

FULL-STACK WEB DEVELOPMENT

React and Next.js

Building Full-Stack Web Applications

Components · Hooks · App Router · Server Components
Server Actions · Prisma · Auth · Testing · Deployment

REACT 19 · TYPESCRIPT · TAILWIND · POSTGRESQL

THENEXBOOKS

React and Next.js

Building Full-Stack Web Applications

First Edition

THENEXBOOKS

React and Next.js
Building Full-Stack Web Applications

First Edition, 2026

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

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

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. React, Next.js, and the libraries in their ecosystem evolve quickly; always consult the official documentation for the versions you use.

This book is an independent publication and is not affiliated with, sponsored by, or endorsed by Meta Platforms, Inc. or Vercel Inc. React is a trademark of Meta Platforms, Inc. Next.js and Vercel are trademarks of Vercel Inc. TypeScript, Tailwind CSS, Prisma, PostgreSQL, Stripe, Docker, GitHub, Playwright, and other product names are trademarks of their respective owners; their use here is for identification only. Savora is a fictional company.

Preface

React changed how we build user interfaces: describe what the screen should look like for a given state, and let the library keep the page in sync. Next.js took the next step, turning React into a full-stack framework where the same components can render on the server, fetch data directly from a database, stream to the browser, and handle form submissions without a separate API layer. Together they have become one of the most common ways to build modern web applications.

They have also changed a lot in a short time. Server Components, Server Actions, the App Router, new caching rules, and React 19's form and transition APIs mean that many tutorials — and many developers' instincts — are out of date. This book teaches the modern way to build with React and Next.js, and explains the mental models behind it, so that you can reason about new features as they arrive instead of memorizing recipes.

We start with React itself: components, state, effects, and the patterns that keep an application understandable as it grows. We then move to Next.js and the App Router, and build a complete full-stack application with a database, authentication, forms, validation, testing, deployment, and production-quality performance, accessibility, and security.

Who This Book Is For

- **Frontend developers** who know HTML, CSS, and JavaScript and want to master React and move into full-stack work.
- **Backend developers** who want to build complete web applications without juggling separate frontend and API projects.
- **React developers** who learned the older patterns and need to understand Server Components, Server Actions, and the App Router.
- **Students and career changers** who want a structured, practical path to building and shipping real web applications.

What You Will Learn

- React fundamentals done right: components, props, state, events, effects, refs, and custom hooks — in TypeScript.
- Performance, transitions, styling with Tailwind CSS, client-side data with TanStack Query, and component testing.
- The Next.js App Router: layouts, dynamic routes, Server and Client Components, data fetching, caching, and streaming.
- Full-stack features: Server Actions, Route Handlers, PostgreSQL with Prisma, authentication and authorization, forms and validation.

- Production quality: SEO and metadata, images and fonts, Core Web Vitals, accessibility, end-to-end testing, deployment, and security.
- Three complete projects that build a real restaurant discovery and booking platform, including payments in test mode and an AI feature.

How This Book Is Organized

Part I — React Foundations (Chapters 1–5) covers the language, components, state, and effects. **Part II — Advanced React** (Chapters 6–10) explores hooks, performance, styling, client-side data, and testing. **Part III — Next.js and the App Router** (Chapters 11–15) introduces routing, Server and Client Components, data fetching, caching, and streaming. **Part IV — Full-Stack Features** (Chapters 16–21) adds mutations, APIs, a database, authentication, forms, and middleware. **Part V — Production Quality** (Chapters 22–25) covers SEO, performance, accessibility, testing, and deployment. **Part VI — Real-World Projects** (Chapters 26–28) brings everything together. The appendices provide a cheat sheet, templates and API reference, a production checklist, and a glossary.

Every chapter ends with a **Try It Yourself** section and a **Key Takeaways** box.

