



PROJECT COURSE

Learning Website with Next.js

Building a family of subdomain sites from one Next.js workspace of shared packages: host-based routing, reading content files at build time, a shared design system, navigation, search and sitemaps, optional logins and progress, migrating from the old site with reviewed redirects, and deployment with backups and a way back. Every chapter is backed by code run on the real content.

12 Chapters

Philip Osztromok

Generated with Claude

Contents

Chapter 1	Monorepo for Many Sites	2
Chapter 2	Subdomains & Middleware	8
Chapter 3	The Content Model in TypeScript	14
Chapter 4	Reading Fragments at Build Time	19
Chapter 5	The Shared Design System	25
Chapter 6	Layout & Navigation	30
Chapter 7	The Languages Site	36
Chapter 8	PDFs, Solutions & Assets	41
Chapter 9	Search, Sitemaps & Metadata	47
Chapter 10	Accounts & Progress (Optional)	54
Chapter 11	Migrating from the Old Site	60
Chapter 12	Deployment, Testing & Operations	66

Learning Website with Next.js

Chapter 1 · Monorepo for Many Sites

The notes are going to become several sites, one for each part: languages, web development, programming and so on. *Learning Website: Framework & Architecture 2* decided which folders belong to which site, and *Learning Website with Django 1* built that as one Django project. This course builds the same thing with Next.js, so the two can be compared fairly. This first chapter sets up the shape: one repository, one small app for each site, and one shared package that holds the site map.

RUN FOR REAL, ON NODE 24, NPM 11, NEXT.JS 16.4, REACT 19.3 AND TYPESCRIPT 5.9

Everything below was installed, tested, type-checked, built and served on this machine, with three apps (portfolio, languages, web development). It was **not** opened in a browser or run on the server.

What a Monorepo Is, and the Alternatives

SHAPE	GOOD FOR	COST	USED HERE
One repository per site	Sites that really are independent	The site map and design are copied into each; they drift apart	No
One app that reads the host name	One deployment	One build and one failure for every site	No (Chapter 2 compares it)
A monorepo: one repository, one app per site, shared packages	Separate builds, one shared source of truth	A shared change means rebuilding every app	Yes

The Layout

```
learning-website-nextjs/
├─ package.json          workspaces: apps/*, packages/*
├─ tsconfig.base.json    one set of TypeScript rules for every app
├─ apps/
│   └─ portfolio/        the front page of osztromok.com (port 3000)
```

```

|   |— languages/          languages.osztromok.com      (port 3001)
|   |— webdevelopment/    webdevelopment.osztromok.com (port 3002)
|— packages/
|   |— sites/             the site map and address helpers, shared by every app
|       |— src/
|           |— index.ts
|           |— index.test.ts

```

npm **workspaces** make this work with no extra tool. The root `package.json` lists the folders that are packages, and `npm install` *links* them to each other instead of downloading them: in `node_modules/@lw/` there are four links to the real folders, so an app that imports `@lw/sites` gets the live source file, not a copy. The install took 27 seconds and added 31 packages, with 0 vulnerabilities reported.

NODE_MODULES IS 369 MB

That is for three apps, and it is why `node_modules` is never committed, and why the install is best done outside a synchronised folder such as OneDrive. pnpm would use far less disk (it shares one copy of each package); it was not run here, so that is the documented behaviour, not a measurement.

The Shared Package: the Site Map

The one thing every site must agree on is which folders belong to which site. It lives in `packages/sites` as TypeScript source with **no build step**: its `exports` field points straight at `src/index.ts`. It is the Python site map from the Django project, written again in TypeScript:

```

export const SITES = {
  languages: { title: "Languages", folders: ["france", "germany", "hungary", "japan", "c"],
  webdevelopment: { title: "Web Development", folders: ["web-development", "web-platform"],
  // ... six more sites: 8 sites and 48 folders in all
} as const;

export type SiteName = keyof typeof SITES;

export function siteForPath(path: string): SiteName { /* the first folder decides; sideb.
export function siteUrl(site: SiteName, env: Environment, path = "/"): string { /* dev o

```

Because `SiteName` is made from the object itself, a typo such as `"lanugages"` is a compile error everywhere it is used. The function `siteForPath` fails loudly with a `NoSiteError` for a path that no site owns, instead of guessing. Seven tests check that no folder belongs to two sites, that there are 48 folders, the sidebar and special-prefix rules, the addresses, and that every site has its own development port. They use Node's own test runner, which runs TypeScript files directly, so no test library was added. All 7 pass. (The first run of `npm test` failed only because the command named a folder instead of a file pattern.)

Is It the Same Map as the Python One?

A one-off script runs the Django project's Python site map, prints it as JSON and compares it key by key with the TypeScript one. Result: **identical, 8 sites and 48 folders**. That also exposes a weakness: there are now two copies of the map, and a person has to remember to compare them. The script is not part of `npm test` because it needs the Django project. In the final site only one framework would be used, which removes the problem.

One App per Site

Each app has its own `package.json`, its own port and its own build. The only unusual line is in its `next.config.ts`:

```
const config: NextConfig = {
  // the shared package is TypeScript source, so Next has to compile it too
  transpilePackages: ["@lw/sites"],
};
```

Next compiles the app's own files but treats packages in `node_modules` as already compiled, so without that line the build stops at the package's first `export`. Links from one site to another come from `siteUrl()`, so one place decides whether a link is `http://systems.localhost:3004/` (development) or `https://systems.osztromok.com/` (set `LW_ENV=prod` when building for the real domain).

REAL RUN	RESULT
<code>npm run typecheck</code>	All three apps pass
<code>npm run build:languages</code> (one app, cold)	Compiled in 2.6 s; 3 static pages; 11.4 s in all

REAL RUN	RESULT
<code>npm run build</code> (all three)	18.6 s
Start each app and fetch it with <code>curl</code>	Languages shows its five folders; web development its three; the front page lists every site
Links on the languages page	The seven <i>other</i> sites, on ports 3002 to 3008; none to itself
A page that does not exist	404

The pages are *static*: they were produced at build time and need no server code to answer. The names `*.localhost` were not opened in a browser; the requests went to `localhost`, and the host name does not matter because each app serves only its own site.

The Price of Sharing

A shared package is shared *source*, not a running service. To see what that means, I changed the title “Languages” to “Languages (changed)” in the package and rebuilt only the languages app:

BUILD	CONTAINS “LANGUAGES (CHANGED)”?
Languages (rebuilt)	Yes
Web development (not rebuilt)	No: its output still has the old title
All three, after changing it back and running <code>npm run build</code>	No, in every app (0 matches)

A change to the package reaches an app only when that app is rebuilt and redeployed, so **after changing the package, every app that uses it must be rebuilt**, or the sites disagree. For three apps that is one command (18.6 s). For eight it would be roughly 50 s, extrapolated and not measured. A tool such as Turborepo can skip the apps a change cannot affect; it was **not used or run** here, so that is what it is for, not a tested recommendation.

WHAT WAS NOT VERIFIED

No browser was used and nothing ran on the server. Only three of the eight apps exist; the other five are copies of the languages app with another name and port. pnpm and Turborepo were not tried. The TypeScript and Python site maps are compared by a script, not by a test that runs with `npm test`.

Hands-On Exercises

EXERCISE 1

Create an npm workspace with a shared `@lw/sites` package holding the site map in TypeScript, with tests that run without a test library. Install it and report the size and the test result.

 [View solution](#)

EXERCISE 2

Add three Next.js apps that import the shared package, type-check and build each one alone and all together, start them on separate ports and fetch them. State what was and was not checked.

 [View solution](#)

EXERCISE 3

Prove the TypeScript site map equals the Python one, then change the shared package and show which apps notice. Explain the cost of sharing and what you chose not to try.

 [View solution](#)

CHAPTER 1 QUICK REFERENCE

- Monorepo: one repository, one app per site (`apps/*`), shared code in `packages/*` ; npm workspaces need no extra tool
- `npm install` links the workspaces (`node_modules/@lw/...` are links); 27 s, 31 packages, 369 MB for three apps
- Shared package = TypeScript source, `exports` pointing at `src/index.ts` , no build step
- Each app needs `transpilePackages: ["@lw/sites"]` in `next.config.ts`
- `SiteName` comes from the site map object, so a mistyped site name is a compile error
- `siteForPath` throws `NoSiteError` for a path no site owns; `siteUrl` gives the dev or prod address (`LW_ENV=prod`)
- Tests: Node's own runner (`node --test ".../*.test.ts"`), 7 pass; the TypeScript and Python site maps are identical (8 sites, 48 folders)

- Builds: one app 11.4 s from cold, all three 18.6 s; pages are static
- A change to a shared package reaches an app only when that app is rebuilt: rebuild every app
- Not tried: pnpm, Turborepo, a browser, the server; five of the eight apps

Learning Website with Next.js

Chapter 2 · Subdomains & Middleware

Chapter 1 gave every site its own app, so the apps never had to ask “which site is this request for?”: Apache sent each host name to a different port. This chapter builds the other shape, **one app that answers for every site**, and measures both. The question it has to answer well is small and dangerous: which site is a request for, judging only by its Host header?

RUN FOR REAL, ON NEXT.JS 16.4

The one-app version was built, started and sent twelve different requests, including hostile ones, and opened in headless Chrome. Apache was **not** available, so the Apache lines are written and untested.

Step 1: Which Site Is This For?

The answer lives in the shared package, so every app and every test uses the same one. The rule is strict: after lower-casing and removing a port and one trailing dot, the host must be *exactly* `languages.localhost` (`languages.osztromok.com` in production). The bare domain is the front page. Anything else is `null`: not ours.

```
export function siteForHost(host: string | null | undefined, env: Environment): HostTarget {
  if (!host) return null;
  const name = host.toLowerCase().replace(/:\d+$/, "").replace(/\.$/, "");
  const base = env === "prod" ? DOMAIN : "localhost";
  if (name === base) return "portfolio";
  for (const site of SITE_NAMES) {
    if (name === `${site}.${base}`) return site;
  }
  return null;
}
```

HOST HEADER

`Languages.LOCALHOST:3010`

RESULT

`languages` (case and port ignored)

HOST HEADER	RESULT
<code>systems.osztromok.com.</code> (production)	<code>systems</code> (trailing dot ignored)
<code>evillanguages.localhost</code> , <code>languages.localhost.evil.com</code> , <code>x.languages.localhost</code>	<code>null</code>
<code>127.0.0.1:3010</code> , an empty header, <code>languages%2elocalhost</code>	<code>null</code>
<code>languages.localhost</code> in production, or the production names in development	<code>null</code>

WHAT WENT WRONG, AND HOW TO AVOID IT

The mistake: my first version also trimmed whitespace, and the test `"languages.localhost\n"` came back as `languages`. A Host header containing a newline is malformed and should be refused, not tidied up, so the function was more forgiving than the rule it implements. **The takeaway:** normalise only what the rule says is equivalent (case, port, trailing dot), and put the awkward inputs in the test before trusting the function. The fix was to remove the trim; all 9 tests pass.

Step 2: The Proxy File

In Next.js 16 the file that runs before every request is `proxy.ts`. (Earlier versions called it `middleware.ts`, and the build output still labels it "Proxy (Middleware)".) It reads the Host header and **rewrites** the request to that site's folder inside the app: `languages.localhost/` is answered by `app/sites/languages`, and the visitor's address does not change.

```
export function proxy(request: NextRequest) {
  const { pathname } = request.nextUrl;

  // /sites/... is the app's internal layout: a visitor must never reach it by typing it
  if (pathname === "/sites" || pathname.startsWith("/sites/")) {
    return new NextResponse("Not found", { status: 404 });
  }

  // only the Host header is trusted
  const target = siteForHost(request.headers.get("host"), environment());
  if (target === null) return new NextResponse("Unknown site", { status: 404 });

  const url = request.nextUrl.clone();
  url.pathname = `/sites/${target}${pathname === "/" ? "" : pathname}`;
```

```
return NextResponse.rewrite(url);
}
```

The page behind it, `app/sites/[site]/page.tsx`, lists the nine values (eight sites and the front page) in `generateStaticParams` and sets `dynamicParams = false`, so each is built in advance and any other value is a 404. The build produced nine static pages and the “Proxy” entry, in 11.2 seconds.

