COMPREHENSIVE GUIDE

Next.js

The Comprehensive Guide

```
export default async function Page() {
  const ai = await generateText({
    model: openai('gpt-4o'),
    prompt: 'Build something amazing'
  });
  return <main>{ai.text}</main>
}
```

From React Fundamentals to AI-Powered Full-Stack Apps

React 19 · App Router · Server Components · Vercel AI SDK · Prisma · Auth.js

AI Includes AI-assisted development workflows

WRITTEN BY
Florian Wessels

1ST EDITION · 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 2026-03-29

Contents

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, Auth0, Vercel, Mixpanel, and Sentry. That's not a random collection of buzzwords; it's the stack we'll work with throughout this book, battle-tested in production with real users and real scale.

Today, I lead product and technology at a Swiss company, where I design and implement 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

Around 90% of engineering teams already use AI tools in some capacity. 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* your 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 you how to fetch data and render a component. They don't show you 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. You know your way around `async/await`, you've written functions and classes, and you 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 you 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, suggests replies based on past solutions, and learns from resolved cases.

Why This Project?

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

- **Authentication and roles** – customers, agents, admins, and viewers, each with different permissions and 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, 28 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 — no project code yet, but 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 26–28): Advanced topics — 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.

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 you — 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 use AI tools in Chapters 5–27 so we build that understanding first. Chapter 28 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 on your desk.

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 16.2**. 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 and 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. Claude Code, Cursor, and Windsurf are all built on VS Code, so extensions and settings work the same way in all of them. If you prefer a different editor (WebStorm, Neovim, Zed), see Appendix F for setup instructions.
- **Git** – for version control
- **Docker** – needed for Chapter 25 (self-hosting). Not required until then.

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 free-tier accounts for the following services. They're introduced progressively – there's no need to sign up for everything before Chapter 1:

- **GitHub** – for source code and version control (from Chapter 5)
- **Vercel** – for deployment (from Chapter 24, free tier is sufficient)
- **Google Cloud Console** – for OAuth login (from Chapter 12, free)
- **AWS** – for S3 file storage and SES email (from Chapter 13, free tier)
- **OpenAI and/or Anthropic** – for AI features (from Chapter 15)

Cost

All tools and services used in this book have free tiers. AI API usage incurs minor costs – typically under $5 USD for all examples in the book. 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

A Note on the State of This Book

This book is published as a work in progress on Leanpub. That means early access to chapters as they're written, and every update is a free download for existing readers.

The current version contains Part I (Chapters 1–4), covering TypeScript, modern JavaScript, and React fundamentals, and Part II (Chapters 5–9), covering Next.js core concepts: project setup, routing, rendering strategies, data fetching, and styling with Tailwind CSS and Headless UI. Parts III–VI and the appendices are in active development.

If you spot errors, have suggestions, or want to share what you'd like to see covered in more depth – we'd love to hear from you. Reader feedback directly shapes this book.

Thank you for joining early. Let's build something real together.

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 you debugging time.

Background

Additional context or a deeper explanation. Safe to skip on first reading without losing the thread. Come back to these when you want 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

Sections like this are where we actively build parts of HelpDesk AI. They're step-by-step, with testable results at the end. Roll up your sleeves.

Code Examples

Every code block has a file path on the first line (either as a title attribute or a comment), so we always know where the code belongs. All code is TypeScript — never JavaScript. Relevant lines are highlighted; we don't highlight entire blocks. The maximum line width is 85 characters, optimized for reading in both digital and print formats.

Terminal Commands

Terminal commands are shown for macOS and Linux. Where behavior differs on Windows, we add a note. All package management uses pnpm — if you prefer npm or yarn, the commands are similar but not always identical.

Cross-References

References to other chapters are specific: "as we saw in section 7.3" — never "as previously mentioned." Forward references are meant to spark curiosity: "we'll explore this in detail in Chapter 19."

* * *

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.

Already know TypeScript?

If you use TypeScript daily, skim or skip this chapter. Here's a quick self-check – if you can confidently explain all of these, 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. That's why we'll add runtime validation with Zod later in the book.

