



PROJECT COURSE

Learning Website with Django

Building a family of subdomain sites from one Django project: host-based routing, importing content from files, a shared design system, navigation, search and sitemaps, a read-only admin and optional logins, migrating from the old site with reviewed redirects, and deployment with backups and a way back. Every chapter is backed by code run on the real content.

12 Chapters

Philip Osztromok

Generated with Claude

Contents

Chapter 1	Project Layout for Many Sites	2
Chapter 2	Subdomain Routing	9
Chapter 3	The Content App	15
Chapter 4	Loading Content from Files	21
Chapter 5	Templates & the Shared Design System	27
Chapter 6	Navigation & Breadcrumbs	33
Chapter 7	The Languages Site	38
Chapter 8	PDFs, Solutions & Static Files	43
Chapter 9	Search & Sitemaps	49
Chapter 10	Admin, Accounts & Progress	55
Chapter 11	Migrating from the Old Site	61
Chapter 12	Deployment, Testing & Operations	67

Learning Website with Django

Chapter 1 · Project Layout for Many Sites

Learning Website: Framework & Architecture 1 to 12 decided *what* the family of sites should be. This course builds it with Django, starting with the languages site. Before any page exists, one decision shapes everything that follows: how should one Django codebase serve eight sites? This chapter makes that decision, lays out the project, and proves the layout runs. You will need Python and a basic knowledge of Django (Django Fundamentals covers it); the course uses Django 6.1.

EVERYTHING HERE WAS RUN

The project in this chapter was created and tested for real, with Django 6.1.2 on Python 3.14. The scripts, test files and outputs are in the exercise solutions, and the numbers quoted below come from those runs.

Where to Build It

The new site is its own project, separate from the folder of course content. Create the project in the `learning-website` folder (or any new folder of your choice) and keep the content where it is. The project finds the content through a setting, `CONTENT_ROOT`, which defaults to a `content` folder beside `manage.py` and can point anywhere with the `LW_CONTENT_ROOT` environment variable. That way the content can stay in its current home until you decide to move it.

Decision 1: How Many Projects, How Many Processes?

OPTION	HOW IT WORKS	TRADE-OFF
A. One project, one process, site chosen from the host name	All eight sites run in the same Django process. The <code>Host</code> header says which site a request is for.	One thing to deploy and monitor. A bug or a crash affects every site.
B. One project, one process per site	The same code runs eight times, each pinned to one site.	Strong isolation and separate restarts, but eight services to manage and eight times the memory.

OPTION	HOW IT WORKS	TRADE-OFF
C. One Django project per site	Eight codebases.	Maximum independence, and the shared parts (design, navigation, importer) get copied eight times. Rejected.

The layout chooses A and keeps B available. A setting called `LW_SITE` pins a process to a single site. Leave it unset and one process serves every site; set it to `languages` and that process answers only to `languages.osztromok.com`. You can start simple and split later without changing any code. A name that is not one of the eight sites stops the start-up with a clear message, so a typo can never serve the wrong site.

Decision 2: `django.contrib.sites` or a Config File?

Django ships a sites framework, and it is the obvious first thought. Exercise 3 tries it and the plain alternative side by side. The results:

	CONTRIB.SITES	CONFIG FILE (THE SITE MAP)
Where the sites are defined	Rows in a database table	One Python file, in the code
A fresh database	Contains a site called <code>example.com</code> that you must replace	Nothing to set up
Host lookup	1 database query (then cached)	0 queries
An unknown host name	Raises <code>Site.DoesNotExist</code> (a server error unless caught)	Gives <code>None</code> , and you choose the response
Copies of the site list	Two: the code and the table, which can disagree	One

The course uses the config file. The content comes from files, the list of sites already exists in one place, and no database is needed to know which site a request is for. Django's own documentation says the sites framework requires a database, and a site table you must keep in step with the code buys nothing here. If you later add database models that belong to one site, or use a Django feature built on it (such as the flatpages app), you can add `django.contrib.sites` then. Nothing in this layout prevents it.

The Layout

```
learning-site/
├─ manage.py
├─ config/
│   ├─ sites_config.py      # the ONE list of sites and their folders
│   ├─ urls.py
│   ├─ wsgi.py
│   └─ settings/
│       ├─ base.py          # shared by every environment
│       ├─ dev.py           # DEBUG on, *.localhost hosts, a throwaway key
│       └─ prod.py          # DEBUG off, exact hosts, secure cookies, key from the env
├─ apps/
│   ├─ core/                # shared helpers and middleware           (Chapter 2)
│   ├─ content/             # pages imported from the content tree (Chapters 3 and 4)
│   ├─ theme/               # shared templates, static, tokens       (Chapter 5)
│   └─ navigation/          # menus and breadcrumbs                 (Chapter 6)
├─ templates/
├─ static/
├─ tests/
└─ requirements/
    ├─ base.txt
    ├─ dev.txt
    └─ prod.txt
```

The skeleton is 25 small files that Exercise 1 writes for you. Three habits keep it healthy as it grows:

- **One app, one job.** Each app is named for the chapter that fills it in. Site-specific code (the languages site, in Chapter 7) gets its own app later, so the shared apps never fill up with `if site == "languages":`.
- **The site map is data, not logic.** Settings, menus, redirects and sitemaps all read `sites_config.py`. A change of mind is a one-line edit.
- **Settings are a package.** `base.py` holds what never changes; `dev.py` and `prod.py` start with `from .base import *` and add only the differences.

The Settings That Matter Most

In a multi-site project a small settings mistake is expensive, because it affects every site at once. These are the ones to get right from day one:

SETTING	VALUE HERE	WHY
<code>SECRET_KEY</code>	Dev: a throwaway string. Prod: read from <code>DJANGO_SECRET_KEY</code> , with no default.	Production refuses to start without it, rather than starting with a guessable key.
<code>ALLOWED_HOSTS</code>	The exact host names from the site map (for example <code>languages.osztromok.com</code>)	No <code>*</code> . Django's documentation allows a leading-dot pattern like <code>.osztromok.com</code> , but exact names refuse any other host.
<code>CSRF_TRUSTED_ORIGINS</code>	<code>https://</code> plus each host	Django requires the scheme in this setting.
<code>SESSION_COOKIE_DOMAIN</code>	<code>None</code>	Each site keeps its own login cookie. Sharing one across subdomains is a deliberate decision (Learning Website: Framework & Architecture 9).
<code>SECURE_HSTS_SECONDS</code>	<code>3600</code>	Browsers remember HSTS, so start with one hour and raise it in steps.
<code>SECURE_PROXY_SSL_HEADER</code>	<code>("HTTP_X_FORWARDED_PROTO", "https")</code>	Apache ends the HTTPS connection and passes the request on, so Django must trust the header it adds.

```
# config/sites_config.py (excerpt)
def site_hosts(env, only=None):
    names = [only] if only else list(SITES)
    suffix = DOMAIN if env == "prod" else "localhost"
    return [f"{name}.{suffix}" for name in names]

# config/settings/prod.py (excerpt)
SECRET_KEY = os.environ["DJANGO_SECRET_KEY"] # required: no default
ALLOWED_HOSTS = site_hosts("prod", PINNED_SITE)
CSRF_TRUSTED_ORIGINS = [f"https://{host}" for host in ALLOWED_HOSTS]
```

TWO DEPLOYMENT WARNINGS THAT STAY ON PURPOSE

Django's `check --deploy` on the production settings is clean except for two warnings. One asks for `SECURE_HSTS_INCLUDE_SUBDOMAINS`, which would tell browsers that every subdomain must use HTTPS. Do not turn it on until every subdomain really does, including sites you have not migrated yet. The other asks for HSTS preload, which is very hard to undo. Both are decisions for later, not oversights.

Environment Variables

VARIABLE	PURPOSE	REQUIRED?
<code>DJANGO_SETTINGS_MODULE</code>	<code>config.settings.dev</code> or <code>config.settings.prod</code>	<code>manage.py</code> defaults to dev; <code>wsgi.py</code> defaults to prod
<code>DJANGO_SECRET_KEY</code>	Signing key in production	Production only
<code>LW_SITE</code>	Pin the process to one site	No
<code>LW_CONTENT_ROOT</code>	Where the content tree lives	No (default: <code>content/</code> beside <code>manage.py</code>)

Secrets never go in the repository: `.env` is in `.gitignore`, and the production key comes from the server's environment (Learning Website: Framework & Architecture 8). A key committed to git has to be treated as public.

A Caveat About `*.localhost`

The dev settings answer to `languages.localhost`, `systems.localhost` and so on, so you can run the sites on your own machine under their real-looking names. Django's own default for debug mode also accepts `.localhost`. Most current browsers send any name ending in `.localhost` to your own computer. But the operating system's resolver may not: on the Windows machine used to write this chapter, Python's `socket.gethostbyname("languages.localhost")` failed even though `localhost` itself worked. So a browser may work while `curl` or a script fails. Django's test client sends the host name itself (`HTTP_HOST`), so the tests are unaffected. Chapter 2 deals with local development in detail, including hosts-file entries and `curl --resolve`.

Start With Tests

The settings are covered by tests from the first day (Exercise 2): every folder belongs to one site, hosts are exact names with no wildcard, a pinned site gives exactly one host, an unknown site name is rejected, and the production settings have an https origin for each host and refuse to load without a secret key. The ten tests run in a few hundredths of a second. One of them failed on its first run, because Django's test runner adds `testserver` to `ALLOWED_HOSTS`; the fix was to leave that name out of the comparison, not to loosen the test.

Hands-On Exercises

EXERCISE 1

Create the project layout from this chapter as real files with a script, in a virtual environment with Django 6.1 installed. Run `manage.py check`, then again with `LW_SITE=languages` and with a nonsense value, and say what each shows.

 [View solution](#)

EXERCISE 2

Write tests for the site map and the settings (unique folders, exact hosts, pinning, https origins, the secret key), run them, and then run `check --deploy` on the production settings. Fix what you can and explain any warning you leave.

 [View solution](#)

EXERCISE 3

Compare `django.contrib.sites` with a plain config lookup: migrate a fresh database, add the eight sites, find the site from a request host with each method, try an unknown host, and count database queries. Decide which to use and why.

 [View solution](#)

CHAPTER 1 QUICK REFERENCE

- **One project, one codebase;** one process for all sites by default, or pin a process to one site with `LW_SITE`
- Sites come from **one config file** (`config/sites_config.py`), not `django.contrib.sites`: 0 queries, one copy of the list, and unknown hosts are yours to handle
- Layout: `config/` (settings package, site map, URLs), `apps/` (core, content, theme, navigation), `templates/`, `static/`, `tests/`, `requirements/`
- Settings: no default secret key in production; exact `ALLOWED_HOSTS`; `CSRF_TRUSTED_ORIGINS` with `https://`; `SESSION_COOKIE_DOMAIN = None`
- HSTS: start at 3600 seconds; leave include-subdomains and preload off until every subdomain is on HTTPS
- Environment variables: `DJANGO_SETTINGS_MODULE`, `DJANGO_SECRET_KEY`, `LW_SITE`, `LW_CONTENT_ROOT`
- `*.localhost` may work in a browser but not in the OS resolver: use hosts-file entries or `curl --resolve` (Chapter 2)
- Create the virtual environment in a short path on Windows (the 260-character limit broke the Django install)
- Test the settings from day one; when a test fails, read the difference before changing either side

Learning Website with Django

Chapter 2 · Subdomain Routing

Chapter 1 gave the project a layout and a list of sites. This chapter makes the project *answer* as those sites: a request for `languages.osztromok.com` must reach the languages pages, and a request for `systems.osztromok.com/hungary/` must not. The mechanism is small, a middleware and one URL configuration per site, but it sits underneath every later chapter, so it is worth building carefully and testing hard.

RUN FOR REAL, ON DJANGO 6.1

The code in this chapter was added to the Chapter 1 project, covered by 24 tests (all passing), and tried with the real development server and `curl`. The Apache and Gunicorn part could not be run on the machine used to write it, and is marked as such.

How a Request Becomes a Site

browser asks for	<code>https://languages.osztromok.com/hungary/hungarian-basic-3/</code>	
1. DNS	<code>languages.osztromok.com</code>	-> the server's address
2. Apache	picks the virtual host by name, ends HTTPS, passes the request on	
3. Gunicorn	runs Django (one process for all sites, or one pinned to a site)	
4. Django	<code>ALLOWED_HOSTS</code> must contain the host name	(otherwise 400)
5. Middleware	host name -> site "languages"	(<code>request.site</code>)
6. URL config	the languages site's own routes	(<code>request.urlconf</code>)
7. View	builds the page for that path	

Steps 4 to 6 are Django and are the subject of this chapter. Steps 2 and 3 are the last section.