Real Requests

REQUEST TO THE ONE APP	RESULT
<code>languages.localhost</code> (curl resolves the name itself)	200, Languages
<code>Host: webdevelopment.localhost:3010</code>	200, Web Development
<code>Host: LANGUAGES.localhost</code>	200, Languages
<code>Host: localhost</code>	200, the front page
<code>Host: example.com</code> , <code>Host: evillanguages.localhost</code>	404, 404
languages host asking for <code>/sites/systems</code> or <code>/sites</code>	404, 404 (the internal layout is hidden)
languages host, <code>/nothing-here</code>	404
<code>X-Forwarded-Host: systems.localhost</code> sent to the languages host	200, <i>Languages</i> : the forged header was ignored
No Host header at all (HTTP/1.0)	404
A file under <code>/_next/static/</code>	200, <code>application/javascript</code> (the matcher skips it)

A headless Chrome opened `http://languages.localhost:3010/` with no special settings and showed the Languages page: browsers resolve `*.localhost` to your own machine themselves, so development needs no hosts-file entries. Only the `Host` header is trusted because `X-Forwarded-Host` is sent by whoever makes the request unless a proxy overwrites it, the same decision as in the Django project.

Step 3: One App or Many? Measure

	MANY APPS (CHAPTER 1)	ONE APP (THIS CHAPTER)
Build	18.6 s for three apps	11.2 s
Memory, running	3 processes, 268 MB (about 89 MB each)	1 process, 84 MB
One site fails	The others keep running	Every site is down together
Change to one site	Rebuild that app	Rebuild everything
Change to shared code	Rebuild every app	Rebuild the one app
Routing	Apache sends each host to a port	Apache sends every host to one port; the proxy file picks the site

HOW MUCH TO TRUST THESE NUMBERS

Memory is the Windows working set of the Node processes after each server had answered a request: an idle, freshly started machine, not a loaded server, and Debian will differ. Build times are one run each. Eight separate apps would be about 700 MB by extrapolation, which was not measured. Treat the table as the *shape* of the cost, not a benchmark.

The Apache Setting That Breaks the One-App Shape

Written but **not run** (there is no Apache here): for the one-app shape every host goes to the same port, and Apache must keep the original Host header.

```
<VirtualHost *:443>
  ServerName languages.osztromok.com
  ProxyPreserveHost On
  ProxyPass / http://127.0.0.1:3010/
</VirtualHost>
```

Without `ProxyPreserveHost On`, Apache sends `Host: 127.0.0.1:3010` to the app, which is “not ours”, so every request is a 404. It cannot be seen in development, which makes it the kind of fault that appears on the day of the move.

Choosing (an opinion, not a measurement): with eight sites on one small server, one app is the lighter choice. With sites that change at different speeds, or may move to different servers,

separate apps are safer. This course keeps the separate apps and treats the one-app version as a tested alternative.

WHAT WAS NOT VERIFIED

Nothing was run behind Apache. The case where the rewritten page is produced on each request (not built in advance) was not tried, and neither was `next dev` with the proxy file. Memory and build figures are single runs on a development machine.

Hands-On Exercises

EXERCISE 1

Write the shared function that decides the site from a Host header, strictly, and a test with hostile inputs. Show the mistake a too-forgiving version makes and fix it.

 [View solution](#)

EXERCISE 2

Build one app that rewrites each request to its site's folder by Host header, hides the internal folder, ignores forwarded hosts and refuses unknown hosts. Send it a table of real requests and open it in headless Chrome.

 [View solution](#)

EXERCISE 3

Compare one app with several on build time and running memory on this machine, write the Apache lines for both shapes, and name the setting that silently breaks the one-app shape. Say how much to trust the numbers.

 [View solution](#)

CHAPTER 2 QUICK REFERENCE

- Two shapes: many apps (Apache routes by host to ports) or one app (a proxy file routes by host inside Next)
- `siteForHost(host, env)` in the shared package: exact names only; case, port and one trailing dot ignored; anything else is `null`
- Normalise only what the rule calls equivalent: a trim made `"languages.localhost\n"` valid, and a test caught it
- Next.js 16: `proxy.ts` (earlier: `middleware.ts`); `NextResponse.rewrite` keeps the visitor's address
- Hide the internal `/sites/...` path (404); trust only `Host`, never `X-Forwarded-Host`; unknown host is a 404, never a default site
- The matcher `/((?!_next/|favicon.ico).*)` keeps Next's own files out of the proxy
- `generateStaticParams` + `dynamicParams = false`: nine pages built in advance, anything else 404
- Browsers resolve `*.localhost` themselves: no hosts-file entries
- Measured here: one app 84 MB and a 11.2 s build; three apps 268 MB and 18.6 s; one failure takes down everything in the one-app shape
- One-app Apache needs `ProxyPreserveHost On`; untested here

Learning Website with Next.js

Chapter 3 · The Content Model in TypeScript

Every site needs the same answer to “what is this file?”: which site it belongs to, what kind of page it is, its title, its course and chapter, and its dates. If each app worked that out for itself, they would drift apart. So the answer becomes one set of TypeScript types and one parser, in a package that every app imports. This chapter builds it, and then does the thing a second implementation owes you: it runs it on the real content and compares every page with what the Django project stored.

RUN FOR REAL, ON 4,414 PAGES

The package has 17 tests of its own (26 with the site package, all passing) and was run over the real content folder. It does **not** read or show page bodies; that is Chapter 4.

The Types

```
export type Kind = "course_chapter" | "sidebar" | "lesson" | "full_page" | "other";

export interface Page {
  readonly path: string;           // relative to the content folder, forward slashes
  readonly site: SiteName;
  readonly kind: Kind;
  readonly title: string;
  readonly summary: string;       // the banner's Topic line, at most 300 characters, or
  readonly courseName: string | null;
  readonly courseNo: number | null;
  readonly chapterNo: number | null;
  readonly created: IsoDate | null;
  readonly updated: IsoDate | null;
}
```

- **The path is BASE-relative:** `hungary/hungarian-basic-3/x.html`, with forward slashes. Nothing in the model knows where the content folder is on disk, so the same pages work on any machine.

- **Dates are ISO text** (`"2026-10-07"`), not `Date` objects, so no time zone can move a date by a day.
- **Everything is `readonly`, and absence is `null`**, never `""` or `0` (the summary is the one string, empty when there is none).
- **The site is a `SiteName`** from the shared package, so a page can only belong to a site that exists.

Reading the Banner

Every page begins with a comment such as `Course: German Basic Conversation 3 / Chapter: At the Station / Topic: ... / Date Created: ... Date Updated: ...`. The parser reads the comment at the very top (a comment further down is *not* a banner), strips the `====` decoration, keeps the first value of a repeated key, and finds the dates with their own pattern. A line with a Course field makes a course chapter; a Category field, a sidebar page; any other comment, a lesson; a complete HTML page is recognised by its doctype and described by its `<title>`; no comment at all is "other".

From a File to a Page

QUESTION	HOW IT IS ANSWERED
Title	Banner Chapter, then <code><title></code> , then the first h1 or h2, then the file name made readable
Course and chapter numbers	From the file name, which has come in five generations
Site	<code>siteForPath()</code> from Chapter 1; a path no site owns is a <code>NoSiteError</code> , never a guess
Dates	Must be real: <code>2026-13-01</code> and <code>2025-02-29</code> are errors, <code>2024-02-29</code> is fine
Character references in a title	Decoded once: <code>&amp;lt;</code> becomes <code>&lt;</code> , not <code><</code>
Long titles	Cut at 300 characters, counting an emoji as one

The five generations of chapter file name, each with a test:

hungarian_basic_conversation_3_1.html	course 3, chapter 1
html_lesson_01_3.html	chapter 1 of course 3 (the numbers are the other way)
js1-4.html, psp-croll-1-2.html	course 1, chapter 4 / 2 (the older prompt-prefers the newer)
setting_up_a_web_server_on_debian_03.html	chapter 3 only
linux_appendix_a.html	no number at all

Reading a Whole Folder

`loadPages(root)` walks the content folder in a stable order and returns `{ pages, errors }`. A bad file is reported in `errors` (not valid UTF-8, over 2 MB, a path that climbs out of the folder, an impossible date, a path no site owns) and never stops the others. But it re-throws anything that is *not* a content problem: a missing folder is `ENOENT`, not “a bad file”, so a real bug cannot hide in the error list. It skips files ending `_print.html`, files starting with `_`, the `pdfs` and `solutions` folders and the archive copy of the kanji pages. `groupCourses()` then groups chapters by folder in number order, so chapter 2 comes before chapter 10 and an unnumbered chapter goes last.

Is It the Same as the Django Importer?

Two implementations of the same rules should give the same answers, and the only honest way to say so is to run both on the real content. Django's stored pages were dumped to JSON, the TypeScript package parsed the same folder, and every field was compared:

COMPARISON	RESULT
Pages found	4,414 in TypeScript, 4,414 in Django; none missing either way
Errors	0
Site, kind, title, chapter number, created, updated, course number, course name, summary	0 differences on every page
Courses	396 in each; 0 differences
Time to read and parse the whole folder	1.5 to 1.9 seconds (two runs)

The real pages are 3,907 course chapters, 410 complete pages, 47 sidebar pages, 43 lessons and 7 others. The time is *not* a comparison with Django, whose import also prepares and indexes every page.

CAN THIS CHECK FAIL? I MADE IT

Zero differences on the first run is only worth something if the comparison can find one, so I broke the parser twice on purpose and ran it again:

- Named character references no longer decoded: **11 title differences**. The check works, and the small table of named references covers every real case.
- The file-name-to-title function changed to give wrong answers: **0 differences**. No real page takes its title from its file name, so that function is checked *only* by its unit tests, not by the real content.

The second result is the useful one: a check you have not seen fail has not been shown to check.

WHAT WAS NOT VERIFIED

Page bodies (Chapter 4), links, and files Django would reject while preparing the body (there were none: both sides found the same 4,414). Only the common named character references are decoded; Python decodes all of them, so a future page with an unusual one in its title would differ and the comparison would show it.

Hands-On Exercises

EXERCISE 1

Define the TypeScript types for a page and a course, and write the banner reader. Explain each decision in the types (relative paths, ISO dates, readonly, null for absence).

 [View solution](#)

EXERCISE 2

Turn a file into a page (title order, file-name numbers, real dates, decoded entities) and a whole folder into pages and errors, refusing unsafe paths and re-throwing real bugs. Test the awkward cases.

 [View solution](#)

EXERCISE 3

Compare the TypeScript reader with the Django importer on the real content, field by field, and show that your comparison can fail by breaking the code on purpose.

 [View solution](#)

CHAPTER 3 QUICK REFERENCE

- `@lw/content` holds the types (`Page` , `Course` , `Kind`) and the reader; every app imports it
- Paths are relative to the content folder with forward slashes; dates are ISO text; fields are readonly; absence is `null`
- Banner: the comment at the very top; Course makes a chapter, Category a sidebar page, any other comment a lesson, a doctype a full page
- Title: banner Chapter, `<title>` , first h1/h2, then the file name; cut at 300 characters
- Five file-name generations give course and chapter numbers; `html_lesson_01_3` is chapter 1 of course 3
- Impossible dates are errors; entities decode once; a path no site owns is a `NoSiteError`
- `loadPages` returns `{ pages, errors }` ; one bad file never stops the rest, but real bugs are re-thrown
- `groupCourses` : by folder, chapters in number order (2 before 10), unnumbered last
- Real run: 4,414 pages, 0 errors, 396 courses, 0 field differences from Django, 1.5 to 1.9 s
- The comparison was shown able to fail (11 differences with entities off), and it does not cover the file-name title fallback

Learning Website with Next.js

Chapter 4 · Reading Fragments at Build Time

Chapter 3 described every file. This chapter shows one: it turns a content file into the HTML a visitor receives, makes every page of the languages site when the site is built, and makes sure the scripts inside those pages still run. Two of the findings came from opening a real browser, not from reading the code.

RUN FOR REAL, ON 561 PAGES OF THE LANGUAGES SITE

34 tests pass; the prepared HTML of all 4,414 pages was compared byte for byte with what the Django project stored; the site was built, served and fetched; and a headless Chrome was driven to test scripts. Not tried: `next dev`, other browsers, and pages with scripts that load other scripts.

Step 1: From a File to a Fragment

A content file is not yet what a site shows. Three steps, the same ones the Django importer takes:

1. **Unwrap.** A complete HTML document gives its body, preceded by the styles from its head; a fragment loses the banner comment at its top.
2. **Fix solution links.** Every `.txt` link is pointed at the course's own `solutions` folder, whatever the old link said.
3. **Site changes.** A site may add one of its own. The languages site marks each line in the language being taught with a `lang` attribute (`lang="hu"`, `lang="ja"`), so screen readers pronounce it correctly and the browser draws Japanese with the right glyph shapes.

```
export function prepareFragment(raw: string, path: string, site: SiteName): string {
  const courseUrlPath = path.includes("/") ? "/" + posix.dirname(path) : "";
  let fragment = rewriteSolutionLinks(extractFragment(raw), courseUrlPath);
  for (const transform of SITE_TRANSFORMS[site] ?? []) fragment = transform(fragment);
  return fragment;
}
```

A body that never closes is refused with a `MalformedDocument` error rather than guessed at. Language marking adds `lang` to whole class names only (`hu`, not `hu-lesson`), leaves an element that already has one alone, and is safe to run twice. Percent-decoding of file names never throws: a cut-off sequence becomes a replacement character. The loader also gained an option to read one site only, decided from the *path* alone, so the languages site reads 561 files, not 4,414.

