PROJECTS

Website Rebuild with Astro

Bring-Your-Own-Backend, Made Honest

A Fifth Arrival at the Same Flexible URL Model

Drizzle ORM, Database-Backed Islands & Auth.js

No Free Admin, No Scaffold — Built Entirely by Hand

Capstone: The Website Rebuild Series, Complete

Philip Osztromok

Generated with Claude

Table of Contents

Website Rebuild with Astro — 12 Chapters

This is the fifth and final course in the Website Rebuild series, alongside the already-complete Next.js, Django, Laravel, and Rails rebuilds. It builds directly on the standalone `astro1` course — this chapter assumes everything through that course's own Chapter 10 is already familiar ground.

Scaffolding the Project

```
npm create astro@latest website-rebuild-with-astro
cd website-rebuild-with-astro
npm run dev
```

Server Output From Day One

```
// astro.config.mjs
import { defineConfig } from 'astro/config';
import node from '@astrojs/node';

export default defineConfig({
  output: 'server',
  adapter: node({ mode: 'standalone' }),
});
```

The standalone `astro1` course started static by default and only reached `output: 'server'` in its own Chapter 10, once a genuine reason showed up. This rebuild has that reason from the very first line — a real admin interface, exactly like every one of the four completed sibling courses needed — so server mode is configured immediately rather than migrated to later.

An Honest Starting Position: No Built-In ORM, No Built-In Auth

CLOSER TO NEXT.JS THAN TO DJANGO, LARAVEL, OR RAILS

Django ships its own ORM and its own `auth` app. Laravel ships Eloquent and its own `Auth` facade. Rails ships ActiveRecord and `has_secure_password`. A freshly scaffolded Astro project has **none** of that — no database layer, no authentication mechanism, nothing opinionated about either. This puts Astro in the same real category as Next.js, not the other three: both are JS/TS-ecosystem frameworks that assume the backend is entirely the developer's own choice.

This isn't a gap to apologize for — it's a genuinely different starting philosophy, and it shapes real decisions later in this course: Chapter 2 reaches for a real ORM (Drizzle, not reused from Next.js's own Prisma, to give this course its own technical texture) precisely because nothing comes built in, and Chapter 9 solves authentication by directly reusing the Next.js rebuild's own `bcryptjs` approach — a genuine, honest case of two siblings sharing a real solution because they share a real ecosystem.

Five Frameworks, Compared Honestly

	NEXT.JS	DJANGO	LARAVEL	RAILS	ASTRO
Built-in ORM?	No	Yes	Yes	Yes	No
Built-in auth?	No	Yes	Yes	Yes	No
Backend philosophy	Bring your own	Batteries included	Batteries included	Batteries included	Bring your own

Coming in Chapter 2

NOT YET CONFIGURED

This chapter deliberately stops before adding a database. `drizzle-orm` and `mysql2` are installed and the self-referencing `Page` table is designed starting in Chapter 2 — this chapter's own job is only to establish the project's real starting shape.

Hands-On Exercises

EXERCISE 1

Scaffold a new Astro project via npm create astro@latest, and confirm the default starter runs correctly in the dev server.

View solution

EXERCISE 2

Configure output: 'server' with the Node adapter in astro.config.mjs, and confirm the dev server still runs correctly in server mode.

View solution

EXERCISE 3

Inspect the freshly scaffolded project's package.json and confirm no ORM or authentication package is present anywhere in its dependencies — contrasting this with what a fresh Rails or Laravel project's own default Gemfile/composer.json already includes.

View solution

CHAPTER 1 QUICK REFERENCE

- `npm create astro@latest` — scaffolds the project
- `output: 'server'` + `@astrojs/node` — configured immediately, not migrated to later
- **No built-in ORM or auth** — genuinely closer to Next.js's own "bring your own backend" than to Django/Laravel/Rails
- **Not yet configured** — the database layer starts in Chapter 2
- **Next chapter:** Designing a Flexible URL & Content Model

Every sibling course reached the same design independently: a real `parent_id` foreign key plus a precomputed `full_path` string, kept in sync. This chapter reaches it a fifth time — with **Drizzle ORM**, a genuinely new TypeScript-first ORM not yet used anywhere on this site, deliberately not a second course built on Prisma.

The Self-Referencing Schema

```
// db/schema.ts
import { mysqlTable, int, varchar, text, AnyMySqlColumn } from 'drizzle-orm/mysql-core';
import { relations } from 'drizzle-orm';

export const pages = mysqlTable('pages', {
  id: int('id').primaryKey().autoincrement(),
  slug: varchar('slug', { length: 255 }).notNull(),
  fullPath: varchar('full_path', { length: 1024 }).notNull().unique(),
  title: varchar('title', { length: 255 }).notNull(),
  body: text('body'),
  parentId: int('parent_id').references(
    (): AnyMySqlColumn => pages.id,
    { onDelete: 'restrict' }
  ),
});

export const pagesRelations = relations(pages, ({ one, many }) => ({
  parent: one(pages, {
    fields: [pages.parentId],
    references: [pages.id],
    relationName: 'parentChild',
  }),
  children: many(pages, { relationName: 'parentChild' }),
}));
```

A REAL TYPESCRIPT WRINKLE, UNIQUE TO A SCHEMA-AS-REAL-CODE ORM

`parentId`'s own `references()` call needs a *function* returning `pages.id`, not a direct reference — because `pages` can't be referenced from inside its own definition while TypeScript is still evaluating it. Prisma's own `schema.prisma` never hits this, since it's a separate declarative DSL file, not real TypeScript being type-checked as it's written. This is a genuine, small cost of Drizzle's own "the schema is real code" design.

Verifying `onDelete: 'restrict'` — Which Layer, Confirmed

MATCHES LARAVEL'S LAYER, NOT RAILS' OR DJANGO'S

Drizzle's `references(..., { onDelete: 'restrict' })` generates a genuine SQL `FOREIGN KEY ... ON DELETE RESTRICT` clause in the migration itself — a real, database-enforced constraint, exactly like Laravel's own `restrictOnDelete()`. This is a different layer entirely from Rails' own `dependent: :restrict_with_error` (an application-level ActiveRecord callback) or Django's own `PROTECT` (an ORM-level check before deletion) — a raw SQL `DELETE` bypassing Drizzle entirely would still be refused here, the same guarantee Laravel's own database constraint provides.

