



WEBSITE ARCHITECTURE COURSE

Learning Website: Framework & Architecture

A blueprint for rebuilding a large learning site as a family of smaller sites: whether to split, mapping content to sites, the content model, a shared design system, navigation, the content pipeline, search and sitemaps, hosting and deployment, accounts, migrating one area at a time, redirects, and launch. Every chapter is backed by scripts run on the real site.

12 Chapters

Philip Osztromok

Generated with Claude

Contents

Chapter 1	Why Split the Site	2
Chapter 2	Mapping Content to Sites	9
Chapter 3	The Content Model	15
Chapter 4	The Shared Design System	21
Chapter 5	Navigation Across Sites	28
Chapter 6	The Content Pipeline	34
Chapter 7	Search, Sitemaps & SEO	40
Chapter 8	Hosting & Deployment	47
Chapter 9	Accounts, Progress & Shared Services	54
Chapter 10	Migrating One Area at a Time	59
Chapter 11	Rewriting Links & Redirects	65
Chapter 12	Testing, Launch & Operations	70

Learning Website: Framework & Architecture

Chapter 1 · Why Split the Site

This learning site began as a handful of pages and has grown into a very large collection of courses, lessons, cheat sheets and tools. Growth is the point, but a single site that holds everything has costs that grow with it: longer builds, a bigger navigation to understand, and one deployment that every change has to pass through. This course is a blueprint for rebuilding the site as a family of smaller sites, such as `languages.osztromok.com` and `webdevelopment.osztromok.com`, with `osztromok.com` itself eventually becoming a portfolio and landing page. This first chapter asks whether that is worth doing, and gives you a way to decide.

A DECISION, NOT A GIVEN

Splitting is a trade, not an upgrade. Part of this chapter argues *against* splitting, because the best reason to split is one you can name and measure. Chapters 2 to 12 assume you have decided to go ahead, starting with the languages area, and keep the old site running while you do.

What Hurts on One Big Site

Before choosing a solution, list the actual problems. These are the ones a single large content site tends to meet:

PROBLEM	WHAT YOU NOTICE
Build time	A static site generator builds every page on every deploy, so the build grows with the page count. A small change to one chapter costs a full rebuild.
Navigation overload	Menus have to list every subject, so a visitor looking for one language course sees dozens of unrelated ones.
One deployment for everything	A mistake in one area can break the whole site, and a fix for one area waits behind unrelated work.
Different needs per area	The language courses have their own colours, dialog blocks and audience. A programming course needs code blocks and runnable examples. One shared layout serves neither well.
Unmanageable content tree	The folder structure grows deeper and wider, and rules for each content type pile up in one place.

Count these for your own site before you decide. Exercise 1 at the end of the chapter shows how to measure how many pages each area holds, which tells you whether the problem is real.

What a Subdomain Is

A **subdomain** is a name added in front of your domain: `languages` in `languages.osztromok.com`. It is set up in DNS, and it can point at a different folder, a different application or even a different server from the main site. A **path** is the part after the host name: `/languages/` in `osztromok.com/languages/`. A path always lives inside one site.

```
; DNS zone file entries (203.0.113.10 is a documentation address)
osztromok.com.      A      203.0.113.10
languages           A      203.0.113.10
webdevelopment     A      203.0.113.10

; or, to cover every subdomain at once (a wildcard record)
*                  A      203.0.113.10
```

All three names above point at the same machine. The web server (Apache or Nginx, covered in Learning Website: Framework & Architecture 8) looks at the host name in each request and decides which site to serve. So a subdomain split does not need more servers; it needs more configuration.

What a Subdomain Split Gives You

- **Independent builds and deploys.** The languages site rebuilds only when language content changes. A failed deploy of one site leaves the others alone.
- **A smaller, focused site.** Navigation, search and layout only describe one area, so a visitor sees what is relevant to them.
- **Area-specific design.** Each site can have its own accent colour, components and rules, while still sharing a common base (Learning Website: Framework & Architecture 4).
- **Room to choose technology per site.** The languages site could be built differently from a future site that needs a database or user accounts.
- **A clear public structure.** `osztromok.com` can become a short portfolio page that links out to each area.

What It Costs

The costs are real, and most of them come from one fact: a subdomain is a different *origin*.

Origins and sites

A browser treats two URLs as the same **origin** only when the scheme, host and port all match.

`languages.osztromok.com` and `osztromok.com` have different hosts, so they are different origins, although they are still the same **site** because they share the registrable domain `osztromok.com`. Exercise 3 turns this into a small script you can run.

FEATURE	PATH SPLIT (ONE ORIGIN)	SUBDOMAIN SPLIT (SEVERAL ORIGINS, ONE SITE)
localStorage and sessionStorage	Shared by every page	Separate for each subdomain
Cookies	Shared	Shared only if set with a <code>Domain</code> attribute for the parent domain
fetch() from one area to another	Allowed	Blocked unless the other site sends CORS headers
Shared fonts and assets	Simple	Cross-origin fonts need CORS headers; a shared asset host is often easier
Links between areas	Relative paths	Absolute URLs, kept in one configuration file

SHARING A LOGIN IS A DESIGN DECISION

A login cookie set for `osztromok.com` with the `Domain` attribute is sent to every subdomain, which is convenient and also widens the damage if one subdomain is compromised. This is why the optional accounts chapter (Learning Website: Framework & Architecture 9) is kept separate: decide whether you need shared accounts at all before you build the sites.

Certificates and setup

Every hostname needs to be covered by an HTTPS certificate. You can get one certificate per subdomain, or one **wildcard** certificate for `*.osztromok.com`. A wildcard covers a single label only, so `languages.osztromok.com` is covered but `a.languages.osztromok.com` is not. With Let's Encrypt a wildcard certificate has to be requested using DNS validation, which means

automating changes to your DNS records. You also add a virtual host to the web server for each site.

Search and discoverability

Search engines can index subdomains, and Google has said it can handle either subdomains or paths. The practical differences are smaller than the folklore suggests, but they are real: each subdomain is crawled and reported separately, you have more places to keep sitemaps and redirects correct, and any search ranking the old combined site earned has to be carried over carefully when you move content (Learning Website: Framework & Architecture 7 and 11). A “domain” property in Google Search Console covers every subdomain, so you can still see the whole picture in one place.

Duplicated effort

Anything shared now has to be shared on purpose: the top navigation, the footer, the colour scheme, the code-copy script, the sitemap logic. A split without a shared design package turns into several slightly different copies of the same site, which is worse than the single site you started with.

Subdomain or Path?

There is a middle option, and for many sites it is the right one: keep one domain and one origin, but split the content and the build. The area still appears at `osztromok.com/languages/`, while the web server or a small proxy sends that path to a separate, independently built site. You get independent builds and deploys without any of the origin-related costs above.

```
# Option A: path split (one origin)
osztromok.com/
├─ /                -> portfolio / landing page
├─ /languages/      -> built from the languages project
└─ /webdevelopment/ -> built from the web development project

# Option B: subdomain split (several origins, one site)
osztromok.com        -> portfolio / landing page
languages.osztromok.com -> built from the languages project
webdevelopment.osztromok.com -> built from the web development project
```

A path split is simpler to run. A subdomain split gives more independence and a clearer identity for each area. This course assumes subdomains because that is the direction you have chosen, but every design in it (the shared content model, the design package, the configuration of links) works for a path split as well, so the decision can be revisited.

A Way to Decide

Score each area, not the whole site. An area is a good candidate for its own site when most of these are true:

SIGNAL	POINTS TOWARDS
It is large, or growing fast	Its own site (build time, navigation)
Its audience differs from the rest	Its own site (separate identity)
It has its own design rules	Its own site with a shared base
It needs different technology	Its own site
Visitors move often between it and other areas	A path split, or keep it together
It needs login shared with other areas	A path split, or a deliberate shared-login design
It is small and stable	Keep it where it is

WHAT THIS SUGGESTS FOR THE LANGUAGES AREA

The language courses are large and growing, have their own accent colours and dialog and vocabulary blocks, and form volumes that are read as a group, which is why they are the first candidate. Many learners will not need to cross over to the programming courses, and no shared login is needed for static content. That fits a separate site. It is a judgement from the criteria above, not a law, and Exercise 2 asks you to write it down properly.

Staying Honest About the Cost

Whatever you decide, plan to keep the old site running while the new one grows. You will not move everything in one weekend, and you should not have to. Moving one area at a time means

each step is small, can be tested, and can be undone. Learning Website: Framework & Architecture 10 describes this approach, usually called the *strangler* pattern, in detail.

Hands-On Exercises

EXERCISE 1

Measure your site. Count the HTML files under each top-level folder of `content/`, rank the areas by size, and write down which areas are big, which are growing fastest, and which would still need many links to each other after a split.

 [View solution](#)

EXERCISE 2

Write a one-page decision record for the languages area. State the context, the two options (path or subdomain), at least six criteria with scores, your decision, and its consequences. Say which criteria you weighted most and why.

 [View solution](#)

EXERCISE 3

Write a short Python script that takes two URLs and reports whether they are *same-origin*, *same-site but cross-origin*, or *cross-site*. Test it on a path pair, two subdomains, an http/https pair and a different port. Explain what each result means for cookies and localStorage.

 [View solution](#)

CHAPTER 1 QUICK REFERENCE

- **Reasons to split:** build time, navigation overload, one deployment for everything, different needs per area
- **Subdomain** = a name in DNS (`languages.osztromok.com`); **path** = part of a URL inside one site (`/languages/`)
- **Origin** = scheme + host + port; **site** = scheme + registrable domain

- A subdomain split makes **several origins within one site**: localStorage is separate, cookies are shared only with a parent `Domain`, and fetch needs CORS
- A **wildcard certificate** (`*.osztromok.com`) covers one label only, and Let's Encrypt needs DNS validation to issue one
- A path split can give independent builds without the origin costs
- Score **each area** (size, growth, audience, design, technology, cross-links, shared login), not the whole site
- Keep the old site running and migrate one area at a time

Learning Website: Framework & Architecture

Chapter 2 · Mapping Content to Sites

Chapter 1 argued that the site should be split. Before building anything, you need an answer to a simpler question: *which content goes on which site?* Get this wrong and every later decision (navigation, redirects, builds) inherits the mistake. This chapter works from the real `content/` folder, builds a map from top-level folders to sites, and turns that map into data a script can check.

Start from What Exists

The `content/` folder has **48 top-level subject folders**, plus a special `sidebar/` folder that holds link pages, tools, standalone lessons and cheat sheets for several subjects. Here are a few of them, to show how varied the content is:

```
content/
├─ france/  germany/  hungary/  japan/           # language courses
├─ web-development/  web-servers/  web-platforms/
├─ programming/  software-development/  databases/  projects/
├─ linux/  operating-systems/  networking/  security/  cloud-and-devops/
├─ ai/  data-science-and-ml/
├─ history/  politics/  philosophy/  music/  science-fiction/
├─ cookery/  practical-life-skills/  youtube/
├─ sidebar/
│   └─ ai/  drinks/  football/  history/
│   └─ linux/  programming/  web-development/
└─ ... # and 22 more
```

This is the raw material. Some folders are obviously related (`web-development`, `web-servers`, `web-platforms`), some belong to a clear audience (`france`, `germany`, `hungary`, `japan` are all someone learning a language), and some are on their own (`cookery`, `football`).

Principles for Grouping

Use these in order, and expect to adjust rather than get it perfect first time:

1. **Group by audience first.** Who comes to this content, and what else do they want? A language learner and a Linux administrator are different visitors.
2. **Group by shared design.** Content that uses the same components (dialog blocks, code blocks, timelines) belongs on one site, so the design package serves it well.
3. **Keep cross-links short.** If two folders link to each other constantly, put them on the same site. Cross-site links work, but they are absolute URLs and a different origin (Learning Website: Framework & Architecture 1).
4. **Balance size and growth.** A site that is ten times bigger than the others has not solved the original problem. Neither does a site with one course in it.
5. **One home per page.** Every page lives on exactly one site. Other sites link to it and never copy it.

A Proposed Map

The map below is a starting proposal, built by applying those principles to the real folders. It is a judgement and not a law: you will move some folders after you see it laid out, and that is the point of writing it down.