The Site Middleware

A **middleware** is a small class that wraps every request. This one looks at the host name, decides the site, and attaches three things to the request:

```
class SiteMiddleware:
    def __call__(self, request):
```

```

host = request.get_host().split(":")[0].lower()      # get_host() enforces ALLOWED_HOSTS
site = settings.HOST_TO_SITE.get(host)
if site is None:
    return HttpResponseNotFound("Unknown site", content_type="text/plain")
request.site = site                                # "languages"
request.site_info = SITES[site]                     # title and folders
request.urlconf = f"config.urlconfs.{site}"        # this site's routes
return self.get_response(request)

```

- `request.get_host()` **does two jobs.** It returns the host name, and it checks it against `ALLOWED_HOSTS`, so an unknown name is refused by Django with a 400 before any of your code runs.
- **Strip the port and lower-case the name** before the lookup: `Languages.Localhost:8000` and `languages.localhost` are the same site.
- **The lookup table comes from the site map.** `HOST_TO_SITE` is built by one function in `sites_config.py`, `host_map(env)`, so the settings and the middleware cannot disagree.
- **Order matters.** The middleware must be listed *before* `CommonMiddleware`, which reads `request.urlconf` (for example when it adds a missing slash).

One URL Configuration per Site

Django normally has one list of routes, named by `ROOT_URLCONF`. But if a middleware sets `request.urlconf`, Django uses that module for the request instead. So each site can have its own routes, built from the site map: the languages site has routes for `france`, `germany`, `hungary`, `japan` and `culture`, and nothing else.

```

def build(site):
    folders = "|".join(re.escape(f) for f in SITES[site]["folders"])
    module = types.ModuleType(f"config.urlconfs.{site}")
    module.urlpatterns = [
        re_path(r"^$", views.home, {"site": site}, name="home"),
        re_path(rf"^(?P<page_path>(?:{folders})(?:/.*)?)$", views.page, {"site": site},
    ]
    sys.modules[module.__name__] = module          # so Django can import it by name
    return module

```

The result is that the *routing itself* enforces the site boundaries. There is no `if site == "languages"` in any view: `/linux/shell-and-scripting/vim/` on the languages site matches no route, so it is a 404, and the same path on the systems site matches and is served. A test checks every folder of every site against every other site (Exercise 1).

ANCHOR THE FOLDER PATTERN

The pattern ends with `(?:/.*)?$`, which means “the folder, then optionally a slash and more”. A looser pattern that only matched the start of the path would let `/hungary-extra/` through as if it were the `hungary` folder. One test exists for exactly that case.

Two Kinds of “Unknown”

REQUEST	RESULT	WHO ANSWERS
A host name that is not in <code>ALLOWED_HOSTS</code>	400	Django, before the middleware
An allowed host with no site (should not happen)	404, “Unknown site”	The middleware
A known site, a folder it does not have	404	That site's URL configuration
A known site, a page that does not exist	404	The view (Chapters 3 and 4)

The root domain, `osztromok.com` and `www`, is not mapped yet: it will become the landing page after the areas have moved, and until then the old site keeps answering for it. A request for it reaching this project would get the 400.

One Process or One per Site

With `LW_SITE` unset, one process serves every site. With `LW_SITE=languages`, the process narrows both `ALLOWED_HOSTS` and `HOST_TO_SITE` to that one site. Real requests to both, from Exercise 2:

REQUEST	ALL-SITES PROCESS	PINNED TO LANGUAGES
<code>languages.localhost</code> , path <code>/</code>	200	200
<code>languages.localhost</code> , path <code>/hungary/x/</code>	200	200

REQUEST	ALL-SITES PROCESS	PINNED TO LANGUAGES
<code>systems.localhost</code> , path <code>/hungary/x/</code>	404 (no such route on that site)	400 (host not allowed)
Host <code>evil.example.org</code>	400	400
Host <code>127.0.0.1:8765</code> (no proxy host name)	400	400

Names on Your Own Machine

To try a named site locally you need the name to reach your machine. The obvious idea is `languages.localhost`, and **it cannot be relied on**. On the Windows machine used to write this course, Python could not resolve it, and on Ubuntu 24.04 under WSL, `getent hosts languages.localhost` returned nothing, although `localhost` worked. Some browsers resolve `*.localhost` themselves, but `curl`, scripts and other tools use the system resolver. Whether it works on Debian depends on the name-service modules installed, so check on your own development machine before relying on it.

OPTION	HOW
1. The hosts file (reliable)	Add one line, <code>127.0.0.1 languages.localhost systems.localhost ...</code> , to <code>/etc/hosts</code> . A five-line script prints it from the site map.
2. <code>curl --resolve</code>	<code>curl --resolve languages.localhost:8000:127.0.0.1 http://languages.localhost:8000/</code> for a quick test with no file edited.
3. <code>curl -H "Host: ..."</code>	<code>curl -H "Host: languages.localhost" http://127.0.0.1:8000/</code> sends the name but connects by address.

Behind Apache and Gunicorn

In production the request reaches Django through two other programs. Django cannot see the original request for itself, so three things must be passed on correctly:

WHAT DJANGO NEEDS TO KNOW	HOW THE PROXY TELLS IT	DJANGO SETTING
The host name the visitor used	Apache: <code>ProxyPreserveHost On</code>	None (the default: use the <code>Host</code> header)

WHAT DJANGO NEEDS TO KNOW	HOW THE PROXY TELLS IT	DJANGO SETTING
Whether the visitor used HTTPS	Apache: <code>RequestHeader set X-Forwarded-Proto "https"</code>	<code>SECURE_PROXY_SSL_HEADER = ("HTTP_X_FORWARDED_PROTO", "https")</code>
Which headers to trust	Trust as few as possible	Leave <code>USE_X_FORWARDED_HOST</code> off

- **Without** `ProxyPreserveHost On` Django would see the address Apache used to reach it, not the visitor's host name. The real request would be refused with a 400, exactly like the `"127.0.0.1"` row above.
- **The HTTPS header is trusted only when you say so.** A test shows Django ignores `X-Forwarded-Proto` by default and believes it once `SECURE_PROXY_SSL_HEADER` is set. That is also what stops `SECURE_SSL_REDIRECT` from looping: Django finally sees that the request was HTTPS.
- `USE_X_FORWARDED_HOST` **is a trap.** With it on, a visitor can pick the site by sending an `X-Forwarded-Host` header. A test shows a request addressed to `languages.localhost` being served as the systems site.

The Gunicorn service and the Apache virtual host are in the Exercise 3 solution. They were **not** run here, because neither program is installed on the machine used to write the course. Test them on the server with `apache2ctl configtest` and `curl -I` before you rely on them.

Hands-On Exercises

EXERCISE 1

Add subdomain routing to the Chapter 1 project: a `host_map()` function, a site middleware (placed before `CommonMiddleware`), one URL configuration per site built from the site map, placeholder views, and tests that check each site on its own host, the port and case handling, unknown hosts, and every folder against every other site.

 [View solution](#)

EXERCISE 2

Start the development server twice (all sites, and pinned to one) and request both with `curl` using `--resolve` and a custom `Host` header. Record the status of each request, then write a script that prints the hosts-file line for your development machine.

 [View solution](#)

EXERCISE 3

Write tests for what Django does behind a proxy (the HTTPS header, the redirect loop, the forwarded host), then write the Gunicorn service and the Apache virtual host, and a checklist for testing them on the server.

 [View solution](#)

CHAPTER 2 QUICK REFERENCE

- A request becomes a site in three steps: `ALLOWED_HOSTS` (400 if unknown), the middleware (host to site), the site's own URL configuration
- The middleware sets `request.site`, `request.site_info` and `request.urlconf`; list it **before** `CommonMiddleware`
- `request.get_host()` also enforces `ALLOWED_HOSTS`; strip the port and lower-case before looking the host up
- Each site's URL configuration is built from the site map, so the **routing enforces the site boundary** (`/linux/...` is a 404 on the languages site)
- Anchor the folder regex: `(?:/.*)?$`, so `/hungary-extra/` does not match `hungary`
- `LW_SITE` pins a process: `ALLOWED_HOSTS` shrinks to one name, so another site's host gets a 400
- Do not rely on `*.localhost`: use hosts-file entries, `curl --resolve` or `curl -H "Host: ..."`
- Behind Apache: `ProxyPreserveHost On`, `X-Forwarded-Proto` with `SECURE_PROXY_SSL_HEADER`, and `USE_X_FORWARDED_HOST` left off
- The Apache and Gunicorn configuration was not run here: test with `apache2ctl configtest` and `curl -I`

Learning Website with Django

Chapter 3 · The Content App

The project can now tell which site a request is for. It cannot yet tell what pages exist. This chapter builds the `content` app: the models that describe a page and a course, and the code that reads a content file and works out those facts from the file itself. It does not import anything yet; Chapter 4 does that. The order is deliberate: first teach the project what a page *is*, then run it over thousands of real ones.

RUN FOR REAL, ON DJANGO 6.1

The app has 53 tests, all passing, and was run over every page of the real content folder in a dry run (Exercise 3). The numbers quoted below are from that run. The page counts differ from the framework course because that course also counted directory index pages; this chapter counts content files.

What the Database Is For

The pages themselves are files, and they stay files: you write them, you commit them, and the PDF builders read them. The database does not replace that. It holds what the site must *query* quickly and repeatedly:

QUESTION THE SITE ASKS	ANSWERED FROM
Which pages are on this site?	The database (site, kind)
What are the chapters of this course, in order?	The database (course, chapter number)
What was updated recently?	The database (updated date)
What does this page say?	The file (Chapter 4 decides how it is read)

A page's identity is its **path**, relative to the content folder, with forward slashes:

`hungary/hungarian-basic-3/hungarian_basic_conversation_3_1.html`. It is unique, it is stable, and the page's address is just that path without `.html`. Nothing needs a separate id.

The Two Models

MODEL	FIELD	NOTES
Course	<code>site</code> , <code>folder</code>	The folder is unique, such as <code>hungary/hungarian-basic-3</code>
	<code>name</code>	From the banner's <code>Course:</code> line
	<code>course_no</code>	Optional
	A course is just a folder of numbered chapters	
Page	<code>path</code>	Unique, relative, forward slashes, ends in <code>.html</code> , never contains <code>..</code>
	<code>site</code> , <code>kind</code>	The site comes from the path; the kind is one of five (below)
	<code>title</code>	Found in the file (below)
	<code>course</code> , <code>chapter_no</code>	Optional; deleting a course keeps its pages
	<code>created</code> , <code>updated</code>	Optional: many older pages have no dates
	Ordered by course, then chapter <i>number</i> , then path	

```

class Page(models.Model):
    class Kind(models.TextChoices):
        COURSE_CHAPTER = "course_chapter", "Course chapter"
        SIDEBAR = "sidebar", "Sidebar page"
        LESSON = "lesson", "Standalone lesson or reference"
        FULL_PAGE = "full_page", "Complete HTML page"
        OTHER = "other", "Other"

    path = models.CharField(max_length=500, unique=True, validators=[validate_content_pa
    site = models.CharField(max_length=40, choices=SITE_CHOICES, db_index=True)
    ...
    class Meta:
        ordering = ["course_id", "chapter_no", "path"]    # chapter_no is a NUMBER

```

Ordering by a number matters. A test inserts chapters 10, 2, 1 and 11 and reads them back as 1, 2, 10, 11. A sort on file names would give 1, 10, 11, 2, a trap from Learning Website: Framework & Architecture 3.

Keeping the Table Honest

- **Unsafe paths are refused.** A path must be relative, use forward slashes, end in `.html` and contain no `..`. Four bad examples are tested, so a path can never point outside the content folder.
- **The site must agree with the path.** `clean()` looks the path up in the site map and rejects a row that says `systems` for a `hungary/...` path. The table cannot contradict the map.
- **A path in no site is an error.** The lookup, `site_for_path()`, raises `NoSiteError` instead of guessing, and the importer will report it.
- **Sidebar pages follow their subject.** `sidebar/linux/...` belongs to the systems site, as decided in Learning Website: Framework & Architecture 2.

Django validates these rules in `full_clean()`, not on every `save()`. The importer will call `full_clean()` explicitly, so a bad row is reported, not stored.

Reading a File

The parser takes a file's text and its path and returns every value a `Page` needs. It starts from the banner shapes catalogued in Learning Website: Framework & Architecture 3, and the tests use a real file of each shape:

BANNER SHAPE	KIND STORED	EXAMPLE PARSED
Course and Chapter, with dates	<code>course_chapter</code>	Hungarian Basic Conversation 3, chapter 1, dated 2026-10-05
Course and Chapter, no dates (older)	<code>course_chapter</code>	Apache In Depth, chapter 1: dates stay empty
Category and Subcategory	<code>sidebar</code>	The Git command line cheat sheet, on the programming site
Free-form headline	<code>lesson</code>	The Python boolean logic lesson: title from its first heading
Free-form, no title line	<code>lesson</code>	The Hungarian alphabet reference: title from the file name
A complete HTML page	<code>full_page</code>	A kanji page: title from <code><title></code>
No banner	<code>other</code>	Seven pages in the real content (below)

The title follows the same order as the live site: the banner's `Chapter:` line, then the page's own `<title>`, then the first `<h1>` or `<h2>`, then the file name made readable. Course chapters have no real heading of their own, so the banner is the only place their title lives. Entities such as `&` are decoded, so a title reads "Fit & Returns", not "Fit & Returns".