The Running Example: Savora

Throughout the book we build **Savora**, a fictional restaurant discovery and table-booking platform. Diners browse restaurants by city and cuisine, read menus and reviews, save favorites, and book tables. Restaurant owners manage their profile, menu, opening hours, and bookings in a dashboard. Savora starts as a small client-side React app and grows, chapter by chapter, into a full-stack Next.js application with a database, authentication, and a production deployment.

Which Versions?

The examples use **React 19**, the **Next.js App Router** in recent versions, and **TypeScript** throughout. Next.js in particular evolves quickly; where a behavior depends on the version — caching defaults and newer caching APIs are the main examples — we say so explicitly and show the stable approach first. Experimental features are always labeled.

Conventions Used in This Book

> **TIP**
> A recommended practice or a technique that makes your work easier.

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

> **WARNING**
>
> A common mistake, a security risk, or a behavior that can cause bugs in production.

Code appears in `monospace`. When a code block shows a file, its path appears above it — for example `app/restaurants/[slug]/page.tsx`.

Contents

Preface

PART I — REACT FOUNDATIONS

1 Why React and Next.js? The Modern Full-Stack Landscape

2 Modern JavaScript and TypeScript for React Developers

3 Components, JSX, and Props

4 State, Events, and Forms

5 Effects, Refs, and Synchronization

PART II — ADVANCED REACT

6 Hooks in Depth: Custom Hooks, useReducer, and Context

7 Rendering and Performance: Memoization, Transitions, and the React Compiler

8 Styling React Applications

9 Client-Side Data and State Management

10 Testing React Components

PART III — NEXT.JS AND THE APP ROUTER

11 Getting Started with Next.js and the App Router

12 Routing: Layouts, Dynamic Routes, and Navigation

13 Server Components and Client Components

14 Data Fetching, Caching, and Revalidation

15 Streaming, Suspense, Loading and Error UI

PART IV — FULL-STACK FEATURES

16 Server Actions and Mutations

17 Route Handlers and APIs

18 Databases with Prisma and PostgreSQL

19 Authentication and Authorization

20 Forms, Validation, and File Uploads

21 Middleware, Internationalization, and Edge Concerns

PART V — PRODUCTION QUALITY

22 SEO, Metadata, Images, Fonts, and Web Performance

23 Accessibility and UX Quality

24 End-to-End Testing and Quality Automation

25 Deployment, Observability, and Security

PART VI — REAL-WORLD PROJECTS

26 Project: The Savora Public Site

27 Project: Savora Bookings and the Owner Dashboard

28 Project: Payments, Real-Time Updates, AI Features, and Next Steps

APPENDICES

A React and Next.js Cheat Sheet

B Project Templates and API Reference

C Production Checklist

D Glossary

PART I

React Foundations

The language and the core ideas: modern JavaScript and TypeScript, components, props, state, events, and effects — everything React applications are built from.

1 · Why React and Next.js?
2 · Modern JavaScript and TypeScript
3 · Components, JSX, and Props
4 · State, Events, and Forms
5 · Effects, Refs, and Synchronization

CHAPTER 1 · PART I — REACT FOUNDATIONS

Why React and Next.js? The Modern Full-Stack Landscape

Before we write a single component, we need a map. This chapter explains how web applications are rendered today, where React and Next.js fit into that picture, and why the combination has become a common choice for full-stack work. We finish by installing the toolchain, creating the first version of Savora, and running it in the browser.

What We Mean by a "Full-Stack Web Application"

A full-stack web application is software that spans the whole path from a person's browser to a database and back. It has a user interface that runs in the browser, logic that runs on a server, and some form of persistent storage. The interesting part is not any single layer but the seams between them: how data travels from the database into HTML, how a click turns into a network request, how the server decides who is allowed to do what, and how the interface stays responsive while all of that happens.

For most of the web's history, those layers were built with different tools and often by different people. A backend team wrote templates in one language, a frontend team sprinkled JavaScript on top, and an API contract sat in between. That separation still makes sense for many organizations, but it has costs. Every feature crosses a boundary, every boundary needs its own types and validation, and every duplicated rule is a chance for the two sides to disagree.

