENCIAL

A linha em branco não é formatação: é a fronteira. Tudo antes dela são cabeçalhos; tudo depois é corpo.

É por isso que qualquer `echo` acidental antes de um `header()` estraga a resposta inteira — o PHP entende que o corpo começou, e os cabeçalhos que vierem depois não têm mais para onde ir.

1.2 Os sete status que você vai usar

```
< HTTP/1.1 201 Created
```

O número é o que o outro programa lê; o texto ao lado é cortesia para humanos.

CÓDIGO	QUANDO	O CORPO
200	deu certo, aqui está	o recurso
201	criei, e está aqui	o recurso criado
204	feito, nada a dizer	vazio
400	o pedido está malformatado	o motivo
401	não sei quem é você	o motivo
404	não existe	o motivo
422	entendi o pedido e ele é inválido	quais campos

TABELA 1.1 Sete cobrem a API inteira deste livro. Os outros existem e são raros.

1.3 O que o PHP enxerga

Todo esse texto chega ao seu programa já separado em três variáveis que existem sozinhas, sem você declarar nada.

API.PHP

php

```

1  <?php
2
3  declare(strict_types=1);
4
5  header('Content-Type: application/json');
6
7  echo json_encode([
8      'metodo' => $_SERVER['REQUEST_METHOD'],
9      'caminho' => $_SERVER['REQUEST_URI'],
10     'tipo' => $_SERVER['CONTENT_TYPE'] ?? '(ausente)',
11     'get' => $_GET,
12     'post' => $_POST,
13 ], JSON_PRETTY_PRINT | JSON_UNESCAPED_SLASHES);

```

Rode o servidor que vem junto com o PHP e mande um formulário comum:

```
$ php -S localhost:8000 api.php
```

```
$ curl -X POST http://localhost:8000/emprestimos \
-d "exemplar_id=812&leitor_id=47"
```

```

1  {
2      "metodo": "POST",
3      "caminho": "/emprestimos",
4      "tipo": "application/x-www-form-urlencoded",
5      "get": [],
6      "post": {
7          "exemplar_id": "812",
8          "leitor_id": "47"
9      }
10 }

```

Superglobal Variável que existe em qualquer lugar do programa sem ser declarada nem recebida: `$_GET`, `$_POST`, `$_SERVER`, `$_FILES`, `$_COOKIE`. Elas são a forma mais antiga de o PHP entregar a requisição, e a mais frágil, porque qualquer linha de qualquer arquivo pode lê-las — e escrevê-las.

Repare nos valores: `"812"`, entre aspas. Tudo que chega por HTTP é texto, inclusive o que parece número. A conversão é trabalho seu, e o lugar dela é a fronteira de entrada.

1.4 Por que o `$_POST` estava vazio

Agora o mesmo pedido, com JSON:

```
$ curl -X POST http://localhost:8000/emprestimos \
-H "Content-Type: application/json" \
-d '{"exemplar_id":812,"leitor_id":47}'
```

```
1  {
2      "metodo": "POST",
3      "caminho": "/emprestimos",
4      "tipo": "application/json",
5      "get": [],
6      "post": []
7  }
```

O aplicativo do Kauã estava certo, e o `$_POST` também.

O PHP preenche o `$_POST` a partir do corpo **apenas quando o `Content-Type` é um dos dois formatos de formulário:** `application/x-www-form-urlencoded`, que é o do exemplo anterior, e `multipart/form-data`, que é o de envio de arquivo. Com qualquer outro tipo, ele não tenta adivinhar: deixa o corpo intacto e o `$_POST` vazio.

O corpo continua lá, inteiro, num lugar com nome esquisito:

API.PHP

php

```
1 $bruto = file_get_contents('php://input');
2
3 $dados = json_decode($bruto, true);
4
5 echo $dados['exemplar_id'], "\n";
```

812

`php://input` é um fluxo de leitura com o corpo cru da requisição, do primeiro ao último byte, sem interpretação nenhuma. `json_decode` com o segundo argumento `true` devolve array associativo em vez de objeto.

ERRO COMUM

Este é, com folga, o primeiro defeito de quem escreve API em PHP: o aplicativo manda JSON, o servidor lê `$_POST`, e a resposta é “campo obrigatório” para um campo que foi enviado.

O sintoma engana porque **o erro está no servidor e a suspeita cai no cliente.**

Antes de acusar quem manda, rode `curl -v`: se a linha `Content-Type` e o corpo estão lá, o problema é de quem lê.

1.5 Cabeçalhos que mudam o comportamento

A maioria dos cabeçalhos é informação. Três decidem o que acontece.

`Content-Type` diz em que formato o corpo **está escrito**. É ele que acabou de decidir se o `$_POST` seria preenchido.

`Accept` diz o que o cliente **aceita receber**. Um cliente que manda `Accept: application/json` está dizendo que não quer a sua página de erro em HTML — e devolver HTML mesmo assim é o que faz um aplicativo mostrar uma tela branca em vez da mensagem.

`Authorization` carrega a credencial. Sai assim, e o `$_SERVER` traduz o nome trocando o traço por sublinhado e acrescentando o prefixo `HTTP_`:

```
> Authorization: Bearer abc123
```

```
$_SERVER['HTTP_AUTHORIZATION']
```

Na resposta, quem escreve cabeçalho é a função `header()`, e o status é `http_response_code()`:

```
header('Content-Type: application/json');  
http_response_code(201);
```

1.6 O servidor não lembra de você

HTTP é **sem estado**: cada requisição chega sozinha, sem nenhuma memória da anterior. O servidor não sabe quem você é, o que você acabou de fazer nem que existe uma tela aberta do outro lado.

Isso é o mesmo modelo que o capítulo 1 do volume 1 descreveu por dentro — cada requisição começa do zero e morre no fim — agora visto de fora, no protocolo.

A consequência prática: **tudo que o servidor precisa lembrar tem que vir junto no pedido, ou estar guardado em algum lugar fora dele**. O navegador resolve isso reenviando um cookie a cada requisição; o PHP usa esse cookie para reencontrar um arquivo de sessão no disco do servidor. Uma API costuma resolver de outro jeito, mandando a credencial no `Authorization` a cada chamada.

Os dois são a mesma ideia: a requisição carrega a identidade, porque a conexão não carrega nada.

1.7 Um endpoint cru

Junte tudo. Sem framework, sem biblioteca, sessenta linhas de nada:

API.PHP

php

```
1  <?php
2
3  declare(strict_types=1);
4
5  header('Content-Type: application/json');
6
7  if ($_SERVER['REQUEST_METHOD'] !== 'POST') {
8      http_response_code(405);
9      echo json_encode(['erro' => 'Use POST']);
10     exit;
11 }
12
13 $tipo = $_SERVER['CONTENT_TYPE'] ?? '';
14
15 if (!str_starts_with($tipo, 'application/json')) {
16     http_response_code(415);
17     echo json_encode(['erro' => 'Envie application/json']);
18     exit;
19 }
20
21 $dados = json_decode(file_get_contents('php://input'), true);
22
23 if (!is_array($dados)) {
24     http_response_code(400);
25     echo json_encode(['erro' => 'JSON inválido']);
26     exit;
27 }
28
29 $faltando = [];
30
31 foreach (['exemplar_id', 'leitor_id'] as $campo) {
32     if (!isset($dados[$campo])) {
33         $faltando[] = $campo;
34     }
35 }
36
37 if ($faltando !== []) {
38     http_response_code(422);
39     echo json_encode([
40         'erro' => 'Campos obrigatórios',
41         'campos' => $faltando,
42     ]);
43     exit;
44 }
45
46 http_response_code(201);
47
48 echo json_encode([
49     'id' => 4471,
50     'exemplar_id' => (int) $dados['exemplar_id'],
```

```

51     'leitor_id' => (int) $dados['leitor_id'],
52 ];

```

REQUISIÇÃO **POST** /emprestimos

Content-Type: application/json

```
{ "exemplar_id": 812, "leitor_id": 47 }
```

RESPOSTA **201 Created**

Content-Type: application/json

```

{
  "id": 4471,
  "exemplar_id": 812,
  "leitor_id": 47
}

```

Duas coisas para reparar antes de seguir.

A primeira: o 405 e o 415 existem porque o arquivo responde a **um** caminho e **um** verbo. Um programa que atende vários precisa decidir qual código roda, e essa decisão tem nome — é a única coisa que falta aqui para isso virar um sistema.

A segunda: os (int) na resposta. O que chegou era texto; o que sai é número. A fronteira converte, e é por isso que ela é o único lugar do programa que precisa saber que HTTP existe.

■ NA SUA CARREIRA

“O cliente não está mandando” e “o servidor não está recebendo” são a mesma discussão vista de dois lados, e ela consome tardes inteiras porque cada lado olha só para o próprio log.

O que encerra em dois minutos é uma troca de curl -v: quem acusa cola a saída inteira, com a linha da requisição, os cabeçalhos e o corpo. A partir daí não há mais opinião — ou o Content-Length é zero, ou não é.

Aprenda a ler essa saída antes de precisar dela numa reunião com o time do aplicativo. É a diferença entre participar da conversa e esperar a conclusão.

RESUMO

- Requisição e resposta são texto com a mesma forma: primeira linha, cabeçalhos, linha em branco, corpo.
- A linha em branco é a fronteira; echo antes de header() a antecipa e estraga a resposta.
- Sete status cobrem a API inteira: 200, 201, 204, 400, 401, 404 e 422.

- `$_GET`, `$_POST` e `$_SERVER` são superglobais, e tudo que chega por elas é texto.
- O `$_POST` só é preenchido com `x-www-form-urlencoded` e `multipart/form-data` com JSON, o corpo está em `php://input`.
- `Content-Type` diz como o corpo está escrito; `Accept`, o que o cliente aceita receber; `Authorization` carrega a credencial.
- HTTP é sem estado: a identidade viaja em cada requisição, porque a conexão não guarda nada.
- `header()` escreve cabeçalho e `http_response_code()` define o status.

✓ PONTO DE CONTROLE

Você lê uma troca inteira em `curl -v` e explica cada linha, sabe dizer sem testar se um `$_POST` vai chegar preenchido, lê o corpo cru de uma requisição JSON e escreve um endpoint que devolve o status certo para pedido malformatado, campo faltando e sucesso.

- 1 Para cada situação, diga qual status a resposta deve ter:
 1. O empréstimo foi registrado com sucesso.
 2. O leitor 913 não existe no cadastro.
 3. O corpo chegou com `leitor_id` mas sem `exemplar_id`.
 4. O cliente mandou o corpo em XML.
 5. A devolução foi registrada e não há nada para devolver no corpo.
- 2 Acrescente ao `api.php` o tratamento do verbo `GET` em `.. /emprestimos/{id}`, devolvendo `200` com o empréstimo ou `404` se o número não existir.

Use estes dados fixos no lugar do banco:

```
$emprestimos = [
    4471 => ['exemplar_id' => 812, 'leitor_id' => 47],
];
```

- 3 Um aplicativo relata que a API do acervo “às vezes devolve HTML”.
... O time da API garante que devolve JSON sempre.

Os dois estão certos. Levante três situações em que uma API PHP devolve HTML sem que ninguém tenha escrito HTML, e diga como evitar cada uma.

CAPÍTULO

2

O que é uma API REST

Seu Juvenal quer um botão que renova tudo. A pergunta difícil não é como fazer — é o que acontece quando alguém aperta duas vezes.

AO FIM DESTE CAPÍTULO

Desenhar os endereços de uma API a partir dos recursos, separar o que é seguro do que é idempotente, escolher o status que já responde metade da pergunta, e saber o que pode e o que não pode mudar depois que alguém começou a consumir.

**Capítulo disponível na
versão completa**

CAPÍTULO

3

Um framework de quarenta linhas

Depois de escrever o roteador, a pergunta da estagiária foi a melhor do projeto: por que não usar este e pronto?

AO FIM DESTE CAPÍTULO

Escrever à mão o mínimo de um framework — front controller, roteador, contêiner e middleware — para conseguir explicar o ciclo de uma requisição em qualquer framework PHP, apontando onde cada peça entra.

ENQUANTO ISSO, NA EMPRESA Vinte minutos

A pasta `public/`, vazia no fim do volume 1, tinha dezenove arquivos `.php` duas semanas depois, e o endereço de cada tela terminava em um deles.

— Isso vira `/emprestar.php?tombo=2117` — disse Tainá. — O aplicativo vai consumir assim?

— Não. Vira `POST /emprestimos`.

— E como?

Dedé puxou o teclado e começou a escrever. Vinte minutos depois havia um arquivo com quarenta e poucas linhas, e as três rotas da biblioteca respondiam JSON.

Tainá leu o arquivo inteiro, de cima a baixo, sem interromper.

— Tá. Por que a gente não usa esse e pronto?

3.1 Um arquivo por página

O projeto da Casa Amarela cresceu do jeito mais natural do mundo: uma tela, um arquivo. É como o PHP foi feito para funcionar, e é o motivo de a linguagem ter conquistado a web.

Três coisas quebram quando o projeto passa de uma dúzia de arquivos.

O endereço vira mapa do disco. `/emprestar.php` conta a quem estiver do lado de fora como a pasta está organizada, e amarra a URL ao nome do arquivo. Renomear vira mudança de contrato.

Cada arquivo repete o começo. Conexão, autoload, cabeçalho de resposta, conferência de credencial. Dezenove vezes, e no dia em que uma delas mudar, dezenove lugares para lembrar.

Não existe um lugar para o que vale para todos. Registrar quanto tempo cada requisição levou, recusar quem não está autenticado, devolver JSON quando o programa quebra: cada um desses é uma linha que precisaria estar em todos os dezenove arquivos.

3.2 Front controller: uma porta só

A saída é antiga e tem nome: **toda requisição entra pelo mesmo arquivo**, que decide o que fazer.