No Lifecycle Hooks — A Genuine, Significant Difference

```ts
// db/pages.ts
import { db } from './client';
import { pages } from './schema';
import { eq } from 'drizzle-orm';

async function computeFullPath(slug: string, parentId: number | null): Promise<string> {
  if (!parentId) return slug;
  const [parent] = await db.select().from(pages).where(eq(pages.id, parentId));
  return `${parent.fullPath}/${slug}`;
}

export async function createPage(slug: string, title: string, parentId: number | null) {
  const fullPath = await computeFullPath(slug, parentId);
  return db.insert(pages).values({ slug, title, fullPath, parentId });
}
```

NOT A GAP — A DELIBERATE DESIGN CHOICE, WORTH NAMING HONESTLY

Every ORM used across this series so far — ActiveRecord, Eloquent, Django's own ORM, even Prisma via its own middleware — offers some form of lifecycle hook or callback for exactly this "recompute a derived field whenever a row is saved" job. Drizzle offers **none** — it's deliberately a thin, close-to-SQL query builder, not a framework with model-level magic. `computeFullPath` has to be called explicitly by every function that creates or updates a page — there's no `before_save`-equivalent silently doing it automatically. This is a real, significant architectural difference, not a smaller version of the same idea.

Five ORMs, Compared Honestly

	DJANGO	LARAVEL	RAILS	ASTRO (DRIZZLE)
Delete-protection layer	Application-level (`PROTECT`)	Database-level (`restrictOnDelete()`)	Application-level (`dependent: :restrict_with_error`)	Database-level (`onDelete: 'restrict'`)
Lifecycle hooks?	Yes — signals	Yes — model events	Yes — callbacks (`before_save`)	No — explicit helper functions only

Migrations

```
npx drizzle-kit generate
npx drizzle-kit migrate
```

`drizzle-kit generate` diffs `schema.ts` against the current migration history and writes a new SQL migration file; `drizzle-kit migrate` applies it.

Hands-On Exercises

EXERCISE 1

Define the self-referencing pages table in Drizzle, including the function-based workaround needed for parentId's own circular reference to pages.id.

View solution

EXERCISE 2

Write and test computeFullPath(), confirming it correctly builds a nested path for a page with a real parent, and returns the bare slug for a page with no parent.

View solution

EXERCISE 3

Confirm onDelete: 'restrict' is a real database-level constraint by attempting a raw SQL DELETE (bypassing Drizzle entirely) against a page that still has children, and observing the database itself refuse it.

View solution

CHAPTER 2 QUICK REFERENCE

- `references(():AnyMySqlColumn => pages.id, ...)` — the function-based workaround for a self-referencing foreign key
- `onDelete: 'restrict'` — a real database-level constraint, matching Laravel's own layer
- **No lifecycle hooks** — Drizzle's own deliberate, minimal design; `full_path` is computed by an explicit helper, not a callback
- `drizzle-kit generate` / `migrate` — Drizzle's own migration workflow
- **Next chapter:** Routing — Astro's Own Catch-All, Now Database-Backed

`astro1`'s own Chapter 3 built `[...path].astro` against Content Collections, and Chapter 10 showed `getStaticPaths()` disappearing entirely in server mode. This chapter is where both findings actually pay off together — the catch-all route, hooked to a real database query, with no pre-declared path list anywhere.

The Catch-All Route, Database-Backed

```
// src/pages/[...path].astro
---
import { db } from '../db/client';
import { pages } from '../db/schema';
import { eq } from 'drizzle-orm';

const { path } = Astro.params;
const fullPath = path ?? '';

const [page] = await db.select().from(pages).where(eq(pages.fullPath, fullPath));
---
<h1>{page.title}</h1>
```

No `getStaticPaths()` anywhere in this file — Chapter 1's own server-mode configuration means every possible `path` value resolves fresh, per request, straight from the `pages` table.

> **GENUINE CONVERGENCE — THE PAYOFF OF THE WHOLE SERIES' OWN THROUGHLINE**
>
> While `astro1`'s own catch-all stayed backed by Content Collections, its final shape looked structurally different from every sibling rebuild's own routing. Here, with a real database query behind it, Astro's routing finally looks the same shape as all four completed siblings: one catch-all route, one lookup by path, resolved per request. The differences that remain are in the ORM and syntax, not in the underlying architecture — a genuine convergence, not a coincidence.

A Real Gotcha: Your Own Catch-All Shadows the Conventional 404 Page

ASTRO'S OWN 404.ASTRO NEVER GETS REACHED

Astro's usual convention is that a `src/pages/404.astro` file is served automatically for any unmatched route. But `[...path].astro` matches *every possible URL*, including genuinely nonexistent ones — so it always wins the routing match first, and `404.astro` is never reached through normal routing at all. A missing page simply falls through this file's own `if (!page)` branch instead — that has to be handled explicitly, not left to Astro's own file-based convention.

Handling Not-Found Correctly

```
// src/pages/[...path].astro
---
const { path } = Astro.params;
const fullPath = path ?? '';

const [page] = await db.select().from(pages).where(eq(pages.fullPath, fullPath));

if (!page) {
  Astro.response.status = 404;
  return Astro.rewrite('/404');
}
---
```

`Astro.rewrite('/404')` renders the real content of `src/pages/404.astro` for this request, while `Astro.response.status = 404` ensures the response actually carries a genuine `404` status code — both are needed together; `rewrite` alone would render the right content with a misleading `200` status.

Routing, Now Genuinely Converged

	NEXT.JS	DJANGO	LARAVEL	RAILS	ASTRO
Mechanism	`[...path]` folder	`<path:full_path>`	`{path?}` + regex	`*path` glob	`[...path].astro`
Resolved by	A database lookup	A database lookup	A database lookup	A database lookup	A database lookup

Hands-On Exercises

EXERCISE 1

Build [...path].astro with a real Drizzle query resolving Astro.params.path against the pages table, and confirm a real stored page renders correctly at its own full path.

View solution

EXERCISE 2

Visit a genuinely nonexistent path with no explicit not-found handling in place yet, and observe exactly what happens — confirming src/pages/404.astro is never actually reached.

View solution

EXERCISE 3

Add the explicit Astro.response.status = 404 plus Astro.rewrite('/404') handling, and confirm the response now carries both a real 404 status code and the correct 404 page content together.

