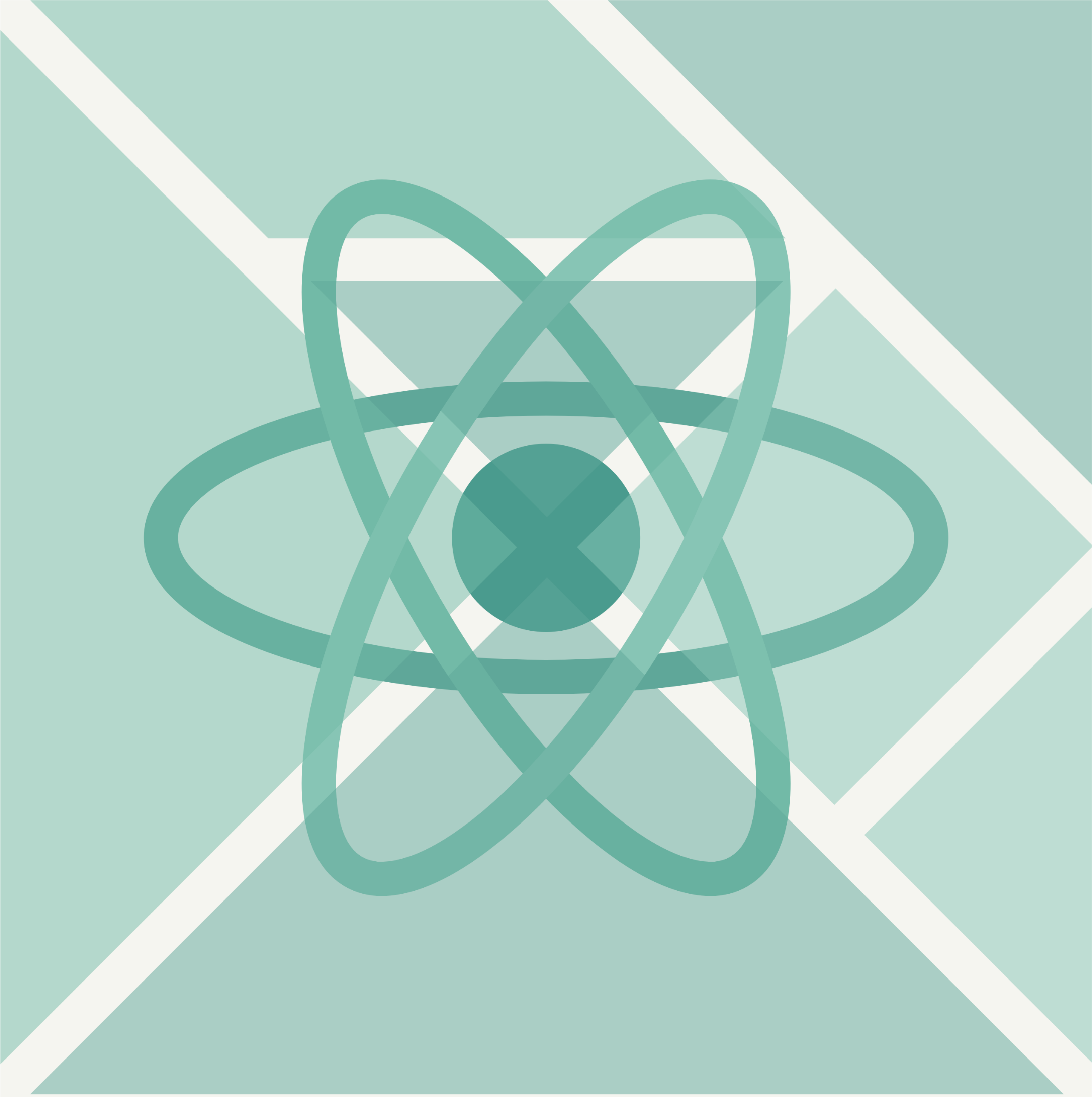

An Elm Primer for React Developers

The Best Way to Learn Real Functional Programming

An Elm Primer for React Developers

The Best Way to Learn Real Functional Programming

Christian Ekrem

This book is available at https://leanpub.com/elm-for-react-devs

This version was published on 2025-10-20

Contents

Introduction

If you're a React developer, chances are you've felt the framework's power and complexity. Hooks are incredibly flexible, TypeScript catches many errors—yet runtime surprises still slip through. `useEffect` dependencies that never feel quite right. Performance bugs requiring strategic `React.memo` and `useCallback`. These aren't signs you're doing React wrong; they're the inherent trade-offs of React's flexibility.

Some of my most-read posts are the deep-dive React articles—especially on `React.memo` and reconciliation. A few are used in React courses and have even been translated to Korean. This popularity reflects a truth: once you step beyond the basics, React's complexity increases dramatically and the good can quickly become challenging faster than you expect.

Elm takes a very different approach. It's not just another UI library; it's a language with a single way of building applications, designed from the ground up for reliability and clarity. With Elm, you trade flexibility for focus. There's exactly one architecture (The Elm Architecture, or TEA), exactly one way to represent side effects, and a type system that works relentlessly to prevent runtime errors. The result is a developer experience that feels—at least at first—strangely quiet. You don't spend hours debugging why your component didn't re-render or why state mutated unexpectedly. Instead, you spend time *modeling* your problem domain.

For React developers, the first encounter with Elm can feel almost too simple. There are no hooks to juggle, no contexts to configure, no class vs. function components. Just a `Model`, an `Update` function, and a `View`. And yet, that simplicity scales to codebases with hundreds of thousands of lines.

So why consider Elm if you already know React?

- **To catch more errors at compile time**. Many React/TypeScript errors show up only in the browser. Elm's compiler is famously strict: if it compiles, it's very likely to work.
- **To simplify mental overhead**. Fewer concepts mean fewer ways to get things wrong. Instead of choosing between a dozen state management solutions, Elm gives you one—TEA.
