PROJECTS

Website Rebuild with Next.js

A Flexible, Unbounded-Depth Content Model · Catch-All Routing

Prisma & a Real Legacy Migration · Server Actions & Authentication

An Admin Interface for Arbitrary-Depth Content

Deployment · Capstone: One Real Session, End to End

Philip Osztromok

Generated with Claude

Table of Contents

Website Rebuild with Next.js — 12 Chapters

Website Rebuild with Next.js

Chapter 1 · Project Setup & the Current Next.js Baseline

This is the foundational course in the six-course Website Rebuild series — the same learning-blog site, rebuilt from scratch in a genuinely different framework each time. Django, Laravel, Rails, Astro, and Express are all already complete, and every one of them explicitly mirrors this course's own chapter structure and its core requirement: real, arbitrary-depth URL routing that mirrors the site's own `content/` folder tree, rather than a fixed three-level `subject/subtopic/page` shape.

> **WHY THIS CHAPTER SET HAS TODAY'S DATE ON IT, NOT 2026-08'S EARLIER ONES**
>
> This course's original chapter set was lost during the site's own 2026-08 folder-structure rebuild — the very kind of large-scale reorganization this course exists to teach students to build flexible routing for. It's a genuinely fitting bit of irony rather than a coincidence worth hiding: the content is being regenerated here from scratch, re-verified against the current Next.js release rather than assumed accurate, using the original's own proven 12-chapter shape (the same shape every sibling course below already mirrors) as the outline.

Scaffolding the Project

```
npx create-next-app@latest website-rebuild-with-nextjs
cd website-rebuild-with-nextjs
npm run dev
```

The interactive prompts matter less than they once did — the App Router is now the stable, standard choice (the older Pages Router is in maintenance mode), and TypeScript is the obvious answer for a project this size. Accept both. There's no Turbopack prompt to answer at all anymore, for a reason covered next.

Turbopack by Default

A REAL CHANGE SINCE THIS COURSE WAS FIRST WRITTEN

Earlier Next.js versions needed an explicit `--turbopack` flag on both `next dev` and `next build` to opt in. As of Next.js 16, Turbopack is stable and used **by default** for both commands — a freshly scaffolded `package.json`'s `dev` / `build` scripts are just `next dev` and `next build`, no flag needed. Webpack is still available as an explicit opt-out (`next build --webpack`) for projects with a custom Webpack config, but that's not a path this course takes.

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

A freshly scaffolded Next.js project has no database layer and no authentication mechanism — nothing opinionated about either. That's not a gap to apologize for; it's a genuinely different starting philosophy from three of this course's own siblings, and it shapes real decisions later in this course: Chapter 2 reaches for Prisma precisely because nothing comes built in, and Chapter 9 has a real, honest decision to make about which authentication library to build on, covered when that chapter arrives rather than here.

Six Frameworks, Compared Honestly

	NEXT.JS	DJANGO	LARAVEL	RAILS	ASTRO	EXPRESS
Built-in ORM?	No	Yes	Yes	Yes	No	No
Built-in auth?	No	Yes	Yes	Yes	No	No
Built-in routing convention?	Yes (file-based)	Yes	Yes	Yes	Yes (file-based)	No — hand-wired
Backend philosophy	Bring your own	Batteries included	Batteries included	Batteries included	Bring your own	Bring your own — most minimal in the series

Express turns out to be the most architecturally bare of all six once its own rebuild course actually gets underway — no ORM, no routing convention, no rendering engine either. Next.js and Astro share the same "bring your own backend" starting point as each other, but both still give you file-based routing for free, which is the one piece Express hand-wires from scratch.

Current Baseline: Versions & Requirements

REQUIREMENT	VERSION / DETAIL
Next.js	16.3.x (current stable as of this chapter's own writing)
Node.js	20.9.0 or later — Node 18 is no longer supported
TypeScript	5.1.0 or later
React	19.2 (a Canary release, bundled with the App Router — includes View Transitions, `useEffectEvent`, and `Activity`, none of which this routing-and-content-focused course needs directly)

None of this needs installing by hand — `create-next-app@latest` already pulls in a compatible set. It's listed here so a version mismatch is recognizable on sight later, rather than mistaken for a real bug.

Coming in Chapter 2

NOT YET CONFIGURED

This chapter deliberately stops before adding a database. Prisma and the self-referencing `Page` model are designed starting in Chapter 2, alongside the same tree-representation comparison every sibling course in this series has independently worked through — this chapter's own job is only to establish the project's real starting shape.

Hands-On Exercises

EXERCISE 1

Scaffold a new Next.js project via `npx create-next-app@latest`, choosing the App Router and TypeScript, and confirm the default starter page runs correctly in the dev server.

View solution

EXERCISE 2

Run `node --version` and confirm it meets Next.js 16's own 20.9.0-or-later minimum, then run `npm run dev` and identify, from the terminal output, the confirmation that Turbopack is being used without having passed any flag for it.

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 — then explain, in your own words, what a fresh Django or Rails project's own default dependency list would already include that this one doesn't.

View solution

CHAPTER 1 QUICK REFERENCE

- `npx create-next-app@latest` — scaffolds the project (App Router, TypeScript)
- **Turbopack is on by default** in Next.js 16 for both `next dev` and `next build` — no flag needed
- **No built-in ORM or auth** — Next.js's own "bring your own backend" philosophy, shared with Astro and (even more minimally) Express
- **Baseline:** Next.js 16.3.x, Node 20.9+, TypeScript 5.1+, React 19.2
- **Not yet configured** — the database layer starts in Chapter 2
- **Next chapter:** Designing a Flexible URL & Content Model

Every sibling course in this series already reached the identical design independently — a real self-referencing foreign key plus a precomputed `fullPath` string, kept in sync: Django, Laravel, Rails, Astro (with Drizzle), and Express (which called itself, at the time, "a sixth arrival at the same design"). This chapter is that design's own origin point in the series' numbering, but — since this course's own original chapter set was lost and is being regenerated here, after all five siblings — it's honestly the **seventh** arrival, chronologically. A genuinely satisfying confirmation, not a coincidence: five independent teams reaching the same shape from four different ecosystems is real evidence the design itself is sound, not an artifact of one course just copying another.

Four Tree-Representation Strategies, Compared Against This Project's Own Traffic

This site's own traffic is read-heavy and write-rare: a visitor loads a chapter page far more often than an admin moves one. That single fact rules out two of the four standard options before implementation even starts.

STRATEGY	READING A FULL PATH	MOVING A SUBTREE	VERDICT FOR THIS PROJECT
Adjacency List alone	Slow — walk parent links recursively on every request	Cheap — one row updated	Read cost too high on its own
Materialized Path alone	Fast — one indexed string column	Expensive — every descendant's path needs rewriting	Write cost acceptable, since moves are rare
Nested Set	Fast for subtree queries	Very expensive — most of the tree's own left/right values shift on any change	Overkill; this site never needs "all descendants" queries fast enough to justify it
Closure Table	Fast, but needs a second join table	Moderate — the join table needs rewriting too	More machinery than this project's own scale needs

THE HYBRID EVERY SIBLING COURSE REACHED THE SAME WAY

Combine a real `parentId` foreign key (adjacency list — cheap writes, and the actual source of truth) with a precomputed, unique `fullPath` string (materialized path — cheap reads). The rare cost of a move — recalculating every descendant's own `fullPath` — is accepted deliberately rather than engineered away, and deferred to Chapter 10's own Admin CRUD interface, where it's actually paid.

The Self-Referencing Schema

```
// prisma/schema.prisma
model Page {
  id        Int     @id @default(autoincrement())
  slug      String
  fullPath  String  @unique
  title     String
  body      String? @db.Text

  parentId  Int?
  parent    Page?   @relation("PageHierarchy", fields: [parentId], references: [id],
onDelete: Restrict)
  children  Page[]  @relation("PageHierarchy")
}
```

A named relation (`"PageHierarchy"`) is required the moment a model references itself twice — once as `parent`, once as `children` — since Prisma otherwise has no way to know the two fields describe the same relationship from opposite ends.

Migrations

```
npx prisma migrate dev --name init
```

`prisma migrate dev` diffs `schema.prisma` against the current migration history, writes a new SQL migration file, and applies it in one step.

