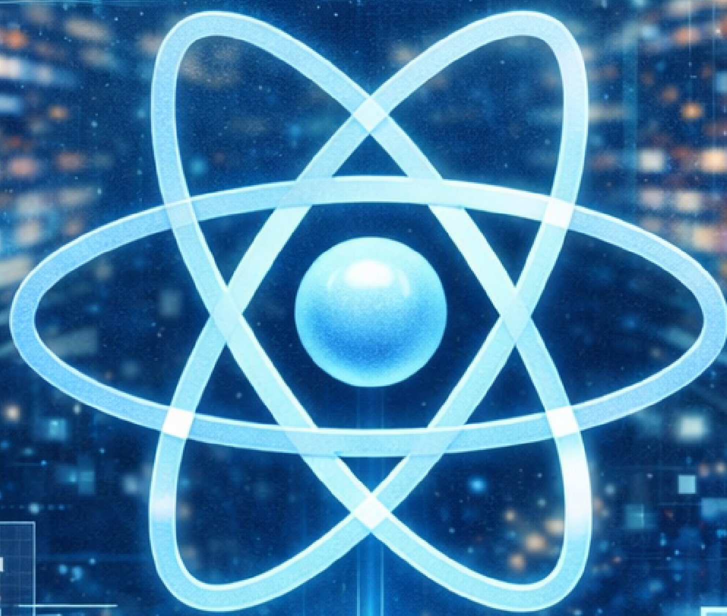
REACT BEST PRACTICES 2026



Advanced Tips and Techniques for Modern Web Development

ENACHE GABRIEL-IONUT

React Best Practices 2026: Scalable Architecture, Design Patterns, Hooks, Performance, and Modern React 19 Techniques — Free Preview

Enache Gabriel-Ionuț

Table of Contents

Preface . . . 1
1. Building a Strong Foundation . . . 3
  1.1. Designing a Scalable React Architecture . . . 3
    1.1.1. Why structure matters . . . 3
    1.1.2. The Fractal Pattern . . . 3
    1.1.3. Folder organization for scalability . . . 3
  1.2. Mastering React without overusing useEffect . . . 7
    1.2.1. When you **don’t** need `useEffect` . . . 7
      Deriving state from props or other state . . . 7
      Resetting state when props change . . . 8
      Handling user events . . . 9
    1.2.2. Common anti-patterns . . . 10
      Effect chains . . . 10
      Fetching data without proper cleanup . . . 10
    1.2.3. The React GG rules . . . 11
2. A Glimpse of What’s Inside . . . 13
  2.1. Stop Memoizing Yourself ***(excerpt)*** . . . 13
    2.1.1. Understanding the React Compiler (React 19+) . . . 13
  2.2. Concurrent UI and Optimistic Updates ***(excerpt)*** . . . 15
    2.2.1. useTransition . . . 15
3. What’s Inside the Full Book . . . 17
  3.1. Part 1 — Building a Strong Foundation . . . 17
  3.2. Part 2 — Managing State and Data Intelligently . . . 17
  3.3. Part 3 — Writing Smarter React Code . . . 17
  3.4. Part 4 — Performance and Modern React Features . . . 18
  3.5. Part 5 — Making React Development Easier . . . 19

Preface

After attending the React Alicante conference in October 2025, I realized that there are many ways to improve my React projects. The thought that I should always be aware of the current React version and closely monitor the changes of the library never came to my mind until then.

And why should I? After all, everything in my projects was working fine, so I thought. The truth is that it wasn't. Not because the actual "working state", but because there were a lot of improvements that could be made that could speed up the development process and improve the quality of the code for years to come… or until a new React version comes out and we need to migrate to these changes. And since the React library is so big, it's not easy to know what's new and what's not.

What I thought back then was that I should simply use the features and patterns that other developers use, because if so many people are using them, they must be good, right? But it wasn't until the conference that I realized that this is not the case.

Attending this conference literally changed my life as a developer and I realized that I was not using the full potential of the React library and I didn't have respect for the library that I was using, so how can I name myself a React developer?

From finding out about new hooks are coming in the new versions of the library to learning about completely new concepts like concurrent React, I realized that there are a lot of things that I didn't know about the library and that I should be aware of.

This is why I decided to write this book. To help you to be more aware of the current React version and use the best practices and newest additions to the library in 2026, which allow you to write the best, cleanest and most modern React code that you can possibly write. I will also show you some tips that I learned along the way to help you structure your project and leverage the best tools and minimize your development time, without lacking efficiency.

