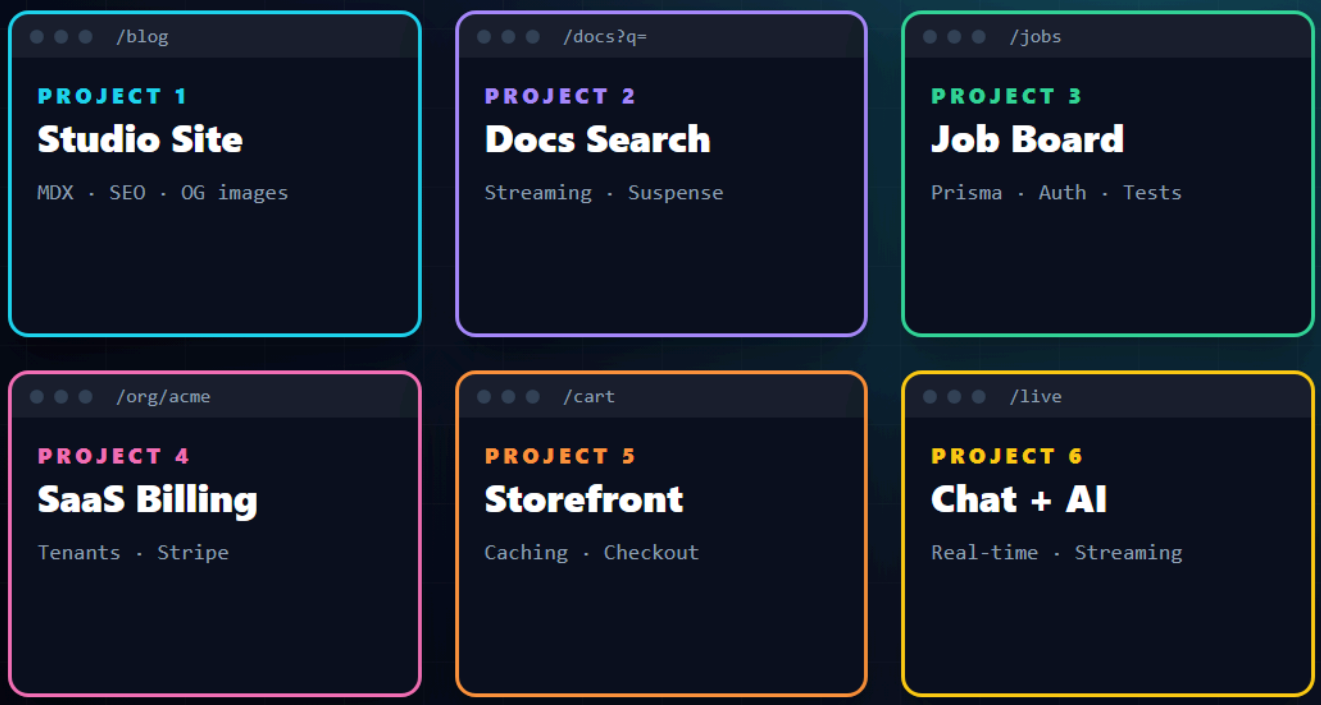


app/ · layout.tsx · page.tsx · actions.ts · route.ts

LEARN BY BUILDING

Next.js Project-Based Guide

Build six real-world apps with the App Router, Server Components, and Server Actions

TypeScript · Tailwind CSS · PostgreSQL + Prisma · Auth.js
Stripe · Playwright · Vercel and Docker

SITE · DOCS · JOBS · SAAS · STORE · CHAT + AI

THENEXBOOKS

Next.js
Project-Based Guide

Build Six Real-World Apps with the App Router, Server Components, and Server Actions

First Edition

THENEXBOOKS

Next.js Project-Based Guide
Build Six Real-World Apps with the App Router, Server Components, and Server Actions

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. Next.js, React, and the libraries and services used in this book change quickly; where behavior differs between versions, this book says so, and you should consult the official documentation for the versions you use. Any performance figures shown are illustrative.

Next.js and Vercel are trademarks of Vercel, Inc. React is a trademark of Meta Platforms, Inc. TypeScript is a trademark of Microsoft Corporation. Tailwind CSS, Prisma, PostgreSQL, Stripe, Docker, and other product names are trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement. This book is not affiliated with, sponsored by, or endorsed by Vercel, Inc. or Meta Platforms, Inc.

