1ST EDITION

FULL-STACK · AI-POWERED

Next.js

The Comprehensive *Guide*

From React Fundamentals to
AI-Powered Full-Stack Apps

Next.js · App Router · Server Components · Vercel AI SDK · Prisma · Auth.js

◆ Includes AI-assisted development workflows

WRITTEN BY

Florian Wessels

2026

Next.js – The Comprehensive Guide

From React Fundamentals to AI-Powered Full-Stack Apps

Florian Wessels

This book is available at https://leanpub.com/comprehensive-nextjs-guide

This version was published on 2026-09-06



This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2026 Florian Wessels

Contents

Introduction . . . . . **i**
About the Author i
Why This Book? i
Who This Book Is For, and Who It's Not For ii
The Practice Project: HelpDesk AI iii
How to Read This Book v
Prerequisites vi
Companion Code and Resources vii
Conventions in This Book vii

Part I: Fundamentals **1**

1. TypeScript: Quick Start for Experienced Developers **2**
1.1. Why TypeScript? 2
1.2. Setting Up the Development Environment 4
1.3. The Type System: From Primitives to Generics 8
1.4. Interfaces vs. Types: When to Use Which 15
1.5. Utility Types: Reusing What Already Exists 17
1.6. Modules, Imports, and `tsconfig.json` 20
1.7. Common Stumbling Blocks When Switching 24
1.8. Building the TypeScript Playground 27
Summary 32

2. Modern JavaScript for TypeScript Developers **34**
2.1. Destructuring: Unpacking Data 34
2.2. Spread and Rest: Copying and Collecting 34
2.3. Template Literals 35
2.4. Arrow Functions and Lexical `this` 35
2.5. Promises, `async/await`, and Error Handling 35
2.6. Array Methods: Thinking in Transformations 36
2.7. Optional Chaining and Nullish Coalescing 37
2.8. Quick Reference: Additional Patterns 37
Summary 38

3. React: Fundamentals and Mental Model **39**
3.1. The Mental Model: UI as a Function of State 39
3.2. Setting Up a React Playground 39
3.3. JSX: Writing UI in TypeScript 39

3.4. Components, Props, and Composition . . . 40
3.5. State with `useState`: Making Components Interactive . . . 41
3.6. Side Effects with `useEffect` . . . 42
3.7. Event Handling and Controlled Forms . . . 42
3.8. Lists and Keys . . . 43
3.9. Putting It All Together: A Mini Ticket Board . . . 43
Summary . . . 44

4. Advanced React . . . 45
4.1. Custom Hooks: Extracting Reusable Logic . . . 45
4.2. Context API: Sharing State Across the Tree . . . 46
4.3. `useReducer`: Complex State Logic . . . 46
4.4. Refs, Memoization, and Performance . . . 47
4.5. Error Boundaries and Suspense . . . 48
4.6. Actions, Optimistic Updates, and the `use` Hook . . . 48
Summary . . . 49

Part II: Next.js Core Concepts . . . 50

5. Next.js: Getting Started and Project Structure . . . 51
5.1. What Is Next.js, and Why Do We Need It? . . . 51
5.2. App Router vs. Pages Router . . . 52
5.3. Creating a Project with `create-next-app` . . . 53
5.4. VS Code Setup . . . 57
5.5. Directory Structure: Where Everything Lives . . . 61
5.6. Configuration: `next.config.ts` in Detail . . . 63
5.7. Code Quality Tooling: ESLint, Prettier, knip, and Husky . . . 66
5.8. Dev Server, Build Process, and Turbopack . . . 74
5.9. The Next.js DevTools . . . 78
Summary . . . 79

6. Routing and Navigation . . . 81
6.1. File-Based Routing in the App Router . . . 81
6.2. Layouts, Templates, and Nested Routes . . . 81
6.3. Dynamic Routes . . . 82
6.4. Route Groups and Parallel Routes . . . 82
6.5. Loading, Error, and Not-Found Boundaries . . . 83
6.6. Navigation: `<Link>`, `useRouter`, `useLinkStatus`, and `redirect` . . . 84
6.7. Search Params and URL State Management . . . 84
6.8. Intercepting Routes and Modals . . . 85
Summary . . . 85

7. Understanding Rendering Strategies . . . 87
7.1. The Rendering Problem . . . 87
7.2. Server Components vs. Client Components . . . 87
7.3. Cached Rendering and the `'use cache'` Directive . . . 88
7.4. Dynamic Rendering . . . 88
7.5. Streaming and Suspense . . . 89
7.6. Cache Revalidation: Keeping Cached Pages Fresh . . . 90

7.7. Partial Prerendering (PPR) . . . 91
7.8. Understanding Hydration . . . 91
7.9. Practical Patterns and Anti-Patterns . . . 92
7.10. What the DevTools Show Now . . . 93
Summary . . . 93

8. Data Fetching . . . 94
8.1. The Data Problem . . . 94
8.2. Fetching Data in Server Components . . . 94
8.3. The `cache()` Function and Request Memoization . . . 95
8.4. Server Actions: Mutations Without API Endpoints . . . 95
8.5. Calling Server Actions from Client Components . . . 96
8.6. Optimistic Updates . . . 97
8.7. Bulk Actions: Mutating Multiple Records . . . 98
8.8. Parallel vs. Sequential Fetching . . . 98
8.9. Caching Strategy: Putting the Layers Together . . . 98
Summary . . . 99

9. Styling and UI Components . . . 100
9.1. The Styling Problem . . . 100
9.2. Tailwind CSS: Setup and Configuration . . . 100
9.3. Building a Shared Component Library . . . 101
9.4. CSS Modules as an Alternative . . . 102
9.5. Font Optimization with `next/font` . . . 102
9.6. Image Optimization with `next/image` . . . 103
9.7. Dark Mode, Theming, and Responsive Layouts . . . 103
9.8. Styling the HelpDesk AI Application . . . 105
Summary . . . 105

Part III, Deepening the Practice Project: Full-Stack Architecture . . . 107

10. Project Architecture and Data Model . . . 108
10.1. Why Architecture Matters . . . 108
10.2. Feature Slices: Organizing Code by Domain . . . 108
10.3. Designing the Data Model as a Prisma Schema . . . 109
10.4. Shared Types and the Scope System . . . 110
Summary . . . 111

11. Database and ORM . . . 112
11.1. From Schema to Running Database . . . 112
11.2. Setting Up Prisma with PostgreSQL . . . 112
11.3. Migrations: Evolving the Schema . . . 112
11.4. Type-Safe Queries with Prisma Client . . . 113
11.5. Seeding the Database . . . 113
11.6. Full-Text Search Across Tickets . . . 114
11.7. Connection Pooling for Serverless . . . 114
11.8. Aside: MongoDB as an Alternative . . . 114
Summary . . . 115

12. Authentication and Authorization **116**
12.1. The Problem: Open Doors Everywhere 116
12.2. Auth.js: The Authentication Layer 116
12.3. Google OAuth: Setting Up the Provider 116
12.4. Session Management 117
12.5. Protecting Routes with the Proxy 117
12.6. Scope-Based Access Control 118
12.7. Two Flows: Customer Portal and Agent Dashboard 118
12.8. The Trade-offs of Email and Password 118
Summary 119

13. Forms, Validation, and File Management **120**
13.1. The Problem: Forms That Fight Back 120
13.2. React Hook Form: Type-Safe Form State 120
13.3. Zod: One Schema, Both Sides 120
13.4. Aside: Joi as an Alternative 121
13.5. Controlled Components with Controller 121
13.6. Rich Text with Tiptap 121
13.7. File Uploads with S3 Presigned URLs 121
13.8. The Complete Forms 122
Summary 122

14. Email, State Management, and Real-Time **123**
14.1. Three Missing Pieces 123
14.2. Transactional Emails with AWS SES 123
14.3. State Management with Zustand 123
14.4. SWR for Server State 124
14.5. Dashboard Visualizations with Recharts 124
14.6. Activity Feed and Webhook Dispatch 125
14.7. Real-Time Updates with Server-Sent Events 125
14.8. SLA Countdown Timer 125
Summary 125

Part IV: AI Features with the Vercel AI SDK **127**

15. Introduction to the Vercel AI SDK **128**
15.1. The Missing Intelligence 128
15.2. Two Libraries, Two Responsibilities 128
15.3. One API, Many Models: The Gateway 128
15.4. Setting Up the AI SDK in HelpDesk AI 128
15.5. Tracking Token Usage From Day One 128
15.6. The AI Settings Page 129
15.7. The First AI Call: A Ticket Title Generator 129
Summary 129

16. Text Generation and Streaming **130**
16.1. The Waiting Problem 130
16.2. `generateText` vs. `streamText` 130
16.3. The Streaming Route Handler 130

16.4. Streaming Into the UI With `useCompletion` . . . 130
16.5. System Prompts: Defining the Job . . . 130
16.6. Generation Parameters . . . 130
16.7. The Right Runtime for AI Routes . . . 131
16.8. Wiring the Summary Into the Ticket Page . . . 131
Summary . . . 131

17. Building a Chat Interface . . . 132
17.1. From One-Shot to Conversation . . . 132
17.2. The `useChat` Hook . . . 132
17.3. The Copilot Route Handler . . . 132
17.4. Multi-Turn Context and Token Limits . . . 132
17.5. Persisting and Resuming Conversations . . . 132
17.6. Keeping the Newest Message in View . . . 132
17.7. The Copilot on the Ticket Page . . . 133
17.8. The Standalone Copilot Page . . . 133
Summary . . . 133

18. Structured Output and Tool Calling . . . 134
18.1. Beyond Free-Form Text . . . 134
18.2. `Output.object`: Type-Safe Structured Output . . . 134
18.3. Auto-Categorization and Sentiment . . . 134
18.4. Tool Calling: Giving the Model Hands . . . 134
18.5. Multi-Step Execution with `stopWhen` . . . 134
18.6. Human-in-the-Loop with `toolApproval` . . . 135
18.7. Scope-Protected Execution . . . 135
Summary . . . 135

19. Retrieval-Augmented Generation (RAG) . . . 136
19.1. Understanding Embeddings . . . 136
19.2. Text Chunking . . . 136
19.3. Setting Up pgvector . . . 136
19.4. The Embedding Model Configuration . . . 136
19.5. The Knowledge Base Becomes Real . . . 136
19.6. The Embedding Pipeline . . . 136
19.7. The Retrieval Pipeline . . . 136
19.8. Prompt Engineering With Retrieved Context . . . 136
19.9. The Payoff: Semantic Tools and Suggested Replies . . . 137
Summary . . . 137

Index . . . 138

Introduction

About the Author

I'm Florian Wessels, CTO, product executive, and a career changer into software development. I hold a B.Sc. in chemical engineering, but I've spent the last decade building software, leading engineering teams, and integrating AI into products.

As CTO of a technology-leading AI SaaS company, I built a product from the ground up, with React, Node.js, Next.js, Prisma, Auth.js, the Vercel AI SDK, Mixpanel, and Sentry, deployed on Vercel. It is still the stack I build on today, self-hosted in Swiss data centers instead. That's not a random collection of buzzwords; it's what we'll work with throughout this book, battle-tested in production with real users and real scale.

Today, I'm CTO of a Swiss company building sovereign AI: an infrastructure and application platform that runs in Swiss data centers, so hospitals, public administrations, and universities can put AI to work on sensitive data without that data ever leaving the country. Before that, I designed and implemented AI solutions for M&A processes. I've published articles on artificial intelligence and prompt engineering in renowned technology magazines. Additionally, I work as an independent consultant, coach, and software engineer helping teams build digital products.

Why does my background matter for this book? Because I've been on both sides: the developer trying to ship features on a Friday afternoon, and the CTO deciding which technologies to bet on. This book reflects both perspectives. It's practical enough for the developer and strategic enough for the tech lead.

Why This Book?

There are many ways to learn Next.js. The official documentation is excellent. YouTube tutorials are plentiful. Online courses cover the fundamentals in hours. So why write a book, and why this one?

The Gap

To the best of our knowledge, no existing book guides developers through a single, cohesive, production-ready project that combines current Next.js with the Vercel AI SDK. Some books touch on AI integration alongside Next.js, but they either rely on additional orchestration layers, target older framework versions, or spread their content across multiple isolated demos rather than one continuous build. Most Next.js books build full-stack projects without any AI integration at all, leaving developers to figure out one of the most relevant modern capabilities on their own.

This book fills that gap. We build one application, HelpDesk AI, from `create-next-app` to production deployment, with AI features woven into the architecture from the start. Not as a chapter bolted on at the end, but as an integral part of how the application works.

AI Integration Is Becoming Table Stakes

AI tools have moved from novelty to routine in most engineering teams. Most modern web applications will benefit from LLM interaction, whether that's a support chatbot, intelligent search, content generation, or automated classification. The question is no longer *if* an application will have AI features, but *when* and *how well*.

This book shows how to build those features into a real application from day one. We'll use the Vercel AI SDK (built by Vercel, the company behind Next.js) to add streaming chat, structured output, tool calling, and retrieval-augmented generation to HelpDesk AI.

From Tutorial to Production

Most tutorials end where the real work begins. They show us how to fetch data and render a component. They don't show us how to handle authentication across roles, validate forms on both client and server, send transactional emails, manage file uploads, monitor errors in production, or deploy with confidence.

That's where this book comes in. We cover the "boring but critical" parts (auth, error handling, testing, security, monitoring, deployment), everything that separates a demo from a product we'd ship to actual users. One project from start to finish, not a collection of disconnected examples.

With our value proposition clear, let's talk about who this book is written for, and who it's not for.

Who This Book Is For, and Who It's Not For

Ideal For

This book is written for developers who want to build real applications with Next.js:

- **Experienced TypeScript or JavaScript developers** who want to use Next.js productively. We know our way around `async/await`, we've written functions and classes, and we want to build something real.
- **Developers coming from other languages** (Java, C#, Python, Go) who are entering the React and Next.js ecosystem. Chapters 1–4 give a fast-track introduction to TypeScript and React with experienced developers in mind.
- **CTOs, tech leads, and architects** evaluating whether Next.js and AI integration fit their products. The practice project mirrors real-world complexity. It's not a to-do app.
- **Teams adopting Next.js** who need a shared reference that covers architecture, conventions, and production patterns in one place.

Not Suitable For

To set honest expectations:

- **Absolute programming beginners.** This is not a "learn to program" book. We assume solid experience in at least one programming language.
- **Developers looking for a pure React book** without backend or full-stack coverage. We go deep into server-side patterns, databases, and deployment.

- **Developers looking for a pure AI or machine learning book.** We use AI APIs and the Vercel AI SDK. We don't train models or dive into neural network architecture.

A Note on the First Four Chapters

The book opens with a compact introduction to TypeScript (Chapters 1–2) and React (Chapters 3–4). These chapters are written for experienced developers switching stacks, not for absolute beginners. If TypeScript and React are already familiar territory, skip ahead to Chapter 5 where we create the Next.js project and start building HelpDesk AI.

These foundational chapters exist so nobody is left behind, not as a gate to pass through. Use them as a reference whenever we need them.

Now let's look at what we're building.

The Practice Project: HelpDesk AI

Throughout this book, we build **HelpDesk AI**, an AI-powered support portal for teams. Customers submit tickets through a public portal. Agents handle them with the help of an AI copilot that automatically categorizes incoming tickets and drafts replies grounded in the cases the team has already resolved.

Why This Project?

We chose HelpDesk AI because it covers the breadth of features found in real-world applications:

- **Authentication and roles**: customers on one side, and owners, admins, agents, and viewers inside an organization, each with a different set of scopes
- **Forms and validation**: ticket submission, reply editors, rich text editing with file attachments
- **Real-time updates**: Server-Sent Events for live ticket feeds and SLA countdown timers
- **File management**: uploads via AWS S3 with presigned URLs
- **Email notifications**: transactional emails via AWS SES for ticket confirmations and agent replies
- **AI features**: streaming chat, retrieval-augmented generation over resolved tickets and knowledge articles, tool calling for automated actions, structured output for classification and sentiment analysis

This same project could be deployed for a real team with minimal adjustments. It's not a toy. It's a product.

Architecture Overview

HelpDesk AI consists of three main areas:

- **Customer portal**: A public-facing site where customers submit tickets and track their status. No login required for submission; registered customers can view their ticket history.
- **Agent dashboard**: An internal application where the support team manages tickets, communicates with customers, and uses the AI copilot for faster resolutions.

- **Knowledge base**: A collection of articles and documentation that serves both customers (self-service) and the AI (as a retrieval source for generating reply suggestions).

We won't dive into the full technology stack here; that comes in Chapter 10 when we design the architecture and data model. For now, the important thing is the shape: three interconnected areas, one codebase, one deployment.

How the Project Grows

The book has 6 parts, 29 chapters, and 7 appendices. Here's how HelpDesk AI evolves across them:

- **Part I** (Chapters 1–4): We build our foundation in TypeScript and React. There's no project code yet, but these are the skills we'll need for everything that follows.
- **Part II** (Chapters 5–9): We scaffold the Next.js project, set up routing, implement rendering strategies, build the data layer, and create a styled component library. By the end of Part II, we have a working application with navigation, data fetching, and a polished UI, all backed by an in-memory data store.
- **Part III** (Chapters 10–14): We go full-stack. A real database with Prisma, authentication with Auth.js, forms with React Hook Form and Zod, file uploads via S3, transactional emails, state management with Zustand, and real-time updates with Server-Sent Events.
- **Part IV** (Chapters 15–19): The AI chapters, the heart of what makes this book different. We integrate the Vercel AI SDK for text generation, build a streaming chat interface, add structured output for auto-categorization, implement tool calling so the AI copilot can take actions, and build a RAG pipeline over resolved tickets and knowledge articles.
- **Part V** (Chapters 20–25): Production readiness. Testing with Vitest and Playwright, performance optimization, security hardening, monitoring with Sentry and Mixpanel, deployment on Vercel, and self-hosting with Docker as an alternative.
- **Part VI** (Chapters 26–29): Advanced topics. SEO and accessibility, internationalization with `next-intl`, the Next.js ecosystem and future, and AI-assisted development with Claude Code.

The **appendices** (A–G) provide quick reference material: a TypeScript cheat sheet, React hooks reference, Next.js CLI and configuration guide, Vercel AI SDK API reference, VS Code setup, alternative IDE configuration, and a glossary.

What the Project Keeps Simple

HelpDesk AI is built to teach, so its data model and rules stay focused on the concept each chapter is about. A few places hold a deliberately simple shape that a production deployment would grow, and each is an additive change rather than a rewrite:

- **Customers are individual users.** A customer is a `User` identified by email, the same table agents live in, told apart by a platform role. A larger product would group several contacts under one customer account or company; that is a new table with a relation, not a redesign.
- **Organizations are flat.** Membership is a role plus a set of scopes, with no teams, departments, or nested org trees. A team layer slots in between the organization and its members when a deployment needs one.

- **Plans and SLAs are minimal.** An organization sets response and resolution deadlines per priority, applied the same to every customer. There are no plan tiers that vary those targets, feature access, or usage limits; a plan model and a few checks add that without reshaping the app.
- **Validation covers shape, not every policy.** Forms and actions validate with Zod (lengths, formats, allowed values) from one schema shared by the client and the server. Domain rules a mature product layers on (deduplicating tickets, scanning attachment contents, enforcing per-tenant quotas) attach at the same boundary.
- **Sign-in covers the common cases, not enterprise SSO.** Auth.js wires Google OAuth alongside an email-and-password provider, which is enough for a customer signing up and an agent logging in. SAML or OIDC single sign-on and further providers are configuration on the same setup.
- **Search and embeddings live in Postgres.** Full-text search and pgvector cover retrieval without a separate search or vector service, which can be swapped in later behind the same query functions.