React and Next.js offer a different arrangement. React gives us one model for describing user interfaces as components. Next.js extends that model to the server, so the same component tree can fetch data, render HTML on the server, stream it to the browser, and then become interactive. The seams do not disappear, but they become explicit, typed, and much easier to reason about. That is the promise this book is built around, and we will spend the next twenty-seven chapters testing it against a realistic application.

A Short History of Rendering on the Web

You do not need to memorize web history to build good applications, but a quick tour explains why today's tools look the way they do. Each generation solved the problems of the previous one and introduced new ones.

Server-rendered pages

The earliest dynamic sites rendered every page on the server. A request arrived, a program queried a database, filled in an HTML template, and sent the result back. Every click on a link or form submission triggered a full page load. This model is simple, fast to first paint, friendly to search engines, and works without JavaScript. Its weakness is interactivity: anything more dynamic than a form submission requires another round trip and a full page refresh.

Progressive enhancement with JavaScript

Next came pages that were rendered on the server and then enhanced in the browser with scripts. Libraries such as jQuery made it practical to update parts of a page without reloading it. The trouble was that the state of the interface now lived in two places: in the HTML the server produced and in whatever the scripts had changed since. Keeping those in sync by hand, with code that found elements and mutated them, became fragile as applications grew.

Single-page applications

Single-page applications (SPAs) moved rendering entirely into the browser. The server sends a nearly empty HTML document and a JavaScript bundle; the bundle builds the interface, fetches data from JSON APIs, and handles navigation without full page loads. SPAs made rich, app-like experiences normal on the web, and React became one of the most popular ways to build them. The trade-offs are real, though: users wait for JavaScript to download and run before they see meaningful content, search engines and link previews may see an empty page, and bundles tend to grow over time.

The return of the server

The current generation tries to keep the best of both worlds. Frameworks render HTML on the server for a fast first view, then "hydrate" it in the browser so it becomes interactive. More recently, React Server Components let some components run *only* on the server, so their code never ships to the browser at all. Next.js is the framework in which this model is most fully developed today, and it is the one we use from Part III onward.

Rendering Strategies at a Glance

The vocabulary of rendering can feel like alphabet soup. The table below gives a short definition of each strategy we will meet in this book. Do not worry about the details yet; each gets its own treatment in later chapters.

Strategy	Where HTML is produced	When	Good for
CSR (client-side rendering)	Browser	After JS loads	Highly interactive tools behind a login

SSR (server-side rendering)	Server	On each request	Personalized or frequently changing pages
SSG (static site generation)	Server or build machine	At build time	Content that rarely changes
Revalidation (often called ISR)	Server	At build, then refreshed	Mostly static pages with periodic updates
RSC (React Server Components)	Server	Build or request	Data-heavy UI with little interactivity

Notice that these are not mutually exclusive. A single Next.js application routinely mixes them: the Savora home page can be generated statically and refreshed every few minutes, a restaurant's detail page can be rendered on the server with fresh availability, and the booking widget inside it can run as a client component that responds instantly to every keystroke. Choosing the right strategy per page, and sometimes per component, is one of the core skills this book teaches.

> **NOTE**
>
> React Server Components are a *component model*, not a rendering mode in the same sense as SSR or SSG. A Server Component can be rendered at build time or at request time. We keep them in the table because, in practice, "does this code run on the server or in the browser?" is the question you will ask most often.