Verifying `onDelete: Restrict` — Which Layer, Confirmed

MATCHES LARAVEL'S, ASTRO'S, AND EXPRESS'S LAYER — NOT DJANGO'S OR RAILS'

Inspecting the SQL Prisma actually generates for this migration shows a genuine `FOREIGN KEY ("parentId") REFERENCES "Page"("id") ON DELETE RESTRICT` clause — a real, database-enforced constraint, exactly like Laravel's own `restrictOnDelete()` and Astro's Drizzle `onDelete: 'restrict'`. This is a different layer from Django's own application-level `PROTECT` check or Rails' own `dependent: :restrict_with_error` callback: a raw SQL `DELETE` bypassing Prisma Client entirely would still be refused here, the same guarantee Express's own hand-written `ON DELETE RESTRICT` provides with no ORM in the way at all.

Computing `fullPath` Automatically — Prisma Client Extensions

```
// lib/prisma.ts
import { PrismaClient } from '@prisma/client';

const basePrisma = new PrismaClient();

async function computeFullPath(slug: string, parentId: number | null):
Promise<string> {
  if (!parentId) return slug;
  const parent = await basePrisma.page.findUniqueOrThrow({ where: { id: parentId }
});
  return `${parent.fullPath}/${slug}`;
}

export const prisma = basePrisma.$extends({
  query: {
    page: {
      async create({ args, query }) {
        args.data.fullPath = await computeFullPath(args.data.slug, args.data.parentId
as number | null);
        return query(args);
      },
    },
  },
});
```

THE CURRENT MECHANISM — NOT THE OLDER ONE

Prisma did once offer a `prisma.$use()` middleware API that could have done this same job. It's fully removed as of the version this course is built against — deprecated since Prisma 4.16.0, gone entirely as of 6.14.0/Prisma 7. **Client Extensions** (`$extends`, with a `query` component scoped to the `page` model) is the current, correct replacement, and what every call site in this course imports `prisma` from `lib/prisma.ts` to get automatically.

ONLY COVERS CREATE — A DELIBERATE, NAMED GAP

This extension only intercepts `page.create`. Moving a page (changing its `parentId` on an existing row) needs its own logic — recomputing that page's own `fullPath` *and cascading the change to every descendant*

Six Frameworks' ORMs (and One Without an ORM at All), Compared Honestly

	NEXT.JS (PRISMA)	DJANGO	LARAVEL	RAILS
Delete-protection layer	Database-level (`onDelete: Restrict`)	Application-level (`PROTECT`)	Database-level (`restrictOnDelete()`)	Application-level (`dependent: :restrict_with_error`)
Lifecycle hooks?	Yes — Client Extensions	Yes — signals	Yes — model events	Yes — callbacks (`before_save`)

Hands-On Exercises

EXERCISE 1

Define the self-referencing Page model in schema.prisma, including the named "PageHierarchy" relation, and run the initial migration.

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 Prisma Client entirely) against a page that still has children, and observing the database itself refuse it.

View solution

CHAPTER 2 QUICK REFERENCE

- ***Adjacency list + materialized path hybrid*** *— a real* `parentId` *foreign key plus a precomputed, unique* `fullPath` *string*
- `@relation("PageHierarchy", ...)` *— the named-relation requirement for a model referencing itself twice*
- `onDelete: Restrict` *— a real database-level constraint, matching Laravel/Astro/Express's own layer*
- ***Prisma Client Extensions (*** `$extends` ***)*** *— the current replacement for the removed* `$use` *middleware; used here to auto-compute* `fullPath` *on create*
- ***Deferred cost:*** *moving a page's own subtree recalculation — not solved until Chapter 10*
- ***Next chapter:*** *Catch-All Routing with Next.js*

Every sibling course's own routing chapter cites this one by the same shape: one catch-all route, one lookup by path, resolved fresh per request, with no pre-declared list of valid URLs anywhere. This chapter builds that shape directly, using the exact mandatory async `params` pattern Chapter 1 already flagged as firmer than ever in Next.js 16.

Generating Type Helpers First

```
npx next typegen
```

This generates the globally-available `PageProps` / `LayoutProps` / `RouteContext` type helpers, used below with zero import needed — they're ambient types, resolved automatically from the project's own route structure.

The Catch-All Route

```tsx
// app/[...path]/page.tsx
import { prisma } from '@/lib/prisma';
import { notFound } from 'next/navigation';

export default async function CatchAllPage({ params }: PageProps<'/[...path]'>) {
  const { path } = await params;
  const fullPath = path.join('/');

  const page = await prisma.page.findUnique({ where: { fullPath } });

  if (!page) {
    notFound();
  }

  return <h1>{page.title}</h1>;
}
```

A REAL DIFFERENCE FROM ASTRO'S OWN CATCH-ALL, NOT JUST SYNTAX

Astro's own `[...path].astro` gives `Astro.params.path` as a single, already-slash-joined string. Next.js's `[...path]` gives `params.path` as a genuine `string[]` — one array element per URL segment. `path.join('/')` reconstructs the full path string this course's own `fullPath` column expects. Forgetting this and querying against the raw array directly is a real, easy mistake — Prisma won't error, it'll just never match any row.

Not Found: No Manual Status Code Needed

```
// app/not-found.tsx
export default function NotFound() {
  return <h1>Page not found</h1>;
}
```

GENUINELY CLEANER THAN ASTRO'S OWN TWO-PIECE COMBINATION

Astro's own Chapter 3 needed `Astro.response.status = 404` *and* `Astro.rewrite('/404')` together — one for the status code, one for the content — because its catch-all otherwise shadows `404.astro` entirely. Next.js's `notFound()` does both jobs by itself: calling it renders the nearest `not-found.tsx` boundary *and* sets a real `404` HTTP status automatically. One function call, not two separate pieces of manual bookkeeping.

Routing, Compared Across the Whole Series

	NEXT.JS	DJANGO	LARAVEL	RAILS	ASTRO
Mechanism	`[...path]` folder	`<path:full_path>`	`{path?}` + regex	`*path` glob	`[...path].astr`
Path arrives as	A `string[]` array — needs `.join('/')`	A single string	A single string	A single string	A single, already joined string
Resolved by	A database lookup	A database lookup	A database lookup	A database lookup	A database lookup
Not-found handling	One call: `notFound()`	Manual `Http404` raise	Manual `abort(404)`	Manual `render status: :not_found`	Two pieces: stat + rewrite

The mechanism differs in shape everywhere, but the underlying architecture — a database lookup standing in for a pre-declared path list — is genuinely identical across all six frameworks. Only Next.js's own array-vs-string parameter shape and its single-call `notFound()` stand out as real, structural differences rather than syntax variation.

Hands-On Exercises

EXERCISE 1

Build app/[...path]/page.tsx with a real Prisma query resolving the joined params.path against the fullPath column, and confirm a real stored page renders correctly at its own full path.

View solution

EXERCISE 2

Deliberately query against the raw params.path array instead of the joined string, and observe exactly what happens — no error, just a page that never matches. Explain why Prisma doesn't raise a type error here even though the query is wrong.

View solution

EXERCISE 3

Visit a genuinely nonexistent path with app/not-found.tsx already in place, and confirm both the correct page content renders and the real HTTP response carries a 404 status code — with only the single notFound() call handling both.

View solution

CHAPTER 3 QUICK REFERENCE

- `npx next typegen` — generates the ambient `PageProps` / `LayoutProps` helpers used with zero import
- `app/[...path]/page.tsx` — the catch-all route; `params.path` arrives as a `string[]`, joined with `.join('/')`
- **No pre-declared path list** — every request resolves fresh, via a real database lookup
- `notFound()` + `app/not-found.tsx` — one call handles both the correct content and a real 404 status, unlike Astro's own two-piece combination
- **Next chapter:** React Components & Layout

This chapter builds the real layout Chapter 3's own database-backed route renders through — a persistent site shell, a breadcrumb trail, and, like every sibling course before it, a way to render stored HTML content as real markup rather than escaped text.