- **Because it's actually fun**. Many developers describe Elm as enjoyable to work with. The compiler points you toward fixes, the language encourages clear modeling, and your app tends to just work once it compiles.

But there's a deeper reason to learn Elm, even if you never use it professionally: **it's the fastest way to truly learn functional programming**. Not watered-down FP, not FP-flavored JavaScript patterns, but real, uncompromising functional programming—the kind that changes how you think about code in any language. Haskell has too many language features and quirks, making it notoriously hard to get started with something practical. Elm, by contrast, is remarkably small and focused—the entire language fits in your head. And crucially, you're working in a domain you already know: building web UIs. This combination of strict FP discipline and immediate practical application makes Elm uniquely effective as a learning tool. If you've ever wanted to "level up" as a developer by learning functional programming, Elm is your shortest path. Especially if you're already familiar with React.

This book is not about convincing you to abandon React. Instead, it's a guided tour for React developers: what Elm looks like, how it compares, and how you might start using Elm—even if only as a single widget inside a React app. By the end, you should have a feel for whether Elm can play a role in your work.

If React has been your main frontend toolkit, think of Elm as a chance to see what frontend development feels like with different trade-offs. It may not be for every project—but once you've experienced the calm of Elm's compiler, you may find yourself wishing more of your codebases worked this way.

And, quite frankly, learning Elm will be worthwhile whether you end up using it professionally or not.

To give you a taste of what I mean, let me show you the kind of guarantees Elm provides. Throughout this book, you'll see what I call "Elm Hooks" in each chapter—compelling previews of what makes Elm delightful for that particular topic. React has hooks for state management, but Elm has hooks that keep you hooked on the language itself.

Let's try one right away, showcasing two of Elm's main selling points: The friendly compiler and the strict type system (or, good cop / bad cop, if you prefer).