SITE (SUBDOMAIN)	FOLDERS IT HOLDS	AUDIENCE
languages	france, germany, hungary, japan, culture	People learning a language
webdevelopment	web-development, web-platforms, web-servers	People building and hosting websites
programming	programming, software-development, maths-for-programmers, retro-computing, game-development, android-development, ios-development, databases, blockchain-and-web3, developer-tools, projects	People learning to write software
systems	linux, operating-systems, networking, cloud-and-devops, security, technical-support, raspberry-pi, windows	Administrators and support engineers
ai	ai, data-science-and-ml	People working with AI and data
humanities	history, politics, philosophy, art-history, classic-literature, science-fiction, sci-fi-project, music, football, ethics, study-	General readers

SITE (SUBDOMAIN)	FOLDERS IT HOLDS	AUDIENCE
	methodologies, science	
lifeskills	practical-life-skills, cookery, freelancing-and-business-skills, youtube	Everyday and side-income skills
creative	creative-and-design-tools, audio-video-production, office-and-productivity-software	People using creative and office software

NOT ALL OF THESE HAVE TO EXIST AT ONCE

Chapter 1 chose to start with languages. This map is the destination, not a plan to build eight sites immediately. The areas you do not move yet simply stay on the existing site. If the map looks too fine grained, merge sites: **humanities** and **lifeskills** could be one general site, for example. What matters is that the map is complete, which the exercise at the end checks by script.

Special Cases

The sidebar folder

sidebar/ is organised by subject, with links, tools, lessons and cheat sheets for each. It is not a subject itself, so it does not get a site of its own. Instead, route each page by its *subject folder*: **sidebar/linux/** goes with the systems site, **sidebar/web-development/** with the webdevelopment site, **sidebar/football/** with humanities, and so on. A Linux cheat sheet then appears next to the Linux courses.

Folders that could go either way

databases is useful for web and non-web work. **projects** mixes web projects with a database engine and an operating system kernel. **culture** (Japan) could sit with languages or with the humanities. Pick one home for each, write down the reason, and let the other sites link to it. Exercise 3 asks you to do exactly that.

Shared pages: the landing page and tools

A few things belong to no area: a site-wide search page, an “about” page, and a list of every site. These live on the main **osztromok.com** domain, which Chapter 1 suggested becoming a portfolio and landing page.

Make the Map Data, Not Prose

A table in a document is for people. The build needs the same information in a form a script can read. Put the map in one file and let everything else (navigation, builds, redirects) read it:

```
# sites.py (excerpt): one source of truth
SITES = {
    "languages": ["france", "germany", "hungary", "japan", "culture"],
    "webdevelopment": ["web-development", "web-platforms", "web-servers"],
    # ... one entry per site
}

# pages under sidebar/ are routed by their subject folder
SIDEBAR_ROUTES = {
    "linux": "systems",
    "web-development": "webdevelopment",
    # ...
}
```

Two properties make this map trustworthy. It is **complete**: every folder on disk appears in it. And it is **unambiguous**: no folder appears twice. A short script can check both every time the folder tree changes, which is the first exercise below.

Naming the Sites

The site name becomes a subdomain label, so it has to be a valid one: letters, digits and hyphens, no spaces or underscores, and at most 63 characters. Beyond that, follow the same rules you use for folders:

- **Short and recognisable.** `languages` is clearer than `langs` and shorter than `language-learning`.
- **Stable.** Renaming a site later means changing every cross-site link and every redirect.
- **One style.** If you use a hyphen in one name, use hyphens in all names. The names above are single words, which avoids the question.
- **Describe the content, not the technology.** `webdevelopment` will still be right if you change framework.

Keep the Path the Same

When a folder moves to a new site, keep everything after the site name unchanged. The page that lives at `japan/hiragana-1/` under `content/` would live at `languages.osztromok.com/japan/hiragana-1/`. Three benefits follow. The redirect rule is a mechanical rewrite of the host name. Links inside the content need little change. And you can predict a page's new address without looking it up.

CHECK THE LIVE URL PATTERN BEFORE YOU RELY ON THIS

This assumes the live site's URLs follow the folder paths. Open several real course pages on the current site and compare their addresses with the paths under `content/`. Where the two differ (the kanji pages, for example, are served from a separate resources path), note the exceptions now and handle them in the redirect map (Learning Website: Framework & Architecture 11).

Hands-On Exercises

EXERCISE 1

Write `check_sites.py`. Given your site map and the path to `content/`, report every folder that is unassigned or assigned twice, any mapped folder that does not exist, and any `sidebar/` subject with no route. Run it on your real folders.

 [View solution](#)

EXERCISE 2

Write a function `site_for(path)` that takes a content-relative path and returns its site, including pages under `sidebar/<subject>/`. Make it fail loudly for a path that belongs nowhere, and test it on at least five paths.

 [View solution](#)

EXERCISE 3

Choose a single home for `databases`, `projects` and `culture`, with a written reason for each. Then write the Apache `RedirectMatch` rule that would send the old language URLs to

`languages.osztromok.com`, keeping the path the same.

 View solution

CHAPTER 2 QUICK REFERENCE

- The `content/` folder has **48 top-level subject folders** plus `sidebar/`
- Group by **audience**, then **shared design**, then **cross-links**, then size and growth
- **One home per page:** other sites link to it, never copy it
- Proposed sites: languages, webdevelopment, programming, systems, ai, humanities, lifeskills, creative
- `sidebar/<subject>/` pages are routed by subject, not given a site of their own
- Keep the map as **data** (one file) and check it is complete and unambiguous by script
- A site name is a subdomain label: letters, digits, hyphens, at most 63 characters; keep it short and stable
- Keep the path after the site name unchanged so redirects are a host-name rewrite
- Check the live URL pattern against the folder paths before relying on that rule

Learning Website: Framework & Architecture

Chapter 3 · The Content Model

Chapter 2 decided which folder belongs to which site. Before any site can be built, you need to know what a *page* is: what kinds of page exist, where their information lives, and how a script can read it. On this site there is no database. The files are the database, and the content model is the description of how to read them. This chapter inspects the real files, finds the shapes they actually take, and defines a model that copes with all of them.

EVERY EXAMPLE HERE WAS RUN

The scripts in this chapter's exercises were run on your real `content/` folder, and the outputs quoted in the solutions are what they printed. Where the real files turned out messier than the rules describe, the chapter says so.

The Kinds of Page

Several kinds of page share the `content/` folder. They differ in where they live, how they are named and what information they carry:

KIND	WHERE IT LIVES	NAMING	NOTES
Course chapter	<code><subject>/<course>/</code>	<code><course_name>_<course#>_<chapter#>.html</code>	Numbered; has a banner with Course and Chapter; may have <code>pdfs/</code> and <code>solutions/</code>
Sidebar page	<code>sidebar/<subject>/</code>	<code>links_</code> , <code>sidebar_</code> , <code>cheat_sheet_</code> prefixes; tools use none	Links, tools, standalone lessons, cheat sheets; banner has Category and Subcategory
Standalone lesson	A lessons folder, such as <code>python-lessons/</code>	<code><language>_lesson_<topic>.html</code>	Not numbered; a free-form banner

KIND	WHERE IT LIVES	NAMING	NOTES
Reference page	Alongside lessons or under reference folders	Descriptive, such as <code>hungarian_alphabet.html</code>	A free-form banner with no title line
Kanji page	Two copies (archive and live)	<code>kanji_<character>.html</code>	A complete HTML page with its own <code><title></code> , not a fragment

Cheat sheets also have a printable twin: `cheat_sheet_git_command_line.html` sits beside `cheat_sheet_git_command_line_print.html`. The model has to know the second file is a variant of the first and not a page of its own.

Anatomy of a Course Folder

```
linux/system-administration/debian-development-machine-setup/
├─ _outline.md                    # planning notes, not published
├─ debian_development_machine_setup_1_1.html  # chapter files sit directly in the folder
├─ debian_development_machine_setup_1_2.html
├─ ...
├─ pdfs/                        # per-chapter PDFs and the course PDF
└─ solutions/
    ├─ devsetup1-1_exercise1.txt    # keyed by the short prompt prefix
    └─ devsetup1-1_exercise2.txt
```

Three details matter for the build. Files that start with an underscore (`_outline.md`) are working notes and are never published. The chapter files follow

`<course_name>_<course#>_<chapter#>.html`, so the course number and chapter number can be read from the filename. And solution files use the *short* prompt prefix (`devsetup1`), not the long course name, so they cannot be found by guessing from the chapter filename. The chapter page links to them, and the build must follow the links.

Fragments and Full Pages

Nearly everything is an **HTML fragment**: a banner comment, a `<style>` block and markup, with no `<html>`, `<head>` or `<body>`. The site wraps each fragment in its own layout. The kanji pages are the exception. They are complete HTML pages, with their own head and title, and so they need different handling: the build must not wrap a full page inside a second page.

The Banner Shapes

Because the site has been built over a long time, the banner comment comes in several shapes. A parser that handles only the newest will silently mislabel the rest. These are the shapes that really exist:

SHAPE	FIELDS	EXAMPLE
Course chapter	Course, Chapter, File, Topic, (Level), dates when present	Hungarian Basic Conversation 3, chapter 1
Course chapter, older	The same, with no dates	Apache In Depth, chapter 1
Sidebar	Category, Subcategory, Chapter, File, dates	The Git command line cheat sheet
Free-form with a headline	A line such as <code>PYTHON LESSON: ...</code> , then Level and dates	The Python boolean logic lesson
Free-form without a title	Level and Focus only	The Hungarian alphabet reference
None (full page)	Nothing: the title is in <code><title></code>	The kanji pages

TOLERATE THE PAST, FIX IT GRADUALLY

You could rewrite every old banner to a single format, but that touches a very large number of files and risks breaking pages that work. A better approach is to make the *parser* accept every shape and return only the fields that are really there. New content then follows the current rules, and old content is updated only when it is rewritten for another reason.

Where Each Field Comes From

Every piece of information about a page lives in exactly one natural place, and the model should read it from there:

FIELD	READ IT FROM	NOTES
Subject and site	The first folder in the path	Through the site map from Learning Website: Framework & Architecture 2
Course name, chapter title	The banner	The reader-facing text, so it needs no guessing
Course number, chapter number	The filename	The banner does not carry them
Created and updated dates	The banner	Optional: missing on older pages
Chapter PDF	The <code>pdfs/</code> folder next to the chapter	Optional: not every course has per-chapter PDFs
Solution files	Links inside the chapter	Follow the links; do not guess filenames
Kind	Banner shape plus folder	Course chapter, sidebar page, lesson, reference or full page

The dates are worth keeping carefully. The site's own rule is that the created date never changes and the updated date moves when the content really changes. That makes the updated date a ready-made value for the `lastmod` field in a sitemap and for a “recently updated” list (Learning Website: Framework & Architecture 7 uses both).

A Model That Survives Real Data

A `Page` record gathers those fields in one place. The design choices that keep it from breaking on old content:

- **Optional fields are really optional.** Dates, PDFs and course numbers can be `None`. A page is not invalid for lacking them.
- **Paths are relative to `content/` and use forward slashes**, so the same record works on Windows, on the Debian server and in a URL.
- **The record holds metadata, not the HTML.** Read the fragment from disk when you build the page, so the index stays small.
- **Sort numbers as numbers.** Filenames sort as text, so chapter 10 appears before chapter 2. A chapter list built from a directory listing is wrong until it sorts on the chapter number.

- **Skip what is not a page:** files beginning with an underscore, print variants, `pdfs/` and `solutions/`.

```
@dataclass
class Page:
    path: str                # relative to content/, forward slashes
    site: str                # from the site map
    kind: str                # course-chapter | sidebar | free-banner | full-page
    title: Optional[str]
    course: Optional[str] = None
    course_no: Optional[int] = None
    chapter_no: Optional[int] = None
    created: Optional[str] = None
    updated: Optional[str] = None
    pdf: Optional[str] = None
```

DO NOT STORE THE SAME FACT TWICE

The model reads the course name from the banner and the course number from the filename, and it does not copy either into a separate list. If a second file ever holds the same information (a hand-edited course index, for example), the two will drift apart. Prefer deriving the data from the files every time the site is built.

Validate Before You Build

Once pages are records, content problems become checkable: a banner whose `File:` line does not match the real filename, a chapter missing from the sequence, a file that does not follow the naming pattern. A validator that runs before the build finds these cheaply. Exercise 3 writes one and runs it on two real courses of different ages. Both pass, and the output shows exactly why the model has to treat dates and PDFs as optional.

Hands-On Exercises

EXERCISE 1

Write `parse_banner(text)` so it handles every banner shape listed in this chapter, including pages with no banner. Run it on one real file of each shape and check the fields it returns.

 [View solution](#)

EXERCISE 2

Define a `Page` data type and a `load_page()` function that builds one from a file path: site, kind, title, course and chapter numbers from the filename, the dates, and the chapter PDF if it exists. Run it on a new course chapter and an older one.

 [View solution](#)

EXERCISE 3

Write `validate_course.py`, which indexes one course folder and reports a wrong banner kind, a `File:` line that does not match the filename, a filename that breaks the pattern, and gaps in the chapter numbers. Run it on a recent course and an older one and explain the differences.