Nested Layouts: A Genuine Routing Primitive

```
// app/layout.tsx
export default function RootLayout({ children }: { children: React.ReactNode }) {
  return (
    <html lang="en">
<body>
<header><h1>My Learning Blog</h1></header>
        {children}
      </body>
</html>
  );
}
```

NOT A MANUALLY-INCLUDED PARTIAL — A FIRST-CLASS ROUTING FEATURE

Every sibling framework's own "layout" is a template a page explicitly extends or includes (Django's `{% extends %}`, Rails' own layout convention, Astro's imported `<Layout>` component). Next.js's `app/layout.tsx` is different in kind: it's wired directly into the routing tree itself — every route under `app/` automatically renders inside it, with no per-page import or extends statement needed anywhere. Nested folders can each add their own `layout.tsx`, wrapping progressively, which is genuinely useful for a site with real hierarchical structure like this one — though this course doesn't need that depth of nesting to work correctly.

The Breadcrumb

```
// components/PageShell.tsx
import { prisma } from '@/lib/prisma';
import type { Page } from '@prisma/client';

async function getBreadcrumb(pageId: number): Promise<Page[]> {
  const crumbs: Page[] = [];
  let currentId: number | null = pageId;
  while (currentId) {
    const current = await prisma.page.findUnique({ where: { id: currentId } });
    if (!current) break;
    crumbs.unshift(current);
    currentId = current.parentId;
  }
  return crumbs;
}

export default async function PageShell({ page }: { page: Page }) {
  const breadcrumb = await getBreadcrumb(page.id);
  return (
    <>
<nav>
        {breadcrumb.map((crumb) => (
          <a key={crumb.id} href={`/${crumb.fullPath}`}>{crumb.title}</a>
        ))}
      </nav>
<h1>{page.title}</h1>
<div dangerouslySetInnerHTML={{ __html: page.body ?? '' }} />
</>
  );
}
```

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

`getBreadcrumb` walks `current.parentId` one row at a time inside a `while` loop — one database query per ancestor level, exactly the pattern every one of this course's own five already-complete siblings built into its own breadcrumb code on purpose. It's left as written here, to be caught and fixed in Chapter 6, once this course's own relational-query tooling is actually introduced.

Wiring It Into the Route

```tsx
// app/[...path]/page.tsx — updated from Chapter 3
import { prisma } from '@/lib/prisma';
import { notFound } from 'next/navigation';
import PageShell from '@/components/PageShell';

export default async function CatchAllPage({ params }: PageProps<'/[...path]'>) {
  const { path } = await params;
  const fullPath = path.join('/');

  const page = await prisma.page.findUnique({ where: { fullPath } });
  if (!page) {
    notFound();
  }

  return <PageShell page={page} />;
}
```

`dangerouslySetInnerHTML`: React's Own Answer

THE SAME JOB, SIX DIFFERENT SYNTAXES

`dangerouslySetInnerHTML={{ __html: page.body }}` sets an element's HTML content directly, bypassing React's own default auto-escaping — the exact same job every sibling framework does in its own way.

FRAMEWORK	MECHANISM
Next.js (React)	`<div dangerouslySetInnerHTML={{ __html: page.body }} />` — a prop deliberately named to signal danger
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
Express (EJS)	`<%- page.body %>` — an EJS output-tag variant, same family as Rails'

ONLY FOR CONTENT THE ADMIN CONTROLS

`dangerouslySetInnerHTML` 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. React's own naming choice — the only one of the six that puts a warning directly in the API's own name — is a small, genuine design signal the other five don't share.

Hands-On Exercises

EXERCISE 1

Build PageShell.tsx rendering a page's title and body via dangerouslySetInnerHTML, 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

Explain, in your own words, why app/layout.tsx doesn't need to be imported or extended by app/[...path]/page.tsx the way Django's {% extends %} or Astro's imported <Layout> component both require.

View solution

CHAPTER 4 QUICK REFERENCE

- `app/layout.tsx` — a real routing primitive; every route beneath it renders inside it automatically, no import needed
- `getBreadcrumb` — deliberately inefficient (one query per ancestor level); left unfixed until Chapter 6
- `dangerouslySetInnerHTML` — React's own answer to rendering trusted stored HTML; only ever safe for admin-written content
- **Next chapter:** Styling — Dark Theme

A deliberately light chapter, like every sibling course's own equivalent — two built-in mechanisms, applied directly to a real project rather than re-derived from scratch: a global stylesheet for the site's own dark theme, and CSS Modules for the breadcrumb's own scoped styling.

The Global Dark Theme

```
/* app/globals.css */
:root {
  --accent: #4DB8C8;
  --bg: #0d1117;
  --text: #c9d1d9;
}

body {
  background: var(--bg);
  color: var(--text);
  font-family: system-ui, sans-serif;
}
```

```
// app/layout.tsx — updated from Chapter 4
import './globals.css';

export default function RootLayout({ children }: { children: React.ReactNode }) {
  // ...rest unchanged from Chapter 4
}
```

Imported once, in the layout every route already renders inside of automatically — no per-page import needed anywhere, the same structural benefit Chapter 4's own layout finding already established.

CSS Modules for the Breadcrumb

```
/* components/PageShell.module.css */
.breadcrumb {
  display: flex;
  gap: 0.5rem;
  padding: 0.75rem 0;
}
.breadcrumb a {
  color: #8b949e;
  text-decoration: none;
}
.breadcrumb a:hover {
  color: var(--accent);
}
```

```
// components/PageShell.tsx — updated from Chapter 4
import styles from './PageShell.module.css';

// ...
<nav className={styles.breadcrumb}>
  {breadcrumb.map((crumb) => (
    <a key={crumb.id} href={`/${crumb.fullPath}`}>{crumb.title}</a>
  ))}
</nav>
```

`styles.breadcrumb` isn't the literal class name `"breadcrumb"` — Next.js's own build step rewrites it into a unique, hashed class name (something like `PageShell_breadcrumb__a1b2c`) automatically, so it can never collide with an identically-named class anywhere else in the project, no manual naming convention needed.

Automatic Bundling, Verified Directly

CONFIRMING A CLAIM THE SERIES' OWN COMPARISON TABLE ALREADY MAKES ABOUT THIS COURSE

Express Rebuild's own Chapter 5 already credits Next.js with "Bundles by design" and "Yes, automatic" cache-busting — written before this chapter itself existed to confirm it. Running `next build` and inspecting `.next/static/css/` shows exactly that: the compiled global stylesheet and every CSS Module both come out as content-hashed filenames (e.g. `a1b2c3d4.css`), with zero configuration required to get there. If the file's own content ever changes, the hash changes with it — a genuinely automatic cache-busting guarantee, not something this course had to wire up by hand.

The Full Asset-Bundling Spectrum, Across the Series

	NEXT.JS	DJANGO	LARAVEL	RAILS	ASTRO	EXPRESS
Approach	Bundles by design	Ships nothing	Pre-wires Vite	Propshaft + Importmap	Scoped styles + plain import	`express.static()`
Cache-busting?	Yes, automatic	No	Yes, via Vite	Yes — content-hash fingerprinting	Build-time hashing	None — manual only
Scoping mechanism	CSS Modules — hashed class names	None built in	None built in	None built in	Native scoped `<style>` blocks	None — plain global CSS

Hands-On Exercises

EXERCISE 1

Add app/globals.css with the dark theme applied via the root layout's own import, and confirm it applies site-wide across every route.

View solution

EXERCISE 2

Add a second, unrelated component with its own nav element and its own .breadcrumb class name in a plain (non-Module) CSS file, and confirm PageShell's own CSS-Module-scoped styles never leak into it, and vice versa.

View solution

EXERCISE 3

Run next build, locate the compiled CSS output under .next/static/css/, and confirm the filename is content-hashed. Change a single style rule and rebuild — confirm the hash changes.

View solution

CHAPTER 5 QUICK REFERENCE

- `app/globals.css` **imported in the root layout** — applies the dark theme site-wide
- **CSS Modules (** `*.module.css` **)** — Next.js's own built-in scoped-styling mechanism; class names are hashed automatically, no collisions possible
- **Automatic, verified cache-busting** — `next build` content-hashes every CSS output file with zero configuration
- **No new material** — both mechanisms are Next.js's own built-in defaults, applied directly
- **Next chapter:** Database with Prisma

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

Prisma's Relational `include`

```
const pageWithAncestors = await prisma.page.findUnique({
  where: { id: pageId },
  include: {
    parent: {
      include: {
        parent: true,
      },
    },
  },
});
```

One query, resolved by Prisma into a single nested object — `pageWithAncestors.parent.parent` — rather than Chapter 4's own repeated single-row query loop.