View solution

CHAPTER 3 QUICK REFERENCE

- `[...path].astro` — no `getStaticPaths()` needed; resolves fresh, per request
- **Genuine convergence** — Astro's routing finally matches the same shape as all four sibling rebuilds
- **The 404 gotcha** — a global catch-all shadows `src/pages/404.astro`; it's never reached via normal routing
- `Astro.response.status = 404` + `Astro.rewrite('/404')` — the correct combination for a real 404, both status and content
- **Next chapter:** Views & Rendering Trusted Content

This chapter builds the real layout Chapter 3's own database-backed route uses — and, like every sibling course before it, needs a way to render stored HTML content as real markup, not escaped text.

The Layout

```
// src/layouts/PageLayout.astro
---
import { db } from '../db/client';
import { pages } from '../db/schema';
import { eq } from 'drizzle-orm';

interface Props {
  page: typeof pages.$inferSelect;
}

const { page } = Astro.props;

async function getBreadcrumb(pageId: number) {
  const crumbs = [];
  let currentId: number | null = pageId;
  while (currentId) {
    const [current] = await db.select().from(pages).where(eq(pages.id, currentId));
    if (!current) break;
    crumbs.unshift(current);
    currentId = current.parentId;
  }
  return crumbs;
}

const breadcrumb = await getBreadcrumb(page.id);
---
<html lang="en">
<head><title>{page.title}</title></head>
<body>
<nav>
      {breadcrumb.map((crumb) => <a href={`/${crumb.fullPath}`}>{crumb.title}</a>)}
    </nav>
<h1>{page.title}</h1>
<div set:html={page.body} />
</body>
</html>
```

THE SAME N+1 PATTERN, PLANTED DELIBERATELY A SIXTH TIME

`getBreadcrumb` walks `current.parentId` one row at a time inside a `while` loop — one database query per ancestor level, exactly the pattern every sibling rebuild course (and the standalone `astro1` course itself) built into its own breadcrumb code on purpose. It's left as written here, to be caught and fixed in Chapter 6.

`set:html`: Astro's Own Answer

THE SAME JOB, FIVE DIFFERENT SYNTAXES

`set:html={page.body}` sets an element's HTML content directly, bypassing Astro's own default auto-escaping — the exact same job Django's `|safe` filter, Laravel's `{!! !!}`, and Rails' `<%== %>` each do in their own frameworks. Astro's version is a template *directive* on the element itself, rather than a filter or a special output-tag syntax — its own genuine syntactic shape, doing an identical job to every sibling.

ONLY FOR CONTENT THE ADMIN CONTROLS

`set:html` is only safe for content this project's own admin interface writes — never for rendering arbitrary user input directly, since it bypasses the exact escaping that protects against injected markup. The same warning every sibling course's own equivalent mechanism carries.

Compared Across the Series

FRAMEWORK	MECHANISM
Django	`{{ page.body|safe }}` — a template filter
Laravel	`{!! $page->body !!}` — a Blade output-tag variant
Rails	`<%== @page.body %>` — an ERB output-tag variant
Astro	`<div set:html={page.body} />` — a template directive on the element

Hands-On Exercises

EXERCISE 1

Build PageLayout.astro rendering a page's title and body via set:html, and confirm real stored HTML (e.g. a tag inside body) renders as actual formatted HTML, not escaped text.

View solution

EXERCISE 2

Build getBreadcrumb() and confirm it produces the correct, correctly-ordered ancestor chain for a real page stored at least three levels deep.

View solution

EXERCISE 3

Temporarily replace set:html={page.body} with plain {page.body} interpolation, and confirm Astro's default behavior actually escapes the stored HTML — the raw tags appear as visible text instead of rendering.

View solution

CHAPTER 4 QUICK REFERENCE

- `set:html={page.body}` — Astro's own directive-based equivalent of `|safe` / `{!! !!}` / `<%== %>`
- **Only for admin-controlled content** — never for arbitrary user input
- `getBreadcrumb` — a deliberately planted N+1 pattern, caught in Chapter 6
- **Plain `{ }` interpolation escapes by default** — `set:html` is a genuine, deliberate opt-out
- **Next chapter:** Styling — Dark Theme

Nothing new to introduce here — this chapter applies `astro1`'s own Chapter 6 material directly: a global stylesheet for the site's own established dark theme, plus a scoped style block for the breadcrumb nav.

The Global Stylesheet

```
/* src/styles/global.css */
body {
  background: #0f1117;
  color: #c9d1d9;
  font-family: system-ui, sans-serif;
  margin: 0;
}
```

```
// src/layouts/PageLayout.astro
---
import '../styles/global.css';
// ...rest unchanged from Chapter 4
---
```

Imported once, in the layout every page already routes through — the same pattern `astro1` Chapter 6 established.

A Scoped Style for the Breadcrumb

```
<nav class="breadcrumb">
  {breadcrumb.map((crumb) => <a href={`/${crumb.fullPath}`}>{crumb.title}</a>)}
</nav>
<style define:vars={{ accent: '#FF5D01' }}>
  .breadcrumb {
    display: flex;
    gap: 0.5rem;
    padding: 0.75rem 0;
  }
  .breadcrumb a {
    color: #8b949e;
    text-decoration: none;
  }
  .breadcrumb a:hover {
    color: var(--accent);
  }
</style>
```

Scoped automatically, with no risk of colliding with any other `nav` element elsewhere on the site — and `define:vars` binds the frontmatter's own accent color directly into the hover state, exactly as `astro1` Chapter 6 demonstrated.

NOTHING NEW — BY DESIGN

This chapter is deliberately light. Every technique here — scoped styles, `define:vars`, a plain imported global stylesheet — was already covered in full in `astro1`'s own Chapter 6. Applying already-known material to a real project is the point, not re-deriving it.

Hands-On Exercises

EXERCISE 1

Add global.css with the dark theme applied via PageLayout's own frontmatter import, and confirm it applies site-wide across every route.

View solution

EXERCISE 2

Add a second, unrelated component with its own nav element styled differently, and confirm the breadcrumb's own scoped styles never leak into it.

View solution

EXERCISE 3

Use define:vars to bind the accent color into the breadcrumb's own hover state, and confirm changing the frontmatter's own accent value changes the rendered hover color.

View solution

CHAPTER 5 QUICK REFERENCE

- `global.css` **imported in the layout** — applies the dark theme site-wide
- **Scoped styles, automatic** — the breadcrumb's own rules never collide with anything else
- `define:vars` — binds a frontmatter value into a real CSS custom property
- **No new material** — everything here is `astro1` Chapter 6, applied directly
- **Next chapter:** The Database: Drizzle ORM

Chapter 4's own `getBreadcrumb` was planted deliberately — one database query per ancestor level. This chapter catches it, resolved via Drizzle's own relational query API.