 [View solution](#)

CHAPTER 3 QUICK REFERENCE

- **The files are the database:** the content model describes how to read them
- Kinds of page: course chapter, sidebar page, standalone lesson, reference page, kanji (full) page
- Nearly all pages are **fragments**; kanji pages are full HTML pages and must not be wrapped again
- Banner shapes: course chapter (with or without dates), sidebar, free-form with a headline, free-form without a title, none
- Course and chapter **numbers come from the filename**; names, titles and dates from the banner
- Solution files use the **short prefix**, so follow the links in the chapter instead of guessing
- Optional means optional: older pages have no dates and no per-chapter PDF
- Sort chapter numbers as integers, never as text
- Skip underscore files, `_print` variants and the `pdfs/` and `solutions/` folders
- Validate before building so content problems fail early

Learning Website: Framework & Architecture

Chapter 4 · The Shared Design System

If the site becomes several sites, they still have to look like members of one family. A visitor who moves from the languages site to the programming site should not feel they have changed websites, and you should not have to change a colour in fifty places. This chapter looks at what the fragments really contain, chooses a design approach that does not require rewriting thousands of files, and sets up the pieces: tokens, shared components, and a safe order for loading them.

MEASURED, NOT ASSUMED

The numbers in this chapter come from a script run over your real fragments (Exercise 1). They show a simpler situation than you might fear: a lot of repetition, but only a few real differences.

What the Fragments Contain Today

Every fragment carries its own `<style>` block, scoped under its own wrapper class. That scoping rule is deliberate, because it stops one fragment's styles leaking into another. The price is repetition. Measuring a few courses gives these results:

FOLDER	FILES	DISTINCT COLOURS	CSS AS SHARE OF FILE BYTES	ACCENT COLOUR
Hungarian Basic 3	12	14	13%	#5aa469
German Basic 1	12	14	17%	#e0a82e
Japanese Basic 1	12	14	17%	#d64550
Apache In Depth	10	19	30%	#f0938a
Sidebar: programming	30	16	12%	#6ee7b7
Django rebuild	12	21	32%	#44b78b

Four findings stand out:

1. **The three language courses share the same palette.** Surfaces `#1a1d27`, borders `#2a2d3a`, text `#e2e4ec`, muted `#a0a4b8` and the purple speaker colour `#7c5cbf` are identical. Only

the accent changes.

2. **There are two neutral families.** The technical courses use a different set (`#161b22` surfaces, `#30363d` borders, `#c9d1d9` text, `#8b949e` muted), with their own accent per course.
3. **The same CSS is copied into every chapter.** A colour used nine times in one chapter appears 108 times in a 12-chapter course.
4. **CSS is 12% to 32% of each file.** A visitor downloads the same styling again on every chapter.

Three Ways to Share Design

APPROACH	HOW IT WORKS	VERDICT
A. Leave it as it is	Each fragment keeps its own CSS. Sites only share the surrounding layout.	Nothing to migrate, but the repetition stays and a brand change means editing every file.
B. One shared stylesheet, all at once	Strip the CSS from every fragment and rely on shared rules.	Cleanest result, but a very large rewrite that touches every file and risks broken pages.
C. Tokens and shared components, gradually	Define colours and spacing as variables. Provide shared component styles as defaults. Fragments keep working and are migrated when convenient.	Recommended: safe at every step.

This course follows approach C. New content uses the shared layer, and old content keeps its own rules until it is rewritten or migrated by script.

Design Tokens

A **token** is a named value: `--surface` instead of `#1a1d27`. In CSS the natural way to hold tokens is custom properties (variables). They cascade, so you can set a different value for a part of the page, which is exactly what per-language accents need.

```
/* tokens.css: the only place a colour is decided */
[data-family="language"] {
  --surface: #1a1d27;  --border: #2a2d3a;
```

```

    --text:    #e2e4ec;    --muted:  #a0a4b8;
  }
  [data-family="technical"] {
    --surface: #161b22;    --border: #30363d;
    --text:    #c9d1d9;    --muted:  #8b949e;
  }
  .hu-lesson { --accent: #5aa469; } /* Hungarian */
  .de-lesson { --accent: #e0a82e; } /* German */

  /* a component using them */
  .tip-box {
    background: var(--surface);
    border: 1px solid var(--border);
    border-left: 4px solid var(--accent);
    color: var(--text);
  }

```

Two neutral families can stay as they are for now. Unifying them into one is a design decision for later; tokens make it a small change when you choose to do it, because every use reads the variable.

A safe way to introduce the variable

Existing fragments hard-code their accent. A script can replace each occurrence *inside the style block only* with `var(--accent, #5aa469)`. The second argument is a fallback: if the variable is not defined, the browser uses the old colour, so the page looks exactly as before. That makes each file's migration independently safe (Exercise 2).

WHY ONE VARIABLE PAYS OFF

The Hungarian, German and Japanese chapters differ from each other by a single accent colour. Once that colour is a variable, a new language is one line of CSS instead of a whole copied stylesheet.

Shared Components

The fragments already use the same class names for the same ideas, which is what makes a shared layer possible:

GROUP	CLASSES IN COMMON USE
Page header	<code>.lesson-head</code> , <code>.course-name</code> , <code>.chapter-sub</code>
Callouts	<code>.tip-box</code> , <code>.warn-box</code> , <code>.finding-box</code>
Code	<code>.code-block</code> , <code>.code-block-wrap</code> , <code>.copy-btn</code>
Tables	<code>.compare-table</code> , <code>.vocab-table</code>
Exercises	<code>.challenge-block</code> , <code>.ch-num</code> , <code>.ch-body</code> , <code>.ch-link</code> , <code>.quick-ref</code>
Language lessons	<code>.dialog-grid</code> , <code>.dialog-card</code> , <code>.vocab-table</code> , <code>.tip-box</code>

A shared `components.css` styles these once, using the tokens. Fragments that still carry their own rules for the same classes keep winning, as the next section explains.

Who Wins When Rules Collide

The existing scoping rule has a useful side effect. A fragment's rule such as `.hu-lesson .tip-box` is more specific (two classes) than a shared `.tip-box` (one class), so the fragment's own rule wins. If you write the shared rule with the same specificity, for example `[data-site=languages] .tip-box`, the two tie, and the later one in the document wins. The layout's stylesheet is in `<head>` and the fragment's `<style>` is inside the body, so the fragment still wins. Both effects protect old pages during a migration.

DO NOT REACH FOR !IMPORTANT

When two rules clash it is tempting to add `!important`. It spreads, because the next rule that needs to override it needs one too. Fix a clash by changing specificity or by deleting the old rule.

Load order is therefore: **tokens, then base, then components, then the fragment's own style**. Exercise 3 writes the token file and checks the specificity rules with a small script.

Per-Area Accents

Colour already tells visitors where they are. These accents are in use today, with the contrast each one gives against the language surface `#1a1d27`:

AREA	ACCENT	CONTRAST
Hungarian	#5aa469	5.56
German	#e0a82e	7.85
Japanese	#d64550	3.86
French	#4f8ef7	5.24
Links pages	#38bdf8	7.85
Sidebar lessons	#fbbf24	10.07
Cheat sheets	#6ee7b7	11.03

WCAG level AA asks for a contrast ratio of at least 4.5 for normal-size text. Every accent passes except the Japanese red, at 3.86. It is fine for borders, buttons and large headings, but weak for small coloured text. The right place to fix this is the token file, for instance with a separate lighter `--accent-text` for that language, not in each chapter (Exercise 3, part D).

Fonts, Printing and PDFs

- **Fonts are tokens too.** The language lessons use a serif body font (Georgia) and the technical ones use the system sans-serif. Put each in a variable such as `--font-body`.
- **Web fonts and origins.** If a font is loaded from another subdomain, the browser requires CORS headers (Learning Website: Framework & Architecture 1). The simplest options are system fonts, or self-hosting the same font files on every site.
- **The PDF builders render these same fragments.** Any change to the shared layer changes the PDFs too, and the copy buttons are hidden for print by an `@media print` rule. After a design change, rebuild one chapter PDF and look at it before trusting the change.

Where the Design System Lives

The tokens, base and components are one small unit, shared by every site and versioned together. In a framework this becomes a package or a static folder:

```
design-system/  
├─ tokens.css      # colours, fonts, spacing  
├─ base.css        # page, typography, links  
├─ components.css  # tip-box, code-block, tables, exercises  
└─ copy-code.js    # the copyCodeBlock() function
```

Each site includes the same four files. The Django course (Learning Website with Django 5) serves them as static files, and the Next.js course (Learning Website with Next.js 5) imports them from a shared workspace package. A change in one place reaches every site on its next build, so treat the design system like code: review it and test it before release.

Hands-On Exercises

EXERCISE 1

Write `palette.py`, which reports for each folder of fragments the number of files, the number of distinct colours in the style blocks, the share of the file bytes that is CSS, and the most common colours. Run it on several courses from different areas and write down what is shared and what differs.

 [View solution](#)

EXERCISE 2

Write a script that replaces a fragment's accent colour, inside its `<style>` blocks only, with `var(--accent, <the same colour>)`. Have it report the number of replacements and check that the braces are still balanced and the markup outside the style block is unchanged. Run it without modifying the original file.

 [View solution](#)

EXERCISE 3

Write a `tokens.css` for the two neutral families and the language accents. Then use a small script to compare the specificity of a shared `.tip-box` rule and a fragment's `.hu-lesson .tip-box` rule, state the load order, and check the contrast of every accent against the language surface.

CHAPTER 4 QUICK REFERENCE

- Every fragment carries its own scoped CSS: **12% to 32% of file size**, copied into every chapter
- The language courses share one palette and differ only in the **accent colour**; the technical courses use a second neutral family
- Approach: **tokens plus shared components, migrated gradually** (not a one-off rewrite)
- Tokens are CSS custom properties; set them per family or per wrapper so accents can differ by area
- Migrate safely with `var(--accent, #oldcolour)`: the fallback keeps old pages unchanged
- A fragment's `.wrapper .class` rule beats a shared `.class` rule; on a tie the later one (the fragment's style) wins
- Load order: **tokens, base, components, then the fragment's own style**
- Avoid `!important`; fix clashes with specificity or by deleting the old rule
- Check contrast: WCAG AA needs 4.5 for normal text, and the Japanese red `#d64550` scores 3.86
- PDF builders use the same HTML, so rebuild a PDF to check after a design change

Learning Website: Framework & Architecture

Chapter 5 · Navigation Across Sites

On a single site, navigation is easy to take for granted: every page is one link away from every other. Once the content is spread over several sites, a visitor needs three things to keep their bearings: a menu for the part of the site they are in, a way to see and reach the other sites, and links inside lessons that still work when they point across a boundary. This chapter starts from how your current navigation is really built, then designs the three layers for the multi-site version.

How Navigation Works Today

The live Astro site has four navigation pieces, and two of them are derived automatically while one is typed by hand:

PIECE	SOURCE	MAINTAINED BY
Top menu (dropdown groups)	<code>NAV_CATEGORIES</code> and <code>STANDALONE_SLUGS</code> in <code>nav-groups.ts</code>	By hand
Home page subject grid	Every top-level folder of the content tree	Automatic
Sidebar tree	The folder tree of the current subject, drilled to the active page	Automatic
Breadcrumb and page titles	The folder path, with the chapter title read from the banner	Automatic

The site tree is built by walking the `content/` folder, and the site is configured with `trailingSlash: 'always'`, so every directory URL ends in `/`. Both facts matter for what follows.

THE HAND-WRITTEN MENU HAS DRIFTED

Exercise 1 compares the menu list with the real folders. Of 49 top-level folders, **13 have no menu entry**: `blockchain-and-web3`, `classic-literature`, `cookery`, `culture`, `football`, `ios-development`, `music`, `philosophy`, `politics`, `sci-fi-project`, `science`, `science-fiction` and `study-methodologies`. They are still reachable from the home page grid and the course index, so

nothing is lost, but the menu does not show them. The file says itself that a folder left out simply does not appear. A list that is maintained by hand and has no check will drift.

Three Layers for Several Sites

With separate sites, one menu cannot do every job. Split navigation into three layers, each with a different scope and a different source:

LAYER	SHOWS	SAME ON EVERY SITE?	SOURCE
1. Global bar	The family of sites, with the current one highlighted	Yes	The site map (one file)
2. Site menu	The subjects within this site	No: one per site	Derived from the site's own folders
3. Page navigation	The sidebar tree, breadcrumb and chapter previous/next	No: depends on the page	Derived from the folder tree

Layer 1: the global bar

A thin bar at the top of every page lists the sites: Languages, Web Development, Programming, Systems and so on, with the current one marked. It is the same markup everywhere, generated from the site map, so adding a site means adding one entry. Eight short names fit comfortably on a desktop. On a narrow screen, collapse them into a `<details>` dropdown, the same no-JavaScript technique the current menu already uses.

