PROJECTS

Website Rebuild with Laravel

Modernizing the Live PHP/Apache Stack in Place

A Flexible, Self-Referencing URL Model

Blade, Eloquent & Laravel's Own Explicit Routing

Zero-Configuration Admin Authentication

Capstone: A Complete, Working Flexible-Routing Site

Philip Osztromok

Generated with Claude

Table of Contents

Website Rebuild with Laravel — 12 Chapters

Website Rebuild with Next.js and Website Rebuild with Django both had to be honest about something: neither one was evolving the site's own actual, currently-live stack — both meant replacing PHP/Apache with something else entirely, language and all. This course doesn't have to make that same trade. The live site is already PHP. Laravel isn't a replacement for what's running today; it's what that same PHP could become if it grew up into a real, modern framework — the same arbitrary-depth routing requirement, solved in the language already sitting on the server.

`composer create-project`: Laravel's Own Install Step

```
composer create-project laravel/laravel osztromok-site
cd osztromok-site
php artisan serve
```

Composer is PHP's own package manager — the direct equivalent of `npm` (Next.js) or `pip` (Django) — and `artisan` is Laravel's own command-line tool, the same job `manage.py` does for Django. `php artisan serve` starts a local dev server exactly the way `runserver` or `next dev` would.

The Generated Structure: One App, Not a Project-Plus-App Split

```
osztromok-site/
├── app/
│   ├── Http/
│   │   └── Controllers/    # request-handling logic
│   └── Models/              # Eloquent models — Chapter 2's own Page model lands
here
├── routes/
│   └── web.php              # the project's own route definitions — Chapter 3's own
territory
├── resources/
│   └── views/               # Blade templates — Chapter 4
├── database/
│   └── migrations/
├── .env
└── artisan
```

A THREE-WAY STRUCTURAL DIFFERENCE, NOT JUST TWO

Django Rebuild 1 named its own project-vs-app split as a structural distinction Next.js has no equivalent for. Laravel doesn't have that split either — like Next.js, there's just *one* application, one directory tree, no separate outer "project" container wrapping independent "apps" inside it. What Laravel *does* keep from Django's own philosophy is heavy, deliberate internal organization by convention — `app/Models`, `app/Http/Controllers`, `routes/web.php`, `resources/views` are all fixed, meaningful locations, not just folders you happened to create. It's a genuine third shape: no project/app split like Next.js, but far more prescriptive internal structure than Next.js imposes on its own single app.

Batteries-Included, Like Django — Not Like Next.js

Laravel ships an ORM (Eloquent), a templating engine (Blade), routing, validation, and authentication scaffolding all together, out of the box — much closer to Django's own "one framework, everything included" philosophy than to Next.js's more compositional style, which needed Prisma and NextAuth.js added as separate, deliberate choices. Chapter 9's own admin authentication work leans directly on this — no separate package required there either, echoing Django Rebuild 9's own advantage for a genuinely similar reason.

`.env` & `APP_KEY`: A Forward Reference

```
# .env
APP_NAME=Osztromok
APP_ENV=local
APP_KEY=                    # empty until generated
DB_CONNECTION=mysql
DB_DATABASE=osztromok_site

php artisan key:generate
# APP_KEY=base64:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx=
```

`php artisan key:generate` creates a real, random `APP_KEY` — Laravel's own direct equivalent of Django's `SECRET_KEY`, used to sign sessions and encrypted data. It matters now for exactly the same reason Django Rebuild 1 flagged `INSTALLED_APPS` ahead of its own Chapter 9: nothing about it needs solving yet, but Chapter 11's own deployment work will come back to it directly.

Where This Course Is Headed

Designing the same flexible, arbitrary-depth content model as an Eloquent self-relationship, Laravel's own routing mechanism for catching a deep path, Blade templates, Vite-based styling, the database via Eloquent (including the same N+1 query problem, solved Laravel's own way), rendering content and the kanji edge case (with a genuinely new Laravel-specific wrinkle), dynamic content and forms through Controllers, admin authentication (this course's own standout "modernize in place" payoff), a hand-built admin CRUD interface, deployment onto the site's own already-live Apache, and a capstone building a genuinely five-level-deep chain live.

Hands-On Exercises

EXERCISE 1

Explain why this course can honestly frame itself as "evolving the existing stack" in a way the Next.js and Django rebuilds genuinely couldn't, referencing what the current live site is already built in.

View solution

EXERCISE 2

Explain why Laravel has no equivalent to Django's own project-vs-app split, even though Laravel is otherwise much closer to Django's "batteries-included" philosophy than to Next.js's more minimal one.

View solution

EXERCISE 3

Explain what `php artisan key:generate` actually does, and why this chapter treats it as a forward reference rather than something needing to be solved right now.

View solution

CHAPTER 1 QUICK REFERENCE

- **This course's own real advantage:** the live site is already PHP — genuinely evolving the existing stack, not replacing it
- `composer create-project laravel/laravel` — Laravel's install step; `artisan` — its own CLI, the same job as `manage.py`
- **No project-vs-app split** — one application, one directory tree, like Next.js — but far more prescriptive internal structure (`app/Models`, `app/Http/Controllers`, `routes/`, `resources/views`)
- **Batteries-included** — Eloquent, Blade, routing, validation, auth all ship together, closer to Django's philosophy than Next.js's own compositional style
- `APP_KEY` / `php artisan key:generate` — Laravel's own direct equivalent of Django's `SECRET_KEY`; a forward reference to Chapter 11
- **Next chapter:** Designing a Flexible URL & Content Model

This is the third time this exact design question gets asked on this site — Website Rebuild with Next.js answered it with Prisma, Website Rebuild with Django answered it with the ORM, and both landed on the identical hybrid: a real adjacency list for structure, plus a stored materialized path for fast reads. That two independent frameworks reached the same answer wasn't a coincidence of either framework's own bias — it's a property of this specific site's own read-heavy, write-rare traffic. Watch Laravel arrive at the same place a third time, independently, for the same reason.

Where Does the Schema Actually Live? A Real Difference

Django's `Page` model declares its own fields directly in the class — `title = models.CharField(...)` — and the migration is auto-generated *from* that class. Prisma works the same way: `schema.prisma` declares the fields, and a migration is generated from the schema file. Laravel inverts this entirely.

The Migration: Hand-Written, and the Real Source of Truth

```
php artisan make:model Page -m

// database/migrations/xxxx_create_pages_table.php
Schema::create('pages', function (Blueprint $table) {
    $table->id();
    $table->string('title');
    $table->string('slug');
    $table->foreignId('parent_id')->nullable()->constrained('pages')-
>restrictOnDelete();
    $table->string('full_path')->unique();
    $table->text('body')->nullable();
    $table->timestamps();
});
```

The `-m` flag generates a migration alongside the model in one step. `->restrictOnDelete()` is Laravel's own direct parallel to Django Rebuild 2's own `on_delete=PROTECT` decision — explicitly refusing a deletion while referenced rows still exist, made explicit here on purpose rather than left to the database's own implicit default behavior.

> **THE CENTRAL FACT THIS CHAPTER IS BUILT ON**
>
> In Django and Prisma, the model/schema file *is* the real definition — read it, and you know every column. In Laravel, the migration is the real schema, hand-written, and the Eloquent model class barely mentions its own columns at all. Asking "what fields does this table actually have?" means reading a completely different file depending on the framework — a genuine, practical habit to adjust, not just a syntax difference.

The Eloquent Model: Two Relationships, Not One Field

```php
// app/Models/Page.php
class Page extends Model
{
    protected $fillable = ['title', 'slug', 'parent_id', 'full_path', 'body'];

    public function parent()
    {
        return $this->belongsTo(Page::class, 'parent_id');
    }

    public function children()
    {
        return $this->hasMany(Page::class, 'parent_id');
    }
}
```

Django's single `ForeignKey('self', related_name='children')` field declaration gave both directions — `page.parent` and `page.children.all()` — from one line. Eloquent needs two separate, explicitly written relationship methods to get the identical two-directional access: `parent()` reads upward, `children()` reads downward, and neither one implies the other automatically.

