Contents

02 — React Thinking Guide for Flagline

Audience: Backend developer with 5+ years PHP/Laravel experience, transitioning to React / Next.js / TypeScript. **Stack:** Next.js 14+ (App Router), React 19, TypeScript, Prisma (PostgreSQL), Redis. **Purpose:** This is not a code reference. It is a thinking guide. It explains the mental models, decision frameworks, and reasoning process that will help you write React code yourself. If you want API surfaces and implementation patterns, see 03 — Next.js Reference. If you want database schemas and queries, see 05 — Database & Prisma Reference.

Table of Contents

---

1. The Mental Model Shift from Server-Rendered PHP to React

Coming from Laravel / Blade / Livewire

When you build a Laravel app, here is your mental model: a request arrives, a controller queries the database, that controller passes variables to a Blade template, Blade interpolates those variables into HTML, and the browser receives a finished document. Every interaction that requires new data means a new HTTP request. The server is always the authority. The browser is a display terminal.

Livewire softens this a bit – it lets you wire PHP properties to DOM elements so that interactions trigger partial re-renders over AJAX – but the mental model is still "the server owns the state, the browser shows it."

React inverts this relationship in a way that matters deeply once you internalize it.

UI as a Function of State

The single most important sentence in this entire document is this: **In React, your UI is a function of state.** Not a template with holes punched in it. Not an HTML document that gets patched. A function. In the mathematical sense: given the same state, you will always get the same UI.

In Blade, you write `<span>{{ $user->name }}</span>`. That is template interpolation – you are telling the engine "put this value here." In React, you write a function that *returns* what the UI should look like given the current state. When state changes, React calls your function again and figures out what changed in the output.

Think of it this way: Blade is a fill-in-the-blanks worksheet. React is a calculator that reruns the formula every time an input changes.

This is not just a syntactic difference. It changes how you think about everything. In Laravel, you think: "When the user clicks this button, I need to update the database, then redirect to a page that will query the new data and render it." In React, you think: "When the user clicks this button, I need to update the state. The UI will take care of itself."

Re-rendering: What Actually Happens

"Re-rendering" is the concept that trips up most backend developers because it sounds expensive. It is not what you think.

When React "re-renders" a component, it does not touch the DOM. It calls your function again to get the new JSX output, then *diffs* that output against the previous output. Only the actual differences get applied to the DOM. This diffing algorithm (called reconciliation) is extremely fast for typical UI trees.

Think of it like this: imagine if Blade re-ran the entire template every time a variable changed, but instead of sending the full HTML to the browser, it computed a minimal set of DOM mutations. That is what React does on the client, but it happens in milliseconds because the diffing happens in memory, not over the network.

The consequence is that you should not be afraid of re-renders. A component re-rendering is just a function being called. The question is never "how do I prevent re-renders?" but rather "is the work being done during this re-render expensive?" For most components, it is not.

The Component Tree as a Call Stack

Here is an analogy that should click immediately for you: the React component tree is like a call stack.

When React renders your page, it starts at the root component and calls it. That function returns JSX that references child components. React calls those. They return JSX referencing their children. React calls those too. All the way down.

In Laravel, you might think of this like nested Blade includes: `@include('partials.sidebar')` inside a layout that `@includes` a header. But the crucial difference is that in Blade, the data flows implicitly through shared variables or `@props`. In React, the data flow is explicit and always downward.

A parent component passes data to a child through “props” – think of them as function arguments. The child cannot reach up and grab data from its parent. The child cannot modify the parent’s state. Data flows down. This is the most important architectural constraint in React, and it is a feature, not a limitation.

Events flow in the opposite direction. A child that needs to tell its parent something does so by calling a callback function that the parent passed down as a prop. The child says “something happened,” the parent decides what to do about it.

In Laravel terms: imagine every `@include` received its data exclusively through a parameter array, and if the partial needed to trigger a controller action, it had to call a closure that was passed in through that same parameter array. No accessing `$request` from inside a partial. No service container injection. Just arguments in, callbacks out.

This constraint makes React applications dramatically easier to reason about as they grow. When something goes wrong, you can trace the data flow by following the props down and the callbacks up. There is no spooky action at a distance.

From Page Loads to State Changes

In Laravel, navigation means a new request. The server generates a new page. The browser replaces its entire document. Session and cookies persist across requests, but the in-memory state of the frontend is destroyed on every navigation.

In a React single-page application, navigation means updating a client-side router’s state. The URL changes, a different component tree renders, but the JavaScript runtime stays alive. In-memory state can persist across navigations. This is why you can have things like a flag toggle that updates optimistically and then syncs in the background – the UI never “resets” because there was no page load.

Next.js with the App Router adds nuance to this, because server components *do* render on the server (more on this in Section 2), but client-side navigations between routes still behave as described above. The mental model is: initial page load is server-rendered (like Laravel), subsequent navigations are client-side transitions (unlike Laravel).

Server Components Bring Back Some of the PHP Mental Model

Here is the twist that makes Next.js particularly approachable for you: **React Server Components are conceptually very close to Blade templates.**

A server component runs on the server, has direct access to the database, renders to HTML, and sends that HTML to the client. The client never downloads the JavaScript for that component. It is, in essence, a server-rendered template.

This means you can think about your page in two layers. The outer structure – layouts, data-heavy sections, content that does not need interactivity – is server-rendered, just like Laravel. The interactive islands within that structure – toggles, dropdowns, real-time feeds, form inputs – are client components that hydrate on the browser.

If you squint, this is not unlike a Blade layout with Livewire components embedded in it. The layout is server-rendered. The Livewire components are interactive islands. The difference is that

React gives you a unified component model across both layers, whereas Laravel has two separate paradigms (Blade vs Livewire) with an awkward boundary between them.