Driftwood Labs and all companies, products, and people in its projects are fictional. Any resemblance to real companies or products is coincidental.

Preface

The fastest way to learn a framework is to build something real with it — and then build something harder. Tutorials that stop at a to-do list leave out almost everything that makes real projects difficult: content that must rank in search, data that must stay consistent, users who must only see their own records, payments that must never be charged twice, pages that must stay fast as the catalog grows, and apps that must keep running after you ship them.

Next.js has become one of the most popular ways to build React applications for production. Its App Router, Server Components, and Server Actions let you put data fetching, rendering, and mutations exactly where they belong — but they also introduce new ideas about where code runs, what gets cached, and how the server and the browser share work. Those ideas click when you use them to solve a concrete problem, not when you read a list of features.

So this book is built around projects. You will build six applications for a fictional product studio, each one harder than the last, and learn every concept at the moment a project needs it: static content and SEO for a marketing site, streaming search for a knowledge base, a database, authentication, and authorization for a job board, multi-tenancy and billing for a SaaS, caching and checkout for a store, and real-time updates and an AI assistant for a team chat app. We explain *why* each decision is made, not just *what* to type.

Who This Book Is For

- **React developers** who want to move from single-page apps to full-stack Next.js.
- **Frontend developers** who want to own the backend parts of their features — data, auth, payments.
- **Backend developers** who want a modern, productive way to build web interfaces on top of their skills.
- **Freelancers and indie builders** who ship complete products and want patterns they can reuse.

You should be comfortable with JavaScript or TypeScript and know React basics — components, props, state, and hooks. Everything else is taught along the way.

What You Will Build

Project	What it is	Chapters
1 · Studio Site	Marketing site with an MDX blog	2–5
2 · Driftwood Docs	Searchable knowledge base	6–8
3 · Driftwood Jobs	Job board with auth and a database	9–15
4 · Ledgerly	Multi-tenant invoicing SaaS	16–20

5 · Driftwood Supply	Storefront with checkout	21–23
6 · Driftwood Live	Real-time chat with an AI assistant	24–25

Chapters 26–28 deploy, monitor, maintain, and review the whole portfolio.

What You Will Learn

- The App Router: layouts, routing, dynamic segments, route groups, parallel and intercepting routes.
- Server and Client Components, data fetching, caching, revalidation, streaming, and Suspense.
- Server Actions, forms, validation with Zod, optimistic UI, and Route Handlers.
- PostgreSQL with Prisma, authentication with Auth.js, and role- and tenant-based authorization.
- File uploads, email, background and scheduled work, and work after the response.
- Stripe subscriptions and checkout with webhooks, rate limiting, audit logs, and security hardening.
- SEO, metadata, Open Graph images, images and fonts, and Core Web Vitals.
- Real-time features and an AI assistant with streaming, tools, and guardrails.
- Testing with Vitest, React Testing Library, and Playwright; deployment to Vercel and Docker; observability and upgrades.

How This Book Is Organized

Part I — Foundations Through Your First Project (Chapters 1–5) explains how the book works and builds the Studio Site. **Part II — Content and Data** (Chapters 6–10) covers Server Components and caching, builds Driftwood Docs, and starts Driftwood Jobs. **Part III — Users, Auth, and Quality** (Chapters 11–15) completes the job board with authentication, authorization, uploads, search, and tests. **Part IV — Building a SaaS** (Chapters 16–20) builds Ledgerly. **Part V — Commerce, Real-Time, and AI** (Chapters 21–25) builds Driftwood Supply and Driftwood Live. **Part VI — Shipping and Maintaining** (Chapters 26–28) deploys, observes, and reviews everything. The appendices provide a cheat sheet, starter templates, a launch checklist, and a glossary.

Each project chapter opens with a short brief, builds step by step, and ends with a recap, practice exercises, and a **Key Takeaways** box.

The Running Example: Driftwood Labs

Every project in this book is a product of **Driftwood Labs**, a fictional four-person studio. Theo writes the brief for each project, Sara designs the interface, Jonah builds most features, and Mei, the tech lead, reviews the architecture and asks the hard questions. Following one team across six products lets

you see not only how to build each app, but how decisions and patterns carry from one project to the next.

Versions and Conventions

Examples use recent versions of Next.js (15 and later) with the App Router, React 19, TypeScript, Tailwind CSS v4, and `pnpm`. Next.js evolves quickly — caching defaults, request APIs, and some file conventions have changed between major versions — so where behavior depends on the version, the text says so and points you to the documentation.

