EXPERT INSIGHT

React Application Architecture for Production

A hands-on guide to architecting, building, and delivering enterprise-ready modern React apps

Second Edition

Alan Alickovic | Anthony Alicea

‹packt›

React Application Architecture for Production

Second Edition

A hands-on guide to architecting, building, and delivering enterprise-ready modern React apps

Alan Alickovic

Anthony Alicea

‹packt›

React Application Architecture for Production

Second Edition

Portfolio Director : Ashwin Nair

Relationship Lead : Bhavya Rao

Project Manager : Vishnu Priya R

Content Engineer : Roshan Ravi Kumar

Technical Editor: Arjun Varma

Copy Editor: Roshan Ravi Kumar

Indexer: Hemangini Bari

Production Designer: Shankar Kalbhor
Growth Lead : Anamika Singh
First published: January 2023
Second edition: May 2026
Production reference: 1110526
Published by Packt Publishing Ltd.
Grosvenor House
11 St Paul's Square
Birmingham
B3 1RB, UK.
ISBN 978-1-83620-297-4
www.packtpub.com

Contributors

About the authors

Alan Alickovic is a software engineer, consultant, and creator of Bulletproof React, one of the most popular architecture guides for building React applications. He has spent more than a decade building scalable web applications across startups and large enterprises, giving him practical insights into what works in real-world scenarios. He believes great software is not just functional, but also simple, scalable, and maintainable.

Anthony Alicea is a software educator and developer with over 25 years of experience building and teaching web development fundamentals. More than 370,000 students have taken his courses on JavaScript, React, and more. Tony is known for helping developers truly understand how things work beneath the surface. He teaches React online at understandingreact.com.

About the reviewers

Gurjit Singh is a Berlin-based Senior Frontend Engineer at Storyblok with over eight years of experience building modern web applications using React, TypeScript, and Node.js. He has a keen interest in developer tooling, along with a focus on performance optimization and building scalable libraries. Previously, at Zendesk, he worked on AI-powered and customer-facing

products at scale. He also led a major open-source project for over five years, growing it to more than 30,000 monthly active users and collaborating with engineers from companies such as Apple and Wix.

Outside of work, Gurjit enjoys speaking at conferences, sharing practical engineering insights, exploring Indian classical music, reading psychology books, and traveling.

Eleke Great is a senior software engineer at Skillseed with seven years of hands-on experience working with JavaScript libraries and frameworks at the production level. He has built and scaled real-world systems and is also the author of *Saturday with Codes* , a LinkedIn newsletter with over 13,000 subscribers.

Table of Contents

Chapter 2: Setup and Project Structure Overview

Chapter 4: Routing and Rendering Strategies

Chapter 8: Improving Application Performance

Chapter 9: Going International

Chapter 11: Testing the Application

Chapter 12: Going to Production

Chapter 13: Evolving the Application

Preface

Building large-scale applications in production with React can be overwhelming due to the number of choices and lack of cohesive resources. This hands-on guide is designed to share practices and examples to help address these challenges in building enterprise-ready applications with React.

In this book, we will first discuss the architectural principles behind scalable React applications. Then, we will lay out the foundation of the project with Vite, TypeScript, ESLint, Prettier, and Husky, and organize it with a feature-based folder structure. We will then build reusable, documented components with Shadcn UI and Storybook, and learn how to handle routing and rendering strategies, including pre-rendering, SSR, CSR, and hybrid approaches using React Router in framework mode.

Once the foundations are in place, we will cover how to communicate with APIs in a type-safe way using OpenAPI-generated types, Zod validation, and React Query for server state. We will explore the right state management tools for each use case, covering local state, global state, form state, and URL state before implementing cookie-based authentication, authorization policies, and content security practices.

Finally, we will improve the quality of the application by optimizing performance with memoization, code splitting, and streaming, adding internationalization with react-i18next, ensuring accessibility by following WCAG principles, and writing a comprehensive test suite with Vitest and Playwright. We will finish by setting up a CI/CD pipeline with GitHub Actions and looking at advanced topics such as enforcing the architecture with AI, React Server Components, feature flags, monorepos, and microfrontends.

By the end of the book, you will be able to efficiently build production-ready applications by following industry practices and expert tips.

Who this book is for

This book is for intermediate-level web developers who already have a good understanding of JavaScript, React, and web development in general and want to build large-scale React applications effectively. Besides experience with JavaScript and React, some experience with TypeScript will be beneficial.

What this book covers

Chapter 1 , Understanding the Architecture of React Applications , explores how to think about applications from an architectural point of view. It starts by covering the importance of good architecture and its benefits. Then, it covers some bad and good practices in React applications. Finally, we will cover the planning of a real React application, an AI Ideas Community Platform, that we will be building throughout the book.

Chapter 2 , Setup and Project Structure Overview , covers laying out the foundation of the project with all the tools and setup for the application that we will be building. It will introduce us to tools such as React Router, Vite, TypeScript, ESLint, Prettier, and Husky. Finally, it will cover the feature-based project structure for the project, which improves the codebase organization.

Chapter 3 , Building and Documenting Components , introduces us to component design principles and Shadcn UI, a copy-paste component library built on top of Radix UI primitives. We will cover how to set it up and use it to build reusable components that can be used throughout the application to keep the UI consistent. Finally, we will learn about documenting those components with Storybook.

Chapter 4 , Routing and Rendering Strategies , dives into React Router in framework mode and different rendering strategies in more depth. First, we will cover the basics such as routing, nested layouts, and data prefetching with loaders. Then, we will explore the different rendering strategies it supports: pre-rendering, SSR, CSR, and hybrid. Finally, we will apply those concepts by building the routes and layouts for our application.

Chapter 5 , Communicating with the API , walks through how to communicate with the backend API in a type-safe way. We will first learn how to generate TypeScript types from an OpenAPI specification and validate API responses at runtime. Then, we will configure React Query and use it to build the API layer for our application, covering queries, mutations, and cache invalidation.

Chapter 6 , Managing Application State , teaches how to use the right state management tool for each use case. We will start with local UI state, then move to global state with Zustand. From there, we will look at form state with

React Hook Form and Zod, and finish with URL state management for features like filters and search parameters.

Chapter 7 , Implementing Authentication and Securing the Application , starts by walking through how to implement authentication for our application using cookie-based sessions. Then, it demonstrates how to protect routes and enforce authorization policies for resource ownership. Finally, it covers security best practices such as content sanitization and security headers.

Chapter 8 , Improving Application Performance , focuses on performance optimization in a React application. It starts by covering how to detect and diagnose performance bottlenecks using the React DevTools Profiler. Then, it covers a range of optimization techniques including memoization, code splitting with lazy loading, server-side streaming, debouncing, infinite scroll, and optimistic UI updates.

Chapter 9 , Going International , guides you through setting up internationalization for a React application. We will first cover how to configure react-i18next and organize translations by feature namespace. Then, we will go through key concepts such as server-side language detection, pluralization, variable interpolation, and locale-aware date formatting. Finally, we will build a language switcher component that stores the user's preference in a cookie, which persists across page reloads.

Chapter 10 , Making the Application Accessible , examines the practices for making your application accessible following WCAG standards. It starts with the POUR principles as a framework for thinking about accessibility. Then, it covers practical techniques such as semantic HTML, skip links, ARIA attributes, live regions for dynamic content announcements, and visible focus styles for keyboard navigation.

Chapter 11 , Testing the Application , takes a practical approach to testing a React application using the testing trophy strategy. We will cover unit and component testing with Vitest and React Testing Library, focusing on complex isolated logic and UI behavior. Then, we will use Playwright for integration and end-to-end tests, covering route mocking and structuring tests with test steps.

Chapter 12 , Going to Production , covers the basics of setting up a CI/CD pipeline with GitHub Actions. We will first configure the CI pipeline with

parallel jobs for linting, type checking, format checking, and all test tiers. Then, we will set up the CD pipeline to trigger on a successful CI run on the main branch and deploy the application to Render.

Chapter 13 , Evolving the Application , touches on some advanced topics for taking the application beyond its current state. We will look at using AI to enforce architectural standards, React Server Components, application observability, feature flags, the backend for frontend pattern, and how to scale the codebase with monorepos and microfrontends.

To get the most out of this book

Previous experience with JavaScript and React and fundamental knowledge of web development will make it a lot easier to follow along with the content of the book. It is also desirable to have some experience with TypeScript and React Router, but it should be possible to follow along without it since we will cover the basics in the book.

Download the example code files

This book includes a complete downloadable code bundle containing all the example projects and files used throughout the chapters. We recommend downloading the bundle so you can follow along smoothly and experiment with the examples.

Use the bundle as a practical starting point. Modify it, extend it, and apply what you learn by creating your own variations as you progress through the chapters.

Get the code bundle

If you bought the book directly from Packt:

1. Go to **packtpub.com**
2. Click your **profile picture** and select **Your Orders**
3. Find this book and click **Download Code**

If you bought this book from Amazon or any other channel partner:

1. Go to `packtpub.com/unlock` or scan the following QR code:

2. Search for this book
3. Sign up or log in to your free Packt account
4. Upload your proof of purchase and download the code bundle locally

Usage note: You're free to use and modify this code for personal learning and non-commercial projects.

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText` : Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. For example: " We could use this same `Counter` component multiple times on a page

A block of code is set as follows:

```
export default function HomePage() {
  return (
    <div>
      <h1>Home</h1>
      <Counter
        initialValue={0}
        label="Click Counter"
        onIncrement={(newValue) => {
          console.log(`Incremented to ${newValue}`);
        }}
        onDecrement={(newValue) => {
          console.log(`Decremented to ${newValue}`);
        }}
      />
    </div>
  );
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
export default function HomePage() {
  return (
    <div>
      <h1>Home</h1>
      <Counter
        initialValue={0}
        label="Click Counter"
        onIncrement={(newValue) => {
          console.log(`Incremented to ${newValue}`);
        }}
        onDecrement={(newValue) => {
          console.log(`Decremented to ${newValue}`);
        }}
```

```
      />
    </div>
  );
}
```

Any command-line input or output is written as follows:

```
npm run dev
```

Bold : Indicates a new term, an important word, or words that you see on the screen. For instance, words in menus or dialog boxes appear in the text like this. For example: "This is an **MVP** (**Minimum Viable Product**).

> **NOTE**
> Warnings or important notes appear like this.

> **TIP**
> Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback : If you have questions about any aspect of this book or have any general feedback, please email us at `customercare@packt.com` and mention the book's title in the subject of your message.

Errata : Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you reported this to us. Please visit `http://www.packt.com/submit-errata` , click **Submit Errata** , and fill in the form.

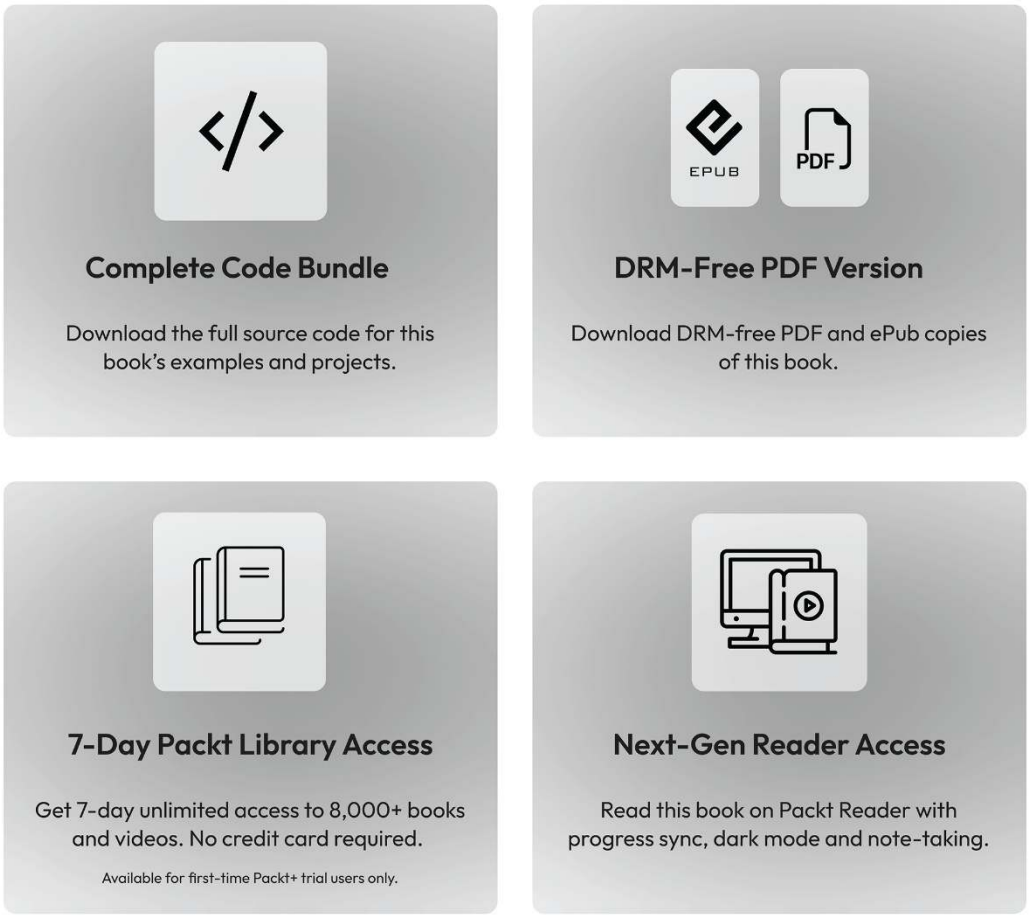
Piracy : If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author : If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit .

Free benefits with your book

This book includes free benefits designed to support your learning and help you apply what you learn effectively. Activate them now for instant access (see the *How to unlock* section for instructions).

Here's a quick overview of what you can instantly unlock with your purchase:



How to unlock

Scan the QR code (or go to `packtpub.com/unlock`). Search for this book by name, confirm the edition, and then follow the steps on the page.

UNLOCK NOW

Note: Have your invoice handy. Purchases made directly from the Packt website don ' t require an invoice.

Share your thoughts

Once you've read *React Application Architecture for Production* , we'd love to hear your thoughts! Scan the QR code below to go straight to the Amazon review page for this book and share your feedback.

https://packt.link/r/1836202970

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

1

Understanding the Architecture of React Applications

React is the most popular JavaScript library in the world for building user interfaces. The job market is full of React-based opportunities.

React has become so popular for multiple reasons: its small API, its massive ecosystem of pre-built components, and its flexibility. Yet for the modern developer, that flexibility can also be a double-edged sword. This is because React doesn't provide a strong opinion on how to structure your application, so every team has to figure it out on their own. That leads to inconsistency across projects, architectural decisions made under deadline pressure without fully understanding the tradeoffs, and codebases that become increasingly difficult to maintain as they grow.

This means every team is faced with important questions: How do you structure your application? How does state flow through it? How do you handle routing, component styles, and every other architectural decision a developer must make? React does not provide a definitive answer.

Yet, good architectural decisions are the difference between building a maintainable app and creating a mess of hard-to-debug spaghetti code.

This book aims to provide you with that missing clarity: how to architect a React application so that it is scalable, maintainable, and can grow with you and your team.

In this chapter, we'll cover the following:

- Discuss the benefits of having a good application architecture
- Explore the architectural challenges of React applications

- Understand the architectural decisions we face when building React applications
- Plan the application we'll build in this book

By the end of this chapter, you'll understand how to approach React applications from an architectural perspective and make informed decisions that set your projects up for long-term success.

NOTE

Your purchase includes a free PDF copy + exclusive extras

Your purchase includes a DRM-free PDF copy of this book, the code bundle, and additional exclusive extras. See the *Free benefits with your book* section in the *Preface* to unlock them instantly and maximize your learning.

Why architecture matters

Every application has an architecture, even those where nobody thought about it. You might open a React project and find 47 components in a single `components/` folder, API calls scattered across random files, and state managed three different ways depending on who wrote the code.

That's architecture. It just wasn't a deliberate one. The question isn't whether your application has architecture, but whether that architecture was chosen deliberately or happened by accident.

When we talk about architecture, we're talking about the high-level decisions that shape how your application works. How is the code organized? How do different parts communicate? What patterns does the team follow? These decisions are difficult to change later because they affect everything built on top of them.

In this chapter, we'll explore why architecture matters in the first place, look at the specific architectural challenges React presents, examine patterns that work and patterns that don't, and then plan the real application we'll build throughout the rest of the book.

Architecture as foundation

When you construct a building, you start with the foundation. You don't start with the walls and figure out the foundation later. Software works the same way, but the consequences aren't as immediate. You can start building features without thinking about architecture, and things will work fine for a while. The problems show up later, when you need to change something.

A solid architectural foundation makes your application resilient to change. And change is the only constant in software. Requirements shift. New features get added. Team members come and go. New technologies appear. Business priorities pivot. An application built on a solid foundation can adapt to these changes. One built on a shaky foundation requires increasingly expensive workarounds until eventually the only option is the most expensive of all: a rewrite.

Coordination and velocity

Good architecture makes it easier to work as a team. When components have clear boundaries and responsibilities, developers can work on different parts of the application without stepping on each other's toes. You can split work across team members more effectively. You can make better estimates because you understand the scope of changes.

This matters even more as teams grow. A solo developer can keep an entire poorly architected application in their head, but that doesn't scale. When a second developer joins, they need to understand the patterns. When a third joins, they need those patterns to be consistent. By the time you have five developers, inconsistent architecture creates constant friction.

Bad architecture also affects AI-assisted development. When you use tools like GitHub Copilot or Claude, they predict based not just on their training data but on your codebase patterns. Consistent architecture means consistent suggestions. Inconsistent architecture means the AI can't figure out which pattern to follow, and you spend more time correcting its suggestions than you save from using it.

Speed through clarity

Counterintuitively, spending time on architecture upfront actually increases development speed. When architectural decisions are made, developers can focus on solving business problems instead of debating technical approaches

for every feature. New features fit into existing patterns. New developers can become productive quickly because the structure is clear.

The alternative is what we might call " *architecture decision fatigue* ." Every new feature requires deciding how it should work, where it should live, and how it should communicate with other parts of the application. These micro-decisions add up. They create inconsistency. They slow everything down.

This is particularly important when working with LLMs for code generation. If you ask an AI to add a feature to a well-architected application, it can follow existing patterns. If you ask it to add a feature to an inconsistent codebase, it has to guess which pattern to follow, and those guesses are often wrong.

Economics of good architecture

Software projects cost money. Most of that cost is people: their time, their effort, their attention. Good architecture reduces costs by making those people more effective. It reduces the time spent debugging. It reduces the time spent searching for where things are. It reduces the time spent working around architectural limitations.

Poor architecture has a compounding cost. Every new feature takes longer because you have to work around existing problems. Every bug is harder to fix because the boundaries between components are unclear. Eventually, you reach a point where adding new features is prohibitively expensive and a rewrite becomes necessary. Rewrites are expensive and risky.

Good architecture also helps you make better business decisions. When you understand your application's structure, you can estimate costs more accurately. You can decide which features to build and which to defer until later.

Architecture and product quality

When developers spend less time fighting the architecture, they have more time to focus on user needs. They can spend time on polish. They can think about edge cases. They can improve error handling. They can make the application more accessible.

Poor architecture creates a vicious cycle. Developers spend time fixing architectural problems. Features ship with rough edges. Users encounter bugs.

Developers spend more time on emergency fixes. There is never time to improve the architecture because there is always something urgent.

Good architecture creates a happy cycle of improving quality. Developers spend time building features. Features ship with better quality. Users are happier. Fewer bugs come back. The team has time to keep the architecture healthy.

The goal of any software is to solve real problems for real people. Everything else is in service to that goal. Architecture matters because it determines whether you can stay focused on that goal or get lost in technical debt.

The React architecture challenge

React is an excellent tool for building user interfaces, but it intentionally doesn't solve certain problems. Understanding what React does and doesn't do helps us understand the architectural decisions we need to make.

React's flexibility is both a blessing and a curse. It's a blessing because the library doesn't force you into patterns that don't fit your problem. It's a curse because you're responsible for every decision. This has helped React's popularity, creating a rich ecosystem of libraries and tools which you can think of as pre-made decisions. This, however, creates a new problem: choice overload.

Let's look at the landscape. The React ecosystem is vast. Take a moment to examine this roadmap from roadmap.sh:

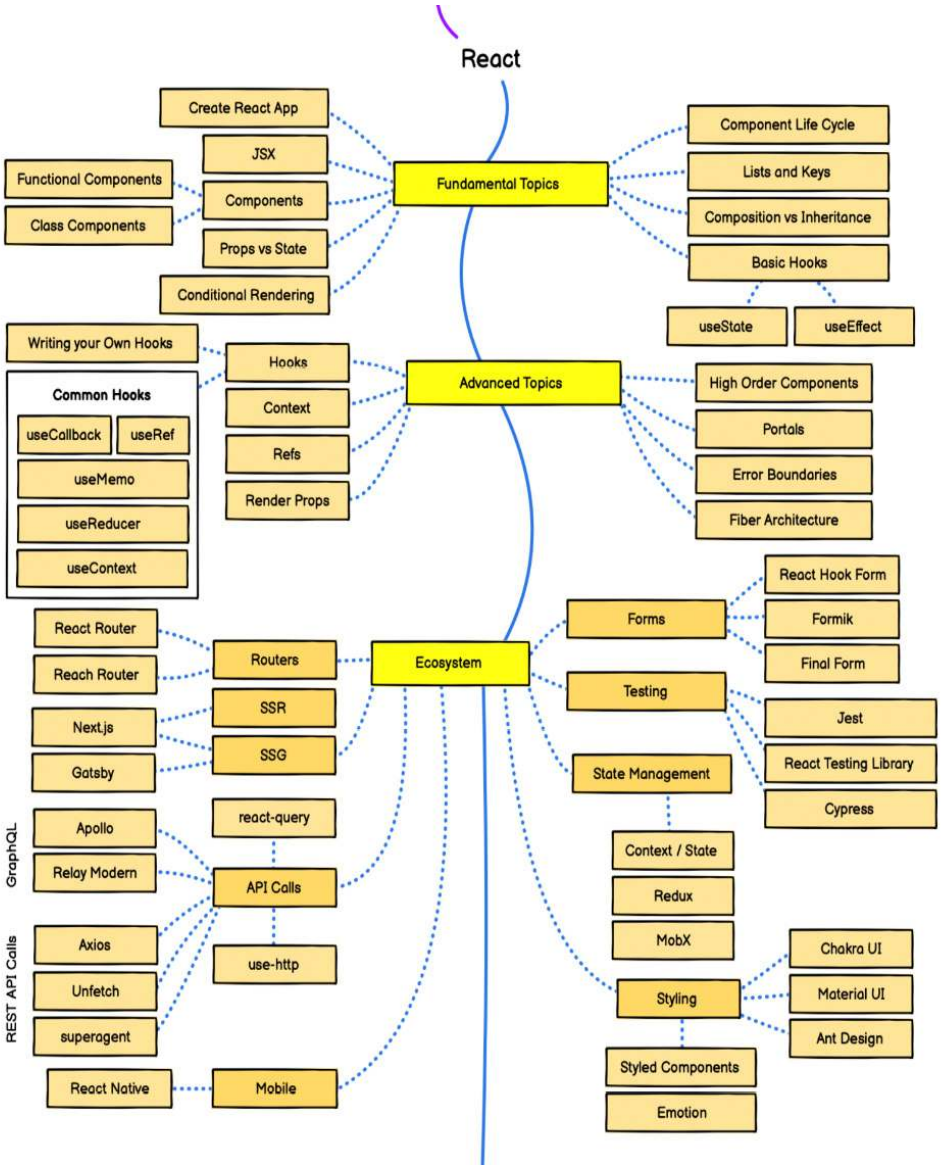


Figure 1.1 – React developer roadmap by roadmap.sh

This diagram shows a fraction of what's available. Each box represents a decision point. Each decision has multiple options. Each option has trade-offs.

Here are the questions that every development team starting a new React project faces.

How do we structure the project?

React doesn't have an opinion about file structure. You could:

- Put everything in one file (though you shouldn't)

- Have a flat folder of components
- Organize by technical role (all components in one folder, all hooks in another)
- Organize by feature
- Organize by domain concept

Dan Abramov, once one of React's core team members, famously said: " *Move files around until it feels right* ." This is helpful, but for some it can be frustrating. It's helpful because it acknowledges that different applications have different needs. A social network and a text editor shouldn't necessarily be organized the same way. It's frustrating because it doesn't give you a starting point.

The structure matters more than you might think. A good structure makes it easy to find things. A poor structure means you spend cognitive energy just navigating the codebase. When you're training an LLM on your codebase or using AI pair programming tools, structure becomes part of the context window. Clear structure helps the AI understand your application.

How do we render our application?

The term " *rendering* " itself has multiple meanings in React, which creates confusion. When we talk about rendering strategy in React, we mean: where and when does our code execute to determine what the UI should look like?

The simplest option is **client-side rendering (CSR)** : all your React code runs in the browser. The server sends a minimal HTML file, JavaScript downloads and executes, and React builds the UI. This is perfect for applications behind authentication or for internal tools.

But what if you want to make it as easy as possible for search engines to crawl your application? Then you might need **server-side rendering (SSR)** : your React code executes on the server to generate HTML, which is sent to the browser. The browser displays it immediately (good for perceived performance), then downloads the JavaScript and "hydrates" the page to make it interactive.

Or perhaps your content doesn't change often, so you can use **static site generation (SSG)** , pre-render HTML at build time, and serve it from a CDN.

Or perhaps you need all three for different parts of your application. React and its various frameworks support all of these, but you need to decide which to

use and where.

React Server Components add another dimension to this question. Some components can run only on the server, never sending their code to the client. This can sometimes help bundle size but creates new rules about what those components can and can't do, and may send more data in the HTML than is saved in the bundle.

How do we manage state?

React provides **useState** and **useReducer** for local state, and the Context API for sharing state across components. For many applications, this is enough. For many others, it is not.

The challenge is that "state" is not one thing. There's local UI state (is this dropdown open?), form state (what has the user typed?), global application state (who is the current user?), server state (what data did we fetch from the API?), and URL state (what page are we on, what are the search parameters?).

Each of these has different characteristics and different requirements. Local UI state is usually simple. Server state is complex because it involves caching, invalidation, background updates, and error handling. Treating all state the same leads to problems.

The ecosystem offers many solutions: Redux, MobX, Zustand, Jotai, Recoil, XState, and more. Then there are specialized tools for server state like TanStack Query (React Query) and SWR. Each has different philosophies and trade-offs.

How do we style components?

Some developers prefer traditional CSS files. Others prefer CSS Modules for scoping. Others prefer utility-first CSS like Tailwind. Others prefer CSS-in-JS solutions, and more.

The choice affects more than just developer preference. It affects performance (build-time vs. runtime), how you think about component reusability, how easily you can maintain consistent design, and your bundle size.

How do we handle data fetching?

React doesn't have built-in data fetching. Instead, you have several options:

- You could use the browser's fetch API directly
- You could use a library like Axios (`https://axios-http.com/`)
- You could use a specialized tool like TanStack Query (`https://tanstack.com/query/latest`) that handles caching and background updates
- You could use the data fetching capabilities of a meta-framework like Next.js (`https://nextjs.org/`)

Where you fetch matters too. In the component? In a custom hook? In a separate data layer? On the server? Each approach has implications for how you test your code and how you handle loading and error states.

How do we handle authentication?

Authentication intersects with several other decisions. Are you using token-based auth (where tokens are stored in `localStorage` or `sessionStorage`)? Or cookie-based auth (where `httpOnly` cookies are automatically included with requests)? Cookie-based auth is more secure but requires server-side session handling. Token-based auth is simpler but vulnerable to XSS attacks if not handled carefully.

How do you protect routes? How do you handle authentication errors? How do you handle token refresh?

These patterns need to be consistent across your application.

How do we test a React application?

React applications can be tested at multiple levels: tests for individual functions, tests for components, end-to-end tests for full user flows, and regression tests for UI changes.

But time is not an infinite resource. Which tests give you the most confidence with the least maintenance burden? Do you test implementation details or user behavior? How do you test components that fetch data? How do you test authenticated flows?

The testing strategy affects your architecture because different architectural patterns are easier or more difficult to test.

The meta-decision

Underlying all of these is a meta-decision: do you use a meta-framework like Next.js, React Router, or TanStack Start? These frameworks make some of the above decisions for you, but they also lock you into their patterns. That's not necessarily bad. Constraints can free you to think less about architecture and more about the app itself. But it's a choice that affects everything else.

Thinking about architectural decisions

Not all architectural decisions are equal. Some are clearly wrong in almost any context. Some are clearly right. Most fall somewhere in between, where the right answer depends on your specific situation. Let's look at some examples.

Patterns that usually don't work

Some architectural patterns appear reasonable at first but cause problems as applications grow. Here are some seen in React codebases and why they break down:

- **Everything in One Place** : Putting all your components in a single flat folder works fine when you have 10 components. By 50 components, finding anything becomes a chore. By 100 components, it's a disaster. The same goes for putting your entire application in a single file. The fact that you can do something doesn't mean you should.
  Why doesn't this work? Because humans are bad at dealing with long, undifferentiated lists. We need structure to make sense of things. Your file system is part of your application's documentation. It should communicate structure, not hide it.
- **One Big Component** : A component that does everything seems efficient at first. You have all the logic in one place. But these components become unmaintainable. They're hard to test because you have to set up the entire world that surrounds the component. They're hard to reuse because they're coupled to too many things. They're hard to reason about because you have to hold too much in your head at once.
- **Large components** : When you're using AI to modify code, large components create problems. The component might fill much of the AI's context window, or the AI might suggest changes that don't account for subtle interactions in the component.

- **Everything global** : Putting all your state in a global store seems like it would make things simple. Everything in one place! But this actually makes things harder. When any component can modify any state, understanding data flow becomes impossible. You can't look at a component and understand what state it depends on. Changes become risky because you don't know what else might break. Global state has its place, but most state should be local. Keep state as close as possible to where it's used, and lift it only when you need to share it.
- **Wrong Tool, Right Problem** : The React ecosystem has specialized tools for specialized problems. Using a generic global state manager to handle server data (data fetched from an API) means reimplementing caching, revalidation, optimistic updates, and background sync. These are complex problems. Tools like TanStack Query solve them, and they're proven at scale. Ignoring specialized tools to avoid adding dependencies sounds pragmatic, but you end up with more code, more bugs, and more maintenance burden. A carefully chosen dependency is the solution, not the problem.
- **Ignoring Security** : Not sanitizing user input is not an architectural decision, it's a mistake. If your application displays user-generated content (and we'll be building one that does), you must sanitize it. Otherwise, you're vulnerable to attacks where malicious users inject scripts that execute in other users' browsers. This is especially important if you're using Markdown or rich text.
- **Ignoring Performance Infrastructure** : Serving your application from a single server in one geographic location means users far from that server experience slow load times. This is a solved problem: content delivery networks (CDNs) distribute your application globally. Users are served from the edge location closest to them. This is table stakes for modern web applications.

Patterns that usually work

On the other hand, certain patterns have proven themselves across teams and codebases of all sizes. These aren't rules; they're decisions that tend to serve you well unless you have a good reason to deviate:

- **Feature-Based Structure** : Organizing code by feature rather than by technical role scales better. Instead of a folder for all components, a folder for all hooks, and a folder for all API calls, you have a folder for user authentication, a folder for the ideas feature, a folder for reviews. Each folder contains everything related to that feature, and teams can work within separate features with fewer merge conflicts. Why does this work? Because features are how we think about applications. When you need to modify how ideas work, you go to the ideas folder. Everything you need is co-located. When a new developer joins, they can understand the application by understanding its features. This also helps with AI-assisted development. When an LLM is helping you modify a feature, all the relevant context is in one place, making it easier for the AI to understand the full scope of what it's working with.
- **Co-located State** : Keeping state as local as possible. If only one component needs it, use useState in that component. If several nearby components need it, lift it to their common parent. Only put state in a global store if it truly needs to be global (user authentication, theme preferences, etc.). This keeps the scope of state clear. When you look at a component, you can see what state it depends on. Changes are less risky because the impact is localized.
- **Appropriate Specialization** : Use specialized tools for specialized problems. Use TanStack Query for server state. Use React Hook Form for complex forms. Use Zod for validation. These tools exist because they solve hard problems well. Your application-specific code should focus on application-specific problems, not on reimplementing generic solutions.
- **Layered Defense** : Security is not a single decision; it's a series of decisions. Use httpOnly cookies for authentication tokens so they can't be accessed by JavaScript. Validate input on both client and server. Sanitize user-generated content before rendering. Use TypeScript to catch type errors at compile time. Use ESLint to catch common mistakes. Each layer catches things the others miss.
- **Small, Focused Components** : Components should do one thing. If a component is getting large, that's a signal to break it apart. Small components are easier to understand, test, modify, and reuse. They are

also easier for AI to work with, since they fit more easily in context windows and their behavior is clearer.

- **Tooling and Analysis** : Use tools to enforce consistency, like ESLint for code quality, Prettier for formatting, and TypeScript for type safety. These tools catch mistakes before code runs. They reduce cognitive load because you don't have to remember rules. They create consistency across the team because the tools enforce it. This is particularly valuable when using AI code generation tools. When the AI suggestions are automatically formatted and type-checked, you catch issues immediately.
  These patterns and anti-patterns give us a menu of architectural possibilities. But patterns in the abstract only get you so far. To see how these ideas play out in practice, we need a real application with real requirements. That's what we'll plan next.

Planning our application

Abstract discussions of architecture are useful, but architecture decisions happen in the context of real applications with real requirements. So, we're going to build one.

Throughout this book, we'll build an **AI Ideas Community Platform** . This is a social application where users can share AI application ideas, browse ideas from others, and write reviews. Think of it as a community hub for people who want to share what they're building or thinking about building in the AI space.

Why this application? Because it touches on most of the architectural decisions we've been discussing. It has public pages (for SEO). It has authenticated pages (for posting ideas and reviews). It has user-generated content (which means we need sanitization). It has forms (ideas and reviews). It has list views with filtering and search. It has relational data (users, ideas, reviews). It is conceptually straightforward enough that we can focus on architecture without getting lost in domain complexity.

This is an **MVP** (**Minimum Viable Product**). We'll build core features, but architect it so it can grow. Future enhancements could include collaboration on ideas, voting, notifications, AI-powered recommendations, analytics, and

more. Good architecture means those features can be added without restructuring everything.

Understanding the requirements

Before making architectural decisions, we need to understand what we're building. Requirements come in two types:

Functional requirements: What it does

These describe the features from a user's perspective. Defining functional requirements clearly before writing code serves two purposes. First, they give us concrete requirements to make architectural decisions against; we're not choosing tools in the abstract, we're choosing tools that fit what we need to build.

Second, they'll become the acceptance criteria to know when a feature is done. As you read through these, think about which of the architectural challenges we discussed earlier each requirement touches.

- **User authentication and management :** The foundation of a community platform is identity. Users need to be able to create accounts, sign in, and manage their presence:
    - **User registration** : New users can create accounts with username, email, and password
    - **User login/logout** : Registered users can authenticate and manage their sessions
    - **User profiles** : Users can view and edit their profiles, including bio information that supports Markdown formatting
    - **User dashboard** : Authenticated users get a personalized dashboard to manage their content
- **Idea management** : The core of the platform is ideas themselves. Users need full control over creating, browsing, and managing AI application ideas:
    - **Create Ideas** : Logged-in users can submit new AI application ideas with a title, short description, detailed description (with Markdown support), and tags

    - **Browse Ideas** : All visitors can browse ideas in a paginated list sorted by creation date
    - **View Idea Details** : Detailed view showing full idea information, including all associated reviews
    - **Edit Ideas** : Users can edit their own ideas to improve and update content
    - **Delete Ideas** : Users can delete their own ideas (with cascade deletion of associated reviews)
    - **Search and Filter** : Users can search ideas by title or description and filter by tags
- **Review System** : Community value comes from feedback. The review system lets users evaluate and discuss each other's ideas through the review system such as:
    - **Write Reviews** : Logged-in users can write reviews for ideas with a 1–5-star rating and text content
    - **View Reviews** : All visitors can see reviews on idea detail pages, sorted by date
    - **Edit Reviews** : Users can update their own reviews
    - **Delete Reviews** : Users can delete their own reviews
    - **Review Constraints** : Users can only write one review per idea
- **Content Discovery** : A platform is only useful if users can find what they're looking for. Discovery features help surface relevant content:
    - **Idea Listing** : Paginated or infinite scroll listing of all ideas
    - **Tag-based filtering** : Filter ideas by selecting tags
    - **Real-time search** : Search functionality with debounced updates
    - **User Content Views** : Users can view all their own ideas and reviews in dedicated pages

Non-functional requirements: How it works

These describe quality attributes and constraints, the characteristics that determine whether the application feels professional and reliable, even when nothing is visibly "broken." Non-functional requirements are easy to overlook

but are important to the success of an application. Each category below defines a standard we'll hold ourselves to throughout the book:

- **Performance :** The application must meet modern web performance standards:
  - Initial page load within 3 seconds on standard broadband
  - Client-side navigation should feel instant (< 100ms)
  - Bundle size under 1MB for initial page load
  - Meet Core Web Vitals thresholds (LCP < 2.5s, FID < 100ms, CLS < 0.1)
- **Usability :** The application must be user-friendly and intuitive:
  - Responsive design that works on devices from 320px to 1920px width
  - Touch-friendly interface for mobile devices
  - Consistent navigation across all routes
  - Clear loading states and skeleton screens during data fetching
  - Intuitive form validation with helpful error messages
- **Accessibility:** The application should be accessible to all users:
  - Meet WCAG 2.1 AA accessibility standards
  - Keyboard navigation support for all interactive elements
  - Proper ARIA labels and semantic HTML
  - Minimum color contrast ratio of 4.5:1 for text
- **Security:** Client-side security measures must be implemented:
  - Input validation and sanitization to prevent XSS attacks
  - Markdown content sanitization before rendering
  - Secure authentication token storage (httpOnly cookies)
  - Protection against common web vulnerabilities
- **Reliability:** The application should handle failures gracefully:
  - Network error handling with retry mechanisms
  - React error boundaries for component error handling
  - Graceful degradation when features are unavailable
  - Appropriate fallback content for failed data fetching

The data model

Our application has three core entities that work together to create the community platform:

- **User**
  - `id` : Unique identifier
  - `username` : Unique username for the user
  - `email` : User's email address
  - `password` : Hashed password for authentication
  - `bio` : User biography supporting Markdown (optional)
  - `createdAt` : Timestamp of account creation
- **Idea**
  - `id` : Unique identifier
  - `title` : The idea's title
  - `shortDescription` : Brief summary of the idea
  - `description` : Detailed description with Markdown support
  - `tags` : Array of tags for categorization
  - `authorId` : Reference to the user who created it
  - `createdAt` : Timestamp of creation
  - `updatedAt` : Timestamp of last update
- **Review**
  - `id` : Unique identifier
  - `content` : Review text content
  - `rating` : Star rating (1–5)
  - `ideaId` : Reference to the idea being reviewed
  - `authorId` : Reference to the User who wrote the review
  - `createdAt` : Timestamp of creation
  - `updatedAt` : Timestamp of last update

The relationships between these entities are as follows:

- A User can create many ideas (one-to-many)
- A User can write many Reviews (one-to-many)

- An Idea can have many Reviews (one-to-many)
- Each Review belongs to one Idea and one User
- A User can only write one Review per Idea (enforced constraint)

This is a clean relational model. It supports everything the MVP needs while staying simple enough to understand quickly.

Architectural decisions

Now we can make informed decisions. We've defined what the application does (functional requirements), how well it needs to do it (non-functional requirements), and what data it works with (the data model). Each architectural decision below is a direct response to these requirements.

We're not picking tools because they're popular; we're picking them because they solve the specific problems our application presents. For each decision, we'll name the approach and explain why it fits our requirements.

Project structure: feature-based

We'll organize by domain features (users, ideas, reviews) rather than by technical role. Each feature folder contains its components, API calls, hooks, types, and utilities. Shared code (auth, UI components, utilities) lives in shared folders.

This scales as the app grows. Features can be developed independently. Related code is co-located. New developers can understand the structure by understanding the features.

Rendering strategy: hybrid

Different pages have different needs. We'll use React Router v7, which supports the following:

- **Static Site Generation (SSG)** for the landing page: pre-rendered at build time, instant loading, perfect for marketing
- **Server-Side Rendering (SSR)** for public idea pages: rendered on-demand on the server, good for SEO and initial load performance
- **Client-Side Rendering (CSR)** for authenticated pages: rich interactivity, fast navigation after initial load

We'll also use code splitting to lazy load routes, prefetching for critical resources, TanStack Query for caching API responses, and skeleton screens for

loading states.

React Router v7 makes this straightforward. We can use a hybrid rendering approach, which means each route can specify its own rendering method.

State management: Multi-tier

Different types of state need different tools:

- Local state (component-specific): useState and useReducer
- Global application state (in the context of our application, it's just notifications system): Zustand
- Server state (API data): TanStack Query
- Form state: React Hook Form with Zod validation
- URL state (search, filters, pagination): Nuqs library to handle URL state - `https://nuqs.dev/`

This avoids over-engineering. Each tool is purpose-built for its use case.

Styling: Tailwind CSS + BaseUI

We'll use Tailwind CSS for utility-first styling. It's fast (build-time, not runtime), it enforces consistency through design tokens, and its purge feature keeps bundles small. Developer experience is excellent once you learn the utilities.

For component primitives (dropdowns, dialogs, etc.), we'll use Base UI. Base UI provides unstyled, accessible components. It handles keyboard navigation, focus management, and ARIA attributes. We style them with Tailwind.

This gives us full design control with excellent accessibility built in.

Authentication: Cookie-based

We'll use httpOnly cookies for authentication. The cookie is set by the server and automatically included with requests. JavaScript can't access it, which protects against XSS attacks.

On the frontend, Zustand will track the current user. Protected routes will redirect unauthenticated users to login. We'll handle auth errors gracefully and persist sessions across page refreshes by checking with the server.

Testing: pragmatic

We'll follow the testing trophy: mostly integration tests, some end-to-end tests for critical paths, and unit tests for complex logic.

- **Unit tests (Vitest)** : Utility functions, custom hooks, calculations
- **Integration tests (Playwright + Mocked API)** : Testing complete pages from user's perspective
- **End-to-end tests (Playwright)** : Critical journeys (register, login, create idea, write review)

We won't over-test presentational components or third-party integrations. The goal is confidence, not coverage percentage.

Summary

We've covered a lot of ground. We talked about why architecture matters, not just in theory but in practice. We explored the architectural landscape of React applications and the decisions you'll face. We looked at patterns that work and patterns that don't. And we planned a real application that we'll build together.

The decisions we made aren't arbitrary. They're driven by our requirements. A different application with different requirements might require different choices. That's the point. Architecture isn't about following rules. It's about understanding tradeoffs and making informed decisions.

In the next chapter, we'll set up our development environment and tooling. We'll install and configure everything we need to build this application. Then we'll start building one piece at a time, seeing how these architectural decisions play out in practice.

The goal isn't just to build an application. It's to understand *why* we build it the way we do. That understanding is what lets you make good architectural decisions in your own projects, with your own requirements, in your own context.

Ready? We're not going to just build a React application. Let's *architect* one.

Get this book's PDF copy, code bundle, and more

Scan the QR code (or go to `packtpub.com/unlock`). Search for this book by name, confirm the edition, and then follow the steps on the page.

UNLOCK NOW

Note: Have your invoice handy. Purchases made directly from the Packt website don ' t require an invoice.

2

Setup and Project Structure Overview

Before writing any feature code, it's worth getting our project set up right. Poor tooling choices and messy folder structures are some of the most common reasons code bases become hard to work with over time. In this chapter, we'll set up the tools and structure that prevent those problems: a meta framework, type checking, linting, formatting, pre-commit checks, and a feature-based project structure. By the end, we'll have a solid foundation that we can build on with confidence.

We'll cover the following topics:

- Choosing a meta framework for the project
- Build tool setup overview
- Type-checking setup overview
- Linting setup overview
- Formatting setup overview
- Pre-commit checks setup overview
- Project structure overview
- Environment variables setup overview

By the end of this chapter, you'll have a good understanding of the tools we'll be using for the project setup and the feature-based project structure to make organizing our code more manageable.

Technical requirements

Before we get started, we need to set up our project. To develop our project, we need the following tools installed on our computer:

- Node.js version 24 or above. npm version 11 or above ships with Node. We can confirm this by executing `node -v` and `npm -v` in the terminal. There are multiple ways to install Node.js and npm. Here is a helpful article that goes into more detail: https://www.nodejsdesignpatterns.com/blog/5-ways-to-install-node-js .
- VS Code (optional), a popular editor for JavaScript and TypeScript. It is open source, has solid TypeScript support, and offers many extensions. It can be downloaded from https://code.visualstudio.com .

The code for this book is available at the book's repo. To access the repository link, follow the steps in the *"Download the example code files"* section in the *Preface* . Clone it and enter the repository root:

```
git clone https://github.com/PacktPublishing/React-Application-Architecture-for-Production-Second-Edition.git
```

The repository contains chapter folders with the code for each chapter, along with a shared `api` folder that includes the API server used across all chapters.

We are working on *Chapter 2* , so navigate to the `chapter-02` directory:

```
cd React-Application-Architecture-for-Production-Second-Edition/chapter-02
```

Next, install the dependencies:

```
npm install
```

We also need to provide the environment variables:

```
cp .env.example .env
```

At this point, the frontend should be ready and running at http://localhost:5173 .

Now, we should have the project code ready.

For more information about the setup details, check out the `README.md` file.

Choosing a meta framework for the project

Before we dive into the meta framework setup, we need to understand what a meta framework is and why we need it. In this section, we'll cover the following topics:

- What is a meta framework, and why do we need it?
- The React meta framework landscape
- Making the right choice
- Why we're using React Router
- How to get started with React Router

What is a meta framework, and why do we need it?

When we start building web applications with libraries such as React, we quickly realize that these tools only solve one part of the problem. React, being a UI library, handles our components and basic state management properly, but what about routing? **Server-side rendering** (**SSR**)? Build optimization? API integration? Suddenly, we're spending more time configuring tools than building features.

This is where meta frameworks come in. They're a layer that sits on top of our UI library or base framework, providing all the application-level infrastructure that we need for our project.

The pain points they solve

Building a production-ready web application involves many decisions and configurations that have nothing to do with our actual business logic. We need to figure out how to handle different types of rendering (client-side, server-side, and static), set up routing with code splitting, optimize our bundles, configure TypeScript, set up linting, handle data fetching, and deploy everything reliably.

If we tried to set all of this up from scratch, we'd spend days (or weeks) just configuring tools before writing a single line of product code. Even then, we'd likely end up with a setup that's hard to maintain, hard to update, and full of subtle issues that only show up in production.

Meta frameworks solve these problems directly. Routing, rendering modes (**client-side rendering** (**CSR**), SSR, and static), code splitting, and data-

fetching patterns are all handled out of the box. Instead of wiring all these pieces together ourselves, we get a well-thought-out system that works from day one, leaving us free to focus on what actually matters: building features and solving problems for our users.

The React meta framework landscape

Since this book is about React, let's focus on the main options available to us in the React ecosystem. The landscape has matured significantly over the last few years, and we now have several solid choices depending on what our project needs.

Next.js

Next.js is the most popular choice in the React world. It's backed by Vercel and has the largest community, which means excellent documentation, plenty of tutorials, and easier hiring. It comes with a lot built in: automatic image optimization, React Server Components, partial prerendering, and more.

React Router (in framework mode)

React Router has been around for a long time as a client-side routing library, but it also ships with a framework mode that is the successor to Remix. In framework mode, it acts as a full meta framework with SSR and data loading built in. It focuses on web standards and progressive enhancement.

TanStack Start

TanStack Start is the newest framework in this space. It emphasizes type safety and developer experience, taking a type-first approach where things such as route parameters, data loading, and navigation are fully typed out of the box.

While still evolving, TanStack Start shows promise for teams that prioritize strong TypeScript integration.

Making the right choice

All three frameworks are solid, production-ready options with strong communities behind them. The right choice for a project depends on team familiarity, hosting requirements, and how much abstraction feels comfortable. For this book, we are using React Router in framework mode.

Why we're using React Router

We chose React Router for two main reasons:

- **Deployment f lexibility** : React Router works consistently across any hosting environment with no preference for any particular platform, giving us more freedom in how we deploy our application.
- **Learning f ocus** : React Router's data-loading model maps closely to how the web platform works natively. This makes it a good fit for a book focused on understanding how React applications work.

Whichever framework we choose, the concepts we cover in this book are applicable across all of them.

How to get started with React Router

To get started with a new project, we can use the `create-react-router` CLI and generate a new project with the following command:

```
npx create-react-router@latest my-app
```

This will prompt us with a list of options to choose from. We can select the default options and the project will be generated.

This is how our initial project structure looks:

```
├── src/
│   └── app/
│       ├── app.css
│       ├── root.tsx
│       ├── routes/
│       │   └── home.tsx
│       ├── routes.ts
│       └── welcome/
│           ├── logo-dark.svg
│           ├── logo-light.svg
│           └── welcome.tsx
├── Dockerfile
├── node_modules/
├── package-lock.json
├── package.json
├── public/
│   └── favicon.ico
├── react-router.config.ts
├── README.md
├── tsconfig.json
└── vite.config.ts
```

The project structure is as follows:

- `src/app` : Application directory
- `src/app/root.tsx` : Root of the application with HTML structure and error handling
- `src/app/routes.ts` : Route configuration file that maps URLs to components
- `react-router.config.ts` : React Router configuration file
- `tsconfig.json` : TypeScript configuration file
- `vite.config.ts` : Vite configuration file

One thing we would notice if we ran the CLI is that the `app` folder is created directly in the root of the project, not in the `src` directory. This is because the CLI generates the project in the current directory. However, we will keep it in the `src` directory just because we want to reference anything that is inside the `src` directory with the `@/` alias, which we will cover in the following sections.

We will cover these files as we go through the book, so we shouldn't worry much about them for now.

Now that we have our meta framework set up, let's understand the build tool that powers it and how we can configure it to our needs.

Build tool setup overview

React Router in framework mode uses Vite as its build tool, which provides fast development and optimized production builds. In this section, we will cover the following topics:

- What is Vite, and why do we need it?
- How Vite works with React Router
- Our Vite configuration
- Development versus production builds

What is Vite, and why do we need it?

Vite is a modern build tool that provides the following:

- **Fast development** : **Hot Module Replacement** (**HMR**) for instant updates during development
- **Optimized builds** : Tree shaking, code splitting, and asset optimization for production
- **Framework agnostic** : Works with React, Vue, Svelte, and more
- **Modern standards** : Native ES modules in development; optimized bundles for production

When we're building React applications, we need a tool that can do the following:

- Bundle our code efficiently
- Handle different file types (TypeScript, JSX, CSS, and images)
- Provide fast development experience
- Optimize for production deployment

Vite solves these problems with a modern approach that's faster than traditional bundlers such as webpack.

How Vite works with React Router

React Router is built on top of Vite, which means the following:

- The **d evelopment server** runs on Vite's dev server with HMR
- The **b uild process** uses Vite's optimizations for production
- **Environment variables** are handled by Vite's system

This integration means we get the best of both worlds: React Router's routing and data-loading capabilities with Vite's fast build system.

Our Vite configuration

We're using Vite to handle our build process, and our configuration is set up in `vite.config.ts` :

```
// vite.config.ts
import { reactRouter } from "@react-router/dev/vite";
import tailwindcss from "@tailwindcss/vite";
import { defineConfig } from "vite";
import tsconfigPaths from "vite-tsconfig-paths";

export default defineConfig({
  plugins: [tailwindcss(), reactRouter(), tsconfigPaths()],
});
```

Here's what is configured here:

- The `tailwindcss()` plugin: This is the Tailwind CSS plugin that handles the CSS layer of the application.
- The `reactRouter()` plugin: This is the React Router plugin that handles routing, SSR, and other framework-specific features.
- The `tsconfigPaths()` plugin: This plugin allows us to use `@/` as an alias for our `src` directory in TypeScript files, so we can write `@/components/button` instead of `../../../components/button`. Don't worry about TypeScript for now; we will cover it in the next section.

Type-checking setup overview

Now that we understand our build tool, we need to set up TypeScript for type checking. In this section, we will cover the following topics:

- What is TypeScript, and why do we need it?
- How TypeScript works
- Our TypeScript configuration
- Running TypeScript checks

TypeScript is a very valuable tool in modern React development. Let's see why and how we can configure it to our needs.

What is TypeScript, and why do we need it?

TypeScript is a programming language that extends JavaScript by adding static type definitions. Think of it as JavaScript with a safety net—it helps us catch errors before they happen in production.

When we're building a React application, or any other JavaScript application in general, we often run into issues such as the following:

- Passing a string where we expected a number
- Calling a function that doesn't exist or with the wrong arguments
- Accessing properties on objects that might be undefined or do not exist
- Forgetting to handle all the possible cases in our code

TypeScript catches these issues at compile time, before our users ever see them.

TypeScript is especially useful for large applications built by large teams. Code written in TypeScript is much better documented than code written in vanilla JavaScript. By looking at the type definitions, we can figure out how a piece of code is supposed to work.

Another reason is that TypeScript makes refactoring much easier because most of the issues can be caught before running the application.

TypeScript also helps inside our IDE with autocomplete, hover information, and signature information, which speeds up our productivity.

How TypeScript works

TypeScript works by analyzing our code and understanding the types of data we're working with. Here is an example:

```
// Function expects a number
function double(value: number) {
  return value * 2;
}

// TypeScript is happy - we passed a number
double(21); // Works fine, returns 42

// TypeScript catches the error - we passed a string
double("21"); // TypeScript error: Argument of type 'string' is not assignable to parameter of type
'number'
```

The key benefit is that TypeScript catches these mistakes at compile time, before our users ever see them.

Our TypeScript configuration

We're using TypeScript to catch errors before they reach production, and our setup is configured to be strict but practical. Let's look at how we've configured it in our `tsconfig.json` file:

```
{
  "include": [
    "**/*",
    "**/.server/**/*",
    "**/.client/**/*",
    ".react-router/types/**/*"
  ],
  "compilerOptions": {
    "lib": ["DOM", "DOM.Iterable", "ES2022"],
```

```
    "types": ["node", "vite/client"],
    "target": "ES2022",
    "module": "ES2022",
    "moduleResolution": "bundler",
    "jsx": "react-jsx",
    "rootDirs": [".", "./.react-router/types"],
    "baseUrl": ".",
    "paths": {
      "@/*": ["src/*"]
    },
    "esModuleInterop": true,
    "verbatimModuleSyntax": true,
    "noEmit": true,
    "resolveJsonModule": true,
    "skipLibCheck": true,
    "strict": true
  }
}
```

Here are the most important parts of the preceding configuration:

- `"strict" : true`: This enables all strict type-checking options. It might seem aggressive, but it catches a lot of potential bugs early.
- `"jsx": "react-jsx"` : This tells TypeScript how to handle our React components. We don't need to import React in every file anymore.
- `"noEmit": true` : We're letting Vite handle the actual compilation, so TypeScript just does the type checking.
- `"paths": { "@/*": ["./src/*"] }` : This lets us use `@/` as an alias for our `src` directory, so we can write absolute paths to our files, such as `@/components/button` instead of `../../../components/button` . Remember `tsconfigPaths ()` ? It allows Vite to resolve paths that are defined here.

We also have a `typecheck` script in our `package.json` file that runs both React Router's type generation and TypeScript compilation:

```
"typecheck": "react-router typegen && tsc"
```

This ensures that our types are always up to date with our routes before we run the type checker.

React Router type generation

The `react-router typegen` command allows React Router to generate TypeScript types for our routes based on the file structure and exports in our route files. This gives us the following:

- **Type-safe route parameters** : We get autocomplete and type checking for route parameters
- **Type-safe loaders and actions** : Our data-loading functions are fully typed
- **Type-safe navigation** : When navigating between routes, we get type safety for params and search parameters

These generated types are placed in the `.react-router/types` directory (which is included in our `tsconfig.json`), and they're regenerated whenever we run the `typecheck` command. This means our route types are always in sync with our actual route files.

Running TypeScript checks

We can run type checking using our configured scripts:

```
# Run type checking once (includes React Router type generation)
npm run typecheck

# Run type checking in watch mode (reruns on file changes)
npx tsc --watch

# Run type checking for specific files
npx tsc --noEmit src/features/ideas/api/create-idea.ts
```

TypeScript catches type errors, but there are other issues that can occur. That's where linting comes in.

Linting setup overview

TypeScript's type system is very powerful for catching errors at compile time, but it doesn't enforce coding standards or catch all potential issues. For example, TypeScript won't complain if we declare variables we never use, write overly complex functions, or use inconsistent formatting. To maintain a clean, consistent, and high-quality code base across our entire team, we need a linter. In this section, we will cover the following topics:

- What is linting, and why do we need it?

- ESLint overview
- Our ESLint configuration
- Running ESLint checks

What is linting, and why do we need it?

Linting is the process of analyzing our source code to find potential errors, bugs, stylistic errors, and suspicious constructs. Think of it as an automated code review that runs every time we write code.

When we're working in a team, everyone will have different coding styles and preferences. Some might prefer single quotes and others double quotes. Some might forget to handle edge cases or write code that's hard to read.

A linter helps us by doing the following:

- **Catching bugs early** : Finding potential issues before they become problems
- **Enforcing a consistent style** : Making sure all our code looks the same
- **Teaching best practices** : Showing us better ways to write code
- **Preventing common mistakes** : Catching things we might miss

ESLint overview

ESLint is the most popular JavaScript/TypeScript linter. It works by parsing our code and then running rules against it to find issues.

For example, we might write the following:

```
const unusedVariable = "hello";
console.log("This is fine");
```

ESLint can detect that `unusedVariable` is declared but never used and warn us about it.

Our ESLint configuration

Our ESLint configuration is quite comprehensive and helps us maintain code quality across the entire team.

Let's break down what we've set up in `eslint.config.js` :

```
import js from '@eslint/js';
import typescript from '@typescript-eslint/eslint-plugin';
```

```
import typescriptParser from '@typescript-eslint/parser';
import { defineConfig } from 'eslint/config';
import prettierConfig from 'eslint-config-prettier';
import checkFile from 'eslint-plugin-check-file';
import importPlugin from 'eslint-plugin-import';
import prettier from 'eslint-plugin-prettier';
import react from 'eslint-plugin-react';
import reactHooks from 'eslint-plugin-react-hooks';
import globals from 'globals';

... the rest of the configuration
```

We're using the flat config format (the new ESLint configuration style) and have several key plugins worth mentioning:

- `@typescript-eslint` : TypeScript-specific linting rules
- `eslint-plugin-react` : React-specific rules
- `eslint-plugin-react-hooks` : Rules for React hooks
- `eslint-plugin-prettier` : Integrates Prettier with ESLint, which we will cover in a moment
- `eslint-plugin-import` : Import/export linting
- `eslint-plugin-check-file` : Filenaming conventions

Here are some of the important rules we've configured:

```
rules: {
  'prettier/prettier': 'error',
  'react/react-in-jsx-scope': 'off',
  'react/prop-types': 'off',
  '@typescript-eslint/no-unused-vars': 'warn',
  '@typescript-eslint/no-explicit-any': 'warn',
  'import/order': [
    'error',
    {
      groups: ['builtin', 'external', 'internal', 'parent', 'sibling', 'index'],
      'newlines-between': 'always',
      alphabetize: { order: 'asc', caseInsensitive: true },
    },
  ],
  'import/no-cycle': ['error', { maxDepth: Infinity }],
  'check-file/filename-naming-convention': [
    'error',
    {
      '**/*.{js,jsx,ts,tsx}': 'KEBAB_CASE',
    },
  ],
}
```

Here is what each rule does:

- `prettier/prettier` : Treats any Prettier formatting issues as ESLint errors, so formatting problems show up alongside linting issues in a single pass.
- `react/react-in- jsx -scope` : In older React versions, we had to import React at the top of every JSX file. Modern React no longer requires this, so we turn this rule off.
- `react/prop-types` : In JavaScript projects, `prop-types` was used to validate component props at runtime. Since we use TypeScript, our props are already statically typed, so this rule is unnecessary.
- `@typescript -eslint/no-unused-vars` : Warns us when we declare variables that are never used, keeping the code clean and avoiding confusion.
- `@typescript -eslint/no-explicit-any` : Warns us when we use the `any` type. Using `any` defeats the purpose of TypeScript's type safety, so this forces us to use proper types instead.
- `import/order` : Enforces a consistent order for imports: built-in modules first, then external packages, then internal imports, with each group separated by a blank line and sorted alphabetically. This makes imports easier to scan at a glance.
- `import/no-cycle` : Prevents circular dependencies between files. Circular imports can cause hard-to-debug issues and are usually a sign of an architectural problem.
- `check-file/filename-naming-convention` : Enforces kebab-case naming for all JavaScript/TypeScript files (e.g., `my- component.tsx` instead of `MyComponent.tsx`). This keeps filenaming consistent across the entire project without having to think about it each time we create a new file.

We also have some custom import rules that help enforce our project structure; we'll cover those in detail when we talk about our project organization.

Which rules we are using is something that should be decided by the whole team or organization. Once decided, we should add them to the ESLint

configuration file and stick to them.

Running ESLint

We can run linting using our configured scripts:

```
# Run linting on all files
npm run lint

# Run linting and automatically fix what can be fixed
npm run lint:fix

# Run linting on specific files or directories
npx eslint src/features/ideas/
npx eslint src/components/button.tsx
```

ESLint will show us errors and warnings, and with the `--fix` flag, it can automatically fix many issues, such as formatting, unused imports, and simple code style problems.

Formatting setup overview

Now that we've covered linting, let's look at code formatting. In this section, we will cover the following topics:

- What is code formatting, and why do we need it?
- How Prettier works
- Configuring Prettier
- Running Prettier checks

What is code formatting, and why do we need it?

Code formatting is about making our code look consistent and readable. When we're working in a team, we all have different preferences for how code should look—some like semicolons while others don't; some prefer single quotes while others prefer double quotes.

The problem is that these differences can make code reviews harder and create unnecessary diffs and conflicts. When we're focused on the logic of our code, we don't want to be distracted by formatting inconsistencies.

How Prettier works

Prettier is a code formatter that automatically formats our code according to a set of rules. It's great because we don't have to debate whether to use semicolons or not; Prettier handles it for us.

Prettier works by doing the following:

1. **Parsing our code** : Understanding the structure of our JavaScript, TypeScript, CSS, and so on
2. **Applying formatting rules** : Making it look consistent
3. **Outputting formatted code** : Giving us back clean, readable code

The beauty of Prettier is that it's consistent across the entire team. Everyone's code looks the same, regardless of their personal preferences.

Configuring Prettier

Prettier has some opinionated defaults, but we can configure it to our needs. We can find the configuration file in `.prettierrc` , which is already set up for us:

```
{
  "semi": true,
  "trailingComma": "all",
  "singleQuote": true,
  "printWidth": 80,
  "tabWidth": 2,
  "useTabs": false
}
```

The key settings here are the following:

- `"trailingComma": "all"` : This adds trailing commas everywhere possible, which makes Git diffs cleaner when we add new items to arrays or objects
- `"singleQuote": true` : We use single quotes for strings, which is a common preference in the React community
- `"printWidth": 80` : We keep lines under 80 characters, which makes code more readable on smaller screens

Prettier is integrated with ESLint through the `eslint-plugin-prettier` plugin, so formatting issues show up as linting errors. This ensures that our code is both properly formatted and follows our linting rules.

Prettier will automatically format our code according to our configuration, and since it's integrated with ESLint, formatting issues will also show up when we run `npm run lint .`

With our type checking, linting, and formatting tools configured, we need to make sure they actually run before code enters our repository. We will see how in the next section.

Pre-commit checks setup overview

Having static code analysis tools such as TypeScript, ESLint, and Prettier is great; we have configured them and can run individual scripts whenever we make some changes to ensure everything is in the best order.

However, there are some drawbacks. Developers can forget to run all checks before committing to the repo, which can still bring problematic and inconsistent code to production.

Fortunately, there is a solution that can fix this problem: whenever we try to commit to the repository, we want to run all checks in an automated way. This is where pre-commit hooks come in.

What are pre-commit hooks, and why do we need them?

Pre-commit hooks are scripts that run automatically before we can commit our code to the repository. They act as a safety net, ensuring that only high-quality code makes it into our code base.

Without pre-commit hooks, we might accidentally commit the following:

- Code that doesn't compile
- Code that has linting errors
- Code that's not properly formatted
- Code that breaks our tests

Pre-commit hooks catch these issues before they end up in a pull request or, even worse, in production.