1	antes	depois
2	/emprestar.php?tombo=2117	POST /emprestimos
3	/listar.php	GET /livros
4	/recibo.php?id=4471	GET /emprestimos/4471/recibo
5		↓
6		tudo entra por index.php

Para o servidor entregar tudo a um arquivo só, ele precisa de uma instrução. No Apache, é um arquivo de configuração na pasta pública:

.HTACCESS	text
<pre>RewriteEngine On RewriteCond %{REQUEST_FILENAME} !-f RewriteRule ^ index.php [L]</pre>	

As duas últimas linhas dizem: se o caminho pedido **não** for um arquivo que existe — uma imagem, um CSS —, mande para o `index.php`. É por isso que todo projeto PHP moderno tem uma pasta `public/` com quase nada dentro: só o `index.php` e os arquivos que devem mesmo ser servidos direto.

No servidor embutido do PHP, a mesma coisa se faz na linha de comando:

```
$ php -S localhost:8000 mini.php
```

3.3 Um roteador em vinte linhas

Com tudo entrando por uma porta, falta decidir para onde vai. Duas classes, e a primeira tem três linhas:

MINI.PHP	php
<pre>1 <?php 2 3 declare(strict_types=1); 4 5 final class Resposta 6 { 7 public function __construct(8 public readonly int \$status, 9 public readonly array \$corpo, 10) {} 11 }</pre>	

MINI.PHP	php
<pre>1 final class Roteador 2 {</pre>	

```

3     public function __construct(private array $rotas) {}
4
5     public function despachar(
6         string $metodo,
7         string $caminho,
8     ): Resposta {
9         foreach ($this->rotas as [$verbo, $padrao, $acao]) {
10             if ($verbo !== $metodo) {
11                 continue;
12             }
13
14             $regex = '#^' . preg_replace(
15                 '#\{(\w+)\}\#',
16                 '(?<$1>[^/]+)',
17                 $padrao,
18             ) . '#';
19
20             if (preg_match($regex, $caminho, $achados) === 1) {
21                 $parametros = array_filter(
22                     $achados,
23                     'is_string',
24                     ARRAY_FILTER_USE_KEY,
25                 );
26
27                 return $acao(...array_values($parametros));
28             }
29         }
30
31         return new Resposta(404, ['erro' => 'Rota não
32             encontrada']);
33     }

```

O laço é simples; a única linha que merece atenção é a que monta a expressão regular. Ela troca `{id}` por um grupo com nome que casa com qualquer coisa que não seja barra. `/livros/{id}` vira `#^/livros/(?<id>[^/]+)$#`.

Depois do casamento, `$achados` traz os pedaços duas vezes: com índice numérico e com o nome. O `array_filter` com `ARRAY_FILTER_USE_KEY` fica só com os de nome — e eles viram os argumentos da função da rota. **Despachar** Escolher qual código roda para uma requisição e chamá-lo. É a única coisa que um roteador faz, e é a peça que todo framework tem no meio. Tabela de rotas de um lado, requisição do outro, uma função no fim. A tabela de rotas é um array, e cada linha tem verbo, padrão e o que fazer:

MINI.PHP

php

```

1     $acervo = [
2         12 => ['titulo' => 'Dom Casmurro', 'ano' => 1899],

```

```

3      31 => ['titulo' => 'Vidas Secas', 'ano' => 1938],
4  ];
5
6  $rotas = [
7      ['GET', '/livros',
8          fn(): Resposta => new Resposta(200, $acervo)],
9
10     ['GET', '/livros/{id}',
11         function (string $id) use ($acervo): Resposta {
12             $id = (int) $id;
13
14             if (!isset($acervo[$id])) {
15                 return new Resposta(404, [
16                     'erro' => "Livro {$id} não existe",
17                 ]);
18             }
19
20             return new Resposta(200, ['id' => $id] +
21                 $acervo[$id]);
22         }],
23     ['POST', '/emprestimos',
24         fn(): Resposta => new Resposta(201, ['id' => 4471])],
25 ];

```

E o front controller, que é o fim do arquivo, tem seis linhas:

MINI.PHP	php
<pre> 1 function enviar(Resposta \$r): void 2 { 3 http_response_code(\$r->status); 4 header('Content-Type: application/json'); 5 echo json_encode(\$r->corpo, JSON_UNESCAPED_UNICODE); 6 } 7 8 \$roteador = new Roteador(\$rotas); 9 10 enviar(\$roteador->despachar(11 \$_SERVER['REQUEST_METHOD'], 12 parse_url(\$_SERVER['REQUEST_URI'], PHP_URL_PATH), 13)); </pre>	

Suba e experimente:

```
$ php -S localhost:8000 mini.php
```

```

1 $ curl -s localhost:8000/livros/31
2 {"id":31,"titulo":"Vidas Secas","ano":1938}
3
4 $ curl -s localhost:8000/livros/99
5 {"erro":"Livro 99 não existe"}
6
7 $ curl -s localhost:8000/nada
8 {"erro":"Rota não encontrada"}

```

● O ESSENCIAL

Repare no que o roteador **não** é: um switch gigante com `$_SERVER` lá dentro. A diferença não é estética. Um switch mistura a decisão de rota com o trabalho de cada rota, e cresce junto com as duas. Aqui a decisão está numa classe que não sabe nada sobre livros, e o trabalho está numa tabela que não sabe nada sobre expressões regulares.

Trocar o roteador por outro não toca em nenhuma rota. É essa separação que faz a palavra “framework” significar alguma coisa.

3.4 O contêiner ingênuo

O projeto tem classes que dependem de outras: um serviço de empréstimo precisa de um repositório, que precisa de uma conexão. Montar isso à mão em cada rota é a repetição de sempre:

```

$servico = new ServicoDeEmprestimo(
    new RepositorioDeLivros(
        new Conexao()
    )
);

```

O `Servicos` do capítulo 29 do volume 1 resolveu isso com uma fábrica escrita à mão para cada classe — e o exercício 2 daquele capítulo já apontava o que as fábricas têm de mecânico: ler o construtor, pedir cada tipo, passar na ordem. Um contêiner que lê o construtor sozinho dispensa as fábricas. Quinze linhas resolvem para sempre:

MINI.PHP

php

```

1 final class Container
2 {
3     private array $feitos = [];
4
5     public function obter(string $classe): object
6     {
7         if (isset($this->feitos[$classe])) {
8             return $this->feitos[$classe];
9         }

```

```

10
11     $construtor = (new ReflectionClass($classe))
12         ->getConstructor();
13
14     $argumentos = [];
15
16     foreach ($construtor?->getParameters() ?? [] as $p) {
17         $tipo = $p->getType();
18
19         if (!$tipo instanceof ReflectionNamedType
20             || $tipo->isBuiltin()) {
21             throw new RuntimeException(
22                 "Não sei construir \${$p->getName()}"
23             );
24         }
25
26         $argumentos[] = $this->obter($tipo->getName());
27     }
28
29     return $this->feitos[$classe]
30         = new $classe(...$argumentos);
31 }
32 }

```

`ReflectionClass` é a parte do PHP que permite ao programa **olhar para o próprio código**: quais métodos uma classe tem, quais parâmetros um método recebe, de que tipo cada um é. O contêiner usa isso para ler a lista de parâmetros do construtor e resolver cada um, recursivamente.

```

$c = new Container();

$servico = $c->obter(ServicoDeEmprestimo::class);

```

```
(conexao criada)
```

Uma linha, e a corrente inteira foi montada. Peça de novo:

```

$outro = $c->obter(ServicoDeEmprestimo::class);

var_dump($servico === $outro);

```

```
bool(true)
```

Nenhuma conexão nova. O contêiner guarda o que já construiu, e é por isso que uma única `Conexao` atende o programa inteiro sem ninguém passar o `$pdo` de função em função.

ERRO COMUM

O contêiner acima é ingênuo de propósito, e o limite aparece rápido:

Não sei construir \$status

Ele só sabe resolver parâmetros que são **classes**. Um `int`, uma `string` ou uma configuração vinda de arquivo não têm como ser adivinhados — alguém precisa dizer o que colocar ali.

Guardar essa frase vale mais que o código: quando um framework reclamar que não consegue resolver uma dependência, é quase sempre isto. Um parâmetro que não é classe, ou uma interface sem ninguém ter dito qual implementação usar.

3.5 Middleware: a cebola antes de ter nome

Falta o lugar de pôr o que vale para todas as rotas. A ideia é embrulhar a ação em camadas: cada camada recebe a requisição, faz a parte dela, chama a de dentro e ainda vê a resposta voltando.

MINI.PHP

php

```
1  $acao = fn(string $req): string => "[acao:{$req}]";
2
3  $middlewares = [
4      fn(string $req, callable $proximo): string
5          => 'log(' . $proximo($req) . ')',
6
7      fn(string $req, callable $proximo): string
8          => 'auth(' . $proximo($req) . ')',
9  ];
10
11 $pipeline = array_reduce(
12     array_reverse($middlewares),
13     fn(callable $proximo, callable $atual): callable
14         => fn(string $req): string => $atual($req, $proximo),
15     $acao,
16 );
17
18 echo $pipeline('GET /livros'), "\n";
```

```
log(auth([acao:GET /livros]))
```

A saída é o desenho. O `log` embrulha o `auth`, que embrulha a ação. A requisição entra de fora para dentro e a resposta sai de dentro para fora, passando pelas mesmas camadas na ordem inversa.

O `array_reduce` com a lista invertida é o que monta essa boneca. Cada passo pega o que já foi montado — o “próximo” — e o embrulha na camada atual.

E o ganho prático aparece quando uma camada decide **não** chamar a seguinte:

```
fn(string $req, callable $proximo): string
=> autenticado($req) ? $proximo($req) : '401',
```

Aí a ação nunca roda. É assim que autenticação, limite de requisições e recusa por permissão funcionam em todo framework PHP — e a partir daqui nenhum deles precisa explicar o mecanismo para você.

3.6 O que ainda falta

O arquivo tem umas cem linhas contando tudo, e atende três rotas. Vale listar, sem pressa, o que ele não faz:

FALTA	O QUE DÁ ERRADO SEM ISSO
validação de entrada	cada rota confere na mão, e uma esquece
tratamento central de erro	um <code>TypeError</code> vira HTML no meio do JSON
camada de banco com migração	o esquema mora na cabeça de alguém
autenticação de verdade	não há sessão, token nem hash de senha
respostas padronizadas	cada rota inventa o formato do erro
testes com requisição falsa	só dá para testar subindo servidor
tarefas fora da requisição	e-mail atrasa a resposta do usuário
cache, log, fila	tudo vira arquivo em <code>/tmp</code>
documentação	o contrato existe só no código

TABELA 3.1 Cada linha dessa tabela é um trabalho de dias, e nenhuma delas é específica da Casa Amarela — é exatamente a mesma lista em qualquer projeto web do mundo.

3.7 O que você acabou de entender

As quatro peças deste capítulo não são uma simplificação didática de um framework. São a **forma** de todos eles.

1	requisição	
2	↓	
3	front controller	index.php
4	↓	
5	middlewares	camada, camada, camada
6	↓	
7	roteador	qual função roda
8	↓	
9	contêiner	monta o que a função precisa
10	↓	
11	sua função	a única parte que é sua
12	↓	
13	resposta	volta pelas camadas

Quando um framework falar em *kernel*, *pipeline*, *service provider* ou *route model binding*, o vocabulário vai ser novo e o desenho não. Ele vai ser este, com mais cuidado, mais casos tratados e mais anos de correção em cima.

■ NA SUA CARREIRA

Você vai ouvir, em algum momento, que “framework é para quem não sabe fazer na mão”. A resposta não é discordar — é concordar e continuar a frase. Fazer na mão levou vinte minutos e produziu cem linhas que atendem três rotas e não tratam erro, não validam, não autenticam e não têm um teste. Completar a tabela da seção anterior é um ano de trabalho, e o resultado seria um framework pior e com um mantenedor só.

O que este capítulo comprou não foi independência: foi a capacidade de **ler** o framework. Quando algo der errado três camadas abaixo do seu código, você vai saber que camadas são essas — e essa é a diferença entre abrir um chamado e abrir o arquivo.

```
acervo/
├── mini.php  front controller, roteador, contêiner, middleware
├── src/
│   ├── Acervo/
│   ├── Circulacao/
│   ├── Emprestimos/
│   └── Leitores/
├── composer.json
└── phpstan.neon
```

RESUMO

- Um arquivo por página amarra a URL ao disco, repete o começo em todo arquivo e não deixa lugar para o que vale para todos.
- Front controller faz tudo entrar por `index.php` o servidor precisa de uma regra de reescrita para isso.

- Roteador é uma tabela de rotas, um casamento de padrão e uma chamada — e não sabe nada sobre o domínio.
- `{id}` no padrão vira grupo com nome na expressão, e o grupo vira argumento.
- Contêiner usa reflexão para ler os parâmetros do construtor e montar a corrente de dependências, guardando o que já construiu.
- Contêiner ingênuo só resolve parâmetro que é classe; o resto alguém precisa declarar.
- Middleware embrulha a ação em camadas; a camada que não chama a próxima interrompe a requisição.
- O que falta no arquivo de cem linhas é, item por item, o que um framework entrega.

✓ PONTO DE CONTROLE

Você explica o ciclo de uma requisição em qualquer framework PHP apontando onde entram front controller, middleware, roteador e contêiner; escreve um roteador com parâmetro de caminho; e sabe dizer por que um contêiner não consegue resolver certa dependência.