Chapter numbers come from the file name, and the names are not uniform

Course and chapter numbers are not in the banner, so the parser reads them from the file name. The real content has several generations of names, and the first version of the parser handled only the newest. Counting the course-chapter pages by pattern shows how much would have been wrong:

PATTERN	EXAMPLE	PAGES	COURSE, CHAPTER
Newest	<code>hungarian_basic_conversation_3_10.html</code>	2,950	3, 10
Older prompt- prefix	<code>js1-4.html</code> , <code>psp-croll-1-2.html</code>	495	1, 4 and 1, 2
One trailing number	<code>setting_up_a_web_server_on_debian_03.html</code>	367	none, 3
Lesson, chapter first	<code>html_lesson_01_3.html</code>	62	3, 1 (the numbers are the other way round)
No number	<code>linux_appendix_a.html</code>	21 at first, 3 once the other forms were handled	none, none

THE REVERSED NAMES WOULD HAVE PASSED UNNOTICED

The HTML course's files, such as `html_lesson_01_3.html`, put the *chapter* first. A parser that always reads "course then chapter" gives every chapter of the advanced HTML course the number 3, and nothing complains. It was found only because the dry run checked that each course's chapters run 1 to N, and twelve of them came out as 3, 3, 3, 3... Check the numbers against a sequence, not just against "is it a number".

The Dry Run

Before any import, Exercise 3 parses every real page and reports what it finds, writing nothing to the database:

RESULT	VALUE
Pages parsed	4,402 of 4,402
By kind	3,895 course chapters, 410 complete pages, 47 sidebar, 43 lessons, 7 with no banner
By site	programming 1,370; humanities 707; systems 653; languages 561; webdevelopment 514; lifeskills 253; ai 187; creative 157
Course folders	395, with no folder using two different course names
Paths in no site	0: the site map is complete
Chapter numbers not running 1 to N	1 course (PHP Fundamentals, because its first chapter has no banner)
Banner <code>File:</code> line differs from the real file	7 pages, all in Setting Up a Web Server on Debian

SEVEN PAGES WITH NO BANNER

Four older lessons (two Hungarian and two Japanese), the hiragana and katakana tile fragments, and `php1-1.html` have no banner comment at all. They are real content problems, not parser problems, and `php1-1.html` is why PHP Fundamentals seems to start at chapter 2. Adding a banner to each is a small fix, listed in the review notes.

Hands-On Exercises

EXERCISE 1

Add the site lookup for a path, the banner parser and the page parser to the Chapter 2 project. Copy a real file of each banner shape into `tests/fixtures`, write tests for each shape and for every file-name pattern, and print what the parser finds in each sample.

 [View solution](#)

EXERCISE 2

Create the `Course` and `Page` models and their migration, and test them against a real database: one row per path, chapters in numeric order, unsafe paths and mismatched sites rejected, and a course deleted without losing its pages. Read the SQL Django generates.

 [View solution](#)

EXERCISE 3

Write a dry-run script that parses every page of the real content folder and reports the counts by kind and site, paths in no site, titles that fell back to the file name, banner file-name mismatches, and courses whose chapter numbers do not run 1 to N. Fix the parser for whatever it finds, then list the content problems that remain.

 [View solution](#)

CHAPTER 3 QUICK REFERENCE

- The files stay the source; the database holds what the site must **query**: pages by site, chapters in order, recent updates
- A page's identity is its **path** (relative, forward slashes, `.html`); its address is the path without `.html`
- Models: `Course` (a folder of chapters) and `Page` (path, site, kind, title, course, chapter number, dates)
- Order by the chapter **number**; a text sort puts 10 before 2
- Kinds: course chapter, sidebar, lesson, complete page, other; only fields that exist are filled in
- Title order: banner `Chapter:`, `<title>`, first h1/h2, then the file name
- File names have generations: `_3_10`, `js1-4`, a single number, `_lesson_01_3` (chapter first), none
- `site_for_path()` raises `NoSiteError` for a path in no site; `clean()` checks the stored site matches
- Validation runs in `full_clean()`, so the importer must call it
- Dry-run before importing: 4,402 pages parse, 0 unmapped, 7 pages have no banner

Learning Website with Django

Chapter 4 · Loading Content from Files

Chapter 3 taught the project what a page is. This chapter fills the database with the real thing: a management command, `import_content`, that reads every file in the content folder, works out what each one is, and creates, updates or skips it. It has to be safe to run over and over, fast enough to run on every deploy, and honest about anything it cannot handle. Every number below comes from running it on the real content folder.

RUN FOR REAL, ON DJANGO 6.1

The importer has 79 tests in the project (all passing, with a temporary content folder for each importer test), and was run on all 4,403 real pages. The timings depend on a folder synced by OneDrive on an ordinary PC, so treat them as relative, not as promises.

Where Does the Page Body Live?

The Chapter 3 models store facts about a page. The page's HTML has to be somewhere, and there are two honest choices:

	IN THE DATABASE	READ FROM THE FILE ON EACH REQUEST
What is deployed	Code plus one database; the content folder need not be on the server	Code, the database <i>and</i> the whole content folder, kept in step
Consistency	A snapshot: metadata and body always come from the same import	A file edited after the import can disagree with its database row
Cost	Disk space, and care not to load bodies for lists	File reads, ideally cached
Speed of a page	One query	One query plus a file read

The course stores the body in the database. The real sizes make it easy: the 4,403 pages total about 92 MB of HTML (an average of 22 KB, the largest 0.77 MB, none over 1 MB), and the finished SQLite file is about 99 MB. A build is then a snapshot: import, test, deploy the database

with the code. The cost to manage is loading bodies by accident, so `Page.objects.light()` defers the body field, and every list, menu and sitemap uses it.

Storing a Fragment Safely

A stored fragment is HTML that will be sent to browsers without escaping, which is exactly the thing Django normally prevents. It is safe here for one reason: **it came from your own files, through the importer, and nothing else can write to it.** Six habits protect that:

- **One place marks it safe.** `Page.html` is the only `mark_safe` in the project. A template that wants the body uses that property; everything else is escaped as usual.
- **Never accept visitor text into `fragment`.** When Chapter 10 adds accounts, comments go in their own model and are escaped.
- **Refuse paths that leave the folder.** `safe_join` resolves the real path, including symbolic links, and rejects anything outside the content root.
- **Set limits.** A file over 2 MB is an error; the real maximum is 0.77 MB, so the limit only catches accidents.
- **Be strict about encoding.** A file that is not valid UTF-8 is reported, not guessed at.
- **Write in one transaction.** A failure part-way leaves the database exactly as it was.

THE PAGES CONTAIN SCRIPTS ON PURPOSE

Interactive tools carry their own `<script>`, and every page has inline styles. That means a strict Content Security Policy that forbids inline code cannot be switched on without rewriting the content. The protection is where the HTML comes from, not a header, which is why that point is worth repeating: only your files, only through the importer.

Turning a File Into a Fragment

Three small, tested functions do the work, matching what the live site does (Learning Website: Framework & Architecture 6):

STEP	WHAT IT DOES
<code>extract_fragment</code>	A complete document gives its <code><body></code> plus the styles from its head. A fragment loses only its leading banner comment. A body with no end is an error.

STEP	WHAT IT DOES
<code>rewrite_solution_links</code>	Every <code>.txt</code> link is rebuilt as <code><course>/solutions/<file name></code> , whatever stale form it had (five are tested).
<code>safe_join</code>	Resolves a path under the content root and refuses to leave it.

The Importer

```
# python manage.py import_content [--dry-run] [--prune] [--force] [--site languages]

for rel in iter_content_paths(root):
    data = open(full, "rb").read()
    digest = hashlib.sha256(data).hexdigest()
    if not force and existing.get(rel) == digest:
        unchanged += 1; continue          # nothing to do
    parsed = parse_page(raw, rel)          # Chapter 3
    fragment = prepare_fragment(raw, rel)   # this chapter
    ...create or update...                  # bulk_create / bulk_update, ...
```

SITUATION	WHAT THE IMPORTER DOES
A new file	Creates the page, and its course if it is a new folder
An unchanged file	Skips it (same SHA-256 hash): no parsing, no write
An edited file	Updates that page only; the other pages keep their import time
A file that has gone	Reports it as missing and keeps the row; <code>--prune</code> deletes it
Not UTF-8, over 2 MB, in no site, or a body with no end	An error for that file only; the rest import; the command exits with an error code
<code>_print</code> pages, underscore files, <code>pdfs</code> , <code>solutions</code> , the kanji archive	Skipped, as on the live site
<code>--dry-run</code>	Reports everything; writes nothing

Deleting by default would be dangerous: a mistake in the content path would look like “every page was deleted” and empty the site. So a missing file is a report, and removal is a deliberate flag.

The Real Run

COMMAND	RESULT
<code>--dry-run</code>	Would create 4,403 pages, 0 errors (13 to 43 seconds)
First import	Created 4,403 pages in 395 courses, 0 errors, about 20 seconds
Second import, nothing changed	4,403 unchanged, about 10 to 12 seconds (almost all of it reading files to hash them)
<code>--force</code>	Updated all 4,403, about 23 to 27 seconds
<code>--site languages</code>	561 pages, 1 to 5 seconds

The stored result: programming 1,371 pages (33 MB of HTML), humanities 707 (9.5 MB), systems 653 (19 MB), languages 561 (7.3 MB), web development 514 (14 MB), life skills 253 (3.2 MB), AI 187 (3.9 MB) and creative 157 (2.0 MB). No stored page still begins with its banner comment, and none lacks a hash.

MEASURE, THEN MOVE THE CHEAP TEST FIRST

The first version of `--site languages` took 16 seconds, because it parsed every file and only then checked the site. The site is known from the *path* alone, so the fixed version checks that first and never reads the other sites' files: 1 to 5 seconds. The same command also showed that a timing is not a fact: the dry run took 13 seconds on one run and 43 on another, depending on how many files were already cached. Repeat a measurement before trusting it.

Slugs: The Path Is the Slug

A *slug* is the short, URL-safe name of a page. Here there is no separate slug field: the page's path *is* its slug, and its address is the path without `.html`. That keeps the address identical to the old site's, which makes redirects trivial (Learning Website: Framework & Architecture 11). Three checks on the real paths made sure that is safe:

- **Letter case:** no two paths differ only by case, so the database's unique path cannot collide when the site moves from Windows to Linux.
- **Unicode form:** all paths are in the NFC form, so the same character cannot be stored two ways.
- **Non-ASCII names:** 282 paths contain non-ASCII characters (the hiragana and katakana pages). A test sends both the percent-encoded and the raw address and gets the page. Links should use the percent-encoded form, which is what browsers send.

Hands-On Exercises

EXERCISE 1

Write the functions that turn a raw file into a stored fragment: unwrap a complete document, drop the banner, rewrite every stale solution link, and refuse unsafe paths. Test each, including five stale link forms and a path that climbs out of the folder. Then test addresses that contain non-ASCII characters.

 [View solution](#)

EXERCISE 2

Add the hash, fragment and import-time fields to the page model, then write the importer and the `import_content` command with `--dry-run`, `--prune`, `--force` and `--site`. Test it against a temporary content folder: creating, skipping unchanged files, updating an edited file, a deleted file, bad encoding, an oversized file, and a path in no site.

 [View solution](#)

EXERCISE 3

Run the importer on the real content folder: a dry run, the first import, a repeat, a forced import and a single site. Record the counts and times, look at what was stored, and find and fix the slowest avoidable step.

 [View solution](#)

CHAPTER 4 QUICK REFERENCE

- The body lives in the database (92 MB for 4,403 pages; the database file is about 99 MB); use `Page.objects.light()` for every list
- A stored fragment is trusted HTML: **one** `mark_safe` (`Page.html`), only your own files, never visitor text
- Protect the import: `safe_join`, a 2 MB limit, strict UTF-8, one transaction
- Change detection is a SHA-256 of the file: unchanged files are skipped without parsing
- A missing file is reported and kept; `--prune` deletes; `--dry-run` writes nothing; `--force` re-imports all; `--site` limits to one site
- One bad file is an error for that file only; the command exits with an error code so a build fails
- The path is the slug and the address is the path without `.html`; no case collisions, all NFC, 282 non-ASCII paths work
- Real run: 4,403 pages, 395 courses, 0 errors, about 20 seconds; repeat runs about 10 seconds
- Check the cheap test (the site, from the path) before the expensive one (reading and parsing the file)

Learning Website with Django

Chapter 5 · Templates & the Shared Design System

The database now holds every page; the routing knows which site a request is for. This chapter makes the pages *appear*: a base template, a reusable `theme` app that carries the shared design, and a real page view that replaces the placeholders from Chapter 2. The design follows the plan in Learning Website: Framework & Architecture 4: tokens, shared components that sit *under* each lesson's own styles, and a safe order for loading them. Here it becomes working code, and it was looked at in a real browser.