> **TIP**
> A recommended practice or a technique that saves you time.

> **NOTE**
> Background information, version details, or a useful aside.

> **WARNING**
> A common mistake that causes bugs, leaks data, or breaks production.

File captions such as *driftwood-docs/app/search/page.tsx* tell you which project and file a snippet belongs to.

Contents

Preface

PART I — FOUNDATIONS THROUGH YOUR FIRST PROJECT

1 How This Book Works: Next.js, the App Router, and Learning by Building

2 Project 1 Kickoff: create-next-app, TypeScript, Tailwind CSS, and Project Structure

3 Project 1: Layouts, Pages, Navigation, and Metadata for a Studio Website

4 Project 1: An MDX Blog with Static Generation, Dynamic Routes, and RSS

5 Project 1: Images, Fonts, SEO, Open Graph, and Your First Deployment

PART II — CONTENT AND DATA

6 Server Components in Practice: Fetching, Caching, and Revalidation

7 Project 2: A Searchable Knowledge Base with Search Params, Streaming, and Suspense

8 Project 2: Loading, Error, and Not-Found UI, and a Search Route Handler

9 Project 3: Modeling the Job Board with PostgreSQL and Prisma

10 Project 3: Server Actions for Posting Jobs: Forms, Zod Validation, and Revalidation

PART III — USERS, AUTH, AND QUALITY

11 Project 3: Authentication with Auth.js: Sessions, OAuth, and Protected Routes

12 Project 3: Authorization, Roles, and the Employer Dashboard

13 Project 3: File Uploads, Email Notifications, and Work After the Response

14 Project 3: Pagination, Filtering, Full-Text Search, and URL State

15 Project 3: Testing the Job Board with Vitest, React Testing Library, and Playwright

PART IV — BUILDING A SAAS

16 Project 4: Planning Ledgerly: Organizations, Members, and Data Isolation

17 Project 4: The Dashboard: Parallel Routes, Charts, and Optimistic UI

18 Project 4: Subscriptions and Billing with Stripe and Webhooks

19 Project 4: Middleware (Proxy), Rate Limiting, Audit Logs, and Security Hardening

20 Project 4: Background Jobs, Scheduled Tasks, and Transactional Email

PART V — COMMERCE, REAL-TIME, AND AI

21 Project 5: The Driftwood Supply Catalog: Rendering and Caching Strategy

22 Project 5: Cart, Checkout, and Orders with Server Actions and Stripe Checkout

23 Project 5: Performance: Core Web Vitals, Images, Bundles, and Prerendering

24 Project 6: Driftwood Live: Real-Time Chat and Presence

25 Project 6: An AI Assistant: Streaming Responses, Tools, and Guardrails

PART VI — SHIPPING AND MAINTAINING

26 Deploying Next.js: Vercel, Docker Standalone Builds, and Environments

27 Observability, Error Tracking, and Keeping Next.js Apps Up to Date

28 Capstone: Reviewing the Driftwood Portfolio — Architecture Decisions, Checklists, and Next Steps

APPENDICES

A Next.js App Router Cheat Sheet

B Project Starter Templates and Snippets

C Production Launch Checklist

D Glossary

PART I

Foundations Through Your First Project

How the book works, the App Router mental model, and Project 1: Driftwood's studio website with layouts, navigation, an MDX blog, SEO, Open Graph images, and a first deployment.

1 · How This Book Works

2 · Project 1 Kickoff

3 · Layouts, Pages, and Metadata

4 · An MDX Blog

5 · Images, Fonts, SEO, and Deployment

CHAPTER 1 · PART I — FOUNDATIONS THROUGH YOUR FIRST PROJECT

How This Book Works: Next.js, the App Router, and Learning by Building

This book teaches Next.js the way most developers actually learn a framework: by shipping real applications, one after another, each a little harder than the last. In this chapter you meet Driftwood Labs, the small studio whose six products you will build, and you get the mental model, prerequisites, and working habits that every later chapter relies on.

Why a Project-Based Book

Most framework documentation is organized by feature. There is a page on routing, a page on data fetching, a page on caching, a page on metadata, and so on. That structure is excellent for reference, and you will keep the official Next.js documentation open in a browser tab for the whole book. But feature-by-feature learning has a known weakness: it rarely tells you *when* a feature matters, *which* alternative to pick, or *how* features interact once an application has users, content, a database, and a deadline.