The first item links back to the root domain, which Chapter 1 suggested becoming a portfolio and landing page. Note that the current Astro configuration uses `https://www.osztromok.com` as the site address. Decide early whether the landing page lives on `www`, on the bare domain, or on both with a redirect, because every global-bar link and every canonical URL depends on it.

Layer 2: the site menu

Each site's menu lists only its own subject folders, taken from the map and the folder tree. Because it is generated, it cannot drift the way the current list did. A site gets a *Sidebar* entry only if some `sidebar/<subject>/` folder is routed to it (Learning Website: Framework & Architecture 2). Exercise 3 builds this and shows the result for two sites.

```
languages.osztromok.com
```

```
|— Global bar:  Languages* | Web Development | Programming | Systems | AI | ...  
|— Site menu:  France Germany Hungary Japan Culture  
└ Page:       sidebar tree / breadcrumb / previous - next
```

```
systems.osztromok.com
```

```
|— Global bar:  Languages | ... | Systems* | ...  
|— Site menu:  Linux Operating Systems Networking ... Sidebar  
└ Page:       sidebar tree / breadcrumb / previous - next
```

Layer 3: page navigation

The sidebar tree and the breadcrumb already work from the folder structure, so they carry over unchanged, with one addition: the first breadcrumb item becomes the site, linking to its home, optionally preceded by the family root. Previous and next links between chapters use the chapter number, sorted as a number (Learning Website: Framework & Architecture 3).

Links Inside Content

The hardest part is not the menus. It is the thousands of links written inside lessons, such as `/linux/shell-and-scripting/vim/`, that assume one site. After a split a link may stay on the same site or cross to another one. Two rules keep this manageable:

1. **Authors keep writing root-relative paths.** Nobody has to remember which site a topic lives on.
2. **The build resolves them.** A resolver looks up the target's site using the same function as Chapter 2. A same-site link stays relative. A cross-site link becomes an absolute URL on the other site.

```
# written in a Hungarian chapter, built on languages.osztromok.com  
/japan/hiragana-1/           -> /japan/hiragana-1/   (same site, stays re  
/linux/shell-and-scripting/vim/ -> https://systems.osztromok.com/linux/shell  
#top                         -> #top    (anchors are never touched)  
https://example.org/page/    -> https://example.org/page/  (external)
```

This has three benefits. A folder that later moves to another site needs one line changed in the map, not thousands of edits. Local development can use another base address (such as

`*.localhost`) from the same table. And a link to a path that belongs to no site fails loudly during the build instead of shipping broken. Exercise 2 writes this resolver and runs it on nine kinds of link.

SOME URLS ARE NOT FOLDER NAMES

The kanji pages are served from `/resources/japanese/kanji/`, which is not a folder under `content/`. The resolver needs an explicit entry for that prefix. Collect these exceptions deliberately: Learning Website: Framework & Architecture 11 audits every link and will find the rest.

Opening Cross-Site Links

- **Keep the visitor in context.** A cross-site link in a lesson should usually open in the same tab, so the browser's Back button works. Reserve a new tab for genuinely external sites.
- **Mark them.** A small marker or label such as “Systems site” tells visitors a link leaves the current site, so a change of menu is no surprise.
- **Mind the origin.** A normal link is fine across origins. What does not work across subdomains is shared `localStorage` and, without CORS, scripted requests (Learning Website: Framework & Architecture 1).

Keeping Navigation Honest

Navigation fails quietly. A missing menu entry or a link to the wrong site does not cause an error; a visitor just cannot find something. Add checks to the build:

- Every folder is assigned to exactly one site (Chapter 2's checker).
- Every site menu entry points at a folder that exists.
- Every internal link in every page resolves to a known site and, after the build, to a file that exists.

SMALL THINGS WORTH FIXING ON THE WAY

The label for `data-science-and-ml` comes out as “Data Science And ML” and `cloud-and-devops` as “Cloud And Devops”, because the live label helper has no `ml` in its acronym list and does not lower-case small words. Fix both once in the shared helper rather than renaming folders.

Hands-On Exercises

EXERCISE 1

Write `nav_gap.py`, which reads the slugs in `nav-groups.ts` and compares them with the real top-level folders of `content/`. Report the folders with no menu entry and any menu entry that points at nothing, and explain why a list like this drifts.

 [View solution](#)

EXERCISE 2

Write `link_for(current_site, href)`, which keeps same-site links relative, turns cross-site links into absolute URLs, and leaves external links, anchors and relative paths alone. Keep the path, query, fragment and trailing slash, make the base address configurable per environment, and test it on at least eight links.

 [View solution](#)

EXERCISE 3

Write `nav_for(site, content)`, which returns a site's menu from the site map and the real folders with readable labels. Add a Sidebar entry only for sites that receive sidebar pages, run it for two sites, and note any label that comes out badly.

 [View solution](#)

CHAPTER 5 QUICK REFERENCE

- Today: the top menu is **hand-written**; the home grid, sidebar tree and breadcrumbs are **derived** from the folder tree
- The hand-written menu has drifted: **13 of 49 folders** are missing from it (still reachable from the home grid and the course index)
- Three layers: **global bar** (all sites), **site menu** (this site's subjects), **page navigation** (tree, breadcrumb, previous/next)
- Generate menus from the site map and folders so they cannot drift

- Authors write root-relative links; the **build resolves them** with `site_for()`
- Same-site links stay relative; cross-site links become absolute URLs; anchors, external and relative links are untouched
- Keep the trailing slash (`trailingSlash: 'always'`) when rebuilding links
- Some URLs are not folder names (the kanji resources): list them explicitly
- Decide the landing page address (`www` , bare domain or both) early
- Add build checks: every folder assigned, every menu entry valid, every internal link resolvable

Learning Website: Framework & Architecture

Chapter 6 · The Content Pipeline

Chapter 3 described the files. This chapter follows one file on its journey to a finished page: how the build reads it, cleans it, finds its title, fixes its links, wraps it in a layout and publishes the assets next to it. The good news is that this pipeline already exists and works. The live Astro site implements it in a few small files. The job here is to describe it clearly enough that the Django and Next.js versions can behave identically, because the content is shared and must look the same whichever framework builds it.

BASED ON THE REAL CODE

Everything in the stage table below is taken from the live site's `transform-html.ts`, `content-fs.ts`, `site-tree.ts`, the page and layout files and the asset-copy script. The exercises re-implement the stages in Python and run them on real files.

The Stages

#	STAGE	WHAT IT DOES
1	Walk and classify	Visit every file under <code>content/</code> and label it: chapter HTML, PDF, solution <code>.txt</code> , or other
2	Detect the shape	A file with a <code><body></code> tag is a complete document; anything else is a fragment
3	Extract the fragment	Complete document: take the body contents and keep the <code><style></code> blocks from the head. Fragment: remove the leading banner comment
4	Rewrite solution links	Point every <code>.txt</code> link at <code><course>/solutions/<file></code>
5	Find the title	Banner <code>Chapter:</code> , else the <code><title></code> tag, else the first <code><h1></code> or <code><h2></code> , else the file name made readable
6	Decide the heading	Add a synthetic <code><h1></code> only if the fragment has none of its own

#	STAGE	WHAT IT DOES
7	Wrap in the layout	Inject the fragment into the page layout, with breadcrumb and previous/next links from the sorted sibling chapters
8	Activate scripts	Re-create any <code><script></code> inside the injected content so that it runs; provide the code-copy function once, globally
9	Publish assets	Mirror every <code>pdfs/</code> and <code>solutions/</code> folder into the output at the same relative path
10	Validate	Check the assumptions against every real file before the templates rely on them

Shape and Extraction

Nearly all files are fragments, and the kanji pages are complete documents (Learning Website: Framework & Architecture 3). The pipeline must handle both, so it tests for a `<body>` tag and extracts accordingly. For a complete document it keeps the head's `<style>` blocks and discards the rest of the head, so the styles are not lost. A document with an opening but no closing body tag is marked malformed so the build can complain.

Titles and Headings

The page title comes from the banner's `Chapter:` line first. This is deliberate: numbered-course chapters have no real heading of their own (they show a styled course name and a breadcrumb line instead), so the banner is the only place that holds the snappy chapter title. When there is no such line, the pipeline falls back to the document's own `<title>`, then to the first heading, then to the filename. The fallback order matters. A kanji page's `<title>` ends in “| osztromok.com”, which is right for the browser tab but wrong for a page heading.

A synthetic `<h1>` is added only when the fragment has no `<h1>`, so nothing is ever titled twice.

Solution Links

Old chapters link to their solution files in many stale ways: absolute paths from before the folders were reorganised, relative variants, a reference to an old backup location, or a bare filename. Instead of listing every variant, the pipeline matches any `.txt` link and rebuilds it

from its file name: `<course path>/solutions/<file name>`. That is robust, with one condition: the file names inside one course's `solutions/` folder must be unique and must exist there.

WHAT A REAL AUDIT FOUND

The pipeline rewrites links but never checks that the target exists. Exercise 2 checks it: of **268** courses with a `solutions/` folder and **10,975** solution links, **seven courses have a problem**. JavaScript Fundamentals has 60 links but only 3 solution files. TypeScript Fundamentals and the three Practical Scripting projects have solution files that no chapter links to. OWASP Top 10 and Excel Fundamentals each have a few links with no file. They are listed in `website_checks.md`; none has been fixed.

Scripts, and the Copy Button

Fragments are injected into the page as HTML text. Browsers do not run `<script>` tags that arrive that way, so the pipeline has two measures, both in the layout:

1. **The code-copy function exists once, globally.** Each code block carries `onclick="copyCodeBlock(this)"`, and the function lives in a normal script in the layout. New chapters must not embed their own copy.
2. **Other scripts are re-created.** When the page loads, every `<script>` inside the content area is replaced by a freshly created one with the same attributes and text, which does run. That is what makes the interactive My Tools pages work.

```
<!-- the markup contract every code block must follow -->
<div class="code-block-wrap">
  <button class="copy-btn" onclick="copyCodeBlock(this)">Copy</button>
  <div class="code-block"> ...code... </div>
</div>

// the global function reads the element right after the button
function copyCodeBlock(btn) {
  const code = btn.nextElementSibling.innerText;
  navigator.clipboard.writeText(code).then(() => { /* show "Copied!" */ });
}
```

Two details are easy to miss. The function relies on the code block being the element *immediately after* the button, so the markup order is part of the contract, and any new framework must keep it. And `navigator.clipboard` only works in a **secure context**, meaning

HTTPS or localhost. Test copying on the real HTTPS subdomains, not just on a development address.

SCRIPTS THAT DEPEND ON EACH OTHER

Re-created scripts run in document order, but only after the page has loaded. A script that expects another to have run first, or that runs while the page is still parsing, can behave differently than in a standalone file. Test every interactive tool on the built site, and make each tool's script self-contained.

Ordering, Neighbours and Assets

The live site sorts chapters with a **natural sort**, so chapter 2 comes before chapter 10 (the trap from Learning Website: Framework & Architecture 3). Previous and next links are the neighbours in that sorted list. The new builds must use the same ordering, or the chapter links will disagree between sites.

PDFs and solution files are not part of the page. A small step before the build mirrors every `pdfs/` and `solutions/` folder into the published output at the *same relative path*, which is why the rewritten links work. It copies a file only when it is missing or newer, so the second run does almost nothing (Exercise 3).

PDFs are build outputs

Each PDF is generated from the chapter HTML by a builder script, and the repository ignores `*.pdf`. That keeps git small, but it has a consequence: the asset-copy step reads PDFs from the working folder, not from git. On a new machine or a build server they will be missing until you regenerate them or sync them from where they are kept. Decide which, document it, and add a check that every course which links to a PDF actually has it.

Validate Before the Templates Rely on It

The live site has a separate script that runs the transform over every real file and reports shapes, malformed documents and unresolved solution links before any template uses them. Keep that habit. A pipeline check that fails the build is cheap, and the alternative is a visitor finding the broken link for you. In the multi-site version run the same validation per site, and once more across all sites together.

Hands-On Exercises

EXERCISE 1

Port the transform stages (detect shape, extract fragment, rewrite solution links, find the title, decide the synthetic heading) to Python without any framework. Run them on a course chapter, a chapter with solution links, a complete kanji page and a cheat sheet, and explain each result.

 [View solution](#)

EXERCISE 2

Write an audit that checks every `.txt` link in a course against that course's `solutions/` folder, and reports links with no file and files no chapter links to. Run it on four courses, then on every course with a `solutions/` folder, and summarise the problems.

 [View solution](#)

EXERCISE 3

Write an idempotent asset-copy step that mirrors every `pdfs/` and `solutions/` folder to an output folder at the same relative path, copying only when a file is missing or newer. Run it twice and then after touching one file, and explain what it shows. Say what you would do about PDFs on a machine that has no copies.