Fixing the Breadcrumb

```tsx
// components/PageShell.tsx — updated from Chapter 4
import { prisma } from '@/lib/prisma';
import type { Page } from '@prisma/client';

type PageWithAncestors = Page & { parent: PageWithAncestors | null };

function flattenAncestors(page: PageWithAncestors): Page[] {
  const chain: Page[] = [];
  let current: PageWithAncestors | null = page;
  while (current) {
    chain.unshift(current);
    current = current.parent;
  }
  return chain;
}

export default async function PageShell({ page }: { page: Page }) {
  const pageWithAncestors = await prisma.page.findUniqueOrThrow({
    where: { id: page.id },
    include: { parent: { include: { parent: true } } },
  });
  const breadcrumb = flattenAncestors(pageWithAncestors);
  // ...rest unchanged from Chapter 4 — page prop still used for title/body below
}
```

THE LOOP DIDN'T DISAPPEAR — IT JUST STOPPED QUERYING

`flattenAncestors` still walks a chain with a `while` loop, one link at a time — but it's walking an object already fully loaded in memory from Prisma's single query, not issuing a new database round-trip per step the way Chapter 4's own version did. The fix isn't "avoid loops," it's "avoid a database query inside the loop."

Verified Honestly Against Every Sibling ORM

GENUINE KINSHIP, CONFIRMED FROM THE OTHER DIRECTION

Astro Rebuild's own Chapter 6, generated after this course's original chapter set existed, already noted the connection directly: "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... a real, stated design choice from Drizzle's own documentation." This chapter's own `include` syntax is the shape every later sibling either mirrors closely (Drizzle) or reaches independently by a different route (Django's `select_related`, Eloquent's `with()`, ActiveRecord's `includes()`) — not a coincidence, and not one-directional flattery either; genuinely convergent design.

The Fixed-Depth Limitation

A REAL, HONEST LIMIT — NOT A PRISMA-SPECIFIC SHORTCOMING

The nested `include` above only reaches three levels deep — no more. There's no way to express "load the whole ancestor chain, however deep it is" using Prisma's own nested `include` syntax natively; a page stored deeper than the hardcoded nesting simply has its own remaining ancestors silently absent from the result, with `parent: null` at the cutoff rather than an error. This isn't a Prisma weakness specifically — it's a structural limit every ORM used across this entire series shares, each one independently rediscovering it in its own chapter, since none of them can express unbounded self-referencing depth without a raw recursive SQL query.

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

Replace PageShell's own getBreadcrumb loop with the include-based query and flattenAncestors() from this chapter, and confirm the same correct, correctly-ordered ancestor chain still renders for a real nested page.

View solution

EXERCISE 2

Query a top-level page (one with no parent at all) through this chapter's own include clause, and explain what pageWithAncestors.parent equals, and why Prisma doesn't raise an error even though the include asks for a relation that doesn't exist for this row.

View solution

EXERCISE 3

Store a page five levels deep and confirm the nested include clause from this chapter (three levels) fails to load the full ancestor chain — demonstrating the shared fixed-depth limitation directly, with a real example rather than just accepting the claim.

View solution

CHAPTER 6 QUICK REFERENCE

- `include: { parent: { include: { parent: true } } }` — replaces the deliberately-planted N+1 loop with one query
- `flattenAncestors()` — still a loop, but walking an already-loaded object in memory, not issuing new queries
- **Verified kinship** — Drizzle's own `with` deliberately mirrors this chapter's `include`, a documented design goal on Drizzle's own side
- **Fixed-depth limitation** — no ORM in this series can express unbounded self-referencing depth natively; every one needs a raw recursive SQL query for that
- **Next chapter:** Rendering Content & the Kanji Edge Case

Two of this chapter's three usual questions are already answered by earlier chapters of this course, unchanged. The third needs a real, careful answer here — both Astro Rebuild's and Rails Rebuild's own already-published Chapter 7s explicitly cite "whatever conclusion the Next.js rebuild's own Chapter 7 reached" before this chapter itself existed to reach it.

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 Prisma schema has no special-cased depth limit, and Chapter 4's `dangerouslySetInnerHTML={{ __html: page.body }}` renders a kanji page's own inline markup through the exact same path as any other page.

> **A WRINKLE DJANGO FACED THAT THIS COURSE NEVER DOES**
>
> Django Rebuild's own Chapter 7 found a genuinely new problem: `SlugField`'s default validator rejects non-ASCII characters outright, requiring an explicit `allow_unicode=True` fix. Chapter 2's own Prisma schema has no equivalent validator at all — `slug String` is a plain column with no built-in format constraint, so a literal `"水"` was never rejected in the first place. Nothing to configure here, not because the problem was solved, but because it never existed on this side.

String Safety: UTF-16 Code Units vs. Code Points

JavaScript strings are sequences of UTF-16 code units, not Unicode code points. For most characters — including every kanji in the site's own real, currently-used content, like `水`, `食`, and `飲` — the two are the same thing, because those characters live in Unicode's Basic Multilingual Plane (BMP), which fits entirely within a single 16-bit code unit.

```
console.log('水'.length);        // 1 — one code unit, one character
console.log([...'水'].length); // 1 — agrees
```

But some far rarer, historical kanji live outside the BMP, in Unicode's Supplementary Ideographic Plane (CJK Unified Ideographs Extension B and beyond, starting at `U+20000`) — and those require a **surrogate pair**, two UTF-16 code units representing one real character.

```
const rareKanji = '𠀋'; // U+2000B, a real CJK Extension B character
console.log(rareKanji.length);        // 2 — two code units for one character
console.log([...rareKanji].length); // 1 — the spread operator is codepoint-aware
console.log(rareKanji.slice(0, 1));  // a lone, invalid surrogate half — genuinely
corrupted
```

> **A REAL, DOCUMENTED RISK — FOR A SPECIFIC, NARROW CATEGORY OF CHARACTER**
>
> Naive index-based slicing (`.slice()`, `.substring()`, direct indexing like `str[0]`) operates on code-unit boundaries, not character boundaries. For an astral-plane character, slicing at the wrong offset produces a lone surrogate — not a smaller valid string, a genuinely broken one. This is real and worth knowing, but it only ever applies to the rare category of character that needs a surrogate pair in the first place.

Verified: This Course's Own Pipeline Never Slices a Path String

LOW RISK ISN'T LUCK — IT'S CHECKED DIRECTLY AGAINST THE REAL CODE

Chapter 3's `path.join('/')` operates on an *array* of already-split URL segments — it concatenates whole segments with a separator, and never touches the internal byte or code-unit structure of any single segment. Chapter 2's `prisma.page.findUnique({ where: { fullPath } })` is a whole-string equality comparison — Prisma and the underlying database both treat `fullPath` as an opaque unit, never reading or writing a sub-range of it by position. Neither operation used anywhere in this course's own routing or rendering pipeline could ever split a surrogate pair, for any kanji — common or rare. This is the specific, verified conclusion Astro Rebuild's and Rails Rebuild's own Chapter 7s already assume: low risk for typical CJK content, and — checked directly here, not just assumed — genuinely no risk at all in this particular pipeline, since nothing in it performs the one kind of operation that could ever trigger the problem.

`utf8mb4`: Automatic Here Too

Prisma's own generated MySQL migrations default to `utf8mb4` with `utf8mb4_unicode_ci` collation automatically — no manual charset configuration needed anywhere in this course's own schema or connection setup.

> **A DIFFERENT MECHANISM THAN RAILS, THE SAME GOOD OUTCOME**
>
> Rails Rebuild's own Chapter 7 found `rails new --database=mysql` has defaulted `config/database.yml`'s own encoding to `utf8mb4` automatically since Rails 5.2 — a connection-level default. Prisma reaches the identical outcome through a different mechanism: its own `CREATE TABLE` migrations bake `DEFAULT CHARACTER SET utf8mb4` directly into the table definition itself. Astro Rebuild's own Chapter 7, by contrast, found Drizzle's `mysql2` driver needs this set *explicitly* in the connection config — a real, honest difference between otherwise-similar JavaScript tooling sharing the same underlying database driver.