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 you've used Java, C#, Go, or Python's type hints, TypeScript will feel familiar – but some differences can trip you 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
```

Version Manager

If you work on multiple projects with different Node.js versions, use `fnm` (Fast Node Manager) or `nvm`. They let you switch Node.js versions per project with 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
```

Alternatively, use Node.js's built-in `corepack`:

```
$ corepack enable
```

Verify:

```
$ pnpm -v
10.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!)

Using npm or yarn Instead

All commands in this book use pnpm. If you prefer npm or yarn, 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.

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).

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 1. 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.

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:

```
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:

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

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

```
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, but we also want narrow type inference on the values. The `satisfies` operator does both:

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

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

How is this different from a plain type annotation? Compare the two approaches:

```
// With a type annotation — values widen to string
const COLORS: Record<TicketPriority, string> = {
  LOW: '#64748b',
  MEDIUM: '#3b82f6',
  HIGH: '#f59e0b',
  URGENT: '#ef4444',
}
COLORS.LOW  // type: string

// With satisfies — values stay narrow
const COLORS2 = {
  LOW: '#64748b',
  MEDIUM: '#3b82f6',
  HIGH: '#f59e0b',
  URGENT: '#ef4444',
} satisfies Record<TicketPriority, string>
COLORS2.LOW  // type: '#64748b'
```

Both versions check that every `TicketPriority` key is present. The difference is what TypeScript infers for the values. With the annotation, `COLORS.LOW` is `string`. With `satisfies`, `COLORS2.LOW` is the literal `'#64748b'`. That narrow inference matters when we pass values to functions that expect specific strings – or when we want autocomplete to show the exact values.

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. This is the pattern we'll use for all labeled lookup objects in HelpDesk AI – statuses, scopes, roles, and more.

Why Not `as const`?

You may see `as const` in other TypeScript codebases. It also preserves literal types, but it makes the entire object deeply readonly and doesn't verify that all keys are present. `satisfies` gives us the same narrow inference plus a structural check – a better fit for our use case. 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:

```
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:

```
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":

```
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:

```
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:

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

`instanceof` checks for class instances:

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

Truthiness checks for nullable values:

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

The `in` operator for objects:

```typescript
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:

```typescript
// 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:

```typescript
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:

```typescript
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'
```

The `extends` keyword here means "T must have at least an `id` property of type `string`."

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):

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

// Interface: via extends
interface TicketWithAgent 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:

```
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 You Have

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

The inverse of `Partial` – 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. Use `Record` with a type annotation when all values have the same broad type. Use `Record` with `satisfies` (as in Section 1.3.4) when we want to preserve narrow literal types on the values.

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. `Exclude<T, U>` and `Extract<T, U>` — Filter Union Members

These work on union types to remove or keep specific 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'
```

1.5.9. Composing Utility Types

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.

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 { Ticket, CreateTicketInput } from './types'
```

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

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

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. When we ran `pnpm tsc --init` earlier, it generated one with many options commented out. Here are the options that `create-next-app` generates for a Next.js project:

Option	Value	Why
`strict`	`true`	Enables all strict type-checking options. Non-negotiable for production code.
`target`	`"ES2017"`	The JavaScript version to compile to. Next.js sets this for broad browser compatibility – the bundler handles modern syntax.
`module`	`"ESNext"`	Use ES modules (not CommonJS) for better tree-shaking.
`moduleResolution`	`"bundler"`	Resolution strategy compatible with Next.js and modern bundlers.
`jsx`	`"react-jsx"`	Use React's automatic JSX transform – no need to `import React` in every file.
`paths`	`{ "@/*": ["./src/*"] }`	Path aliases so we can write `@/shared/lib/db` instead of `../../../shared/lib/db`.
`noUncheckedIndexedAccess`	`true`	Array access and object indexing return `T | undefined`. Prevents silent `undefined` bugs.
`skipLibCheck`	`true`	Skips type-checking `.d.ts` files. Speeds up compilation without reducing safety for our own code.

`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`.

When we create the HelpDesk AI project, `create-next-app` will auto-generate a `tscon-`

`fig.json` with these core options plus Next.js-specific additions.

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.

We won't write our own `.d.ts` files in this book. It's 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 name: string = null  // allowed

// With strictNullChecks (our setting):
const name: string = null  // Error!
const name: 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:

```ts
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:

```ts
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, create a declaration file to silence the error:

```ts
// types/untyped-lib.d.ts
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:

```ts
// 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:

```ts
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
```

This feels wrong at first if you're used to nominal typing. 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):

```typescript
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 the Hands-On section.

