Contents

03 – Next.js Thinking Guide for Flagline 2
- Table of Contents 2
- 1. The App Router Mental Model 2
  - The filesystem IS your router 2
  - Layouts as persistent shells 3
  - Route groups as organizational boundaries 3
  - How to think about the Flagline structure 4
  - The nested layout model and why it matters for a dashboard 5
  - Loading states and error boundaries: thinking in segments 5
- 2. Server Components vs Client Components – The Decision Framework 6
  - The core question 6
  - The Laravel mental model bridge 6
  - The "push the boundary down" principle 7
  - Walking through Flagline's dashboard, page by page 7
  - The serialization boundary 8
  - Common mistakes and how to avoid them 9
  - When the whole page should be a client component 9
- 3. Server Actions – Thinking About Mutations 9
  - The mental model shift 9
  - Validation: Zod replaces Form Requests 10
  - How server action return values work 10
  - When server actions make sense vs API endpoints 11
  - The revalidation mental model 11
- 4. Middleware Thinking 12
  - What middleware IS in Next.js 12
  - What to do in middleware 13
  - What NOT to do in middleware 13
  - The matcher pattern 13
  - How to structure the middleware function 14
- 5. The "Two APIs" Architecture 14
  - The key architectural decision 14
  - Why separate them? 15
  - The thinking process for each request 15
  - How they share the database and Redis 16
  - The data flow for a flag toggle 16
- 6. Authentication Thinking 17
  - The core question: where does the session live? 17
  - Session vs JWT: when to pick which 17
  - How auth is checked at different layers 18
  - The multi-tenancy model 18
  - Session data vs database lookups: the freshness tradeoff 19
- 7. Caching Mental Model 19
  - Why caching is confusing in Next.js 19
  - The key question: how fresh does this data need to be? 20
  - The layers, and when each applies 20
  - The Flagline caching strategy 21

The revalidation mental model: tags vs paths . . . 21
When to opt out entirely . . . 22
8. Deployment Thinking . . . 23
The Vercel model vs the Forge model . . . 23
Environment variable scoping . . . 23
Preview deployments as a workflow tool . . . 24
How this compares to deploying the full Flagline stack . . . 24
The build step: what happens and why it matters . . . 25
Thinking about database migrations in deployment . . . 25
Bringing It All Together . . . 26
The adjustment period . . . 26

03 – Next.js 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, Vercel. **Tone:** This is a thinking guide, not a code reference. It explains how to reason about Next.js architecture decisions. Code snippets are minimal and serve only to anchor concepts.

Table of Contents

1. The App Router Mental Model
2. Server Components vs Client Components – The Decision Framework
3. Server Actions – Thinking About Mutations
4. Middleware Thinking
5. The “Two APIs” Architecture
6. Authentication Thinking
7. Caching Mental Model
8. Deployment Thinking

---

1. The App Router Mental Model

The filesystem IS your router

In Laravel, routing is declarative and centralized. You open `routes/web.php`, scan through it, and you know every URL your application responds to. You register routes explicitly: `Route::get('/dashboard/{project}/flags', [FlagController::class, 'index'])`. If you forget to add a line, the route does not exist.

Next.js App Router inverts this entirely. There is no routes file. Instead, the directory structure under `app/` is the routing table. If the file `app/dashboard/page.tsx` exists, the URL `/dashboard` exists. If you delete the file, the route disappears. There is nothing to register, nothing to configure, nothing to forget to wire up.

This sounds simple, but it changes how you think about your application's structure. In Laravel, you design routes first (the URLs, the verbs, the middleware) and then create controllers to handle them. In Next.js, you design the filesystem first, and the routes emerge from that structure. Your file tree is not just an organizational convenience – it is an architectural decision that directly determines your URL hierarchy, your layout nesting, and your code-splitting boundaries.

The shift matters because it means you cannot separate "where the code lives" from "what URL it responds to." In Laravel, you could put a controller anywhere and point a route at it. Here, the physical location of `page.tsx` files defines the URL. This constraint is actually freeing once you internalize it: looking at the file tree tells you everything about the app's routing, without needing to cross-reference a separate routes file.