Enabling Relational Queries

```
// db/client.ts
import { drizzle } from 'drizzle-orm/mysql2';
import mysql from 'mysql2/promise';
import * as schema from './schema';

const connection = await mysql.createConnection(process.env.DATABASE_URL);
export const db = drizzle(connection, { schema });
```

Passing the full `schema` module — including Chapter 2's own `pagesRelations` — into `drizzle()` is what unlocks `db.query.pages`'s own relational API, distinct from the plain `db.select()` query builder used so far.

Fixing the Breadcrumb

```
const pageWithAncestors = await db.query.pages.findFirst({
  where: eq(pages.id, pageId),
  with: {
    parent: {
      with: {
        parent: {
          with: {
            parent: {
              with: { parent: true }
            }
          }
        }
      }
    }
  }
});
```

`with` eagerly loads the parent chain in place of Chapter 4's own repeated single-row query loop, resolved in a small, bounded number of queries rather than one per level.

Verified Honestly Against Prisma

NOT A COINCIDENCE — A DOCUMENTED DESIGN GOAL

Drizzle's own relational query API — the `with` keyword, the nested-object shape — deliberately mirrors Prisma's own `include`, both in naming and in structure. This is a real, stated design choice from Drizzle's own documentation: offer a developer experience close to Prisma's while keeping the underlying query builder closer to raw SQL. The similarity between `with: { parent: true }` here and Prisma's own `include: { parent: true }` from the Next.js rebuild's own database chapter is genuine kinship, not an accident.

The Same Shared Limitation, a Fifth Time

FIXED DEPTH, EXACTLY LIKE EVERY SIBLING ORM BEFORE IT

The nested `with` call above only reaches four levels deep — no more. This is the identical limitation already established across every ORM in this series: Prisma's own nested `include`, Django's chained `select_related`, Eloquent's dotted `with()`, and ActiveRecord's nested `includes()` all share this exact same fixed-depth requirement for a self-referencing, arbitrary-depth tree. None of the five ORMs used across this entire Website Rebuild series can express "load the whole ancestor chain, however deep it is" without a raw recursive SQL query — a real, consistent, honestly-repeated finding, not a Drizzle-specific shortcoming.

Five ORMs' Eager Loading, Compared

FRAMEWORK	SYNTAX
Next.js (Prisma)	`include: { parent: { include: { parent: true } } }`
Django	`select_related('parent__parent__parent')`
Laravel (Eloquent)	`with('parent.parent.parent')`
Rails (ActiveRecord)	`includes(parent: { parent: :parent })`
Astro (Drizzle)	`with: { parent: { with: { parent: true } } }`

Hands-On Exercises

EXERCISE 1

Configure db/client.ts to pass the full schema (including pagesRelations) into drizzle(), enabling db.query.pages's own relational API.

View solution

EXERCISE 2

Replace PageLayout's own getBreadcrumb loop with a db.query.pages.findFirst({ with: ... }) call, and confirm the same correct ancestor chain still renders.

View solution

EXERCISE 3

Store a page six levels deep and confirm the nested with clause from this chapter (only four levels) fails to load the full ancestor chain — demonstrating the shared fixed-depth limitation directly.

View solution

CHAPTER 6 QUICK REFERENCE

- `drizzle(connection, { schema })` — enables `db.query`'s own relational API
- `db.query.pages.findFirst({ with: ... })` — replaces the deliberately-planted N+1 loop
- **Verified kinship** — Drizzle's `with` deliberately mirrors Prisma's own `include`, a documented design goal
- **Shared limitation, a fifth time** — every ORM in this series needs a fixed-depth nested call; none expresses unbounded ancestor depth natively
- **Next chapter:** Rendering Content & the Kanji Edge Case

Two of this chapter's three usual questions are already answered by earlier chapters, unchanged. The third has a real, small, honest wrinkle worth covering.

Routing & Rendering: Already Solved

Kanji's two original constraints — a fixed-depth hierarchy, and no self-contained-fragment convention — never applied to this course's own design in the first place. Chapter 2's arbitrary-depth Drizzle schema has no special-cased depth limit, and Chapter 4's `set:html={page.body}` renders a kanji page's own inline markup through the exact same path as any other page. Nothing new to solve here, matching every sibling course's own resolution.

String Safety: Genuinely Already Solved Too

THE IDENTICAL RESOLUTION AS NEXT.JS, BECAUSE IT'S THE IDENTICAL RUNTIME

Astro runs on Node.js — the exact same JavaScript engine, and the exact same UTF-16 string model, that the Next.js rebuild already worked with. Whatever string-slicing safety conclusion that course's own Chapter 7 reached applies here completely unchanged, since this isn't two frameworks reaching a similar answer independently — it's the literal same language and runtime being reused a second time within this series.

A Real, Small, Honest Difference: `mysql2`'s Own Charset

```
// db/client.ts
const connection = await mysql.createConnection({
  uri: process.env.DATABASE_URL,
  charset: 'utf8mb4',
});
```

NOT AUTOMATIC THE WAY RAILS' OWN GENERATOR WAS

Rails Rebuild's own Chapter 7 found a genuinely positive default: `rails new --database=mysql` has configured `utf8mb4` automatically since Rails 5.2, with nothing to set manually. The Node `mysql2` driver Drizzle uses has no equivalent automatic default — `charset: 'utf8mb4'` has to be set explicitly in the connection config, or the connection falls back to a less complete charset that risks the same historical 3-byte truncation issue Rails' own chapter described. A small, real, honest difference — not every framework in this series reached the same quiet win Rails did.

Kanji Across the Series

	NEXT.JS	DJANGO	LARAVEL	RAILS	ASTRO
String-slicing risk	None — UTF-16, same as Astro	None — Python 3 strings	Real — `mb_substr()` needed	None — Ruby strings	None — same runtime as Next.js
`utf8mb4` handling	Manual (Prisma config)	Manual (Django DB config)	Manual (migration charset)	Automatic since Rails 5.2	Manual — `mysql2` connection option

Hands-On Exercises

EXERCISE 1

Confirm the two original kanji constraints are already resolved by Chapters 2 and 4, without writing any new code for this chapter.

View solution

EXERCISE 2

In a Node REPL, confirm '水のページ'.length and slicing behavior are safe for typical kanji content — the same JavaScript string model the Next.js rebuild already relied on.