Recomputing `full_path`: A Model Event, Not an Overridden `save()`

```php
protected static function booted()
{
    static::saving(function (Page $page) {
        $page->full_path = $page->parent
            ? $page->parent->full_path . '/' . $page->slug
            : $page->slug;
    });
}
```

Django recomputes `full_path` by directly overriding `save()` — an imperative method replacement. Laravel reaches the identical practical result through a genuinely different mechanism: **model events**. `static::saving(...)` registers a closure to run automatically every time a `Page` is about to be saved, without touching `save()` itself at all. Eloquent fires a whole family of these (`creating`, `created`, `saving`, `saved`, `deleting`, and more) — an event-listener pattern, not a method override.

$FILLABLE: MASS-ASSIGNMENT PROTECTION

Only fields listed in `$fillable` can be set through `Page::create([...])` or `$page->fill([...])` — a genuinely Laravel-specific convention with no direct one-line equivalent in Django or Prisma, guarding against a real class of bug where unexpected form input silently overwrites a column nobody meant to expose.

Hands-On Exercises

EXERCISE 1

Explain the real difference in "where the schema lives" between Django's `Page` model, Prisma's `schema.prisma`, and Laravel's own migration-plus-Eloquent-model pair.

View solution

EXERCISE 2

Explain why `Page.php` needs two separate relationship methods, `parent()` and `children()`, to get the same two-directional access Django's single `ForeignKey('self', related_name='children')` field provided from one declaration.

View solution

EXERCISE 3

Explain why `full_path` is recomputed inside a `static::saving()` closure rather than by overriding a `save()` method directly, and what "model event" means in this context.

View solution

CHAPTER 2 QUICK REFERENCE

- **The same hybrid, a third time** — adjacency list plus materialized path, independently reached by Prisma, Django, and now Eloquent
- **The schema lives in the migration**, not the Eloquent model class — a real, practical difference from Django and Prisma
- `php artisan make:model Page -m` — generates a model and its migration together
- `->restrictOnDelete()` — Laravel's own explicit parallel to Django's `on_delete=PROTECT`
- `parent()` / `children()` — two explicit relationship methods, replacing Django's single self-referencing field
- **Model events (`static::saving()`)** — Eloquent's own mechanism for "run this before every save," genuinely different from overriding `save()`
- `$fillable` — Laravel's own mass-assignment protection, with no direct Django/Prisma equivalent
- **Next chapter:** Routing: Laravel's Own Mechanism for Arbitrary-Depth Routing

Three courses, three genuinely different technical answers to the identical problem. Next.js repurposed the file system itself as the routing table. Django added a dedicated, purpose-built `path` converter type to its route vocabulary. Laravel does neither — it takes the same generic regex-constraint mechanism used to validate *any* route parameter, and simply points it at a permissive pattern.

Route Parameters Don't Match Slashes By Default, Either

Laravel's ordinary `{path}` syntax behaves exactly like Django's default converters — it stops at the first `/`. There's no separate "path" type to reach for instead. The fix is Laravel's own general-purpose `->where()` constraint, normally used for things like requiring a parameter to be numeric, applied here with a pattern permissive enough to match anything:

```
// routes/web.php
Route::get('/{path?}', [PageController::class, 'show'])
    ->where('path', '.*');
```

THE CENTRAL FACT THIS CHAPTER IS BUILT ON

Next.js repurposes the file system as the routing table. Django introduces a genuinely new concept — a dedicated `path` converter type, selected instead of `str` / `slug` / `int`. Laravel introduces nothing new at all: `->where()` already existed for ordinary parameter validation, and matching a deep path is just that same existing, general-purpose mechanism pointed at `'.*'` instead of something like `'[0-9]+'`. Three real mechanisms, three different amounts of new machinery required.

A Genuine Elegance: One Route for Home *and* Everything Else

The `?` in `{path?}` makes the parameter optional — visiting the bare root URL resolves it to `null` rather than failing to match at all, and Laravel doesn't even attempt the `where()` regex check when the segment is absent. Django's own `<path:full_path>` converter structurally requires at least one character, which is exactly why Django Rebuild 3 needed a second, separate `path('', ...)` entry just for the homepage. Laravel's optional parameter avoids that split entirely — one route, both jobs.

The Controller

```php
// app/Http/Controllers/PageController.php
class PageController extends Controller
{
    public function show(?string $path = null)
    {
        if (!$path) {
            return view('pages.home');
        }

        $page = Page::where('full_path', trim($path, '/'))->firstOrFail();
        return view('pages.show', ['page' => $page]);
    }
}
```

`trim($path, '/')` is PHP's own version of Chapter 2 of the Django course's `rstrip('/')` normalization — `trim()` strips both leading and trailing slashes, a slightly broader defense than `rstrip()` alone offered. `firstOrFail()` is Eloquent's direct answer to Django's own `get_object_or_404` — a missing page throws a `ModelNotFoundException` that Laravel's own exception handler automatically turns into a real 404 response, without a manual check anywhere in this method.

Two Real Production Gotchas

ROUTE:CACHE REJECTS CLOSURE-BASED ROUTES OUTRIGHT

`php artisan route:cache` pre-compiles every route into a single cached file for a real production performance boost — but it flatly refuses to cache any route defined with an inline closure instead of a Controller reference; the command errors out rather than silently skipping it. This is exactly why the route above already uses `[PageController::class, 'show']` rather than a closure — writing it any other way would work perfectly in development and then break route caching the moment it mattered, in production.

REGISTRATION ORDER MATTERS HERE TOO

Laravel matches routes in the order they're registered, top to bottom, exactly like Django's own `urlpatterns` — the same underlying principle recurring for a third time in this course's own family. The permissive `{path?}` catch-all has to be the last route registered, or it will swallow requests meant for anything declared after it, admin routes very much included.

Laravel vs. the Other Two Mechanisms

	NEXT.JS	DJANGO	LARAVEL
How it's declared	A folder named `[...slug]`	A dedicated `path` converter type	The ordinary `{param}` syntax plus an explicit `where()` regex
New concept required	A file-naming convention	A new converter type to learn	None — reuses an already-general-purpose mechanism
Home + catch-all	Handled by separate route files	Two separate `path()` entries required	One route, via the optional `?` modifier

Hands-On Exercises

EXERCISE 1

Explain the mechanism difference between Django's `<path:full_path>` converter and Laravel's `{path}` plus `->where('path', '.*')` — both achieve slash-matching, but by genuinely different means. Name what's actually different.

View solution

EXERCISE 2

Explain why `php artisan route:cache` would fail if the catch-all route were defined using an inline closure instead of the `[PageController::class, 'show']` array syntax.

View solution

EXERCISE 3

Explain why `Route::get('/{path?}', ...)` can handle both the homepage and every nested path with one single route entry, something Django's own `<path:full_path>` converter structurally couldn't do without a second, separate entry.

View solution

CHAPTER 3 QUICK REFERENCE

- **Route parameters don't match slashes by default** — the same limitation Django's non-`path` converters have
- `->where('path', '.*')` — Laravel's own fix; an existing general-purpose regex mechanism, not a new dedicated type
- `{path?}` — the optional-parameter modifier; one route handles both the homepage and every nested path
- `trim($path, '/')` — normalizes both leading and trailing slashes before the database lookup
- `firstOrFail()` — Eloquent's own answer to `get_object_or_404`; a missing row becomes an automatic 404
- `route:cache` **rejects closures** — only Controller-referenced routes can be cached for production
- **Registration order matters** — the catch-all must be registered last, the same principle as Django's own `urlpatterns` ordering
- **Next chapter:** Views: Blade Templates

Django Rebuild 4 made a real point of what DTL structurally *can't* do — no arbitrary loops, no genuine tree-walking logic, nothing but a narrow tag vocabulary. Blade sits in a different place entirely: it's a real templating layer, but one that actually allows PHP to run inside it. That's not a small syntax difference — it changes what's even possible to attempt, and this chapter is honest about both what that buys you and why disciplined Laravel code mostly chooses not to use it anyway.