Is It the Same as Django's?

The Django project stored the prepared fragment of every page. A script there writes a SHA-256 of each; another here prepares every page and compares:

CHECK	RESULT
Pages prepared	4,415 in 10.6 s (one is this chapter's own new file, not yet in the Django database)
Identical to Django's stored fragment	4,414 of 4,414 ; different: 0; failed to prepare: 0
Language marking switched off on purpose	4,253 identical, 161 different : the check works
File names in solution links not decoded, or decoded with the wrong text encoding	0 different

WHAT THE LAST ROW MEANS

No real page has a percent-encoded `.txt` file name, so that decoding (and its UTF-8 handling) is checked *only* by its unit tests, not by the real content. The comparison proves what the real pages exercise, and only that.

Step 2: Make Every Page When the Site Is Built

Next.js can make a page at build time from a list of addresses. Here the list is the content folder: one address for each page of the site, so a visitor receives a finished HTML file and no code runs when they ask for it.

```
// every page of the site is made when the site is built; any other address is a 404
export const dynamicParams = false;

export function generateStaticParams() {
```

```
return [...sitePages().values()].map((page) => ({ path: urlSegments(page) }));
}
```

- `LW_CONTENT_ROOT` has **no default**: a build that quietly reads the wrong folder is worse than one that stops.
- A content file that cannot be read **stops the build**, naming the file. (None did.)
- The page title and description come from the banner. A page with its own `<h1>` shows it; every other page gets one made from its title.
- `dangerouslySetInnerHTML` is React's name for "put this text in as HTML". It is safe *here* only because the text is one of the site's own files, read at build time. It must never be given text a visitor typed.

REAL BUILD AND FETCH	RESULT
Build the languages site	567 pages made (the 561 pages plus Next's own and the test bench), 4.1 s of page generation, 14.6 s in all, 59 MB
A Hungarian lesson	200, 97 KB, heading "Buying Clothes: Sizes, Fit & Returns", 38 <code>lang="hu"</code> marks
A German lesson, a hiragana lesson	200, 65 KB and 51 KB
A page with a Japanese address (hiragana あ)	200: made at build time, found by its percent-encoded address
An address that does not exist	404
The test bench address, in a normal build	404

Step 3: Scripts Inside a Fragment

1,844 of the 4,415 pages have a `<script>` in their fragment (copy buttons and interactive tools; on the languages site only 2). They have to work, and there is a trap. A browser runs the scripts in the page it *first downloads*. But when a visitor **clicks a link**, Next fetches the new page's data and inserts it into the page, and scripts inserted that way never run. The cure is to replace each script with a fresh copy of itself, which the browser does run, and a tiny client component does that.

To see it, I made a test bench (it exists only in a build made with `LW_LAB=1`): two pages with the same counting script, one with the component and one without, and an index that links to both. A real headless Chrome was driven through the DevTools protocol, using only Node's own WebSocket:

HOW THE VISITOR ARRIVES	FIRST VERSION	FIXED VERSION
Open the page <i>without</i> the component, directly	script ran 1 time	script ran 1 time
Open the page <i>with</i> the component, directly	script ran 2 times (a bug)	script ran 1 time
Click a link to the page without the component	waiting: the script never ran (the trap)	waiting (the same: no component, no run)
Click a link to the page with the component	script ran 1 time	script ran 1 time
Click, go back, click again	not tested	script ran 2 times in the window: once per visit

WHAT WENT WRONG, AND HOW TO AVOID IT

The mistake: my first component re-created *every* script it found, including on the page that was downloaded directly, where the browser had already run them. The script ran twice; a script that adds a click handler would have added it twice. Reading the code would not have shown it; counting in a real browser did. **The takeaway:** re-run scripts only for pages that arrived by navigation, and test *both* ways of arriving with a script that counts how often it ran. **The fix:** a mark in the layout, set after the first page is hydrated. A child's effect runs before its parent's, so on the first page the component sees the mark missing (the page was downloaded: do nothing), and on every later page it sees it (the page came by navigation: re-run). Two small markers stop React's development mode, which runs every effect twice, from running a script twice.

WHAT WAS NOT VERIFIED

`next dev` (the markers are written for its double effects but it was not run), scripts that load other scripts with `src=` (the copy keeps the attributes; none was tried), and browsers other than Chrome. There are no styles, menus or navigation yet (Chapters 5 and 6), and no PDFs (Chapter 8).

Hands-On Exercises

EXERCISE 1

Write the steps that turn a content file into a fragment (unwrap, fix solution links, a site change), test the awkward inputs, and prove the result is identical to Django's on all 4,414 real pages. Show the comparison can fail, and say which parts the real data does not exercise.

 [View solution](#)

EXERCISE 2

Make every page of the languages site at build time from the content folder, with no default content path, a build that stops on an unreadable file, and 404 for everything else. Build it, serve it and fetch pages including one with a non-ASCII address.

 [View solution](#)

EXERCISE 3

Make scripts inside fragments run after a client-side navigation, but not twice on a direct load. Build a test bench and drive a real headless Chrome through it; record the failure of your first version and the fix.

 [View solution](#)

CHAPTER 4 QUICK REFERENCE

- `prepareFragment`: unwrap a document or drop the banner, point `.txt` links at the course's `solutions` folder, apply the site's own change
- Languages site change: add `lang` to whole class names `hu`, `de`, `fr`, `jp` (as `ja`); safe to run twice
- `loadPages(root, { site })` decides the site from the path alone, so other sites' files are never read
- Identical to Django on 4,414 of 4,414 pages; the check fails (161 different) when language marking is off; file-name decoding is only unit-tested
- `generateStaticParams` + `dynamicParams = false`: one page per content file, everything else 404

- `LW_CONTENT_ROOT` has no default; an unreadable content file stops the build
- `dangerouslySetInnerHTML` only for the site's own files, never for visitor text
- Real build: 567 pages, 14.6 s, 59 MB; non-ASCII addresses work
- Scripts: they run on the page first downloaded, not after a link click; `RunScripts` re-creates them, but only for navigated pages
- Found in Chrome: the first version ran a script twice on a direct load; tested both ways of arriving; not tested in `next dev`

Learning Website with Next.js

Chapter 5 · The Shared Design System

Eight sites should look like one family, and each should be recognisable at a glance. This chapter moves the Django project's theme into a package, `@lw/ui`, that every app imports: one set of stylesheets, a frame component, and a colour for each site. It answers one real question about Next.js (CSS Modules or global CSS?) and then measures whether the result looks the same as the Django site, instead of saying so.

RUN FOR REAL, ON NEXT.JS 16.4

41 tests pass. The languages and web development apps were built with the design system, served, screenshotted, and a real headless Chrome compared 27 computed styles of one lesson with the Django site. The site is **dark only**, as the Django one is: there is no light theme.

Tokens: the One Place a Colour Is Decided

`tokens.css` holds every colour as a variable, and nothing else in the design writes a colour of its own. It is the Django project's file unchanged. Two neutral families exist (a warmer one for the language courses, a cooler one for the technical courses), and each site has **one accent**. The accent is a CSS variable on the `<html>` element, so one stylesheet serves every site and only that value differs:

```
// in each app's layout
import "@lw/ui/styles.css";
import { SiteLayout, htmlProps } from "@lw/ui";

<html lang="en-GB" {...htmlProps("languages")}>
  <body><SiteLayout site="languages">{children}</SiteLayout></body>
</html>

// the result in the built page
<html lang="en-GB" data-site="languages" data-family="language" style="--accent:#a78bfa">
```

SITE	FAMILY	ACCENT
Languages	language	#a78bfa purple
Web Development	technical	#38bdf8 sky blue
Programming	technical	#44b78b green
Systems	technical	#f0938a coral
AI and Data	technical	#fbbf24 amber
Humanities	language	#fb923c orange
Life Skills	language	#6ee7b7 mint
Creative Tools	technical	#f472b6 pink

Rules That Are Tests

- **Every site has a style, and its own accent.**
- **Every accent is readable as text** on both neutral surfaces: a contrast of at least 4.5 (WCAG level AA). The test *reads the real colours out of* `tokens.css` instead of copying them into the test, so changing a token changes what the test sees. The contrast function itself is checked against known values (black on white is exactly 21; `#777777` on white is 4.48, the famous "just fails").
- **No** `!important` in any shared stylesheet.
- **Shared component rules stay weaker than a fragment's own** (no ids, at most two parts), explained below.

CSS Modules or Global CSS?

CSS Modules rename every class to something unique, so one can never clash with another. That is exactly right for the site's *own* parts and exactly wrong for the classes the lesson fragments use. A fragment is a file of HTML written long ago, and it says `class="tip-box"`. If `tip-box` were a module class, the stylesheet would hold a renamed class and the fragment would match nothing. So the choice follows from one fact: **who writes the class name.**

STYLES FOR	KIND	WHY
The site's frame: header, footer, skip link	CSS Module	Only the site's own code names them; nothing in a fragment ever does
<code>.tip-box</code> , <code>.code-block</code> , tables, <code>.page-title</code>	Global CSS	Fragments name them, so the names must not change

The built output shows the difference:

```
class names in the built page:      NMm9mq_bar NMm9mq_footer NMm9mq_name NMm9mq_skip
selectors in the built stylesheet: .NMm9mq_skip{... :root{ *,:before,:after{ .tip-box
                                     (global
```

A fragment also brings its *own* scoped `<style>` (for example `.hu-lesson .tip-box { ... }`). The shared rules use a single class, so the fragment's rule is more specific and wins. That is deliberate: an old page keeps its look. A test reads `components.css` and fails if any selector has an id or more than two parts.

Does It Look the Same as the Django Site?

Both were started, and a real headless Chrome, driven through the DevTools protocol, opened the **same Hungarian lesson** on each and read the computed style of 27 properties: the accent variable, the body, the main column, the page title, a tip box, a dialog card, a vocabulary table, a speaker label.

RUN	RESULT
As built	26 identical, 0 different , 1 absent on both sites (a link in the page body that this page does not have)
Page background and text tokens each changed by one step	24 identical, 2 different : <code>rgb(15, 17, 23)</code> against <code>rgb(15, 17, 24)</code> , and 236 against 237
Radius changed from 8 to 10 px and the language surface by one step	26 identical, 0 different

THE LAST ROW IS A FINDING, NOT A FAILURE

The lesson carries its *own* scoped stylesheet, which sets its own radius and surfaces, so changing those tokens changes nothing the lesson shows. The shared tokens only reach the page frame and anything a fragment leaves unstyled. So this check proves that the frame and the page-level colours match, and it says **nothing** about a token that every fragment overrides. Without the middle row (a change the check *can* see) its first result would be worth much less.

A *style guide* page (`/styleguide` , in both apps) shows every shared building block at once. Looked at by eye: the Languages site is purple on the warmer neutral surface, the Web Development site is sky blue on the cooler one, and the warning and finding boxes keep their own colours on both.

WHAT WAS NOT VERIFIED

Only one lesson page was compared with Django (the other languages and the other sites were not), and only in Chrome. The design is dark only: there is no light theme and no `prefers-color-scheme` switch. Fonts are the system fonts; font loading is Chapter 8. The menus, breadcrumbs, search and account rules of the Django stylesheet come in the chapters that need them.

Hands-On Exercises

EXERCISE 1

Move the design tokens into a shared package and give each site its own accent through a CSS variable on the `html` element. Write tests that read the real stylesheet: every accent readable on both surfaces, no two the same, every site styled.

 [View solution](#)

EXERCISE 2

Decide which styles are CSS Modules and which are global, from the fact of who writes the class name. Build the site frame as a module, keep the fragment-facing classes global, and show the difference in the built output. Guard the specificity rule with tests.

 [View solution](#)

EXERCISE 3

Measure whether a real lesson looks the same on the Django site and the Next.js site by comparing computed styles in a real browser. Show that your check can fail, and say what it cannot see.

 [View solution](#)

CHAPTER 5 QUICK REFERENCE

- `@lw/ui`: `styles.css` (tokens, base, components), `SiteLayout`, `htmlProps(site)`, `StyleGuide`, contrast helpers
- Colours live only in `tokens.css`; each site's accent is `--accent` on `<html>`, from `SITE_STYLE` in the site package
- Two neutral families (language, technical); eight accents, all different, all at least 4.5 contrast on both surfaces
- Modules for the site's own frame; global CSS for classes that fragments write (`tip-box`, `code-block`, tables)
- Built proof: module classes become `NMm9mq_bar`; `.tip-box` keeps its name
- Shared rules: one class each, no ids, no `!important`, so a fragment's own scoped rule wins
- Parity with Django: 26 of 26 comparable styles identical; the check fails on a one-step colour change; it cannot see tokens that fragments override
- Dark only; no light theme; system fonts; only one lesson and Chrome compared

Learning Website with Next.js

Chapter 6 · Layout & Navigation

A visitor needs three answers on every page: which site is this, where in it am I, and where can I go next. This chapter builds them in the Next.js App Router with layouts, and works out every one of them from the pages themselves, so none can drift from the content. As elsewhere in this course, the result is compared with the Django project's on the real content, and then clicked through in a real browser, which found a bug the comparison could not.