Keep in mind that this book **is not intended to be a comprehensive guide to React, nor a guide for beginners**. It is intended to be a guide for mid to senior React developers to help you use the best practices and the newest additions to the library in 2026, which allow you to write the best, cleanest and most modern React code that you can possibly write.

I hope you enjoy reading this book and learn a lot of new things as much as I enjoyed writing and learning from it.

Chapter 1. Building a Strong Foundation

1.1. Designing a Scalable React Architecture

1.1.1. Why structure matters

How you structure a project is one of the most important decisions when building in React.

Suppose you're looking for a `LoadingButton` component. Ignore your IDE's `Command+P` for a moment; imagine manually scrolling through folders to find it.

A `LoadingButton` used inside a `RegisterForm` isn't limited to that form. `LoginForm` might need it too to show a loading state, and so might other components. With that in mind, `LoadingButton` should live in a shared folder, not only inside the `RegisterForm` or `LoginForm` folders.

But what if we need a component used across the entire application? Take a modal, used for login, registration, confirmations, cookie consent, and more. That's a global component; it should be accessible from anywhere. By "globally accessible" we mean placing it alongside other global components in a folder where any part of the app can import it.

1.1.2. The Fractal Pattern

To make placement decisions consistent and clear, we can use the ***"Fractal Pattern"***—a React project structure designed for indefinite scalability. A few benefits include:

- Component locations are logical and predictable
- Enables building complex applications without tangled concerns
- Scaling stays straightforward because functionality is split into small, focused pieces

1.1.3. Folder organization for scalability

Imagine a complex application with many components and pages. Using the Fractal Pattern, a sensible root structure looks like this:

```
src/
├── components/
│   Modal.tsx
│   LoadingButton.tsx
├── routes/
│   HomePage.tsx
├── utils/
│   auth.ts
├── types/
│   props.d.ts
├── hooks/
│   useModal.ts
│   useAuth.ts
├── contexts/
│   AuthContext.tsx
```

- `components/` - global components used across the application
- `routes/` - page-level route components, such as `HomePage`
- `utils/` - application utility functions
- `types/` - shared TypeScript types
- `hooks/` - global hooks
- `contexts/` - React contexts

The key idea is that—except for `contexts/` and `routes/`—these top-level folders hold global code intended for reuse across the app. At the project root, we keep folders that contain global building blocks.

Let's look at a more complete and nested example:

```
src/
├── routes/
│   HomePage.tsx
│   homePage/
│     HeroSection.tsx
│     HomePageNavbar.tsx
│     utils/
│       getHomePageData.ts
│     homePageNavbar/
│       Notifications.tsx
│       notifications/
│         utils/
│           getNotifications.ts
```

As you can see, we have a `HomePage` route. Some pieces are used **only** within `HomePage`, because it composes them.

For example, `HomePage` is composed of `HeroSection` and `HomePageNavbar`.

Because they relate exclusively to `HomePage`, we place them in the `homePage` folder. The convention is to use the component's name in `camelCase` for the folder. Inside that folder, we place the `HeroSection` and `HomePageNavbar` components.

To fetch page data, we place `getHomePageData.ts` in a `utils` subfolder under `homePage`.

Similarly, the `HomePageNavbar` component contains a `Notifications` component, so we place it in the `homePageNavbar` folder.

Lastly, we add a `utils` folder under `notifications` to hold logic for fetching the current user's notifications. With this approach, everything is split into small, manageable pieces, making code easy to find.

If we later need a component shared by both `HomePageNavbar` and `HomePage` (for example, a `LoginButton` shown in the hero section and in the navbar), we should place it in a new `components` folder scoped to the `homePage` feature, like so:

```
src/
├── routes/
│   HomePage.tsx
│   homePage/
│     components/         <--- components folder (shared within homePage)
│       LoginButton.tsx   <--- used by both HomePage and HomePageNavbar
│     HeroSection.tsx
│     HomePageNavbar.tsx
│     utils/
│       getHomePageData.ts
│     homePageNavbar/
│       Notifications.tsx
│       notifications/
│         utils/
│           getNotifications.ts
```

Because it might look a little bit trivial and redundant and you might ask yourself "Why do we need a `LoginButton` component? It's just a button with a loading state and a label". I simply chose `LoginButton` as an example, because implementation details are not important here and I

> want you to focus on the architecture.

Exercise 1: Where should you place a new hook called `useProfile` that fetches the user's profile data?