Template Inheritance: `@extends` and `@section`

```
{{-- resources/views/layouts/base.blade.php --}}
<!DOCTYPE html>
<html lang="en">
<head>
    <title>@yield('title', 'Osztromok.com')</title>
</head>
<body class="dark-theme">
    <nav>@yield('breadcrumb')</nav>
    <main>@yield('content')</main>
</body>
</html>
```

```
{{-- resources/views/pages/show.blade.php --}}
@extends('layouts.base')

@section('title', $page->title)

@section('breadcrumb')
    @foreach ($breadcrumb as $crumb)
        <a href="/{{ $crumb->full_path }}/">{{ $crumb->title }}</a> /
    @endforeach
@endsection

@section('content')
    <h1>{{ $page->title }}</h1>
    <div>{!! $page->body !!}</div>
@endsection
```

`@yield` / `@extends` / `@section` do the identical structural job as Django's `{% block %}` / `{% extends %}` — a shell layout, filled in by a child template.

Blade's Real Difference: PHP Is Actually Allowed

Django Rebuild 4 built the breadcrumb walk in the Controller specifically because DTL *couldn't* do it in the template. Blade genuinely could:

```
{{-- This actually runs — DTL structurally could not do this --}}
@php
    $breadcrumb = [];
    $node = $page;
    while ($node) {
        array_unshift($breadcrumb, $node);
        $node = $node->parent;
    }
@endphp

@foreach ($breadcrumb as $crumb)
    <a href="/{{ $crumb->full_path }}/">{{ $crumb->title }}</a> /
@endforeach
```

`@php` / `@endphp` is a real block of genuine PHP, executed directly inside the template, with full access to loops, conditionals, and any expression the language allows. This isn't a workaround or an edge case — it's an intentional, documented Blade feature.

Why the Controller Still Wins Anyway

THE CENTRAL FACT THIS CHAPTER IS BUILT ON

Blade's capability difference from DTL is genuine — the code above actually runs. But disciplined Laravel code still keeps logic like this in the Controller almost all the time, exactly like Django's own code does. The destination — thin views, real logic living elsewhere — is usually the same. What's different is the *enforcement mechanism*: Django's DTL makes the tree-walking loop structurally impossible to write in a template at all. Blade makes it possible, and relies entirely on developer discipline and community convention to keep it out of templates anyway. One is a language limitation; the other is a norm you have to choose to follow.

```php
// app/Http/Controllers/PageController.php — the recommended version
$breadcrumb = [];
$node = $page;
while ($node) {
    array_unshift($breadcrumb, $node);
    $node = $node->parent;
}

return view('pages.show', ['page' => $page, 'breadcrumb' => $breadcrumb]);
```

Auto-Escaping and `{!! !!}`: Blade's Own `|safe`

`{{ $page->title }}` auto-escapes by default, exactly like Django's `{{ }}` and React's JSX. `{!! $page->body !!}` is Blade's own raw-output syntax, the direct equivalent of Django's `|safe` filter — and the same trust-boundary reasoning applies without modification: appropriate here specifically because `page.body` is written exclusively by the authenticated admin, never acceptable for anything a visitor could submit through a form.

Three Points on a Spectrum

	NEXT.JS (REACT/JSX)	DJANGO (DTL)	LARAVEL (BLADE)
Logic in the view layer	Full JavaScript, no restriction at all	Structurally impossible — tags and filters only	Genuinely possible via `@php`, discouraged by convention
What enforces "thin views"	Developer discipline, same as Laravel	The language itself	Developer discipline, same as Next.js/React

Django is the outlier here, not Laravel — both Next.js and Laravel allow real logic in the view layer and rely on convention to keep it disciplined; Django is the one framework in this course's own trio that makes the restriction structural.

Hands-On Exercises

EXERCISE 1

Explain the genuine capability difference between Blade and Django's DTL, illustrated by the breadcrumb example. What's actually possible in Blade that was structurally impossible in DTL?

View solution

EXERCISE 2

Even though Blade can embed the breadcrumb loop directly via `@php`, this chapter still recommends building it in the Controller. Explain why, and explain what's different about how Django enforces the same outcome versus how Laravel does.

View solution

EXERCISE 3

Explain the parallel between Blade's `{!! !!}` and Django's `|safe` filter, including why the identical trust-boundary reasoning (admin-authored content only) applies to both.

View solution

CHAPTER 4 QUICK REFERENCE

- `@extends` / `@section` / `@yield` — Blade's own template inheritance, structurally the same job as Django's `{% extends %}` / `{% block %}`
- `@php` / `@endphp` — real, executable PHP inside a template — a genuine capability Django's DTL doesn't have
- **The Controller still wins** — by convention, not by language limitation; Blade's "thin views" discipline is a choice, Django's is enforced
- `{{ }}` auto-escapes by default; `{!! !!}` — Blade's own direct equivalent of Django's `|safe`, same trust boundary
- **Django is the outlier** — both Next.js and Laravel permit real view-layer logic and rely on convention; Django alone makes it structurally impossible
- **Next chapter:** Styling — Dark Theme

Django Rebuild 5 had no bundler at all — `staticfiles` just serves plain files, exactly as written. Next.js bundles CSS as an inseparable part of the framework's own build system. Laravel lands on neither extreme: `composer create-project` already installed and pre-configured a real, independent tool — Vite — without folding bundling into Laravel itself. A genuine third position, not a compromise between the other two.

The `@vite` Directive

```
{{-- resources/views/layouts/base.blade.php --}}
<head>
    @vite(['resources/css/app.css'])
</head>
```

Django's own `{% static %}` tag points at a plain, unprocessed file — served exactly as it sits on disk. `@vite` points at a *source* file that genuinely gets processed — Sass, PostCSS, autoprefixing, and (in development) live hot-reloading are all real, available capabilities the moment they're needed, not something bolted on afterward.

`vite.config.js` : Already Generated, Nothing to Set Up

```
// vite.config.js — generated automatically by composer create-project
import { defineConfig } from 'vite';
import laravel from 'laravel-vite-plugin';

export default defineConfig({
    plugins: [
        laravel({
            input: ['resources/css/app.css', 'resources/js/app.js'],
            refresh: true,
        }),
    ],
});
```

THE CENTRAL FACT THIS CHAPTER IS BUILT ON

Next.js's own bundler is invisible — baked into the framework's build system, no separate config file most projects ever touch. Django ships nothing at all — `staticfiles` genuinely has no bundling concept. Laravel does neither: it hands you a real, independently-maintained tool, already wired up and ready, but never claims to *be* that tool itself. `vite.config.js` existing as its own real, separate file — not hidden inside Laravel's own internals — is the visible proof of that distinction.

A Real Gotcha: Two Processes Running at Once, in Development

```
# Terminal 1 — the PHP application itself
php artisan serve

# Terminal 2 — the Vite dev server, genuinely required for @vite to resolve anything
npm run dev
```

FORGETTING NPM RUN DEV BREAKS THE PAGE, NOT JUST THE STYLING

In development, `@vite` doesn't read a static file at all — it connects to a running Vite dev server on its own port, expecting a live process to actually answer. Forget to start `npm run dev`, and there's nothing there to connect to: the page can show a connection error, or fail to load any asset at all, not just a missing stylesheet. This is a genuinely different failure shape from Django Rebuild 5's own `DEBUG=False` gotcha — Django's problem only appears in production, once a specific setting flips; Laravel's version can happen on literally the very first day of local development, the moment one of two required processes isn't running.

Production: `npm run build` and the Manifest

```
npm run build
# Generates public/build/manifest.json plus versioned, hashed asset files
# @vite now reads that manifest directly — no dev server involved, or expected, at
all
```

This is Laravel's own direct parallel to Django Rebuild 11's forward-referenced `collectstatic` step — a real, necessary build action that has to happen as part of deployment, not something to discover was missing after the site is already live.