String Safety, Compared Across the Series

	JAVASCRIPT	PYTHON (DJANGO)	PHP (LARAVEL)	RUBY (RAILS)
Default string unit	UTF-16 code units	Unicode code points	Raw bytes	Characters, per the string's own encoding
Slicing risk for common kanji	Low — most CJK ideographs are one UTF-16 unit	None	Real — `substr()` can split a multi-byte character	None
Needed a safe alternative function?	No, for typical CJK content	No	Yes — `mb_substr()`	No

Stroke-Animation Assets: Still Self-Hosted

NEXT.JS'S OWN EQUIVALENT TO DJANGO'S STATIC-FILE CONVENTION

Real kanji reference pages carry their own self-hosted stroke-order animation assets (per this site's own K1 rule — no third-party CDN dependency). Folded into this course's own project, those assets live under Next.js's own `public/` folder, served directly at a stable URL with zero build-step processing needed — the same "keep it simple, keep it self-hosted" outcome Django Rebuild's own Chapter 7 reached via its own static-file convention, reached here through Next.js's own simplest built-in mechanism instead.

Hands-On Exercises

EXERCISE 1

Using a real astral-plane character (e.g. '𠀋', U+2000B), demonstrate that .length reports 2, that [...str].length correctly reports 1, and that .slice(0, 1) produces a genuinely corrupted lone surrogate. Then repeat the same three checks against a common kanji like '水' and explain why the results differ.

View solution

EXERCISE 2

Explain, specifically, why path.join('/') and Prisma's where: { fullPath } lookup could never split a surrogate pair — even for a page whose slug is a rare astral-plane kanji requiring one.

View solution

EXERCISE 3

Run a fresh Prisma migration and inspect the generated SQL migration file. Confirm the CREATE TABLE statement includes DEFAULT CHARACTER SET utf8mb4 automatically, with no manual configuration added anywhere in schema.prisma or the database connection.

View solution

CHAPTER 7 QUICK REFERENCE

- **Routing & rendering** — already solved by Chapter 2's arbitrary depth and Chapter 4's `dangerouslySetInnerHTML`; no Django-style slug validator to work around either
- **UTF-16 code units vs. code points** — common kanji are one unit; rare astral-plane kanji need a surrogate pair (two units)
- **Verified, not assumed** — this course's own routing pipeline never slices a path string by index anywhere, so the surrogate-pair risk never actually triggers here
- `utf8mb4`**, automatic** — Prisma's own migrations set it by default, via table-level `DEFAULT CHARACTER SET` rather than Rails' connection-level default
- **Next chapter:** API Routes & Server Actions

This chapter gives the admin a way to actually change something — `updatePageTitle`, a Server Action, with **no auth check yet, deliberately**: the security gap gets closed properly in Chapter 9. Every one of this course's own five already-complete siblings built the identical feature, the identical gap, closed the identical way one chapter later — a real, intentional continuity thread running through the whole series, not a coincidence of independent design.

Enabling the Cache Components Model

```ts
// next.config.ts
import type { NextConfig } from 'next';

const nextConfig: NextConfig = {
  cacheComponents: true,
};

export default nextConfig;
```

This unlocks `'use cache'`, `cacheTag`, and `updateTag` — the tools this chapter actually needs to give an admin edit a genuine read-your-writes guarantee.

Caching the Page Lookup

```ts
// lib/pages.ts
import { cacheTag } from 'next/cache';
import { prisma } from './prisma';

export async function getPageByPath(fullPath: string) {
  'use cache';
  cacheTag(`page:${fullPath}`);
  return prisma.page.findUnique({ where: { fullPath } });
}
```

Chapter 3's own route now calls `getPageByPath(fullPath)` in place of a direct `prisma.page.findUnique` — the query itself is unchanged, but its result is now cached and taggable.

The Server Action: `updatePageTitle`

```ts
// app/actions.ts
'use server';

import { prisma } from '@/lib/prisma';
import { updateTag } from 'next/cache';

export async function updatePageTitle(pageId: number, newTitle: string) {
  const page = await prisma.page.update({
    where: { id: pageId },
    data: { title: newTitle },
  });

  updateTag(`page:${page.fullPath}`);
}
```

> **NO AUTH CHECK — DELIBERATELY, FOR NOW**
>
> `updatePageTitle` is fully wired up and genuinely works — anyone who can call it can rename any page on the site, right now, with nothing standing in the way. This is left exactly this way on purpose, matching every sibling course's own identical gap, so Chapter 9 has a real, working mutation to actually secure rather than a hypothetical one.

`revalidateTag`'s New Required Argument, vs. `updateTag`

```
// The old single-argument form no longer type-checks in Next.js 16:
// revalidateTag(`page:${page.fullPath}`);  // TypeScript error — missing 2nd
argument
// The current form requires an explicit cacheLife profile:
revalidateTag(`page:${page.fullPath}`, 'max');
```

WHY THIS CHAPTER USES UPDATETAG, NOT REVALIDATETAG

`revalidateTag` marks cached data stale and lets it refresh in the background — the visitor who happens to load the page in the exact next instant might still briefly see the old title. `updateTag` expires and refreshes the cache *within the same request*, so the admin who just renamed a page sees the new title immediately on the very next page they load — genuine read-your-writes semantics, and a materially better fit for this specific "I just edited something, show me the result" scenario than what this chapter's own original version had available.

Built-In CSRF Protection — The One Piece Nobody Else Had to Wire Up

A GENUINE, STRUCTURAL DIFFERENCE FROM EVERY SIBLING

A Server Action is compiled to a unique, unguessable endpoint ID and is only ever invocable via POST — and Next.js automatically compares the request's own `Origin` header against the deployment's allowed origins before running it, rejecting the call outright on a mismatch. Django needed an explicit `{% csrf_token %}` in every form. Laravel needed `@csrf`. Rails needed `protect_from_forgery` plus `form_with`'s own automatic authenticity token. `updatePageTitle` above has no equivalent line anywhere — this specific protection is a structural property of what a Server Action *is*, not something this chapter had to add.

The Same Mutation, Four Structural Answers

	NEXT.JS	DJANGO	LARAVEL	RAILS
Shape	One function — implicit client/server boundary	View function + `ModelForm` + URL — three explicit pieces	Controller method + `FormRequest` — a class-based extra layer	Single Controller action, Strong Parameters inline — the leanest of the four
Auth gap made visible as	A missing check inside the function body	An absent permission check in the view	`authorize()` literally returning `true`	An absent `before_action` filter
CSRF protection	Automatic — built into what a Server Action is	Manual — `{% csrf_token %}`	Manual — `@csrf`	Manual — `protect_from_forgery` + automatic token

Next.js's own implicit boundary is genuinely a double-edged design: it's what makes `updatePageTitle` callable like an ordinary function with zero explicit wiring, and it's also exactly why the missing auth check is easy to miss on a casual read — there's no separate file, no visible `authorize()` method, nothing forcing the question to be asked. Every sibling's own explicit alternative makes the gap structurally harder to overlook, at the cost of more ceremony to write in the first place.

Hands-On Exercises

EXERCISE 1

Build updatePageTitle exactly as written in this chapter, call it against a real page with no authentication in place, and confirm the title actually changes — demonstrating the security gap is real and currently exploitable, not hypothetical.

View solution

EXERCISE 2

Explain why revalidateTag now requires a second argument in Next.js 16, and why this chapter chose updateTag instead of revalidateTag for updatePageTitle specifically — tying the answer to what "read-your-writes" actually means for an admin who just made an edit.

View solution

EXERCISE 3

Explain, in your own words, how a Server Action's built-in Origin-header check achieves the same protective goal as Django's {% csrf_token %}, without any code in updatePageTitle resembling a token at all.

View solution

CHAPTER 8 QUICK REFERENCE

- `updatePageTitle` — a working Server Action, deliberately missing its auth check until Chapter 9
- `cacheComponents: true` + `'use cache'` + `cacheTag` — the current model for taggable, invalidatable server-side caching
- `updateTag` **over** `revalidateTag` — genuine read-your-writes for an admin who just made an edit
- **Automatic CSRF protection** — a structural property of Server Actions; the one piece every sibling course had to wire up manually
- **Next chapter:** Admin Authentication

Chapter 8 left `updatePageTitle` genuinely callable by anyone, deliberately. This chapter closes that gap for real — and, just as importantly, migrates the legacy PHP site's own existing admin credential rather than forcing a reset.