How pre-commit hooks work

Git hooks are scripts that Git runs at certain points in the Git workflow. The pre-commit hook runs just before a commit is created, and if it exits with a non-zero status, the commit is blocked.

Check the following diagram to understand how pre-commit hooks work:

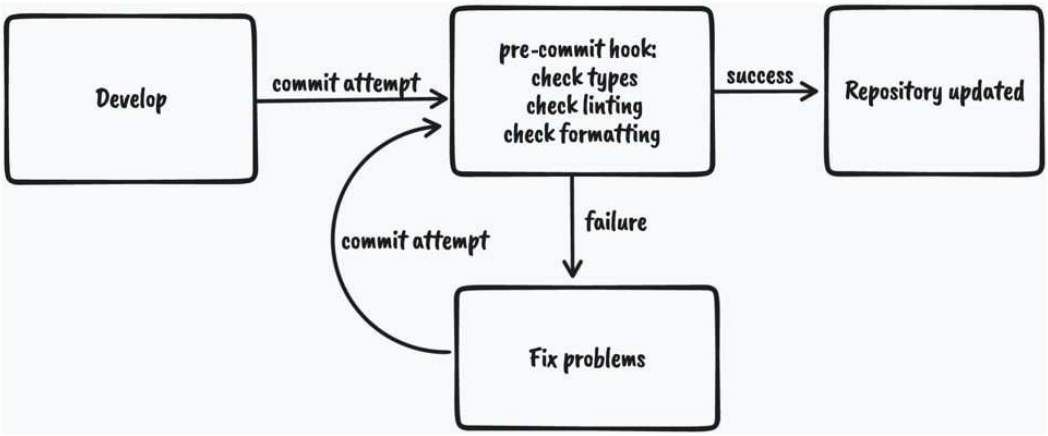


Figure 2.1 – How pre-commit hooks work

This means we can't commit bad code; we have to fix the issues first. It might seem annoying at first, but it prevents us from accidentally breaking the build or introducing bugs or inconsistencies.

Our pre-commit workflow

As we can see, whenever we attempt to commit to the repository, the Git pre-commit hook will run and execute the scripts that will do the checking. If all the checks pass, the changes will be committed to the repository; otherwise, we will have to fix the issues and try again.

To enable this flow, we will use `husky` and `lint-staged` :

- `husky` is a tool that allows us to run Git hooks. We want to run the pre-commit hook to run the checks before committing our changes.
- `lint-staged` is a tool that allows us to run those checks only on files that are in the staging area of Git. This improves the speed of code checking since doing that on the entire code base might be too slow.

Before starting, we need to make sure the Git repository is initialized if it isn't already. Then we can install `husky` and `lint-staged` with the following commands:

```
npm install --save-dev husky lint-staged
```

Then, we would need to initialize `husky` to enable Git hooks:

```
npx husky init
```

A new folder named `.husky` will be created with a file named `pre-commit` that will contain the following content:

```
npm test
```

Let's modify the file to run the `lint-staged` script:

```
npm run lint-staged
```

The `husky` pre-commit hook will run `lint-staged`, so we should define what commands `lint-staged` should run inside the `lint-staged.config.js` file:

```
// infra/lint-staged.config.js
export default {
  '*.{js,jsx,ts,tsx}': ['eslint --fix'],
  '*.{json,md,css,scss,html}': ['prettier --write'],
  '*.{ts,tsx}': ['bash -c "npm run typecheck"'],
};
```

This means the following:

- JavaScript/TypeScript files get linted and fixed
- Other files just get formatted with Prettier
- TypeScript files also get type-checked

The beauty of this setup is that it's fast (only checks changed files) and comprehensive (catches formatting, linting, and type issues).

Running pre-commit checks

We can run our pre-commit checks manually:

```
# Run all pre-commit checks (linting, formatting, type checking)
npm run pre-commit

# Run only lint-staged (checks only changed files)
npm run lint-staged

# Run individual checks
npm run lint
npm run format
npm run typecheck
```

These checks also run automatically when we try to commit code, but we can run them manually to catch issues before committing.

It's worth noting that since we are now in the `chapters/chapter-02` directory, the pre-commit hook will not work if we try to run it from the current setup. The `.husky` folder would need to be in the root of the repository to work.

With our development tooling in place, let's look at how we organize our code files and folders in our project so we can keep our code base scalable and maintainable.

Project structure overview

How we organize our files and folders has a huge impact on how easy it is to navigate and maintain our code base. Let's explore different approaches and understand why we chose a feature-based structure for our application.

What is project structure, and why does it matter?

When we're building a React application, we have to decide how to organize the following:

- Components (reusable UI pieces)
- Pages (full screens)
- API calls (data fetching)
- State management (application data)
- Utilities (helper functions)
- Tests (quality assurance)

The wrong structure can make our code base hard to navigate, leading to the following:

- Difficulty finding related code
- Unclear dependencies between different parts
- Harder code reviews
- Slower development as the project grows

Different approaches to project structure

There are several common approaches to organizing React applications, each with its own trade-offs. We will go through each one next.

File type structure

Here's how a file-type-based structure typically looks in a project:

```
src/
├── components/
├── pages/
├── hooks/
├── utils/
└── types/
```

The following are the advantages of this approach:

- Simple to get started with
- Easy to find all components in one place
- Clear separation by file type (components, pages, hooks, utils, and types)

The following are the disadvantages of this approach:

- Hard to find related code when working on a feature
- Components become tightly coupled across features
- Difficult to scale as the application grows as the code base becomes too large to navigate
- Code reviews become harder (need to look in multiple folders)
- Difficult to remove a feature when not used anymore because it is all over the place

Feature-based structure

Here's how a feature-type-based structure typically looks in a project:

```
src/
├── app/
├── components/
├── features/
│   ├── auth/
│   ├── ideas/
│   ├── profile/
│   └── reviews/
└── lib/
```

The following are the advantages of this approach:

- Easy to find all code related to a feature
- Features can be developed independently
- Clear boundaries prevent tight coupling
- Scales well as the application grows
- Team members can work on different features without conflicts

The following are the disadvantages of this approach:

- Slightly more complex to set up initially
- Need to think about where shared code belongs
- Requires discipline to maintain boundaries, but this is solved by ESLint rules

Why we chose the feature-based structure

Here's why this is the right choice for our application:

- **Clear feature boundaries** : Our app has distinct features, such as authentication, ideas management, user profiles, and reviews. Each feature has its own components, API calls, and logic.
- **Clear code flow** : The code flow is clear and easy to understand.
- **Team collaboration** : Different developers can work on different features (auth, ideas, and reviews) without stepping on each other's toes.
- **Scalability** : As we add new features, we can add them as new feature folders without affecting existing code.
- **Code reviews** : When reviewing a feature, we only need to look at the files in that feature folder, making reviews more focused and efficient.
- **Deployment** : In the future, we could potentially split features into separate repos or micro-frontends or deploy them independently.
- **Ease of removal** : If a feature is not used anymore, it can be removed easily without affecting the rest of the code base.

The slight complexity of setting up boundaries is worth it because it pays dividends as our application grows and our team expands.

Here's how we will structure our `src` directory:

```
src/
├── app/              # Core application setup and entry points
├── components/       # Reusable UI components (not feature-specific)
├── config/           # Configuration files
├── features/         # Feature-based modules
│   ├── auth/        # Authentication feature
│   ├── ideas/       # Ideas management feature
│   ├── profile/     # User profile feature
│   └── reviews/     # Reviews feature
├── hooks/           # Custom React hooks (shared across features)
├── lib/             # Utility functions and third-party integrations
├── stores/          # Global state management
├── types/           # TypeScript type definitions
```

Each feature folder contains everything related to that feature:

```
features/ideas/
├── api/              # API calls for this feature
├── components/       # Feature-specific components
├── config/           # Feature configuration (query keys, etc.)
├── hooks/            # Feature-specific hooks
└── locales/          # Internationalization for this feature
```

This structure has several benefits:

- **Easy to find code** : When we need to work on ideas, we know exactly where to look
- **Clear boundaries** : Each feature is self-contained with its own components, API calls, and logic
- **Scalable** : Adding new features doesn't affect existing ones
- **Team-friendly** : Different developers can work on different features without conflicts
- **Modular** : Features can be developed, tested, and deployed independently

How the code flows

Let's take a look at how the code flows in our application.

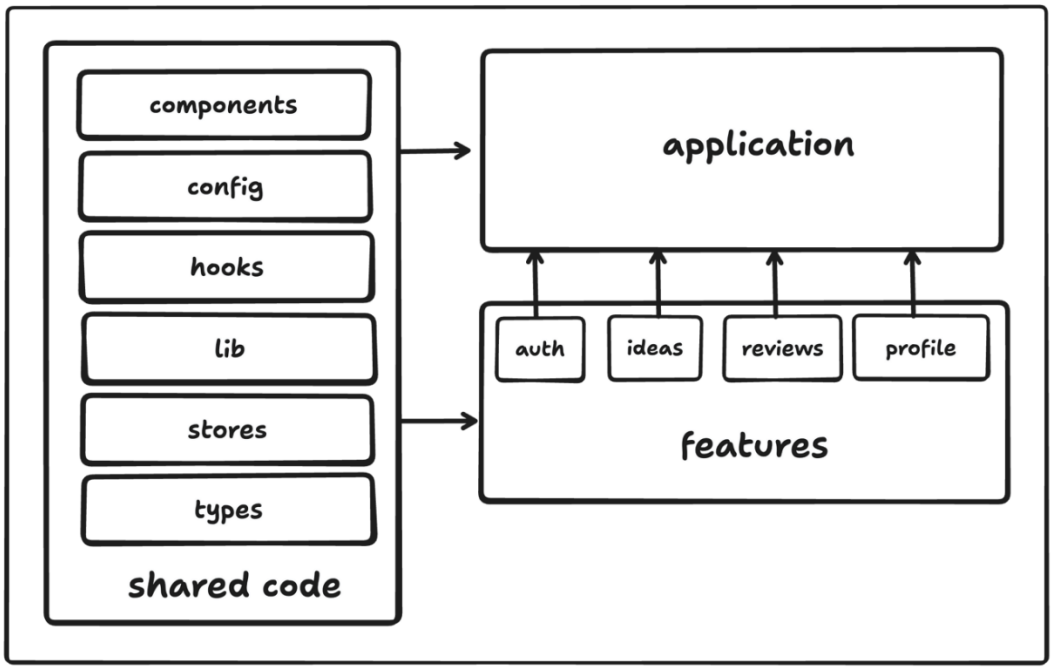


Figure 2.2 – Code flow in our code base

The code flows in a clear, unidirectional manner. We will go through each level next.

Application (top-level)

The `app` directory is at the top of our code base hierarchy.

- It can import from anywhere: features, components, hooks, lib, stores, and types
- Example: A route in `app/routes/dashboard/ideas/ ideas.tsx` can import from `features/ideas/components/ idea- list.tsx`
- This is where most of the code is composed together to form the application

Features (mid level)

Features sit in the middle of the hierarchy.

- They can import from shared utilities (components, hooks, lib, stores, and types)
- They can import from other features only if explicitly allowed
- They cannot import from the `app` directory

- Example: The `ideas` feature can import from `features/auth/hooks/use-user` because `auth` is an allowed dependency
- Example: The `ideas` feature cannot import from `features/profile` or `features/reviews`

Shared utilities (bottom-level)

Shared utilities are at the bottom of the hierarchy.

- They can only import from each other
- They cannot import from features or app
- Example: A component in `components` can use a hook from `hooks` or a utility from `lib`
- Example: A hook in `hooks` cannot import from `features/auth`

Why this matters

Let's look at a real example. Imagine we're displaying a list of ideas:

```
// ALLOWED: App imports from feature
// app/routes/dashboard/ideas/ideas.tsx
import { IdeaList } from '@/features/ideas/components/idea-list';

// ALLOWED: Feature imports from auth feature (explicit dependency)
// features/ideas/components/idea-list.tsx
import { useUser } from '@/features/auth/hooks/use-user';

// ALLOWED: Feature imports from shared utilities
// features/ideas/components/idea-list.tsx
import { Button } from '@/components/ui/button';
import { formatDate } from '@/lib/date';

// FORBIDDEN: Feature cannot import from app
// features/ideas/api/get-ideas.ts
import { loader } from '@/app/routes/dashboard/ideas/ideas'; // ESLint error!

// FORBIDDEN: Feature cannot import from non-allowed feature
// features/ideas/components/idea-list.tsx
import { ProfileCard } from '@/features/profile/components/profile-card'; // ESLint error!

// FORBIDDEN: Shared utility cannot import from feature
// components/idea-card.tsx
import { useUser } from '@/features/auth/hooks/use-user'; // ESLint error!
```

This unidirectional flow ensures the following:

- Shared utilities remain truly reusable and don't have hidden dependencies
- Features stay independent and can be developed/tested in isolation
- The app layer orchestrates everything without being imported by lower layers
- Dependencies are explicit and controlled

Enforcing project structure with ESLint

Having such a standard is great and the code flow is clear, but how do we enforce it? If we leave it to our developers to follow the rules, we will have to rely on them to do the right thing. Humans make mistakes. They should not have to worry about these kinds of things while building features and solving actual problems. This is where ESLint comes into the picture.

We will use `import/no-restricted-paths` from `eslint -plugin-import` to set these constraints.

To keep the `eslint.config.js` file clean, we will use the `infra/eslint-import-rules.js` file to generate the rules for import constraints.

If we open the `eslint.config.js` file, we'll see the following configuration:

```
// eslint.config.js
import { importRules } from './infra/eslint-import-rules.js';

export default [
  // ... other config
  {
    rules: {
      'import/no-restricted-paths': [
        'error',
        {
          zones: importRules, // Our custom rules are applied here
        },
      ],
    },
  },
];
```

Each entry in `zones` defines a restriction: a `target` (the files being protected) and a `from` (the files they are not allowed to import from).

Now let's walk through each constraint defined in `infra/eslint-import-rules.js` .

Constraint 1: Shared utilities must remain independent

To enforce this, we define the following rule:

```
// infra/eslint-import-rules.js
zones.push({
  target: [
    './src/components/**',
    './src/config/**',
    './src/hooks/**',
    './src/lib/**',
    './src/stores/**',
    './src/types/**',
    './src/utils/**',
  ],
  from: ['./src/features/**', './src/app/**'],
  message:
    'Shared utilities should not import from features or app directories.',
});
```

This rule ensures that shared folders (components, config, hooks, lib, stores, types, and utils) cannot import from features or the app. This keeps our shared code truly reusable. If a utility needs feature-specific logic, it should accept that logic as a parameter or be moved into the feature.

Constraint 2: Features cannot import from the app

To maintain this boundary, we define another rule:

```
zones.push({
  target: `./src/features/**/**`,
  from: `./src/app/**/**`,
  message: `Features should not import from app directory.`,
});
```

This maintains unidirectional flow where features cannot import from `app` . This keeps features independent of how they're used in routes, allowing them to be reused throughout the application.

Constraint 3: Feature dependencies are explicit and controlled

This is configured with the following rule:

```
// infra/eslint-import-rules.js
const features = [
  { name: 'auth', allowedFeatures: [] },
  { name: 'ideas', allowedFeatures: ['auth'] },
  { name: 'profile', allowedFeatures: ['auth'] },
  { name: 'reviews', allowedFeatures: ['auth'] },
];
```

This creates a dependency configuration where we have the following:

- `auth` is the foundation; it cannot import from other features
- `ideas` , `profile` , and `reviews` can only import from `auth`
- No feature can import from features at the same level (e.g., `ideas` cannot import from `profile`)

This prevents circular dependencies and keeps features independent. If we need shared code between features, we move it to the `components` or `lib` directory.

Why this matters

These ESLint rules prevent common problems in large code bases:

- **Prevents circular dependencies** : A clear hierarchy makes circular imports impossible
- **Reduces tight coupling** : Features stay independent with well-defined interfaces
- **Stops architectural drift** : Violations are caught immediately, preventing shortcuts
- **Makes dependencies explicit** : There are no hidden dependencies in shared utilities
- **Enables isolated testing** : Each feature can be tested independently
- **Improves onboarding** : Architecture is self-documenting through ESLint config

ESLint in action

When we violate these rules, ESLint gives us clear error messages:

```
// src/features/profile/components/profile-stats.tsx
import { IdeaCard } from '@/features/ideas/components/idea-card';
// ESLint Error: profile feature should not import from ideas features.
```

The solution depends on our use case:

- **Move to shared** : If multiple features use a component or a hook, or anything that needs to be used across multiple features, move it to one of the shared folders, such as `src /components` or `src /lib` .
  Example: A generic `Card` component used by both `ideas` and `profile`
- **Pass as props** : If the component is specific to one feature, keep it there and pass data through `props` .
  Example: A profile displaying the idea count by receiving it as a prop
- **Add dependency** : Only if there's a legitimate architectural reason for one feature to depend on another, update `allowedFeatures` in `infra/eslint-import-rules.js` .
  Example: `profile` legitimately needs to use `auth` 's session check

These rules automatically check that we're following the architecture. When we try to import from the wrong place, ESLint will tell us immediately. This keeps our features independent and our code easy to understand.

Now that we have our project structure and code quality tools in place, there's one final piece of our setup: handling configuration that changes between environments. Environment variables allow us to configure our application without hardcoding values.

Environment variables setup overview

Our application needs different configurations for development, staging, and production. Environment variables let us manage these configurations without changing our code. Let's see how to set them up properly with type safety and validation.

What are environment variables, and why do we need them?

Without environment variables, we'd have to hardcode configuration values into our code:

```
// Bad: hardcoded values
const API_URL = "https://api.production.com";
const API_KEY = "1234567890";
```

This creates several problems:

- We can't use different APIs for different environments
- We'd accidentally commit sensitive information to our repository
- We'd have to change the code every time we want to switch environments

Our environment variable system

We've set up a robust environment variable system that validates our configuration at startup. Here's how it works:

```
// src/config/env.ts
const envMapping = {
  API_URL: 'VITE_API_URL',
} as const;

export const envSchema = z.object({
  API_URL: z.url('API_URL must be a valid URL'),
});

const parseEnv = () => {
  const rawEnv: Record<string, string | undefined> = {};

  for (const [cleanKey, viteKey] of Object.entries(envMapping)) {
    rawEnv[cleanKey] = import.meta.env[viteKey];
  }

  try {
    return envSchema.parse(rawEnv);
  } catch (error) {
    // ... error handling
  }
};

export const env = parseEnv();
```

Here's why we've set it up this way:

- **Clean variable names** : We use `envMapping` to map the environment variable names to the clean variable names, such as `API_URL` , internally instead of `VITE_API_URL` . We need to do this because Vite prefixes the environment variable names with `VITE_` .
- **Type safety** : Zod validates that our environment variables are the right type.

- **Clear error messages** : If a variable is missing or invalid, we get a helpful error message.

Note that we're not just reading environment variables; we're validating them and providing a clean, typed interface for the rest of our application to use.

We can now import and use environment variables, as follows:

```
import { env } from '@/config/env';

console.log(env.API_URL);
```

This approach prevents runtime errors caused by missing or invalid environment variables, and it makes it clear what configuration our application needs to run, as the whole setup is centralized.

Setting up environment variables

In our project, we provide a `. env.example` file that lists all the environment variables that our application needs:

```
# .env.example
VITE_API_URL=http://localhost:9999
```

To set up our local development environment, we copy this file to `.env` :

```
cp .env.example .env
```

Then we update the values in `.env` to match our local setup. The `.env` file is git-ignored, so we can safely add our own values without committing them to the repository.

If we try to run the application without setting the environment variables, we'll get the following error:

```
3:10:35 PM [vite] Internal server error: Environment validation failed:
API_URL (VITE_API_URL): API_URL must be a valid URL

Please check your .env file and ensure all required variables are set.
```

This is great because it prevents us from deploying the application with missing environment variables. The error message clearly shows which variable is missing and what the expected format is.

With all these tools and structures in place—our meta framework, TypeScript, linting, formatting, pre-commit hooks, project organization, and environment configuration—we have a solid foundation for building maintainable React applications.

Summary

In this chapter, we established the foundation for building a production-ready React application.

We started by choosing **React Router** as our meta framework for its flexibility in supporting both CSR and SSR without locking us into a specific hosting provider.

We also covered **Vite** as our build tool and how we can configure it to our needs.

We then set up our development workflow with **TypeScript** for type safety, **ESLint** for code quality, **Prettier** for consistent formatting, and **pre-commit** hooks to automatically catch issues before they reach our repository.

The most important architectural decision was adopting a feature-based project structure. Instead of grouping files by type (components, hooks, and utils), we organized them by feature (auth, ideas, profile, and reviews). We enforced feature boundaries with ESLint rules, preventing circular dependencies and keeping our code base modular as it grows.

Finally, we implemented environment variable validation using **Zod** to ensure our configuration is correct before the application even starts, catching configuration errors early.

With these tools and patterns in place, we have everything we need to build maintainable, scalable React applications.

Get this book's PDF copy, code bundle, and more

Scan the QR code (or go to `packtpub.com/unlock`). Search for this book by name, confirm the edition, and then follow the steps on the page.

UNLOCK NOW

Note: Have your invoice handy. Purchases made directly from the Packt website don ' t require an invoice.

3

Building and Documenting Components

In React, everything is a component. This paradigm allows us to split user interfaces into smaller parts, making it easier to develop larger user interfaces and applications. It also adheres to the DRY (Don't Repeat Yourself) principle by enabling component reusability since we can reuse the same components in multiple places.

In this chapter, we will learn how to build the foundational components for our application's design system. This will make the application UI more consistent and easier to understand and maintain. We will also learn how to document these components with Storybook, a great tool that serves as a catalog for all our reusable UI elements.

We'll cover the following topics:

- Anatomy of a component
- Creating our component library
- Documenting components

By the end of this chapter, we'll have a solid understanding of component architecture and a reusable component library documented with Storybook that we can use throughout our application.

Technical requirements

Before we get started, we need to set up our project. To develop our project, we need the following tools installed on our computer:

- Node.js version 24 or above. npm version 11 or above ships with Node. We can confirm this by executing `node -v` and `npm -v` in the terminal. There are multiple ways to install Node.js and npm. Here is a helpful article that goes into more detail: https://www.nodejsdesignpatterns.com/blog/5-ways-to-install-node-js .
- VS Code (optional), a popular editor for JavaScript and TypeScript. It is open source, has solid TypeScript support, and offers many extensions. It can be downloaded from https://code.visualstudio.com .

The code for this book is available at the book's repo. To access the repository link, follow the steps in the *"Download the example code files"* section in the *Preface* . Clone it and enter the repository root:

```
git clone https://github.com/PacktPublishing/React-Application-Architecture-for-Production-Second-Edition.git
```

The repository contains chapter folders with the code for each chapter, along with a shared `api` folder that includes the API server used across all chapters.

We are working on *Chapter 3* , so navigate to the `chapter-03` directory:

```
cd React-Application-Architecture-for-Production-Second-Edition/chapter-03
```

Next, install the dependencies:

```
npm install
```

We also need to provide the environment variables:

```
cp .env.example .env
```

At this point, the frontend should be ready and running at http://localhost:5173 .

Now we should have the project code ready.

For more information about the setup details, check out the `README.md` file.

Anatomy of a component

Before we start building our component library, we need to understand what a component is and how it works. In this section, we'll cover the following:

- What is a component?
- Component props
- Component state
- Event handlers
- TypeScript and components

Let's explore each of these concepts using a practical example.

What is a component?

A component is a self-contained, reusable piece of UI that encapsulates its own structure, behavior, and styling. Components combine to form complete user interfaces.

Here's a simple example:

```
// src/components/counter.tsx
import { useState } from 'react';

export type CounterProps = {
  initialValue: number;
  label: string;
  onIncrement?: (newValue: number) => void;
  onDecrement?: (newValue: number) => void;
};

export function Counter(props: CounterProps) {
  const [value, setValue] = useState(props.initialValue);

  const handleIncrement = () => {
    const newValue = value + 1;
    setValue(newValue);
    props.onIncrement?.(newValue);
  };

  const handleDecrement = () => {
    const newValue = value - 1;
    setValue(newValue);
    props.onDecrement?.(newValue);
  };

  return (
    <div>
      <label>{props.label}</label>
      <div className="flex items-center gap-2">
        <button onClick={handleDecrement}>-</button>
        <p>{value}</p>
```

```
        <button onClick={handleIncrement}>+</button>
      </div>
    </div>
  );
}
```

This component demonstrates the key parts of a React component:

- **The component function** : At its core, a component is just a JavaScript function that returns JSX (JavaScript XML), which looks like HTML but is actually JavaScript. The function describes what should appear on the screen. Under the hood, the JSX compiler transforms markup into plain `React.createElement` calls, so `<button onClick={fn}>Click</button>` becomes: `React.createElement('button', { onClick: fn }, 'Click')` .
- **Props (p roperties)** : The `props` parameter is how we pass data into components. Props can be anything – strings, numbers, functions, objects, arrays, or even other components. They make components dynamic and reusable.
- **TypeScript t ypes** : Notice the `CounterProps` type definition. TypeScript will prevent us from passing a string where we need a number or forgetting required props. It also gives us excellent IntelliSense in our IDE.
- **State** : The `useState` hook lets the component remember values between renders. When we call `setValue` , React re-renders the component with the new value, updating what users see onscreen.
- **Event h andlers** : Functions such as `handleIncrement` respond to user interactions. When someone clicks the increment button, we update the state and optionally call a callback prop.

Using the component

Once we've defined a component, we can use it anywhere in our application:

```
// src/app/routes/home.tsx
import { Counter } from '@/components/counter';

export default function HomePage() {
  return (
    <div>
      <h1>Home</h1>
      <Counter
```

```
        initialValue={0}
        label="Click Counter"
        onIncrement={(newValue) => {
          console.log(`Incremented to ${newValue}`);
        }}
        onDecrement={(newValue) => {
          console.log(`Decremented to ${newValue}`);
        }}
      />
    </div>
  );
}
```

When this renders, users will see a " **Click Counter** " label, the current count value, and two buttons. Clicking the buttons updates the displayed number and logs messages to the console.

We could use this same `Counter` component multiple times on a page, each with different `initialValue` and `label` props, and they'd all work independently.

Now that we understand component basics, let's build a component library that will serve as a base for our application.

Creating our component library

When building a production application, we need a consistent set of UI components – buttons, inputs, dialogs, forms, and more. We have several options for how to get these components. In this section, we'll cover the following:

- What is a component library, and why do we need one?
- Different approaches to component libraries
- What is Shadcn UI and why did we choose it?
- How to set up Shadcn UI

Let's start by understanding what a component library is and why we need one.

What is a component library, and why do we need one?

Without a component library, we'd build each button, input, or modal from scratch every time we need one. This leads to the following:

- **Inconsistent UI** : Different buttons styled differently across the app
- **Repeated code** : The same components built multiple times
- **Harder maintenance** : Need to update the same component in many places
- **Accessibility issues** : Easy to miss keyboard navigation, screen readers, and ARIA attributes
- **Slower development** : Building everything from scratch takes time

A component library solves these problems by providing a set of pre-built, tested, accessible components that we can reuse throughout our application.

Different approaches to component libraries

There are several ways to get a component library for our React application:

Option 1: Install a package (Material-UI, Ant Design, Chakra UI, Mantine)

We install a library via `npm` and import components:

```
import Button from '@mui/material/Button';

<Button variant="contained">Click me</Button>
```

Pros:

- Quick to get started
- Well-tested and maintained
- Good documentation and community

Cons:

- Large bundle sizes (we ship the entire library even if we use a few components)
- Limited customization (following the library's design system)
- Upgrades can break things

At the other end of the spectrum, we have the option to build everything ourselves.

Option 2: Build from scratch

We create all components ourselves from basic HTML elements.

Pros:

- Full control over everything
- No extra dependencies
- Exactly what we need

Cons:

- Time-consuming to build
- Need to handle accessibility ourselves
- Need to test everything
- Reinventing the wheel

There's also a middle ground that combines the benefits of both approaches.

Option 3: Shadcn UI (copy-paste components)

Shadcn UI takes a different approach – it's not an npm package. Instead, it's a collection of accessible components that we copy directly into our project. The components are built with Radix UI or Base UI (which are headless component libraries for accessibility and behavior) and styled with Tailwind CSS. We will use Base UI for our project as it is being more actively maintained at the time of writing this book, which is always something to consider when choosing a component library.

Pros:

- Components live in our code base (full control and customization)
- No runtime dependencies to manage
- Smaller bundle sizes (only install and ship what we use)
- Built on Base UI for functionality and accessibility
- Easy to modify and extend

Cons:

- Need to manually update components if Shadcn UI releases improvements or in case of any upstream updates of Base UI or Tailwind
- Initial setup required

For this project, we'll use Shadcn UI. Let's look at why it fits our needs.

Why we chose Shadcn UI

For this project, we chose Shadcn UI for the following reasons:

- **Ownership** : The components become part of our code base, so we can customize them however we need without worrying about library constraints or breaking changes from upstream
- **Modern s tack** : It uses Tailwind CSS for styling, which aligns with modern React development practices and makes styling intuitive and fast
- **Accessibility** : Built on Base UI, which handles complex accessibility requirements (keyboard navigation, focus management, ARIA attributes) that would take significant time to implement correctly ourselves
- **Bundle Size** : We only include components we actually use, keeping our application lean
- **Developer Experience** : The CLI makes adding components simple, and since they're in our code base, we can see exactly how they work

Configuring Shadcn UI

Shadcn UI components can be added to our project using the CLI or manually. Let's look at both approaches.

Using the CLI

The Shadcn CLI sets up all the necessary files and configurations for us if we run the following:

```
npx shadcn@latest init --base=base --preset=vega
```

We are choosing Base UI as the component library and Vega as the preset. This command will do the following:

- Install necessary dependencies
- Create configuration files
- Set up Tailwind CSS styles
- Configure path aliases

The CLI generates a `components.json` file that contains the configuration for generating components:

```
{
  "$schema": "https://ui.shadcn.com/schema.json",
  "style": "base-vega",
  "rsc": false,
  "tsx": true,
  "tailwind": {
    "config": "",
    "css": "src/app/app.css",
    "baseColor": "neutral",
    "cssVariables": true,
    "prefix": ""
  },
  "iconLibrary": "lucide",
  "rtl": false,
  "aliases": {
    "components": "@/components",
    "utils": "@/lib/utils",
    "ui": "@/components/ui",
    "lib": "@/lib",
    "hooks": "@/hooks"
  },
  "menuColor": "default",
  "menuAccent": "subtle",
  "registries": {}
}
```

Here's what this configuration does:

- `style` : The design style to use (`base-vega` to use Base UI as the component library with Vega preset)
- `tailwind.css` : Where to find our Tailwind CSS file
- `aliases` : Path mappings that match our TypeScript configuration, which Shadcn UI needs to know because it will use these folders to store required components and utilities
- `iconLibrary` : Icon library to use (`lucide-react`)

Adding components

Now we can add components using the CLI:

```
npx shadcn@latest add button
```

This command will generate and add the component to our code base. The generated button component looks like this:

```tsx
// src/components/ui/button.tsx

import { Button as ButtonPrimitive } from '@base-ui/react/button';
import { cva, type VariantProps } from 'class-variance-authority';

import { cn } from '@/lib/utils';

const buttonVariants = cva(
  "group/button inline-flex shrink-0 items-center justify-center rounded-md border border-transparent bg-clip-padding text-sm font-medium whitespace-nowrap transition-all outline-none select-none focus-visible:border-ring focus-visible:ring-3 focus-visible:ring-ring/50 disabled:pointer-events-none disabled:opacity-50 aria-invalid:border-destructive aria-invalid:ring-3 aria-invalid:ring-destructive/20 dark:aria-invalid:border-destructive/50 dark:aria-invalid:ring-destructive/40 [&_svg]:pointer-events-none [&_svg]:shrink-0 [&_svg:not([class*='size-'])]:size-4",
  {
    variants: {
      variant: {
        default: 'bg-primary text-primary-foreground hover:bg-primary/80',
        outline:
          'border-border bg-background shadow-xs hover:bg-muted hover:text-foreground aria-expanded:bg-muted aria-expanded:text-foreground dark:border-input dark:bg-input/30 dark:hover:bg-input/50',
        secondary:
          'bg-secondary text-secondary-foreground hover:bg-secondary/80 aria-expanded:bg-secondary aria-expanded:text-secondary-foreground',
        ghost:
          'hover:bg-muted hover:text-foreground aria-expanded:bg-muted aria-expanded:text-foreground dark:hover:bg-muted/50',
        destructive:
          'bg-destructive/10 text-destructive hover:bg-destructive/20 focus-visible:border-destructive/40 focus-visible:ring-destructive/20 dark:bg-destructive/20 dark:hover:bg-destructive/30 dark:focus-visible:ring-destructive/40',
        link: 'text-primary underline-offset-4 hover:underline',
      },
      size: {
        default:
          'h-9 gap-1.5 px-2.5 in-data-[slot=button-group]:rounded-md has-data-[icon=inline-end]:pr-2 has-data-[icon=inline-start]:pl-2',
        xs: "h-6 gap-1 rounded-[min(var(--radius-md),8px)] px-2 text-xs in-data-[slot=button-group]:rounded-md has-data-[icon=inline-end]:pr-1.5 has-data-[icon=inline-start]:pl-1.5 [&_svg:not([class*='size-'])]:size-3",
        sm: 'h-8 gap-1 rounded-[min(var(--radius-md),10px)] px-2.5 in-data-[slot=button-group]:rounded-md has-data-[icon=inline-end]:pr-1.5 has-data-[icon=inline-start]:pl-1.5',
        lg: 'h-10 gap-1.5 px-2.5 has-data-[icon=inline-end]:pr-3 has-data-[icon=inline-start]:pl-3',
        icon: 'size-9',
        'icon-xs':
          "size-6 rounded-[min(var(--radius-md),8px)] in-data-[slot=button-group]:rounded-md
```

```
[&_svg:not([class*='size-'])]:size-3",
        'icon-sm':
          'size-8 rounded-[min(var(--radius-md),10px)] in-data-[slot=button-group]:rounded-md',
        'icon-lg': 'size-10',
      },
    },
    defaultVariants: {
      variant: 'default',
      size: 'default',
    },
  },
);

function Button({
  className,
  variant = 'default',
  size = 'default',
  ...props
}: ButtonPrimitive.Props&VariantProps<typeof buttonVariants>) {
  return (
    <ButtonPrimitive
      data-slot="button"
      className={cn(buttonVariants({ variant, size, className }))}
      {...props}
    />
  );
}

export { Button, buttonVariants };
```

These are key features of the generated component:

- **Type-safe** : Uses TypeScript for props validation
- **Variants** : Multiple visual styles (`default` , `destructive` , `outline` , `secondary` , `ghost` , `link`)
- **Sizes** : Different sizes (`default, sm, lg, icon`)
- Extensible: Can override styles with the className prop
- **Accessible** : Built on Base UI primitives

We can generate multiple components at once by providing multiple components to the `add` command:

```
npx shadcn@latest add button card
```

Creating components manually

We can also add components manually by visiting the Shadcn UI website:

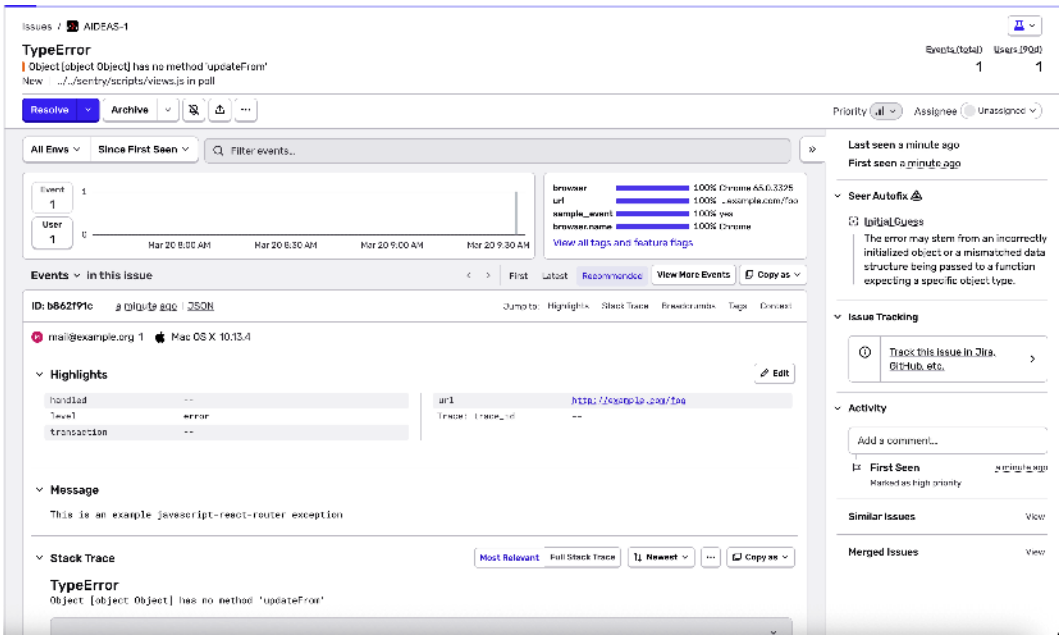


Figure 3.1 – Installing Shadcn UI components manually

The manual process involves the following:

- Creating the component file in `src/components/ui/`
- Copying the component code from the website to the component file
- Installing any required dependencies manually

While the CLI is faster, the manual approach gives us more control over what's being added to our project.

Now that we have our components, we need a way to document and try them out.

For the latest setup instructions, always check the official Shadcn UI documentation at https://ui.shadcn.com/

Documenting components

Our `Button` component has different variants and sizes. We need a way to document and test these different states. In this section, we'll explore two approaches:

- Creating a dedicated route in our application
- Using Storybook for component documentation

We can start from the simplest approach.

Creating a component route

A simple way to view our components is to create a dedicated route. We can create a components route in our application:

```
// src/app/routes/components.tsx
import { Button } from '@/components/ui/button';

export default function Components() {
  return (
    <div className="container mx-auto py-8 space-y-12">
      <div>
        <h1 className="text-3xl font-bold mb-8">Components in Action</h1>
        <section className="space-y-6">
          <h2 className="text-2xl font-semibold">Button Component</h2>

          <div className="space-y-4">
            <h3 className="text-lg font-medium">Variants</h3>
            <div className="flex flex-wrap gap-4">
              <Button variant="default">Default</Button>
              <Button variant="destructive">Destructive</Button>
              <Button variant="outline">Outline</Button>
              <Button variant="secondary">Secondary</Button>
              <Button variant="ghost">Ghost</Button>
              <Button variant="link">Link</Button>
            </div>
          </div>

          <div className="space-y-4">
            <h3 className="text-lg font-medium">Sizes</h3>
            <div className="flex flex-wrap items-center gap-4">
              <Button size="sm">Small</Button>
              <Button size="default">Default</Button>
              <Button size="lg">Large</Button>
              <Button size="icon-sm">ρ</Button>
              <Button size="icon">ρ </Button>
              <Button size="icon-lg">ρ </Button>
            </div>
          </div>
        </section>
      </div>
    </div>
  );
}
```

This renders all the button variants:

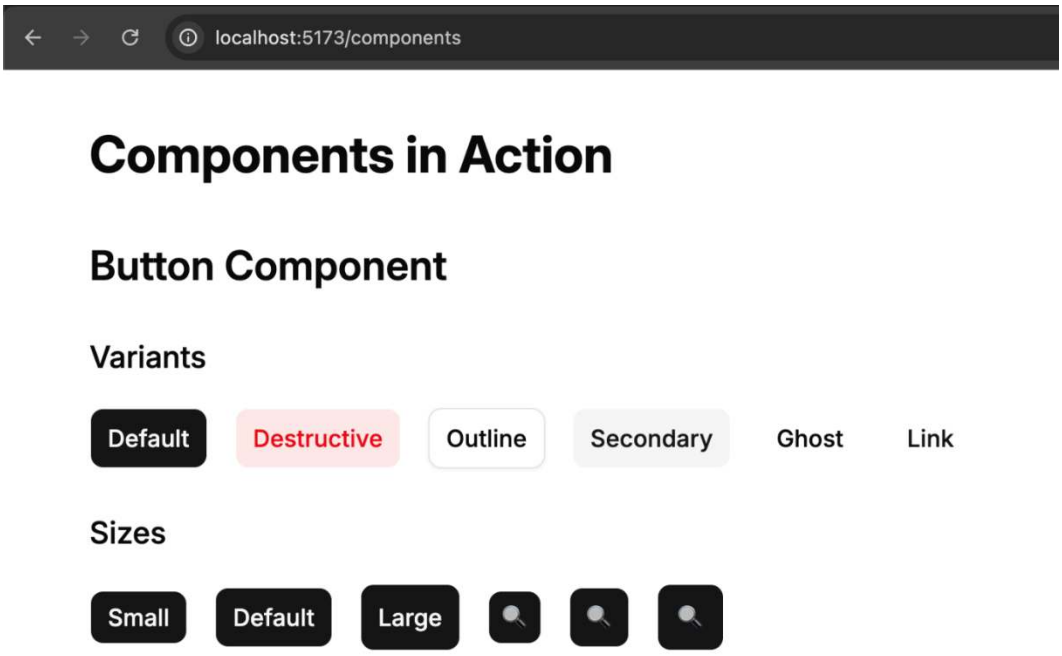


Figure 3.2 – Documenting components on a dedicated application route

This approach works without much additional setup, but has several problems:

- **Exposes components in the application** : Our application routes should focus on features, not component documentation
- **Difficult to maintain** : Adding every component and variant to one route becomes crowded
- **No interactivity** : We can't easily test different prop combinations
- **No isolation** : Components aren't developed in isolation from the application

We need a better solution. That's where Storybook comes in.

What is storybook?

Storybook is a tool for developing and testing UI components in isolation. It serves as a catalog of our application's components.

These are the benefits of using Storybook:

- **Component isolation** : Develop and test components without running the full application.
- **Visual testing** : See all variants of a component in one place.

- Interactive playground: Experiment with different prop combinations in real time.
- **Living documentation** : Stories stay up to date with our code.
- **Team collaboration** : Designers and product managers can review components without technical setup.
- **Regression testing** : Catch visual bugs by comparing screenshots of stories. If you want to dive deeper into regression testing with Storybook, check out their docs at `https://storybook.js.org/docs/writing-tests/visual-testing` .

Let's see how we can configure Storybook to work with our project.

Configuring Storybook

We can use Storybook CLI to install and configure everything automatically.

Storybook is initialized in the project with the following command:

```
npm create storybook@latest
```

This command will do the following:

- Install Storybook and its dependencies
- Create a `.storybook` directory with configuration files
- Add Storybook scripts to our `package.json`
- Detect our project setup (React, Vite, TypeScript) and configure accordingly

There are a few things we need to configure to make Storybook work with our project.

Configuring Vite for storybook

Since we're using React Router with Vite, we need to make a small adjustment to our Vite configuration. The React Router plugin should only run when we're building our application, not when running Storybook.

We need to update our `vite.config.ts` :

```
// vite.config.ts
import { reactRouter } from "@react-router/dev/vite";
import tailwindcss from "@tailwindcss/vite";
import { defineConfig } from "vite";
import tsconfigPaths from "vite-tsconfig-paths";

const isStorybook = process.env.STORYBOOK === 'true';

export default defineConfig({
  plugins: [tailwindcss(), !isStorybook && reactRouter(), tsconfigPaths()],
});
```

Here's what we changed:

- `const isStorybook = process.env.STORYBOOK === 'true';` : Detects if we're running Storybook.
- `!isStorybook && reactRouter()` : Only loads the React Router plugin when we're not running Storybook. We don't need it in Storybook; that's why we are disabling it.

This prevents conflicts between React Router and Storybook in the Vite setup.

Configuring Storybook preview

The `.storybook/preview.ts` file controls how our stories are displayed. We need to import our application styles so our components look the same in Storybook as they do in our application:

```
// .storybook/preview.ts
import type { Preview } from '@storybook/react-vite'
import '../src/app/app.css';

const preview: Preview = {
  parameters: {
    controls: {
      matchers: {
       color: /(background|color)$/i,
       date: /Date$/i,
      },
```

```
    },
  },
};

export default preview;
```

Here's what this configuration does:

- `import '../src/app/app.css'` : Imports our Tailwind CSS styles so components render with proper styling
- `parameters.controls.matchers` : Tells Storybook to provide color pickers for color props and date pickers for date props

Running Storybook

The CLI added two new scripts to our `package.json` :

```
{
  "scripts": {
    "storybook": "STORYBOOK=true storybook dev -p 6006",
    "build-storybook": "STORYBOOK=true storybook build"
  }
}
```

We can now run Storybook with the following:

```
# Start Storybook development server
npm run storybook

# Build static Storybook for deployment
npm run build-storybook
```

The `storybook` command starts a development server on port `6006` . When we open `http://localhost:6006` in our browser, we'll see the Storybook interface with all our documented components.

Documenting components with Storybook

Now that we have Storybook configured, let's create our first story. Stories are examples of how a component can be used with different props.

Creating a story file

Story files use the naming convention `ComponentName.stories.tsx` . We'll use **Component Story Format version 3** (**CSF3**), which is the modern way to write Storybook stories. Let's create a story for our `Button` component:

```
// src/components/ui/button.stories.tsx
import type { Meta, StoryObj } from '@storybook/react-vite';

import { Button } from './button';

const meta = {
  title: 'UI/Button',
  component: Button,
  parameters: {
    layout: 'centered',
  },
  tags: ['autodocs'],
  argTypes: {
    variant: {
      control: 'select',
      options: [
        'default',
        'destructive',
        'outline',
        'secondary',
        'ghost',
        'link',
      ],
    },
    size: {
      control: 'select',
      options: ['default', 'sm', 'lg', 'icon'],
    },
    disabled: {
      control: 'boolean',
    },
  },
} satisfies Meta<typeof Button>;

export default meta;
type Story = StoryObj<typeof meta>;

export const Default: Story = {
```

```
  args: {
    children: 'Button',
    variant: 'default',
  },
};

export const Destructive: Story = {
  args: {
    children: 'Delete',
    variant: 'destructive',
  },
};

// ... more stories ...
```

Let's break down what each part does:

Meta configuration

The `meta` object defines the configuration for all stories of this component:

- `title: 'UI/Button'` : Organizes the story in Storybook's sidebar under **UI > Button**
- `component: Button` : Tells Storybook which component these stories are for
- `parameters: { layout: 'centered' }` : Centers the component in the canvas
- `tags: ['autodocs']` : Automatically generates documentation from the component's props and stories
- `argTypes` : Defines interactive controls for component props in Storybook's controls panel

ArgTypes configuration

The `argTypes` section creates interactive controls in Storybook:

- `variant` : Creates a dropdown with all variant options
- `size` : Creates a dropdown with all size options
- `asChild` and `disabled` : Create toggle switches for boolean props

Individual stories

Each exported story represents a different state or variation of the component:

- `Default` : Shows the default button

- `Destructive` : Shows the destructive variant (for delete actions)
- `Outline` , `Secondary` , `Ghost` , `Link` : Show different visual styles
- `Small` and `Large` : Shows different sizes
- `Disabled` : Shows the disabled state

Each story has an args object that defines the props passed to the component. These can be modified in Storybook's controls panel to see how the component responds to different inputs.

Viewing stories in Storybook

When we run `npm run storybook` and open `http://localhost:6006` , we'll see the Storybook interface:

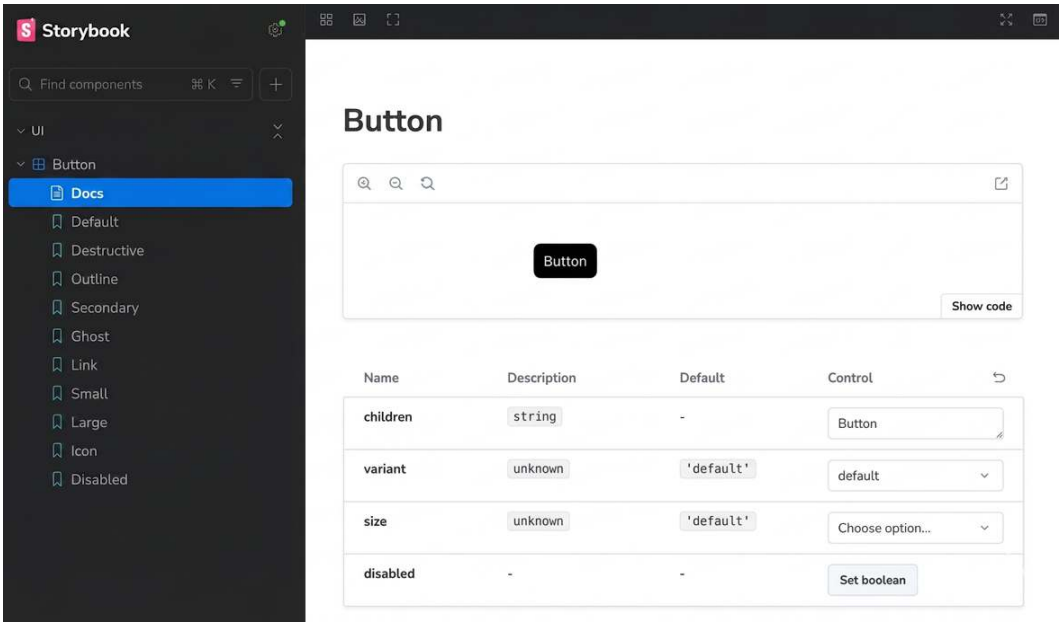


Figure 3.3 – Viewing documented components in Storybook

The Storybook interface has several key areas:

- **Sidebar (left)** : Lists all our components and their stories
- **Canvas (center)** : Shows the rendered component
- **Controls panel (bottom)** : Interactive controls to modify component props
- **Docs tab** : Auto-generated documentation showing all variants and prop descriptions

We can do the following:

- Click on different stories to see different states
- Use the controls panel to change props interactively
- Switch to the **Docs** tab to see all variants at once
- Copy code examples for how to use the component

With our component library and documentation system in place, we can build our application UI with confidence.

Summary

In this chapter, we built the foundation for our application's user interface.

We started by understanding what components are – self-contained pieces of UI that use props for input, state for managing values, event handlers for interactions, and TypeScript for type safety.

Next, we evaluated different approaches to component libraries. We chose Shadcn UI because it gives us full control by copying components directly into our code base, while still benefiting from Base UI for accessibility and Tailwind CSS for styling. We can use the shadcn CLI to add components, or we can copy the code manually.

Finally, we set up Storybook for component documentation. We configured Storybook to work with React Router and Vite, created story files that showcase different component states, and learned how Storybook provides an isolated environment for developing and testing components. This gives us a living catalog that designers, developers, and stakeholders can reference.

With our component library and documentation system in place, we're ready to start building the features of our application.

Get this book's PDF copy, code bundle, and more

Scan the QR code (or go to `packtpub.com/unlock`). Search for this book by name, confirm the edition, and then follow the steps on the page.

Note: Have your invoice handy. Purchases made directly from the Packt website don ' t require an invoice.

4

Routing and Rendering Strategies

When a user navigates to our application's URL, the application needs to decide which UI should be shown to the user. This is where routing comes in. Beyond just displaying the right page, we also need to consider how and when that page gets rendered: whether on the server, in the browser, or a combination of both. These decisions can have a significant impact on performance, user experience, and SEO.

We'll cover the following topics:

- Routing with React Router
- Rendering strategies
- Adding meta tags to pages
- Adding page layouts

By the end of this chapter, we'll have a solid understanding of how routing works in modern React applications, how to configure routes effectively, and how to choose the right rendering strategy for each page to optimize performance and user experience.

Technical requirements

Before we get started, we need to set up our project. To develop our project, we need the following tools installed on our computer:

- Node.js version 24 or above. npm version 11 or above ships with Node. We can confirm this by executing `node -v` and `npm -v` in the terminal. There are multiple ways to install Node.js and npm. Here is a helpful article that goes into more detail:

https://www.nodejsdesignpatterns.com/blog/5-ways-to-install-node-js .

- VS Code (optional), a popular editor for JavaScript and TypeScript. It is open source, has solid TypeScript support, and offers many extensions. It can be downloaded from https://code.visualstudio.com .

The code for this book is available at the book's repo. To access the repository link, follow the steps in the *"Download the example code files"* section in the *Preface* . Clone it and enter the repository root:

```
git clone https://github.com/PacktPublishing/React-Application-Architecture-for-Production-Second-Edition.git
```

The repository contains chapter folders with the code for each chapter, along with a shared `api` folder that includes the API server used across all chapters.

We are working on *Chapter 4* , so navigate to the `chapter-04` directory:

```
cd React-Application-Architecture-for-Production-Second-Edition/chapter-04
```

Next, install the dependencies:

```
npm install
```

We also need to provide the environment variables:

```
cp .env.example .env
```

At this point, the frontend should be ready and running at http://localhost:5173 .

Now we should have the frontend ready.

For more information about the setup details, check out the `README.md` file.

Routing with React Router

Routing in React Router is very straightforward. Everything starts with a route configuration file.

Configuring routes in React Router

In React Router, we can configure the routes in the `src/app/routes.ts` file. This is a special file that React Router expects to find in the `src/app` directory and

uses it to manage the routes of the application.

Here's a simple example:

```
// src/app/routes.ts

import {
  type RouteConfig,
  route,
} from '@react-router/dev/routes';

export default [
  route('/', './routes/home.tsx'),
] satisfies RouteConfig;
```

We define the routes using the `route` function. The first argument is the path of the route, and the second argument is the path to the route module. In this example, the route path is `/`, and the route module is defined in `./routes/home.tsx`.

But what does a route module look like?

React Router expects a route module to export the UI component of the page as the default export, so it would look something like this:

```
// src/app/routes/home.tsx

export default function HomePage() {
  return <div>Welcome to the home page!</div>;
}
```

Now, if the user navigates to `/`, the `HomePage` component will be rendered and the text `Welcome to the home page!` will be displayed.

But what if we had a page that needed to have a dynamic path? For example, the idea detail page. In such cases, we would have a path like this: `/ideas/1`, where `1` is the ID of the idea. Since there could be many more ideas with different IDs, it would be very impractical to define a separate route for each idea, so we would have to use a dynamic route instead.

In that case, we could define the route like this:

```
// src/app/routes.ts

export default [
```

```
  route('/ideas/:id', './routes/ideas/idea.tsx'),
] satisfies RouteConfig;
```

This configuration would define a route that matches any path that starts with `/ideas/` , and `:id` is a dynamic parameter that will be passed to the component.

```
// src/app/routes/ideas/idea.tsx

export default function IdeaPage(props: Route.ComponentProps) {
  const ideaId = props.params.id;
  return <div>Idea {ideaId}</div>;
}
```

This would mean that whenever the user navigates to `/ideas/1` , the `IdeaPage` component will be rendered and `id` would be passed as a prop to the component via `props.params.id` .

Once we have the routes configured, we need a mechanism to navigate between the pages in the application. We don't want to type the URL in the browser every time we want to navigate to a page; there is a better way.

Navigating between pages

To define links between the pages in web applications, we usually use the `<a>` tag to link to other pages.

In traditional multi-page applications, clicking a link causes the browser to request a completely new HTML document from the server. This results in a visible page refresh and interrupts the user experience.

Modern frameworks such as React Router solve this problem by using client-side routing. When a user clicks a link, JavaScript intercepts the navigation, updates the URL in the browser, and then loads and renders the appropriate component without requesting a new HTML document from the server. This creates a smooth, app-like experience as we don't see the full page refresh.

They are able to do this by providing the `Link` and `NavLink` components to define links between the pages in a declarative way.

Using Link for basic navigation

The `Link` component is the most common way to navigate between pages. It renders an anchor tag (`<a>`) that intercepts clicks to prevent full page reloads:

```tsx
// src/components/navigation.tsx
import { Link } from 'react-router';

<Link to="/" className="text-xl font-bold">
  AIdeas
</Link>
```

When a user clicks this link, React Router updates the URL to `/` and renders the corresponding route component without a page reload. The `to` prop specifies the destination path.

Using NavLink for navigation with active states

The `NavLink` component is similar to the `Link` component, but it also extends it with the ability to apply styling based on whether the link matches the current route. This is very useful for navigation menus where we want to highlight the current page:

```tsx
// src/components/navigation.tsx
import { NavLink } from 'react-router';

<NavLink
  to="/ideas"
  end
  className={({ isActive }) =>
    cn(
      'flex items-center gap-2 text-sm hover:text-primary px-3 py-2 rounded-md',
      isActive && 'font-semibold',
    )
  }
>
  <Lightbulb className="h-4 w-4" /> Discover Ideas
</NavLink>
```

The `className` prop accepts a function that receives an `isActive` boolean. When the current URL matches the `to` path, `isActive` is `true`, allowing us to apply different styles. In this example, we make the text bold when the link is active.

Notice the `end` prop on this `NavLink`. By default, `NavLink` considers a route "active" if the current URL starts with the `to` path. This means `/ideas/123` would mark both `/ideas` and `/ideas/123` as active. The `end` prop tells `NavLink` to only consider the link active if it's an exact match, preventing the ideas index route from being highlighted when viewing sub-routes.

`Link` and `NavLink` trigger navigation when the user clicks the link, but what if we want to navigate programmatically in response to user actions other than clicking a link, such as after a form submission or when handling a button click?

Using useNavigate for programmatic navigation

React Router provides the `useNavigate` hook, which can be used to navigate programmatically. Let's see a simple example:

```
import { useNavigate } from 'react-router';

function MyComponent() {
  const navigate = useNavigate();

  const handleSubmit = async (data) => {
    await saveData(data);
    // Navigate to success page
    navigate('/success');
  };

  return <form onSubmit={handleSubmit}>...</form>;
}
```

The `useNavigate` hook returns a function that can be used to navigate to a different path. The function takes the path as an argument and navigates to it.

Now that we have a basic understanding of how routing works in React Router, let's move on to which rendering strategies we can use to build the pages of our application.

Rendering strategies

When thinking about our application pages, we need to think about their needs to get a better understanding of how they should be rendered. Some factors to consider are: frequency of page content updates, number of pages, SEO requirements, performance requirements, infrastructure, and whether the page content is public or private.

We can choose some of the following rendering strategies:

- **Server-side rendering (SSR)**
- **Client-side rendering (CSR)**
- **Hybrid rendering (SSR + CSR)**

- **Static p re-rendering**

We will dive deeper into each strategy by implementing it in our application to learn more about why we would use each of them.

Server-side rendering

Server-side rendering (**SSR**) means that the page content is generated on the server for each request. This allows us to fetch fresh data from our database or API and render personalized content while still providing fast initial page loads and excellent SEO.

Let's visualize how server-side rendering works:

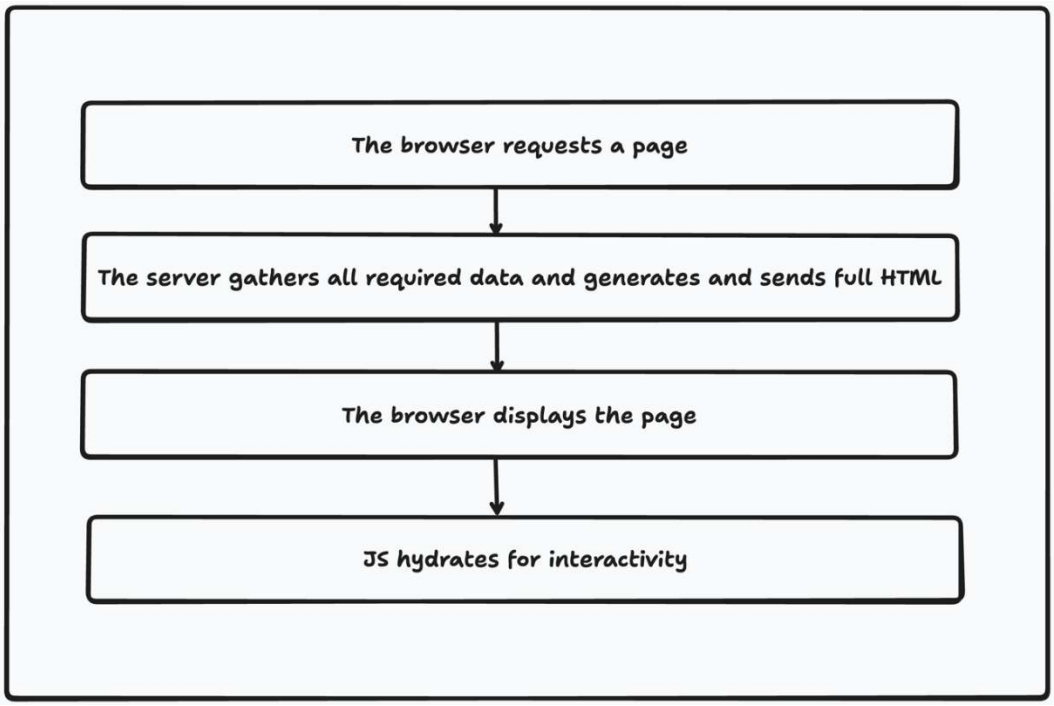


Figure 4.1 – Server-side rendering flow

Every time a user visits an SSR page, the server fetches the data from the API and renders the HTML page content. The browser receives the complete HTML and displays it immediately, then JavaScript is loaded to make the page interactive.

What are the main benefits of using SSR?

- **SEO-friendly** : Search engines see complete content since everything is already rendered on the server

- **Fast perceived load** : Content is visible immediately without the need to wait for JavaScript to load
- **Fresh data** : Always current from the database or API since the data is fetched on the server on every request
- **Works without JavaScript** : The content is visible even if JavaScript fails or is disabled

So, when should we use SSR? SSR is ideal for pages that need fresh, dynamic content with good SEO, such as a product detail page on an e-commerce platform, a profile page on a social network platform, and so on.

In the context of our application, the idea details page is a great example of a page that needs SSR because it needs to show the idea details and the reviews, which can change frequently, and it needs to be SEO-friendly.

Creating a server-rendered idea detail page

Since we are rendering the page on the server, we need a way to load all the data there and pass it to the page:

```
// src/app/routes/ideas/idea.tsx

export async function loader({ params }: Route.LoaderArgs) {
  const [idea, reviews] = await Promise.all([
    getIdeaById(params.id),
    getReviewsByIdea({ id: params.id }),
  ]);
  return routerData({
    idea,
    reviews,
  });
}
```

In React Router, this can be done via the `loader` function, which is exported from the `route` module and executed on the server side before the component is rendered. It allows us to fetch and provide the data for the page component. The loader receives `params.id` from the URL path `/ideas/:id` , fetches data from our API, and passes it to the component:

```
export default function IdeaDetailPage({ loaderData }: Route.ComponentProps) {
  const idea = loaderData?.idea;

  const reviews = loaderData?.reviews?.data;
```

```
  return (
    <div className="container mx-auto px-4 py-8">
      {/* ... */}
      <div className="max-w-4xl mx-auto">
        <IdeaDetails idea={idea} currentUser={null} />
        <div className="space-y-6">
          <div className="flex items-center justify-between">
            <h2 className="text-xl font-semibold">
              Reviews ({reviews?.length || 0})
            </h2>
          </div>
          <ReviewsList
            reviews={reviews}
            emptyMessage="No reviews yet. Be the first to review this idea!"
          />
        </div>
      </div>
    </div>
  );
}
```

Then, the component receives the data from the `loader` function via the `loaderData` prop and can use it to render the page content.

Another great thing here is that the entire route module is type-safe via TypeScript types coming from `./+types/idea`, which is automatically generated by React Router.

But what if something goes wrong with the loader and it throws an error, or the request fails? To handle errors from the loaders, we can export `ErrorBoundary` from the `route` module:

```
export function ErrorBoundary({ error }: { error: Error }) {
  return (
    <div className="container mx-auto px-4 py-8 max-w-3xl">
      {/* ... */}
    </div>
  );
}
```

If the `loader` function throws an error, the `ErrorBoundary` component will be rendered, and the error will be displayed to the user.

Once the `route` module is created, we need to register it in the `src/app/routes.ts` file.

```
// src/app/routes.ts

export default [
  // ...
  route('/ideas/:id', './routes/ideas/idea.tsx'),
] satisfies RouteConfig;
```

Now, if we navigate to `/ideas/1`, the `IdeaDetailPage` component will be rendered, and the idea details will be displayed.

