PROJECTS

Website Rebuild with Django

A Flexible, Self-Referencing URL Model

Django's Own Explicit Routing, Views & ORM

The Kanji Edge Case, Resolved Again

Batteries-Included Admin Authentication

Capstone: A Complete, Working Flexible-Routing Site

Philip Osztromok

Generated with Claude

Table of Contents

Website Rebuild with Django — 12 Chapters

Website Rebuild with Next.js rebuilt this same real site — Philip's own learning blog, currently served as HTML fragments out of a MySQL database via PHP/Apache — specifically to support genuinely **arbitrary-depth** content nesting: `programming/general-purpose-languages/java/fundamentals/chapter-1`, five levels deep, rather than the site's current fixed three-level `subject/subtopic/page` model. This course rebuilds the exact same site, with the exact same requirement, in Django instead — not to declare one framework "better," but because Django solves the identical arbitrary-depth problem through a genuinely different mechanism: its own URL dispatcher paired with a self-referencing database model, rather than Next.js's file-based catch-all routes. Chapter 2 gets to that model directly; this chapter is about standing up a real, working Django baseline first.

`django-admin startproject`: Project vs. App

```
# Creates the overall project — settings, URL config, WSGI/ASGI entry points
django-admin startproject osztromok_site
cd osztromok_site

# Creates an APP — a self-contained, reusable unit of functionality inside the project
python manage.py startapp content
```

A STRUCTURAL SPLIT NEXT.JS SIMPLY DOESN'T HAVE

Django's **project** (the settings/configuration container — `osztromok_site`, here) and **app** (a focused, in-principle-reusable piece of functionality — `content`, here) are two genuinely different things, both required, related but not interchangeable. Next.js has no equivalent split at all: there's just "the app," full stop, with routing and configuration living directly inside it rather than in a separate outer container. A Django project routinely hosts several independent apps side by side (this course's own `content` app, plus `django.contrib.admin` and the other built-in apps introduced below); a Next.js project has no comparable notion of several separately pluggable "apps" cooperating inside one project.

The Generated Skeleton

```
osztromok_site/
├── manage.py                  # the command-line entry point for everything —
runserver, migrate, etc.
├── content/                   # the app just created — will hold the Page model
starting Chapter 2
│   ├── migrations/
│   ├── models.py
│   ├── views.py
│   └── apps.py
└── osztromok_site/
    ├── settings.py             # INSTALLED_APPS, DATABASES, middleware — the
project-wide configuration
    ├── urls.py                 # the project's own root URL configuration
    ├── wsgi.py
    └── asgi.py
```

`settings.py` and the project-level `urls.py` are the two files this course returns to most — the first for configuration, the second for wiring the flexible URL dispatch Chapter 3 builds.

Batteries-Included, Visible From the First Run

```
# settings.py — already present, nothing added by hand yet
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]
```

A real admin interface (`django.contrib.admin`) and a full authentication system (`django.contrib.auth`) are already listed — before a single line of this course's own code has been written. Website Rebuild with Next.js needed to install and wire up NextAuth.js separately, starting only at its own Chapter 9. This course reaches the identical admin-login requirement in its own Chapter 9 too — but largely by *using* what `startproject` already generated, not by adding a third-party package first. Worth remembering now; it becomes a genuinely concrete payoff later.

Confirming the Baseline Actually Runs

```
# Applies the migrations the built-in apps above already define (auth, sessions,
etc.)
python manage.py migrate

python manage.py runserver
# Starting development server at http://127.0.0.1:8000/
```

Nothing about this site's own actual content exists yet — this step exists purely to confirm the project itself is sound before Chapter 2 starts modeling real data on top of it.

> **A FIRST PRACTICAL HABIT**
>
> Run `python manage.py migrate` before `runserver` on a genuinely fresh project, every time — the built-in apps in `INSTALLED_APPS` already define real database tables (for sessions, users, and more), and skipping this step is a common source of a confusing first-run error that has nothing to do with anything this course will actually build.

Where This Course Is Headed

Designing the same flexible, arbitrary-depth URL and content model Next.js Rebuild 2 designed — now as a self-referencing Django model — then Django's own URL dispatch mechanism, the MVT view/template model, dark-theme styling, the database via Django's ORM, rendering content (including the same kanji edge case Next.js Rebuild 7 resolved), dynamic content and forms, admin authentication, a custom admin CRUD interface for the tree structure, deployment, and a capstone building a genuinely five-level-deep chain live.

Hands-On Exercises

EXERCISE 1

Explain, in your own words, what a Django "project" is and what a Django "app" is, and why Next.js has no real equivalent to that specific two-level split.

View solution

EXERCISE 2

This chapter lists `django.contrib.admin` and `django.contrib.auth` as already present in `INSTALLED_APPS` immediately after `startproject`. Explain why this is called out as a genuine advantage over the Next.js rebuild's own starting point, specifically regarding Chapter 9's admin authentication work.

View solution

EXERCISE 3

Explain why `python manage.py migrate` is run before `runserver` on a fresh project, even though this course hasn't defined a single model of its own yet.

View solution

CHAPTER 1 QUICK REFERENCE

- **The rebuild's own real requirement** — arbitrary-depth content nesting, the same one Website Rebuild with Next.js was built around
- `django-admin startproject` — creates the project (settings, root URL config); `manage.py startapp` — creates an app inside it
- **Project vs. app** — a genuine structural distinction with no real Next.js equivalent
- `settings.py` / `urls.py` — the two files this course returns to most
- `INSTALLED_APPS` — admin and auth are already present out of the box, unlike Next.js's own NextAuth.js add-on requirement
- `manage.py migrate` before `runserver` — applies the built-in apps' own tables even before this course defines any of its own
- **Next chapter:** Designing a Flexible URL & Content Model

Everything else in this course rests on getting one design decision right: how to store a content tree that can genuinely go five levels deep — `programming/general-purpose-languages/java/fundamentals/chapter-1` — without the schema itself assuming a fixed depth. There are four standard ways to represent a tree in a relational database, each with real, different tradeoffs. This chapter picks one, deliberately, rather than reaching for the first option that comes to mind.

Four Ways to Store a Tree

STRATEGY	READING A FULL PATH	MOVING A SUBTREE
Adjacency list	Slow — walk one parent link at a time, one query per level, to reconstruct a deep path	Fast — change one row's `parent_id`
Materialized path	Fast — one lookup by the stored path string	Slow — every descendant's stored path has to be rewritten
Nested sets	Fast for whole-subtree queries	Very slow — nearly every other row's left/right values shift on any change
Closure table	Fast, flexible — a separate ancestor-descendant join table answers most queries directly	Moderate — real extra complexity and an extra table to keep in sync

This site's own real traffic pattern is heavily read-weighted — pages get *rendered* constantly, and *reorganized* rarely, by one admin, deliberately. That single fact rules out nested sets outright (punishing exactly the operation that matters least here) and makes a closure table's own extra complexity hard to justify for a project this size. A pure adjacency list alone would make every single page render slow, which is precisely backwards for a read-heavy site.

THE DESIGN THIS CHAPTER SETTLES ON: BOTH, TOGETHER

A self-referential **adjacency list** stays the real, authoritative structure — the actual parent/child relationships the admin interface (Chapter 10) edits directly. A **materialized path**, stored as an ordinary indexed column, exists purely as a fast-lookup cache derived from that structure — the thing Chapter 3's URL dispatch actually queries against. Reads stay fast (one indexed lookup by path); the real tree structure stays explicit and query-able for anything the adjacency list is naturally better at, like rendering a page's own breadcrumb or a sibling list. This is the same hybrid shape Website Rebuild with Next.js reached with Prisma, arrived at independently here for the identical reason: this specific site's own read-heavy, write-rare traffic profile, not a framework preference.

The Model

```
# content/models.py
from django.db import models

class Page(models.Model):
    title = models.CharField(max_length=200)
    slug = models.SlugField(max_length=100)

    parent = models.ForeignKey(
        'self',
        null=True,
        blank=True,
        on_delete=models.PROTECT,
        related_name='children',
    )

    full_path = models.CharField(max_length=500, unique=True, db_index=True)
    body = models.TextField(blank=True)

    class Meta:
        unique_together = ('parent', 'slug')

    def __str__(self):
        return self.full_path

    def save(self, *args, **kwargs):
        if self.parent:
            self.full_path = f"{self.parent.full_path}/{self.slug}"
else:
            self.full_path = self.slug
        super().save(*args, **kwargs)
```

`parent = models.ForeignKey('self', ...)` is Django's own syntax for a self-referential relationship — the string `'self'` stands in for "this same model," since the class isn't fully defined yet at the point the field itself is declared. `related_name='children'` is what makes `some_page.children.all()` work later, walking downward instead of only upward through `parent`.