Elm Hook

Given the following Elm code

```elm
type Animal = Cat | Dog | Penguin


animalToString : Animal -> String
animalToString animal =
    case animal of
        Dog ->
            "dog"

        Cat ->
            "cat"

        -- Ooops, forgot about that Penguin!
```

The compiler answers the following:

```
-- MISSING PATTERNS --------------- /Users/cekrem/code/animals/src/Main.elm
This `case` does not have branches for all possibilities:

306|>    case animal of
307|>        Dog ->
308|>            "dog"
309|>
310|>        Cat ->
311|>            "cat"

Missing possibilities include:

    Penguin

I would have to crash if I saw one of those. Add branches for them!

Hint: If you want to write the code for each branch later, use `Debug.todo` as a
placeholder. Read <https://elm-lang.org/0.19.1/missing-patterns> for more
guidance on this workflow.
```

Sure beats all those `ensureNever` helpers you've written in TypeScript, huh?

About the Author

Christian Ekrem is a Staff Engineer with over 12 years of experience building production systems at scale. He works daily in a production Elm codebase with over 125,000 lines of code at Lovdata, Norway's leading legal information provider. This real-world experience—not just toy projects—informs every comparison and recommendation in this book.

His journey spans multiple platforms and languages: six years at Vipps MobilePay (Norway's leading mobile payment platform) working across backend development in Go and React, Android development in Kotlin, and eventually becoming the first engineer to work across both simultaneously. This diversity gives him a unique perspective on what makes code maintainable and genuinely better.

Christian's background extends beyond tech—he's taught at a special-needs high school and worked as a film director. This varied experience taught him how to explain complex concepts clearly and meet people where they are, which is exactly what this book aims to do for React developers learning Elm.

Why this book exists: Christian believes Elm is the fastest way for developers to truly learn functional programming. Whether you adopt Elm professionally or not, learning it will make you a better developer in any language. This book is his way of sharing that insight with React developers ready to level up.

You can read more of his writing on technology and functional programming at cekrem.github.io[1].

[1] https://cekrem.github.io

Part I: From React to Elm: Getting Started

Chapter 1: Elm: Delightful Constraints

Remember that `Penguin` example from the introduction? That kind of exhaustive checking isn't just for pattern matching; it's actually how Elm approaches everything. Let me show you what this means for everyday development.

You refactor a type. You rename a field from `status` to `orderStatus`. You update the code, run your tests, and ship it. A week later, production errors start rolling in: `Cannot read property 'status' of undefined`.

You missed one usage. It was buried in an error handler that only runs when a specific edge case triggers. TypeScript didn't catch it because that file had an `any` type. Your tests didn't catch it because you hadn't written a test for that exact scenario. ESLint was silent because the code was syntactically fine.

This isn't a story about carelessness. You're a good developer. You ran the tests. You checked your work. But in a codebase of any size, it's impossible to keep every usage of every field in your head. You rely on tools to catch what you forget.

And sometimes, your tools don't catch everything.

React Recommends, Elm Requires and Enables

Here's something I've noticed working with both React and Elm: they're heading in the same direction, but taking different paths to get there.

Look at how React has evolved:

- Hooks moved us toward functional components and immutable state
- Redux brought predictable state management to the mainstream
- TypeScript went from optional to essential for serious projects
- Server Components push side effects to the server

Each change pushes React toward functional programming principles. Immutability, pure functions, explicit state management—these are all things the React community now considers best practices.

A senior React developer I know put it this way: "Good React code in 2025 looks suspiciously like Elm code from 2015."

But here's the key difference: **React recommends functional programming. Elm requires it.** And in doing so, it frees you from worrying about entire classes of bugs and moreover *enables* you to focus on solving actual problems instead of debugging state mutations.

In React, you can still mutate variables. You can mix paradigms. You can create runtime errors. The language allows it. The community discourages it, but JavaScript doesn't stop you.