---

2. Server Components vs Client Components

The Thinking Framework, Not a Rule List

You will find plenty of “use server components when X, use client components when Y” checklists online. Those are fine as references, but they do not teach you how to think. Here is the thinking framework.

For every piece of UI you are about to build, ask yourself one question: **Does this UI need to react to something the user does?**

“React to something” means: does it need to respond to clicks, track hover state, manage form input, animate based on user scroll, update based on a timer, or maintain any in-memory state between renders?

If the answer is no, it is a server component. It is like a Blade partial. It fetches data, renders HTML, and is done. The flag list page header that shows the project name and environment? Server component. The pricing table on your marketing site? Server component. The sidebar navigation links? Probably a server component (active-state highlighting can be handled differently, as we will discuss).

If the answer is yes, it is a client component. The flag toggle switch that the user clicks? Client component. The search input that filters the flag list as you type? Client component. The dropdown menu that opens on click? Client component.

The Serialization Boundary

Here is the concept that matters most and gets explained least: the **serialization boundary** between server and client components.

When a server component renders a client component as one of its children, React needs to send data from the server world to the client world. That data must be serializable – it must survive being turned into JSON, sent over the wire, and reconstituted on the other side.

This means you can pass strings, numbers, booleans, arrays, plain objects, and dates as props from a server component to a client component. You cannot pass functions, class instances, database connections, or anything that only makes sense in a Node.js runtime.

Think of it like this: in Laravel, when you pass data from a controller to a Blade view, you can pass anything PHP can handle because it all runs in the same process. But if you were passing data from a PHP backend to a JavaScript frontend via an API response, you would have to serialize it to JSON. The server-to-client component boundary is that same JSON serialization boundary, except it happens automatically within your component tree instead of at an explicit API endpoint.

This has a practical consequence for how you design your components. If you have a server component that queries a list of flags from the database and you need a client component to render the interactive flag table, the server component fetches the flags, serializes them as props, and the client component receives plain data. The client component never talks to the database.

Pushing Client Boundaries Down

The most important architectural principle for server/client composition is this: **push the client boundary as far down the component tree as possible.**

Here is what this means in practice. Imagine you are building the flag detail page. It shows the flag name, description, creation date (all static, read-only data), and a toggle switch (interactive). The naive approach is to make the entire page a client component because it "has interactive parts." The better approach is to make the page a server component that fetches the flag data and renders the static parts directly, then embeds a small `FlagToggle` client component that receives only the props it needs: the current state and a callback.

Why does this matter? Three reasons.

First, server components send zero JavaScript to the client. The flag name, description, and metadata are rendered as HTML on the server. The browser does not need to download, parse, and execute JavaScript to display them. It is only the toggle switch – a tiny component – that adds to your JavaScript bundle.

Second, server components can directly access your database, your Redis cache, your environment variables. No API layer needed. In Flagline, a server component on the flag list page can call Prisma directly: `const flags = await prisma.flag.findMany(...)`. This is exactly like a Laravel controller querying Eloquent and passing the result to a Blade view.

Third, it keeps your client components small, focused, and easy to test. A `FlagToggle` that receives `flagId`, `isEnabled`, and `onToggle` as props is trivial to reason about. A `FlagDetailPage` client component that fetches its own data, manages loading states, handles errors, and also renders the toggle is a mess.

The mental model is: your server components are the scaffolding. They fetch data, define layout, and render everything that does not need interactivity. Client components are the interactive joints embedded within that scaffolding. Think small, focused, interactive widgets – not large page-level client components.

The `"use client"` Directive as a Fence

When you put `"use client"` at the top of a file, you are drawing a fence. Everything in that file and everything it imports is now in client territory. This fence is one-way: client components cannot import server components.

However – and this is the subtlety that catches people – a client component can *render* server components if they are passed as children or props. The server component is rendered on the server and the result (not the code) is passed through. This is how you compose server and client code without turning everything into a client component.

The practical consequence: be thoughtful about where you place `"use client"`. If you put it on a layout component high in the tree, everything inside that layout becomes client code. If you put it on a small leaf component, only that leaf and its imports are client code.

In Flagline, the dashboard layout (sidebar, topbar, content area) should be a server component. The sidebar might have a collapsible section – only that collapsible trigger is a client component.

The topbar might have a user menu dropdown – only that dropdown is a client component. The rest is static, server-rendered HTML.

---

3. State Management Thinking

Coming from Laravel

In Laravel, "state" lives in a few places: the database (persistent), the session (per-user, across requests), the request itself (per-request), and maybe Redis or cache (transient). You never agonize over where to put state because the answer is almost always "the database" with "the session" as the runner-up.

In React, state can live in more places, and choosing the right place is one of the most important architectural decisions you make. Get it wrong and you end up with state synchronization bugs, unnecessary complexity, or UI that feels broken because the URL does not reflect what the user sees.

The Decision Framework: Where Should This State Live?

When you need to track a piece of state, ask these questions in order:

1. Should this state survive a page refresh and be shareable via URL?

If yes, put it in the URL (query parameters or path segments). This is the most overlooked state location by React beginners, and it is the most natural one for a backend developer.

In Flagline, the selected project and environment are URL state: `/dashboard/my-app/production/flags`. If the user copies this URL and sends it to a teammate, the teammate should land on the same project, same environment, same flag list. If the user refreshes, they should not lose their place. This is not "React state" at all – it is routing, and it works exactly like Laravel routes.

Search filters and pagination are also URL state. If you are on page 3 of the audit log with a filter for "toggle" events, that should be `?page=3&event=toggle` in the URL. A backend developer would never store pagination in a PHP session variable – they would use query parameters. Apply the same instinct here.