 [View solution](#)

CHAPTER 6 QUICK REFERENCE

- Stages: **walk, shape, extract, solution links, title, heading, layout, scripts, assets, validate**
- A file with `<body>` is a complete document; everything else is a fragment
- Title order: banner `Chapter:`, then `<title>`, then first h1/h2, then the file name
- Add a synthetic `<h1>` only if the fragment has none
- Any `.txt` link is rewritten to `<course>/solutions/<file name>`; the file must exist there
- The audit found **7 of 268** courses with solution problems (10,975 links checked)

- `copyCodeBlock` lives once in the layout and needs the code block right after the button; clipboard needs HTTPS or localhost
- Scripts inside content are re-created on page load so that they run
- Sort chapters naturally (2 before 10); previous/next are the neighbours in that order
- Mirror `pdfs/` and `solutions/` at the same relative path, copying only if missing or newer
- PDFs are build outputs, ignored by git: regenerate or sync them, and check every linked PDF exists

Learning Website: Framework & Architecture

Chapter 7 · Search, Sitemaps & SEO

A site can be well organised and still be hard to find. Visitors need to search it, and search engines need to know what is on it and which address is the real one. When the content moves to several sites, three things become more important: each site must describe itself, the old addresses must lead to the new ones without losing ground, and no page should be listed under two addresses. This chapter looks at what the live site has today, then designs search, sitemaps, canonical addresses and redirects for the family of sites.

WHERE THE FACTS COME FROM

Statements about the live site come from its source and built output. Statements about Google and Pagefind come from their documentation, read while writing this chapter. Search engines change, so treat the behaviour described as guidance and check the current documentation before launch.

What Exists Today

FEATURE	ON THE LIVE SITE
Search	A course-name and tag filter on the Course Index page (client-side). No full-text search of lesson content
Sitemap	None: no <code>sitemap.xml</code> in the built output
robots.txt	None
Canonical link, meta description, social tags	None: the layout's head holds only the character set, viewport and title
Redirects	None: the <code>.htaccess</code> only turns off directory listings
Size	About 4,900 published pages in the content tree (5,014 directory index pages in the build)

That is a clean slate rather than a problem: the new sites can include all of this from the start, and none of it has to be migrated.

Search

Search comes in three levels, and each one builds on the one before:

LEVEL	WHAT IT SEARCHES	HOW
1. Course finder	Course names and tags in one site	The existing client-side filter, built per site from that site's own courses
2. Full-text search per site	The text of every lesson on one site	A static index generated after the build and searched in the browser; no server needed
3. Search across all sites	Every site at once	Merge the per-site indexes in one search page, usually on the root domain

A static site search tool such as **Pagefind** fits the second level: it indexes the built HTML files after the build and ships the index as static files, so the sites need no search server. For the third level its documentation describes a multisite feature: a search page loads other sites' indexes with `mergeIndex`, given each index's `bundlePath`, and an `indexWeight` option ranks one index above another. Two caveats from the documentation matter here. Indexes on separate domains must be served with **CORS headers**, which is the origin issue from Learning Website: Framework & Architecture 1 again, and a merged index uses the main instance's language support, which matters for a site with Hungarian, Japanese, German and French text.

INDEX THE CONTENT, NOT THE CHROME

Menus, footers and the global bar appear on every page. If they are indexed, every search matches every page. Whatever tool you use, mark the lesson body (the content area) as the part to index, and leave the navigation out. Also decide what you do *not* want searchable, such as the print variants of cheat sheets, which would appear as duplicates.

Sitemaps

A sitemap is a list of the URLs you want search engines to know about, optionally with the date each page was last changed. The rules that matter, from Google's documentation:

- A single sitemap is limited to **50,000 URLs or 50 MB** uncompressed. Larger sites use several files and a sitemap index.

- A sitemap's scope depends on where it lives: unless you submit it through Search Console, it affects only the directory it sits in and below. Put each one at the **site root**.
- Sitemaps can span several hosts, or a site's sitemap can be hosted centrally with each site's `robots.txt` pointing to it. A simple **one sitemap per site** avoids needing either.

Exercise 1 builds the sitemaps from the real content tree. The result for the proposed sites:

SITE	PAGES	WITH A BANNER DATE	SITEMAP FILES
programming	1,540	633	1
humanities	792	695	1
systems	739	205	1
languages	594	250	1
webdevelopment	583	53	1
lifeskills	284	227	1
ai	216	27	1
creative	182	20	1

Every site is far below the limit. The page counts add to slightly more than the 4,925 distinct pages, because each of the six sites that receives sidebar pages has its own `/sidebar/` index page.

ONLY 42% OF PAGES HAVE A DATE

Just 2,110 of the 4,925 pages carry a `Date Updated` in their banner (the rule that added it applies to new and rewritten files only). The sitemap script gives those pages a `lastmod` and leaves the rest without one. Do not fill in a date you cannot trust, such as the file's modified time after a bulk rewrite: a wrong date is worse than no date.

robots.txt, Canonical Links and Descriptions

robots.txt belongs to a host

Google's documentation states that a `robots.txt` file applies only to the host, protocol and port that serve it, and that it must sit at the root. So `languages.osztromok.com` needs its own, and it is also the natural place to name that site's sitemap:

```
# https://languages.osztromok.com/robots.txt
User-agent: *
Allow: /

Sitemap: https://languages.osztromok.com/sitemap.xml
```

A new site that is not ready to be public should disallow crawling, and the day it launches that line must change. It is the kind of thing that is forgotten, so it goes on the launch checklist (Learning Website: Framework & Architecture 12).

One address per page

A **canonical link** tells search engines which address is the real one when a page can be reached in several ways (with and without `www`, over HTTP and HTTPS, with and without a trailing slash, or from an old and a new site). Google recommends an *absolute* URL, and recommends that a page include a canonical link to itself. Treat it as a strong signal, not a command. To keep the signals consistent, make three things agree on the same string: the canonical link, the sitemap entry and the internal links. That includes the decision about `www` from Learning Website: Framework & Architecture 5, and the trailing slash that the site's configuration already requires.

```
<title>Buying Clothes: Sizes, Fit & Returns</title>
<meta name="description" content="Hungarian Basic Conversation 3, chapter 1: buying clothes" />
<link rel="canonical" href="https://languages.osztromok.com/hungary/hungarian-basic-3/hungarian-basics-3-buying-clothes/" />
```

The meta description does not need new writing: the banner's `Topic:` line is already a written summary of each chapter, and can be shortened for the tag.

Moving Without Losing Ground

The part of SEO that can really hurt is the move itself. Google's site-move guidance gives a few rules that fit this migration well:

- **Use permanent redirects**, a 301 or 308, from each old URL to its new URL.
- **Keep them a long time:** generally at least a year, and indefinitely is better for visitors who have old links.
- **A section-by-section move is acceptable** for large sites, which matches moving languages first and the rest later.
- **Expect weeks, not days:** a small or medium site can take a few weeks for most pages to move.
- The **Change of Address** tool in Search Console applies to domain or subdomain changes (for example one subdomain to another), and is not needed for path-only changes. Whether it applies when a section moves from a path to a new subdomain is a question to check in Search Console at the time.

Redirect each old URL to its exact new URL in a **single hop**, not to the new site's home page, and not through a chain. Exercise 3 builds the rule for the languages area from the site map and tests it against every old page URL.

DO NOT COPY CONTENT TO TWO SITES

A page that exists on two sites at once competes with itself, and a canonical link then has to sort it out. The rule from Chapter 2 (one home per page) is also the best SEO rule: link across sites, never copy.

What Splitting Does to Search Visibility

Honest expectations help. Google has said it can handle subdomains and paths, but there is no guarantee that a moved section ranks the same afterwards, and for a while the old and new addresses are both known. Reduce the risk instead of trusting it: redirect every old URL, keep internal links consistent, submit each new site's sitemap, watch the Search Console reports for crawl and coverage errors, and move one area at a time so that a problem is found while it is small.

Hands-On Exercises

EXERCISE 1

Write a script that walks `content/`, decides which files are published pages (skipping print variants, underscore files, `pdfs/`, `solutions/` and the kanji archive), groups the URLs by site, and writes a sitemap per site with `lastmod` where the banner has a date. Report pages per site, date coverage and the number of sitemap files needed.

 [View solution](#)

EXERCISE 2

Write helpers that produce a page's title, meta description and canonical link, and a site's `robots.txt` with its `Sitemap:` line. Make the canonical the same string as the sitemap's `<loc>`, and reject paths that would break that match.

 [View solution](#)

EXERCISE 3

Build the Apache `RedirectMatch` rule for the languages area from the site map, and test it against every old page URL in that area: count the matches, confirm each lands on the same path on the new host, and confirm that another area is left alone. Say what this test cannot prove.

 [View solution](#)

CHAPTER 7 QUICK REFERENCE

- Today: a course-name filter only; **no** full-text search, sitemap, robots.txt, canonical link or meta description
- Search levels: course finder, full-text per site (static index, built after the site), merged search across sites (needs CORS)
- Index the lesson body, not menus or the global bar
- Sitemap limits: **50,000 URLs or 50 MB** each; one sitemap per site, at the site root
- Only 42% of pages have a banner date: give `lastmod` to those and leave the rest out
- `robots.txt` applies only to its own host, protocol and port: **one per site**
- Canonical links: absolute, self-referential, and the same string as the sitemap and the internal links
- Moves: 301 or 308, one hop, exact URL to exact URL, kept at least a year; section-by-section is fine
- Never copy a page to two sites: link across sites instead

- Block crawling on a site that is not ready, and put reversing it on the launch checklist

Learning Website: Framework & Architecture

Chapter 8 · Hosting & Deployment

Design and content decide what the sites are. Hosting decides whether they stay up, whether a bad release can be undone, and whether anything private leaks out. The good news is that the multi-site version runs on the same kind of machine you already have: one Debian server running Apache, with a certificate from Certbot. What changes is the arrangement: one virtual host per site, a safer way to publish a build, and a backup plan that covers the things git does not.

A REAL FINDING FROM THIS CHAPTER'S EXERCISES

While testing a secrets scanner (Exercise 3) on the real folders, a **database password** turned up in plain text in **five content files**, in the matching built pages, and in one PHP file in the build. Published pages are public, so treat that password as exposed and change it. The chapter's values are never shown. See the section on secrets below, and `website_checks.md`.

How It Works Today

As you described it, the content reaches the server in two steps: your PC syncs to a folder under your home on `debserver`, and that folder is synced to the web root, so you always hold several copies. The server runs Apache with one virtual host, taken from the course Setting Up a Web Server on Debian:

```
# /etc/apache2/sites-available/000-default.conf (as described in the course)
<VirtualHost *:80>
    ServerName      osztromok.com
    ServerAlias      www.osztromok.com
    DocumentRoot    /var/www/html
    <Directory /var/www/html>
        Options      -Indexes +FollowSymLinks
        AllowOverride All          # lets .htaccess work
        Require      all granted
    </Directory>
</VirtualHost>
```

Certificates come from Certbot, which proves ownership of a name over HTTP. That is a good foundation. Two weaknesses are worth fixing while you are changing it anyway: a sync that overwrites live files can show visitors a half-copied site and has no way back, and everything sits in one document root.

The Target Layout

Each site gets its own folder and its own virtual host. Inside each folder, the live site is not a normal directory but a link called `current` that points at one release folder:

```
/var/www/
├── html/                                # the old single site, kept while migrating
├── languages/
│   ├── current -> releases/20261006-140545-328689964
│   └── releases/
│       ├── 20261006-140544-200467284/
│       └── 20261006-140545-328689964/
├── webdevelopment/
│   ├── current -> releases/...
│   └── releases/
└── ...                                # one folder per site
```

DNS and Virtual Hosts

Each site needs a DNS record pointing at the server (an `A` record per subdomain, or a wildcard record; Chapter 1 shows the entries). Apache then chooses the site from the host name in the request, which is called **name-based virtual hosting**: several `<VirtualHost *:80>` blocks share one address, each with its own `ServerName`. A request for a name that matches none of them goes to the *first* virtual host Apache loads for that address and port, so keep one deliberate default, and let the existing `000-default.conf` play that role, so a mistyped record never shows the wrong site.

Because the virtual hosts differ only by name, generate them from the site map (Exercise 1) rather than copying one by hand:

```
# /etc/apache2/sites-available/languages.osztromok.com.conf
<VirtualHost *:80>
```

```

ServerAdmin    webmaster@osztromok.com
ServerName     languages.osztromok.com
DocumentRoot  /var/www/languages/current

<Directory /var/www/languages/current>
    Options          -Indexes +FollowSymLinks
    AllowOverride    None
    Require          all granted
</Directory>

ErrorLog  ${APACHE_LOG_DIR}/languages-error.log
CustomLog ${APACHE_LOG_DIR}/languages-access.log combined
</VirtualHost>

# always test before reloading
sudo a2ensite languages.osztromok.com.conf
sudo apache2ctl configtest
sudo systemctl reload apache2

```

`AllowOverride None` is deliberate. The new builds need no `.htaccess`; every rule lives in the virtual host, which is faster and cannot be changed by a stray file in the content.