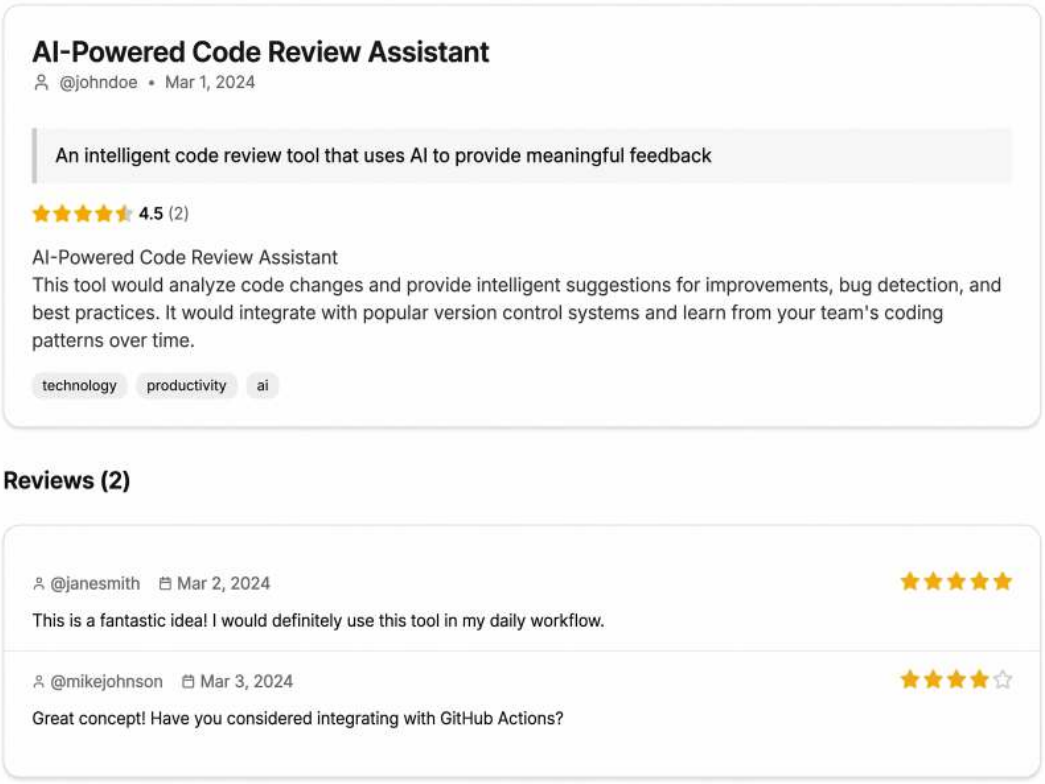


Figure 4.2 – Server-side rendered idea detail page

To verify that SSR is working, we can disable JavaScript in the browser and reload the page. All the content should still be visible.

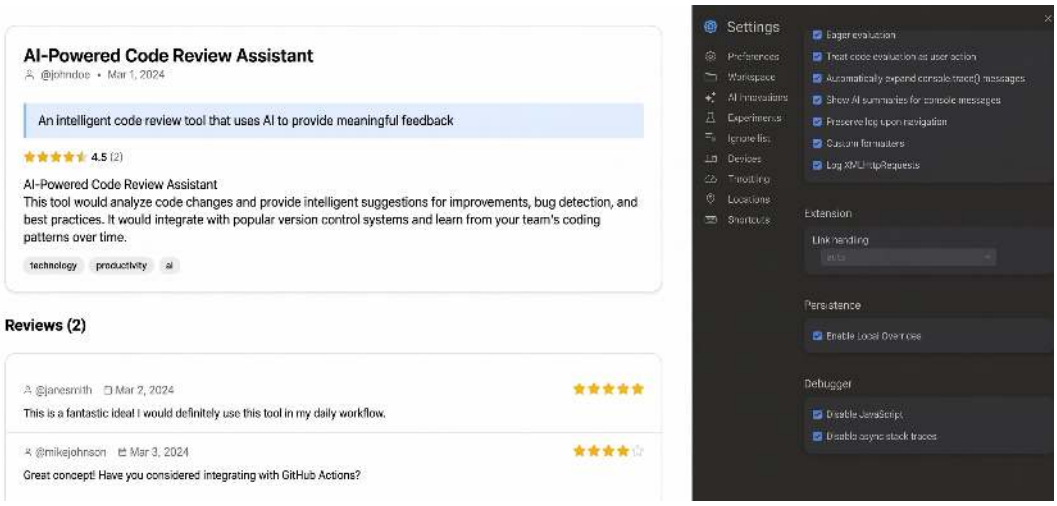


Figure 4.3 – Server-side rendered idea detail page without JavaScript

SSR is powerful, but sometimes we don't need SEO or immediate content visibility. For authenticated pages behind a login, client-side rendering is more than sufficient.

Client-side rendering

Client-side rendering (**CSR**) sends the minimal HTML from the server and fetches data using JavaScript after the page loads. This is ideal for authenticated pages where SEO doesn't matter and we want to reduce server load.

Here's how CSR works in our application:

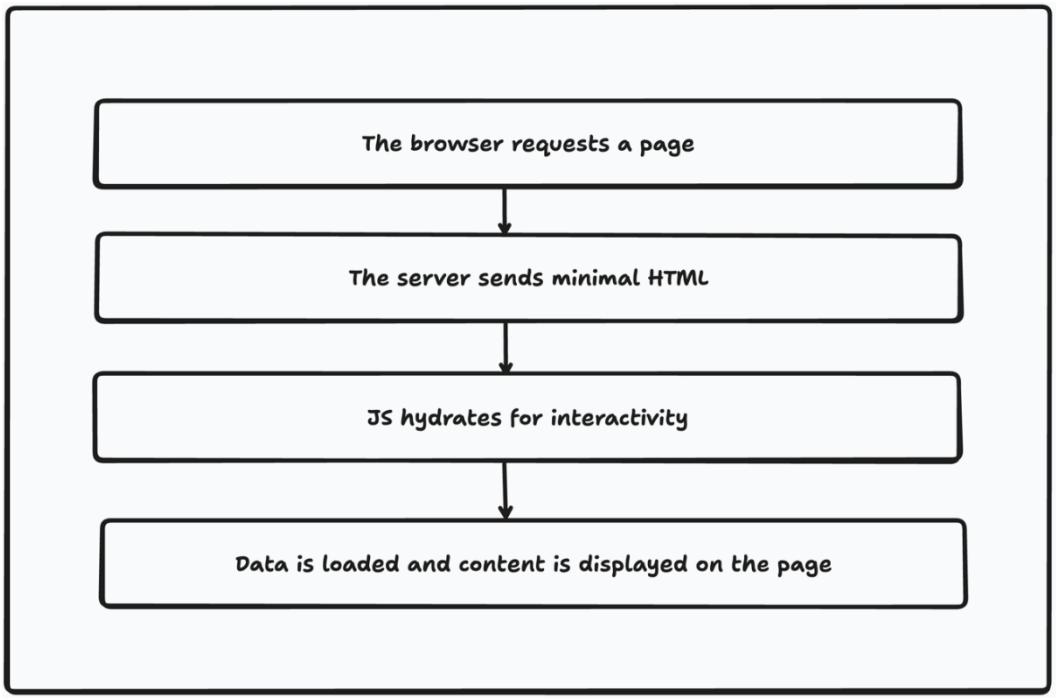


Figure 4.4 – CSR flow

When a user visits `/dashboard/ideas` , the server sends minimal HTML. JavaScript loads, the page content is rendered, and then the component fetches data. The page shows the loading state initially, and then updates the page content when the data arrives.

Some of the benefits of using CSR are as follows:

- **Reduces server load** : No server rendering per request; we only need to send the minimal HTML to the client
- **Simpler infrastructure** : No need to configure a server or a build process, which is great for reducing the costs of running the application

Some of the trade-offs of using CSR are as follows:

- Users see loading states instead of content initially, which is not as good an experience as seeing the content immediately.
- Worse SEO (fine for auth-required pages) since the page content is not generated on the server. Search engines are able to execute JavaScript and see the page content, but it's not as good as having the page content immediately available.
- Requires JavaScript to be enabled in the browser. If JavaScript is disabled, the page content will not be fully visible.

So, when should we use CSR? CSR is best for pages where SEO isn't a concern, and we want to reduce the server load. Our dashboard pages (`/dashboard/ideas` and `/dashboard/reviews`) display user-specific content that doesn't need SEO and are protected by authentication.

Creating a client-rendered dashboard page

The dashboard ideas page shows the current user's ideas. Instead of fetching in a loader, it fetches data on the client using a custom hook:

```
// src/app/routes/dashboard/ideas/ideas.tsx
import { useCurrentUserIdeasQuery } from '@/features/ideas/api/get-current-user-ideas';
import { IdeasList } from '@/features/ideas/components/ideas-list';

export default function MyIdeasPage() {
  const ideasQuery = useCurrentUserIdeasQuery();
  const ideas = ideasQuery.data?.data;

  return (
    <div className="container mx-auto px-4 py-8">
      <div className="max-w-4xl mx-auto space-y-6">
        <div className="flex items-center justify-between">
          <div>
            <h1 className="text-3xl font-bold mb-2">My Ideas</h1>
            <p className="text-muted-foreground">
              Manage and track your submitted ideas
            </p>
          </div>
        </div>

        <IdeasList
          ideas={ideas}
          isLoading={ideasQuery.isLoading}
          emptyMessage="You haven't created any ideas yet."
          error={ideasQuery.error}
        />
      </div>
    </div>
  );
}
```

The `MyIdeasPage` component is very simple, as it only fetches data on the client using a custom hook and renders the page content.

Now that we have the route module for this page, we need to register it in the `src/app/routes.ts` file.

```
// src/app/routes.ts

export default [
  // ...
  route('/dashboard/ideas', './routes/dashboard/ideas/ideas.tsx'),
] satisfies RouteConfig;
```

If we navigate to `/dashboard/ideas`, the `MyIdeasPage` component will be rendered, and the current user's ideas will be displayed.

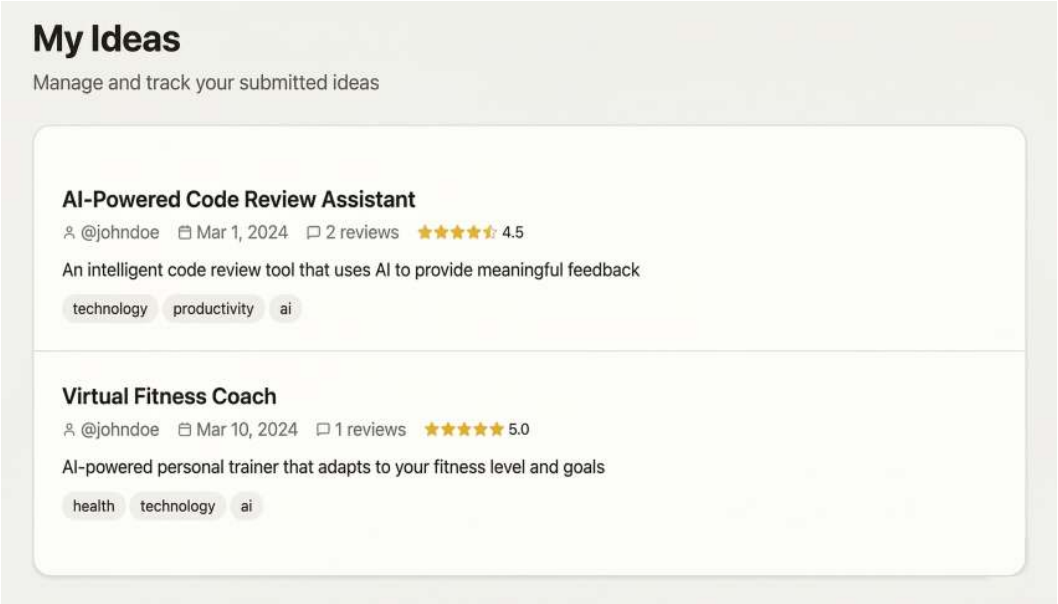


Figure 4.5 – Client-rendered dashboard ideas page

To verify what is shown initially, we can disable JavaScript in the browser and reload the page.

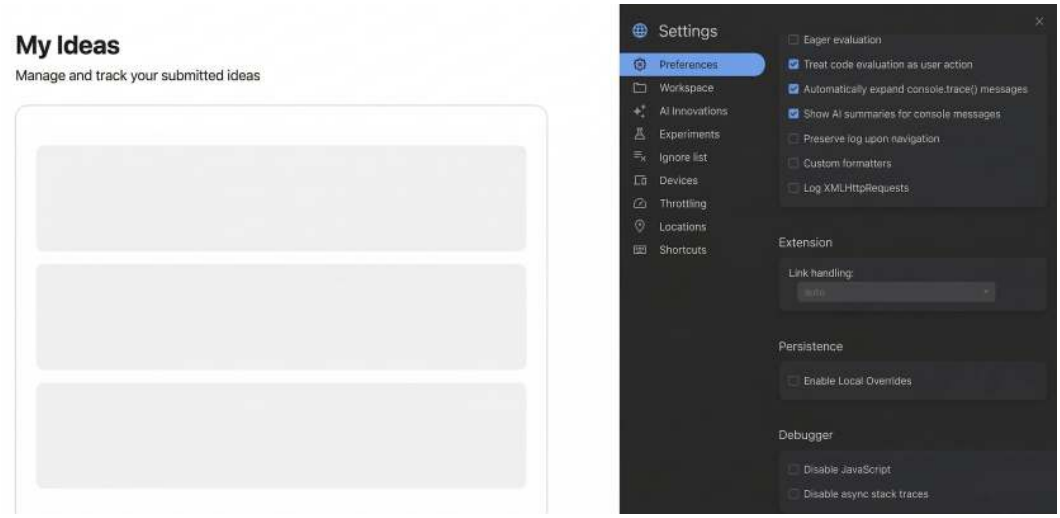


Figure 4.6 – Client-rendered dashboard ideas page without JavaScript

Initially, we only see the loading state, and then, once the data is fetched on the client side, the page content is updated.

Sometimes we need the best of both worlds: immediate content from the server plus additional client-side data fetching. That's where hybrid rendering comes in.

Hybrid rendering

Hybrid rendering is a combination of SSR and CSR. It allows us to have the best of both worlds: immediate content from the server plus additional client-side data fetching.

Here's how hybrid rendering works in our application:

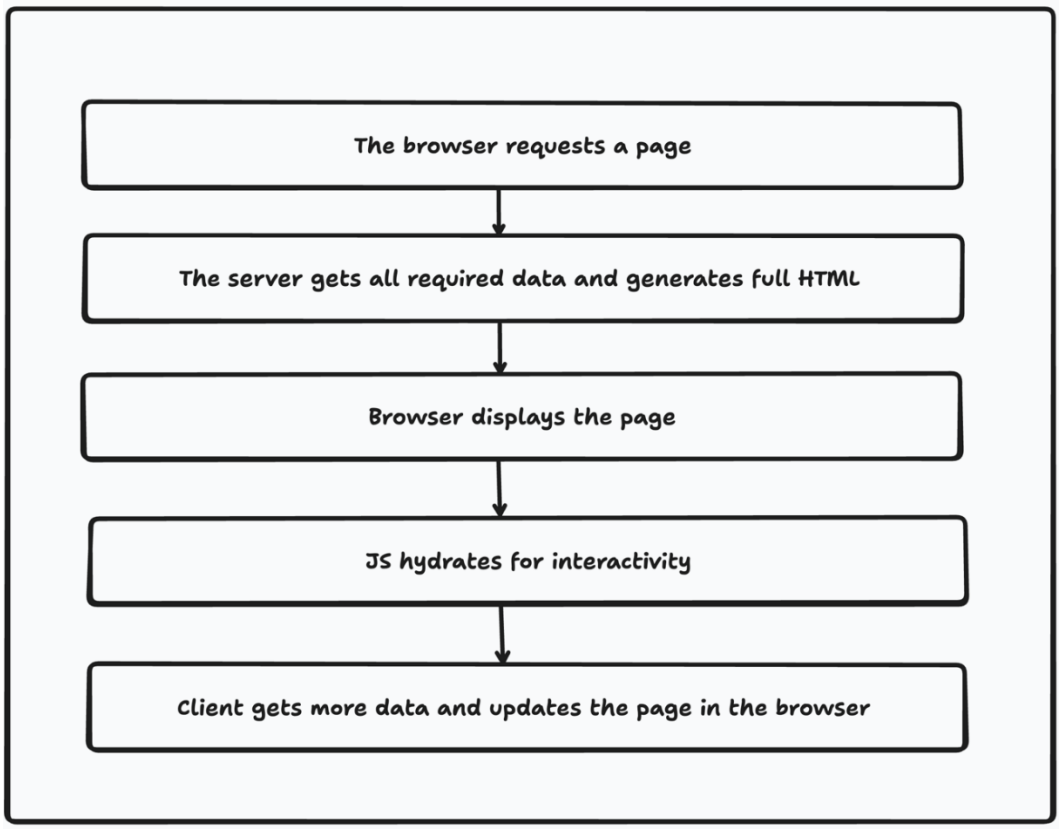


Figure 4.7 – Hybrid rendering flow

When a user visits `/profile/johndoe` , the server loader fetches profile data and renders HTML with complete profile details. The browser displays this immediately. After JavaScript hydrates, the page component is rendered and

the rest of the data is fetched on the client side, showing loading states until data arrives.

Hybrid rendering is ideal for pages that need both immediate critical content visibility and additional content that can be fetched on the client side. Here are the main use cases:

- Public pages that need SEO but also show personalized content
- Pages with critical content (server-rendered) and supplementary content (client-rendered)
- Balancing performance and functionality

Our profile page (`/profile/:username`) uses hybrid rendering for the following reasons:

- The profile information needs to be visible immediately for SEO
- The user's ideas and reviews can load progressively on the client

Let's see how this can be implemented in our application.

Creating a hybrid profile page

The profile page loads critical data on the server (profile info) and additional ideas and reviews data that can be fetched on the client side:

First, we need to create the `route` module for the profile page:

```
// src/app/routes/profile.tsx

import { data as routerData, Link } from 'react-router';

import { ErrorMessage } from '@/components/error-message';
import { Button } from '@/components/ui/button';
import { UserIdeas } from '@/features/ideas/components/user-ideas';
import { getProfileByUsername } from '@/features/profile/api/get-profile-by-username';
import { ProfileDetails } from '@/features/profile/components/profile-details';
import { UserReviews } from '@/features/reviews/components/user-reviews';

import type { Route } from './+types/profile';

export async function loader({ params }: Route.LoaderArgs) {
  const profile = await getProfileByUsername(params.username);

  return routerData({
    profile,
  });
}
```

```
export default function ProfilePage({
  params,
  loaderData,
}: Route.ComponentProps) {
  const profile = loaderData?.profile;

  return (
    <div className="container mx-auto px-4 py-8">
      <div className="max-w-4xl mx-auto">
        <ProfileDetails profile={profile} />
        <div className="mb-8">
          <UserIdeas username={params.username} />
        </div>
        <UserReviews username={params.username} />
      </div>
    </div>
  );
}
```

As we can see, the profile data is fetched in the `loader` function on the server side and then passed to the component via the `loaderData` prop.

Components responsible for fetching and rendering ideas and reviews are client-side components:

```
// src/features/ideas/components/user-ideas.tsx
import { useIdeasByUserQuery } from '../api/get-ideas-by-user';
import { IdeasList } from '@/features/ideas/components/ideas-list';

export function UserIdeas({ username }: { username: string }) {
  const ideasQuery = useIdeasByUserQuery({ username });
  const ideas = ideasQuery.data?.data;

  return (
    <div>
      <h2 className="text-xl font-semibold mb-6">
        Ideas by {username} ({ideas?.length || 0})
      </h2>
      <IdeasList
        ideas={ideas}
        isLoading={ideasQuery.isLoading}
        emptyMessage={`${username} hasn't shared any ideas yet.`}
        error={ideasQuery.error}
      />
    </div>
  );
}
```

Also, the `UserReviews` component:

```
// src/features/reviews/components/user-reviews.tsx
import { useReviewsByUserQuery } from '../api/get-reviews-by-user';
import { ReviewsList } from '@/features/reviews/components/reviews-list';

export function UserReviews({ username }: { username: string }) {
  const reviewsQuery = useReviewsByUserQuery({ username });
  const reviews = reviewsQuery.data?.data;

  return (
    <div>
      <h2 className="text-xl font-semibold mb-6">
        Reviews by {username} ({reviews?.length || 0})
      </h2>

      <ReviewsList
        reviews={reviews}
        isLoading={reviewsQuery.isLoading}
        showIdeaTitle
        emptyMessage={`${username} hasn't written any reviews yet.`}
        error={reviewsQuery.error}
      />
    </div>
  );
}
```

Both components fetch data on the client using a custom hook. They show loading states while fetching and update when data arrives. But that's fine, as we want to load the profile data as soon as possible and then load the ideas and reviews data progressively on the client side.

Now that we have the `route` module for this page, we need to register it in the `src/app/routes.ts` file.

```
// src/app/routes.ts

export default [
  // ...
  route('/profile/:username', './routes/profile.tsx'),
] satisfies RouteConfig;
```

If we navigate to `/profile/johndoe`, the `ProfilePage` component will be rendered, and the profile details will be displayed.

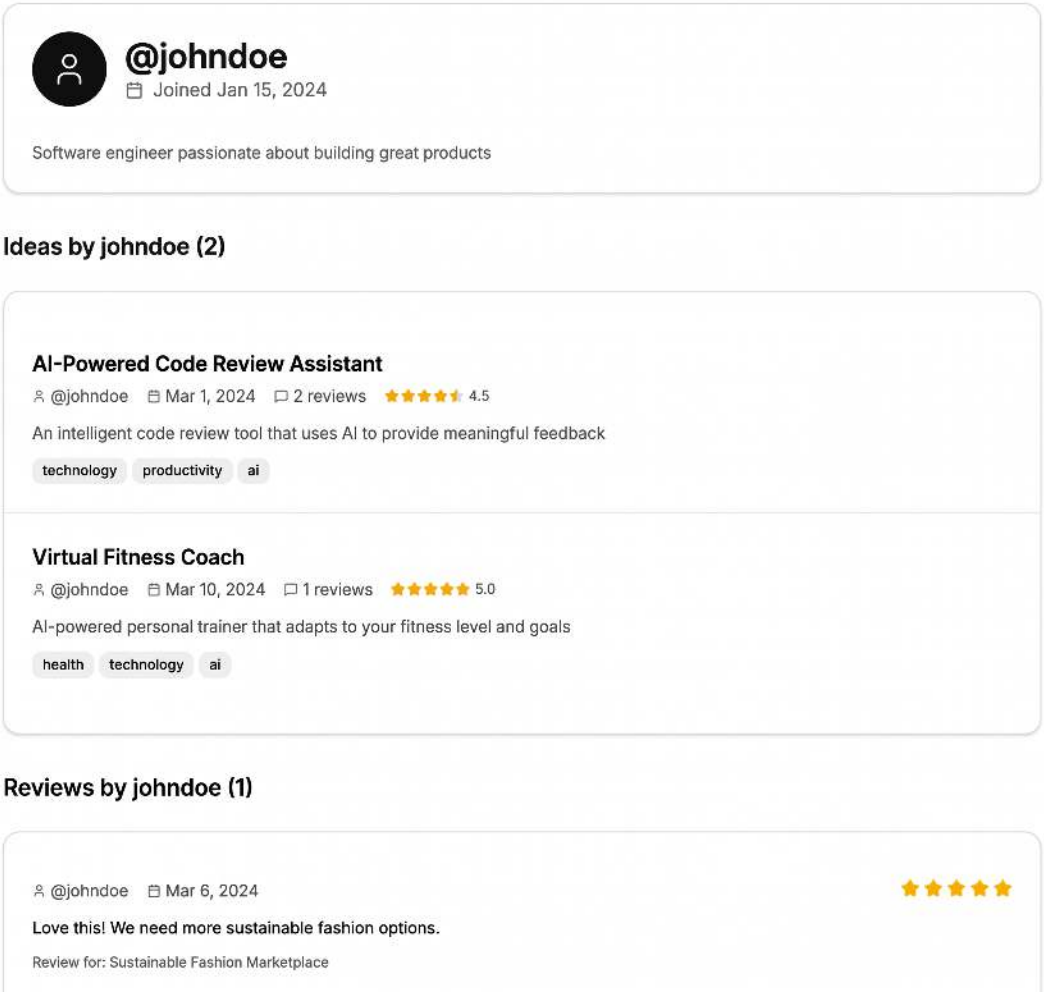


Figure 4.8 – Hybrid-rendered profile page

To verify what is shown initially, we can disable JavaScript in the browser and reload the page.

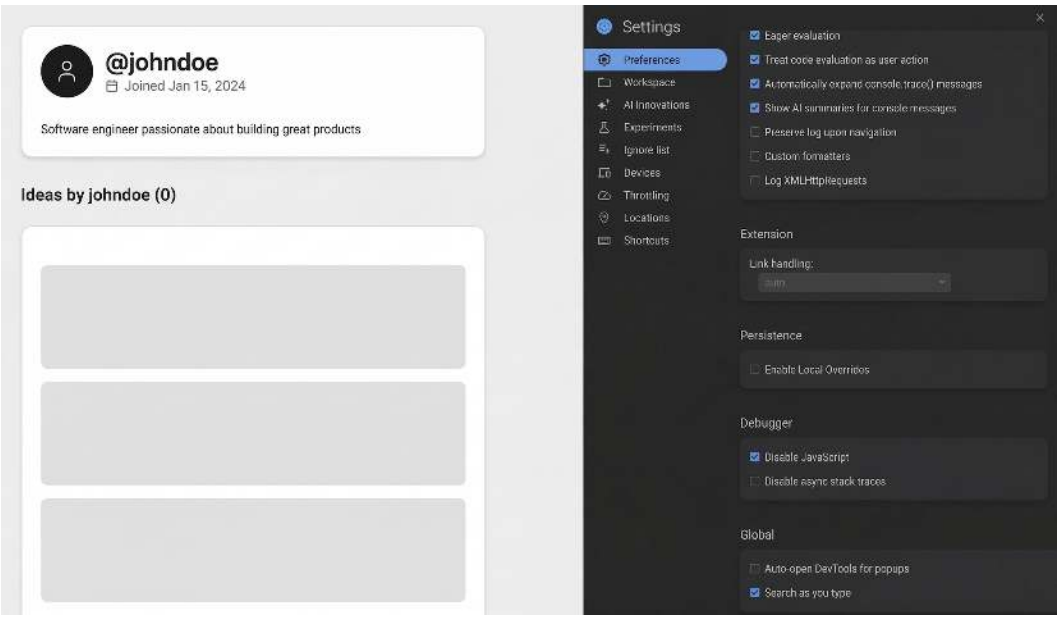


Figure 4.9 – Hybrid-rendered profile page without JavaScript

As we can see, the profile data is visible, but the ideas and reviews are not, so we can only see the loading states. Once the data is fetched on the client side, the page content is updated.

But what if we had pages that are static, so the page content should not be generated on every request? That's where static pre-rendering comes in.

Static pre-rendering

Static pre-rendered pages are pages that are generated at build time and served as HTML files. They load quickly, work without JavaScript, and are excellent for content that doesn't change frequently.

Let's see how static pre-rendering works visually:

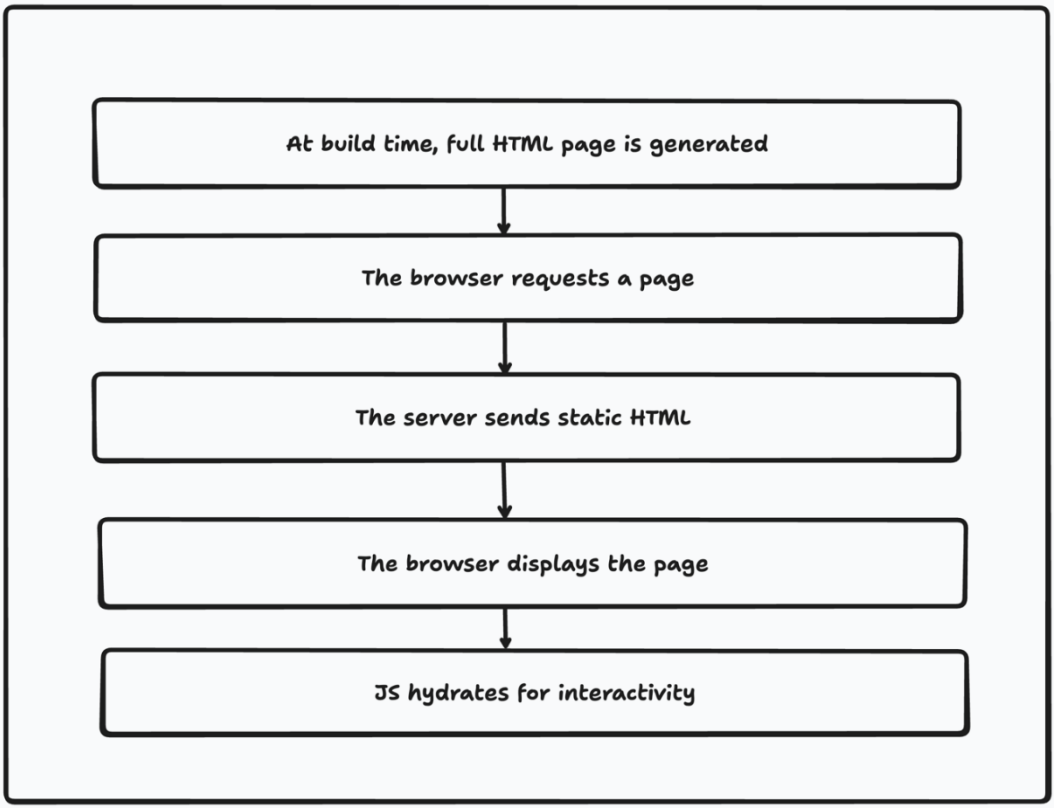


Figure 4.10 – Pre-rendering flow

At build time, we fetch the data (if needed) for the page and generate the HTML page content, which will be served as a static file. The browser will then load the HTML file and display the content.

Pre-rendering is ideal for pages with static or rarely changing content that benefits from fast load times and SEO. Here are the main use cases:

- Marketing pages and landing pages with static content
- Blog posts and documentation that don't change frequently
- Pages that don't require user-specific data
- Content that's the same for all visitors

Our home and about pages are perfect candidates for pre-rendering because they display static marketing content that's identical for all visitors.

Creating pre-rendered home and about pages

Our home and about pages are standard routes with a component that renders the UI:

```
// src/app/routes/home.tsx

export default function HomePage() {
  // ...
}
```

Also, the about page:

```
// src/app/routes/about.tsx

export default function AboutPage() {
  // ...
}
```

Then we need to register the routes in the `src/app/routes.ts` file:

```
// src/app/routes.ts

export default [
  // ...
  index('routes/home.tsx'),
  route('/about', './routes/about.tsx'),
] satisfies RouteConfig;
```

To enable pre-rendering, we configure React Router to generate HTML at build time in the `react-router.config.ts` file:

```
// react-router.config.ts
import type { Config } from '@react-router/dev/config';

export default {
  ssr: true,
  appDirectory: 'src/app',
  async prerender() {
    return ['/', '/about'];
  },
} satisfies Config;
```

We're telling React Router to pre-render the home and about pages at build time. This will generate the HTML files for the home and about pages and store them in the `build` directory, ready to be served as pre-rendered HTML files.

If we run the `npm run build` command, we will see that the HTML files for the home and about pages are generated in the `build` directory.

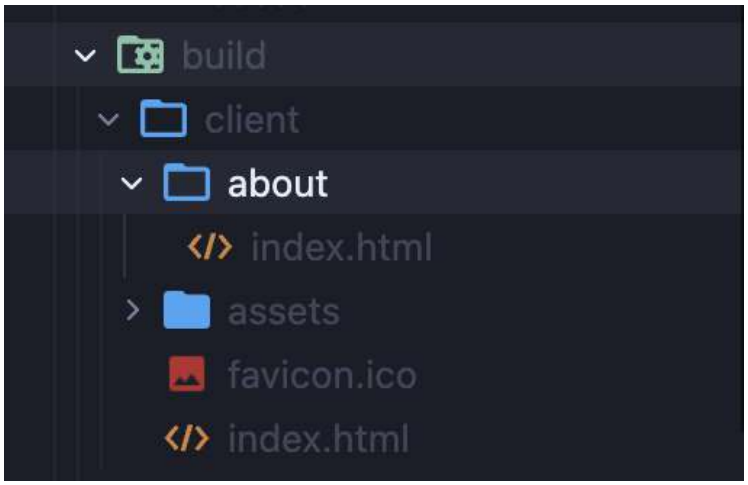


Figure 4.11 – Pre-rendered home and about page files

As we can see, the HTML files are generated in the `build` directory, and they are ready to be served as pre-rendered HTML files.

We have already mentioned that rendering page content on the server is important for SEO, but we still have one missing part related to SEO: the meta tags.

Adding meta tags to pages

React Router supports two ways of adding meta tags to pages:

- Exporting the `meta` function from the `route` module
- Using the `<meta>` tag

The `meta` function is a function that can be defined in the `route` module. It returns an array of meta tags that will be added to the page.

```
// src/app/routes/home.tsx

export function meta() {
  return [
    { title: 'AIdeas - Share and Discover AI Ideas' },
    { name: 'description', content: 'AIdeas - A community platform for sharing, reviewing, and
discovering innovative AI application ideas' },
  ];
}
```

Since React 19, it is recommended to use the built-in `<meta>` element provided by React instead of the `meta` function.

```tsx
// src/app/routes/home.tsx

export default function HomePage() {
  return (
    <div className="container mx-auto px-4 py-16">
      <title>AIdeas - Share and Discover AI Ideas</title>
      <meta name="description" content="AIdeas - A community platform for sharing, reviewing, and discovering innovative AI application ideas" />
      {/* ... */}
    </div>
  );
}
```

We can do this for every page. If that's the case, wouldn't it be better to create a reusable component for the meta tags?

Let's create the `Seo` component, which will be used to add `title` and `meta` tags to the page. To keep it simple, we will only add the `title` and the `description` meta tag to the page, but it can be extended to add more meta tags in the future.

```tsx
// src/components/seo.tsx

export function Seo({ title, description }: { title: string; description: string }) {
  return (
    <>
      <title>{title}</title>
      <meta name="description" content={description} />
    </>
  );
}
```

Now we can use this component in our pages:

```tsx
// src/app/routes/home.tsx

export default function HomePage() {
  return (
    <div className="container mx-auto px-4 py-16">
      <Seo title="AIdeas - Share and Discover AI Ideas" description="AIdeas - A community platform for sharing, reviewing, and discovering innovative AI application ideas" />
      {/* ... */}
    </div>
  );
}
```

This makes our pages much better optimized for SEO, which will help us to improve the search engine rankings of our application.

We are almost there. The final missing piece of our application pages is the navigation. We don't have a clear way to navigate between the pages. Since the navigation is a common component that is rendered on every page, it would make sense to make it visible on every page.

Adding page layouts

Currently, our application pages are working fine, but there is no navigation present, so there is no clear way to navigate from the home page to the dashboard. We already have the navigation component available at `src/components/navigation.tsx` . While we could add it to every page separately, it is better to add it to layouts because layouts avoid unnecessary rerenders when navigating between the pages.

Layouts are components that wrap the content of pages and provide a common structure around them. A common use case for layouts is navigation. We want to keep the navigation visible on every page to provide a consistent user experience. In our application, there are 2 levels of navigation:

- The main navigation is the one that is rendered on every page
- The dashboard navigation is the one that is rendered on the dashboard pages

To achieve this, we will create the layout components that will wrap the routes of the application.

We will first create the main layout component that will wrap the routes of the application:

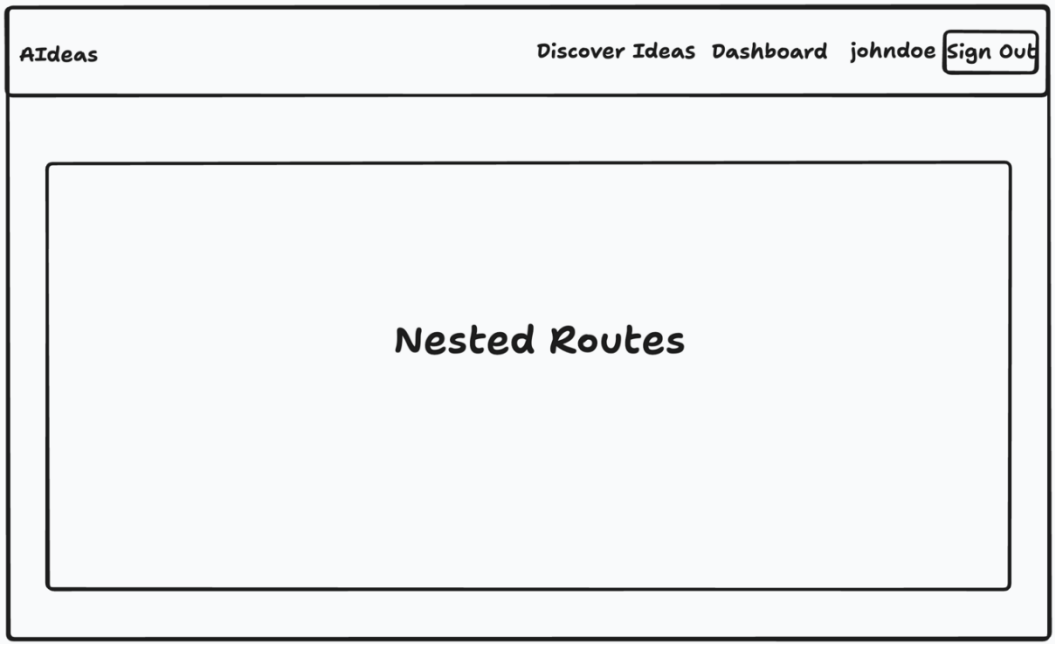


Figure 4.12 – Root layout structure

As we can see, the main layout should show the navigation at the top of the page and the content of the page below it.

In React Router, a layout component renders shared UI elements and includes an `<Outlet />` component where child routes are rendered. We can think of the `Outlet` as the placeholder for the routes the layout is wrapping. Here's our root layout:

```
// src/app/routes/layout.tsx

import { Outlet } from 'react-router';

import { Navigation } from '@/components/navigation';

export default function Layout() {
  return (
    <div>
      <Navigation />
      <main className="min-h-screen bg-background">
        <Outlet />
      </main>
    </div>
  );
}
```

As we can see, the layout component renders the navigation at the top of the page and the content of the page below it.

To make the layout wrap our pages, we need to register the layout in the `routes` configuration file:

```
// src/app/routes.ts

export default [
  layout('./routes/layout.tsx', [
    route('ideas/:id', './routes/ideas/idea.tsx'),
    route('ideas', './routes/ideas/ideas.tsx'),
    // ...
  ]),
] satisfies RouteConfig;
```

The layout wraps routes where the navigation should be visible. When a user visits the `/ideas/1` page, the layout will be rendered, and the navigation will be present at the top of the page.

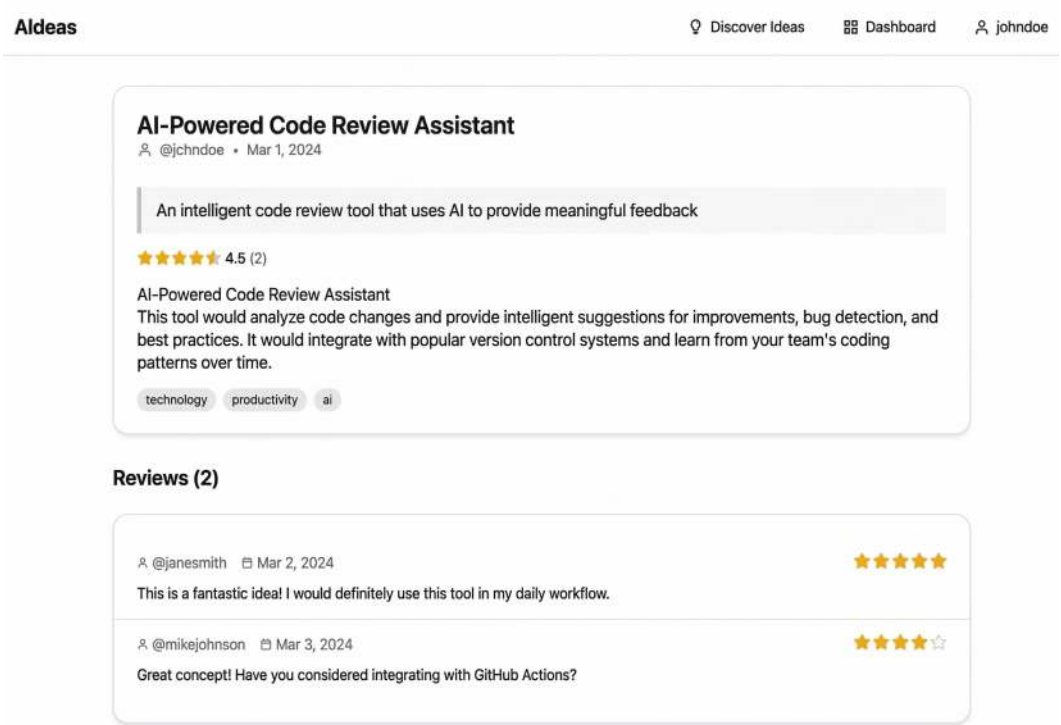


Figure 4.13 – Top-level navigation

Now we have a clear way to navigate between the home page, the ideas page, and the dashboard.

Our dashboard also needs navigation to navigate between the dashboard pages.

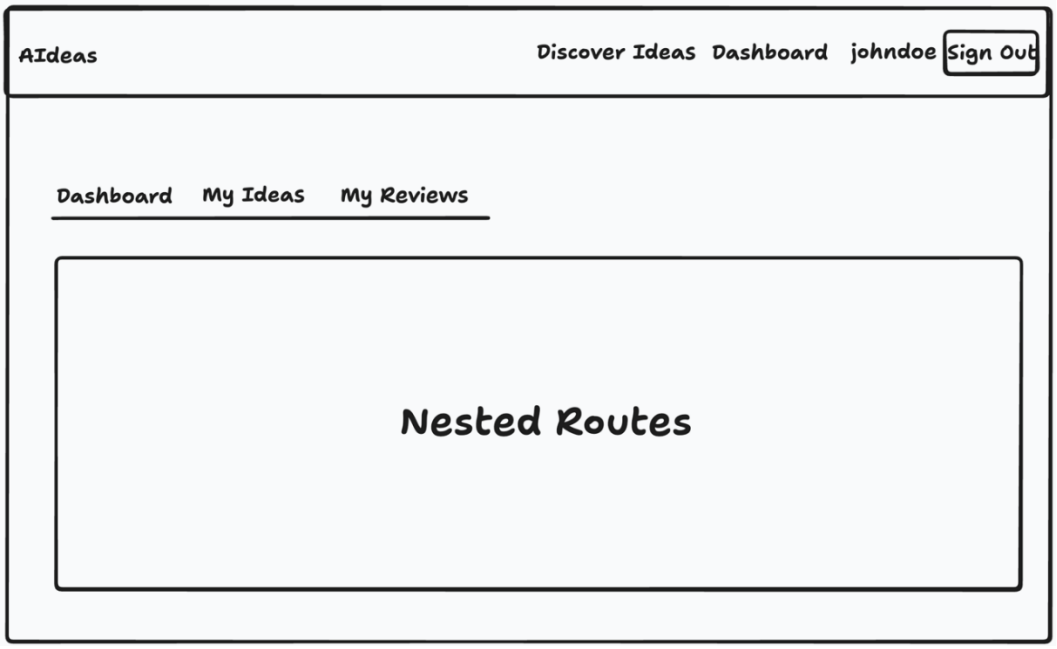


Figure 4.14 – Dashboard navigation

Let's create the dashboard layout component that will wrap the dashboard routes.

```
// src/app/routes/dashboard/layout.tsx

import { LayoutDashboard, Home, MessageSquare } from 'lucide-react';
import { NavLink, Outlet } from 'react-router';

import { cn } from '@/lib/utils';

export default function DashboardLayout() {
  // ...

  return (
    <div className="container mx-auto px-4 py-8">
      <div className="max-w-4xl mx-auto">
        <nav className="mb-6">
          {/* ... */}
        </nav>
        <Outlet />
      </div>
    </div>
  );
}
```

Now we need to register the dashboard layout in the `src/app/routes.ts` file:

```
// src/app/routes.ts

export default [
  layout('./routes/layout.tsx', [
    route('ideas/:id', './routes/ideas/idea.tsx'),
    route('ideas', './routes/ideas/ideas.tsx'),
    layout('./routes/dashboard/layout.tsx', [
      route('dashboard', './routes/dashboard/dashboard.tsx'),
      route('dashboard/ideas', './routes/dashboard/ideas/ideas.tsx'),
      route('dashboard/reviews', './routes/dashboard/reviews.tsx'),
    ]),
    // ...
  ]),
] satisfies RouteConfig;
```

Now we can navigate to `/dashboard` and see the dashboard page. Also, if we navigate to `/dashboard/ideas` or `/dashboard/reviews`, the dashboard navigation will be visible as well since it is a nested layout.

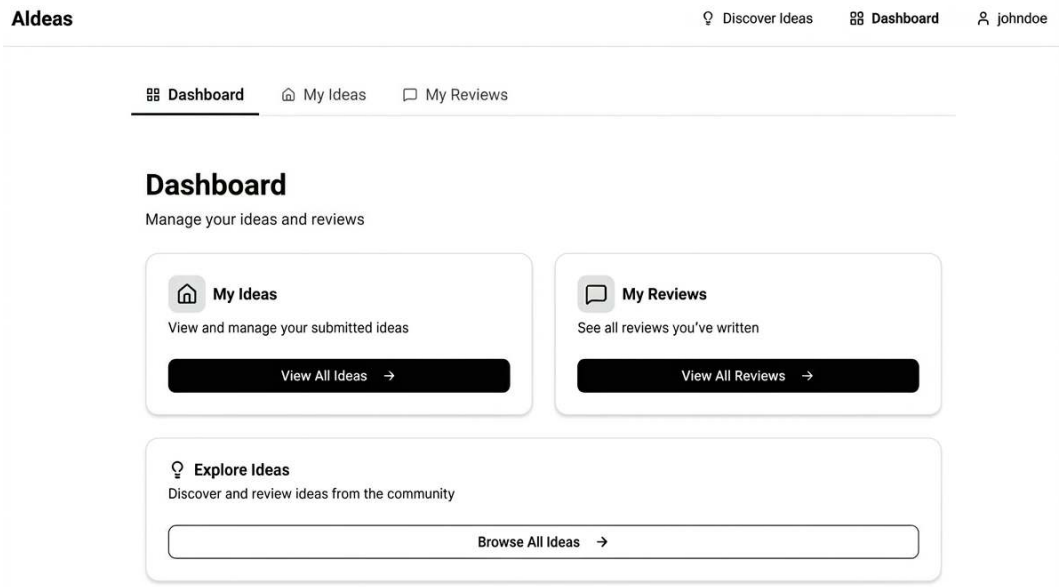


Figure 4.15 – Dashboard page

As we can see, the dashboard navigation is visible, as well as the top-level navigation.

Now we have a clear way to navigate between top-level and dashboard pages, which improves the user experience of our application.

The final route structure defined in the `src/app/routes.ts` file looks like this:

```
// src/app/routes.ts

import {
  type RouteConfig,
  index,
  layout,
  prefix,
  route,
} from '@react-router/dev/routes';

export default [
  layout('./routes/layout.tsx', [
    index('routes/home.tsx'),
    route('about', './routes/about.tsx'),
    ...prefix('dashboard', [
      layout('./routes/dashboard/layout.tsx', [
        index('./routes/dashboard/dashboard.tsx'),
        route('ideas', './routes/dashboard/ideas/ideas.tsx'),
        route('reviews', './routes/dashboard/reviews.tsx'),
      ]),
    ]),
    route('profile/:username', './routes/profile.tsx'),
    route('ideas/:id', './routes/ideas/idea.tsx'),
    route('ideas', './routes/ideas/ideas.tsx'),
    route('*', './routes/not-found.tsx'),
  ]),
] satisfies RouteConfig;
```

The configuration is quite straightforward, but there are a couple of things we didn't cover earlier:

- The `prefix` function is used to group routes under a common path prefix. All routes inside `prefix('dashboard', [...])` will start with `/dashboard`.
- The `index` function is used to define the default route for a path. For example, `index('routes/home.tsx')` renders the home page when the URL is `/`.
- The `*` route is used to define the catch-all route. It matches any URL that doesn't match a previous route and renders the not found page.

In case we have too many routes and the configuration becomes complex, we can compose them in the following way:

```
const dashboardRoutes = [/* ... */];
const profileRoutes = [/* ... */];
const ideasRoutes = [/* ... */];
```

```
const rootRoutes = [
  index('routes/home.tsx'),
  route('about', './routes/about.tsx'),
  ...dashboardRoutes,
  ...profileRoutes,
  ...ideasRoutes,
route('*', './routes/not-found.tsx'),
];
```

Now we have all the pages of the application configured and ready to be used.

Summary

In this chapter, we explored routing and rendering strategies in React Router, covering the essential architectural decisions for building modern React applications.

We started by learning how to configure routes using the `route` function in `src/app/routes.ts`, including dynamic routes with parameters. We then covered navigation between pages using the `Link` and `NavLink` components for declarative navigation, and the `useNavigate` hook for programmatic navigation.

We explored four rendering strategies, each suited for different use cases:

- **Server-side rendering** (**SSR**) generates page content on the server for each request. We used SSR for the idea detail page because it needs fresh data, good SEO, and immediate content visibility. This approach is ideal for public pages with dynamic content that changes frequently.
- **Client-side rendering** (**CSR**) loads minimal HTML and fetches data using JavaScript after the page loads. We used CSR for the dashboard pages because they're behind authentication, don't need SEO, and benefit from reduced server load. This approach is perfect for authenticated pages where SEO doesn't matter.
- **Hybrid rendering** (**SSR + CSR**) combines both strategies to get the best of both worlds. We used hybrid rendering for the profile page, loading critical profile data on the server for SEO while fetching ideas and reviews on the client for progressive enhancement. This approach works well for public pages that need both immediate content and additional client-side interactivity.

- Static pre-rendering generates HTML at build time for pages with static content. We configured the home and about pages to be pre-rendered using the `prerender` function in `react-router.config.ts`. This approach is ideal for marketing pages and content that doesn't change frequently.

We then added SEO optimization by creating a reusable `Seo` component that adds `title` and `meta` tags to pages, improving search engine rankings for our application.

Finally, we implemented page layouts to provide consistent navigation across the application. We created a root layout with top-level navigation and a dashboard layout with dashboard-specific navigation, using the `layout` function and `<Outlet />` component to render shared UI elements.

By understanding these routing and rendering patterns, we can make informed architectural decisions about how to build performant, SEO-friendly, and user-friendly React applications. The key is matching the rendering strategy to the specific requirements of each page: SEO needs, authentication requirements, data freshness, and user experience goals.

Get this book's PDF copy, code bundle, and more

Scan the QR code (or go to `packtpub.com/unlock`). Search for this book by name, confirm the edition, and then follow the steps on the page.

UNLOCK NOW

Note: Have your invoice handy. Purchases made directly from the Packt website don't require an invoice.

5

Communicating with the API

Every modern application needs to communicate with a backend API to fetch data, submit forms, and keep content up to date. While routing shows users the right page, the API layer provides the actual content for those pages. Setting this up with type safety, caching, and error handling makes our application easier to develop and maintain.

We'll cover the following topics:

- Creating an API client
- Generating TypeScript types and validation schemas from OpenAPI specification
- Setting up React Query
- Creating the API layer for the application
- Integrating with the application

By the end of this chapter, we'll have a complete API layer with automatic type safety, caching, and seamless integration with both server-side and client-side rendering.

Technical requirements

Before we get started, we need to set up our project. To develop our project, we need the following tools installed on our computer:

- Node.js version 24 or above. npm version 11 or above ships with Node. We can confirm this by executing `node -v` and `npm -v` in the terminal. There are multiple ways to install Node.js and npm. Here is a helpful article that goes into more detail:

https://www.nodejsdesignpatterns.com/blog/5-ways-to-install-node-js .

- VS Code (optional), a popular editor for JavaScript and TypeScript. It is open source, has solid TypeScript support, and offers many extensions. It can be downloaded from https://code.visualstudio.com .

The code for this book is available at the book's repo. To access the repository link, follow the steps in the " *Download the example code files* " section in the *Preface* . Clone it and enter the repository root:

```
git clone https://github.com/PacktPublishing/React-Application-Architecture-for-Production-Second-Edition.git
```

The repository contains chapter folders with the code for each chapter, along with a shared `api` folder that includes the API server used across all chapters.

We are working on *Chapter 5* , so navigate to the `chapter-05` directory:

```
cd React-Application-Architecture-for-Production-Second-Edition/chapter-05
```

Next, install the dependencies:

```
npm install
```

We also need to provide the environment variables:

```
cp .env.example .env
```

At this point, the frontend should be ready and running at http://localhost:5173 .

We also need to run the API server.

Open a new terminal window and navigate to the `api` directory:

```
cd React-Application-Architecture-for-Production-Second-Edition/api
```

Run the setup script for *Chapter 5* to configure everything:

```
npm run setup 05
```

Then start the API server:

```
npm run dev
```

The API server should now be running on `http://localhost:9999` .

For more information about the setup details, check out the `README.md` file.

Creating API client

Every application needs a way to communicate with its backend API. In modern browsers, we can use the `fetch` API to make HTTP requests, and while we could use `fetch` directly, that would mean defining base URLs, headers, error handling, and request formatting in every request. Instead of doing that, we'll build a centralized API client that defines these things once and provides a clean interface for it to be used throughout the application.

An API client is just a wrapper around the `fetch` API that provides a consistent interface for making HTTP requests. It will abstract away some repetitive things we need to do for every request such as setting the base URL, headers, and error handling.

Let's start by defining which parameters we need to pass to the API client:

```
// src/lib/api.ts

type RequestOptions<TBody = unknown> = {
  method?: 'GET' | 'POST' | 'PUT' | 'PATCH' | 'DELETE';
  headers?: Record<string, string>;
  body?: TBody;
  params?: Record<string, string | number | boolean | undefined | null>;
};
```

The wrapper will expect the URL of the request and the options object that contains the following parameters:

- The method of the request which can be one of the following: `GET` , `POST` , `PUT` , `PATCH` , `DELETE` .
- The headers of the request which can be a record of key-value pairs.
- The body of the request.
- The parameters of the request, which can be a record of key-value pairs. These are the query parameters (e.g. `?page=1&limit=10`).

Now let's see what our wrapper looks like:

```
// src/lib/api.ts
```

```
import { env } from '@/config/env';

async function fetchApi<T, TBody = unknown>(
  url: string,
  options: RequestOptions<TBody> = {},
): Promise<T> {
  const { method = 'GET', headers = {}, body, params } = options;

  const fullUrl = new URL(url, env.API_URL);
  if (params) {
    Object.entries(params).forEach(([key, value]) => {
      if (value != null) {
        fullUrl.searchParams.set(key, String(value));
      }
    });
  }

  const makeRequest = async (): Promise<Response> => {
    try {
      return await fetch(fullUrl, {
        method,
        headers: {
          Accept: 'application/json',
          ...(body ? { 'Content-Type': 'application/json' } : {}),
          ...headers,
        },
        body: body ? JSON.stringify(body) : undefined,
      });
    } catch (error) {
      if (error instanceof TypeError) {
        const networkError = new Error(
          'Network error. Please check your connection.',
        );

        throw networkError;
      }
      throw error;
    }
  };

  let response = await makeRequest();

  // ...
}
```

Sending the request is pretty straightforward. We use params (if provided) to form the full URL of the request by appending the query parameters to the base URL. We also set some default headers such as `Accept: application/json` and `Content-Type: application/json` if the body is

present. We also make sure the body is stringified before being sent to the API because the API server expects it to be a JSON object.

Now let's see how we handle the response:

```
// src/lib/api.ts

// ...

async function fetchApi<T, TBody = unknown>(
  url: string,
  options: RequestOptions<TBody> = {},
): Promise<T> {

  // ...

  let response = await makeRequest();

  if (!response.ok) {
    let message = response.statusText;
    try {
      const errorData = await response.json();
      message = errorData.message || message;
    } catch {
 // response body may not be JSON, use the statusText as the message instead
    }

    throw new Error(message);
  }

  try {
    return await response.json();
  } catch {
    throw new Error('Invalid response from server');
  }
}
```

If the response is not successful, we handle it properly. If everything is fine, we return the JSON response.

The `fetchApi` function is the core of our API client. It handles the actual HTTP request and response, error handling, etc, but we won't be using it directly in our application. Instead, we'll use a more convenient interface that is easier to use and understand.

```
// src/lib/api.ts

export const api = {
```

```
  get<T>(url: string, options?: RequestOptions): Promise<T> {
    return fetchApi<T>(url, { ...options, method: 'GET' });
  },
  post<T, TBody = unknown>(
    url: string,
    body?: TBody,
    options?: RequestOptions,
  ): Promise<T> {
    return fetchApi<T, TBody>(url, { ...options, method: 'POST', body });
  },
  put<T, TBody = unknown>(
    url: string,
    body?: TBody,
    options?: RequestOptions,
  ): Promise<T> {
    return fetchApi<T, TBody>(url, { ...options, method: 'PUT', body });
  },
  patch<T, TBody = unknown>(
    url: string,
    body?: TBody,
    options?: RequestOptions,
  ): Promise<T> {
    return fetchApi<T, TBody>(url, { ...options, method: 'PATCH', body });
  },
  delete<T>(url: string, options?: RequestOptions): Promise<T> {
    return fetchApi<T>(url, { ...options, method: 'DELETE' });
  },
};
```

This object provides a simple interface with methods for each HTTP verb: `get`, `post`, `put`, `patch`, and `delete`. Each method wraps the `fetchApi` function and passes the appropriate HTTP method. This makes our API calls cleaner and more readable throughout the application.

Now if we want to use the API client in our application, we can do it like this:

```
import { api } from '@/lib/api';

const idea = await api.get<Idea>('/ideas/1');

const newIdea = await api.post<Idea>('/ideas', { title: 'New Idea' });
```

What's great with this approach is that we can control which parameters we pass to `fetchApi` for each method. For example, there is not much point in passing the body to the `get` method since it shouldn't have a body. This way, every method is type-safe and we can't pass the wrong parameters.

With our API client in place, we need to ensure the data we send and receive matches what the backend expects. Let's see how we can generate TypeScript types from the API specification to keep everything in sync.

Generating TypeScript types and validation schemas from OpenAPI specifications

When our data comes from the API, there is no way for TypeScript to know what the data is. While we could manually define types for what gets returned from the API, there is no guarantee that the data is always correct. For example, if a property is added or removed from an API response object, our frontend application would not know about that which could break the application.

To make our API calls type-safe, we need to provide proper types for the request and response data.

What is OpenAPI and why generate types?

OpenAPI is a standard format for describing REST APIs. Our backend provides an OpenAPI specification at the `${API_URL}/doc` endpoint that lists every endpoint, the data it accepts, and what it returns.

Instead of manually writing and maintaining types for every endpoint, we generate them from the OpenAPI specification. This keeps our frontend types in sync with the backend automatically. When the API changes, we regenerate the types and TypeScript will tell us exactly what needs to be updated.

Instead of manually writing and maintaining types for every endpoint, we generate them from the OpenAPI specification. This gives us a contract between the backend and the frontend that keeps our frontend types in sync with the backend automatically. When the API changes, we regenerate the types and TypeScript will tell us exactly what needs to be updated.

This contract is only as reliable as the spec itself, though. It is important that the backend keeps the OpenAPI spec accurate and up to date, otherwise the generated types will be misleading.

Configuring type generation

To generate types from the OpenAPI specification, we use the `@hey-api/openapi-ts` package. First, let's set up the configuration for the generator tool in `openapi-ts.config.ts` :

```
// openapi-ts.config.ts

export default {
  input: `${process.env.VITE_API_URL}/doc`,
  output: {
    format: 'prettier', // Run prettier to format the generated code.
    path: './src/types/generated', // Output directory for the generated types.
  },
  plugins: [
    {
      name: '@hey-api/typescript', // Generate TypeScript types.
      exportFromIndex: false,
    },
    'zod', // We can also generate runtime validation schemas with Zod.
  ],
};
```

To run the generator, we can use the following command:

```
npm run generate:openapi
```

It is defined in the `package.json` file as follows:

```
// package.json
"generate:openapi": "dotenv -e .env -- openapi-ts"
```

Once we run the command, we can find TypeScript types generated in the `src/types/generated/types.gen.ts` file:

```
// src/types/generated/types.gen.ts

// ...

export type User = {
  id: string;
  email: string;
  username: string;
  bio: string;
  createdAt: string;
  updatedAt: string;
};

export type Idea = {
```

```
  id: string;
  title: string;
  shortDescription: string;
  description: string;
  tags: Array<string>;
  authorId: string;
  author: UserSummary;
  reviewsCount: number;
  avgRating: number | null;
  createdAt: string;
  updatedAt: string;
};


export type Review = {
  id: string;
  content: string;
  rating: number;
  authorId: string;
  author: UserSummary;
  ideaId: string;
  idea?: IdeaSummary;
  createdAt: string;
  updatedAt: string;
};

// ...
```

These generated types describe the data structures our API returns. The generator creates a type for each request body, response, and data model defined in the OpenAPI specification.

In addition to TypeScript types, we can also generate validation schemas with Zod that can be used to validate the data at runtime.

We can find the Zod schemas generated in the `src/types/generated/zod.gen.ts` file:

```
// src/types/generated/zod.gen.ts

// ...

export const zUser = z.object({
  id: z.string(),
  email: z.string(),
  username: z.string(),
  bio: z.string(),
  createdAt: z.string(),
  updatedAt: z.string(),
```

```
});

export const zReview = z.object({
  id: z.string(),
  content: z.string(),
  rating: z.number(),
  authorId: z.string(),
  author: zUserSummary,
  ideaId: z.string(),
  idea: z.optional(zIdeaSummary),
  createdAt: z.string(),
  updatedAt: z.string(),
});

// ...
```

These schemas mirror the TypeScript types but provide runtime validation. We can use them to verify that the data we receive from the API matches what we expect, catching issues that TypeScript alone cannot detect at runtime.

They can be used to validate the data at runtime like this:

```
const idea = zIdea.parse(idea);
```

If the data is not valid, the Zod schema will throw a validation error. We don't need to worry about this yet; we will see how these validation schemas are used when we start building the API layer and forms.