2. Is this state derived entirely from other state or from props?

If yes, do not store it at all. Compute it. If you have a list of flags and a search query, the filtered list is not state – it is a computation. Storing derived state is a common React mistake that leads to synchronization bugs. You end up with the search query saying one thing and the filtered list showing another because you forgot to update both.

The rule of thumb: if you can calculate it from existing data, calculate it. In Laravel terms: you would not store a `$filteredUsers` variable in the session and try to keep it in sync with the database. You would just run the query with the filter applied. Same principle.

3. Does only one component need this state?

If yes, put it in local component state with `useState`. The open/closed state of a dropdown menu, the current value of a text input before submission, the optimistic toggle state of a flag switch – these are all local to a single component.

This is the "default" location for state. When in doubt, start here and promote the state to a higher level only when you discover a concrete need.

4. Do a parent and a few of its children need to share this state?

If yes, lift the state up to the nearest common ancestor and pass it down through props. This is the fundamental React pattern, and it works well for shallow trees. If the flag list page needs to know which flag is selected (to highlight it in the list and show its details in a panel), that state lives in the page component and gets passed to both the list and the panel as props.

5. Do many components across different parts of the tree need this state?

If yes, and the data changes infrequently, use React Context. The authenticated user's session information is the canonical example: many components (the sidebar, the topbar, permission checks on buttons, the billing page) need to know who is logged in. This data changes rarely (only on login/logout). Context is the right tool.

If the data changes frequently and is accessed by many components, Context starts to cause performance problems because every consumer re-renders on every change. This is where external state management libraries (Zustand, Jotai) earn their keep. But for Flagline, you are unlikely to need one. URL state, local state, and a user context will cover nearly everything.

State Placement for Flagline, Concretely

Here is how the framework applies to the specific state in the Flagline dashboard:

URL state (path segments and query params): - Selected project: `/dashboard/[projectSlug]/...` - Selected environment: `/dashboard/[projectSlug]/[environment]/...` - Flag detail view: `/dashboard/[projectSlug]/[environment]/flags/[flagKey]` - Search/filter: `?q=dark&status=active` - Pagination: `?page=3` - Sorting: `?sort=name&dir=asc`

Local component state (`useState`): - Flag toggle optimistic state (the switch shows "on" immediately, before the server confirms) - Dropdown open/closed - Modal open/closed - Form input values (before submission) - Text being typed in a search box (debounced before it hits the URL) - Expanded/collapsed state of rule builder sections

Context: - Authenticated user session (user object, permissions, organization membership) - Theme preference (light/dark), if you implement it - Toast/notification queue (a context that any component can push messages to)

Server state (managed by TanStack Query, not raw React state): - Flag list data (fetched from the server, cached, refreshed on mutation) - Audit log entries - Project settings - Team member list

This last category is important. "Server state" – data that lives in your database and is fetched into the client – should not be managed with `useState`. It should be managed by a data-fetching library like TanStack Query that understands caching, refetching, and staleness. More on this in Section 5.

---

4. Custom Hooks as the Abstraction Layer

Coming from Laravel

In Laravel, when you find yourself repeating logic across multiple controllers, you extract it into a service class, a trait, or an action class. A `ToggleFlagAction` might encapsulate the business logic of toggling a flag, used by both the web controller and the API controller. A `HasPermissions` trait might add authorization methods to multiple models.

React hooks are the equivalent abstraction. They let you extract reusable stateful logic out of components. The key word is "stateful" – hooks are not just shared utility functions. They can manage state, trigger side effects, subscribe to external data sources, and compose other hooks. They are behaviors you can attach to any component.

The Mental Model: Behaviors, Not Utilities

Think of a custom hook as a behavior you plug into a component. When you write `const { flags, isLoading, toggleFlag } = useFlags(projectId, environment)`, you are saying: "This component now has flag-fetching behavior. It knows how to get flags, it tracks whether they are loading, and it can toggle them."

The component does not care how `useFlags` fetches the data. Maybe it uses TanStack Query. Maybe it uses SWR. Maybe it is an in-memory mock during testing. The component just knows it has flags, a loading state, and a toggle function.

Compare this to Laravel: if you inject a `FlagService` into a controller, the controller does not care if the service talks to MySQL, Redis, or an external API. The interface is what matters. Hooks are the same idea, but they also encapsulate the reactive lifecycle – when to fetch, when to refetch, how to handle errors.

When to Extract a Hook

The instinct from Laravel is to extract early and build a service layer before you need it. In React, this often leads to premature abstraction. Here is a better heuristic.

Extract a hook when you find yourself copying stateful logic between two or more components. Not "when you might someday need it in another component" but when you actually do. The reason is that hooks bundle together state + effects + derived values into a single unit. If you extract too early, you are guessing at the right interface, and you will probably guess wrong.

Do extract a hook when the logic is complex enough that it obscures the component's rendering intent. If your component is 80% data-fetching and effect-management code and 20% JSX, the rendering intent is buried. Move the logic into a hook so the component reads clearly: "I use flags, and here is how I display them."

Do not extract a hook for pure computations. If you have a function that formats a date or filters an array, that is a utility function, not a hook. Hooks are specifically for logic that involves React's state or effect primitives.

Hooks for Flagline: The Thinking Behind Each

Let me walk through the hooks you would likely build for Flagline and, more importantly, the reasoning process that leads to each one.

`useFlags(projectId, environment)` – This exists because every page within an environment needs the flag list, and the flag list involves fetching, caching, refetching on mutation, and real-time updates via SSE. That is a lot of behavior to repeat. The hook wraps a TanStack Query call, sets up the SSE subscription to invalidate the cache when another user modifies a flag, and exposes the flags, loading state, and error state. Any component that needs the flag list just calls this hook and gets reactive data.