Where a simplification carries operational weight, the chapter that owns it says so: Chapter 25 covers scaling the single database; Chapter 22 hardens the public endpoints. The goal throughout is a system we can read end to end, then extend where our own product needs more.

With the project in mind, let's figure out the best path through the book.

How to Read This Book

Reading Paths by Background

The right starting point depends on where we're coming from:

- **Switching from Java, C#, Python, or another language?** Start at Chapter 1. Part I is written for experienced developers entering the TypeScript and React ecosystem. It moves fast, but it doesn't skip the "why" behind each concept.
- **Experienced TypeScript developer?** Skim Part I or use it as a reference. Start at Part II (Chapter 5) where we create the Next.js project.
- **React developer without Next.js experience?** Skim Chapters 3–4 (React basics and advanced patterns), then dive in from Chapter 5. The shift from client-side React to Next.js Server Components is the biggest learning curve; Chapter 7 is where it clicks.
- **Experienced Next.js developer?** Jump directly to the parts that interest us: Part IV for AI integration, Part V for production patterns, or Part VI for advanced topics like internationalization and AI-assisted development.

Build Along

Our strong recommendation: build HelpDesk AI alongside reading. The learning effect is significantly greater than reading alone. Every chapter has hands-on sections that guide through the implementation step by step, with testable results at the end.

Each chapter builds on the previous one, but the GitHub repo has branches per chapter, so it's possible to jump in at any point without having typed every line from the beginning.

A Note on AI Coding Assistants

This book teaches Next.js by having us write code by hand, understanding what each line does and why. AI coding assistants like Claude Code, GitHub Copilot, or Cursor can accelerate development dramatically, but they work best when the developer understands the underlying patterns.

We deliberately don't let AI tools write our code in Chapters 5–28 so we build that understanding first. In Chapter 28 we do connect Claude Code to the MCP server we have just built, but there it is the client under test rather than the thing writing, one of several that could take that place. Chapter 29 covers AI-assisted development in depth: how to configure Claude Code for Next.js, customize project conventions, and use it effectively once the fundamentals are solid.

Linear vs. Reference

Parts I–III reward linear reading: concepts compound as the project grows. Parts IV–VI can be read more selectively, though they assume familiarity with the project we built in Parts II–III.

The appendices (A–G) are standalone reference material, designed for quick lookups rather than linear reading.

This book works as both a tutorial and a reference. Read it front to back the first time, then keep it within reach for lookups.

Now let's make sure we have everything we need to get started.

Prerequisites

Prior Knowledge

We assume solid programming experience in at least one language and a basic understanding of HTTP, HTML, and CSS. No React or Next.js experience is required; Chapters 1–4 cover the fundamentals.

Target Version

This book targets **Next.js 16.3**. All code examples, configurations, and patterns are based on this version.

Hardware and Software

We need a modern computer with at least 8 GB RAM, though 16 GB makes the production builds of the later chapters considerably more comfortable, plus the following software:

- **Node.js** (LTS version), the runtime that powers the entire JavaScript toolchain, including Next.js
- **pnpm**, our package manager of choice (we'll cover the why and the setup in Chapter 1)
- **VS Code**, our primary IDE. Cursor and Windsurf are forks of it, so the extensions and settings from Appendix E apply there unchanged. If we prefer a different editor (WebStorm, Neovim, Zed), see Appendix F for setup instructions.
- **Git**, for version control

- **Docker**, which runs our PostgreSQL database from Chapter 11 onward and packages the application for self-hosting in Chapter 25



Tip: No Setup Needed Yet

There's no need to install anything before starting the book. Section 1.2 walks through Node.js and pnpm setup step by step.

Accounts

We'll need accounts for the following services, each introduced in the chapter that first needs it. There's no need to sign up for anything before Chapter 1:

- **Google Cloud Console**, for OAuth login (Chapter 12, free)
- **AWS**, for S3 file storage (Chapter 13) and SES email (Chapter 14). The free tier covers both, though signup asks for a card
- **Vercel**, for the AI Gateway (Chapter 15) and deployment (Chapter 24)
- **Sentry** and **Mixpanel**, for error tracking and product analytics (Chapter 23), both on free tiers
- **GitHub**, which Vercel deploys from (Chapter 24) and which provides the container registry and CI pipeline we use for self-hosting (Chapter 25)
- **Anthropic**, for Claude Code (Chapters 28 and 29). This is the one item without a free tier; the cost note below covers it

Cost

Every service above has a free tier that carries the work in this book, with two exceptions worth naming up front. AI usage through the Gateway is billed per token: all of Part IV runs to a few dollars, typically under $5 USD, and new Vercel accounts start with free credits that usually absorb even that. And Claude Code, in Chapters 28 and 29, authenticates with a Claude subscription or an Anthropic API key, the one thing here with no free tier at all. Nothing else in the book depends on it, so those two chapters can be read rather than built along. There won't be a surprise bill.

With our tools ready, let's look at the companion resources.

Companion Code and Resources

The complete source code of the HelpDesk AI project is available on GitHub, organized by chapters and branches. Each chapter has its own branch, so it's possible to jump in at any point or compare our own code with the reference implementation.

- **GitHub repository:** https://github.com/flossels/helpdesk-ai
- **Feedback and contact:** nextjs@flossels.ch, for questions, typos, and suggestions

Conventions in This Book

Throughout the book, we'll encounter several types of callout boxes. Each has a distinct purpose:



Tip

A recommendation based on real-world experience. Time-savers and best practices that make development smoother.



Warning

A stumbling block we've seen developers hit repeatedly. Read these before making the mistake; they'll save us debugging time.



Background

Additional context or a deeper explanation. Safe to skip on first reading without losing the thread. Come back to these later to understand the "why" behind a pattern.



Aside

A reference to an alternative approach, a related topic, or a deeper dive that lives outside the main narrative.



Hands-On

A short exercise where we write the code ourselves instead of reading it. These belong to Part I, where we're building TypeScript and React fluency before the project exists. From Part II on, the project work lives in the numbered sections themselves, each ending in something we can run.

In the chapters, callouts longer than a line carry a heading in the form "Tip: Descriptive Title" so the type and topic are visible at a glance; single-line asides skip it.

Code Examples

Every block that shows a project file carries that file's path as a title above the code, so we always know where it belongs. Blocks without a title are terminal sessions, command output, or short snippets that illustrate a point rather than land in the project. Application code is TypeScript throughout, never JavaScript; configuration, SQL, YAML, and the Prisma schema appear in their own languages. The maximum line width is 85 characters, optimized for reading in both digital and print formats.

Terminal Commands

Terminal commands assume a POSIX shell, which means macOS, Linux, or WSL on Windows. They lean on pipes, line continuations, and shell scripts that PowerShell and `cmd` spell differently, and we do not print a second version of each one. From Chapter 25 onward the project also expects Docker. All package management uses pnpm. If we prefer npm or yarn, the commands are similar but not always identical.

With our conventions established, let's get started. Chapter 1 begins with a rapid tour of TypeScript, the language that ties everything in this book together.

Part I: Fundamentals

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

1. TypeScript: Quick Start for Experienced Developers

Every JavaScript project hits the same wall eventually. A function receives the wrong argument type, a property name is misspelled, or a refactoring breaks code in a file nobody touched, and none of these errors surface until a user reports them in production. As codebases grow, the lack of a type system turns every change into a guessing game. Developers spend more time reading code and running manual tests than building features.

The entire stack we'll use throughout this book (React, Next.js, Prisma, the Vercel AI SDK) is written in TypeScript and expects TypeScript. Without a solid grasp of the type system, every code example from Chapter 2 onward will feel like reading a foreign language with half the words missing.

In this chapter, we'll set up a TypeScript development environment from scratch, work through the type system from primitives to generics, and build a set of standalone examples using HelpDesk AI domain concepts. By the end, we'll be able to write, run, and reason about TypeScript confidently, and recognize every type pattern that appears in the rest of this book.



Tip: Already Know TypeScript?

If TypeScript is part of the daily toolkit, skim or skip this chapter. Here's a quick self-check. Anyone who can confidently explain all of these can move on to Chapter 2:

- Discriminated unions and type narrowing
- `type` vs. `interface`, when to use which
- Utility types: `Pick`, `Omit`, `Partial`, `Record`
- `satisfies` operator
- `import type` and why it matters
- The `strict` flag in `tsconfig.json`

1.1. Why TypeScript?

Consider a function that formats a ticket for display:

```
function formatTicket(ticket) {
  return `${ticket.trackingId}: ${ticket.titel}`
}
```

This code runs without errors. JavaScript doesn't care that we wrote `titel` instead of `title`, or that `ticket` could be `null`. The bug goes unnoticed until a customer sees "HD-0042: undefined" on their screen.

TypeScript catches this class of error before the code runs. It's a strict syntactic superset of JavaScript that adds static type checking at compile time. When we annotate our function with types, the compiler immediately flags the typo:

```
type Ticket = {
  trackingId: string
  title: string
  status: 'OPEN' | 'RESOLVED'
}

function formatTicket(ticket: Ticket): string {
  return `${ticket.trackingId}: ${ticket.titel}`
  //                                  ~~~~~
  // Property 'titel' does not exist
  // on type 'Ticket'. Did you mean 'title'?
}
```

The error appears the moment we type it, in the editor, before we save the file, long before the code reaches a browser or server.

1.1.1. What TypeScript Gives Us

Compile-time type checking. Types are checked when we compile (or when the editor runs its background analysis). Entire categories of bugs (wrong argument types, missing properties, incompatible return values) disappear before the code runs.

IDE superpowers. The type system powers autocompletion, inline documentation, and refactoring tools. Rename a property in one place, and the editor updates every reference across the project. This isn't magic. It's the type information telling the IDE what connects to what.

Types as documentation. A function signature like `createTicket(input: CreateTicketInput): Promise<ActionResult<Ticket>>` tells us exactly what goes in and what comes out: no need to read the implementation or search for comments that may be outdated.

Gradual adoption. TypeScript is a superset of JavaScript. Every `.js` file is (in theory) a valid `.ts` file. We can adopt TypeScript incrementally, file by file. In practice, enabling `strict` mode means some JavaScript patterns need adjustment, but that's a feature, not a bug.

1.1.2. What TypeScript Does Not Do

TypeScript has clear boundaries. Understanding them prevents frustration later.

Types are erased at runtime. The TypeScript compiler removes all type annotations when it produces JavaScript. There is no runtime type checking: if bad data arrives from an API response or a form submission, the types won't stop it.

TypeScript doesn't replace tests. Types prevent a class of bugs (wrong shapes, missing properties, incompatible assignments), but they can't verify business logic, catch off-by-one errors, or confirm that a function returns the right value. Tests remain essential.

TypeScript doesn't run in the browser. Browsers execute JavaScript. TypeScript must be compiled first, a step that Next.js handles automatically for us.

1.1.3. For Developers Coming from Other Typed Languages

If we're coming from Java, C#, Go, or Python's type hints, TypeScript will feel familiar, but some differences can trip us up.

Feature	Java / C#	Go	Python	TypeScript
Type system	Nominal	Structural	Gradual (hints)	Structural
Runtime types	Yes	Yes	Yes (duck typing)	No (erased)
Null handling	Nullable refs	Zero values	`None`	`strictNullChecks`
Generics	Yes	Yes	Yes	Yes
Enums	Full-featured	`iota` constants	`Enum` class	Quirky; prefer string unions
Union types	Sealed interfaces (Java 17+)	No	`Union` (typing)	First-class

The biggest mental shift is **structural typing**. We'll cover it with code examples in Section 1.7.2, along with other common stumbling blocks.

With the "why" covered, let's set up the tools we need to write and run TypeScript.

1.2. Setting Up the Development Environment

TypeScript and the tools around it (the compiler, the dev server, the package manager) all run on Node.js. Before we write a single line of TypeScript, we need a working local environment.

1.2.1. Installing Node.js

Node.js is a JavaScript runtime built on Chrome's V8 engine. It runs JavaScript (and compiled TypeScript) outside the browser. The entire modern JavaScript toolchain (package managers, compilers, bundlers, dev servers, and frameworks like Next.js) runs on Node.js.

Next.js requires Node.js for everything: the dev server (`next dev`), the build process (`next build`), Server Components, Server Actions, and API routes all execute on Node.js. Even when we deploy to Vercel, the production runtime is Node.js.

Install Node.js 24 (LTS) or later from nodejs.org or use a version manager. Verify the installation:

```
$ node -v
v24.x.x
```



Tip: Version Manager

When working on multiple projects with different Node.js versions, use `fnm` (Fast Node Manager) or `nvm`. They switch Node.js versions per project based on a `.node-version` file. For this book, Node.js 24 or later is required.

1.2.2. Choosing a Package Manager: pnpm

A package manager installs, updates, and manages the third-party libraries our project depends on, from React and Next.js to Prisma and the AI SDK. Three options compete for the job: **npm** (ships with Node.js), **yarn** (Meta, 2016), and **pnpm** (performant npm).

This book uses **pnpm** throughout. Here's why:

- **Disk efficiency.** pnpm uses a content-addressable store and hard links. Dependencies are stored once globally and linked into projects, saving gigabytes across multiple projects.
- **Strictness.** pnpm creates a strict `node_modules` structure that prevents accessing packages not declared in `package.json` (so-called phantom dependencies). This catches dependency issues early.
- **Speed.** Consistently faster than npm for installs and CI pipelines.
- **Vercel compatibility.** Vercel auto-detects `pnpm-lock.yaml` and uses pnpm in the build pipeline, no extra configuration needed.

Install pnpm globally:

```
$ npm install -g pnpm
```

Verify:

```
$ pnpm -v
11.x.x
```

For readers coming from npm, here's a quick reference:

npm	pnpm	Notes
`npm install`	`pnpm install`	Install all dependencies
`npm install <pkg>`	`pnpm add <pkg>`	Add a dependency
`npm install -D <pkg>`	`pnpm add -D <pkg>`	Add a dev dependency
`npx <cmd>`	`pnpm dlx <cmd>`	Run a package without installing
`npm run <script>`	`pnpm <script>`	Run a package.json script
`package-lock.json`	`pnpm-lock.yaml`	Lock file (commit this!)



Aside: Using npm or yarn Instead

All commands in this book use pnpm. To use npm or yarn instead, substitute the commands using the table above. The code itself is identical; only the CLI commands differ. We recommend sticking with pnpm to follow along without friction.

1.2.3. VS Code and Extensions

VS Code is the editor we'll use throughout this book. It's free, widely adopted, and has first-class TypeScript support built in, no extensions needed for the examples in this chapter. Install it from code.visualstudio.com.

We'll install project-specific extensions (ESLint, Prettier, Error Lens, and more) when we set up the HelpDesk AI project in Chapter 5.



Aside: IDE Choice

Any editor with TypeScript language server support will work (WebStorm, Neovim with LSP, Zed), but all screenshots and settings references in this book are based on VS Code.

1.2.4. Our First TypeScript Project

Let's create a minimal project to verify the toolchain works. Create a new directory and initialize it:

```
$ mkdir ts-playground && cd ts-playground
$ pnpm init
$ pnpm add -D typescript tsx
```

We install two dev dependencies: `typescript` (the compiler) and `tsx` (a tool that runs `.ts` files directly without a separate compilation step, ideal during development).

The install ends by refusing to run a build script:

```
[ERR_PNPM_IGNORED_BUILDS] Ignored build scripts: esbuild

Run "pnpm approve-builds" to pick which dependencies should be
allowed to run scripts.
```

pnpm does not let a freshly installed package run install scripts on our machine until we allow it. This is a supply-chain guardrail: a new dependency cannot execute code the moment it lands in `node_modules`. The package here is **esbuild**, which `tsx` bundles to compile TypeScript on the fly; its install script sets up the platform binary `tsx` runs on.

pnpm has already written the question down for us, in a new file at the project root:

Figure 1. pnpm-workspace.yaml

```
allowBuilds:
  esbuild: set this to true or false
```

The placeholder is what pnpm is waiting on, and until it becomes `true` or `false` every later `pnpm` command stops at the same message. Since we know why esbuild needs to build, we answer it:

```
$ pnpm approve-builds
```

pnpm lists the packages with pending scripts. Use the arrow keys and the space bar to select `esbuild`, then press Enter to submit the selection. pnpm then asks a second question, and this one matters:

```
? The next packages will now be built: esbuild.
  Do you approve? (y/N)
```

The capital `N` is the default, so pressing Enter here declines and records nothing at all, leaving the placeholder in place. Type `y`. pnpm runs the build script and writes `esbuild: true` into `pnpm-workspace.yaml`, and the warning does not return. We meet this same guardrail again when we scaffold the real project in Chapter 5, where the generator fills that file in for us before pnpm ever asks.



Warning: Approve Only Packages We Recognize

Approve a build script only when the reason it needs to run is clear, the way esbuild needs its binary. When an unfamiliar transitive dependency asks to run a script, the safe answer is `false`, which is a decision like any other and lets the install finish. Leaving the placeholder untouched is the one option that blocks us.

Generate a `tsconfig.json`:

```
$ pnpm tsc --init
```

Since `typescript` is a local dev dependency, `pnpm tsc` runs the locally installed compiler. This creates a `tsconfig.json` with sensible defaults. We'll explore its options in detail in Section 1.6. For now, the defaults are fine.

Create our first TypeScript file:

Figure 2. hello.ts

```
const greeting: string = 'Hello, TypeScript!'
console.log(greeting)
```

Run it:

```
$ pnpm tsx hello.ts
Hello, TypeScript!
```

No compilation step, no `node hello.js`: `tsx` handles everything. During development, this is how we'll run standalone TypeScript files.



Tip: Why Not Compile First?

The traditional workflow is `tsc` to compile, then `node` to run. `tsx` skips the intermediate `.js` file by compiling on the fly. We use `tsx` for development convenience. Next.js has its own compiler and never calls `tsc` directly. We'll see that in Chapter 5.

Our environment is ready. Let's explore the type system that makes TypeScript worth the setup.

1.3. The Type System: From Primitives to Generics

TypeScript's type system is the foundation everything else builds on. We'll start with the basics and work our way up to generics, the most versatile pattern in the language.

1.3.1. Primitive Types

TypeScript has the same primitives as JavaScript, but with explicit type annotations:

```
const name: string = 'HelpDesk AI'
const ticketCount: number = 42
const isResolved: boolean = true
const nothing: null = null
const missing: undefined = undefined
```

Most of the time, we don't need the annotations. TypeScript infers types from the assigned value:

```
const name = 'HelpDesk AI'     // inferred: string
const ticketCount = 42         // inferred: number
const isResolved = true        // inferred: boolean

function getTicketCount() {
  return 42                    // return type inferred: number
}
```

When to annotate, when to infer: Let TypeScript infer when the type is obvious from the value. Add explicit annotations for function parameters (TypeScript can't infer those) and when the inferred type is too broad or too narrow for our needs.

1.3.2. `any` vs. `unknown`: The Escape Hatches

Two special types exist for situations where we don't know the type upfront.