RUN FOR REAL, ON 4,417 PAGES AND 545 FOLDERS

50 tests pass; about 9,400 navigation answers were compared with the Django project's (0 differences) and three planted mistakes were all caught; the languages site was built (601 pages, 13 s) and clicked through in headless Chrome. The web development app shows its menu but has no content routes yet, so its menu links lead nowhere.

Three Layers, All Derived

LAYER	WHAT IT SHOWS	WORKED OUT FROM
1. The bar of sites	All eight sites, the current one marked	The site map (so the addresses differ between development and production by themselves)
2. The site menu	This site's subjects that really have pages	The pages' first folder, in the site map's order
3. The breadcrumb and chapter links	Site, folder, course, page; previous and next chapter	The page's path, its course's name, and chapter <i>numbers</i>

The logic is a package of plain functions, `@lw/navigation`, with no React in it, so it can be tested without a browser. A few of its rules:

- **Labels:** `data-science-and-ml` becomes "Data Science and ML" and `cloud-and-devops` "Cloud and DevOps": small joining words stay lower case, a list of acronyms is upper-cased, and a few product names have fixed spellings.
- **Breadcrumbs:** a course folder is labelled with the *course's name*, not its folder name; the page you are on is not a link.

- **Previous and next:** by chapter *number*, so chapter 10 follows 9, not 1.
- **Folder listings:** sub-folders with page counts, then the folder's own pages in chapter order. The folder pages are the site's index, and nothing was written to make them.
- **Links between sites:** `resolveLink` keeps a link inside the site, gives a link into another site that site's absolute address, leaves external links, anchors and `.txt` files alone, and *throws* for a path no site owns: a broken link should fail, not be guessed.

WHAT WENT WRONG, AND HOW TO AVOID IT (1)

The mistake: my first version of the index class declared a field in its constructor

(`constructor(readonly site: SiteName, ...)`). Node runs TypeScript by stripping the types, and refused it: "parameter property is not supported in strip-only mode". **The takeaway:** write TypeScript that is only types plus plain JavaScript (declare the field on its own line), and the same files run in Node, in the tests and in Next.

Is It the Same as Django's?

A script in the Django project writes what its navigation says for the real content; the TypeScript version is asked the same questions:

COMPARED	COUNT	DIFFERENT
Site menus	8	0
Previous and next chapter of every page	4,417	0
Breadcrumb of every page	4,417	0
Listing of every folder (sub-folders, counts, pages, course name, breadcrumb)	545	0

Can the comparison fail? Three deliberate breakages, each undone afterwards:

PLANTED MISTAKE	WHAT THE COMPARISON REPORTED
Small words ("and", "of") no longer kept small	5 menus, 1,158 breadcrumbs and 197 folder lists different
Previous chapter chosen with <code><=</code> instead of <code><</code>	3,907 previous/next different
Chapters sorted as text (10 before 2)	3,115 previous/next different

Layouts

In the App Router a **layout** wraps the pages below it and is *not drawn again* when the visitor moves between them, which is what a site frame wants. Two are used:

- **The root layout** (`app/layout.tsx`): the page frame, the bar of sites and the site's menu, the same on every page.
- **A nested layout** (`app/[...path]/layout.tsx`) for content pages: it makes the breadcrumb. It is not redrawn when the visitor moves, but it *is* handed the address's segments, so the breadcrumb can be made from them.

```
export default async function ContentLayout({ children, params }) {
  const { path } = await params;
  const page = content.pageFor(path);
  const crumbs = page
    ? nav.breadcrumbs(/* the page's folder *//, page.title)
    : nav.breadcrumbs(decodeURIComponent(path.join("/")));
  return <><Breadcrumbs crumbs={crumbs} />{children}</>;
}
```

One route handles *two kinds* of address: a page, or a folder (a listing). `generateStaticParams` lists both, a page wins if a page and a folder share an address, and anything else is a 404. Links inside the site use Next's `Link` (a click loads only what is new); the bar of sites uses plain links because they go to other hosts. The menu is a **client component** for one reason only: the root layout is not redrawn when the visitor moves, so something has to ask the browser where we are now (`usePathname`) to mark the current subject. Everything else sends no script.

REAL BUILD AND FETCH (LANGUAGES SITE)	RESULT
Build	601 pages made (561 pages, the folder listings, the home page, the 404 page and a few of Next's own), 3.0 s of generation, 13.2 s in all
Bar of sites	Eight links; Languages marked current, the others on their own ports
Site menu	France, Germany, Hungary (current), Japan, Culture
Breadcrumb	Languages > Hungary > Hungarian Basic Conversation 3 > Work & Careers: Applying for a Job & Interviews

REAL BUILD AND FETCH (LANGUAGES SITE)

RESULT

`/hungary`

A listing: Hungarian Alphabet 1 (20), Basic 1 (12), Basic 2 (12), Basic 3 (12), Hungarian Lessons (12)

Click Through It

Reading the HTML shows that the links are there, not what happens when they are clicked. A real headless Chrome opened a lesson, put a marker on the page's `window`, and clicked through the navigation. If the marker survives a click, the page was *not* reloaded.

STEP	ADDRESS AFTERWARDS	RELOADED?	MENU MARKED	FIRST RUN	AFTER THE FIX
Open chapter 5 directly	..._3_5	(start)	Hungary	ok	ok
Click Next	..._3_6	no	Hungary	ok	ok
Click Previous	..._3_5	no	Hungary	ok	ok
Click the course in the breadcrumb	/hungary/hungarian-basic-3	no	Hungary	ok	ok
Click Japan in the menu	/japan	no	none in the first run; Japan after the fix	bug	ok
Browser Back	/hungary/hungarian-basic-3	no	Hungary	ok	ok

WHAT WENT WRONG, AND HOW TO AVOID IT (2)

The mistake: the menu marks an item current when the path starts with the item's address. The addresses end in a slash (`/japan/`) but Next serves `/japan`, which does not start with `/japan/`. Pages *inside* a subject worked, so looking at lesson pages never showed it; only a click on a subject's own page did. **The takeaway:** test the case at the edge of a rule (the subject's own page, not only pages below it), and compare addresses without relying on a trailing slash. **The fix:** compare with the slash removed, and require a slash after the subject (`pathname === base || pathname.startsWith(base + "/")`), which also stops `/ai` matching `/ai-tools`.

WHAT WAS NOT DONE

The web development app has a menu but no content routes, so its menu links (including Sidebar) lead to a 404 until that site's pages are built; only the languages site is complete. `resolveLink` is tested but not yet applied to links *inside* lesson pages (Chapter 11). Keyboard use and a screen reader were not tried, and only Chrome was used.

Hands-On Exercises

EXERCISE 1

Write the navigation as plain, tested functions: labels, the bar of sites, the site menu, breadcrumbs, previous and next chapter, folder listings and cross-site links. Compare every answer with the Django project's on the real content, and plant three mistakes to show the comparison catches them.

 [View solution](#)

EXERCISE 2

Build the navigation components and a root layout plus a nested layout that makes the breadcrumb from the address. Make one route serve both pages and folder listings, keep the menu as the only client component, build the languages site and fetch it.

 [View solution](#)

EXERCISE 3

Drive a real headless Chrome through the navigation and check that each click avoids a page reload and that the menu follows. Record any bug you find, fix it, and say what you did not test.

 [View solution](#)

CHAPTER 6 QUICK REFERENCE

- Three layers, all derived: the bar of sites (site map), the site menu (the pages' first folders), the breadcrumb and chapter links (path, course name, chapter numbers)

- `@lw/navigation`: `folderLabel`, `globalBar`, `NavIndex` (`menu`, `breadcrumbs`, `neighbours`, `listing`), `resolveLink`
- Course folders are labelled with the course's name; previous and next go by chapter *number*
- Root layout: frame, bar of sites, menu. Nested layout `[...path]/layout.tsx`: the breadcrumb, from `params`
- A layout is not redrawn when the visitor moves; only the menu is a client component, to mark the current subject with `usePathname`
- One route for pages and folder listings; a page wins over a folder of the same address; anything else 404
- `Link` inside the site (no reload); plain links to other sites
- Identical to Django on 8 menus, 4,417 prev/next, 4,417 breadcrumbs, 545 folder lists; three planted mistakes all caught
- Found in Chrome: the menu did not mark a subject's own page (trailing slash); fixed
- Node's type stripping refuses constructor parameter properties: declare fields on their own line

Learning Website with Next.js

Chapter 7 · The Languages Site

The earlier chapters built the machinery; this one finishes the first real site. The languages site needs a front page that shows each language and its courses in the order a learner meets them, a colour for each language that stays readable, and answer blocks that work. As before, the grouping is checked against the Django project on the real content, and the answer blocks are tested in a real browser.

RUN FOR REAL, ON 20 COURSES AND 4,417 PAGES

56 tests pass; the front page's data is identical to Django's; the site builds (601 pages, 17 s from cold); a headless Chrome opened and closed an answer block. Keyboard use, a screen reader and other browsers were not tried.

The Front Page: Grouped the Way a Learner Thinks

It is not an alphabetical list of folders. Each language has its courses in the order they are met:

GROUP	HOLDS	HOW A COURSE IS PLACED
Reading and writing	Alphabets and kana	The folder name contains alphabet, hiragana, katakana or kanji; ordered by that, then course number
Survival <Language>	Basic Conversation 1 to 3	The folder name contains <code>-basic-</code> and the course number is 3 or less
Everyday <Language>	Basic Conversation 4 to 6	The same, with a number above 3
Culture	Courses about the country	The folder is under <code>culture/</code> ; ordered by name
Other courses	Anything else with numbered chapters	Ordered by name

This is decided by plain functions in a package, `@lw/languages-site`, so it needs no browser to test. A new course appears on the front page when its folder first has a numbered chapter: there is no list to edit. The real site has 8 groups over 20 courses: French has Survival French; German

has Survival and Everyday German; Hungarian has Reading and writing and Survival Hungarian; Japanese has Reading and writing (5 courses), Survival Japanese and Culture (5 courses).

```
export function classify(course): { group; key } {
  const name = course.folder.slice(course.folder.lastIndexOf("/") + 1);
  if (course.folder.startsWith("culture/")) return { group: "Culture", key: [course.name] };
  const writing = WRITING.findIndex((word) => name.includes(word));
  if (writing >= 0) return { group: "Reading and writing", key: [writing, course.courseNo] };
  if (name.includes("-basic-")) {
    const number = course.courseNo ?? 0;
    return { group: number <= 3 ? "Survival" : "Everyday", key: [number] };
  }
  return { group: "Other courses", key: [course.name] };
}
```

Is It the Same as Django's?

A script in the Django project writes what its front page holds; the TypeScript version is asked the same question:

LANGUAGE	GROUPS	COURSES	LESSONS	COMPARED WITH DJANGO
French	1	1	3	identical
German	2	4	0	identical
Hungarian	2	4	12	identical
Japanese	3	11	300	identical

TWO PLANTED MISTAKES: ONE CAUGHT, ONE NOT

- **"Basic 3" treated as Everyday** (`<= 2` instead of `<= 3`): German and Hungarian came out *different*. Caught.
- **Sort changed to `localeCompare`** (the browser's language rules instead of code-point order): **0 different** on the real content. *Not* caught, because the five Japanese culture courses happen to sort the same either way. A *unit test* with chosen awkward names (`Banana`, `Zebra`, `apple`) does catch it (55 passed, 1 failed).

The takeaway: real data proves only the cases it contains. A test with deliberately awkward data covers the rest.

A Colour for Each Language, and Text You Can Read

Each language's section carries two CSS variables, `--lang-accent` (the border) and `--lang-text` (the heading text), set inline on the section. French is blue, German gold, Hungarian green and Japanese red, from their flags. One catch: Japanese's red (`#d64550`) is only 3.86 against the dark surface, and text needs 4.5. So Japanese keeps its true red for the border and uses a lighter red of the same hue (`#e8707a`) for its heading. A test keeps every language's text colour at 4.5 or more, reads the real surface colour out of `tokens.css`, and checks that Japanese's accent alone would fail, which is why the second colour exists.

The page's stylesheet is a CSS Module: only this page uses those classes, so by Chapter 5's rule they are not global. A screenshot in headless Chrome shows a coloured left border and name for each language, the group titles in small capitals, and each course with its chapter count.