NOTE: We still need to remember to run the generator command manually to generate the types since the types could still go out of sync if the API changes and we forget to regenerate the types. We could solve this with the help of a CI/CD pipeline. We will cover CI/CD in the future chapters, but here is a brief overview of how we can solve this problem:

If the frontend and backend apps are in the same repository, we can configure our CI/CD pipeline to automatically generate types whenever the API specification changes. The typecheck step in CI will then verify that the frontend code is compatible with the new types. If the API changes break existing frontend code, the typecheck will fail, preventing the merge and alerting us to fix the breaking changes. We can visualize this process like this:

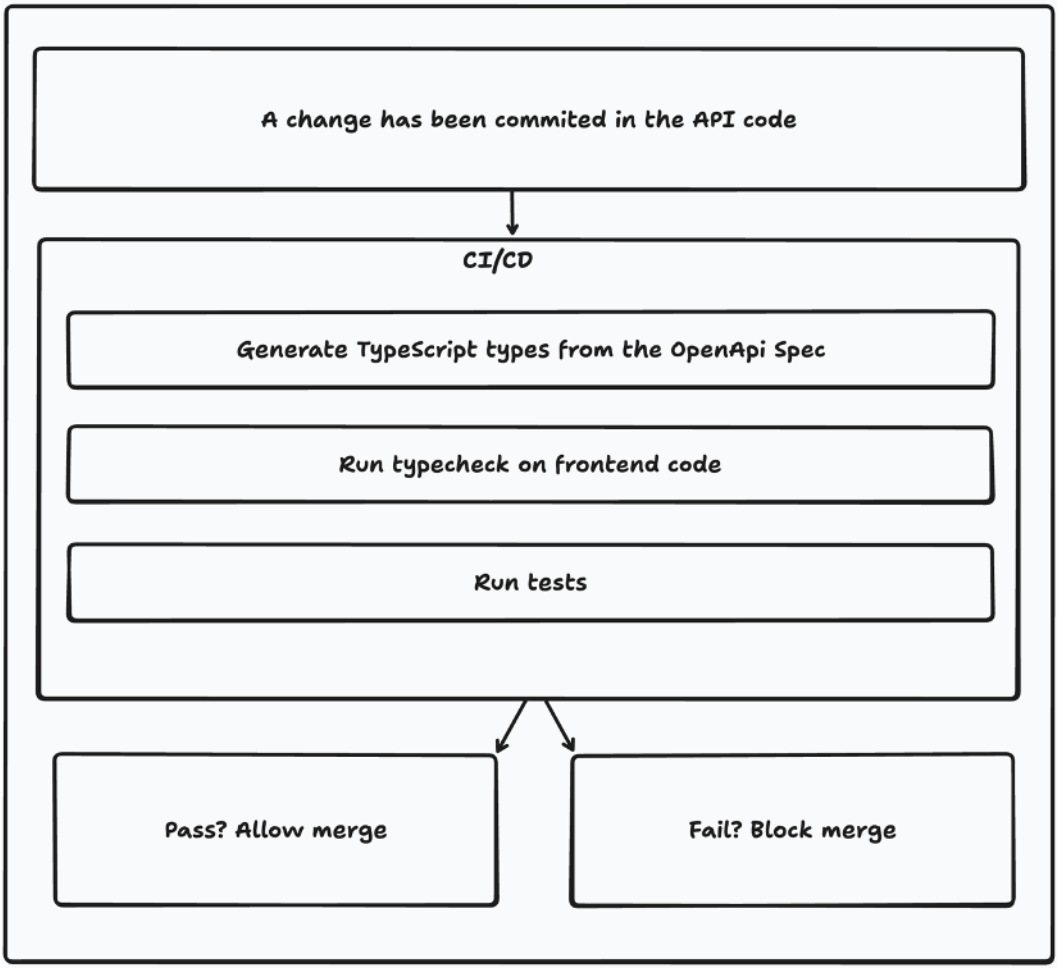


Figure 5.1 – Flow when backend and frontend are in the same repository

If they are in different repositories, we can configure a CI/CD pipeline in the backend repo to trigger CI/CD in the frontend repository whenever the API specification changes. This workflow would generate the updated TypeScript types and create a pull request. The frontend's CI pipeline will then run typechecking and tests on this PR, catching any breaking changes. We can review the PR to see the changes and fix any breaking changes before merging. Here is how that would look like:

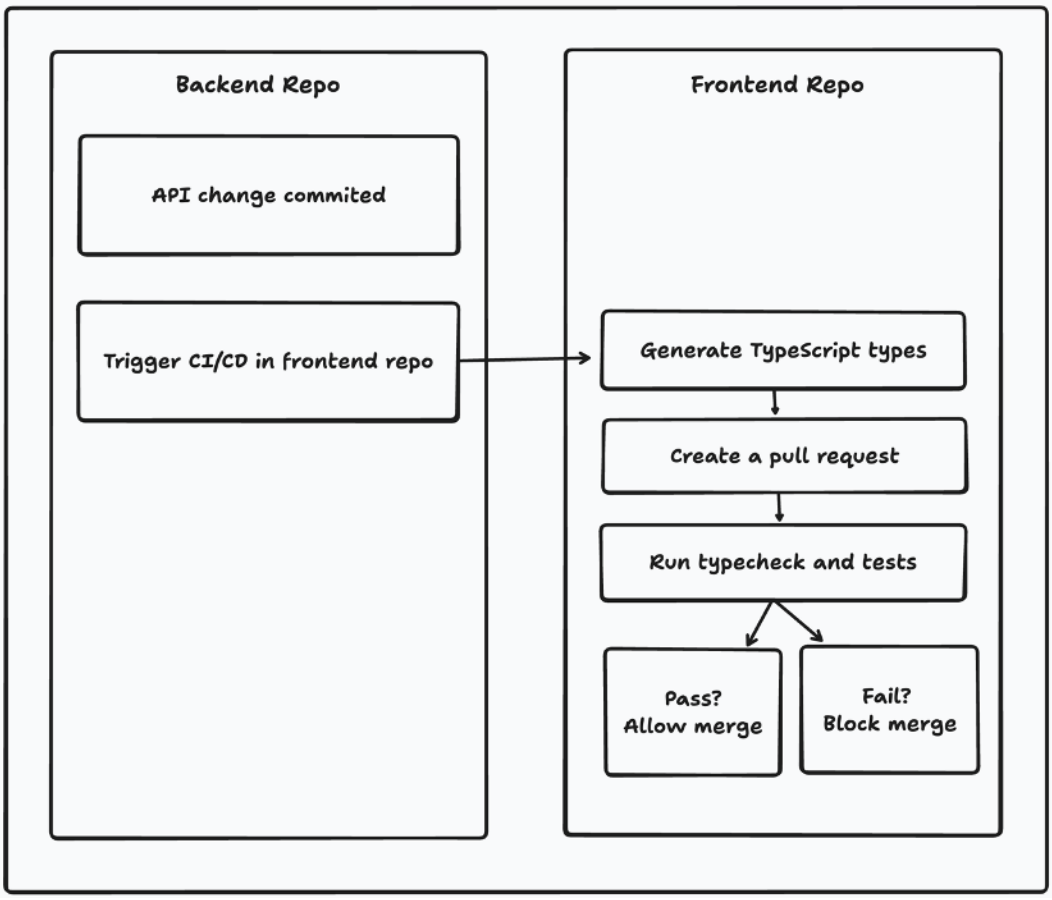


Figure 5.2 – Flow when backend and frontend are in different repositories

With our types and validation schemas generated and always in sync with the backend, we have a good foundation to build type-safe and validated API calls. Next, we need to set up React Query to manage how we fetch and cache this data.

Setting up React Query

When making API calls in React applications, we have a lot of things to handle such as loading states, error handling, caching, request deduplication, and more.

This is where React Query comes in to simplify the process of handling server state.

Why React Query?

React Query is a library that manages asynchronous state in React applications. Without it, we would need to manually track loading states,

handle errors, implement caching, prevent duplicate requests, and decide when to refetch data. React Query handles all of this for us automatically, letting us focus on building features instead of reinventing the wheel with our own solution.

Configuring React Query for the application

React Query needs a `QueryClient` instance to manage queries. We'll create a function that configures the client with sensible defaults for our application:

```
// src/lib/react-query.ts

import { QueryClient, type QueryClientConfig } from '@tanstack/react-query';

export function createQueryClient(config: Partial<QueryClientConfig> = {}) {
  return new QueryClient({
    defaultOptions: {
      queries: {
        staleTime: 60 * 1000,
      },
    },
    ...config,
  });
}
```

We set a stale time of 60 seconds, meaning data stays fresh for one minute before React Query considers refetching it.

Now we need to make the QueryClient available to our entire application by wrapping our routes in the `QueryClientProvider` :

```
// src/app/root.tsx

import { QueryClientProvider } from '@tanstack/react-query';
import { useState } from 'react';
import { Outlet } from 'react-router';

import { createQueryClient } from '@/lib/react-query';

export default function App() {
  const [queryClient] = useState(() => createQueryClient());
  return (
    <QueryClientProvider client={queryClient}>
      <Outlet />
    </QueryClientProvider>
  );
}
```

We create the QueryClient instance once using the `useState` hook with a lazy initializer. This ensures the client is created only once and persists for the lifetime of the application. The `QueryClientProvider` makes the client available within the application via the `useQueryClient` hook.

With React Query configured, our application now has the infrastructure to handle server state efficiently. The next step is to build our API layer, the functions and hooks that will fetch and mutate data throughout our application.

Creating API layer for the application

Now that we have our API client and React Query set up, we can build the actual API layer that our application will use.

Before we start building the API layer, we need to organize the query keys.

Organizing query keys

React Query uses query keys to identify and cache queries. Each unique key represents a unique piece of data in the cache.

We could define these keys inline in each query, but organizing them in one place makes them easier to manage and reuse. Let's look at the query keys for the ideas feature:

```
// src/features/ideas/config/query-keys.ts

import type { GetAllIdeasData } from '@/types/generated/types.gen';

export const ideasQueryKeys = {
  all: ['ideas'] as const,
  lists: () => [...ideasQueryKeys.all, 'list'] as const,
  list: (params?: GetAllIdeasData['query']) =>
    [...ideasQueryKeys.lists(), params] as const,
  details: () => [...ideasQueryKeys.all, 'detail'] as const,
  detail: (id: string) => [...ideasQueryKeys.details(), id] as const,
  current: () => [...ideasQueryKeys.all, 'current'] as const,
  byUser: (username: string) =>
    [...ideasQueryKeys.all, 'user', username] as const,
  tags: () => [...ideasQueryKeys.all, 'tags'] as const,
} as const;
```

This structure follows a pattern where we start from a root key (`ideas`) and compose more specific keys for different types of queries. For example,

`detail(id)` creates a key like `['ideas', 'detail', '123']`. This makes it easy to invalidate related queries, we can invalidate all idea lists by targeting `ideasQueryKeys.lists()`, or just a specific detail by targeting `ideasQueryKeys.detail(id)`.

We are now ready to define our queries.

Defining queries

Each query follows the same pattern: a fetcher function that makes the API call, query options that configure the query, and a custom hook that makes it easy to use. Let's see this pattern in action by fetching an idea by its ID:

```
// src/features/ideas/api/get-idea-by-id.ts
import { queryOptions, useQuery } from '@tanstack/react-query';
import { api } from '@/lib/api';
import type { GetIdeaByIdResponse } from '@/types/generated/types.gen';
import {
  zGetIdeaByIdData,
  zGetIdeaByIdResponse,
} from '@/types/generated/zod.gen';
import { ideasQueryKeys } from '../config/query-keys';

// Fetcher function for making API call to fetch an idea by its ID
export async function getIdeaById(id: string): Promise<GetIdeaByIdResponse> {
  // Validate input data using the Zod schema
  // In this example, we need to validate the path parameter `id`
  // If it is not provided, the Zod schema will throw a validation error.
  const validatedData = zGetIdeaByIdData.parse({ path: { id } });

  // Make the API request
  const response = await api.get<GetIdeaByIdResponse>(
    `/ideas/${validatedData.path.id}`,
  );

  // Validate response data using the Zod schema
  // In this example, we need to validate the response data
  return zGetIdeaByIdResponse.parse(response);
}

// Query options factory for the query
export function getIdeaByIdQueryOptions(id: string) {
  return queryOptions({
    queryKey: ideasQueryKeys.detail(id), // Use the query key from the query keys
    queryFn: () => getIdeaById(id), // Use the fetcher function to fetch the data
  });
}
```

```
// Custom hook for fetching an idea by ID that returns the query
export function useIdeaByIdQuery({
  id,
  options,
}: {
  id: string;
  options?: Omit<
    ReturnType<typeof getIdeaByIdQueryOptions>,
    'queryKey' | 'queryFn'
  >;
}) {
  return useQuery({
    ...getIdeaByIdQueryOptions(id), // Using the query options factory to create the query
    ...options, // Allowing to override the query options
  });
}
```

This pattern gives us three things: a fetcher function that handles the API call and validation, a query options factory that configures the query, and a custom hook that allows us to use the query in components. The options parameter allows components to override specific settings like enabling or disabling the query.

Now let's see how mutations work for operations that modify data on the server.

Defining mutations

Mutations are for operations that modify data on the server. They follow the same pattern as queries: a fetcher function, mutation options, and a custom hook. Here's how we create a new idea:

```
// src/features/ideas/api/create-idea.ts

import {
  mutationOptions,
  useMutation,
  type UseMutationOptions,
} from '@tanstack/react-query';

import { api } from '@/lib/api';
import type {
  CreateIdeaData,
  CreateIdeaResponse,
} from '@/types/generated/types.gen';
import {
  zCreateIdeaData,
```

```
  zCreateIdeaResponse,
} from '@/types/generated/zod.gen';

// Fetcher function for making API call to create an idea
export async function createIdea(
  data: CreateIdeaData['body'],
): Promise<CreateIdeaResponse> {
  // Validate input data using the Zod schema
  // In this example, we need to validate the body of the request
  // If it is not provided, the Zod schema will throw a validation error.
  const validatedData = zCreateIdeaData.parse({ body: data });

  // Make the API request
  const response = await api.post<CreateIdeaResponse>(
    '/ideas',
    validatedData.body,
  );

  // Validate response data using the Zod schema
  // In this example, we need to validate the response data
  return zCreateIdeaResponse.parse(response);
}

// Mutation options factory for the mutation
export function getCreateIdeaMutationOptions() {
  return mutationOptions({
    mutationFn: createIdea,
  });
}

// Custom hook for creating an idea that returns the mutation
export function useCreateIdeaMutation({
  options,
}: {
  options?: Omit<
    UseMutationOptions<CreateIdeaResponse, Error, CreateIdeaData['body']>,
    'mutationFn'
  >;
}) {
  return useMutation({
    ...getCreateIdeaMutationOptions(), // Using the mutation options factory to create the mutation
    ...options, // Allowing to override the mutation options
  });
}
```

The mutation pattern follows the same structure as queries: a fetcher function, mutation options factory, and a custom hook. The key difference is that mutations are for operations that change data, like creating, updating, or deleting resources. Unlike queries, mutations don't run automatically. We

trigger them by calling the `mutate` method, which we will see in action in a moment.

Now that we have our API layer defined, we can integrate it into our application.

Integrating with the application

We've built all the pieces: the API client, generated types, React Query configuration, and our API layer. Now it's time to put it all together and see how these pieces work in different rendering scenarios.

Queries

We can use our API layer on the client or the server side. Let's explore how.

Client-side usage

The simplest way to use React Query is on the client side. We call a custom hook in our component, and React Query fetches the data, manages the cache, and provides loading and error states. Here's how we fetch the current user's ideas:

```
// src/app/routes/dashboard/ideas/ideas.tsx

import { useCurrentUserIdeasQuery } from '@/features/ideas/api/get-current-user-ideas';
import { IdeasList } from '@/features/ideas/components/ideas-list';


export default function MyIdeasPage() {
  const ideasQuery = useCurrentUserIdeasQuery(); // Use the custom hook to use the query
  const ideas = ideasQuery.data?.data;

  return (
    <div className="container mx-auto px-4 py-8">
      <div className="max-w-4xl mx-auto space-y-6">
        {/* More UI code here */}
        <IdeasList
          ideas={ideas}
          isLoading={ideasQuery.isLoading} // Show loading state while the query is loading
          emptyMessage="You haven't created any ideas yet."
          error={ideasQuery.error} // Show error state if the query fails
        />
      </div>
    </div>
  );
}
```

The hook returns a query object that contains `data` , `isLoading` , and `error` properties. React Query handles all the complexity, fetches the data, manages the loading state, caches the result, and handles errors. We just need to display the data in our component.

This works great for client-side data fetching. But what if we need to fetch data on the server and send it to the client for better performance and SEO?

Server-side usage

For server-side rendering, we fetch data in the loader function and pass it to the query as initial data. This gives us the benefits of SSR while still using React Query. Here's how we fetch an idea and its reviews on the server:

```
// src/app/routes/ideas/idea.tsx

import {
  getReviewsByIdea,
  useReviewsByIdeaQuery,
} from '@/features/reviews/api/get-reviews-by-idea';


import type { Route } from './+types/idea';

// We are using the loader function to fetch the data on the server side
export async function loader({ params }: Route.LoaderArgs) {
  const [idea, reviews] = await Promise.all([
    getIdeaById(params.id),
    getReviewsByIdea(params.id),
  ]);
  return routerData({
    idea,
    reviews,
    error: null,
    meta: {
      title: idea.title,
      description: idea.shortDescription,
    },
  });
}

export default function IdeaDetailPage({
  params,
  loaderData, // We are receiving the data from the loader function
}: Route.ComponentProps) {
  const ideaId = params.id as string;
  const user = useUser();
```

```
  const ideaQuery = useIdeaByIdQuery({
    id: ideaId,
    options: {
      // Use the data from the loader function as initial data
      ...(loaderData?.idea && { initialData: loaderData.idea }),
    },
  });
  const idea = ideaQuery?.data;

  const reviewsQuery = useReviewsByIdeaQuery({
    id: ideaId,
    options: {
      // Use the data from the loader function as initial data
      ...(loaderData?.reviews && { initialData: loaderData.reviews }),
    },
  });

  // more code here...
}

export function ErrorBoundary({ error }: { error: Error }) {
  // ...
}
```

In this approach, we fetch the data in the loader function and pass it as `initialData` to the query. This gives us server-side rendering while still using React Query for all its benefits. The query uses the initial data immediately, then React Query takes over for any refetching or updates. If any of the queries fail, we can handle the errors in the `ErrorBoundary` component.

This works well when the query is used directly in the page component. But what if a component deep in the tree needs the data? We could pass it down through props, but that leads to prop drilling. There's a better way.

Server-side usage via HydrationBoundary

React Query provides a `HydrationBoundary` component that solves the prop drilling problem. Instead of passing data as props, we prefetch queries on the server, dehydrate the cache, and rehydrate it on the client. Any component can then use the queries and get the prefetched data.

Here's how it works:

```
// src/app/routes/ideas/ideas.tsx
```

```tsx
import { dehydrate, HydrationBoundary } from '@tanstack/react-query';

import {
  getIdeasQueryOptions,
  useIdeasQuery,
} from '@/features/ideas/api/get-ideas';
import { createQueryClient } from '@/lib/react-query';

import type { Route } from './+types/ideas';

// We are using the loader function to prefetch the query on the server side
// and return the dehydrated state to the component
export async function loader() {
  const queryClient = createQueryClient();

  // But we are not fetching the data directly.
  // Instead, we are prefetching the query using the query options factory.
  // This will populate the cache with the data from the server side.
  await queryClient.prefetchQuery(getIdeasQueryOptions());

  return routerData({
    meta: {
      title: 'AIdeas - Discover AI Ideas',
      description:
        'AIdeas - Browse and explore innovative AI application ideas from the community',
    },
    // Dehydrate the query client state which will be used to hydrate the component on the client
side
    dehydratedState: dehydrate(queryClient),
  });
}

// Component that uses the hydrated data
export default function IdeasPage({ loaderData }: Route.ComponentProps) {
  return (
    // Wrap the component in the HydrationBoundary component to hydrate the component on the client
side
    <HydrationBoundary state={loaderData.dehydratedState}>
      <Ideas />
    </HydrationBoundary>
  );
}


// This component could be rendered anywhere in the application
// Queries used here will always have access to the initial data
// from the server side because of dehydrated state
function Ideas() {
  const ideasQuery = useIdeasQuery({});
```

```
  const allIdeas = ideasQuery.data?.data || [];

  // more code here...
}
export function ErrorBoundary({ error }: { error: Error }) {
  // ...
}
```

Here's how this works:

1. The loader function prefetches the query on the server using `queryClient.prefetchQuery()`. This populates the React Query cache with data from the server.
2. We dehydrate the query client state using the `dehydrate` function. This serializes the cache into a format that can be sent to the client.
3. The page component wraps its content in a `HydrationBoundary` component, passing the dehydrated state.
4. On the client, the `HydrationBoundary` rehydrates the cache with the server data. Any component that uses the same query will immediately have access to that data.

This approach is powerful because components anywhere in the tree can use the query without prop drilling. The `Ideas` component doesn't need to receive the data as props—it just uses the query hook and gets the server-prefetched data from the cache.

Now let's see how mutations work in our application.

Mutations

Now let's see how we can use the `useCreateIdeaMutation` hook to create a new idea:

```
// src/app/routes/dashboard/ideas/new.tsx

import { useQueryClient } from '@tanstack/react-query';
import { useNavigate } from 'react-router';

import { ErrorMessage } from '@/components/error-message';
import { useCreateIdeaMutation } from '@/features/ideas/api/create-idea';
import { IdeaForm } from '@/features/ideas/components/idea-form';
import { ideasQueryKeys } from '@/features/ideas/config/query-keys';

export default function NewIdeaPage() {
```

```
  const navigate = useNavigate();
  const queryClient = useQueryClient();

  // We are using the mutation hook to get the mutation object:
  const createIdeaMutation = useCreateIdeaMutation({
    options: {
      onSuccess: () => {
        // We are using the query keys to invalidate the queries and refresh the data
        queryClient.invalidateQueries({ queryKey: ideasQueryKeys.lists() });
        queryClient.invalidateQueries({ queryKey: ideasQueryKeys.current() });
        navigate('/dashboard/ideas');
      },
    },
  });

  return (
    <div>
      <div className="space-y-6">
        <div>
          <h1 className="text-3xl font-bold mb-2">Create New Idea</h1>
          <p className="text-muted-foreground">
            Share your AI idea with the community
          </p>
        </div>

        {/* We are displaying the error state if the mutation fails */}
        {createIdeaMutation.error && (
          <ErrorMessage error={createIdeaMutation.error} />
        )}
        <IdeaForm
          // We are using mutate method to trigger the mutation
          onSubmit={createIdeaMutation.mutate}
          onCancel={() => navigate('/dashboard/ideas')}
          // We are showing the loading state if the mutation is pending
          isSubmitting={createIdeaMutation.isPending}
        />
      </div>
    </div>
  );
}
```

The mutation hook provides a `mutate` method that we pass to the form's `onSubmit`. When the form is submitted, the mutation executes. We use `isPending` to show loading state and `error` to display any errors. When the mutation succeeds, we invalidate the related queries so they refetch with the updated data, then navigate to the ideas list.

This pattern keeps our mutation logic clean and separate from the UI. The form just calls `mutate` with the data, and React Query handles the rest.

Summary

In this chapter, we built a complete API layer for our application.

We started by creating an API client that wraps the *fetch* API with error handling and consistent configuration. This gives us a single place to manage how we communicate with our backend.

Next, we set up type generation from our OpenAPI specification. Now we can run a command to generate TypeScript types and Zod validation schemas that match our backend. This catches breaking changes at compile time and provides runtime validation.

We then configured React Query to manage server state. React Query handles caching, loading states, error handling, and request deduplication automatically, so we don't have to manage these manually.

With the foundation ready, we built our API layer following a consistent pattern: fetcher function, options factory, and custom hook. Each API function validates input and output using Zod schemas for end-to-end type safety.

Finally, we explored three ways to integrate queries: client-side fetching with hooks, server-side fetching with initial data, and server-side prefetching with hydration boundaries. Each approach fits different use cases, and they all work seamlessly with React Query.

Our API layer ensures data is type-safe, cached efficiently, and always up to date, whether we're rendering on the server or the client.

Get this book's PDF version and more

Scan the QR code (or go to `packtpub.com/unlock`). Search for this book by name, confirm the edition, and then follow the steps on the page.

UNLOCK NOW

Note: Keep your invoice handy. Purchases made directly from Packt don't require an invoice.

6

Managing Application State

State management is an important aspect of every interactive web application. It determines how data flows through our application, how the UI responds to user interactions, and how to keep the application synchronized with the API. Without proper state management, our applications would be difficult to maintain and prone to bugs.

State refers to application data that changes over time and affects what is rendered in the UI. When state changes, React re-renders the components that depend on that state, updating the UI to reflect the new data. This mechanism is what makes interactive web applications responsive and dynamic.

In React applications, there are different types of state depending on where the data comes from, how long it should persist, and which components need access to it. Understanding these different types of state helps us make better architectural decisions and write more maintainable code. We'll cover the following topics:

- Managing local state
- Sharing state globally across components
- Handling asynchronous data from the server
- Managing form state with validation
- Using URL state for persistence and sharing

By the end of this chapter, we'll have an understanding of the different types of state in React applications and when to use each approach. We'll see practical examples from our application that demonstrate real-world state management patterns.

Technical requirements

Before we get started, we need to set up our project. To develop our project, we need the following tools installed on our computer:

- Node.js version 24 or above. npm version 11 or above ships with Node. We can confirm this by executing `node -v` and `npm -v` in the terminal. There are multiple ways to install Node.js and npm. Here is a helpful article that goes into more detail: `https://www.nodejsdesignpatterns.com/blog/5-ways-to-install-node-js` .
- VS Code (optional), a popular editor for JavaScript and TypeScript. It is open source, has solid TypeScript support, and offers many extensions. It can be downloaded from `https://code.visualstudio.com` .

The code for this book is available at the book's repo. To access the repository link, follow the steps in the " *Download the example code files* " section in the *Preface* . Clone it and enter the repository root:

```
git clone https://github.com/PacktPublishing/React-Application-Architecture-for-Production-Second-Edition.git
```

The repository contains chapter folders with the code for each chapter, along with a shared `api` folder that includes the API server used across all chapters.

We are working on *Chapter 6* , so navigate to the `chapter-06` directory:

```
cd React-Application-Architecture-for-Production-Second-Edition/chapter-06
```

Next, install the dependencies:

```
npm install
```

We also need to provide the environment variables:

```
cp .env.example .env
```

At this point, the frontend should be ready and running at `http://localhost:5173` .

We also need to run the API server.

Open a new terminal window and navigate to the `api` directory:

```
cd React-Application-Architecture-for-Production-Second-Edition/api
```

Run the setup script for *Chapter 6* to configure everything:

```
npm run setup 06
```

Then start the API server:

```
npm run dev
```

The API server should now be running on `http://localhost:9999` .

For more information about the setup details, check out the `README.md` file.

Managing local state

Local state is the simplest and most fundamental type of state in React. It lives within a single component and is only accessible to that component and its children if passed down as props. This encapsulation makes local state easy to reason about and maintain.

We should try to keep the state as close to the component that needs it as possible. If a piece of state is only needed within a small portion of the application and does not need to be persisted, there's no reason to make it more complicated. This principle of keeping state colocated helps us avoid unnecessary complexity and makes our components more reusable.

Let's look at a practical example from our application. When we try to create a new review from the idea detail page, we need to track whether the review modal is open or closed.

First, we need to click the "Write Review" button:

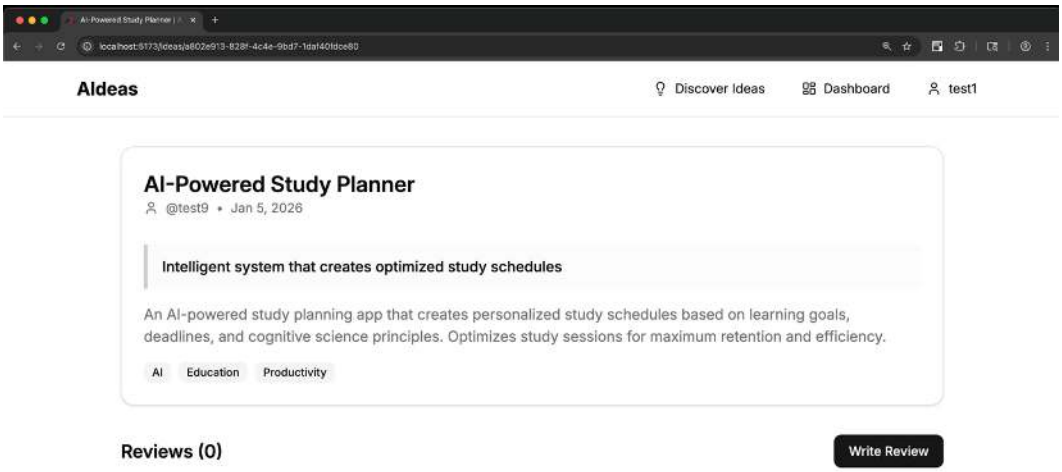


Figure 6.1 – Write Review button

When we click the button, the review modal should open:

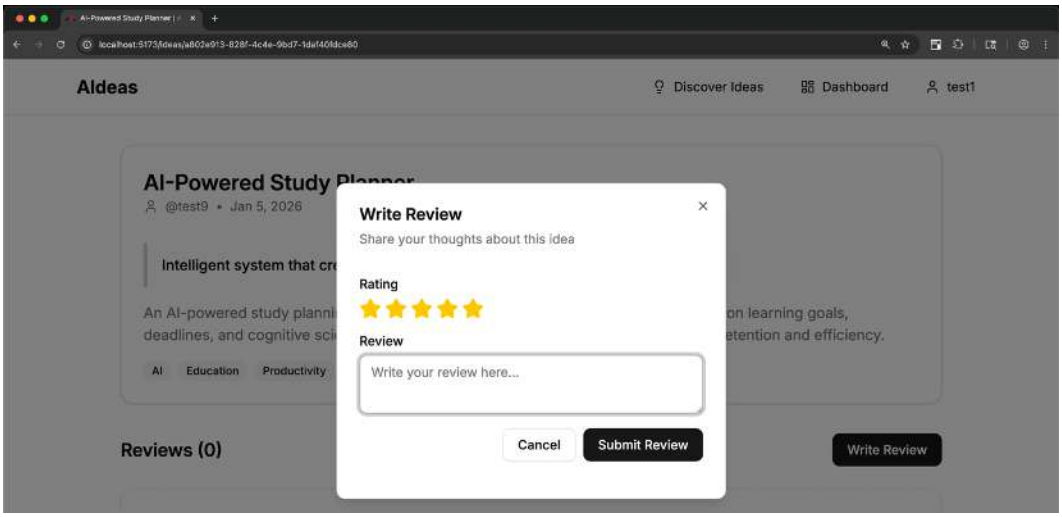


Figure 6.2 – Review modal in action

Here is the code for the `CreateReview` component:

```
// src/features/reviews/components/create-review.tsx

import { useState } from 'react';

import { Button } from '@/components/ui/button';
import { ReviewFormModal } from '@/features/reviews/components/review-form-modal';

export function CreateReview({ ideaId }: { ideaId: string }) {

  const [isReviewModalOpen, setIsReviewModalOpen] = useState(false);

  const createReviewMutation = useCreateReviewMutation({
```

```
    options: {
      onSuccess: () => {
        setIsReviewModalOpen(false);
      },
    },
  });


  const handleOpenReviewModalForCreate = () => {
    setIsReviewModalOpen(true);
  };


  const handleCloseReviewModal = () => {
    setIsReviewModalOpen(false);
    createReviewMutation.reset();
  };

  // ...

  return (
    <>
      <Button
        onClick={handleOpenReviewModalForCreate}
        disabled={createReviewMutation.isPending}
      >
        Write Review
      </Button>
      <ReviewFormModal
        isOpen={isReviewModalOpen}
        onClose={handleCloseReviewModal}
        onSubmit={handleReviewSubmit}
        isSubmitting={createReviewMutation.isPending}
        ideaId={ideaId}
        error={createReviewMutation.error}
      />
    </>
  );
}
```

In this example, we're using the `useState` hook to manage the modal's open/closed state. The `isReviewModalOpen` boolean tracks whether the modal is currently displayed, and we have two handler functions to open and close it. This is a good example of using local state because:

- The modal state is only needed within this component
- No other components in our application need to know if this modal is open

- The state doesn't need to persist when the component unmounts
- The logic is self-contained and easy to understand

When the user clicks the "Write Review" button, we call `handleOpenReviewModalForCreate` which sets the state to `true` , opening the modal. When the user closes the modal or submits the review, we call `handleCloseReviewModal` which sets the state back to `false` .

This pattern works great for UI state that's isolated to a single component. However, as our application grows, we'll encounter situations where multiple components need access to the same state or we need to update the state from multiple parts of the application. That's where global state comes in.

Sharing state globally

Global state is application state that can be accessed or modified from any component in the application that is subscribed to the changes, regardless of where it sits in the component tree. Unlike local state, which is encapsulated within a single component, global state is shared across multiple parts of our application.

The most common use cases for global state are features that need to be triggered or accessed from anywhere in the application. For example, showing notifications, managing theme preferences, or tracking the current user's shopping cart.

In our application, we use global state for the notifications system. When a user creates a review, updates their profile, or encounters an error, we want to show a toast notification. Since these actions can happen from any page or component, we need a way to trigger notifications globally.

There are several approaches to managing global state in React:

- **Prop drilling** - Passing state down through many levels of components, but it becomes difficult to manage quickly as the application grows. Intermediate components must accept and forward props they never read, so they become coupled to data they do not use which also makes the application difficult to maintain.
- **Context API** - React's built-in solution for sharing state. It's worth noting that context API is just a mechanism for passing data down

through the component tree, not a full state management solution and can cause performance issues if not used carefully, since all consumers are rerendered when the provider value changes.

- **Redux** - A powerful but verbose state management library. It adds boilerplate, but it's very popular.
- **Zustand** - A lightweight state management library with minimal boilerplate which is our choice for this project.

We chose Zustand for our application because it provides a simple API, good TypeScript support, and good performance without the complexity of larger solutions like Redux. This is exactly what we need.

Let's see how we implement our notifications store using Zustand:

```
// src/stores/notifications.ts

import { create } from 'zustand';

import { uid } from '@/lib/uid';

export type NotificationType = 'info' | 'warning' | 'success' | 'error';

export type Notification = {
  id: string;
  type: NotificationType;
  title?: string;
  duration?: number;
  message?: string;
};

type NotificationsStore = {
  notifications: Notification[];
  actions: {
    showNotification: (notification: Omit<Notification, 'id'>) => void;
    dismissNotification: (id: string) => void;
  };
};

const useNotificationsStore = create<NotificationsStore>((set, get) => ({
  notifications: [],
  actions: {
    showNotification: (notification) => {
      const id = uid();
      const duration = notification.duration ?? 5000;

      set((state) => ({
        notifications: [
```

```
          ...state.notifications,
          { id, duration, ...notification },
        ],
      }));
      if (duration > 0) {
        setTimeout(() => {
          get().actions.dismissNotification(id);
        }, duration);
      }
    },
    dismissNotification: (id) => {
      set((state) => ({
        notifications: state.notifications.filter(
          (notification) => notification.id !== id,
        ),
      }));
    },
  },
}));

export const useNotifications = () =>
  useNotificationsStore((state) => state.notifications);

export const useNotificationActions = () =>
  useNotificationsStore((state) => state.actions);
```

This store demonstrates several important patterns for global state management. We're creating a Zustand store with the `create` function, which takes a callback function that defines our state structure. The store contains a `notifications` array to hold all active notifications and an `actions` object with methods to show and dismiss notifications.

The `showNotification` action generates a unique ID for each notification, adds it to the notifications array, and sets up an automatic dismissal after the specified duration (defaulting to 5 seconds). The `dismissNotification` action removes a notification from the array by filtering it out based on its ID.

An important performance optimization we're using here is separating the state and actions into different selector hooks. Instead of exporting the store directly, we export two custom hooks: `useNotifications` for accessing the notifications array and `useNotificationActions` for accessing the action methods. This separation means that components using only the actions won't re-render when the notifications array changes, and it improves

performance. With our store defined, we need a component to display the notifications to users:

```
// src/components/notifications.tsx

import {
  useNotifications,
  useNotificationActions,
  type Notification,
} from '@/stores/notifications';

type NotificationItemProps = {
  notification: Notification;
};

export function NotificationItem({ notification }: NotificationItemProps) {
  // ...
}

export function Notifications() {
  const notifications = useNotifications();

  return (
    <div
      aria-live="assertive"
      className="pointer-events-none fixed inset-0 z-50 flex items-end px-4 py-6 sm:items-start sm:p-
6"
    >
      <div className="flex w-full flex-col items-center space-y-4 sm:items-end">
        {notifications.map((notification) => (
          <NotificationItem key={notification.id} notification={notification} />
        ))}
      </div>
    </div>
  );
}
```

The `Notifications` component is straightforward. It uses the `useNotifications` hook to access the notifications array from our global store and renders a `NotificationItem` for each notification to show them to the user.

To make notifications available throughout our application, we need to render this component at the root level. This ensures that no matter which page the user is on, notifications will appear:

```tsx
// src/app/root.tsx

import { Notifications } from '@/components/notifications';

export default function App() {
  const [queryClient] = useState(() => createQueryClient());
  return (
    <QueryClientProvider client={queryClient}>
      <Notifications />
      <Outlet />
    </QueryClientProvider>
  );
}
```

By placing the `Notifications` component at the root of our application, it will be rendered from every page. The `< Outlet / >` component renders our page content, while the `Notifications` component sits above it, ready to display notifications triggered from anywhere in the application.

Now comes the real power of global state: we can show notifications from any component without prop drilling or complex state passing.

Here's how we can notify the user when a review is created:

```tsx
// src/features/reviews/components/create-review.tsx

import { useState } from 'react';

import { useCreateReviewMutation } from '@/features/reviews/api/create-review';
import { useNotificationActions } from '@/stores/notifications';

export function CreateReview({ ideaId }: { ideaId: string }) {

  const [isReviewModalOpen, setIsReviewModalOpen] = useState(false);

  const { showNotification } = useNotificationActions();

  const createReviewMutation = useCreateReviewMutation({
    options: {
      onSuccess: () => {
        // trigger the notification to be shown
        showNotification({
          type: 'success',
          title: 'Review created',
        });
        setIsReviewModalOpen(false);
      },
    },
```

```
    });
    // ...
}
```

In the `CreateReview` component, we use the `useNotificationActions` hook to get access to the `showNotification` action. When the review is successfully created, we call this action with the notification details. The notification appears in the top-right corner of the screen, automatically disappears after 5 seconds, and can be dismissed by the user, as shown in Figure 6.3:

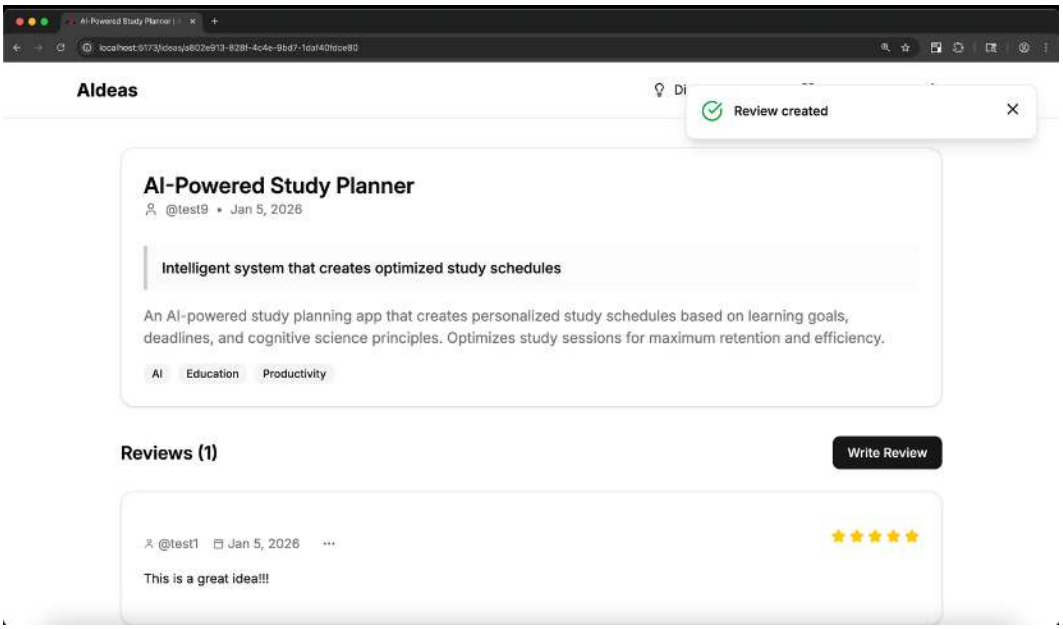


Figure 6.3 – Notifications in action

This is the benefit of global state management. We don't need to pass the notification system down through props or worry about where in the component tree we are. Any component can show a notification by simply calling `showNotification` . This makes our code cleaner and our features easier to implement.

Now what if we have a state that comes from the server? For example, we want to fetch the current user's reviews from the server and display them in the dashboard reviews page. This is where asynchronous state comes in.

Handling asynchronous state

Asynchronous state is state that comes from external sources like API requests. Unlike local or global state that we control entirely on the client, async state is a bit more challenging because it is loaded asynchronously so it can be loading, failing with errors, becoming stale, and needing to be synchronized between the source and the application.

Managing async state manually with `useState` and `useEffect` quickly becomes complex. We need to track loading states, handle errors, prevent race conditions, implement caching, and manage refetching. This is where React Query becomes super useful. It provides a powerful and declarative way to manage all aspects of asynchronous data.

We already explored React Query in the previous chapter, but let's look at how it fits into our overall state management strategy once again.

Let's imagine we have this component:

```
const loadData = () => api.get('/data');

const DataComponent = () => {
  const [data, setData] = useState();
  const [error, setError] = useState();
  const [isLoading, setIsLoading] = useState();
  useEffect(() => {
    setIsLoading(true);
    loadData()
      .then((data) => {
        setData(data);
      })
      .catch((error) => {
        setError(error);
      })
      .finally(() => {
        setIsLoading(false);
      });
  }, []);
  if (isLoading) return <div>Loading</div>;
  if (error) return <div>{error}</div>;
  return <div>{data}</div>;
};
```

This is fine if we fetch data from an API only once, but in most cases, we need to fetch it from many different endpoints. We can see that there is a certain amount of boilerplate here:

- The same data, error, and isLoading pieces of state need to be defined

- Different pieces of state must be updated accordingly
- The data is thrown away as soon as we move away from the component

That's where React Query comes in. We can update our component to the following:

```
import { useQuery } from '@tanstack/react-query';

const loadData = () => api.get('/data');

const DataComponent = () => {
  const { data, error, isLoading } = useQuery({
    queryKey: ['data'],
    queryFn: loadData,
  });
  if (isLoading) return <div>Loading</div>;
  if (error) return <div>{error}</div>;
  return <div>{data}</div>;
};
```

This is much cleaner and more maintainable. We don't need to define the same pieces of state over and over again. We can just use the `useQuery` hook to fetch the data and React Query handles the rest.

Let's see how this works in practice. When we create reviews in our application, we should see them on the dashboard reviews page:

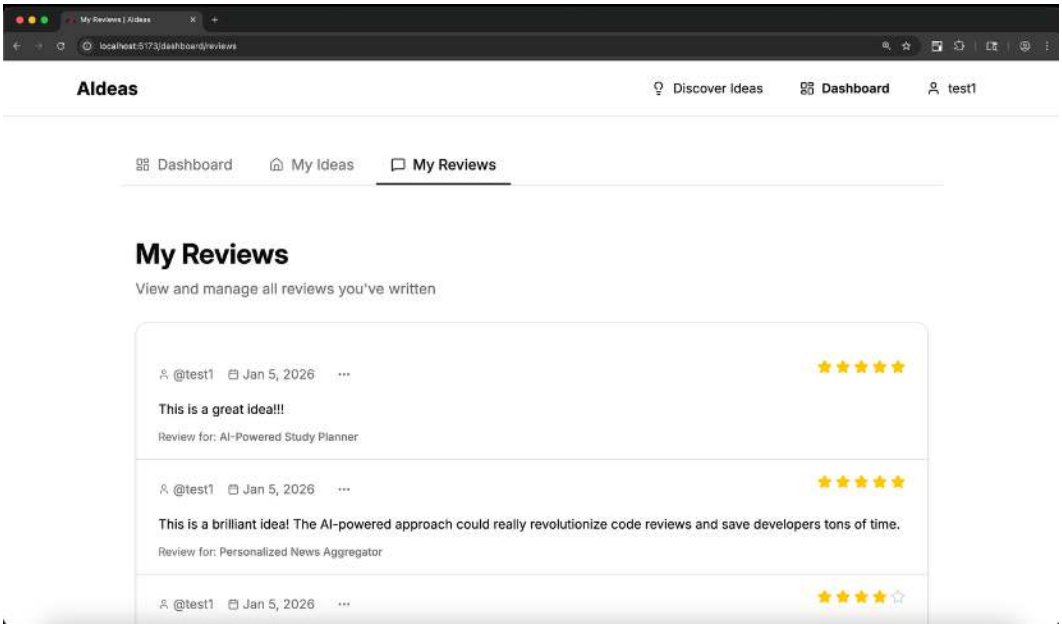


Figure 6.4 – Dashboard reviews page

Let's look at the code responsible for fetching the current user's reviews:

```
// src/features/reviews/api/get-current-user-reviews.ts

import { queryOptions, useQuery } from '@tanstack/react-query';

import { api } from '@/lib/api';
import type { GetReviewsByUserResponse } from '@/types/generated/types.gen';
import { zGetReviewsByUserResponse } from '@/types/generated/zod.gen';

import { reviewsQueryKeys } from '../config/query-keys';

export async function getCurrentUserReviews(): Promise<GetReviewsByUserResponse> {
  const response = await api.get<GetReviewsByUserResponse>('/reviews/current');

  return zGetReviewsByUserResponse.parse(response);
}

export function getCurrentUserReviewsQueryOptions() {
  return queryOptions({
    queryKey: reviewsQueryKeys.current(),
    queryFn: () => getCurrentUserReviews(),
  });
}

export function useCurrentUserReviewsQuery({
  options,
}: {
  options?: Omit<
    ReturnType<typeof getCurrentUserReviewsQueryOptions>,
    'queryKey' | 'queryFn'
  >;
} = {}) {
  return useQuery({
    ...getCurrentUserReviewsQueryOptions(),
    ...options,
  });
}
```

This code demonstrates our pattern for async state management. We create three parts:

- **Fetcher function** (`getCurrentUserReviews`) - The actual API call with input and output validation
- **Query options factory** (`getCurrentUserReviewsQueryOptions`) - Configuration for React Query including the cache key, the fetcher function, and any conditions

- **Custom hook** (`useCurrentUserReviewsQuery`) - A convenient hook that wraps React Query's `useQuery` with our options

This pattern provides several benefits. The fetcher function can be used independently for server-side data fetching or testing. The query options can be used for prefetching data, and the custom hook provides a clean API for components to use.

Now let's see how we use this in a component:

```
// src/app/routes/dashboard/reviews.tsx

import { Seo } from '@/components/seo';
import { useCurrentUserReviewsQuery } from '@/features/reviews/api/get-current-user-reviews';
import { ReviewsList } from '@/features/reviews/components/reviews-list';

export default function MyReviewsPage() {
  const reviewsQuery = useCurrentUserReviewsQuery();
  const reviews = reviewsQuery.data?.data;

  return (
    <div className="container mx-auto px-4 py-8">
      <Seo
        title="My Reviews | AIdeas"
        description="View and manage all reviews you've written for AI application ideas"
      />
      <div className="max-w-4xl mx-auto space-y-6">
        <div>
          <h1 className="text-3xl font-bold mb-2">My Reviews</h1>
          <p className="text-muted-foreground">
            View and manage all reviews you've written
          </p>
        </div>

        <ReviewsList
          reviews={reviews}
          isLoading={reviewsQuery.isLoading}
          showIdeaTitle
          emptyMessage="You haven't written any reviews yet. Explore ideas and share your feedback!"
          error={reviewsQuery.error}
        />
      </div>
    </div>
  );
}
```

The `MyReviewsPage` component demonstrates how simple async state management becomes with React Query. We call our custom hook, and React

Query handles everything else with the `useQuery` hook under the hood. We get the data, loading state, and error state automatically. When the component gets rendered, React Query fetches the data. If another component has already fetched reviews for the current user, React Query returns the cached data immediately without duplicating the request.

React Query also handles many complex scenarios for us:

- **Automatic caching** : Data is cached and reused across components
- **Background refetching** : Stale data is refreshed automatically in the background
- **Deduplication** : Multiple components requesting the same data trigger only one network request
- **Error handling** : Failed requests are retried automatically
- **Loading states** : We get granular control over loading states for better UX

This is why we treat async state differently from other types of state. Libraries like React Query are specifically designed to handle the complexities of server data, and they do it much better than we could with manual state management.

It's worth noting that async state is not just for data that comes from the API, but any data that is loaded asynchronously. For example, we can use React Query to fetch data from a browser API like the Geolocation API or the Web Speech API.

Managing form state

Forms are used to collect user information in web applications. Whether users are creating content, updating their profile, or submitting reviews, they're using forms. Managing form state effectively is important for a good user experience.

We could use local state to track the form values, but form state includes not just the current values of inputs, but also validation errors, submission status, which fields have been touched, and more. Managing all this manually becomes error-prone and repetitive. That's where libraries like React Hook Form come in.

React Hook Form provides a performant and flexible way to handle forms with minimal re-renders. Combined with Zod for schema validation, it gives us type-safe forms with automatic validation. Let's look at our review form as an example. This form demonstrates several key concepts of form state management. Let's break down what's happening:

```tsx
// src/features/reviews/components/review-form.tsx

export function ReviewForm({
  onSubmit,
  isSubmitting,
  onModalClose,
  initialReview,
  error,
  ideaId,
}: ReviewFormProps) {
  const form = useForm<CreateReviewData['body']>({
    resolver: zodResolver(zCreateReviewData.shape.body),
    defaultValues: {
      content: initialReview?.content || '',
      rating: initialReview?.rating || 5,
      ideaId: initialReview?.ideaId || ideaId || '',
    },
  });

  const handleClose = () => {
    form.reset();
    onModalClose();
  };

  return (
    <Form {...form}>
      <form
        onSubmit={form.handleSubmit(onSubmit)}
        className="space-y-4 py-4"
      >
        {/* ... */}
      </form>
    </Form>
  );
}
```

First, we initialize the form with the `useForm` hook from React Hook Form. We pass it a `zodResolver` with our Zod schema to get the field validation. Notice how we provide the same schema that validates the request body in our API.

This schema is automatically generated from the OpenAPI specification, ensuring frontend and backend validation stay synchronized.

The `defaultValues` can be either from an existing review we're editing or empty values for creating a new review.

The form manages all the complexity of form state for us:

- **Field values** : Current input values are tracked automatically
- **Validation** : Zod schemas validate on blur and submit
- **Error messages** : Validation errors are displayed next to fields
- **Form submission** : The `handleSubmit` wrapper prevents default behavior and validates before calling our callback
- **Reset functionality** : We can reset the form to initial values with `form.reset()`

Now let's look at how to add fields to the form:

```
// src/features/reviews/components/review-form.tsx

export function ReviewForm({
  // ...
  return (
    <Form {...form}>
      <form {/* ... */}>
        <FormField
          control={form.control}
          name="rating"
          render={({ field }) => (
            <FormItem>
              <FormLabel>Rating</FormLabel>
              <FormControl>
                <div className="flex space-x-1 mt-2">
                  {Array.from({ length: 5 }).map((_, i) => (
                    <Star
                      key={i}
                      className={`h-6 w-6 cursor-pointer ${
                        i < field.value
                          ? 'fill-yellow-400 text-yellow-400'
                          : 'text-gray-300'
                      }`}
                      onClick={() => field.onChange(i + 1)}
                    />
                  ))}
                </div>
              </FormControl>
              <FormMessage />
```

```
            </FormItem>
          )}
        />

        <FormField
          control={form.control}
          name="content"
          render={({ field }) => (
            <FormItem>
              <FormLabel>Review</FormLabel>
              <FormControl>
                <Textarea
                  placeholder="Write your review here..."
                  rows={4}
                  disabled={isSubmitting}
                  {...field}
                />
              </FormControl>
              <FormMessage />
            </FormItem>
          )}
        />

{error&&<ErrorMessage error={error} />}

        <div className="flex justify-end space-x-2">
          <Button
            type="button"
            variant="outline"
            onClick={handleClose}
            disabled={isSubmitting}
          >
            Cancel
          </Button>
          <Button type="submit" disabled={isSubmitting}>
            {isSubmitting ? 'Submitting...' : 'Submit Review'}
          </Button>
        </div>
      </form>
    </Form>
  );
}
```

The `FormField` components connect React Hook Form to our UI components. Each field uses the `render` prop pattern to access the field's value, change handler, and validation state. This gives us complete control over how each field is rendered while React Hook Form handles the state management behind the scenes.

Here is how the form looks in action:

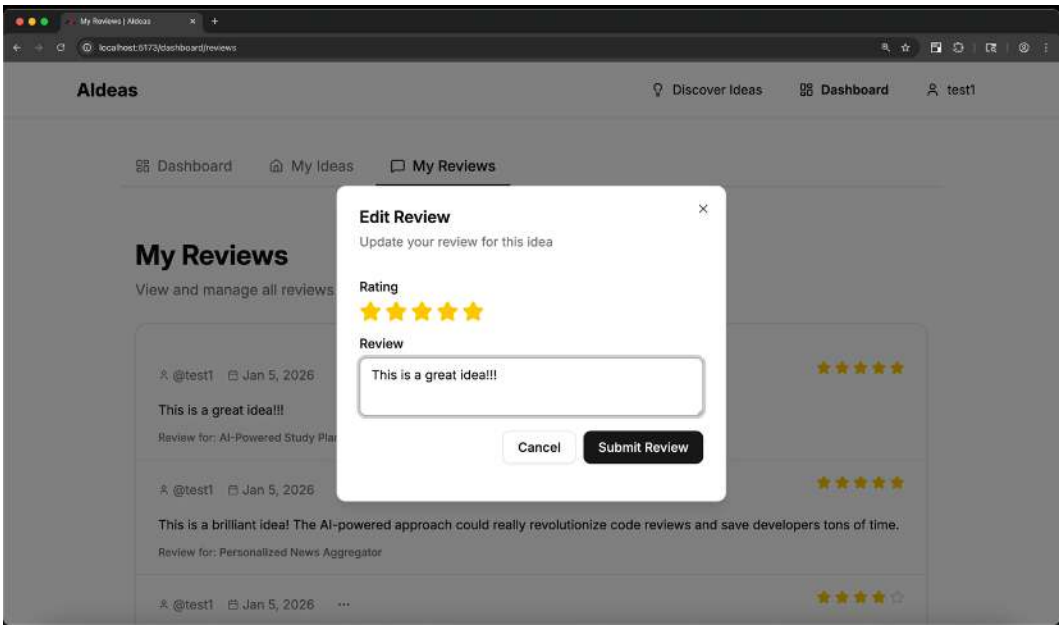


Figure 6.5 – Review form in action

With form state covered, let's explore the final type of state we'll discuss in this chapter: URL state.

Persisting state in the URL

URL state, also called URL search parameters or query parameters, is a special type of state that lives in the browser's address bar. It's one of the most underutilized but very powerful types of state in web applications.

Why would we want to store state in the URL? URL state has unique advantages:

- **Persistable** : URL state persists across page reloads.
- **Shareable** : Users can share URLs that preserve the exact state of the page
- **Bookmarkable** : Users can bookmark URLs and return to the exact same view
- **Browser navigation** : Back and forward buttons work naturally with URL state
- **Server-side rendering** : URL state is available on the server for initial data fetching

In our application, we use URL state for search filters and sorting on the ideas page. When a user searches for ideas, filters by tags, or changes the sort order, these preferences are reflected in the URL. This means they can share a link to a filtered view or refresh the page without losing their filters.

While React Router has built-in functionality to manage query params via the `useSearchParams` hook, there is a library called nuqs which provides a nicer way for managing URL state with React hooks. To get started, we need to wrap our application with the nuqs adapter:

```
// src/app/root.tsx

import { NuqsAdapter } from 'nuqs/adapters/react-router/v7';

export default function App() {
  const [queryClient] = useState(() => createQueryClient());
  return (
    <QueryClientProvider client={queryClient}>
      <NuqsAdapter>
        <Notifications />
        <Outlet />
      </NuqsAdapter>
    </QueryClientProvider>
  );
}
```

The `NuqsAdapter` provides the context needed for nuqs to work with React Router. Now let's look at how we use it to manage search and filter state:

```
// src/features/ideas/hooks/use-search-and-filters.ts

import { useQueryState, parseAsArrayOf, parseAsString } from 'nuqs';

export function useSearchAndFilters() {
  const [urlSearchTerm, setUrlSearchTerm] = useQueryState(
    'search',
    parseAsString.withDefault(''),
  );

  const [selectedTags, setSelectedTags] = useQueryState(
    'tags',
    parseAsArrayOf(parseAsString).withDefault([]),
  );

  const [sortBy, setSortBy] = useQueryState(
    'sortBy',
    parseAsString.withDefault(''),
```

```
  );

  const toggleTag = (tag: string) => {
    setSelectedTags((currentTags) => {
      const newTags = currentTags.includes(tag)
        ? currentTags.filter((t) => t !== tag)
        : [...currentTags, tag];
      return newTags.length > 0 ? newTags : null;
    });
  };

  const clearFilters = () => {
    setUrlSearchTerm('');
    setSelectedTags(null);
    setSortBy(null);
  };

  const hasActiveFilters =
    selectedTags.length > 0 || urlSearchTerm.length > 0 || sortBy.length > 0;

  return {
    searchTerm: urlSearchTerm,
    urlSearchTerm,
    selectedTags,
    sortBy,
    params: {
      ...(selectedTags.length > 0 && { tags: selectedTags.join(',') }),
      ...(urlSearchTerm && { search: urlSearchTerm }),
      ...(sortBy && { sortBy }),
    },
    hasActiveFilters,
    setSearchTerm: setUrlSearchTerm,
    setSelectedTags,
    toggleTag,
    setSortBy,
    clearFilters,
  };
}
```

This custom hook demonstrates the power of URL state management. Let's break down what's happening:

We use `useQueryState` from `nuqs` for each piece of filter state: search term, selected tags, sort order, and current page. Each call to `useQueryState` returns the current value from the URL and a setter function that updates both the URL and the state. The `parseAs` functions ensure type safety and proper serialization.

The hook includes several useful features:

- **Automatic URL synchronization** : When we call the setter functions, the URL updates automatically
- **Type-safe parsing** : The `parseAs` functions convert URL strings to the correct types
- **Default values** : Each parameter has a sensible default if not present in the URL
- **Computed params** : We build an object ready to pass to our API queries
- **Helper functions** : Functions like `toggleTag` and `clearFilters` make the API convenient to use

One important detail is the `useEffect` that resets the page to 1 when filters change. This prevents users from ending up on page 10 of a filtered result set that only has 2 pages.

Now let's see how we use this hook in the search and filters UI component:

```
// src/features/ideas/components/idea-search-and-filters.tsx


import { useSearchAndFilters } from '@/features/ideas/hooks/use-search-and-filters';

export function IdeaSearchAndFilters() {
  const {
    searchTerm,
    selectedTags,
    sortBy,
    hasActiveFilters,
    setSearchTerm,
    toggleTag,
    setSortBy,
    clearFilters,
  } = useSearchAndFilters();

  const tagsQuery = useTagsQuery();
  const availableTags = tagsQuery.data?.data || [];

  return (
    <div className="space-y-4">
      <div className="relative">
        <Search className="absolute left-3 top-1/2 transform -translate-y-1/2 h-4 w-4 text-muted-
foreground" />
        <Input
          placeholder="Search ideas by title or description..."
          value={searchTerm}
```

```
          onChange={(e) => setSearchTerm(e.target.value)}
          className="pl-10"
        />
      </div>

      <div className="flex flex-wrap items-center gap-2">
        <span className="text-sm font-medium">Filter by tags:</span>
        {availableTags.map((tag) => (
          <Badge
            key={tag}
            variant={selectedTags.includes(tag) ? 'default' : 'outline'}
            className="cursor-pointer"
            onClick={() => toggleTag(tag)}
          >
            {tag}
          </Badge>
        ))}
      </div>
      {/* ... */}
    </div>
  );
}
```

The `IdeaSearchAndFilters` component uses our custom hook to provide an interactive filtering interface. Users can search by text, filter by tags, and change the sort order. All of these changes are immediately reflected in the URL, which means:

- Typing in the search box updates the URL with `?search=text`
- Clicking a tag adds it to the URL with `?tags=tag1,tag2`
- Changing the sort order updates the URL with `?sortBy=rating`
- All parameters can be combined: `?search=ai&tags=ml,nlp&sortBy=rating&page=2`

The beauty of this approach is that every user action creates a shareable URL. Users can bookmark their favorite filtered views or share links with other users. The browser's back button and forward button work naturally, letting users navigate through their filter history.

Now let's see how this is used in our application on the ideas page:

```
// src/app/routes/ideas/ideas.tsx

export async function loader({ request }: Route.LoaderArgs) {
  const url = new URL(request.url);
  const searchParams = url.searchParams;
```

```
  const ideasParams = {
    search: searchParams.get('search') || undefined,
    tags: searchParams.get('tags') || undefined,
    sortBy: searchParams.get('sortBy') || undefined,
  } as GetAllIdeasData['query'];
  const queryClient = createQueryClient();

  await Promise.all([
    queryClient.prefetchQuery(getIdeasQueryOptions(ideasParams)),
    queryClient.prefetchQuery(getTagsQueryOptions()),
  ]);

  return routerData({
    dehydratedState: dehydrate(queryClient),
  });
}

export default function IdeasPage({ loaderData }: Route.ComponentProps) {
  return (
    <HydrationBoundary state={loaderData.dehydratedState}>
      <Ideas />
    </HydrationBoundary>
  );
}

function Ideas() {
  const { params } = useSearchAndFilters();

  const ideasQuery = useIdeasQuery({
    params: params as GetAllIdeasData['query'],
  });

  const allIdeas = ideasQuery.data?.data || [];

  return (
    <div className="container mx-auto px-4 py-8">
      <Seo
        title="Discover AI Ideas | AIdeas"
        description="Browse and explore innovative AI application ideas from the community"
      />
      <div className="mb-8">
        <h1 className="text-3xl font-bold mb-4">Discover AI Ideas</h1>
        <p className="text-muted-foreground mb-6">
          Browse and explore innovative AI application ideas from the community
        </p>
        <IdeaSearchAndFilters />
      </div>

      <IdeasList
```

```
          ideas={allIdeas}
          isLoading={ideasQuery.isLoading && allIdeas.length === 0}
          emptyMessage={'No ideas available yet'}
          error={ideasQuery.error}
        />
      </div>
    );
  }
```

This page brings together multiple types of state management in a cohesive example. Let's trace how the different state types work together:

The `loader` function runs on the server before the page renders. It reads the URL search parameters and uses them to prefetch data with React Query. This ensures the page loads with the correctly filtered data on the initial server-side render.

The `Ideas` component uses our `useSearchAndFilters` hook to access the current URL state and build the params object. These params are passed to `useIdeasQuery`, which fetches the ideas from the API on the client side.

When users interact with the search and filters, the URL updates automatically. Because we pass the URL params to React Query's query key, any change to the filters triggers a new data fetch. React Query is smart enough to show the cached data while fetching the updated results in the background, providing a smooth user experience.

The pagination component updates the page parameter in the URL, which triggers a new query for the next page of results. All of this happens seamlessly, with the URL always reflecting the current state of the page.

Here is how the URL state is used in the ideas page:

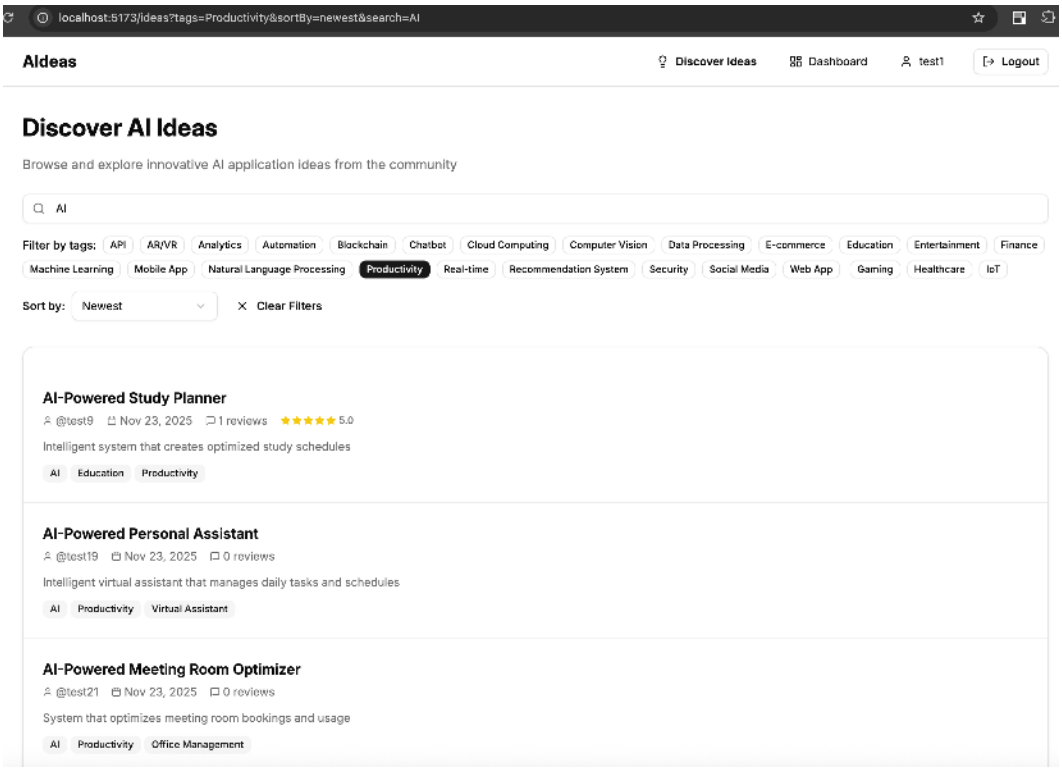


Figure 6.6 – URL state usage in ideas page

Notice the URL params are reflected in the UI. The search term is displayed in the search input, the selected tags are displayed as badges, and the sort order is displayed in the select dropdown.

This example demonstrates how different types of state work together in a real application. URL state drives what data we fetch, async state manages the server data, and local state handles UI interactions. Each type of state has its role, and using the right approach for each piece of state creates a maintainable and performant application.

Summary

In this chapter, we explored the five main types of state in React applications and learned when and how to use each approach.