`useToggleFlag()` – You might wonder why this is separate from `useFlags`. The reason is separation of concerns. `useFlags` is about reading data. `useToggleFlag` is about writing data. It wraps a mutation (TanStack Query's `useMutation`) that does optimistic updating: it flips the flag in the local cache immediately, sends the request to the server, and rolls back if the server returns an error. Having it as a separate hook means any component with a toggle button can use it, whether it is in the flag list, the flag detail view, or a bulk-operations panel.

`useAuditLog(projectId, options)` – The audit log involves pagination, filtering, and real-time updates (new entries appear at the top). The hook manages the cursor-based pagination state, the filter parameters, and the SSE subscription that prepends new entries. Without this hook, every component that shows audit entries would need to manage all that machinery.

`useSSE(channel)` – This is a low-level hook that the others build on. It establishes an EventSource connection to the server, handles reconnection with exponential backoff, and calls a provided callback when messages arrive. `useFlags` and `useAuditLog` both use `useSSE` internally but provide different handlers for the incoming messages. Extracting SSE management into its own hook means you have one place to get reconnection logic right.

`usePermission(action, resource)` – Authorization checks happen everywhere in the dashboard: can this user toggle flags in production? Can they invite team members? Can they delete the project? The hook reads the user's role from the auth context and returns a boolean. Components use it to conditionally render buttons, disable inputs, or show upgrade prompts. This is similar to how you might use a `@can` directive in Blade, except it runs on the client side.

Notice the pattern: each hook encapsulates a specific behavior that multiple components need. The hooks compose each other (`useFlags` uses `useSSE`). And the components that use them stay clean and focused on rendering.

---

5. Data Fetching Mental Model

Coming from Laravel

In Laravel, data fetching is simple and unified. A controller method queries the database (via Eloquent or raw queries), does whatever processing is needed, and passes the result to the view. There is one pattern. It always works the same way. The data is always fresh because you just queried for it.

React has multiple data-fetching patterns, and this is genuinely confusing for newcomers. The confusion is not a sign that React is over-engineered – it is a consequence of having code that runs in two different environments (server and client) with different capabilities and constraints.

The Three Patterns and When to Use Each

Pattern 1: Server component fetching (closest to what you know)

A server component is a function that runs on the server. It can `await` a database query and use the result directly in its JSX. This is, conceptually, identical to a Laravel controller passing data to a Blade view.

This is the default pattern for any data that is needed on initial page load and does not need to update in response to client-side interactions. The flag list page's initial load, the project settings page, the team members list – all of these are server component fetches. The data flows from database to server component to HTML, and the client receives a fully rendered page.

The beauty of this pattern is that there is no loading state for the initial render. The page arrives with data already in it, just like a Laravel response. Subsequent client-side navigations to that page will show a loading state while Next.js fetches the server component's output, but the first render is instant.

Pattern 2: TanStack Query (the client-side cache layer)

Once data is on the client and needs to stay fresh, be refetched, or be mutated optimistically, you need a client-side data management layer. TanStack Query is the standard tool for this.

Think of TanStack Query as a smart cache that sits between your components and your API. It handles fetching, caching, background refetching, deduplication (if three components request the same data, only one request fires), and cache invalidation.

In Laravel terms, TanStack Query is like a request-scoped cache layer that you never had to build because every request starts fresh. In a React app, the "request" (the user's session) lasts indefinitely, so you need something to manage the lifecycle of remote data.

You use TanStack Query on the client when: data needs to refresh periodically, data needs to update after a mutation, you want optimistic updates, or multiple components need to share the same cached data.

Pattern 3: Server Actions (form submissions and mutations)

Server Actions are functions that run on the server but can be called from client components. They are the React equivalent of a form POST to a Laravel controller method.

In Laravel, you write `<form action="/flags" method="POST">`, the form submits, the controller handles it, and you redirect. Server Actions work the same way conceptually, but with a better developer experience: the function is defined in your codebase (not matched by a route), TypeScript validates the arguments, and you can call it from a client component without building an API endpoint.

Use Server Actions for any mutation that does not need optimistic updates: creating a project, updating settings, inviting a team member. For mutations that do need optimistic updates (flag toggles), you will typically call an API endpoint via TanStack Query's `useMutation` instead, because TanStack Query gives you fine-grained control over the optimistic cache update.

The Decision Framework

When you need data in a component, ask:

1. **Is this the initial render of a page or layout?** Fetch in a server component. This is the default.
2. **Does this data need to stay fresh on the client after initial load?** Use TanStack Query to manage it on the client.
3. **Do I need to mutate data and care about optimistic UI?** Use TanStack Query's `useMutation` with optimistic updates.
4. **Do I need to mutate data and a form submission UX is fine?** Use a Server Action.

In practice, many pages use a combination. The flag list page might fetch the initial flag data in a server component, pass it to a client component as initial data for TanStack Query, and then TanStack Query takes over for refetching, mutations, and real-time invalidation. This hybrid approach gives you the best of both worlds: fast initial render (server), live-updating data (client).

Caching and Staleness

In Laravel, every request queries the database. You might add Redis caching for hot paths, but the default is fresh data on every request. In React with TanStack Query, you need to think about staleness.

TanStack Query lets you configure a `staleTime` – how long data is considered fresh. If data is fresh, TanStack Query serves it from cache without refetching. If data is stale, it serves the cached version immediately (so the UI is never blank) and refetches in the background.