ON_DELETE=PROTECT IS A DELIBERATE CHOICE, NOT THE DEFAULT

Django's own default for a `ForeignKey` is `on_delete=CASCADE` — deleting a parent silently deletes every one of its children too, recursively, all the way down. For a content tree that's a genuinely dangerous default: deleting one page by mistake could wipe out an entire subject's worth of course chapters without any warning. `PROTECT` instead refuses the deletion outright while children still exist, forcing a deliberate decision (delete the children first, or reparent them) rather than an accidental cascade.

Two Fields, Two Different Jobs

`slug` is just this one page's own single URL segment — `"chapter-1"`, not the whole path. `full_path` is the complete route, built by walking up through `parent` once, at save time, and stored so nothing downstream ever has to walk that chain again. `SlugField` also comes with real, built-in validation — letters, numbers, hyphens, and underscores only — rejecting anything that couldn't cleanly become a URL segment before it ever reaches the database.

A COST BEING DELIBERATELY DEFERRED, NOT AVOIDED

Recomputing `full_path` in `save()` only ever updates the one page being saved — moving a page that already has children doesn't yet cascade that update down through its descendants' own stored `full_path` values. That's a real gap, left open on purpose: Chapter 10's own admin move/reparent feature is exactly where this gets paid off properly, once there's an actual UI action ("move this page") to hang the cascade logic on. Noting it now, rather than only discovering it later, is the point.

Where This Course Is Headed

Django's own URL dispatch mechanism built directly on this model (a genuinely different approach from Next.js's file-based catch-all routes), the MVT view/template model, dark-theme styling, the database migration that makes this model real, rendering content (including the kanji edge case), dynamic content and forms, admin authentication, the admin CRUD interface that finally pays off this chapter's own deferred move-cascade cost, deployment, and a capstone building a genuinely five-level-deep chain live.

Hands-On Exercises

EXERCISE 1

Explain why a pure adjacency list alone, with no materialized path, would make this site's own page rendering slow — and why nested sets were ruled out for the opposite reason. Use this chapter's own read-heavy/write-rare framing.

View solution

EXERCISE 2

Explain what would happen if `Page.parent` used Django's own default `on_delete=CASCADE` instead of `PROTECT`, and why that default is described as "genuinely dangerous" for this specific model.

View solution

EXERCISE 3

Explain the deferred cost this chapter names around moving a page that already has children. Why is it acceptable to leave that gap open in this chapter rather than solving it here?

View solution

CHAPTER 2 QUICK REFERENCE

- **Four tree strategies** — adjacency list, materialized path, nested sets, closure table — each with a different read/write tradeoff
- **This project's choice: both** — a self-referential adjacency list as the real structure, a stored materialized `full_path` as a fast-lookup cache
- `ForeignKey('self', ...)` — Django's syntax for a self-referential relationship; `related_name='children'` enables walking downward
- `on_delete=PROTECT` — chosen deliberately over Django's own default `CASCADE`, to prevent silently deleting an entire subtree
- `slug` **vs.** `full_path` — one URL segment vs. the complete, precomputed route
- **Deferred cost:** moving a page with children doesn't yet cascade-update descendants' own `full_path` — Chapter 10's own job
- **Next chapter:** URL Dispatch: Django's Own Mechanism for Arbitrary-Depth Routing

Website Rebuild with Next.js solved arbitrary-depth routing with a special filename: a folder literally named `[...slug]`, whose presence alone tells Next.js "match anything here, and hand me the segments as an array." Django has no filename convention that does anything at all — every route is a line of ordinary Python, read top to bottom, matched in the exact order it's declared. Both frameworks reach the same end result; the actual mechanism underneath is genuinely different, and that difference is this chapter's whole subject.

Path Converters: What Actually Makes a Catch-All Possible

Django's `path()` function matches URL segments using named **converters** — `str`, `int`, `slug`, `uuid`. Every one of those stops at the first `/` it encounters. Only one converter doesn't: `path`, which matches literally anything, forward slashes included — the single piece of syntax this whole chapter is built around.

```
# osztromok_site/urls.py
from django.contrib import admin
from django.urls import path
from content import views

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', views.homepage, name='homepage'),
    path('<path:full_path>/', views.page_detail, name='page_detail'),
]
```

`<path:full_path>` means "match anything, including slashes, and pass it to the view as a keyword argument named `full_path`" — the direct equivalent of Next.js's own `[...slug]` array param, just delivered as a single string instead of an array of segments.

THE CENTRAL FACT THIS CHAPTER IS BUILT ON

Next.js's own routing table is *implicit* — the folder structure on disk **is** the routing table, discovered by convention. Django's is *explicit* — `urlpatterns` is an ordinary Python list, read and matched top to bottom, that you write and can see in full in one file. Neither approach is objectively better; they're two genuinely different philosophies (convention-over-configuration vs. explicit-is-better-than-implicit) that this whole course keeps surfacing, starting with Chapter 1's own project/app split.

The View: One Lookup, Not a Walk

```python
# content/views.py
from django.shortcuts import render, get_object_or_404
from .models import Page

def page_detail(request, full_path):
    full_path = full_path.rstrip('/')
    page = get_object_or_404(Page, full_path=full_path)
    return render(request, 'content/page_detail.html', {'page': page})
```

This is exactly the payoff Chapter 2's own `full_path` field was built for: `full_path` arrives from the URL already in the identical shape it was stored in, so resolving any URL — no matter how deep — is one indexed database lookup, never a walk back up through `parent`.

Three Real Edge Cases

PATTERN ORDER MATTERS — THE CATCH-ALL HAS TO COME LAST

Django matches `urlpatterns` in declaration order and stops at the first match. If `path('<path:full_path>/', ...)` were listed *before* `path('admin/', ...)`, a request for `/admin/` would never reach the real admin route at all — `full_path` would greedily match `"admin"` first, and Django would call `page_detail` looking for a `Page` that was never meant to exist. The catch-all pattern has to be the very last entry in the list, always.

THE HOMEPAGE NEEDS ITS OWN EXPLICIT ROUTE

`<path:full_path>` requires at least one real character — it doesn't match an empty string, so the bare root URL `/` would 404 with only the catch-all pattern in place. `path('', views.homepage, name='homepage')` exists specifically to handle that one case the catch-all structurally can't.

A TRAILING-SLASH MISMATCH IS A GENUINE, EASY WAY TO BREAK EVERY LOOKUP

Django's own convention appends a trailing slash to normalized URLs, so a request typically arrives as `/programming/python/` — but Chapter 2's `save()` method built `full_path` with *no* trailing slash (`"programming/python"`). Querying with the raw, unstripped `full_path` parameter would never match a single real row, for every single page, silently. `full_path.rstrip('/')` exists purely to close that exact gap before the query ever runs.

Django vs. Next.js: The Same Job, Two Mechanisms

	NEXT.JS	DJANGO
How the catch-all is declared	A folder named `[...slug]` — a filename convention	A `path()` call using the `path` converter — an explicit line of code
What the handler receives	An array of individual URL segments	A single string, the full remaining path
Where routing logic lives	Discovered from the folder structure on disk	Declared explicitly in `urlpatterns`, read top to bottom

Hands-On Exercises

EXERCISE 1

Explain why Next.js's `[...slug]` file-based catch-all and Django's `path('<path:full_path>/', ...)` achieve the identical end result through genuinely different mechanisms. Name the actual mechanism each framework relies on.

View solution

EXERCISE 2

Explain why the catch-all pattern must be listed last in `urlpatterns`, using a concrete example of what would actually break if it were listed before the `admin/` route instead.

View solution

EXERCISE 3

Explain why `full_path.rstrip('/')` is necessary inside `page_detail`, referencing exactly how `full_path` was built back in Chapter 2.