View solution

EXERCISE 3

Omit charset: 'utf8mb4' from the mysql2 connection config, and confirm the connection does NOT default to it automatically the way Rails' own generator did — a real, observable difference between the two frameworks.

View solution

CHAPTER 7 QUICK REFERENCE

- **Routing/rendering** — already solved by Chapters 2 and 4, no new work
- **String safety** — identical to Next.js, since both share the same Node.js/V8 runtime
- `charset: 'utf8mb4'` — has to be set explicitly on the `mysql2` connection; no automatic default the way Rails 5.2+ has
- **Not every sibling reached the same quiet win** — a real, honest difference from Rails' own Chapter 7
- **Next chapter:** Dynamic Content & Forms

Django needed a separate `ModelForm` class. Laravel needed a separate `FormRequest` class. Rails needed Strong Parameters, at least filtering which fields are allowed. Astro needs none of that structurally — but the reason is worth being honest about.

The Endpoint, With No Validation at All

```ts
// src/pages/api/pages/[id]/title.ts
import type { APIRoute } from 'astro';
import { db } from '../../../../db/client';
import { pages } from '../../../../db/schema';
import { eq } from 'drizzle-orm';

export const POST: APIRoute = async ({ params, request }) => {
  const { title } = await request.json();

  await db.update(pages).set({ title }).where(eq(pages.id, Number(params.id)));

  return new Response(JSON.stringify({ status: 'ok' }));
};
```

THE LEANEST ANSWER, FOR A REAL REASON

Nothing here checks that `title` is even a string, that it exists at all, or that it's a reasonable length. A request body of `{ "title": 12345 }` or `{}` is happily passed straight into a real database write. This isn't a smarter, more minimal design than Rails or Laravel's own answers — it's simply what's left when a framework provides no validation layer at all.

Closing the Gap: Zod, Deliberately

```
import { z } from 'zod';

const titleSchema = z.object({
  title: z.string().min(1).max(255),
});

export const POST: APIRoute = async ({ params, request }) => {
  const body = await request.json();
  const result = titleSchema.safeParse(body);

  if (!result.success) {
    return new Response(JSON.stringify({ error: result.error.flatten() }), { status: 400
});
  }

  await db.update(pages).set({ title: result.data.title }).where(eq(pages.id,
Number(params.id)));

  return new Response(JSON.stringify({ status: 'ok' }));
};
```

The same Zod library already used for Content Collections in `astro1` Chapter 5 closes this gap. `safeParse` returns a typed, validated result rather than throwing — the same discipline Zod's own schema already demonstrated, applied here to a plain API route instead of stored content.

The Deliberate No-Auth-Check-Yet Gap — the Bare Minimum Version

Matching every sibling course's own Chapter 8, this endpoint has no authentication check yet, to be closed in Chapter 9. Astro's own version of the gap is genuinely the most minimal of all five: Rails left a missing `before_action` filter, Laravel left a visible `authorize()` placeholder — Astro has no conventional place for either concept to even go yet, since there's no framework-level auth mechanism at all.

A Real, Unaddressed Limitation: No CSRF Protection

NOT SOLVED BY THIS COURSE, NAMED HONESTLY INSTEAD

Rails ships `protect_from_forgery`. Laravel ships `VerifyCsrfToken` middleware. Django ships its own CSRF middleware. Astro ships **none** of that — a request from any origin can hit this endpoint with no built-in same-origin or token check standing in the way. A production version of this project would need to add that protection by hand (a same-origin check on the request's own `Origin` header, or a real CSRF token scheme). This course names the gap honestly rather than building a full solution for it, matching the same "honest scope note" convention every capstone in this series already uses.

Five Structural Answers, Compared

	NEXT.JS	DJANGO	LARAVEL	RAILS	ASTRO
Validation layer	Manual (Zod, by choice)	`ModelForm`	`FormRequest`	Strong Parameters	Manual (Zod, by necessity)
CSRF protection built in?	No	Yes	Yes	Yes	No

Hands-On Exercises

EXERCISE 1

Build the /api/pages/[id]/title.ts endpoint with no validation, then send a malformed request body (e.g. a numeric title) and confirm it's written to the database completely unchecked.

View solution

EXERCISE 2

Add the Zod schema and safeParse validation, and confirm the same malformed request is now rejected with a 400 response instead of reaching the database.

View solution

EXERCISE 3

In a local dev environment, confirm no CSRF protection exists by successfully calling this endpoint from a fetch request issued from a different origin/port, with no same-origin check blocking it.

View solution

CHAPTER 8 QUICK REFERENCE

- **The leanest endpoint of all five siblings** — no separate validation class, but only because none is provided by default
- **Zod's** `safeParse` — the deliberate fix, reused from `astro1` Chapter 5's own Content Collections
- **The gap looks different here** — no conventional place for an auth check to even go yet, closed in Chapter 9
- **No CSRF protection at all** — a real, unaddressed limitation, named honestly rather than solved
- **Next chapter:** Admin Authentication

Astro has no framework-level auth mechanism at all — Chapter 1's own finding. This chapter doesn't invent a new solution from scratch; it reuses one that already exists in this exact series.

Not Just the Same Technique — the Same Library

AUTH.JS HAS AN OFFICIAL ASTRO INTEGRATION

The Next.js rebuild's own Chapter 9 used NextAuth.js's Credentials provider with a `bcryptjs` comparison inside it. NextAuth.js was rebranded **Auth.js** and now ships official integrations for multiple frameworks — including `@auth/astro`. This isn't a case of two frameworks independently reaching for a similar solution; it's the literal same underlying authentication library, reused here through its own dedicated Astro package, because both frameworks share the same Node.js/npm ecosystem.

Setting Up `@auth/astro`

```
npm install @auth/astro @auth/core bcryptjs
```

```
// auth.config.ts
import { defineConfig } from 'auth-astro';
import Credentials from '@auth/core/providers/credentials';
import bcrypt from 'bcryptjs';
import { db } from './db/client';
import { adminUsers } from './db/schema';
import { eq } from 'drizzle-orm';

export default defineConfig({
  providers: [
    Credentials({
      credentials: {
        email: { label: 'Email' },
        password: { label: 'Password', type: 'password' },
      },
      async authorize(credentials) {
        const [user] = await db.select().from(adminUsers).where(eq(adminUsers.email,
credentials.email));
        if (!user) return null;

        const valid = await bcrypt.compare(credentials.password, user.passwordHash);
        return valid ? { id: user.id, email: user.email } : null;
      },
    }),
  ],
});
```

`authorize()` is Auth.js's own callback shape — identical to the one the Next.js rebuild's own Chapter 9 already wrote — receiving submitted credentials and returning either a real user object or `null`.