Place `useProfile` in the project root `hooks/` folder, since multiple parts of the application may reuse it.

```
src/
├── hooks/
│   useProfile.ts
├── routes/
│   HomePage.tsx
│   homePage/
│     components/
│       LoginButton.tsx
│     HeroSection.tsx
│     HomePageNavbar.tsx
│     utils/
│       getHomePageData.ts
│     homePageNavbar/
│       Notifications.tsx
│       notifications/
│         utils/
│           getNotifications.ts
```

Exercise 2: What if you decide that you want a general Navbar component that should be the same across all pages? Where should you place it?

Place it in the project root `components/` folder, since it is a general component that should be used across all pages and delete the `homePageNavbar` component.

```
src/
├── components/
│   Navbar.tsx
├── hooks/
│   useProfile.ts
├── routes/
│   HomePage.tsx
│   homePage/
│     components/
│       LoginButton.tsx
│     HeroSection.tsx
│     utils/
```

```
│       getHomePageData.ts
```

As a conclusion, these are the main principles that you should follow when designing a scalable React architecture:

- All components should be ***CamelCased***.
- All nodes should be ***lowerCamelCased***.
- If a component/hook/util etc. is only used within a specific feature, place it in a folder with the name of the feature.
- If a component/hook/util etc. is used across multiple features, place it in the project root `components/`, `hooks/`, `utils/`, `types/` folders.
- When you want to write tests, consider using a ***tests*** folder and place the tests directory in the same folder as the component (e.g., `/routes/` `tests` `/HomePage.test.tsx`).

1.2. Mastering React without overusing useEffect

`useEffect` might be the most misused hook in React. I know, because I was guilty of this too. And if you've been writing React for more than a year, chances are you've been there too: you need to do something "after a state changes", so you reach for `useEffect`. It feels natural. But most of the time, you just don't need it.

This chapter is about developing an intuition for when `useEffect` is actually the right tool — and when it's not.

1.2.1. When you don't need `useEffect`

Let me show you the most common situations where developers reach for `useEffect` when they really shouldn't.

Deriving state from props or other state

This is probably the number one misuse I see in code reviews. You have some state, and you want to compute another value based on it, so you write something like this:

./components/ProductList.tsx

```
function ProductList({ products }: { products: Product[] }) {
  const [filteredProducts, setFilteredProducts] = useState(products);
```

```
  useEffect(() => {
    setFilteredProducts(products.filter((p) => p.inStock));
  }, [products]);

  return (
    <ul>
      {filteredProducts.map((p) => (
        <li key={p.id}>{p.name}</li>
      ))}
    </ul>
  );
}
```

This pattern causes an unnecessary extra render. React renders once with the old `filteredProducts`, then the effect runs, updates the state, and React renders again. Two renders instead of one, for no good reason. And there's no actual reason to store `filteredProducts` in state at all — you can compute it directly during render:

./components/ProductList.tsx

```
function ProductList({ products }: { products: Product[] }) {
  const filteredProducts = products.filter((p) => p.inStock);

  return (
    <ul>
      {filteredProducts.map((p) => (
        <li key={p.id}>{p.name}</li>
      ))}
    </ul>
  );
}
```

That's it. No `useEffect`, no extra state, no double render. If you're worried about performance, wrap it in `useMemo`. But honestly, only do that if the computation is actually expensive — filtering a small array is fast enough that memoization isn't worth the added complexity.

Resetting state when props change

Another classic. You want to reset some local state when a prop changes, so you write this:

```
function UserProfile({ userId }: { userId: string }) {
  const [comment, setComment] = useState('');

  useEffect(() => {
    setComment('');
  }, [userId]);
```

```
  return (
    <textarea
      value={comment}
      onChange={(e) => setComment(e.target.value)}
    />
  );
}
```

Again, this causes an extra render. The component renders with the stale `comment`, the effect fires and clears it, and React renders again. The correct approach is to use the `key` prop. When `key` changes, React unmounts the component and mounts a fresh one, resetting all local state automatically:

```
<UserProfile key={userId} userId={userId} />
```

That's the whole fix. One line. When `userId` changes, `key` changes, and React creates a completely fresh instance of `UserProfile` with empty state. No `useEffect` needed, no extra render.

This is one of the most underused tricks in React. I use it all the time.