Layouts as persistent shells

In Laravel, you define Blade layouts with `@extends('layouts.dashboard')` and inject content via `@yield('content')`. Every page that uses the dashboard layout explicitly declares that relationship. If you navigate from the flags page to the settings page, the entire page re-renders – the sidebar, the nav, everything. The layout is a template that gets re-stamped for every request.

Next.js layouts work differently in a way that matters enormously for a dashboard application. A `layout.tsx` file wraps every page in its directory subtree, and critically, it persists across navigations within that subtree. When a user navigates from `/dashboard/my-app/production/flags` to `/dashboard/my-app/production/audit-log`, the dashboard shell (sidebar, top navigation, organization picker) does not unmount and remount. It stays alive. Only the content area re-renders.

Think about why this matters for Flagline specifically. The dashboard has a sidebar with the organization picker, a list of projects, and navigation links. It has a top bar with the user menu. Below that, there are environment tabs. And then there is the actual page content. If each navigation destroyed and rebuilt the sidebar, you would lose scroll position, you would lose the open/closed state of dropdowns, and you would see a flash of loading state for UI that has not actually changed. Layout persistence eliminates all of that.

The nesting model compounds this benefit. Flagline has multiple layout levels: the root layout (HTML shell, providers), the dashboard layout (sidebar, top bar), the project layout (environment tabs, project context), and possibly a flags layout (for the split-panel view with parallel routes). Each layer persists independently. Navigate from one environment to another within the same project, and only the environment layout and below re-render. Navigate from one project to another, and only the project layout and below re-render. The dashboard shell stays stable throughout.

This is the single most important difference from Laravel's approach to layouts. In Laravel, layouts are templates. In Next.js, layouts are live, persistent components that maintain state across navigations.

Route groups as organizational boundaries

Here is where the thinking gets interesting. Flagline has three fundamentally different sections: marketing pages (landing, pricing, docs, blog), authentication pages (login, signup, forgot password), and the dashboard (everything behind auth). Each section needs a different visual shell.

Marketing pages have a public navbar and footer. Auth pages have a centered card layout. The dashboard has the sidebar and top bar.

In Laravel, you would handle this with different Blade layouts and middleware groups. In Next.js, you use route groups – directories wrapped in parentheses like `(marketing)`, `(auth)`, and `(dashboard)`. The key thing to understand is that parentheses mean "this directory affects layout nesting but does NOT appear in the URL." So `app/(marketing)/pricing/page.tsx` responds to `/pricing`, not `/marketing/pricing`. And `app/(auth)/login/page.tsx` responds to `/login`, not `/auth/login`.

Each route group can have its own `layout.tsx`. The marketing group gets the public navbar and footer. The auth group gets the centered card. The dashboard group gets the sidebar. The URLs stay clean, but the layout hierarchy is cleanly separated.

Think of route groups as organizational boundaries for your code that happen to also be layout boundaries. They answer the question: "Which pages share a common shell?" Group them together. The URL structure is independent from the layout structure, which is a flexibility that Laravel's approach does not offer – in Laravel, if you want a different layout, you change the `@extends` call in the Blade template, but the code organization is separate from the routing.

How to think about the Flagline structure

When designing the file tree for Flagline, the thinking process goes like this. Start with the user experience: what are the distinct "contexts" a user can be in?

First, the public context. A visitor lands on the marketing site. They see a navbar with links to pricing, docs, and a login button. They see a footer. Every page in this context shares that shell. This becomes the `(marketing)` route group.

Second, the authentication context. The user clicks "Sign Up" or "Log In." The navbar disappears. They see a clean, centered card on a minimal background. This is a different shell entirely. This becomes the `(auth)` route group.

Third, the dashboard context. The user is logged in. They see a sidebar, a top bar, and content. Everything here requires authentication. This becomes the `(dashboard)` route group.