- 1 Acrescente ao `mini.php` a rota `DELETE /livros/{id}`, que devolve `204` sem corpo quando o livro existe e `404` quando não existe.
Depois explique por que o `enviar()` precisa de um ajuste.
- 2 O roteador atual casa `{id}` com qualquer coisa que não seja barra.
.. Isso faz `GET /livros/abc` entrar na rota e chegar em `(int) 'abc'`, que é `0`.
Acrescente ao padrão a possibilidade de exigir um formato, para que `/livros/{id:\d+}` só case com dígitos — e `/livros/abc` devolva `404` sem nunca chamar a função.
- 3 Junte as peças: faça o `mini.php` usar o contêiner e uma cadeia de dois middlewares de verdade.
... O primeiro middleware mede quanto tempo a requisição levou e acrescenta o cabeçalho `X-Tempo`. O segundo recusa com `401` qualquer requisição sem o cabeçalho `Authorization`, **exceto** `GET`.
Escreva o código e diga em que ordem os dois devem ficar na lista, e por quê.

CAPÍTULO

4

O que é o Laravel

A resposta para a pergunta da estagiária tem duas partes: o que o framework faz por você, e o que ele decide sem perguntar.

AO FIM DESTE CAPÍTULO

Descrever o ciclo de uma requisição Laravel apontando, em cada etapa, o equivalente no micro-framework do capítulo anterior — e saber nomear o que a convenção cobra em troca do que entrega.

**Capítulo disponível na
versão completa**

PARTE 2

Dentro do Laravel

Estrutura, configuração, rotas, controllers e respostas. O framework chegando depois do problema, que é a única ordem em que ele faz sentido.

NESTA PARTE

- 5 O primeiro projeto
- 6 Configuração, ambiente e Artisan
- 7 Rotas e controllers
- 8 Requests e responses
- 9 Blade, quando a tela ainda é a resposta

CAPÍTULO

5

O primeiro projeto

Seu Juvenal viu a tela de boas-vindas, leu o nome do framework em letras grandes e perguntou se já dava para cadastrar os livros.

AO FIM DESTES CAPÍTULOS

Criar o projeto da Casa Amarela com o Laravel, entender o papel de cada pasta de topo, saber o que o `.env` guarda e por que ele não vai para o Git, e fazer a primeira rota responder.

ENQUANTO ISSO, NA EMPRESA Já tá pronto?

Dedê projetou a tela na parede da sala da associação para mostrar que o ambiente estava de pé. Fundo claro, o nome do framework no meio, alguns links de documentação em volta.

Seu Juvenal olhou por uns segundos.

— Bonito. Já tá pronto?

— Isso é a tela que vem de fábrica.

— Mas tem o nome do sistema ali.

— Tem o nome do framework.

Seu Juvenal apontou para a parede com o queixo, do jeito que a Vera aponta o monitor.

— Pra mim tá escrito que funciona.

5.1 `composer create-project`

Um comando cria o projeto inteiro:

```
$ composer create-project laravel/laravel casa-amarela
```

```
1  Creating a "laravel/laravel" project at "./casa-amarela"
2  Installing laravel/laravel (v12.0.0)
3  Created project in /home/dede/casa-amarela
4
5  > @php -r "file_exists('.env') || copy('.env.example', '.env');"
6
7  Loading composer repositories with package information
8  Updating dependencies
9  Package operations: 107 installs, 0 updates, 0 removals
10   - Installing symfony/polyfill-mbstring (v1.31.0)
11   - Installing illuminate/support (v12.0.0)
12   ...
13  Generating optimized autoload files
14
15  > @php artisan key:generate --ansi
16
17  INFO  Application key set successfully.
```

Cento e sete pacotes. É bem mais que os três do `var-dumper`, e a diferença é o que o capítulo anterior listou: fila, e-mail, sessão, validação, console, cache e o resto.

Repare nas duas últimas linhas, porque elas não são instalação: são o projeto se preparando. O `.env` foi criado a partir do `.env.example`, e uma

chave de aplicação foi gerada. Você vai ouvir falar das duas em cinco minutos.

5.2 A rota que responde em dois minutos

```
$ cd casa-amarela
$ php artisan serve
```

```
INFO Server running on [http://127.0.0.1:8000].

Press Ctrl+C to stop the server
```

Abra `routes/web.php` e acrescente quatro linhas:

ROUTES/WEB.PHP	php
<pre>1 Route::get('/saude', function () { 2 return [3 'status' => 'ok', 4 'hora' => now()->toIso8601String(), 5]; 6 });</pre>	

```
$ curl -s localhost:8000/saude
{"status":"ok","hora":"2026-01-13T09:41:12-03:00"}
```

Três coisas aconteceram sem você pedir.

A rota devolveu um **array**, e chegou JSON. O Laravel converte array e objeto automaticamente, e já manda o `Content-Type` certo.

O `now()` existe sem nenhum `use`. É uma das funções de conveniência que o framework registra globalmente — e devolve um objeto de data, não um texto.

E o endereço é `/saude`, não `/saude.php`. O front controller do capítulo 3 está lá, em `public/index.php`, fazendo exatamente o que o seu fazia.

★ VOCÊ SABIA

O esqueleto já vem com uma rota de saúde pronta, em `/up`, configurada no `bootstrap/app.php`. Ela existe para o serviço de monitoramento bater e saber se a aplicação está viva.

A `/saude` deste capítulo é sua, para você ver a rota funcionar. Num projeto de verdade, quem responde ao monitoramento é a `/up`.

5.3 As pastas que importam

O projeto tem doze pastas de topo. Seis delas você vai abrir todo dia.

```

casa-amarela/
├── app/      o seu código
├── bootstrap/ app.php monta a aplicação; cache/ é gerado
├── config/   um arquivo por assunto
├── database/ migrações, seeders e factories
├── public/   index.php e arquivos servidos direto
├── resources/ views Blade, CSS e JS de origem
├── routes/   web.php e console.php
├── storage/  log, cache, arquivos enviados, sessão
├── tests/
├── vendor/   do Composer, fora do Git
├── .env      configuração desta máquina, fora do Git
└── artisan  o comando de terminal do projeto

```

Três merecem um parágrafo agora.

app/ começa quase vazia: um controller base, um model de usuário e um provedor. Isso é de propósito — o Laravel não adivinha a sua arquitetura, e as pastas que você criar aqui dentro são decisão sua.

storage/ é a única pasta em que a aplicação **escreve**. Log, cache compilado das views, sessões e uploads moram aí. É também a origem do primeiro erro de quase todo mundo.

public/ é a única que o servidor web deve enxergar. Tudo que estiver fora dela — o **.env**, o **vendor/**, o seu código — fica inalcançável pela internet, e é isso que separa este projeto da pasta do Sistema, em que o backup de 2019 podia ser baixado por qualquer um.

5.4 O primeiro erro de permissão

Mais cedo ou mais tarde, e sempre no servidor:

```

The stream or file "/var/www/casa-amarela/storage/logs/laravel.log"
could not be opened in append mode: Failed to open stream:
Permission denied

```

A causa é sempre a mesma: quem roda o PHP no servidor não é você. É um usuário do sistema — **www-data** no Debian e no Ubuntu, **nginx** ou **apache** em outras distribuições — e ele precisa poder escrever em duas pastas.

```

$ sudo chown -R www-data:www-data storage bootstrap/cache
$ sudo chmod -R 775 storage bootstrap/cache

```

ERRO COMUM

A receita que aparece em fórum é `chmod -R 777 storage`. Ela funciona, e significa “qualquer usuário do servidor pode escrever aqui”.

Num servidor compartilhado, isso inclui os outros sites hospedados na mesma máquina. Num servidor só seu, inclui qualquer processo que um invasor consiga rodar com qualquer usuário.

O 775 com o dono certo resolve o mesmo problema e não abre a porta. A diferença entre os dois comandos é de dez segundos para escrever e de anos para descobrir que foi por ali.

5.5 `.env` : a configuração desta máquina

<code>.ENV</code>	text
1	<code>APP_NAME="Casa Amarela"</code>
2	<code>APP_ENV=local</code>
3	<code>APP_KEY=base64:0sT3qk9...</code>
4	<code>APP_DEBUG=true</code>
5	<code>APP_URL=http://localhost</code>
6	
7	<code>DB_CONNECTION=mysql</code>
8	<code>DB_HOST=127.0.0.1</code>
9	<code>DB_PORT=3306</code>
10	<code>DB_DATABASE=casa_amarela</code>
11	<code>DB_USERNAME=root</code>
12	<code>DB_PASSWORD=senha</code>

O `.env` guarda o que **muda de máquina para máquina**: endereço do banco, senha, se os erros aparecem na tela, para onde vão os e-mails. A sua máquina tem um, o servidor tem outro, e os dois nunca são iguais.

Três regras, e as três têm consequência.

O `.env` **não entra no Git**. Ele já vem no `.gitignore` do esqueleto. O que entra é o `.env.example`, com as mesmas chaves e sem os valores — é ele que conta a quem clonar o projeto o que precisa ser preenchido.

A `APP_KEY` **é usada para criptografar**. Sessões e dados cifrados dependem dela. Trocar a chave num sistema no ar derruba todas as sessões abertas; rodar sem ela dá erro na primeira requisição que precisar de criptografia. Ela é gerada uma vez, por ambiente, e guardada com as senhas.

`APP_DEBUG=true` **nunca vai para produção**. Com ela ligada, um erro devolve a página de diagnóstico do Laravel — que mostra o trecho do código, os valores das variáveis e, dependendo do ponto, o conteúdo do próprio `.env`.

● **O ESSENCIAL**

No Laravel 11 em diante o banco padrão do `.env.example` é SQLite, porque ele funciona sem instalar nada.

O projeto da Casa Amarela já tem um MySQL desde o capítulo 12 do volume 1, com quatro tabelas e dados dentro. Troque para `mysql` e aponte para o banco que já existe.

5.6 Servir o projeto: escolha uma

Quatro formas aparecem na documentação, e a única decisão errada é tentar as quatro no primeiro dia.

FORMA	QUANDO SERVE
<code>php artisan serve</code>	agora, e para o livro inteiro
Laravel Sail	quando a equipe precisa do mesmo ambiente
Valet	macOS, vários projetos ao mesmo tempo
Docker próprio	quando produção já é Docker

TABELA 5.1 O `artisan serve` é o servidor embutido do PHP com as rotas do projeto. Não serve para produção e serve perfeitamente para aprender.

⚡ **ERRO COMUM**

A tentação do primeiro dia é começar por Docker, “porque é assim que se faz profissionalmente”. Em geral é, e no primeiro dia o resultado é duas horas depurando volume, permissão e rede — nenhuma das quais tem a ver com PHP. Container resolve um problema real: fazer a sua máquina parecer com o servidor. Esse problema aparece quando existe um servidor e existe uma equipe. Antes disso, ele é só um problema a mais.

5.7 O que o `index.php` faz

Vale abrir o arquivo, porque ele tem doze linhas e você já conhece nove:

PUBLIC/INDEX.PHP

php

```

1  <?php
2
3  use Illuminate\Foundation\Application;
4  use Illuminate\Http\Request;
5
6  define('LARAVEL_START', microtime(true));
7
8  require __DIR__.'/../vendor/autoload.php';
9
10 $app = require_once __DIR__.'/../bootstrap/app.php';
11
12 $app->handleRequest(Request::capture());

```

Autoload do Composer, a aplicação montada pelo `bootstrap/app.php`, a requisição capturada das superglobais e entregue. Depois disso vêm as camadas, o roteador, o seu código e a resposta — o desenho do capítulo anterior, com nomes de gente grande.

```

casa-amarela/
├── .env          com o MySQL do projeto, fora do Git
├── .env.example as mesmas chaves, sem valores
├── bootstrap/app.php
├── public/index.php
└── routes/web.php com GET /saude respondendo

```

RESUMO

- `composer create-project laravel/laravel` cria o esqueleto, copia o `.env` e gera a `APP_KEY`.
- Uma rota que devolve array vira JSON com o cabeçalho certo, sem conversão à mão.
- `app/` começa quase vazia de propósito; a arquitetura interna é sua.
- `storage/` e `bootstrap/cache/` precisam ser graváveis pelo usuário do servidor web — com 775 e o dono certo, não com 777.
- `public/` é a única pasta que o servidor web enxerga; `.env` e `vendor/` ficam fora do alcance da internet.
- O `.env` guarda o que muda de máquina; ele fica fora do Git e o `.env.example` entra no lugar.
- `APP_KEY` cifra sessão e dados; `APP_DEBUG=true` em produção mostra código e configuração na tela.
- `php artisan serve` basta para aprender; Docker resolve um problema que ainda não existe no primeiro dia.

✓ **PONTO DE CONTROLE**

Você cria um projeto Laravel, sobe o servidor, escreve uma rota que responde JSON, explica o papel de cada pasta de topo e sabe dizer por que o `.env` não entra no repositório e o que a `APP_KEY` protege.

- 1 Um colega clonou o repositório do projeto e recebeu, na primeira requisição:

```
No application encryption key has been specified.
```

Diga o que aconteceu, qual comando resolve e por que esse erro não pode ser evitado versionando o `.env`.

- 2 Acrescente à `/saude` três informações úteis para quem monitora: a versão do PHP, se a aplicação está em modo de depuração e se o banco responde.

Cuidado com o que você expõe: a rota é pública.

- 3 O deploy da Casa Amarela é feito copiando a pasta do projeto para o servidor. Na primeira tentativa, a aplicação subiu e a página quebrou com erro de permissão em `storage/logs`.

O estagiário rodou `chmod -R 777 storage` e funcionou.

Escreva o que você diria na revisão: por que funcionou, qual é o risco concreto, qual é a correção, e como evitar que a pasta volte a ser copiada com dono errado no próximo deploy.

CAPÍTULO

6

Configuração, ambiente e Artisan

A multa passou a sair zerada em produção, e em nenhuma outra máquina. A causa foi um comando de otimização rodado no deploy.