Projects answer those questions naturally. When you build a marketing site, you discover why static generation and good metadata matter. When you build a job board, you discover that forms, validation, authentication, and authorization are one connected problem rather than four separate topics. When you build a subscription product, you learn that billing webhooks, background work, and audit logs are where most of the operational risk lives. The concepts are the same ones listed in the documentation, but you meet each one at the moment a real requirement makes it necessary, and that context is what makes it stick.

This book therefore has a simple rule: **every concept is taught at the moment a project needs it**. You will not find a chapter that lists every option of every API before you use any of them. Instead, each chapter starts from a short product brief, identifies what the brief requires, and builds exactly that, explaining the decisions and trade-offs along the way. By the end you will have written six complete applications, and the reference material will make far more sense because you will recognize every feature from a place where you used it.

Who This Book Is For

You will get the most from this book if you already write JavaScript comfortably, have at least a working familiarity with TypeScript, and have built something with React. You should know what a component is, how props flow down, how `useState` and `useEffect` behave, and why keys matter in

lists. You do not need to be a React expert, and short refreshers appear whenever a project leans on a React feature that beginners tend to skip, such as context, refs, or transitions.

Backend experience helps but is not required. From Project 3 onward you will work with a PostgreSQL database through Prisma, write Server Actions that change data, and handle authentication. Each of those topics is introduced from first principles as far as Next.js is concerned. If you have never written SQL, you will still be able to follow along, although you may want a general SQL primer on hand for the more advanced queries in Chapter 14.

What Next.js Is, in One Page

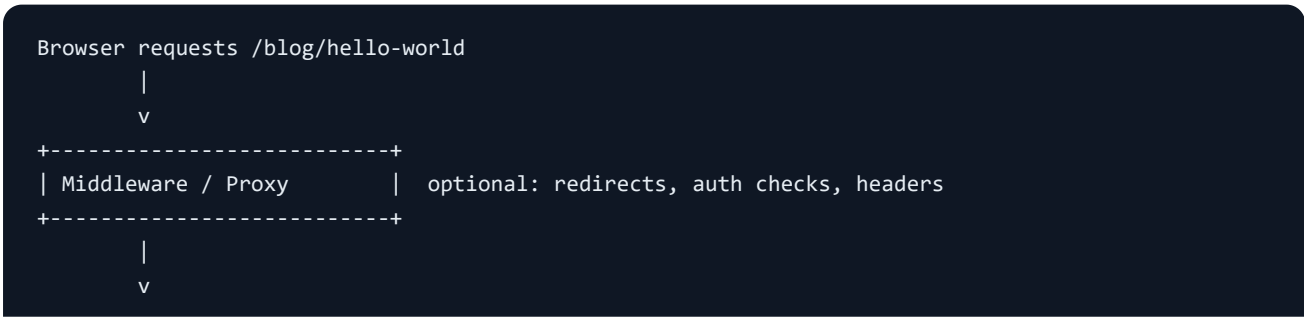
Next.js is a React framework. React itself is a library for describing user interfaces as a tree of components; it does not decide how URLs map to screens, how data is fetched on the server, how pages are bundled and optimized, or how an application is deployed. Next.js makes those decisions for you, with conventions you can learn once and reuse in every project.

The modern way to build with Next.js is the **App Router**, which lives in a folder called `app/`. In the App Router, folders define URL segments and special file names define what happens at each segment. A `page.tsx` file makes a segment publicly routable; a `layout.tsx` file wraps every page beneath it in shared UI; `loading.tsx`, `error.tsx`, and `not-found.tsx` describe what to show while waiting, when something fails, and when something does not exist. A `route.ts` file turns a segment into an HTTP endpoint instead of a page. Those few conventions carry most of the framework.

The second big idea is **React Server Components**. By default, every component in the `app/` directory is a Server Component: it runs on the server, can read files and query databases directly, and sends only rendered output to the browser, not its own JavaScript. When a component needs interactivity, such as state, event handlers, or browser APIs, you mark its file with the `"use client"` directive and it becomes a Client Component that is also hydrated in the browser. Learning where to draw that boundary is one of the core skills this book develops, starting in Chapter 3.

The third big idea is **Server Actions**: async functions marked with `"use server"` that the browser can call, typically by submitting a form. They let you write data mutations next to the UI that triggers them, without building a separate REST endpoint for every button. You will use them heavily from Chapter 10 onward.