How the pieces look on a timeline

It helps to picture what the browser receives in each model. The ASCII sketch below compares a classic SPA with a server-rendered React page for the same Savora restaurant list.

```
Client-side rendering (SPA)
  Browser  ── GET / ──────────────────────────> Server
  Browser  <── empty HTML + bundle.js ─────────── Server
  Browser  [download JS] [run JS] ── GET /api/restaurants ──> API
  Browser  <── JSON ──────────────────────────────────────── API
  Browser  [render list]            first meaningful content here

Server rendering with hydration
  Browser  ── GET / ──────────────────────────> Server
                                  Server [query DB] [render HTML]
  Browser  <── full HTML (list visible) ──────── Server
  Browser  [show list]              first meaningful content here
  Browser  [download JS] [hydrate]  interactive from here
```

Server rendering moves meaningful content earlier, at the cost of work on the server for each request (or at build time, for static pages). Hydration still requires JavaScript before buttons respond, which is one reason React Server Components exist: parts of the page that never need to respond to clicks can skip hydration entirely because their code is never sent to the browser.

What React Actually Is

React describes itself as a library for building user interfaces. That modest description is accurate. React on its own does not include a router, a data layer, a build tool, or a server. What it does provide is a small set of ideas that turn out to be remarkably powerful.

Components

A React component is a function that returns a description of some UI. Components can use other components, so a page is a tree: an `App` contains a `RestaurantList`, which contains many `RestaurantCard` components, each of which contains a `RatingStars`. Because each component is just a function, you can test it, reuse it, and reason about it in isolation.

Declarative rendering

Instead of writing step-by-step instructions to change the page ("find this element, add a class, remove that child"), you describe what the UI should look like for a given state. When the state changes, React works out which parts of the real page need to change and applies the minimum set of updates. This process is called *reconciliation*. You write code that answers "what should the screen show now?" and React handles "how do we get from the old screen to the new one?"

One-way data flow

Data flows down the tree through *props*, and changes flow back up through callbacks. A parent passes a restaurant to a card; the card calls a function the parent gave it when the user taps "Save to favorites". This makes it much easier to trace why the screen looks the way it does, because every piece of data has a clear owner.

Hooks and state

Components remember things between renders using *hooks* such as `useState`. Hooks also let components synchronize with the outside world, share logic, and, since React 19, work with asynchronous Actions for forms and mutations. We introduce state properly in Chapter 4 and go deeper in Chapters 5 and 6.

Here is the smallest piece of Savora we can write. Do not type it yet; we will set up a project shortly. Just notice how it reads like a description of the output rather than a set of instructions.

src/components/RestaurantCard.tsx

```
type RestaurantCardProps = {
  name: string;
  city: string;
  cuisine: string;
};

export function RestaurantCard({ name, city, cuisine }: RestaurantCardProps) {
  return (
    <article className="restaurant-card">
```

```
      <h2>{name}</h2>
      <p>
        {cuisine} · {city}
      </p>
    </article>
  );
}
```

The angle-bracket syntax inside the function is JSX, a syntax extension that compiles to ordinary JavaScript function calls. Chapter 3 covers its rules in detail. For now, the important idea is that `RestaurantCard` is a plain function that takes data and returns UI.

Why React Needs a Framework for Full-Stack Work

Because React is deliberately small, real applications need more: routing between pages, code splitting so users do not download everything at once, a way to fetch data, a server to render HTML, image optimization, handling of environment variables and secrets, and a build pipeline for production. You can assemble all of these yourself, and for years many teams did. The React team's own documentation now recommends starting new full-stack projects with a framework rather than wiring those pieces by hand.

Next.js is one such framework, built by Vercel and developed in close collaboration with the React team. Its App Router, introduced in Next.js 13 and the default for new projects in recent versions, is built on React Server Components. That gives it several properties that matter for Savora:

- **File-system routing.** Folders under `app/` become URLs. A file at `app/restaurants/[slug]/page.tsx` handles every restaurant detail page.
- **Server Components by default.** Components can query a database directly with `await`, and their code stays on the server.
- **Server Actions.** A form can call a server function directly, which is exactly what a booking flow needs.
- **Flexible rendering.** Each route can be static, dynamic, or revalidated on a schedule or on demand.
- **Built-in optimizations.** Image, font, and script handling, metadata for SEO, and streaming with Suspense.
- **Deployment options.** It runs on Vercel, on other hosting platforms with Next.js support, or self-hosted in a Node.js server or Docker container.