AO FIM DESTE CAPÍTULO

Configurar o projeto sem espalhar `env()` pelo código, entender por que o cache de configuração derruba quem faz isso, e usar o Artisan como modelo mental em vez de lista de comandos decorada.

**Capítulo disponível na
versão completa**

CAPÍTULO

7

Rotas e controllers

O índice do Sistema tinha 137 linhas, e cada linha era um nome de arquivo. Ninguém nunca leu esse índice inteiro.

AO FIM DESTES CAPÍTULOS

Registrar as rotas da API desenhadas antes, com nome, restrição e busca automática do registro — mantendo os controllers pequenos e sabendo justificar o formato escolhido para cada um.

**Capítulo disponível na
versão completa**

CAPÍTULO

8

Requests e responses

A estagiária enviou uma capa de livro chamada ../teste.php. O Sistema guardou, e o servidor executou.

AO FIM DESTES CAPÍTULOS

Entrar e sair da aplicação com objetos: ler o dado certo do Request, devolver resposta com status e cabeçalho corretos, e receber arquivo sem confiar em nada que o cliente mandou.

**Capítulo disponível na
versão completa**

CAPÍTULO

9

Blade, quando a tela ainda é a resposta

A Vera não quer aplicativo. Ela quer um campo de busca e um botão, como o do Sistema, que funcionava.

AO FIM DESTE CAPÍTULO

Entregar o painel que a bibliotecária usa no balcão — listagem, formulário e ação — entendendo o escape automático, o token de formulário e onde fica o limite entre Blade e front-end de verdade.

**Capítulo disponível na
versão completa**

PARTE 3

Eloquent sobre o SQL que você já sabe

O ORM entra depois do SQL. Cada método tem uma consulta correspondente que você consegue escrever à mão — e é isso que separa usar o Eloquent de depender dele.

NESTA PARTE

- 10 Migrations, seeders e factories
- 11 Eloquent
- 12 Relacionamentos

CAPÍTULO

10

Migrations, seeders e factories

Onze e quarenta de uma quinta-feira. O terminal escreveu Dropping all tables e ele leu o endereço do banco na linha de cima.

AO FIM DESTE CAPÍTULO

Versionar o esquema do banco em arquivos que contam a história das mudanças, alterar coluna sem derrubar a aplicação, e ter dados de desenvolvimento que qualquer pessoa da equipe consegue reproduzir.

**Capítulo disponível na
versão completa**

CAPÍTULO

11

Eloquent

O cadastro de leitor aceitava um campo que o formulário não tinha. Quem descobriu isso virou bibliotecária-chefe em nove segundos.

AO FIM DESTE CAPÍTULO

Usar o ORM sabendo qual consulta cada método produz, proteger o cadastro contra campos que ninguém pediu, converter valores na entrada e na saída do banco, e reconhecer quando um model virou depósito.

ENQUANTO ISSO, NA EMPRESA Perfil admin

Tainá estava testando o cadastro público de leitor com `curl`, porque formulário esconde campo e ela queria ver o que a API aceitava de verdade.

```
$ curl -s -X POST localhost:8000/api/leitores \
  -H "Content-Type: application/json" \
  -d '{ "nome": "Teste", "documento": "11122233344",
        "perfil": "admin" }'
```

```
{ "id": 4102, "nome": "Teste", "documento": "11122233344",
  "perfil": "admin" }
```

Ela leu duas vezes.

- Dedé, o formulário de cadastro tem campo de perfil?
- Não. Perfil só a Vera muda, na tela dela.
- Então por que ele aceitou?

Dedé abriu o model. A segunda linha da classe era:

```
protected $guarded = [];
```

- Isso quer dizer que nada é protegido.
- Isso quer dizer que eu sou administradora agora.

11.1 Eloquent não é SQL mágico

Cada método do Eloquent produz uma consulta, e você consegue escrever todas elas à mão — foi o que os capítulos da Parte 2 do volume 1 fizeram.

Este capítulo inteiro segue uma regra: **nenhum método aparece sem a consulta ao lado.**

```
Livro::where('assunto', 'infantil')->orderBy('titulo')->get();
```

```
SELECT * FROM livros WHERE assunto = ? ORDER BY titulo ASC
```

O `?` não é enfeite: o Eloquent monta consulta preparada, sempre, pelo motivo do capítulo 15 do volume 1. O valor viaja separado do comando.

● O ESSENCIAL

O ORM não poupa você de saber SQL. Ele poupa você de **escrever** o SQL repetitivo — o `SELECT` de cinco colunas, o `INSERT` com oito campos, o `JOIN` óbvio. O SQL que importa continua sendo escrito por alguém. A diferença é que, com o ORM, você só precisa escrever o que é interessante.

11.2 O model e as convenções

APP/MODELS/LIVRO.PHP

php

```

1  <?php
2
3  declare(strict_types=1);
4
5  namespace App\Models;
6
7  use Illuminate\Database\Eloquent\Model;
8
9  class Livro extends Model
10 {
11     protected $fillable = [
12         'titulo', 'autor', 'isbn', 'assunto', 'ano',
13     ];
14 }
```

Seis linhas úteis, e o model já sabe ler e gravar. As convenções que ele assumiu:

CONVENÇÃO	VALOR ASSUMIDO
tabela	plural do nome da classe
chave primária	id
datas	created_at e updated_at

TABELA 11.1 Todas são ajustáveis, e a primeira precisa ser ajustada com frequência num projeto em português.

⚠ ERRO COMUM

O pluralizador do Laravel fala inglês. Ele resolve `Livro` para `livros` por acaso — porque acrescentar `s` funciona —, e erra o resto:

CLASSE	O LARAVEL PROCURA	A TABELA SE CHAMA
<code>Livro</code>	<code>livros</code>	<code>livros</code>
<code>Exemplar</code>	<code>exemplars</code>	<code>exemplares</code>
<code>Leitor</code>	<code>leitores</code>	<code>leitores</code>
<code>Emprestimo</code>	<code>emprestimos</code>	<code>emprestimos</code>

TABELA 11.2 Dois acertos e dois erros, e o erro aparece como `Table 'casa_amarela.exemplars' doesn't exist` na primeira consulta.

A correção é uma linha, e vale declarar **em todos os modelos** de um projeto em português — inclusive nos que dariam certo, para que ninguém precise lembrar de quais são quais:

```
protected $table = 'exemplares';
```

11.3 Active Record: o objeto que sabe se salvar

```
$livro = new Livro();  
$livro->titulo = 'Vidas Secas';  
$livro->autor = 'Graciliano Ramos';  
$livro->assunto = 'literatura';  
$livro->save();
```

```
INSERT INTO livros (titulo, autor, assunto, updated_at, created_at)  
VALUES (?, ?, ?, ?, ?)
```

```
$livro->ano = 1938;  
$livro->save();
```

```
UPDATE livros SET ano = ?, updated_at = ? WHERE id = ?
```

O mesmo método faz as duas coisas, e o objeto decide qual pela existência da chave. Isso é o padrão **Active Record**: o registro e o comportamento no mesmo objeto.

É conveniente e tem um custo que aparece em dois pontos. O objeto carrega uma dependência do banco para todo lugar aonde for — e testar a regra que vive dentro dele exige banco. É a razão de este livro manter

a regra de negócio em serviços, e usar o model para o que ele faz bem: ir e voltar da tabela.

11.4 Buscar, e o `null` que escapa

MÉTODO	SQL	QUANDO NÃO ACHA
<code>find(12)</code>	<code>WHERE id = ?</code>	devolve <code>null</code>
<code>findOrFail(12)</code>	<code>WHERE id = ?</code>	lança, e vira 404
<code>first()</code>	<code>LIMIT 1</code>	devolve <code>null</code>
<code>findOrFail()</code>	<code>LIMIT 1</code>	lança, e vira 404
<code>value('titulo')</code>	<code>SELECT titulo ... LIMIT 1</code>	devolve <code>null</code>

TABELA 11.3 Os quatro primeiros diferem numa coisa só, e é a mais importante.

```
$livro = Livro::find($id);
echo $livro->titulo;
```

```
Attempt to read property "titulo" on null
```

O `find` devolveu `null` e o erro aparece na linha seguinte — ou trinta linhas depois, ou na view. A versão com `findOrFail` falha no lugar certo, com a resposta certa:

```
$livro = Livro::findOrFail($id);
```

```
404 Not Found
```

• O ESSENCIAL

Use `find` quando `null` for **uma resposta possível** que você vai tratar ali mesmo. Use `findOrFail` quando a ausência for um erro.

Na prática, dentro de um controller com route model binding do capítulo 7, você raramente escreve qualquer um dos dois: o framework já buscou.

E o método que parece inofensivo e não é:

```
Livro::all();
```

```
SELECT * FROM livros
```

Quatro mil linhas na memória para mostrar vinte. Numa tabela de empréstimos com anos de histórico, é a consulta que derruba o servidor numa terça à tarde. O `paginate(20)` existe para isso.

11.5 Mass assignment

Volta ao `perfil=admin`.

```
Leitor::create($request->all());
```

O `create` recebe um array e preenche o registro com ele. A pergunta é: **quais campos ele aceita?**

```
protected $fillable = ['nome', 'documento', 'telefone'];
```

Com `$fillable`, ele aceita esses três e **ignora em silêncio** qualquer outro. O `perfil` que a Tainá mandou é descartado.

A alternativa, `$guarded`, é a lista do que **não** pode ser preenchido — e `$guarded = []` quer dizer “nada é proibido”, que é exatamente o que o `model` do `Leitor` dizia.

ERRO COMUM

`$guarded = []` aparece em tutorial porque tira um obstáculo de quem está aprendendo. Ele transforma toda coluna da tabela num campo público de formulário.

O dano depende do que existe na tabela: `perfil`, `saldo`, `aprovado`, `id_da_empresa`. Em qualquer sistema com mais de um nível de acesso, essa linha é uma escalada de privilégio esperando alguém curioso.

A regra: **`$fillable` sempre, com a lista escrita à mão**. Escrever a lista é o momento em que você decide, campo a campo, o que o mundo de fora pode preencher.

E existe uma proteção a mais, que transforma o descarte silencioso em erro:

APP/PROVIDERS/APPSERVICEPROVIDER.PHP

php

```
1 public function boot(): void
2 {
3     Model::preventSilentlyDiscardingAttributes(
4         ! $this->app->isProduction()
5     );
6 }
```

Em desenvolvimento, mandar um campo que não está no `$fillable` passa a lançar exceção. É como você descobre, na sua máquina, que o formulário está enviando um campo que o model ignora — em vez de descobrir pela ausência do dado, três semanas depois.

11.6 Casts: o tipo certo dos dois lados

O banco guarda texto, número e data. O seu código quer enums e objetos de valor. O `casts()` é a tradução, nos dois sentidos:

APP/MODELS/EXEMPLAR.PHP

php

```
1  protected function casts(): array
2  {
3      return [
4          'estado' => StatusExemplar::class,
5          'adquirido_em' => 'immutable_date',
6      ];
7  }
```

```
$exemplar = Exemplar::find(1);
```

```
$exemplar->estado;           // StatusExemplar::Bom
$exemplar->estado->rotulo();  // 'Disponível'
$exemplar->adquirido_em;     // DateTimeImmutable
```

A coluna continua sendo `VARCHAR(20)` com `'bom'` dentro. O que muda é que nenhum ponto do seu código volta a comparar string solta — a lição do capítulo 23 do volume 1 passa a valer também na fronteira do banco.

Para o `Dinheiro`, que não é enum nem data, a conversão é uma classe:

APP/CASTS/DINHEIROCAST.PHP

php

```
1  <?php
2
3  declare(strict_types=1);
4
5  namespace App\Casts;
6
7  use App\Emprestimos\Dinheiro;
8  use Illuminate\Contracts\Database\Eloquent\CastsAttributes;
9
10 class DinheiroCast implements CastsAttributes
11 {
12     public function get($model, $chave, $valor, $atributos): ?
13     Dinheiro
14     {
15         return $valor === null
16             ? null
17             : Dinheiro::emCentavos((int) $valor);
18     }
19 }
```

```

18
19     public function set($model, $chave, $valor, $atributos): ?int
20     {
21         return $valor?->centavos;
22     }
23 }

```

```
'multa_em_centavos' => DinheiroCast::class,
```

Agora `$emprestimo->multa_em_centavos` devolve um `Dinheiro`, com `formatado()` e `mais()` — e a soma com um inteiro de dias, que o capítulo 23 do volume 1 fechou, continua fechada depois de passar pelo banco.

11.7 Accessor: valor calculado com cara de coluna

APP/MODELS/EMPRESTIMO.PHP

php

```

1  use Illuminate\Database\Eloquent\Casts\Attribute;
2
3  protected function diasDeAtraso(): Attribute
4  {
5      return Attribute::make(
6          get: fn (): int => max(
7              0,
8              $this->devolver_ate->diffInDays(now(), absolute:
false),
9          ),
10         );
11 }

```

```
$emprestimo->dias_de_atraso; // 3
```

Parece coluna e não é: é calculado a cada leitura. Vale a pena quando o valor é derivado de outros que já estão na tabela — e é exatamente a razão de não existir uma coluna `dias_de_atraso`, pelo motivo do capítulo 14 do volume 1: número calculado não diverge da realidade.

11.8 Scope: a consulta que ganha nome

APP/MODELS/EMPRESTIMO.PHP

php

```

1  use Illuminate\Database\Eloquent\Builder;
2
3  public function scopeEmAberto(Builder $q): void
4  {
5      $q->whereNull('devolvido_em');
6  }
7
8  public function scopeVencidos(Builder $q): void
9  {
10     $q->whereDate('devolver_ate', '<', now());
11 }

```

```
Emprestimo::emAberto()->vencidos()->count();
```

```

SELECT COUNT(*) FROM empréstimos
WHERE devolvido_em IS NULL AND date(devolver_ate) < ?