Within the dashboard, there is further nesting. A project has multiple environments (development, staging, production). Each environment has flags, segments, and an audit log. The project-level layout provides environment tabs. The environment-level layout might provide sub-navigation. This nesting emerges naturally from the directory structure: `(dashboard)/dashboard/[projectSlug]/[environment]/flags/page.tsx` gives you the URL `/dashboard/my-app/production/flags` with four layers of layout wrapping it.

The `[projectSlug]` and `[environment]` directories are dynamic segments – the Next.js equivalent of Laravel's `{project}` route parameters. They work the same conceptually: the value from the URL is passed to the page component as a prop. But because the dynamic segment is also a directory, it can have its own layout, its own loading state, and its own error boundary. In Laravel, you would need to manually handle all of those concerns in the controller. Here, the framework handles them based on where you put files.

There is also the catch-all route pattern: `[...slug]` captures all remaining URL segments as an array. This is useful for the docs section of Flagline, where `/docs/sdk/react/hooks` should resolve to a specific documentation page without you defining a route for every possible path. The `slug` parameter would be `["sdk", "react", "hooks"]`, and your page component joins them to look up the right content. This is like Laravel's `{path?}` wildcard route parameter but more flexible because it captures the segments as an array rather than a single string.

The nested layout model and why it matters for a dashboard

Let me trace through what happens when a Flagline user navigates to `/dashboard/my-app/production/flags`. The framework assembles the response by nesting layouts:

The root layout renders the HTML element, the body, and wraps everything in providers (session, theme, etc.). Inside that, the dashboard layout checks authentication and renders the sidebar and top bar. Inside that, the project layout fetches the project by slug, verifies access, and renders the environment tabs. Inside that, the environment layout might set some context. And finally, the flags page renders the actual flag list.

Now the user clicks the "Audit Log" tab. Only the content below the environment layout re-renders. The sidebar does not re-fetch. The project layout does not re-query. The environment tabs do not flicker. The transition is near-instant because most of the layout tree is already hydrated and stable.

Now the user switches to a different project. The project layout and everything below it re-renders (new project data, new environment tabs). But the dashboard shell is still stable.

This cascading persistence model is what makes Next.js feel like a single-page application while still being server-rendered. Laravel cannot replicate this without building a full SPA with something like Inertia.js, and even then, Inertia does not have the concept of persistent nested layouts in the same way.

The implication for how you write code is significant. Each layout level should fetch only the data it needs for its own UI. The dashboard layout fetches the user session and organization list (for the sidebar). The project layout fetches the project and its environments (for the tabs). The page itself fetches the flags or audit entries or settings. Each layer is responsible for its own data, and the framework handles composing them efficiently.

Loading states and error boundaries: thinking in segments

Each route segment (each directory level with a `page.tsx`) can also have a `loading.tsx` and an `error.tsx`. This is another area where the filesystem-as-architecture principle pays off.

A `loading.tsx` file provides a streaming fallback. When the page component is doing async work (fetching data from Prisma), the loading component renders immediately while the page streams in. In Laravel, this has no direct equivalent – you would need client-side JavaScript or Turbo Frames to show a loading state while the server prepares the response. Here, the framework handles it: the browser receives the layout shell with the loading skeleton first, and the actual page content replaces it once the data is ready.

An `error.tsx` file catches errors in its subtree. If the flags page throws an exception (database timeout, unexpected null), the error boundary renders instead of crashing the entire page. The dashboard layout, sidebar, and navigation all remain functional. The user sees an error message with a retry button in the content area only. In Laravel, an exception would typically result in a full-page error view, losing all layout context. Here, errors are contained to the segment that threw them.

The thinking process for where to place loading and error files is: what is the smallest unit that should show a loading state independently? For Flagline, the flags list should have its own loading skeleton, but the dashboard shell should not show a loader when only the flags list is loading. So `loading.tsx` goes in the flags directory, not in the dashboard layout directory. Similarly, if the flags query fails, only the flags content area should show an error, not the entire dashboard. So `error.tsx` also goes at the flags directory level.