The Stylesheet: The Same Palette, a Third Time

```
/* resources/css/app.css */
:root {
    --bg: #0f1117;
    --surface: #161b22;
    --border: #30363d;
    --text: #c9d1d9;
    --accent: #FF6B57;
}

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

The same background, surface, border, and text colors from Chapter 5 of both sibling courses, reused here for the third and final time in this "same site, three frameworks" trilogy — deliberate continuity, not repetition for its own sake. A visitor moving between any of the three shouldn't be able to tell which one they're looking at.

Hands-On Exercises

EXERCISE 1

Explain the genuine three-way difference in asset bundling between Next.js, Django, and Laravel. Is Laravel "a bundler" the same way Next.js is? Why or why not?

View solution

EXERCISE 2

Explain why forgetting to run `npm run dev` alongside `php artisan serve` breaks the page's assets, and how this failure differs in timing from Django's own `DEBUG=False` static-file gotcha.

View solution

EXERCISE 3

Explain what changes about how `@vite` resolves its assets between development (with `npm run dev` running) and production (after `npm run build` has already executed).

View solution

CHAPTER 5 QUICK REFERENCE

- **A genuine third position** — Next.js bundles by framework design, Django has no bundler at all, Laravel ships a real separate tool (Vite) pre-wired, without being one itself
- `@vite([...])` — points at real source files, genuinely processed, unlike Django's `{% static %}` serving plain files as-is
- `vite.config.js` — already generated by `composer create-project`, nothing to set up manually
- **Two processes in development** — `php artisan serve` and `npm run dev` both required; forgetting the second breaks the page immediately, not just in production
- `npm run build` — generates the production manifest `@vite` reads instead of a dev server; Laravel's own parallel to Django's `collectstatic`
- **The same palette, a third time** — deliberate visual continuity across all three rebuild courses and the live site itself
- **Next chapter:** The Database: Eloquent ORM

The migration and model exist since Chapter 2 — this chapter is about actually living with Eloquent day to day: running the migration for real, Laravel's own interactive shell, and the exact same N+1 query trap Django Rebuild 6 caught, already sitting in this course's own Chapter 4 breadcrumb code too.

Running the Migration & `php artisan tinker`

```
php artisan migrate

php artisan tinker
>>> $root = Page::create(['title' => 'Programming', 'slug' => 'programming']);
>>> $child = Page::create(['title' => 'Python', 'slug' => 'python', 'parent_id' =>
$root->id]);
>>> $child->full_path
=> "programming/python"
```

`tinker` is Laravel's own interactive REPL — the direct equivalent of Django's own `manage.py shell`. `Page::create([...])` mass-assigns through Chapter 2's own `$fillable` list — the same protection is doing real work on every call here, not just a one-time setup detail.

Eloquent's Query Builder: Chained, Then a Real Collection

```
$pages = Page::where('title', 'like', '%Python%')->orderBy('title')->get();
// ->get() is a TERMINAL method — the query runs immediately, right here
// $pages is now a real Eloquent Collection — already fetched, not still queryable
```

Chained methods like `->where()` and `->orderBy()` build up the query without running it — `->get()` (or `->first()`, `->count()`) is what actually executes it. What comes back from a terminal method is a **Collection** — a real, already-fetched, enhanced array-like object with its own useful methods (`->map()`, `->filter()`, `->pluck()`), not something you can call `->where()` on again to keep refining the same database query.

A Real, Live N+1 Example — Again

```
// Chapter 4's own Controller breadcrumb loop, revisited
$node = $page;
while ($node) {
    array_unshift($breadcrumb, $node);
    $node = $node->parent;   // each access not already loaded triggers its own query
}
```

HONEST, NOT OVERSTATED: THE SAME PATTERN, A THIRD TIME IN THIS COURSE'S OWN FAMILY

Every `$node->parent` access that hasn't already been loaded triggers its own fresh query — a genuine N+1 pattern, exactly like the one Django Rebuild 6 caught in its own identical breadcrumb code. And exactly the same reasoning applies here too: breadcrumb depth is bounded (this site never goes more than a handful of levels deep), so this stays a small, currently acceptable cost — worth naming honestly rather than pretending the code is already perfectly optimized, not worth an urgent fix on its own.

`->with()`: Eager Loading, But Not a JOIN

```php
// Without eager loading — N+1, a fresh query every iteration
$pages = Page::whereNotNull('parent_id')->get();
foreach ($pages as $page) {
    echo $page->parent->title;
}

// With eager loading — exactly TWO queries total, not N+1
$pages = Page::with('parent')->whereNotNull('parent_id')->get();
foreach ($pages as $page) {
    echo $page->parent->title;   // no extra query — already loaded
}
```

A REAL, VERIFIED MECHANISM DIFFERENCE — NOT JUST DIFFERENT SYNTAX

Django's `select_related('parent')` fetches the related rows in **one** query, via a genuine SQL `JOIN`. Eloquent's `with('parent')` solves the identical N+1 problem, but through a genuinely different mechanism: it runs a **second**, separate query — a batched `WHERE id IN (...)` against every parent ID collected from the first query's results — then stitches the two result sets together in PHP memory. Both fully eliminate the N+1 pattern; one does it in a single joined query, the other in two batched ones. Neither is "wrong" — they're just genuinely different SQL underneath an almost identical-looking call.

Hands-On Exercises

EXERCISE 1

Explain what `Page::where(...)->get()` actually returns, and why you can't chain another `->where()` onto the result afterward the way you might expect from a "queryset" mental model.

View solution

EXERCISE 2

Explain why the Controller's own breadcrumb loop from Chapter 4 has the identical N+1 pattern Django Rebuild 6 caught in its own breadcrumb code, and why this chapter still treats it as acceptable for now.

View solution

EXERCISE 3

Explain the real mechanical difference between Django's `select_related` (one query, a JOIN) and Eloquent's `with()` (two queries, a batched `WHERE IN`). Both solve N+1 — how do they actually differ underneath?

View solution

CHAPTER 6 QUICK REFERENCE

- `php artisan tinker` — Laravel's own interactive shell, the direct equivalent of Django's `manage.py shell`
- **Eloquent's query builder** — chained methods build the query lazily; a terminal method (`->get()`, `->first()`) executes it immediately and returns a real, already-fetched Collection
- **A live N+1 example, again** — Chapter 4's own breadcrumb loop, honestly acceptable for now given its bounded depth
- `with('parent')` — Eloquent's eager loading; solves N+1 in exactly two queries total
- **A real mechanism difference:** Django's `select_related` uses one query with a JOIN; Eloquent's `with()` uses two queries with a batched `WHERE IN`, not a join
- **Next chapter:** Rendering Content & the Kanji Edge Case

Two chapters have already quietly dismantled both of the reasons kanji pages were ever kept separate from the rest of this site's content. This chapter folds them into the unified tree — and finds a genuinely different kind of wrinkle than Django Rebuild 7 hit, living at a completely different layer of the stack.

Both Original Constraints, Already Resolved

Kanji pages were kept separate for two real reasons: a fixed, three-level content hierarchy with no natural slot for a flat grid of individual characters, and no self-contained-fragment convention, since a kanji page is a complete standalone document with its own inline stroke-animation markup, not a plain fragment. Chapter 2's own arbitrary-depth `Page` model removes the first constraint entirely — `japanese/kanji/水` fits exactly as naturally as any deep course path. Chapter 4's own `{!! $page->body !!}` removes the second — a kanji page's stroke-animation markup can live in `body` and render through the identical mechanism every other page already uses.

The Resolution: One Row Per Kanji Character

```
$kanjiParent = Page::where('full_path', 'japanese/kanji')->firstOrFail();

Page::create([
    'title' => '水 (mizu) — water',
    'slug' => '水',
    'parent_id' => $kanjiParent->id,
    'body' => '<!-- stroke-animation markup and inline CSS -->',
]);

// full_path becomes: japanese/kanji/水
```

A Genuinely Different Kind of Wrinkle: Storage, Not Validation

Django's own kanji wrinkle was an application-level problem — `SlugField`'s built-in validator rejecting the character outright, before the row could ever save. Laravel's `slug` column is just a plain string with no such built-in validator, so that exact failure mode doesn't exist here at all. The genuine risk lives one layer deeper: whether the *database itself* can correctly store a multi-byte character in the first place.