Verifying the Legacy Hash

The legacy site's own admin credential is a PHP-generated bcrypt hash, tagged `$2y$`. `bcryptjs` — the exact same package the Next.js rebuild already used — verifies it correctly with no special configuration, the same finding already confirmed twice in this series: once directly (Next.js), once as a cross-ecosystem nuance worth double-checking (Rails' own Ruby `bcrypt` gem). Here, it isn't even a cross-ecosystem question — it's literally the same JavaScript library, doing the same job, unchanged.

Closing Chapter 8's Gap

```
// src/pages/api/pages/[id]/title.ts
import { getSession } from 'auth-astro/server';

export const POST: APIRoute = async ({ params, request }) => {
  const session = await getSession(request);
  if (!session) {
    return new Response(null, { status: 401 });
  }

  // ...rest unchanged from Chapter 8
};
```

`getSession(request)` is exactly the piece Chapter 8's own gap was waiting for — the first genuine "place for an auth check to go" this course has had.

Five Frameworks' Own Auth Stories

	NEXT.JS	DJANGO	LARAVEL	RAILS	ASTRO
Mechanism	NextAuth.js / Auth.js	Built-in `auth` app	Built-in `Auth` facade	`has_secure_password`	`@auth/astro` — the same Auth.js library
Bcrypt match?	Yes, via `bcryptjs`	No — needed reconfiguration	Yes, zero config	Yes, with a real cross-ecosystem tag nuance	Yes — the identical library and result as Next.js

Hands-On Exercises

EXERCISE 1

Set up @auth/astro with a Credentials provider verifying against the legacy admin's stored bcrypt hash, and confirm a correct login succeeds while an incorrect password fails.

View solution

EXERCISE 2

Add the getSession() check to the title-update endpoint from Chapter 8, and confirm an unauthenticated request is now rejected with a 401 status.

View solution

EXERCISE 3

Confirm bcryptjs correctly verifies the legacy $2y$-tagged PHP hash with no configuration change, and explain why this isn't even a cross-ecosystem question here, unlike the Rails rebuild's own Chapter 9.

View solution

CHAPTER 9 QUICK REFERENCE

- `@auth/astro` — Auth.js's own official Astro integration, the literal same library as Next.js's own NextAuth.js
- `authorize()` — the identical Credentials-provider callback shape the Next.js rebuild already wrote
- `bcryptjs` — verifies the legacy `$2y$`-tagged PHP hash with no configuration, the same package the Next.js rebuild used
- `getSession(request)` — closes Chapter 8's own gap, the first real place for an auth check in this course
- **Next chapter:** Admin CRUD Interface

Django has a free admin. Laravel and Rails don't, but each has some code-generation help — Laravel's `make:model -mcr`, Rails' own `rails generate scaffold`. Astro has neither. This is genuinely the most minimal starting point of all five siblings.

The Parent Picker

The same searchable flat list every sibling course reached for — every other page's own `fullPath`, filterable client-side, rather than a nested tree widget, for the same scale-justified reason established since the Next.js rebuild's own Chapter 10.

Paying Off Chapter 2's Deferred Cost

```
// lib/pages.ts
export async function movePage(pageId: number, newParentId: number | null) {
  const [page] = await db.select().from(pages).where(eq(pages.id, pageId));
  const newFullPath = await computeFullPath(page.slug, newParentId);

  await db.update(pages)
    .set({ parentId: newParentId, fullPath: newFullPath })
    .where(eq(pages.id, pageId));

  await updateDescendantPaths(pageId);
}

async function updateDescendantPaths(pageId: number) {
  const [parent] = await db.select().from(pages).where(eq(pages.id, pageId));
  const children = await db.select().from(pages).where(eq(pages.parentId, pageId));

  for (const child of children) {
    const newPath = `${parent.fullPath}/${child.slug}`;
    await db.update(pages).set({ fullPath: newPath }).where(eq(pages.id, child.id));
    await updateDescendantPaths(child.id);
  }
}
```

Chapter 2 already established that Drizzle has no lifecycle hooks — so `updateDescendantPaths` is a fully explicit recursive walk, no different in spirit from Laravel's own `updateDescendantPaths()` or Rails' own `update_descendant_paths!`, just written without any framework callback machinery underneath it at all.

Delete Refusal — a Precise Callback to Two Earlier Findings

```
// src/pages/api/pages/[id].ts
export const DELETE: APIRoute = async ({ params }) => {
  try {
    await db.delete(pages).where(eq(pages.id, Number(params.id)));
    return new Response(JSON.stringify({ status: 'ok' }));
  } catch (err) {
    return new Response(
      JSON.stringify({ error: "Can't delete a page that still has children — move or delete them first." }),
      { status: 409 }
    );
  }
};
```

THE SAME LAYER AS LARAVEL — NEEDING THE SAME CEREMONY

Chapter 2 verified `onDelete: 'restrict'` as a genuine database-level constraint, the same layer Laravel's own `restrictOnDelete()` operates at. That means deleting a page with children here raises a real, raw database exception — requiring the same `try` / `catch` ceremony Laravel's own Chapter 10 needed, not the smoother built-in error-population Rails found for itself.

A PRECISE CORRECTION, NOT A BROAD RULE

It would be tempting to conclude "database-level protection needs a try/catch, application-level protection doesn't" — but that's not quite right either. Django's own `PROTECT` is application-level, yet it still raises a real `ProtectedError` exception that needs its own `try` / `except`. Rails' own smoother, non-exception path — `destroy` simply returning `false` with `errors` already populated — was a specific design choice in ActiveRecord's own API, not a trait every application-level protection shares. Astro needs a `try` / `catch` for the same reason Laravel *and* Django both effectively do, each for their own reason.

Five Admin Interfaces, Compared

	DJANGO	LARAVEL	RAILS	ASTRO
Free admin?	Yes — live, runtime-introspected	No	No	No
Scaffold generator?	N/A — has the admin instead	`make:model -mcr`	`rails generate scaffold`	None at all
Delete-with-children handling	Caught `ProtectedError`	Caught `QueryException`	Returns `false`, no exception	Caught raw database exception

Hands-On Exercises

EXERCISE 1

Build the searchable flat parent picker for the admin interface, excluding the page itself from its own candidate-parent list.