```

O prefixo `scope` some na chamada, e os dois se combinam em qualquer ordem.

O ganho é o de sempre: a definição de “em aberto” passa a morar num lugar só. No dia em que empréstimo cancelado também contar como fechado, a mudança é uma linha — e não uma busca por `whereNull('devolvido_em')` em dezessete arquivos.

11.9 Ver o SQL antes de confiar

```
Livro::where('assunto', 'infantil')->orderBy('titulo')->toSql();
```

```
select * from `livros` where `assunto` = ? order by `titulo` asc
```

Para ver tudo que a requisição fez, com os valores e o tempo:

APP/PROVIDERS/APPSERVICEPROVIDER.PHP

php

```

1  if ($this->app->environment('local')) {
2      DB::listen(function ($consulta): void {
3          Log::debug($consulta->sql, [
4              'valores' => $consulta->bindings,
5              'ms' => $consulta->time,
6          ]);
7      });
8  }

```

● O ESSENCIAL

Ligue isso no primeiro dia de qualquer projeto com ORM e olhe o log depois de abrir três telas.

É a forma mais rápida de descobrir que a listagem de empréstimos está fazendo duzentas e uma consultas — e esse número tem nome, tem causa conhecida e é o assunto do próximo capítulo.

11.10 Model gigante: os sinais

O model começa com seis linhas e, se ninguém disser nada, chega a trezentas. Três sinais de que isso aconteceu:

Ele tem método que não fala com o banco. Cálculo de multa, regra de limite, decisão de prazo. Nada disso precisa de tabela para existir, e tudo isso passa a precisar quando mora ali.

Ele importa coisas que não são dados. Um model que usa `Mail`, `Http` ou `Storage` deixou de representar uma linha e virou um processo.

Ele tem um método que você não consegue testar sem banco. Este é o sinal mais confiável, porque é verificável: se para conferir a regra da multa você precisa inserir um empréstimo, a regra está no lugar errado.

A saída não é quebrar o model em cinco. É **tirar de lá o que não é persistência**: a regra vai para o serviço ou para o objeto de valor, e o model fica com colunas, casts, escopos e relações.

■ NA SUA CARREIRA

“Model gordo ou controller gordo?” é uma discussão sem fim, e ela é a pergunta errada. As duas respostas juntam duas coisas que mudam por motivos diferentes: a forma de guardar e a regra do negócio.

O teste que resolve, e que você pode aplicar em qualquer projeto: **se a Casa Amarela trocasse o MySQL por outra coisa, quanto do código precisaria mudar?** Tudo que precisasse mudar é persistência. O resto é regra, e regra não devia estar num arquivo que só existe por causa de uma tabela. Não é um argumento sobre pureza: é sobre onde o teste fica barato.

```

casa-amarela/
├── app/
│   ├── Models/
│   │   ├── Livro.php      $table, $fillable
│   │   ├── Exemplar.php   cast do enum
│   │   ├── Leitor.php     $fillable sem perfil
│   │   └── Empréstimo.php  scopes, accessor, cast de Dinheiro
│   ├── Casts/
│   │   └── DinheiroCast.php
│   ├── Providers/
│   │   └── AppServiceProvider.php  DB::listen e a proteção de atri-
└── butos

```

RESUMO

- Todo método do Eloquent produz uma consulta, e ela é preparada.
- O pluralizador fala inglês: declare `$table` em projeto português.
- Active Record põe registro e comportamento no mesmo objeto — cômodo para persistir, caro para testar regra.
- `find` devolve `null` `findOrFail` lança e vira 404.
- `all()` traz a tabela inteira para a memória.
- `$fillable` é a lista do que o mundo de fora pode preencher; `$guarded = []` é escalada de privilégio esperando alguém curioso.
- `preventSilentlyDiscardingAttributes` transforma descarte em erro fora de produção.
- Casts convertem nos dois sentidos: enum, data imutável e objeto de valor.
- Accessor é valor calculado; scope é consulta com nome, e os dois moram num lugar só.
- `toSql()` e `DB::listen` mostram o que foi realmente executado.
- Model que tem regra sem banco, que chama serviço externo ou que não se testa sem tabela deixou de ser model.

✓ PONTO DE CONTROLE

Você escreve consultas com Eloquent e mostra o SQL correspondente, protege o cadastro com `$fillable`, converte enum e objeto de valor com casts, dá nome a uma consulta com scope, e reconhece pelos três sinais quando a regra vazou para dentro do model.

1 Escreva o SQL que cada chamada produz:

```
Livro::where('ano', '>=', 2000)->count();
Exemplar::where('estado', 'bom')->pluck('tombo');
Leitor::orderBy('nome')->paginate(20);
Emprestimo::whereNull('devolvido_em')->latest()->first();
```

2 O model `Leitor` tem `$guarded = []` e a tabela tem as colunas `nome`, `documento`, `telefone`, `perfil` e `bloqueado_em`.

Corrija o model e escreva o que muda no controller de cadastro público e no controller da tela da Vera, que **precisa** poder mudar o perfil.

3 Este model chegou para revisão. Ele funciona.

```
1 class Emprestimo extends Model
2 {
3     protected $guarded = [];
4
5     public function calcularMulta(): float
```

```
6      {
7          $dias = (strtotime('now')
8              - strtotime($this->devolver_ate)) / 86400;
9
10         if ($dias <= 0) {
11             return 0;
12         }
13
14         $valor = $dias * 0.8;
15
16         if ($this->leitor->perfil === 'estudante') {
17             $valor = $valor / 2;
18         }
19
20         Mail::to($this->leitor->email)
21             ->send(new AvisoDeMulta($valor));
22
23         return $valor;
24     }
25 }
```

Aponte os problemas por categoria — segurança, correção, desenho — e diga onde cada pedaço deveria morar.

CAPÍTULO

12

Relacionamentos

A listagem do acervo levava quatro segundos. O log mostrou 143 consultas para mostrar 47 livros.

AO FIM DESTES CAPÍTULO

Modelar as ligações do domínio com o tipo certo de relacionamento, reconhecer o N+1 contando consultas, e corrigi-lo provando a correção com o mesmo contador.

**Capítulo disponível na
versão completa**

PARTE 4

A API de verdade

O que separa um CRUD de exercício de uma API que outra pessoa consome: validação, formato de saída, paginação, filtros e erro honesto.

NESTA PARTE

- 13 O CRUD completo
- 14 Validation e Form Requests
- 15 API Resources
- 16 Paginação, filtros e buscas
- 17 Erros padronizados

CAPÍTULO

13

O CRUD completo

Seu Juvenal perguntou se aquilo não era só um CRUD. A resposta tinha onze itens e estava no caderno da estagiária.

AO FIM DESTE CAPÍTULO

Ligar rotas, controllers, models e banco com as cinco operações devolvendo o status certo, fazer uma operação de negócio caber num commit, e terminar sabendo listar o que ainda está errado.

**Capítulo disponível na
versão completa**

CAPÍTULO

14

Validation e Form Requests

O mesmo livro estava três vezes no acervo. O ISBN era igual nos três — a não ser pelos hífens.

AO FIM DESTE CAPÍTULO

Recusar a entrada inválida na porta, tirar as regras de forma de dentro do controller, fazer o PATCH parcial funcionar sem apagar campo, e saber dizer o que não é validação e por isso não vai para lá.

**Capítulo disponível na
versão completa**

CAPÍTULO

15

API Resources

A coluna foi criada para a equipe anotar o que não dizia na frente do leitor. Durante onze dias, o leitor leu.

AO FIM DESTE CAPÍTULO

Separar o formato da resposta da estrutura da tabela, decidir campo a campo o que sai, carregar relacionamentos sem voltar ao N+1, e tratar a resposta como um contrato que outra pessoa já está usando.

**Capítulo disponível na
versão completa**

CAPÍTULO

16

Paginação, filtros e buscas

A busca por “acafrão” não encontrava “Açafrão”. Metade do acervo tinha sido cadastrada em teclados de licitação, sem cedilha.

AO FIM DESTE CAPÍTULO

Entregar uma listagem que aguenta o acervo inteiro: paginada com teto do servidor, ordenada de forma estável, filtrável sem escada de if, e com uma busca que encontra o livro que a pessoa digitou do jeito que deu.

**Capítulo disponível na
versão completa**

CAPÍTULO

17

Erros padronizados

O leitor ligou dizendo que deu erro. O log tinha 4.200 linhas e nenhuma pista de qual delas era dele.

AO FIM DESTES CAPÍTULOS

Fazer toda resposta de erro da API ter o mesmo formato, traduzir as exceções do domínio em status numa linha, nunca vaziar detalhe interno, e entregar ao suporte um número que leva direto à linha certa do log.

ENQUANTO ISSO, NA EMPRESA Deu erro

O telefone da biblioteca tocou às 10h20. Vera atendeu, ouviu, e passou para Tainá sem dizer nada, o que já era uma forma de dizer.

— Oi, é o Wellington. Eu tentei pegar um livro pelo aplicativo e deu erro.

— Que erro?

— Deu erro. Apareceu “Server Error”. Aí eu tentei de novo e deu de novo.

— Que horas foi isso?

— Agora. Quer dizer, uns dez minutos atrás. Ou quinze.

Tainá abriu o `laravel.log` de homologação, que naquela semana também atendia os primeiros testes com leitores de verdade. Quatro mil e duzentas linhas desde as nove. Procurou “Wellington”: nada — o log não tinha nome de ninguém. Procurou “ERROR”: trinta e uma ocorrências entre 10h00 e 10h20.

Escolheu a que parecia mais com empréstimo. Um `QueryException` com um *stack trace* de oitenta linhas. Passou quarenta minutos nele, achou a causa, abriu a tarefa.

Dedé leu a tarefa depois do almoço.

— Esse erro é do importador noturno. Rodou de novo às dez porque o Cléber disparou à mão.

— E o do Wellington?

— Deve ser um dos outros trinta.

Wellington ligou de novo às 14h. Tinha conseguido pegar o livro no balcão. Queria saber se o aplicativo ia cobrar duas vezes.

17.1 Três formatos de erro na mesma API

Faça o inventário do que a API da Casa Amarela devolve hoje quando algo dá errado. São quatro situações, e cada uma sai de um jeito:

```

1 // 422, validação do Form Request
2 {"message": "O título é obrigatório.",
3  "errors": {"título": ["O título é obrigatório."]}
4
5 // 409, abort() no controller
6 {"message": "Exemplar indisponível"}
7
8 // 404, route model binding
9 {"message": "No query results for model [App\\Models\\Livro]
10 99999"}
11
12 // 500, com APP_DEBUG=true
13 {"message": "SQLSTATE[23000]: Integrity constraint violation...",
14  "exception": "Illuminate\\Database\\QueryException",
15  "file": "/var/www/vendor/laravel/framework/...",
16  "line": 822,
17  "trace": [ ... 80 itens ... ]}

```

Quatro formatos parecidos o bastante para enganar e diferentes o bastante para quebrar. O aplicativo precisa de um `if` para cada um — e o `if` que lê `message` para decidir o que fazer quebra no dia em que alguém traduzir a frase.

E o último vaza, na ordem: o SQL com o nome da tabela e da restrição, o nome da classe interna, o caminho do projeto no servidor, a versão do framework pela estrutura de pastas, e oitenta linhas de caminho de execução. É um mapa do sistema entregue a quem fez a requisição.

• O ESSENCIAL

Um formato de erro, para a API inteira, é mais importante do que o formato ser perfeito. O cliente escreve o tratamento **uma vez** e confia nele. Três formatos certos, cada um num canto, são piores que um formato só, imperfeito, em todos os lugares.

17.2 O formato

A Casa Amarela adota um formato com quatro chaves:

```

1 {
2   "tipo": "exemplar-indisponível",
3   "mensagem": "O exemplar 2117 está emprestado.",
4   "campos": { "exemplar_id": ["Exemplar indisponível." ] },
5   "incidente": "01JHQ4Z8K3M2X9V7B5N1P0R6TW"
6 }

```

tipo é um identificador estável, em texto, que o cliente usa para decidir. Ele **nunca muda**, nem quando a mensagem é reescrita. É o que o `if` do aplicativo compara.

`mensagem` é para uma pessoa ler. Pode mudar, ser traduzida, ganhar acento. Nenhum código deve decidir nada com base nela.

`campos` é opcional e aparece quando o erro tem um campo culpado — sempre no 422, às vezes no 409. É o mesmo mapa do `errors` do capítulo 14.

`incidente` é opcional e aparece quando a falha é do servidor. É o número que o Wellington lê para a Tainá pelo telefone.

★ VOCÊ SABIA

Existe uma especificação para isso, a RFC 9457, *Problem Details for HTTP APIs*, com as chaves `type`, `title`, `status`, `detail` e `instance`. O formato da Casa Amarela é uma versão em português da mesma ideia. Se a sua API vai ser consumida por muitas equipes que você não conhece, seguir a RFC ao pé da letra poupa uma conversa em cada integração. Se é o aplicativo de uma biblioteca, as chaves em português são mais fáceis de ler, e o que importa é a disciplina — um formato, sempre.

17.3 O handler, onde toda exceção termina

No Laravel 11, o lugar em que toda exceção não capturada chega é `O bootstrap/app.php`, no bloco `withExceptions`. É o `set_exception_handler` do capítulo 21 do volume 1, com o framework em volta.