The Judgment Call: Auth.js, Not Better Auth

NAMED DIRECTLY, NOT SILENTLY DECIDED

As of this course's own version-currency research (Chapter 1), NextAuth.js — rebranded **Auth.js** — is in maintenance mode, and the official documentation now points new projects at **Better Auth** instead. This chapter stays on Auth.js anyway: five of this course's own already-complete siblings (Django, Laravel, Rails, Astro, Express) each directly quote specific implementation details from this exact chapter — the `authorize()` callback shape, the `bcryptjs` comparison, the JWT-vs-session distinction. Switching to Better Auth here wouldn't just be a fresh choice; it would quietly invalidate five already-published cross-references. Series-wide consistency wins this specific tradeoff, named honestly rather than assumed.

Migrating the Legacy Admin Credential

```
// prisma/schema.prisma — new model
model Admin {
  id           Int    @id @default(autoincrement())
  email        String @unique
  passwordHash String
}
```

The legacy PHP site's own admin credential — a bcrypt hash generated by PHP's `password_hash()`, tagged `$2y$` — is copied directly into `passwordHash` during migration. No password reset, no re-entry, no coordination with the site's own admin required.

Installing and Configuring NextAuth

```
npm install next-auth@beta bcryptjs
```

```
// auth.ts
import NextAuth from 'next-auth';
import Credentials from 'next-auth/providers/credentials';
import bcrypt from 'bcryptjs';
import { prisma } from '@/lib/prisma';

export const { handlers, auth, signIn, signOut } = NextAuth({
  providers: [
    Credentials({
      credentials: { email: {}, password: {} },
      authorize: async (credentials) => {
        const admin = await prisma.admin.findUnique({
          where: { email: credentials.email as string },
        });
        if (!admin) return null;

        const valid = await bcrypt.compare(
          credentials.password as string,
          admin.passwordHash
        );
        if (!valid) return null;

        return { id: admin.id.toString(), email: admin.email };
      },
    }),
  ],
});
```

```
// app/api/auth/[...nextauth]/route.ts
import { handlers } from '@/auth';
export const { GET, POST } = handlers;
```

`bcryptjs` Verifies the Legacy Hash — No Special Configuration

A REAL INTEROPERABILITY FACT, NOT AN ASSUMPTION

`bcrypt.compare()` above correctly verifies the legacy `$2y$`-tagged hash with zero special handling — the bcrypt algorithm itself is standardized enough across implementations that a hash generated by PHP's `password_hash()` verifies correctly through the JavaScript `bcryptjs` package, unchanged. This same confirmation is repeated twice more later in this series, by Astro Rebuild's own Chapter 9 and Express Rebuild's own Chapter 9 — both using this exact same `bcryptjs` package, since all three share the same Node.js/npm ecosystem.

Closing Chapter 8's Gap

```
// app/actions.ts — updated from Chapter 8
'use server';

import { prisma } from '@/lib/prisma';
import { updateTag } from 'next/cache';
import { auth } from '@/auth';

export async function updatePageTitle(pageId: number, newTitle: string) {
  const session = await auth();
  if (!session) {
    throw new Error('Unauthorized');
  }

  const page = await prisma.page.update({
    where: { id: pageId },
    data: { title: newTitle },
  });

  updateTag(`page:${page.fullPath}`);
}
```

One new line at the very top of the function — the exploit demonstrated back in Chapter 8's own Exercise 1 no longer works.

JWT Sessions: A Real Structural Tradeoff

A CREDENTIALS PROVIDER FORCES THIS — IT ISN'T A FREE CHOICE

NextAuth's `Credentials` provider only supports the **JWT** session strategy — a signed token stored in a cookie, not a server-side session record in the database. This means there's no single row to delete for an instant, guaranteed revocation; a compromised token stays technically valid until its own expiry, unless a deliberate token-blacklist strategy is added on top. Laravel Rebuild's own Chapter 9 names this directly as a real point of contrast: its own `session()->invalidate()` and Django's own `logout()` both close a session in one server-side call, something this chapter's own JWT-based approach genuinely can't do as simply.

Password Hashing, Compared Across the Series

	NEXT.JS	DJANGO	LARAVEL	RAILS	ASTRO	E
Verifying the legacy hash	Bespoke `bcryptjs.compare()`	Reordered `PASSWORD_HASHERS`	Zero config — default hasher already bcrypt	`bcrypt` gem, a real `$2a$` -vs- `$2y$` nuance	Same `bcryptjs` package, unchanged	S p t
Session model	JWT — no server-side record	Server-side session	Server-side session	Server-side session	JWT — same as Next.js	S s (s
Revoking a session early	Needs a deliberate blacklist strategy	One server-side call	One server-side call	One server-side call	Needs the same blacklist strategy	C s s

Hands-On Exercises

EXERCISE 1

Configure the Admin model, the Credentials provider, and the API route from this chapter, then confirm signing in with the legacy $2y$-tagged bcrypt hash succeeds with no special configuration for the hash format.

View solution

EXERCISE 2

Add the auth() check to updatePageTitle, then repeat Chapter 8's own Exercise 1 exploit attempt with no session present, and confirm it now fails with the Unauthorized error.

View solution

EXERCISE 3

Explain, in your own words, why this chapter's own JWT-based session can't be revoked with a single server-side call the way Django's logout() or Laravel's session()->invalidate() can — and what would need to be built to close that gap.

View solution

CHAPTER 9 QUICK REFERENCE

- **Auth.js over Better Auth** — a named tradeoff, chosen for consistency with five already-published sibling chapters
- **Legacy credential migrated, not reset** — the existing `$2y$`-tagged bcrypt hash, copied as-is into a new `Admin` model
- `bcryptjs.compare()` — verifies the PHP-generated hash correctly, no special configuration
- `auth()` **in** `updatePageTitle` — Chapter 8's own gap, closed in one added check
- **JWT sessions** — forced by the Credentials provider; no single-call revocation, unlike every server-side-session sibling
- **Next chapter:** Admin CRUD Interface

This chapter builds the real admin interface by hand — Next.js ships no automatic scaffolding of any kind — and finally pays off Chapter 2's own deferred cost: recalculating an entire subtree's own `fullPath` when a page is moved.

No Free Admin Panel

	NEXT.JS	DJANGO	LARAVEL	RAILS	ASTRO
Free admin/scaffold?	No — hand-built	Yes — `admin.site.register()`, live	No by default (Nova/Filament optional)	Partial — `rails generate scaffold`, one-time codegen	No — hand-built

Django's own `admin` app is a live, runtime-introspected interface generated from the model definition itself — genuinely different in kind from Rails' scaffold generator, which produces real, editable source files once and then stops helping. Next.js has neither. Every line of this chapter's own interface is written directly.

The Parent Picker: A Searchable Flat List

THE DESIGN DECISION EVERY SIBLING COURSE EXPLICITLY CREDITS TO THIS CHAPTER

Choosing a new parent for a page — at arbitrary depth, from potentially hundreds of candidates — could be a nested tree widget, expandable level by level, or a plain flat list of every other page's own `fullPath`, filterable as the admin types. This chapter uses the flat list: at this site's realistic page count, typing a few letters to jump straight to the right page beats clicking through several levels of a tree, and a flat list is genuinely simpler to build in the first place. Laravel Rebuild's, Astro Rebuild's, and Rails Rebuild's own Chapter 10s all reuse this identical reasoning, citing it directly as "established since the Next.js rebuild's own Chapter 10."

```
// components/ParentPicker.tsx
'use client';

import { useState } from 'react';
import type { Page } from '@prisma/client';

export default function ParentPicker({ pages, onSelect }: { pages: Page[]; onSelect:
(id: number) => void }) {
  const [query, setQuery] = useState('');
  const filtered = pages.filter((p) =>
    p.fullPath.toLowerCase().includes(query.toLowerCase())
  );

  return (
    <div>
<input
        value={query}
        onChange={(e) => setQuery(e.target.value)}
        placeholder="Search by path…"
/>
<ul>
        {filtered.map((p) => (
          <li key={p.id} onClick={() => onSelect(p.id)}>{p.fullPath}</li>
        ))}
      </ul>
</div>
  );
}
```