This per-segment granularity is one of the most powerful architectural tools in the App Router. It lets you design resilient UIs where failures in one section do not cascade to unrelated sections.

> **Coming from Laravel:** Think of this as if your Blade layouts were live Vue or Livewire components that persisted between page navigations, and the framework automatically figured out which layers to re-render based on what changed in the URL. Loading files are like having a built-in Turbo Frame placeholder for each section of your page. Error files are like having per-section exception handlers instead of a global `render` method in your exception handler. The granularity is finer than anything in the Laravel world.

---

2. Server Components vs Client Components – The Decision Framework

The core question

Stop thinking in terms of rules to memorize. Instead, ask one question about every piece of UI you build: "Does this need to respond to user interaction or access browser state?"

If the answer is no – if the component just displays data, renders HTML, and does not need to react to clicks, track hover states, manage form inputs, or read from browser APIs like `window` or `localStorage` – it should be a server component. Server components run on the server, have direct access to the database and environment variables, never ship their code to the browser, and are the default in the App Router. You do not need to do anything special to make a component a server component. It just is one, by default.

If the answer is yes – if the component needs `useState`, `useEffect`, `onClick`, `onChange`, keyboard event handlers, or any browser API – it must be a client component. You mark it with `"use client"` at the top of the file, and its code gets bundled and sent to the browser.

This is not a performance optimization to think about later. It is a fundamental architectural boundary that determines where your code runs, what it has access to, and how much JavaScript the user's browser downloads. Getting this right from the start is the difference between a fast dashboard and a sluggish one.

The Laravel mental model bridge

If you have used Blade with Alpine.js, you already understand this intuitively. Your Blade template is the server component: it runs on the server, has access to controllers, models, and the database, and outputs HTML. The Alpine.js directives sprinkled in (`x-data`, `x-show`, `@click`) are the client components: they add interactivity to specific elements without making the entire page interactive.

If you have used Livewire, the analogy is even closer. A Livewire component is like a client component – it has state, it responds to user actions. But the page it sits on is still server-rendered Blade. The key insight is the same: keep the interactive surface area as small as possible. You would not make an entire Blade view a Livewire component just because one button needs an onClick handler. The same principle applies here.

The "push the boundary down" principle

This is the most important architectural principle for building a Next.js dashboard. When you find yourself needing interactivity, do not make the entire page a client component. Instead, push the `"use client"` boundary as far down the component tree as possible.

Consider the Flagline flags page. It shows a toolbar with search and filters, a table of flags, and each flag row has a toggle switch and a context menu. The instinct from a Laravel background might be: "This page has interactive elements, so I need to make it a client page." Resist that instinct.

Instead, think about it layer by layer. The page itself fetches the flag data from Prisma – that is a server concern. The table that renders the list of flags is just mapping data to rows – that is a server concern. The toolbar with the search input needs to handle user input – that is a client concern. The toggle switch on each flag row needs to respond to clicks and call a server action – that is a client concern. The context menu on each row needs to track open/closed state – that is a client concern.

So the page is a server component. The table can be a server component. But the toolbar, the toggle switch, and the context menu are small, focused client components. The server component fetches the data and passes it down as props to these interactive islands.

Why does this matter? Three reasons. First, every line of code in a client component gets shipped to the browser as JavaScript. A server component that fetches 50 flags from Prisma and renders them in a table sends only the HTML – zero JavaScript for that rendering logic. If you made the whole page a client component, all of that rendering code would be in the browser bundle, plus you would need to fetch the data via an API call instead of directly querying Prisma. Second, server components can be async – they can `await` database queries directly in the render function. Client components cannot do this. Third, server components have access to secrets, environment variables, and the filesystem. Client components run in the browser and have access to none of those things.

Walking through Flagline's dashboard, page by page

Let me walk through the reasoning for each major piece of the Flagline dashboard.