RUN FOR REAL, ON DJANGO 6.1

The project has 92 tests (all passing). Pages from the imported content were served by the development server and photographed with headless Chrome. The Chapter 2 placeholder tests had to be rewritten, which is covered below.

The Theme App

Everything visual lives in one app, `apps/theme`, so a change to the design is a change in one place and every site gets it:

```
apps/theme/
├── context_processors.py      # tells every template which site it is on
├── templates/theme/
│   ├── base.html             # the page: head, site bar, main, footer
│   ├── page.html             # a stored page inside base.html
│   └── home.html
└── static/theme/
    ├── tokens.css             # the ONLY place a colour is decided
    ├── base.css               # typography, layout, links
    ├── components.css         # tip-box, code-block, tables, exercises (shared default
    └── copy-code.js           # copyCodeBlock(), and scripts inside a page body
```

One Element Restyles the Page

The base template puts three facts about the site on the `<html>` element:

```
<html lang="en-GB" data-site="languages" data-family="language" style="--accent: #a78bfa">
```

- `data-family` picks the **neutral colours**. The language courses use one set and the technical courses another (the two sets measured in the framework course), so `tokens.css` defines each set once under `[data-family="..."]`.
- `style="--accent: ..."` sets the site's **own colour**, a CSS variable that links, the site bar's underline, table headings and the shared boxes all read.
- The values come from one dictionary in the site map, `SITE_STYLE`, so a ninth site is one more line and no template changes.

The four language colours (Hungarian green, German gold, Japanese red, French blue) are set on the *lesson's own wrapper*, in `tokens.css`. A Hungarian chapter therefore keeps its green inside a site whose chrome is violet, which is what the screenshot of the languages site shows.

Shared Defaults Under Old Styles

Every existing page carries its own scoped CSS. If the shared styles beat it, old pages change; if they lose, old pages stay exactly as they are and new pages use the shared look. The cascade gives you the second for free, provided the shared rules stay *less specific* than a fragment's own:

RULE	SPECIFICITY	WHERE IT COMES FROM
<code>.tip-box</code>	one class	components.css (shared default)
<code>.devsetup1-lesson .tip-box</code>	two classes	the fragment's own style: wins
A tie	equal	the later rule wins, and the fragment's <code><style></code> is in the body, after the links

Two tests keep that promise: the shared CSS never contains `!important`, and no shared selector uses an id or more than two parts. The screenshot of the systems site shows the result: the Debian chapter keeps its pink headings, green tip box and yellow warning box, inside a site with a salmon accent.

Accents You Can Read

A colour that is lovely and unreadable is a bug. Each accent has a test that checks its contrast against the surface it sits on and against the page background, to the 4.5 that WCAG level AA asks for normal text:

SITE	ACCENT	ON SURFACE	ON BACKGROUND
languages	<code>#a78bfa</code>	6.18	6.93
webdevelopment	<code>#38bdf8</code>	8.07	8.81
programming	<code>#44b78b</code>	6.91	7.54
systems	<code>#f0938a</code>	7.62	8.31
ai	<code>#fbbbf24</code>	10.36	11.30
humanities	<code>#fb923c</code>	7.43	8.34
lifeskills	<code>#6ee7b7</code>	11.03	12.38
creative	<code>#f472b6</code>	6.53	7.12

Every site's accent passes, and no two are the same. These are the *site* accents; the Japanese lesson red is the lessons' own colour and is still only 3.86 (Learning Website: Framework & Architecture 4), which a lighter text variant in `tokens.css` would fix.

The Page View

```
def page(request, site, page_path):
    path = page_path.rstrip("/") + ".html"          # address -> file path
    page = get_object_or_404(Page, path=path, site=site) # another site's page: 404
    show_h1 = "<h1" not in page.fragment.lower()
    return render(request, "theme/page.html", {"page": page, "show_h1": show_h1})
```

- **The address is the path.** `/hungary/x/chapter/` is the file `hungary/x/chapter.html`. There is no other lookup key.

- **The site check is in the query.** A row whose stored site disagrees with the address is a 404, even though the route matched.
- **The heading rule is the live site's.** A page without its own `<h1>` gets one made from its title (course chapters have no real heading of their own); a page with one does not get a second.
- **Escaping is on by default.** A page's title is escaped; its stored body is marked safe in exactly one place (Chapter 4). A test stores a title of `<b>x</b>` and a body of `<b>y</b>` and checks that the first is shown as text and the second as markup.

A TEST OF PLACEHOLDER OUTPUT IS A TEST OF THE PLACEHOLDER

Replacing the Chapter 2 views broke 11 tests, because they asked the placeholder what it returned. They were rewritten to ask the real question: “does this site's URL configuration have a route for this path?” uses `django.urls.resolve`, and “which site answered?” looks for `data-site` in the page. A routed-but-missing page and an unrouted path are both 404 now, so a status code can no longer tell the two apart.

Looking at It

The tests cannot say whether a page looks right. Serving the imported content and photographing it with headless Chrome (mapping the site names to your own machine inside the browser, so no hosts file is needed) showed:

- **Languages:** a violet-underlined site bar; the chapter title as a large heading, made from the banner; the course name in the lesson's own green; dialog cards with purple speaker letters.
- **Systems:** the same layout with a salmon underline; the Debian chapter in its own pink, with its green tip box, yellow warning and table.

Only desktop width was looked at, and a third screenshot (programming) was taken but not examined. A phone width and the print preview remain to check (Learning Website: Framework & Architecture 12).

Hands-On Exercises

EXERCISE 1

Build the theme app: a `SITE_STYLE` dictionary, a context processor, the three stylesheets (tokens, base, components), the copy-code script, and the base, page and home templates. Explain how one element restyles a page and why a fragment's own CSS still wins.

 [View solution](#)

EXERCISE 2

Replace the placeholder views with a real page view, rewrite the tests that depended on the placeholders, and add tests for contrast, the shared CSS rules, stylesheet order, escaping, the heading rule and the 404 cases.

 [View solution](#)

EXERCISE 3

Serve the imported content with the development server, fetch a stylesheet with `curl`, and photograph one page from two different sites with headless Chrome. Look at the images and write down what you see, and what you did not check.

 [View solution](#)

CHAPTER 5 QUICK REFERENCE

- One `theme` app holds the whole design: `tokens.css`, `base.css`, `components.css`, `copy-code.js` and the templates
- The `<html>` element carries `data-site`, `data-family` and `style="--accent: ..."`; the values come from `SITE_STYLE`
- Load order: tokens, base, components, then the page body (with the fragment's own `<style>`)
- Shared rules stay at one class, with no ids and no `!important`, so a fragment's own two-part rule wins
- Language colours (`.hu-lesson` and the others) live on the lesson wrapper, so they survive a different site accent
- Every accent is tested at 4.5 or more on its surface and on the page background
- The page view: address to path, site checked in the query, one `<h1>` only if the body has none, title escaped, body marked safe once

- A test that asserts on placeholder output breaks when the placeholder goes: test routing with `resolve()`, and the site with `data-site`
- Look at the pages in a real browser; headless Chrome with `--host-resolver-rules` needs no hosts file

Learning Website with Django

Chapter 6 · Navigation & Breadcrumbs

A visitor needs to know three things at every moment: *which site am I on, what is on this site, and where am I within it*. Learning Website: Framework & Architecture 5 designed three layers of navigation for that. This chapter builds them in Django, and keeps one rule throughout: **every layer is derived from data that already exists**, so none of them can drift out of date the way the live site's hand-written menu has.

RUN FOR REAL, ON DJANGO 6.1

The project has 125 tests (all passing). The navigation was checked against all 4,403 imported pages and photographed with headless Chrome. The one thing that could not be checked is a true phone-width layout; the chapter says so where it matters.

Three Layers, Three Sources

LAYER	ANSWERS	DERIVED FROM
1. Global bar	Which site am I on, and what are the others?	The site map (<code>SITES</code>); the current site is marked
2. Site menu	What is on this site?	The folders in the site map that really have pages in the database
3. Breadcrumb, previous/next	Where am I?	The page's path, its course and its chapter number

All three are exposed to every template by the context processor from Chapter 5, so a template only has to loop over `global_bar`, `site_menu` and `breadcrumbs`.

Layer 1: The Global Bar

It lists the eight sites in the site map's order, the same on every page, with the current one marked `aria-current` (so a screen reader says so as well as the colour). Each address comes from one setting, `SITE_URL_TEMPLATE`:

```
# config/settings/prod.py
SITE_URL_TEMPLATE = "https://{site}.osztromok.com"
# config/settings/dev.py
SITE_URL_TEMPLATE = "http://{site}.localhost:" + os.environ.get("LW_DEV_PORT", "8000")
```

Changing the domain, or the development port, is one line, and the bar follows. It scrolls sideways on a narrow screen instead of wrapping into several rows.

Layer 2: The Site Menu

The menu is the site's folders, in the site map's order, *that have pages*. It reads the database once and is kept in Django's cache for an hour; the importer clears it when it finishes, so a new import shows at once. After the first request, the menu costs no database query, which a test proves with `assertNumQueries(0)`.

DERIVED MENUS CANNOT DRIFT

The live site's hand-written menu was missing 13 of its 49 top-level folders (Learning Website: Framework & Architecture 5). The derived menus were checked against the real pages: **all 49 folders that have pages appear in a site menu**, none appears in two (except the sidebar, which each site shows only its own part of), and no folder in the site map is empty.

Readable labels

A menu entry is a folder name made readable. The live site's helper capitalised every word, giving "Data Science And ML" and "Cloud And Devops". The new one lower-cases small joining words, upper-cases acronyms, and has a short list of names with their own capitals:

FOLDER NAME	LIVE SITE	HERE
<code>data-science-and-ml</code>	Data Science And ML	Data Science and ML
<code>ccloud-and-devops</code>	Cloud And Devops	Cloud and DevOps
<code>maths-for-programmers</code>	Maths For Programmers	Maths for Programmers
<code>ios-development</code>	Ios Development	iOS Development

The sidebar needs routes too

Building the menu exposed a gap from Chapter 2: its URL configuration was made from the folders in the site map, so `/sidebar/...` was a 404 on every site. The sidebar belongs to the site its *subject* is routed to (`sidebar/linux` to systems, `sidebar/football` to humanities), and `/sidebar/` is that site's own index. The pattern keeps the sidebar index as a separate alternative (`sidebar/?`) so that `/sidebar/football/x/` can never match on the systems site. Tests check both directions.

Layer 3: Where You Are

The breadcrumb

The trail is built from the page's path: *Site > folder > folder > this page*. A course folder is labelled with the course's real name ("Hungarian Basic Conversation 3", one query for all the folders on the path), any other folder with its readable name, and the page you are on is plain text, not a link.

Previous and next

They follow the chapter *number*, so chapter 10 comes after 9, and a page with no course or number simply has neither link. The links carry `rel="prev"` and `rel="next"`.

Every folder is a page

A breadcrumb that links to a folder is a promise that the folder page exists. So the page view serves either a page or a folder: a folder lists its sub-folders (with page counts) and its own pages, and a course folder lists its chapters in numeric order under the course's name. An address that is neither is a 404, and another site's folder is a 404 on this one.

```
def page(request, site, page_path):
    bare = page_path.strip("/")
    found = Page.objects.filter(path=bare + ".html", site=site).first()
    if found is not None:
        ...render the page, with breadcrumbs and previous/next...
    content = listing(site, bare)                # a folder?
    if content is None:
        raise Http404("No such page or folder")
    ...render the listing...
```

Links Between Sites

The global bar links across sites with absolute addresses. For links *inside page content*, the resolver from the framework course is included (`resolve_link`): a same-site link stays root-relative, a cross-site link becomes an absolute address on the right site, and a path that belongs to no site raises an error rather than being guessed. It is not applied to page bodies today because the content has no real cross-site links (Learning Website: Framework & Architecture 11); it is ready for the first one.

What It Costs, and What Was Not Checked

- **Queries:** a chapter page costs 4 database queries and a folder listing 4 (the page, the course names for the breadcrumb, the two neighbours). The menus add none, because they are cached.
- **Looked at:** screenshots of the Hungarian course listing and of the Linux folder show the global bar with the current site in the accent colour, the site menu with the active subject underlined, the breadcrumb, and the twelve chapters numbered in order.
- **Not verified: phone width.** A headless screenshot at 390 pixels did not look like a true phone layout (it seemed to be laid out about 500 pixels wide and cropped, so one menu item looked cut off), so the narrow layout has not been checked. Test it with a real browser's device toolbar.

Hands-On Exercises

EXERCISE 1

Build the global bar and the cached site menu from the site map and the database, write readable folder labels (small words, acronyms, product names), add the link resolver, and make the importer clear the cached menus.

 [View solution](#)

EXERCISE 2

Add breadcrumbs, previous and next links and folder listings, make the page view serve a page or a folder, route the sidebar to the right site, and test all of it, including numeric chapter order and the sidebar boundaries.