Honest trade-offs

Next.js is not free of costs, and it is worth naming them up front so you can make an informed choice. The App Router introduces a mental model — which code runs where, what gets cached, and when — that takes time to learn. Caching behavior has changed between major versions, so older tutorials may describe defaults that no longer apply; we point out those differences when they matter. The

framework is large, and some advanced features ship first as experimental options before becoming stable. Finally, although Next.js can be self-hosted, a few features are easiest to use on platforms that support them directly, so you should check your hosting provider's documentation before relying on them.

Alternatives worth knowing

React has other capable frameworks. React Router, in its framework mode (the continuation of the Remix project), emphasizes web standards, loaders, and actions. TanStack Start is a newer full-stack framework built around TanStack Router. Astro focuses on content-heavy sites and lets you add React components as interactive "islands". Outside React, Vue has Nuxt and Svelte has SvelteKit, both of which solve similar problems in their own ecosystems. Everything you learn about React in Part I and Part II transfers directly to the React-based options, and much of what you learn about rendering strategies in Part III transfers to all of them.

> **TIP**
> When you evaluate a framework, build the same small feature — say, a list page with a filter and a form that saves data — in each candidate. An afternoon of hands-on comparison tells you more about fit than any feature checklist.

Meet Savora

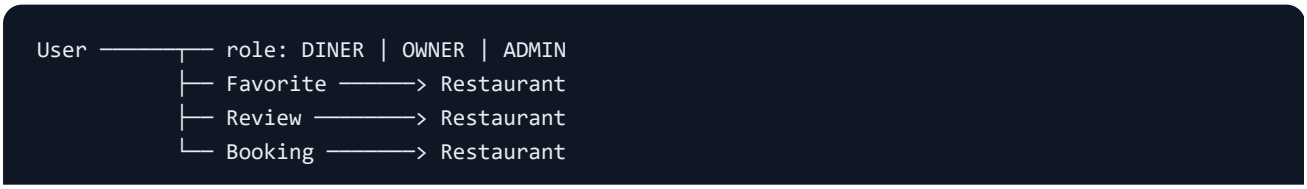
Throughout this book we build **Savora**, a fictional restaurant discovery and table-booking platform served from `savora.example.com`. It is large enough to exercise every important feature of React and Next.js, yet familiar enough that you will not need to learn a new business domain alongside the technology.

Who uses Savora

- **Diners** browse restaurants by city (Lisbon, Osaka, Austin, Melbourne) and cuisine, read menus and reviews, save favorites, and book a table for a date, time, and party size.
- **Restaurant owners** manage their profile, menu, opening hours, and availability, and they confirm or cancel bookings in an owner dashboard.
- **Admins** moderate reviews and keep the catalog healthy.

The core data model

We will refine these entities as we go, but it helps to know them now. Chapter 2 turns them into TypeScript types, and Chapter 18 turns them into a Prisma schema backed by PostgreSQL.



```
                  └── status: PENDING | CONFIRMED | CANCELLED
                              | COMPLETED | NO_SHOW

Restaurant ─┬── City
            ├── Cuisine
            ├── MenuItem (many)
            ├── OpeningHours (per weekday)
            └── TimeSlot (bookable capacity per date and time)
```

How Savora evolves

The application grows with your skills. We deliberately start without a server so you can focus on React itself, and then move the same ideas into Next.js once the fundamentals are solid.

Part	What Savora becomes
I–II	A client-side React app built with Vite and TypeScript: restaurant cards, search and filters, a booking form, favorites
III–IV	A Next.js App Router application with public pages, a booking flow using Server Actions, PostgreSQL via Prisma, authentication, and an owner dashboard
V–VI	A production-ready product: SEO, performance, accessibility, testing, deployment, payments in test mode, real-time updates, and an AI menu assistant

Why start with a Vite single-page app if the destination is Next.js? Because React concepts — components, props, state, effects, hooks — are easier to learn when there is no server in the picture. Once you understand what a component does in the browser, the question "what changes when it runs on the server?" becomes much easier to answer.