View solution

CHAPTER 3 QUICK REFERENCE

- **Path converters** — `str` / `int` / `slug` / `uuid` all stop at a `/`; only `path` matches through it
- `<path:full_path>` — Django's own catch-all syntax; delivers one string, unlike Next.js's array of segments
- **Explicit vs. implicit routing** — Django's `urlpatterns` is ordinary Python code; Next.js's routing table is the folder structure itself
- **Pattern order matters** — the catch-all must be the last entry, or it swallows requests meant for earlier routes
- **The homepage needs its own explicit** `path('', ...)` — the catch-all can't match an empty path
- `full_path.rstrip('/')` — reconciles Django's own trailing-slash convention against Chapter 2's own no-trailing-slash `full_path` storage
- **Next chapter:** Views & Templates: Django's MVT Model

Chapter 2 built the Model; Chapter 3 wired the routing that finds one. This chapter is the other two letters of MVT — the View and the Template — and Django's own answer to what Website Rebuild with Next.js 4 solved with React components. The mechanism is genuinely different: no JSX, no component props, and — deliberately — no arbitrary code running inside a template file at all.

Template Inheritance: `extends` and `block`

```
<!-- content/templates/content/base.html -->
<!DOCTYPE html>
<html lang="en">
<head>
    <title>{% block title %}Osztromok.com{% endblock %}</title>
</head>
<body class="dark-theme">
    <nav>{% block breadcrumb %}{% endblock %}</nav>
    <main>{% block content %}{% endblock %}</main>
</body>
</html>
```

```
<!-- content/templates/content/page_detail.html -->
{% extends "content/base.html" %}

{% block title %}{{ page.title }}{% endblock %}

{% block content %}
    <h1>{{ page.title }}</h1>
    <div>{{ page.body|safe }}</div>
{% endblock %}
```

`base.html` defines the page's overall shell — the same job Next.js Rebuild 4's own shared `Layout` component did — with named `{% block %}` slots left open. `page_detail.html` extends it and only fills in the specific blocks it actually needs; everything else in the shell is inherited unchanged.

The Django Template Language: Deliberately Not a Programming Language

`{{ variable }}` outputs a value, auto-escaped by default — the same automatic XSS protection JSX gives Next.js, just via a different mechanism. `{% if %}` / `{% for %}` and a limited set of filters (`|truncatewords`, `|date`, and the rest) cover most real templating needs. What DTL deliberately *doesn't* have is arbitrary function calls, real expressions, or general-purpose logic — no way to write a genuine loop-inside-a-loop tree walk, or call an arbitrary Python function with arguments, directly inside a template file.

Building the Breadcrumb: Logic in the View, Not the Template

```
def page_detail(request, full_path):
    full_path = full_path.rstrip('/')
    page = get_object_or_404(Page, full_path=full_path)

    breadcrumb = []
    node = page
    while node:
        breadcrumb.insert(0, node)
        node = node.parent

    return render(request, 'content/page_detail.html', {
        'page': page,
        'breadcrumb': breadcrumb,
    })
```

Walking `node.parent` upward — Chapter 2's own adjacency list, not the materialized `full_path` Chapter 3 used for the lookup — happens entirely in Python, producing an already-flat list before the template ever sees it. The template's own job is trivial by comparison:

```
{% block breadcrumb %}
    {% for crumb in breadcrumb %}
        <a href="/{{ crumb.full_path }}/">{{ crumb.title }}</a> /
    {% endfor %}
{% endblock %}
```

THE CENTRAL FACT THIS CHAPTER IS BUILT ON

This split — a real, arbitrary Python loop in the view; a simple, flat `{% for %}` in the template — is Django's own "thin templates" philosophy made concrete. Every genuinely tricky part of the breadcrumb (walking an unknown number of levels upward, in the correct order) lives in ordinary, testable Python, exactly where Chapter 2's own model design intended the adjacency list to be used. The template only ever iterates a list that already exists — it was never capable of doing that walk itself, and that's a deliberate design choice, not a missing feature: a template language that *could* run arbitrary logic is also one where a template-injection-style bug becomes possible in the first place.

|SAFE IS ONLY APPROPRIATE BECAUSE OF WHERE PAGE.BODY ACTUALLY COMES FROM

`{{ page.body }}` alone would auto-escape any HTML inside `page.body`, turning real markup into visible `<p>` text on the page — not what's wanted, since `body` genuinely stores HTML. `|safe` disables that escaping, and it's the right call *here* specifically because `page.body` is written exclusively by the authenticated site admin (Chapter 9), never submitted directly by an anonymous visitor. Applying `|safe` to anything a visitor could actually type into a form would reopen exactly the XSS risk Django's own auto-escaping exists to close.

Context Processors: Site-Wide Data Without Repeating Yourself

```
# content/context_processors.py
def site_settings(request):
    return {'current_year': 2026}

# settings.py — added to TEMPLATES['OPTIONS']['context_processors']
# 'content.context_processors.site_settings',
```

Once registered, `{{ current_year }}` is available in *every* template, site-wide, with no view ever needing to pass it explicitly — the same "shared, always-available data" goal a Next.js layout component solves by simply always rendering, just reached through Django's own request-scoped context mechanism instead.

Django's Templates vs. Next.js's React Components

	NEXT.JS	DJANGO
Composition unit	A React component — real JavaScript, JSX, props, children	A template file — `{% %}` / `{{ }}` tags only
Shared layout	A shared `Layout` component wrapping page content	Template inheritance — `{% extends %}` + named `{% block %}` slots
Logic inside the view layer	Full JavaScript, directly inside the component	Deliberately restricted — real logic belongs in the view (Python), not the template

Hands-On Exercises

EXERCISE 1

Explain why the breadcrumb is built by walking `page.parent` in the view (Python) rather than attempting that same walk directly inside the template with DTL tags. What does this illustrate about Django's own "thin templates" philosophy?

View solution

EXERCISE 2

Explain why `{{ page.body|safe }}` is used instead of plain `{{ page.body }}`, and why this specific use of `|safe` is considered acceptable rather than a security risk.

View solution

EXERCISE 3

Explain what a context processor does, using the `current_year` example, and why it's a better fit here than passing `current_year` explicitly from every individual view function.

View solution

CHAPTER 4 QUICK REFERENCE

- `{% extends %}` / `{% block %}` — template inheritance; Django's own answer to a shared Next.js layout component
- **DTL is deliberately limited** — `{{ }}` / `{% %}` tags and filters only, no arbitrary code execution
- **Breadcrumb built in the view** — walking `page.parent` in Python, handed to the template as an already-flat list
- **"Thin templates"** — real logic belongs in views/models, not templates; a deliberate separation-of-concerns choice
- `|safe` — disables auto-escaping; only appropriate for trusted, admin-authored content, never for visitor-submitted input
- **Context processors** — make data available to every template automatically, without passing it from every view
- **Next chapter:** Styling — Dark Theme

Website Rebuild with Next.js 5 gave the rebuilt site its dark theme through Next.js's own bundling pipeline. Django has no equivalent built-in bundler at all — its `staticfiles` app organizes and serves plain files directly, nothing more. That's a genuine difference worth understanding before writing a single line of CSS, not just a syntax swap.

Django's `staticfiles` App

```
<!-- content/templates/content/base.html -->
{% load static %}
<!DOCTYPE html>
<html lang="en">
<head>
    <link rel="stylesheet" href="{% static 'content/style.css' %}">
</head>
```

`{% load static %}` makes the `{% static %}` tag available in that template; `{% static 'content/style.css' %}` resolves to the correct served URL for that file, wherever `STATIC_URL` actually points.

Why `static/content/...`, Not Just `static/...`

```
content/
├── static/
│   └── content/
│       └── style.css       # the extra "content/" nesting is deliberate
├── templates/
│   └── content/
│       ├── base.html
│       └── page_detail.html
```

THE SAME NAMESPACING IDEA AS CHAPTER 4'S TEMPLATES, APPLIED TO STATIC FILES

`collectstatic` (covered properly in Chapter 11) merges *every* installed app's own `static/` folder into one single, flat `STATIC_ROOT` directory for production. If two different apps both had a `static/style.css`, one would silently overwrite the other during that merge — there'd be no error, just a mysteriously wrong stylesheet in production. Nesting each app's files under its own name (`static/content/style.css`) avoids the collision entirely, the exact same reasoning Chapter 4's `templates/content/...` nesting already relied on.