```
// config/database.php — Laravel's own default since 5.4+
'mysql' => [
    // ...
'charset'   => 'utf8mb4',
    'collation' => 'utf8mb4_unicode_ci',
],
```

USUALLY ALREADY FINE — CONFIRM IT, DON'T JUST ASSUME IT

MySQL's older, plain `utf8` charset is a limited, 3-byte-max subset of real UTF-8 — genuinely insufficient for some characters, though most common kanji (this one included) happen to fit within its limits anyway. `utf8mb4` is the real, complete UTF-8 encoding, and it's already Laravel's own sensible default on a fresh install. The right habit isn't assuming this is a problem — it's confirming `config/database.php` actually still says `utf8mb4`, especially on any project connecting to an older or hand-configured MySQL instance where that default might have been overridden.

The Real, Active Risk: Multi-Byte-Unsafe String Functions

```
// DANGEROUS — substr() operates on raw BYTES, not real characters
$truncated = substr($title, 0, 10);   // can slice a multi-byte kanji character in half
// SAFE — mb_substr() respects real UTF-8 character boundaries
$truncated = mb_substr($title, 0, 10);
```

A KANJI CHARACTER IS MULTIPLE BYTES, AND PHP'S PLAIN STRING FUNCTIONS DON'T KNOW THAT

A character like `水` is encoded as three real bytes in UTF-8, but plain PHP functions like `substr()`, `strlen()`, and `strtoupper()` operate on raw bytes, with no awareness that some "characters" actually span several of them. Slicing at exactly the wrong byte offset produces broken, invalid UTF-8 — a real, well-known PHP trap for any future code that ever needs to truncate or manipulate a kanji title programmatically. The `mb_`-prefixed functions (`mb_substr`, `mb_strlen`, and the rest) exist specifically to operate on actual characters instead.

THE CENTRAL FACT THIS CHAPTER IS BUILT ON

Django's kanji wrinkle lived at the application's own input-validation layer — a field type actively refusing the character. Laravel's genuine wrinkle, where it exists at all, lives one layer deeper: the database's own character-set configuration, and any hand-written PHP string manipulation that isn't multi-byte-aware. Same underlying fact — this isn't plain ASCII — surfacing as a real risk at a completely different layer of each framework's own stack.

Stroke-Animation Assets Under Vite

The stroke-order library's own CSS/font/data files move into `resources/`, processed through Chapter 5's own Vite pipeline just like every other asset — still genuinely self-hosted, still never loaded from a third-party CDN. The one non-negotiable property of the original kanji pages carries forward completely unchanged.

Hands-On Exercises

EXERCISE 1

Name kanji's two original constraints and explain why neither one still applies in this Laravel rebuild, tying each one back to a specific earlier chapter.

View solution

EXERCISE 2

Explain why Laravel's own kanji-related wrinkle is genuinely different in *kind* from Django's `SlugField` / `allow_unicode` issue — what layer of the stack does each one actually live at?

View solution

EXERCISE 3

Explain why `substr($title, 0, 10)` is a real risk for a kanji title while `mb_substr($title, 0, 10)` is not, referencing how multi-byte UTF-8 characters are actually encoded.

View solution

CHAPTER 7 QUICK REFERENCE

- **Both original constraints already resolved** — Chapter 2's arbitrary-depth model, Chapter 4's `{!! !!}` rendering
- **No `SlugField`-style validator in Laravel** — the `slug` column is plain, unvalidated string storage
- **The real wrinkle lives at the database layer** — `utf8mb4` vs. the older, limited `utf8` charset; already Laravel's own sensible default, worth confirming rather than assuming
- **The real, active risk:** plain PHP string functions (`substr()`) operate on bytes, not characters — `mb_substr()` and the other `mb_` functions are the safe alternative
- **Different layer, same underlying fact** — Django's wrinkle is application-level validation; Laravel's is database storage plus string-handling discipline
- **Stroke-animation assets** — moved into Vite's own pipeline, still genuinely self-hosted
- **Next chapter:** Dynamic Content & Forms

Django Rebuild 8 answered Next.js's implicit Server Actions with three explicit pieces: a view function, a `ModelForm`, a URL. Laravel's own answer keeps that same explicit spirit, but adds a genuine extra layer neither sibling needed — a real `Controller` class, already established since Chapter 3. This chapter builds the same deliberately-incomplete `updateTitle` mutation both sibling courses built, closed the same way, one chapter later, in Chapter 9.

The Route: Automatic Lookup via Model Binding

```
// routes/web.php
Route::post('/admin/pages/{page}/title', [PageController::class, 'updateTitle'])
    ->name('pages.updateTitle');
```

`{page}` in the route triggers Laravel's own **route model binding** — as long as the Controller method's own parameter is type-hinted `Page $page`, Laravel automatically looks the row up before the method body even runs. Neither Django's function-based views nor a plain Next.js Server Action have quite this automatic an equivalent — Django's own view still needs an explicit `get_object_or_404` call inside the function body.

The Form Request: Validation *and* Authorization, Structurally Separated

```
php artisan make:request UpdatePageTitleRequest

// app/Http/Requests/UpdatePageTitleRequest.php
class UpdatePageTitleRequest extends FormRequest
{
    public function authorize()
    {
        return true;   // deliberately no auth check yet — closed in Chapter 9
    }

    public function rules()
    {
        return [
            'title' => 'required|string|max:200',
        ];
    }
}
```

THE GAP IS LITERALLY VISIBLE IN THE CODE, NOT JUST ABSENT

Laravel's own `FormRequest` class has a dedicated `authorize()` method — a structural separation between "is this data valid" (`rules()`) and "is this user actually allowed to do this" (`authorize()`) that neither Django's `ModelForm` nor a bare Next.js Server Action enforces quite so explicitly. Returning `true` here isn't just *missing* a check the way the sibling courses' own gaps were — it's a real, visible line of code that will change to a genuine condition in Chapter 9. The deliberate incompleteness is structurally on the page, not just an absence someone has to notice.

The Controller Method

```
public function updateTitle(UpdatePageTitleRequest $request, Page $page)
{
    $page->update($request->validated());
    return redirect()->route('page.show', ['path' => $page->full_path]);
}
```

Type-hinting `UpdatePageTitleRequest` is what actually triggers validation — Laravel runs `rules()` automatically, before this method body even starts, and redirects back with errors on failure without a single line of validation-checking code written here. `$request->validated()` returns only the fields that passed — safe to hand straight to `->update()`, the same protection Chapter 2's own `$fillable` already provides against anything unexpected slipping through.

`@csrf`: The Same Requirement, the Same Gotcha

```
<form method="POST" action="{{ route('pages.updateTitle', $page) }}">
    @csrf
    <input type="text" name="title" value="{{ $page->title }}">
    <button type="submit">Save</button>
</form>
```

`@csrf` is Blade's own direct equivalent of Django's `{% csrf_token %}` — Laravel's `VerifyCsrfToken` middleware rejects every POST by default unless the matching hidden token is present, and forgetting `@csrf` produces the identical "form looks correct, submits fine, still fails" confusion Django Rebuild 8 already warned about.

Three Explicit Answers, Compared

	DJANGO	LARAVEL
Pieces required	A view function, a `ModelForm`, a URL	A Controller method, a `FormRequest`, a route
Object lookup	Explicit `get_object_or_404` inside the view	Automatic, via route model binding
Validation vs. authorization	Not structurally separated	Two distinct methods — `rules()` and `authorize()`

Hands-On Exercises

EXERCISE 1

Explain what route model binding does in `Route::post('/admin/pages/{page}/title', ...)`, and why the Controller method never needs to call `Page::findOrFail()` itself.

View solution

EXERCISE 2

Explain why `authorize()` currently returns `true`, and why this chapter treats that as a visible, structural placeholder rather than simply describing "no check at all" the way the sibling courses did.

View solution

EXERCISE 3

Explain what `$request->validated()` actually returns, and why it's safe to pass directly into `$page->update(...)`.

View solution

CHAPTER 8 QUICK REFERENCE

- **Route model binding** — `{page}` plus a type-hinted `Page $page` parameter auto-resolves the row, no manual lookup needed
- `FormRequest` — a dedicated class with `rules()` for validation and `authorize()` for permission, structurally separated
- `authorize() { return true; }` — the deliberate no-auth-check-yet gap, made visible in code rather than just absent, closed in Chapter 9
- `$request->validated()` — only the fields that passed validation; safe to mass-update directly
- `@csrf` — Blade's own direct equivalent of `{% csrf_token %}`; the same "why won't my form submit" trap if forgotten
- **Genuinely more structure than Django** — a real Controller class layer, plus explicit validation/authorization separation Django's function-based views don't require
- **Next chapter:** Admin Authentication

Next.js Rebuild 9 wrote a bespoke `bcryptjs.compare()` call inside a NextAuth Credentials provider to match the existing site's own legacy bcrypt hash. Django Rebuild 9 had to reorder `PASSWORD_HASHERS` to add `BCryptSHA256PasswordHasher`, since Django's own default is PBKDF2, not bcrypt. Laravel needs neither workaround — its default hasher *already is* bcrypt.

Laravel's Default Hasher: Already Bcrypt