Handling user events

If you're using `useEffect` to react to a user action, stop. `useEffect` is for synchronizing with external systems, not for reacting to events.

```
// Don't do this
function SubmitButton({ onSuccess }: { onSuccess: () => void }) {
  const [submitted, setSubmitted] = useState(false);

  useEffect(() => {
    if (submitted) {
      onSuccess();
    }
  }, [submitted, onSuccess]);

  return <button onClick={() => setSubmitted(true)}>Submit</button>;
}
```

```
// Do this instead
function SubmitButton({ onSuccess }: { onSuccess: () => void }) {
  const handleClick = () => {
    onSuccess();
  };
```

```
  return <button onClick={handleClick}>Submit</button>;
}
```

Keep it simple. If something needs to happen because of a user click, put it in the click handler. Not in an effect that runs when some flag state changes.

1.2.2. Common anti-patterns

Effect chains

One `useEffect` sets some state, which triggers another `useEffect`, which sets more state. This is known as an ***effect chain*** and it's a nightmare to debug. Every time you add another link to the chain, you introduce another render cycle and another potential source of bugs.

```
// Effect chains are a red flag
useEffect(() => {
  setUserData(fetchUserData(userId));
}, [userId]);

useEffect(() => {
  if (userData) {
    setPermissions(calculatePermissions(userData.role));
  }
}, [userData]);

useEffect(() => {
  if (permissions) {
    setMenuItems(buildMenu(permissions));
  }
}, [permissions]);
```

If you find yourself with effects that depend on state set by other effects, that's a strong signal that you're modeling the problem wrong. Step back and think about what you're actually trying to derive, and whether you can do it in a single pass.

Fetching data without proper cleanup

Fetching data in `useEffect` is one of the legitimate uses of the hook, but most implementations you'll find in the wild are broken:

```
// This is broken
function UserCard({ userId }: { userId: string }) {
  const [user, setUser] = useState<User | null>(null);

  useEffect(() => {
```

```
    fetchUser(userId).then((data) => setUser(data)); // no cleanup!
  }, [userId]);

  return user ? <div>{user.name}</div> : <p>Loading...</p>;
}
```

The problem is that if `userId` changes before the first fetch finishes, both requests are in flight simultaneously. Whichever finishes last wins, and you might end up showing the wrong user's data. You need to cancel the previous request, or at least ignore its result:

```
useEffect(() => {
  let cancelled = false;

  fetchUser(userId).then((data) => {
    if (!cancelled) setUser(data);
  });

  return () => {
    cancelled = true;
  };
}, [userId]);
```

That's better. But honestly, in 2026, you almost certainly shouldn't be writing raw data fetching in `useEffect` at all. Use a library like **TanStack Query** instead. It handles caching, deduplication, cancellation, and background refetching out of the box, and you literally can't forget the cleanup because the library handles it for you:

./components/UserCard.tsx

```
import { useQuery } from '@tanstack/react-query';

function UserCard({ userId }: { userId: string }) {
  const { data: user, isLoading } = useQuery({
    queryKey: ['user', userId],
    queryFn: () => fetchUser(userId),
  });

  return isLoading ? <p>Loading...</p> : <div>{user?.name}</div>;
}
```

That's the same functionality with zero boilerplate and zero bugs related to stale data or missing cleanup.

1.2.3. The React GG rules

The React team published a page in the docs called "You Might Not Need an Effect". It's

required reading. Let me give you a practical summary of the rules I follow personally:

- **If you can compute it during rendering, don't store it in state** — derived values don't need `useEffect`, just compute them inline or wrap in `useMemo`.
- **If it's triggered by a user event, handle it in the event handler** — not in `useEffect`.
- **If you want to reset a component when a prop changes, use the `key` prop** — don't reach for `useEffect` to manually clear state.
- **If you need to synchronize with an external system** — a WebSocket, a third-party library, browser APIs like `ResizeObserver` — then `useEffect` is the right tool. That's what it was designed for.
- **If you're fetching data, use TanStack Query or SWR** — don't reinvent this with a raw `useEffect`.

The overarching theme is this: `useEffect` is for synchronization with external systems, not for managing application logic. Keep that in mind and most of the wrong uses disappear naturally.

If you want to go deeper on this topic, the official React docs have an excellent section: **You Might Not Need an Effect**. It covers even more cases with great examples. Worth bookmarking.

Chapter 2. A Glimpse of What's Inside

The following are brief excerpts from two of the most impactful chapters in the book — on the React Compiler and Concurrent UI. They are included here to give you a taste of the depth and practical focus you will find throughout.