The Stylesheet: Reusing This Site's Own Established Palette

```
/* content/static/content/style.css */
:root {
    --bg: #0f1117;
    --surface: #161b22;
    --border: #30363d;
    --text: #c9d1d9;
    --accent: #44B78B;
}

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

These aren't arbitrary color choices — they're the exact same background, surface, border, and text colors every generated lesson fragment across this whole site already uses. The point of a rebuild is continuity: a visitor moving between an old page and a newly-migrated one shouldn't be able to tell which framework rendered it.

Development vs. Production: A Real Gotcha

STATIC FILES STOP BEING SERVED THE MOMENT DEBUG=FALSE

Django's own dev server (`runserver`) serves static files automatically — but only while `DEBUG=True`. Chapter 11's own production setup requires `DEBUG=False`, and the instant that flag flips, Django stops serving `static/` files itself entirely. A site that rendered its own CSS perfectly throughout every chapter of local development can suddenly appear completely unstyled the moment it's deployed — not a bug in the CSS, a missing production step: `python manage.py collectstatic` has to gather every app's files into `STATIC_ROOT` first, and something else (WhiteNoise, or nginx directly) has to actually serve from there, since Django itself won't.

No Built-In Bundler: A Genuine Difference From Next.js

	NEXT.JS	DJANGO
CSS handling	Runs through Next.js's own build pipeline — bundling, minification, and optimization all built in	Plain files, served as-is by `staticfiles` — no bundling unless you add a separate tool
If bundling/SCSS is genuinely wanted	Already there by default	A separate package (e.g. `django-compressor`) — an explicit, deliberate addition, not the default

For a project this size, plain hand-written CSS with no build step is a completely reasonable choice — but it's worth knowing the absence of a bundler is a real, structural difference from Next.js, not something this course is simply choosing to skip covering.

> **A FORWARD REFERENCE**
>
> Chapter 11's own deployment work covers `collectstatic` and WhiteNoise in real depth — this chapter only needs to establish that the step exists and why, so it isn't a surprise later.

Hands-On Exercises

EXERCISE 1

Explain why this project's stylesheet lives at `content/static/content/style.css` — with "content" appearing twice — rather than simply `content/static/style.css`.

View solution

EXERCISE 2

Explain why a Django site that displays its own CSS perfectly throughout local development can suddenly appear completely unstyled the moment it's deployed with `DEBUG=False`, and name the command that fixes it.

View solution

EXERCISE 3

Explain the genuine structural difference between how Next.js handles CSS and how Django's `staticfiles` app handles it — not just "different syntax," but a real difference in what each framework does by default.

View solution

CHAPTER 5 QUICK REFERENCE

- `{% load static %}` / `{% static 'path' %}` — makes and resolves a static file URL inside a template
- `static/<app_name>/...` — namespacing convention avoiding collisions once `collectstatic` merges every app's files into one folder
- **This site's own established dark-theme palette** — reused deliberately, for real visual continuity across the rebuild
- **Static files are only auto-served while** `DEBUG=True` — production needs `collectstatic` plus WhiteNoise or nginx
- **No built-in bundler** — a genuine structural difference from Next.js, not just different syntax
- **Next chapter:** The Database with Django's ORM

The `Page` model has existed as Python since Chapter 2 — nothing about it is real in an actual database yet. This chapter makes it real, the same job Website Rebuild with Next.js 6 gave to Prisma, and a second real-world instance of relational modeling this site already worked through once, in Food Tracker (Django) 2. Along the way, this chapter also covers the single most common way a Django ORM query quietly costs far more than it looks like it should — and points directly at a real, live example already sitting in this course's own code.

Migrations: `makemigrations` & `migrate`

```
python manage.py makemigrations content
python manage.py migrate
```

`makemigrations` compares `models.py` against what Django last knew about and writes a migration file describing the difference. `migrate` actually applies it to the database. Prisma's own workflow (`prisma migrate dev`) does conceptually the same two-step job, but the artifact it produces is different: Prisma generates a raw `.sql` migration file directly, where Django generates real, executable **Python** describing the change.

What a Migration File Actually Contains

```
# content/migrations/0001_initial.py
from django.db import migrations, models
import django.db.models.deletion

class Migration(migrations.Migration):
    initial = True
    operations = [
        migrations.CreateModel(
            name='Page',
            fields=[
                ('id', models.AutoField(primary_key=True, serialize=False)),
                ('title', models.CharField(max_length=200)),
                ('slug', models.SlugField(max_length=100)),
                ('full_path', models.CharField(db_index=True, max_length=500,
unique=True)),
                ('body', models.TextField(blank=True)),
                ('parent', models.ForeignKey(
                    blank=True, null=True,
                    on_delete=django.db.models.deletion.PROTECT,
                    related_name='children', to='content.page',
                )),
            ],
        ),
    ]
```

Even the self-referential `parent` field appears exactly as declared back in Chapter 2 — Django resolved `'self'` down to a real, explicit reference (`to='content.page'`) once it generated this file.

The Django Shell: Testing the ORM Interactively

```
python manage.py shell

>>> from content.models import Page
>>> root = Page.objects.create(title="Programming", slug="programming")
>>> child = Page.objects.create(title="Python", slug="python", parent=root)
>>> child.full_path
'programming/python'
```

Prisma Studio does the same job through a GUI; the Django shell is a plain Python REPL with the ORM already imported and ready — no separate tool to install or launch.

QuerySets Are Lazy

```
qs = Page.objects.filter(title__icontains="python")   # NO query has run yet
qs = qs.exclude(slug="old-python")                     # still no query
list(qs)   # NOW — exactly one combined query actually runs, right here
```

A `QuerySet` only describes a query — it doesn't execute anything until it's actually evaluated (iterated, converted with `list()`, sliced for display, and so on). Chaining `.filter()` / `.exclude()` across several lines never means several queries; it builds up one query description that runs exactly once, at the moment something finally needs real rows.

The N+1 Problem — Including One Already Sitting in This Codebase

```
# Chapter 4's own breadcrumb loop
node = page
while node:
    breadcrumb.insert(0, node)
    node = node.parent   # each access after the first is a SEPARATE query
```

HONEST, NOT OVERSTATED: A REAL N+1 PATTERN ALREADY EXISTS IN THIS COURSE

Every `node.parent` access that hasn't already been loaded triggers its own fresh query — a five-level-deep page means up to five separate round trips just to build one breadcrumb. That's a genuine N+1 pattern, and it's worth naming rather than pretending the code written so far is already perfectly optimized. It's also, honestly, a small and currently acceptable cost: breadcrumb depth is bounded (this site never goes deeper than a handful of levels), so five small queries per page view isn't the kind of problem that demands an immediate fix. The identical pattern applied to something *unbounded* — a page listing showing many rows, each needing its own parent's title — is where this stops being a minor inefficiency and becomes a real one.

`select_related`: Fixing It With a Real JOIN