Here is a simplified picture of what happens when someone visits a page in an App Router application. Do not worry about every box yet; each one gets its own chapter.



```
+--------------------------+
| Router matches app/ tree  |  app/blog/[slug]/page.tsx + parent layouts
+--------------------------+
        |
        v
+--------------------------+
| Render Server Components  |  read files, query DB, fetch APIs
| (static or dynamic)       |  cached output reused when allowed
+--------------------------+
        |
        v
+--------------------------+
| Stream HTML + RSC payload |  Suspense boundaries stream in order
+--------------------------+
        |
        v
Browser shows HTML, then hydrates Client Components only
```

Notice that the browser receives HTML first. That is why Next.js sites can be fast and search-engine friendly even when they contain a lot of React: the heavy lifting happens on the server, and the browser downloads JavaScript only for the interactive islands.

Static, dynamic, and cached

Every route is rendered either ahead of time (static) or per request (dynamic), and caching decides how long rendered output and fetched data are reused. Caching behavior has changed between major versions, and it continues to evolve. In Next.js 15, `fetch` requests and `GET` Route Handlers are not cached by default, and you opt in with options like `export const revalidate` or `fetch(url, { next: { revalidate } })`. Newer versions introduce Cache Components and the `"use cache"` directive, which fold Partial Prerendering into a more explicit model. This book explains the stable concepts, notes where recent versions differ, and asks you to check the documentation for the exact release you are using. Chapter 6 is devoted entirely to this subject.

Meet Driftwood Labs

Every project in this book belongs to one fictional company: **Driftwood Labs**, a four-person product studio that designs, builds, and runs a small portfolio of web applications. Using a single company keeps the book reading as one story while letting each project stand on its own. You can skip a project and come back to it later; nothing in Project 4 requires code from Project 3, although the ideas build on each other.

The studio lives at `driftwoodlabs.example`. The `.example` top-level domain is reserved for documentation, so no domain in this book points to a real website, and every email address in the code is safe to copy into test data. The client companies you will encounter, such as Harbor & Pine Coffee, Acme Plumbing, and Nova Studio, are fictional as well.

The people

Four people appear throughout the book. They are not decoration: each one represents a perspective a real team brings to a project, and their requests shape the code.

Person	Role	What they care about
Theo	Product and operations	Writes the brief for each project: goals, must-haves, and what is out of scope
Mei	Tech lead	Reviews architecture, pushes for simple boundaries, security, and maintainability
Jonah	Full-stack developer	Builds features quickly and asks the practical questions you will ask too
Sara	Designer	Owns the look and feel, accessibility, and UI requirements for every screen

Each project chapter opens with **Theo's brief**, a short block that describes the goal, the must-have features, and what is explicitly out of scope. Briefs are deliberately short. Real product briefs are often vague, and part of the skill you are building is turning a vague requirement into concrete routes, components, and data models. When Mei reviews a design decision or Sara adds a UI constraint, the text calls it out so you can see where a requirement came from.

The Six Projects

The book is organized around six Driftwood products. They are ordered by complexity, and each introduces a new layer of the stack. The table below is your roadmap; come back to it whenever you want to see where you are.

#	Project (repo)	What it is	Key topics	Chapters
1	Studio Site (`driftwood-site`)	Marketing site with an MDX blog	Layouts, metadata, static generation, RSS, images, fonts, OG images, SEO, deployment	2–5
2	Driftwood Docs (`driftwood-docs`)	Searchable knowledge base	Caching, search params, streaming, Suspense, error and not-found UI, Route Handlers	6–8
3	Driftwood Jobs (`driftwood-jobs`)	Job board	PostgreSQL, Prisma, Server Actions, Zod, Auth.js, roles, uploads, email, search, testing	9–15
4	Ledgerly (`ledgerly`)	Multi-tenant invoicing SaaS	Organizations, data isolation, parallel routes, Stripe billing, webhooks, proxy, rate limits, jobs	16–20
5	Driftwood Supply (`driftwood-supply`)	Merch and template storefront	Catalog caching, cart, Stripe Checkout, orders, Core Web Vitals	21–23
6	Driftwood Live (`driftwood-live`)	Team chat with an AI assistant	Real-time updates, presence, streaming AI responses, tools, guardrails	24–25

After the six projects, Chapters 26 through 28 step back to the portfolio level: deploying to Vercel and to self-hosted Docker containers, adding observability and error tracking, keeping dependencies current, and finally reviewing the architecture decisions across all six applications.