 [View solution](#)

EXERCISE 3

Check the navigation against the real imported pages: which folders appear in which menu, whether any folder is missing or in two menus, and how many queries a page costs. Look at it in a browser, and write down what you could not verify.

 [View solution](#)

CHAPTER 6 QUICK REFERENCE

- Three layers, all derived: the **global bar** (site map), the **site menu** (folders with pages), the **breadcrumb and previous/next** (path, course, chapter number)
- Site addresses come from one setting, `SITE_URL_TEMPLATE`; the current site and active subject use `aria-current`
- The menu is cached for an hour and cleared by the importer; a repeat request makes no query for it
- Labels: small words lower case, acronyms upper case, product names their own capitals ("Data Science and ML", "Cloud and DevOps")
- Course folders are labelled with the course name in the breadcrumb; the page you are on is not a link
- Previous/next use the chapter **number**; a page with no number has neither
- Every folder is a page (sub-folders with counts, then the pages); an unknown folder, or another site's, is a 404
- The sidebar routes to the site its subject belongs to; `/sidebar/` is that site's own index; the index is a separate pattern so subjects cannot leak
- `resolve_link` turns cross-site links absolute and raises for a path in no site
- All 49 real folders appear in a menu; a page costs 4 queries; phone width still needs a real-browser check

Learning Website with Django

Chapter 7 · The Languages Site

So far everything has been shared by all eight sites. This chapter builds the first thing that belongs to just one: the **languages site**, the area chosen to move first (Learning Website: Framework & Architecture 10). It needs a few things the others do not: it must know which language is which, group its courses the way a learner thinks, and mark the language of every line being taught. The lesson is as much about *where* that code goes as about what it does: a small app of its own, plugged into the shared pipeline, with nothing language-specific added to the shared apps.

RUN FOR REAL, ON DJANGO 6.1

The project has 152 tests (all passing). The changes were run on the real content: 561 languages pages, and the results were checked against counts taken from those pages *before* the code was written.

Count First

Before writing the markup code, the classes the lesson fragments really use were counted. Each line in the language being taught sits in an element with a class named after the language, and the English line has the class `en`:

CLASS	LINES	PAGES	HTML LANGUAGE CODE
<code>de</code> (German)	1,285	48	<code>de</code>
<code>hu</code> (Hungarian)	1,274	63	<code>hu</code>
<code>jp</code> (Japanese)	687	35	<code>ja</code> , not <code>jp</code>
<code>fr</code> (French)	363	15	<code>fr</code>

The lesson wrappers are `.jp-lesson` (95 pages), `.hu-lesson` (67), `.de-lesson` (48) and `.fr-lesson` (15), plus four more used by the Japanese culture courses, which have no colour of their own yet.

Why Mark the Language?

- **Pronunciation.** A screen reader switches to the right voice for `lang="hu"` text. Without it Hungarian is read with English rules.
- **Glyph shapes.** The same kanji are drawn with different shapes for Chinese and Japanese. `lang="ja"` tells the browser to use the Japanese forms.
- **Spelling and hyphenation** use the right language.

```
<span class="jp">おはようございます。</span>  
becomes  
<span lang="ja" class="jp">おはようございます。</span>
```

The pass is small, and the care is in the edges. It reads *whole* class names (so `hu` matches but `hu-lesson` does not), leaves a tag that already has a `lang` attribute alone, copes with attributes spread over several lines, ignores text that merely mentions a class (an escaped example or a CSS rule), and gives the same result when run twice. Each of those is a test.

Plugging It In Without Touching the Shared Apps

The shared importer should not know about Hungarian. Instead, `apps/content/fragments.py` keeps a small registry, and the languages app adds its function to it when Django starts:

```
# apps/site_languages/apps.py  
def ready(self):  
    from apps.content.fragments import register_transform  
    from .markup import add_language_attributes  
    register_transform("languages", add_language_attributes)
```

The importer calls `prepare_fragment(raw, path, site)`, which runs the shared steps and then that site's registered functions. Registering twice does not run a function twice, and a page of any other site is untouched. The real check: after importing, there are **0** of these attributes on any other site, and on the languages site the counts match the table above exactly (1,285, 1,274, 687 and 363).

A Gotcha: When the Pipeline Changes but the Files Do Not

Chapter 4's importer skips a file whose hash has not changed. But the hash says whether the *file* changed, not whether the way it is *processed* did. After adding the markup pass, every page would still say “unchanged” and the new attributes would only reach pages someone happened to edit.

```
PIPELINE_VERSION = "2"          # raise it whenever a processing step changes
digest = hashlib.sha256(PIPELINE_VERSION.encode() + b"\0" + data).hexdigest()
```

With the version inside the hash, raising it re-imports every page *once* and then settles. On the real content: **4,403 updated, then 4,406 unchanged** on the next run (the extra three are new chapters added in the meantime). A test does the same on a temporary folder.

The Front Page

The languages site gets a front page of its own, built from the courses in the database. The courses are grouped by the rule recorded when the plan for six courses in two volumes was made:

GROUP	WHAT GOES IN IT	REAL EXAMPLE
Reading and writing	Alphabet, hiragana, katakana, kanji courses, in that order	Hungarian Alphabet 1; Hiragana 1 to 3; Katakana 1 and 2
Survival <Language>	Basic Conversation courses 1 to 3	Survival German: courses 1, 2, 3
Everyday <Language>	Basic Conversation courses 4 to 6	Everyday German: course 4
Culture	Courses about the country	Japanese Music; Manga & Anime; five in all
Lessons and reference	Standalone lessons, not numbered courses	12 Hungarian pages, 3 French, 300 Japanese

The groups are produced by one small function with tests, so the rule is written down once, in code. The result on the real data: French 1 course; German 4 (Survival German and Everyday

German); Hungarian 4 (Reading and writing, Survival Hungarian); Japanese 11 (Reading and writing, Survival Japanese, Culture). A language with no numbered courses still has its card; a lessons link only appears where lessons exist.

Colours, and a fix for the Japanese red

Each language's card sets two variables: `--lang-accent` for the border and `--lang-text` for the heading. They are equal for French, German and Hungarian. Japanese is the exception that Learning Website: Framework & Architecture 4 predicted: its red, `#d64550`, scores only **3.86** against the dark surface, under the 4.5 that WCAG AA asks for normal text. The heading therefore uses a lighter red of the same hue, `#e8707a`, which scores **5.63**. One test checks every language's text colour, and another checks that the original red is below 4.5, so nobody "simplifies" the variant away later.

Answers Without a Script

The "Try It Yourself" answers are in a native `<details>` element. It opens and closes with no JavaScript, so it works in the stored pages exactly as written. 196 of the 561 languages pages use it.

What Is Not Done Here

- **The Japanese culture courses** use their own wrapper classes (`jfilmtv1-lesson`, `jlit1-lesson` and so on), which have no colour in the shared tokens, so they appear in the site's accent. Adding four lines to `tokens.css` would fix it.
- **PDFs and solution files** are the next chapter.
- The front page was looked at in a desktop-width screenshot only.

Hands-On Exercises

EXERCISE 1

Count the language classes in the real pages, then build the languages app: a config of languages, a markup pass that adds `lang` attributes (whole class names only, idempotent), and a registry in the shared pipeline so the app plugs in without changing the shared code.

 [View solution](#)

EXERCISE 2

Group the courses (writing, survival, everyday, culture), build the front page with a card per language, give Japanese a readable text colour, and let a site name a front page of its own. Test the grouping, the colours and the front page.

 [View solution](#)

EXERCISE 3

Add a pipeline version to the hash so a change in processing re-imports every page once. Import the real content, and check that the stored language attributes match the counts you took before writing the code, that no other site changed, and that the front page groups the real courses correctly.

 [View solution](#)

CHAPTER 7 QUICK REFERENCE

- Site-specific code lives in its own app (`apps/site_languages`) and plugs in through `register_transform(site, function)` in `ready()`
- Count the real markup before coding: 1,285 `de`, 1,274 `hu`, 687 `jp`, 363 `fr` lines
- `lang` attributes fix pronunciation, Japanese glyph shapes and spelling; Japanese is `ja`, not `jp`
- The markup pass: whole class names only, skips existing `lang`, multi-line tags, idempotent
- **PIPELINE_VERSION** inside the file hash: processing changes re-import every page once (4,403 updated, then unchanged)
- Groups: Reading and writing, Survival (courses 1 to 3), Everyday (4 to 6), Culture; Lessons and reference linked where they exist
- Per-language card colours: `--lang-accent` and `--lang-text`; Japanese text uses `#e8707a` (5.63) because `#d64550` is 3.86
- Answers use a native `<details>`: no script needed (196 pages)
- Not yet: colours for the Japanese culture courses, PDFs and solution files

Learning Website with Django

Chapter 8 · PDFs, Solutions & Static Files

A page is not the only thing a visitor downloads. Every course has a PDF, most chapters link to solution files, and every page loads a stylesheet and a script. This chapter sorts those files into three kinds, decides who serves each one, and keeps the addresses that old pages already link to working after the split. It also fixes a small problem with the copy button on code blocks that only shows up when the clipboard is unavailable.

RUN FOR REAL, ON DJANGO 6.1 AND WHITENOISE 6.12

The project has 180 tests (all passing). Real PDFs and solution files were mirrored from the content folder and fetched with `curl` from the development server, and the copy script was loaded in headless Chrome. The Apache configuration could not be run here and is marked as such.

Three Kinds of File

KIND	EXAMPLES	SIZE	SERVED BY
Page bodies	The 4,403 pages	92 MB in the database	Django, from the database (Chapter 4)
Static files	The theme's three stylesheets and <code>copy-code.js</code>	A few kilobytes	WhiteNoise, inside the Django process
Assets	Course PDFs, solution <code>.txt</code> files	8,394 files, 666 MB	Apache in production (Django only in development)

The split matters because the three kinds want different things. Static files are tiny, shared by every site and can be cached for years. Assets are large, belong to one site, and benefit from a real web server's handling of big files. Pages are the only thing that needs Django's templates.

Assets: One Folder per Site

Pages link to assets at the *same relative path* as in the content folder: the importer rewrote every stale solution link to `/<course>/solutions/<file>`. To keep those addresses working, the

`collect_assets` command mirrors every `pdfs/` and `solutions/` file into `ASSET_ROOT/<site>/<the same path>`:

```
assets/
├─ languages/
│   └─ hungary/hungarian-basic-3/
│       ├── pdfs/Hungarian_Basic_Conversation_3_Course.pdf
│       └─ solutions/...
└─ systems/
    └─ linux/system-administration/debian-development-machine-setup/
        ├── pdfs/...
        └─ solutions/devsetup1-1_exercise1.txt
```

On the real content, a dry run for every site finds **8,394 files (666 MB)**, all of which belong to a site: nothing was skipped. Languages has 236 files (195 MB) and systems 1,419 (90 MB). A repeat run copies nothing and takes 0.3 seconds, because a file is copied only when it is missing or its size or modified time differs. A file in a folder that belongs to no site is reported, not guessed. Removal needs `--prune`.

THE SITE BOUNDARY IS A FOLDER

Because each site has its own folder, a site *cannot* serve another site's files: there is nothing there to serve. The same PDF requested on the systems site gets a 404. This holds whether Django or Apache serves the file, because both read the per-site folder.

Serving Assets: Django in Development, Apache in Production

Django's `django.views.static.serve` can send the files, and its documentation is blunt that the view is "not hardened for production use". So it is used only while `SERVE_ASSETS_WITH_DJANGO` is on (development), with a route that only matches `.pdf` and `.txt` files inside a `pdfs` or `solutions` folder of that site. In production the route does not exist, and Apache serves the same folder directly.

Real requests to the development server, with `curl`:

REQUEST	RESULT
The Hungarian course PDF, on the languages site	200, <code>application/pdf</code> , 3.9 MB, with a Last-Modified date
The same PDF, on the systems site	404
A Debian chapter's solution link	200, <code>text/plain</code> , <code>X-Content-Type-Options: nosniff</code>
The same file again, with If-Modified-Since	304 : the browser keeps its copy
The first 100 bytes of the PDF	200 with the whole file: Django does not honour range requests

- **Solutions are plain text with `nosniff`**, so a browser may not decide a `.txt` file is something it should run.
- **Range requests are why Apache is better.** A PDF viewer asks for pieces of a large file so the first page appears quickly; a 206 reply is what Apache sends and what Django's view cannot.
- **Tests cover the dangerous cases:** a path that climbs out with `..` or `%2e%2e` is refused, a missing file and a directory are 404, a `.sh` file in a solutions folder is not routed, and the link the importer wrote reaches the file.
- **The course page lists its downloads:** the PDF as a download link, and the solution files in a collapsible section when there are any (the language courses have none).

THE APACHE CONFIGURATION WAS NOT RUN

The production configuration (an `AliasMatch` to the site's own asset folder, a `ProxyPassMatch` exclusion so the proxy does not claim those addresses first, and a cache header for PDFs) is in the Exercise 2 solution. Apache is not installed on the machine used to write this course, so test it with `apache2ctl configtest` and `curl -I`: a range request should return 206 from Apache.