```
# Without select_related — one extra query PER ROW, in the loop (N+1)
pages = Page.objects.filter(parent__isnull=False)
for p in pages:
    print(p.parent.title)   # a fresh query, every single iteration
# With select_related — ONE query total, parent JOINed in up front
pages = Page.objects.filter(parent__isnull=False).select_related('parent')
for p in pages:
    print(p.parent.title)   # no extra query — already fetched via the JOIN
```

THE CENTRAL FACT THIS CHAPTER IS BUILT ON

The ORM hides the SQL — it never hides the *cost*. Recognizing when a chain of attribute access like `.parent.parent` is secretly N separate queries, rather than one, is a genuine, core ORM skill, not a rare edge case. This project's own Chapter 4 breadcrumb code is presented here honestly, as a live example of exactly that pattern, currently small enough not to matter — the same habit of noticing it now is what prevents it from quietly becoming a real problem once this project grows.

Hands-On Exercises

EXERCISE 1

Explain why `qs = Page.objects.filter(...).exclude(...)`, written across two separate lines, doesn't run two separate database queries. What actually triggers the query to run?

View solution

EXERCISE 2

Explain why Chapter 4's breadcrumb-building loop is a real N+1 query pattern, and why this chapter treats it as an acceptable, bounded cost right now rather than something demanding an immediate fix.

View solution

EXERCISE 3

Given a page listing that loops over many pages and accesses `page.parent.title` for each one, write the corrected queryset using `select_related`, and explain specifically what problem it solves.

View solution

CHAPTER 6 QUICK REFERENCE

- `makemigrations` / `migrate` — generate a real Python migration file describing model changes, then apply it
- **Django migrations are Python, not raw SQL** — a genuine mechanical difference from Prisma's own `.sql` migration files
- `python manage.py shell` — an interactive Python REPL with the ORM ready to go, Django's own answer to Prisma Studio
- **QuerySets are lazy** — chaining `.filter()` / `.exclude()` builds one query description; nothing runs until it's actually evaluated
- **The N+1 problem** — an unguarded relationship access inside a loop triggers one query per row, not one query total
- **A real, current example** — Chapter 4's own breadcrumb loop, honestly named as N+1, currently small enough to be acceptable
- `select_related('field')` — fetches a related row via a real SQL JOIN up front, turning N+1 queries into one
- **Next chapter:** Rendering Content & the Kanji Edge Case

Every kanji page on the live site today is a real, deliberate exception: a self-contained HTML file with its own self-hosted stroke-order animation, living outside the site's normal subject/subtopic/page hierarchy entirely, dual-written into two separate folders rather than driven by a database row. Website Rebuild with Next.js 7 resolved this same edge case for its own architecture; this chapter reaches the identical resolution independently, for the same underlying reason — and finds one genuinely new, Django-specific wrinkle along the way that Next.js never had to solve.

Why Kanji Was Kept Separate, Originally

Two real constraints, not an arbitrary decision:

- **Fixed depth.** The site's existing three-level `subject/subtopic/page` model had no natural slot for "a flat grid of individual characters" — kanji didn't fit the shape every other course/lesson already used.
- **No self-contained-fragment convention.** A kanji page isn't a bare content fragment meant to drop into a shared template — it's a complete, standalone document with its own inline stroke-animation markup and CSS, genuinely different in shape from an ordinary lesson page.

Neither Constraint Survives This Rebuild

BOTH ORIGINAL REASONS ARE ALREADY RESOLVED BY EARLIER CHAPTERS

Chapter 2's own `Page` model supports genuinely arbitrary depth — `japanese/kanji/水` fits exactly as naturally as any five-level course path, no special case required. Chapter 4's own `{{ page.body|safe }}` already renders real, admin-authored HTML directly — a kanji page's stroke-animation markup and inline CSS can live in `body` and render through the identical mechanism every other page already uses. The two constraints that justified keeping kanji separate in the first place were both artifacts of the *old* three-level, fragment-only model — neither one is a property of this rebuild's own architecture at all.

The Resolution: One Page Row Per Kanji Character

```
kanji_parent = Page.objects.get(full_path="japanese/kanji")

Page.objects.create(
    title="水 (mizu) — water",
    slug="水",
    parent=kanji_parent,
    body="<!-- stroke-animation markup and inline CSS, same as the current live page
-->",
)
# full_path becomes: japanese/kanji/水
```

A New Wrinkle Django-Specific: Non-ASCII Characters in a Slug

SLUGFIELD'S DEFAULT VALIDATOR REJECTS THE KANJI CHARACTER ITSELF

Chapter 2's original `slug = models.SlugField(max_length=100)` only accepts ASCII letters, numbers, hyphens, and underscores by default — a literal `"水"` fails that validation outright, immediately, before the row would ever save. This is a genuinely new problem, one Next.js's own file-based routing never had to confront the same way.

```
# content/models.py — Chapter 2's slug field, revisited
slug = models.SlugField(max_length=100, allow_unicode=True)
```

`allow_unicode=True` is Django's own built-in answer to exactly this scenario — real, non-ASCII characters become valid slug content, not just tolerated as an escape hatch. The alternative (romanizing to something like `"sui"` and keeping the real character only in `title`) was considered and rejected here: it would be less faithful to the source material, and Chapter 3's own `path` converter already handles UTF-8 URL segments correctly with zero changes needed — nothing about routing has to adapt to make this work.

Stroke-Animation Assets: Still Self-Hosted, Now Under Chapter 5's Own Convention

```
content/
└── static/
    └── content/
        └── animcjk/          # the stroke-order library's own CSS/font/data files
            ├── animcjk.css
            └── ...
```

The animation library's own asset files move into the exact same app-namespaced static folder Chapter 5 already established — `staticfiles` serves them locally, the same as always. The one non-negotiable property of the original kanji pages carries forward unchanged: these assets are genuinely self-hosted, never loaded from a third-party CDN, so a kanji page's stroke animation never depends on any outside service staying online.

> **THE |SAFE TRUST BOUNDARY ISN'T REOPENED, JUST REUSED**
>
> Rendering a kanji page's own `<style>`/animation markup through `{{ page.body|safe }}` relies on the exact same reasoning Chapter 4 already established: this is safe specifically because kanji content, like every other page's `body`, is admin-authored and migrated deliberately — never submitted by an anonymous visitor. Nothing new is being trusted here; it's the same trust boundary, applied to one more kind of content.

Hands-On Exercises

EXERCISE 1

Name the two original reasons kanji pages were kept separate from the rest of the site's content model, and explain why this chapter concludes neither one still applies in this rebuild specifically.

View solution

EXERCISE 2

Explain why Chapter 2's original `slug = models.SlugField(max_length=100)` needed to change specifically to support kanji pages, and what `allow_unicode=True` actually does.

View solution

EXERCISE 3

Explain why moving the stroke-animation CSS/font files into `content/static/content/animcjk/` still satisfies the original "self-hosted, not CDN-dependent" requirement the kanji pages were built around.

View solution

CHAPTER 7 QUICK REFERENCE

- **Two original constraints:** a fixed three-level hierarchy with no natural slot for kanji, and a fragment-only rendering model kanji pages never fit
- **Both resolved already** — Chapter 2's arbitrary-depth `Page` model and Chapter 4's `{{ page.body|safe }}` rendering remove both constraints entirely
- **The resolution:** one `Page` row per kanji character, e.g. `japanese/kanji/水`
- **A genuinely new wrinkle:** `SlugField`'s default validator is ASCII-only; kanji slugs need `allow_unicode=True`
- **Stroke-animation assets** — moved into `static/content/animcjk/`, still fully self-hosted, never CDN-dependent
- **The `|safe` trust boundary isn't new** — the same reasoning from Chapter 4, applied to one more kind of admin-authored content
- **Next chapter:** Dynamic Content & Forms

Website Rebuild with Next.js 8 gave 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 that course's own Chapter 9. This chapter builds the identical feature for this rebuild — the same page title, the same deliberate gap, closed the same way in this course's own Chapter 9 — but through a mechanism with no real equivalent in Next.js: three separate, explicit pieces, rather than one function.

Django Has No Single "Server Actions" Equivalent

A Next.js Server Action is a function that runs on the server, callable seemingly directly from a form — the client/server boundary is deliberately blurred, hidden behind what looks like an ordinary function call. Django keeps that boundary fully explicit instead: a real HTTP POST, to a real URL, handled by a real view function, validated by a real form class. Nothing about a Django mutation is hidden — every piece can be read, in isolation, in its own file.

The Form: `ModelForm`

```
# content/forms.py
from django import forms
from .models import Page

class PageTitleForm(forms.ModelForm):
    class Meta:
        model = Page
        fields = ['title']
```

A `ModelForm` derives its fields and validation directly from the model itself — Chapter 2's own `Page` definition — the same "batteries-included, no separate schema to maintain" theme running since Chapter 1's `INSTALLED_APPS`.

The View

```python
# content/views.py
from django.shortcuts import redirect, get_object_or_404
from .forms import PageTitleForm
from .models import Page

def update_page_title(request, full_path):
    page = get_object_or_404(Page, full_path=full_path.rstrip('/'))

    if request.method == 'POST':
        form = PageTitleForm(request.POST, instance=page)
        if form.is_valid():
            form.save()
        return redirect('page_detail', full_path=full_path)
```

NO PERMISSION CHECK YET — DELIBERATELY, NOT AN OVERSIGHT

Anyone who knows or guesses this URL can currently rename any page — there's no login requirement anywhere in this view. That's the exact same gap Next.js Rebuild 8 left open on purpose in its own `updatePageTitle` Server Action, and it gets closed the same way here: Chapter 9's admin authentication work is what makes leaving it open right now the correct order to build in, not a mistake to fix immediately.

The Template: `{% csrf_token %}`