For Flagline, the right staleness settings depend on the data: - Flag list: low `staleTime` (maybe 30 seconds). Flags change frequently and correctness matters. But you also have SSE pushing invalidations, so stale data gets replaced quickly. - Audit log: moderate `staleTime` (a few minutes). Audit entries are immutable once written. The only thing that changes is new entries appearing, and SSE handles that. - Project settings: high `staleTime` (10+ minutes). Settings change rarely. No need to hammer the server.

Why Optimistic Updates Matter for Flag Toggles

Here is a UX argument that should resonate with your product instincts.

When a user toggles a feature flag, they expect the switch to flip instantly. If the toggle sends a request to the server and waits for the response before updating the UI, there is a 200-500ms delay where the switch feels "stuck." That delay destroys the feeling of direct manipulation. It makes the dashboard feel like a web page from 2010.

Optimistic updates solve this: the UI updates immediately (the switch flips), the request goes to the server in the background, and if the server rejects it (permissions error, conflict), the UI rolls back. The user experience goes from "click… wait… update" to "click, done."

This is the single most impactful UX pattern for a feature flag dashboard, because flag toggling is the most frequent and most latency-sensitive action. It is worth getting right.

The mental model is simple: trust the user's intent, update the UI, verify with the server. If the server disagrees, apologize and roll back. In the vast majority of cases, the server will agree, and the user will never know the request was asynchronous.

---

6. Real-Time Updates Thinking

The Problem

Flagline is a collaborative tool. Multiple team members might be looking at the same flag list simultaneously. When one person toggles a flag, the others need to see the change. Without real-time updates, they are looking at stale data and might make incorrect decisions ("I see dark mode is off, let me turn it on" – not realizing a colleague just turned it on).

SSE as a Push Channel

Server-Sent Events (SSE) is the simplest real-time technology that solves this problem. The mental model is: the client opens a persistent HTTP connection to the server. The server sends events down that connection whenever something changes. The client uses those events to update its local state.

Compare the three real-time options:

Polling means the client asks "anything new?" every N seconds. It is simple but wasteful – most responses are "no" – and there is always a delay equal to half the polling interval on average. For Flagline, polling every 5 seconds means a flag toggle might take up to 5 seconds to appear on another user's screen.

WebSockets are bidirectional, persistent connections. The client and server can both send messages at any time. They are powerful but complex: you need a WebSocket server, you lose HTTP semantics (cookies, headers, caching), and load balancers need special configuration. WebSockets are the right choice when you need bidirectional communication – chat apps, multiplayer games, collaborative editors.

SSE is unidirectional: server to client only. It runs over standard HTTP. It automatically reconnects if the connection drops. It works with existing load balancers, proxies, and CDNs. Cookies and headers work normally. It is dramatically simpler than WebSockets.

For Flagline, SSE is the right choice because the communication is one-directional: the server notifies clients about changes, clients do not need to send real-time messages back (they use normal HTTP requests for mutations). You do not need the complexity of WebSockets for a dashboard that displays updates.

How SSE Integrates with TanStack Query

This is the key architectural insight: **SSE does not replace your data-fetching layer. It augments it.**

Your components still fetch data via TanStack Query. They still have cached flag lists. SSE acts as a signal that tells TanStack Query "the data you have is stale – refetch it."

When the SSE connection receives a `flag.updated` event, your `useSSE` hook does not try to patch the local data with the event payload. Instead, it calls TanStack Query's `invalidate-`

`Queries` for the affected query key. TanStack Query then refetches the data from the server and updates every component that depends on it.

Why this approach instead of patching local state? Because it keeps the server as the source of truth. The SSE event might not contain the full flag object. Another update might have happened between the SSE event and now. By refetching, you are guaranteed to get the latest data. The brief delay (one fetch round-trip) is imperceptible to the user.

The alternative – having the SSE event carry the full updated object and patching the local cache directly – is faster but more fragile. It works until you have complex derived data, permissions-filtered views, or multiple cache keys that all need updating. For Flagline, the invalidation approach is simpler and sufficient.

Reconnection and Exponential Backoff

SSE connections will drop. The user's network will blip. Your server will deploy. The browser will throttle inactive tabs. You must handle reconnection.

The `EventSource` API has built-in reconnection, but its default behavior is to retry immediately with a fixed interval. This is fine for occasional disconnects but problematic for server outages: if your server goes down and 10,000 clients all reconnect the instant it comes back up, you have a thundering herd that might bring it right back down.

Exponential backoff solves this: the first retry is after 1 second, the second after 2 seconds, the fourth after 4, then 8, then 16, capping at some maximum (say, 60 seconds). Add random jitter so all clients do not retry at the exact same moment.

Your `useSSE` hook should implement this. When the connection is lost, the hook enters a reconnecting state. Components can check this state and show a subtle banner: "Reconnecting to live updates…" This keeps the user informed without being alarming.

When SSE vs WebSockets vs Polling: The Framework

Ask yourself these questions:

1. **Does the server need to push updates to the client?** If no, you do not need real-time at all. Fetch on demand.
2. **Does the client need to send real-time messages to the server?** If yes, you need WebSockets. If no, SSE is sufficient.
3. **How many concurrent connections will you have?** SSE connections are long-lived HTTP connections. At Flagline's scale (hundreds to low thousands of concurrent dashboard users), this is fine. At millions of connections, you would need a dedicated real-time service regardless of the protocol.
4. **Do you need to push data to clients who are not currently looking at the page?** Neither SSE nor WebSockets help here. You need push notifications (web push, mobile push).

For Flagline, the answer is SSE for the dashboard, and no real-time for the SDK (the SDK uses polling or server-side evaluation, depending on the integration pattern).