```
// config/hashing.php
return [
    'driver' => 'bcrypt',   // already the out-of-the-box default
    ...
];
```

THIS COURSE'S OWN STANDOUT "MODERNIZE IN PLACE" PAYOFF

The legacy site's own admin credential is a bcrypt hash — the same algorithm PHP's own `password_hash()` has defaulted to for years. Laravel's `Hash` facade already defaults to that exact algorithm, with nothing to configure. Django needed a config change; Next.js needed a hand-written comparison function; Laravel needs neither — the framework's own out-of-the-box default already matches the legacy hash.

The Login Flow: `Auth::attempt()`

```php
// app/Http/Controllers/AuthController.php
class AuthController extends Controller
{
    public function showLogin()
    {
        return view('admin.login');
    }

    public function login(Request $request)
    {
        $credentials = $request->validate([
            'email' => 'required|email',
            'password' => 'required',
        ]);

        if (Auth::attempt($credentials)) {
            $request->session()->regenerate();
            return redirect()->intended('/admin');
        }

        return back()->withErrors(['email' => 'Invalid credentials.']);
    }

    public function logout(Request $request)
    {
        Auth::logout();
        $request->session()->invalidate();
        $request->session()->regenerateToken();
        return redirect('/');
    }
}
```

`Auth::attempt($credentials)` internally calls `Hash::check($credentials['password'], $user->password)` using whatever driver `config/hashing.php` names — bcrypt, by default. Since the legacy hash is itself bcrypt, that one built-in call is the entire authentication mechanism: no custom comparison function, no hasher class swap. `session()->regenerate()` issues a fresh session ID on login, guarding against session-fixation — the same reason Django Rebuild 9's own login flow rotated its session key.

A LIGHTER PARALLEL TO DJANGO'S AUTOMATIC HASH UPGRADE

Django Rebuild 9's login flow quietly upgraded the stored hash to a stronger format on a successful login. Laravel offers the same idea manually via `Hash::needsRehash($user->password)` — checked and re-saved on login if the hash's own cost factor is out of date — but unlike Django, it isn't automatic by default; a route has to call it explicitly.

Protecting Chapter 8's Mutation — Closing the Gap

```
// routes/web.php
Route::middleware('auth')->group(function () {
    Route::post('/admin/pages/{page}/title', [PageController::class, 'updateTitle'])
        ->name('pages.updateTitle');
});
```

```
// app/Http/Requests/UpdatePageTitleRequest.php
public function authorize()
{
    return Auth::check();   // Chapter 8's placeholder, now a real check
}
```

Chapter 8's `authorize()` — deliberately left as a visible `return true;` placeholder — now returns `Auth::check()`. The `auth` middleware on the route provides one layer of protection; the `FormRequest`'s own `authorize()` is a second, independent layer that fails the request with a 403 even if the route protection were ever misconfigured — the same defense-in-depth reasoning Django Rebuild 9's `@staff_member_required` decorator embodied for its own view.

Session-Based, Like Django — Not Like NextAuth's JWT

Laravel's default `web` guard is session-based — the same model Django's own session middleware already used. Next.js Rebuild 9's NextAuth setup, by contrast, issued a signed JWT stored in a cookie, with no server-side session record to invalidate directly. Laravel and Django both close a session in one server-side call (`session()->invalidate()` / Django's own `logout()`); revoking a NextAuth JWT before its own expiry needs a deliberate token-blacklist strategy that a plain session model doesn't.

Three Frameworks, Three Different Amounts of Work

	NEXT.JS	DJANGO	LARAVEL
Default hasher	None — hand-rolled Credentials provider	PBKDF2	Bcrypt
Work to match the legacy hash	Write a manual `bcryptjs.compare()` call	Add `BCryptSHA256PasswordHasher` to `PASSWORD_HASHERS`	None — the default already matches
Session model	Signed JWT cookie	Server-side session	Server-side session

NO SELF-REGISTRATION ROUTE

This is a single-admin site, not a multi-user application — there's deliberately no registration route or sign-up form. Only the one legacy admin credential, already present in the `users` table, can log in.

Hands-On Exercises

EXERCISE 1

Explain why Laravel needs no hasher-configuration change to authenticate against the legacy site's own bcrypt hash, while Django Rebuild 9 needed to add `BCryptSHA256PasswordHasher` to `PASSWORD_HASHERS`.

View solution

EXERCISE 2

Explain what changed in `UpdatePageTitleRequest::authorize()` in this chapter, and why both the route's `auth` middleware and the `FormRequest`'s own check together provide defense in depth rather than being redundant.

View solution

EXERCISE 3

Explain why revoking a Next.js/NextAuth JWT session before its natural expiry is harder than ending a Laravel or Django session, and what a JWT-based approach would need to add to close that gap.

View solution

CHAPTER 9 QUICK REFERENCE

- **Standout payoff** — Laravel's default `Hash` driver is already bcrypt, matching the legacy hash with zero configuration
- `Auth::attempt($credentials)` — validates the password against the stored hash using the configured driver, in one call
- `session()->regenerate()` — issues a fresh session ID on login, guarding against session fixation
- `Hash::needsRehash()` — Laravel's manual equivalent of Django's automatic hash-upgrade-on-login
- **Defense in depth** — `auth` middleware on the route, plus `authorize()` now returning `Auth::check()` on the `FormRequest` itself
- **Session-based** — like Django, unlike Next.js's own JWT-cookie NextAuth setup
- **Next chapter:** Admin CRUD Interface

Django Rebuild 10 got a working CRUD interface almost for free — a single `admin.site.register(Page)` call, thanks to Django's own admin app introspecting the model and generating list, edit, and delete views automatically. Laravel ships nothing like that. This is a real, honest philosophical difference, not just a missing convenience method — and it's where this chapter finally pays off Chapter 2's own deferred cost: recomputing `full_path` for an entire subtree when a page is reparented.

No Free Admin — A Real Philosophical Difference

Laravel's own official ecosystem does offer admin-panel packages — **Nova** (paid, first-party) and **Filament** (free, community-maintained) both generate CRUD interfaces from model definitions, genuinely comparable in spirit to Django's own built-in admin. But neither ships with the framework itself the way Django's `admin` app does out of `INSTALLED_APPS` — installing one is an explicit, separate decision, not a default. This chapter builds the interface by hand instead, matching Next.js Rebuild 10's own approach and, more importantly, because hand-building it is the only way to actually teach the reparenting logic this chapter needs to cover.

Listing Pages: The Same Searchable Flat List

Next.js Rebuild 10 solved its own parent-picker problem — choosing a new parent for a page at arbitrary depth — with a searchable flat list rather than a nested tree widget, for an explicit, scale-justified reason: at this site's realistic page count, a flat list with client-side filtering is simpler to build and just as fast to use as a tree view, and a tree view's own advantage (seeing hierarchy at a glance) matters less than being able to type a few letters and jump straight to a page. That reasoning carries over unchanged here.