Course and Chapter Pages

Nothing language-specific was needed. A course is a folder listing headed with the course's name, and a chapter has a breadcrumb and previous and next links, all from Chapter 6. The lessons themselves keep their own accent colour (green Hungarian, gold German, red Japanese, blue French) from their own scoped style, as Chapter 5 found, and their `lang` marks from Chapter 4.

Answer Blocks

Most language lessons end with a "Try It Yourself" block whose answers are hidden until asked for. In the files they are plain `<details>` elements with a `<summary>` ("Show answer"): no script, so nothing to re-run and nothing to break. 196 of the site's 601 pages contain them. That is a claim about a real browser, so it was tested in one:

GERMAN BASIC CONVERSATION 1, CHAPTER 1 (5 BLOCKS)	FIRST BLOCK
Before the click	closed, 24 px high
After one click on "Show answer"	open, 60 px high
After a second click	closed, 24 px high

A MISTAKE IN THE TEST ITSELF

What went wrong: my first measurement asked “is the answer's own element visible?” and reported “visible” whether the block was open or closed, because a closed `<details>` still reports a size for its content. **The takeaway:** a check that gives the same answer in both states checks nothing, so before trusting one, run it in a state where the answer is known to be “no”. It was replaced with the height of the whole block, which does change.

WHAT WAS NOT VERIFIED

Keyboard operation of the blocks (browsers open a `<details>` with Enter or Space on its summary, but that was not tried), a screen reader, other browsers, and the print layout. Only the languages site is complete; the others have a menu but no pages yet.

Hands-On Exercises

EXERCISE 1

Group the language courses the way a learner meets them (reading and writing, survival, everyday, culture, other), with readable text colours for every language. Prove the result against the Django project on the real content, and show with planted mistakes what the real data does and does not catch.

 [View solution](#)

EXERCISE 2

Build the languages front page as a server component with a CSS Module, a section and colour for each language, and a lessons line only where lessons exist. Build the site, fetch the page and check its sections, groups and colours.

 [View solution](#)

EXERCISE 3

Test in a real browser that a lesson's “Show answer” blocks open and close. Make sure your measurement gives different answers in the open and closed states, and say what you did not test.

CHAPTER 7 QUICK REFERENCE

- Front page: each language (French, German, Hungarian, Japanese), its courses in groups: Reading and writing, Survival, Everyday, Culture, Other courses
- Group by folder name and course number; sort numbers by number and names by code point (not `localeCompare`)
- Identical to Django: 4 sections, 8 groups, 20 courses; a "Basic 3 is Everyday" mistake was caught, a sort mistake was not (a unit test catches it)
- Two CSS variables per language section: `--lang-accent` (border) and `--lang-text` (heading); Japanese's text red is lighter for contrast 4.5
- The front page's classes are a CSS Module; the data comes from `groupCourses` at build time, so a new course appears by itself
- Course and chapter pages are Chapter 6's: a course is a listing headed with its name, a chapter has a breadcrumb and previous/next
- "Try It Yourself": plain `<details>`, no script; tested: closed 24 px, open 60 px, closed again
- A check that cannot tell open from closed checks nothing: run it in a state where the answer is "no"
- Not tried: keyboard, a screen reader, other browsers, print

Learning Website with Next.js

Chapter 8 · PDFs, Solutions & Assets

A page is not the only thing a visitor downloads. Courses have PDFs, chapters link to solution files, and 1,923 pages have a code block with a Copy button. This chapter sorts out how each is handled in Next.js, keeps the addresses that old pages already link to, and finds out whether the image and font tools that Next.js offers have anything to do here. Each part was checked on the real files, and the Copy button in a real browser.

RUN FOR REAL, ON 8,431 FILES AND 4,419 PAGES

63 tests pass. The assets of every site were counted, the languages site's were copied and served, the headers were read with `curl`, and a headless Chrome clicked the Copy button through 12 checks. Nothing was run behind Apache, and the **real clipboard was never written** (that would overwrite what you had copied).

Three Kinds of File

KIND	EXAMPLES	HOW IT IS HANDLED
Pages	4,419 content files	Made at build time (Chapter 4)
Assets	Course PDFs, solution <code>.txt</code> files (8,431)	Copied into the app's <code>public/</code> folder at the <i>same relative path</i>
The site's own files	Stylesheets, scripts (including the Copy button's)	Compiled and hashed by Next

Assets: Same Address, Copied In

A page links to a PDF or a solution at the *same relative path* as in the content folder: `/hungary/hungarian-basic-3/solutions/ex1.txt`. Those addresses must keep working after the split. Next serves everything in an app's `public/` folder as it is, at the same path, so each site's build **copies its own** PDFs and solutions there:

- **A site gets only its own files.** The site is decided from the path (the same function as for pages), and a file that belongs to no site is reported and never copied.
- **Only what changed is copied** (missing, or a different size or time), and the copy keeps the original's modified time, so a second run copies nothing.
- `--dry-run` shows what would happen; `--prune` removes only files that a *previous run* copied and that no longer exist.
- **PDFs are build artefacts.** The Python builders make them, git ignores them (`*.pdf`), and this build only *copies* the ones that exist. On a fresh checkout there are none until the builders have run.

REAL RUN	RESULT
Languages, first copy	236 files, 186 MiB, 1.6 s
The same again	0 copied, 236 unchanged, 0.3 s
Every site, dry run	Languages 236, web development 1,016, programming 3,658, systems 1,419, AI 317, humanities 889, life skills 412, creative 484: 8,431 files , none skipped (about 640 MiB of copies across eight apps)

Serving Them

REQUEST (CURL, RUNNING APP)	RESULT
A solution <code>.txt</code>	200, <code>text/plain; charset=UTF-8</code> , <code>ETag</code> , <code>Last-Modified</code> ; a repeat with the ETag gives 304
A PDF	200, <code>application/pdf</code> , <code>Accept-Ranges: bytes</code> ; a request for bytes 0–99 gives 206 Partial Content
A missing file under a solutions folder	404
<code>/solutions/../../../../package.json</code> and the <code>%2e%2e</code> form	404, 404

One header is added to the two folders: `X-Content-Type-Options: nosniff`, so a browser never guesses a file's type. To see why, a file containing `<script>alert(1)</script><h1>hi</h1>` was put in a solutions folder (and removed afterwards). It is served as `text/plain`, and Chrome showed it as *text* with the tags escaped: nothing ran.

THREE MISTAKES, AND HOW TO AVOID EACH

1. **A bookkeeping file inside `public/`.** My list of copied files (needed for `--prune`) was kept in the target folder, and `public/` is served to visitors. A request for it was answered, with an error (500), which is its own bad sign. It is now kept *next to* the folder, and a test checks that only content is in `public/`. *Whatever a folder serves, nothing in it should be for you only.*
2. **A wrong comment.** I wrote that Next sends no validators for files in `public/`, and added a one-hour cache rule. I had filtered the header list too tightly to see them: Next *does* send `ETag` and `Last-Modified`, and its default `max-age=0` means “ask again each time, cheaply”. The rule and the comment were removed. *Look at all the headers before explaining them.*
3. **A test that assumed a language.** My “only content is in public” test expected one folder and found two, because its own sample files included a French one and French belongs to the languages site. *Work out what a test expects from the same rule the code uses.*

Downloads on Course Pages, and Dead Links

A course folder's page now lists its PDFs, and its solution files folded away in a `<details>`, taken from what was really copied, so a PDF that does not exist cannot be linked.

`/hungary/hungarian-basic-3` shows a Downloads heading with its course PDF and chapter PDFs; `/hungary`, which has none of its own, shows no heading.

A second check goes the other way: every link to a `.pdf` or `.txt` in every page, against the files the builds would copy. **11,188 links, 8,431 files, 178 links to a file that does not exist:** 174 on the web development site (solution files for the scripting and back-end courses), 3 on creative, 1 on systems. It is the same number the Django project found, reached independently. It was *not* made to fail the build, because 178 known problems would stop every build until someone decides what to do about them.

Images and Fonts: Is There Anything to Optimise?

Next.js has `next/image` (resizes and converts images) and `next/font` (loads fonts without a layout jump). Whether to use them depends on what the pages really contain, so the fragments were surveyed:

FOUND IN THE 4,419 PAGES	COUNT
<code></code> tags	0
Image or font files in the content folder	0
Pages with <code>@font-face</code>	5 (all web development)
Pages loading Google Fonts	3 (2 web development, 1 systems)
Pages with <code>url()</code> in their CSS	40
Pages with a Copy button	1,923 (none on the languages site)

So `next/image` has **nothing to work on**: no page uses an image, and the old site's pictures live in its `resources/` folder (code samples and exercises), not in the content. The site's fonts are the system fonts (Chapter 5), and the few pages with their own fonts bring them themselves, so `next/font` would not replace those either. Neither is adopted. Revisit this when the first image is added to a page, and use `next/image` for it then.

The Copy Button as a Client Component

On 1,923 pages a code block is written as `<button class="copy-btn" onclick="copyCodeBlock(this)">`. The page body is inserted as text, so a script inside it would never define the function (Chapter 4). It is defined **once, for the whole site**, by a small client component in the layout. The button copies the code block right *after* it, using the browser's clipboard when it is available (HTTPS or localhost), the old hidden-text-box command when it is not, and saying **"Copy failed"** if both fail.

HEADLESS CHROME, ON THE TEST BENCH	RESULT
The page defines <code>copyCodeBlock</code> ; the button starts as "Copy"	pass
Modern clipboard: asked to copy exactly the code block's text; button says "Copied!"; back to "Copy" after 1.5 s	pass, pass, pass
The clipboard refuses: the old command is used, on the right text; "Copied!"	pass, pass
No clipboard at all: the old command is used	pass
Nothing works: the button says "Copy failed"; no code block after the button: "Copy failed", no crash	pass, pass

HEADLESS CHROME, ON THE TEST BENCH

RESULT

After a *link click*: the function still exists and copies the right text

pass, pass

WHAT THIS TEST DOES NOT DO, AND WHETHER IT CAN FAIL

It never writes the **real** system clipboard, on purpose: that would overwrite whatever you had copied. The browser's clipboard call and the old copy command are replaced by recorders, so the test sees which text is asked for, which route is taken and what the button says. The last step, pasting, has not been seen: please click a Copy button on a real page once. All 12 passed first time, which proves little, so two deliberate breakages were rebuilt and run: a button that always claims "Copied!" gave **11 passed, 1 failed**; copying the element *before* the button gave **6 passed, 6 failed**. *Be wary of a test until you have seen it fail for the right reason.*

WHAT WAS NOT VERIFIED

Apache is not in front, so none of the serving was checked behind a proxy (in production Apache can serve these folders directly: Chapter 12). The PDFs themselves were not rebuilt. The Copy button was tested in Chrome only, and never against the other 1,922 pages' real code blocks, only the test bench's.

Hands-On Exercises

EXERCISE 1

Copy each site's PDFs and solutions into its app at the same relative paths, copying only what changed and pruning only what you made. Serve them and check the types, ranges, cache validators and path tricks with curl. Record the mistakes you make.

 [View solution](#)

EXERCISE 2

List a course's downloads on its page from the files that really exist, check every PDF and solution link in every page, and survey the pages' images and fonts to decide whether `next/image` and `next/font` are worth using.

 [View solution](#)

EXERCISE 3

Make the Copy button work on every page with one client component, and test every route it can take in a real headless Chrome without touching the real clipboard. Show the test fails when the button lies.

 [View solution](#)

CHAPTER 8 QUICK REFERENCE

- Assets keep their **same relative path**: a site's build copies its own PDFs and solutions into the app's `public/`
- Only changed files are copied (size and time); `--prune` removes only what an earlier run copied; the list of copied files is kept *next to* `public/`, never in it
- 8,431 files in all (about 640 MiB of copies); languages 236 files, 186 MiB, 1.6 s the first time, 0.3 s after
- PDFs are built by the Python builders and ignored by git: the site build copies the ones that exist
- Next sends `ETag` and `Last-Modified` for `public/` files (304 on a repeat), range requests give 206; add `nosniff` for `/pdfs/` and `/solutions/`
- A course page lists its PDFs and (folded) its solutions, from the files that exist
- 11,188 links to PDFs and solutions checked: 178 point at missing files (174 web development), the same number as Django
- 0 images and 0 font files in the content: `next/image` and `next/font` not adopted
- Copy button: one client component in the layout defines `copyCodeBlock`; modern clipboard, then the old command, then "Copy failed"; 12 checks pass, and the test fails when the button lies
- Not done: Apache in front, rebuilding the PDFs, a real paste, other browsers

Learning Website with Next.js

Chapter 9 · Search, Sitemaps & Metadata

A static site has no server to ask, so search has to be a file, and a search engine has to be told, by files and tags, what the site contains. This chapter builds a search that runs in the visitor's browser, a sitemap, one canonical address for every page, and the Open Graph tags that describe a page when it is shared. As in earlier chapters, the results are compared with the Django project on the real content, and two of the differences turned out to be mistakes in the *content*.