---

7. Performance Thinking

The Most Important Message: Do Not Optimize Prematurely

This needs to be said directly because backend developers coming to React often hear about `React.memo`, `useMemo`, `useCallback`, and immediately start wrapping everything in them "just in case." This is counterproductive. It makes your code harder to read, introduces potential bugs (stale closures from incorrect dependency arrays), and often makes performance worse because memoization has its own cost.

React is fast by default for the kind of UI you are building. A feature flag dashboard is not a real-time data visualization with 10,000 DOM nodes updating 60 times per second. It is forms, tables, toggles, and text. React can re-render these components hundreds of times without the user noticing.

What Actually Causes Performance Problems

There are exactly two categories of performance problems in React apps:

1. Rendering expensive subtrees unnecessarily.

If you have a component that takes 50ms to render (maybe it processes a large list, does complex formatting, or renders hundreds of DOM nodes), and it re-renders every time an unrelated piece of state changes, you have a real problem. The solution is not to memoize every component preemptively – it is to identify the expensive one (by measuring) and either memoize it specifically or restructure the state so it does not re-render unnecessarily.

The key insight is that a re-render of a cheap component (a toggle switch, a text label, a button) costs essentially nothing. Microseconds. You do not need to prevent it. You only need to worry when a re-render triggers expensive work.

2. Sending too much JavaScript to the browser.

This is the performance concern that matters most for Flagline and that backend developers often overlook. Every npm package you import, every client component you write, every utility function you use in client code – all of it gets bundled into JavaScript that the user's browser must download, parse, and execute.

A 500KB JavaScript bundle adds 1-2 seconds to your initial load on a mobile connection. A 2MB bundle is unusable on slow connections. Server components help enormously here because they send zero JavaScript to the client, but your client components still need to be lean.

For the Flagline dashboard, this means: be intentional about which npm packages you use in client components. A date-formatting library that is 2KB is fine. A rich-text editor that is 200KB should be lazy-loaded. Chart libraries should be dynamically imported so they only load on the analytics page, not on every page.

For the Flagline SDK (`@flagline/js`), bundle size is a hard product requirement. Your SDK runs in your customers' applications. Every kilobyte you add to the SDK is a kilobyte added to every app that uses Flagline. Keep the SDK tiny. This is a completely different concern from dashboard performance.

When Memoization Actually Matters

Use `React.memo` when you have measured (with React DevTools Profiler or the browser performance tab) that a specific component is re-rendering frequently and each render is expensive. Wrap that specific component in `React.memo`. Do not wrap everything.

Use `useMemo` when you have a computation inside a component that is genuinely expensive (sorting a list of 10,000 items, parsing a complex rule tree) and the inputs to that computation do not change on most re-renders. For formatting a date or filtering a list of 50 flags, `useMemo` is unnecessary.

Use `useCallback` primarily when you are passing a callback to a memoized child component (because if the callback reference changes every render, the child's memo is useless). Outside of that specific use case, `useCallback` is usually noise.

The rule of thumb: write your code without memoization. If the UI feels slow, measure. If you find a bottleneck, memoize that specific thing. This is the same principle as profiling a Laravel app: you do not add Redis caching to every query before you have identified which query is slow.

The "Measure First" Principle

React DevTools has a Profiler tab. It shows you exactly which components rendered, how long each render took, and what caused the render. Use it before reaching for any optimization.

The browser's Performance tab (the flame chart) shows you the full picture: JavaScript execution, layout, paint, network. If your app feels slow, the Profiler will tell you if React rendering is the bottleneck or if the problem is somewhere else (slow network requests, heavy CSS, large images).

Nine times out of ten, the performance problem is not React rendering. It is an unoptimized API endpoint, an N+1 query, a missing database index, or a large uncompressed image. Your backend instincts will serve you well here – the same diagnostic process applies.

---

8. Error Handling and Loading States

The Three States of Async UI

Every piece of UI that depends on asynchronous data has three possible states: loading, success, and error. This is true in Laravel too, but you rarely think about it because the server resolves all three before sending the response. The browser only ever sees the success state (or a full-page error).

In React, the client is often responsible for fetching data, which means the client must handle all three states explicitly. A component that fetches a list of flags needs to show a skeleton while loading, the flag list on success, and an error message if the fetch fails. Forgetting to handle loading or error states is the most common source of broken UX in React apps.

Suspense Boundaries as Loading Fences

React's Suspense is a mechanism for declaratively handling loading states. You wrap a part of your component tree in a `<Suspense>` boundary and provide a fallback UI. When any component inside that boundary is loading (suspended), React shows the fallback instead of the component tree.

Think of Suspense boundaries as "loading fences." They answer the question: "When this part of the page is loading, what should the user see?"

The placement of these fences is an important design decision. Consider the Flagline dashboard:

If you put a single Suspense boundary around the entire dashboard content area, then when any data is loading, the entire content area shows a spinner. This is simple but provides a poor experience – the sidebar and topbar are ready immediately, so why should the whole page look like it is loading?

If you put Suspense boundaries around each independent section – the flag list, the flag detail panel, the audit log sidebar – each section loads independently. The flag list might appear while the audit log is still loading. This is more granular and provides a better experience, but it can look chaotic if sections pop in at different times.

The right approach is somewhere in between. Group related content that should load together within the same Suspense boundary. The flag list and its filters are one loading unit. The flag detail panel is another. The audit log feed is another. Each gets its own boundary with an appropriate skeleton (a faded-out version of the expected layout, not a generic spinner).

In Next.js specifically, `loading.tsx` files act as automatic Suspense boundaries at the route level. A `loading.tsx` inside `app/(dashboard)/dashboard/[projectSlug]/[environment]/flags/` will show while the flags page's server component is running its data fetch. This maps closely to how you might show a loading screen in a Laravel app that has a slow controller.