Setting Up Your Toolchain

You need four things to follow along: Node.js, a package manager, a code editor, and a modern browser. If you already have them, skim this section to check your versions and then jump to creating the project.

Node.js

Node.js runs JavaScript outside the browser. We use it for the development server, the build tools, and later for Next.js itself. Install a current **Long-Term Support (LTS)** release from `nodejs.org`. Vite and Next.js both publish minimum Node.js versions in their documentation, and those minimums rise over time, so an up-to-date LTS release is the safest choice.

If you work on several projects, a version manager makes life easier. Tools such as `nvm` (macOS and Linux), `nvm-windows`, `fnm`, or `Volta` let you install multiple Node.js versions and switch between them per project. Check your installation from a terminal:

```
node --version
npm --version
```

Package managers

Node.js ships with `npm`, which is all you need for this book. Alternatives such as `pnpm`, `yarn`, and `bun` are popular and work fine; each has its own lockfile and slightly different commands. We show `npm` commands throughout. If you prefer another tool, translate the commands and stick with one manager per project, because mixing lockfiles causes confusing dependency differences between machines.

Editor

Any editor works, but one with strong TypeScript support will save you hours. Visual Studio Code is a common choice because TypeScript's language service is built in, giving you autocompletion, inline errors, and refactoring for `.ts` and `.tsx` files out of the box. Useful extensions include ESLint, Prettier, and Tailwind CSS IntelliSense (which we use from Chapter 8). WebStorm and other JetBrains IDEs, Zed, and Neovim with a TypeScript language server are equally capable.

Browser and React Developer Tools

Use a current version of Chrome, Edge, Firefox, or Safari. Install the **React Developer Tools** browser extension from your browser's extension store. It adds a Components panel that shows the component tree, props, state, and hooks, and a Profiler panel that records renders. We use both repeatedly, especially in Chapter 7.

> **WARNING**
>
> Install browser extensions and Node.js only from official sources: your browser's extension store and `nodejs.org` (or your operating system's trusted package manager). Look-alike packages and extensions do appear from time to time, and developer machines are attractive targets because they often hold access tokens and deployment credentials.

Creating the Savora Project with Vite

Vite is a build tool and development server. In development it serves your source files over native ES modules and applies changes almost instantly with Hot Module Replacement (HMR). For production it bundles and optimizes the code. Its official templates include one for React with TypeScript, which is exactly what we want.

Open a terminal in the folder where you keep projects and run:

```
npm create vite@latest savora -- --template react-ts
cd savora
npm install
```

```
npm run dev
```

The extra `--` passes the `--template` flag through `npm` to Vite's scaffolding tool. Depending on the version, the tool may ask a follow-up question or two; accept the defaults. When the dev server starts, it prints a local URL, typically `http://localhost:5173`. Open it and you will see the starter page with a counter button. Click the button, then edit `src/App.tsx` and save: the page updates without losing the counter's value. That is HMR at work.

What the template gives you

The exact files vary slightly between template versions, but you should see something close to this:

```
savora/
├── index.html            the single HTML page Vite serves
├── package.json          scripts and dependencies
├── vite.config.ts        Vite configuration (React plugin)
├── tsconfig.json         references the two configs below
├── tsconfig.app.json     TypeScript settings for src/
├── tsconfig.node.json    TypeScript settings for config files
├── eslint.config.js      linting rules
├── public/               static files copied as-is
└── src/
    ├── main.tsx          entry point: mounts React into the page
    ├── App.tsx           root component
    ├── App.css
    ├── index.css
    └── assets/
```

Open `index.html` first. It contains an element such as `<div id="root"></div>` and a script tag that loads `/src/main.tsx`. That is the whole HTML document for a single-page app: an empty container and a script that fills it.