```
// app/Http/Controllers/Admin/PageController.php
public function edit(Page $page)
{
    $allPages = Page::orderBy('full_path')
        ->where('id', '!=', $page->id)
        ->get(['id', 'full_path']);

    return view('admin.pages.edit', [
        'page' => $page,
        'allPages' => $allPages,
    ]);
}
```

The page itself is excluded from its own candidate-parent list — reparenting a page under itself, or under one of its own descendants, would corrupt the tree. A simple client-side `<input>` filters the `<select>` options by `full_path` substring, giving fast lookup without a full tree-rendering component.

Paying Off Chapter 2's Deferred Cost: `movePage()`

```php
// app/Models/Page.php
public function movePage(?Page $newParent): void
{
    $this->parent_id = $newParent?->id;
    $this->full_path = $newParent
        ? $newParent->full_path . '/' . $this->slug
        : $this->slug;
    $this->save();

    $this->updateDescendantPaths();
}

protected function updateDescendantPaths(): void
{
    foreach ($this->children as $child) {
        $child->full_path = $this->full_path . '/' . $child->slug;
        $child->save();
        $child->updateDescendantPaths();   // recurse down the subtree
    }
}
```

EXPLICIT RECURSION, NOT AN IMPLICIT CASCADE

Django Rebuild 10 paid this same cost through an *implicit* mechanism — overriding `save()` once, then re-triggering every descendant's own `save()` in breadth-first order, so each node recomputes its own `full_path` from its own parent without any code explicitly walking the tree. Laravel's Eloquent models don't have an equivalent built-in cascade for a computed field like this, so `updateDescendantPaths()` walks the subtree explicitly and recursively instead. Same real cost, same result, genuinely different mechanism — Django's cascade lives inside the model's own `save()` override; Laravel's lives in a dedicated method that's called on purpose.

Refusing, Not Cascading, on Delete

```
public function destroy(Page $page)
{
    try {
        $page->delete();
    } catch (\Illuminate\Database\QueryException $e) {
        return back()->withErrors([
            'delete' => "Can't delete a page that still has children — move or delete
them first.",
        ]);
    }

    return redirect()->route('admin.pages.index');
}
```

Chapter 2's own `restrictOnDelete()` constraint means the database itself refuses to delete a page with existing children — the raw failure is a `QueryException` from a foreign-key violation. Catching it here and returning a plain, readable error message turns that low-level database refusal into an honest, expected UI outcome, rather than a raw stack trace reaching the admin.

Deliberately no automatic cascading delete: reparenting or removing an entire subtree is left as a conscious, separate action.

Three Admin Interfaces, Compared

	NEXT.JS	DJANGO	LARAVEL
Free admin panel	None — built by hand	`admin.site.register()`, generated automatically	None by default — Nova/Filament exist as separate packages
Reparent cascade	Explicit `movePage()` walk	Implicit — re-triggered `save()` cascade	Explicit `movePage()` + recursive `updateDescendantPaths()`
Delete-with-children	Refused, error surfaced in the UI	Refused via `on_delete=PROTECT`, surfaced in the admin	Refused via `restrictOnDelete()`, caught and surfaced

A TREE VIEW WOULD SCALE DIFFERENTLY

The flat searchable list works well at this site's realistic page count. A collection large enough to make scrolling a flat list impractical would eventually justify a real nested tree widget — a genuine future trade-off, not something this chapter treats as settled forever.

Hands-On Exercises

EXERCISE 1

Explain the real, honest philosophical difference between Django's free automatic admin and Laravel's total absence of one, and why this chapter builds the interface by hand rather than reaching for a package like Filament.

View solution

EXERCISE 2

Explain how `Page::movePage()` recomputes `full_path` for the moved page and all of its descendants, and how this differs mechanically from Django Rebuild 10's own breadth-first `save()` re-trigger approach.

View solution

EXERCISE 3

Explain why deleting a page with children fails with a friendly error instead of silently cascading, and which earlier chapter's decision this behavior traces back to.

View solution

CHAPTER 10 QUICK REFERENCE

- **No free admin by default** — Nova (paid) and Filament (free) exist as separate packages, not a framework default like Django's `admin` app
- **Searchable flat list** — same parent-picker approach as Next.js Rebuild 10, for the same scale-justified reason
- `movePage()` — updates the moved page's own `full_path`, then calls `updateDescendantPaths()`
- `updateDescendantPaths()` — explicit recursive walk, Laravel's own answer to Django's implicit `save()`-cascade mechanism
- **Delete refusal** — Chapter 2's `restrictOnDelete()` throws a `QueryException`, caught here and turned into a readable UI error
- **Next chapter:** Deployment

Next.js Rebuild 11 stood up PM2, nginx, and certbot from nothing. Django Rebuild 11 stood up gunicorn behind a new nginx reverse proxy from nothing. Neither course could reuse anything already running on the live server, because the live server wasn't running Next.js or Django. This course's own deployment chapter gets to be genuinely different: the live site is *already* PHP behind Apache — so this chapter evolves that exact stack instead of replacing it, the strongest "modernize in place" payoff of the whole course.

Production Environment Settings

```
# .env (production)
APP_ENV=production
APP_DEBUG=false
APP_KEY=base64:...
```

`APP_DEBUG=false` parallels Django's own `DEBUG=False` from two directions at once: it hides Laravel's own detailed error pages (stack traces, file paths, query dumps) from real visitors, and it disables debug-only behavior that would otherwise leak information or cost performance in production. `APP_KEY`, generated once via `php artisan key:generate`, parallels Django's own `SECRET_KEY` directly — it's what Laravel uses to encrypt cookies and sign sessions. Rotating it has the identical consequence Django Rebuild 11 already flagged for `SECRET_KEY`: every existing session and encrypted cookie becomes unreadable, forcing every logged-in user to sign in again.

Reusing the Live Apache: PHP-FPM via `mod_proxy_fcgi`

```
# Apache VirtualHost — the site's own already-running Apache
<VirtualHost *:443>
    ServerName osztromok.com
    DocumentRoot /var/www/laravel-rebuild/public

    <Directory /var/www/laravel-rebuild/public>
        AllowOverride All
        Require all granted
    </Directory>

    <FilesMatch \.php$>
        SetHandler "proxy:unix:/run/php/php8.3-fpm.sock|fcgi://localhost"
    </FilesMatch>

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

NO NEW PROXY STACK — THE SAME APACHE, EXTENDED

`mod_proxy_fcgi` — the exact module Apache In Depth's own Chapter 7 covered as Apache's answer to reverse proxying — hands PHP requests straight to a PHP-FPM pool over a Unix socket. There's no nginx being introduced, no PM2 process manager, no separate reverse-proxy layer to learn or operate: the same Apache install already serving the live site's existing pages gets one more `VirtualHost` block and a `FilesMatch` directive routing PHP requests to FPM. TLS termination, logging, and the certificate already configured for the live domain are all reused unchanged.

Production Optimizations: The `artisan` Cache Commands

```
php artisan config:cache
php artisan route:cache
php artisan view:cache
php artisan migrate --force
```

> **ROUTE:CACHE STILL REJECTS CLOSURES**
>
> Chapter 3 already flagged this: `route:cache` fails outright if any route is defined as a closure rather than a Controller-method reference. Every route in this course has pointed at a Controller method since Chapter 3 specifically so this command would work without rewriting anything at deployment time — the payoff of that earlier discipline lands here.

`migrate --force` is needed because Artisan normally prompts for confirmation before running migrations when `APP_ENV=production` — a safety guard that would otherwise block an unattended deploy script. `config:cache` and `route:cache` combine every config file and route definition into single, fast-loading compiled files, avoiding the cost of re-parsing them on every request.

Building Assets: Chapter 5's Vite Payoff

```
npm run build
```

This produces the compiled, hashed asset files and the manifest Chapter 5's `@vite` directive reads at render time — the same role Django Rebuild 11's own `collectstatic` played for that course's static files, and the same underlying build step Chapter 5 warned would need to actually run before deployment, not just in local dev.