We started with local state, the simplest form of state management using React's `useState` hook. Local state is perfect for UI state that's self-contained within a single component, such as whether a modal is open or closed. We learned to always start with local state and only move to more complex solutions when necessary.

Next, we examined global state management with Zustand. Global state is useful for data that needs to be accessed from multiple components throughout the application, such as notifications or authentication status. We implemented a notifications system that can be triggered from any component, demonstrating how global state enables features that transcend component boundaries.

We then looked at asynchronous state management with React Query. Server state has unique challenges including loading states, errors, caching, and synchronization. React Query handles all of these complexities for us, providing a declarative API for fetching, caching, and updating server data. We saw how this integrates with our API layer to create a robust data fetching strategy.

Form state management was our next topic, using React Hook Form combined with Zod for schema validation. Forms require managing multiple pieces of state including field values, validation errors, touched fields, and submission status. React Hook Form handles all of this efficiently with minimal re-renders, while Zod provides type-safe validation.

Finally, we explored URL state management with nuqs. URL state is special because it's shareable, bookmarkable, and works naturally with browser navigation. We implemented a complete search and filter system where all user preferences are stored in the URL, making every filtered view shareable and persistent across page reloads.

Throughout the chapter, we saw how these different types of state work together in a real application. The ideas page demonstrates this perfectly: URL state drives what data to fetch, async state manages the server data, form state handles user input, global state shows notifications, and local state manages UI interactions. Understanding when to use each type of state is crucial for building maintainable React applications.

Get this book's PDF version and more

Scan the QR code (or go to `packtpub.com/unlock`). Search for this book by name, confirm the edition, and then follow the steps on the page.

UNLOCK NOW

Note: Keep your invoice handy. Purchases made directly from Packt don't require an invoice.

7

Implementing Authentication and Securing the Application

Authentication and authorization are fundamental to building secure applications. Think of authentication as checking in at a hotel; it verifies who the user is by confirming their identity through login credentials. Authorization is like the key card you receive; it determines what the user can do once they're in. In the context of our application, authentication confirms someone is a registered user, while authorization ensures only the author of an idea can edit or delete it.

Beyond authentication and authorization, we need to protect our application and its users from security vulnerabilities by preventing malicious code from running on behalf of the user. We'll cover the following topics:

- Understanding authentication and authorization
- Implementing authentication with httpOnly cookies
- Protecting parts of the application with authorization policies
- Building registration, login, and logout flows
- Protecting authenticated routes
- Preventing XSS attacks with content sanitization
- Securing the application with security headers

By the end of this chapter, we'll have a secure authentication system in place that protects the application and implements security practices to keep our application and users safe.

Technical requirements

Before we get started, we need to set up our project. To develop our project, we need the following tools installed on our computer:

- Node.js version 24 or above. npm version 11 or above ships with Node. We can confirm this by executing `node -v` and `npm -v` in the terminal. There are multiple ways to install Node.js and npm. Here is a helpful article that goes into more detail: `https://www.nodejsdesignpatterns.com/blog/5-ways-to-install-node-js` .
- **VS Code** (optional), a popular editor for JavaScript and TypeScript. It is open source, has solid TypeScript support, and offers many extensions. It can be downloaded from `https://code.visualstudio.com` .

The code for this book is available at `https://github.com/PacktPublishing/React-Application-Architecture-for-Production-Second-Edition` on GitHub. Clone it and enter the repository root:

```
git clone https://github.com/PacktPublishing/React-Application-Architecture-for-Production-Second-Edition.git
```

The repository contains chapter folders with the code for each chapter, along with a shared `api` folder that includes the API server used across all chapters.

We are working on *Chapter 7* so navigate to the `chapter-07` directory:

```
cd React-Application-Architecture-for-Production-Second-Edition/chapter-07
```

Next, install the dependencies:

```
npm install
```

We also need to provide the environment variables:

```
cp .env.example .env
```

At this point, the frontend should be ready and running at `http://localhost:5173` .

We also need to run the API server.

Open a new terminal window and navigate to the `api` directory:

```
cd React-Application-Architecture-for-Production-Second-Edition/api
```

Run the setup script for *Chapter 7* to configure everything:

```
npm run setup 07
```

Then start the API server:

```
npm run dev
```

The API server should now be running on `http://localhost:9999` .

For more information about the setup details, check out the `README.md` file.

Authentication

Authentication is the process of verifying who a user is. When a user logs into our application, we need to verify their identity and then remember who they are as they navigate through different pages and make requests to our API. This is fundamental to building applications that serve personalized content and protect sensitive data.

Without authentication, we can't distinguish between different users or restrict access to certain features. Every user would see the same content, and anyone could access any data. This works for public websites, but most applications need to know who is using them.

We'll implement authentication using a token-based approach. When users log in with their credentials, our API will verify them and send back authentication tokens. These tokens will be stored in `httpOnly` cookies, which are cookies that JavaScript can't access. This is important because it protects the tokens from being stolen by malicious scripts that might run on our page, as httpOnly cookies can't be accessed by JavaScript on the client-side. It is worth noting that while `httpOnly` cookies reduce the risk of cross-site scripting (XSS) attacks stealing our tokens, they don't protect against cross-site request forgery (CSRF) attacks on their own. To help mitigate CSRF, the `SameSite` cookie attribute can be used; setting it to `Lax` or `Strict` instructs browsers to limit when the cookie is sent with cross-origin requests. To learn more about `SameSite` , you can check `https://web.dev/articles/samesite-cookies-explained` .

The following diagram shows how the authentication flow works:

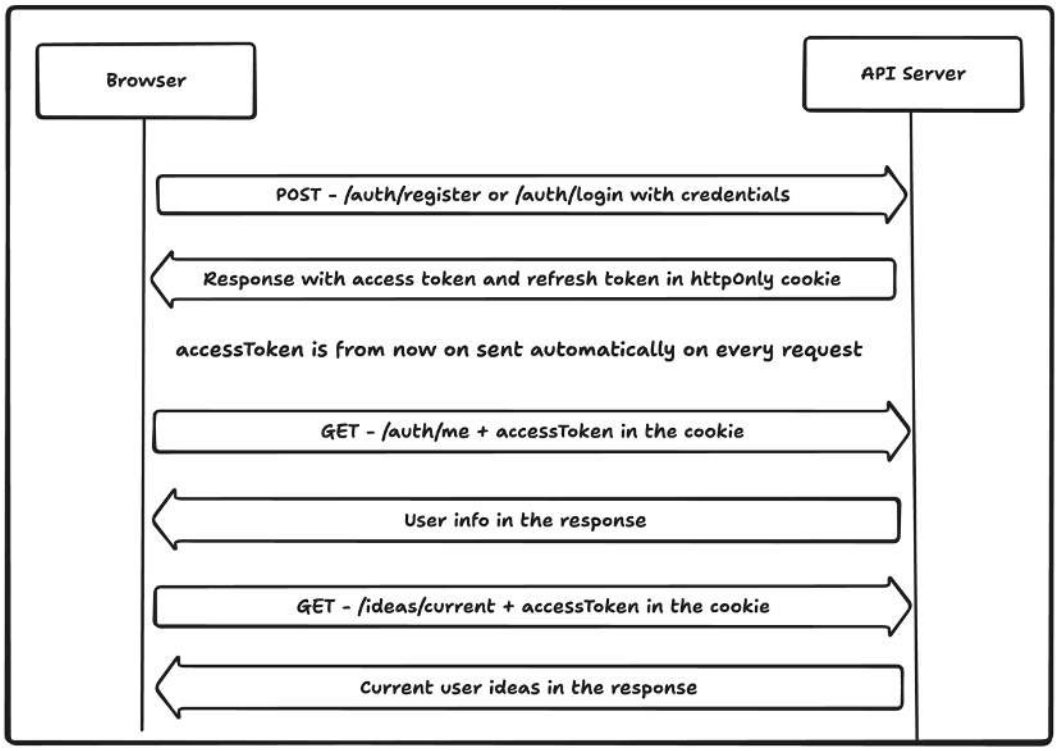


Figure 7.1 – The authentication flow

In this flow, the user submits their credentials to the API, which validates them and returns authentication tokens. These tokens are stored as httpOnly cookies in the browser and automatically sent with subsequent requests to identify the user.

Our API uses two types of tokens to handle authentication:

- **Access tokens** : Short-lived tokens (usually 15 minutes) that prove the user's identity for each request. These are sent with every API call to identify the user.
- **Refresh tokens** : Longer-lived tokens (usually 7 days) that can generate new access tokens when the old ones expire. This lets users stay logged in without having to re-enter their credentials every 15 minutes.

This two-token system balances security and user experience. Access tokens expire quickly to limit the damage if they're intercepted, while refresh tokens let users stay logged in for longer periods without compromising security.

To support this authentication flow, we need to extend our API client to handle tokens, implement registration and login pages, and protect routes that require authentication.

Extending the API client

When our access token expires, the API will respond with a 401 Unauthorized status. Instead of logging the user out immediately, we should try to get a new access token using the refresh token. This way, users can stay logged in without noticing that their token expired.

First, let's update the client to always send cookies automatically on the client side:

```
// src/lib/api.ts

async function fetchApi<T, TBody = unknown>(
  url: string,
  options: RequestOptions<TBody> = {},
): Promise<T> {
  // ...

  const makeRequest = async (accessToken?: string): Promise<Response> => {
    try {
      return await fetch(fullUrl, {
        // ...
        credentials: typeof window !== 'undefined' ? 'include' : undefined,
      });
    } catch (error) {
      // ...
    }
  };
}
```

We are sending cookies automatically on the client-side by setting the `credentials` option to `include`. The default value is `same-origin` which means cookies are only sent for requests to the same origin. But our API is running on a different origin, so we need to send cookies explicitly. On the server-side, we will have to pass cookies explicitly in the headers like this:

```
const response = await api.get<GetCurrentUserResponse>('/auth/me', {
    headers: {
      Cookie: cookieHeader ?? '',
    },
});
```

We also want to support refreshing the access token.

```
// src/lib/api.ts
```

```
async function fetchApi<T, TBody = unknown>(
  url: string,
  options: RequestOptions<TBody> = {},
): Promise<T> {
  // ...

  // Handle 401 errors with automatic token refresh
  const isAuthenticationError =
    !response.ok &&
    response.status === 401 &&
    !url.endsWith('/auth/login') &&
    !url.endsWith('/auth/register')


  if (isAuthenticationError) {
    // Extract cookie header from options for server-side requests
    // On client-side, this will be undefined and credentials: 'include' will be used
    const cookieHeader = headers.Cookie;

    try {
      // Attempt to refresh the token
      const { accessToken } = await refreshToken(cookieHeader);

      // Retry the original request with the new token
      response = await makeRequest(accessToken);
    } catch (refreshError) {
      // If refresh fails, the original 401 error will be handled below
      console.warn('Token refresh failed:', refreshError);
    }
  }

  // ...
}
```

When a request fails with a 401 status, we automatically attempt to refresh the token and retry the original request with the new access token.

This setup ensures that users stay logged in seamlessly even when their access tokens expire, and it works correctly both in the browser and on the server.

The following diagram illustrates the token refresh process:

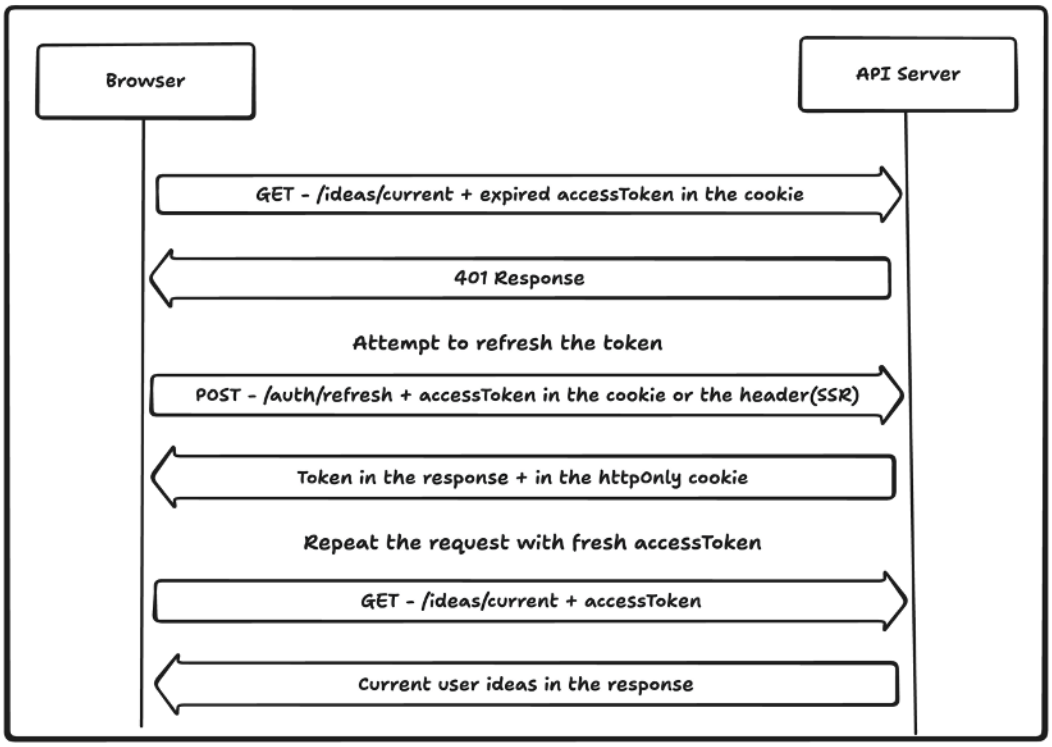


Figure 7.2 – The refresh token flow

When an API request fails with a 401 status, the client automatically calls the refresh endpoint using the refresh token cookie. If the refresh token is valid, the API returns a new access token, and the client retries the original request. This all happens in the background without the user being aware of it.

Now that we have the API client ready to handle authentication, we need a way for users to get those tokens in the first place. Users can either register as a new user or log in with existing credentials.

Registration page

To register a new user, we need to create a new route and a form component that when submitted will call the registration endpoint that will create a new user and return the auth tokens in the cookies.

```
// src/app/routes/auth/register.tsx

export default function RegisterPage() {
  const navigate = useNavigate();

  const registerUserMutation = useRegisterUserMutation();

  const onSubmit = (data: RegisterUserData['body']) => {
```

```
    registerUserMutation.mutate(data, {
      onSuccess: () => {
        navigate('/dashboard');
      },
    });
  };

  return (
    <div className="container mx-auto px-4 py-16">
      <div className="max-w-md mx-auto">
        <Card>
          <CardHeader>
            <CardTitle className="text-2xl text-center">Register</CardTitle>
          </CardHeader>
          <CardContent>
            <RegisterForm
              onSubmit={onSubmit}
              error={registerUserMutation.error}
              isPending={registerUserMutation.isPending}
            />
            <div className="text-center mt-4">
              <p className="text-sm text-muted-foreground">
                Already have an account?{' '}
                <Link to="/auth/login" className="text-primary hover:underline">
                  Log In
                </Link>
              </p>
            </div>
          </CardContent>
        </Card>
      </div>
    </div>
  );
}
```

The registration page looks like this:

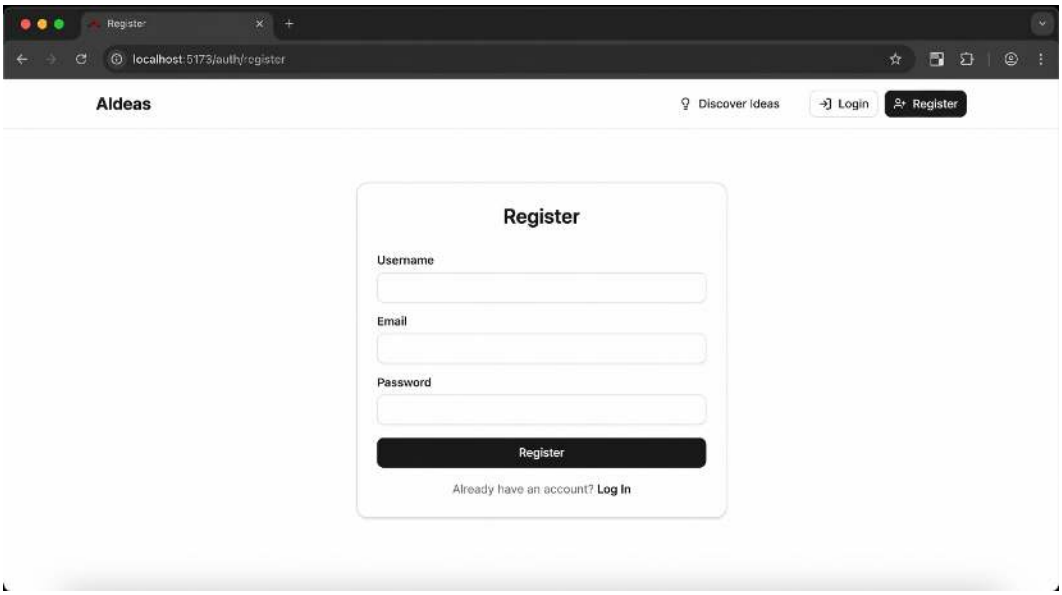


Figure 7.3 – The registration page

This registration page is straightforward. It renders a form inside a card component and handles the submission by calling the registration mutation. When registration succeeds, we navigate the user to the dashboard. The form component receives the error and loading states from the mutation so it can show error messages and disable inputs while the request is in progress.

The actual registration logic lives in the mutation hook.

```
// src/features/auth/api/register.ts

export async function registerUser(
  data: RegisterUserData['body'],
): Promise<RegisterUserResponse> {
  const response = await api.post<RegisterUserResponse>('/auth/register', {
    body: data,
  });

  return zRegisterUserResponse.parse(response);
}

export function getRegisterUserMutationOptions() {
  return mutationOptions({
    mutationFn: registerUser,
  });
}

export function useRegisterUserMutation({
  options,
}: {
```

```
  options?: Omit<
    UseMutationOptions<RegisterUserResponse, Error, RegisterUserData['body']>,
    'mutationFn'
  >;
} = {}) {
  return useMutation({
    ...getRegisterUserMutationOptions(),
    ...options,
  });
}
```

The registration function sends the user's credentials to the `/auth/register` endpoint. When registration succeeds, the API automatically sets the authentication tokens in `httpOnly` cookies in the response, so the user is immediately logged in without any additional steps.

Now we need to configure the route to render the registration page. This is done in the `routes.ts` file.

```
// src/app/routes.ts

export default [
  route('auth/register', './routes/auth/register.tsx'),
  // ...
];
```

With registration in place, we also need a way for existing users to log back into the application.

Login page

The login flow is very similar to registration. We need a page with a form where users can enter their credentials, and when they submit the form, we call the login endpoint to authenticate them.

```
// src/app/routes/auth/login.tsx

export default function LoginPage() {
  const navigate = useNavigate();

  const loginUserMutation = useLoginUserMutation();

  const onSubmit = (data: LoginUserData['body']) => {
    loginUserMutation.mutate(data, {
      onSuccess: () => {
        navigate('/dashboard');
      },
```

```
    });
  };

  return (
    <div className="container mx-auto px-4 py-16">
      <div className="max-w-md mx-auto">
        <Card>
          <CardHeader>
            <CardTitle className="text-2xl text-center">Log In</CardTitle>
          </CardHeader>
          <CardContent>
            <LoginForm
              onSubmit={onSubmit}
              error={loginUserMutation.error}
              isPending={loginUserMutation.isPending}
            />
            <div className="text-center mt-4">
              <p className="text-sm text-muted-foreground">
                Don't have an account?{' '}
                <Link
                  to="/auth/register"
                  className="text-primary hover:underline"
                >
                  Register
                </Link>
              </p>
            </div>
          </CardContent>
        </Card>
      </div>
    </div>
  );
}
```

The login page looks like this:

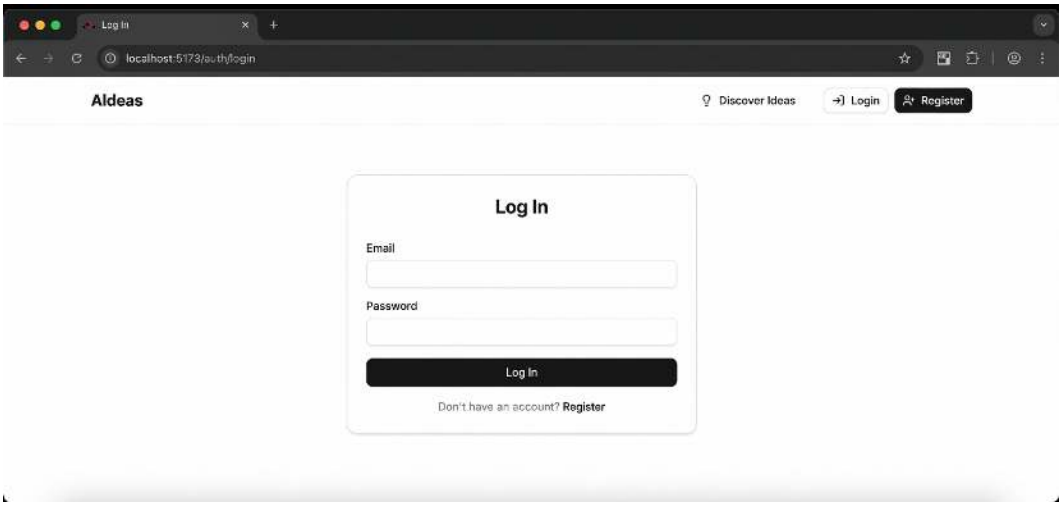


Figure 7.4 – The login page

The login page follows the same pattern as registration. It handles form submission with a mutation and navigates to the dashboard on success.

The mutation calls the login endpoint to verify the user's credentials.

```
// src/features/auth/api/login.ts

export async function loginUser(
  data: LoginUserData['body'],
): Promise<LoginUserResponse> {
  const response = await api.post<LoginUserResponse>('/auth/login', {
    body: data,
  });

  return zLoginUserResponse.parse(response);
}

export function getLoginUserMutationOptions() {
  return mutationOptions({
    mutationFn: loginUser,
  });
}

export function useLoginUserMutation({
  options,
}: {
  options?: Omit<
    UseMutationOptions<LoginUserResponse, Error, LoginUserData['body']>,
    'mutationFn'
  >;
} = {}) {
  return useMutation({
    ...getLoginUserMutationOptions(),
    ...options,
  });
}
```

Just like registration, the login endpoint sets authentication tokens in httpOnly cookies when the credentials are valid. The user is then authenticated and can access protected areas of the application.

We also need to configure the route to render the login page. This is done in the `routes.ts` file.

```
// src/app/routes.ts

export default [
```

```
  route('auth/login', './routes/auth/login.tsx'),
  // ...
];
```

Now that users can register and log in, we need to provide a way for them to log out.

Logging out the user

Logging out is different from registration and login because users don't fill out a form. Instead, they click the "Logout" button in the navigation, and we immediately call the logout endpoint to clear their authentication tokens.

Our navigation component has a logout button that calls an `onLogout` callback. We'll implement this callback in the layout component that wraps our application.

```
// src/app/routes/layout.tsx

import { Outlet, useNavigate } from 'react-router';

import { Navigation } from '@/components/navigation';
import { useLogoutUserMutation } from '@/features/auth/api/logout';
import { useUser } from '@/features/auth/hooks/use-user';

export default function Layout() {
  const user = useUser();
  const navigate = useNavigate();
  const logoutUserMutation = useLogoutUserMutation({
    options: {
      onSuccess: () => navigate('/'),
    },
  });
  const handleLogout = async () => {
    logoutUserMutation.mutate();
  };

  return (
    <div>
      <Navigation user={user} onLogout={handleLogout} />
      <main className="min-h-screen bg-background">
        <Outlet />
      </main>
    </div>
  );
}
```

The layout component uses the `useUser` hook to get the current user and passes it to the navigation. When the user clicks the logout button, the `handleLogout` function is called, which triggers the logout mutation. After a successful logout, we navigate the user back to the home page.

Now let's implement the logout mutation.

```
// src/features/auth/api/logout.ts

export async function logoutUser(): Promise<LogoutUserResponse> {
  const response = await api.post<LogoutUserResponse>('/auth/logout', {
    body: {},
  });

  return zLogoutUserResponse.parse(response);
}

export function getLogoutUserMutationOptions() {
  return mutationOptions({
    mutationFn: logoutUser,
  });
}

export function useLogoutUserMutation({
  options,
}: {
  options?: Omit<
    UseMutationOptions<LogoutUserResponse, Error, void>,
    'mutationFn'
  >;
} = {}) {
  return useMutation({
    ...getLogoutUserMutationOptions(),
    ...options,
  });
}
```

The logout function calls the `/auth/logout` endpoint, which clears the authentication tokens from the cookies. After this, the user is no longer authenticated and will need to log in again to access protected pages.

But how does our application know if we are authenticated or not? We need a way to access the current user's information throughout our application.

Accessing the user in the app

Usually, we need to access the current user's information in many parts of our application. The navigation shows the user's name, the dashboard displays user-specific data, and we need to decide which features to show based on user permissions. Instead of fetching the user in every component, we'll fetch it once and make it available throughout the application.

We'll use React Router's middleware to load the user on the server and make it available to all components. First, we need a function that fetches the current user's data from the API.

```
// src/features/auth/api/get-me.ts

export async function getMe(
  cookieHeader?: string,
): Promise<GetCurrentUserResponse> {
  const response = await api.get<GetCurrentUserResponse>('/auth/me', {
    headers: {
      Cookie: cookieHeader ?? '',
    },
  });
  return zGetCurrentUserResponse.parse(response);
}
```

This function calls the `/auth/me` endpoint to get the current user's data. It accepts an optional `cookieHeader` parameter because we'll call this on the server where cookies need to be passed explicitly.

Now we need to fetch this user data and make it available throughout our application using middleware. First, we need to enable middleware in our application.

```
// react-router.config.ts

import type { Config } from '@react-router/dev/config';

export default {
  // ...
  future: {
    v8_middleware: true,
  },
} satisfies Config;
```

This is required by React Router as this is a new feature that will be available in the next major version of React Router by default.

Now let's create the user middleware:

```ts
// src/app/middleware/user.ts

import { createContext, type MiddlewareFunction } from 'react-router';

import { getMe } from '@/features/auth/api/get-me';
import type { CurrentUser } from '@/types/generated/types.gen';

export const userContext = createContext<CurrentUser | null>();

function hasAuthCookies(request: Request): boolean {
  const cookieHeader = request.headers.get('Cookie');
  if (!cookieHeader) return false;

  return (
    cookieHeader.includes('accessToken=') ||
    cookieHeader.includes('refreshToken=')
  );
}

export const userMiddleware: MiddlewareFunction = async (
  { request, context },
  next,
) => {
  try {
    const existingUser = context.get(userContext);

    if (existingUser !== undefined) {
      return next();
    }
  } catch {
  }

  if (!hasAuthCookies(request)) {
    context.set(userContext, null);
    return next();
  }

  try {
    const cookieHeader = request.headers.get('Cookie') || '';
    const user = await getMe(cookieHeader);
    context.set(userContext, user);
  } catch {
    context.set(userContext, null);
  }

  return next();
};
```

This middleware runs on every request and fetches the current user if authentication cookies are present. Let's break down how it works:

- **Check for existing user** : If the user is already in the context, we skip the fetch to avoid duplicate requests.
- **Check for auth cookies** : We only attempt to fetch the user if auth cookies are present in the request.
- **Fetch and store user** : If cookies exist, we fetch the user data and store it in the context.
- **Handle errors gracefully** : If fetching fails (invalid token, network error, etc.), we set the user to null instead of crashing the application.

The middleware stores either the user object or null in React Router's request context (not to be confused with React's Context API), so downstream code always knows whether a user is authenticated.

To activate this middleware, we need to add it to the middleware array in our root file.

```
// src/app/root.tsx

export const middleware = [userMiddleware];

// ...
```

By adding `userMiddleware` to the middleware array, it will run on every request before any loaders execute. This ensures the user is available in the context when we need it.

Now we can access the user from the context in our root loader.

```
// src/app/root.tsx

export async function loader({ context }: Route.LoaderArgs) {
  const user = context.get(userContext);
  return data({ user });
}
```

The root loader gets the user from the context and returns it. This makes the user data available to all components through React Router's loader data mechanism.

With the user data loaded in the root, we can create a custom hook to access it from any component.

```
// src/features/auth/hooks/use-user.ts

import { useRouteLoaderData } from 'react-router';

import type { RootLoaderData } from '@/types/app-context';

export function useUser() {
  const rootData = useRouteLoaderData<RootLoaderData>('root');
  return rootData?.user ?? null;
}
```

The `useUser` hook gets the root loader data and extracts the user from it. This hook can be called from any component in our application to access the current user.

Here's how we use it in practice.

```
// src/app/routes/layout.tsx
import { useUser } from '@/features/auth/hooks/use-user';

export default function Layout() {
  const user = useUser();

  // ...
}
```

Any component can now call `useUser()` to get the current user. If the user is authenticated, it returns their data. If not, it returns null.

We have authentication working and can access the user throughout our app, but we still need to protect certain routes so that only authenticated users can access them.

Protecting routes

Some routes in our application should only be accessible to authenticated users. For example, the dashboard pages don't make sense for users who aren't logged in. We need a way to protect these routes and redirect unauthenticated users to the login page.

We can create a middleware that checks if a user is authenticated. If not, it redirects them to the login page before the route even loads.

```ts
// src/app/middleware/protected.ts
import { redirect, type MiddlewareFunction } from 'react-router';

import { userContext } from './user';

export const protectedMiddleware: MiddlewareFunction = async (
  { context },
  next,
) => {
  const user = context.get(userContext);

  if (!user) {
    throw redirect('/auth/login');
  }

  return next();
};
```

This middleware is simple but powerful. It gets the user from the context and checks if they're authenticated. If there's no user, it throws a redirect to the login page, which stops the request from continuing. If there is a user, it calls `next()` to continue processing the request.

The order of middleware execution matters. The user middleware must run before the protected middleware because the protected middleware needs to access the user that was loaded by the user middleware.

Now we can apply this middleware to protect specific routes. Since our dashboard routes are all nested under the dashboard layout, we can add the middleware there to protect all of them at once.

```ts
// src/app/routes/dashboard/layout.tsx

export const middleware = [protectedMiddleware];

export default function DashboardLayout() {
  // ...
}
```

By adding `protectedMiddleware` to the dashboard layout, all routes nested under the dashboard will check for authentication. If a user tries to access the dashboard without being logged in, they'll be redirected to the login page.

With authentication in place, we know who our users are and can protect entire routes. However, knowing who a user is doesn't tell us what they're

allowed to do in the application. This is where authorization comes in.

Authorization

Authentication tells us who users are, but authorization determines what they can do. These are two different concepts that work together. Authentication answers "Are you who you say you are?" while authorization answers "Are you allowed to do this action?"

For example, authentication lets us know that a user is logged in as John. Authorization then determines whether John can edit a particular idea. Maybe John can only edit his own ideas, not ideas created by other users. This kind of permission checking is authorization.

Without authorization, authenticated users could perform any action in the application. They could delete other users' content, modify data they don't own, or access features they shouldn't have access to. Authorization rules define the boundaries of what each user can do.

In our application, we need authorization for several scenarios:

- Users can only edit and delete their own ideas
- Users can't review their own ideas
- Users can only review each idea once
- Users can only edit their own profile

We'll implement authorization using a policy-based approach. Policies are functions that take the current user and a resource, then return whether the user has permission to perform an action on that resource.

Let's create authorization policies for the different resources in our application.

```
// src/features/auth/lib/authorization-policies.ts

import type {
  CurrentUser,
  Idea,
  Review,
  User,
} from '@/types/generated/types.gen';

export const IdeaPolicies = {
  canCreate: (currentUser: CurrentUser | null): boolean => {
```

```
    const isAuthenticated = !!currentUser;
    return isAuthenticated;
  },

  canEdit: (currentUser: CurrentUser | null, idea: Idea): boolean => {
    const isIdeaAuthor = currentUser?.id === idea.authorId;
    return isIdeaAuthor;
  },

  canDelete: (currentUser: CurrentUser | null, idea: Idea): boolean => {
    const isIdeaAuthor = currentUser?.id === idea.authorId;
    return isIdeaAuthor;
  },
};

export const ReviewPolicies = {
  canCreate: (
    currentUser: CurrentUser | null,
    idea: Idea,
    existingReviews?: Review[],
  ): boolean => {
    const isAuthenticated = !!currentUser;
    if (!isAuthenticated) return false;

    const isIdeaAuthor = currentUser.id === idea.authorId;
    if (isIdeaAuthor) return false;

    const hasAlreadyReviewed = existingReviews?.some(
      (review) => review.authorId === currentUser.id,
    );
    if (hasAlreadyReviewed) return false;

    return true;
  },

  canEdit: (currentUser: CurrentUser | null, review: Review): boolean => {
    const isReviewAuthor = currentUser?.id === review.authorId;
    return isReviewAuthor;
  },

  canDelete: (currentUser: CurrentUser | null, review: Review): boolean => {
    const isReviewAuthor = currentUser?.id === review.authorId;
    return isReviewAuthor;
  },
};

export const ProfilePolicies = {
  canEdit: (currentUser: CurrentUser | null, user: User): boolean => {
    const isUserAuthor = currentUser?.id === user.id;
    return isUserAuthor;
```

```
  },
};
```

These authorization policies encapsulate our business rules for who can do what. Each policy is a simple function that returns true or false based on the current user and the resource being accessed.

The policies follow a clear pattern:

- **Idea policies** : Users must be logged in to create ideas and can only edit or delete ideas they authored
- **Review policies** : Users must be logged in, can't review their own ideas, and can't review the same idea twice
- **Profile policies** : Users can only edit their own profile

By centralizing these rules in one place, we make them easy to update and maintain. If we need to change who can do what, we just update the policy functions.

Now we need a way to use these policies in our components. We'll create a hook that wraps the authorization policies and makes them easy to use throughout our application.

```
// src/features/auth/hooks/use-authorization.ts

import type { Idea, Review, User } from '@/types/generated/types.gen';

import {
  IdeaPolicies,
  ProfilePolicies,
  ReviewPolicies,
} from '../lib/authorization-policies';

import { useUser } from './use-user';

export function useAuthorization() {
  const currentUser = useUser();

  return {
    // Ideas
    canCreateIdea: () => IdeaPolicies.canCreate(currentUser),
    canEditIdea: (idea: Idea) => IdeaPolicies.canEdit(currentUser, idea),
    canDeleteIdea: (idea: Idea) => IdeaPolicies.canDelete(currentUser, idea),

    // Reviews
    canCreateReview: (idea: Idea, reviews?: Review[]) =>
```

```
      ReviewPolicies.canCreate(currentUser, idea, reviews),
    canEditReview: (review: Review) =>
      ReviewPolicies.canEdit(currentUser, review),
    canDeleteReview: (review: Review) =>
      ReviewPolicies.canDelete(currentUser, review),

    // Profile
    canEditProfile: (user: User) => ProfilePolicies.canEdit(currentUser, user),
  };
}
```

The `useAuthorization` hook provides a convenient way to check permissions in our components. It gets the current user with the `useUser` hook and passes it to the appropriate policy functions. This gives us a clean API for authorization checks throughout our application.

Instead of importing policy functions and the current user in every component, we can just call `useAuthorization()` and get all the permission checks we need.

Let's see how we use this hook in a real component.

```
// src/app/routes/dashboard/ideas/ideas.tsx

import { Plus } from 'lucide-react';
import { Link } from 'react-router';

import { Button } from '@/components/ui/button';
import { useAuthorization } from '@/features/auth/hooks/use-authorization';
import { useCurrentUserIdeasQuery } from '@/features/ideas/api/get-current-user-ideas';
import { IdeasList } from '@/features/ideas/components/ideas-list';

import type { Route } from './+types/ideas';

export default function MyIdeasPage() {
  const ideasQuery = useCurrentUserIdeasQuery();
  const ideas = ideasQuery.data?.data;

  const { canCreateIdea } = useAuthorization();

  return (
    <div className="container mx-auto px-4 py-8">
      <div className="max-w-4xl mx-auto space-y-6">
        <div className="flex items-center justify-between">
          <div>
            <h1 className="text-3xl font-bold mb-2">My Ideas</h1>
            <p className="text-muted-foreground">
              Manage and track your submitted ideas
```

```
            </p>
          </div>
          {canCreateIdea() && (
            <Link to="/dashboard/ideas/new">
              <Button>
                <Plus className="mr-2 h-4 w-4" />
                New Idea
              </Button>
            </Link>
          )}
        </div>

        <IdeasList
          ideas={ideas}
          isLoading={ideasQuery.isLoading}
          emptyMessage="You haven't created any ideas yet."
          error={ideasQuery.error}
        />
      </div>
    </div>
  );
}
```

In this example, we use the `useAuthorization` hook to check if the current user can create ideas. Based on that check, we conditionally render the "**New Idea**" button. If the user doesn't have permission, the button simply doesn't appear in the UI.

This pattern works throughout our application. We check permissions before showing action buttons, before navigating to edit pages, and in forms before enabling submit buttons. The UI adapts to what each user is allowed to do.

It's important to note that authorization checks in the UI are about user experience, not security. They prevent users from seeing buttons for actions they can't perform, which avoids confusion and error messages. However, we must also enforce authorization on the server side in our API. A determined user could bypass client-side checks, so the backend must verify permissions before allowing any action on the server side.

With authentication and authorization in place, users can log in and the application knows what each user can do. But we still need to protect against security threats that can affect any web application, regardless of authentication.

Securing the application

Authentication tells us who users are, but security is about protecting them and our application from malicious attacks. Even with perfect authentication, our application can be vulnerable to attacks that steal data, hijack user sessions, or execute malicious code.

Web applications face many security threats, but two of the most common are Cross-Site Scripting (XSS) attacks and missing security headers. XSS attacks inject malicious scripts into our application, while missing security headers leave our application vulnerable to various browser-based attacks. Let's look at how to protect against these threats.

Content sanitization

Cross-Site Scripting (XSS) is one of the most common web security vulnerabilities. It happens when an attacker injects malicious JavaScript code into our application, which then executes in other users' browsers. For example, if a user can submit content that includes `< script >` tags and we display that content without sanitization, the script will run in every user's browser who views it.

This is particularly dangerous because the malicious script runs with the same permissions as our application. It can read cookies, access local storage, make API requests on behalf of the user, or redirect users to malicious sites. Even though we're using httpOnly cookies for authentication (which JavaScript can't access), XSS attacks can still perform actions as the logged-in user.

In our application, users can create ideas with descriptions in markdown format. Markdown gets converted to HTML for display, which means we're rendering HTML that came from user input. If we don't sanitize this HTML, users could inject malicious scripts into their markdown content.

We'll use DOMPurify to sanitize the HTML before rendering it. DOMPurify removes any potentially dangerous content like `< script >` tags, inline event handlers, and dangerous attributes while keeping safe HTML elements like headings, lists, and links.

```
// src/components/markdown-renderer.tsx
```

```tsx
import DOMPurify from 'isomorphic-dompurify';
import { useMemo } from 'react';
import { remark } from 'remark';
import remarkGfm from 'remark-gfm';
import remarkHtml from 'remark-html';

export type MarkdownRendererProps = {
  content: string;
  className?: string;
};

export function MarkdownRenderer({
  content,
  className = '',
}: MarkdownRendererProps) {
  const htmlContent = useMemo(() => {
    try {
      // Process markdown to HTML
      const result = remark()
        .use(remarkGfm) // GitHub Flavored Markdown
        .use(remarkHtml, { sanitize: false })
        .processSync(content);

      const sanitizedHtml = DOMPurify.sanitize(result.toString(), {
        ALLOWED_TAGS: [
          'p',
          'br',
          'strong',
          'em',
          'u',
          's',
          'code',
          'pre',
          'h1',
          'h2',
          'h3',
          'h4',
          'h5',
          'h6',
          'ul',
          'ol',
          'li',
          'blockquote',
          'a',
          'table',
          'thead',
          'tbody',
          'tr',
          'th',
          'td',
```

```
            'hr',
          ],
          ALLOWED_ATTR: ['href', 'title', 'class'],
          ALLOW_DATA_ATTR: false,
          ALLOWED_URI_REGEXP:
            /^(?:(?:(?:f|ht)tps?|mailto):|[^a-z]|[a-z+.-]+(?:[^a-z+.-:]|$))/i,
        });

        return sanitizedHtml;
      } catch (error) {
        console.error('Error processing markdown:', error);
        return `<p>Error rendering markdown content</p>`;
      }
    }, [content]);

    return (
      <div
        className={`prose prose-sm max-w-none dark:prose-invert ${className}`}
        dangerouslySetInnerHTML={{ __html: htmlContent }}
      />
    );
  }
```

Let's break down how this component protects against XSS:

- **Two-step processing** : First, we convert markdown to HTML using remark. Then, we sanitize the HTML with `DOMPurify` . We intentionally disable remark's built-in sanitization because `DOMPurify` does a better job.
- **Strict allowlist** : We only allow specific HTML tags that are safe and necessary for text content. Tags like `<script>` , `<iframe>` , and `<object>` are automatically blocked.
- **Limited attributes** : We only allow safe attributes like `href` , `title` , and `class` . Event handlers like `onclick` are blocked.
- **Safe protocols** : For links, we only allow `http:` , `https:` , and `mailto:` protocols. This prevents `javascript:` links that could execute code.

The result is that users can write rich markdown content with formatting, links, and lists, but they can't inject malicious scripts or dangerous HTML elements.

Content sanitization protects against XSS attacks, but there's another layer of security we need: security headers.

Security headers

Security headers are special HTTP headers that tell the browser how to behave when handling our application. They're like rules that the browser follows to protect users from various attacks. Without these headers, browsers use default behaviors that aren't always secure.

Modern browsers support many security headers that can prevent attacks like clickjacking, MIME-type sniffing, and cross-site scripting. By setting these headers on all our responses, we create multiple layers of defense that work together to protect our application and users.

Let's implement a function that applies all the necessary security headers to our responses.

```
// src/lib/security-headers.ts

export function applySecurityHeaders(
  responseHeaders: Headers,
  nonce: string,
): void {
  const isProd = (process.env.NODE_ENV as string) === 'production';
  const apiUrl = process.env.VITE_API_URL as string | undefined;
  const securityHeaders = getSecurityHeaders(isProd, apiUrl, nonce);

  Object.entries(securityHeaders).forEach(([key, value]) => {
    responseHeaders.set(key, value);
  });
}

function getSecurityHeaders(
  isProd: boolean,
  apiUrl?: string,
  nonce?: string,
): Record<string, string> {
  return {
    'X-Frame-Options': 'DENY',
    'X-Content-Type-Options': 'nosniff',
    'Referrer-Policy': 'strict-origin-when-cross-origin',

    ...(isProd
      ? { 'Strict-Transport-Security': 'max-age=63072000; includeSubDomains' }
      : {}),

    'Content-Security-Policy': [
      "default-src 'self'",
      `script-src 'self' 'nonce-${nonce}'
```

```
      `style-src 'self' 'unsafe-inline' https://fonts.googleapis.com`,
      "font-src 'self' https://fonts.gstatic.com",
      "img-src 'self' data: https:",
      `connect-src 'self' ${apiUrl}`,
    ].join('; '),
  };
}
```

Here's what each security header does:

- `X-Frame-Options: DENY` : Prevents our site from being embedded in an iframe. This stops clickjacking attacks where attackers overlay invisible iframes on top of buttons to trick users into clicking them.
- `X-Content-Type-Options: nosniff` : Prevents browsers from guessing the content type of files. Without this, browsers might execute a file as JavaScript even if we said it was an image, which could lead to XSS attacks.
- `Referrer-Policy: strict-origin-when-cross-origin` : Controls what information is sent in the `Referer` header. This balances privacy (not leaking full URLs to other sites) with functionality (allowing same-origin analytics).
- `Strict-Transport-Security` : Forces browsers to only use HTTPS connections to our site. This prevents attackers from intercepting traffic by downgrading users to HTTP. We only enable this in production since we use HTTP in development.
- `Content-Security-Policy` : This is the most powerful security header. It controls where the browser can load resources from, which we'll explain in detail next.

The **Content-Security-Policy** (**CSP**) header tells the browser which sources are safe to load resources from. Without CSP, the browser will load scripts, styles, images, and other resources from anywhere. CSP creates an allowlist of trusted sources.

Our CSP has several directives:

- `default-src 'self'` : By default, only load resources from our own origin.
- `script-src 'self' 'nonce-...'` : Only load scripts from our origin or scripts with a matching nonce.

- `style-src 'self' 'unsafe-inline' https://fonts.googleapis.com` : Load styles from our origin, allow inline styles (needed for some libraries), and allow Google Fonts.
- `font-src 'self' https://fonts.gstatic.com` : Load fonts from our origin and Google Fonts CDN.
- `Img-src 'self' data: https:` Load images from our origin, data URIs (for inline images), and any HTTPS source.
- `Connect-src 'self' [API_URL]` :Only make API requests to our origin and our API server.

The nonce (number used once) in the `script-src` directive is important. Instead of allowing all inline scripts with `unsafe-inline` , we generate a unique random value for each request and only allow scripts with that specific nonce. This prevents injected scripts from running even if an attacker manages to insert them into our HTML. To learn more about nonce in CSP you can check out `https://content-security-policy.com/nonce` .

To implement nonces, we need to generate a new random nonce for each request and use it both in the CSP header and in our script tags. Let's create a middleware that generates nonces.

```
// src/app/middleware/nonce.ts

import { randomBytes } from 'node:crypto';

import { createContext, type MiddlewareFunction } from 'react-router';

export function generateNonce(): string {
  return randomBytes(16).toString('base64');
}

export const nonceContext = createContext<string>();

export const nonceMiddleware: MiddlewareFunction = async (
  { context },
  next,
) => {
  const nonce = generateNonce();
  context.set(nonceContext, nonce);

  return next();
};
```

This middleware generates a random nonce for each request and stores it in the context. The nonce will be used both in the CSP header and in our script tags so that the browser knows our scripts are legitimate.

Now we need to apply the security headers to every response that our application sends.

```
// src/app/entry.server.tsx

import { applySecurityHeaders } from '@/lib/security-headers';

import { nonceContext } from './middleware/nonce';

export const streamTimeout = 5_000;

export default function handleRequest(
  request: Request,
  responseStatusCode: number,
  responseHeaders: Headers,
  routerContext: EntryContext,
  loadContext: RouterContextProvider,
) {
  return new Promise((resolve, reject) => {
    // ...

    const nonce = loadContext.get(nonceContext);

    applySecurityHeaders(responseHeaders, nonce);

    // ...

    const { pipe, abort } = renderToPipeableStream(
      <ServerRouter context={routerContext} url={request.url} nonce={nonce} />,
      {
        nonce,
        // ...
      },
    );
  });
}
```

In the `entry.server.tsx` file, we handle server-side rendering of our React application. This is where we have access to the response headers before they're sent to the browser, making it the perfect place to apply our security headers.

The key changes are:

- **Get the nonce** : We retrieve the nonce from the context that was set by our nonce middleware
- **Apply security headers** : We call `applySecurityHeaders` to add all security headers to the response
- **Pass nonce to React** : We pass the nonce to the `ServerRouter` component so it can be used in script tags throughout our application

The nonce is passed to React's rendering function, which will automatically add it to the script tags that React generates. This ensures that our application's JavaScript can run while blocking any injected scripts.

Finally, we need to make the nonce available in our root layout so we can add it to the Scripts and `ScrollRestoration` components that React Router provides.

```
// src/app/root.tsx

import { QueryClientProvider } from '@tanstack/react-query';
import { NuqsAdapter } from 'nuqs/adapters/react-router/v7';
import { useState } from 'react';
import {
  data,
  isRouteErrorResponse,
  Links,
  Meta,
  Outlet,
  Scripts,
  ScrollRestoration,
  useLoaderData,
} from 'react-router';

import { Notifications } from '@/components/notifications';
import { createQueryClient } from '@/lib/react-query';

import type { Route } from './+types/root';
import './app.css';
import { nonceContext, nonceMiddleware } from './middleware/nonce';
import { userContext, userMiddleware } from './middleware/user';

export const middleware = [nonceMiddleware, userMiddleware];

export async function loader({ context }: Route.LoaderArgs) {
  const user = context.get(userContext);
  const nonce = context.get(nonceContext);
  return data({ user, nonce });
}
```

```
// ...

export function Layout({ children }: { children: React.ReactNode }) {
  const { nonce } = useLoaderData<typeof loader>();

  return (
    <html lang="en">
      <head>
        <meta charSet="utf-8" />
        <meta name="viewport" content="width=device-width, initial-scale=1" />
        <Meta />
        <Links />
      </head>
      <body>
        {children}
        <ScrollRestoration nonce={nonce} />
        <Scripts nonce={nonce} />
      </body>
    </html>
  );
}
```

In the root layout, we load the nonce from the loader data and pass it to the `ScrollRestoration` and `Scripts` components. These components will add the nonce attribute to their script tags, allowing them to execute under our Content Security Policy.

The middleware executes in order: first `nonceMiddleware` generates the nonce, then `userMiddleware` loads the user. Both values are stored in the context and loaded in the root loader, making them available to the entire application.

With these security measures in place, our application is protected against common web vulnerabilities. The combination of httpOnly cookies, content sanitization, and security headers creates multiple layers of defense that work together to keep our users safe.

Summary

In this chapter, we implemented a complete authentication and authorization system and secured our application against common threats. We started by extending our API client to handle authentication tokens automatically, including automatic token refresh when access tokens expire. This provides a seamless user experience where users stay logged in without interruption.

We built registration, login, and logout flows that use httpOnly cookies to store authentication tokens securely. These cookies can't be accessed by JavaScript, protecting them from XSS attacks. We created middleware to load the current user on every request and made the user available throughout the application with a simple `useUser` hook.

To protect sensitive routes, we implemented a protected middleware that redirects unauthenticated users to the login page. By adding this middleware to layout components, we can protect entire sections of our application with a single line of code.

We implemented authorization policies to control what authenticated users can do. These policies define who can create, edit, and delete different resources in our application. The `useAuthorization` hook makes these policies easy to use in components, allowing us to show or hide UI elements based on user permissions.

Beyond authentication and authorization, we implemented security measures to protect against attacks. We used DOMPurify to sanitize user-generated HTML content, preventing XSS attacks while still allowing users to create rich markdown content. We added comprehensive security headers including Content Security Policy with nonces to create multiple layers of defense against various browser-based attacks.

These security practices aren't just nice to have—they're essential for any production application. Authentication tells us who users are, authorization controls what they can do, and security measures protect both our users and our application from malicious actors.

Get this book's PDF version and more

Scan the QR code (or go to `packtpub.com/unlock`). Search for this book by name, confirm the edition, and then follow the steps on the page.

UNLOCK NOW

Note: Keep your invoice handy. Purchases made directly from Packt don't require an invoice.

8

Improving Application Performance

Performance is one of the most important aspects of any web application. When users interact with our application, they expect it to respond as soon as possible. Delays of even a few hundred milliseconds can feel sluggish and frustrating.

For every application, performance matters in user experience. Users are more likely to engage with content and complete actions when the application responds quickly. Slow applications can impact business metrics and lose users. Even small delays can significantly impact conversion rates and user engagement.

Fortunately, with the right architectural decisions, we can build applications that feel much faster to users.

We'll cover the following topics:

- Detecting performance issues
- Optimizing components
- Code splitting and lazy loading
- Streaming content from the server
- Debouncing user input
- Handling large data sets with pagination and virtualization
- Optimistic updates

By the end of this chapter, we'll have implemented several performance optimizations that make our application feel snappy and responsive, even when dealing with slow networks or large amounts of data.

Technical requirements

Before we get started, we need to set up our project. To be able to develop our project, we'll need the following things installed on our computer:

- Node.js version 24 or above, npm version 11 or above ships with Node. We can confirm that by executing `node -v` and `npm -v` in the terminal. There are multiple ways to install Node.js and npm. Here is a great article that goes into more detail: https://www.nodejsdesignpatterns.com/blog/5-ways-to-install-node-js .
- VS Code (optional) is a popular editor for JavaScript and TypeScript: open source, solid TypeScript support, and extensions. It can be downloaded from https://code.visualstudio.com .

The code for this book is available at the book's repo. To access the repository link, follow the steps in the *"Download the example code files"* section in the *Preface* . Clone it and enter the repository root:

```
git clone https://github.com/PacktPublishing/React-Application-Architecture-for-Production-Second-Edition.git
```

The repository contains chapter folders with the code for each chapter, plus a shared `api` folder with the API server used across all chapters.

We are on *Chapter 8* , so we need to navigate to the `chapter-08` directory:

```
cd React-Application-Architecture-for-Production-Second-Edition/chapter-08
```

Then we need to install dependencies:

```
npm install
```

We also need to provide the environment variables:

```
cp .env.example .env
```

Now we should have the frontend running at http://localhost:5173 .

We also need to have our API server running.

Let's open a new terminal window and navigate to the `api` directory:

```
cd React-Application-Architecture-for-Production-Second-Edition/api
```

Now we need to run the setup script for *C hapter 8* to configure everything for us:

```
npm run setup 08
```

Then we need to run the API server:

```
npm run dev
```

We should see the API server running on `http://localhost:9999` .

For more information about the setup details, check out the `README.md` file.

Detecting performance issues

Before we can fix performance problems, we need to identify them. React applications can suffer from unnecessary re-renders, which waste CPU cycles and make the UI feel unresponsive. Understanding when and why components re-render is crucial for building fast React applications.

React DevTools is the best way to understand how our application renders and identify components that might be causing slowdowns. It's a browser extension created by the React team that gives us insight into what React is doing behind the scenes.

The **React DevTools Profiler** tab lets us record a session and see exactly which components rendered, how long each render took, and what triggered the render. To use it, we open the **Profiler** tab, click the record button, interact with our application, and then stop recording. The profiler shows us a flame graph of renders, where wider bars indicate longer render times.

The profiler output looks like this:

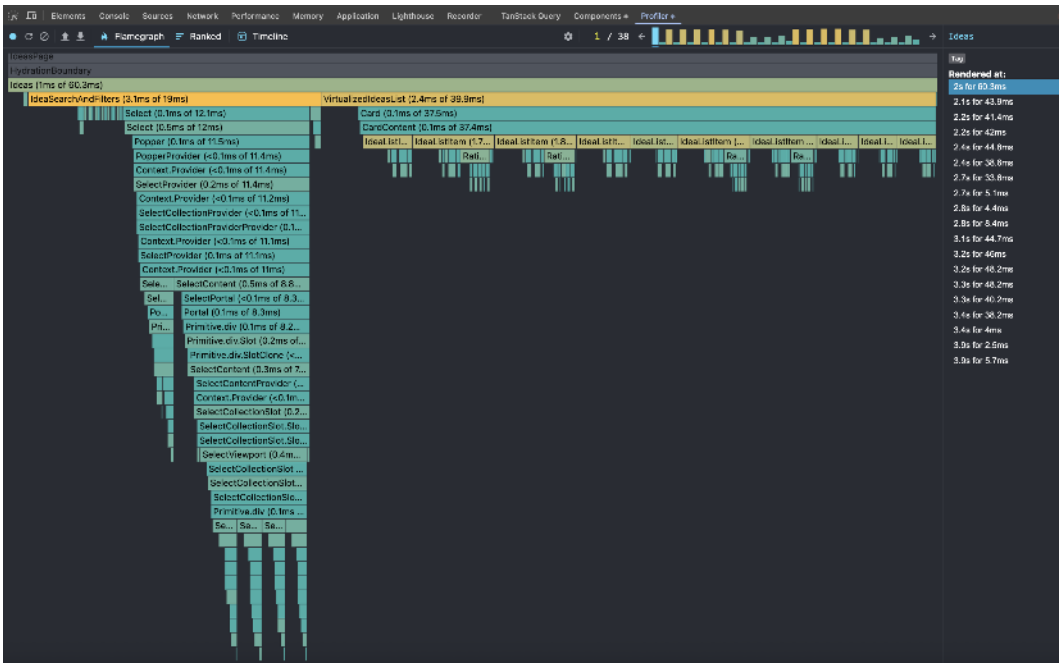


Figure 8.1 – Profiler output

When looking at the profiler output, we can see which components are rendered, how much time rendering took, and how many times they rendered. We should focus on three things:

- **Components that render frequently** - If a component re-renders too often, it might be doing unnecessary work
- **Components with long render times** - These are the most impactful to optimize since they block the UI for longer
- **Cascading renders** - When one component update causes many child components to re-render, we might have a state management issue

The key insight is that not all re-renders are bad. React is designed to re-render efficiently. We should only optimize when we see actual performance problems. Premature optimization can make code harder to understand without providing real benefits.

Optimizing components

Since React is all about components, we need to have the right approach when it comes to building and composing them together for optimal performance. The way we structure our components directly impacts how often they re-render and how much work React must do.

Here are some important techniques for optimizing components for best performance:

- Colocate state
- Memoize expensive calculations
- Use component composition

Let's take a look at a practical example of how to apply these techniques.

Colocating state

State colocation means keeping state as close as possible to where it's used. If only one component needs a piece of state, that state should live in that component, not in a parent. This is one of the most impactful performance optimizations because it directly controls which components re-render.

Consider a dashboard that has a user profile section and an ideas list. Here's a common mistake where all state lives in one component:

```
function Dashboard() {
  const [user, setUser] = useState({ name: '', email: '' });
  const [ideas, setIdeas] = useState([
    { id: 1, title: 'Smart City Noise Monitoring' },
    { id: 2, title: 'AI-Powered Personal Assistant' },
  ]);
  const [searchTerm, setSearchTerm] = useState('');

  const filteredIdeas = ideas.filter(idea =>
    idea.title.toLowerCase().includes(searchTerm.toLowerCase())
  );

  return (
    <div>
      <div>
        <h2>Profile</h2>
        <input
          value={user.name}
          onChange={e => setUser({ ...user, name: e.target.value })}
          placeholder="Name"
        />
        <input
          value={user.email}
          onChange={e => setUser({ ...user, email: e.target.value })}
          placeholder="Email"
        />
      </div>
```

```
      <div>
        <h2>My Ideas</h2>
        <input
          value={searchTerm}
          onChange={e => setSearchTerm(e.target.value)}
          placeholder="Search ideas..."
        />
        {filteredIdeas.map(idea => (
          <div key={idea.id}>{idea.title}</div>
        ))}
      </div>
    </div>
  );
}
```

What's wrong with this? Every time we type in the profile name field, the entire component re-renders, including the ideas list. When we search for ideas, the profile inputs re-render too. Everything affects everything because all state lives in one place.

We can fix this by splitting into focused components where each piece of state lives in the component that uses it. Look how each component manages its own state:

```
function UserProfile() {
  const [user, setUser] = useState({ name: '', email: '' });

  return (
    <div>
      <h2>Profile</h2>
      <input
        value={user.name}
        onChange={e => setUser({ ...user, name: e.target.value })}
        placeholder="Name"
      />
      <input
        value={user.email}
        onChange={e => setUser({ ...user, email: e.target.value })}
        placeholder="Email"
      />
    </div>
  );
}

function IdeasList() {
  const [ideas] = useState([
    { id: 1, title: 'Smart City Noise Monitoring' },
    { id: 2, title: 'AI-Powered Personal Assistant' },
```

```
  ]);
  const [searchTerm, setSearchTerm] = useState('');

  const filteredIdeas = ideas.filter(idea =>
    idea.title.toLowerCase().includes(searchTerm.toLowerCase())
  );

  return (
    <div>
      <h2>My Ideas</h2>
      <input
        value={searchTerm}
        onChange={e => setSearchTerm(e.target.value)}
        placeholder="Search ideas..."
      />
      {filteredIdeas.map(idea => (
        <div key={idea.id}>{idea.title}</div>
      ))}
    </div>
  );
}

function Dashboard() {
  return (
    <div>
      <UserProfile />
      <IdeasList />
    </div>
  );
}
```

Now state is properly colocated. Let's break down what changed:

- `user` state lives in `UserProfile` - Only the profile section needs this data, so it owns the state
- `ideas` and `searchTerm` live in `IdeasList` - The ideas section manages its own data and search functionality
- `Dashboard` has no state - It's purely a composition component that arranges the layout

The result? When we type in the search box, only `IdeasList` re-renders. When we edit our profile, only `UserProfile` re-renders. The parent `Dashboard` never re-renders because it has no state. State colocation directly prevents unnecessary re-renders by keeping components isolated from each other's changes.

Memoizing expensive calculations

With our state properly colocated, our components are optimized for most scenarios. But what if a component performs an expensive calculation that doesn't need to run on every render?

Let's say our ideas list needs to calculate statistics about the filtered results, and we also want to let users toggle between list and grid views:

```
function IdeasList() {
  const [ideas] = useState([
    { id: 1, title: 'Smart City Noise Monitoring' },
    { id: 2, title: 'AI-Powered Personal Assistant' },
  ]);
  const [searchTerm, setSearchTerm] = useState('');
  const [viewMode, setViewMode] = useState<'list' | 'grid'>('list');

  const filteredIdeas = ideas.filter(idea =>
    idea.title.toLowerCase().includes(searchTerm.toLowerCase())
  );

  const stats = calculateIdeaStatistics(filteredIdeas);

  return (
    <div>
      <h2>My Ideas</h2>
      <div>
        <input
          value={searchTerm}
          onChange={e => setSearchTerm(e.target.value)}
          placeholder="Search ideas..."
        />
        <button onClick={() => setViewMode(viewMode === 'list' ? 'grid' : 'list')}>
          {viewMode === 'list' ? 'Grid View' : 'List View'}
        </button>
      </div>

      <div>
        <p>Total: {stats.total}</p>
        <p>Average length: {stats.averageLength}</p>
      </div>

      <div className={viewMode === 'grid' ? 'grid grid-cols-2 gap-4' : ''}>
        {filteredIdeas.map(idea => (
          <div key={idea.id}>{idea.title}</div>
        ))}
      </div>
    </div>
```

```
  );
}
```

The problem here is that `calculateIdeaStatistics` runs on every render, even when it doesn't need to. When the user toggles the view mode, the component re-renders, but the filtered ideas haven't changed, so recalculating the statistics is wasted work. We can fix this by memoizing the expensive calculation with `useMemo`:

```
function IdeasList() {
  const [ideas] = useState([
    { id: 1, title: 'Smart City Noise Monitoring' },
    { id: 2, title: 'AI-Powered Personal Assistant' },
  ]);
  const [searchTerm, setSearchTerm] = useState('');
  const [viewMode, setViewMode] = useState<'list' | 'grid'>('list');

  const filteredIdeas = useMemo(() => {
    return ideas.filter(idea =>
      idea.title.toLowerCase().includes(searchTerm.toLowerCase())
    );
  }, [ideas, searchTerm]);

  const stats = useMemo(() => {
    return calculateIdeaStatistics(filteredIdeas);
  }, [filteredIdeas]);

  // ...
}
```

Now when the user toggles the view mode, React sees that `ideas` and `searchTerm` haven't changed, so it skips the filtering and returns the cached `filteredIdeas`. Similarly, since `filteredIdeas` is the same reference, the expensive `calculateIdeaStatistics` calculation is skipped too.

NOTE

We should not memoize everything. `useMemo` has overhead, so we should only use it for calculations that are actually expensive. To make this decision, we need to determine if we have another piece of state in our component that is irrelevant to the expensive calculation. In the context of our example, the viewMode causes re-renders without affecting the memoized value of `stats`. We can use the React DevTools Profiler to identify which calculations are causing slow re-renders before reaching for `useMemo`.

Using component composition

Component composition allows us to build complex UIs from simple, reusable pieces while maintaining excellent performance characteristics. When done correctly, composition gives us automatic optimization through stable prop references.

Let's say we want to add a visual indicator when the user profile section is focused. A common approach is to add focus tracking directly to the component:

```
function UserProfile() {
  const [user, setUser] = useState({ name: '', email: '' });
  const [isFocused, setIsFocused] = useState(false);

  return (
    <div
      style={{ border: isFocused ? '2px dashed blue' : 'none' }}
      onFocus={() => setIsFocused(true)}
      onBlur={() => setIsFocused(false)}
    >
      <h2>Profile</h2>
      <input
        value={user.name}
        onChange={e => setUser({ ...user, name: e.target.value })}
        placeholder="Name"
      />
      <input
        value={user.email}
        onChange={e => setUser({ ...user, email: e.target.value })}
        placeholder="Email"
      />
    </div>
  );
}
```

The problem with this approach is that focus changes re-render the entire profile form, and form input changes re-render the focus handling logic. These are unrelated concerns that shouldn't affect each other.