Static Files with WhiteNoise

The theme's files are a handful of small static files. WhiteNoise serves them from inside the Django process, which needs no web-server configuration at all. Two settings do the work: the middleware, placed directly after `SecurityMiddleware`, and a storage class that gives every file a hashed name:

```
# config/settings/prod.py
STORAGES = {
    "default": {"BACKEND": "django.core.files.storage.FileSystemStorage"},
    "staticfiles": {"BACKEND": "whitenoise.storage.CompressedManifestStaticFilesStorage"}
}
# then, before deploying: python manage.py collectstatic
```

The tests read the real behaviour instead of assuming it:

- After `collectstatic` each file has a name with a 12-character hash (`tokens.5f6ee8c98574.css`) and a compressed `.gz` copy, and the pages link to the hashed names.
- A hashed file is sent with `Cache-Control: max-age=315360000, public, immutable` : ten years, never revalidated. **A first guess in the test, one year, was wrong**, and the test failed and said so. The unhashed name is cached for 60 seconds only.
- Static files are sent with `Access-Control-Allow-Origin: *`, which suits a stylesheet shared between sites (Learning Website: Framework & Architecture 1), and means anything under `/static/` can be read by any web page.
- A file missing from the manifest is an *error* in production, so a typo in a template shows up at once.

One small thing from the first full run: WhiteNoise printed a warning on every start because the folder `collectstatic` files did not exist in development. The development settings now create it.

The Copy Button, Revisited

The first version used the modern clipboard and did nothing if it was unavailable. It is only available on HTTPS and on `localhost`, so the development server opened from another machine over plain HTTP would fail *without a word*. The new version falls back to a hidden text box and the older copy command, and shows "Copy failed" if both fail, so the button never lies.

It was tested in headless Chrome with a page that inserts a body *as text*, as the site does, with a script in it, and then clicks the button:

CHECK	RESULT
A script inserted as text: has it run before the page loads?	No (as expected)
And after the page loads and <code>copy-code.js</code> has re-created it?	Yes: the interactive tools will work
Click the button (normal context)	"Copy failed"
Click the button (forced insecure context, so the fallback runs)	"Copy failed"

WHAT THIS DOES AND DOES NOT PROVE

Script re-creation is verified. **The copy itself could not be verified:** in headless Chrome the clipboard is not permitted and the older command also fails, so both paths report failure. That does prove the fallback is reached and that the button says so instead of staying silent. Whether copying succeeds has to be tried by hand in a real browser, on an HTTPS address and on a plain HTTP one. It is in the review notes.

Hands-On Exercises

EXERCISE 1

Write the command that mirrors every `pdfs/` and `solutions/` file into one folder per site at the same relative path, copying only what is missing or changed, with a dry run, a one-site option and an opt-in prune. Run it on the real content and report the counts.

 [View solution](#)

EXERCISE 2

Serve the mirrored files from the development server through a route that only matches PDFs and text files in `pdfs` and `solutions` folders of that site. Test the boundaries and the dangerous paths, fetch real files with `curl`, and write the Apache configuration for production, saying clearly that it is untested.

 [View solution](#)

EXERCISE 3

Serve the theme's static files with WhiteNoise and hashed names, and test the real cache headers. Then make the copy button fall back when the clipboard is unavailable, and test the script in headless Chrome. State exactly which parts the browser test can and cannot verify.

 [View solution](#)

CHAPTER 8 QUICK REFERENCE

- Three kinds of file: page bodies (database), static files (WhiteNoise), assets (PDFs and solutions: Apache in production)
- Assets keep the **same relative path**; `collect_assets` mirrors them to `ASSET_ROOT/<site>/...`: 8,394 files, 666 MB, 0 skipped
- The site boundary is a folder: another site's PDF is a 404 because it is not there
- Django's `static.serve` is development only (its docs say so); it does not honour range requests
- Solutions are served as `text/plain` with `nosniff`; a repeat request with `If-Modified-Since` gets 304
- Only `.pdf` and `.txt` in `pdfs` or `solutions` folders are routed; `..` and `%2e%2e` are refused
- WhiteNoise: middleware right after `SecurityMiddleware`; `CompressedManifestStaticFilesStorage`; run `collectstatic` before deploying
- Hashed static files: `max-age=315360000, public, immutable` (ten years); unhashed: 60 seconds; both with `Access-Control-Allow-Origin: *`
- Copy button: modern clipboard, then a fallback, then "Copy failed"; script re-creation verified in Chrome; the copy itself needs a manual test
- The Apache configuration is untested: `apache2ctl configtest` and `curl -I` (expect 206 for a range request)

Learning Website with Django

Chapter 9 · Search & Sitemaps

With 4,400 pages, a visitor who knows what they want should not have to click through menus to find it, and a search engine should be told what each site contains. This chapter adds a search box that only searches the current site, a sitemap for each site, one canonical address for every page, and a `robots.txt` that keeps crawlers out until launch.

RUN FOR REAL, ON DJANGO 6.1 AND SQLITE

The project has 217 tests (all passing). Every search and sitemap below was run on the real database of 4,408 pages. The PostgreSQL version is described, **not run**, because this project uses SQLite.

Choosing a Search Engine

OPTION	EXTRA TO RUN	GOOD FOR	USED HERE
SQLite FTS5	Nothing: it is inside SQLite	One server, a few thousand pages	Yes
PostgreSQL full-text search	PostgreSQL	The same size, once the database is PostgreSQL	Described only
Elasticsearch or OpenSearch	A separate server	Millions of documents, typo tolerance	No: far more than this needs

The Index

The page body is stored as HTML, so the importer first turns it into plain text (tags removed, `<script>` and `<style>` ignored, a space at every block tag so table cells do not join into one word). The text goes into two FTS5 tables, each using the page's own id as its row id:

TABLE	TOKENIZER	HOLDS	WHY
page_fts	<code>unicode61</code> <code>remove_diacritics 2</code>	Every page (4,408)	Word search; "szeretnek" finds "Szeretnék"

TABLE	TOKENIZER	HOLDS	WHY
page_fts_cjk	trigram	Pages with Japanese (421)	Japanese has no spaces, so there are no words to find

```
CREATE VIRTUAL TABLE page_fts USING fts5(
    title, body, site UNINDEXED,
    tokenize='unicode61 remove_diacritics 2'
)
```

The `site` column is stored but not searched, so one table serves all eight sites and each search filters on it. The importer keeps the index in step: it re-indexes new and changed pages after saving them and removes pages from the index before pruning them. The importer's version number went up, so the first import after this chapter re-imported every page once.

A BUG THAT ONLY THE REAL RUN FOUND

What went wrong: the first real import crashed with “not all arguments converted during string formatting”. I had written `?` placeholders (SQLite's own) in raw SQL sent through Django's cursor, which expects `%s`. The tests passed because query logging is off in tests, and the development server logs queries. **How to avoid it:** use `%s` with Django's cursor, and run the real command with the real settings, not only the test suite.

Real import: 4,406 pages updated and 2 created in 38 seconds, no errors, 4,408 rows in each index table (421 in the Japanese one). The database grew from about 99 MB to 162 MB after a `VACUUM`, so the indexes cost roughly 63 MB.

Turning Typed Text into a Query

FTS5 has its own query language: quotes, `AND`, `OR`, `NEAR`, brackets, a leading minus, column filters like `title:x`. Text typed by a visitor that is not valid in it is an *error*, and text that is valid can change the meaning. So the visitor's text is never put in as it is. Only word characters are kept, each word is quoted, at most eight are used, and the last one may be the start of a word:

TYPED	QUERY SENT
polite cond	"polite" "cond"*

TYPED	QUERY SENT
<code>"; DROP TABLE x; -- OR NEAR(</code>	<code>"DROP" "TABLE" "x" "OR" "NEAR"*</code>
<code>!!! ???</code>	(nothing: no search is run)

Results are ranked with `bm25`, with a title match worth ten times a body match. Each result carries a snippet. The match is wrapped in control characters (which cannot appear in page text), the snippet is HTML-escaped, and only then are the markers turned into `<mark>` tags, so page text that contains a script is shown as text.

Japanese

The trigram table finds any run of **three or more** characters. One or two characters (水, 水を) are too short for it, so those are answered by scanning the 421 Japanese pages with `LIKE`, with `%` and `_` escaped. That scan took 5 ms; it will slow down if the number of Japanese pages grows a lot.

Real Searches

SITE	SEARCH	RESULTS	TIME	FIRST RESULT
languages	<code>szeretnek</code> (no accents)	18	12 ms	Buying Clothes: Sizes, Fit & Returns
languages	<code>Szeretnék</code>	18	3 ms	the same page
languages	<code>brotchen</code>	1	1 ms	Ordering Food & Drink at a Café
systems	<code>partition</code>	44	8 ms	Multi-Boot With More Than Two OSes & Advanced Partitioning
programming	<code>decorat</code> (a beginning)	63	9 ms	Decorators
languages	<code>ありがとう</code>	25	5 ms	Za Ji Zu
languages	<code>水</code> (one character)	5	5 ms	Days, Dates & Time
ai	<code>zzzqqq</code>	0	0 ms	(none)
languages	<code>"; DROP TABLE content_page; --</code>	0	1 ms	(none; no error)

A search on one site never returns another site's pages: the `site` filter is part of every query. The search box sits in every site's header and goes to that site's own `/search/`. Results come twenty at a time with Previous and Next links, and the empty search shows a short help line. The page was checked in headless Chrome: the matched word is highlighted inside each snippet.

The PostgreSQL Version (not run)

If the database ever becomes PostgreSQL, the usual route is a generated `SearchVectorField` with a GIN index, queried with `SearchQuery` and ranked with `SearchRank`. The FTS5 migration and functions here do nothing on other databases (they check the database vendor), so the switch would replace this app's two modules, not the rest of the project. This was not run here.

Sitemaps

A sitemap is a list of a site's pages for search engines. Each site has its own at `/sitemap.xml`, built with Django's `django.contrib.sitemaps` from that site's pages only, with each page's own address:

SITE	PAGES	SITEMAP ENTRIES	DUPLICATES	WITH A DATE	SIZE
languages	561	561	0	250	69 KB
programming	1,376	1,376	0	654	189 KB
ai	187	187	0	27	21 KB

- The limit is 50,000 URLs per sitemap; the largest site has 1,376, so one file is enough.
- A `lastmod` date appears only where the page's banner has a date. No date is invented, because a wrong date teaches a crawler to ignore the field.
- Addresses with non-ASCII characters are percent-encoded (`hiragana_%E3%81%82/`).
- Behind the proxy the addresses are `https`, because the proxy settings from Chapter 2 tell Django the original request was secure.

One Address per Page

Every page now has a `<link rel="canonical">` and a `<meta name="description">` (from the banner's Topic line, or the first 160 characters of text). The important rule is that the sitemap and

the canonical link get their address from **the same function**, so they cannot disagree. This was checked on real pages: 75 sampled pages on the languages site (including ones with Japanese in the address) and 60 each on programming and ai, and every page's canonical link was in its site's sitemap. Listing pages and the front page do not have a canonical link yet.

robots.txt

Each host answers its own `/robots.txt`. Until launch it says `Disallow: /`; setting `LW_ALLOW_CRAWLING=1` switches it to `Allow: /` plus a `Sitemap:` line naming that site's own sitemap. Putting it behind a setting makes opening the sites to search engines one deliberate line on the launch checklist.

WHAT WAS NOT VERIFIED

Nothing was submitted to a search engine. `robots.txt` was only read with crawling off in the real run (the on case is covered by tests). The PostgreSQL search is not run. Whether the sitemaps are accepted by search engines can only be seen after launch.

Hands-On Exercises

EXERCISE 1

Make every page searchable on its own site: turn the stored HTML into plain text, create the two FTS5 tables, and keep them in step with the importer. Run it on the real content and report the row counts, the time and the growth in database size, including any bug the real run finds.

 [View solution](#)

EXERCISE 2

Build the search page so typed text can never be an error or an injection: quoted words, a prefix on the last one, bm25 with the title weighted, escaped snippets with highlighting, and a Japanese path. Test hostile input and run real searches on each site.

 [View solution](#)

EXERCISE 3

Add a sitemap and a `robots.txt` for each site and a canonical link on every page, all using the same address function. Parse the real sitemaps, compare their counts with the page counts, and check sampled pages' canonical links against them.