`any` disables type checking entirely. TypeScript treats an `any` value as compatible with everything (no errors, no autocompletion, no safety):

```
let data: any = 'hello'
data = 42          // no error
data.foo.bar.baz   // no error (but crashes at runtime)
```

`unknown` is the safe alternative. It accepts any value, but we must narrow the type before using it:

```
let data: unknown = 'hello'

// data.toUpperCase()  // Error: 'data' is of type 'unknown'

if (typeof data === 'string') {
  data.toUpperCase()   // OK: TypeScript knows it's a string
}
```

The rule is straightforward: **avoid `any`**. When we don't know the type, use `unknown` and narrow it. This is especially important for `catch` blocks, where the error is `unknown` by default:

```
try {
  await fetchTickets()
} catch (error: unknown) {
  if (error instanceof Error) {
    console.error(error.message)
  }
}
```

1.3.3. Arrays and Tuples

Arrays use the shorthand `Type[]` syntax:

```
const ids: string[] = ['abc', 'def', 'ghi']
const counts: number[] = [1, 2, 3]
```

Tuples are fixed-length arrays where each position has a specific type:

```
// [trackingId, priority level]
const ticket: [string, number] = ['HD-0042', 3]

const [id, priority] = ticket
// id: string, priority: number
```

Tuples with `readonly` prevent accidental modifications:

```ts
const config: readonly [string, number] = ['host', 8080]
// config[0] = 'other'  // Error: read-only
```

1.3.4. Union Types

A union type represents a value that can be one of several types. We write it with the pipe `|` operator:

```ts
let id: string | number = 'abc-123'
id = 42  // also valid
```

Union types become far more interesting with literal types (specific string or number values):

```ts
type TicketStatus = 'OPEN' | 'IN_PROGRESS' | 'WAITING' | 'RESOLVED' | 'CLOSED'

let status: TicketStatus = 'OPEN'       // OK
// status = 'PENDING'                   // Error
```

This is how we model constrained values in TypeScript, and it's the pattern we'll use for statuses, priorities, roles, and scopes throughout HelpDesk AI.

String unions often come with a lookup object: a mapping from each value to a label, a color, or a configuration. We want TypeScript to guarantee that the object covers every union member. The `satisfies` operator does that:

```ts
type TicketPriority = 'LOW' | 'MEDIUM' | 'HIGH' | 'URGENT'

const PRIORITY_COLORS = {
  LOW: '#64748b',
  MEDIUM: '#3b82f6',
  HIGH: '#f59e0b',
  URGENT: '#ef4444'
} satisfies Record<TicketPriority, string>
```

If we add a fifth priority to the union and forget to add it here, the compiler tells us. If we mistype a key, the compiler tells us.

A plain annotation (`const PRIORITY_COLORS: Record<TicketPriority, string> = { ... }`) checks those same two things, so for an object this simple the two forms are interchangeable. The difference shows up as soon as the target type is broader than what we actually wrote. An annotation *replaces* the literal's type with the declared one; `satisfies` checks the literal against the declared type and then keeps what the literal knows:

```ts
type PriorityStyle = {
  color: string
  weight: 'normal' | 'bold'
}

// The annotation replaces the literal's type
const A: Record<TicketPriority, PriorityStyle> = {
  LOW: { color: '#64748b', weight: 'normal' },
  // ...
}
A.LOW.weight  // type: 'normal' | 'bold'

// satisfies keeps it
const B = {
  LOW: { color: '#64748b', weight: 'normal' },
  // ...
} satisfies Record<TicketPriority, PriorityStyle>
B.LOW.weight  // type: 'normal'
```

Note where this does and doesn't apply. `weight` is declared as a union of two literals, so there is a narrower answer for `satisfies` to preserve. `color` is declared as `string`, and `string` is as narrow as that field ever gets: `A.LOW.color` and `B.LOW.color` are both `string`. `satisfies` never *adds* precision, it only declines to throw precision away.

That is why it is our default for labeled lookup objects in HelpDesk AI (statuses, scopes, roles, and more): it gives the same completeness check as an annotation and costs us nothing when there is nothing extra to keep.



Background: Why Not `as const`?

Other TypeScript codebases often use `as const`. It also preserves literal types, and more of them, since it freezes every value to its literal. What it doesn't do is check the object against a target type at all, so nothing catches a missing or misspelled key, and it makes the whole object deeply readonly whether we wanted that or not. `satisfies` keeps the structural check. Throughout this book, we use `satisfies` instead of `as const` for typed lookup objects.

1.3.5. Discriminated Unions

Discriminated unions are the most important pattern in TypeScript for modeling states. Each variant in the union has a shared property (the "discriminant") with a different literal value:

```typescript
type TicketEvent =
  | { kind: 'created', ticketId: string }
  | { kind: 'assigned', ticketId: string, agentId: string }
  | { kind: 'resolved', ticketId: string, resolution: string }
```

The `kind` property is the discriminant. When we check it, TypeScript automatically narrows the type in each branch:

```typescript
function describeEvent(event: TicketEvent): string {
  switch (event.kind) {
    case 'created':
      return `Ticket ${event.ticketId} was created`
    case 'assigned':
      // TypeScript knows agentId exists here
      return `Assigned to ${event.agentId}`
    case 'resolved':
      // TypeScript knows resolution exists here
      return `Resolved: ${event.resolution}`
  }
}
```

No type assertion, no casting: TypeScript narrows automatically based on the discriminant check. We'll use this pattern extensively: for API responses (Section 1.3.8), for Server Action results in HelpDesk AI, and anywhere we need to model multiple states with different shapes.

1.3.6. Intersection Types

Where union types mean "this OR that", intersection types mean "this AND that":

```typescript
type WithTimestamps = {
  createdAt: Date
  updatedAt: Date
}

type Ticket = {
  id: string
  subject: string
  status: TicketStatus
}

type TicketWithTimestamps = Ticket & WithTimestamps
// Has all properties from both types
```

Intersection types are useful for composing types from smaller building blocks, defining reusable "mixins" that we attach to different base types:

```ts
type WithAudit = {
  createdBy: string
  updatedBy: string
}

type AuditedTicket = Ticket & WithTimestamps & WithAudit
// Has id, subject, status, createdAt, updatedAt,
// createdBy, updatedBy
```

We'll see this pattern often when combining base types with additional metadata. Use intersections when the pieces are independently useful; use a single type when the fields always belong together.

1.3.7. Type Narrowing

Type narrowing is how TypeScript figures out the specific type within a broader one. We've already seen it with discriminated unions. Here are the other common patterns:

`typeof` checks for primitives:

```ts
function formatId(id: string | number): string {
  return typeof id === 'string' ? id.toUpperCase() : `#${id}`
}
```

`instanceof` checks for class instances:

```ts
function formatError(error: unknown) {
  return error instanceof Error ? error.message : String(error)
}
```

Truthiness checks for nullable values:

```ts
function greet(name: string | null): string {
  return name ? `Hello, ${name}` : 'Hello, guest'
}
```

The `in` operator for objects:

```ts
type Agent = { name: string; role: 'AGENT' }
type Customer = { name: string; email: string }

function identify(user: Agent | Customer): string {
  return 'role' in user ? `Agent: ${user.name}` : `Customer: ${user.email}`
}
```

TypeScript tracks narrowing through the control flow of our code. Once we've checked a condition, the type is narrowed in the corresponding branch, no manual assertions needed.

1.3.8. Generics

Generics let us write functions, types, and interfaces that work with any type while preserving type information. Consider a function that wraps a value in an array:

```
// Without generics, loses type information
function toArray(value: unknown): unknown[] {
  return [value]
}

const result = toArray('hello')
// result: unknown[], we lost the string type
```

With generics, we keep the type:

```
function toArray<T>(value: T): T[] {
  return [value]
}

const result = toArray('hello')
// result: string[], type preserved
```

The <T> is a type parameter, a placeholder that TypeScript fills in when the function is called. We don't need to specify it explicitly; TypeScript infers it from the argument.

Generic constraints limit which types are acceptable:

```
function getProperty<T extends { id: string }>(
  obj: T,
  key: keyof T
) {
  return obj[key]
}

const ticket = { id: 'abc', subject: 'Help!' }
getProperty(ticket, 'subject')  // OK
// getProperty(42, 'toString')  // Error: number has no 'id'
```

Two pieces of syntax are doing the work here. The `extends` keyword means "T must have at least an `id` property of type `string`." And `keyof T` is the union of T's property names, so `key` accepts `'id'` or `'subject'` for the object above and nothing else. We'll lean on `keyof` again for typed lookup objects later in the book.

Generic types work the same way as generic functions:

```typescript
type ApiResponse<T> =
  | { success: true; data: T }
  | { success: false; error: string }

// Usage:
type TicketResponse = ApiResponse<{
  ticketId: string
  trackingId: string
}>

// Expands to:
// | { success: true; data: { ticketId: string; trackingId: string } }
// | { success: false; error: string }
```

This `ApiResponse<T>` pattern is a preview of the `ActionResult<T>` type we'll use for every Server Action in HelpDesk AI. Generics make it reusable across all actions while keeping each one fully typed.

We've covered the core type system. Next, we'll tackle a question that comes up in every TypeScript project: should we use `type` or `interface`?

1.4. Interfaces vs. Types: When to Use Which

TypeScript offers two ways to define object shapes: `type` and `interface`. Both can do the same thing in most cases, and the debate over which to use can get heated. Let's look at the differences that matter in practice.

1.4.1. What Both Can Do

Both define object shapes and can be used interchangeably for most purposes:

```typescript
// As a type
type TicketA = {
  id: string
  subject: string
  status: TicketStatus
}

// As an interface
interface TicketB {
  id: string
  subject: string
  status: TicketStatus
}
```

Both support extending (adding fields):

```typescript
// Type: via intersection
type AssignedTicketA = TicketA & { agentId: string }

// Interface: via extends
interface AssignedTicketB extends TicketB {
  agentId: string
}
```

1.4.2. Where They Differ

The differences are few, but meaningful:

Feature	`interface`	`type`
Declaration merging	Yes, two `interface` blocks with the same name merge automatically	No, duplicate name is an error
Union types	Cannot define unions	Can define unions: `type A = B | C`
Mapped types	Cannot use mapped types directly	Full support
`extends` keyword	Yes	Via intersection (&)
Primitives, tuples	Cannot represent	Can represent

Declaration merging is the most significant practical difference. When two `interface` declarations share the same name, TypeScript merges them:

```typescript
interface User {
  id: string
  name: string
}

interface User {
  email: string
}

// Result: User has id, name, AND email
```

This is rarely what we want in application code, but it's essential when extending third-party library types. For example, Auth.js uses declaration merging to let us extend the `Session` type. We'll use this technique when we set up authentication for HelpDesk AI.

1.4.3. Our Convention

For this book and the HelpDesk AI project, we use **`type` as the default**:

- `type` for object shapes, function signatures, input/output types, unions

- `interface` only when we need declaration merging (e.g., extending third-party types)

The choice matters less than consistency. Pick one, stick with it. We picked `type`.

```
// Our convention: type for everything
type Ticket = {
  id: string
  subject: string
  status: TicketStatus
}

type CreateTicketInput = {
  subject: string
  description: string
  priority?: TicketPriority
}

// Exception: interface when extending library types
interface Session {
  user: {
    id: string
    organizationId: string
    scopes: string[]
  }
}
```

With the `type` vs. `interface` question settled, let's look at how TypeScript helps us avoid writing the same shapes over and over.

1.5. Utility Types: Reusing What Already Exists

In a real project, types often overlap. A `Ticket` has 15 fields, but the list view needs only 5 of them. An update form sends the same fields as the create form, but all of them are optional. Writing each variation by hand leads to duplication and drift.

TypeScript's utility types solve this by deriving new types from existing ones.

1.5.1. `Partial<T>`: All Fields Optional

`Partial<T>` makes every property in `T` optional. Useful for update operations where only changed fields are sent:

```typescript
type Ticket = {
  id: string
  subject: string
  status: TicketStatus
  priority: TicketPriority
}

type TicketUpdate = Partial<Ticket>
// {
//   id?: string
//   subject?: string
//   status?: TicketStatus
//   priority?: TicketPriority
// }
```

1.5.2. `Required<T>`: All Fields Required

`Required` is the inverse of `Partial`. It forces all optional properties to be required:

```typescript
type Config = {
  host?: string
  port?: number
  debug?: boolean
}

type ResolvedConfig = Required<Config>
// All three fields are now required
```

1.5.3. `Pick<T, K>`: Select Specific Fields

`Pick` creates a type with only the selected properties. Ideal for list views and summaries:

```typescript
type Ticket = {
  id: string
  trackingId: string
  subject: string
  description: string
  status: TicketStatus
  priority: TicketPriority
  createdAt: Date
  updatedAt: Date
}

type TicketListItem = Pick<Ticket, 'id' | 'trackingId' | 'subject' | 'status' |
↪  'priority'>
// Only the five fields we need for the list
```

1.5.4. `Omit<T, K>`: Exclude Specific Fields

`Omit` is the inverse of `Pick`; it creates a type with everything except the listed properties:

```
type User = {
  id: string
  name: string
  email: string
  passwordHash: string
}

type PublicUser = Omit<User, 'passwordHash'>
// { id: string; name: string; email: string }
```

1.5.5. `Record<K, V>`: Key-Value Maps

`Record` creates a type where all keys are of type `K` and all values are of type `V`. We saw it combined with `satisfies` in Section 1.3.4 for lookup objects. It's also useful as a standalone type annotation:

```
type Role = 'OWNER' | 'ADMIN' | 'AGENT' | 'VIEWER'

const DEFAULT_TICKET_LIMIT: Record<Role, number> = {
  OWNER: Infinity,
  ADMIN: Infinity,
  AGENT: 50,
  VIEWER: 20
}
```

The compiler ensures every key in the union is present: if we add a new role later, TypeScript tells us we missed an entry. Here the values are plain numbers with nothing narrower to preserve, so an annotation is as good as `satisfies`. Reach for `satisfies` (as in Section 1.3.4) once the value type has literal detail worth keeping.

1.5.6. `Readonly<T>`: Immutable Objects

`Readonly` makes all properties read-only:

```
type AppConfig = Readonly<{
  apiUrl: string
  maxRetries: number
}>

const config: AppConfig = {
  apiUrl: 'https://api.example.com',
  maxRetries: 3
}

// config.apiUrl = 'other'  // Error: read-only
```

1.5.7. `ReturnType<T>`: Extract a Function's Return Type

`ReturnType` extracts the return type of a function type. Combined with `typeof`, it's useful for deriving types from existing functions without duplicating them:

```
function createTicket(): { id: string, trackingId: string, status: 'OPEN' } {
  return {
    id: crypto.randomUUID(),
    trackingId: 'HD-0001',
    status: 'OPEN'
  }
}

type NewTicket = ReturnType<typeof createTicket>
// { id: string; trackingId: string; status: 'OPEN' }
```

1.5.8. Composing Utility Types

Two more utilities operate on unions rather than object shapes: `Exclude<T, U>` and `Extract<T, U>` remove or keep specific union members:

```
type TicketStatus = 'OPEN' | 'IN_PROGRESS' | 'WAITING' | 'RESOLVED' | 'CLOSED'

type ActiveStatus = Exclude<TicketStatus, 'CLOSED'>
// 'OPEN' | 'IN_PROGRESS' | 'WAITING' | 'RESOLVED'

type TerminalStatus = Extract<TicketStatus, 'RESOLVED' | 'CLOSED'>
// 'RESOLVED' | 'CLOSED'
```

Utility types can be combined. When the combination becomes hard to read, extract it into a named type:

```
// Readable composition
type TicketSummary = Pick<Ticket, 'id' | 'subject' | 'status'>

// Getting complex: extract a name
type TicketFormData = Omit<Partial<Ticket>, 'id' | 'createdAt' | 'updatedAt'>
```

The rule of thumb: if a composition is used more than once or reads like a puzzle, give it a name.



Aside: The Full Utility-Type Catalog

We cover the utility types HelpDesk AI relies on. TypeScript ships more, and the mapped and conditional types that power them are beyond this quick start. The full catalog, and everything else this chapter compresses, is in the official TypeScript Handbook at typescriptlang.org/docs/handbook. Appendix A collects the utility types used in this book.

We've seen how to build types. Now let's look at how to organize them across files.

1.6. Modules, Imports, and `tsconfig.json`

1.6.1. ES Modules in TypeScript

TypeScript uses the same module system as modern JavaScript: ES Modules with `import` and `export`.

Named exports, the convention for this book:

```
export type Ticket = {
  id: string
  subject: string
  status: TicketStatus
}

export type CreateTicketInput = {
  subject: string
  description: string
}
```

Importing:

```
import { createTicket } from './actions/createTicket'
```

Re-exports bundle multiple modules into a single entry point:

```
export type { Ticket, CreateTicketInput } from './types'
export { createTicket } from './actions/createTicket'
```

The `export type` on the first line is not decoration. `Ticket` and `CreateTicketInput` are types, and our configuration requires saying so, which is what the next section is about.

1.6.2. Type-Only Imports

TypeScript provides a special `import type` syntax for imports that are only used as types, not as values at runtime:

```
import type { Ticket } from './types'
import type { TicketStatus } from '@prisma/client'
```

Why does this matter? Regular imports can create side effects and prevent tree-shaking (the process of removing unused code from the bundle). Type-only imports are guaranteed to be erased during compilation; they never end up in the JavaScript output. They also prevent circular dependency issues.

Our convention: **always use `import type` for types**. We'll enforce this with an ESLint rule once we set up the HelpDesk AI project.

1.6.3. Understanding `tsconfig.json`

The `tsconfig.json` file configures the TypeScript compiler. Running `pnpm tsc --init` back in Section 1.2.4 generated one for us, with a curated set of options switched on and the rest left as commented hints. Here is the generated file, trimmed to the active options:

Figure 3. tsconfig.json (generated by tsc –init)

```jsonc
{
  "compilerOptions": {
    // Environment Settings
    "module": "nodenext",
    "target": "esnext",
    "types": [],

    // Other Outputs
    "sourceMap": true,
    "declaration": true,
    "declarationMap": true,

    // Stricter Typechecking Options
    "noUncheckedIndexedAccess": true,
    "exactOptionalPropertyTypes": true,

    // Recommended Options
    "strict": true,
    "jsx": "react-jsx",
    "verbatimModuleSyntax": true,
    "isolatedModules": true,
    "noUncheckedSideEffectImports": true,
    "moduleDetection": "force",
    "skipLibCheck": true
  }
}
```

Each active option, in the order the file lists them:

Option	Value	Why
`module`	`"nodenext"`	Resolves imports the way modern Node.js does, honoring the `type` field in `package.json`. It sets the matching resolution strategy too, so there is no separate `moduleResolution` entry.
`target`	`"esnext"`	Compiles to the latest JavaScript. We run on current Node.js, so there is nothing to down-level.

Option	Value	Why
`types`	`[]`	Don't pull in any `@types/*` global declarations automatically; add them explicitly when needed. Keeps the global namespace clean.
`sourceMap`	`true`	Emits source maps so debuggers and stack traces point back to the original `.ts` lines.
`declaration`	`true`	Emits `.d.ts` type declarations alongside the JavaScript, which matters when a package is consumed as a library.
`declarationMap`	`true`	Adds source maps for the declarations, so "Go to definition" lands in the `.ts` source rather than the generated `.d.ts`.
`noUncheckedIndexedAccess`	`true`	Array access and object indexing return `T | undefined`. Prevents silent `undefined` bugs.
`exactOptionalPropertyTypes`	`true`	An optional property (`prop?: T`) can be absent but not set to `undefined` explicitly. Keeps "missing" and "present but undefined" distinct.
`strict`	`true`	Enables all strict type-checking options at once. Non-negotiable for production code.
`jsx`	`"react-jsx"`	Uses React's automatic JSX transform; no need to `import React` in every file.
`verbatimModuleSyntax`	`true`	Leaves imports and exports exactly as written and requires `import type` for type-only imports (Section 1.6.2), so the emitted JavaScript is predictable.
`isolatedModules`	`true`	Requires every file to be compilable on its own, which is what bundlers and fast transpilers (SWC, esbuild) need.
`noUncheckedSideEffectImports`	`true`	Flags a side-effect-only import (`import './x'`) when the module cannot be found, instead of silently ignoring it.