If you are in a hurry, you can read Chapters 1 through 5 and then jump to the project that matches your work. Chapter 2 establishes the setup that every project reuses, so later chapters will often say "same setup as Chapter 2" and move straight to what is new.

Prerequisites and Tools

You need a small set of tools installed before Chapter 2. None of them are exotic, and all are free.

- **Node.js.** Install a current Long-Term Support (LTS) release. Next.js documents a minimum Node.js version for each major release, and that minimum rises over time, so check the installation page of the documentation for the version you plan to use. A version manager such as `nvm`, `fnm`, or Volta makes switching painless.
- **pnpm.** This book uses `pnpm` as its package manager because it is fast, disk-efficient, and strict about dependencies. Recent Node.js releases ship with Corepack, which can enable pnpm for you. Chapter 2 shows the npm equivalents once, and you can translate freely if you prefer npm, Yarn, or Bun.
- **Git.** Every project is its own repository, and the deployment chapters assume you can push to a remote such as GitHub, GitLab, or Bitbucket.
- **An editor with TypeScript support.** Visual Studio Code is a common choice, but any editor that runs the TypeScript language server works. Install the ESLint and Tailwind CSS extensions if your editor offers them.
- **A modern browser with developer tools.** You will use the Network and Performance panels regularly, especially in Chapters 6 and 23.
- **Later projects:** PostgreSQL (installed locally or through Docker) from Chapter 9, a Stripe account in test mode from Chapter 18, and Docker from Chapter 26. Each chapter tells you when to install something, so you do not need these on day one.

Check your setup with a few commands in a terminal. The exact version numbers you see will differ from anything printed in a book, and that is fine.

terminal

```
node --version
corepack enable
pnpm --version
git --version
```

If `corepack enable` fails with a permissions error, install pnpm through its standalone installer instead, or run your terminal with the rights your operating system requires for global installs. If you use a version manager, make sure the shell you run commands in has actually picked up the Node.js

version you expect; mixing a system Node.js with a managed one is the most common source of confusing install errors.

> **WARNING**
>
> Do not follow tutorials that tell you to install packages with `sudo`. Global installs that require administrator rights usually indicate a Node.js installation owned by the system rather than your user, and they cause file-permission problems later. A version manager avoids the issue entirely.

Versions and How This Book Handles Change

Next.js moves quickly. Major versions arrive regularly, and each one tends to change a default, rename a file convention, or add a new way to do something that already had an older way. A book cannot freeze the framework, so this one follows a few explicit rules to stay useful as releases continue.

- **The baseline is Next.js 15 and later, React 19, and TypeScript.** Code in this book targets recent versions of the App Router. It does not use the older Pages Router except where a comparison helps.
- **Stable concepts are stated plainly.** Folder-based routing, layouts, Server Components, Server Actions, the Metadata API, and Route Handlers are foundational, and the text describes them with confidence.
- **Version-specific behavior is hedged.** When a default or file name has changed recently, the text says "in recent versions" and describes both the old and new form. For example, the request interceptor that older releases call **middleware** is called **proxy** in recent versions (`proxy.ts`), and the book mentions both names.
- **Async request APIs are used throughout.** Since Next.js 15, `params`, `searchParams`, `cookies()`, and `headers()` are asynchronous, so the code always awaits them. If you are maintaining an older application you may see synchronous access; Chapter 27 discusses upgrade strategies.
- **No invented numbers.** When the book discusses performance, it describes what to measure and how, rather than quoting benchmark results. Any sample metric in a table is labeled illustrative.

> **NOTE**
>
> When a code listing in this book disagrees with the current documentation, trust the documentation for the exact API, then use the book for the reasoning behind the design. The reasoning ages much more slowly than the syntax. The release notes and the official upgrade guides are the best place to learn what changed between two versions.

The same attitude applies to the libraries around Next.js. Tailwind CSS v4 configures itself mostly from CSS rather than a JavaScript config file, and its details continue to evolve. Auth.js, the authentication library used in Project 3, has changed its API across versions, and credible alternatives exist. Stripe adds and renames features regularly. Whenever a chapter depends on a third-party API, it names the concept first, shows the code for recent versions, and points you to the provider's documentation for the latest details.

How Each Chapter Is Structured

Project chapters follow a consistent rhythm so you always know where you are.