 [View solution](#)

CHAPTER 9 QUICK REFERENCE

- Search: SQLite FTS5, no extra server; `page_fts` (words, accents ignored) and `page_fts_cjk` (trigram, Japanese pages only)
- The importer indexes new and changed pages and removes pruned ones; the first import after the change re-imports everything
- Never put typed text into a query: keep word characters, quote each, prefix on the last, at most eight
- Ranking: `bm25` with the title weighted 10 times; snippets are escaped before `<mark>` is added
- Japanese: 3 or more characters use the trigram table; 1–2 characters use a `LIKE` scan
- Django's cursor wants `%s` placeholders, not `?`; the real run found what the tests could not
- Real numbers: 4,408 pages indexed in 38 s; about 63 MB added to the database; searches take 1–13 ms
- Per-site `/sitemap.xml` (50,000 URL limit) with a date only where the page has one; non-ASCII addresses percent-encoded
- Sitemap address and canonical link come from the same function, so they cannot disagree
- `robots.txt` blocks everything until `LW_ALLOW_CRAWLING=1`; the PostgreSQL search and search-engine acceptance are untested

Learning Website with Django

Chapter 10 · Admin, Accounts & Progress

Until now every visitor is anonymous and nothing on the site can be changed through the site itself. This chapter adds three small things: an admin for looking at what was imported, an optional login, and a button that remembers which chapters you have finished. The interesting part is not the code, which is short, but the decisions about what *not* to allow.

RUN FOR REAL, ON DJANGO 6.1 AND SQLITE

The project has 245 tests (all passing). The admin and the progress pages were used on the real database of 4,408 pages, with a temporary user (random password, never printed, deleted afterwards). The buttons were not clicked in a real browser, and behaviour with several worker processes was not tested.

Who Is Allowed to Do What

WHO	CAN	CANNOT
Anyone	Read every page, search, see sitemaps	Anything that changes data
Logged-in reader	Mark chapters finished; see their own progress	See anyone else's progress; use the admin
Owner (superuser)	Use the admin to look at pages and courses, and to make users	Edit or delete a page (on purpose)

The Admin: Its Own Host, Read-Only

Django's admin is the quickest way to see what is in the database, but two decisions matter more than the setup:

1. **It has its own host.** `admin.osztromok.com` (and `admin.localhost` in development) has its own URL configuration containing the admin and nothing else. The content sites do not contain the admin at all, so `/admin/` is a plain 404 on every one of them: there is nothing to

find or guess at. The middleware from Chapter 2 gained one branch: if the host is the admin host, use the admin's URL configuration.

2. **It is read-only.** The files are the source of truth and the importer overwrites the database from them, so a title edited in the admin would silently vanish at the next import. A form that loses your work without telling you is worse than no form, so the add, change and delete permissions are switched off for pages, courses and progress.

```
class ReadOnlyAdmin(admin.ModelAdmin):
    def has_add_permission(self, request):
        return False

    def has_change_permission(self, request, obj=None):
        return False

    def has_delete_permission(self, request, obj=None):
        return False
```

The page body is left out of the list (the query defers it) and out of the detail form, so listing 4,408 pages stays cheap. On the real database:

ADMIN REQUEST	RESULT
All pages	200, 4,408 pages, 130 ms, 36 KB
Filter: languages site, course chapters	200, 246 results, 75 ms
Page detail	200, no text box for the body
An attempted change to a page	403 (refused)
<code>/admin/</code> on the languages site	404
The admin host with no login	302 to the login page

Times come from Django's test client, so they leave out the network. The tests also check that an ordinary user and a staff user with no permissions are turned away, and that a posted change and a posted delete leave the page unchanged.

Accounts: Optional, and Per Site

Nobody needs an account to read anything. A login only lets a site remember what you have finished. There is **no public sign-up**: the owner makes users in the admin, so there is no registration form to attack or to fill with spam.

- **One login per site.** The session cookie has no `Domain`, so a browser sends it to the host that set it and no other. Logging in on the languages site does not log you in on the systems site. Sharing a login would need `SESSION_COOKIE_DOMAIN=".osztromok.com"`, which also hands the cookie to every subdomain you create later; the project keeps them apart. (The test client has one cookie jar for all hosts, so the test checks the cookie has no `Domain` instead of trying to see the separation.)
- **The cookie** is `HttpOnly` and `SameSite=Lax` (Django's defaults), and `Secure` in production. A login lasts 30 days.
- `?next=` **cannot send you elsewhere.** Django follows it only when it points at the same host. Tested with `https://evil.example/`, `//evil.example/` and another of the sites: each goes to `/`.
- **Logging out is a POST** with the CSRF token, so a link or an image on another page cannot do it.

Limiting Wrong Passwords

Django does not limit login attempts by itself. After 5 wrong passwords for one user name, the login answers `429` for 15 minutes, *even to the right password*. The count is kept for the *name* as typed, whether or not that name exists, so the reply never reveals which names are real. Three costs are worth knowing:

WHAT THIS LIMIT DOES NOT DO

- Someone who knows your user name can lock you out for 15 minutes. That is acceptable for a personal site; a public one would also count per address.
- The count lives in Django's cache, which by default is separate for each process. With several Gunicorn workers each keeps its own count, so the real limit is up to 5 *per worker*. A shared cache (Redis or the database cache) makes it exact. **Not tested here.**
- Behind a proxy every request seems to come from the proxy's address, which is another reason the count is by name and not by address.

Progress

A logged-in reader sees a button at the bottom of every page: *Mark as finished*, or, once done, *Mark as not finished*. The `/progress/` page lists each course on the site with how many chapters are finished. On the real content, five chapters of Hungarian Basic Conversation 3 marked finished gave “5 chapters finished” and “5 of 12” in 13 ms, and chapter 8 of the same course still showed the unfinished button.

Path, Not Foreign Key

A progress row stores the page's **path** (`hungary/x/c1.html`), not a link to the page row. The importer may delete and recreate a page, and a foreign key with the usual cascade would delete the reader's progress with it. With a path, a test deletes a page, recreates it and finds the progress still there. The cost is the other direction: if a file is *renamed*, its progress points at a path that no longer exists and counts for nothing. Renaming a chapter file is rare; losing progress on every re-import would not be.

The Safety Checks, Each With a Test

ATTEMPT	RESULT
Mark a page without logging in	302 to the login page; nothing saved
Mark by GET instead of POST	405
Mark without the CSRF token	403
Mark a page of another site, or a path that does not exist	404
Mark the same page twice	One row (a unique constraint)
Undo for a page that another user finished	That user's row is untouched
View the overview without logging in	302 to the login page
Delete a user	Their progress goes too

The redirect after marking is built from the page that was found, not from anything the visitor sent, so there is nothing to turn into an open redirect. The cached menus are unaffected, because nothing user-specific is cached. The page itself now differs for each logged-in reader, so a page that shows the button must never be cached as a whole.

WHAT WAS NOT VERIFIED

The buttons were not clicked in a real browser (the run used the test client, which sends the same requests); the login page was checked in headless Chrome. The lockout with several worker processes and the production cookie settings behind Apache were not tested. Nothing was run against PostgreSQL.

Hands-On Exercises

EXERCISE 1

Put the Django admin on its own host with its own URL configuration, make it read-only for pages, courses and progress, and keep the page body out of its lists. Test that an ordinary user, a staff user with no permissions and an attempted edit are all refused, and run it on the real database.

 [View solution](#)

EXERCISE 2

Add an optional, per-site login with no public sign-up. Limit wrong passwords per user name, make logging out a POST, and test the `?next=` attacks, the CSRF check and the lockout. State the limits of the lockout honestly.

 [View solution](#)

EXERCISE 3

Add a “Mark as finished” button and a per-course progress page. Store the page by path so progress survives a re-import, and test the wrong site, wrong user, no login, no CSRF token, a deleted page and a deleted user.

 [View solution](#)

CHAPTER 10 QUICK REFERENCE

- The admin lives on `admin.osztromok.com` with its own URL configuration; `/admin/` is a 404 on every content site

- The admin is read-only: the files are the source of truth and the importer would overwrite an edit
- No public sign-up: the owner makes users in the admin
- One login per site: the session cookie has no `Domain`; a shared login needs `SESSION_COOKIE_DOMAIN` and shares the cookie with every subdomain
- `?next=` is followed only on the same host; log out is a POST with the CSRF token
- 5 wrong passwords for a name give 429 for 15 minutes, counted by name; the cache is per process unless you use a shared one
- Progress is stored by **path**, so a re-import keeps it; a renamed file loses it
- Marking: POST, login and CSRF required; the page must be on this site; the redirect comes from the found page
- Real run: admin list of 4,408 pages in 130 ms; progress page 13 ms; 245 tests pass
- Not verified: real-browser clicks, several workers, production cookies behind Apache

Learning Website with Django

Chapter 11 · Migrating from the Old Site

The old site is live and has bookmarks, search results and links from other places pointing at it. A migration that just switches it off turns every one of them into a 404. This chapter does the opposite: it measures exactly what the old site answers, finds what is missing on the new sites, suggests redirects for a person to approve, and moves one site at a time. The numbers come from the real old site and the real new database.

RUN FOR REAL, ON DJANGO 6.1

The old site is the folder it is served from (`debservice/website`), checked against the 4,410 pages and 8,403 mirrored files of the new project. The project has 280 tests (all passing). **The Apache lines were not run**, and the redirect suggestions have not been reviewed by a person.

Step 1: Measure, Don't Remember

The old site is a folder of built files: `/hungary/x/y/` is the file `hungary/x/y/index.html` . That folder is the complete list of old addresses, so a command reads it and asks each new site about each address:

RESULT	MEANING
same	The old address answers 200 on the site that now owns its folder
redirected	It answers a redirect that ends on a page that exists
missing	Anything else, including a redirect to a page that does not exist

OLD SITE	COUNT	BEFORE ANY REDIRECT
Pages (<code>index.html</code>)	3,205	3,102 same (97.5% of the 3,182 a site owns), 80 missing
PDFs and solution files	5,831	5,544 in place, 166 missing (of the 5,710 a site owns)

OLD SITE	COUNT	BEFORE ANY REDIRECT
Owned by no site	144 pages and files	Mostly <code>resources/php</code> (83), <code>resources/js</code> (30) and <code>resources/csset</code> (19), plus the old front page and <code>/admin/</code>
Dynamic pages (<code>.php</code>)	23	The Anime Vault: not migrated at all

That is the useful result: **97.5% of old pages already have the same address**, because the folder structure was kept, and 80 do not. Reading the list showed why:

- **32 moved or renamed lessons**, for example `hungary/hungarian-language/...` became `hungary/hungarian-lessons/...`, and ten Japanese lessons that used to sit under `ai/claude-tools/claude-lessons/` are now under `japan/`.
- **23 `_print` pages**: the old site had a printable twin of many pages, and the new one does not.
- **25 others**: index pages of renamed folders, old “tentative” course folders that were removed, and `japan/kanji-tiles`.

The 166 missing files are mostly older copies of Node.js and Express solution files (120, in `text-files/` folders the courses no longer use) and 36 in `projects/nextjs-rebuild/solutions`. The command exits with an error until the list is empty, so it can be run again after every change.

Step 2: Redirects That Someone Reviews

An old address that moved needs a permanent (301) redirect. Redirects are *decisions*, not text imported from files, so they live in the database, are editable in the admin, and come from a CSV. The matching is deliberately modest:

1. If **exactly one** new page has the same file name (ignoring case, and `-` versus `_`), propose it.
2. If several do, list them for a person. If none do, say so.
3. A name ending in `_print` falls back to the page without it.

A file name is not proof of the same lesson, so every row of the CSV starts with an *empty* decision column, and only rows where you write `yes` are loaded. Loading a file with no “yes” does nothing (“0 created, 0 updated, 32 skipped”). On the real data the first version proposed 32

of the 80, and the `_print` rule added 23, giving **55 proposals, none ambiguous, 25 with nothing to suggest.**

```
site,old_path,new_site,new_path,why,decision
languages,hungary/hungarian-language/hungarian_lesson_01,languages,hungary/hungarian-les
ai,ai/claude-tools/claude-lessons/japanese_lesson_dates,languages,japan/japanese-language
```

The second row is a redirect to a *different site*: the old `ai` address becomes a full address on the languages site, built from the same site-address setting as the global bar. The redirect is tried only when there is no page and no folder listing at that address, so a redirect can never hide a real page (a test checks this), and a redirect belongs to one site.

WHAT THIS DOES AND DOES NOT PROVE

To measure the mechanism I loaded all 55 proposals into my scratch database with `--accept-all`, a test shortcut. The check then reported **3,102 same, 55 redirected, 25 missing** (before: 80 missing). I looked at a sample of the rows but **did not review all 55**; you should before any of them reaches a real database. A 301 is cached hard by browsers, so a wrong one is hard to undo. Redirects for PDF and solution files are not implemented, so the 166 missing files still need a decision.

Step 3: Links Between Sites

A page written for one site that links to `/linux/...` with a plain absolute path breaks after the split: that path is on the systems site. Editing thousands of files, and writing a host name into them, would be the wrong fix. Instead the link is rewritten **when the page is shown**: a link into another site's folders gets that site's address in front, taken from the same setting that differs between development and production. The stored page is unchanged (a test checks this).

On the real content the rewriter found **nothing to do**: of 11,691 internal links, every one points into its own site. That is a good result for the site map, and it means the rewriter is exercised only by tests today. It leaves alone external links, `//cdn...` links, relative links, `#fragments`, `/search/` and static files, and keeps query strings.

Broken Links

A second tool reads every stored page and checks each absolute internal link against the pages, folders, redirects and mirrored files. On the real data (3 seconds): **11,715 links checked, 483**

broken.