In Elm, it's simply impossible. Not hard. Not discouraged. Impossible. The language won't compile if you try to mutate data.

This isn't about React being bad—React's flexibility is a feature. But that flexibility comes with a cost: you have to maintain the discipline yourself.

When Constraints Give Freedom

It sounds paradoxical: how can stricter constraints give you more freedom? But consider a common debugging scenario in JavaScript:

```
const user = { name: "Ada", age: 29 };
someFunction(user);
console.log(user.name); // What's the name now?
```

You can't know without reading `someFunction`. Maybe it mutates the user. Maybe it doesn't. Maybe it mutates it conditionally? You have to trace through the code to be sure. Even the most elaborate typings can't give complete compile-time guarantees that an object will not be mutated; it's simply not possible. There are ways to prevent this in *runtime* with recursive variations of `Object.freeze` and the like (but be mindful that, ironically, even the `Object.freeze` method can be mutated!). And again, imposing such restrictions in TypeScript require discipline.

This isn't just a theoretical problem. I've spent hours debugging issues where data was mutated in unexpected places. A function I thought was safe was actually changing my state. The bug only appeared in specific conditions, which is why the tests didn't catch it.

Now look at the Elm equivalent:

```
user = { name = "Ada", age = 29 }

-- This doesn't compile, whether done inline like here or in a function you pass it \
to:
user.name = "Grace"  -- ERROR: Elm doesn't have variable mutation

-- The right way:
updatedUser = { user | name = "Grace" }  -- Creates a new record
```

The compiler makes mutation impossible. When you pass `user` to a function, you know it comes back unchanged. Not because you trust the function author. Not because code review caught it. But because the language doesn't allow anything else. If you've ever used Rust, you know that the distinction between a variable and a mutable variable is an important one. In Elm, they're all immutable.

This constraint eliminates entire categories of bugs. You stop wondering "who changed this value?" because nothing can. The constraint gives you freedom from a whole class of debugging sessions.

The Debugging Clarity

This immutability guarantee changes how you reason about code. In React, when state is wrong, you have to trace backward: Where was this set? What changed it? Did something mutate it accidentally?

In Elm, when state is wrong, you look at your `update` function. That's it. That's the only place state changes. If the state is wrong, the logic in `update` is wrong. No hidden mutations. No stale closures. No wondering if some other component changed something.

At our production app (a 120k+ lines Elm codebase), we've had entire months with zero runtime exceptions in our Elm code. Not because we're better developers than when we wrote TypeScript. But because the compiler catches those errors before the code runs.

Elm Hook

You know that moment when you refactor a type but forget to update one place that uses it? In Elm, you literally cannot compile until you fix every single usage. The compiler won't let you ship the incomplete refactor. It's impossible to "forget one spot."

Refactoring with Confidence

This reliability becomes especially valuable when making large-scale changes.

Let's say you need to add a new state to your application—maybe a `Paused` state for a game, or a `Refreshing` state for data loading. In React, you'd:

1. Add `'paused'` to your TypeScript union type
2. Search the codebase for places that check the state
3. Update each one, hoping you found them all
4. Test manually, hoping you caught the edge cases
5. Ship and monitor for bugs

In Elm, you:

1. Add `Paused` to your union type

2. Try to compile
3. The compiler lists every place that needs updating
4. Fix each one
5. When it compiles, you're done

I refactored a complex state machine recently—47 places needed updates. The compiler found all 47. I fixed them one by one. When the code compiled, I deployed it. No bugs. No forgotten edge cases. The compiler had verified completeness.

That's not a guarantee of correctness—I can still have logic bugs. But it's a guarantee that every code path handles every state. No gaps.

The Architectural Discipline

If you've read about Clean Architecture or SOLID principles, you know they're good ideas. Single Responsibility, Dependency Inversion, separation of concerns—these patterns lead to maintainable code.

But they're also discipline. You have to remember to follow them. Code review has to catch violations. It's easy to cut corners when you're rushing.