The flags list page. The page component itself is a server component. It receives route parameters (`projectSlug`, `environment`), queries Prisma for the flags, and renders the result. It does not need `useState` or `useEffect`. It does not respond to clicks. It just fetches and renders. The flag toolbar (search input, filter dropdown, "Create Flag" button) is a client component because it needs to track input state and potentially update URL search params on user interaction. Each flag row's toggle switch is a client component because it needs an `onChange` handler. The rest of the row (the flag name, key, description, last-modified date) is display-only and can be rendered by the server.

The flag detail / edit page. If a user clicks on a flag to see its details and edit its targeting rules, the detail view fetches the flag data (server component) and passes it to a rule editor. The rule editor is a client component – it involves drag and drop, conditional logic builders, and complex interactive state. This is the right place for a client component because the entire purpose of this UI is interaction.

The audit log. This is almost entirely a server concern. Fetch the audit entries from Prisma, render them in a table. Pagination is handled via URL search params and `<Link>` components, which do not require client-side state. The entire page can be a server component. If you add a date range filter, that specific filter widget becomes a client component, but the table remains server-rendered.

The settings pages. Member list: server component – just fetching and displaying a list. Invite member form: client component – it has controlled inputs and form state. API key display with "click to reveal": client component – it needs `useState` to toggle between masked and revealed states. Danger zone with the delete project button: the page is a server component (it checks authorization), but the delete button with its confirmation dialog is a client component.

The sidebar. This is a client component because it is collapsible (needs state for open/closed), and the organization picker dropdown has interactive behavior (search, keyboard navigation, selection). However, the data for the sidebar (the list of organizations and projects) is fetched in the dashboard layout (a server component) and passed to the sidebar as props. The sidebar does not fetch its own data – it receives it.

The environment tabs. Client component, because they need to highlight the active tab based on the current URL (using `usePathname()`). But they receive the list of environments as props from the project layout (a server component).

Notice the pattern: server components own the data. Client components own the interaction. Data flows down from server to client via props. Interaction flows up from client to server via server actions (which we will discuss next).

The serialization boundary

When a server component passes props to a client component, those props cross a serialization boundary. The data must be serializable to JSON (with some extensions like Date, Map, and Set). You cannot pass functions across this boundary – with one critical exception: server actions, which we cover in the next section.

This means you cannot do something like pass an `onDelete` callback from a server component to a client component. Instead, the client component imports the server action directly and calls it. This feels unusual if you are used to the React pattern of passing callbacks down, but it is the intended architecture.

Prisma query results are plain objects by default, so they cross the boundary without issues. You do not need to manually serialize them. Just be aware that if you return a Prisma result that includes deeply nested relations, all of that data is serialized and sent to the client. Select only what the client component needs.

Common mistakes and how to avoid them

Marking a page as `"use client"` because one child needs interactivity. This is the most frequent mistake. You need a search input on the flags page, so you put `"use client"` at the top of the page file. Now the entire page is a client component: it cannot query Prisma directly, it ships all its code to the browser, and you need to fetch data via an API call. The fix: keep the page as a server component and extract the search input into a small client component.

Importing a client component into a server component and being confused about the boundary. A server component can render a client component as a child. The boundary is at the `"use client"` file, not at the import. When a server component imports and renders a client component, the server component remains a server component. The client component and everything it imports become client-side code. But the server component's code stays on the server.

Trying to use `async/await` in a client component. Client components cannot be async functions. If you need to fetch data, either do it in a parent server component and pass it as props, or use a client-side data fetching library (like SWR or React Query) with a `useEffect`. The server component approach is almost always preferable because it avoids client-side loading states and keeps the data fetching on the server.

Passing non-serializable props to a client component. If you query Prisma and get back an object with a `Decimal` type (for currency) or a `BigInt`, these may not serialize cleanly across the boundary. Use Prisma's `select` to choose only the fields you need, and convert any exotic types to primitives before passing them as props.