Error Boundaries as Error Fences

Error boundaries are the error equivalent of Suspense boundaries. They catch JavaScript errors in their child component tree and display a fallback UI instead of crashing the entire page.

The placement logic is similar. Consider what should happen when different parts of the Flagline dashboard fail:

If the flag list fails to load (database error, network issue), the user should see an error message in the flag list area with a retry button. The sidebar, topbar, and navigation should still work. The user might want to switch to a different project or check the audit log.

If the entire page fails at the layout level (auth token expired, the project does not exist), a full-page error is appropriate. Redirect to login or show a 404.

If a single flag toggle fails (permission denied, concurrent modification), you do not want an error boundary at all – you want an inline error message or a toast notification. Error boundaries are for unexpected failures, not for expected error responses.

The framework is: - **Route-level error boundary (`error.tsx` in Next.js):** Catches crashes in the entire page. Shows a "Something went wrong" message with a retry button. The layout (sidebar, topbar) survives. - **Section-level error boundaries:** Wrap independently-loadable sections. If the audit log component throws, the flag list still works. - **Inline error handling:** For expected errors (validation failures, permission denials), handle them in the component with conditional rendering or toast notifications. Do not use error boundaries for these.

In Laravel terms: error boundaries are like `try/catch` blocks in your controllers, but scoped to regions of the UI rather than individual statements. The `error.tsx` file is like Laravel's exception handler rendering a custom error page. Section-level error boundaries are like wrapping individual Blade partials in their own error handling – something you could do in Laravel but rarely needed to.

---

9. Component Design Thinking for This Dashboard

This section walks through the thought process – not the implementation – for three representative components in the Flagline dashboard. The goal is to show you how a React developer thinks about decomposition, state, and interaction before writing any code.

Thinking Through the RuleBuilder

The RuleBuilder is the most complex UI component in Flagline. It is where users define targeting rules for a flag: "enable this flag for users in the US on a Pro plan whose account was created after January 2024." Rules can be nested with AND/OR operators.

First question: what is the data shape?

Before thinking about UI at all, think about the data structure. Targeting rules are a tree: a top-level group (AND/OR) contains conditions, and each condition can itself be a group. This is a recursive data structure. In Flagline's database, this is stored as JSONB (see 05 — Database & Prisma Reference for the schema). The component's job is to let the user visually edit this tree.

Second question: where does the state live?

The rule tree is form state. It lives locally in the component (or in a form library) until the user saves. It should not be in the URL (too complex), not in context (only one component needs it), and not in TanStack Query (it has not been persisted yet).

The form has two modes: editing (the user is building rules) and saved (the rules are persisted to the database). The transition is explicit – a "Save" button. This is a standard form pattern, not a real-time-sync pattern.

Third question: how do I decompose the UI?

Think recursively, matching the data structure. A `RuleGroup` component renders an AND/OR selector and a list of children. Each child is either a `RuleCondition` (a single condition: attribute, operator, value) or another `RuleGroup` (a nested group). `RuleGroup` renders itself recursively for nested groups.

This is one of those cases where the component tree mirrors the data tree. When you see a recursive data structure, the component structure should be recursive too.

Fourth question: what interactions does the user need?

Adding a condition to a group. Removing a condition. Changing a group's operator (AND to OR). Adding a nested group. Dragging to reorder conditions (maybe – consider if this is worth the complexity). Each of these is a mutation to the rule tree.

The callbacks for these mutations flow up from the leaf conditions to the root group. A `RuleCondition` does not mutate the tree directly. It calls an `onChange` prop. The parent `RuleGroup` handles the update to its portion of the tree. This is the standard “state up, events up” pattern.

Fifth question: should I use a form library?

For a complex nested form like this, a form library (React Hook Form, for example) provides two benefits: it manages the form state efficiently (no unnecessary re-renders), and it handles validation. But the RuleBuilder’s recursive structure is awkward to express in most form libraries’ API. You might end up fighting the library more than it helps you.

The pragmatic approach: start with plain `useState` holding the rule tree. Add validation manually. If performance becomes an issue (the rule tree gets very deep and every keystroke re-renders the entire tree), then consider a form library or memoization. Do not reach for the complex tool before you have a concrete problem.

Thinking Through the FlagToggle

The FlagToggle is the most interaction-dense component. It is a switch that enables or disables a flag in an environment. It looks simple but involves subtle complexity.

First question: what happens when the user clicks?

The toggle should flip immediately (optimistic update). A request goes to the server. If the server confirms, nothing visible happens – the UI is already correct. If the server rejects (permission error, flag is locked), the toggle flips back and an error message appears.

This is the optimistic update pattern described in Section 5. The key design decision is: what is the source of truth during the optimistic window? The local state says “enabled.” The server has not confirmed yet. If another component reads the flag state from TanStack Query’s cache, does it see the optimistic value or the last confirmed value? The answer is: TanStack Query’s optimistic update mechanism handles this. When you perform an optimistic mutation, TanStack Query updates the cache immediately, so all consumers see the optimistic value. If the mutation fails, TanStack Query rolls back the cache.

Second question: what about permissions?

Not every user should be able to toggle every flag. A viewer should see the toggle but not be able to click it. A developer might toggle in staging but not production. The toggle component needs to know the user’s permissions for this specific action.

The thinking here is: the toggle component receives a `disabled` prop or checks permissions internally via `usePermission('flag.toggle', { environment })`. The component itself does not enforce authorization – it just disables the interactive element. The server enforces authorization on the actual mutation. The client-side check is purely UX (do not show a clickable toggle to someone who cannot use it).

Third question: what about concurrent modifications?

If two users toggle the same flag at the same time, what happens? Both see the optimistic update locally. Both requests hit the server. The server processes them sequentially. The final state depends on ordering. SSE then pushes the final state to both clients.