Certificates

There are two sensible ways to cover the subdomains:

OPTION	HOW	TRADE-OFF
One certificate listing every name	Certbot with a <code>-d</code> for each name, using the same HTTP method you already use	Simple, with no DNS automation. Adding a site means re-running the command with the new name.
A wildcard certificate	<code>*.osztromok.com</code> , issued with the DNS challenge	New sites need no certificate change, but you must automate DNS record changes, and a wildcard covers one label only.

For eight sites the first option is easier: one certificate with ten names (the root, `www` and the eight sites). Check that renewal runs on its own (`systemctl list-timers` should list the Certbot timer) and that it still works after you add names.

Atomic Deploys and Rollback

The deploy script copies a finished build into a new timestamped release folder, then switches `current` to it with one atomic rename. A visitor sees the old site or the new site, never a mixture, and the previous release stays on disk, so going back is one command:

```
# the heart of deploy.sh
rsync -a --delete "$BUILD"/ "$REL"/          # copy into the new release
ln -sf "$REL" "$BASE/current.new"          # make the new link beside the old
mv -Tf "$BASE/current.new" "$BASE/current"  # one atomic rename switches it
```

TWO BUGS THE TEST CAUGHT

The first version of the script pruned old releases by modified time, and `rsync -a` copies folder times from the build, so it deleted the wrong release. Worse, two deploys in the same second shared a release name, so a *refused* deploy emptied and then deleted the live site. The fixes: name releases with nanoseconds, create them with `mkdir` without `-p` so an existing name fails, prune by name, and never delete the live release. The test now deploys three times, rolls back, deploys again, and tries an empty build. Test any script that deletes things by trying to make it hurt you.

Backups

You already keep several copies. The new layout changes what each copy has to contain, because two of the most important things are not in git:

WHAT	WHERE IT LIVES	IN GIT?
Content, rules, scripts	<code>website-content</code> and <code>learning-website</code>	Yes (private repos)
PDFs	The <code>pdfs/</code> folders	No (ignored; build outputs)
The built sites	<code>/var/www/<site>/releases/</code>	No (rebuildable)
Server configuration	<code>/etc/apache2/sites-available/</code>	No
Certificates and account	<code>/etc/letsencrypt/</code>	No (and private)

WHAT	WHERE IT LIVES	IN GIT?
Any database (accounts, progress)	MySQL or similar	No

PDFs are the one that deserves attention: they can be rebuilt from the HTML, but only with the builder scripts and some time, so your copies on the PC, server and backups are effectively their only safe store. Apply the simple 3-2-1 idea: three copies, on two kinds of storage, one of them away from the machine. Then **test a restore** at least once, because a backup you have never restored is a hope, not a plan.

Secrets Do Not Belong in Content or in the Web Root

Exercise 3 uses two scanners. The first looks for things that *look* like secrets and is noisy: it reported 66 candidates in 46 files, mostly example passwords in lessons, and it missed three of the five files that really hold a known password. The second searches for an exact string you already know is secret, and found that password in 5 content files, in 5 built pages and in one PHP file in the build. The lessons to take:

- **Change the exposed credential first.** Once a secret has been published, deleting the text does not make it private again.
- **Use obvious placeholders in lessons** (`your-db-password-here`), even in a course about your own server.
- **Keep real credentials outside the content tree and the web root**, in a file only the account that needs it can read, and outside every git repository.
- **Make the scan part of the build** so that it fails before publishing, not after.

A Rollout Checklist

1. DNS record for the new subdomain, and wait for it to resolve.
2. Virtual host generated, enabled, `configtest` passed, Apache reloaded.
3. Certificate includes the name; HTTPS works with no warning.
4. The site's `robots.txt` blocks crawling while it is a test site; it allows crawling when it goes live (Learning Website: Framework & Architecture 7).

5. First deploy through the release script; `current` points where expected.
6. Secrets scan passes; links and pages checked (Learning Website: Framework & Architecture 12).
7. Redirects from the old paths switched on last, once the new site is verified.
8. Rollback rehearsed once before it is needed.

Hands-On Exercises

EXERCISE 1

Generate an Apache virtual host for each site from the site map, plus the single certbot command covering every name. Check for duplicate names or document roots, balanced tags, valid subdomain labels, and names missing from the certificate command. Say what the check cannot prove.

 [View solution](#)

EXERCISE 2

Write `deploy.sh` and `rollback.sh`: copy each build into a timestamped release, switch `current` atomically, refuse a build without `index.html`, prune old releases safely, and roll back in one command. Write a test that deploys three times, rolls back, deploys again and tries an empty build, and run it.

 [View solution](#)

EXERCISE 3

Write a pattern scanner and an exact-match scanner for secrets. Neither may print the values it finds. Run both on the content tree and the built site, compare what each catches, and write down the steps to take if a real credential is found.

 [View solution](#)

CHAPTER 8 QUICK REFERENCE

- Same stack as today: Debian, Apache, Certbot; **one virtual host per site**, generated from the site map
- Layout: `/var/www/<site>/releases/<timestamp>/` with a `current` link as the document root
- Name-based hosting: unknown names go to the **first** virtual host loaded, so keep a deliberate default
- Test and reload: `a2ensite`, `apache2ctl configtest`, `systemctl reload apache2`
- Certificates: one certificate with every name (HTTP method) or a wildcard (DNS method, one label only)
- Deploy atomically: `ln -sf` then `mv -Tf`; refuse a build with no `index.html`; roll back by moving the link
- Prune releases by name, never the live one, and give each release a unique name
- Back up what git does not hold: **PDFs**, server config, certificates, any database; test a restore
- **Secrets**: never in content or the web root; rotate anything that was published; scan before every deploy
- A pattern scan is a heuristic (it missed 3 of 5 known files); an exact-match scan finds what you already know

Learning Website: Framework & Architecture

Chapter 9 · Accounts, Progress & Shared Services

This is the optional chapter. Everything so far has been about publishing static pages, which needs no accounts at all. The question here is whether the sites should *remember* visitors: which chapters they have finished, which course they were on, perhaps a bookmark or a comment. The honest answer might be “no”, and a good design chapter says so. If the answer is “yes”, there are several ways to do it, and the multi-site layout makes some of them harder than they look.

DECIDE ON THE NEED FIRST

Accounts are the most expensive feature you can add to a static site: they bring security work, stored personal data, password resets and legal duties. Add them only if a feature you really want cannot work without them. Chapters 1 to 8 and 10 to 12 do not depend on this chapter.

What Exists Today

- **The learning site has no visitor accounts.** Its admin page is a read-only status dashboard, protected by Apache Basic Auth through an `.htaccess` file, not by a login system of its own.
- **The Anime Vault is a separate PHP and MySQL application with real accounts.** It has its own login, registration, roles and admin pages. It sits inside the static site's `public/` folder, so the static build carries it along.

The Anime Vault is a useful reference because it is a working, real account system. Reading its code (Exercise 3) shows bcrypt password hashing, prepared statements in 60 places, a session cookie that is HttpOnly, SameSite and Secure on HTTPS, a fresh session id at login and a CSRF token. That is a sound base. It also raises a design point for the new layout: a dynamic application with its own database does not belong inside the output folder of a static site. Give it its own site (for example `anime.osztromok.com`) with its own document root, and the static builds stay purely static.

Five Options

OPTION	WHAT IT GIVES	COST
A. No accounts	Nothing remembered	None. Simplest and safest.
B. Browser storage	"Mark as done" and progress bars, saved in the visitor's own browser	Per site only (see below), lost if the visitor clears their data or changes device
C. Export and import	B plus a file the visitor can save and load on another device	A little more code; the visitor holds their own data
D. A shared account service	One login, progress follows the visitor across sites and devices	The full cost of accounts, on every site
E. Separate apps keep separate accounts	Each dynamic application (like the Anime Vault) has its own login	Several logins, but a problem in one cannot touch the others

For a learning site, option B or C covers most of the value: "I have done chapters 1 to 4" does not need a server. They are also the options that need no personal data from visitors. Choose D only if you want progress to follow a visitor across devices or sites, or if you want comments.

Browser Storage Meets Subdomains

Chapter 1 explained that each subdomain is a separate origin. Browser storage follows origins, so `localStorage` on `languages.osztromok.com` cannot be read from `systems.osztromok.com`. Progress saved by the languages site is invisible on the systems site. That is fine for per-site progress bars, which is what most visitors want. It means a single "continue where you left off" list across all sites needs more than browser storage: either the shared cookie or a server.

STORAGE CAN BE EMPTY AT ANY TIME

Private windows, cleared site data and blocked storage all happen, and a script that reads `localStorage` can throw an error. Wrap every read and write in a `try` block, and make the page work normally when storage is missing.

If You Do Add Accounts

One identity, or several?

A **shared** login is a cookie with a `Domain` attribute for `osztromok.com`, which the browser sends to every subdomain. It is convenient, and it ties all sites together: a script-injection flaw or a handed-over subdomain on any one site becomes a risk for every site. The alternative is a host-only cookie using the `__Host-` prefix, which browsers only accept with `Secure`, `Path=/` and no `Domain`. That cookie can never be shared, so each site has its own session, normally obtained through a central login page that redirects back (single sign-on). Exercise 2 builds both headers and checks them.

Talking to an account service from another site

If a page on `languages.osztromok.com` calls an API on `account.osztromok.com` to save progress, the browser treats it as a cross-origin request (Learning Website: Framework & Architecture 1). The API has to allow that origin explicitly, and a request that carries cookies needs the API to name the exact origin rather than a wildcard and to allow credentials. Every extra allowed origin is extra attack surface, so allow only the sites that need it.

Passwords and sessions

- Store only a **salted, deliberately slow hash** (bcrypt, scrypt or Argon2), never the password.
- Compare in **constant time**, and regenerate the session id at login.
- Protect state-changing requests with a **CSRF token**, and use POST for them, not GET.
- Set the cookie `Secure`, `HttpOnly` and `SameSite`.
- Limit login attempts, and plan password reset before launch, not after.

The course Authentication & Session Security covers each of these in depth. Do not write your own scheme from this summary: use a well-reviewed library or framework feature (Django and Next.js both have one).

What to Store, and Why Less Is Better

Progress needs very little: who, which page, when. The identity of a page is the problem. The best stable identifier is the page's root-relative path from Chapter 3, with an *alias table* from old paths to new ones for renames and moves, generated from the same redirect map you build in Learning Website: Framework & Architecture 11. Exercise 1 builds this with SQLite.

Once a visitor's email address or progress is tied to an identity, you are holding personal data. If you are based in the United Kingdom, UK data protection law applies to you as the person holding it. In general that means: collect only what you need, say what you keep and why, let people see and delete their data, keep it secure, and have a plan for a breach. This is not legal advice; read the Information Commissioner's Office guidance for small organisations before you launch anything that stores personal data. Options A to C avoid most of this, which is a good reason to prefer them.

Hands-On Exercises

EXERCISE 1

Build a small progress store with SQLite: a completed-chapters table keyed by user and page path, an alias table for renamed paths, and functions to mark a chapter done and to report a course's progress. Test it on a real course, including marking a chapter twice and renaming one.

 [View solution](#)

EXERCISE 2

Write functions that build a shared session cookie and a host-only `__Host-` cookie, and a checker that reports missing attributes. Then hash and verify a password with a random salt and a constant-time comparison. Say what each design choice protects against.

 [View solution](#)

EXERCISE 3

Review the Anime Vault's login code with a script that checks admin pages for authentication, POST handlers for CSRF checks, SQL for interpolated variables and output for unescaped request values. Then open every flagged file and decide which findings are real.

 [View solution](#)

CHAPTER 9 QUICK REFERENCE

- Today: the learning site has **no visitor accounts**; the Anime Vault is a separate PHP app with its own login
- Options: none, browser storage, export/import, a shared account service, or separate apps with separate accounts
- Browser storage is **per origin**: progress saved on one subdomain is not visible on another
- Always guard storage with `try`; pages must work without it
- A shared cookie (`Domain=osztromok.com`) ties every subdomain's security together; `__Host-` cookies cannot be shared
- Cross-origin API calls with cookies need the exact origin allowed, plus credentials
- Passwords: salted, slow hash; constant-time compare; new session id at login; CSRF token on POST
- A stable page id is the page's path, plus an alias table for renames and moves
- Keep dynamic apps out of the static output: give them their own site
- Personal data brings legal duties: prefer options that avoid collecting it (this is not legal advice)

Learning Website: Framework & Architecture

Chapter 10 · Migrating One Area at a Time

You now have a design for the family of sites: a site map, a content model, a shared design layer, three layers of navigation, a pipeline, search and sitemaps, and hosting. What remains is the hardest part, which is getting from the site you have to the sites you want without breaking what works. The safe way is not to rebuild everything and switch over on one weekend. It is to move one area at a time while the old site keeps running. This chapter works through that plan for the languages area, using measurements of your real site.