RUN FOR REAL, ON 4,420 PAGES

76 tests pass; the text of all 4,420 pages is identical to what Django indexed; 35 searches find the same pages as Django's; the built languages site (561 pages) has a canonical address in its sitemap for every page; and a headless Chrome made 18 successful searches and clicks. The *order* of results is not identical to Django's, there are no highlighted snippets, and nothing was run behind Apache.

Search That Is a File

There is no search server and no database. When the site is built, an **index** of every page is written out as one JSON file, `/search-index.json`. The browser fetches it the first time somebody searches, then looks words up in it. Three small, tested jobs make it work:

1. **htmlToText:** the words a person would read on a page (tags, scripts and styles removed, character references decoded).
 2. **buildSearchIndex:** for every word, which pages it is on and how often.
 3. **search:** a query against the index.
- **Words** are lower-cased with their accents taken off, so "Szeretnék" and "szeretnek" are one word.
 - **Japanese** has no spaces, so it is stored as single characters, pairs and groups of three, and a query uses the biggest group that fits. (Taking accents off ㇿ would turn it into ㇿ, so Japanese is cut out first and never folded.)
 - **A search needs every word;** the last may be only the beginning of a word, so "polite cond" finds "conditional".

- **Typed text is never a query language.** It is split into units and looked up in tables, so no input can be an error or an injection.
- **Ranking** is BM25 (a word counts more when it is rare and frequent, less on a long page) with the title weighted ten to one.
- **The index is compact:** each page number is written as the difference from the one before, in base 36 (`3,7,1:2` is pages 3, 10 and 11, the last with the word twice).

The package has **two doors**, because a browser must never load code that reads files: the main one (used at build time) and `@lw/search/client` (prepare and search only), which the page imports.

Is It the Same as Django's?

Step 1: the text

The Django index stores each page's plain text. A fingerprint of each was dumped and compared with the text made here.

RUN	PAGES WITH IDENTICAL TEXT
First	4,413 of 4,420 (7 different)
After three fixes	4,420 of 4,420

The seven pages came from three causes:

- `</>` (a React fragment written in a code sample) stayed as text here; a parser ignores it.
- `⅖` (two fifths) was a real HTML name my decoder did not know. (Earlier a survey of every page found 57 names that pages use and the shared decoder did not know; it was filled in, and a lower-case fallback was removed because `Δ` and `δ` are different letters.)
- **Two pages have a mistake in their content:** a raw `<style> block` and a raw `<script src>` written in a sentence, where `<style>` was meant. An unclosed `<style>` or `<script>` swallows the *rest* of the page, in a browser and in Python's parser. So a browser hides the rest of both pages, and the Django index holds **524 of 54,617 characters** of the Build Tooling page. Here an unclosed one swallows the rest too, so the two agree; the real fix is in the two content files (it is in the review notes).

Step 2: the searches

35 queries were put to the Django search and to this one: words, accents, beginnings, Japanese of 1, 2, 3 and 5 characters, all eight sites, and injection-like text.

WHAT WAS COMPARED	FIRST VERSION	FINAL
The same <i>set</i> of pages	31 of 35	35 of 35
The same first three, in the same order	5 of 35 (a title-first rule)	23 of 35, which is 20 of the 32 queries that find anything (BM25)

The four differences in the sets were: “*ありが*” found one page too many (with *pairs* of characters a page that has the first pair in one place and the second in another counts, so **groups of three** were added to the index), and three queries that found the Build Tooling page, which Django could not (the swallowed text above). The order differs because SQLite and this code compute BM25 a little differently. I stopped there on purpose: matching another engine's order to the last result is not the goal; the same pages, in a sensible order, is.

SITE	PAGES	INDEX	GZIPPED
Languages	561	937 KB	320 KB (273 KB with brotli)
Web development	514	966 KB	338 KB
Programming	1,388	2.7 MB	937 KB
Systems	653	1.5 MB	513 KB
Humanities	707	1.5 MB	561 KB

A search takes a median of 0.04 ms (at worst 1.7 ms) once the index is loaded. It is fetched once per visit, and only when somebody searches.

TWO MISTAKES OF MY OWN, IN THE TESTS

I worked a base-36 number out in my head (“dx”; the real answer is “dl”): *compute expected values with a tool, not by hand*. And my test page for “pieces that are not together” had the exact characters written out, so it correctly matched: *build test data so that the thing it is meant to prove is really absent*. Both were mistakes in the *tests*, which the tests then found.

The Search Page

A client component reads `?q=`, fetches the index once, searches it, and shows 20 results at a time. A plain form in the header goes to it, so it works without any script. The page is marked `noindex` (a list of results is not something to find in a search engine). In a headless Chrome, **18 checks passed**:

- The same counts as the Django search (18 for "szeretnek" and for "Szeretnék", 1 for "brotchen", 10 for "polite cond", 25 for "ありがとう" and "ありがとう", 5 for "水", 0 for nonsense).
- An injection-like query is a search with no results. HTML typed into the box (an `<img onerror>`) is shown as text and nothing runs.
- "the" gives 348 results in pages of 20 (the last page shows 8; a page number past the end shows none, without a crash).
- The header box with accented text finds pages; clicking a result moves to it without a reload; the index is fetched once.
- From opening the page to seeing results: 64 to 136 ms after the first search, which took 439 ms because it loads the index.

FOUND ON THE WAY

The index is 995 KB on the wire here. `next start` compresses pages (a 107 KB page went out as 20 KB) but *not* this file, and a generated file has no `ETag`, so a browser would download it again on every visit. Two changes: `Cache-Control: public, max-age=3600` for the file (after a rebuild a visitor can see an older index for up to an hour), and compression left to Apache, which would send about 320 KB. **The Apache part is not tested** (Chapter 12). A Turbopack build warning ("dynamic filesystem access causes tracing of the whole project") came from Chapter 8's code that lists a folder's downloads; it only runs while the site is built, so it is marked to be ignored, and the warning is gone.

One Address for Every Page

The sitemap, the canonical link and the Open Graph address all come from **one function**, so they cannot disagree. **Decision:** the canonical form has *no trailing slash* (Next's way), where the Django project used one. Old addresses with a slash are redirected (308), so no link anyone made is lost, and each page has exactly one address:

```
/hungary/hungarian-basic-3/hungarian_basic_conversation_3_1/ 308 to /hungary/hungari
/hungary/hungarian-basic-3/ 308 to /hungary/hungari
/search/ to /search /sitemap.xml/ to /sitemap.xml
```

The built site was checked page by page (`check-seo.mjs` reads every built page):

CHECK ON THE 561 LANGUAGES PAGES	RESULT
Addresses in the sitemap (all different), with a <code>lastmod</code>	561, of which 250 have a date (the same 250 as Django; no date is invented)
An address with non-ASCII letters	percent-encoded: <code>hiragana_%E3%81%82</code>
The page's canonical link is in the sitemap	561 of 561
Sitemap addresses that are not a page	0
The description is identical to the one Django stored (the banner's Topic line, or the first 160 characters)	561 of 561
Open Graph complete (title, description, url equal to the canonical link, type article, site name, card)	561 of 561
<code>robots.txt</code> by default; the search page	<code>Disallow: /</code> (crawling is blocked until launch); <code>noindex</code>
Built for the real domain with crawling allowed	addresses begin <code>https://languages.osztromok.com</code> , all 561 still match; <code>Allow: /</code> and a <code>Sitemap:</code> line

Can the check fail? Because the sitemap and the canonical link normally use the same function, it takes a change to *one* of them to see it work: giving only the canonical link a trailing slash made all 561 pages fail. Note that `robots.txt` is decided when the site is *built* here (set `LW_ALLOW_CRAWLING=1` for the real build); the Django project decides when it is asked.

WHAT WAS NOT DONE

The results show a page's summary, **not the matching words highlighted**: the index holds no page text, to stay small (the Django search showed highlighted snippets). There is **no** `og:image`: no page has an image, so there is nothing to show (one picture per site would be your decision). Only the

languages app has search and a sitemap yet. The order of results is not Django's. Nothing was run behind Apache, and only Chrome was used.

Hands-On Exercises

EXERCISE 1

Build a search that is a file: turn pages into plain text, build an index of words (and Japanese characters), and search it in the browser without ever treating typed text as a query language. Test the awkward cases and rank by BM25 with the title weighted.

 [View solution](#)

EXERCISE 2

Prove the new search against the Django project's: first the plain text of every page, then 35 real queries. Explain every difference you find, including any that turn out to be errors in the content, and say what you chose not to match.

 [View solution](#)

EXERCISE 3

Add the search page, a sitemap, `robots.txt`, canonical links and Open Graph tags, all from one address function. Check every built page, build for the real domain, and drive the search in a real headless Chrome.

 [View solution](#)

CHAPTER 9 QUICK REFERENCE

- Search is a file: `/search-index.json` is made at build time and fetched once per visit; `@lw/search/client` is the only part the browser loads
- Words are lower-cased with accents removed; Japanese is stored as characters, pairs and groups of three; every word must match and the last may be a beginning

- Typed text is split and looked up, never run as a query language; ranking is BM25 with the title weighted ten to one
- The text is identical to Django's on 4,420 of 4,420 pages; 35 of 35 queries find the same pages; the first three agree on 20 of 32 non-empty queries
- An unclosed `<style>` or `<script>` in the text of a page swallows the rest (two real pages): fix the content
- Index sizes: languages 937 KB (320 KB gzipped); the largest site 2.7 MB (937 KB); search takes under 2 ms
- One function gives the address for the sitemap, the canonical link and Open Graph; no trailing slash, and old addresses with one redirect (308)
- 561 of 561 canonical links in the sitemap, descriptions identical to Django's, Open Graph complete; 250 pages with a `lastmod`
- `robots.txt` blocks all until `LW_ALLOW_CRAWLING=1` at build time; the search page is `noindex`
- Not done: highlighted snippets, `og:image`, compression of the index (Apache), other sites, other browsers

Learning Website with Next.js

Chapter 10 · Accounts & Progress (Optional)

Everything so far is made in advance and is the same for everybody. Remembering *who someone is* and *what they have finished* needs a program that runs when asked, and a database. This chapter adds exactly that, as little of it as possible: four small routes, one SQLite file, and no change to the pages themselves. The interesting part is not the code but what is locked, how each lock was tested, and what it costs.

RUN FOR REAL, ON NEXT.JS 16.4 AND NODE 24

90 tests pass; the login and progress API was tested over real HTTP (35 checks); the login pages were used in a headless Chrome (16 checks); and four planted mistakes were each caught. Not tested: HTTPS (so the `Secure` cookie flag), Apache, a second server, other browsers, a screen reader.

Do You Need This at All?

OPTION	COST	USED
Keep progress in the browser	Nothing to run; but per browser, and lost with it	No
Our own small API and SQLite	A Node process; the security is our job	Yes
A library such as Auth.js	A large dependency, and still a database	No
A hosted login service	Another company holds the passwords	No

The Accounts Package

The decisions live in a package of plain functions with no web framework (`@lw/accounts`), so they can be tested hard without a browser. The database is SQLite through Node's own built-in `node:sqlite` : nothing to install. (In Node 24 it is still marked experimental and prints a warning when it starts: its interface could change.)

- **No sign-up.** The owner makes accounts with a small tool, so there is no registration form to attack or fill with junk. The password is read from the environment, not typed on the

command line, so it is not left in the shell history.

- **A password is never stored**, only a script hash with its own random salt. One check takes about 29 ms here: script is slow on purpose, and so is guessing.
- **A login token is 32 random bytes, and the database holds only a hash of it**, so a copied database cannot log in as anyone.
- **“Wrong password” and “no such user” look the same**: the same words, and about the same time (a name that does not exist is checked against a decoy hash). Otherwise the words or the delay would tell an attacker which names are real.
- **Five wrong passwords lock a name for 15 minutes**, even for the right password; names that do not exist count too. The cost: anyone who knows your user name can lock you out for 15 minutes.
- **Sessions last 30 days and are checked against the server each time**, so logging out really ends them.
- **Progress is stored by page path**, so a rebuild never loses it (renaming a page file would). Deleting a user deletes their sessions and progress.

A WEAKNESS IN MY FIRST TESTS, FOUND BY CHANGING THE CODE

I changed the number of allowed failures from 5 to 50 and **every test still passed**, because the tests imported the constant and so followed it. *A test that follows a constant cannot notice the constant being changed.* I added a test that writes the policy out in plain numbers (4 wrong passwords: not locked; the 5th: locked; 899 s later still locked; 900 s later free). With it, the same change fails the suite. A second planted change (sessions never expiring) was caught by an existing test.