View solution

EXERCISE 2

Build movePage() and updateDescendantPaths(), and confirm a real reparent operation correctly cascades full_path updates to every descendant, not just the moved page itself.

View solution

EXERCISE 3

Attempt to delete a page with children via the DELETE endpoint, confirm the raw database constraint throws, and confirm the try/catch turns it into a real 409 response with a readable message.

View solution

CHAPTER 10 QUICK REFERENCE

- **No free admin, no scaffold generator** — the most minimal starting point of all five siblings
- `movePage()` / `updateDescendantPaths()` — fully explicit, since Drizzle has no lifecycle hooks
- **Delete refusal needs a real** `try` / `catch` — the same ceremony as Laravel, and precisely, as Django too
- **A corrected rule** — Rails' own smooth path was ActiveRecord's own API design, not a universal application-level trait
- **Next chapter:** Deployment

Chapter 1's `output: 'server'` plus `@astrojs/node` already made this deployment story real from day one — this chapter just runs it in production.

Building & Running the Standalone Server

```
npm run build
node ./dist/server/entry.mjs
```

The Node adapter's `'standalone'` mode produces a real, self-contained HTTP server — `dist/server/entry.mjs` — listening on its own port (`4321` by default), no different in role from Rails' own Puma process or Django's own `gunicorn`.

Secrets: The Plainest Story of All Five

```
# .env — excluded via .gitignore
DATABASE_URL=mysql://user:pass@localhost/website
AUTH_SECRET=a-real-random-secret-value
```

`AUTH_SECRET` is what Auth.js uses to sign sessions — the direct equivalent of Django's `SECRET_KEY`, Laravel's `APP_KEY`, or Rails' `master.key`. Astro's own version of this story is the simplest of the five: a plain `.env` file, matching Next.js, Django, and Laravel's own convention exactly — not Rails' own reversed, encrypted-commit approach.

Keeping the Process Alive: PM2

```
npm install -g pm2
pm2 start ./dist/server/entry.mjs --name website-astro
pm2 save
pm2 startup
```

A standalone Node process needs a real supervisor to restart it on crash or reboot — the same underlying need `gunicorn` has in the Django rebuild's own deployment chapter, just answered here with PM2 instead of a systemd service unit.

Apache's Own Third Mechanism: `mod_proxy_http`

```
# Apache VirtualHost — the site's own already-running Apache
<VirtualHost *:443>
    ServerName osztromok.com

    ProxyPreserveHost On
    ProxyPass / http://127.0.0.1:4321/
    ProxyPassReverse / http://127.0.0.1:4321/

    SSLEngine on
    SSLCertificateFile /etc/letsencrypt/live/osztromok.com/fullchain.pem
    SSLCertificateKeyFile /etc/letsencrypt/live/osztromok.com/privkey.pem
</VirtualHost>
```

A THIRD DISTINCT APACHE MECHANISM, NOT A REPEAT OF LARAVEL'S OR RAILS'

`mod_proxy_http` forwards requests over plain HTTP to any backend server — genuinely different from Laravel's own `mod_proxy_fcgi` (a FastCGI-specific protocol talking to a PHP-FPM process pool) and from Rails' own `mod_passenger` (embedding process management directly into Apache itself). Astro's Node server is just an ordinary HTTP server, so plain HTTP proxying is the natural fit — the same underlying idea Django's own `gunicorn`-behind-a-reverse-proxy architecture uses, but here reusing the site's own already-live Apache the way Laravel and Rails did, rather than standing up nginx from nothing the way Django's own Chapter 11 had to.

Migrations in Production

```
npx drizzle-kit migrate
```

Five Deployment Stories, Compared

	NEXT.JS	DJANGO	LARAVEL	RAILS	ASTRO
App server process	Node (PM2)	gunicorn	PHP-FPM	Passenger-embedded	Node adapter (PM2)
Reverse proxy	nginx (new)	nginx (new)	Apache (already live)	Apache (already live)	Apache (already live)
Apache module used	N/A	N/A	`mod_proxy_fcgi`	`mod_passenger`	`mod_proxy_http`
Secrets model	Plain `.env`	Plain `.env`	Plain `.env`	Encrypted, committed	Plain `.env`

THREE "MODERNIZE IN PLACE" STORIES, ONE ARCHITECTURE

Laravel, Rails, and now Astro all reuse the site's own already-live Apache — three genuinely different mechanisms (`mod_proxy_fcgi`, `mod_passenger`, `mod_proxy_http`) achieving the same real goal. Django and Next.js, needing a genuinely different runtime Apache had no existing path for, stood up nginx from nothing instead.

Hands-On Exercises

EXERCISE 1

Build the production server (npm run build, then run dist/server/entry.mjs directly), and confirm it serves the site correctly on its own configured port.

View solution

EXERCISE 2

Configure PM2 to run the server, then simulate a crash (kill the process directly) and confirm PM2 restarts it automatically.

View solution

EXERCISE 3

Configure the Apache VirtualHost with mod_proxy_http reverse-proxying to the Node server, and confirm the site is reachable through Apache's own domain rather than the raw Node port directly.

View solution

CHAPTER 11 QUICK REFERENCE

- `dist/server/entry.mjs` — the Node adapter's own standalone production server
- **Plain** `.env` — the simplest secrets story of all five siblings, matching Next.js/Django/Laravel
- **PM2** — the same underlying need as Django's own `gunicorn` supervisor
- `mod_proxy_http` — a third distinct Apache mechanism, genuinely different from `mod_proxy_fcgi` and `mod_passenger`
- **Three "modernize in place" stories** — Laravel, Rails, and Astro all reuse the site's own live Apache; Django and Next.js stood up nginx from nothing
- **Next chapter:** Capstone — A Complete, Working Flexible-Routing Site

Website Rebuild with Astro

Chapter 12 · Capstone: A Complete, Working Flexible-Routing Site

Sam — the same site admin persona from every capstone in this series, reused a fifth and final time — logs into this course's own deployed Astro version. Every step below draws on a specific earlier chapter, closing with an attribution table, an honest scope note, and the close of the entire five-framework Website Rebuild series.

STEP 1 — VISITING THE LIVE DEPLOYMENT

Sam opens the site. Apache's own `mod_proxy_http` forwards the request to the standalone Node server, kept alive under PM2's own supervision on the same already-live Apache the site has always run behind.