BOOTSTRAP/APP.PHP

php

```
1  ->withExceptions(function (Exceptions $exceptions) {
2      $exceptions->shouldRenderJsonWhen(
3          fn (Request $r) => $r->is('api/*') || $r->expectsJson(),
4      );
5
6      $exceptions->render(
7          fn (Throwable $e, Request $r) => $r->is('api/*')
8              ? RespostaDeErro::para($e)
9              : null,
10     );
11 }
```

O `render` recebe toda exceção. Se a requisição é da API, uma classe decide a resposta. Se não é, devolve `null`, e o Laravel faz o que faria normalmente — mostra a página de erro do painel Blade.

A classe que decide é uma tradução, e uma tradução é um `match`:

APP/HTTP/RESPOSTADEERRO.PHP

php

```
1  final class RespostaDeErro
2  {
```

```

3      public static function para(Throwable $e): JsonResponse
4      {
5          return match (true) {
6              $e instanceof ValidationException
7                  => self::validacao($e),
8              $e instanceof ExcecaoDeDominio
9                  => self::dominio($e),
10             $e instanceof ModelNotFoundException,
11             $e instanceof NotFoundHttpException
12                 => self::corpo(404, 'nao-encontrado',
13                     'O recurso pedido não existe.'),
14             $e instanceof AuthenticationException
15                 => self::corpo(401, 'nao-autenticado',
16                     'É preciso entrar para continuar.'),
17             $e instanceof AuthorizationException
18                 => self::corpo(403, 'sem-permissao',
19                     'Você não tem permissão para isso.'),
20             $e instanceof ThrottleRequestsException
21                 => self::corpo(429, 'muitas-requisicoes',
22                     'Muitas tentativas. Aguarde um pouco.'),
23             default => self::falhaInterna($e),
24         };
25     }
26
27     // ...
28 }

```

Uma lista, uma leitura. Quem quer saber o que a API responde para cada tipo de falha lê um arquivo.

17.4 Exceção de domínio vira status numa linha

As três exceções do capítulo 21 do volume 1 — `ExemplarIndisponivel`, `LimiteDeEmprestimosAtingido`, `LeitorComPendencia` — já carregam dados. O que falta é dizerem, cada uma, o seu `tipo`. Uma classe mãe resolve:

APP/EMPRESTIMOS/EXCECAODEDOMINIO.PHP

php

```

1  abstract class ExcecaoDeDominio extends RuntimeException
2  {
3      abstract public function tipo(): string;
4
5      public function status(): int
6      {
7          return 409;
8      }
9
10     public function campos(): ?array
11     {
12         return null;
13     }
14 }

```

APP/EMPRESTIMOS/EXEMPLARINDISPONIVEL.PHP

php

```

1  final class ExemplarIndisponivel extends ExcecaoDeDominio
2  {
3      public function __construct(
4          public readonly int $tombo,
5          public readonly StatusExemplar $estado,
6      ) {
7          parent::__construct(sprintf(
8              'O exemplar %d está %s.',
9              $tombo,
10             mb_strtolower($estado->rotulo()),
11         ));
12     }
13
14     public function tipo(): string
15     {
16         return 'exemplar-indisponivel';
17     }
18
19     public function campos(): array
20     {
21         return ['exemplar_id' => ['Exemplar indisponível.']];
22     }
23 }

```

E a tradução, no `RespostaDeErro`, é genérica:

```

1  private static function dominio(ExcecaoDeDominio $e):
   JsonResponse
2  {
3      return self::corpo(
4          $e->status(),
5          $e->tipo(),
6          $e->getMessage(),
7          $e->campos(),
8      );
9  }

```

Uma exceção nova do domínio — `ReservaExpirada`, digamos — herda de `ExcecaoDeDominio`, declara o seu `tipo`, e sai da API no formato certo sem que ninguém toque no handler.

REQUISIÇÃO **POST** `/api/emprestimos`

Content-Type: application/json

```
{ "exemplar_id": 2117, "leitor_id": 47 }
```

RESPOSTA **409 Conflict**

Content-Type: application/json

```
{
  "tipo": "exemplar-indisponivel",
  "mensagem": "O exemplar 2117 está emprestado.",
  "campos": { "exemplar_id": [ "Exemplar indisponível." ] }
}
```

É a resposta que o exercício 3 do capítulo 14 prometeu: o status diz a verdade, e a tela sabe qual campo destacar.

Com isso, os `abort(409, ...)` do controller do capítulo 13 podem ser trocados por `throw new ExemplarIndisponivel(...)`. E o controller deixa de conhecer status HTTP para regra de negócio — o que o capítulo 19 vai transformar em princípio.

⚡ ERRO COMUM

A tentação, depois de ter o handler, é usar exceção para tudo — inclusive para o que não é erro. “Livro não tem exemplares” não é exceção; é uma lista vazia. “Leitor sem empréstimos” é `[]`, com `200`. A régua é a do capítulo 21 do volume 1: exceção é para quando a operação **não pode continuar**. Uma consulta que não encontrou nada continuou e terminou bem.

17.5 O ⁴⁰⁴ do binding: útil e genérico demais

O ⁴⁰⁴ automático do route model binding é um presente, e vem com uma mensagem que não deveria sair:

```
No query results for model [App\Models\Livro] 999999
```

Ela revela o namespace e o nome da classe interna. Não é uma falha grave — não abre porta nenhuma —, mas é informação que o cliente não precisa e que um curioso anota.

O `match` acima já troca a mensagem pela genérica. E existe uma decisão a tomar sobre **quando o 404 é a resposta certa para outra coisa**: o leitor 47 pede o empréstimo 312, que existe e é de outra pessoa. A resposta é 403 ou 404?

RESPOSTA DIZ AO CLIENTE	
403	“isso existe, e não é seu”
404	“isso não existe para você”

TABELA 17.1 Os dois estão corretos em HTTP. O primeiro confirma a existência do recurso, e isso às vezes é informação demais.

Para empréstimos, confirmar que o número 312 existe diz pouco. Para uma rota como `/leitores?documento=...`, confirmar que um CPF está cadastrado na biblioteca diz muito. A regra da Casa Amarela: 404 quando a existência do recurso é, ela mesma, um dado pessoal. O capítulo 22 aplica.

17.6 500 não vaza nada

A última linha do `match` — O default — é a mais importante, porque é a que ninguém planejou:

APP/HTTP/RESPOSTADEERRO.PHPphp

```
1 private static function falhaInterna(Throwable $e): JsonResponse
2 {
3     $status = $e instanceof HttpExceptionInterface
4         ? $e->getStatusCode()
5         : 500;
6
7     return self::corpo(
8         $status,
9         'falha-interna',
10        'Algo deu errado do nosso lado. Informe o '
11        . 'código ao suporte.',
12        incidente: Incidente::atual(),
13    );
14 }
```

A resposta tem uma frase honesta e um número. Nenhuma linha de SQL, nenhum nome de classe, nenhum caminho de arquivo.

O detalhe continua existindo — **no log**, onde quem precisa pode ler. O Laravel registra a exceção antes de chamar o `render`, e o que vai para o cliente e o que vai para o log são decisões separadas.

▲ ATENÇÃO

Nada disso vale com `APP_DEBUG=true`. Com ele ligado, o Laravel acrescenta à resposta a exceção, o arquivo, a linha e o *trace* completo, por cima do que o handler montou, para ajudar em desenvolvimento. Em produção, `APP_DEBUG=false` é a primeira linha da lista do capítulo 28, e a que mais aparece em relatório de invasão. Há buscadores que indexam páginas de erro do Laravel com o modo de depuração ligado — e elas trazem, com frequência, o conteúdo do `.env`.

17.7 Código de incidente: o número que o suporte pede

O Wellington não tinha como dizer qual das 31 linhas era a dele. Com o incidente, ele tem: está na tela do aplicativo, e o aplicativo pode até oferecer um botão para copiar.

APP/SUPPORT/INCIDENTE.PHP

php

```

1  final class Incidente
2  {
3      public static function atual(): string
4      {
5          if (!Context::has('incidente')) {
6              Context::add('incidente', (string) Str::ulid());
7          }
8
9          return Context::get('incidente');
10     }
11 }
```

`Context` é uma área de dados que vive durante a requisição, e o Laravel **acrescenta tudo o que estiver nela a cada linha de log** escrita depois. Quando a exceção é registrada, a linha sai com o incidente anexado:

```
[2026-02-10 10:14:07] production.ERROR: SQLSTATE[23000]...
{"exception":"...", "incidente": "01JHQ4Z8K3M2X9V7B5N1P0R6TW"}
```

E a busca que levou quarenta minutos passa a levar um comando:

```
$ grep 01JHQ4Z8K3M2X9V7B5N1P0R6TW storage/logs/laravel.log
```

O ULID é um identificador único que começa pelo instante em que foi gerado, então os incidentes do mesmo minuto ficam próximos numa ordenação. É um detalhe pequeno que ajuda quando alguém diz “foi mais ou menos às dez”.

Por enquanto, o incidente nasce no momento do erro. No capítulo 20, ele passa a nascer **no início de toda requisição**, e cada linha de log daquela requisição — não só a do erro — carrega o mesmo número. É a diferença entre achar a exceção e achar a história inteira que levou até ela.

17.8 401 e 403 não são a mesma coisa

A lista do handler tem os dois, e eles são confundidos o bastante para merecer seção própria.

401 Unauthorized quer dizer: **não sei quem você é**. Faltou o token, ou ele expirou, ou é inválido. O cliente deve mandar a pessoa entrar de novo.

403 Forbidden quer dizer: **sei quem você é, e você não pode**. O token é válido. Entrar de novo não resolve nada.

O nome oficial do 401 é infeliz — ele diz *unauthorized* e significa “não autenticado”. A confusão vem daí, e a consequência é concreta: um aplicativo que recebe 401 quando deveria receber 403 manda a pessoa para a tela de login, ela entra, tenta de novo, volta para a tela de login. Ciclo infinito, com a pessoa convencida de que a senha está errada.

STATUS	PERGUNTA QUE FALHOU	O CLIENTE FAZ
401	quem é você?	pede login
403	você pode?	mostra “sem permissão”
404	isso existe?	mostra “não encontrado”
409	o mundo permite?	explica e oferece saída
422	o pedido está certo?	destaca os campos

TABELA 17.2 Cinco status, cinco perguntas, cinco comportamentos da tela. É a tabela que o aplicativo do leitor implementa, e é a razão de cada uma existir.

■ NA SUA CARREIRA

O formato de erro de uma API é a parte do contrato que mais custa mudar depois, porque todo cliente o trata em um lugar central — e mudar esse lugar central muda o comportamento de todas as telas ao mesmo tempo. Se você estiver no começo de uma API, gaste uma tarde nisso antes de ter vinte rotas. Se estiver numa API que já tem três formatos, o caminho é o de sempre: o formato novo passa a sair em todas as rotas, com os campos antigos mantidos ao lado até o último cliente migrar. É trabalhoso e é invisível, e é o tipo de coisa que diferencia quem mantém sistema de quem só escreve rota.

```

casa-amarela/
├── bootstrap/app.php    withExceptions → RespostaDeErro
├── app/Http/
│   └── RespostaDeErro.php    um match, a API inteira
├── app/Support/
│   └── Incidente.php        ULID no Context
└── app/Emprestimos/
    ├── ExcecaoDeDominio.php    tipo(), status(), campos()
    ├── ExemplarIndisponivel.php
    ├── LimiteDeEmprestimosAtingido.php
    └── LeitorComPendencia.php

```

■ O PROJETO AGORA

Fim da Parte 4. A API da Casa Amarela recusa entrada malformada na porta, devolve só o que decidiu devolver, lista o acervo inteiro com busca, filtro e teto, e responde todo erro no mesmo formato — com um número que leva o suporte à linha certa do log.

No caderno da Tainá, uma linha nova: *“o cliente não lê a mensagem, lê o tipo”*.

RESUMO

- Um formato de erro para a API inteira vale mais que três formatos corretos.
- `tipo` é estável e o cliente decide por ele; `mensagem` é para gente e pode mudar.
- O `render` no `withExceptions` recebe toda exceção; um `match` traduz `tipo` em resposta.
- Exceções de domínio herdam de uma classe mãe com `tipo`, `status` e `campos` uma nova sai no formato certo sem tocar no handler.
- O `404` do binding precisa de mensagem genérica; `403` ou `404` para recurso alheio é uma decisão sobre o que a existência revela.
- `500` não leva SQL, classe, arquivo nem *trace*; o detalhe vai para o log.
- `APP_DEBUG=true` em produção anula o handler e vaza o sistema.
- O `incidente` no `Context` aparece em toda linha de log e na resposta, e transforma uma busca de quarenta minutos num `grep`.
- `401` é “não sei quem é você”; `403` é “sei, e você não pode”.

✓ PONTO DE CONTROLE

Toda resposta de erro da API tem o mesmo formato, as exceções de domínio viram `409` sem `abort` no controller, nenhuma falha interna vaza detalhe, e você consegue explicar, para cada um dos cinco status de erro, o que a tela do cliente deve fazer ao recebê-lo.

- 1 Para cada situação, dê o status e o `tipo` que a API da Casa Amarela devolve:
 1. O token do aplicativo expirou.

2. O leitor tenta renovar um empréstimo que já foi renovado duas vezes.

3. O corpo do `POST /emprestimos` veio sem `leitor_id`.

4. O banco de dados está fora do ar.

5. Uma atendente tenta apagar um livro, e só admin pode.

2 Escreva `LeitorComPendencia` herdando de `ExcecaoDeDominio`. Ela carrega o id do leitor e a multa em aberto como `Dinheiro`, e a mensagem deve dizer o valor formatado.

Depois responda: o valor da multa deve aparecer na resposta da API? Para quem?

3 Uma pessoa do time propõe tirar o `RespostaDeErro` e, em vez dele, pôr um `try/catch` em cada controller, “para cada rota ter controle total da sua resposta de erro”.