We can use composition to separate these concerns:

```
function FocusTracker({ children }) {
  const [isFocused, setIsFocused] = useState(false);

  return (
    <div
```

```
      style={{ border: isFocused ? '2px dashed blue' : 'none' }}
      onFocus={() => setIsFocused(true)}
      onBlur={() => setIsFocused(false)}
    >
      {children}
    </div>
  );
}

function UserProfile() {
  const [user, setUser] = useState({ name: '', email: '' });

  return (
    <div>
      <h2>Profile</h2>
      <input
        value={user.name}
        onChange={e => setUser({ ...user, name: e.target.value })}
        placeholder="Name"
      />
      <input
        value={user.email}
        onChange={e => setUser({ ...user, email: e.target.value })}
        placeholder="Email"
      />
    </div>
  );
}

function Dashboard() {
  return (
    <div>
      <FocusTracker>
        <UserProfile />
      </FocusTracker>
      <IdeasList />
    </div>
  );
}
```

Now, when we focus on a section, only `FocusTracker` re-renders to update its border style. Since the `UserProfile` component manages its own state independently, focus changes in the `FocusTracker` component don't trigger re-renders in `UserProfile` because the two components are isolated from each other's state updates.

This composition pattern gives us two optimization benefits:

- **Props optimization** - The `children` prop is created once in the parent and passed down. Even though `FocusTracker` re-renders when focus changes, React doesn't re-render the children because the prop reference is stable.
- **Rendering optimization** - Each component only re-renders when its own state changes. Focus updates are isolated to `FocusTracker` and form changes are isolated to `UserProfile` .

When components are properly structured with colocated state, memoized calculations, and thoughtful composition, they naturally re-render less often. This is why good component architecture is a great step towards optimizing the application performance.

These were some basic React optimizations. Now let's look at some more advanced techniques that go beyond React for optimizing different aspects of our application.

Code splitting and lazy loading

When users visit our application for the first time, the browser downloads all the JavaScript code before the page becomes interactive. For large applications, this can be several megabytes of code. But here's the thing: users might not need all the features of the application, so why make them download code they'll never use?

Code splitting allows us to break our JavaScript bundle into smaller chunks that load on demand. Instead of downloading all the code upfront, users only download what they need. This is especially valuable for large applications, where users might never visit certain pages.

We should at least code split at the route level, so that the code for each page is loaded only when the user navigates to it. The good news is that React Router in framework mode automatically code-splits routes. However, we should still consider code splitting and lazy loading heavy components within the pages, like complex charts, editors, or features that many users won't use.

React provides the `lazy` function for loading components on demand. Here's a simple example:

```
import { lazy, Suspense } from 'react';
```

```
const HeavyComponent = lazy(() => import('@/components/heavy-component'));

function Ideas() {
  const [showHeavyComponent, setShowHeavyComponent] = useState(false);

  return (
    <div>
      <button onClick={() => setShowHeavyComponent(true)}>View Heavy Component</button>
      <Suspense fallback={<div>Loading...</div>}>
{showHeavyComponent&&<HeavyComponent />}
      </Suspense>
    </div>
  );
}
```

The `lazy` function takes a function that returns a dynamic import. When the component is first rendered, React will load the JavaScript file containing `HeavyComponent`. The `Suspense` component shows a fallback while the code is loading.

Note that we should only code split and lazy load components that are actually heavy, since we don't want to make additional chunk requests without a significant benefit. To understand the size of the chunks, we can use the analyzer plugin for Vite. To get the report, we can run the `analyze` command and then open the report in the browser.

```
npm run analyze
```

This will open a report in the browser.

Figure 8.2 – Bundle analyzer report

Here we can explore which parts of the application are heavy and should be code split and lazy loaded.

Streaming content from the server

Server-side rendering (SSR) improves initial page load by rendering HTML on the server. However, traditional SSR typically has a limitation: the server must wait for all data before sending anything to the browser. If one API call is slow, the entire page is delayed.

This creates a frustrating experience. Imagine a page that shows an idea and its reviews. The idea data loads in 100 ms, but the reviews take 2 seconds. With traditional SSR, users wait 2 seconds to see anything, even though the idea was ready almost immediately.

The solution is to stream the content as it arrives. Fast content appears immediately, while slower content loads progressively in the background.

Let's implement streaming for the idea detail page. The idea data loads first because users need to see it immediately. Reviews can load afterward since they are less critical for the user to see.

```
// src/app/routes/ideas/idea.tsx

export async function loader({ params }: Route.LoaderArgs) {
  const idea = await getIdeaById(params.id);
  const reviewsPromise = getReviewsByIdea(params.id);

  return routerData({
    idea,
    reviewsPromise,
  });
}
```

Notice the difference in how we handle the two pieces of data. We `await` the idea, which means the server waits for it before sending anything. But we don't `await` the reviews; we pass the promise directly to the component. The `reviewsPromise` is sent to the component as an unresolved promise. React's streaming renderer detects the pending promise through the Suspense boundary and starts sending the available HTML to the browser immediately, without waiting for the reviews to resolve.

In the component, we use React's `Suspense` to handle the loading state for the reviews:

```
export default function IdeaDetailPage({
  params,
  loaderData,
}: Route.ComponentProps) {
  const ideaId = params.id as string;

  const ideaQuery = useIdeaByIdQuery({
    id: ideaId,
    options: {
      ...(loaderData?.idea && { initialData: loaderData.idea }),
    },
  });
  const idea = ideaQuery?.data ?? loaderData?.idea;

  return (
    <div className="container mx-auto px-4 py-8">
      <Seo
        title={`${idea?.title || 'Idea'} | AIdeas`}
        description={idea?.shortDescription || 'View idea details'}
      />
      <div className="max-w-4xl mx-auto">
        <IdeaDetails idea={idea} />
        <Suspense fallback={<ReviewsSkeleton />}>
          <IdeaReviews
            idea={idea}
            reviewsPromise={loaderData.reviewsPromise}
          />
        </Suspense>
      </div>
    </div>
  );
}
```

The `Suspense` component wraps the reviews section. While the reviews promise is still pending, React shows the `ReviewsSkeleton` fallback. When the data arrives, the skeleton is seamlessly replaced with the actual reviews.

But how does the `IdeaReviews` component access the data from the promise? React provides the `use` hook for exactly this purpose:

```
function IdeaReviews({
  idea,
  reviewsPromise,
}: {
  idea: Idea;
```

```
  reviewsPromise: Promise<ReviewListResponse>;
}) {
  const reviewsData = use(reviewsPromise);

  // Now we can use reviewsData to render the reviews
  // ...
}
```

The `use` hook from React unwraps the promise. When the promise resolves, React re-renders the component with the data. If the promise is still pending, `use` suspends the component, which is why we need the `Suspense` wrapper above. It's worth noting that the use hook is available from React 19. Alternatively, we can use the `Await` component from React Router to unwrap the promise in a similar way:

```
import { Await } from "react-router";

function IdeaReviews({
  idea,
  reviewsPromise,
}: {
  idea: Idea;
  reviewsPromise: Promise<ReviewListResponse>;
}) {
  return (
    <Await promise={reviewsPromise}>
      {({ data }) => (
        // Use data to render the reviews
      )}
    </Await>
  );
}
```

With streaming SSR, the idea details appear immediately while a skeleton shows where reviews will appear. When the reviews finish loading, they seamlessly replace the skeleton. This makes the page feel faster because users can start reading the idea while reviews load in the background.

Debouncing user input

When users type in a search input, each keystroke can trigger a new API request. If someone searches for education-related ideas and types " *Education* " quickly, that's nine characters and most likely nine separate network

requests. Most of these requests are wasted because the user hasn't finished typing yet.

We can see this problem in action from the network tab in the browser.

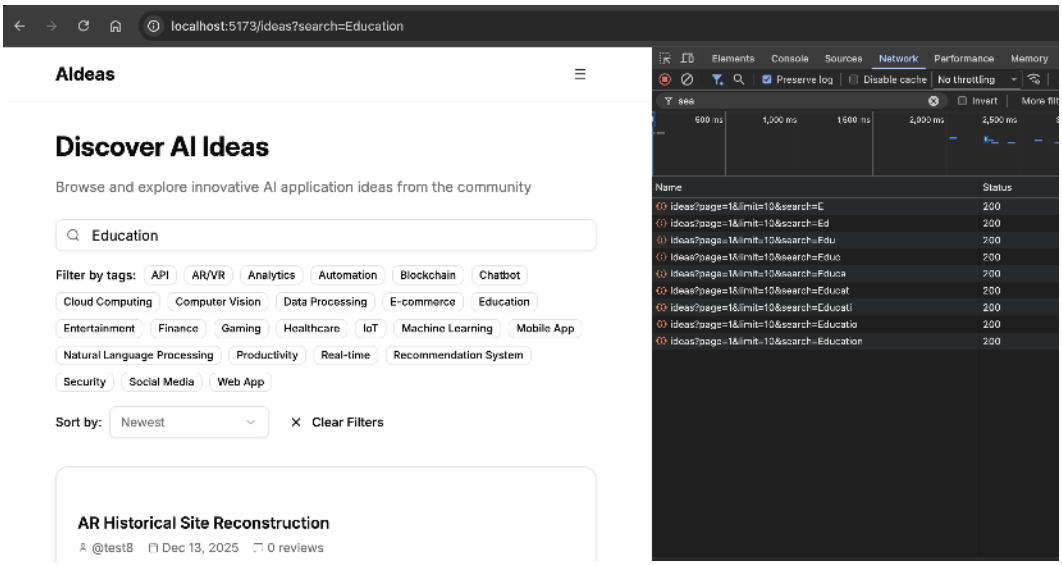


Figure 8.3 – Network requests while typing without debouncing

As we can see, there are 9 network requests being made while the user is typing. This wastes bandwidth, puts unnecessary load on the server, and can cause race conditions where responses arrive out of order. Imagine if the response for "Edu" arrives after the response for "Education"; the UI would show the wrong results, which is a bad user experience.

Debouncing solves this by waiting for a pause in user input before making the request. Instead of firing on every keystroke, we wait until the user stops typing for a specified duration (usually 300–500 ms), then make a single request with the final value.

To debounce the input, let's start by creating a reusable `useDebouncedValue` hook:

```
// src/hooks/use-debounced-value.ts
import { useEffect, useState } from 'react';

export function useDebouncedValue<T>(value: T, delay: number): T {
  const [debouncedValue, setDebouncedValue] = useState<T>(value);

  useEffect(() => {
    const timer = setTimeout(() => {
      setDebouncedValue(value);
```

```
    }, delay);

    return () => clearTimeout(timer);
  }, [value, delay]);

  return debouncedValue;
}
```

The hook accepts any value and a delay in milliseconds. Here's how it works: when the value changes, it starts a timer. If the value changes again before the timer completes, the timer resets. Only when the value stays stable for the full delay does the debounced value update.

Now let's use this hook in our search and filters hook:

```
// src/features/ideas/hooks/use-search-and-filters.ts

export function useSearchAndFilters(debounceMs: number = 500) {
  const [urlSearchTerm, setUrlSearchTerm] = useQueryState(
    'search',
    parseAsString.withDefault(''),
  );

  const debouncedSearchTerm = useDebouncedValue(urlSearchTerm, debounceMs);

  return {
    searchTerm: urlSearchTerm,
    debouncedSearchTerm,
    params: {
      ...(debouncedSearchTerm && { search: debouncedSearchTerm }),
      // ...other params
    },
    setSearchTerm: setUrlSearchTerm,
    // ...other actions
  };
}
```

We're returning two different values here, and this is important. The `searchTerm` updates immediately when users type, keeping the input field responsive; users see their characters appear instantly. The `debouncedSearchTerm` only updates after the user pauses, and we use this debounced value in our `params` object for API requests.

In the component, the search input binds directly to `searchTerm`:

```
// src/features/ideas/components/idea-search-and-filters.tsx
```

```
export function IdeaSearchAndFilters() {
  const {
    searchTerm,
    setSearchTerm,
    // ...
  } = useSearchAndFilters();

  return (
    <div className="space-y-4">
      <div className="relative">
        <Search className="absolute left-3 top-1/2 transform -translate-y-1/2 h-4 w-4 text-muted-
foreground" />
        <Input
          placeholder="Search ideas by title or description..."
          value={searchTerm}
          onChange={(e) => setSearchTerm(e.target.value)}
          className="pl-10"
        />
      </div>
      {/* ... */}
    </div>
  );
}
```

The input shows what users type instantly, but the `params` object used by the query only updates after the user stops typing for 500 ms:

```
// src/app/routes/ideas/ideas.tsx

function Ideas() {
  const { params } = useSearchAndFilters();

  const ideasInfiniteQuery = useIdeasInfiniteQuery({
    params: params as GetAllIdeasData['query'],
  });

  // ...
}
```

After implementing debouncing, we can see the difference in the network tab.

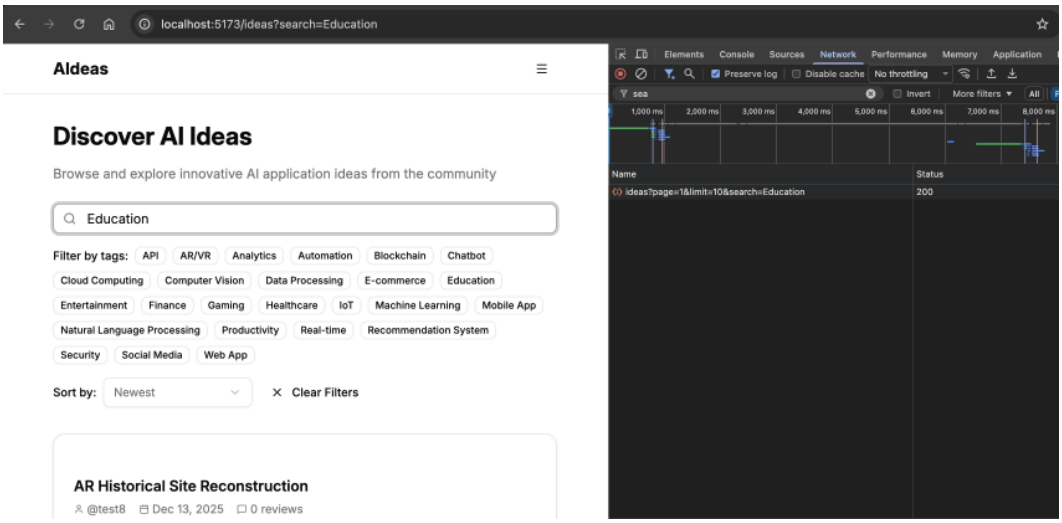


Figure 8.4 – Network requests while typing with debouncing

As we can see, there is only one network request being made after the user stops typing for 500 ms. Debouncing is one of the simplest optimizations we can make, but the impact is significant. This reduces API calls from nine to one per search, improving both client performance and server load.

Large data set optimization

When dealing with large data sets, performance can become a bottleneck for both the server and the client. The server must process potentially thousands of records, increasing response time and bandwidth usage. The client must render all those records, which can be slow and memory-intensive, especially on mobile and lower-end devices.

We can address these challenges with two techniques: pagination, which reduces the amount of data transferred, and virtualization, which reduces the number of list items rendered on the page.

Pagination

Instead of returning all ideas at once, we can implement infinite pagination to fetch a set of ideas, then fetch more ideas as the user scrolls or clicks a "**Load More**" button. This way, users start with a small, fast-loading set of data and can load more when they need it.

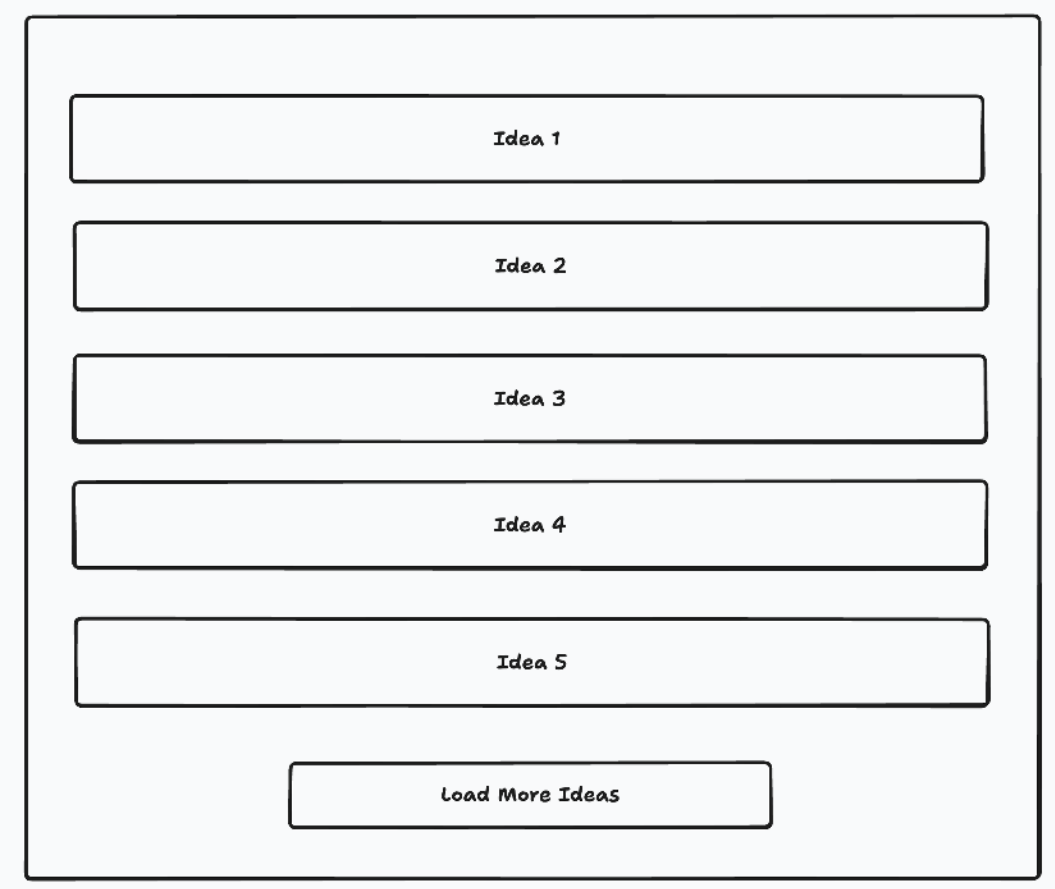


Figure 8.5 – Infinite pagination

React Query provides the `useInfiniteQuery` hook specifically for this pattern. It manages multiple pages of data and knows when there are more pages available. Let's update our ideas API to support infinite loading.

```
// src/features/ideas/api/get-ideas.ts

export async function getIdeas(
  params?: GetAllIdeasData['query'],
): Promise<GetAllIdeasResponse> {
  const validatedData = zGetAllIdeasData.parse({ query: params });

  const response = await api.get<GetAllIdeasResponse>('/ideas', {
    params: { ...validatedData.query, limit: 10 },
  });

  return zGetAllIdeasResponse.parse(response);
}

export function getIdeasInfiniteQueryOptions(
  params?: GetAllIdeasData['query'],
```

```
) {
  return infiniteQueryOptions({
    queryKey: ideasQueryKeys.list(params),
    queryFn: ({ pageParam }) =>
      getIdeas({ ...params, page: pageParam.toString() }),
    getNextPageParam: ({ pagination }) => pagination.nextPage,
    initialPageParam: 1,
  });
}

export function useIdeasInfiniteQuery({
  params,
  options,
}: {
  params?: GetAllIdeasData['query'];
  options?: Omit<
    ReturnType<typeof getIdeasInfiniteQueryOptions>,
    'queryKey' | 'queryFn'
  >;
}) {
  return useInfiniteQuery({
    ...getIdeasInfiniteQueryOptions(params),
    ...options,
  });
}
```

Let's break down the key parts:

- `limit: 10` – We request only 10 ideas per page instead of all ideas at once. This keeps each request fast and the response small.
- `getNextPageParam` – The function extracts the next page number from the API response. React Query uses this to know when there are more pages to load. If it returns `undefined` , there are no more pages.
- `initialPageParam: 1` - The first page to load when the query runs.

Now let's use this in the ideas page:

```
// src/app/routes/ideas/ideas.tsx

export async function loader({ request }: Route.LoaderArgs) {
  // ...

  await Promise.all([
    queryClient.prefetchInfiniteQuery(
      getIdeasInfiniteQueryOptions(ideasParams),
    ),
    queryClient.prefetchQuery(getTagsQueryOptions()),
  ]);
```

```
  return routerData({
    dehydratedState: dehydrate(queryClient),
  });
}

function Ideas() {
  const { params } = useSearchAndFilters();

  const ideasInfiniteQuery = useIdeasInfiniteQuery({
    params: params as GetAllIdeasData['query'],
  });

  const allIdeas =
    ideasInfiniteQuery.data?.pages.flatMap((page) => page.data) || [];

  return (
    <div className="container mx-auto px-4 py-8">
      <Seo
        title="Discover AI Ideas | AIdeas"
        description="Browse and explore innovative AI application ideas from the community"
      />
      <div className="mb-8">
        <h1 className="text-3xl font-bold mb-4">Discover AI Ideas</h1>
        <p className="text-muted-foreground mb-6">
          Browse and explore innovative AI application ideas from the community
        </p>
        <IdeaSearchAndFilters />
      </div>

      <IdeasList
        ideas={allIdeas}
        isLoading={ideasInfiniteQuery.isLoading && allIdeas.length === 0}
        emptyMessage="No ideas available yet"
        error={ideasInfiniteQuery.error}
      />

      {allIdeas.length > 0 && ideasInfiniteQuery.hasNextPage && (
        <div className="flex justify-center mt-8">
          <Button
            onClick={() => ideasInfiniteQuery.fetchNextPage()}
            disabled={ideasInfiniteQuery.isFetchingNextPage}
          >
            {ideasInfiniteQuery.isFetchingNextPage
              ? 'Loading more ideas...'
              : 'Load more ideas'}
          </Button>
        </div>
      )}
    </div>
```

```
  );
}
```

The `pages` array contains all the pages we've loaded so far. Each page has its own `data` array of ideas. We use `flatMap` to combine all the ideas from all pages into a single array that we can pass to our list component.

When the user clicks "Load more ideas", `fetchNextPage` loads the next page and adds it to the `pages` array. React Query handles all the caching and state management for us. The `hasNextPage` boolean tells us whether there are more pages to load, so we can hide the button when we've reached the end.

Pagination improves server performance and initial load time, but as users load more pages, we can end up with hundreds of items in the DOM. Let's explore how we can optimize this.

List virtualization

Even with pagination, users might load 10, 20, or 50 pages of data. If each page has 10 items, that's 500 DOM elements, each with its own event listeners, layout calculations, and memory usage. On slower devices, this can make the page feel sluggish. This can be demonstrated by inspecting the DOM in the browser.

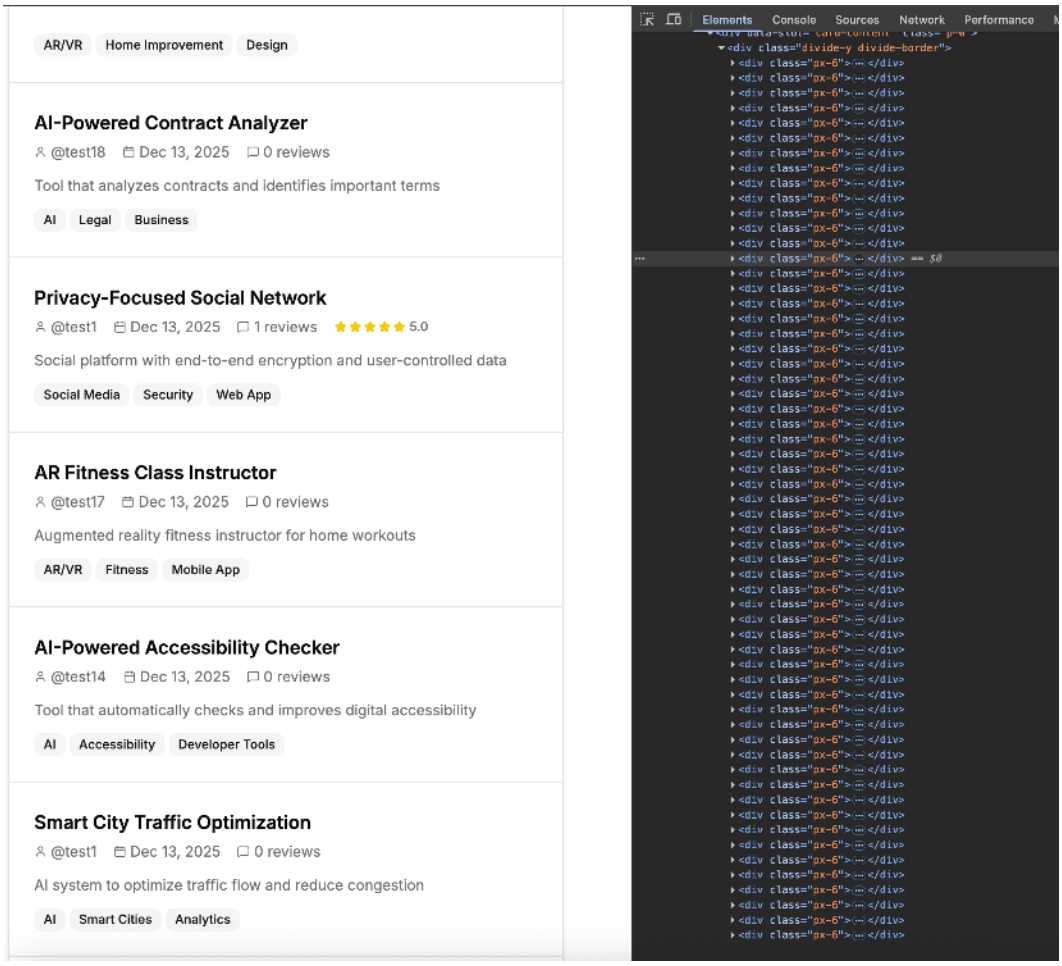


Figure 8.6 – DOM inspection

As we can see, there are more DOM elements being rendered than the user can see at a time. This can be a performance bottleneck, especially on slower devices. This is where list virtualization comes in.

List virtualization is a technique that only renders items currently visible in the viewport. Instead of creating DOM elements for all 500 items in a list, we might only render the 10 that are visible, plus a few extra for smooth scrolling. As the user scrolls, items are mounted and unmounted as they enter and leave the viewport.

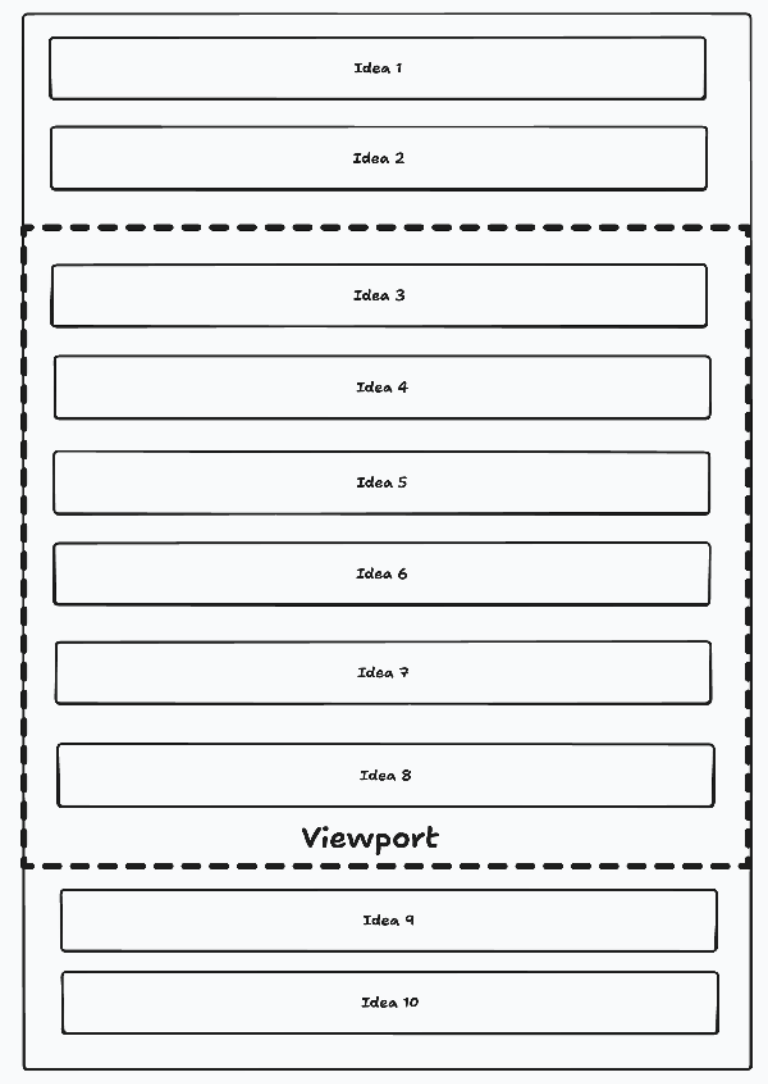


Figure 8.7 – List virtualization

We'll use the `@tanstack/react-virtual` library to implement this. The library handles all the complex scroll math and element positioning for us. Here's a virtualized version of our ideas list:

```
// src/features/ideas/components/virtualized-ideas-list.tsx

import { useWindowVirtualizer } from '@tanstack/react-virtual';

export function VirtualizedIdeasList({
  ideas,
  isLoading,
  emptyMessage,
  error,
}: VirtualizedIdeasListProps) {
```

```
  const virtualizer = useWindowVirtualizer({
    count: ideas?.length || 0,
    estimateSize: () => 150,
    overscan: 5,
  });

  // ...

  return (
    <Card>
      <CardContent className="p-0">
        <div
          style={{
            height: `${virtualizer.getTotalSize()}px`,
            width: '100%',
            position: 'relative',
          }}
        >
          {virtualizer.getVirtualItems().map((virtualItem) => {
            const idea = ideas[virtualItem.index];
            return (
              <div
                key={virtualItem.key}
                data-index={virtualItem.index}
                ref={virtualizer.measureElement}
                style={{
                  position: 'absolute',
                  top: 0,
                  left: 0,
                  width: '100%',
                  transform: `translateY(${virtualItem.start}px)`,
                }}
              >
                <div className="px-6 border-b border-border last:border-b-0">
                  <IdeaListItem idea={idea} />
                </div>
              </div>
            );
          })}
        </div>
      </CardContent>
    </Card>
  );
}
```

The `useWindowVirtualizer` hook takes three key options:

- `count` - The total number of items in the list. The virtualizer needs to know this to calculate the total scroll height.

- `estimateSize` - A function that returns the estimated height of each item. This doesn't need to be exact, as the virtualizer will measure actual sizes as items render. A good estimate just helps with initial layout and prevents layout shifts and jumpy scrolling during initial renders.
- `overscan` - How many items to render outside the visible area. A higher value makes scrolling smoother (items are ready before they become visible) but renders more items. 5 is a good starting point.

The virtualizer gives us several methods that make the magic happen:

- `getTotalSize()` - Returns the total height needed to contain all items. We use this for the container height so the scrollbar is the right size.
- `getVirtualItems()` - Returns only the items that should be rendered right now. This is the key to virtualization, instead of mapping over all 500 ideas, we only map over the 15-20 that are visible.
- `measureElement` - Measure each item's actual size with a ref callback. This is important because item heights might vary.

Since virtualization is a client-side optimization that depends on scroll position, we need to use the regular list during server-side rendering because the server doesn't know what's visible in the viewport or what the scroll position is. We can conditionally use the virtualized list on the client after hydration:

```
// src/app/routes/ideas/ideas.tsx

function Ideas() {
  // ...

  const isClient = useIsClient();

  const ListComponent = isClient ? VirtualizedIdeasList : IdeasList;

  return (
    <div className="container mx-auto px-4 py-8">
      {/* ... */}

      <ListComponent
        ideas={allIdeas}
        isLoading={ideasInfiniteQuery.isLoading && allIdeas.length === 0}
        emptyMessage="No ideas available yet"
        error={ideasInfiniteQuery.error}
```

```
        />

        {/* ... */}
      </div>
    );
  }
```

We can verify virtualization is working by inspecting the DOM. Even with hundreds of ideas loaded, only a small number of DOM elements exist.

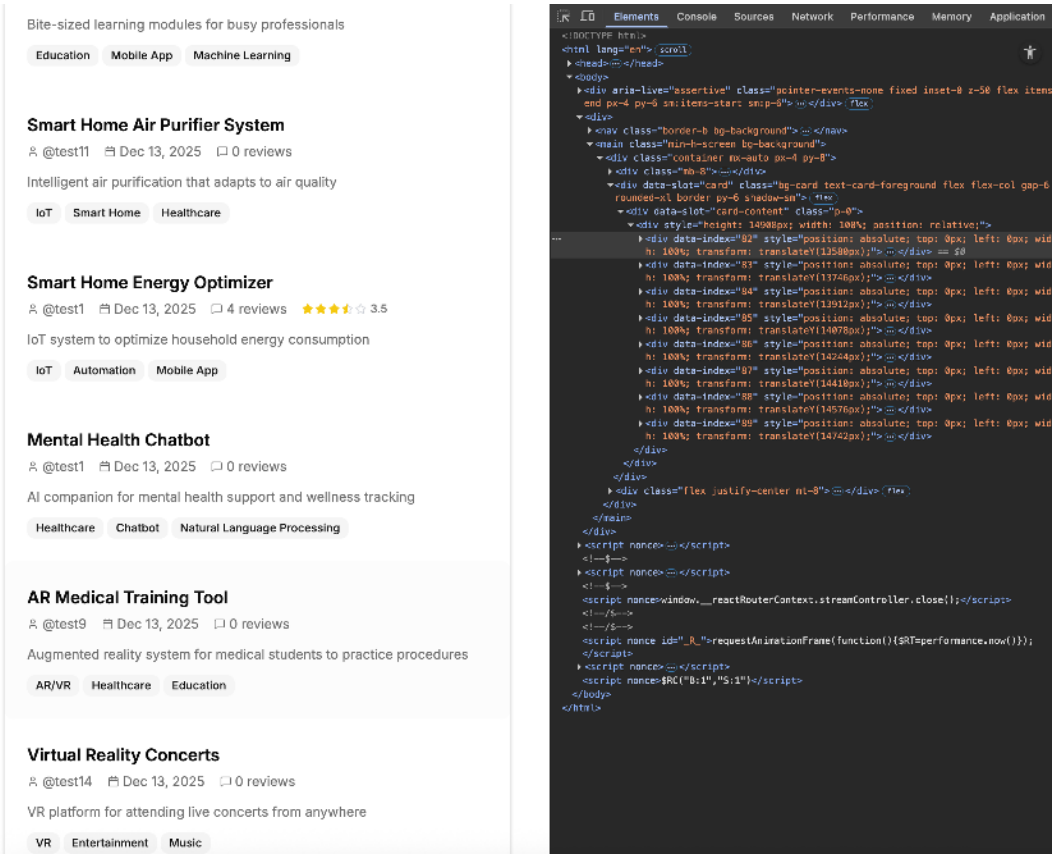


Figure 8.8 – List virtualization DOM

As we can see, only the items visible in the viewport are rendered. The rest are created as the user scrolls, keeping the DOM lean regardless of how many items are in the list.

Optimistic updates

When a user submits a form, they typically have to wait for the server to respond before seeing their changes. This round trip can take some time depending on the network conditions and the server response time,

potentially making the application feel slow. Think about posting a review: we click submit, see a loading spinner, wait for the server, and finally see our review appear. It works, but it sometimes feels sluggish.

Optimistic updates solve this by immediately showing the expected result before the server confirms it. When a user submits a review, we show it right away as if it already succeeded. In most cases, the server confirms the action moments later, and the user never notices the difference. The application feels instant.

The approach has three parts:

- Update immediately - When the user submits, update the UI as if the request succeeded
- Handle errors - If the request fails, roll back to the previous state and show an error
- Sync with server - When the request succeeds, replace optimistic data with real server data

Let's implement optimistic updates for creating reviews.

First, we handle the optimistic update in `onMutate` . This callback runs before the request is sent to the server:

```
// src/features/reviews/api/create-review.ts

export function useCreateReviewMutation({
  options,
  onSuccess,
  onError,
  ideaId,
}: {
  options?: Omit<
    UseMutationOptions<CreateReviewResponse, Error, CreateReviewData['body']>,
    'mutationFn'
  >;
  onSuccess: () => void;
  onError: () => void;
  ideaId: string;
}) {
  const queryClient = useQueryClient();
  const user = useUser();

  return useMutation({
    ...getCreateReviewMutationOptions(),
    ...options,
```

```
    onMutate: async (newReview) => {
      // Cancel any outgoing refetches to prevent them from overwriting our optimistic update
      await queryClient.cancelQueries({
        queryKey: reviewsQueryKeys.byIdea(ideaId),
      });
      await queryClient.cancelQueries({
        queryKey: reviewsQueryKeys.current(),
      });

      // Snapshot the previous values
      const previousReviewsByIdea =
        queryClient.getQueryData<GetReviewsByIdeaResponse>(
          reviewsQueryKeys.byIdea(ideaId),
        );
      const previousCurrentReviews =
        queryClient.getQueryData<GetReviewsByIdeaResponse>(
          reviewsQueryKeys.current(),
        );

      // Optimistically update the cache with the new review
      if (user) {
        const optimisticReview: Review = {
          id: `temp-${Date.now()}`,
          content: newReview.content,
          rating: newReview.rating,
          authorId: user.id,
          author: {
            username: user.username,
            email: user.email,
            id: user.id,
          },
          ideaId,
          createdAt: new Date().toISOString(),
          updatedAt: new Date().toISOString(),
        };

        queryClient.setQueryData<GetReviewsByIdeaResponse>(
          reviewsQueryKeys.byIdea(ideaId),
          (old) => ({
            data: old?.data
              ? [optimisticReview, ...old.data]
              : [optimisticReview],
          }),
        );

        queryClient.setQueryData<GetReviewsByIdeaResponse>(
          reviewsQueryKeys.current(),
          (old) => ({
            data: old?.data
              ? [optimisticReview, ...old.data]
```

```
              : [optimisticReview],
          }),
        );
      }

      // Return context with the previous values for rolling back if the request fails
      return { previousReviewsByIdea, previousCurrentReviews };
    },
    // ...
  });
}
```

Let's break down what's happening here. The `onMutate` callback runs before the mutation starts, giving us a chance to update the UI immediately:

- Cancel in-flight queries - Cancel any queries that might be fetching reviews. If we don't do this, a pending refetch that happens after our optimistic update could overwrite it with stale data, causing the UI to briefly revert.
- Snapshot previous values - Save the current cache data so we can restore it if the request fails. This is our rollback plan.
- Create optimistic review - Build a review object that looks like what the server will return. Use a temporary ID because we don't know the real ID yet.
- Update the cache - Add the optimistic review to the cache. React Query will immediately re-render any components that use this data.
- Return context - Return the previous values so we can access them in `onError` if we need to roll back.

Next, we handle errors and success:

```
export function useCreateReviewMutation({
  // ...
}) {
  const queryClient = useQueryClient();
  const user = useUser();

  return useMutation({
    // ...
    onError: (_err, _newReview, context) => {
      // Rollback to the previous values on error
      if (context?.previousReviewsByIdea) {
        queryClient.setQueryData(
          reviewsQueryKeys.byIdea(ideaId),
```

```
            context.previousReviewsByIdea,
          );
        }
        if (context?.previousCurrentReviews) {
          queryClient.setQueryData(
            reviewsQueryKeys.current(),
            context.previousCurrentReviews,
          );
        }
        onError();
      },
      onSuccess: () => {
        // Refetch to get the server data with the real ID
        queryClient.invalidateQueries({
          queryKey: reviewsQueryKeys.byIdea(ideaId),
        });
        queryClient.invalidateQueries({
          queryKey: reviewsQueryKeys.current(),
        });
        onSuccess();
      },
    });
  }
```

If the request fails, we restore the previous cache data from the context we returned in `onMutate`. The optimistic review disappears, and we call `onError` so the component can show an error message.

If the request succeeds, we invalidate the queries to refetch from the server. This replaces our optimistic review (with its temporary ID) with the real review from the server. The user won't notice this swap because refetching happens in the background and the review stays in place, but now it has the correct ID and any server-side modifications.

The user experience is much better now. They see their review appear immediately when they submit. If something goes wrong, the review disappears and they see an error message. Most of the time, the server confirms the action, and users don't even notice the background refetch that swaps in the real data.

Summary

In this chapter, we explored several techniques for making our React application faster and more responsive. We learned how to use React

DevTools to identify performance issues and then addressed those issues with targeted optimizations.

We started with state management best practices: collocating state and keeping components focused. When state lives close to where it's used, fewer components re-render when it changes.

We then looked at code splitting and lazy loading to reduce initial bundle sizes. By loading code on demand, users only download what they need for the current page.

Streaming slow content during SSR allowed us to show content progressively instead of waiting for all data. Fast content appears immediately, while slower content loads in the background.

Debouncing reduced unnecessary API calls when users type in search fields. Instead of 9 requests for typing "Education," we make just 1 request after the user pauses.

For large data sets, we combined pagination with list virtualization. Pagination keeps API responses small and fast. Virtualization keeps the DOM lean by rendering only visible items, no matter how many items the user has loaded.

Finally, optimistic updates made mutations feel instant by showing changes before server confirmation. The UI updates immediately, and we sync with the server in the background.

These techniques complement each other. A well-optimized application uses the right tool for each situation: debouncing for user input, pagination for large lists, streaming for slow data, and optimistic updates for mutations. The goal is always the same: make the application feel instant and responsive to users.

Get this book's PDF copy, code bundle, and more

Scan the QR code (or go to `packtpub.com/unlock`). Search for this book by name, confirm the edition, and then follow the steps on the page.

UNLOCK NOW

Note: Have your invoice handy. Purchases made directly from the Packt website don ' t require an invoice.

9

Going International

If we want our application to reach users around the world, we need to support multiple languages. This is where internationalization comes in. You might see it abbreviated as i18n, since there are 18 letters between the "i" and "n" in "internationalization" .

When users can interact with an application in their own language, they feel more comfortable and are more likely to keep using it. But internationalization is about more than just translating text. We also need to think about how dates are formatted, how numbers are displayed, how pluralization works (one item vs. two items), and even the direction text flows. Some languages, like Arabic and Hebrew, read from right to left.

If we set up internationalization correctly from the start, adding new languages later becomes straightforward. We add translation files and the application just works.

We'll cover the following topics:

- Understanding internationalization architecture
- Setting up i18n in our application
- Using translations in the application
- Switching between languages in the application

By the end of this chapter, we'll have a fully internationalized application that supports multiple languages, with type-safe translations and a smooth user experience when switching between languages.

Technical requirements

Before we get started, we need to set up our project. To develop our project, we need the following tools installed on our computer:

- Node.js version 24 or above. npm version 11 or above ships with Node. We can confirm this by executing `node -v` and `npm -v` in the terminal. There are multiple ways to install Node.js and npm. Here is a helpful article that goes into more detail: https://www.nodejsdesignpatterns.com/blog/5-ways-to-install-node-js .
- VS Code (optional), a popular editor for JavaScript and TypeScript. It is open source, has solid TypeScript support, and offers many extensions. It can be downloaded from https://code.visualstudio.com .

The code for this book is available at https://github.com/PacktPublishing/React-Application-Architecture-for-Production-Second-Edition on GitHub. Clone it and enter the repository root:

```
git clone https://github.com/PacktPublishing/React-Application-Architecture-for-Production-Second-Edition.git
```

The repository contains chapter folders with the code for each chapter, along with a shared `api` folder that includes the API server used across all chapters.

We are working on *Chapter 9* , so navigate to the `chapter-09` directory:

```
cd React-Application-Architecture-for-Production-Second-Edition/chapter-09
```

Next, install the dependencies:

```
npm install
```

We also need to provide the environment variables:

```
cp .env.example .env
```

At this point, the frontend should be ready and running at http://localhost:5173 .

We also need to run the API server.

Open a new terminal window and navigate to the `api` directory:

```
cd React-Application-Architecture-for-Production-Second-Edition/api
```

Run the setup script for *Chapter 9* to configure everything:

```
npm run setup 09
```

Then start the API server:

```
npm run dev
```

The API server should now be running on `http://localhost:9999` .

For more information about the setup details, check out the `README.md` file.

Understanding internationalization architecture

The core idea behind internationalization is simple: we separate the text content from the code. Instead of writing "Welcome" directly in a component, we use a key like `welcome` , and the i18n system looks up the appropriate text based on the user's language.

Why do we do this? Think about what happens without this separation. If we hardcode "Welcome" in our component and later need to support Spanish, we'd have to go through every component, find every string, and add conditional logic to show the Spanish version. Or have a separate version of the application for every language. That would be messy, error-prone, and would not make sense at all.

With i18n, the component always uses translation keys like `t('welcome')` . The translation system handles the rest. When we add Spanish, we just create a Spanish translation file with the same keys. The components don't change at all.

Where to store translations

Before building our i18n system, we need to decide where our translations will live. This choice affects how translators work, how we deploy updates, and what tools we can use.

There are two main approaches:

- In-codebase translations store translation files directly in the source code repository, typically as JSON, YAML, or TypeScript files. This

approach is simple and works well for most projects. Translations get version controlled alongside the code, so when you add a feature, you add its translations in the same pull request. Changes to translations go through the same review process as code changes. The main limitation is that updating a translation requires a code deployment; you can't fix a typo without pushing a new version.

- External translation services like Lokalise, Crowdin, or Phrase separate translations from code. These platforms provide web interfaces where translators work without touching the codebase. They're valuable for larger teams, especially when working with professional translators who shouldn't need codebase access. Many offer features like translation memory, automated suggestions, and workflow management. However, they add complexity, cost, and integration overhead that may not be justified for smaller projects.

For our application, we're using in-codebase translations stored as TypeScript files. This gives us type safety (TypeScript can validate our translation keys), simplicity (no external dependencies), and tight integration with our development workflow. As our project grows, we could migrate to an external service if needed, but starting simple makes sense.

How our i18n system works

Now let's understand the complete flow of how internationalization works in our server-rendered React application. This will help you see how all the pieces we'll build fit together.

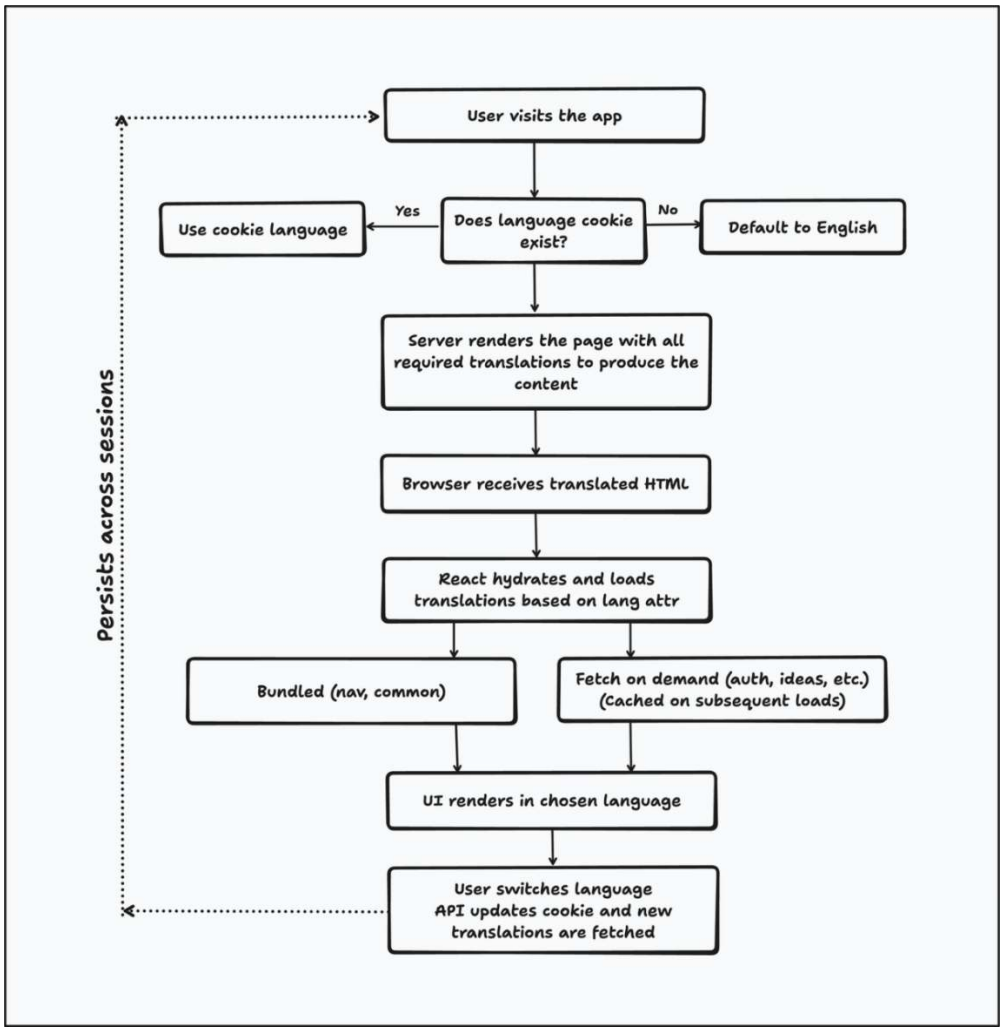


Figure 9.1 – i18n system flow

The diagram shows how translations are flowing through our application:

Language detection and storage

When a user visits our application, we need to know which language to show them. The detection is straightforward: we check for a language preference cookie. If the cookie exists, we use that language. If it doesn't exist, we default to English.

Users explicitly choose their language using the language switcher we'll build. When they make a selection, we store it in a secure HTTP-only cookie. This means their choice persists across browser sessions and page reloads. Both the server and client read this same cookie, ensuring consistent language detection in both environments.

Server-side rendering with translations

When the server receives a request, it reads the language cookie before rendering the page. The server has access to all translation files, so it can immediately render the entire page in the correct language. This is important for SEO since search engines see translated content. It's also important for performance since users see content in their language instantly, without waiting for JavaScript to load and run.

The server sends fully translated HTML to the browser. If a Spanish user requests the home page, they receive HTML with "Bienvenido" already in it, not a key like `welcome` that needs to be translated client-side.

Client-side hydration

When the JavaScript loads in the browser, React needs to "hydrate"—attach event listeners and make the page interactive. For this to work without flashing or re-rendering, the client needs the same translations the server used.

We bundle the most commonly used translations directly in the JavaScript. These are translations that appear on every page, such as navigation, common, etc. They're available immediately when JavaScript initializes.

The client detects the current `lang` uage from the lang attribute, which is set on the server, and loads those bundled translations. Now, when React hydrates, it uses the same translation keys and gets the same text the server rendered. No flashing or layout shifts.

Loading translations dynamically

Not all translations are bundled with the JavaScript. If we bundled every translation for every feature in every language, the initial download would be huge. Most users never visit every page, so they would download translations they never use.

Instead, we load translations on demand. When a user navigates to the `auth` entication pages, the client fetches the auth namespace from our API. When they visit the `ideas` section, it fetches the ideas namespace. This happens automatically. We configure which namespaces each route needs, and react-i18next handles the fetching.

These translation files are cached aggressively, so after the first visit to a section, subsequent visits are instant.

Switching languages

When a user switches languages, we update the cookie through an API call and tell the client-side i18n system to change languages. If the new language's translations aren't loaded yet, they're fetched from the API. Once everything is ready, the entire UI re-renders in the new language. The cookie ensures that if the user reloads the page or comes back later, they see the same language.

This architecture gives us the best of both worlds: fast server-rendered pages with full translations for SEO and initial load, and efficient client-side updates that only load what's needed. The same components work in both environments because they always use translation keys, never hard-coded strings.

Now let's build this system piece by piece.

Setting up i18n in our application

Now that we understand the architecture, let's implement it. This involves organizing translations with namespaces, configuring react-i18next, serving translations via API, adding type safety, and setting up language and direction attributes.

Organizing translations with namespaces

As we add more features, our translation files could easily grow to thousands of lines. Finding and updating translations would become difficult. Namespaces solve this problem by allowing us to split translations into logical groups.

Instead of one massive file containing every translation, we create smaller files organized by feature or page. The home page has its own translations, authentication has its own, and so on. This makes translations easier to find and maintain. It also helps with performance, since we can load only the translations we actually need for the current page.

We organize our translations at two levels: app-level translations for things used everywhere and feature-scoped translations that live alongside their features.

App-level translations

For translations used throughout the application, we create files in the `app/locales` directory:

```
// src/app/locales/en/common.ts

export default {
  cancel: 'Cancel',
  delete: 'Delete',
  deleting: 'Deleting...',
  // ...
};
```

This `common` namespace contains translations that appear on many pages.

We can also create namespace files for specific pages:

```
// src/app/locales/en/home.ts

export default {
  meta: {
    description: 'A community platform for sharing and discovering AI ideas',
    title: 'AIdeas - Share and Discover AI Ideas',
  },
  subtitle: 'A community platform for sharing and discovering AI ideas',
  title: 'AIdeas - Share and Discover AI Ideas',
  // ...
};
```

We nest related translations together to group related translations. The `meta` object groups all metadata-related translations. When we need to find a translation, the nesting makes it obvious where to look.

Feature-scoped translations

For larger features, we keep translations close to the feature code itself. This is the same principle we use for organizing components and hooks, where we keep related things together:

```
// src/features/auth/locales/en.ts

export default {
  alreadyHaveAccount: 'Already have an account?',
  creatingAccount: 'Creating account...',
  dontHaveAccount: "Don't have an account?",
  // ...
};
```

By placing feature translations in `features/auth/locales`, we make the auth feature self-contained. If we ever need to extract this feature into a separate package or reuse it elsewhere, all its translations come along.

Combining translations

All these separate translation files need to come together so the i18n system can access them. We create an index file that combines everything:

```
// src/app/locales/en/index.ts

import type { ResourceLanguage } from 'i18next';

import authTranslations from '@/features/auth/locales/en';
import ideasTranslations from '@/features/ideas/locales/en';
import profileTranslations from '@/features/profile/locales/en';
import reviewsTranslations from '@/features/reviews/locales/en';

import about from './about';
import common from './common';
import components from './components';
import dashboard from './dashboard';
import home from './home';
import navigation from './navigation';
import notFound from './not-found';

export default {
  common,
  notFound,
  home,
  about,
  dashboard,
  navigation,
  components,
  auth: authTranslations,
  ideas: ideasTranslations,
  reviews: reviewsTranslations,
  profile: profileTranslations,
} satisfies ResourceLanguage;
```

This gives us clear namespaces: `common`, `home`, `auth`, `ideas`, and so on. When we use translations in components, we reference them as `common:cancel` or `auth:loginTitle`. The namespace prefix makes it clear where each translation comes from.

Configuring React-i18next

React-i18next is the React integration for i18next, which is a very popular i18n library in the JavaScript ecosystem. It gives us hooks and components that make working with translations in React straightforward.

To get started, we create a centralized configuration file that defines our supported languages and how the system should behave:

```
// src/config/i18n.ts

export const languages = {
  en: 'English',
  es: 'Español',
} as const;

export type Language = keyof typeof languages;
const supportedLanguages = Object.keys(languages) as Language[];

export const i18nConfig = {
  defaultNS: 'common' as const,
  fallbackLng: 'en' as const,,
  supportedLanguages,
  backend: {
    loadPath: '/api/locales/{{lng}}/{{ns}}',
  },
  detection: {
    order: ['htmlTag'],
    caches: [],
  },
  cookieName: 'lng' as const,
};
```

Let's break down what each setting does:

- **languages** - Maps language codes to display names. The `as const` makes TypeScript treat these as literal types for better type safety.
- **supportedLanguages** - The list of language codes we support, extracted from the languages object.
- **defaultNS** - The namespace to use when we don't specify one. Since `common` contains frequently used translations, it is a sensible default.
- **fallbackLng** - The language to use if the user's preferred language isn't available. Default is English.
- **backend.loadPath** - URL pattern for loading translations that aren't bundled. The `{{ lng }}` and `{{ns}}` placeholders get replaced with

the language and namespace. We will create this API endpoint shortly.

- **detection.order** - How the client detects the user's language. We configure it to read from the HTML `lang` attribute, which the server sets based on the cookie.
- **cookieName** - The name of the cookie that stores the user's language preference

Now let's use this configuration in our application.

Creating the i18next middleware

Since we're using server-side rendering, we need middleware that sets up i18n before rendering pages:

```
// src/app/middleware/i18next.ts

import { createCookie } from 'react-router';
import { createI18nextMiddleware } from 'remix-i18next/middleware';

import { resources } from '@/app/locales';
import { i18nConfig } from '@/config/i18n';

export const localeCookie = createCookie(i18nConfig.cookieName, {
  path: '/',
  sameSite: 'lax',
  secure: process.env.NODE_ENV === 'production',
  httpOnly: true,
});

export const [i18nextMiddleware, getLocale, getInstance] =
  createI18nextMiddleware({
    detection: {
      supportedLanguages: i18nConfig.supportedLanguages,
      fallbackLanguage: i18nConfig.fallbackLng,
      cookie: localeCookie,
    },
    i18next: {
      resources,
      defaultNS: i18nConfig.defaultNS,
    },
  });
```

This middleware does three important things. It creates a secure cookie to remember the user's language preference. It detects the user's language from the cookie. It initializes an i18next instance with all our translations so we can render pages in the correct language on the server.

Initializing on the server

The server has access to all translation files through the middleware we just configured. The middleware reads the language cookie and initializes i18next with the appropriate language before rendering.

In our server entry point, we wrap the application with the i18next provider so all components can access translations:

```
// src/app/entry.server.tsx

import { I18nextProvider } from 'react-i18next';
import { ServerRouter } from 'react-router';
import type { EntryContext } from 'react-router';

import { getInstance } from './middleware/i18next';

export default function handleRequest(
  request: Request,
  responseStatusCode: number,
  responseHeaders: Headers,
  routerContext: EntryContext,
  loadContext: RouterContextProvider,
) {
  // ...

  const { pipe, abort } = renderToPipeableStream(
    <I18nextProvider i18n={getInstance(loadContext)}>
      <ServerRouter context={routerContext} url={request.url} nonce={nonce} />
    </I18nextProvider>,
    {
      // ...
    },
  );

  // ...
}
```

The `I18nextProvider` makes the i18next instance available to all components in the tree, similar to how React Context works.

We also need to set the `lang` and `dir` attributes on the HTML element during server rendering. These attributes are important since browsers, search engines, and screen readers all rely on them. The `lang` attribute tells screen readers how to pronounce text, helps search engines serve content to the right

users, and informs browsers about translation options. The dir attribute controls text direction for right-to-left languages like Arabic and Hebrew.

We set both attributes in the root layout component, which is rendered on the server:

```
// src/app/root.tsx

import { useTranslation } from 'react-i18next';
import { useLoaderData } from 'react-router';

import { i18nextMiddleware } from '@/app/middleware/i18next';

export const middleware = [nonceMiddleware, userMiddleware, i18nextMiddleware];

export async function loader({ context }: Route.LoaderArgs) {
  // ...
}

export function Layout({ children }: { children: React.ReactNode }) {
  const { nonce } = useLoaderData<typeof loader>();
  const { i18n } = useTranslation();

  return (
    <html lang={i18n.language} dir={i18n.dir(i18n.language)}>
      {/* ... */}
    </html>
  );
}
```

The key parts here are:

- `middleware` array - Includes `i18nextMiddleware` so it runs on every request and sets up i18n before rendering.
- `Layout` component - Gets the i18n instance from `useTranslation()` and uses it to set `i18n.language` on the `lang` attribute and `i18n.dir()` for the `dir` attribute. The middleware has already initialized i18next with the correct language from the cookie.

The `i18n.dir()` method returns `'ltr'` for left-to-right languages like English and `'rtl'` for right-to-left languages like Arabic. When we set `dir="rtl"`, the browser handles most layout changes automatically.

With the server configured, the middleware runs on every request, detects the `lang`uage from the cookie, and renders fully translated HTML with the

correct lang and `dir` attributes.

Initializing on the client

Now that the server has rendered the page with the correct `lang` uage and set the `lang` attribute, the client needs to initialize i18next to match. The client reads the language from the lang attribute and uses plugins to load additional translations on demand.

As we discussed in the overview, we use partial bundling: bundle only `common` and `navigation` namespaces (used on every page) and fetch other namespaces on demand. Here's how we configure this:

```
// src/app/entry.client.tsx

import i18next from 'i18next';
import I18nextBrowserLanguageDetector from 'i18next-browser-languagedetector';
import Fetch from 'i18next-fetch-backend';
import { startTransition, StrictMode } from 'react';
import { hydrateRoot } from 'react-dom/client';
import { I18nextProvider, initReactI18next } from 'react-i18next';
import { HydratedRouter } from 'react-router/dom';

import { i18nConfig } from '@/config/i18n';

import { resources } from './locales';

async function main() {
  await i18next
    .use(initReactI18next)
    .use(Fetch)
    .use(I18nextBrowserLanguageDetector)
    .init({
      defaultNS: i18nConfig.defaultNS, // common
      partialBundledLanguages: true,
      resources: {
        en: {
          common: resources.en.common,
          navigation: resources.en.navigation,
        },
        es: {
          common: resources.es.common,
          navigation: resources.es.navigation,
        },
      },
      ns: ['common', 'navigation'],
      fallbackLng: i18nConfig.fallbackLng, // en
      detection: i18nConfig.detection, // order: ['htmlTag'], caches: []
```

```
      backend: i18nConfig.backend, // loadPath: '/api/locales/{{lng}}/{{ns}}'
    });

  startTransition(() => {
    hydrateRoot(
      document,
      <I18nextProvider i18n={i18next}>
        <StrictMode>
          <HydratedRouter />
        </StrictMode>
      </I18nextProvider>,
    );
  });
}

main().catch((error) => console.error(error));
```

Let's break down what's happening here:

- `use(initReactI18next)` - Connects i18next to React so the `useTranslation` hook works.
- `use(Fetch)` - Enables loading translations from our API when needed.
- `use(I18nextBrowserLanguageDetector)` - Provides the detection mechanism. We configure it to read from the HTML *lang* attribute.
- `partialBundledLanguages : true` - Tells i18next that some translations are bundled while others will be loaded later. Without this flag, i18next would think any namespace not in the bundle is missing.
- `resources` - Includes only `common` and `navigation` namespaces. These appear on every page, so we bundle them. Other namespaces load when needed.
- `ns: ['common', 'navigation']` - Preloads these namespaces at startup.
- `fallbackLng` - The language to use when the requested language isn't available. Set to 'en' as our default.
- `detection` - Configures how to detect the user's language. We read from the HTML `lang` attribute (set by the server) and don't cache the detection result since we rely on the server-set cookie.
- `backend` - Configures how to fetch namespaces that aren't bundled. Points to our API endpoint at `/ api /locales/{{ lng }}/{{ns}} .`

We initialize i18next before hydrating React to make sure translations are ready when components render. However, before we do that, we need to create an API endpoint that serves translations on demand.

Serving translations via API

For the dynamic loading to work, we need an API endpoint that serves translations on demand:

```ts
// src/app/routes/api/locales.ts

import { cacheHeader } from 'pretty-cache-header';
import { data } from 'react-router';
import { z } from 'zod';

import { resources } from '@/app/locales';
import type { Language } from '@/config/i18n';

import type { Route } from './+types/locales';

export async function loader({ params }: Route.LoaderArgs) {
  const lng = z
    .enum(Object.keys(resources) as Array<Language>)
    .safeParse(params.lng);

  if (lng.error) return data({ error: lng.error }, { status: 400 });

  const namespaces = resources[lng.data];

  const ns = z
    .enum(Object.keys(namespaces) as Array<keyof typeof namespaces>)
    .safeParse(params.ns);

  if (ns.error) return data({ error: ns.error }, { status: 400 });

  const headers = new Headers();

  if (process.env.NODE_ENV === 'production') {
    headers.set(
      'Cache-Control',
      cacheHeader({
        maxAge: '5m',
        sMaxage: '1d',
        staleWhileRevalidate: '7d',
        staleIfError: '7d',
      }),
    );
  }
```

```
  return data(namespaces[ns.data], { headers });
}
```

This endpoint matches the URL pattern `/api/locales/{{lng}}/{{ns}}` that we configured earlier. It validates both parameters using Zod and returns the requested translations as JSON.

Since translations don't change often, we cache them aggressively. This means users rarely wait for translation requests after the first load.

Adding type safety to translation keys

Translation keys are strings, which means TypeScript can't catch typos by default. If we write `t('welcom')` instead of `t('welcome')`, we won't know until we see the broken translation in the browser.