1. **Theo's brief.** A short statement of goals, must-haves, and out-of-scope items.
2. **The plan.** A breakdown of the brief into routes, components, data, and decisions, often with an ASCII diagram or a table.
3. **The file tree.** The new and changed files for the chapter, so you can see the shape of the work before writing any code.
4. **Step-by-step build.** Complete, runnable TypeScript listings with explanations of why each piece exists and what the alternatives were.
5. **What we built.** A recap that ties the code back to the brief.
6. **Practice.** Three to five extension exercises with brief solution sketches. These are where the learning deepens; do at least one per chapter.
7. **Key Takeaways.** A short summary box you can skim later as a refresher.

Along the way you will see three kinds of callout boxes. A **Tip** offers a shortcut or a better way to work. A **Note** adds context, such as a version difference or a related concept. A **Warning** flags a mistake that is easy to make and expensive to discover later, such as leaking a secret to the browser or caching private data.

Reading the code listings

Every code listing has a caption above it that shows the file path relative to the repository root, such as `driftwood-site/app/layout.tsx`. When a listing shows a whole file, you can copy it as is. When a listing shows only part of a file, the surrounding prose says so, and a comment like `// ...existing imports` marks the elided code. Terminal commands have the caption `terminal` and should be run from the repository root unless the text says otherwise.

Listings are kept short enough to read on a page. Long JSX attributes and Tailwind class lists are broken across lines, which is also a good habit in your own code because it makes diffs easier to review. Here is a small example that shows the style you will see throughout the book: a Server Component that reads a value, a typed props object, and clean, wrapped markup.

driftwood-site/components/section-heading.tsx

```
type SectionHeadingProps = {
  eyebrow?: string;
  title: string;
  children?: React.ReactNode;
};

export function SectionHeading({ eyebrow, title, children }: SectionHeadingProps) {
  return (
    <header className="mx-auto max-w-2xl text-center">
      {eyebrow && (
        <p className="text-sm font-semibold uppercase tracking-wide text-teal-700">
          {eyebrow}
        </p>
      )}
      <h2 className="mt-2 text-3xl font-bold tracking-tight text-slate-900">
        {title}
      </h2>
      {children && <div className="mt-4 text-lg text-slate-600">{children}</div>}
    </header>
  );
}
```

This component has no `"use client"` directive, so it is a Server Component. It renders to HTML on the server and ships no JavaScript of its own to the browser. You will see this pattern constantly: most of an App Router application is plain, typed components like this one, and interactivity is added only where the product requires it.

A Quick Tour of the App Router Vocabulary

Before Chapter 2 creates the first project, it helps to know the words you will read and write every day. The table below is a preview, not something to memorize. Each item is taught properly in the project where it first matters, and Appendix A repeats it as a cheat sheet.

Term	What it means	First used
`page.tsx`	Makes a route segment publicly accessible and renders its UI	Chapter 2
`layout.tsx`	Shared UI that wraps child segments and persists across navigation	Chapter 3
Route group `(name)`	Folder that organizes routes without affecting the URL	Chapter 3
Metadata API	Typed `metadata` or `generateMetadata` exports for titles and tags	Chapter 3
Dynamic segment `[slug]`	A URL part that becomes a parameter, such as a post slug	Chapter 4
`generateStaticParams`	Lists dynamic paths to prerender at build time	Chapter 4
`route.ts`	A Route Handler that returns an HTTP response instead of a page	Chapter 4
`opengraph-image.tsx`	Generates the social sharing image for a route	Chapter 5

`loading.tsx`	Instant loading UI backed by a Suspense boundary	Chapter 7
`error.tsx`	Error boundary UI for a segment	Chapter 8
Server Action	An async server function callable from forms and client code	Chapter 10
Parallel route `@slot`	Renders several independent views in one layout	Chapter 17
Middleware / proxy	Code that runs before a request reaches a route	Chapter 19

Server Components versus Client Components, briefly

Because the Server and Client Component split is the idea people find most unfamiliar, here is the short version that will carry you through Project 1. A Server Component can be `async`, can read from the file system or a database directly, and can keep secrets such as API keys, because its code never reaches the browser. It cannot use state, effects, event handlers, or browser-only APIs. A Client Component can do all of those interactive things, but everything it imports is bundled and sent to the browser, so it must never contain secrets.