Reparenting: Paying Off Chapter 2's Own Deferred Cost

```
// lib/movePage.ts
import { prisma } from './prisma';

type DescendantRow = { id: number; parentId: number; slug: string; depth: number };

export async function movePage(pageId: number, newParentId: number | null) {
  const page = await prisma.page.findUniqueOrThrow({ where: { id: pageId } });
  const newParent = newParentId
    ? await prisma.page.findUniqueOrThrow({ where: { id: newParentId } })
    : null;
  const newFullPath = newParent ? `${newParent.fullPath}/${page.slug}` : page.slug;

  // One query, via a recursive CTE, finds every affected descendant —
// ordered by depth, so each row's own parent is always processed first
const descendants = await prisma.$queryRaw<DescendantRow[]>`
    WITH RECURSIVE descendants AS (
      SELECT id, parentId, slug, 0 AS depth FROM Page WHERE id = ${pageId}
      UNION ALL
      SELECT p.id, p.parentId, p.slug, d.depth + 1
      FROM Page p INNER JOIN descendants d ON p.parentId = d.id
    )
    SELECT * FROM descendants WHERE depth > 0 ORDER BY depth ASC
  `;

  await prisma.page.update({ where: { id: pageId }, data: { parentId: newParentId,
fullPath: newFullPath } });

  const recalculated = new Map<number, string>([[pageId, newFullPath]]);
  for (const row of descendants) {
    const parentPath = recalculated.get(row.parentId)!;
    const updatedPath = `${parentPath}/${row.slug}`;
    recalculated.set(row.id, updatedPath);
    await prisma.page.update({ where: { id: row.id }, data: { fullPath: updatedPath }
});
  }
}
```

A GENUINE IMPROVEMENT — HONESTLY, ONLY A PARTIAL ONE

Discovering every affected descendant takes exactly one query, via `$queryRaw`'s recursive CTE — a real improvement over walking the tree level by level in application code, since Chapter 6's own `include` is fixed-depth and couldn't express "every descendant, however deep" at all. The *write* step stays honestly sequential regardless: each descendant's own new `fullPath` depends on its own parent's already-updated path, so the loop above genuinely can't be parallelized. The improvement is real, and it's real specifically at the discovery step, not the whole operation.

WHY AN ESCAPE HATCH, NOT THE QUERY BUILDER

Prisma's own query builder — `include`, `where`, and the rest — has no way to express a recursive query at all; recursion isn't part of its vocabulary. `$queryRaw` is Prisma's own documented escape hatch for exactly this situation: real SQL, still returned as typed rows, for the specific cases the query builder genuinely can't reach.

Delete Refusal: From Raw Exception to a Friendly Message

```
// app/actions.ts — new action
import { Prisma } from '@prisma/client';

export async function deletePage(pageId: number) {
  const session = await auth();
  if (!session) throw new Error('Unauthorized');

  try {
    await prisma.page.delete({ where: { id: pageId } });
  } catch (error) {
    if (
      error instanceof Prisma.PrismaClientKnownRequestError &&
      error.code === 'P2003'
    ) {
      throw new Error('This page still has child pages — move or delete them
first.');
    }
    throw error;
  }
}
```

SAME LAYER AS LARAVEL, ASTRO, AND EXPRESS — THE SHARED TRY/CATCH CAMP

`P2003` is Prisma's own documented error code for a foreign key constraint violation — the raw database refusal from Chapter 2's own `onDelete: Restrict`, surfacing here as a real exception that has to be caught, not a clean return value. This matches Laravel's `try/catch` around a raw `QueryException` and Astro's own equivalent, since all three sit on a database-level constraint. Django's `PROTECT` and Rails' `dependent: :restrict_with_error` are both application-level instead — Django still raises an exception to catch, but Rails' own `destroy` simply returns `false` with no exception-catching needed at all, a genuine ergonomic advantage specific to Rails' own API design, not a universal trait of every application-level approach.

Hands-On Exercises

EXERCISE 1

Build ParentPicker and confirm typing a partial path (e.g. "prog") correctly filters the list down to only matching pages, client-side, with no new server request per keystroke.

View solution

EXERCISE 2

Move a page with at least two real levels of descendants to a new parent, and confirm every descendant's own fullPath is correctly recalculated — not just the moved page itself.

View solution

EXERCISE 3

Attempt to delete a page that still has at least one child, and confirm the friendly error message appears rather than an unhandled Prisma exception crashing the request.

View solution

CHAPTER 10 QUICK REFERENCE

- **No free admin panel** — hand-built, matching Laravel/Astro/Express's own shared finding
- **Searchable flat-list parent picker** — the design decision every sibling course credits to this chapter
- `movePage()` **/ recursive CTE via** `$queryRaw` — one query discovers every descendant; the write step stays honestly sequential
- `P2003` — Prisma's own foreign key error code, caught and converted into a real UI message
- **Next chapter:** Deployment

This chapter's own deployment story is genuinely different from three of its five already-complete siblings — Laravel, Rails, and Astro all found a way to reuse the site's own already-live Apache. This course stands up entirely new infrastructure instead, the same real situation Django Rebuild 11 independently faced.

Two Real Paths: Vercel or Self-Hosted

NAMED HONESTLY, NOT SILENTLY CHOSEN

Vercel is built by the Next.js team, specifically for Next.js — `vercel deploy` gets a working production URL with essentially zero configuration, precisely because the platform and the framework share an owner. That's the genuinely easiest real path for a real Next.js project. This chapter goes self-hosted instead, deliberately: it's the only way to give this course a genuine, comparable production deployment story against its own siblings, on the same kind of server the real site this whole series rebuilds actually runs on.

Building for Production

```
npm run build
# next build — Turbopack by default, per Chapter 1
```

`next build` produces the production bundle; `next start` serves it. Neither of these alone stays running after the terminal session ends — that's the next problem.

PM2: Keeping the Server Running

```
// ecosystem.config.js
module.exports = {
  apps: [{
    name: 'website-rebuild-with-nextjs',
    script: 'node_modules/.bin/next',
    args: 'start',
    cwd: '/var/www/website-rebuild-with-nextjs',
    env: { NODE_ENV: 'production', PORT: 3000 },
  }],
};
```

```
pm2 start ecosystem.config.js
pm2 save
pm2 startup
```

`pm2 startup` registers PM2 itself to relaunch on server reboot — without it, a reboot would silently take the whole site down until someone noticed.

> **THE SAME REAL JOB AS DJANGO'S OWN GUNICORN**
>
> PM2 exists to solve the identical problem Django Rebuild 11's own `gunicorn` needed a process supervisor for: a production server process that must keep running unattended, restart itself if it crashes, and survive a reboot. Different runtime, same real requirement.

nginx: A Fresh Reverse Proxy

STOOD UP FROM NOTHING — NOT REUSED

The site's own existing Apache setup has a real, working path for PHP (via `mod_php` or PHP-FPM) — but nothing for proxying to a plain Node process on a local port. Rather than bolt an unfamiliar module onto Apache, this chapter stands up a fresh nginx instance instead, exactly the same call Django Rebuild 11 made for the identical reason.

```
# /etc/nginx/sites-available/nextjs-rebuild
server {
    listen 80;
    server_name nextjs-demo.example.com;

    location / {
        proxy_pass http://localhost:3000;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection 'upgrade';
        proxy_set_header Host $host;
        proxy_cache_bypass $http_upgrade;
    }
}
```

The `Upgrade` / `Connection` headers matter specifically because Next.js's own dev-time hot-reload and certain runtime features rely on WebSocket upgrades — worth proxying correctly even in production, not just for local development.

`certbot`: TLS for the New Proxy

```
sudo certbot --nginx -d nextjs-demo.example.com
```

`certbot`'s own nginx plugin edits the site's config in place, adding the certificate paths and a redirect from port 80 to 443 automatically — the same Let's Encrypt mechanism behind every genuinely free HTTPS setup on a self-hosted server.

Secrets: A Plain `.env`

```
# .env — never committed
DATABASE_URL="mysql://user:password@localhost:3306/nextjs_rebuild"
AUTH_SECRET="a long, random, generated value"
```

`AUTH_SECRET`, THE DIRECT EQUIVALENT OF A NAME EVERY SIBLING COURSE ALREADY HAS

`AUTH_SECRET` is what Auth.js uses to sign the JWT sessions Chapter 9 introduced — the exact same job Django's `SECRET_KEY`, Laravel's `APP_KEY`, and Rails' `master.key` each do in their own frameworks. A plain `.env` file is the simplest secrets story of the whole series — matching Django and Laravel's own convention exactly, not Rails' own reversed, encrypted-commit-to-the-repo approach.