Four Routes: the Only Dynamic Part

`/api/login`, `/api/logout`, `/api/me` and `/api/progress` are marked *force-dynamic*: they run when asked, never at build time, never cached. Every POST goes through the same locks:

LOCK	WHAT IT STOPS
The request must come from <i>our own</i> pages (the <code>Origin</code> header must equal our host)	Another site making a visitor's browser send a request with their cookie. Even another of <i>our</i> sites is refused: each site has its own login
The request must say it is JSON	A plain form posted from another site

LOCK	WHAT IT STOPS
Cookie: HttpOnly, SameSite=Lax, no Domain, 30 days, Secure in production	Script reading the cookie; another site sending it; it leaking to other subdomains; plain http in production
Body read as text and limited to 4 KB; types checked	Huge bodies; a user name that is a list rather than text
A progress path must be a real page of <i>this</i> site	Made-up paths, <code>../../etc/passwd</code> , other sites' pages
Answers about a person: <code>no-store</code> , depending on the cookie	A shared cache showing one person's answer to another

35 Checks Over Real HTTP

A script made up passwords for the run (kept in memory, never printed), started against the running app, and checked, among others: wrong password and unknown name give the same `401` and the same words and no cookie; a login with no Origin, from another site, or from another of our sites gets `403`; a form post gets `415`; GET on login gets `405`; after 5 wrong passwords the right one gets `429` with `Retry-After: 900` while another name is unaffected; the cookie has the right flags and the token is not in the body; marking a page twice is one; four kinds of bad path are `404`; another user can neither see nor undo it; logging out clears the cookie and the *old* cookie stops working at once; and a made-up cookie is just "nobody". **35 passed.** Taking the same-origin check out and rebuilding gave **31 passed, 4 failed.**

The Pages Stay Static

A lesson page requested *with* a valid login cookie is, byte for byte, the page everyone gets: 106,396 bytes, no user name, no "Mark as finished", still saying "Log in", cacheable, and setting no cookie. That matters: if a page could differ by user, a shared cache could show one person's page to another. Everything personal arrives afterwards from `/api/me`, which is never cached. So the page is sent logged-out, and the browser fills in the rest:

ROUTE KIND	WHAT
○ static	The front page, <code>/login</code> , <code>/progress</code> , <code>/search</code> , <code>/sitemap.xml</code> , <code>/robots.txt</code> , <code>/search-index.json</code>

ROUTE	KIND	WHAT
• static, from a list		Every page and folder page of the languages site: 605 HTML files in all
f dynamic		<code>/api/login</code> , <code>/api/logout</code> , <code>/api/me</code> , <code>/api/progress</code>

The content is still built in advance, but the site is no longer just files: it needs a Node process (`next start`) for four routes. Two ways to deploy it, for Chapter 12: the whole app as a Node server (what was tested), or the pages as plain files with only the accounts as a tiny separate server behind Apache's `ProxyPass` for `/api` (possible, because the accounts package has no web framework in it; not built).

In a Real Browser

A headless Chrome typed a made-up password into the real form. **16 checks passed:**

- Logged out, the header says “Log in” and the link remembers the page to come back to; a wrong password shows “Wrong user name or password.” and stays.
- The right password returns to the lesson. The header then shows the user's name and “Log out”, **77 ms** after the page loaded; “Mark as finished” appears.
- Clicking it says “You have finished this page.”; after a *full reload* it still does (kept on the server).
- `/progress` shows “Hungarian Basic Conversation 3: 1 of 12” with a progress bar; “Mark as not finished” undoes it without a reload.
- Logging out brings back “Log in” and removes the button; logged out, `/progress` asks you to log in.

The 77 ms is the **flash**: a logged-in visitor briefly sees “Log in” and no button before the browser has asked who they are. A page that cannot be personalised in advance has to do this; logged-out visitors, who are most people, see nothing change.

AN OPEN REDIRECT, AND A PLANTED MISTAKE THAT DID NOTHING

After logging in the visitor is sent to the address in `?next=`. If that could be anything, a link such as `/login?next=//evil.example` would send a freshly logged-in visitor to a site that looks the same. Only addresses inside the site are followed (`//evil.example`, `https://...`, `javascript:...` and backslash tricks all give `/`); a unit test lists them. **My first attempt to plant this mistake did nothing:** my edit

did not match the code, so the code was unchanged and the test passed. I noticed because the count of changed lines was 0. Done properly, the test failed, with the browser heading for `//evil.example`. *Confirm that a planted mistake is really in the code before believing the test that passes.*

WHAT WAS NOT VERIFIED, AND WHAT TO KNOW

HTTPS (the `Secure` flag is added only by a production build and was not seen on a real response); a second server sharing the database (SQLite handles several processes on one machine, not a network drive); other browsers; a screen reader; a password manager. **For Apache (Chapter 12):** the Origin check compares `Origin` with the `Host` header, so Apache must pass the original Host on (`ProxyPreserveHost On`) or every login will be refused with 403. There is no password change or reset (no email is set up: the owner deletes and re-makes the user), and the other sites have no accounts yet.

Hands-On Exercises

EXERCISE 1

Write accounts as a small tested package: hashed passwords, session tokens stored only as hashes, an identical answer for wrong password and unknown name, a logout, and progress that cannot leak between users. Prove each rule by changing the code and watching the tests fail.

 [View solution](#)

EXERCISE 2

Build the four API routes with an origin check, a JSON check, safe cookie flags and strict input checks, then test every lock over real HTTP and show that removing one makes the tests fail. Show that a page requested by a logged-in browser is still the same for everyone.

 [View solution](#)

EXERCISE 3

Add the login page, a header that shows who is logged in, a Mark as finished button and a progress page, all filled in by the browser after the static page loads. Close the open redirect, test the whole thing in a real browser, and list what is and is not static.

CHAPTER 10 QUICK REFERENCE

- Own small API + SQLite (`node:sqlite` , built in, experimental in Node 24); no sign-up; accounts made by the owner with `accounts-admin.mjs` (password from the environment)
- Passwords: script hash with a random salt (about 29 ms a check); session token: 32 random bytes, only its hash stored; sessions 30 days, checked on the server
- Wrong password and unknown name: same answer, same words, similar time; 5 wrong passwords lock a name for 15 minutes (the cost: anyone can lock you out)
- Every POST: same `Origin` as our host (403), JSON (415), body under 4 KB, types checked; cookie `HttpOnly`, `SameSite=Lax`, no Domain, `Secure` in production
- Progress by page path, only for real pages of this site (404 otherwise); another user cannot see or undo it
- Pages stay static and identical for everyone; the browser asks `/api/me` (`no-store`) afterwards: a logged-in visitor sees a flash of about 77 ms
- Only `?next=` addresses inside the site are followed (an open redirect otherwise)
- A test that follows a constant cannot notice the constant changing: write the policy out in plain numbers; and check that a planted mistake is really in the code
- 35 HTTP checks (31 pass, 4 fail when the origin check is removed), 16 browser checks, 90 tests
- Needs a Node server for four routes; Apache must keep the Host header; not tested: HTTPS, Apache, other browsers; no password reset

Learning Website with Next.js

Chapter 11 · Migrating from the Old Site

The old site is live and has bookmarks, search results and links from elsewhere. A migration that just switches it off turns every one of them into a 404. This chapter does what the Django course's chapter on migration did, in TypeScript and in Next.js's own configuration: measure exactly what the old site answers, suggest redirects for a person to approve, keep them in the app's configuration, check the links, and move one site at a time. Because the Django project has done the same measurements, every number here can be compared with an independent implementation.

RUN FOR REAL, ON THE OLD SITE AND 4,422 PAGES

100 tests pass. The old site's folder (`debserver/website`) was read, 9,000 of its addresses were checked, and the languages app was asked about 329 of them over HTTP. **The Apache lines were not run**, and the redirect suggestions have **not been reviewed by a person**: the lists that are committed are empty.

Step 1: Measure, Don't Remember

The old site is a folder of built files (`/hungary/x/y/` is the file `hungary/x/y/index.html`), so that folder is the complete list of old addresses. A check asks, of each one, whether the new sites have it, *without starting a server*: from their pages, the folders those pages sit in (each has a listing), the files that will be copied, and the decided redirects. An address is **same, redirected** (a decided redirect that ends on something that exists) or **missing**. A redirect to a page that does not exist is *not* success.

OLD SITE	COUNT	BEFORE ANY REDIRECT
Pages	3,205	3,102 same (of the 3,182 a site owns), 80 missing
PDFs and solution files	5,831	5,544 in place, 166 missing
Owned by no site	144	Mostly <code>resources/php</code> , <code>resources/js</code> , <code>resources/csset</code> , the old front page and <code>/admin/</code>

OLD SITE	COUNT	BEFORE ANY REDIRECT
Dynamic pages (.php)	23	The Anime Vault: not migrated

So most of the site already works at the same address, because the folder structure was kept. Reading the 80 missing pages: **32** are lessons whose folder was renamed or moved, **23** are `_print` pages (the old site had a printable twin of many pages), and **25** are the index pages of renamed folders, removed "tentative" course folders, and `japan/kanji-tiles`. These are *exactly the numbers the Django project found*, from a separate implementation.

Step 2: Redirects You Review First

A permanent redirect is remembered hard by browsers, so a *wrong* one is hard to undo. So: suggest, then review, then build.

- Suggest.** If exactly *one* new page has the same file name (ignoring capitals and `-` against `_`), propose it; if several do, list them; if none, say so; a name ending in `_print` falls back to the page without it. Result: **55 suggestions, 0 ambiguous, 25 with nothing to suggest**, and all 55 point where the Django project's do.
- Review.** The suggestions go in a file whose *decision* column is empty. Only rows where you write `yes` become redirects. Loading the file before any review gives *0 redirects (55 rows skipped)*, so **the lists that are committed are empty**.
- Build.** `next.config.ts` reads the site's list and Next answers the redirects itself. A redirect's source is a *pattern* in Next, so the characters `( ) : * + ? { }` in an old address are made plain. A page that moved to *another* site gets a full address.

```
async redirects() {
  const file = join(process.env["LW_REDIRECTS_DIR"] ?? join(process.cwd(), "..", "..", "l"),
    return existsSync(file) ? nextRedirects(JSON.parse(readFileSync(file, "utf8"))) as Redi
},
```

Over Real HTTP

To see the mechanism work, I made a *scratch* list with every suggestion accepted (`--accept-all`), which prints "NOT reviewed": 19 for languages, 8 programming, 11 systems, 11 AI, 6

humanities; not committed), built the languages app with it, and asked the running app about **every old address the languages site owns**:

ASKED OVER HTTP	SAME	REDIRECTED	MISSING
319 old pages	297	19	3 (<code>france/french-language</code> , <code>hungary/hungarian-language</code> , <code>japan/kanji-tiles</code>)
10 old files	4	0	6 (in <code>resources/hungarian</code> and the old <code>pdfs</code> folder)

The HTTP answers and the check from Step 1 **agree on all 329 addresses**. All redirects are `308`. Note that an old link *with* a trailing slash takes **two hops**: first Next's own tidy-up to the address without the slash, then the real redirect. It works and each hop is cheap, but it is not one.

CAN THE SUGGESTER FAIL?

Taking out the print-version rule and running the report again gave "32 suggestions, compared with the Django project's 55: same 32, **only there 23**". Put back.

Step 3: Links

Broken links

Every absolute internal link of every page is checked against the pages, folders, files and redirects: **11,751 links, 483 broken** (305 to pages, 178 to files), in a few seconds (not timed precisely). That is the Django project's result, category by category. 282 of the page links are to the hiragana and katakana *tile* pages, which exist on the old site and not yet on the new one; 174 of the file links are solution files under `/web-development/scripting-and-backend` that were moved or never made. It is *not* set to fail the build: 483 known problems would stop every build until each is decided.

Links between sites

A page written for the old single site that links to `/linux/...` breaks after the split, because `/linux/` is on the systems site. The fix is made **when the page is built**, not in the files: a link into another site's folders gets that site's address in front. External links, `//` links, anchors, `/search/` and the site's own links are left alone. On the real content this found **0 links to**

rewrite: every page links only inside its own site. So the function is tested only on made-up examples, and a function that has never met real data deserves suspicion.

Step 4: One Site at a Time

While the old site still answers on `osztromok.com`, a site moves by telling the *old* host's Apache to send that site's folders to its new subdomain. To go back, remove the line.

```
RedirectMatch 301 ^/(france|germany|hungary|japan|culture|resources/japanese|resources/h
```

Without Apache, the same pattern was run as a regular expression over every old address: of **8,892** that a site owns, none was matched by the wrong site or by two, and none of the **144** that no site owns was matched by any rule (the Django project's numbers).