Option	Value	Why
`moduleDetection`	`"force"`	Treats every file as a module, so a file without imports or exports never becomes an accidental global script.
`skipLibCheck`	`true`	Skips type-checking `.d.ts` files. Speeds up compilation without reducing safety for our own code.

The file also lists a **Style Options** group (`noUnusedLocals`, `noImplicitReturns`, `noFallthroughCasesInSwitch`, and others) commented out. These are optional lint-style checks a project can opt into; we leave them off here.



Warning: `strict: true` Is Non-Negotiable

The `strict` flag enables `strictNullChecks`, `strictFunctionTypes`, `noImplicitAny`, and several other checks. Turning it off or leaving individual checks disabled creates a false sense of safety. Every project in this book runs with `strict: true`.

These defaults give us a strict, modern TypeScript setup for the playground. Every project we build later starts from a `tsconfig.json` too, generated for us by its own tooling, but the options mean the same thing wherever they appear. Two of them, `declaration` and `declarationMap`, produce the `.d.ts` files worth understanding on their own.

1.6.4. Declaration Files

Declaration files (`.d.ts`) contain type definitions without implementation. We encounter them in two places:

Third-party libraries. Many npm packages ship their own `.d.ts` files (Prisma, the AI SDK, Next.js itself). For libraries that don't, the `@types` scope on npm provides community-maintained declarations, for example, `@types/node` for Node.js APIs.

Generated types. Tools like Prisma generate `.d.ts` files based on our schema. Next.js generates route types when `typedRoutes` is enabled. These are consumed automatically. We rarely need to think about them.

Writing one ourselves stays rare, and when we do it is to extend a library's types rather than to describe a package that has none. For now it is enough to know they exist and what they do when we see them in `node_modules`.

Understanding how modules and configuration work completes the picture of TypeScript's architecture. Before we move on, let's address the stumbling blocks that catch developers coming from other languages.

1.7. Common Stumbling Blocks When Switching

Every language has its quirks, and TypeScript is no exception. This section covers the most common pain points for developers switching from JavaScript, Java/C#, or Python.

1.7.1. From JavaScript to TypeScript

"My code was working, now TypeScript complains." The adjustment period is real. TypeScript's type checker flags code that JavaScript runs without error. That's the point: those errors were always there, hidden.

`strictNullChecks` is the most impactful strict-mode flag. With it enabled, `null` and `undefined` are not assignable to other types:

```
// Without strictNullChecks (BAD: don't do this):
const agentName: string = null  // allowed

// With strictNullChecks (our setting):
const assignee: string = null         // Error!
const reviewer: string | null = null  // OK, explicit
```

This forces us to handle `null` and `undefined` explicitly, which prevents an enormous class of runtime errors.

The `!` non-null assertion is a tempting escape hatch:

```
const element = document.getElementById('app')
// element is HTMLElement | null

element!.textContent = 'Hello'
// The ! tells TypeScript: "trust me, this isn't null"
```

The problem: "trust me" is exactly the kind of reasoning that causes bugs. Prefer a null check:

```
const element = document.getElementById('app')
if (element) element.textContent = 'Hello'
```

Use `!` only when we're certain the value isn't null and the alternative is impractical, which is rare.

Third-party libraries without types. Most popular packages ship their own type definitions or have community-maintained ones in the `@types` scope (e.g., `pnpm add -D @types/node`). Occasionally, a package has no types at all. In that case, a declaration file silences the error. TypeScript picks up any `.d.ts` file that `tsconfig.json` includes, so the file name is for us rather than for the compiler; the convention is to name it after the package and keep it with the project's other declarations, in a `types/` directory:

```
declare module 'untyped-lib'
```

This tells TypeScript "this module exists, treat its exports as `any`." It's a last resort, but it unblocks us without disabling type checking globally.

1.7.2. From Java/C# to TypeScript

Structural typing vs. nominal typing. This is the biggest conceptual difference. In Java, two classes with identical fields are different types:

```
// Java
class Point { int x; int y; }
class Coordinate { int x; int y; }

Point p = new Coordinate(1, 2);  // Compile error!
```

In TypeScript, types are compared by shape, not by name:

```
type Point = { x: number, y: number }
type Coordinate = { x: number, y: number }

const p: Point = { x: 1, y: 2 }
const c: Coordinate = p  // OK, same shape
```

To developers used to nominal typing, this feels wrong at first. It's a deliberate design choice that matches how JavaScript objects actually work.

No runtime type information. In Java, `instanceof` works for any class. In TypeScript, `instanceof` works for classes but not for `type` or `interface`, because types are erased at runtime (as we discussed in Section 1.1.2):

```
type InternalNote = { text: string, agentId: string }
type CustomerReply = { text: string, email: string }

type Message = InternalNote | CustomerReply

function isInternal(message: Message) {
  // message instanceof InternalNote  // ERROR: not a class
  return 'agentId' in message         // Use 'in' check instead
}
```

Enum pitfalls. TypeScript enums exist but have quirks. Numeric enums allow reverse mapping (which leaks implementation details), tree-shaking doesn't always work, and they introduce a runtime object for what should be a compile-time concept. We prefer the string union types we introduced in Section 1.3.4:

```typescript
// Prefer this:
type Role = 'OWNER' | 'ADMIN' | 'AGENT' | 'VIEWER'

// Over this:
enum Role {
  OWNER = 'OWNER',
  ADMIN = 'ADMIN',
  AGENT = 'AGENT',
  VIEWER = 'VIEWER'
}
```

String unions are simpler, tree-shakeable, and work naturally with discriminated unions. When we need a lookup object (mapping a value to a label or color), we combine `Record` with the `satisfies` operator. We'll build a complete example of this pattern in Section 1.8.

1.7.3. From Python to TypeScript

Type hints vs. enforced types. Python's type hints are optional suggestions: the interpreter ignores them entirely. TypeScript types are enforced by the compiler. Code that violates the types doesn't compile.

Structural typing IS duck typing, formalized. Python developers are used to duck typing ("if it quacks like a duck..."). TypeScript's structural type system is the same idea, expressed formally: if an object has the right shape, it fits the type.

The `any` escape hatch. Python developers often reach for `Any` when type annotations get complicated. TypeScript's `any` works the same way and is equally dangerous. As we covered in Section 1.3.2, prefer `unknown` and narrow the type explicitly before accessing properties.

With the type system and common pitfalls covered, let's put everything together using HelpDesk AI domain concepts.

1.8. Building the TypeScript Playground



Hands-On: Building the TypeScript Playground

Let's apply what we've covered by building four standalone TypeScript files. Each one demonstrates a key pattern using HelpDesk AI domain concepts: ticket statuses, API responses, user types, and scope definitions.

We'll work in the `ts-playground` project we created in Section 1.2.

1.8.1. Step 1: Ticket Status (String Unions and Type Narrowing)

Create a file that models ticket statuses with discriminated unions and demonstrates type narrowing:

Figure 4. ticketStatus.ts

```ts
type TicketStatus = 'OPEN' | 'IN_PROGRESS' | 'WAITING' | 'RESOLVED' | 'CLOSED'
type TicketPriority = 'LOW' | 'MEDIUM' | 'HIGH' | 'URGENT'

type Ticket = {
  trackingId: string
  subject: string
  status: TicketStatus
  priority: TicketPriority
  assigneeId: string | null
}

function getStatusMessage(ticket: Ticket): string {
  switch (ticket.status) {
    case 'OPEN':
      return `${ticket.trackingId} is waiting`
    case 'IN_PROGRESS':
      return `${ticket.trackingId} is being handled`
    case 'WAITING':
      return `${ticket.trackingId} is waiting for a response`
    case 'RESOLVED':
      return `${ticket.trackingId} has been resolved`
    case 'CLOSED':
      return `${ticket.trackingId} is closed`
  }
}

function requiresAssignee(ticket: Ticket): boolean {
  return (
    ticket.status === 'IN_PROGRESS' &&
    ticket.assigneeId === null
  )
}

// Test it
const ticket: Ticket = {
  trackingId: 'HD-0042',
  subject: 'Login not working',
  status: 'IN_PROGRESS',
  priority: 'HIGH',
  assigneeId: null
}

console.log(getStatusMessage(ticket))
console.log('Needs assignee:', requiresAssignee(ticket))
```

Run it:

```
$ pnpm tsx ticketStatus.ts
HD-0042 is being handled
Needs assignee: true
```

The `switch` covers all five status values. If we add a sixth status later and forget to handle it, TypeScript will flag the missing case: the function promises to return `string`, but the new branch would fall through without a return value. Under the hood, TypeScript narrows the type of `ticket.status` in each `case` branch. When all members of the union are handled, the type after the last `case` is `never`, a type that represents "this can't happen." If we add a sixth status, the type after the last branch is no longer `never`, and TypeScript knows we missed something.

1.8.2. Step 2: API Response (Generics)

Create a generic response type that previews the pattern we'll use for Server Actions:

Figure 5. apiResponse.ts

```ts
type ApiResponse<T> = { success: true, data: T } | { success: false, error: string }

function createTicket(subject: string): ApiResponse<{
  ticketId: string,
  trackingId: string
}> {
  if (subject.length < 5) {
    return {
      success: false,
      error: 'Subject must be at least 5 characters.'
    }
  }

  return {
    success: true,
    data: {
      ticketId: crypto.randomUUID(),
      trackingId: 'HD-0001'
    }
  }
}

// Test both paths
const fail = createTicket('Hi')
if (!fail.success) console.log('Error:', fail.error)

const ok = createTicket('Login page broken')
if (ok.success) console.log('Created:', ok.data.trackingId)
```

Run it:

```
$ pnpm tsx apiResponse.ts
Error: Subject must be at least 5 characters.
Created: HD-0001
```

The generic `T` makes `ApiResponse` reusable: each function defines its own success payload, and the consumer always knows whether to access `.data` or `.error`.

1.8.3. Step 3: User Types (Utility Types)

Apply `Pick`, `Omit`, `Partial`, and `Readonly` to a `User` type:

Figure 6. userTypes.ts

```ts
type User = {
  id: string
  name: string
  email: string
  passwordHash: string
  role: 'OWNER' | 'ADMIN' | 'AGENT' | 'VIEWER'
  organizationId: string
  createdAt: Date
}

// For displaying in a list, only what the UI needs
type UserListItem = Pick<User, 'id' | 'name' | 'email' | 'role'>

// For profile updates, only editable fields, all optional
type UserUpdate = Partial<Pick<User, 'name' | 'email'>>

// Demonstrate the types
const listItem: UserListItem = {
  id: 'usr-001',
  name: 'Alice',
  email: 'alice@example.com',
  role: 'AGENT'
}

const update: UserUpdate = {
  name: 'Alice B.'
  // email is optional; we can omit it
}

console.log('List item:', listItem)
console.log('Update:', update)
```

Run it:

```
$ pnpm tsx userTypes.ts
List item: {
  id: 'usr-001',
  name: 'Alice',
  email: 'alice@example.com',
  role: 'AGENT'
}
Update: { name: 'Alice B.' }
```

Instead of writing four separate types by hand, we derived them from a single `User` type. When `User` gains a new field, the derived types stay in sync automatically.

1.8.4. Step 4: Scope Definitions (`satisfies` and `Record`)

Build a typed lookup object covering four of the permission scopes we'll use in HelpDesk AI:

Figure 7. scopeDefinitions.ts

```ts
type Scope =
  | 'tickets:read'
  | 'tickets:write'
  | 'knowledge:read'
  | 'ai:use'

type ScopeInfo = {
  label: string
  description: string
  category: 'tickets' | 'knowledge' | 'ai'
}

const SCOPES = {
  'tickets:read': {
    label: 'View Tickets',
    description: 'Read and search tickets',
    category: 'tickets'
  },
  'tickets:write': {
    label: 'Edit Tickets',
    description: 'Create and reply to tickets',
    category: 'tickets'
  },
  'knowledge:read': {
    label: 'View Articles',
    description: 'Read knowledge base articles',
    category: 'knowledge'
  },
  'ai:use': {
    label: 'Use AI Copilot',
    description: 'Generate suggestions and summaries',
    category: 'ai'
```

```
  }
} satisfies Record<Scope, ScopeInfo>

// Type-safe access; autocomplete works
console.log(SCOPES['tickets:read'].label)
// 'View Tickets'

// Helper: check if a user has a scope
function hasScope(userScopes: Scope[], required: Scope): boolean {
  return userScopes.includes(required)
}

const viewerScopes: Scope[] = [
  'tickets:read',
  'knowledge:read'
]

console.log('Can read tickets:', hasScope(viewerScopes, 'tickets:read'))
console.log('Can write tickets:', hasScope(viewerScopes, 'tickets:write'))
```

Run it:

```
$ pnpm tsx scopeDefinitions.ts
View Tickets
Can read tickets: true
Can write tickets: false
```

The `satisfies Record<Scope, ScopeInfo>` check ensures that every scope in our union has a corresponding entry. If we add a new scope to the `Scope` type, the compiler tells us we're missing it in `SCOPES`. We keep the playground at four scopes; HelpDesk AI's real permission system defines fourteen, built on exactly this pattern.

Summary

In this chapter, we:

- Set up a TypeScript development environment with Node.js, pnpm, VS Code, and `tsx` for running TypeScript directly
- Explored the type system from primitives and unions to discriminated unions and generics
- Built a generic `ApiResponse<T>` type that previews the `ActionResult<T>` pattern used throughout HelpDesk AI
- Established our convention: `type` over `interface`, string unions over enums, `satisfies` over `as const`
- Applied utility types (`Pick`, `Omit`, `Partial`, `Record`) to derive new types from existing ones instead of duplicating
- Configured `tsconfig.json` and understood why `strict: true` is non-negotiable

- Built four standalone TypeScript files that demonstrate every pattern using HelpDesk AI domain concepts

We've covered TypeScript's type system, but modern TypeScript code also relies heavily on ES2015+ JavaScript features: destructuring, spread operators, `async/await`, and array methods that appear on every page of this book. In the next chapter, we'll bring those up to speed.

2. Modern JavaScript for TypeScript Developers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

2.1. Destructuring: Unpacking Data

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

2.1.1. Renaming During Destructuring

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

2.1.2. Default Values

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

2.1.3. Nested Destructuring

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

2.1.4. Array Destructuring

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

2.1.5. Destructuring in Function Parameters

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

2.2. Spread and Rest: Copying and Collecting

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

2.2.1. The Spread Operator (`...`)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

2.2.2. Rest Parameters

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

2.2.3. TypeScript and Spread

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

2.3. Template Literals

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

2.4. Arrow Functions and Lexical `this`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

2.4.1. Arrow Function Syntax

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

2.4.2. When to Use Which

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

2.5. Promises, `async`/`await`, and Error Handling

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

2.5.1. The Problem: Asynchronous Operations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

2.5.2. Promises

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

2.5.3. `async/await`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

2.5.4. Error Handling with `try/catch`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

2.5.5. Throwing vs. Returning Errors

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

2.5.6. `finally`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

2.6. Array Methods: Thinking in Transformations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

2.6.1. The Problem

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

2.6.2. `map`: Transforming Every Element

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

2.6.3. `filter`: Selecting Elements

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

2.6.4. `find` and `findIndex`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

2.6.5. `some` and `every`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

2.6.6. `reduce`: Aggregating Data

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

2.6.7. `flatMap`: Map and Flatten

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

2.6.8. Deduplicating with `Set`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

2.7. Optional Chaining and Nullish Coalescing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

2.7.1. Optional Chaining (`?.`)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

2.7.2. Nullish Coalescing (`??`)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

2.7.3. Nullish Coalescing Assignment (`??=`)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

2.8. Quick Reference: Additional Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

3. React: Fundamentals and Mental Model

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

3.1. The Mental Model: UI as a Function of State

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

3.1.1. React's Core Idea

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

3.1.2. Why This Matters for Backend Developers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

3.1.3. What React Is, and What It Is Not

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

3.2. Setting Up a React Playground

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

3.2.1. Creating a Vite Project

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

3.2.2. Cleaning Up the Template

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

3.3. JSX: Writing UI in TypeScript

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

3.3.1. JSX Rules

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

3.3.2. Expressions in JSX

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

3.3.3. Conditional Rendering

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

3.3.4. TypeScript and JSX

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

3.3.5. Hands-On: A First Component

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

3.4. Components, Props, and Composition

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

3.4.1. Functional Components

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

3.4.2. Typing Props

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

3.4.3. The `children` Prop

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

3.4.4. Component Composition

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

3.4.5. Thinking in Components

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

3.4.6. Composition vs. Inheritance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

3.4.7. When Composition Breaks Down

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

3.5. State with `useState`: Making Components Interactive

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

3.5.1. `useState`: The Basics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

3.5.2. How Re-rendering Works

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

3.5.3. State Updates Are Asynchronous

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

3.5.4. Multiple State Variables

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

3.5.5. Hands-On: Building a Toggle Panel

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

3.6. Side Effects with `useEffect`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

3.6.1. `useEffect`: The Basics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

3.6.2. The Dependency Array

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

3.6.3. When NOT to Use `useEffect`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

3.7. Event Handling and Controlled Forms

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

3.7.1. Event Handling in React

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

3.7.2. TypeScript Event Types

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

3.7.3. Controlled Components

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

3.7.4. A First Look at `useRef`: Focus Management

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

3.7.5. Hands-On: Building the Ticket Form

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

3.8. Lists and Keys

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

3.8.1. Rendering Lists with `map`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

3.8.2. Why Keys Matter

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

3.8.3. Filtering and Conditional Rendering in Lists

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

3.8.4. Hands-On: Building the Ticket List

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

3.9. Putting It All Together: A Mini Ticket Board

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

3.9.1. Shared Types and Sample Data

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

3.9.2. The Status Filter

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

3.9.3. Updating Components for Shared Types

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

3.9.4. Wiring It All Together

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

3.9.5. Adding a Search Input

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

3.9.6. Verification

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

3.9.7. What's Missing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

4. Advanced React

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

4.1. Custom Hooks: Extracting Reusable Logic

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

4.1.1. The Problem

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

4.1.2. What a Custom Hook Is

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

4.1.3. Building Custom Hooks Step by Step

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

4.1.3.1. `useDebounce`: Delaying a Rapidly Changing Value

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

4.1.3.2. `useLocalStorage`: Persisting State Across Refreshes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

4.1.3.3. `useTicketFilters`: A Domain-Specific Hook

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

4.1.4. Rules for Custom Hooks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

4.1.5. usehooks-ts: Don't Reinvent the Wheel

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

4.1.6. Testing Custom Hooks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

4.2. Context API: Sharing State Across the Tree

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

4.2.1. The Problem: Prop Drilling

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

4.2.2. Context: A Teleporter for Data

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

4.2.3. Building a Theme Context

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

4.2.4. When to Use Context

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

4.2.5. When NOT to Use Context

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

4.3. `useReducer`: Complex State Logic

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

4.3.1. The Problem

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

4.3.2. `useReducer`: The Pattern

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

4.3.3. `useReducer` + Context

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

4.3.4. `useState` vs. `useReducer`: Decision Guide

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

4.4. Refs, Memoization, and Performance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

4.4.1. `useRef`: Escaping the Render Cycle

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

4.4.2. `useMemo`: Caching Expensive Computations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

4.4.3. `useCallback`: Stabilizing Function References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

4.4.4. `React.memo`: Skipping Re-renders

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

4.4.5. The Performance Optimization Mindset

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

4.4.6. React Compiler: Automatic Memoization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

4.5. Error Boundaries and Suspense

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

4.5.1. Error Boundaries

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

4.5.2. Suspense

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

4.5.3. Error Boundaries + Suspense Together

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

4.6. Actions, Optimistic Updates, and the `use` Hook

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

4.6.1. Actions and `useActionState`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

4.6.2. `useFormStatus`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

4.6.3. `useOptimistic`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

4.6.4. The `use` Hook

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

4.6.5. `ref` as a Prop

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

Part II: Next.js Core Concepts

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

5. Next.js: Getting Started and Project Structure

We've spent four chapters building our TypeScript and React foundation. We can write type-safe code, compose components, manage state with hooks, and handle side effects. But try to build a real application with React alone and we'll hit walls fast: there's no built-in router, no server-side rendering, no way to create API endpoints, and no build pipeline that optimizes our code for production. We're left choosing, configuring, and maintaining a dozen separate tools, and that stack becomes its own project to manage.