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 in a hands-on exercise using HelpDesk AI domain concepts.

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.

Step 1: Ticket Status — String Unions and Type Narrowing

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

Figure 2. ticketStatus.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.

Step 2: API Response — Generics

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

Figure 3. apiResponse.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`.

Step 3: User Types — Utility Types

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

Figure 4. userTypes.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 creating a new user – no server-generated fields
type CreateUserInput = Omit<User, 'id' | 'createdAt'>

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

// For configuration – immutable after creation
type UserConfig = Readonly<Pick<User, 'id' | 'role' | 'organizationId'>>

// 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',
   ...
}
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.

Step 4: Scope Definitions — `satisfies` and `Record`

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

Figure 5. scopeDefinitions.ts

```
type Scope =
  | 'tickets:read'
  | 'tickets:write'
  | 'tickets:assign'
  | 'tickets:delete'
  | 'knowledge:read'
  | 'knowledge:write'
  | 'settings:manage'
  | 'members:manage'
  | 'ai:use'
  | 'analytics:view'

type ScopeInfo = {
  label: string
  description: string
```

```
  category: 'tickets' | 'knowledge' | 'admin' | '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'
  },
  'tickets:assign': {
    label: 'Assign Tickets',
    description: 'Assign tickets to agents',
    category: 'tickets'
  },
  'tickets:delete': {
    label: 'Delete Tickets',
    description: 'Soft-delete tickets',
    category: 'tickets'
  },
  'knowledge:read': {
    label: 'View Articles',
    description: 'Read knowledge base articles',
    category: 'knowledge'
  },
  'knowledge:write': {
    label: 'Edit Articles',
    description: 'Create and edit articles',
    category: 'knowledge'
  },
  'settings:manage': {
    label: 'Manage Settings',
    description: 'Organization and category settings',
    category: 'admin'
  },
  'members:manage': {
    label: 'Manage Members',
    description: 'Invite agents, assign roles',
    category: 'admin'
  },
  'ai:use': {
    label: 'Use AI Copilot',
    description: 'Generate suggestions and summaries',
    category: 'ai'
  },
  'analytics:view': {
```

```
    label: 'View Analytics',
    description: 'Dashboard metrics and reports',
    category: 'admin'
  }
} 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 agentScopes: Scope[] = [
  'tickets:read',
  'tickets:write',
  'ai:use'
]

console.log('Can read tickets:', hasScope(agentScopes, 'tickets:read'))
console.log('Can manage settings:', hasScope(agentScopes, 'settings:manage'))
```

Run it:

```
$ pnpm tsx scopeDefinitions.ts
View Tickets
Can read tickets: true
Can manage settings: 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`. This pattern becomes the foundation for HelpDesk AI's permission system.

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.

2.7. 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.

`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.

`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.

`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 you'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 your code for production. You'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 your Core Web Vitals score. It needs a build pipeline that bundles, minifies, and code-splits your 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 — 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 you write the component.
- **API routes** — Backend endpoints live alongside your 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. You 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 is seamless, but Chapter 25 covers self-hosting with Docker and AWS. The framework doesn't lock you in.

5.1.3. Where This Chapter Fits

If you're coming from a backend framework like Spring Boot, Express, or Django, the concept of a framework wrapping a UI library will feel familiar. Next.js is to React what Spring Boot is to Java's Servlet API – it provides structure, opinions, and batteries-included infrastructure.

If you've built React SPAs with Vite or Create React App, the key shift is this: Next.js renders on the server by default. Components run on the server, not in the browser, unless you explicitly opt them into client-side behavior. That's a paradigm shift, and Chapter 7 is dedicated to understanding it.

If you already know Next.js from the Pages Router era, this chapter focuses on Next.js 16.2 with the App Router. The rendering model, file conventions, and data fetching patterns are different from what you've used.

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 isn't deprecated – it continues to work in Next.js 16.2 and will be maintained. But all new features and improvements go into the App Router. For new projects, there's no reason to choose the Pages Router. If you encounter tutorials or documentation that reference `getServerSideProps`, `_app.tsx`, or the `pages/` directory, they're using 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 every reader.

5.3.1. Scaffolding the HelpDesk AI Project

Open your terminal and run:

```
$ pnpm create next-app@latest helpdesk-ai \
    --ts --tailwind --eslint --app --src-dir \
    --react-compiler --empty
```

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.

The script asks two interactive questions. Accept the defaults for both:

- **Import alias** (`@/*` by default) – Keep the default. This maps `@/*` to `src/*`, so we can write `import { db } from '@/shared/lib/db'` instead of relative paths like `../../../shared/lib/db`.
- **Include AGENTS.md** – Say yes. This generates `AGENTS.md` and `CLAUDE.md` at the project root, pointing AI coding agents (Claude Code, Cursor, GitHub Copilot) to the bundled Next.js documentation in `node_modules/next/dist/docs/` instead of relying on potentially outdated training data. If you don't use AI coding tools, you can ignore these files. We'll cover AI-assisted development in detail in Chapter 28.

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

Tip

If you need to fall back to webpack (for an unsupported loader or plugin), pass `--webpack` to `create-next-app`. You can also switch after the fact by running `next dev --webpack` or `next build --webpack`.

5.3.2. What Gets Generated

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

```
helpdesk-ai/
├── .next                # Next.js 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
├── AGENTS.md            # AI agent instructions
├── CLAUDE.md            # Claude-specific instructions
├── 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  # Workspace configuration
├── 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:

Figure 6. 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,
```

```
}: Readonly<{
  children: React.ReactNode;
}>) {
  return (
    <html lang="en">
      <body>{children}</body>
    </html>
  );
}
```

`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 the highlighted line:

Figure 7. tsconfig.json (relevant section)

```
{
  "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.

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

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

`AGENTS.md` and `CLAUDE.md` – The AI agent guides we mentioned above. `CLAUDE.md` imports `AGENTS.md` via `@AGENTS.md`. We'll explore how to customize these files and work with AI coding agents in Chapter 28.

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 you run the code. Typos in variable names, missing imports, incorrect types, formatting inconsistencies – all caught inline, in real time, as you 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. Claude Code, Cursor, and Windsurf are all built on VS Code – extensions and settings work the same way in all of them.

Warning

This section uses VS Code terminology. If you use a different editor (WebStorm, Neovim, Zed), the terminal-based tools (pnpm, ESLint, Prettier, Husky) work identically. You'll need to find equivalent settings for your editor's TypeScript, ESLint, and formatting integration.

5.4.1. Essential Extensions

Install these five 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 you 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

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

Alternatively, install them from the terminal. If the `code` command is not in your PATH, open the command palette (`Cmd+Shift+P` / `Ctrl+Shift+P`) 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
```

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 your global VS Code configuration:

Figure 8. .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 you save a file, Prettier formats it. You 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 9. .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, you 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 you inspect variables, step through code, and evaluate expressions – the same workflow you'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 16.2.1 (Turbopack)
- Local:         http://localhost:3000
- Network:       http://192.168.1.100:3000
- Debugger port: 9230
✓ Ready in XXXms
```

Open `http://localhost:3000` in your browser. With the `--empty` flag, you'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 10. 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 11. src/app/layout.tsx

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

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

export default function RootLayout({
  children,
}: Readonly<{
  children: React.ReactNode
}>) {
  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 you didn't intend or hide a page you 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 you 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)
`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**. You 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