Escreva os argumentos a favor que ela provavelmente tem, os três custos concretos da proposta, e o caso em que um `try/catch` no controller é, de fato, a escolha certa.

PARTE 5

Arquitetura e segurança

Onde a regra de negócio mora, quantas camadas o seu projeto realmente precisa, e as duas perguntas que toda requisição responde: quem é você e o que você pode.

NESTA PARTE

- 18 Service Container e injeção de dependência
- 19 Services: onde a regra de negócio mora
- 20 Middleware
- 21 Autenticação com Sanctum
- 22 Autorização: Gates e Policies

CAPÍTULO

18

Service Container e injeção de dependência

Dedé abriu o contêiner de vinte linhas ao lado do contêiner do Laravel. A estagiária reconheceu a reflexão antes dele apontar.

AO FIM DESTE CAPÍTULO

Entender o mecanismo que monta os objetos da aplicação, declarar dependência pelo construtor, escolher entre bind, singleton e scoped, e trocar uma implementação no teste sem tocar em quem a usa.

**Capítulo disponível na
versão completa**

CAPÍTULO

19

Services: onde a regra de negócio mora

Vera leu o método em voz alta e corrigiu uma regra que a equipe tinha entendido errado havia quatro meses. Levou dezoito segundos.

AO FIM DESTE CAPÍTULO

Decidir onde cada regra vive, tirar a regra de empréstimo do controller e pô-la num service que é uma fronteira de transação e não conhece HTTP, e saber quantas camadas o projeto realmente precisa.

**Capítulo disponível na
versão completa**

CAPÍTULO

20

Middleware

O middleware de log gravava o corpo inteiro de cada requisição. Durante três meses, isso incluiu o campo senha.

AO FIM DESTES CAPÍTULO

Entender a pipeline que envolve toda rota, escolher entre middleware global, de grupo e de rota, pôr a ordem a favor, limitar tentativas, e escrever um middleware próprio sabendo o que não pertence a ele.

**Capítulo disponível na
versão completa**

CAPÍTULO

21

Autenticação com Sanctum

*O Sistema guardava as senhas em MD5, sem sal.
Na hora do almoço, na máquina de um
desenvolvedor, sessenta por cento caíram em
quatro minutos.*

AO FIM DESTES CAPÍTULOS

Responder “quem é você” com segurança: guardar senha do jeito certo, migrar as que estão erradas, emitir e revogar tokens com Sanctum, e escrever um login cuja resposta de erro não entregue nada — nem no texto, nem no tempo.

**Capítulo disponível na
versão completa**

CAPÍTULO

22

Autorização: Gates e Policies

O leitor ligou animado: tinha descoberto que, trocando um número no endereço, dava para ver os livros de todo mundo.

AO FIM DESTES CAPÍTULOS

Responder “o que você pode” — inclusive quando o recurso é de outra pessoa: separar papel de permissão, escrever Policies por recurso, restringir listagens por quem pergunta, e testar cada permissão, porque ninguém testa à mão.

**Capítulo disponível na
versão completa**

PARTE 6

Depois da resposta

Eventos, filas, cache e log. O trabalho que não cabe dentro da requisição — e o que você vai querer ter medido quando algo quebrar.

NESTA PARTE

- 23 Events, jobs e filas
- 24 Cache, logs e o que se mede

CAPÍTULO

23

Events, jobs e filas

O aviso de devolução foi enviado 1.400 vezes para a mesma pessoa. O job não era idempotente, e o worker reiniciou no meio.

AO FIM DESTE CAPÍTULO

Tirar da requisição o trabalho que não precisa acontecer antes da resposta, desacoplar com eventos sem esconder o fluxo, rodar filas com worker que reinicia no deploy, e escrever jobs que podem falhar, voltar e rodar de novo sem repetir o efeito.

**Capítulo disponível na
versão completa**

CAPÍTULO

24

Cache, logs e o que se mede

A página do acervo ficou rápida e passou a mostrar como disponível um livro que estava emprestado. Por seis horas, que era o tempo de validade do cache.

AO FIM DESTE CAPÍTULO

Tornar a aplicação observável e rápida, nessa ordem: medir antes de otimizar, cachear com uma estratégia de invalidação escrita, e registrar logs estruturados que dá para procurar — sem nunca gravar o que não pode ser gravado.

**Capítulo disponível na
versão completa**

PARTE 7

Provar e publicar

Testes que dizem o que estão tentando provar, documentação que nasce do código, e o caminho até o dia do deploy sem plantão de madrugada.

NESTA PARTE

- 25 Testes: o que estamos tentando provar
- 26 Testes de feature, HTTP e banco
- 27 Documentação da API
- 28 Git, CI e o dia do deploy

CAPÍTULO

25

Testes: o que estamos tentando provar

A multa por devolução adiantada tinha sido corrigida em outubro. Em fevereiro, voltou por outro caminho. Havia correção; não havia teste.

AO FIM DESTE CAPÍTULO

Escrever testes rápidos sobre a regra de negócio, sem banco e sem HTTP: separar a decisão da consulta, cobrir as onze condições com uma tabela de casos, escolher o dublê certo, e transformar cada defeito corrigido num teste que impede a volta dele.

**Capítulo disponível na
versão completa**

CAPÍTULO

26

Testes de feature, HTTP e banco

A esteira ficou vermelha por três dias num teste que passava em toda máquina local. Faltava um ORDER BY — no teste e no código.

AO FIM DESTE CAPÍTULO

Verificar o contrato da API de ponta a ponta e as garantias que só o banco dá: requisições reais contra rotas reais, banco isolado a cada teste, permissões provadas pelo lado da recusa, e fakes de infraestrutura que não escondem o que deveriam testar.

**Capítulo disponível na
versão completa**

CAPÍTULO

27

Documentação da API

A integração do aplicativo levou três semanas. A única documentação da API era uma frase: “pergunta pro Dedé”.

AO FIM DESTES CAPÍTULO

Publicar uma documentação que nasce do código e por isso não desatualiza: um esquema OpenAPI gerado das rotas, dos Form Requests e dos resources, com os erros descritos, exemplos executáveis, e um plano escrito para tirar um campo do ar sem quebrar quem o usa.

**Capítulo disponível na
versão completa**

CAPÍTULO

28

Git, CI e o dia do deploy

No dia 31 de março, às sete da manhã, o papel dobrado com a senha do FTP saiu da pasta do projeto pela última vez.

AO FIM DESTE CAPÍTULO

Sair do “funciona na minha máquina”: um histórico que dá para ler, uma esteira que recusa o que não passa, um deploy em ordem com migration, cache e worker no lugar certo, uma verificação de saúde honesta — e, no fim, um requisito novo implementado de ponta a ponta, sozinho.

**Capítulo disponível na
versão completa**

PARTE 8

O sistema no ar

O dia seguinte ao deploy. Os pedidos que só aparecem quando gente de verdade usa o sistema: foto de celular, e-mail que sai assinado por quem não devia e fila que cresce mais rápido do que esvazia.

NESTA PARTE

- 29 Upload de arquivos
- 30 Mail e notificações
- 31 Queues na prática

CAPÍTULO

29

Upload de arquivos

A Vera fotografou trezentas capas com o celular novo. Metade não subiu. A outra metade subiu deitada.

AO FIM DESTES CAPÍTULOS

Receber arquivos grandes sem ser barrado por um limite que você não configurou, validar imagem pelo conteúdo e pelas dimensões, escolher entre disco privado e público, processar a imagem fora da requisição, trocar um arquivo sem deixar órfão, e testar tudo sem tocar no disco.

**Capítulo disponível na
versão completa**

CAPÍTULO

30

Mail e notificações

O primeiro e-mail da Casa Amarela foi assinado por um tal de Laravel, de hello@example.com. A Dona Marlene imprimiu e levou ao balcão como prova do golpe.

AO FIM DESTE CAPÍTULO

Separar e-mail que é documento de aviso que é notificação, configurar quem assina o que sai, levar o mesmo aviso por e-mail, WhatsApp e histórico no aplicativo com uma classe só, pôr cada canal na fila sem que a falha de um repita o outro, falar português, chegar à caixa de entrada e testar sem mandar nada.

**Capítulo disponível na
versão completa**

CAPÍTULO

31

Queues na prática

O aviso das nove chegou às três da tarde. Na frente dele, na mesma fila, estavam duzentos mil livros do Seu Juvenal.

AO FIM DESTE CAPÍTULO

Medir a fila pelo tempo de espera, separar trabalhos em filas com prioridade e workers próprios, importar em lote com progresso, encadear passos que dependem um do outro, respeitar o ritmo do provedor sem gastar tentativas, impedir trabalho em dobro e acertar o par timeout e `retry_after` que causa a maior parte das repetições.

**Capítulo disponível na
versão completa**

Encerramento

O calendário de farmácia continua na parede da Vertexo, com vinte e cinco sábados riscados e um circulado.

**Capítulo disponível na
versão completa**

Respostas

O que é HTTP

1. 201 — criou um recurso novo.
2. 404 — o recurso pedido não existe.
3. 422 — o pedido está bem formado e o conteúdo é inválido. Não é 400: o JSON estava correto, a regra é que não foi cumprida.
4. 415 — o formato do corpo não é aceito. Também não é 400, e a diferença importa para o cliente saber se deve corrigir o dado ou o cabeçalho.
5. 204 — feito, sem corpo. Devolver 200 com {} funciona e obriga o cliente a interpretar um corpo vazio de propósito.

O par que mais confunde é o 3 e o 4 contra o 400. Uma régua que resolve: 400 é “não consegui nem entender o pedido”; 415 é “entendi o formato e não aceito”; 422 é “entendi tudo e o conteúdo está errado”.

2

..

API.PHP	php
1	<?php
2	
3	declare(strict_types=1);
4	
5	header('Content-Type: application/json');
6	
7	\$emprestimos = [
8	4471 => ['exemplar_id' => 812, 'leitor_id' => 47],
9	];
10	
11	\$caminho = parse_url(\$_SERVER['REQUEST_URI'], PHP_URL_PATH);
12	
13	if (\$_SERVER['REQUEST_METHOD'] === 'GET'
14	&& preg_match('#^/emprestimos/(\d+)\$#', \$caminho,
15	\$partes)) {
16	\$id = (int) \$partes[1];
17	
18	if (!isset(\$emprestimos[\$id])) {
19	http_response_code(404);
20	echo json_encode(['erro' => "Empréstimo {\$id} não
21	existe"]);
22	exit;
23	}
24	echo json_encode(['id' => \$id] + \$emprestimos[\$id]);
25	exit;
26	}
27	
28	http_response_code(404);
29	echo json_encode(['erro' => 'Rota não encontrada']);

Três detalhes que decidem se isso funciona.

`parse_url(..., PHP_URL_PATH)` separa o caminho da consulta. Sem ele, `/emprestimos/4471?formato=curto` não casaria com a expressão, e o cliente receberia `404` por ter mandado um parâmetro a mais.

A expressão exige `\d+` e ancora nas duas pontas. Sem as âncoras, `/velhos/emprestimos/4471/extra` também casaria.

E o `(int)` na conversão do número: `$partes[1]` é texto, como tudo que vem da URL, e `$emprestimos` tem chaves inteiras. `isset($emprestimos['4471'])` com string funciona por conversão automática de chave — funciona por acidente, e acidente não é desenho.

3 Um: erro fatal com `display_errors` ligado. Um `TypeError` não tratado faz o PHP imprimir a mensagem e a pilha. Se o servidor estiver configurado para formatar isso em HTML, o cliente recebe uma página no meio do que devia ser JSON — muitas vezes **depois** de um pedaço do JSON já ter sido enviado.

Evita-se com `display_errors` desligado no servidor e um tratador de exceções registrado que responda em JSON com status `500`.

Dois: o servidor web respondendo antes do PHP. `404` de rota inexistente, `413` de corpo grande demais, `502` quando o PHP não respondeu — nenhuma dessas passa pelo seu código, e o padrão do servidor é uma página HTML.

Evita-se configurando as páginas de erro do servidor para o formato da API, e é o tipo de coisa que só aparece quando alguém testa o caminho ruim.

Três: saída antes do `header()`. Uma linha em branco depois do `?>` num arquivo incluído, um `echo` de depuração esquecido, um aviso impresso. O PHP manda os cabeçalhos padrão, que incluem `Content-Type: text/html`, e o seu `header()` chega tarde:

```
Warning: Cannot modify header information - headers already sent
```

Evita-se não fechando o `?>` em arquivo que só tem PHP — e é por isso que essa regra existe.

O que as três têm em comum vale mais que as três: **o caminho de erro da sua API precisa ser testado como o caminho de**

sucesso. Quase todo time testa o 201 e descobre o formato do 500 em produção.

Um framework de quarenta linhas

1

```
1      ['DELETE', '/livros/{id}',
2          function (string $id) use ($acervo): Resposta {
3              $id = (int) $id;
4
5              if (!isset($acervo[$id])) {
6                  return new Resposta(404, [
7                      'erro' => "Livro {id} não existe",
8                  ]);
9              }
10
11              return new Resposta(204, []);
12          }],
```

O `enviar()` precisa de ajuste porque, como está, ele sempre imprime o corpo — e `json_encode([])` produz `[]`, dois caracteres. Uma resposta 204 com dois bytes de corpo contraria o próprio status, e alguns clientes tratam isso como resposta malformada.

```
1  function enviar(Resposta $r): void
2  {
3      http_response_code($r->status);
4
5      if ($r->status === 204) {
6          return;
7      }
8
9      header('Content-Type: application/json');
10     echo json_encode($r->corpo, JSON_UNESCAPED_UNICODE);
11 }
```

Repare onde a correção mora: no envio, não na rota. Se cada rota precisasse lembrar de não imprimir nada, uma delas esqueceria — e é a mesma razão que justifica o front controller existir.