That's the gap Next.js fills. It wraps React with a file-based router, multiple rendering strategies, API routes, image and font optimization, and a production-grade build system. Instead of assembling infrastructure, we write application code.

In this chapter, we'll scaffold the HelpDesk AI project with `create-next-app`, set up VS Code for an optimal development experience, walk through the generated directory structure, configure `next.config.ts`, set up a professional code quality pipeline with ESLint, Prettier, `knip`, and Husky, and run both the development and production builds. By the end, we'll have a running project with automated quality checks on every commit: the foundation everything else builds on.

5.1. What Is Next.js, and Why Do We Need It?

React is a UI library. It handles one thing well: rendering components based on state. But a production application needs more than rendering. It needs routing so users can navigate between pages. It needs server-side rendering so search engines can index content and users see a fast initial load. It needs API endpoints for backend logic.

It needs image optimization so a 4 MB hero image doesn't ruin our Core Web Vitals score. It needs a build pipeline that bundles, minifies, and code-splits our JavaScript.

Without a framework, developers assemble these pieces by hand. Choose a router (React Router, TanStack Router). Configure a bundler (Vite, webpack). Set up SSR with a custom Express server. Wire up image optimization with a CDN. That glue code becomes its own maintenance burden, and every project reinvents the same wheel.

5.1.1. What Next.js Solves

Next.js is a React framework. Not a replacement for React, but a layer built on top of it that provides the infrastructure every web application needs:

- **File-based routing**: Drop a `page.tsx` into a folder and it becomes a URL. No router configuration, no route tables.
- **Multiple rendering strategies**: Server-side rendering, static generation, incremental regeneration, and streaming, all available per route, chosen by how we write the component.
- **API routes**: Backend endpoints live alongside our pages. A `route.ts` file in the `app/` directory handles HTTP requests.
- **Automatic optimization**: Images are resized and served in modern formats. Fonts are self-hosted without layout shift. JavaScript is code-split per route.

- **Zero-config TypeScript**: TypeScript works out of the box. No `tsconfig.json` tweaking, no separate build step.
- **Built-in build pipeline**: Turbopack (a Rust-based bundler) handles development and production builds with fast cold starts and incremental compilation.

The relationship is straightforward: React is the view layer. Next.js is the full-stack framework built on top of it. We write React components; Next.js decides where and when to render them, how to route to them, and how to ship them to the browser.

5.1.2. Next.js and Vercel

Next.js is an open-source project maintained by Vercel, a cloud platform for frontend deployment. The relationship matters because Vercel both develops the framework and offers the hosting platform most optimized for it. Features like Edge Functions, preview deployments, and the AI Gateway are Vercel-specific.

That said, Next.js runs anywhere Node.js runs. We'll deploy on Vercel in Chapter 24 because the integration needs almost no configuration, but Chapter 25 covers self-hosting with Docker and AWS. The framework doesn't lock us in.

5.1.3. Where This Chapter Fits

How much of this chapter feels new depends on the background we bring to it. Backend developers who know Spring Boot, Express, or Django will recognize the pattern of a framework wrapping a library: Next.js is to React what Spring Boot is to Java's Servlet API, providing structure, opinions, and batteries-included infrastructure.

Developers arriving from React SPAs built with Vite face the key shift: Next.js renders on the server by default. Components run on the server, not in the browser, unless we explicitly opt them into client-side behavior. That's a paradigm shift, and Chapter 7 is dedicated to understanding it.

And for anyone who knows Next.js through the Pages Router, this chapter focuses on Next.js 16.3 with the App Router: a different rendering model, different file conventions, and different data fetching patterns.

Now that we know what Next.js provides, let's look at its two architectural approaches, and why we're choosing one over the other.

5.2. App Router vs. Pages Router

Next.js offers two routing architectures: the **Pages Router** and the **App Router**. They coexist in the same framework, but they represent different eras of thinking about React applications.

5.2.1. Two Architectures, One Framework

The Pages Router came first. It introduced file-based routing to the React ecosystem and popularized `getServerSideProps` and `getStaticProps` as patterns for server-side data fetching. Every component in the Pages Router is a Client Component: it ships JavaScript to the browser and hydrates on the client.

The App Router arrived with Next.js 13 and became the recommended approach from Next.js 14 onward. It's built on React Server Components, a fundamentally different rendering model. Components are Server Components by default: they run on the server, send only their rendered HTML to the browser, and include zero client-side JavaScript. Layouts are nested and persistent. Data fetching happens inside the component itself, without separate functions.

5.2.2. Key Differences at a Glance

Aspect	Pages Router	App Router
File convention	`pages/about.tsx`	`app/about/page.tsx`
Data fetching	`getServerSideProps`, `getStaticProps`	Server Components, `fetch` with caching
Layouts	`_app.tsx` + `_document.tsx` (global only)	Nested `layout.tsx` per route segment
Default component type	Client Components	Server Components
Streaming	Not supported	Built-in with `loading.tsx` and Suspense
Metadata	`next/head`	`metadata` export or `generateMetadata`

5.2.3. Why the App Router

The App Router reduces client-side JavaScript by default. Server Components don't ship code to the browser; they send rendered HTML. For HelpDesk AI, that means the ticket list, the knowledge base, and the dashboard render on the server without adding to the browser's JavaScript payload.

Nested layouts prevent full-page re-renders. When a user navigates from one ticket to another, the sidebar and header stay mounted. Only the ticket detail area re-renders. With the Pages Router, every navigation remounts the entire page.

Streaming improves perceived performance. Instead of waiting for all data to load before showing anything, the App Router can stream parts of the page as they become ready. A slow database query doesn't block the entire page from rendering.

This book uses the App Router exclusively. Every code example, every hands-on section, and every deployment targets the App Router.



Background: The Pages Router

All new features and improvements go into the App Router; for new projects, there's no reason to choose the Pages Router. Tutorials and documentation that reference `getServerSideProps`, `_app.tsx`, or the `pages/` directory are describing the Pages Router.

Now that we've made our architectural choice, let's create the project.

5.3. Creating a Project with `create-next-app`

Next.js provides a scaffolding tool called `create-next-app` that generates a complete project with all configuration in place. Instead of answering interactive prompts, we'll use explicit CLI flags: reproducible, copy-pasteable, and identical for everyone following along.

5.3.1. Scaffolding the HelpDesk AI Project

Open a terminal and run:

```
$ pnpm create next-app@latest helpdesk-ai \
    --ts --tailwind --eslint --app --src-dir \
    --react-compiler --empty --no-agents-md
```

Let's break down the flags:

- `--ts`: TypeScript. The entire book and project use TypeScript.
- `--tailwind`: Installs Tailwind CSS. We'll configure it in Chapter 9, but having it from day one means generated styles work immediately.
- `--eslint`: Installs ESLint with the Next.js rule sets (`eslint-config-next`). We'll extend the configuration in section 5.7.
- `--app`: Uses the App Router (the default, but explicit is better than implicit).
- `--src-dir`: Places application code inside a `src/` directory. This separates source code from root-level configuration files.
- `--react-compiler`: Enables the React Compiler for automatic memoization (Chapter 4). We'll configure this in `next.config.ts` in section 5.6.
- `--empty`: Skips the default template content. We don't need Next.js's starter page.
- `--no-agents-md`: Skips the `AGENTS.md` and `CLAUDE.md` files that guide AI coding agents, so Chapter 29 can set them up with AI-assisted development in view. Skipping them here is only half the job, because `next dev` writes both files as soon as it detects a coding agent in the environment. We finish the job with one line of configuration in section 5.6.

With every choice passed as a flag, `create-next-app` runs without prompts. It fills the one option we left out, the **import alias**, with its default and prints it. That default, `@/*` mapped to `src/*`, is exactly what we want: it lets us write `import { db } from '@/shared/lib/db'` instead of relative paths like `../../../shared/lib/db`.

One more thing happens automatically: **Turbopack** is the default bundler. No flag needed: both `next dev` and `next build` use it.

The install runs straight through, and one generated file explains why. Back in Chapter 1, pnpm held up the toolchain until we decided whether `tsx`'s esbuild dependency was allowed to run an install script. That guardrail is still active: a new dependency does not get to execute code the moment it lands in `node_modules`. One package in this tree carries such a script, `unrs-resolver` (a native module resolver), and `create-next-app` answers for it before it calls pnpm, by writing a `pnpm-workspace.yaml` at the project root:

Figure 8. pnpm-workspace.yaml

```yaml
allowBuilds:
  sharp: false
  unrs-resolver: false
```

`false` means the package is installed but its script stays unrun. `unrs-resolver` does not need it: the package ships prebuilt binaries for the common platforms, and the script is the fallback that compiles from source when none of them fits. The generator lists `sharp`, Next.js's image optimizer, as a standing answer beside it. The version installed here has no install script at all, so pnpm never asks about it, and the entry costs nothing while covering the release that adds one back.

That leaves pnpm with nothing to decide, which is why the message from Chapter 1 stays away. It returns the moment a dependency arrives carrying a script nobody has ruled on: Chapter 10 hits it with Prisma, Chapter 20 hits it with esbuild, and that one we answer with `true`, because Vitest cannot transform our TypeScript until esbuild has fetched its binary. pnpm writes the undecided package into `allowBuilds` with a placeholder value and waits for us to replace it.

This is the same file Chapter 1 left us with a single `esbuild: true` in it. There we answered the question ourselves; here the generator answers it before pnpm gets to ask.

5.3.2. What Gets Generated

After the command finishes, `cd helpdesk-ai` and look at what was created:

```
helpdesk-ai/
├── .next/               # Generated types and build output
├── node_modules         # npm packages
├── src/
│   └── app/
│       ├── globals.css  # Global styles (Tailwind)
│       ├── layout.tsx   # Root layout
│       └── page.tsx     # Home page
├── .gitignore           # Git ignore file
├── eslint.config.mjs    # ESLint flat config
├── next.config.ts       # Framework configuration
├── next-env.d.ts        # Auto-generated type declarations
├── package.json         # Dependencies and scripts
├── pnpm-lock.yaml       # Dependency lockfile
├── pnpm-workspace.yaml  # pnpm settings (build-script approvals)
├── postcss.config.mjs   # PostCSS for Tailwind
├── README.md            # Project readme
└── tsconfig.json        # TypeScript configuration
```

Let's walk through the important files:

`src/app/layout.tsx`: The root layout. Every page in the application is wrapped by this component. It defines the `<html>` and `<body>` tags, imports global CSS, and sets metadata. Both the `lang` attribute and the metadata API return in Chapter 26, where we cover accessibility and SEO in depth:

Figure 9. src/app/layout.tsx (generated)

```tsx
import type { Metadata } from "next";
import "./globals.css";

export const metadata: Metadata = {
  title: "Create Next App",
  description: "Generated by create next app",
};

export default function RootLayout({ children }: LayoutProps<"/">) {
  return (
    <html lang="en">
      <body>{children}</body>
    </html>
  );
}
```

`LayoutProps` appears without an import, and that's deliberate: Next.js generates it as a global type on every `next dev`, `next build`, and `next typegen`, keyed by the route path. `LayoutProps<"/">` is therefore the props of the layout at `/`, and it knows that this layout receives `children`. Chapter 6 puts the helper to work on routes that carry parameters and slots, where deriving the type beats writing it by hand.

`src/app/page.tsx`: The home page. With the `--empty` flag, this is a minimal component. This file makes the `/` route accessible: remove it and `http://localhost:3000` returns a 404.

`next.config.ts`: The framework configuration file. We'll customize this extensively in section 5.6.

`tsconfig.json`: TypeScript configuration with Next.js defaults. The `paths` section maps `@/*` to `./src/*`, so we can write `@/shared/lib/db` instead of `../../../shared/lib/db`. We'll add one option that the generator doesn't include: `noUncheckedIndexedAccess`. With it enabled, array access and object indexing return `T | undefined` instead of `T`; the compiler forces us to handle the case where an element doesn't exist. Open `tsconfig.json` and add it to `compilerOptions`:

Figure 10. tsconfig.json (relevant section)

```json
{
  "compilerOptions": {
    "noUncheckedIndexedAccess": true,
    "paths": {
      "@/*": ["./src/*"]
    }
  }
}
```

We discussed this option in Chapter 1. From here on, every array or record lookup in our HelpDesk AI project goes through a proper `undefined` check, one less category of runtime bugs to worry about.

`next-env.d.ts`: Auto-generated type declarations for Next.js. Do not edit this file. It's regenerated on every build. It pulls in Next.js's TypeScript types so that special files like `page.tsx` and `layout.tsx` are recognized by the type system.

`.next/`: Generated output. `create-next-app` runs `next typegen` once the install finishes, which is why the route types are in place before we have started anything ourselves: that is what makes `LayoutProps<"/">` above resolve. Every `next dev` and `next build` writes more into this directory, and section 5.8.6 covers what lands there. We never edit it, and the generated `.gitignore` excludes it.

`eslint.config.mjs`: ESLint configuration using the flat config format (ESLint 9). We'll extend it in section 5.7.

`package.json`: Dependencies plus four scripts, `dev`, `build`, `start`, and `lint`. One pinned version is worth adjusting right away. The generator asks for `@types/node@^20`, and this project runs on Node.js 24, so the type definitions describe an older standard library than the one we're using. Bring them in line:

```
$ pnpm add -D @types/node@^24
```

`postcss.config.mjs`: Configures PostCSS with the Tailwind CSS plugin. This is what connects Tailwind to the build pipeline.

Before we start the dev server, let's set up our editor; it'll make every step from here on more productive.

5.4. VS Code Setup

A good editor setup eliminates entire categories of bugs before we run the code. Typos in variable names, missing imports, incorrect types, formatting inconsistencies: all caught inline, in real time, as we type. The few minutes we invest here save hours of debugging later.

VS Code is free, open source, and the most widely used editor in the web development community. Cursor and Windsurf are forks of it, so extensions and settings work the same way in both. Claude Code runs in the terminal and connects to VS Code through an extension rather than replacing it, so the settings below apply there too.



Warning: Working in a Different Editor

This section uses VS Code terminology. With a different editor (WebStorm, Neovim, Zed), the terminal-based tools (pnpm, ESLint, Prettier, Husky) work identically; the editor's TypeScript, ESLint, and formatting integration needs equivalent settings.

5.4.1. Essential Extensions

Install these six extensions. Each one addresses a specific gap:

Extension	ID	Purpose
ESLint	`dbaeumer.vscode-eslint`	Inline linting: red and yellow underlines for errors and warnings as we type
Prettier	`esbenp.prettier-vscode`	Format on save: consistent formatting without thinking about it
Tailwind CSS IntelliSense	`bradlc.vscode-tailwindcss`	Class name autocompletion, CSS preview on hover, lint for invalid classes
Prisma	`Prisma.prisma`	Syntax highlighting and autocompletion for `.prisma` files (needed from Chapter 11)
Error Lens	`usernamehw.errorlens`	Shows errors and warnings inline, directly on the line (no hovering needed)
Pretty TypeScript Errors	`yoavbls.pretty-ts-errors`	Reformats long TypeScript errors into readable, indented blocks instead of one dense line

Install them from the command palette (`Cmd+Shift+P` on macOS, `Ctrl+Shift+P` on Windows/Linux) → "Extensions: Install Extension".

Alternatively, install them from the terminal. If the `code` command is not on the PATH, open the command palette and run "Shell Command: Install 'code' command in PATH". Then:

```
code --install-extension dbaeumer.vscode-eslint
code --install-extension esbenp.prettier-vscode
code --install-extension bradlc.vscode-tailwindcss
code --install-extension Prisma.prisma
code --install-extension usernamehw.errorlens
code --install-extension yoavbls.pretty-ts-errors
```

5.4.2. Project Settings

Create a `.vscode/` directory at the project root with a `settings.json` file. These settings apply only to this project; they won't affect the global VS Code configuration:

Figure 11. .vscode/settings.json

```
{
  "editor.formatOnSave": true,
  "editor.defaultFormatter": "esbenp.prettier-vscode",
  "editor.codeActionsOnSave": {
    "source.fixAll.eslint": "explicit",
    "source.organizeImports": "never"
  },
  "js/ts.tsdk.path": "node_modules/typescript/lib"
}
```

Five settings, each with a specific reason:

- **`editor.formatOnSave`**: Every time we save a file, Prettier formats it. We never think about indentation, semicolons, or trailing commas again.
- **`editor.defaultFormatter`**: Uses Prettier (not VS Code's built-in formatter) for all file types.
- **`source.fixAll.eslint`**: Auto-fixes ESLint errors on save. If ESLint can fix it (missing import type annotation, wrong quote style), it does so automatically.
- **`source.organizeImports: "never"`**: Disables VS Code's built-in import organizer. We'll let ESLint handle import ordering instead; the two conflict if both run.
- **`js/ts.tsdk.path`**: Uses the project's TypeScript version instead of VS Code's bundled one. This ensures type checking matches what the build sees.