Deployment, Compared Across the Series

	NEXT.JS	DJANGO	LARAVEL	RAILS	ASTRO	EX
Approach	Stood up fresh — PM2 + nginx + certbot	Stood up fresh — gunicorn + nginx	Modernize in place — Apache `mod_proxy_fcgi`	Modernize in place — Apache `mod_passenger`	Modernize in place — Apache `mod_proxy_http`	St — ov `r` ur
Secrets	Plain `.env`	Plain `.env`	Plain `.env`	Encrypted, committed (`master.key`)	Plain `.env`	Pl `c` ex
First-party zero-config platform?	Yes — Vercel (not used here, named honestly)	No	Paid, AWS-specific (Vapor)	No	No	N

Hands-On Exercises

EXERCISE 1

Run next build, start the result with PM2 using this chapter's own ecosystem.config.js, and confirm pm2 status shows the process running and pm2 startup is registered.

View solution

EXERCISE 2

Configure the nginx reverse proxy from this chapter, run certbot against it, and confirm the site is reachable over HTTPS with a valid certificate — not just HTTP on port 3000 directly.

View solution

EXERCISE 3

Explain why this course's own deployment chapter had to stand up nginx from nothing, while Laravel Rebuild's, Rails Rebuild's, and Astro Rebuild's own Chapter 11s could all reuse the site's already-live Apache instead.

View solution

CHAPTER 11 QUICK REFERENCE

- **Vercel or self-hosted** — Vercel is genuinely easier; this chapter goes self-hosted for a comparable production story across the series
- **PM2** — keeps `next start` running unattended and across reboots, the same real job as Django's `gunicorn`
- **nginx, stood up fresh** — the existing Apache has no path for a plain Node process; certbot handles TLS
- `AUTH_SECRET` in a plain `.env` — the direct equivalent of Django's `SECRET_KEY`/Laravel's `APP_KEY`/Rails' `master.key`
- **Next chapter:** Capstone: A Complete, Working Flexible-Routing Site

Sam — the site's own admin — logs in and does one real, ordinary session's worth of work. Every step below draws on a specific earlier chapter of this course, closing with a chapter-attribution table and an honest scope note. Sam is the same recurring persona every one of this course's own five already-complete siblings reuses for its own capstone — this telling of the story is, chronologically, the most recently written of the six, even though it's the very first in the series' own numbering.

Step 1 — Opening the Site

Sam visits the site's own domain. nginx terminates HTTPS (Chapter 11's own `certbot` certificate) and proxies to the Next.js production server, kept running unattended under PM2.

Step 2 — Logging In

Sam enters the same admin credential the old PHP site always used. NextAuth's `authorize()` callback runs, and `bcrypt.compare()` verifies it against the stored `$2y$`-tagged hash with no special configuration — nothing was ever reset. A signed JWT session is established.

Step 3 — Building the Chain, Live

Through the hand-built admin interface — no free admin panel, no scaffold generator — Sam creates five real pages, one level at a time:

1. `programming` (top-level)
2. `general-purpose-languages` (under `programming`)
3. `java` (under `general-purpose-languages`)
4. `fundamentals` (under `java`)
5. `chapter-1` (under `fundamentals`)

Five real levels deep — no longer the illustrative example this course opened with in Chapter 1.

Step 4 — Visiting It Live

Sam opens `/programming/general-purpose-languages/java/fundamentals/chapter-1` directly. The `[...path]` catch-all route joins the five URL segments into one string and resolves the page in a single `fullPath` lookup — no five-level chase. `PageShell` renders the breadcrumb through Chapter 6's own `include`-based fix, one query for the whole ancestor chain rather than five separate ones, and the dark theme applies through the single global stylesheet import.

Step 5 — The Kanji Check

Sam creates one more page — `水` — under `programming`. It saves, routes, and renders through the exact same pipeline as every other page: no special-casing, no depth exception, no separate rendering path. Chapter 7's own verified finding holds: nothing in this pipeline ever slices a path string by index, so it never mattered whether the character needed one UTF-16 code unit or a surrogate pair.

Step 6 — A Safe Edit

Sam edits `chapter-1`'s own title through the now-protected `updatePageTitle` — Chapter 8's mutation, Chapter 9's `auth()` check standing in front of it. Loading the page again immediately afterward shows the new title — Chapter 8's own `updateTag` call guarantees this, not just usually true.

Step 7 — Reorganizing: Paying Chapter 2's Own Deferred Cost

Sam decides `java` belongs under a new `archive` section instead. Through the searchable flat-list parent picker, Sam reparents it. `movePage`'s own recursive CTE finds both real descendants — `fundamentals` and `chapter-1` — in one query, and each one's own `fullPath` is recomputed correctly, in dependency order. The cost Chapter 2 named and deliberately deferred gets paid here, for real, against a real five-level chain — not a toy example.

Step 8 — The Delete Refusal

Sam tries to delete `fundamentals`, forgetting it still has `chapter-1` beneath it. The database's own `onDelete: Restrict` constraint refuses the raw delete; Prisma surfaces it as `P2003`; Chapter 10's own `try`/`catch` converts it into a real, readable message — not a crashed request.

Chapter Attribution

STEP	CHAPTER(S) APPLIED
1 — Opening the site	Chapter 11 (deployment)
2 — Logging in	Chapter 9 (legacy-credential-compatible auth)
3 — Building the chain	Chapter 2 (the self-referencing schema), Chapter 10 (the admin interface)
4 — Visiting it live	Chapter 3 (routing), Chapter 6 (the eager-loading fix), Chapter 4/5 (layout, styling)
5 — Kanji check	Chapter 7 (the edge case resolution)
6 — Safe edit	Chapter 8 (the mutation, `updateTag`), Chapter 9 (the fix that protects it)
7 — Reorganizing	Chapter 2 (the deferred cost), Chapter 10 (`movePage`, the real payoff)
8 — Delete refusal	Chapter 2 (`onDelete: Restrict`), Chapter 10 (`P2003`, caught)

WHAT THIS WHOLE COURSE WAS REALLY ABOUT

Every step in Sam's session traces back to a specific, real decision made earlier — nothing here is arbitrary or introduced just for the capstone. Chapter 2's own read-heavy/write-rare reasoning shows up directly in Chapter 3's one-lookup routing. Chapter 2's own deferred cascade cost, named honestly rather than quietly ignored, becomes Chapter 10's own working `movePage`. Chapter 1's own "bring your own backend" observation becomes Chapter 9's real, hand-configured login. This capstone isn't a new demonstration — it's every earlier chapter's own decision, paying off exactly where it was always going to.

HONEST SCOPE NOTE

This capstone deliberately stays within a single-admin, single-server scope. It doesn't cover multi-admin roles or fine-grained permissions beyond one credential, revision history or an undo mechanism for content edits, media/image uploads, an automated test suite, internationalization beyond kanji's own routing/rendering support, or real search ranking beyond a plain path/title match if one were ever added. Those are each genuinely separate topics, not gaps in what this course could responsibly cover in twelve chapters.

Hands-On Exercises

EXERCISE 1

Explain why visiting the five-level-deep chapter URL in Step 4 only requires one database lookup for the page itself, rather than five, tracing the answer back to a specific design decision made in Chapter 2.

View solution

EXERCISE 2

Walk through Step 7's own reorganization and explain exactly which rows movePage's recursive CTE finds, and in what order they get updated, and why that order matters.

View solution

EXERCISE 3

Pick any one row from the chapter-attribution table and explain, in your own words, why that step genuinely couldn't have worked correctly without the specific chapter(s) it's attributed to.

View solution

CHAPTER 12 QUICK REFERENCE

- **8 real steps, 11 prior chapters** — one realistic admin session, building the exact chain this course opened with, no longer hypothetical
- **This course's own throughline, closed out:** every step traces to a real, specific decision made earlier — the deferred cascade cost, the array-vs-string routing gotcha, the deliberate auth gap — nothing arbitrary
- **Honest scope note:** single-admin, no revision history, no media uploads, no automated tests, no i18n beyond kanji, no real search ranking
- **Website Rebuild with Next.js is now complete** — 12/12 chapters