2.1. Stop Memoizing Yourself *(excerpt)*

I used to add `useMemo` and `useCallback` to almost everything. If a function was defined inside a component, `useCallback` it. If a value was computed from props or state, `useMemo` it. I thought I was being a good developer, writing performant code, being responsible.

Then I learned two things that changed how I think about this. First: premature memoization has a real cost. Every `useMemo` and `useCallback` allocates memory for the cached value and runs a comparison on every render to decide whether to recompute. If you're memoizing cheap operations, you're adding overhead to avoid overhead that didn't exist. Second: the React Compiler does this automatically now. Better than you can. And it only memoizes what actually needs memoizing.

2.1.1. Understanding the React Compiler (React 19+)

The React Compiler — previously known as React Forget — is a build-time compiler that automatically applies memoization to your components and hooks. You write regular React code and the compiler figures out what to cache and what not to. It was released as stable in the React 19 ecosystem and is available as a Babel plugin.

Setting it up in a Vite project takes less than five minutes:

```
npm install -D babel-plugin-react-compiler eslint-plugin-react-compiler
```

./vite.config.ts

```
import { defineConfig } from 'vite';
import react from '@vitejs/plugin-react';

export default defineConfig({
  plugins: [
    react({
      babel: {
        plugins: [['babel-plugin-react-compiler', {}]],
      },
```

```
    }),
  ],
});
```

That's all. Enable the plugin and the compiler starts analyzing your code at build time. Here is what this means in practice:

```
// Before the compiler: defensive memoization everywhere
function ProductFilter({ products, category }: ProductFilterProps) {
  const filteredProducts = useMemo(
    () => products.filter((p) => p.category === category),
    [products, category]
  );

  const handleSelect = useCallback(
    (id: string) => { onSelect(id); },
    [onSelect]
  );

  const sortedProducts = useMemo(
    () => [...filteredProducts].sort((a, b) => a.name.localeCompare(b.name)),
    [filteredProducts]
  );

  return (
    <ul>
      {sortedProducts.map((p) => (
        <li key={p.id} onClick={() => handleSelect(p.id)}>{p.name}</li>
      ))}
    </ul>
  );
}
```

```
// After the compiler: just write normal code
function ProductFilter({ products, category, onSelect }: ProductFilterProps) {
  const filteredProducts = products.filter((p) => p.category === category);
  const sortedProducts = [...filteredProducts].sort((a, b) =>
    a.name.localeCompare(b.name)
  );

  return (
    <ul>
      {sortedProducts.map((p) => (
        <li key={p.id} onClick={() => onSelect(p.id)}>{p.name}</li>
      ))}
    </ul>
  );
}
```

The second version is cleaner, easier to read, and produces equivalent or better performance. The full chapter in the book covers when manual memoization still

makes sense, what to remove from your existing codebase, and a step-by-step migration path for legacy projects.

Continue reading in the full book...

2.2. Concurrent UI and Optimistic Updates *(excerpt)*

I hate sluggish UIs. Every time I click a button and the whole page freezes for half a second, or a spinner pops up and I just have to stand there and wait, I feel like something has gone wrong. And for a long time, I accepted this as the cost of doing async work in React.

React introduced two hooks that completely changed how I think about user interactions: `useTransition` and `useOptimistic`. The first one lets you tell React which state updates are urgent and which ones can wait, so your UI never feels janky again. The second one lets you show the user the **result** of their action immediately, before the server even responds.

2.2.1. useTransition

Let me start with a problem: filtering through thousands of records in the browser. Every time the user typed into a search input, the UI would visibly stutter. The input itself lagged because React was busy re-rendering the massive filtered list on every keystroke.

Here is what the problematic version looks like:

```
export default function SearchPage() {
  const [query, setQuery] = useState('');
  const [filteredItems, setFilteredItems] = useState(items);

  const handleSearch = (value: string) => {
    setQuery(value);
    // This blocks the UI — runs synchronously on every keystroke
    setFilteredItems(items.filter(item =>
      item.name.toLowerCase().includes(value.toLowerCase())
    ));
  };

  return (
    <>
      <input value={query} onChange={e => handleSearch(e.target.value)} />
      <ItemList items={filteredItems} />
```

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

Now here is the same thing with `useTransition`:

./components/SearchPage.tsx

```
import { useState, useTransition } from 'react';

export default function SearchPage() {
  const [query, setQuery] = useState('');
  const [filteredItems, setFilteredItems] = useState(items);
  const [isPending, startTransition] = useTransition();

  const handleSearch = (value: string) => {
    setQuery(value); // urgent: update the input immediately

    startTransition(() => {
      // non-urgent: React can defer this if needed
      setFilteredItems(items.filter(item =>
        item.name.toLowerCase().includes(value.toLowerCase())
      ));
    });
  };

  return (
    <>
      <input value={query} onChange={e => handleSearch(e.target.value)} />
      {isPending && <span className="searching-indicator">Searching...</span>}
      <ItemList items={filteredItems} />
    </>
  );
}
```

The input is now always responsive. React prioritizes the `setQuery` update, so typing is never blocked. The full chapter also covers `useOptimistic` — which lets your UI reflect the result of an action ***before*** the server responds — and how to combine both hooks with `useActionState` for form submissions that feel instant.

Continue reading in the full book...

Chapter 3. What's Inside the Full Book

This preview covers two complete chapters and two excerpts. Here is everything waiting for you in the full edition:

3.1. Part 1 — Building a Strong Foundation

Chapter 1: Designing a Scalable React Architecture ***(included in this preview)***
The Fractal Pattern for project structure. Clear rules for where every component, hook, and utility belongs, no matter how large your app grows.

Chapter 2: The Composition Mindset
How to stop building prop-heavy components and start building composable ones. The "Donut" pattern for clean, flexible component APIs your whole team will thank you for.

Chapter 3: Mastering React without Overusing useEffect ***(included in this preview)***
The most misused hook in React. Learn when you genuinely need it, and how to replace the patterns you've been using it for with simpler, faster alternatives.

3.2. Part 2 — Managing State and Data Intelligently

Chapter 4: React Context Done Right
When Context is the right tool and when it absolutely is not. How to structure it for performance, and when to reach for Zustand instead.

Chapter 5: The Art of Custom Hooks
How to extract and reuse stateful logic cleanly. A real-world walkthrough building a `useModal` hook that composes, scales, and eliminates boilerplate across your entire app.

Chapter 6: Data Consistency — Enums, Constants, and `as const`
Why magic strings are a silent source of bugs, and how the `as const` assertion gives you type-safe, readonly configurations that TypeScript actually understands.

3.3. Part 3 — Writing Smarter React Code

Chapter 7: Functional Programming in React
Pure functions, immutability, and function composition — not as abstract theory but as

practical tools for writing React that's predictable, testable, and easy to debug.

Chapter 8: Modern Forms and Validation
React Hook Form + Zod is the industry standard in 2026. This chapter shows you why, with practical patterns for dynamic fields, type-safe validation, and zero unnecessary re-renders.

Chapter 9: The State of Styling in 2026
Why Tailwind has won, the `cn` utility for conditional classes, and an honest look at when CSS-in-JS still makes sense.

Chapter 10: Keep Your JSX Clean
Practical rules for keeping render methods readable. The `&&` operator gotcha that renders `0` on your screen, and how to handle conditional rendering cleanly.

3.4. Part 4 — Performance and Modern React Features

Chapter 11: Efficient Rendering with Virtualized Lists
How to render 100,000 items without breaking a sweat. TanStack Virtual in practice, including variable-height rows and memoized list items.

Chapter 12: Stop Memoizing Yourself *(excerpt included)*
The React Compiler handles memoization automatically. This chapter shows you what to remove from your codebase, what to keep, and how to migrate legacy performance code.

Chapter 13: React Suspense and Streaming UIs
How Suspense actually works under the hood, code splitting with `React.lazy`, data fetching with `useSuspenseQuery`, and the new `use()` hook in React 19.

Chapter 14: Concurrent UI and Optimistic Updates *(excerpt included)*
`useTransition` and `useOptimistic` — the two hooks that make your UIs feel genuinely instant. Real patterns for search interfaces, optimistic todo lists, and form submissions.

Chapter 15: New React Primitives — Activity and ViewTransition
`<Activity>` for preserving state in hidden UI, and `<ViewTransition>` for native-app-like animations. Build smooth tabbed interfaces without an animation library.

3.5. Part 5 — Making React Development Easier

Chapter 16: Leveraging AI in Your React Workflow
How to use Cursor, Claude, and v0.dev to write better code faster — without outsourcing your understanding. Cursor Rules, AI pair programming, and knowing when to trust the output.

Chapter 17: Writing Tests with Vitest and React Testing Library
What to test, what to skip, and why. A pragmatic approach: unit test your utilities, integration test your critical flows, and stop chasing 100% coverage.

Get the full book at: **https://gabrielenache.gumroad.com/l/best-ways-to-improve-your-react-project**