Now open `src/main.tsx`. In recent templates it looks like this:

src/main.tsx

```tsx
import { StrictMode } from "react";
import { createRoot } from "react-dom/client";
import "./index.css";
import App from "./App.tsx";

createRoot(document.getElementById("root")!).render(
  <StrictMode>
    <App />
  </StrictMode>,
);
```

Three things happen here. `createRoot` tells React to take control of the `#root` element. The `!` after `getElementById` is a TypeScript non-null assertion: we promise the element exists, because we can see it in `index.html`. And `render` draws the `App` component inside `StrictMode`, a development-only

helper that runs extra checks. One of those checks deliberately calls some of your functions twice in development to surface bugs caused by impure code; we explain why in Chapters 3 and 5. It has no effect in production builds.

The package.json scripts

The template defines a few scripts you will use constantly:

- `npm run dev` starts the development server with HMR.
- `npm run build` type-checks the project and produces an optimized bundle in `dist/`.
- `npm run preview` serves the built `dist/` folder locally so you can check the production build.
- `npm run lint` runs ESLint across the project.

One detail surprises many newcomers: Vite's dev server does *not* type-check your code. It strips types and serves the JavaScript, which is why it is so fast. Type errors appear in your editor and when you run `npm run build`, which calls the TypeScript compiler first. Keep an eye on the editor's Problems panel, or run the compiler in watch mode in a second terminal if you want continuous feedback.

Our First Savora Screen

Let's replace the starter counter with something that looks like Savora. We will hard-code a few restaurants for now; Chapter 9 fetches them from an API and Chapter 14 from a database. Create a `src/data` folder and add some sample data:

src/data/restaurants.ts

```
export type Restaurant = {
  id: string;
  name: string;
  city: string;
  cuisine: string;
  priceLevel: 1 | 2 | 3 | 4;
};

export const restaurants: Restaurant[] = [
  { id: "r1", name: "Casa Alma", city: "Lisbon", cuisine: "Portuguese", priceLevel: 2 },
  { id: "r2", name: "Kiji Ramen", city: "Osaka", cuisine: "Japanese", priceLevel: 1 },
  { id: "r3", name: "Juniper Smoke", city: "Austin", cuisine: "Barbecue", priceLevel: 2 },
  { id: "r4", name: "Laneway Table", city: "Melbourne", cuisine: "Modern Australian",
    priceLevel: 3 },
];
```

Next, create the card component we sketched earlier in `src/components/RestaurantCard.tsx`, and then replace the contents of `src/App.tsx`:

src/App.tsx

```
import { RestaurantCard } from "./components/RestaurantCard";
import { restaurants } from "./data/restaurants";
import "./App.css";

export default function App() {
  return (
    <main className="page">
      <header>
        <h1>Savora</h1>
        <p>Find a table you will remember.</p>
      </header>

      <section aria-label="Restaurants" className="restaurant-grid">
        {restaurants.map((r) => (
          <RestaurantCard key={r.id} name={r.name} city={r.city} cuisine={r.cuisine} />
        ))}
      </section>
    </main>
  );
}
```

Finally, give it a little structure. Replace `src/App.css` with a few rules — styling gets a full chapter later, so keep it minimal:

src/App.css

```
.page {
  max-width: 960px;
  margin: 0 auto;
  padding: 2rem 1rem;
  font-family: system-ui, sans-serif;
}

.restaurant-grid {
  display: grid;
  gap: 1rem;
  grid-template-columns: repeat(auto-fill, minmax(220px, 1fr));
}

.restaurant-card {
  border: 1px solid #ddd;
  border-radius: 12px;
  padding: 1rem;
}
```

Save the files and the browser updates to show four cards. Open React Developer Tools, select the Components panel, and click on a `RestaurantCard`. You will see its props — `name`, `city`, `cuisine` — exactly as `App` passed them.