We can fix this by telling TypeScript about our translation structure:

```
// src/app/types/i18next.d.ts

import 'i18next';

import type { resources } from '@/app/locales';
import { i18nConfig } from '@/config/i18n';

declare module 'i18next' {
  interface CustomTypeOptions {
    defaultNS: typeof i18nConfig.defaultNS;
    resources: typeof resources.en;
  }
}
```

This tells TypeScript that our translations follow the structure of `resources.en`. Now, when we type `t('`, our editor shows all available keys. Typos get caught at compile time with a red underline, not at runtime when a user reports the bug.

We use the English translations as the source of truth. If a key exists in English, TypeScript expects it. If we reference a key that doesn't exist, TypeScript warns us. This also makes sure we don't have any missing translations in other languages; otherwise, TypeScript will warn us.

With our i18n system fully configured, we're ready to use translations in our application components.

Using translations in the application

React-i18next provides the `useTranslation` hook to access translations. This is how we replace hardcoded strings with translated text.

Let's look at a simple example from our home page:

```
// src/app/routes/home.tsx

import { useTranslation } from 'react-i18next';
import { Link } from 'react-router';

import { Button } from '@/components/ui/button';

export default function HomePage() {
  const { t } = useTranslation(['home']);

  return (
    <div className="container mx-auto px-4 py-16">
      <Seo
        title={t('meta.title')}
        description={t('meta.description')}
      />
      <div className="text-center mb-16">
        <h1 className="text-4xl md:text-6xl font-bold mb-6">
          {t('title')}
        </h1>
        <p className="text-xl text-muted-foreground mb-8 max-w-2xl mx-auto">
          {t('subtitle')}
        </p>
        {/* ... */}
      </div>
      {/* ... */}
    </div>
  );
}
```

We pass `['home']` to `useTranslation` to tell it which namespace to load. The hook returns a t function we use to get translations.

When we call `t('home:title')`, we're asking for the `title` key from the `home` namespace. The colon separates the namespace from the key. For nested translations like `t('home:meta.title')`, we use dot notation to navigate into the `meta` object and get the `title` key.

Using the default namespace

Since we configured `common` as our default namespace, we can access its translations without the namespace prefix:

```
import { useTranslation } from 'react-i18next';

function DeleteButton() {
  const { t } = useTranslation();

  return <button>{t('delete')}</button>;
}
```

Here, `t('delete')` looks up the `delete` key in the `common` namespace automatically. This saves typing for translations we use frequently.

Interpolation

Static translations only get us so far. Real applications need to insert dynamic values like usernames and counts.

Interpolation lets us insert variables into translation strings. We use double curly braces as placeholders:

```
// src/app/locales/en/common.ts

export default {
  welcome: 'Welcome, {{username}}!',
  // ...
};
```

To use this, we pass an object with the variable values:

```
function WelcomeMessage({ username }: { username: string }) {
  const { t } = useTranslation();

  return <h1>{t('welcome', { username })}</h1>;
}
```

When rendered with `username="John"` , this displays `"Welcome, John!"` . The `{{username}}` placeholder gets replaced with whatever value we pass.

This is powerful because translators can position the variable anywhere in the sentence. In some languages, the greeting might be structured differently, like `"John, welcome!"` The translator controls the word order, and the developer just provides the values.

Pluralization

Different quantities often require different wording. "1 idea" vs "2 ideas" in English. React-i18next handles this with the `_one` and `_other` suffixes:

```
// src/features/ideas/locales/en.ts

export default {
  ideasBy_one: '{{count}} Idea by {{username}}',
  ideasBy_other: '{{count}} Ideas by {{username}}',
  // ...
};
```

When we call the translation with a `count` variable, i18next automatically picks the right form:

```
function UserIdeas({ count, username }: { count: number; username: string }) {
  const { t } = useTranslation(['ideas']);

  return <p>{t('ideas:ideasBy', { count, username })}</p>;
}
```

If `count` is 1, i18next uses `ideasBy_one` and shows "`1 Idea by John`". For any other number, it uses `ideasBy_other` and shows "`5 Ideas by John`".

For Spanish:

```
// src/features/ideas/locales/es.ts

export default {
  ideasBy_one: '{{count}} Idea de {{username}}',
  ideasBy_other: '{{count}} Ideas de {{username}}',
  // ...
};
```

The code stays the same; only the translation files differ.

Formatting dates

Dates are tricky because different regions format them differently. January 15, 2024, in the US becomes 15/01/2024 in many European countries. Even month names change; "January" in English is "enero" in Spanish.

JavaScript has built-in support for this through the `Intl.DateTimeFormat` API:

```
// src/lib/date.ts
```

```
export function formatDate(date: string | Date, locale: string = 'en'): string {
  const d = date instanceof Date ? date : new Date(date);

  if (isNaN(d.getTime())) {
    console.error(`Invalid date: "${date}"`);
    return '';
  }

  return new Intl.DateTimeFormat(locale, {
    year: 'numeric',
    month: 'short',
    day: 'numeric',
    timeZone: 'UTC',
  ).format(d);
}
```

This function formats dates according to the locale. For English, we might get "Jan 15, 2024". For Spanish, we might get "15 ene 2024".

To use this in components, we get the current language from the i18n instance:

```
// src/features/ideas/components/idea-card.tsx

import { useTranslation } from 'react-i18next';

import { formatDate } from '@/lib/date';

export function IdeaCard({ idea }: IdeaCardProps) {
  const { i18n } = useTranslation();

  return (
    <div>
      <h2>{idea.title}</h2>
      <p>Created: {formatDate(idea.createdAt, i18n.language)}</p>
    </div>
  );
}
```

The `useTranslation` hook gives us both the t function and the `i18n` instance. We pass `i18n.language` to our `formatDate` function, and dates display in the correct format for the user's language.

Switching between languages in the application

Users need a way to change their language preference. This requires both a UI component for the user to select their language and an API endpoint to persist that choice. Let's build both pieces.

The language switcher component

The switcher is a dropdown that shows available languages and lets users pick one:

```
// src/components/language-switcher.tsx

import { Check, Globe } from 'lucide-react';
import { useTranslation } from 'react-i18next';

import { Button } from '@/components/ui/button';
import {
  DropdownMenu,
  DropdownMenuContent,
  DropdownMenuItem,
  DropdownMenuTrigger,
} from '@/components/ui/dropdown-menu';
import { i18nConfig, languages, type Language } from '@/config/i18n';
import { useNotificationActions } from '@/stores/notifications';

export function LanguageSwitcher() {
  const { t, i18n } = useTranslation(['components', 'common']);
  const { showNotification } = useNotificationActions();

  const handleLanguageChange = async (language: Language) => {
    const formData = new FormData();
    formData.append(i18nConfig.cookieName, language);

    try {
      const response = await fetch('/api/set-language', {
        method: 'POST',
        body: formData,
      });

      if (response.ok) {
        await i18n.changeLanguage(language);
        showNotification({
          type: 'success',
          title: t('common:languageChanged', {
            lng: language,
          }),
        });
      }
    } catch (error) {
      console.error('Failed to change language:', error);
```

```
    }
  };

  return (
    <DropdownMenu>
      <DropdownMenuTrigger
        render={
          <Button variant="ghost" size="sm" className="h-8 w-8 p-0">
            <Globe className="h-4 w-4" />
            <span className="sr-only">
              {t('components:languageSwitcher.switchLanguage')}
            </span>
          </Button>
        }
      />
      <DropdownMenuContent align="end">
        {Object.entries(languages).map(([key, value]) => (
          <DropdownMenuItem
            key={key}
            onClick={() => handleLanguageChange(key as Language)}
            className="cursor-pointer"
          >
            <span className="flex items-center gap-2">
              {i18n.language === key && <Check className="h-4 w-4" />}
              <span className={i18n.language === key ? '' : 'ml-6'}>
                {value}
              </span>
            </span>
          </DropdownMenuItem>
        ))}
      </DropdownMenuContent>
    </DropdownMenu>
  );
}
```

The language change happens in two steps. First, we tell the server to update the cookie so the preference persists. Then we update the client-side i18n instance so the UI updates immediately. Without both steps, the change either wouldn't persist across page reloads or wouldn't be visible until refresh.

The API endpoint

For the component to work, we need an API endpoint that sets the language cookie:

```
// src/app/routes/api/set-language.ts

import { data } from 'react-router';
```

```
import z from 'zod';

import { localeCookie } from '@/app/middleware/i18next';
import { i18nConfig, languages, type Language } from '@/config/i18n';

import type { Route } from './+types/set-language';

const languageSchema = z.enum(
  Object.keys(languages) as [Language, ...Language[]],
);

export async function action({ request }: Route.ActionArgs) {
  const formData = await request.formData();

  const language = languageSchema.safeParse(
    formData.get(i18nConfig.cookieName),
  );

  if (!language.success) {
    return data({ success: false }, { status: 400 });
  }

  return data(
    { success: true },
    {
      headers: {
        'Set-Cookie': await localeCookie.serialize(language.data),
      },
    },
  );
```

This endpoint validates that the requested language is one we support, then sets the cookie. The next time the user loads a page, the server will detect their preference from this cookie and render the page in the correct language.

Summary

In this chapter, we built a complete internationalization system for our application. We started by understanding why separating text from code makes translations manageable.

We organized translations using namespaces to keep files small and focused. App-level namespaces like `common` contain shared translations, while feature-scoped translations live alongside their feature code. This organization makes translations easy to find and maintain.

We set up react-i18next with both server and client initialization. The server renders pages with correct translations, and the client hydrates smoothly. We created an API endpoint for serving translations on demand, enabling partial bundling, which keeps the initial JavaScript bundle small.

Type safety catches translation errors early. By telling TypeScript about our translation structure, we get autocomplete and compile-time checking for translation keys. We also configure language and direction attributes so browsers, screen readers, and search engines understand our content.

We learned how to use the `useTranslation` hook to access translations in components, and how interpolation and pluralization handle dynamic content. We implemented localized date formatting using the browser's built-in Intl API.

Finally, we built a language switcher that persists the user's choice in a cookie. The preference applies across sessions, giving users a consistent experience.

With internationalization in place, our application can serve users around the world in their preferred languages.

10

Making the Application Accessible

Imagine you're a user who relies on the keyboard to navigate through the application. Perhaps a motor impairment makes using a mouse difficult, or perhaps it's simply your preference. You open the application, press Tab to move through the interface, and nothing happens. No focus indicators, no logical tab order, no way forward. You close the tab and move on. For you, the application is unusable. **Accessibility** , often abbreviated as **a11y** , ensures that our applications can be used by everyone, including people with disabilities. This includes users who rely on screen readers, navigate with keyboards, have low vision, or experience motor impairments. Building accessible applications isn't just the right thing to do. It's often a legal requirement and expands our potential user base significantly.

We'll cover the following topics:

- Understanding accessibility fundamentals
- Laying the architectural foundation with semantic HTML and accessible component libraries
- Practical accessibility optimizations

By the end of this chapter, we'll learn how to think about accessibility from the start. We'll see how making the right choices early on makes our applications more maintainable and helps us reach more users.

Technical requirements

Before we get started, we need to set up our project. To be able to develop our project, we'll need the following things installed on our computer:

- Node.js version 24 or above, npm version 11 or above ships with Node. We can confirm that by executing `node -v` and `npm -v` in the terminal. There are multiple ways to install Node.js and npm. Here is a great article that goes into more detail: https://www.nodejsdesignpatterns.com/blog/5-ways-to-install-node-js .
- VS Code (optional) is a popular editor for JavaScript and TypeScript: open source, solid TypeScript support, and extensions. It can be downloaded from https://code.visualstudio.com .

The code for this book is available at the book's repo. To access the repository link, follow the steps in the " *Download the example code files* " section in the *Preface* . Clone it and enter the repository root:

```
git clone https://github.com/PacktPublishing/React-Application-Architecture-for-Production-Second-Edition.git
```

The repository contains chapter folders with the code for each chapter, plus a shared `api` folder with the API server used across all chapters.

We are on *Chapter 10* , so we need to navigate to the `chapter-10` directory:

```
cd React-Application-Architecture-for-Production-Second-Edition/chapter-10
```

Then we need to install dependencies:

```
npm install
```

We also need to provide the environment variables:

```
cp .env.example .env
```

Now we should have the frontend running at http://localhost:5173 .

We also need to have our API server running.

Let's open a new terminal window and navigate to the `api` directory:

```
cd React-Application-Architecture-for-Production-Second-Edition/api
```

Now we need to run the setup script for *Chapter 10* to configure everything for us:

```
npm run setup 10
```

Then we need to run the API server:

```
npm run dev
```

We should see the API server running on `http://localhost:9999` .

For more information about the setup details, check out the `README.md` file.

Understanding accessibility fundamentals

Before diving into technical details, let's see why accessibility matters and which principles guide our work.

- **It's about people** : More than one billion people experience some form of disability. These disabilities range from permanent conditions like blindness to temporary situations like a broken arm to situational limitations like trying to use a phone in bright sunlight. When we build inaccessible applications, we're excluding a significant portion of potential users.
- **It's good for business** : Accessible applications reach more users. Every barrier we remove opens our application to more people. Major tech companies have found that prioritizing accessibility doesn't just help users with disabilities; it often improves the experience for everyone.
- **It's often required by law** : Many countries have laws requiring digital accessibility. In the United States, the **Americans with Disabilities Act** (**ADA**) applies to many websites. The European Union has the European Accessibility Act. Similar laws exist worldwide. Organizations face lawsuits when their digital properties aren't accessible.
- **It makes better code** : Accessible code tends to be better code. It requires us to use good practices. When we use semantic HTML, our markup becomes more meaningful. When we structure content logically, it becomes easier to maintain. When we test with keyboards and screen readers, we find bugs that affect all users.

Accessibility principles

The **Web Content Accessibility Guidelines (WCAG)** provide a framework for making web content accessible. WCAG is organized around four principles, often remembered by the acronym *POUR* :

- Perceivable - Information must be presentable in ways users can perceive. This means providing text alternatives for images, captions for videos, and sufficient color contrast.
- Operable - Users must be able to operate the interface. This means supporting keyboard navigation, providing users enough time to read content, and avoiding content that causes seizures.
- Understandable - Users must be able to understand the information and operation of the interface. This means using clear language, consistent navigation, and helpful error messages.
- Robust - Users must be able to rely on content being interpreted reliably by a wide variety of technologies. This means using valid HTML and following standards.

We will use these principles throughout this chapter to focus on architectural decisions that have the greatest impact on real users.

Laying the architectural foundation

The foundation has two important parts: using semantic HTML and using accessible component libraries. It's important to get these correct from the start, as they will affect the entire application.

Semantic HTML

HTML isn't just markup. It's an accessibility API that browsers and assistive technologies understand. When we use the right HTML element, we get behavior for free. When we use the wrong one, we have to rebuild that behavior ourselves.

Think about a `< button >` versus a `< div >` with an `onClick` handler. They can look identical and both respond to clicks. But under the hood, they're different.

A button element comes with built-in behavior. It responds to keyboard events. Screen readers announce it as a button. Form submissions recognize it. Focus management systems know how to handle it.

When we use a `div` instead, we lose all of that. We need to add `tabIndex` to make it focusable, `onKeyDown` handlers to respond to Enter and Space, and `role="button"` to tell screen readers what it is. And even then, we still fall

short. A `div` can never participate in form submission or have the full semantics of `disabled`. We are not fixing the problem; we are patching around it.

In the context of our application, when users filter ideas by tag we use the `Badge` component. When we click on a badge, we toggle the tag filter. That sounds like a button's behavior. But our `Badge` component renders a `<span>` by default:

```
// src/components/ui/badge.tsx

function Badge({
  className,
  variant = 'default',
  render,
  ...props
}: useRender.ComponentProps<'span'>&VariantProps<typeof badgeVariants>) {
  return useRender({
    defaultTagName: 'span',
    props: mergeProps<'span'>(
      {
        className: cn(badgeVariants({ className, variant })),
      },
      props,
    ),
    render,
    state: {
      slot: 'badge',
      variant,
    },
  });
}
```

How do we keep the `Badge` styling while using a button element? We need to make sure our design system is flexible enough to support overrides when needed. Fortunately, we can achieve that easily by using the `render` prop, which allows us to provide a custom element to render. When we provide it, the `Badge` component passes its styling to the provided element instead of creating its own span element. This lets us provide a button and get Badge's appearance:

```
// src/features/ideas/components/idea-search-and-filters.tsx

<Badge
  key={tag}
```

```
  variant={selectedTags.includes(tag) ? 'default' : 'outline'}
  className="cursor-pointer"
  render={
    <button
      type="button"
      aria-pressed={selectedTags.includes(tag)}
      aria-label={t('ideas:tagStatus', {
        tag,
        status: selectedTags.includes(tag)
          ? t('common:selected')
          : t('common:notSelected'),
      })}
      onClick={() => toggleTag(tag)}
    >
      {tag}
    </button>
  }
/>
```

The button element brings everything we need. It's keyboard focusable. It responds to keyboard events. Screen readers announce it as a button.

If we used divs or spans instead, we'd need to manually add all of this behavior to every single tag filter. By using a button, we get all of that behavior from the platform: keyboard handling, ARIA role, and event semantics without writing a single line of JavaScript.

If we wanted to go further, we could make an abstraction with something like `SelectableBadge` on top of the `Badge` component so we don't have to repeat this pattern everywhere.

Defining page landmarks

Buttons and links handle interactive elements, but semantic HTML also provides structural elements that help users navigate the page itself. Elements like `< header >`, `< main >`, `< nav >`, and `< footer >` create landmarks that screen readers can jump between.

Let's look at our application's layout structure:

```
// src/app/routes/layout.tsx

export default function Layout() {
  // ...
  return (
    <div className="min-h-screen flex flex-col">
      {/* ... */}
```

```
      <header>
        <Navigation user={user} onLogout={handleLogout} />
      </header>
      <main id="main-content" className="flex-1 bg-background">
        <Outlet />
      </main>
      <footer className="border-t py-6 px-4 text-center text-sm text-muted-foreground">
        © {new Date().getFullYear()} AIdeas. All rights reserved.
      </footer>
    </div>
  );
}
```

The `<header>`, `<main>`, and `<footer>` elements create landmarks that screen readers can navigate. Users can jump directly to the main content or footer without tabbing through everything. This works without JavaScript, survives CSS failures, and provides meaning to assistive technologies.

Using accessible component libraries

Semantic HTML handles simple elements, but complex UIs need more. Building accessible interactive components from scratch is hard, and the difficulty doesn't end at initial implementation. Browsers and assistive technologies behave inconsistently and evolve independently, so keeping components correct over time is where teams most often fall short. A dropdown menu needs keyboard navigation with arrow keys, focus management, Escape to close, disabled state handling, screen reader announcements, and correct ARIA attributes. Get any of it wrong and some users can't use the menu.

This is where component libraries matter. We already covered how to choose a component library in *Chapter 3*. Besides their baked-in functionality, they should also provide accessibility support and allow us to focus on application logic and styling.

Take dialogs as an example. A properly accessible dialog needs:

- Focus trapping - Focus stays inside the dialog until it closes
- Return focus - Focus returns to the trigger when dismissed
- Keyboard handling - Escape closes the dialog, Tab cycles through content

- ARIA attributes - Proper `role` , `aria-modal` , `aria-` , `labelledby` set automatically
- Body scroll locking - Prevents page content from scrolling behind the modal
- Portal rendering - Renders dialog at document root for proper stacking

When we use the right component library, we get all of this for free. That's why making the right decision here is very important, as it will affect the entire application and may be difficult to change later.

Practical accessibility optimizations

Semantic HTML and accessible component libraries handle most accessibility concerns, but some situations need extra work. Let's look at the most common optimizations and where they fit in the application.

Skip links for keyboard navigation

When users navigate with a keyboard, they tab through every interactive element. If our navigation has five links, keyboard users must press Tab five times on every page just to reach the main content. For users who use only keyboards, this gets exhausting.

The solution is a skip link: a hidden anchor that navigates directly to the main content. It's the first focusable element on the page.

```
// src/app/routes/layout.tsx

export default function Layout() {
  const { t } = useTranslation(['navigation']);
  // ...

  return (
    <div className="min-h-screen flex flex-col">
      <a
        href="#main-content"
        className="sr-only focus:not-sr-only focus:absolute focus:top-4 focus:left-4 focus:z-50
focus:bg-background focus:text-foreground focus:px-4 focus:py-2 focus:border focus:rounded
focus:ring-2 focus:ring-ring"
      >
        {t('navigation:skipToContent')}
      </a>
      <header>
        <Navigation user={user} onLogout={handleLogout} />
      </header>
```

```
        <main id="main-content" className="flex-1 bg-background">
          <Outlet />
        </main>
        <footer className="border-t py-6 px-4 text-center text-sm text-muted-foreground">
          © {new Date().getFullYear()} AIdeas. All rights reserved.
        </footer>
      </div>
    );
}
```

The skip link is visually hidden by default (`sr-only`) but becomes visible when focused. It targets the `main` element via `id="main-content"` , so users can bypass navigation with a single Tab and Enter.

If we press the Tab button, the skip link becomes visible:

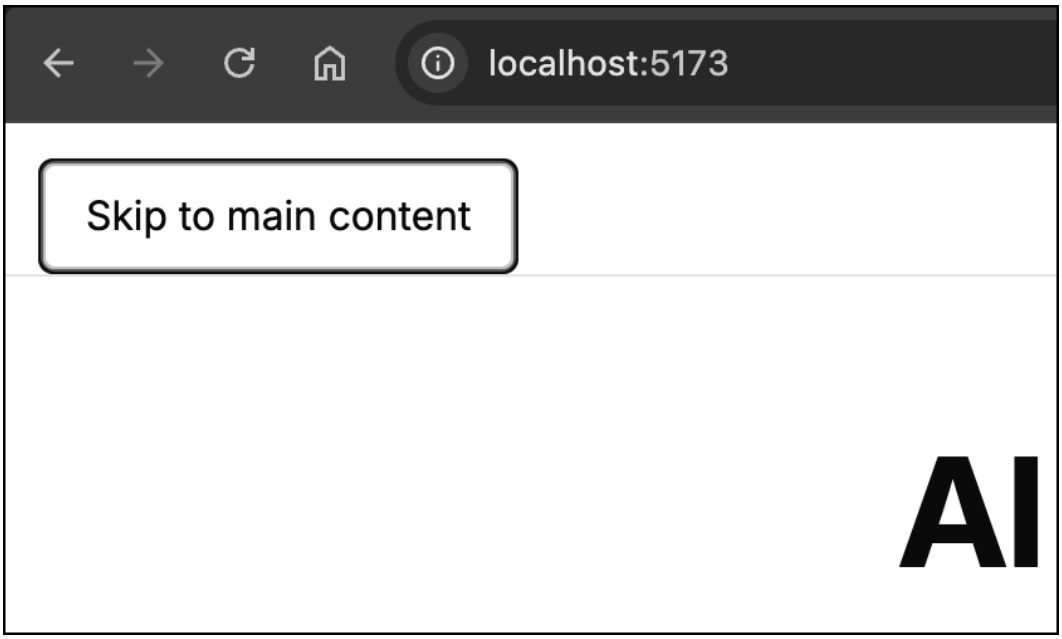


Figure 10.1: Skip link becomes visible when focused

This takes us directly to the main content, skipping the navigation and other less important elements.

Providing accessible labels

When we show a button with only an icon (like a three-dot menu button), sighted users know that it opens a menu. However, screen reader users hear only "button" with no information about what it does since there is no visible text.

Every interactive element needs an accessible name. For any element that doesn't have visible text, we use `aria-label` :

```
// src/features/ideas/components/idea-actions.tsx

<DropdownMenuTrigger
  render={
    <Button
      variant="ghost"
      size="sm"
      className="h-8 w-8 p-0"
      aria-label={t('ideas:ideaActionsMenu')}
    >
      <MoreHorizontal className="h-4 w-4" aria-hidden="true" />
    </Button>
  }
/>
```

The `aria-label` provides the accessible name that screen readers announce: "Idea actions menu, button." The icon itself gets `aria-hidden="true"` because it's purely visual, so it's not announced by screen readers. This same pattern applies anywhere we need to provide text that screen readers can announce but sighted users don't need to see: close buttons, menu buttons, icon toggles, or any visual representation that doesn't translate well to text.

Announcing dynamic changes

When users select a tag filter, React updates the list without a page reload. Sighted users see the list change, but screen reader users don't get any notification. They don't know if the filter worked or how many results are showing. For dynamic applications, this creates a gap between action and feedback.

React applications update without page reloads, and screen readers don't announce DOM changes unless they affect focus or a live region. To solve this, we can use ARIA live regions.

Let's look at how we announce search results to screen readers:

```
// src/app/routes/ideas/ideas.tsx

function Ideas() {
  const { t } = useTranslation(['ideas', 'common']);
  const { params } = useSearchAndFilters();
```

```
  const ideasInfiniteQuery = useIdeasInfiniteQuery({
    params: params as GetAllIdeasData['query'],
  });

  const allIdeas =
    ideasInfiniteQuery.data?.pages.flatMap((page) => page.data) || [];

  // ...

  return (
    <div className="container mx-auto px-4 py-8">
      <div className="mb-8">
        {/* ... */}
        <IdeaSearchAndFilters />
      </div>

      <div
        aria-live="polite"
        aria-atomic="true"
        className="sr-only"
        role="status"
      >
        {t('ideas:resultsFound', { count: allIdeas.length })}
      </div>

      <ListComponent
        ideas={allIdeas}
        isLoading={ideasInfiniteQuery.isLoading && allIdeas.length === 0}
        emptyMessage={t('ideas:noIdeasAvailable')}
        error={ideasInfiniteQuery.error}
      />
    </div>
  );
}
```

The live region is visually hidden (`sr-only`), but screen readers monitor it. When `allIdeas.length` changes, screen readers announce "5 results found." The `aria-live="polite"` means the announcement waits for a pause in speech. Use `aria-live="assertive"` for critical updates like errors. The `role="status"` indicates this is status information. To scale it even further, we can consider centralizing announcements in a shared utility or hook to keep live region logic consistent.

Summary

Accessibility is important for multiple reasons. It's about people, it's good for business, it's often required by law, and it leads to better code. We can use the WCAG POUR principles to guide our decisions and build applications that are accessible to everyone.

Semantic HTML and accessible component libraries form the foundation. Semantic HTML gives us keyboard support and screen reader compatibility for simple interactions. Accessible component libraries handle complex patterns, managing focus and ARIA attributes so we don't have to.

Beyond the foundation, we covered three key optimizations. Skip links let keyboard users jump directly to main content. Accessible labels provide context for icon-only buttons and visual elements. Live regions announce dynamic changes so screen reader users know when content updates.

Get this book's PDF copy, code bundle, and more

Scan the QR code (or go to `packtpub.com/unlock`). Search for this book by name, confirm the edition, and then follow the steps on the page.

UNLOCK NOW

Note: Have your invoice handy. Purchases made directly from the Packt website don't require an invoice.

11

Testing the Application

Testing is what gives us confidence that our application works correctly. When we build features, fix bugs, or refactor code, we need to know that our changes don't break existing functionality. Without tests, every change becomes risky, and we end up spending more time manually checking that everything still works.

In this chapter, we'll learn how to test our application using different testing approaches. Each approach serves a different purpose and together they create a comprehensive testing strategy that catches bugs early and keeps our application reliable.

We'll cover the following topics:

- Understanding why testing is important
- Unit testing
- Integration testing
- End-to-end testing

By the end of this chapter, we'll have a solid testing strategy that gives us confidence to ship changes quickly and safely.

Technical requirements

Before we get started, we need to set up our project. To be able to develop our project, we'll need the following things installed on our computer:

- Node.js version 24 or above, npm version 11 or above ships with Node. We can confirm that by executing `node -v` and `npm -v` in the terminal. There are multiple ways to install Node.js and npm. Here is a

great article that goes into more detail: https://www.nodejsdesignpatterns.com/blog/5-ways-to-install-node-js

- VS Code (optional) is a popular editor for JavaScript and TypeScript: open source, solid TypeScript support, and extensions. It can be downloaded from `https://code.visualstudio.com` .

The code for this book is available at the book's repo. To access The repository link, follow the steps in the *"Download the example code files"* section in the *Preface* . Clone it and enter the repository root:

```
git clone https://github.com/PacktPublishing/React-Application-Architecture-for-Production-Second-Edition.git
```

The repository contains chapter folders with the code for each chapter, plus a shared `api` folder with the API server used across all chapters.

We are on *chapter 11* , so we need to navigate to the `chapter -11` directory:

```
cd React-Application-Architecture-for-Production-Second-Edition/chapter-11
```

Then we need to install dependencies:

```
npm install
```

We also need to provide the environment variables:

```
cp .env.example .env
```

Now we should have the frontend running at `http://localhost:5173` .

We also need to have our API server running.

Let's open a new terminal window and navigate to the api directory:

```
cd React-Application-Architecture-for-Production-Second-Edition/api
```

Now we need to run the setup script for *Chapter 11* to configure everything for us:

```
npm run setup 11
```

Then we need to run the API server:

```
npm run dev
```

We should see the API server running on `http://localhost:9999` .

For more information about the setup details, check out the `README.md` file.

Why testing is important

Imagine making a small change to your authentication logic and accidentally breaking the login flow for all users. Or refactoring a component and unknowingly removing a feature that users depend on. Without tests, these bugs are only discovered in production when users report them.

Tests act as a safety net. They verify that our code works as expected and alert us immediately when something breaks. This means we can refactor with confidence, add new features without fear, and catch bugs before they reach our users.

But not all tests are equally valuable. We need to think strategically about what to test and how to test it. Here are the three main types of tests we'll use:

- **Unit tests** verify individual pieces of code in isolation. They're fast to run and easy to write, but they can't tell us if different parts of our application work together correctly. We use unit tests for utility functions, custom hooks, and reusable components that have clear inputs and outputs.
- **Integration tests** verify how multiple parts work together. They're more valuable than unit tests because they verify that our components, hooks, and API calls integrate correctly. Most of our tests should be integration tests because they give us more confidence while still running relatively fast.
- **End-to-end tests** verify the entire application from the user's perspective. They run the complete system with both frontend and backend, simulating real user interactions. These tests are slower and more expensive to maintain, so we use them selectively to verify critical user journeys work from start to finish.

We can use the testing trophy approach to decide what tests to write. We'll write mostly integration tests, some end-to-end tests for critical paths, and unit tests for complex business logic.

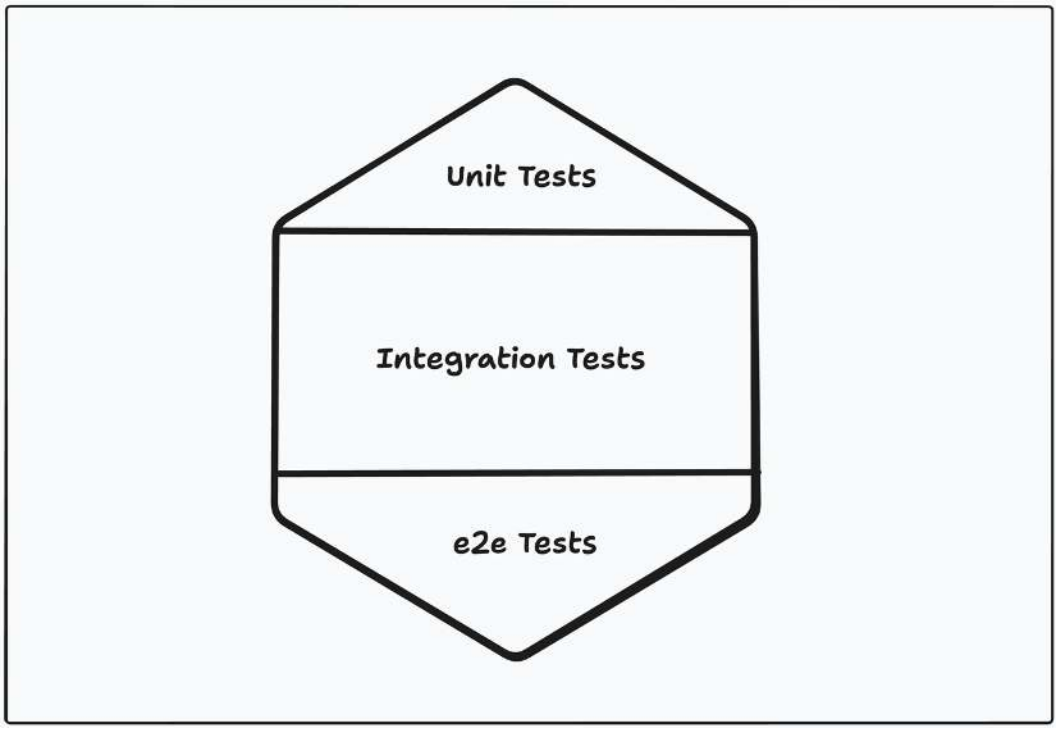


Figure 11.1 – Testing trophy

The idea behind the testing trophy is to focus on integration tests for most functionality, use unit tests for complex logic that's hard to test otherwise, and reserve end-to-end tests for the most important user flows. This gives us good coverage without slowing down our development workflow.

Now let's see how to implement each type of test in our application.

Unit testing

Unit testing is a testing method where individual pieces of code (functions, hooks, components) are tested in isolation. External dependencies are replaced with mocks or stubs, ensuring tests are fast, deterministic, and focused solely on the unit's own logic.

The following diagram shows how unit tests work. We test a single piece of code in isolation, providing inputs and verifying the outputs without involving any external systems:

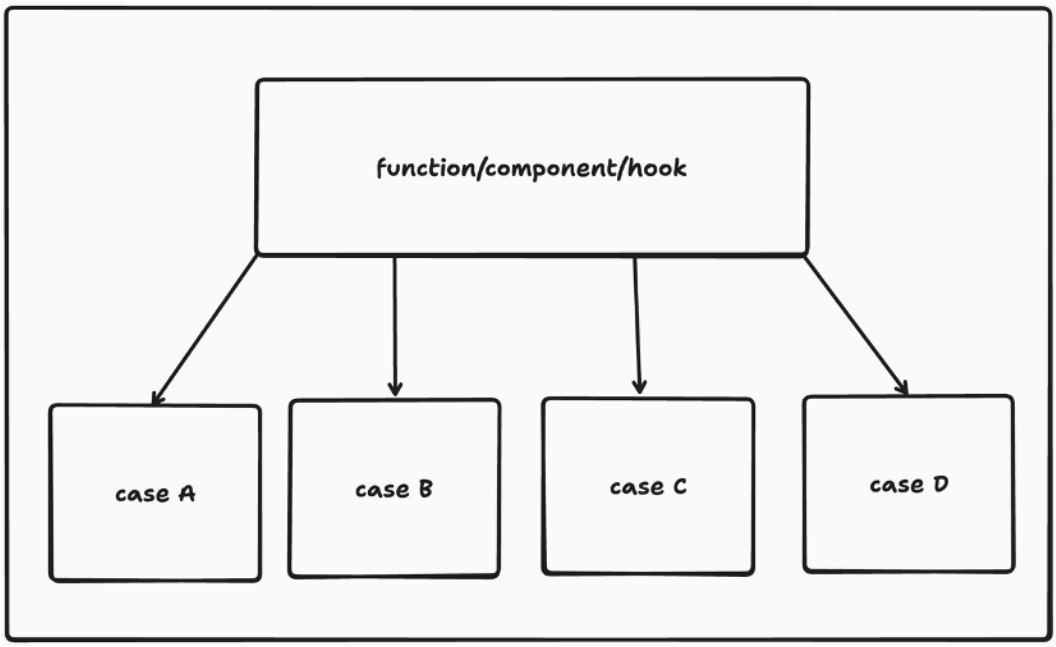


Figure 11.2 – Unit testing

For writing unit tests, we can use **Vitest** as our test runner and **React Testing Library** for testing React components and hooks. Vitest is fast, has a great developer experience, and integrates seamlessly with our Vite-based application.

Let's look at how to unit test different parts of our application. Utility functions are perfect candidates for unit testing. They have clear inputs and outputs, no side effects, and don't depend on external state. For example, our authorization policies have more logic and edge cases to cover, so they benefit from thorough unit testing:

```
// src/features/auth/lib/__tests__/authorization-policies.test.ts

const createUser = (id: string): CurrentUser => ({
  // ...
});

const createIdea = (authorId: string): Idea => ({
  // ...
});

const createReview = (authorId: string): Review => ({
  // ...
});

describe('IdeaPolicies', () => {
```

```
  describe('canCreate', () => {
    it('should return true for authenticated users', () => {
      expect(IdeaPolicies.canCreate(createUser('1'))).toBe(true);
    });

    it('should return false for unauthenticated users', () => {
      expect(IdeaPolicies.canCreate(null)).toBe(false);
    });
  });

  describe('canEdit', () => {
    it('should return true when user is the author', () => {
      const user = createUser('1');
      const idea = createIdea('1');
      expect(IdeaPolicies.canEdit(user, idea)).toBe(true);
    });

    it('should return false when user is not the author', () => {
      const user = createUser('2');
      const idea = createIdea('1');
      expect(IdeaPolicies.canEdit(user, idea)).toBe(false);
    });

    it('should return false when user is null', () => {
      const idea = createIdea('1');
      expect(IdeaPolicies.canEdit(null, idea)).toBe(false);
    });
  });

  describe('canDelete', () => {
    it('should return true when user is the author', () => {
      const user = createUser('1');
      const idea = createIdea('1');
      expect(IdeaPolicies.canDelete(user, idea)).toBe(true);
    });

    it('should return false when user is not the author', () => {
      const user = createUser('2');
      const idea = createIdea('1');
      expect(IdeaPolicies.canDelete(user, idea)).toBe(false);
    });
  });
});

describe('ReviewPolicies', () => {
  // ...
});
```

This test suite verifies that our `IdeaPolicies` and `ReviewPolicies` return the correct values across different scenarios: authenticated vs. unauthenticated users, authors vs. non-authors, and different resource types. Each test is focused on a single, specific behavior.

Utility functions like these are the easiest things to unit test because they're pure logic with no dependencies. But what about testing React components? Let's look at that next.

Integration testing

Unit tests should be fast and focused. They should verify that individual pieces work correctly in isolation. But they can't tell us if we wired everything correctly and our application works when all these pieces come together. That's where integration tests come in.

Our unit tests might pass, but our application can still break. Integration testing catches these gaps by testing multiple parts of the application working together. Instead of testing a part of the application in isolation, we test how it works in the context of the application and all its parts. The only thing we mock is the API layer.

This is the most valuable type of testing for most applications. A good approach for integration testing is to test individual pages. This ensures we are testing the complete flow of a route, including loaders, error boundaries, and user interactions within that page.

The following diagram illustrates how integration tests cover more of the application stack compared to unit tests:

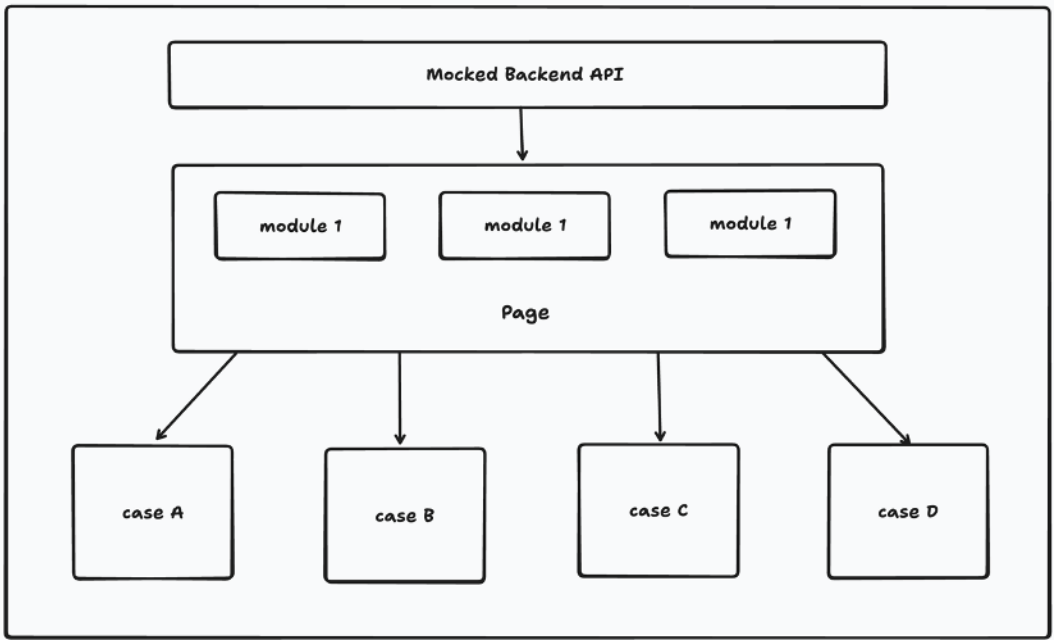


Figure 11.3 – Integration testing

As the diagram shows, integration tests cover the full component tree of a page, including routing and data loading. The only boundary we mock is the API layer, which gives us control over the data without needing a running backend.

We have two options for what tools to use for integration testing:

- **Vitest and React Testing Library with mocked API requests** but it becomes tricky when testing server-side rendered pages and data passing from server to client. React Testing Library can render components, but it can't test how loaders fetch and pass data to the route components.
- **Playwright with mocked API requests** . This approach is powerful because we're testing our application exactly as users experience it, in a real browser with real interactions but without depending on a backend server. By mocking API requests, we can control exactly what data our application receives and test different scenarios like success, errors, and edge cases. This gives us the sweet spot between speed and confidence.

Since our application uses React Router with server-side rendering, the second option is a better fit. While Playwright is primarily an end-to-end testing tool, it works well for integration testing too. Playwright has a built-in

`page.route()` mocking mechanism but it can only intercept requests made by the browser. In an SSR application, the server also makes API requests from the loaders, and those requests bypass Playwright's client-side interception. Until that's supported by Playwright, we can use the **Mocky Balboa** library to mock API requests via `mocky.route()` during both server-side rendering and client-side navigation. It intercepts requests in both environments, so the same mock data is used whether the page is server-rendered on first load or client-rendered during navigation. This gives us consistent, reliable test behavior regardless of how the page is loaded.

Let's test the ideas page to see how this works in practice:

```
// src/app/routes/ideas/__tests__/ideas.integration.test.ts

import { test, expect } from 'testing/fixtures/integration.fixture';
import { generateIdea } from 'testing/test-data';

test.describe('Ideas page', () => {
  test.beforeEach(async ({ mocky, API_URL }) => {
    const tags = ['technology', 'design', 'business', 'health'];

    mocky.route(`${API_URL}/ideas/tags`, (route) => {
      return route.fulfill({
        body: JSON.stringify({ data: tags }),
        status: 200,
      });
    });
  });

  test.describe('Display', () => {
    test('displays ideas list', async ({ page, mocky, API_URL }) => {
      const ideas = [generateIdea(), generateIdea()];
      mocky.route(`${API_URL}/ideas**`, async (route) => {
        return route.fulfill({
          body: JSON.stringify({
            data: ideas,
            pagination: {
              page: 1,
              limit: 10,
              total: ideas.length,
              totalPages: 1,
              prevPage: null,
              nextPage: null,
            },
          }),
          status: 200,
        });
```

```
      });

      await page.goto('/ideas');

      await expect(page.getByTestId('idea-list-item').first()).toContainText(
        ideas[0].title,
      );
      await expect(page.getByTestId('idea-list-item').nth(1)).toContainText(
        ideas[1].title,
      );
    });
  });

  test.describe('Error handling', () => {
    test('shows error when ideas fail to load', async ({
      page,
      mocky,
      API_URL,
    }) => {
      mocky.route(`${API_URL}/ideas**`, (route) => {
        return route.fulfill({
          body: JSON.stringify({ message: 'Failed to fetch ideas' }),
          status: 500,
        });
      });

      await page.goto('/ideas');

      await expect(page.getByTestId('ideas-skeleton')).not.toBeVisible();

      await expect(page.getByText('Failed to fetch ideas')).toBeVisible({
        timeout: 10000,
      });
    });
  });

  test.describe('Search and filter', () => {
    // ...
  });

  test.describe('Pagination', () => {
    // ...
  });
});
```

This integration test verifies that our ideas page works correctly by testing it in a real browser with different API responses. Let's break down the key concepts:

- **Mocking API routes** - The `mocky.route()` function intercepts network requests and returns mock data. This gives us complete control over what data our application receives. We can test success scenarios, error scenarios, empty states, and edge cases without needing a real backend. All we need is to mock the API routes that are used in the application and perform assertions on the page.
- **Testing user interactions** - We interact with the page exactly as a user would: filling in search inputs, clicking filter buttons, changing sort options. The tests verify that these interactions produce the expected results.
- **Descriptive test organization** - We organize tests into describe blocks by feature area: Display, Error handling, Search and filter, and Pagination. This makes it easy to understand what we're testing and to find specific tests later.

The power of integration tests is that they test realistic scenarios. We're not testing whether a component renders correctly in isolation, we're testing whether users can actually search for ideas, filter by tags, and load more results. These are the behaviors that matter to our users.

Integration tests give us confidence that features work correctly without the maintenance burden of end-to-end tests. But things can still go wrong, so there's still value in testing the complete system with both frontend and backend.

End-to-end testing

End-to-end testing is where we test the complete application with both frontend and backend running together. Unlike integration tests where we mock the API, end-to-end tests are executed in a production-like environment where they hit the API, interact with the database, and verify that the entire system works as a cohesive whole.

These tests are the most expensive to write and maintain. They're slower to run, more likely to be flaky, and require more infrastructure. But they're also the most realistic since they test exactly what users will experience in production.

Because of their cost, we use end-to-end tests selectively. We focus on critical user journeys, the core flows that must work for our application to be useful. For our application, the critical journeys are user authentication, creating an idea, reviewing someone else's idea, and managing a user's profile.

The following diagram shows how end-to-end tests interact with the full system:

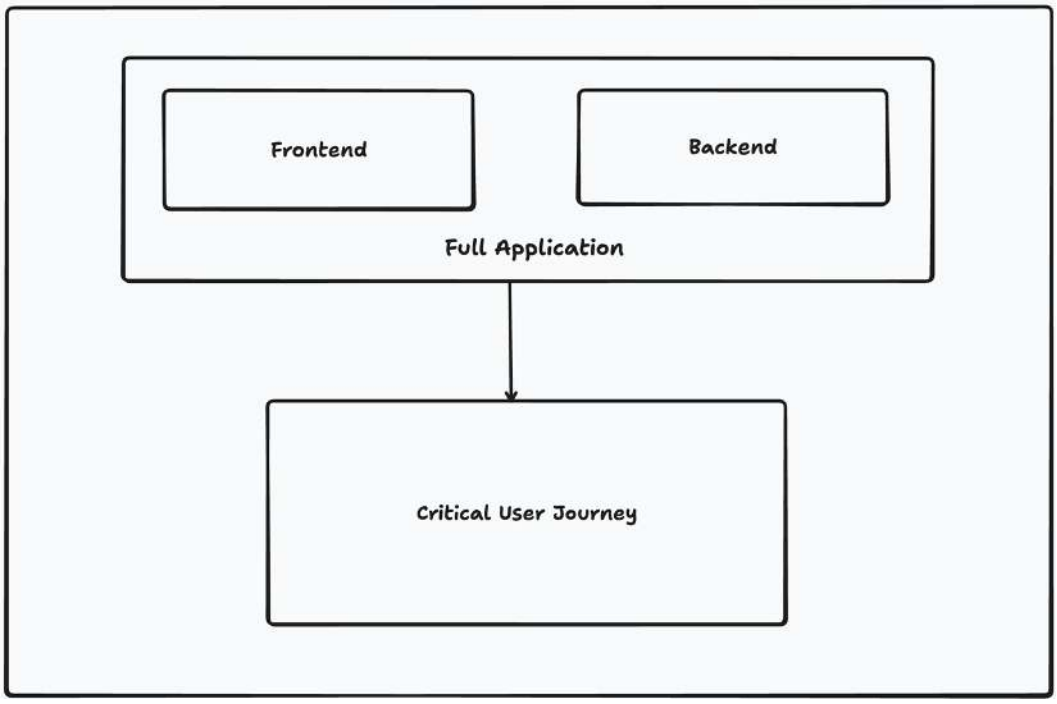


Figure 11.4 – End-to-end testing

Unlike integration tests where we mocked the API, end-to-end tests go through the entire stack. The browser makes requests to our frontend, which communicates with the real API, which reads and writes to the real database. Nothing is mocked and the entire flow is tested.

We'll use Playwright to run these tests in a headless browser, simulating real user interactions:

```
// testing/e2e/smoke.spec.ts

import { test, expect } from '../fixtures/e2e.fixture';
import { generateIdea, generateReview, generateUser } from '../test-data';

test.describe('Smoke Test', () => {
  const testUser = {
    ...generateUser(),
```

```
  password: 'password123',
};

const testIdea = generateIdea();
const testReview = generateReview();
const updatedReviewContent = 'Updated review content';

test('Complete User Journey', async ({ page }) => {
  await test.step('register a new user', async () => {
    await page.goto('/auth/register');

    await page
      .getByRole('textbox', { name: 'Username' })
      .fill(testUser.username);
    await page.getByRole('textbox', { name: 'Email' }).fill(testUser.email);
    await page.getByLabel('Password').fill(testUser.password);

    await page.getByRole('button', { name: 'Register' }).click();

    await expect(page).toHaveURL('/dashboard');
  });

  await test.step('log out the user', async () => {
    await page.getByRole('button', { name: 'Logout' }).click();
    await expect(page).toHaveURL('/');
  });

  await test.step('login with the same user', async () => {
    await page.getByRole('link', { name: 'Login' }).click();
    await expect(page).toHaveURL('/auth/login');

    await page.getByRole('textbox', { name: 'Email' }).fill(testUser.email);
    await page.getByLabel('Password').fill(testUser.password);

    await page.getByRole('button', { name: 'Login' }).click();

    await expect(page).toHaveURL('/dashboard');
  });

  await test.step('create a new idea', async () => {
   // ...
  });

  await test.step('edit the idea', async () => {
    // ...
  });

  await test.step('delete the idea', async () => {
    // ...
  });
```

```
    await test.step('discover ideas', async () => {
      // ...
    });

    await test.step('write a review for an idea of someone else', async () => {
      // ...
    });

    await test.step('edit the review', async () => {
      // ...
    });

    await test.step('delete the review', async () => {
      // ...
    });

    await test.step('update the user profile', async () => {
      // ...
    });
  });
});
```

This smoke test walks through the entire user journey from registration to profile updates. It's called a "smoke test" because it verifies that nothing is on fire and the core functionality works from end to end.

Let's look at what makes this test effective:

- **Test steps for clarity** - Use `test.step()` to break the test into logical sections. This makes the test easier to understand and helps us identify exactly which step failed if something goes wrong.
- **Complete user journey** - Cover everything a user might do: register, log out, log in, create content, edit it, delete it, discover other content, interact with it, and update their profile. This is realistic usage.
- **No mocking** - Unlike integration tests, we don't mock any API calls. This test hits the real backend, creates real database records, and verifies that the entire system works together.

End-to-end tests like this are our final safety net. If this test passes, we have strong confidence that our application works in production. But we usually don't need many of these since they are more expensive and complex to set up and run, so one or two comprehensive smoke tests covering the critical paths is usually enough depending on the application complexity.

With unit tests for complex logic, integration tests for features, and end-to-end tests for critical journeys, we have a testing strategy that catches bugs at multiple levels while keeping our test suite maintainable and fast.

Summary

In this chapter, we learned how to test our application using three complementary approaches. Each type of test serves a different purpose and together they create a comprehensive testing strategy.

We started with unit tests, which verify that individual functions, hooks, and components work correctly in isolation. These tests are fast and focused, making them perfect for testing utility functions and complex logic that's hard to test otherwise.

We then moved to integration tests, which test how multiple parts of our application work together. These are the most valuable tests for most applications because they verify realistic scenarios without the maintenance burden of end-to-end tests. By mocking only the API layer, we can test our application exactly as users experience it while maintaining fast, reliable tests.

Finally, we covered end-to-end tests that verify the entire system works from start to finish. These tests are expensive to maintain, so we use them selectively for critical user journeys. When they pass, they give us confidence that our application works in production.

The key insight is to focus most of our testing effort on integration tests, use unit tests for complex isolated logic, and reserve end-to-end tests for the most important user flows. This balanced approach gives us strong confidence without slowing down our development.

With a solid testing strategy in place, we can refactor code, add new features, and fix bugs knowing that our tests will catch any regressions. This is what lets us move fast while keeping our application reliable.

Get this book's PDF copy, code bundle, and more

Scan the QR code (or go to `packtpub.com/unlock`). Search for this book by name, confirm the edition, and then follow the steps on the page.

UNLOCK NOW

Note: Have your invoice handy. Purchases made directly from the Packt website don't require an invoice.

12

Going to Production

Our application is finally ready to meet its first users. We've built all the features, set up linting, written unit and integration tests, and added end-to-end tests. All of that gives us confidence that the code works. But right now, every one of those checks has to be run manually on our local machine. Every time we want to make a change to the application, we need to run all the scripts ourselves and then trigger the deployment manually. That's slow, error-prone, and honestly pretty tedious.

What we really want is for all of this to happen automatically. Every time we push code changes, a pipeline should run the checks, and if everything passes, deploy the new version without manual steps. That's exactly what CI/CD gives us.

We'll cover the following topics:

- What is CI/CD
- Using GitHub Actions
- Configuring the pipeline for continuous integration
- Configuring the pipeline for continuous deployment

By the end of this chapter, we'll have a fully automated CI/CD pipeline that runs our checks on every push and deploys the application to Render whenever all checks pass.

Technical requirements

Before we get started, we need to set up our project. To be able to develop our project, we'll need the following things installed on our computer:

- Node.js version 24 or above. npm version 11 or above ships with Node. We can confirm that by executing `node -v` and `npm -v` in the terminal. There are multiple ways to install Node.js and npm. Here is a great article that goes into more detail: https://www.nodejsdesignpatterns.com/blog/5-ways-to-install-node-js .
- VS Code (optional) is a popular editor for JavaScript and TypeScript: open source, solid TypeScript support, and extensions. It can be downloaded from https://code.visualstudio.com .

The code for this book is available at the book's repo. To access the repository link, follow the steps in the *"Download the example code files"* section in the *Preface* . Clone it and enter the repository root:

```
git clone https://github.com/PacktPublishing/React-Application-Architecture-for-Production-Second-Edition.git
```

The repository contains chapter folders with the code for each chapter, plus a shared `api` folder with the API server used across all chapters.

We are in *Chapter 12* so we need to navigate to the `chapter-12` directory:

```
cd React-Application-Architecture-for-Production-Second-Edition/chapter-12
```

Then we need to install dependencies:

```
npm install
```

We also need to provide the environment variables:

```
cp .env.example .env
```

Now we should have the frontend ready running at http://localhost:5173 .

We also need to have our API server running.

Let's open a new terminal window and navigate to the `api` directory:

```
cd React-Application-Architecture-for-Production-Second-Edition/api
```

Now we need to run the setup script for *Chapter 12* to configure everything for us:

```
npm run setup 12
```

Then we need to run the API server:

```
npm run dev
```

We should see the API server running on `http://localhost:9999` . For more information about the setup details, check out the `README.md` file.

What is CI/CD?

Continuous integration/continuous deployment (CI/CD) is the practice of delivering application changes to users in an automated, reliable way. Instead of manually running checks and clicking deploy, the pipeline does it for us every single time.

The two parts each serve a different purpose:

- Continuous Integration (CI) is the automated process of verifying that code changes are correct before they get merged. This means running builds, tests, and any other quality checks we care about.
- Continuous Deployment (CD) is the automated process of pushing those verified changes to a production server so users get the latest version without anyone having to manually trigger a release.

For our application, the process will look like this:

1. Run all code quality checks such as linting, type checking, and format checking
2. Run unit tests
3. Run integration tests
4. Run end-to-end tests
5. If all of the above pass on the main branch, deploy the application to production

The following diagram shows how all of these steps fit together.

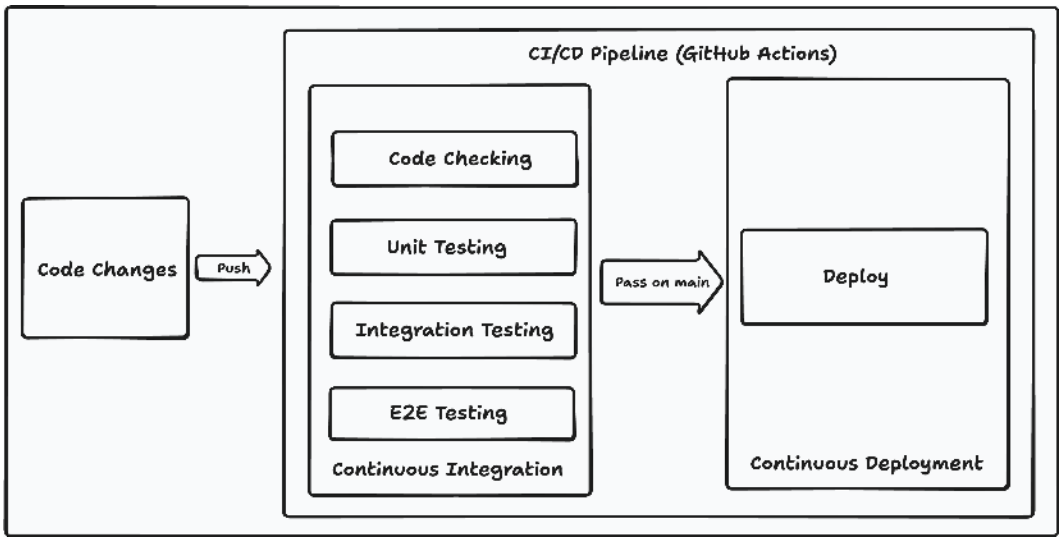


Figure 12.1 – Pipeline overview

This kind of setup really pays off as a team grows. Instead of relying on each developer to remember to run checks and deploy correctly, the pipeline handles it consistently every time code is pushed.

Now that we understand what CI/CD is, let's look at the tool we'll use to build our pipeline.

Using GitHub Actions

For automating our CI/CD pipeline, we will use **GitHub Actions** , a platform built directly into GitHub. We define automated workflows that run in response to things that happen in our repository, like a push or a pull request, and GitHub takes care of running them. There is no external CI server to set up or maintain.

Before we write our first workflow, let's get familiar with the key concepts so we can understand the terminology used in GitHub Actions.

Workflows

A **workflow** is an automated process made up of one or more jobs. We define workflows as YAML files inside the `.github/workflows` folder of our repository. A workflow runs whenever a specific event fires such as a push or a pull request. We can also trigger one manually from the GitHub UI using the `workflow_dispatch` event. A repository can have as many workflows as we need.

Events

An **event** is what kicks off a workflow. It could be something like pushing code to the repository, opening a pull request, or even a scheduled time. Events can also be triggered externally via an HTTP request, which is useful for more advanced setups.

Jobs

A **job** is a set of steps that run together as part of a workflow. Each step is either a shell command we write or a pre-built action. By default, all jobs in a workflow run in parallel, though we can configure a job to wait for another to finish first.

Steps

A **step** is a single unit of work within a job, either a shell command we write or an action from the marketplace. Steps within a job always run sequentially, and each step shares the same environment as the ones before it.

Actions

An **action** is a reusable piece of work that handles a common task. The GitHub Actions Marketplace (`https://github.com/marketplace?type=actions`) has many pre-built actions. Think of them like npm packages, but for CI pipelines. We'll use a few of them, like actions for checking out code, setting up Node.js, and deploying to a platform like Render.

Runners

A **runner** is the server that actually executes a workflow when it's triggered. GitHub provides hosted runners on Ubuntu, macOS, and Windows. If we need something more custom, we can also set up our own self-hosted runners.

Now that we are familiar with the building blocks, let's put them together and build the CI/CD pipeline. Let's start with the CI pipeline.

Configuring the pipeline for continuous integration

Our CI pipeline will have four jobs running in parallel: **code quality checks, unit tests** , **integration tests** , and **end-to-end tests** . Running them in parallel

means we get results faster since there's no waiting for one job to finish before the next one starts.

Let's create the `.github/workflows/ci.yml` file and start with the top-level workflow definition:

```yaml
# .github/workflows/ci.yml

name: Continuous Integration
on:
  - push
  - pull_request
jobs:
  # We will define the jobs here
```

Let's break down what we've defined here:

- `name` - The display name of the workflow shown in the GitHub Actions UI
- `on` - The events that trigger this workflow. We're listening to push and pull_request, so the pipeline runs whenever code is pushed to any branch or a pull request is opened
- `jobs` - The container where all our jobs will live

Now let's add the first job, which handles code quality checks:

```yaml
# .github/workflows/ci.yml

# ...
jobs:
  code-checks:
    name: Code Checks
    runs-on: ubuntu-latest
    defaults:
      run:
        working-directory: ./chapter-12
    steps:
      - uses: actions/checkout@v6
      - uses: actions/setup-node@v6
        with:
          node-version: 24
      - run: npm ci
      - run: npm run lint
      - run: npm run typecheck
      - run: npm run format:check
```

Let's break down how this job works:

- `runs-on: ubuntu-latest` - Runs this job on the latest Ubuntu runner, which is the standard choice for Node.js projects
- `defaults.run.working-directory` - Sets the working directory for all run steps to `./chapter-12` . Our repository holds multiple chapters, so we need to make sure the commands run against the right folder.
- `actions/checkout@v6` - Uses a GitHub action from the marketplace that checks out our repository code onto the runner so the following steps can actually access it.
- `actions/setup-node@v6` - Uses a GitHub action from the marketplace that installs Node.js version 24 on the runner
- `npm ci` - Installs dependencies from `package-lock.json` exactly as it is without updating anything, making it faster and producing a clean, reproducible install, which is exactly what we want in CI.
- `npm run lint` ,`npm run typecheck` ,`npm run format:check` - Runs each quality check in sequence. If any one of them fails, the job stops and the whole pipeline is marked as failed.

Now let's add the unit tests job:

```yaml
# .github/workflows/ci.yml

# ...
jobs:
  # ...
  unit-tests:
    name: Unit Tests
    runs-on: ubuntu-latest
    defaults:
      run:
        working-directory: ./chapter-12
    steps:
      - uses: actions/checkout@v6
      - uses: actions/setup-node@v6
        with:
          node-version: 24
      - run: npm ci
      - run: cp .env.example .env
      - run: npm run test:unit
```

This follows the same pattern as before, with one small addition: we copy `.env.example` to `.env` before running the tests. Our application needs environment variables to run, and `.env.example` has all the required keys with safe placeholder values that work in CI.

Next, the integration tests job:

```
# .github/workflows/ci.yml

# ...
jobs:
  # ...
  integration-tests:
    name: Integration Tests
    runs-on: ubuntu-latest
    defaults:
      run:
        working-directory: ./chapter-12
    steps:
      - uses: actions/checkout@v6
      - uses: actions/setup-node@v6
        with:
          node-version: 24
      - run: npm ci
      - run: npx playwright install --with-deps
      - run: cp .env.example .env
      - run: npm run test:integration
      - uses: actions/upload-artifact@v4
        if: failure()
        with:
          name: playwright-report-integration
          path: chapter-12/playwright-report/
          retention-days: 30
```

Because our integration tests use Playwright, we need to install its browsers with `npx playwright install --with-deps` before running the tests. The last step is interesting; it uploads the Playwright HTML report as a downloadable artifact, but only when the job fails (`if: failure()`). There's no point uploading it on every successful run, but when something breaks, having the report available makes debugging much easier. GitHub keeps it for 30 days before cleaning it up automatically.

And finally, the end-to-end tests job:

```
# .github/workflows/ci.yml
```

```yaml
# ...
jobs:
  # ...
  e2e-tests:
    name: E2E Tests
    runs-on: ubuntu-latest
    defaults:
      run:
        working-directory: ./chapter-12
    steps:
      - uses: actions/checkout@v6
      - uses: actions/setup-node@v6
        with:
          node-version: 24
      - run: npm ci
      - run: npx playwright install --with-deps
      - run: cp .env.example .env
      - run: npm run test:e2e
      - uses: actions/upload-artifact@v4
        if: failure()
        with:
          name: playwright-report-e2e
          path: chapter-12/playwright-report/
          retention-days: 30
```

This is almost identical to the integration tests job. The differences are the test command (`npm run test:e2e`) and the artifact name (`playwright-report-e2e`). Keeping the names separate means we can tell at a glance whether a failure came from integration tests or end-to-end tests.