```
<form method="post" action="{% url 'update_page_title' full_path=page.full_path %}">
    {% csrf_token %}
    {{ form.as_p }}
    <button type="submit">Save</button>
</form>
```

`{{ form.as_p }}` renders the whole `ModelForm`'s fields automatically — one line, no manual `<input>` tags to keep in sync with the model.

> **FORGETTING {% CSRF_TOKEN %} ISN'T A SECURITY HOLE — IT'S A BROKEN FORM**
>
> Django protects every POST request with CSRF verification by default, at the framework level, whether the template remembers to include a token or not. Leave `{% csrf_token %}` out of a form, and the submission is simply **rejected** — a 403 Forbidden, every time, no exceptions. This is one of the single most common "why won't my form submit" confusions for anyone new to Django: the form looks completely correct, submits fine in isolation, and still fails, because the one hidden field the framework actually requires was never rendered.

Next.js Server Actions vs. Django's Explicit Three Pieces

	NEXT.JS	DJANGO
The mutation itself	One function, marked to run on the server	A view function, a `ModelForm`, and a URL — three separate, explicit pieces
Client/server boundary	Deliberately blurred — looks like calling a normal function	Fully explicit — a real HTTP POST to a real, inspectable URL
CSRF-equivalent protection	Built into the framework's own Server Action handling	Built into Django, but requires an explicit `{% csrf_token %}` in the template

THE CENTRAL FACT THIS CHAPTER IS BUILT ON

Django's version is genuinely more code than a single Next.js Server Action — and that's the tradeoff, not a downside to smooth over. Every mutation stays fully visible: a real URL you can find in `urls.py`, a real view you can read top to bottom, a real form whose validation rules live in one place. This is the same explicit-over-implicit theme that's run through this entire course, from Chapter 1's project/app split through Chapter 3's own routing — Django keeps choosing to show its work rather than hide it behind a more convenient abstraction.

Hands-On Exercises

EXERCISE 1

Explain why `update_page_title` has no permission or login check yet, and why this chapter treats that as a deliberate, temporary gap rather than an oversight.

View solution

EXERCISE 2

Explain what happens if `{% csrf_token %}` is left out of the form template, and why this happens even though the form otherwise looks completely correct.

View solution

EXERCISE 3

Explain the real difference between how a Next.js Server Action handles a mutation and how this chapter's own view + `ModelForm` + URL combination handles the identical kind of mutation.

View solution

CHAPTER 8 QUICK REFERENCE

- **No "Server Actions" equivalent** — Django keeps the client/server boundary fully explicit instead of blurring it
- `forms.ModelForm` — auto-derives fields and validation directly from a model
- **A real view, a real URL, a real form** — three separate, explicit, individually-readable pieces per mutation
- **No auth check yet, deliberately** — the same gap Next.js Rebuild 8 left open, closed the same way in this course's own Chapter 9
- `{% csrf_token %}` — required inside every POST form; omitting it means a guaranteed 403, not a subtle bug
- **Explicit over implicit** — more code than a Server Action, but every piece stays visible and independently readable
- **Next chapter:** Admin Authentication

`django.contrib.auth` has been sitting in `INSTALLED_APPS` since Chapter 1's very first look at `settings.py` — this chapter is the first time it actually gets used for something. Website Rebuild with Next.js 9 needed a third-party package, NextAuth.js, configured with a custom credentials check to authenticate against the existing site's own bcrypt password hash. Django needs no separate package at all — just one real compatibility problem to solve first.

The Real Problem: Django's Default Hasher Isn't Bcrypt

The existing live site stores its admin password as a bcrypt hash — a string starting `$2b$...`. Django's own default password hasher is PBKDF2, a different algorithm entirely. Authenticating against the existing hash without forcing an admin password reset means teaching Django to recognize bcrypt specifically, not switching Django's default to it.

```
# settings.py
PASSWORD_HASHERS = [
    'django.contrib.auth.hashers.PBKDF2PasswordHasher',        # Django's own default
— used for any NEW password
'django.contrib.auth.hashers.BCryptSHA256PasswordHasher',   # recognizes the EXISTING
site's bcrypt hash
]
```

Django ships `BCryptSHA256PasswordHasher` built in — it simply isn't enabled by default. Once listed, Django's own authentication reads a stored hash's own algorithm prefix and picks the matching hasher automatically, regardless of which one appears first in the list.

Importing the Existing Admin — Without Resetting the Password

```
from django.contrib.auth.models import User

admin = User(username='philip', is_staff=True, is_superuser=True)
admin.password = '$2b$12$existingHashFromTheOldSite...' # assigned directly — not hashed again
admin.save()
```

ASSIGNED DIRECTLY, NEVER PASSED THROUGH SET_PASSWORD()

`set_password()` takes a *plaintext* password and hashes it — calling it here would hash the already-hashed string, corrupting it completely. The existing value is already a real bcrypt hash; it belongs directly in `.password`, unchanged, so Django's own authentication can compare a future login attempt against it exactly as the old PHP site would have.

A Quiet Payoff: The Hash Upgrades Itself

The first time this admin successfully logs in, Django's own `authenticate()` verifies the submitted password against the stored bcrypt hash — and then, a well-known built-in Django behavior, silently re-hashes that same password using the *first* hasher in `PASSWORD_HASHERS` (PBKDF2, Django's preferred one) and saves it. No admin action, no visible step — the stored hash simply becomes a modern Django-native one the moment it's actually used for the first time.

The Login View

```
# osztromok_site/urls.py
from django.contrib.auth import views as auth_views

urlpatterns = [
    path('admin-login/',
auth_views.LoginView.as_view(template_name='content/login.html'), name='login'),
    # ...
]
```

`LoginView` is another built-in — no custom view needed to handle a login form, verify credentials, or start a session; only a template to render it in.

Closing Chapter 8's Own Gap, For Real

```
from django.contrib.auth.decorators import staff_member_required

@staff_member_required
def update_page_title(request, full_path):
    # unchanged from Chapter 8 — now genuinely protected
    ...
```

One decorator, one line, and Chapter 8's own deliberately-open `update_page_title` now requires a logged-in staff account before it runs at all — exactly the fix that chapter's own warn-box promised would arrive here.

NextAuth.js vs. Built-In `django.contrib.auth`

	NEXT.JS	DJANGO
Auth package	NextAuth.js — a third-party dependency, added deliberately	`django.contrib.auth` — already installed since Chapter 1, no new package
Checking the existing bcrypt hash	A custom `authorize()` callback written by hand inside the Credentials provider	A built-in hasher class, enabled by adding it to `PASSWORD_HASHERS`
The login form itself	Built by hand, wired to the Credentials provider	`LoginView`, built in — only a template needed

THE CENTRAL FACT THIS CHAPTER IS BUILT ON

Nothing here is "adding" authentication to the project — it's *using* capability that was already sitting there since Chapter 1's very first `INSTALLED_APPS` listing. That's the concrete payoff of the advantage Chapter 1 named honestly up front: Django's batteries-included admin/auth apps meant this chapter's real work was solving one genuine compatibility problem (bcrypt vs. PBKDF2), not building an authentication system from nothing the way a bare framework would have required.

Hands-On Exercises

EXERCISE 1

Explain why `BCryptSHA256PasswordHasher` had to be added to `PASSWORD_HASHERS` to authenticate against the existing site's admin password, given that PBKDF2 is Django's own default hasher.

View solution

EXERCISE 2

Explain why the existing admin's bcrypt hash is assigned directly to `user.password` rather than passed through `set_password()`.

View solution

EXERCISE 3

Explain what happens to the stored password hash the first time the admin successfully logs in, once both `BCryptSHA256PasswordHasher` and `PBKDF2PasswordHasher` are listed in `PASSWORD_HASHERS`.

View solution

CHAPTER 9 QUICK REFERENCE

- `django.contrib.auth` — already installed since Chapter 1; this chapter finally uses it
- **The real problem:** the existing site's password is bcrypt-hashed; Django defaults to PBKDF2
- `PASSWORD_HASHERS` + `BCryptSHA256PasswordHasher` — built into Django, just not enabled by default; recognizes the existing hash without a password reset
- **Assign the existing hash directly to** `.password` — never through `set_password()`, which would hash it a second time
- **Automatic hash upgrade** — the first successful login silently re-hashes the password with Django's own preferred (first-listed) hasher
- `LoginView` — built in; only a template is needed
- `@staff_member_required` — the real fix for Chapter 8's own deliberately-open `update_page_title` gap
- **Next chapter:** Admin CRUD Interface

Food Tracker (Django) 3 already showed that Django's admin can turn a model into a working CRUD interface with a single line of registration, before any custom view exists at all. That's still true here — and, exactly like Website Rebuild with Next.js 10 discovered, it still isn't enough on its own for a genuinely browsable tree. This chapter's real job is the one Chapter 2 deliberately deferred: moving a page that already has children, and doing it correctly.

Free CRUD From One Line