When the whole page should be a client component

It is rare, but it happens. If you are building a page that is entirely interactive – say, a visual rule builder where the entire screen is a drag-and-drop canvas, or a real-time monitoring dashboard with WebSocket-driven updates – then making the page a client component is the right call. The key is that this should be a deliberate architectural decision, not a default. Ask yourself: "Is there any part of this page that is just displaying data without interaction?" If yes, keep that part as a server component and make only the interactive parts client components.

> **Coming from Laravel:** The server/client split maps to Blade/Alpine more cleanly than you might expect. The big difference is that in Laravel, you consciously add JavaScript. In Next.js, you consciously remove it by defaulting to server components. The mental model is inverted: instead of asking "where do I need JavaScript?", ask "where can I avoid it?"

---

3. Server Actions – Thinking About Mutations

The mental model shift

In Laravel, the lifecycle of a form submission is well-defined. The user fills out a form. The form POSTs to a named route. That route hits a controller method. The controller validates the input via a Form Request, does the work, and redirects back with a success message or validation errors.

Server actions in Next.js replace most of this pipeline. Instead of defining a POST route and a controller, you define an async function marked with `"use server"`. That function can be passed directly to a form's `action` attribute. When the form is submitted, the function runs on the server. No API route needed. No fetch call. The framework handles the serialization, the network transport, and the response.

The key mental model is this: a server action is a function that lives on the server but can be called from the client. The `"use server"` directive tells the framework to create a network endpoint for this function automatically. You never see the endpoint – there is no URL to think about. You just call the function, and it executes on the server.

Validation: Zod replaces Form Requests

In Laravel, Form Requests are classes that define validation rules. They run automatically before the controller method. If validation fails, the framework redirects back with error messages.

The equivalent in the Next.js world is Zod schemas. You define a schema that describes the expected shape and constraints of the input, and then you validate the incoming data against it in the server action. But there is an important difference: validation is not automatic. You must explicitly call the schema's `safeParse` method and handle the result yourself.

This is actually an advantage for a thinking developer. In Laravel, the magic of Form Requests sometimes obscures what is happening. Here, you see exactly what is being validated, when, and how. You define a Zod schema like `z.object({ key: z.string().min(1).regex(/^[a-z][a-z0-9_-]*$/), name: z.string().min(1).max(128) })`, call `schema.safeParse(data)`, and check whether it succeeded. If it failed, you return the errors. The calling component displays them.

The thinking process for validation in a server action is: first, verify authentication (is there a session?). Second, parse and validate the input (does the data match the schema?). Third, verify authorization (does this user have permission to do this?). Fourth, execute the mutation. Fifth, revalidate any cached data that is now stale. This order matters – you want to fail fast and cheaply before doing expensive database work.

How server action return values work

This is a subtle but important point. In Laravel, the flow after a form submission is: validate, act, redirect. The redirect is what delivers feedback to the user – you flash a success message or validation errors to the session, and the subsequent GET request picks them up.

Server actions do not redirect by default. They return a value. The calling component receives that value and re-renders with it. This means you design your server action return types to carry the information the UI needs: success/failure status, validation errors keyed by field name, and any messages.

The standard pattern is to define a return type with `success: boolean`, an optional `message` string, and an optional `errors` record mapping field names to error message arrays. The calling component uses React's `useActionState` hook (or the older `useFormState`) to manage this state across submissions. The hook gives you the current state, a form action function, and a pending boolean – everything you need to build a responsive form without managing the state manually.

Think of `useActionState` as the equivalent of Laravel's `@error('field')` directive in Blade, but typed and without the redirect-flash-session dance. The form submits, the server action runs, the return value updates the component state, and the UI reflects the result. All within a single page, no navigation required.

One nuance: server actions support progressive enhancement. If JavaScript fails to load, the form still submits as a standard HTML POST. The server action runs, and Next.js renders the page with the updated state on the response. This means your forms work without JavaScript – a property that Laravel forms have by default but most React applications lose.