5.4.3. Launch Configuration for Debugging

Create a `.vscode/launch.json` for debugging Next.js server-side code:

Figure 12. .vscode/launch.json

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "Next.js: debug server-side",
      "type": "node",
      "request": "launch",
      "runtimeExecutable": "pnpm",
      "runtimeArgs": ["dev"],
      "skipFiles": ["<node_internals>/**"],
      "env": {
        "NODE_OPTIONS": "--inspect"
      }
    }
  ]
}
```

With this configuration, we can set breakpoints in Server Components, Server Actions, and API routes, then press F5 to start the dev server with the debugger attached. When execution hits a breakpoint, VS Code pauses and lets us inspect variables, step through code, and evaluate expressions: the same workflow we'd use for any Node.js application.

5.4.4. First Start

Let's verify the project works. Press F5 in VS Code to start the dev server using the launch configuration we created. The terminal shows output like:

```
▲ Next.js 16.3.X (Turbopack)
- Local:         http://localhost:3000
- Network:       http://192.168.1.100:3000
- Debugger port: 9230
✓ Ready in XXXms
✓ Running next.config.ts took XXXms
```

Open `http://localhost:3000` in the browser. With the `--empty` flag, we'll see a minimal page showing "Hello world!", ready for us to replace with our own content.

5.4.5. Replacing the Default Content

Let's replace the generated `page.tsx` with a placeholder for HelpDesk AI:

Figure 13. src/app/page.tsx

```tsx
export default function Home() {
  return (
    <main>
      <h1>HelpDesk AI</h1>
      <p>Coming soon.</p>
    </main>
  )
}
```

And update the root layout metadata:

Figure 14. src/app/layout.tsx

```tsx
import './globals.css'
import type { Metadata } from 'next'

export const metadata: Metadata = {
  title: 'HelpDesk AI',
  description: 'AI-powered support portal for teams'
}

export default function RootLayout({ children }: LayoutProps<'/'>) {
```

```
  return (
    <html lang="en">
      <body>{children}</body>
    </html>
  )
}
```

Save both files. The browser updates automatically, no full page reload needed. We'll look at how this works (Hot Module Replacement and Fast Refresh) in section 5.8.

We have a running project. Let's understand what the generated project structure means.

5.5. Directory Structure: Where Everything Lives

Next.js has strong opinions about file placement. Not every file in every directory is equal; some files have special meaning, and putting code in the wrong place can create a route we didn't intend or hide a page we did.

5.5.1. The `src/` Directory

We used the `--src-dir` flag, which places application code inside `src/`. The result is a clean separation:

```
helpdesk-ai/
├── src/            # Application code
│   └── app/        # Routing and pages
├── next.config.ts  # Framework config
├── tsconfig.json   # TypeScript config
├── package.json    # Dependencies
└── ...             # Other config files
```

We already walked through the individual files in section 5.3.2. The key takeaway for the directory structure: root-level files are configuration, application code lives in `src/`. This separation becomes more valuable as the project grows: configuration files stay out of the way when navigating source code.

5.5.2. The `app/` Directory: Routing as File System

The `app/` directory is where Next.js's file-based routing lives. Every file in `app/` has a specific role; this isn't a free-form directory where we drop any file. Chapter 6 covers routing in depth, but understanding the file conventions now prevents confusion later.

Next.js recognizes these special files:

File	Purpose
`page.tsx`	Makes a route accessible; without it, the URL returns 404
`layout.tsx`	Shared UI that wraps child routes (persists across navigation)
`loading.tsx`	Loading UI shown while a route segment loads (Suspense boundary)
`error.tsx`	Error UI shown when a route segment throws (Error Boundary)
`global-error.tsx`	Error UI for the root layout, which `error.tsx` cannot reach
`not-found.tsx`	UI shown for 404 responses
`route.ts`	API endpoint (no UI); handles HTTP requests
`template.tsx`	Like layout, but re-renders on every navigation
`default.tsx`	Fallback UI for parallel routes

The key rule: **only `page.tsx` and `route.ts` create accessible routes**. We can place component files, utility functions, and type definitions alongside route files in `app/`; Next.js won't expose them as URLs. But keeping route files mixed with non-route files makes the directory harder to navigate. We'll establish a clean separation in Chapter 10 with feature slices.



Warning: One Route File per Directory

Placing `page.tsx` and `route.ts` in the same directory causes a conflict: they both try to handle the same URL. A directory should contain either a `page.tsx` (for UI routes) or a `route.ts` (for API endpoints), never both.

5.5.3. The `public/` Directory

Because we used the `--empty` flag, `create-next-app` didn't generate a `public/` directory. We'll create it when we need it. The purpose of `public/` is to serve static assets: a file at `public/logo.png` becomes accessible at `http://localhost:3000/logo.png`. No import needed, no build step. Use this for favicons, Open Graph images, robots.txt, and other files that need a stable URL.

5.5.4. Planning Ahead: The Full HelpDesk AI Structure

Over the coming chapters, the project grows into this structure:

```
src/
├── app/                    # Routing layer (thin)
│   ├── [locale]/           # Language segment (Ch. 27)
│   │   ├── (public)/       # Customer portal routes
│   │   ├── (auth)/         # Login/signup routes
│   │   └── (dashboard)/    # Agent dashboard routes
│   └── api/                # Route handlers, outside the segment
├── features/               # Feature slices (Ch. 10)
│   ├── tickets/
│   ├── knowledge/
```

```
│   ├── auth/
│   └── copilot/
├── shared/                 # Cross-feature code (Ch. 10)
│   ├── components/
│   ├── hooks/
│   ├── lib/
│   └── types/
├── i18n/                   # Locale routing and lookup (Ch. 27)
└── emails/                 # Email templates (Ch. 14)

prisma/                     # Database schema (Ch. 11), at project root
```

We don't create all of these directories now. Each chapter adds what it needs. This preview helps us see where we're heading: `app/` stays a thin routing layer that imports from feature-specific directories. The architecture chapter (Chapter 10) explains the reasoning behind this structure.

With the directory layout understood, let's configure the framework itself.

5.6. Configuration: `next.config.ts` in Detail

Every Next.js project has a `next.config.ts` file at its root. This is the control center: it tells Next.js how to behave during development, what to optimize during builds, and how to handle redirects, environment variables, and more.

5.6.1. The Configuration File

The generated `next.config.ts` contains a single option, `reactCompiler: true`, because we used the `--react-compiler` flag:

Figure 15. next.config.ts (generated)

```ts
import type { NextConfig } from "next";

const nextConfig: NextConfig = {
  /* config options here */
  reactCompiler: true,
};

export default nextConfig;
```

The `NextConfig` type provides autocomplete for every option. We'll point this out in a tip below. Let's add the options HelpDesk AI needs:

Figure 16. next.config.ts

```ts
import type { NextConfig } from 'next'

const nextConfig: NextConfig = {
  reactStrictMode: true,
  reactCompiler: true,
  typedRoutes: true,
  agentRules: false,
  logging: {
    browserToTerminal: 'warn'
  },
  experimental: {
    typedEnv: true
  }
}

export default nextConfig
```

Save the file. If the dev server is running, the terminal shows:

```
⚠ Found a change in next.config.ts. Restarting the server to apply the changes...
```

Next.js watches `next.config.ts` and restarts automatically whenever it changes, no manual restart needed.

5.6.2. What Each Option Does

`reactStrictMode: true`: React Strict Mode double-invokes effects and renders during development to help us find bugs. If a `useEffect` has a cleanup problem, Strict Mode exposes it immediately instead of letting it surface in production. There's no performance cost: the double invocations only happen in development.

`reactCompiler: true`: Enables the React Compiler (Chapter 4). The compiler inserts memoization automatically at build time, eliminating the need for manual `useMemo`, `useCallback`, or `React.memo`.

`typedRoutes: true`: Enables statically typed routes. When enabled, Next.js generates TypeScript types for all our routes during `next dev` and `next build`. The `<Link href="...">` component and `useRouter().push()` will type-check URLs at compile time: a typo in a route path becomes a TypeScript error, not a 404 in production. The `PageProps`, `LayoutProps`, and `RouteContext` helpers we met in the generated layout are a separate mechanism that needs no flag: Next.js writes those on every `next dev`, `next build`, and `next typegen` regardless. `typedRoutes` adds the link and navigation checking on top of them.

`agentRules: false`: Turns off one piece of automatic behavior. When `next dev` detects an AI coding agent in the environment, it writes an `AGENTS.md` and a `CLAUDE.md` at the project root, pointing the agent at the version-matched documentation in `node_modules/next/dist/docs/` rather than at whatever it learned during training. That default is a good one, and it is the reason those files keep reappearing in a diff after they are deleted. We switch it off here for a single reason:

those two files are the subject of Chapter 29, where we write them deliberately and decide what belongs in them. Until then, a file the project has never explained has no business appearing in it. Chapter 29 removes this line and lets the generation resume.

`logging.browserToTerminal: 'warn'`: Forwards browser console output to the terminal during development. We cover this in detail in section 5.8.2.

`experimental.typedEnv: true`: Generates TypeScript declarations for our environment variables. During development, Next.js reads our `.env` files and creates a `.d.ts` file in `.next/types/` with type information for each variable. This gives us autocompletion when typing `process.env.` in the editor.

5.6.3. Options We'll Add Later

This isn't the final configuration. Options arrive as features need them, and it is worth knowing the shape of what is coming so the file's growth reads as a plan rather than accretion. Chapter 7 adds `cacheComponents`, the switch that changes how every page in this book is rendered, along with a `cacheLife` profile and `partialPrefetching`; Chapter 9 adds `images.remotePatterns` for avatars we do not host; Chapter 12 turns on `authInterrupts` for the authorization boundaries. Part V brings the rest: a flag that opens the app to the end-to-end suite in Chapter 20, an image quality list in Chapter 21, security headers in Chapter 22, finer logging in Chapter 23, and the deployment settings for both hosting targets in Chapter 25.

The file also changes shape twice, and neither change comes from us. Chapter 23 wraps the export in `withSentryConfig` and Chapter 27 wraps it again in the next-intl plugin, so the closing `export default nextConfig` grows into a call around a call. One line goes the other way: the `agentRules: false` we set above comes back out in Chapter 29, once those files are ours to write.

Two of those are worth flagging now. `cacheComponents` is the one option here that changes a default rather than adding a capability, which is why it gets a chapter rather than a line. And `experimental` is not a warning label to avoid: `typedEnv` sits there today and does useful work, but options can move out of it or change shape between releases, so anything under that key is a thing to re-check when upgrading.



Tip: Let Autocomplete Do the Work

There's no need to memorize the configuration API. The `NextConfig` type provides autocomplete for every option: type the property name and let IntelliSense do the rest. The Next.js documentation bundled at `node_modules/next/dist/docs/` is the authoritative reference for the installed version.

5.6.4. Environment Variables

Next.js looks a variable up in several places and stops at the first one that has it. From highest priority to lowest:

1. `process.env`: whatever the shell or the hosting platform already set
2. `.env.development.local`, `.env.production.local`: per-environment local overrides
3. `.env.local`: local overrides that apply to every environment

4. `.env.development`, `.env.production`: per-environment defaults
5. `.env`: defaults for all environments

The `.local` files sit above the others, which makes `.env.local` the right place for secrets during development: it wins over anything a teammate has checked in. One exception is worth knowing now, because it decides how the test suite gets its configuration later: `.env.local` is skipped entirely when `NODE_ENV` is `test`, so the end-to-end run in section 20.6 reads `.env.test` instead of our development secrets. By default, none of these are committed to git: the generated `.gitignore` ignores every `.env*` file, so secrets never leave our machine.

Create two empty files at the project root: `.env.example` and `.env.local`. They start empty on purpose. We fill them in as we go, because each feature brings its own variables: the database adds `DATABASE_URL`, authentication adds its Google credentials, the AI Gateway adds its key, and so on. Rather than list variables we don't use yet, each chapter adds the ones its feature needs, so the files grow alongside the project.

The two files play different roles. `.env.example` is the shared template: it lists every variable the project needs, with empty or placeholder values and no secrets. Since `.gitignore` ignores every `.env*` file, we allow this one back in with a `!.env.example` line, so the variable list can be committed and shared with the team. The real values go in `.env.local`, which stays out of version control.



Aside: The Folder Our Editor Writes

One entry in `.gitignore` is the one a generator cannot guess, because it depends on the editor rather than the framework. JetBrains IDEs (WebStorm, IntelliJ) keep per-project state in `.idea/`, down to which panels are open and where the window sits. It earns a group of its own at the end of the file:

```
# editors
/.idea/
```

VS Code is the case that argues against turning that into a blanket rule. Its `.vscode/` folder holds the settings and the launch configuration from section 5.4, and both are worth sharing: whoever clones the repository gets the same format-on-save behavior and the same debugger. So `.vscode/` stays in version control while `.idea/` stays out. The line to draw is not editor folders in general, it is whatever only one machine cares about.

Our configuration is in place. The project type-checks routes, forwards browser logs, and loads environment variables. We'll add more options as features require them. Next, we'll set up the tools that keep our code clean.

5.7. Code Quality Tooling: ESLint, Prettier, knip, and Husky

Without automated checks, code style drifts, unused code accumulates, and bugs slip through review. In a book project, consistency matters even more: every code example must follow the same conventions. We'll set up four tools that work together: ESLint catches bugs, Prettier formats code, knip finds dead code, and Husky runs all of them before every commit.

5.7.1. ESLint: Catching Bugs and Enforcing Rules

ESLint analyzes our code for potential errors and style violations. It catches bugs that TypeScript misses: leaked renders in JSX, missing hook dependencies, inconsistent import styles, and more.

5.7.1.1. The Generated Configuration

`create-next-app` generates an `eslint.config.mjs` that pulls in two configurations, `eslint-config-next/core-web-vitals` and `eslint-config-next/typescript`. Together they cover performance rules and TypeScript-specific checks. This is a solid starting point, but not enough for a production project.

5.7.1.2. Extending the Configuration

Install the additional plugins we need:

```
$ pnpm add -D eslint-plugin-react \
    eslint-plugin-react-hooks \
    @stylistic/eslint-plugin \
    eslint-config-prettier \
    eslint-plugin-prettier
```



Background: We Stay on ESLint 9

`create-next-app` installs ESLint 9, and we keep it there. A newer major (ESLint 10) is out, but the plugins this config builds on (`eslint-plugin-react`, `eslint-plugin-import`, and through them `eslint-config-next`) still target ESLint 9's rule API. Until they ship support for the newer major, installing it breaks the lint run. Moving up is a one-line change the day the plugins catch up.

Now replace the generated `eslint.config.mjs` with our extended configuration:

Figure 17. eslint.config.mjs

```
import { defineConfig, globalIgnores } from 'eslint/config'
import nextVitals from 'eslint-config-next/core-web-vitals'
import nextTypescript from 'eslint-config-next/typescript'
import react from 'eslint-plugin-react'
import reactHooks from 'eslint-plugin-react-hooks'
import stylistic from '@stylistic/eslint-plugin'
import eslintConfigPrettier from 'eslint-config-prettier'
import eslintPluginPrettier from 'eslint-plugin-prettier'

const eslintConfig = defineConfig([
  ...nextVitals,
  ...nextTypescript,
  react.configs.flat.recommended,
```

```
  reactHooks.configs.flat.recommended,
  stylistic.configs.recommended,
  eslintConfigPrettier,
  {
    plugins: {
      prettier: eslintPluginPrettier
    },
    rules: {
      'prettier/prettier': 'error',
      'import/order': [
        'error',
        {
          groups: ['builtin', 'external', 'internal', 'type'],
          pathGroups: [
            { pattern: 'react', group: 'external', position: 'before' },
            { pattern: 'next', group: 'external', position: 'before' },
            { pattern: 'next/**', group: 'external', position: 'before' },
            { pattern: '@/shared/**', group: 'internal', position: 'before' },
            { pattern: '@/**', group: 'internal', position: 'after' }
          ],
          pathGroupsExcludedImportTypes: ['react', 'next', 'type'],
          'newlines-between': 'ignore'
        }
      ],
      'react/react-in-jsx-scope': 'off',
      '@typescript-eslint/no-unused-vars': [
        'error',
        {
          argsIgnorePattern: '^_',
          ignoreRestSiblings: true
        }
      ],
      '@typescript-eslint/consistent-type-imports': 'error',
      'no-restricted-imports': [
        'error',
        {
          patterns: [
            {
              regex: '^\\.\\.?/(?!.*\\.css$)',
              message: 'Use the @/ alias, not relative imports.'
            }
          ]
        }
      ],
      'no-console': ['warn', { allow: ['warn', 'error'] }]
    }
  },
  globalIgnores(['node_modules/**', '.next/**', 'out/**', 'build/**',
  ↪  'next-env.d.ts'])
])
```

```
63
64 export default eslintConfig
```

This configuration layers multiple rule sets:

- **`eslint-config-next/core-web-vitals`**: Next.js-specific rules for performance (image optimization, link usage)
- **`eslint-config-next/typescript`**: TypeScript rules tuned for Next.js
- **`eslint-plugin-react`**: React-specific rules (JSX structure, component patterns)
- **`eslint-plugin-react-hooks`**: Hook rules (dependency arrays, call order)
- **`@stylistic/eslint-plugin`**: Code style rules (spacing, quotes, semicolons)
- **`eslint-config-prettier`**: Disables all rules that conflict with Prettier (formatting is Prettier's job)
- **`eslint-plugin-prettier`**: Runs Prettier as an ESLint rule so formatting violations appear as ESLint errors

The `react/react-in-jsx-scope` rule is turned off because Next.js automatically imports React. We don't need `import React from 'react'` at the top of every file.

The `@typescript-eslint/no-unused-vars` rule is escalated from `"warn"` (the default in `eslint-config-next/typescript`) to `"error"` because warnings exit zero and block neither commits nor CI. The two options make the rule practical: `argsIgnorePattern: "^_"` allows prefixing intentionally unused parameters with an underscore (common in callbacks like `array.map((_item, index) => ...)`), and `ignoreRestSiblings` allows destructuring to omit properties (`const { unwanted, ...rest } = props`) without triggering an error on `unwanted`.

The `import/order` rule fixes the sequence of the import block, which otherwise drifts per file and per author. The `groups` list sets the coarse order: Node built-ins, then packages from `node_modules`, then our own code behind the `@/` alias, then type-only imports. The `pathGroups` entries refine it: `react` and `next` come before other packages because they are the framework we're reading against, and inside our own code `@/shared/**` precedes the rest. `pathGroupsExcludedImportTypes` keeps those three out of the exclusion the rule applies by default, so the positions we asked for actually take effect, and `newlines-between: 'ignore'` leaves blank lines inside the block alone. The rule is auto-fixable, so `pnpm lint --fix`, or a save with the settings from section 5.4.2, sorts the block for us rather than turning it into busywork.

The `consistent-type-imports` rule from `@typescript-eslint` enforces the `import type` convention we established in Chapter 1 (section 1.6.2): type-only imports are erased at compile time and never reach the bundle. With the rule set to `"error"`, ESLint auto-fixes the import style on save.

The `no-restricted-imports` rule bans relative imports between project files. Inside a feature it's tempting to write `import { getTickets } from './queries/getTickets'`, but as files move and features nest, `../../` chains turn brittle and hard to read. We require the `@/` alias instead: `@/features/tickets/queries/getTickets` points at the same file no matter where the importing file lives. The pattern carves out one exception: a component may import its own co-located `*.css` stylesheet relatively, because a stylesheet belongs next to the component it styles. The regex looks dense, but it says exactly that: block anything starting with `./` or `../`, unless it ends in `.css`.

The `no-console` rule warns on `console.log` and `console.info` calls while allowing `console.warn` and `console.error`. Forgotten debug logs have no place in production code; this rule catches them before they ship.

5.7.1.3. Running ESLint

We run ESLint directly, which gives us full control over flags and configuration:

```
$ pnpm lint
```

`create-next-app` already wrote a `lint` script, so this works right away: it runs `eslint`, which scans the project from the current directory and reports errors and warnings. To auto-fix what ESLint can:

```
$ pnpm lint --fix
```

With the VS Code ESLint extension, we see errors inline as we type: red underlines for errors, yellow for warnings. Combined with Error Lens, the error message appears right on the line.

5.7.2. Prettier: Consistent Formatting

Prettier handles formatting: indentation, line breaks, semicolons, quote style, and trailing commas. By running Prettier on every save, we eliminate all formatting discussions. The code looks the same regardless of who wrote it.

5.7.2.1. Configuration

Install Prettier and the Tailwind CSS plugin:

```
$ pnpm add -D prettier prettier-plugin-tailwindcss
```

Create a `.prettierrc` at the project root:

Figure 18. .prettierrc