Once we commit the `ci.yml` file and push it, GitHub picks it up automatically and starts running the pipeline. We can watch all four jobs running in the Actions tab:

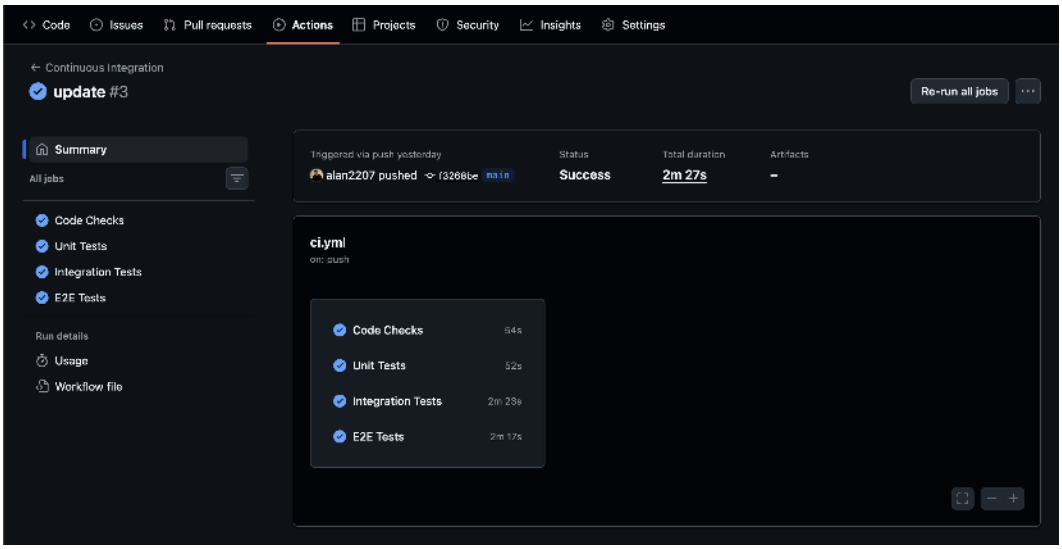


Figure 12.2 – Continuous Integration pipeline

All four jobs run at the same time. The pipeline is marked as successful only when every job passes. If any one of them fails, GitHub notifies us so we can fix it before it reaches production.

With CI sorted, let's move on to the deployment side.

Configuring the pipeline for continuous deployment

There are many different options for deploying a React Router application such as serverless platforms (Vercel, Netlify, and Cloudflare Pages), long-lived server platforms (Fly.io, Render, Railway) or self-hosted solutions on a VPS. For our application, we will use `**Render**` to deploy our application as it is simple to use and has a generous free tier.

Our deployment pipeline has a single job that triggers a deployment, but only after CI passes and only for pushes to the `main` branch. We definitely don't want a deployment triggered every time someone pushes to a feature branch.

Creating the deployment workflow

Let's create a new file at `.github/workflows/cd.yml` :

```
# .github/workflows/cd.yml

name: Continuous Deployment
run-name: ${{ github.event.workflow_run.head_commit.message }}
```

```yaml
on:
  workflow_run:
    workflows: [Continuous Integration]
    branches: [main]
    types: [completed]
jobs:
  deploy:
    runs-on: ubuntu-latest
    if: github.event.workflow_run.conclusion == 'success' && github.event.workflow_run.event ==
'push'
    env:
      RENDER_SERVICE_ID: ${{ secrets.RENDER_SERVICE_ID }}
      RENDER_API_KEY: ${{ secrets.RENDER_API_KEY }}
    steps:
      - if: env.RENDER_SERVICE_ID != '' && env.RENDER_API_KEY != ''
        uses: JorgeLNJunior/render-deploy@v1.5.0
        with:
          service_id: ${{ secrets.RENDER_SERVICE_ID }}
          api_key: ${{ secrets.RENDER_API_KEY }}
          wait_deploy: true
```

Let's break down the key parts:

- `run-name` - Sets the name of each workflow run to the commit message that triggered it. This makes it easy to see at a glance what is being deployed without having to dig into the details.
- `workflow_run` - This trigger fires whenever another named workflow completes. Instead of triggering on a specific event, we're watching for Continuous Integration to finish on the `main` branch.
- `if: conclusion == 'success' && event == 'push'` - This is the critical gate. The deploy job only runs if CI finishes successfully, and only if it was triggered by a push.
- `JorgeLNJunior/render-deploy@v1.5.0` - A pre-built action that handles the Render deployment for us. It needs our service ID and API key, which we pass in as repository secrets so they never appear in the code.
- `wait_deploy: true` - Keeps the job running until Render finishes the deployment. This means the workflow accurately reflects whether the deployment actually succeeded rather than just that the request was sent.

Before this workflow can deploy anything, we need to create a service on Render and obtain the credentials it needs.

Setting up Render

To get started, go to `https://dashboard.render.com` and log in to the Render dashboard. Click **New** , and select **Web Service** from the dropdown.

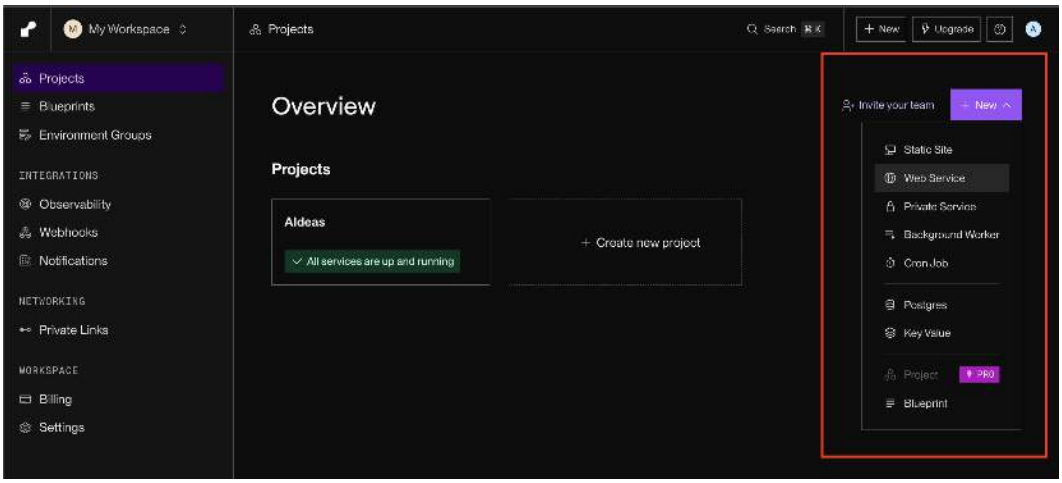


Figure 12.3 – Creating a new service on Render

This takes us to a page where we connect Render to our GitHub repository. We pick the repository that holds our application code.

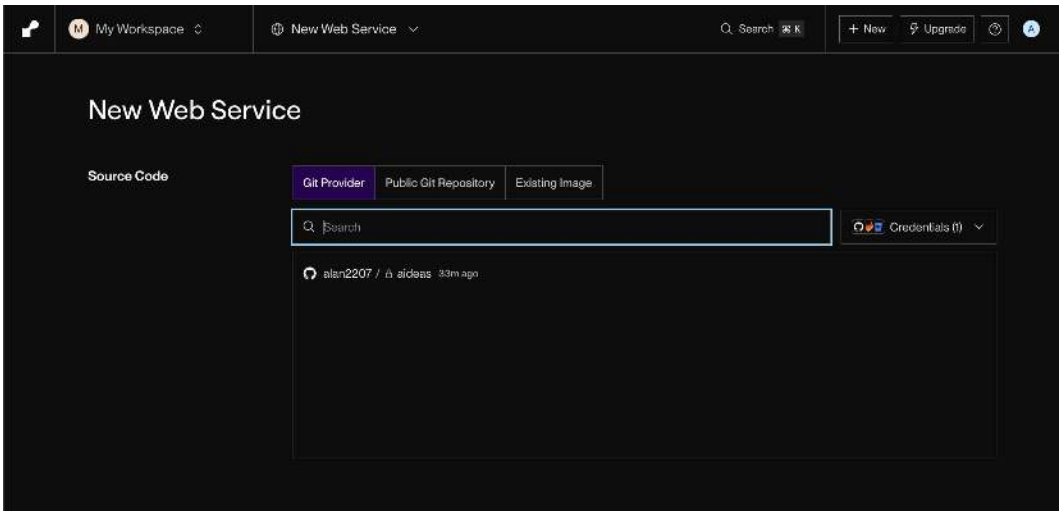


Figure 12.4 – Connect the service to the repository

Once the repository is connected, Render asks us to configure the service. Here we set the service name, the branch to deploy from, the build command, and the start command.

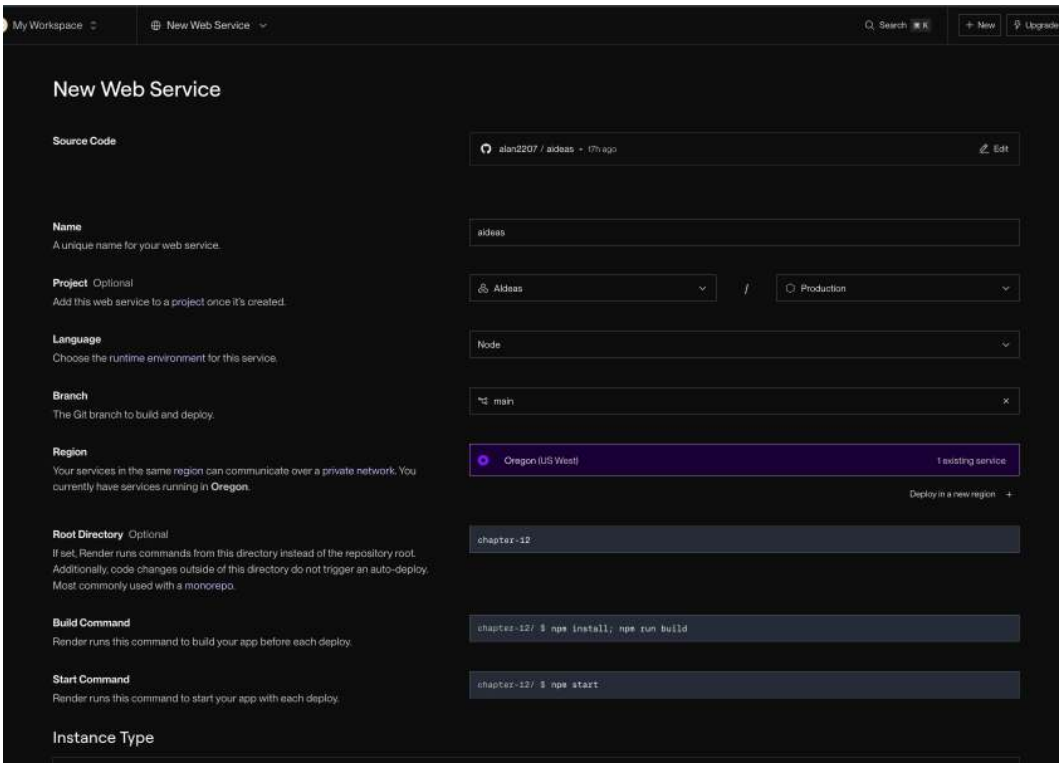


Figure 12.5 – Configure new service

Scrolling down, we can also choose the instance type and add any environment variables the application needs at runtime.

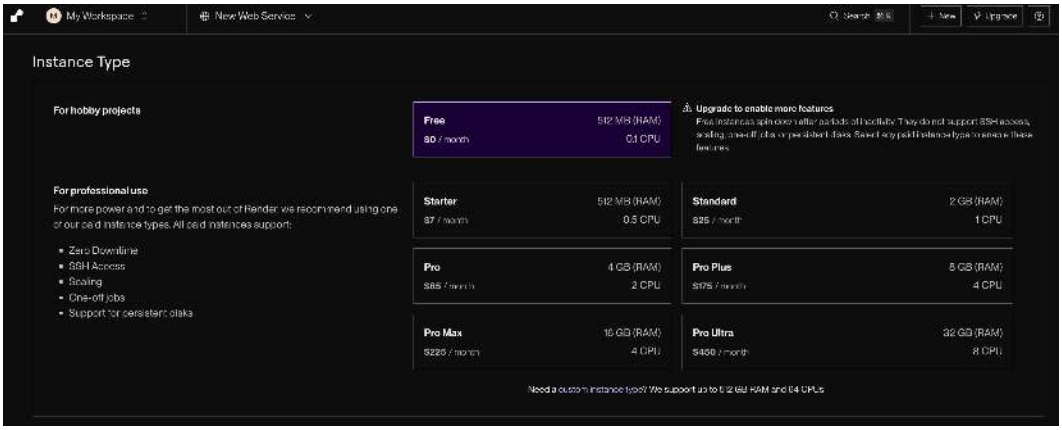


Figure 12.6 – Configure instance and environment

By default, Render automatically deploys on every push to the configured branch. We don't want that because we want GitHub Actions to be the only thing that triggers deployments. Let's expand the **Advanced** section and flip **Auto-Deploy** to **Off** to disable automatic deployments.

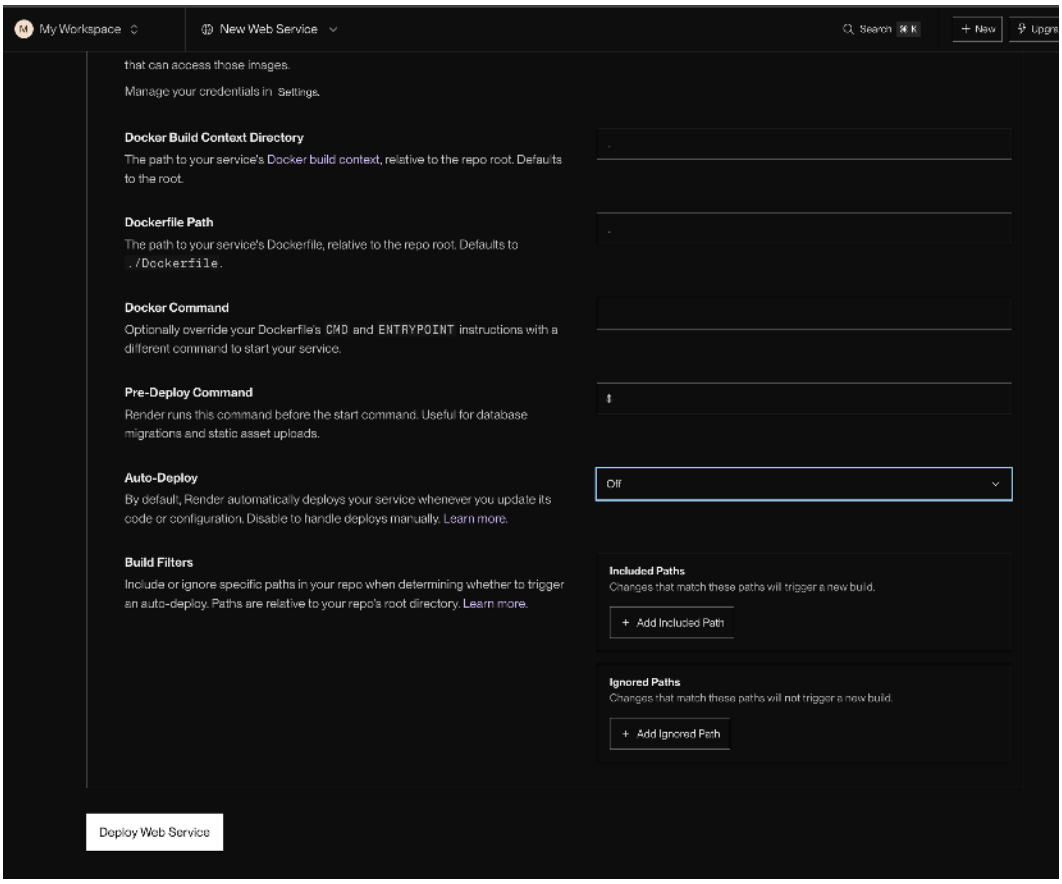


Figure 12.7 – Disable automatic deployments

With that turned off, Render will sit and wait until our CD workflow tells it to deploy.

Getting the service ID and API key

Now we need two things from Render: the service ID and an API key. Our workflow will use these to authenticate with Render's API and start the deployment.

The service ID is right there on the service's settings page.

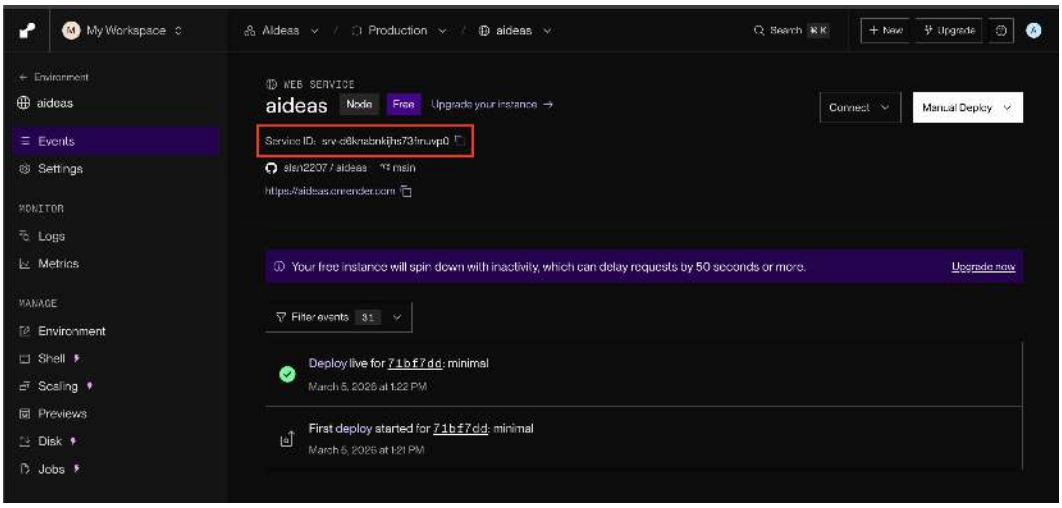


Figure 12.8 – Get the service ID

For the API key, we head to **Account Settings** and then to the **API Keys** section to create a new key.

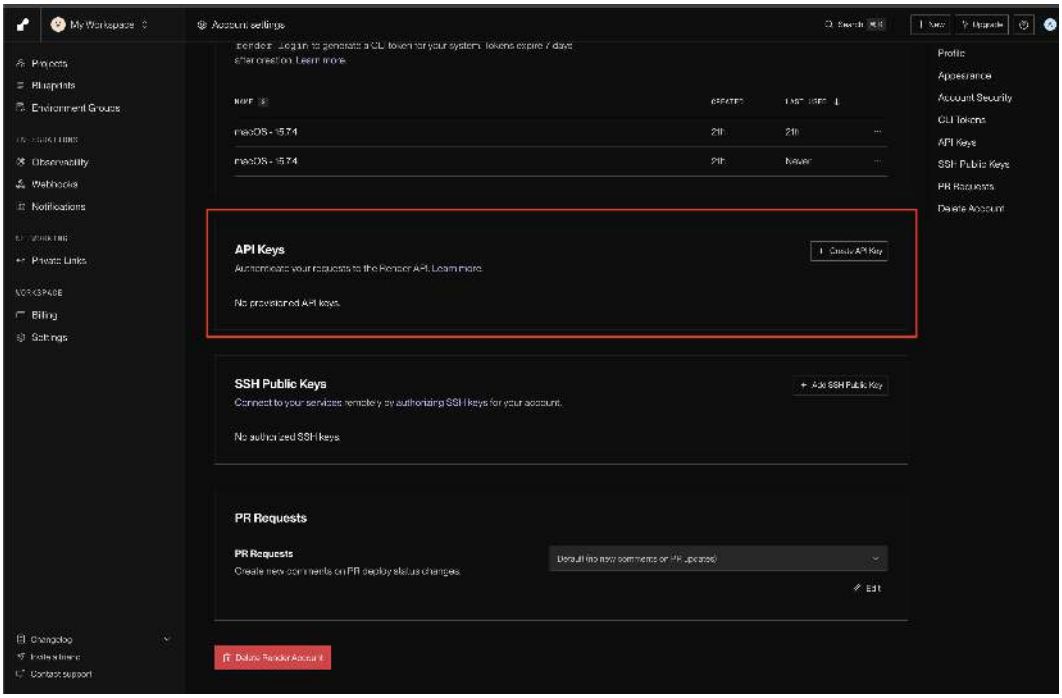


Figure 12.9 – Get the API key

Now we need to give our GitHub workflow access to these credentials.

Adding secrets to the repository

We provide these credentials to the GitHub workflow through repository secrets, which are encrypted values that workflows can read at runtime but

never appear in our code or logs.

Go to the **Settings** tab of the repository, navigate to **Secrets and Variables → Actions** , and add two secrets: `RENDER_SERVICE_ID` and `RENDER_API_KEY` .

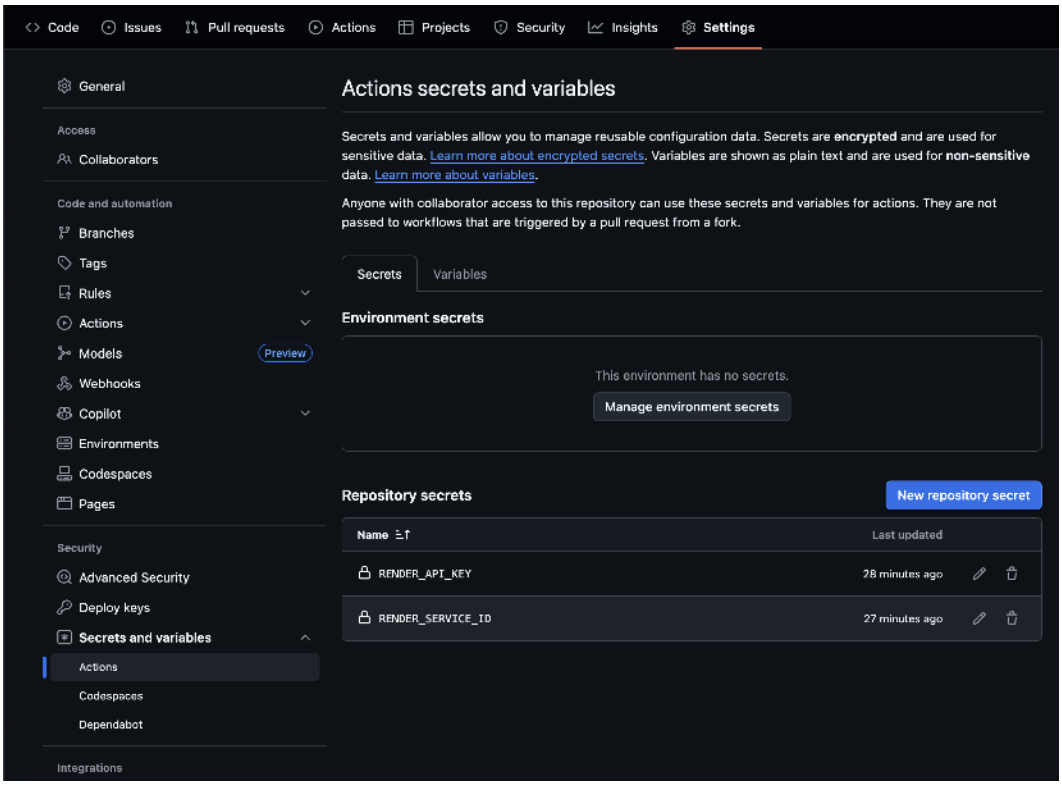


Figure 12.10 – Add the service ID and API key to the repository secrets

These names match exactly what we referenced in `cd.yml` with `${{ secrets.RENDER_SERVICE_ID }}` and `${{ secrets.RENDER_API_KEY }}` . When the workflow runs, GitHub automatically injects the values.

Verifying the deployment

Let's push a new commit to `main` and see the whole thing in action. The CI pipeline runs first, and once it passes, the CD workflow kicks off automatically:

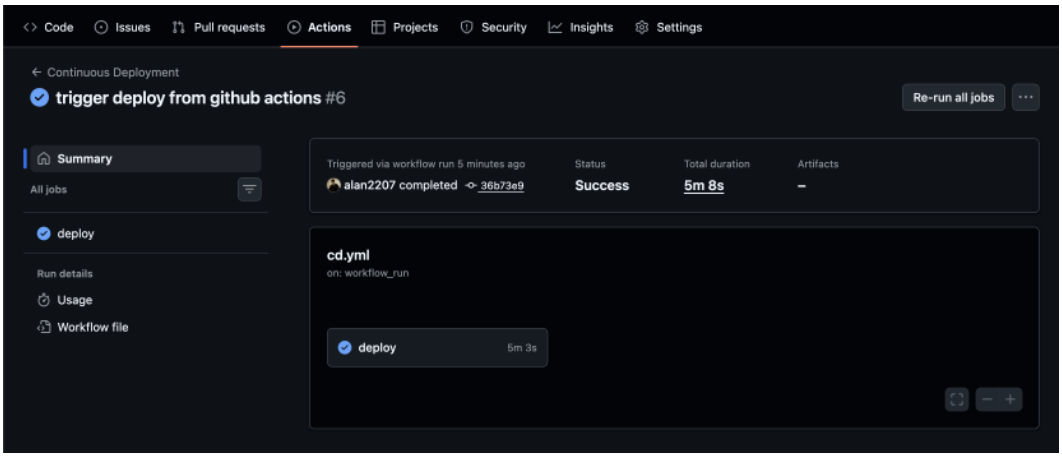


Figure 12.11 – Continuous Deployment pipeline

We can watch the deployment progress in real time. Over on the Render dashboard, we can see the new deployment has been triggered:

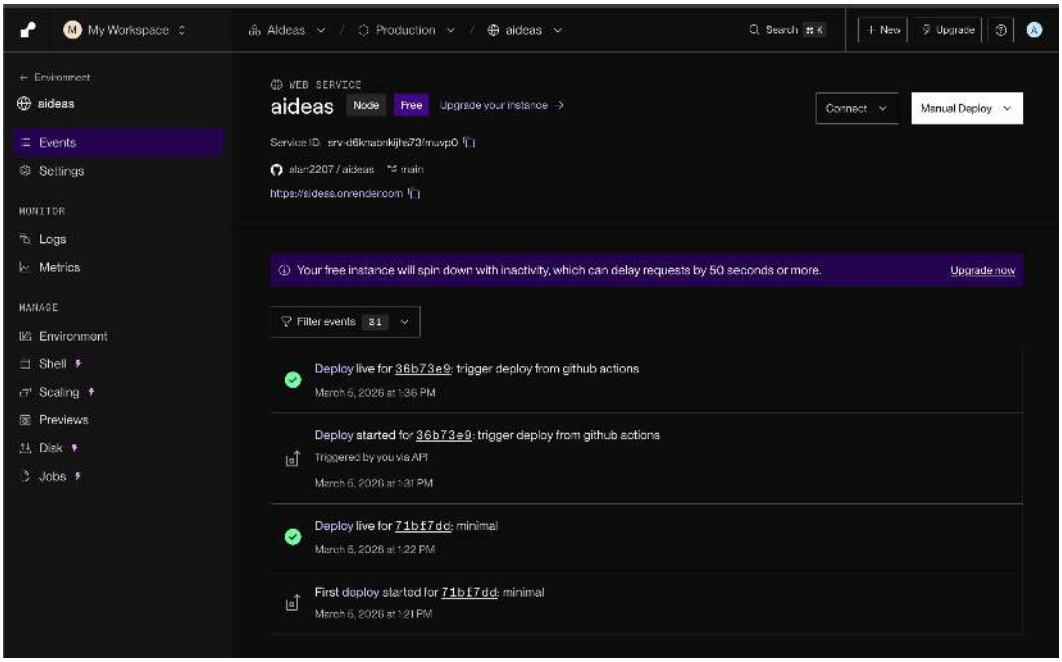


Figure 12.12 – Deployment status on Render

And once it finishes, our application is live.

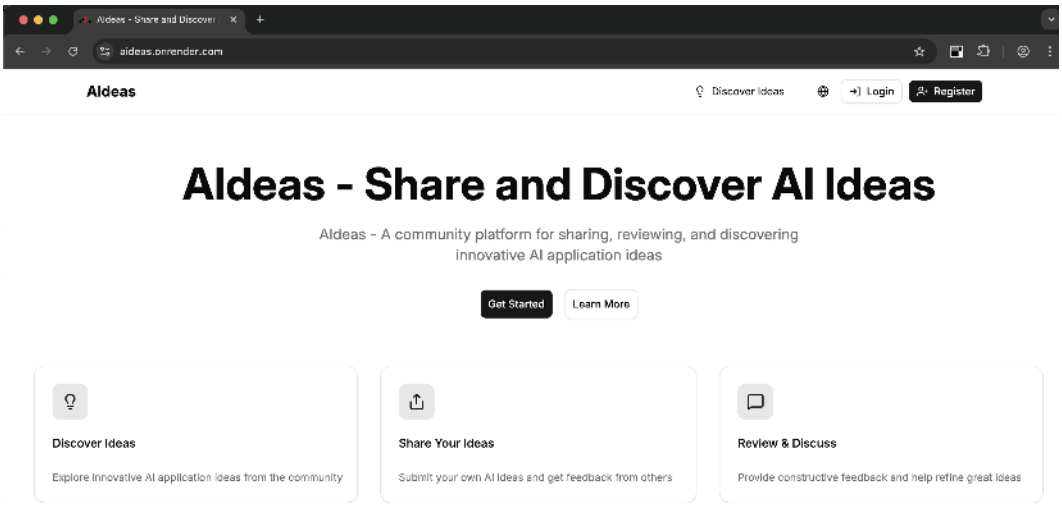


Figure 12.13 – Live application

From now on, every push to main that passes CI will automatically result in a fresh deployment. No more running scripts by hand, no more manually triggering releases—the pipeline handles all of it.

Summary

In this chapter, we set up a complete CI/CD pipeline using GitHub Actions and Render.

We started by understanding what CI/CD is and why it matters. Rather than manually running checks and deploying every time, a pipeline does it reliably and consistently on every push. We then went through the core concepts of GitHub Actions: workflows, events, jobs, actions, runners, and steps, which gave us the vocabulary to understand what we were building.

From there, we created the CI workflow with four parallel jobs covering code quality checks, unit tests, integration tests, and end-to-end tests. We set up Playwright report uploads that only fire on failure so we always have something to debug when things go wrong.

Finally, we built the CD workflow that watches for CI to pass on main and automatically deploys to Render. We configured the Render service, turned off its built-in auto-deploy, and wired up the credentials through GitHub repository secrets.

Get this book's PDF copy, code bundle, and more

Scan the QR code (or go to packtpub.com/unlock). Search for this book by name, confirm the edition, and then follow the steps on the page.

UNLOCK NOW

Note: Have your invoice handy. Purchases made directly from the Packt website don't require an invoice.

13

Evolving the Application

At this point, we have built a full-blown React application from scratch and taken it all the way to production. We have a solid architecture, automated tests, a CI/CD pipeline, and a live deployment.

But reaching production is just the beginning of the journey. Real applications evolve. Teams grow. Requirements change. This chapter introduces several topics that naturally come up as applications mature.

We will cover the following topics:

- Using AI to enforce application architecture
- React Server Components
- Application monitoring and observability
- Feature flags and A/B testing
- Scaling the API layer
- Monorepos
- Microfrontends

By the end of this chapter, we will understand what each of these topics solves, when it becomes worth reaching for, and where to go to learn more.

Using AI to enforce application architecture

We all use AI tools to some extent for generating parts of code. However, one of the most underrated uses of AI coding tools is not generating code, but maintaining the architecture we already defined. As a codebase grows, it is easy for patterns to drift. A component imports something it should not. A feature reaches directly into another feature's internals and breaks our feature isolation. These small violations compound over time until the architecture

exists only in people's heads. Sure, we have already configured tools for linting, testing, and type checking to catch some of these violations, but if we are using AI, we need to tell it about the architecture we want to enforce and how the generated code should fit into it.

AI can be very efficient and helpful, but only if it understands the constraints we have established.

Understanding the right mental model

We can think of AI as a very capable developer who just joined the team. They are smart and fast, but they have no idea about the specific architectural decisions we have made. Without context, they will write code that works but might not fit the patterns we have established.

The key insight is that AI needs to understand our constraints, not just our code. When we describe our architecture clearly—such as which layers exist, what depends on what, and where specific types of logic live—we can use AI to generate code that actually fits.

Context files

Every major AI coding tool has some mechanism for reading project-level instructions, as a plain-text markdown file at the root of the repository that the tool picks up automatically. The specific filename varies: Cursor uses `.cursorrules`, Claude Code reads a `CLAUDE.md`, GitHub Copilot reads `.github/copilot-instructions.md`, and others each have their own convention, and those conventions shift as tools evolve.

The underlying idea is the same for all tools. Pick the file your current tool expects, write the rules there, and treat it like any other source file: versioned, reviewed, and updated alongside the code it describes. If your team uses multiple tools, the simplest approach is to keep one authoritative file and symlink or copy it to whatever names each tool expects.

Writing rules that work

The quality of AI's output depends on the precision of the rules it is given. Vague guidance produces vague results.

This would be a weak rule:

```
Abstract API calls. Do not make them directly in components.
```

If we used AI to create a piece of code that makes API calls and it used our weak rule, it would probably not make API calls directly, but there are no further instructions on how to abstract it, which means there are many ways to do it. Instead, we can use a stronger rule:

```
Abstract API calls. Never call fetch() directly in a component or hook. All API calls go
through src/lib/api.ts using its HTTP verb methods (api.get, api.post, etc.). Every file
in src/features/<name>/api/ exports exactly three things: a fetcher function that validates input and
output with Zod, a query options factory for use in loaders and tests, and a custom hook that wraps
the options factory with useQuery or useMutation.
```

The strong rule specifies the exact constraint, the correct alternative, and the expected shape of the resulting code. AI can apply it reliably. The weak rule requires judgment that the AI doesn't have.

A useful structure for each rule is:

- **What is required** : the positive behavior and its location
- **What is forbidden** : the explicit anti-pattern to avoid
- **Why** : a brief reason, which helps AI reason correctly about edge cases

It is also worth including rules about what *not* to do, not just what to do. Explicit prohibitions like "never call `fetch()` directly from a component" tend to be more effective than positive descriptions alone because they prevent the most common shortcut.

Here is a trimmed version of what a project's context file might look like:

```
# Architecture Guide

## Feature Isolation

Features follow a unidirectional dependency graph enforced by ESLint's
`import/no-restricted-paths` rule in `infra/eslint-import-rules.js`.
Features may only import from other features if explicitly listed in
`allowedFeatures` in that file. If code is needed by multiple features,
move it to the shared layer instead.

## Data Fetching

Abstract API calls. Never call `fetch()` directly in a component. All API calls go through
`src/lib/api.ts`. Every file in `src/features/<name>/api/` exports
three things: a fetcher function (Zod-validated in and out), a query
options factory, and a custom hook.
```

```
...
```

Practical use cases

There are a few places where AI enforcement can be useful:

- Code reviews: Paste a diff and ask the AI to check it against the architecture rules. It can often catch violations that reviewers miss because they are focused on the logic of a change.
- Scaffolding: Ask the AI to scaffold a new component, utility, API call, or test suite following the established patterns. With good context provided, it will create the right folder structure, put files in the right places, and follow naming conventions.
- Refactoring: Describe the changes and ask the AI to implement them without changing behavior. This is tedious work that AI handles well.
- Onboarding: New team members can ask the AI to explain the project structure or walk through a feature and get answers that reflect the actual patterns rather than generic advice.

In all these cases, the AI is most useful when the rules are concrete. The more precisely we describe what belongs where and why, the more reliably it enforces those decisions.

Keeping rules up to date

We need to make sure that the context file is updated regularly. A context file that has not been updated in months might be worse than no file at all. AI will confidently enforce rules that no longer apply and generate code that follows abandoned patterns.

Treating the rules file as living documentation helps. A good practice is to update it as part of any pull request that changes an architectural convention, where the same PR that introduces a new pattern should add the rule describing it. Code review is the right checkpoint: if a PR changes where certain logic lives, the reviewer should ask whether the rules file reflects the new convention.

With architecture in a better position, let's look at another area that becomes increasingly important as the application matures: how it performs and behaves for real users in production.

React server components

React Server Components (**RSC**) are a relatively new model that changes where React components run. Traditionally, all React components run in the browser. With RSC, some components run only on the server; they can fetch data, access databases directly, and never send their JavaScript to the client.

Why server components?

The main appeal is performance. When a component runs on the server, the browser doesn't need to download its JavaScript bundle, wait for it to execute, and then fetch data. The server does the work and sends back finished HTML. This is especially valuable for content-heavy pages or parts of the UI that don't need interactivity.

The mental model takes some getting used to because there is a clear boundary between server components (data fetching, no hooks, no browser APIs) and client components (interactive, can use state and effects). Getting that boundary wrong produces confusing errors.

If you're building an application where initial page load performance is critical or you want to simplify data fetching by running it closer to the database, RSC is worth exploring. The `https://react.dev/reference/rsc/servercomponents` and `https://www.joshwcomeau.com/react/server-components/` are both good starting points. Keep in mind the mental model shift required, and once you understand where the server/client boundary sits, most of the confusion will clear up.

Application monitoring and observability

Deploying to production is when the real surprises start. Users hit edge cases we never thought to test. Errors happen on devices and browsers we don't own and never tested our application on. Pages load slowly under conditions we can't reproduce locally.

Observability is how we find out about these problems before users give up and leave.

Error tracking with Sentry

Sentry is the most widely used error tracking tool in the JavaScript ecosystem. It captures unhandled exceptions, records the stack trace, the user's browser and device, the URL they were on, and recent network requests—everything we need to reproduce and fix the issue.

Adding Sentry to a React application takes a few minutes. After installing the SDK and initializing it with our DSN (the project identifier we get from Sentry's dashboard), errors are captured automatically:

```
// src/app/entry.client.tsx

import * as Sentry from "@sentry/react-router";
import { startTransition, StrictMode } from "react";
import { hydrateRoot } from "react-dom/client";
import { HydratedRouter } from "react-router/dom";
Sentry.init({
  dsn: "https://examplePublicKey@o0.ingest.sentry.io/0",
  sendDefaultPii: true,
});
startTransition(() => {
  hydrateRoot(
    document,
    <StrictMode>
      <HydratedRouter />
    </StrictMode>,
  );
});
```

From there, Sentry groups similar errors together, tracks how often they occur, and can alert us via Slack, email, or PagerDuty when something new breaks.

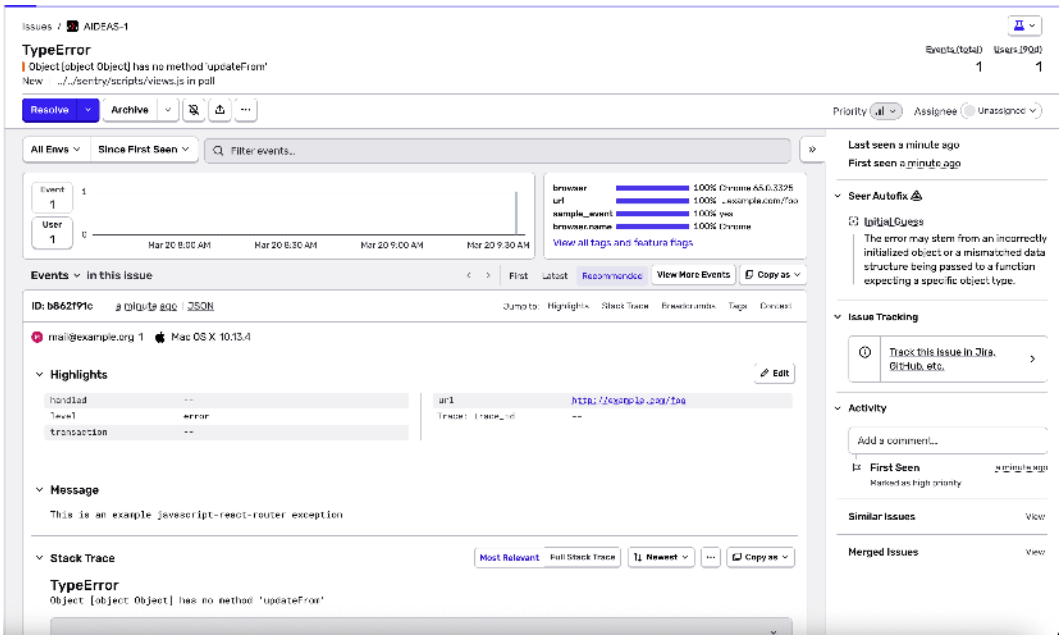


Figure 13.1 – Sentry error tracking dashboard

For most production applications, this is the first monitoring tool worth adding.

Performance monitoring

Beyond errors, we want to understand how fast our application feels to real users. The Web Vitals are a set of metrics defined by Google that measure the parts of performance users actually notice:

- **LCP (Largest Contentful Paint)** - How quickly the main content loads
- **CLS (Cumulative Layout Shift)** - How much the page jumps around during load
- **INP (Interaction to Next Paint)** - How quickly the page responds to user actions

Sentry captures Web Vitals automatically once performance monitoring is enabled. Alternatively, Lighthouse CI can run performance audits as part of the CI pipeline and fail the build if scores drop below a threshold.

Structured logging and alerts

Structured logging means attaching consistent metadata to every error or event you want to track. An unstructured log like this:

```
console.error("Something went wrong");
```

This gets captured and stored, but tells you nothing useful. You can't filter by user, route, or component, it's just a string in a haystack. By passing a structured object instead, each field gets indexed individually:

```
console.error("Something went wrong", {
  route: "/dashboard",
  componentName: "Dashboard",
  userId: "123",
});
```

Now you can search all errors from a specific component, filter by user, and set up alerts when error rates spike after a deploy.

To make this useful in production, these logs need to be stored somewhere so we can browse them. Tools like Datadog and New Relic provide browser SDKs that automatically intercept `console` calls and ship them to their platform where they can be queried and alerted on.

Feature flags

Feature flags are a way to control which users see which features at runtime without deploying new code. Instead of releasing a feature to everyone at once, we wrap it in a flag check. The flag is off by default. We can turn it on for internal users first, then a small percentage of real users, then everyone.

This has a few important benefits. It decouples deployment from release, so we can merge code early without exposing incomplete features. It makes rollbacks instant; if something goes wrong, we flip the flag rather than reverting a deploy.

Simple implementation vs dedicated tools

For a small number of flags, we can start with environment variables or a simple database table. A flag is just a boolean we check before rendering something:

```
// src/lib/flags.ts

export const flags = {
  newIdeaDashboard: import.meta.env.VITE_FLAG_NEW_IDEA_DASHBOARD === 'true',
};

// src/components/dashboard.tsx
```

```
import { flags } from "@/lib/flags";

function Dashboard() {
  if (!flags.newIdeaDashboard) return <LegacyDashboard />;

  return <NewIdeaDashboard />;
}
```

This works fine for a few flags but doesn't scale well. When flags multiply, we want to be able to toggle them without redeploying, target specific users or segments, and audit who changed what and when.

Dedicated tools like LaunchDarkly and Flagsmith provide all of this. They have React SDKs that handle flag evaluation, real-time updates, and targeting rules.

Here is the same example using Flagsmith:

```
// src/components/dashboard.tsx
import { useFlags } from "flagsmith/react";

function Dashboard() {
  const { new_idea_dashboard } = useFlags(["new_idea_dashboard"]);

  if (!new_idea_dashboard.enabled) return <LegacyDashboard />;

  return <NewIdeaDashboard />;
}
```

The flag starts disabled. For example, we can enable it for internal users first, then gradually roll it out to everyone, all from the Flagsmith dashboard, without touching the code.

A/B testing

Feature flags are also the foundation of **A/B testing** . Instead of a simple on/off flag, we assign users to variant A or variant B and measure which performs better based on the collected metrics. Most dedicated feature flag tools support this natively. If you are already using flags for safe deployments, adding experimentation on top is a small step.

```
// src/components/dashboard.tsx
import { useFlags } from "flagsmith/react";

function Dashboard() {
  const { new_idea_dashboard } = useFlags(["new_idea_dashboard"]);
```

```
  if (!new_idea_dashboard.enabled) return <LegacyDashboard />;

  if (new_idea_dashboard.value === "A") return <NewIdeaDashboardA />;
  if (new_idea_dashboard.value === "B") return <NewIdeaDashboardB />;
}
```

Flagsmith consistently assigns each user to one variant, so the same user always sees the same experience. We compare engagement between the two variants in our analytics tool and roll out the winner to everyone by updating the flag.

Scaling the API layer

REST APIs work well for most applications, but as applications grow, two challenges tend to emerge. First, the frontend needs more data than any single REST endpoint provides, leading to multiple round-trip requests (under-fetching) or bloated endpoints that return everything just in case (over-fetching). Second, different clients (web, mobile, third-party) need the same data shaped differently.

The BFF pattern

The **Backend for Frontend (BFF)** pattern addresses both problems by adding a thin API layer that sits in front of our backend and is tailored to a specific client's needs.

Instead of a generic API that all clients share, each client gets its own BFF. The web BFF knows exactly what data the web app needs and aggregates it in one request. If the underlying data comes from three different services, the BFF handles the fan-out and returns a single, composed response. The following diagram shows how this sits between the clients and the backend:

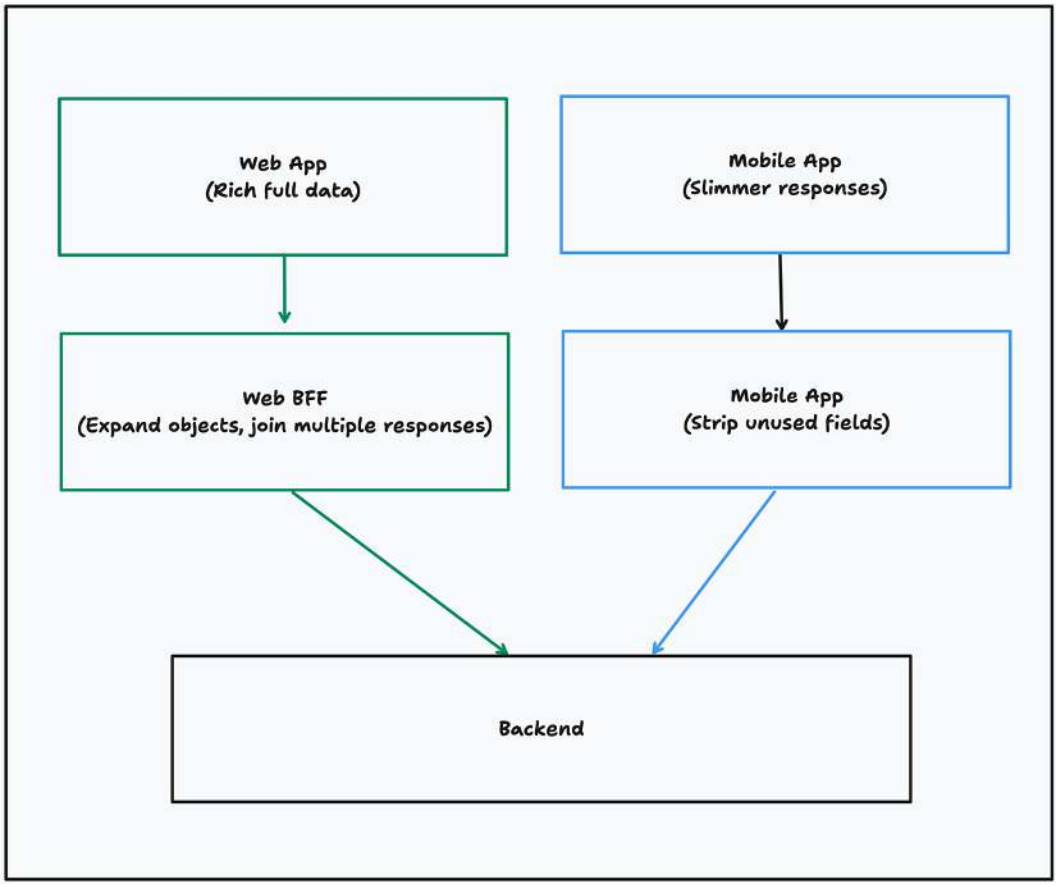


Figure 13.2 – The BFF pattern

In this setup, each client talks only to its own BFF. The BFF handles reshaping the data from backend services and returns exactly what that client needs in a single response.

This keeps backend services clean and generic while giving each client exactly what it needs. The downside is more code to maintain since we are adding another layer between the client and the backend. It makes the most sense when multiple clients with different needs are calling the same backend.

When those multiple clients start to emerge (a web app, a mobile app, perhaps a public API), the question of how to organize all the related code starts to matter. That's where monorepos become relevant.

Monorepos

As projects grow, it's common to end up with multiple related codebases: the main web app, the dashboard app, a mobile app, a component library, shared

TypeScript types, and utility functions. When these live in separate repositories, keeping them in sync becomes painful. A change to the component library requires updating, versioning, publishing, and installing a new version across every consumer. A breaking change in shared types requires coordinating across multiple teams.

A **monorepo** is a single repository that contains multiple applications and packages. Code changes happen together, so a breaking change and all the consumers that need updating can be in the same pull request. The following diagram shows what this looks like in practice:

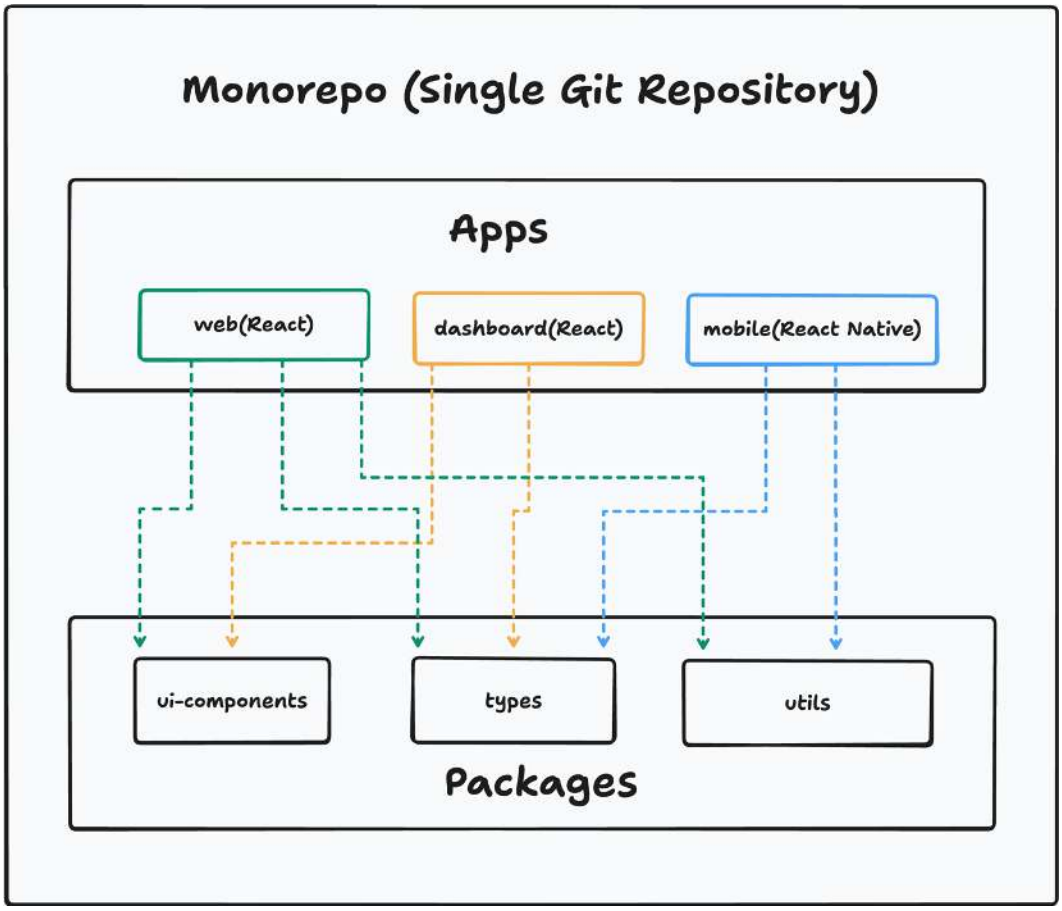


Figure 13.3 – A monorepo with multiple applications

Here, each app is its own package, but they all share code from the common packages in the same repository. A change to a shared package is immediately available to every app without a publish-and-install cycle.

Sharing packages across apps

The main idea behind a monorepo is sharing code between applications. Common candidates are:

- UI component library - Shared React components used by multiple apps
- TypeScript types - API response types shared between the frontend and backend
- Utility functions - Date formatting, validation, and other helpers

Each of these lives as its own package in the monorepo. Applications import them like any other dependency, but changes are reflected immediately upon a new deployment without upgrading the dependencies. This keeps things consistent and reduces the chance of applications drifting out of sync. It also brings the risk of breaking changes that can propagate immediately upon deployment, so we need to be careful.

Microfrontends

We can think of **microfrontends** as microservices for the frontend. Instead of one team owning the entire frontend, different teams own different parts of the application. Each team builds and deploys their slice independently. The following diagram shows what that split looks like:

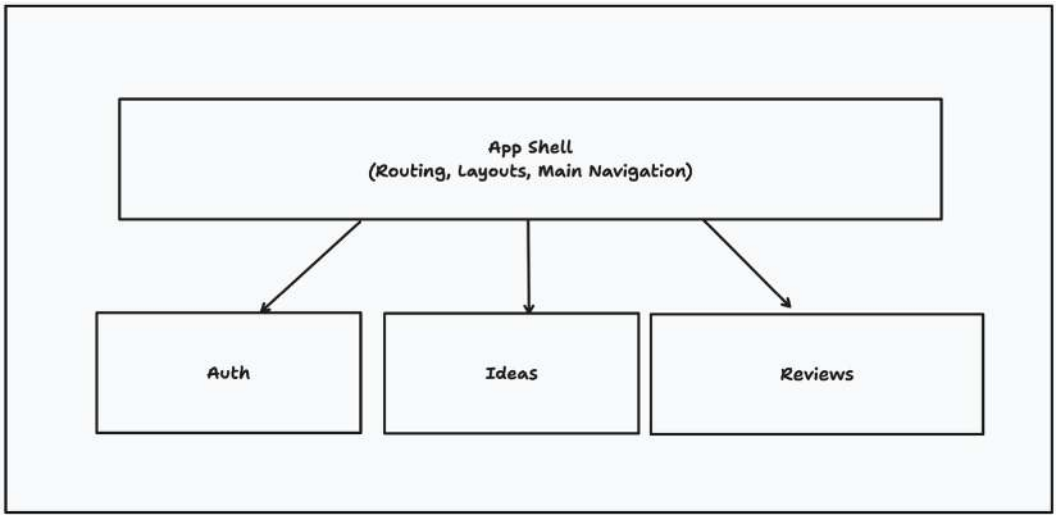


Figure 13.4 – A microfrontend architecture

Each team has full ownership of their slice, their own codebase, their own pipeline, and their own deployment process. A shell application stitches the slices together at runtime.

When microfrontends make sense

Microfrontends solve an organizational problem more than a technical one. If a single team can ship the entire frontend, a monolith is simpler. Microfrontends become worth considering when:

- Multiple large teams need to deploy the frontend independently without coordinating releases
- Different parts of the application have different technology requirements
- The frontend has grown so large that build times and codebase navigation slow teams down.

For most applications, this point never arrives. Introducing microfrontend complexity too early is a classic case of overengineering.

Trade-offs to understand before adopting

Microfrontends come with real costs:

- **Operational complexity** - Each microfrontend is a separate deployment with its own pipeline, versioning, and monitoring
- **Shared dependencies** - Managing shared libraries across independently deployed apps is tricky. Version mismatches can cause subtle bugs or bloated bundles.
- **Consistent UX** - Maintaining a consistent look and feel across teams requires coordination, usually through a shared design system
- **Performance** - Loading multiple independent bundles at runtime has overhead that needs to be managed carefully

The teams that succeed with microfrontends tend to invest heavily in the shared infrastructure (design systems, deployment tooling, contract testing) before splitting the frontend. Without that foundation, the organizational benefits don't outweigh the added complexity.

Summary

In this chapter, we explored several topics that become relevant as an application matures beyond its initial launch.

We started with using AI to enforce architecture, looking at how context files give AI coding tools the constraints they need to generate code that fits established patterns. We covered what makes a rule effective: specificity, explicit prohibitions, a clear reason, and why keeping those rules up to date matters as much as writing them in the first place.

From there, we looked at React Server Components and how moving rendering to the server changes the performance and data-fetching model, then covered application monitoring and observability: capturing errors with Sentry, tracking Web Vitals, and structuring logs so they are queryable in production.

We then explored feature flags as a way to decouple deployment from release, enabling safe rollouts and instant rollbacks without a full redeploy. We saw how the same mechanism naturally extends to A/B testing.

Finally, we examined patterns for scaling the API layer with the BFF pattern, organizing multiple related codebases with monorepos, and splitting frontend ownership across teams with microfrontends, along with the trade-offs and organizational signals that indicate when each is worth the added complexity.

Not all of these will be relevant at the same time. The right move is to understand they exist, recognize the signals that indicate when a topic has become worth investing in, and then dig in when the time comes. The codebase we have built throughout this book is a solid foundation for any of these directions.

Let's remember, good software is never finished; it evolves. Keep iterating, keep learning, and build things worth building.

Get this book's PDF copy, code bundle, and more

Scan the QR code (or go to `packtpub.com/unlock`). Search for this book by name, confirm the edition, and then follow the steps on the page.

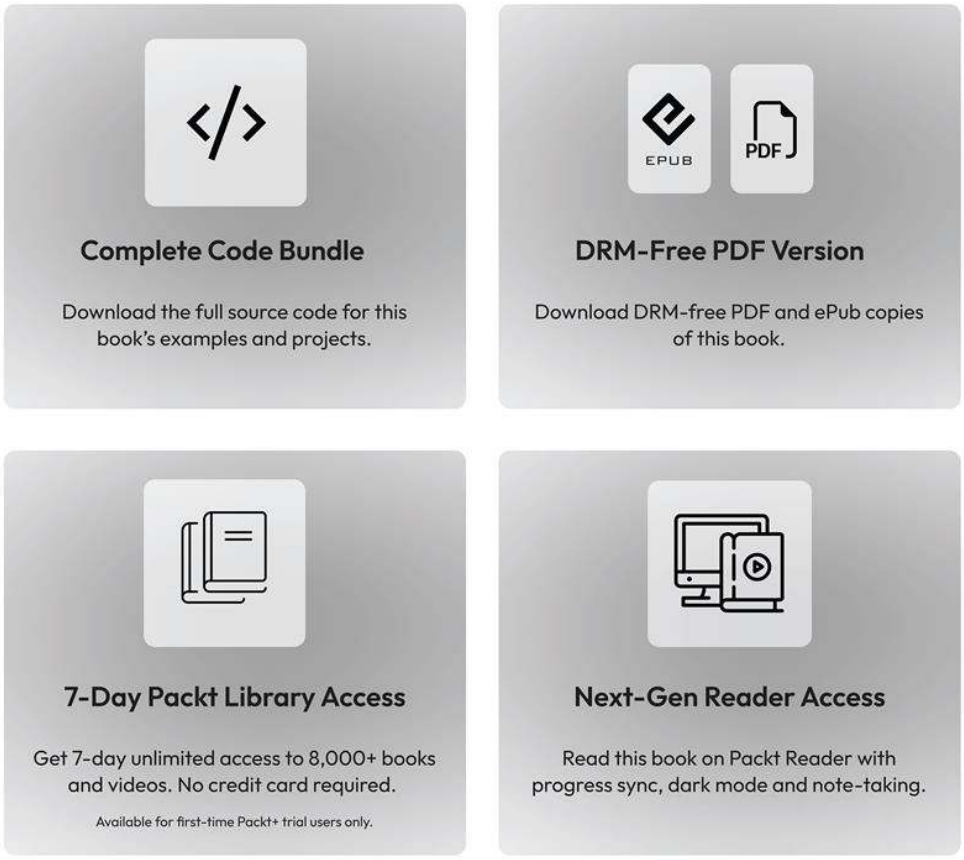
UNLOCK NOW

Note: Have your invoice handy. Purchases made directly from the Packt website don't require an invoice.

14

Unlock Your Exclusive Benefits

Your copy of this book includes the following exclusive benefits:



Follow the guide below to unlock them. The process takes only a few minutes and needs to be completed once.

Unlock this Book's Free Benefits in 3 Easy Steps

Step 1

Keep your purchase invoice ready for *Step 3* . If you have a physical copy, scan it using your phone and save it as a PDF, JPG, or PNG.

For more help on finding your invoice, visit `https://www.packtpub.com/en-us/unlock?step=1.`

NOTE

Note : If you bought this book directly from Packt, no invoice is required. After *Step 2* , you can access your exclusive content right away.

Step 2

Scan the QR code or go to `packtpub.com/unlock` .

On the page that opens (similar to *Figure 1 4 .1* on desktop), search for this book by name and select the correct edition.

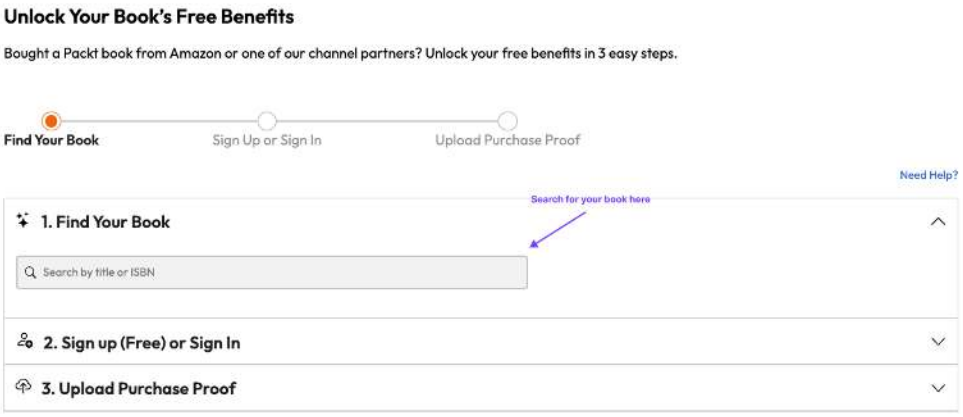


Figure 14.1 – Packt unlock landing page on desktop

Step 3

After selecting your book, sign in to your Packt account or create one for free. Then upload your invoice (PDF, PNG, or JPG, up to 10 MB). Follow the on-screen instructions to finish the process.

Need Help

If you get stuck and need help, visit `https://www.packtpub.com/unlock-benefits/help` for a detailed FAQ on how to find your invoices and more. This QR code will take you to the help page.

NOTE

Note : If you are still facing issues, reach out to `customercare@packt.com` .

packtpub.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

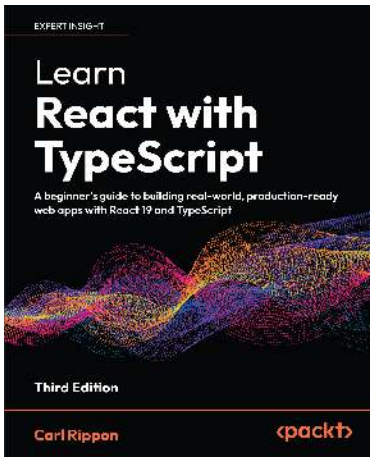
Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

At www.packtpub.com , you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

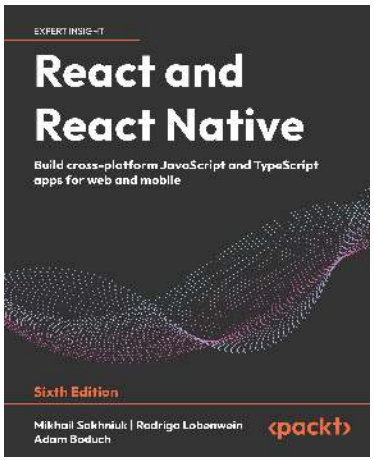


Learn React with TypeScript

Carl Rippon

ISBN: 978-1-83664-317-3

- Apply React styling techniques to create stunning and visually engaging UIs
- Leverage server components to seamlessly integrate with client components for optimized performance
- Fetch and manage data efficiently in React for a smooth, responsive user experience
- Build interactive, validated forms with TypeScript and server actions to handle user input
- Share state efficiently across components using Zustand



React and React Native

Mikhail Sakhniuk, Rodrigo Lobenwein, Adam Boduch

ISBN: 978-1-83702-029-4

- Explore React architecture, component properties, state, and context
- Work with React Hooks for handling functions and components
- Fetch data from a server using the Fetch API, GraphQL, and WebSockets
- Dive into internal and external state management strategies
- Build robust user interfaces (UIs) for web apps using Material-UI
- Perform unit testing for your components with Vitest and mocking techniques
- Manage app performance with server-side rendering, lazy components, and Suspense

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packt.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share your thoughts

Now you've finished *React Application Architecture for Production* , we'd love to hear your thoughts! Scan the QR code below to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.

https://packt.link/r/1836202970

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Index

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

React Application Architecture for Production