File Permissions: `storage/` and `bootstrap/cache/`

Laravel writes logs, compiled views, and cached files into `storage/` and `bootstrap/cache/` at runtime — both directories need to be writable by the web server's own user (typically `www-data`), a detail easy to miss on a fresh deploy and one of the most common sources of a confusing "permission denied" error on a brand-new Laravel install.

No Laravel-Vercel Equivalent

LARAVEL VAPOR EXISTS, BUT IT'S A DIFFERENT PATH THAN THIS COURSE TAKES

Laravel Vapor is the closest thing to a serverless deployment platform for Laravel — genuinely comparable in spirit to Vercel's own role for Next.js. It's a paid, AWS-specific product built around Lambda, not a free default the way Vercel's own free tier is for Next.js. This course deploys the traditional way, onto the site's own already-live server, matching the honest gap Django Rebuild 11 already named for its own framework: there's no free, zero-config serverless equivalent here either.

Three Deployment Stories, Compared

	NEXT.JS	DJANGO	LARAVEL
Process manager	PM2 (new)	gunicorn (new)	PHP-FPM (already running)
Reverse proxy	nginx (new)	nginx (new)	Apache (already live, extended)
TLS	certbot, configured fresh	certbot, configured fresh	Already configured for the live domain
Serverless equivalent	Vercel (free tier)	None	Laravel Vapor (paid, AWS-specific)

Hands-On Exercises

EXERCISE 1

Explain why this course's own deployment chapter can reuse Apache directly, while both the Next.js and Django rebuild courses had to stand up an entirely new nginx reverse proxy from nothing.

View solution

EXERCISE 2

Explain why `route:cache` works without any changes at deployment time in this course specifically, tracing the reason back to a decision made in an earlier chapter.

View solution

EXERCISE 3

Explain what `APP_KEY` is used for, and what happens to existing logged-in users if it's rotated — drawing the parallel to Django's own `SECRET_KEY`.

View solution

CHAPTER 11 QUICK REFERENCE

- **Strongest "modernize in place" payoff** — PHP-FPM behind the site's own already-live Apache, via `mod_proxy_fcgi`
- `APP_DEBUG=false` — hides detailed error pages and disables debug-only behavior, paralleling Django's `DEBUG=False`
- `APP_KEY` — Laravel's own `SECRET_KEY` equivalent; rotating it invalidates every existing session/encrypted cookie
- `config:cache` / `route:cache` / `view:cache` — precompile for production; `route:cache` still rejects closures
- `migrate --force` — bypasses the production confirmation prompt for unattended deploys
- `npm run build` — Chapter 5's Vite payoff, producing the manifest `@vite` reads at render time
- **No free serverless equivalent** — Laravel Vapor exists but is paid and AWS-specific, unlike Vercel's free tier for Next.js
- **Next chapter:** Capstone — A Complete, Working Flexible-Routing Site

Website Rebuild with Laravel

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

Sam — the same site admin from the Next.js and Django rebuild capstones, reused a third time because it's genuinely the same site being rebuilt a third time — logs into this course's own deployed Laravel version. Every step below draws on a specific earlier chapter, closing with an attribution table and an honest scope note.

STEP 1 — VISITING THE LIVE DEPLOYMENT

Sam opens the site in a browser. It's served by the same Apache instance that's always run this domain, now also handing PHP requests to a Laravel PHP-FPM pool over `mod_proxy_fcgi`. `APP_DEBUG=false` means a stray error would show a clean generic page, not a stack trace.

STEP 2 — LOGGING IN

At `/admin/login`, Sam enters the legacy admin credential. `Auth::attempt()` checks it against the stored bcrypt hash using Laravel's own default hasher — no configuration was ever needed to make this work, since Laravel's default already matched the legacy hash's algorithm. `session()->regenerate()` issues a fresh session ID.

STEP 3 — BUILDING THE FIVE-LEVEL CHAIN

Through the admin CRUD interface, Sam creates `programming`, then `general-purpose-languages` under it, then `java`, then `fundamentals`, then `chapter-1` — five real levels deep, no longer the illustrative example Chapter 1 opened with. Each save recomputes that page's own `full_path` from its parent, exactly as Chapter 2's self-referencing `Page` model and Chapter 6's Eloquent ORM were designed to do.

STEP 4 — VIEWING THE PAGE

Visiting `/programming/general-purpose-languages/java/fundamentals/chapter-1`, Chapter 3's `{path?}` wildcard route resolves the full five-segment path in one route definition, Chapter 4's Blade template renders the page with a breadcrumb built in the Controller rather than the template, and Chapter 5's Vite-bundled dark-theme CSS applies.

STEP 5 — EDITING THE TITLE, NOW ACTUALLY PROTECTED

Sam edits the Chapter 1 page's title. Chapter 8's route and `UpdatePageTitleRequest` handle the submission, with `@csrf` protecting the form — but this time, `authorize()` returns Chapter 9's real `Auth::check()` instead of the placeholder `true` Chapter 8 deliberately left in place. Logged out, the exact same request would now be refused.

STEP 6 — REORGANIZING: PAYING THE REAL CASCADE COST

Sam decides `fundamentals` belongs under a different parent. Chapter 10's `movePage()` updates `fundamentals`' own `full_path`, then `updateDescendantPaths()` recurses down and updates `chapter-1`'s `full_path` too — the real, concrete payment of the cascade cost Chapter 2 flagged and deliberately deferred all the way back 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. The database's own `restrictOnDelete()` constraint from Chapter 2 refuses the operation; Chapter 10's `try`/`catch` in `destroy()` turns the resulting `QueryException` into a plain, readable message instead of a raw error page.

STEP 8 — CONFIRMING THE KANJI MIGRATION HELD UP

Sam visits `/japanese/kanji/水`. Chapter 7's resolution holds: no application-level slug validator blocks the character the way Django's own `SlugField` once needed `allow_unicode=True` to allow, and the database's `utf8mb4` charset stores the multi-byte character correctly. The page renders through the same rendering path as every other page on the site — no special-cased kanji route required.

STEP 9 — A QUICK N+1 SANITY CHECK

Sam checks the query log while loading the breadcrumb-heavy Chapter 1 page. Chapter 6's `with()` eager loading is doing its job — one batched query for the ancestor chain, not one query per level, the same fix that was needed after the N+1 pattern was first caught live in Chapter 4's own breadcrumb code.

STEP 10 — CONFIRMING THE DEPLOY ITSELF IS HEALTHY

Sam confirms `npm run build`'s output is being served (Chapter 5/11), and that `route:cache` succeeded without complaint — every route across the whole application has pointed at a Controller method since Chapter 3, specifically so this moment would be uneventful.

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, 6 — Eloquent ORM, 10 — Admin CRUD
4. Viewing the page	3 — Routing, 4 — Views: Blade, 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 — `restrictOnDelete()`, 10 — Admin CRUD
8. Kanji migration	7 — Rendering Content & the Kanji Edge Case
9. N+1 sanity check	6 — Eloquent ORM
10. Deploy health check	3 — Routing, 5 — Styling, 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

THE COURSE'S OWN THROUGHLINE, CLOSED

Every chapter framed itself against "modernize in place" — reusing the live site's own PHP/Apache stack rather than replacing it outright. That framing pays off concretely in this capstone: Chapter 9's login needed no hasher configuration, and Chapter 11's deployment needed no new reverse-proxy layer, both because the legacy stack Laravel is evolving was already compatible with what Laravel defaults to.

Hands-On Exercises

EXERCISE 1

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

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 separate points in this capstone where the course's own "modernize in place" framing paid off concretely, and explain what each one avoided having to do.

View solution

COURSE COMPLETE

- **12/12 chapters** — Website Rebuild with Laravel is now complete
- **Standout wins named honestly throughout** — zero-config bcrypt matching (Ch9), Apache reuse (Ch11), real PHP inside Blade via `@php` (Ch4)
- **Real limits named honestly too** — no free admin panel (Ch10), an extra Controller-class layer Django didn't require (Ch8)
- **Third sibling in the Website Rebuild series** — alongside Website Rebuild with Next.js and Website Rebuild with Django
- **Outstanding:** Website Rebuild with Ruby on Rails remains the last unbuilt variant