```
{
  "semi": false,
  "singleQuote": true,
  "trailingComma": "none",
  "tabWidth": 2,
  "printWidth": 130,
  "plugins": ["prettier-plugin-tailwindcss"]
}
```

These choices are deliberate:

- **`semi: false`**: No semicolons. TypeScript doesn't need them, and they add visual noise.
- **`singleQuote: true`**: Single quotes for strings. Consistent with the Next.js ecosystem.
- **`trailingComma: "none"`**: No trailing commas. A matter of preference. We chose clarity over diff optimization.

- **`tabWidth: 2`**: Two spaces per indentation level. The standard in the JavaScript and TypeScript ecosystem.
- **`printWidth: 130`**: Line width for source files. Code blocks in the book are narrower (85 characters), but source code in the repository can be wider.
- **`plugins`**: The Tailwind CSS plugin sorts utility classes in a consistent, canonical order. `bg-blue-600 text-white p-4` becomes `p-4 bg-blue-600 text-white`: layout before colors.

We also need a `.prettierignore` to exclude files that Prettier shouldn't touch:

Figure 19. .prettierignore

```
pnpm-lock.yaml
pnpm-workspace.yaml
```

The lock file is machine-generated, and Prettier would break its structure. `pnpm-workspace.yaml` holds pnpm's project settings, including the `allowBuilds` approvals, so we keep Prettier out of it and let pnpm own the format.



Tip: Plugin Order Matters

The `prettier-plugin-tailwindcss` must be the last plugin in the `plugins` array. Other Prettier plugins may conflict with it if they run after it.

5.7.2.2. Running Prettier

Two scripts in `package.json` drive it: `format` runs `prettier --write .` and `format:check` runs `prettier --check .`. They appear in the complete script list at the end of this section, alongside the others we collect along the way.

`pnpm format` reformats all files. `pnpm format:check` verifies formatting without changing files (useful in CI pipelines where we want to catch violations without modifying code).

5.7.3. knip: Detecting Dead Code and Unused Dependencies

As a project grows, code gets deleted, refactored, and moved. What often stays behind are unused imports, exported functions that nothing calls, and dependencies in `package.json` that no file references. knip finds all of them.

5.7.3.1. Setup

Install knip:

```
$ pnpm add -D knip
```

Next.js's file conventions are the part a dead-code tool has to be told about, because route files like `page.tsx` and `layout.tsx` are entry points that nothing imports. A checker that traced imports alone would report every route in the project as dead. knip has a Next.js plugin that recognizes those conventions on its own, and it activates as soon as it sees `next` in `package.json`, so there is nothing to configure here. The first `knip.config.ts` arrives in section 7.9.7, where one file needs an exception the conventions cannot cover.

Run it:

```
$ pnpm knip
```

On a fresh project, knip should report no issues. As the project grows, run it periodically, especially after refactoring, to catch stale imports and orphaned exports.

5.7.4. Husky: Pre-Commit Hooks

Code quality checks work only if they run consistently. Relying on developers to remember `pnpm validate` before every commit doesn't scale. Pre-commit hooks solve this: they run checks automatically before every `git commit`.

5.7.4.1. Setup

Install Husky:

```
$ pnpm add -D husky
```

Initialize Husky:

```
$ pnpm husky init
```

This creates a `.husky/` directory with a `pre-commit` hook. Replace its contents:

Figure 20. .husky/pre-commit

```
pnpm validate
```

That's it: no additional configuration. The pre-commit hook runs the same `validate` script we'll define in the next section.

5.7.4.2. How It Works

The flow is:

1. We stage files with `git add`
2. We run `git commit`
3. Husky intercepts and triggers the pre-commit hook
4. `pnpm validate` runs type checking, ESLint, Prettier check, and knip on the entire project

5. If any check fails, the commit is blocked
6. Fix the issues, stage again, commit again

Running on the full project (not only staged files) prevents a subtle problem: our staged changes might be clean, but they could break something in a file we didn't touch. A full-project check catches this.



Warning: Bypassing the Pre-Commit Hook

Pre-commit hooks can be bypassed with `git commit --no-verify`. Use this only in genuine emergencies. If a hook consistently blocks commits, fix the underlying issue rather than skipping the check.

5.7.5. Bringing It All Together

Running `pnpm lint`, `pnpm format:check`, and `pnpm knip` separately is fine for debugging, but for everyday use we want a single command that runs all checks in sequence and stops on the first failure. That's what the `validate` script does.

Here's the complete `scripts` section for `package.json`:

Figure 21. package.json (complete scripts)

```
{
  "scripts": {
    "dev": "next dev",
    "build": "next build",
    "start": "next start",
    "lint": "eslint --max-warnings 0",
    "lint:fix": "eslint --fix .",
    "format": "prettier --write .",
    "format:check": "prettier --check .",
    "prepare": "husky",
    "validate": "tsc --noEmit && pnpm lint && pnpm format:check && pnpm knip",
    "verify": "pnpm validate && pnpm build",
    "validate:fix": "pnpm lint:fix && pnpm format"
  }
}
```

The `validate` script chains four checks with &&: if any check fails, the rest don't run. This gives fast feedback: fix the first problem, run `pnpm validate` again.

1. **`tsc --noEmit`** runs the TypeScript compiler without producing output files: its sole purpose is to verify that every file in the project type-checks. A missing prop, a wrong return type, or an incompatible argument gets caught here before ESLint even starts. The `--noEmit` flag is important: Next.js handles compilation itself, so we don't need `tsc` to produce JavaScript.

We only want the type errors. Type checking comes first because a type error often causes cascading lint failures; fixing the type error first avoids chasing false positives.

`next build` type-checks too, and it does so by running the project's own `tsc` binary rather than loading TypeScript's JavaScript API. That detail is what makes **TypeScript 7**, a native port that type-checks several times faster, usable at all: its JavaScript API is not available, so a build that needed it would have nothing to call. Installing `typescript@^7` is the whole adoption, and `experimental.useTypeScriptCli: false` is the way back to the API-based checker if a project needs it.

HelpDesk AI stays on TypeScript 5 anyway, and the reason is worth knowing before spending an afternoon on it: `typescript-eslint` does not support TypeScript 7, and refuses to load rather than degrading. The compiler gets faster and the linter stops running, which is a poor trade for a `validate` script built on both. This is a question of when, not whether, so it is worth retrying once the linting stack catches up.

2. **`pnpm lint`** runs ESLint across the project, and `--max-warnings 0` is what makes it mean something. ESLint exits `0` when only warnings remain, so without the flag `validate` reports success with warnings sitting in its output, and a warning that fails nothing is a warning nobody reads. Plenty of rules arrive at warning level rather than error: the accessibility rules bundled with `core-web-vitals`, and the React Compiler, which warns rather than errors when a component uses an API it cannot memoize safely. The cost of ignoring that one is not a broken build but a component that quietly stops being optimized. The flag turns every one of them into something that has to be dealt with on the day it appears, which is far cheaper than inheriting a backlog of them later.
3. **`pnpm format:check`** verifies that all files match the Prettier configuration.
4. **`pnpm knip`** detects unused exports, dependencies, and files.

The `validate:fix` script is the auto-repair counterpart: it runs `eslint --fix` to fix linting issues and `prettier --write` to reformat all files. Type errors can't be auto-fixed; they require manual correction. Use `validate` to check, `validate:fix` to fix.

`verify` adds the production build on top, and it exists because a green `validate` does not mean the application builds. Every check in `validate` reads one file at a time. The build is the only step that resolves the whole import graph, and some of the most expensive mistakes in an App Router project live in that graph rather than in any single file: a Client Component that reaches a server-only module is perfectly typed, perfectly linted, and fails the build. We meet that exact failure later, and it is worth knowing now that `validate` cannot see it. Run `validate` while working, `verify` before considering something finished.

The `prepare` script runs automatically after `pnpm install`, ensuring Husky's git hooks are set up for every developer who clones the project. No manual initialization needed.

The Husky pre-commit hook we configured in section 5.7.4 runs `pnpm validate`, so every commit is checked automatically. Let's verify. Stage a file with a deliberate error, an unused variable, for example:

```
$ echo 'const unused = 42' >> src/app/page.tsx
$ git add src/app/page.tsx
$ git commit -m "test: verify pre-commit hook"
```

The commit should fail with an ESLint error about the unused variable. Remove the line, stage again, and the commit succeeds. The safety net is in place.

Our code quality pipeline is in place. Let's see how the development and production builds work.

5.8. Dev Server, Build Process, and Turbopack

In section 5.4.4, we started the dev server by pressing `F5` in VS Code. Under the hood, the launch configuration runs `pnpm dev`, the command that powers the Next.js development server. Let's look at what that command does, how the production build works, and the bundler behind both.

5.8.1. The Dev Server

```
$ pnpm dev
```

This is the command our launch configuration executes. It starts the Next.js development server on `http://localhost:3000`. We can also run it directly from the terminal, useful when we don't need the debugger. Behind the scenes:

- **Turbopack** compiles our TypeScript and JSX into JavaScript that the browser and server can execute. When we change a single file, only that file and its dependents are recompiled.
- **Hot Module Replacement (HMR)** pushes the updated code to the browser without a full page reload, the same mechanism we saw with Vite in section 3.2. Change a CSS class and the style updates. Change a component's markup and the UI re-renders.
- **Fast Refresh** preserves React component state during HMR. If a counter component shows `5` and we change the button label, the counter stays at `5` after the update. State is preserved unless we change the component's structure in a way that forces a reset.

5.8.2. Browser Log Forwarding

When debugging client-side code, we normally switch to the browser's developer tools to read `console.log` output. Next.js can forward that output to the terminal instead.

We enabled this in `next.config.ts` with `logging.browserToTerminal: 'warn'`. With this setting, whenever a Client Component calls `console.warn()` or `console.error()` in the browser, the message appears in the terminal:

```
[browser] Something went wrong (src/app/page.tsx:8:17)
```

The output includes the file path, line number, and column, so we can jump directly to the source. Set the value to `true` to include `console.log` and `console.info` as well, or `'error'` to limit it to errors only.

This feature is especially useful when running AI coding agents like Claude Code. These agents operate from the terminal and can see client-side errors without needing access to a browser.

5.8.3. Turbopack Filesystem Caching

Turbopack caches compiled artifacts on disk between dev server restarts. The first `pnpm dev` takes a moment to compile everything. Subsequent starts reuse the cache and only recompile files that changed, often finishing in under a second.

The same applies to `pnpm build`. Both caches are on by default, as `turbopackFileSystemCacheForDev` and `turbopackFileSystemCacheForBuild` under `experimental`, and either can be set to `false` to opt out. The dev cache lives in `.next/dev/cache/turbopack`, the build cache in `.next/cache/turbopack`.

That directory split matters for CI. A build only starts warm if `.next/cache` survives from one run to the next, which a containerized build starting from a clean layer does not do unless the directory is explicitly cached or mounted. On a build machine that never preserves it, turning the build cache off avoids writing a cache nothing will read.

Writing compiled artifacts to disk also keeps them out of memory, which is the other half of why a long dev session stays manageable. Turbopack evicts what it is not actively using rather than holding every compiled route for the life of the process, so the dev server's footprint after an afternoon of work resembles its footprint after ten minutes. Both behaviors are on by default and neither needs configuration; they are worth knowing about mainly because a dev server whose memory climbs through the day is a problem the framework now absorbs rather than one we work around.



Background: From webpack to Turbopack

Turbopack was created by the team behind webpack: Tobias Koppers joined Vercel to build the successor in Rust. The performance gains come from incremental computation: Turbopack tracks dependencies between modules at a granular level, so a change to one file recompiles only what's affected.

5.8.4. The Production Build

Development mode optimizes for speed: fast compilation, HMR, helpful error messages. Production mode optimizes for the user: small bundles, fast load times, no development overhead.

Run a production build:

```
1 $ pnpm build
```

The build goes through several steps:

1. **Type checking**: TypeScript verifies all types. Errors here block the build.
2. **Route analysis**: Next.js determines which routes are static and which are dynamic.
3. **Pre-rendering**: Static pages are rendered at build time and saved as HTML files.
4. **Bundling**: Client-side JavaScript is bundled, minified, and code-split per route.
5. **Output**: Everything goes into the `.next/` directory.

Note that `next build` does not run ESLint. Linting happens through `pnpm validate` and the pre-commit hook, not during the build. This keeps builds fast and concerns separated: linting is a development-time check, not a build-time gate.

The separation cuts both ways, which is why `pnpm verify` from section 5.7.5 exists. The pre-commit hook deliberately runs the fast checks only, so committing stays quick; `verify` is the one to run before pushing, and at the end of any session that moved code across the server and client boundary.

The terminal output shows each route and its rendering strategy:

```
Route (app)
┌ ○ /
└ ○ /_not-found


○  (Static)  prerendered as static content
```

The ○ symbol means the route was pre-rendered as static HTML. It's served directly from a CDN with no server computation. For HelpDesk AI, the placeholder page is static. As we add data fetching in Chapter 8, routes will become dynamic. Chapter 7 covers the different rendering strategies and their symbols in detail.

5.8.5. Running the Production Build

To verify the production build locally:

```
$ pnpm start
```

This starts a production server on `http://localhost:3000`. The behavior matches what our users will see: optimized bundles, no development warnings, no HMR. Use this to verify that nothing breaks between development and production: some issues (strict mode violations, missing environment variables) only surface in production builds.

5.8.6. Understanding the Build Output

The `.next/` directory contains the build output:

- **`.next/static/`**: Immutable assets such as CSS bundles, JavaScript chunks, and media files. These are served with long cache headers; browsers cache them indefinitely.
- **`.next/server/`**: Server-rendered pages and API routes. These run on the server when a request comes in.
- **`.next/types/`**: Generated TypeScript types for routes and environment variables.

We never edit files in `.next/`; the directory is regenerated on every build. It's already in `.gitignore`.

We now have a complete development environment: a running project, an editor that catches errors, a code quality pipeline that enforces standards, and build tooling that produces optimized output. Let's verify everything works end-to-end.

5.8.7. Final Verification

Run through this checklist to confirm the project is fully set up:

```
# Dev server starts without errors
$ pnpm dev
# → http://localhost:3000 shows "HelpDesk AI"