The two kinds compose freely with one rule of direction: a Server Component can render a Client Component and pass it serializable props, and a Client Component can receive Server Components as `children`, but a Client Component cannot import a Server Component directly. The practical consequence is a design habit you will practice constantly: keep interactive pieces small, push them to the leaves of the tree, and let the server do everything else.

example: the boundary in practice

```
// app/page.tsx (Server Component: no directive needed)
import { NewsletterForm } from "@/components/newsletter-form";
import { getLatestPosts } from "@/lib/posts";

export default async function HomePage() {
  const posts = await getLatestPosts(3); // runs on the server
  return (
    <main>
      <ul>
        {posts.map((post) => (
          <li key={post.slug}>{post.title}</li>
        ))}
      </ul>
      <NewsletterForm /> {/* interactive leaf */}
    </main>
  );
}

// components/newsletter-form.tsx (Client Component)
"use client";
import { useState } from "react";

export function NewsletterForm() {
  const [email, setEmail] = useState("");
  return (
```

```
      <input value={email} onChange={(e) => setEmail(e.target.value)} />
    );
  }
```

Only `NewsletterForm` ships JavaScript. The list of posts is rendered once on the server and arrives as HTML. Keep this picture in mind; Chapter 3 builds a real version of it with an active navigation link and a mobile menu.

How to Use This Book

There is more than one good way to read a project-based book. Pick the approach that matches your goals and your time.

Build along (recommended)

The most effective approach is to type the code yourself, chapter by chapter, in your own repositories. Typing forces you to read every line, and the small mistakes you make and fix are some of the most valuable lessons in the book. When a listing does not work, compare it carefully with the text, read the error message fully, and check that your file is in the right folder; most App Router surprises come from a file with the wrong name or in the wrong place.

Jump to what you need

If you already ship Next.js applications and want a specific skill, such as multi-tenant data isolation or Stripe webhooks, jump to that project. Skim Chapter 2 first so you recognize the shared conventions, and use the Key Takeaways boxes of earlier chapters to fill in any gaps.

> **TIP**
>
> Commit after every major step in a chapter, with a message that names the step, such as `feat: add blog index page`. When something breaks two steps later, `git diff` against the last good commit is usually the fastest way to find the cause, and your commit history becomes a personal index of the book.

The Shared Conventions Across All Projects

Although the six projects are independent repositories, they share conventions so that you can move between them without relearning where things live. Chapter 2 sets these up in detail for Project 1, and later projects reuse them with a one-line reference. Here is the summary so you know what to expect.

Area	Convention
Language	TypeScript in strict mode everywhere, including config files where supported
Routing	App Router in `app/`, with route groups to organize sections

Components	Shared UI in `components/`, primitives in `components/ui/`
Logic	Server-side helpers and data access in `lib/`
Imports	The `@/` alias pointing at the repository root
Styling	Tailwind CSS v4 imported from `app/globals.css`
Environment	`.env.example` committed, `.env.local` ignored, validated in `lib/env.ts`
Quality	ESLint, Prettier, and a type check, combined in one `pnpm check` script
Validation	Zod schemas for anything that crosses a trust boundary

These conventions are not the only reasonable choices. Some teams prefer a `src/` directory, co-locate components inside route folders, or use a different linter. The point is consistency: once a team agrees on where things go, code review gets faster and new contributors get productive sooner. Mei's rule at Driftwood is simple: "Pick a convention, write it down, and do not argue about it again until it hurts."

Working Habits That Pay Off

A few habits make the difference between following along and actually learning. They are the same habits the Driftwood team uses on real client work.

Run the production build early and often. The development server is forgiving and optimized for fast feedback. The production build, run with `pnpm build` followed by `pnpm start`, is where type errors, prerendering failures, and accidental dynamic rendering show up. The build output also tells you which routes are static and which are dynamic, which is one of the most useful signals you have. Make a habit of running a production build at the end of each chapter.

Keep secrets on the server. Only environment variables prefixed with `NEXT_PUBLIC_` are exposed to browser code. Everything else stays on the server. This rule is simple, and breaking it by accident is one of the most serious mistakes you can make, so Chapter 2 builds validation around it from the start.

What We Covered

This chapter did not build anything yet, but it set up everything the build chapters rely on. You now know why the book is organized around projects, what Next.js adds on top of React, and the three ideas at the core of the App Router: file-system routing with special files, Server Components by default, and Server Actions for mutations. You met Driftwood Labs and its team, saw the six projects and the order in which they build on each other, and learned how the book treats a fast-moving framework: stable concepts stated plainly, version-specific behavior hedged, and the official documentation as the final authority on exact syntax.