```
# content/admin.py
from django.contrib import admin
from .models import Page

admin.site.register(Page)
```

`/admin/content/page/` now lists every page, with working add/edit/delete forms — genuinely free, matching Food Tracker (Django) 3's own precedent exactly.

Why the Free Admin Isn't Enough Here

The default admin's own widget for `parent` is a plain dropdown listing *every* `Page`, by its `__str__` (Chapter 2's own `full_path`). For a site with hundreds of pages, that's one enormous, flat, alphabetically-sorted list — no indentation, no sense of depth, no way to browse "show me this page's children." Genuinely unusable once the tree has any real size, for exactly the reason Next.js Rebuild 10 needed its own custom parent picker.

A Cheap Intermediate Step

```
class PageAdmin(admin.ModelAdmin):
    list_display = ['title', 'full_path', 'parent']
    search_fields = ['title', 'full_path']
    raw_id_fields = ['parent']

admin.site.register(Page, PageAdmin)
```

`raw_id_fields` replaces the giant dropdown with a search popup instead — a real, worthwhile improvement for almost no code. It's still a flat *search*, though, not a browsable tree — genuinely better, not genuinely sufficient.

The Real Build: Paying Off Chapter 2's Own Deferred Cost

Chapter 2's own `save()` method only ever recomputes the *one* page being saved's `full_path` — moving a page with children was left as a known gap, deferred specifically until there was a real admin action to attach it to. This is that action.

```
def move_page(page, new_parent):
    page.parent = new_parent
    page.save()   # recalculates THIS page's own full_path, per Chapter 2's own
save()
# Cascade to every descendant, PARENT-FIRST — a simple breadth-first walk
    to_process = list(page.children.all())
    while to_process:
        child = to_process.pop(0)
        child.save()   # re-derives full_path from child.parent.full_path — already
correct by now
        to_process.extend(child.children.all())
```

THE CENTRAL FACT THIS CHAPTER IS BUILT ON

`move_page` never manually edits a single `full_path` string. It doesn't need to — Chapter 2's own `save()` already recomputes `full_path` from `parent.full_path` every time it runs, for *any* page. The only real trick is processing order: as long as each page is saved only *after* its own parent's `full_path` is already correct, calling plain `.save()` down through the whole subtree — breadth-first, level by level — produces exactly the right cascade, with zero string manipulation anywhere. Chapter 2's own model design didn't just defer this problem; it was quietly built to make the eventual solution this simple.

ORDER GENUINELY MATTERS

Saving a grandchild before its own direct child would still leave that grandchild computing its `full_path` from a *stale* parent value — the breadth-first queue above exists specifically to guarantee every page is only processed once its own parent is already correct, one level of the tree at a time.

Deletion Still Respects `PROTECT`

Chapter 2's own `on_delete=PROTECT` decision surfaces directly in the admin: attempting to delete a page that still has children raises a clear, real error rather than silently cascading — Django's admin handles a `PROTECT` violation gracefully out of the box, showing exactly which related objects are blocking the deletion, rather than a raw server error.

Hands-On Exercises

EXERCISE 1

Explain why Django's free automatic admin, even after adding `raw_id_fields` to the `parent` field, still isn't sufficient for genuinely browsing or managing this project's own tree structure.

View solution

EXERCISE 2

Explain why `move_page`'s own cascade loop processes descendants in parent-first (breadth-first) order rather than in any order, and why it never manually edits a `full_path` string anywhere.

View solution

EXERCISE 3

Explain what happens, per Chapter 2's own `on_delete=PROTECT` decision, if an admin tries to delete a page that still has children — and why that's the safer default for this specific project.

View solution

CHAPTER 10 QUICK REFERENCE

- `admin.site.register(Page)` — free, working CRUD, one line, no custom code
- **The free admin's real limit** — the `parent` dropdown is a flat, unstructured list, unusable for browsing a real tree
- `raw_id_fields` — a cheap improvement (a search popup instead of a giant select), still not a browsable tree
- `move_page` — pays off Chapter 2's own deferred full_path-cascade cost, with zero manual string manipulation
- **Parent-first, breadth-first order** — the only real trick; each page's `save()` only produces the right result once its own parent is already correct
- `on_delete=PROTECT`, revisited — the admin surfaces a real, clear error rather than a silent cascade or a raw server crash
- **Next chapter:** Deployment

Chapter 5 promised this moment would come back around: flip `DEBUG` to `False`, and static files quietly stop being served. This chapter is where that promise gets kept, alongside everything else a real production deployment actually needs — reusing the same lessons Food Tracker (Django) 12 already worked through, and landing on a genuinely different deployment story than Website Rebuild with Next.js 11's own Vercel-or-self-hosted split.

`manage.py runserver` Was Never for Production

Every chapter so far has used `runserver` — and Django's own documentation is explicit that it was never meant to handle real production traffic: no real concurrency, no process management, not hardened for the open internet.

`gunicorn`: Django's Own Production WSGI Server

```
gunicorn osztromok_site.wsgi:application --bind 127.0.0.1:8000 --workers 3
```

`gunicorn` is a real WSGI application server — multiple worker processes, genuine concurrency, built to sit behind a proper reverse proxy rather than face the internet directly.

`DEBUG=False`: Two Things Happen at Once

The first is safety: `DEBUG=True` shows a detailed error page on any unhandled exception — full traceback, source code, local variable values, even settings — genuinely dangerous information to hand any random visitor who happens to trigger an error. The second, per Chapter 5, is that Django's own automatic static file serving stops working entirely the moment `DEBUG=False` — two unrelated-sounding consequences from one single setting.

`ALLOWED_HOSTS`: Required the Moment `DEBUG=False`

```
ALLOWED_HOSTS = ['osztromok.com', 'www.osztromok.com']
```

Django refuses any request whose `Host` header isn't in this list — a real defense against HTTP Host-header attacks, and a genuinely common "why is production suddenly returning 400 Bad Request" surprise for anyone who forgets to set it before their first production deploy.

`SECRET_KEY`: Never the Auto-Generated Dev Value

```
import os
SECRET_KEY = os.environ['DJANGO_SECRET_KEY']
```

`SECRET_KEY` signs session cookies and every CSRF token Chapter 8's own `{% csrf_token %}` generates — real security infrastructure, not a cosmetic setting, and it belongs in an environment variable, never hardcoded and committed to a repository alongside the code.

> **ROTATING SECRET_KEY LOGS OUT EVERY ADMIN, IMMEDIATELY**
>
> Because `SECRET_KEY` is what actually signs session data, changing it invalidates every existing session at once — including Chapter 9's own imported admin account. This isn't a bug to chase down; it's a direct, expected consequence of what the key actually does. A surprise "why am I suddenly logged out" moment the first time it happens, not a sign anything went wrong.

Finally Resolving Chapter 5's Own Forward Reference

```
python manage.py collectstatic --noinput
# gathers every app's own static/ folder — including Chapter 5's
content/static/content/...
# and Chapter 7's animcjk/ assets — into one STATIC_ROOT, ready for nginx to serve
directly
```

nginx as Reverse Proxy