When server actions make sense vs API endpoints

This is a decision that comes up repeatedly in a SaaS like Flagline. The general framework is:

Use a server action when the mutation is initiated by a user interacting with your Next.js UI. Creating a flag, toggling a flag, inviting a team member, updating project settings – all of these are actions triggered by a human clicking something in the dashboard. Server actions are perfect here because they give you a direct function call from the UI to the server, with automatic type safety, no manual API endpoint management, and built-in integration with React's form handling and optimistic updates.

Use a Next.js route handler (an API endpoint in `app/api/`) when the request comes from outside your UI. Stripe sends a webhook to `/api/webhooks/stripe`. A CI/CD pipeline triggers a deployment flag via `/api/internal/deploy`. Another service in your infrastructure calls an internal endpoint. These are not user-initiated UI actions – they are machine-to-machine communications that need a stable URL.

Use the separate Fastify evaluation API for requests from customer SDKs. This is the high-throughput path where customer applications check flag values. It has entirely different performance requirements and scaling characteristics (covered in section 5).

The mistake to avoid is creating API route handlers for every mutation and then calling them with `fetch` from client components. That pattern was common in the Pages Router era. With the App Router, server actions eliminate the need for most internal API endpoints. If you find yourself writing a `route.ts` that is only called by your own dashboard UI, you probably want a server action instead.

There is one gray area: client components that need to fetch data dynamically (for example, an autocomplete that searches flags as the user types). In this case, you could use a server action that returns data (server actions are not limited to mutations – they can be read-only too), or you could create a route handler and call it with `fetch` from the client. Both work. Server actions are simpler for one-off data fetches. Route handlers are better if you need standard HTTP features like streaming, cursor-based pagination, or content negotiation. Use your judgment based on the complexity of the data-fetching pattern.

The revalidation mental model

After a mutation, you need to answer the question: "What cached views are now stale?"

In Laravel, you might call `Cache::forget('flags.list')` after creating a flag. In Next.js, the equivalent is `revalidatePath` and `revalidateTag`. But the mental model is broader because

Next.js caches rendered pages, not just data.

When you create a flag, the flag list page is stale. You call `revalidatePath('/dashboard/my-app/production/flags')` to tell Next.js that the cached version of that page (if any) should be regenerated on the next request. Alternatively, if you have tagged your data fetches with cache tags, you call `revalidateTag('project:abc:flags')` to invalidate everything that depends on that tag.

The thinking process is: first, what data did I change? Second, what pages display that data? Third, should I invalidate by path (specific pages) or by tag (everything that uses this data)?

Path-based revalidation is simpler to reason about but less precise. Tag-based revalidation is more granular but requires you to set up a consistent tagging scheme. For Flagline, a reasonable approach is to use tags scoped to the entity: `project:{id}:flags`, `project:{id}:members`, `env:{id}:audit-log`. When a flag changes, invalidate the flags tag. When a member is invited, invalidate the members tag. This way, the audit log page is not unnecessarily revalidated when someone toggles a flag.

One subtlety: revalidation is asynchronous and happens on the next request. If you toggle a flag and expect the UI to update immediately, you need to pair revalidation with either optimistic updates (using React's `useOptimistic` hook) or a `router.refresh()` call from the client component. The revalidation tells the server-side cache to regenerate; the optimistic update makes the UI feel instant to the user.

> **Coming from Laravel:** Server actions replace the route-controller-form request pipeline with a single function. The most important conceptual shift is that there is no redirect. In Laravel, after a form submission, you typically `return redirect()->back()->with('success', 'Flag created')`. In Next.js, the server action returns data directly to the calling component. The component re-renders with the new state. The URL can stay the same or change programmatically. You are not doing the HTTP redirect dance – you are doing an RPC call.

---

4. Middleware Thinking

What middleware IS in Next.js

In Laravel, middleware is a pipeline of handlers that wrap your request. You can have many middleware classes, compose them into groups, and assign them to routes or groups of routes. Each middleware can inspect the request, modify it, inspect the response, or short-circuit the pipeline entirely.