The Strangler Pattern

The approach is usually called the **strangler** pattern, after a vine that grows around a tree until it can stand on its own. The new sites grow around the old one. For a while both exist. Each area that moves is verified, redirected and then retired from the old site, so the old site shrinks as the new ones grow, until there is nothing left to retire. At every moment there is a working site, and until an area is retired there is a way back.

WHY NOT MOVE EVERYTHING AT ONCE?

A single big switch gives you one test, one chance, and a very large set of things that can go wrong together. Moving one area gives you a small problem you can understand, and teaches you what to fix before the next area. Google's own site-move guidance accepts a section-by-section move for large sites (Learning Website: Framework & Architecture 7).

Why Languages First

You chose languages first because it is the fastest-growing area. The measurements say it is also the *easiest* area to move. Exercise 1 scanned the 593 language pages for links:

LINK TYPE	COUNT	WHAT IT MEANS FOR THE MOVE
From languages to another area	0	Nothing in these pages needs a cross-site link
Into languages from another area	0	No other page needs updating

LINK TYPE	COUNT	WHAT IT MEANS FOR THE MOVE
Within languages	225	Stay relative; they keep working
To resource files (<code>/resources/japanese/...</code>)	23	The files must move with the area
To the kanji tiles page (a special route)	30	That page moves with the area
To a page that does not exist	282	An existing bug: fix it first

A clean cut with no links in or out is rare. It means the cross-site navigation work from Chapter 5 is not even needed for the first move, and every later area can be measured the same way before its turn.

DO NOT MOVE A BUG

The 282 broken links are the back-links on the hiragana and katakana character pages: 141 pages each point at `/japan/hiragana/hiragana-tiles` or `/japan/katakana/katakana-tiles`. The folders contain the tile content (`hiragana_tiles_cms_content.html` and its katakana twin), but only the kanji tiles file has a page that routes it. Those links are dead on the current site (checked against the built output). Fix them on the old site first, so the new site starts clean and you can tell whether a later failure is new.

Update: this was fixed after the course was written, by adding the two missing pages (`hiragana-tiles.astro` and `katakana-tiles.astro`, modelled on the kanji one). A rebuild and the link check of Chapter 12 then showed the 282 broken links gone.

The Phases

PHASE	WHAT HAPPENS	THE WAY BACK
0. Prepare	Measure, fix known bugs, scan for secrets, decide the exceptions	Nothing has changed yet
1. Build in parallel	New subdomain, certificate and virtual host; first release deployed; crawling blocked	Delete the new site; the old one is untouched
2. Verify	Compare pages, links, files and visuals against the old site	The old site still serves everyone
3. Rehearse redirects	Turn on the redirect rule as 302 (temporary) and test every old URL	Remove the rule

PHASE	WHAT HAPPENS	THE WAY BACK
4. Switch	Allow crawling, submit the sitemap, change redirects to 301 , replace the old menu entry with a link	Switch the rule back to 302 or remove it
5. Watch	Logs, Search Console, visitor reports, for several weeks	As above
6. Retire	Remove the area from the old site's build; keep redirects for at least a year	Restore from the content folder and the release history

The block on crawling in phase 1 matters because, until the switch, both sites would serve the same pages. Search engines should find the pages in one place only (Chapter 7's rule about one address per page).

Verify with Scripts, Not Eyes

During phase 2 the question is "did every page survive?". Compare two sets: the pages the content tree says should exist, and the pages that were actually built, in both directions. Exercise 2 does this on today's build as a rehearsal:

WHAT THE REHEARSAL SHOWED

Of 4,925 expected pages, **none was missing** from the build. The build also contains **89 pages that nobody planned**: mostly the printable versions of cheat sheets and lessons (names ending in `_print`), some resource folders, the home page, the admin dashboard and the course index. So the print versions are real, public pages. Decide what you want for them (keep, point a canonical link at the main page, or ask search engines not to index them), and then the migration will not surprise you with them.

Use the same check on the new site, with the area's expected pages against its output folder. The goal is zero missing pages and an extras list you can explain line by line.

The Exceptions List

Pages move by their folder, but some things are not content folders. Exercise 3 inventories them from the real `public/` folder:

ITEM	SIZE	DECISION
<code>/resources/japanese/</code>	312 files, 1.6 MB	Moves with the languages area
<code>/resources/hungarian/</code>	11 files, 104.8 MB	Moves with the area, after checking (see below)
<code>/japan/kanji-tiles/</code>	one page	Moves with the area
Other resource folders	csset, jquery, js, js2021, linux, php, postman	Not language assets: they stay until their own areas move

LOOK AT THE 105 MB FOLDER BEFORE YOU COPY IT

`/resources/hungarian/` holds PDFs, a video and screenshots, and their names look like a published Hungarian textbook and a lesson recording. Before copying them to a new public site, check that you have the right to publish them (the course Copyright & Fair Use covers this), and consider whether 100 MB of video belongs in the web root at all. A move is a good moment to clean up.

Running Side by Side

For the duration of the migration the old site's virtual host does not change; the new one is added beside it (Chapter 8). The old site's menu and the new site's global bar link to each other so that visitors can find their way. The only place the two meet is the redirect rule on the old site, which stays off until phase 3. During the move, treat the content folder as the single source for both: the old site builds from it, the new site builds from it, and a correction in one place appears in both until the area is retired.

After the Switch

- **Redirects stay** for at least a year, and ideally forever, because old links live in other people's bookmarks and pages.
- **The old menu entry changes** from a dropdown of language courses to a single link to the new site.
- **Search Console:** add the new site, submit its sitemap, and watch crawl errors and coverage for the old addresses.

- **Then repeat** for the next area: measure its links, list its exceptions, fix its bugs, and follow the same phases.

Hands-On Exercises

EXERCISE 1

Write a link inventory for one area: count its links to its own pages, to other areas, to special routes and to resource files, count links into it from other areas, and check that every target exists. Run it on the languages area and say what it tells you about the move.

 [View solution](#)

EXERCISE 2

Write a parity check that compares the pages the content tree expects with the pages in a build output, reporting pages missing from the build and pages built but not expected. Run it on the current site's build and explain the extras.

 [View solution](#)

EXERCISE 3

Inventory the resource folders and special routes that do not move by the folder rule, decide where each goes, and write a seven-phase cutover plan for the languages area with a way back at every phase.

 [View solution](#)

CHAPTER 10 QUICK REFERENCE

- **Strangler pattern:** new sites grow around the old; move, verify, redirect, retire one area at a time
- Languages is the first area: **0 links in or out** across 593 pages, so it is a clean cut
- Fix existing bugs first: **282 dead back-links** (hiragana and katakana tile pages)
- Phases: prepare, build in parallel (crawling blocked), verify, rehearse with 302, switch with 301, watch, retire

- Verify with scripts: expected pages against built pages, in both directions
- The current build has **0 missing** pages and **89 unplanned** ones (mainly print variants)
- List what does not move by folder: resource folders and special routes
- Check `/resources/hungarian/` (104.8 MB) before copying it to a new site
- Keep redirects at least a year; change the old menu entry into a link; repeat for the next area

Learning Website: Framework & Architecture

Chapter 11 · Rewriting Links & Redirects

Splitting a site breaks two kinds of link: the links *inside* your pages that assume one site, and the links *to* your pages that live in search engines, bookmarks and other people's websites. The first kind sounds like the bigger job, because there are thousands of pages. The second kind is the one that really matters. This chapter starts by measuring, because the measurement changes the plan.

MEASURE BEFORE YOU PLAN

Every figure in this chapter comes from a script run over the real content folder (Exercises 1 to 3). The result is surprising: the feared job, rewriting links in thousands of pages, turns out to be almost nothing.

The Link Audit

The census read every `href` and `src` attribute in 4,483 HTML files:

KIND OF LINK	COUNT	SHARE
Root-relative (<code>/path/</code>)	10,516	85.1%
Relative (<code>path</code> , <code>../path</code>)	1,264	10.2%
Absolute, external	305	2.5%
Anchor only (<code>#top</code>)	271	2.2%
Absolute, on the site's own domain	0	0%
Other (empty, mailto)	2	0.0%

Four facts follow from the census and the cross-site check that came after it:

1. **90% of all links (11,152) are solution `.txt` links.** They are written in many stale forms, and the build already rewrites every one of them from its file name (Learning Website: Framework & Architecture 6). No work.

2. **No page links to its own site by absolute URL.** There is nothing like `https://www.osztromok.com/...` to rewrite.
3. **Only 55 links leave their own site, and none of them is a real cross-site link.** Thirty go to the kanji tiles page, which belongs to the languages area. Twenty-five are example addresses inside lessons, such as `/about` in a web framework tutorial, not links to your own pages.
4. **Relative links are safe:** there are only 43 once solution links are set aside, and none crosses a site boundary.

WHY THERE ARE SO FEW CROSS-LINKS

Cross-references between chapters are written as *plain text*: a course name and chapter number, as the site's own rule requires, never as a link. That habit, adopted for readability, turns out to make the split almost free. If you later want those references clickable, generate the links from the course name deliberately (a "see also" list) instead of scattering paths through the text.

A Rewriter You Hope Not to Need

The content needs no rewriting today, but the first cross-site link someone adds will, so a correct tool belongs in the build. Rules for a safe rewriter:

- **Rewrite only what is needed:** a root-relative link whose target belongs to a different site becomes an absolute URL on that site; everything else is left exactly as it is.
- **Never touch** external links, anchors, relative paths or solution links.
- **Report what you cannot place** (a path in no site) instead of guessing.
- **Keep the path, query, fragment and trailing slash** exactly.
- **Have a dry run** that reports changes without writing files.

Exercise 2 builds this on top of the link resolver from Chapter 5. Its dry run over all 4,483 real pages changes nothing, as the audit predicted, and it reports 23 links it cannot place, which are the example addresses inside lessons.

```
# demo page built for the languages site
/linux/shell-and-scripting/vim/#modes -> https://systems.osztromok.com/linux/shell-and
/sidebar/football/                    -> https://humanities.osztromok.com/sidebar/footl
/japan/hiragana-1/                   (same site: left alone)
```

```
solutions/x_exercise1.txt      (solution link: left alone)
/no-such-area/page/           (reported as unknown, never guessed)
```

Redirects: Where the Real Work Is

Visitors, bookmarks, search results and other people's pages all use the old addresses. Every one

of them is a link to the old site. The old site is not a link to the new site, but the new site is a link to the old site.

not be written by hand, and their **order matters**: specific rules must come before general ones.

ORDER	RULE TYPE	EXAMPLE
1	Resource folders that move with an area	<code>/resources/japanese/...</code> and <code>/resources/hungarian/...</code> go to the languages site
2	Sidebar subjects	<code>/sidebar/linux/...</code> goes to the systems site, <code>/sidebar/football/...</code> to humanities
3	Ordinary folders	<code>/hungary/...</code> goes to languages, <code>/linux/...</code> to systems

```
# generated from the site map: the first lines of 17 rules
RedirectMatch 301 ^/resources/japanese(/.*)?$ https://languages.osztromok.com$0
RedirectMatch 301 ^/resources/hungarian(/.*)?$ https://languages.osztromok.com$0
RedirectMatch 301 ^/sidebar/ai(/.*)?$ https://ai.osztromok.com$0
RedirectMatch 301 ^/sidebar/linux(/.*)?$ https://systems.osztromok.com$0
# ... then one rule per site for its ordinary folders
```

Exercise 3 generates all 17 rules and tests every old URL. The result: all 4,925 pages, plus two sample resource files, are redirected to the right site; none is matched by no rule; none is sent to the wrong site; and no URL matches rules for two different sites. The only path left unmatched on purpose is `/sidebar/` itself, because the sidebar splits across six sites, so decide where the old index page should go (the root landing page is a sensible choice).

A SCRIPT TEST IS NOT A SERVER TEST

These rules were checked with Python's regular expressions. They have not been run through Apache. Before launch, test each rule on the real server with `curl -I`, and look especially at: the `$0` substitution (the whole matched path); **non-ASCII addresses**, which are common here (the hiragana

and katakana pages have names such as `hiragana_あ.html`) and must be matched and re-encoded correctly; and the address with and without a trailing slash.