For Flagline, this is acceptable. Flag toggles are idempotent (toggling to "on" when it is already "on" is a no-op). The SSE update will correct any briefly-stale UI. You do not need conflict resolution for this interaction – it is not a collaborative text editor. Acknowledge the possibility, decide it is acceptable, and move on.

Fourth question: should this be one component or several?

The toggle itself is a pure UI element: a switch that is either on or off and fires a callback when clicked. The optimistic update logic, permission check, and error handling are all behavior layered on top. You might structure this as a generic `Switch` UI component (no business logic, just visual toggle) used by a `FlagToggle` component (which adds the Flagline-specific behavior: permissions, optimistic mutation, error toast).

This separation means the `Switch` component is reusable for any on/off UI (notification preferences, feature visibility settings), while `FlagToggle` is specific to the flag-toggling use case.

Thinking Through the AuditLogFeed

The AuditLogFeed shows a chronological list of actions taken in the project: flag created, flag toggled, member invited, rules changed. It needs pagination (there could be thousands of entries), filtering (by event type, by actor), and real-time updates (new entries appear at the top).

First question: what is the loading pattern?

The initial page load should show the most recent entries. The user can scroll down to load more (infinite scroll) or click "Load more" (button-based pagination). This is cursor-based pagination, not page-based, because new entries are constantly being added at the top and page-based pagination would shift.

In Laravel, you have used `$query->cursorPaginate()`. The same concept applies here. Each batch of results comes with a cursor pointing to the last item. The next request says "give me entries after this cursor."

TanStack Query has an `useInfiniteQuery` hook designed for exactly this pattern. It manages an array of "pages" (batches of results) and provides a `fetchNextPage` function.

Second question: how do real-time entries integrate?

When a new audit log entry is created (someone toggled a flag), the SSE channel pushes a notification. The question is: do you prepend the new entry to the local list or refetch the first page?

Prepending is faster but introduces a subtlety: the new entry might not match the current filter. If the user is filtering for "flag.toggled" events and the new event is "member.invited," you should not show it. Rather than replicating filter logic on the client, the simpler approach is to invalidate the query and let the server re-filter. The brief refetch is imperceptible.

However, if no filters are active (the common case), you could prepend the entry directly from the SSE event for instant feedback, while still refetching in the background to guarantee consistency.

Third question: how do I handle the interaction between real-time and pagination?

If the user has scrolled down and loaded three pages of results, and a new entry arrives at the top, what happens? If you prepend it, the entire list shifts down by one item, and the user's scroll position jumps. This is jarring.

The better UX is: if the user is scrolled to the top (viewing the latest entries), new entries appear immediately. If the user has scrolled down, show a “New entries available” banner at the top. When clicked, it scrolls to the top and shows the new entries.

This is a UX decision, not a technical one. But it informs your component design: the AuditLogFeed needs to know whether the user is scrolled to the top. This is a piece of local state derived from a scroll position – a `useRef` on the scroll container and an intersection observer on the first entry, or a scroll event listener.

Fourth question: how should I decompose this?

- `AuditLogFeed`: The container. Manages the infinite query, the SSE subscription, the scroll tracking, and the “new entries” banner. This is a client component (it has interaction and state).
- `AuditLogEntry`: A single row. Renders the event type icon, the actor name, the description, and the timestamp. This is likely a client component too (it is a child of a client component), but it is pure – given the same props, it always renders the same thing.
- `AuditLogFilters`: A set of filter controls (event type dropdown, date range picker, actor search). Client component. Updating filters changes the URL query params (because filters should be shareable and survive refresh).

The separation here is driven by the different responsibilities: the feed manages the data lifecycle, each entry handles its own rendering, and the filters manage user input.

The General Approach

Across all three examples, notice the pattern of questions:

1. **What is the data shape?** Start with data, not UI.
2. **Where does the state live?** Apply the framework from Section 3.
3. **How does the UI decompose?** Let the data structure guide the component structure.
4. **What interactions does the user need?** List them before implementing.
5. **What are the edge cases?** Concurrent modifications, error states, empty states, loading states. Think about these before coding.
6. **Should this be one component or several?** Separate business logic from presentation. Separate reusable UI from domain-specific behavior.

If you ask these questions for every non-trivial component, you will arrive at clean designs without needing to follow prescriptive component architecture rules. The questions are the framework.

Wrapping Up

The shift from Laravel to React is not primarily a syntax change. It is a change in where computation happens (server vs client vs both), how state is managed (explicit and granular vs implicit and global), and how the UI stays in sync with data (reactive re-rendering vs full-page re-rendering).

The good news is that many of the instincts you have built over five years of PHP development transfer directly. The importance of separating concerns. The value of thin controllers (thin components). The discipline of keeping the database as the source of truth. The practice of building simple solutions first and optimizing only when you have measured a problem.

Server components, in particular, should feel familiar. They are server-rendered templates with direct database access. The mental model is not alien – it is Blade with better composability.

The genuinely new concepts – reactive state, the component lifecycle, the serialization boundary between server and client, optimistic updates, client-side caching – are worth investing time to understand deeply. Do not just learn the APIs. Understand why they exist and what problems they solve. Once you have the mental model, the APIs are just syntax.

Build the Flagline dashboard one component at a time. Start with server components (they feel like home). Add client components where you need interactivity. Extract hooks when you see repeated patterns. Reach for optimization tools only when you have measured a problem. Trust the framework to be fast by default.

The questions in Section 9 – about data shape, state placement, decomposition, interactions, and edge cases – are the most valuable takeaway from this guide. They are the questions a senior React engineer asks unconsciously. Make them conscious, ask them deliberately, and you will write good React code from the start.