WHAT THE REAL DATA CAN AND CANNOT PROVE

I planted a mistake: the pattern no longer needed a slash or the end after the folder, so `/linuxfoo/` would match the systems rule. On the *real* addresses the result was **0 wrong matches**, because no real address looks like that; a *unit test* with that awkward address failed, as it should. The real addresses show the rules cover what exists; only an awkward test shows they do not cover too much. Both are needed.

WHAT WAS NOT VERIFIED

The Apache lines were never run: use `apachectl configtest`, then `curl -I` on one address of that site expecting a 301 and the right `Location`. The 55 suggestions are unreviewed. Only the languages site has an app, so only its redirects were tried over HTTP. The tile pages and the 25 unmatched pages were not built or decided. One of my own tests was wrong, not the code: I expected 11 links and the page set had 10 (I had counted by eye).

Hands-On Exercises

EXERCISE 1

Write the check that reads the old site's folder and says, for every address, whether the new sites have it (same, redirected or missing). Run it on the real old site and explain what the missing ones turned out

to be.

 [View solution](#)

EXERCISE 2

Suggest redirects for the missing pages into a file for review, load only the rows marked yes into the app's configuration, and prove over real HTTP what the languages site does with every old address it owns. Keep the committed list empty until a person has reviewed it.

 [View solution](#)

EXERCISE 3

Check every internal link of every page, rewrite links into other sites when a page is built, and generate the per-site Apache cut-over lines. Test the lines against every real old address and against an awkward one, and say what each of the two can prove.

 [View solution](#)

CHAPTER 11 QUICK REFERENCE

- The old site is a folder of built files: read it to get the real list of addresses; 3,205 pages, 5,831 PDFs and solutions, 23 `.php`, 144 owned by no site
- Same / redirected / missing, worked out without a server; a redirect to nothing is not success; 3,102 same, 80 missing before; 55 redirected and 25 missing after the suggestions
- Suggest (one page with that file name, or a print twin), review (decision column), build: only rows marked `yes` become redirects; the committed lists are empty
- Redirects live in `next.config.ts` (`redirects()`), read from `redirects/<site>.json`; special characters in an old address are escaped; another site gets a full address
- Over HTTP: 319 pages and 10 files asked; 19 redirected (308); the HTTP and logic checks agree on all 329; an old link with a slash takes two hops
- Link check: 11,751 links, 483 broken (305 pages, 178 files), the same as Django; 0 links point into another site
- Links into another site are rewritten when the page is built, not in the files

- Apache cut-over: one `RedirectMatch 301` line per site; 8,892 old addresses matched by exactly their own site, 0 of the 144 unowned matched; not run in Apache
- Real data proves rules cover what exists; an awkward test proves they do not cover too much
- Not done: Apache, reviewing the 55 suggestions, the tile pages, the other sites' apps

Learning Website with Next.js

Chapter 12 · Deployment, Testing & Operations

The last chapter is about the day after the code works: how to know a release is healthy before anyone sees it, how to get the accounts back after a mistake, how Apache and systemd run the sites, and how to undo a release. As in the other chapters, what could be run was run, and what could not is said plainly.

RUN FOR REAL, AND NOT RUN

Run: the production-settings check, the languages app built with `LW_ENV=prod` and started with `next start` and looked at over HTTP, a backup and a restore of the real accounts database, the generated Apache and systemd files, and 114 tests. **Not run:** Apache, systemd, cron, certificates, and `deploy.sh` (Windows has none of them; the script has only had `bash -n`).

One Node Process Per Site, Behind Apache

There were two shapes to choose from: a **static export** (just files, any web server) or a **Node server** (`next start`). The login API, the health address and the per-request checks need a server, so each site runs as its own Node process listening on `127.0.0.1` only, on the port it already has (portfolio 3000, languages 3001, and so on). Apache handles HTTPS and passes each host name to its port. A static export of the pages alone stays possible later if the accounts move elsewhere.

Both the virtual hosts and the systemd units are **generated from the site map** (`node ops.mjs config`), so they cannot disagree with it. Only sites that have an app get a virtual host; a site without one keeps answering from the old site until its turn.

```
<VirtualHost *:443>
    ServerName languages.osztromok.com
    SSLEngine on
    ...
    AddOutputFilterByType DEFLATE application/json application/xml text/plain

    ProxyPreserveHost On
    RequestHeader set X-Forwarded-Proto "https"
    ProxyPass          / http://127.0.0.1:3001/
```

```
ProxyPassReverse / http://127.0.0.1:3001/  
</VirtualHost>
```

THE LINE THAT FAILS QUIETLY

`ProxyPreserveHost On`. The login check compares the browser's `Origin` with the `Host` the app receives. Without that line the app sees `127.0.0.1:3001`, every login is refused as cross-site, and every page still works, so nobody notices. The smoke test below looks at exactly this.

Compression, Measured

Next compresses its pages: a lesson page was 59,774 bytes plain and 12,406 bytes with gzip. What a *route handler* sends is not compressed: asked with `Accept-Encoding: gzip`, `/search-index.json` came back as 1,018,766 bytes, exactly the same as without, and so did the sitemap. Compressed with gzip it would be 326,259 bytes. So the virtual host asks Apache to compress JSON, XML and text. **Untested:** I could not run Apache, so I have not seen this work, nor seen it leave alone a response Next has already compressed (that is its documented behaviour).

Before the Release: Settings

`node ops.mjs check` stops the release on mistakes that would otherwise show up as a strange site: `LW_ENV` not `prod` (every link would point at localhost), no content folder, no data folder or one *inside* the release folder (the next release would not see the accounts), and the experiment pages switched on. With a good environment it says *production settings: ok*. With development settings, no data folder and the experiment pages on, it printed three errors and exited with 1.

While It Runs: Health and a Smoke Test

`/healthz` answers 200 only when the database answers *and* the site has pages (`{ "status": "ok", "pages": 561 }`), is never cached, and says nothing else. The **smoke test** then visits the running site as a visitor would, with the `Host` header Apache will send, and checks 11 things: the health address; the first, the middle and the last page of the site; that a missing address answers 404 and not 200; that a real PDF is served with `nosniff`; robots, sitemap and

search index; and two logins, one from another site's page (must be 403) and one from our own (must get past the lock, 400). Against the running languages app: **11 checks passed**. With the app stopped, all 11 failed with `ECONNREFUSED` instead of hanging.

THREE MISTAKES IN MY OWN SCRIPT THAT ONLY THE REAL RUN FOUND

1. **Trailing slash.** I asked for pages as `/x/`; the site answers with a 308 to `/x`. Three checks failed with "308". *Ask for the address the site really uses.*
2. **The Host header.** Node's `fetch()` does not send the `Host` you give it, so the own-origin login got 403 instead of 400. `curl` with the same header got 400. The script uses `node:http`. Notice that the check for a login from *another* site had passed all along, for the wrong reason.
3. **A non-ASCII address.** One sampled page has `ℓ` in its address and `node:http` refuses it unescaped, so addresses are percent-encoded, as a browser does.

The unit tests, which use a pretend site, could not have found any of the three.

Backups You Have Restored

The only data that cannot be rebuilt from the content folder is the accounts database. A plain file copy is the wrong tool: the database runs in WAL mode, so recent writes sit in `accounts.sqlite-wal`, and a copy made mid-write can be damaged. `VACUUM INTO` asks SQLite itself for a complete, consistent copy while the site runs. The order matters: the copy is made under a temporary name, **checked**, only then renamed, and only after that are backups beyond the newest 14 removed, so a broken backup can never push out a good one.

A **restore** checks the backup before touching anything, keeps the database it replaces (`accounts.sqlite.before-restore`), and moves the old `-wal` and `-shm` files away with it, because they belong to the old database. Run for real: two accounts, a backup, a third account, a restore; the third was gone and the three-account database was kept. A file that is not a database was refused: *refusing to restore bad.sqlite: file is not a database*, and the accounts were unchanged.

WHAT A TEST CANNOT PROVE, AND A TEST THAT PROVED NOTHING

- (1) Removing the check on the *new* backup still fails a test, but only because that test also counts rows; a backup that opens but is damaged inside is not something any test makes, so that check is not proved. (2) My first restore demonstration used a third account called "cy"; the site rightly refuses

names under three letters, so nothing changed after the backup and the restore had nothing to undo. A test that cannot fail is not a test: I redid it with a valid name.

A Release With a Way Back

`deploy.sh release <ref>` builds a new folder next to the old one, then: **1.** backs up the database; **2.** checks the settings; **3.** installs, tests and builds; **4.** switches the `current` symlink (atomic) and restarts the services; **5.** runs the smoke test against every running site, and if one check fails, rolls back by itself. A rollback is moving the symlink and restarting. There are no migrations (tables are only created if missing), so old code normally still works with the database; if a release changed a table, restore the backup taken at its start.

NOT RUN — AND WHAT I WOULD EXPECT TO GO WRONG FIRST

Apache (no virtual host, no `mod_deflate`, no certificate); systemd; cron; `deploy.sh` (only `bash -n`); a real rollback. The unit sets `ProtectSystem=strict` with only the data folder writable, and Next may want to write to its own `.next/cache`, so the first start on the server may fail until that folder is added to `ReadWritePaths`. Only the languages app exists, so the other seven sites have no unit and no virtual host yet.

A first rollout that cannot hurt

1. Install on the server and run the settings check.
2. Start the languages unit on its port and run the smoke test against it directly (`curl -H "Host: ..."`).
3. Only then add its virtual host (`apachectl configtest`, then reload) and run the smoke test over HTTPS from outside.
4. Add the `RedirectMatch` line for its folders to the *old* site (Chapter 11), and watch the logs for 404s.
5. Keep the old site's lines in place for a few weeks, so going back is removing one line.

Hands-On Exercises

EXERCISE 1

Write the production-settings check, a health address and a smoke test that visits a running site with a given Host header. Run them against the built app, and say what each of the mistakes you hit along the way was.

 [View solution](#)

EXERCISE 2

Back up the accounts database while the site is running, verify the copy before trusting it, prune old backups safely, and restore one. Show with a real run that an account made after the backup is gone after the restore.

 [View solution](#)

EXERCISE 3

Generate the Apache virtual hosts and systemd units from the site map, measure what is and is not compressed, and write a release script with an automatic rollback. List exactly what you could not run.

 [View solution](#)

CHAPTER 12 QUICK REFERENCE

- One Node process per site (`next start -p PORT -H 127.0.0.1`); Apache does HTTPS and passes the host on; config and units generated from the site map
- `ProxyPreserveHost On` is required, or every login is refused as cross-site while pages still work
- Next compresses pages but not route-handler responses: search index 1,018,766 bytes either way; Apache `DEFLATE` added (untested)
- `node ops.mjs check`: production settings; `/healthz`: 200 only with a database and pages, never cached
- `node ops.mjs smoke <url> <host>`: 11 checks as a visitor; use `node:http` (fetch drops the Host header), no trailing slash, encode non-ASCII addresses
- Back up with `VACUUM INTO`, verify, rename, then prune; never overwrite; restore verifies first and keeps the old database
- `deploy.sh`: backup, check, build, switch symlink, restart, smoke test, automatic rollback; rollback = move the symlink back

- Not run: Apache, systemd, cron, certificates, `deploy.sh`; likely first failure: the unit's write permission for `.next/cache`

Course Complete

You have built a Next.js workspace that serves several sites from shared packages, with the tools to move the old site across without breaking its addresses. What each chapter left you able to do:

CHAPTER	YOU CAN NOW...
1 · Monorepo for Many Sites	Lay out several apps and shared packages in one npm workspace
2 · Subdomains & Middleware	Choose the site from the host name, strictly, in <code>proxy.ts</code> or as separate apps
3 · The Content Model in TypeScript	Parse each file's banner into typed pages, shared by every site
4 · Reading Fragments at Build Time	Generate every page statically and inject fragments safely, with their scripts running once
5 · The Shared Design System	Share one theme with a per-site accent, keeping fragment CSS apart from module CSS
6 · Layout & Navigation	Derive menus, breadcrumbs and links between sites from the data
7 · The Languages Site	Give one site its own front page, accent colour and answer blocks
8 · PDFs, Solutions & Assets	Serve downloads as build artefacts with safe headers, and add a Copy button
9 · Search, Sitemaps & Metadata	Add a static client-side search index, sitemaps and canonical metadata
10 · Accounts & Progress (Optional)	Add logins and progress without breaking the static pages
11 · Migrating from the Old Site	Measure old addresses, review redirects and cut over one site at a time
12 · Deployment, Testing & Operations	Check settings, monitor, back up, restore and release with a way back

WHAT IS STILL YOURS TO DO

Review the 55 suggested redirects; decide the 25 unmatched pages; run the Apache and systemd files on a real server (this course only generated and read them); build apps for the other six sites from the same packages. Then compare this course with *Learning Website with Django*: both end at the same numbers on the same content, and that agreement is what makes the choice between them one of preference, not of what works.