# All quality checks pass in one command
$ pnpm validate
# → ESLint, Prettier, and knip all pass

# Production build succeeds
$ pnpm build
# → Build completes without errors

# Production server works
$ pnpm start
# → http://localhost:3000 serves the production build
```

If all checks pass, the foundation is solid. One thing has been on screen since the first `pnpm dev` and we have walked past it every time.

5.9. The Next.js DevTools

In the bottom-left corner of every page the dev server serves sits a small dark badge with the Next.js logo. It reads as decoration at first glance, or as something that will eventually need dismissing. It is neither: it is the framework's own instrument panel, and it answers questions about the running application that the page itself cannot show us.

Click it and a short menu opens:

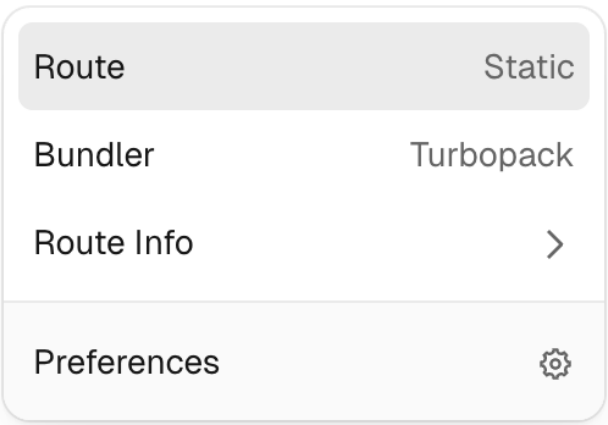


Figure 22. The DevTools Menu on a Fresh Project

Four entries, and each one answers a question we would otherwise have to infer.

Route reports how the page we are looking at was produced. On our scaffold it reads `Static`, meaning Next.js rendered it once and serves the same HTML to everyone. That value is a single word

now and becomes the most consequential fact about a page later, when routes start reading data that differs per request.

Bundler reads `Turbopack`, which confirms what section 5.8.1 claimed rather than asking us to take it on faith. It is a useful line the day a project inherits a configuration from somewhere else and nobody is sure which bundler is actually running.

Route Info opens a submenu listing the files that compose the current route: which layout wraps it, which page renders it, which boundaries sit in between. On a scaffold with one page that is a short list. It stops being short as soon as Chapter 6 starts nesting layouts, and it is the fastest way to answer "which layout is this heading coming from" without reading the directory tree.

Preferences holds the placement and size of the overlay, and one setting worth knowing before we meet it by accident. **Hide Dev Tools for this session** removes the badge until the dev server restarts, and it sits close enough to the other entries that a stray click finds it. If the badge disappears and we did not mean to hide it, restarting `pnpm dev` brings it back.



Warning: The Overlay Is a Development Tool Only

Everything in this section exists in `next dev` and nowhere else. A production build has no badge, no menu, and no route information, which is the correct trade: none of this should ship to a visitor. It also means the overlay cannot answer questions about production behavior. What it reports is true of the server we are running right now.

The menu is not a fixed list. It reflects what the project has switched on, which is why ours is short: we have configured very little so far. As the configuration grows, the menu grows with it, and the entries that appear are worth reading as a summary of what we have turned on.

Summary

In this chapter, we:

- Established **what Next.js is**, a React framework that provides routing, rendering strategies, API routes, and build infrastructure so we can focus on application code
- Chose the **App Router** over the Pages Router for its Server Components, nested layouts, and streaming support
- Scaffolded the **HelpDesk AI project** with `create-next-app` and understood every generated file
- Configured **VS Code** with extensions, project settings, and a debug launch configuration for server-side code
- Explored the **directory structure**: `src/app/` for routes, `src/` for application code, `public/` for static files, and configuration at the root
- Configured **`next.config.ts`** with `reactCompiler`, `typedRoutes`, `agentRules`, `browserToTerminal` logging, and `typedEnv`
- Set up a **code quality pipeline**: `tsc --noEmit` catches type errors, ESLint catches bugs, Prettier formats code, knip detects dead code, and a `pnpm validate` script bundles all four, enforced by a Husky pre-commit hook on every commit

- Ran the **dev server with Turbopack**, created a **production build**, and understood the build output
- Opened the **Next.js DevTools**, the badge that has been in the corner since the first `pnpm dev`, and read what its four entries report about the running route

We have a running project with a single page. But a real application needs multiple pages, navigation between them, and layouts that persist across routes. In the next chapter, we'll build the routing structure for HelpDesk AI: public portal, authentication pages, and the agent dashboard, all organized with route groups and nested layouts.

6. Routing and Navigation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

6.1. File-Based Routing in the App Router

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

6.1.1. A First Example

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

6.1.2. Colocation: Non-Route Files in `app/`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

6.2. Layouts, Templates, and Nested Routes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

6.2.1. Layouts: Persistent Shared UI

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

6.2.2. Nested Layouts

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

6.2.3. When Layouts Re-Render

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

6.2.4. Templates: Layout's Short-Lived Sibling

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

6.2.5. Nested Routes in Practice

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

6.2.6. Preserved Pages: React's Activity Component

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

6.3. Dynamic Routes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

6.3.1. Single Dynamic Segments: `[param]`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

6.3.2. Catch-All Segments: `[...slug]`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

6.3.3. Optional Catch-All: `[[...slug]]`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

6.3.4. Generating Static Params

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

6.4. Route Groups and Parallel Routes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

6.4.1. Route Groups: Layouts Without URL Segments

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

6.4.2. The HelpDesk AI Route Groups

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

6.4.3. The Public Layout

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

6.4.4. The Auth Layout

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

6.4.5. The Dashboard Layout

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

6.4.6. Parallel Routes: Multiple Slots in One Layout

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

6.5. Loading, Error, and Not-Found Boundaries

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

6.5.1. `loading.tsx`: Instant Feedback While Data Loads

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

6.5.2. `error.tsx`: Graceful Error Recovery

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

6.5.3. `global-error.tsx`: The Last Safety Net

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

6.5.4. `not-found.tsx`: Handling Missing Content

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

6.6. Navigation: `<Link>`, `useRouter`, `useLinkStatus`, and `redirect`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

6.6.1. `<Link>`: The Default Navigation Primitive

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

6.6.2. Prefetching Behavior

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

6.6.3. Additional `<Link>` Props

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

6.6.4. `useLinkStatus`: Tracking Navigation State

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

6.6.5. `useRouter`: Programmatic Navigation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

6.6.6. `redirect`: Server-Side Redirects

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

6.7. Search Params and URL State Management

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

6.7.1. Accessing Search Params

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

6.7.2. Type-Safe Query Strings with `qs`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

6.7.3. Updating Search Params from Client Components

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

6.7.4. Example: Ticket Inbox Filters

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

6.8. Intercepting Routes and Modals

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

6.8.1. Intercepting Routes: The `(.)` Convention

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

6.8.2. File Structure for Ticket Preview

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

6.8.3. When to Use Intercepting Routes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

7. Understanding Rendering Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

7.1. The Rendering Problem

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

7.1.1. Where and When

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

7.1.2. What We'll Cover

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

7.2. Server Components vs. Client Components

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

7.2.1. Server Components: The Default

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

7.2.2. Client Components: Opting Into Interactivity

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

7.2.3. The Key Mental Shift

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

7.2.4. The Decision Tree

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

7.2.5. The `'use client'` Boundary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

7.2.6. The Import Rule

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

7.2.7. Composing Server and Client Components

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

7.2.8. HelpDesk AI Component Mapping

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

7.3. Cached Rendering and the `'use cache'` Directive

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

7.3.1. Caching Is Opt-In

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

7.3.2. The `'use cache'` Directive

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

7.3.3. Reading the Build Output

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

7.4. Dynamic Rendering

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

7.4.1. What Triggers Dynamic Rendering?

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

7.4.2. `connection()`: Explicit Dynamic Opt-In

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

7.4.3. Draft Mode: Previewing Unpublished Content

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

7.4.4. Dynamic Content in HelpDesk AI

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

7.5. Streaming and Suspense

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

7.5.1. How Streaming Works

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

7.5.2. `loading.tsx` Is a Suspense Boundary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

7.5.3. Granular Streaming with `<Suspense>`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

7.5.4. Nested Suspense

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

7.5.5. Streaming in HelpDesk AI

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

7.5.6. When a Streamed Region Fails

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

7.6. Cache Revalidation: Keeping Cached Pages Fresh

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

7.6.1. `cacheLife()`: Time-Based Cache Duration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

7.6.2. Custom Cache Profiles

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

7.6.3. `cacheTag()` and `revalidateTag()`: On-Demand Invalidation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

7.6.4. `revalidatePath()`: Path-Based Invalidation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

7.6.5. Cache Revalidation in HelpDesk AI

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

7.6.6. Time-Based vs. On-Demand: When to Use Which

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

7.7. Partial Prerendering (PPR)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

7.7.1. Cached Shell, Dynamic Holes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

7.7.2. How PPR Works

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

7.7.3. PPR and Cache Components

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

7.7.4. PPR in HelpDesk AI

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

7.7.5. Prefetching the Shell

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

7.8. Understanding Hydration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

7.8.1. The Hydration Timeline

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

7.8.2. Hydration Mismatches

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

7.8.3. How to Avoid Mismatches

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

7.8.4. Hydration in HelpDesk AI

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

7.9. Practical Patterns and Anti-Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

7.9.1. Pattern: Push `'use client'` Down the Tree

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

7.9.2. Pattern: Server Component Fetches, Client Component Displays

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

7.9.3. Pattern: Serializable Props Only

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

7.9.4. Anti-Pattern: `'use client'` on Everything

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

7.9.5. Anti-Pattern: Importing Server Code in Client Components

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

7.9.6. Anti-Pattern: Conditional `'use client'`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

7.9.7. The `server-only` and `client-only` Packages

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

7.10. What the DevTools Show Now

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

8. Data Fetching

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

8.1. The Data Problem

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

8.1.1. Two Sides of the Data Problem

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

8.1.2. What We'll Build

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

8.1.3. A Note on the Data Store

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

8.2. Fetching Data in Server Components

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

8.2.1. Upgrading the Data Store

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

8.2.2. Query Functions with Parameters

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

8.2.3. Wiring the Inbox Page

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

8.2.4. When to Cache, and When Not To

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

8.2.5. Separating Queries from Components

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

8.3. The `cache()` Function and Request Memoization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

8.3.1. The Problem: Duplicate Queries

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

8.3.2. React `cache()` for Request Deduplication

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

8.3.3. `cache()` vs. `'use cache'`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

8.3.4. Our Convention: Queries as Cached Functions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

8.4. Server Actions: Mutations Without API Endpoints

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

8.4.1. The Concept

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

8.4.2. The `'use server'` Directive

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

8.4.3. The `ActionResult<T>` Pattern

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

8.4.4. Introducing Zod for Input Validation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

8.4.5. The Five-Step Pattern

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

8.4.6. Creating a Ticket

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

8.4.7. Replying to a Ticket

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

8.4.8. Updating Ticket Status

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

8.4.9. Error Handling in Server Actions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

8.5. Calling Server Actions from Client Components

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

8.5.1. The Direct Call Pattern

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

8.5.2. `useActionState`: Form State from Server Actions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

8.5.3. `useFormStatus`: Pending UI in Nested Components

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

8.5.4. The Submit Page

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

8.5.5. Updating the Reply Form

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

8.6. Optimistic Updates

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

8.6.1. The Problem

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

8.6.2. The `useOptimistic` Hook

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

8.6.3. When to Use Optimistic Updates

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

8.7. Bulk Actions: Mutating Multiple Records

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

8.7.1. The Server Action

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

8.7.2. The Bulk Selection UI

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

8.8. Parallel vs. Sequential Fetching

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

8.8.1. The Waterfall Problem

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

8.8.2. Parallel Fetching with `Promise.all`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

8.8.3. When Sequential Is Correct

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

8.8.4. Streaming as an Alternative

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

8.9. Caching Strategy: Putting the Layers Together

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

8.9.1. The Three Layers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

8.9.2. The Caching Decision

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

8.9.3. Invalidating the Cache After Mutations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

8.9.4. The Tracking Page

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

8.9.5. HelpDesk AI Caching Strategy

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

9. Styling and UI Components

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

9.1. The Styling Problem

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

9.1.1. Why We Need a Styling Strategy

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

9.1.2. Why Tailwind CSS + Headless UI

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

9.1.3. What We'll Build

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

9.2. Tailwind CSS: Setup and Configuration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

9.2.1. Verifying the Installation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

9.2.2. Configuration Lives in CSS

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

9.2.3. The HelpDesk AI Color System

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

9.2.4. The `cn()` Helper

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

9.2.5. Tailwind Class Order

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

9.3. Building a Shared Component Library

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

9.3.1. Why a Component Library

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

9.3.2. How Headless UI Works

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

9.3.3. Spinner

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

9.3.4. Button

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

9.3.5. Icons with Heroicons

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

9.3.6. Dialog

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

9.3.7. The Rest of the Library

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

9.4. CSS Modules as an Alternative

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

9.4.1. When Tailwind Isn't the Right Fit

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

9.4.2. A Brief Example

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

9.4.3. Trade-Offs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

9.5. Font Optimization with `next/font`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

9.5.1. The Problem

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

9.5.2. `next/font`: Self-Hosted, Zero Layout Shift

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

9.5.3. Implementation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

9.6. Image Optimization with `next/image`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

9.6.1. The Problem

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

9.6.2. What `next/image` Does

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

9.6.3. Known Dimensions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

9.6.4. Fill Mode

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

9.6.5. Remote Images

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

9.6.6. The `public/` Directory

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

9.7. Dark Mode, Theming, and Responsive Layouts

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

9.7.1. Dark Mode with Tailwind CSS

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

9.7.1.1. The Strategy

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

9.7.1.2. The Theme Provider

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

9.7.1.3. The Toggle Component

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

9.7.1.4. Applying Dark Styles

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

9.7.1.5. Theme-Aware Design Tokens

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

9.7.2. Responsive Layouts

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

9.7.2.1. Mobile-First with Tailwind's Breakpoints

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

9.7.2.2. The Dashboard Layout

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

9.7.2.3. The Public Portal

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

9.7.2.4. Consistent Spacing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

9.8. Styling the HelpDesk AI Application

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

9.8.1. What We're Styling

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

9.8.2. Dashboard Layout

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

9.8.3. Ticket Inbox

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

9.8.4. Ticket Detail

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

9.8.5. Public Portal

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

9.8.6. Auth Layout

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

9.8.7. Loading Skeletons

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

Part III, Deepening the Practice Project: Full-Stack Architecture

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

10. Project Architecture and Data Model

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

10.1. Why Architecture Matters

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

10.1.1. The Two Extremes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

10.1.2. The Stack We're Committing To

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

10.2. Feature Slices: Organizing Code by Domain

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

10.2.1. By Domain, Not by Type

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

10.2.2. Where We Stand

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

10.2.3. The Target Structure and Its Rules

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

10.2.4. Turning the Data Store into a Private Layer

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

10.2.5. Extracting Queries into the Slice

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

10.2.6. Relocating the Server Actions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

10.2.7. Domain Components and What Stays Shared

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

10.3. Designing the Data Model as a Prisma Schema

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

10.3.1. A Quick Look at Prisma's Schema Language

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

10.3.2. Users, Organizations, and Membership

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

10.3.3. Classifying Tickets: Categories and Tags

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

10.3.4. The Ticket at the Center

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

10.3.5. Communication: Replies and Internal Notes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

10.3.6. Attachments: One Polymorphic File Model

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

10.3.7. The Knowledge Base: Articles

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

10.3.8. Configuration and the Audit Trail

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

10.3.9. Design Decisions at a Glance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

10.3.10. Installing Prisma and Validating the Schema

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

10.4. Shared Types and the Scope System

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

10.4.1. Re-exporting Prisma Types

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

10.4.2. Feature-Specific Types

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

10.4.3. Input Types vs. Output Types

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

10.4.4. Scope-Based Authorization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

11. Database and ORM

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

11.1. From Schema to Running Database

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

11.2. Setting Up Prisma with PostgreSQL

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

11.2.1. What Prisma Still Needs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

11.2.2. The Prisma Client Singleton

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

11.2.3. Connecting to PostgreSQL

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

11.3. Migrations: Evolving the Schema

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

11.3.1. The First Migration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

11.3.2. Evolving the Schema

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

11.3.3. Development vs. Production

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

11.4. Type-Safe Queries with Prisma Client

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

11.4.1. The Four Operations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

11.4.2. Loading Only What the UI Needs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

11.4.3. Filtering, Sorting, and Pagination

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

11.4.4. Relations and Transactions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

11.4.5. Mutations Through Server Actions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

11.5. Seeding the Database

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

11.5.1. Wiring Up the Seed

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

11.5.2. Writing the Seed

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

11.6. Full-Text Search Across Tickets

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

11.6.1. How PostgreSQL Searches Text

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

11.6.2. Indexed, Ranked Search

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

11.7. Connection Pooling for Serverless

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

11.7.1. The Pooler and the Direct Connection

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

11.8. Aside: MongoDB as an Alternative

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

11.8.1. When a Document Database Fits

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

11.8.2. Prisma with MongoDB

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

12. Authentication and Authorization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

12.1. The Problem: Open Doors Everywhere

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

12.2. Auth.js: The Authentication Layer

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

12.2.1. Auth.js or a Hosted Provider?

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

12.2.2. Installing and Configuring

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

12.2.3. The Auth API Route

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

12.2.4. The Auth.js Database Tables

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

12.2.5. Environment Variables

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

12.3. Google OAuth: Setting Up the Provider

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

12.3.1. Creating a Google OAuth App

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

12.3.2. The Login Page

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

12.4. Session Management

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

12.4.1. JWT vs. Database Sessions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

12.4.2. Enriching the Session

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

12.4.3. Teaching TypeScript About the New Fields

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

12.4.4. Reading the Session

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

12.5. Protecting Routes with the Proxy

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

12.5.1. The Proxy

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

12.5.2. Authorization Error Pages

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

12.6. Scope-Based Access Control

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

12.6.1. Enforcing Scopes in Server Actions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

12.6.2. Enforcing Isolation in Queries

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

12.6.3. Gating the UI

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

12.7. Two Flows: Customer Portal and Agent Dashboard

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

12.7.1. The Customer Portal

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

12.7.2. The Agent Flow and Onboarding

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

12.8. The Trade-offs of Email and Password

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

13. Forms, Validation, and File Management

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

13.1. The Problem: Forms That Fight Back

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

13.2. React Hook Form: Type-Safe Form State

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

13.2.1. The Core API

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

13.2.2. Calling the Server Action

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

13.3. Zod: One Schema, Both Sides

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

13.3.1. The Ticket Schema

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

13.3.2. Plugging Zod into the Form

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

13.3.3. Validating Again on the Server

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

13.3.4. Refinements and Transforms

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

13.3.5. The Same Move, Four More Times

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

13.4. Aside: Joi as an Alternative

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

13.5. Controlled Components with Controller

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

13.6. Rich Text with Tiptap

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

13.6.1. The Editor

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

13.6.2. The Toolbar

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

13.6.3. Storing Rich Text

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

13.7. File Uploads with S3 Presigned URLs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

13.7.1. Provisioning the Bucket

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

13.7.2. The S3 Client

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

13.7.3. The Upload Handshake

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

13.8. The Complete Forms

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

13.8.1. Canned Responses

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

14. Email, State Management, and Real-Time

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

14.1. Three Missing Pieces

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

14.2. Transactional Emails with AWS SES

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

14.2.1. Why AWS SES

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

14.2.2. Templates as React Components

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

14.2.3. Rendering and Sending

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

14.2.4. Triggering Email Without Blocking the User

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

14.3. State Management with Zustand

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

14.3.1. Why We Need a Store at All

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

14.3.2. Zustand Fundamentals

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

14.3.3. Persisting Drafts Across a Refresh

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

14.3.4. Hydrating a Store from the Server

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

14.4. SWR for Server State

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

14.4.1. Server State Is a Different Animal

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

14.4.2. When Server Components, When SWR

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

14.4.3. Seeding the Cache from the Server

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

14.4.4. The Three-Way Decision

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

14.5. Dashboard Visualizations with Recharts

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

14.6. Activity Feed and Webhook Dispatch

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

14.6.1. Logging Every Event

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

14.6.2. Webhook Dispatch

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

14.7. Real-Time Updates with Server-Sent Events

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

14.7.1. Why Server-Sent Events

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

14.7.2. An Event Bus and an SSE Endpoint

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

14.7.3. Consuming the Stream

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

14.8. SLA Countdown Timer

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

Part IV: AI Features with the Vercel AI SDK

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

15. Introduction to the Vercel AI SDK

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

15.1. The Missing Intelligence

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

15.2. Two Libraries, Two Responsibilities

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

15.3. One API, Many Models: The Gateway

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

15.4. Setting Up the AI SDK in HelpDesk AI

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

15.4.1. The Model Resolver

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

15.4.2. Configuration in One Place

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

15.5. Tracking Token Usage From Day One

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

15.5.1. The Tracking Utility

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

15.5.2. Enforcing a Budget

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

15.6. The AI Settings Page

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

15.7. The First AI Call: A Ticket Title Generator

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

15.7.1. Reading the Response

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

16. Text Generation and Streaming

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

16.1. The Waiting Problem

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

16.2. `generateText` vs. `streamText`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

16.3. The Streaming Route Handler

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

16.3.1. The Guard Ladder

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

16.3.2. Failing After the Headers Are Gone

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

16.4. Streaming Into the UI With `useCompletion`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

16.5. System Prompts: Defining the Job

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

16.6. Generation Parameters

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

16.6.1. Temperature, and What `0` Does Not Buy

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

16.6.2. Reasoning Effort, the Dial That Replaced It

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

16.6.3. `maxOutputTokens` and the Knobs That Do Nothing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

16.7. The Right Runtime for AI Routes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

16.8. Wiring the Summary Into the Ticket Page

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

17. Building a Chat Interface

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

17.1. From One-Shot to Conversation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

17.2. The `useChat` Hook

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

17.2.1. The Chat Component

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

17.3. The Copilot Route Handler

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

17.4. Multi-Turn Context and Token Limits

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

17.4.1. The Output Budget Pays for Thinking First

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

17.5. Persisting and Resuming Conversations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

17.6. Keeping the Newest Message in View

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

17.7. The Copilot on the Ticket Page

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

17.8. The Standalone Copilot Page

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

18. Structured Output and Tool Calling

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

18.1. Beyond Free-Form Text

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

18.2. `Output.object`: Type-Safe Structured Output

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

18.2.1. Why Not Just Ask for JSON?

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

18.2.2. Streaming a Structured Result

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

18.3. Auto-Categorization and Sentiment

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

18.4. Tool Calling: Giving the Model Hands

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

18.5. Multi-Step Execution with `stopWhen`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

18.5.1. Budgeting a Whole Turn

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

18.6. Human-in-the-Loop with `toolApproval`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

18.7. Scope-Protected Execution

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

19. Retrieval-Augmented Generation (RAG)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

19.1. Understanding Embeddings

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

19.2. Text Chunking

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

19.3. Setting Up pgvector

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

19.4. The Embedding Model Configuration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

19.5. The Knowledge Base Becomes Real

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

19.6. The Embedding Pipeline

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

19.7. The Retrieval Pipeline

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

19.8. Prompt Engineering With Retrieved Context

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

19.9. The Payoff: Semantic Tools and Suggested Replies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/comprehensive-nextjs-guide.

Index

App Router, 52
async/await, 35
Auth.js, i, iv
authentication, ii
AWS S3, iii
AWS SES, iii

Claude Code, iv
Client Components, 52
create-next-app, 51

declaration files, 24
discriminated union, 11
Docker, iv, vii

embedding pipeline, 136
environment variables, 63
error.tsx, 62
ES Modules, 21
esbuild, 6
ESLint, 51

Fast Refresh, 61

generics, 14
global-error.tsx, 62

HelpDesk AI, i
HMR, 61
Hot Module Replacement, 61
Husky, 51
Husky, pre-commit hook, 72

import type, 21

keyof, 14
knip, 51

layout.tsx, 55
loading.tsx, 53

max-warnings flag, 74
Mixpanel, i
multi-turn conversation, 132

next-intl, iv
next.config.ts, 51
Next.js, i, 51
Next.js DevTools, 80

Node.js, 4
not-found.tsx, 62

page.tsx, 51
Pages Router, 52
Playwright, iv
pnpm, vi, 5, 6, 71
pnpm verify, 74
Prettier, 51
Prisma, i, iv
Promises, 35

RAG, ii
React, i
React Compiler, 54, 74
React Hook Form, iv
retrieval-augmented generation, ii
route.ts, 51

satisfies, 10, 27
Sentry, i
Server Components, v, 53
Server-Sent Events, iii
streaming, ii

Tailwind CSS, 54
text chunking, 136
token usage, 128
tool calling, ii
tsc, noEmit, 73
tsconfig.json, 22, 52
tsx, 6
Turbopack, 52
type narrowing, 13
TypeScript, ii, 8, 51
TypeScript 7, 74

union type, 10
unknown type, 9
utility types, 17

Vercel, i
Vercel AI SDK, i
Vitest, iv
VS Code, iv, 6, 51

Zod, iv
Zustand, iv