The Elm Architecture enforces these patterns by default. Again: not as guidelines—as requirements:

- **Single Responsibility**: The compiler forces you to separate View, Update, and Model
- **Pure Functions**: Mutation is impossible, so all your functions are automatically pure
- **Explicit Effects**: Side effects must go through Commands; you can't just call an API in your update logic
- **Exhaustive Handling**: Pattern matching forces you to handle every case

Where other languages offer SOLID as "best practices" you should follow if you're disciplined, they're a mandatory part of Elm. The compiler is your relentless architecture mentor.

What This Costs You

Let's be direct: Elm's strictness has real costs.

The ecosystem is smaller. React has thousands of libraries. Elm has hundreds. You'll find yourself writing more from scratch.

The learning curve is steeper. Functional programming is different if you're coming from JavaScript. Pattern matching, union types, immutability—these take time to internalize.

Your team needs to learn. Hiring is harder. Onboarding takes longer. Not every developer wants to learn a niche language.

You lose flexibility. Sometimes you just want to mutate a value and move on. Elm won't let you. You have to do it the "right" way, even when the shortcut would probably work.

For some projects, these costs aren't worth it. If you're prototyping, exploring, or building something simple, React's flexibility is valuable. You want to move fast, not satisfy a strict compiler. Doubly so if React has become second nature: when going fast, using familiar tools is a win in and by itself! And honestly, for most projects today, React is still the pragmatic choice—and that's perfectly fine.

But for other projects—production applications where bugs are expensive, complex state machines, financial tools, healthcare systems—Elm's guarantees are worth the upfront cost.

And, to be completely honest: At this point I personally prefer Elm even for the occasional whimsical side-project that won't hurt a fly no matter how hard it crashes. As you'll hopefully discover for yourself before too long: Elm is kind of addictive!

What Elm Teaches You

Here's what I've realized: Elm's value isn't just in using it. It's in what it teaches you.

Learning Elm changes how you think about code. Mutable state starts looking suspicious. You design types that actually prevent bugs instead of just documenting intent. You write better React because these patterns become second nature.

The React community is moving toward Elm's ideas—hooks, immutability, type safety, isolated effects. These aren't Elm-specific. They're functional programming principles that apply everywhere. And Elm is uniquely effective at teaching them because you're building real UIs, not studying academic theory.

This is Elm's hidden superpower: **it's the fastest way to truly learn functional programming**. Not because it teaches you monads and functors (Elm never even mentions them), but because it makes functional programming impossible to avoid. Try to mutate a variable? Compiler says no. Try to ignore a case? Compiler says no. Try to hide side effects? Nope. You can't cheat your way around it, so you learn to think functionally.

Compare this to learning FP through Haskell or OCaml. Those languages are powerful, but they're also large and complex. Haskell has lazy evaluation, type classes, monad transformers, dozens of language extensions. By the time you understand enough to build something useful, months have passed.

Elm's entire language fits in your head in a weekend. No classes. No inheritance. No async/await, no promises, no null, no undefined. Just functions, types, and one architecture pattern. The smallness isn't a limitation—it's what makes learning fast.

And crucially, you're building in a domain you already know. If you're coming from React, you understand components, events, state updates. Elm uses different mechanics, but the same concepts. You're not learning abstract category theory—you're building the same UIs you built yesterday, just with different guarantees.

As I'll argue again and again throughout this book: This matters even if you never use Elm professionally. The functional thinking you develop transfers directly to *any* language. Modeling with types, making illegal states unrepresentable, treating data immutably—these patterns make you write better TypeScript, better Python, better anything. And if you eventually need to learn Haskell, F#, or OCaml, you'll have a solid head start since you already understand the core concepts.

In the next chapter, we'll stop talking philosophy and look at code. We'll build the same application in both React and Elm, side by side. You'll see exactly what these guarantees look like in practice—the syntax, the patterns, the developer experience.

Are you ready?