Keeping Links Healthy Afterwards

- **Check the built site, not the source.** After each build, follow every internal link in the output and confirm it reaches a file. The 282 dead tile links found in Chapter 10 are exactly what this finds.
- **Check external links on a schedule.** The 305 external links (the most common hosts are `code.claude.com`, `github.com`, `linkedin.com` and `youtube.com`) rot over time. A monthly check is plenty.
- **Watch the 404s.** The server log shows which old addresses visitors still ask for, which is the best list of redirects you missed.
- **Never reuse an old address** for different content while its redirect still exists.

Hands-On Exercises

EXERCISE 1

Write a link census that counts every `href` and `src` by kind (root-relative, relative, absolute on your own domain, external, anchor). Then write a second script that resolves each link to a site and counts the links that leave their own site, setting solution links aside. Summarise what it means for the migration.

 [View solution](#)

EXERCISE 2

Write a link rewriter that changes only root-relative links to another site, leaves everything else alone, reports unknown targets instead of guessing, and has a dry-run mode. Test it on a small demo page and then in dry-run mode over all the real pages.

 [View solution](#)

EXERCISE 3

Generate the complete ordered set of Apache redirect rules from the site map, including resource folders and sidebar subjects, and test every old page URL against it: each must match exactly one site, the right one. List what the test cannot prove and how you will check it on the server.

 [View solution](#)

CHAPTER 11 QUICK REFERENCE

- Audit first: 12,358 link attributes in 4,483 files; **90% are solution links** the build already fixes
- **0 absolute links** to the site's own domain; only 43 relative links, none crossing a site
- Only 55 links leave their site and **none is a real cross-site link** (kanji tiles route and example URLs in lessons)
- Cross-references are plain text (course name and chapter number), which is why the split is cheap
- A safe rewriter changes only cross-site root-relative links, reports unknowns, and has a dry run
- Redirect order: resource folders, then sidebar subjects, then ordinary folders; generate from the site map
- 17 rules cover all 4,925 pages with no conflicts; `/sidebar/` itself needs a decision
- Python tests are not Apache tests: try `curl -I` on the server, especially non-ASCII and trailing-slash cases
- After launch: check the built site's links, check external links monthly, read the 404 log

Learning Website: Framework & Architecture

Chapter 12 · Testing, Launch & Operations

A design is only as good as the checks that prove it works and the routine that keeps it working. This last chapter gathers the tools from the whole course into a launch gate, adds the one test that has been missing (following every link in the *built* site), looks at size and cost, and sets out what to watch after launch and what to do when something goes wrong. It ends with a short roadmap for the landing page at the root domain and a recap of the course.

What to Test, and With Which Tool

You have already written most of the checks. The point now is to run them together, every time, and to fail the build when one fails:

QUESTION	CHECK	FROM
Does every folder belong to exactly one site?	Coverage checker for the site map	Chapter 2
Are the pages well formed?	Course validator (banners, filenames, chapter numbers)	Chapter 3
Are accents readable?	Contrast check on the tokens	Chapter 4
Is each menu complete?	Menu gap check against the real folders	Chapter 5
Do solution links and assets exist?	Solution audit; asset copy	Chapter 6
Are sitemaps, canonicals and robots right?	Sitemap and tag generators; redirect test	Chapter 7
Is the deploy safe? Is anything secret in the build?	Deploy and rollback test; secrets scan	Chapter 8
Did every page survive the move?	Parity check, expected against built	Chapter 10
Do all redirects land correctly?	Redirect map test; <code>curl -I</code> on the server	Chapter 11
Does every link in the finished site work?	Built-site link check	This chapter

Put them in one script, in this order, cheapest first, and stop on the first failure. Run it before every deploy. The checks that depend on a server (certificates, Apache's handling of non-ASCII addresses) cannot be simulated, so they live on the launch checklist below.

The Built-Site Link Check

Every other check looks at the source or at a plan. This one follows every internal link in the finished output and confirms it reaches a file. Run on the current site, it followed about 289,000 links in 5,014 pages and found **205 distinct broken targets on 521 pages**. The top of the list:

TARGET	PAGES	WHAT IT IS
<code>/japan/hiragana/hiragana-tiles</code>	141	The dead back-link found in Chapter 10 (since fixed)
<code>/japan/katakana/katakana-tiles</code>	141	Its katakana twin (since fixed)
<code>.../javascript/fundamentals/solutions/js1-1_challenge*.txt</code>	2 each	Missing solution files (Chapter 6 found 54)
<code>/about</code> , <code>/predictor/</code> , <code>/mark-used/<id>/</code>	2 to 6	Example addresses inside lessons, not real links

SUSPECT THE CHECKER FIRST

The first version reported **4,994 broken links to `/anime/`**, on almost every page. That folder is a PHP application with `index.php`, not `index.html`, and the checker only accepted the second. When one target is “broken” everywhere, the checker is more likely to be wrong than the site. Fix the checker, then read what is left. The remaining 205 targets were not all triaged; only the top of the list was.

HTML Checks and Visual Checks

A link checker will not tell you that a page is malformed or looks wrong. Two inexpensive additions are worth having, and neither was run for this course:

- **An HTML validator** over a sample of each page type (a chapter, a cheat sheet, a kanji page). The W3C Nu validator is the usual tool. Fragments need wrapping in a page first, since they have no `<html>` element of their own.

- **A visual sample.** Open one page of each type on each site, in a narrow window as well as a wide one, in dark and print preview. Pay special attention to a language page with its accent colour, a code block with its copy button, and a table on a phone width.

The Budget: Size, Time and Cost

Exercise 2 measured the current build by the site each file would belong to:

SITE	FILES	MB	PDF SHARE
programming	5,182	264	78%
languages	1,149	210	93%
systems	2,160	118	75%
webdevelopment	1,720	95	76%
humanities	1,686	57	63%
Others (ai, lifeskills, creative)	1,901	77	71% to 80%
Shared / not in a site	590	123	73%
Total	14,388	945	79%

PDFs are 79% of the bytes. The HTML is small. That changes the deployment design from Chapter 8: copying the whole build into every release wastes disk on PDFs that did not change. Exercise 2 tests an `rsync` option, `--link-dest`, which hard-links unchanged files to the previous release. Three releases of a 40 MB build dropped from 115 MB to 39 MB.

A FIX THAT LOOKED RIGHT AND SERVED STALE PAGES

My first `--link-dest` test looked correct by disk size, but the live page still showed version 1 after three deploys. `rsync` treats a file as unchanged when its size and modified time match, and the two versions had the same size and were written within the same second. On a real build the opposite happens: every file gets a new modified time, so nothing is ever linked. Adding `--checksum` compares contents and fixes both. Test the live content after changing a deploy, not just the disk usage.

Build **time** is its own budget. A full build of the current site (`npm run build`) was timed after the course was written: **4,996 pages in about 1 minute 35 seconds** in total. Write the number

down, and treat a large jump as a reason to look, not a number to accept.

The Launch Checklist

For each site, in order. Nothing here is optional:

1. DNS record resolves; virtual host enabled; `apache2ctl configtest` passes (Chapter 8).
2. HTTPS works for every name with no warning; renewal rehearsed with `certbot renew --dry-run`.
3. Pre-deploy gate passes: coverage, validation, menus, solutions, secrets scan, parity.
4. Built-site link check shows no new broken target.
5. `robots.txt` and sitemap are right for the stage: crawling blocked while testing, allowed at launch.
6. Canonical links use the final address; the old site is not yet redirecting.
7. Redirects rehearsed as 302 on the server with `curl -I`, including non-ASCII and trailing-slash cases, then switched to 301.
8. Old site's menu changed to link to the new site.
9. Sitemap submitted; Search Console watched for the first weeks.
10. Rollback rehearsed once, and a restore from backup tested.

Operating the Sites

The server log is the best monitor you have, and it costs nothing. Exercise 3 summarises it (on a synthetic sample, since no real log was available): status classes, the most requested missing pages, and the number of redirects served. A routine that stays small enough to actually do:

WHEN	WHAT
Daily (automatic)	Disk space (PDF-heavy releases fill disks); certificate renewal dry run
Weekly	Top missing pages from each access log; the Apache error logs; the built-site link check
Monthly	External link check (305 external links today); Search Console coverage and crawl errors
Quarterly	A real restore from backup; review of who has access to what

When It Goes Wrong

- **A bad release:** run the rollback script. The previous release is still on disk, and the switch is one rename.
- **A bad redirect rule:** switch it back to 302 or remove it; the old site is still there until the area is retired.
- **A certificate problem:** check the renewal timer and `certbot renew --dry-run` first; a name missing from the certificate is the usual cause after adding a site.
- **A leaked credential:** change it first, then clean the content, then redeploy (Chapter 8). Do not rely on deleting the text.
- **A search drop after a move:** check redirects return a single 301 to the exact new page, sitemaps are submitted, and the canonical links agree. Wait several weeks before drawing conclusions.

Roadmap for the Root Domain

Once areas have moved, `osztromok.com` becomes a portfolio and landing page. It is the simplest site of all, and it can be built last and with the same pipeline:

1. **Decide the address first:** bare domain, `www`, or both with one redirecting to the other (Chapter 5). Everything else depends on it.
2. **Content:** a short introduction, a card for each site generated from the site map (so it can never list a site that does not exist), and contact details.
3. **Keep the old site alive for the areas not yet moved**, on its own virtual host or a path, until the last area is retired.
4. **Order the remaining areas by how clean a cut each is:** run the link inventory from Chapter 10 on each, start with the cleanest, and fix its existing bugs before moving it.
5. **Retire the old site last**, keeping the redirects for at least a year.

Hands-On Exercises

EXERCISE 1

Write a checker that follows every internal link in a built site and reports the broken targets, counting each once as a distinct target and once per page affected. Run it on the current build, triage the top entries, and explain how you would tell a checker bug from a site bug.

 [View solution](#)

EXERCISE 2

Measure the build by the site each file would belong to and the share that is PDFs. Then test an `rsync --link-dest` deploy against a plain copy on a 40 MB file over three releases, and make sure the live page shows the newest version.

 [View solution](#)

EXERCISE 3

Write a summariser for Apache's combined log format that counts requests by status class and lists the most requested missing pages and the number of redirects. Test it on a small sample you invent, and write the monitoring routine you will actually follow.

 [View solution](#)

Course Complete

You have finished Learning Website: Framework & Architecture. In twelve chapters you went from asking whether the site should be split to a plan you can follow, with scripts, measurements and a rollback at each step. Here is the whole journey.

CHAPTER	TOPIC	YOU CAN NOW...
1	Why Split the Site	weigh a subdomain split against a path split, and decide area by area
2	Mapping Content to Sites	assign every folder to exactly one site and check it by script
3	The Content Model	read a page's metadata from its file and handle every banner shape

CHAPTER	TOPIC	YOU CAN NOW...
4	The Shared Design System	share a design with tokens and migrate old pages safely
5	Navigation Across Sites	build a global bar, site menus and cross-site links that cannot drift
6	The Content Pipeline	follow a file through the build and audit solution links and assets
7	Search, Sitemaps & SEO	generate sitemaps, canonicals and robots files per site
8	Hosting & Deployment	host several sites, deploy atomically, roll back and keep secrets out
9	Accounts, Progress & Shared Services	decide whether to add accounts and design them safely if so
10	Migrating One Area at a Time	move one area with a way back at every phase
11	Rewriting Links & Redirects	audit every link and generate and test the redirect map
12	Testing, Launch & Operations	gate a release, launch with a checklist and run the sites afterwards

WHAT NEXT?

This course is the blueprint. Two courses implement it: **Learning Website with Django** and **Learning Website with Next.js**. Both start with the languages site, because Chapter 10 showed it is the cleanest first move. Before you start either, fix the three open findings from this course: the exposed database password (Chapter 8), the dead hiragana and katakana tile links (Chapter 10), and the question of the 105 MB Hungarian resources folder.

CHAPTER 12 QUICK REFERENCE

- Run all the checks together before every deploy, cheapest first, and stop on the first failure
- The built-site link check found **205 distinct broken targets on 521 pages** (about 289,000 links followed)
- When one target is broken on almost every page, suspect the checker first
- **PDFs are 79% of the build** (745 of 945 MB); use `rsync --link-dest` with `--checksum` for cheaper releases

- Measure build time yourself; the number to watch is the trend
- Launch checklist: DNS, vhost and `configtest`, certificate and renewal dry run, gate, link check, crawl rules, redirects as 302 then 301, old menu link, sitemap, rollback and restore rehearsed
- Monitor: disk and certificate daily; 404s and error logs weekly; external links and Search Console monthly; a restore test quarterly
- Rollback: switch the release link; turn a redirect back to 302; the old site stays until an area is retired
- Root domain last: a landing page generated from the site map, with the address (bare or `www`) decided first