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)
│   ├── (public)/       ← Customer portal routes
│   ├── (auth)/         ← Login/signup routes
│   └── (dashboard)/    ← Agent dashboard routes
├── features/           ← Feature slices (Ch. 10)
│   ├── tickets/
│   ├── knowledge/
│   ├── auth/
│   └── copilot/
├── shared/             ← Cross-feature code (Ch. 10)
│   ├── components/
│   ├── lib/
│   └── types/
prisma/                 ← Database schema (Ch. 11)
emails/                 ← Email templates (Ch. 14)
```

We don't create all of these directories now. Each chapter adds what it needs. This preview helps you 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 12. 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 13. next.config.ts

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

const nextConfig: NextConfig = {
  reactStrictMode: true,
  reactCompiler: true,
  typedRoutes: true,
  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 you 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 your 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. This also generates `PageProps`, `LayoutProps`, and `RouteContext` type helpers that provide typed access to route parameters – no manual type definitions or imports needed.

`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 your environment variables. During development, Next.js reads your `.env` files and creates a `.d.ts` file in `.next/types/` with type information for each variable. This gives you autocompletion when typing `process.env.` in your editor.

5.6.3. Options We'll Add Later

This isn't the final configuration. We'll add more options in later chapters as features require them – caching, security headers, and deployment settings each bring their own configuration needs.

Tip

You don't need to memorize the configuration API. The `NextConfig` type provides autocomplete for every option – type the property name and let IntelliSense guide you. The Next.js documentation bundled at `node_modules/next/dist/docs/` is the authoritative reference for your installed version.

5.6.4. Environment Variables

Next.js loads environment variables from multiple files in a specific order:

1. `.env` – Defaults for all environments (committed to git)
2. `.env.local` – Local overrides (not committed – in `.gitignore`)
3. `.env.development` – Development-specific (committed)
4. `.env.production` – Production-specific (committed)

Later files override earlier ones. `.env.local` always takes precedence, which makes it the right place for secrets during development.

Create two files at the project root:

Figure 14. .env.example

```
# Database
DATABASE_URL=postgresql://...

# Authentication (Chapter 12)
AUTH_SECRET=
AUTH_GOOGLE_ID=
AUTH_GOOGLE_SECRET=

# AI (Chapter 15)
OPENAI_API_KEY=

# AWS (Chapter 13–14)
AWS_ACCESS_KEY_ID=
AWS_SECRET_ACCESS_KEY=
AWS_REGION=eu-central-1
AWS_S3_BUCKET=

# Public (visible in browser)
NEXT_PUBLIC_APP_URL=http://localhost:3000
```

Figure 15. .env.local

```
# Local development overrides
NEXT_PUBLIC_APP_URL=http://localhost:3000
```

The `.env.example` file documents which variables the project needs – it's committed to git and helps new team members set up their environment. The `.env.local` file holds actual values and is excluded from version control by default (it's already in the generated `.gitignore`).

Warning

Any environment variable prefixed with `NEXT_PUBLIC_` is embedded into the client-side JavaScript bundle and visible to anyone who inspects the page source. Never prefix secrets – database URLs, API keys, authentication secrets – with `NEXT_PUBLIC_`.

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 your 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.

The Generated Configuration

`create-next-app` generates an `eslint.config.mjs` with two configurations commonly referred to as `next/core-web-vitals` and `next/typescript`. In code, you import them from `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.

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
```

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

Figure 16. 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',
      'react/react-in-jsx-scope': 'off',
```

```
      '@typescript-eslint/no-unused-vars': [
        'error',
        {
          argsIgnorePattern: '^_',
          ignoreRestSiblings: true
        }
      ],
      '@typescript-eslint/consistent-type-imports': 'error',
      'no-console': ['warn', { allow: ['warn', 'error'] }]
    }
  },
  globalIgnores([
    "node_modules/**",
    ".next/**",
    "out/**",
    "build/**",
    "next-env.d.ts",
  ]),
]);

export default eslintConfig;
```

This configuration layers multiple rule sets:

- **`next/core-web-vitals`** – Next.js-specific rules for performance (image optimization, link usage)
- **`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 (replaces deprecated formatting rules from ESLint core)
- **`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 – you 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 `next/typescript`) to `"error"`. Warnings don't cause ESLint to exit with a non-zero code, which means they won't block commits or fail CI pipelines. Unused variables are dead code – they should break the build, not whisper politely. 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 `@typescript-eslint/consistent-type-imports` rule enforces `import type { ... }` for type-only imports – the convention we established in Chapter 1. Without this rule, it's easy to write `import { User } from './types'` instead of `import type { User } from`

`'./types'`. The difference matters: type imports are erased at compile time and never end up in the JavaScript bundle. Regular imports can cause unnecessary side effects and circular dependency issues. With this rule set to `"error"`, ESLint auto-fixes the import style for you on save.

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.

Running ESLint

In Next.js 16, the `next lint` command was removed. We run ESLint directly instead – which gives us full control over flags and configuration:

```
$ pnpm lint
```

This runs `eslint .`. ESLint scans all files and reports errors and warnings. To auto-fix what it can:

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

With the VS Code ESLint extension, you see errors inline as you 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.

Configuration

Install Prettier and the Tailwind CSS plugin:

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

Create a `.prettierrc` at the project root:

Figure 17. .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 18. .prettierignore

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

Both files are machine-generated. The lock file must not be reformatted – Prettier would break its structure. The workspace configuration file is managed by pnpm and should stay untouched as well.

Tip

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.

Running Prettier

Add two scripts to `package.json`:

Figure 19. package.json (scripts section)

```
{
  "scripts": {
    "format": "prettier --write .",
    "format:check": "prettier --check ."
  }
}
```

`pnpm format` reformats all files. `pnpm format:check` verifies formatting without changing files – useful in CI pipelines where you 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.

Setup

Install knip:

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

Create a `knip.config.ts` at the project root. Next.js's file conventions require special handling – route files like `page.tsx` and `layout.tsx` are entry points that no other file imports, so knip needs to know about them:

Figure 20. knip.config.ts

```ts
import type { KnipConfig } from 'knip'

const config: KnipConfig = {
  next: {
    entry: [
      'src/app/**/page.tsx',
      'src/app/**/layout.tsx',
      'src/app/**/route.ts',
      'src/app/**/loading.tsx',
      'src/app/**/error.tsx',
      'src/app/**/not-found.tsx',
      'src/app/**/global-error.tsx',
      'src/app/**/unauthorized.tsx',
      'src/app/**/forbidden.tsx',
      'src/proxy.ts'
    ]
  },
  ignoreDependencies: ['postcss']
}

export default config
```

The `next` key tells knip about Next.js file conventions. The `entry` array lists all files that serve as entry points – knip starts its analysis from these files and traces imports outward. The `ignoreDependencies` array lists packages that are used indirectly (PostCSS is referenced in `postcss.config.mjs`, which knip doesn't trace by default).

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`.

Setup

Install Husky:

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

Initialize Husky:

```
$ pnpm exec husky init
```

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

Figure 21. .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.

How It Works

The flow is:

1. You stage files with `git add`
2. You 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: your staged changes might be clean, but they could break something in a file you didn't touch. A full-project check catches this.

Warning

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 22. package.json (complete scripts)

```
{
  "scripts": {
    "dev": "next dev",
    "build": "next build",
    "start": "next start",
    "lint": "eslint",
    "lint:fix": "eslint --fix .",
    "format": "prettier --write .",
    "format:check": "prettier --check .",
    "prepare": "husky",
    "validate": "tsc --noEmit && pnpm lint && pnpm format:check && pnpm knip",
    "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.
2. **`pnpm lint`** runs ESLint across the project.
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.

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`. You can also run it directly from the terminal – useful when you don't need the debugger. Behind the scenes:

- **Turbopack** compiles your TypeScript and JSX into JavaScript that the browser and server can execute. When you 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 you change the button label, the counter stays at `5` after the update. State is preserved unless you change the component's structure in a way that forces a reset.

5.8.2. Browser Log Forwarding

When debugging client-side code, you 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 you 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.

This cache lives in `.next/` and is enabled by default for development (`experimental.turbopackFileSyst` defaults to `true`). For production builds, filesystem caching is opt-in via `experimental.turbopackFileSystemCacheForBuild: true` – we'll enable it in Chapter 24 when we set up CI/CD.

Background

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:

```
$ 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 – that was removed in Next.js 16. Linting happens through `pnpm validate` and the pre-commit hook, not during the build. This makes builds faster and keeps concerns separated: linting is a development-time check, not a build-time gate.

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 your 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: CSS bundles, JavaScript chunks, 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.

You 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. Every subsequent chapter builds on this setup.

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`, `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

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.

6.3. 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. Instant Navigation with `unstable_instant`

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. `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.

6.6.7. `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. Two Caching Models

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. 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.

7.6. 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.

7.8. 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.

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.

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. Dropdown

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

9.3.8. SearchableSelect (Combobox)

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

9.3.9. Tabs

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

9.3.10. Badge

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

9.3.11. Card

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.

HelpDesk AI Image Use Cases

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.

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.

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.

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.

Avoiding the Flash of Wrong Theme

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.

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.

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.

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.

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.

Index