2

A troca é na linha que monta a expressão:

..

```
1  $regex = '#^' . preg_replace_callback(
2      '#\{(\w+)(?:\{([\^}]+\)}?)?\}#',
3      fn(array $p): string
4          => '(?<' . $p[1] . '>' . ($p[2] ?? '[^/]+') . ')',
5      $padrao,
6  ) . '$#';
```

O padrão agora reconhece duas formas: `{id}`, sem restrição, e `{id:\d+}`, com. O `preg_replace_callback` é necessário porque a subs-

tuição passou a depender do que foi encontrado — com o `preg_replace` simples não havia como escolher entre o formato informado e o padrão.

A rota vira:

```
['GET', '/livros/{id:\d+}', ...]
```

E `/livros/abc` não casa com nenhuma rota, caindo no 404 do fim do laço.

Vale reparar no que isso é: a primeira **validação** do mini framework, e ela acontece antes da função da rota existir. É por isso que todo framework oferece algo parecido — a alternativa é toda rota começar com um `if` que confere o formato do próprio endereço.

3

...

MINI.PHP

php

```
1  $medirTempo = function (array $req, callable $proximo):
Resposta {
2      $comeco = hrtime(true);
3
4      $resposta = $proximo($req);
5
6      $ms = (hrtime(true) - $comeco) / 1_000_000;
7      header(sprintf('X-Tempo: %.1fms', $ms));
8
9      return $resposta;
10 };
11
12 $exigirCredencial = function (array $req, callable $seg):
Resposta {
13     if ($req['metodo'] !== 'GET'
14         && !isset($_SERVER['HTTP_AUTHORIZATION'])) {
15         return new Resposta(401, ['erro' => 'Credencial
ausente']);
16     }
17
18     return $seg($req);
19 };
20
21 $acao = fn(array $req): Resposta => $roteador->despachar(
22     $req['metodo'],
23     $req['caminho'],
24 );
25
26 $pipeline = array_reduce(
27     array_reverse([$medirTempo, $exigirCredencial]),
28     fn(callable $proximo, callable $atual): callable
29     => fn(array $req): Resposta => $atual($req,
$proximo),
30     $acao,
```

```

31 );
32
33 enviar($pipeline([
34     'metodo' => $_SERVER['REQUEST_METHOD'],
35     'caminho' => parse_url($_SERVER['REQUEST_URI'],
PHP_URL_PATH),
36 ]));

```

A ordem é medição primeiro, credencial depois, e o motivo é o que cada uma faz com a resposta.

O middleware de tempo precisa ser a camada mais de fora para medir **tudo**, inclusive o tempo gasto recusando alguém. Se ele estivesse por dentro, as requisições rejeitadas com 401 sairiam sem X-Tempo — e são justamente as que você vai querer medir no dia em que alguém estiver martelando a API.

O de credencial precisa estar por fora da ação e por dentro da medição: ele interrompe, e interromper cedo é o ponto. Uma requisição sem credencial não deve chegar ao roteador, muito menos ao banco.

A régua geral, que serve para qualquer cadeia: **o que observa fica por fora; o que recusa fica logo em seguida; o que trabalha fica no meio.**

O primeiro projeto

- 1 O `.env` não veio junto — e não deveria vir. Sem ele, não há `APP_KEY`.

```

$ cp .env.example .env
$ php artisan key:generate

```

Versionar o `.env` “resolveria” o erro e criaria três problemas piores. A senha do banco de produção entraria no repositório, onde fica para sempre no histórico mesmo depois de removida. Todos os ambientes passariam a usar a mesma chave de criptografia. E cada pessoa da equipe sobrescreveria o endereço de banco das outras a cada `git pull`.

O `.env.example` existe exatamente para isso: ele leva as **chaves** sem os **valores**, e o erro acima é o lembrete de que falta um passo de dez segundos.

2

..

ROUTES/WEB.PHP

php

```

1  Route::get('/saude', function () {
2      try {
3          DB::connection()->getPdo();
4          $banco = 'ok';
5      } catch (\Throwable $e) {
6          $banco = 'falhou';
7      }
8
9      return response()->json([
10         'status' => $banco === 'ok' ? 'ok' : 'degradado',
11         'php' => PHP_VERSION,
12         'debug' => config('app.debug'),
13         'banco' => $banco,
14     ], $banco === 'ok' ? 200 : 503);
15 });

```

O cuidado pedido no enunciado está em duas decisões.

O catch não devolve a mensagem da exceção. O rastro de uma falha de conexão contém host, usuário e às vezes senha — e esta rota é pública. Quem monitora precisa saber que falhou; quem investiga olha o log.

O status muda junto. Devolver 200 com `"status":"degradado"` obriga o monitoramento a interpretar o corpo. O 503 é lido por qualquer ferramenta sem configuração nenhuma.

Uma decisão discutível, deixada de propósito: expor `PHP_VERSION` numa rota pública conta a um invasor qual versão atacar. Em produção, o comum é ou proteger a rota, ou devolver só `status`, e deixar o detalhe para uma rota interna.

3

...

Por que funcionou. 777 dá permissão de escrita a todo mundo, o que inclui o usuário que roda o PHP. O erro some porque o problema — o dono errado — deixou de importar.

O risco concreto. Qualquer processo do servidor passa a poder escrever em `storage/`. Numa hospedagem compartilhada, isso inclui os outros sites da máquina. E `storage/` não guarda só log: guarda sessões e as views Blade compiladas, que são **arquivos PHP que a aplicação executa**. Permissão de escrita ali é permissão de execução por tabela.

A correção. Dono certo e permissão de grupo:

```

$ sudo chown -R www-data:www-data storage bootstrap/cache
$ sudo chmod -R 775 storage bootstrap/cache

```

Como não repetir. O problema de fundo não é a permissão: é o deploy por cópia de pasta, que refaz o dono a cada vez e depende de alguém lembrar de corrigir. Duas saídas, em ordem de esforço.

A barata: pôr os dois comandos no passo de publicação, para que rodem sempre, sem depender de memória.

A certa: não copiar `storage/` no deploy. Ela guarda estado — log, sessão, upload — e estado não é parte do código. O caminho comum é mantê-la fora da pasta versionada e apontar para ela, de forma que o deploy troque só o código e não toque em nada que a aplicação escreveu.

Eloquent

1

```
SELECT COUNT(*) FROM livros WHERE ano >= ?
```

```
SELECT tombo FROM exemplares WHERE estado = ?
```

```
SELECT COUNT(*) FROM leitores;
SELECT * FROM leitores ORDER BY nome ASC LIMIT 20 OFFSET 0
```

```
SELECT * FROM emprestimos
WHERE devolvido_em IS NULL
ORDER BY created_at DESC LIMIT 1
```

Três observações que separam quem leu de quem entendeu.

O `pluck` traz **só a coluna pedida**, não a linha inteira — é a diferença entre trazer oito mil linhas completas e trazer oito mil números.

O `paginate` faz **duas** consultas: uma conta o total, para saber quantas páginas existem, e outra traz a página. É a causa mais comum de uma listagem lenta em tabela grande, e a razão de existir o `simplePaginate`, que dispensa a contagem.

E o `latest()` ordena por `created_at`, não por `id`. Num banco em que os registros foram importados fora de ordem, os dois resultados diferem.

2

..

APP/MODELS/LEITOR.PHP

php

```

1  class Leitor extends Model
2  {
3      protected $table = 'leitores';
4
5      protected $fillable = ['nome', 'documento', 'telefone'];
6
7      protected function casts(): array
8      {
9          return ['bloqueado_em' => 'immutable_datetime'];
10     }
11 }

```

O cadastro público não muda **uma linha** — e é esse o ponto. Ele já fazia `Leitor::create($dados)`, e agora os campos a mais são descartados.

A tela da Vera não usa atribuição em massa para o perfil. Ela atribui direto, o que ignora o `$fillable` de propósito:

```

$leitor->perfil = $request->enum('perfil', Perfil::class);
$leitor->save();

```

Isso parece contornar a proteção e é exatamente o desenho correto: o `$fillable` protege contra **o que vem de fora em bloco**. A atribuição direta é uma decisão escrita no código, num controller que só quem tem a permissão da Vera alcança.

`bloqueado_em` fica fora dos dois: ele muda por uma ação com nome — bloquear um leitor — e não por formulário.

3

...

Segurança. `$guarded = []` de novo, agora numa tabela que tem `multa_em_centavos` e `devolvido_em`. Um cliente poderia registrar um empréstimo já devolvido, sem multa.

Correção. Três defeitos, todos já vistos:

A diferença de datas em segundos dividida por 86.400 ignora fuso e os dias de mudança de horário, e devolve `float` com casas decimais.

O valor em `float` acumula erro a cada soma — a conta é em centavos.

E 0.8 está solto, divergindo de `config/biblioteca.php`, que existe desde o capítulo 6.

Desenho. Dois problemas, e o segundo é grave.

O cálculo não precisa de banco para existir. Ele é regra, e está num arquivo que só existe por causa de uma tabela — então testar

“meia multa para estudante” exige inserir empréstimo, leitor e exemplar.

E o método **envia e-mail**. Um método chamado `calcularMulta` que manda mensagem para o leitor é uma surpresa: qualquer relatório que percorra duzentos empréstimos para somar multas acabou de disparar duzentos e-mails.

Onde cada pedaço vai.

A conta de dias e o valor vão para um serviço — `CalculadoraDeMultas` — que recebe as datas e a configuração e devolve `Dinheiro`. Ele se testa sem banco.

A regra do estudante vai junto, porque é regra de negócio, e vira um caso de teste explícito.

O envio do aviso sai daqui inteiro. Ele é uma consequência de um acontecimento — a multa foi registrada —, não parte de calcular um número.

E o model fica com `$fillable`, os casts, os scopes e as relações. Umas quinze linhas.

Erros padronizados

1. 401, `nao-autenticado`. O aplicativo leva a pessoa ao login.
2. 409, com um tipo de domínio — algo como `renovacao-esgotada`. O pedido está certo; a regra não deixa.
3. 422, com `campos.leitor_id`. O tipo pode ser `validacao`.
4. 500 — ou 503, se a falha for detectada como indisponibilidade —, `falha-interna`, com `incidente`. Nenhuma menção a banco na mensagem.
5. 403, `sem-permissao`. Ela está autenticada; o papel dela não permite.

O item 4 é o que mais tenta vazar: “banco de dados indisponível” parece uma mensagem honesta e útil. Ela diz a quem está atacando que o ataque funcionou.

2

..

APP/EMPRESTIMOS/LEITORCOMPENDENCIA.PHP

php

```

1  final class LeitorComPendencia extends ExcecaoDeDominio
2  {
3      public function __construct(
4          public readonly int $leitorId,
5          public readonly Dinheiro $multa,
6      ) {
7          parent::__construct(sprintf(
8              'Há uma multa de %s em aberto.',
9              $multa->formatado(),
10             ));
11     }
12
13     public function tipo(): string
14     {
15         return 'leitor-com-pendencia';
16     }
17 }

```

Deve aparecer? Para o próprio leitor e para a equipe, sim: é a informação que permite resolver. O leitor quer saber quanto pagar.

O cuidado é que a mensagem vai para **quem fez a requisição**, e no painel da atendente quem fez a requisição é a atendente — que tem direito ao dado. No aplicativo, só o próprio leitor consegue pedir empréstimo em nome dele, então só ele vê.

Se algum dia existir uma rota em que um leitor aja em nome de outro — reserva para um dependente, por exemplo —, essa mensagem passa a vazar a dívida de uma pessoa para outra. Vale anotar o risco na classe, porque é o tipo de coisa que a pessoa que criar a rota nova não vai lembrar de conferir.

3

...

A favor, provavelmente. Localidade: quem lê o controller vê o que acontece em cada falha, sem abrir outro arquivo. Flexibilidade: uma rota específica pode querer uma resposta diferente para a mesma exceção.

Custo um: o formato diverge. Trinta controllers com `try/catch` são trinta lugares montando o JSON de erro, e em seis meses são quatro formatos diferentes. É o problema que o capítulo começou resolvendo.

Custo dois: o `default` `some`. O handler central garante que **toda** exceção não prevista vira um `500` seguro com incidente. O `try/catch` no controller trata as que o autor lembrou; a que ele não lembrou sobe sem tratamento — ou, pior, é capturada por um `catch (\Throwable $e)` que devolve a mensagem crua.

Custo três: o controller volta a crescer e a conhecer status. Cada método ganha dez linhas de `catch` que não são a responsabilidade dele.

Quando o `try/catch` no controller é certo: quando a exceção muda o **caminho** e não só a resposta. O painel Blade do capítulo 9 captura `ExemplarIndisponivel` para devolver a pessoa ao formulário com o erro no campo — ali, o comportamento é outro, não só o formato. E quando uma rota precisa tentar uma alternativa: se a capa não for encontrada no armazenamento, devolver a capa padrão.

A régua: o controller captura quando **faz algo diferente** com a falha. Quando só vai formatar, o handler já faz.

Glossário

Despachar Escolher qual código roda para uma requisição e chamá-lo. É a única coisa que um roteador faz, e é a peça que todo framework tem no meio. Tabela de rotas de um lado, requisição do outro, uma função no fim.

Superglobal Variável que existe em qualquer lugar do programa sem ser declarada nem recebida: `$_GET`, `$_POST`, `$_SERVER`, `$_FILES`, `$_COOKIE`. Elas são a forma mais antiga de o PHP entregar a requisição, e a mais frágil, porque qualquer linha de qualquer arquivo pode lê-las — e escrevê-las.

Conheça as demais obras dessa coleção



Java One

Do primeiro println à API REST em produção



PHP One

Volume 1 · Fundamentos do PHP

Visite a editora Pensar Bem →

pensarbem.com.br