STEP 2 — LOGGING IN

Sam enters the legacy admin credential. `@auth/astro`'s own Credentials provider runs its `authorize()` callback, and `bcryptjs` — the identical package the Next.js rebuild already used — verifies it against the stored `$2y$`-tagged hash with no configuration needed.

STEP 3 — BUILDING THE FIVE-LEVEL CHAIN

Through the hand-built admin interface — no free admin, no scaffold generator, the most minimal starting point of all five siblings — Sam creates `programming`, then `general-purpose-languages`, then `java`, then `fundamentals`, then `chapter-1`. Each save calls Chapter 2's own explicit `computeFullPath` helper, since Drizzle has no lifecycle hooks to do it automatically.

STEP 4 — VIEWING THE PAGE

Visiting the full five-segment path, Chapter 3's `[...path].astro` resolves it with a single Drizzle query — no `getStaticPaths()` anywhere — and Chapter 4's `PageLayout` renders the breadcrumb and `set:html={page.body}` content, styled by Chapter 5's dark theme.

STEP 5 — EDITING THE TITLE, NOW ACTUALLY PROTECTED

Sam edits the Chapter 1 page's title. Chapter 8's endpoint and Zod validation handle the submission exactly as before — but Chapter 9's `getSession(request)` check is now in place, the first genuine "place for an auth check" this course ever had. Logged out, the identical request now returns a `401` instead.

STEP 6 — REORGANIZING: PAYING THE REAL CASCADE COST

Sam decides `fundamentals` belongs under a different parent. Chapter 10's `movePage` updates `fundamentals`' own `fullPath` first, then `updateDescendantPaths` recurses down and updates `chapter-1`'s path too — the real payment of the cascade cost Chapter 2 deliberately deferred at the very start of the course.

STEP 7 — ATTEMPTING TO DELETE A PAGE WITH CHILDREN

Sam tries to delete `java`, which still has `fundamentals` beneath it. Chapter 2's `onDelete: 'restrict'` database constraint refuses the operation, and Chapter 10's `try`/`catch` turns the raw database exception into a real, readable `409` response — the same ceremony Laravel's own capstone needed, not Rails' smoother built-in path.

STEP 8 — CONFIRMING THE KANJI MIGRATION HELD UP

Sam visits the kanji page. Chapter 7's resolution holds on both fronts: the routing and rendering were already solved by Chapters 2 and 4, and JavaScript's own UTF-16 string model — the same one Next.js already relied on — never risked corrupting the character. The database's own `utf8mb4` charset, set explicitly on the `mysql2` connection since no automatic default existed here, stores it correctly.

STEP 9 — A QUICK N+1 SANITY CHECK

Sam checks the query log while loading the breadcrumb-heavy Chapter 1 page. Chapter 6's `db.query.pages.findFirst({ with: ... })` is doing its job — the same deliberately-planted N+1 pattern caught a fifth time, resolved via Drizzle's own relational API, verified as genuinely mirroring Prisma's own `include`.

STEP 10 — CONFIRMING THE DEPLOY ITSELF IS HEALTHY

Sam confirms PM2 shows the Node process running cleanly, and that Apache's own `mod_proxy_http` is forwarding requests correctly — the last confirmation in a series that has now rebuilt the same real site five separate times.

Chapter Attribution

STEP	CHAPTER(S) APPLIED
1. Visiting the deployment	11 — Deployment
2. Logging in	9 — Admin Authentication
3. Building the five-level chain	2 — URL & Content Model, 10 — Admin CRUD
4. Viewing the page	3 — Routing, 4 — Views, 5 — Styling
5. Editing the title	8 — Dynamic Content & Forms, 9 — Admin Authentication
6. Reorganizing	2 — deferred cascade cost, 10 — `movePage`
7. Delete refusal	2 — `onDelete: 'restrict'`, 10 — Admin CRUD
8. Kanji migration	7 — Rendering Content & the Kanji Edge Case
9. N+1 sanity check	6 — The Database: Drizzle ORM
10. Deploy health check	11 — Deployment

Honest Scope Note

WHAT THIS COURSE DELIBERATELY DOESN'T COVER

- No multi-admin roles or permission levels — a single legacy admin credential only
- No revision history or audit log of content edits
- No media/image upload handling
- No automated tests or CI pipeline
- No internationalization beyond the kanji character-storage edge case itself
- No rate limiting or lockout policy on the login form
- **No CSRF protection** — a real, unaddressed gap named honestly in Chapter 8, never solved

ASTRO'S OWN HONEST CONTRIBUTIONS TO THIS SERIES

Two genuinely Astro-specific findings, grounded in real, verified facts: the TypeScript circular-reference workaround Drizzle's self-referencing schema needs (Chapter 2), and `mod_proxy_http` as a genuinely third distinct Apache deployment mechanism (Chapter 11) — alongside real, direct reuse rather than manufactured uniqueness, where it was honest to reuse: the identical `bcryptjs` package and the identical Auth.js library the Next.js rebuild already used (Chapter 9).

The Website Rebuild Series Is Now Complete

Five frameworks, one real site, rebuilt five separate times: Next.js, Django, Laravel, Ruby on Rails, and now Astro. Each course found its own genuine advantages and honest limits rather than declaring any single architecture simply "the best" — the same standard the Food Tracker Quartet set for comparative courses on this site.

Hands-On Exercises

EXERCISE 1

Trace Step 6's reorganization through the code: what does movePage() update first, and what does updateDescendantPaths() update afterward?

View solution

EXERCISE 2

Explain what changed between Step 5 in this capstone and the identical-looking action back in Chapter 8, and name the exact line of code responsible for the difference.

View solution

EXERCISE 3

Name two genuinely Astro-specific findings from this course — not shared with any sibling — and explain the real technical fact each one is grounded in.

View solution

COURSE COMPLETE

- **12/12 chapters** — Website Rebuild with Astro is now complete
- **Standout findings named honestly throughout** — Drizzle's TypeScript workaround (Ch2), genuine Prisma kinship (Ch6), the literal same Auth.js library as Next.js (Ch9), a third Apache mechanism (Ch11)
- **Real limits named honestly too** — no built-in validation or CSRF protection (Ch8), no free admin or scaffold generator (Ch10)
- **Fifth and final course in the Website Rebuild series** — alongside Next.js, Django, Laravel, and Rails
- **The entire five-framework Website Rebuild series is now complete**