```
server {
    listen 80;
    server_name osztromok.com;

    location /static/ {
        alias /path/to/staticfiles/;
    }

    location / {
        proxy_pass http://127.0.0.1:8000;
        proxy_set_header Host $host;
    }
}
```

nginx serves everything under `/static/` directly from `STATIC_ROOT` — no Python involved at all for a plain file — and forwards everything else to `gunicorn`. The same reverse-proxy pattern this site's own Web Servers Fundamentals and Nginx In Depth courses already cover in real depth, applied here rather than re-taught from scratch.

No Django-Specific Zero-Config Platform

A GENUINE, HONEST DIFFERENCE FROM NEXT.JS

Vercel is built by the Next.js team, specifically for Next.js — near-zero-config deployment is possible precisely because the platform and the framework share an owner. Django has no equivalent: no company-run platform built specifically around Django's own conventions the same way. Django deployment is always closer to the traditional self-hosted pattern shown here (or a general-purpose PaaS like Render or Railway, which supports Django well but wasn't built *for* it specifically) — which, honestly, mirrors this project's own already-live PHP/Apache reality more closely than Next.js's own Vercel path ever could.

	NEXT.JS	DJANGO
Zero-config platform	Vercel — built by the same team, for this exact framework	None — no Django-specific equivalent exists
Self-hosted path	PM2 + nginx, covered as the alternative to Vercel	gunicorn + nginx — the only real path shown here

Hands-On Exercises

EXERCISE 1

Explain why `manage.py runserver` is explicitly not meant for production use, and what `gunicorn` provides that it doesn't.

View solution

EXERCISE 2

Explain the two separate things that happen the moment `DEBUG=False` is set, and why leaving `DEBUG=True` active in production would be a real security problem.

View solution

EXERCISE 3

Explain why rotating `SECRET_KEY` immediately logs out every currently-logged-in admin, including the account imported back in Chapter 9.

View solution

CHAPTER 11 QUICK REFERENCE

- `gunicorn` — the real production WSGI server; `runserver` is explicitly dev-only
- `DEBUG=False` — disables both the detailed error pages (a real information leak) and automatic static file serving (Chapter 5's own forward reference)
- `ALLOWED_HOSTS` — required once `DEBUG=False`; rejects unrecognized `Host` headers
- `SECRET_KEY` — signs sessions and every CSRF token; belongs in an environment variable, never hardcoded
- **Rotating `SECRET_KEY` logs everyone out** — a direct, expected consequence, not a bug
- `collectstatic` — gathers every app's static files into `STATIC_ROOT`, finally resolving Chapter 5's own forward reference
- **nginx + gunicorn** — the same reverse-proxy pattern this site's own Web Servers courses already teach in depth
- **No Vercel-equivalent for Django** — a genuine, honest structural difference, not a gap in this course's own coverage
- **Next chapter:** Capstone — A Complete, Working Flexible-Routing Site

Website Rebuild with Django

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

Sam is the site's own admin — the same real person who worked through Website Rebuild with Next.js's own capstone, now doing the identical job against this Django rebuild instead. Every chapter since Chapter 1 gets used here, in the order a genuine admin session would actually reach them, building the exact five-level-deep chain this whole course was built around from the very first paragraph — no longer a hypothetical example.

STEP 1 — LOGGING IN

Sam visits `/admin-login/` and signs in with the same password the old PHP site always used — Chapter 9's own bcrypt-compatibility work means nothing had to be reset. This is the first real login since that chapter shipped, and it silently upgrades Sam's own stored hash to Django's preferred PBKDF2 format, exactly as promised.

STEP 2 — BUILDING THE REAL CHAIN

Through the admin (Chapter 10's own `raw_id_fields`-improved parent picker), Sam creates five real pages, one level at a time:

```
programming = Page.objects.create(title="Programming", slug="programming")
gpl = Page.objects.create(title="General-Purpose Languages", slug="general-
purpose-languages", parent=programming)
java = Page.objects.create(title="Java", slug="java", parent=gpl)
fundamentals = Page.objects.create(title="Java Fundamentals",
slug="fundamentals", parent=java)
chapter1 = Page.objects.create(title="Chapter 1: Getting Started",
slug="chapter-1", parent=fundamentals)

chapter1.full_path
# 'programming/general-purpose-languages/java/fundamentals/chapter-1'
```

No special case anywhere — Chapter 2's own model handles five levels exactly as easily as one.

STEP 3 — VISITING IT LIVE

Sam opens `/programming/general-purpose-languages/java/fundamentals/chapter-1/` directly. Chapter 3's own `<path:full_path>` catches the whole thing in one pattern; `rstrip('/')` normalizes it; one indexed lookup against Chapter 2's own `full_path` field finds the row — no walk, no five-level chase. Chapter 4's breadcrumb renders all five ancestors correctly, and Chapter 5's dark theme wraps the whole page.

STEP 4 — REORGANIZING IT, AND PAYING THE REAL CASCADE COST

Sam decides "General-Purpose Languages" belongs under a new top-level "Languages" page instead of directly under "Programming" — the exact kind of move Chapter 2 deliberately deferred, and Chapter 10 actually built:

```
languages = Page.objects.create(title="Languages", slug="languages")
move_page(gpl, languages)

chapter1.refresh_from_db()
chapter1.full_path
# 'languages/general-purpose-languages/java/fundamentals/chapter-1'
```

`move_page` was only ever called with `gpl` — `chapter1` itself was never directly touched. Its own `full_path` is still correct anyway, three levels below the page that actually moved, because Chapter 10's own breadth-first cascade walked all the way down through `java` and `fundamentals` before ever reaching it. This is the real, working payoff of a cost Chapter 2 named honestly and left open eight chapters ago.

STEP 5 — CONFIRMING THE KANJI MIGRATION HELD UP

Sam visits `/japanese/kanji/水/`. Chapter 7's `allow_unicode=True` slug resolves correctly through Chapter 3's own UTF-8-safe `path` converter — no special-casing needed anywhere in the routing layer. The stroke-order animation renders from the self-hosted `static/content/animcjk/` assets Chapter 5's own convention organized, through the exact same `{{ page.body|safe }}` trust boundary every other page already uses.

STEP 6 — A SAFE EDIT

Sam notices a typo and fixes it through `update_page_title` — Chapter 8's own mutation, now genuinely protected by Chapter 9's `@staff_member_required`. A stranger who found this same URL months ago, back when Chapter 8 first shipped it, could have renamed anything; today, they can't.

STEP 7 — SHIPPING IT

`collectstatic` gathers every app's own static files — Chapter 5's stylesheet, Chapter 7's stroke-animation assets — into one `STATIC_ROOT`. `gunicorn` runs the real application; nginx serves `/static/` directly and proxies everything else. `DEBUG=False`, `ALLOWED_HOSTS` set, `SECRET_KEY` read from the environment — Chapter 11's own checklist, complete.

Chapter Attribution

STEP	CHAPTER(S) APPLIED
1 — Logging in	Chapter 9 (bcrypt-compatible auth, automatic hash upgrade)
2 — Building the chain	Chapter 2 (the self-referencing model), Chapter 10 (the admin interface)
3 — Visiting it live	Chapter 3 (routing), Chapter 4 (breadcrumb/templates), Chapter 5 (styling)
4 — Reorganizing	Chapter 2 (the deferred cost), Chapter 10 (`move_page`, the real payoff)
5 — Kanji check	Chapter 7 (the edge case resolution)
6 — Safe edit	Chapter 8 (the mutation), Chapter 9 (the fix that protects it)
7 — Shipping it	Chapter 11 (deployment)

WHAT THIS WHOLE COURSE WAS REALLY ABOUT

Every single step in Sam's session traces back to a specific, real decision made earlier in this course — nothing here is arbitrary or newly introduced just for the capstone. Chapter 2's own read-heavy/write-rare design 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 `move_page`. Chapter 1's own "batteries-included" observation about `INSTALLED_APPS` becomes Chapter 9's real 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 staff flag, revision history or an undo mechanism for content edits, media/image uploads, an automated test suite, internationalization beyond kanji's own `allow_unicode` slug support, or real search ranking beyond a plain title/path 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 3 only requires one database lookup, rather than five, tracing the answer back to a specific design decision made in Chapter 2.

View solution

EXERCISE 2

Explain why `chapter1.full_path` is already correct after Step 4's `move_page(gpl, languages)` call, even though `chapter1` was never passed into that function directly.

View solution

EXERCISE 3

Explain why Step 6's page-title edit is described as something "a stranger could have done months ago" but not today. What specifically changed between Chapter 8 and Chapter 9 to make that true?

View solution

CHAPTER 12 QUICK REFERENCE — COURSE COMPLETE

- **7 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 batteries-included auth, the one-lookup routing — 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 Django is now complete** — 12/12 chapters