BROKEN LINKS	COUNT	WHAT IT IS
<code>/japan/hiragana/hiragana-tiles</code> and <code>/japan/katakana/katakana-tiles</code>	282	Tile pages that exist on the old site and not yet on the new one (like <code>japan/kanji-tiles</code>)
Solutions under <code>/web-development/scripting-and-backend/...</code>	174	The solution-link problem already noted for several courses in the review file
<code>/resources/japanese</code>	23	To check
Excel and web-application-security	4	To check

Step 4: One Site at a Time

While the old site still answers on `osztromok.com` , a site moves by telling the *old* host's Apache to send that site's folders to the new subdomain. A command prints the line for one or more sites:

```
python manage.py apache_cutover languages
RedirectMatch 301 ^/(france|germany|hungary|japan|culture|resources/japanese|resources/h
```

Without Apache, I ran the same pattern as a Python regular expression over every old address: of **8,892** page and file addresses that a site owns, **none** was matched by the wrong site or by two sites, and none of the 144 addresses that no site owns was matched. Tests check that no folder is claimed by two sites, that `/linuxfoo/` does not match the systems rule, and that sidebar pages follow their subject (for example `/sidebar/football/` goes to the humanities site).

WHAT WAS NOT VERIFIED

The Apache lines were not run (there is no Apache here). Before using them: `apachectl configtest` , then `curl -I` on one address of that site, expecting a 301 and the right `Location` , and keep the old files in place until you have checked. The redirect suggestions are unreviewed, and nothing was tried against a browser's cache.

Hands-On Exercises

EXERCISE 1

Write the command that lists every address of the old site (a folder of built files) and checks each against the new sites as same, redirected or missing, failing until nothing is missing. Run it on the real old site and explain what the missing ones turned out to be.

 [View solution](#)

EXERCISE 2

Build the redirect table, the view hook and the suggest-then-review CSV workflow, matching by file name only when exactly one page fits. Run it on the real data, load the suggestions into a scratch database, measure again, and list what still needs a person.

 [View solution](#)

EXERCISE 3

Rewrite cross-site links when a page is shown, write the broken-link checker, and generate the per-site Apache redirect lines. Test the Apache pattern against every real old address, and say clearly what could not be run.

 [View solution](#)

CHAPTER 11 QUICK REFERENCE

- Measure the old site from its own folder: 3,205 pages, 5,831 PDFs and solution files, 23 dynamic pages
- 97.5% of old pages already had the same address (3,102 of 3,182); 80 did not
- Results: same, redirected (must end on a real page), missing; the command fails until nothing is missing
- Redirects are decisions: a database table, editable in the admin, loaded from a CSV where only rows marked `yes` count
- Matching: exactly one page with the same file name; several or none are left for a person; `_print` falls back to the page

- A redirect never hides a real page, belongs to one site, and may point to another site's full address
- Cross-site links are rewritten when shown, not in the files; real content had none (11,691 internal links, all within their own site)
- Broken-link check on real data: 483 of 11,715, mostly the hiragana and katakana tile pages and old solution links
- `apache_cutover <site>` prints one `RedirectMatch 301` line; tested as a regular expression against 8,892 real addresses; not run in Apache
- Not done: redirects for PDF and solution files, the tile pages, and a person's review of the 55 suggestions

Learning Website with Django

Chapter 12 · Deployment, Testing & Operations

The sites work on a development machine. This chapter is about the part that decides whether they keep working: running them with the production settings, knowing when something is wrong, backing up what cannot be rebuilt, and releasing a new version in a way that can be undone. It is also where honesty matters most, because the target is a Debian server and this was written on Windows, so a large part of it **could not be run**. Each section says which part was.

WHAT WAS RUN AND WHAT WAS NOT

Run: Django's deployment check, the production settings on a real server process answering real HTTP (90 checks, 0 failed), a database backup, a restore, a full rebuild from the files, and 304 tests (all passing). **Not run:** Gunicorn (it needs Linux), Apache, systemd, cron, certificates, and the deploy script (its syntax was checked, nothing more). Treat those as a first draft to try on the server, one step at a time.

Where Each Thing Runs

PART	JOB	RUN HERE?
Apache	HTTPS, certificates, the redirect from HTTP, PDFs and solutions straight from disk, everything else passed on	No
Gunicorn	Runs Django (3 workers, 2 threads each) on a unix socket only Apache can reach	No
Django + WhiteNoise	Pages, search, sitemaps, logins; the theme's static files with hashed names	Yes, with the production settings
SQLite	Pages, search index, accounts, progress, redirects	Yes
Content folder	The source of truth for pages; rebuilds the database	Yes

Step 1: Check the Production Settings

The test suite runs under the development settings, so it cannot say whether the production ones work. Django's deployment check can: `manage.py check --deploy` with the production settings reports two warnings, about HSTS for subdomains and HSTS preloading. They are **left on deliberately**. Including subdomains would make browsers insist on HTTPS for every name under `osztromok.com`, including any old host without a certificate, and preloading is close to permanent. HSTS is set to 3600 seconds; raise it in steps only when every host is known to work over HTTPS.

Step 2: Know When It Is Broken

Every site (and the admin host) has a `/healthz/` address. It answers `200` only when the process can really serve: the database answers *and* holds pages (for a site, that site's pages). Otherwise it answers `503`. It is never cached and says nothing else, so a monitor or the deploy script can call it freely.

The **smoke test** goes further. It sends real HTTP requests to a running copy and checks, for each site: health, the front page, that the front page links a stylesheet and the stylesheet loads, `robots.txt`, the sitemap, search, and a few pages from the sitemap whose canonical link must equal the sitemap address. For the admin it checks that it asks for a login and that `/admin/` is a 404 on the content sites. Without a connect address it uses each site's real name, so DNS, the certificate and Apache's choice of virtual host are tested too.

RUN (PRODUCTION SETTINGS, DEVELOPMENT SERVER STANDING IN FOR GUNICORN)	RESULT
<code>curl</code> plain HTTP with no proxy header	301 to <code>https://languages.osztromok.com/</code> (correct)
<code>smoke_test</code> with <code>--forwarded-proto https</code> , as Apache would send	90 checks, 0 failed in 6 seconds, 8 sites and the admin
The same without that header	Fails (every request is a 301); exit code 1
Nothing listening	"could not connect" (a clear error, not a traceback)

The failing runs matter as much as the passing one: a check that has never been seen to fail has not been shown to check anything.

Step 3: Back Up What the Files Cannot Rebuild

The pages are in the content folder and can always be rebuilt. But accounts, reading progress and redirect decisions exist *only* in the database, and so does the search index (which takes a minute to rebuild). SQLite has an online backup that is safe while the site runs. Copying the file with `cp` is not: a write in the middle gives a damaged copy.

- The backup command copies with SQLite's own backup, then **opens the copy** the way a restore would: integrity check, the page and search tables present, pages not zero. A copy that fails is deleted and reported.
- It keeps the newest few (14 in the cron entry) and deletes *only* files named exactly like its own, so a file you put in the folder is safe.
- A test found a real bug: a damaged file made the check *raise* instead of reporting "not ok". It now reports the reason.

REAL RUN	RESULT
Back up the real database	4,410 pages, 4,410 indexed, 159 MB, 12 seconds
Restore it (copy in, point <code>LW_DATABASE</code> at it) and search	4,410 pages; "szeretnek" 18 results, "ありがとう" 25
Rebuild from nothing: empty database, <code>migrate</code> , <code>import_content</code>	4,411 pages in 42 seconds; the same search answers

So pages and search come back from the files in about a minute; accounts, progress and redirects come back only from a backup. A backup kept on the same disk does not survive the disk, so copy them elsewhere and check the copy.

Step 4: Apache, Certificates and Gunicorn (Not Run)

The Apache configuration is **generated from the site map** (`apache_vhosts`), so it cannot disagree with it: one HTTPS host for each of the eight sites and the admin host, an HTTP host for each that redirects to HTTPS, the site's own asset folder served by Apache, and `X-Forwarded-Proto` set so Django knows the visitor used HTTPS. The asset pattern was tested against good and bad addresses, including `../pdfs/a.pdf` and `/x/./pdfs/a.pdf`, which it must not match.

Certificates

- **One wildcard certificate** for `osztromok.com` and `*.osztromok.com`: needs the DNS challenge, and a new site needs no new certificate. The generated configuration assumes this.
- **One certificate per name** (nine names now): no DNS access needed, but every new site is a new certificate and a new place for renewal to fail.

A certificate that quietly expires is the most common way sites like this fail. The smoke test, run from outside by name over HTTPS, notices an expired or wrong certificate.

Gunicorn

A small configuration file (3 workers, 2 threads, a 30-second timeout, workers restarted after about 2,000 requests so a slow leak cannot grow forever) and a systemd unit that runs it on a unix socket as an unprivileged user, with a read-only file system apart from the data, asset and backup folders. The secret key lives in an environment file readable only by that user; the production settings refuse to start without it.

Step 5: A Release With a Way Back

```
deploy.sh release <git-ref>
  1. new folder for the release (a git worktree) + install requirements
  2. backup the database           # migrations only go forward
  3. check --deploy, migrate, collectstatic, import_content, collect_assets
  4. switch the current symlink, reload Gunicorn
  5. smoke test over HTTPS; if it fails, roll back at once

deploy.sh rollback           # point the symlink at the previous release, reload
```

THE HONEST LIMIT OF A ROLLBACK

Rolling back restores the old *code*. If the release ran a migration, the old code may not match the new database. Keep migrations backward compatible (add a column in one release, use it in the next), or restore the backup taken at the start of that release, which loses anything written since. The script says so when it rolls back. It passes a syntax check only: **it was never run**, and it needs a sudoers rule allowing the deploy user to reload that one service.

Hands-On Exercises

EXERCISE 1

Run Django's deployment check with the production settings and decide each warning. Add a health address to every site, write a smoke test that uses real HTTP, and run it against a server started with the production settings, showing both a passing and a failing run.

 [View solution](#)

EXERCISE 2

Write a database backup that is safe while the site runs, checks its own copy, keeps only the newest few and never deletes a file it did not make. Back up the real database, restore it and search it, and time a rebuild from the content files.

 [View solution](#)

EXERCISE 3

Generate the Apache virtual hosts from the site map, and write the Gunicorn configuration, the systemd unit and a release script with a rollback. Test what can be tested, and state plainly which parts were not run.

 [View solution](#)

CHAPTER 12 QUICK REFERENCE

- `check --deploy` with the production settings; the two HSTS warnings are left on until every host is known to work over HTTPS
- `/healthz/` on every site: 200 only if the database answers and holds pages; never cached
- `smoke_test`: real HTTP to every site; by name over HTTPS it also tests DNS, the certificate and the virtual host
- Real result: 90 checks, 0 failed under the production settings; fails (exit code 1) without the proxy header

- Back up with SQLite's online backup, never `cp`; the copy is opened and checked; only the newest few are kept
- Real result: 159 MB backed up in 12 s, restored and searched; full rebuild from the files in 42 s
- Only the database holds accounts, progress, redirects and the search index; the pages always come back from the files
- Apache hosts are generated from the site map (`apache_vhosts`); one wildcard certificate covers every site
- Release: backup, checks, migrate, collectstatic, import, switch symlink, reload, smoke test, roll back if it fails
- Not run: Gunicorn, Apache, systemd, cron, certificates, `deploy.sh` (syntax checked only)

Course Complete

You have built a Django project that serves eight sites from one codebase, one database and one design system, and the tools to move the old site onto it without breaking its addresses. What each chapter left you able to do:

CHAPTER	YOU CAN NOW...
1 · Project Layout for Many Sites	Lay out one project with a site map as its single source of truth
2 · Subdomain Routing	Pick the site from the host name, with a separate URL configuration per site
3 · The Content App	Model courses and pages and read the banner comment of each file
4 · Loading Content from Files	Import 4,400 pages with change detection, re-runnable and safe
5 · Templates & the Shared Design System	Share one theme across sites, each with its own accent colour
6 · Navigation & Breadcrumbs	Derive menus, breadcrumbs and chapter links from the data
7 · The Languages Site	Give one site its own front page and lesson-specific handling
8 · PDFs, Solutions & Static Files	Serve three kinds of file by three routes, with safe caching
9 · Search & Sitemaps	Add per-site full-text search, sitemaps, canonical links and robots.txt
10 · Admin, Accounts & Progress	Add a read-only admin on its own host, optional logins and progress

CHAPTER	YOU CAN NOW...
11 · Migrating from the Old Site	Measure old addresses, review redirects and cut over one site at a time
12 · Deployment, Testing & Operations	Check production settings, monitor, back up, restore and release with a way back

WHAT IS STILL OPEN

The server side is still a plan: Apache, Gunicorn, systemd, cron, certificates and the release script all need trying on Debian, and the 55 redirect suggestions need reviewing by a person. The next course, Learning Website with Next.js, builds the same sites on a different stack so the two can be compared.