Next.js middleware is conceptually similar but structurally different in ways that matter. There is exactly one middleware file: `middleware.ts` at the project root. Not one per concern – one total. It runs before every matched request, and it runs at the edge.

"At the edge" is the crucial detail. Edge middleware runs on Vercel's global CDN nodes, not on your origin server. This means it executes closer to the user (lower latency for redirects) but in a restricted environment. You cannot import Prisma. You cannot open a TCP connection to your database. You cannot run heavy computation. You have access to the Web APIs (`fetch`, `crypto.subtle`, `Request`, `Response`) and not much else.

Think of it as a bouncer at the door of a club. The bouncer can check your ID (parse a JWT from a cookie), tell you to go somewhere else (redirect), stamp your hand (set a header or cookie), or refuse entry (return a 403). But the bouncer cannot go look up your account details in the back office (query the database) or run a background check (heavy computation). Those things happen inside the club (in server components and server actions).

What to do in middleware

Given the edge constraints, middleware in Flagline should handle exactly three categories of concerns.

Authentication redirects. Check if a request to `/dashboard/*` has a valid session cookie. If not, redirect to `/login`. Check if a request to `/login` already has a valid session. If so, redirect to `/dashboard`. This does not require a database lookup – you are parsing a JWT from a cookie, which is a lightweight cryptographic operation that the edge runtime can handle.

Security headers. Set `X-Frame-Options`, `X-Content-Type-Options`, `Referrer-Policy`, and other security headers on every response. This is a one-liner per header and makes sense to do at the edge so every response gets them, including cached static pages.

Tenant resolution for custom domains. If Flagline supports custom domains for customer-facing flag endpoints, middleware can read the `Host` header, determine which tenant the request belongs to, and set a header (`x-flagline-tenant-id`) that downstream server components or route handlers can read. The middleware itself does not look up the tenant in the database – it either uses a lightweight lookup (like Upstash Redis via HTTP) or passes the raw domain through and lets the server component resolve it.

What NOT to do in middleware

Do not put authorization logic in middleware. Authorization (checking whether user X can access project Y in organization Z) requires database queries, and the edge runtime cannot run Prisma queries against your PostgreSQL database. Authorization belongs in server components and server actions, where you have full Node.js access.

Do not put rate limiting in middleware unless you use an edge-compatible solution like Upstash (which provides an HTTP-based Redis client). Traditional rate limiting with a local Redis connection will not work at the edge.

Do not try to replicate Laravel's rich middleware pipeline. In Laravel, you might have `auth`, `verified`, `role:admin`, `throttle:60,1` all chained together. In Next.js, middleware handles the lightweight gate checks (is there a session?), and the heavier checks (is this user an admin of this organization?) happen deeper in the stack.

The matcher pattern

The matcher config determines which routes middleware runs on. The thinking is: middleware should run on routes that need pre-processing, and it should NOT run on routes that serve static assets (images, CSS, JS bundles).

A common pattern is a negative regex that excludes static file paths: `"/((?!_next/static|_next/image|favi`
This means "run on everything except Next.js static assets and the favicon." You then use `if` statements inside the middleware function to apply different logic to different path prefixes.

This is less declarative than Laravel's route middleware groups, and that is intentional. Having one middleware function with explicit conditionals makes it easier to understand the execution flow. You open one file and you can see every pre-request check your application makes, in order.

How to structure the middleware function

The temptation is to cram every concern into middleware because it runs before everything else. Resist this. Think of middleware as a series of early-exit checks, ordered from most common to least common.

First, check if the route is one that needs protection (dashboard, onboarding). If the user is not authenticated, redirect to login. This handles the vast majority of middleware logic and applies to the most requests.

Second, check if the route is an auth page (login, signup). If the user IS already authenticated, redirect to the dashboard. This prevents the confusing experience of a logged-in user seeing a login form.

Third, handle any special concerns like custom domain resolution or security headers.

Each check should be independent – it should not depend on the result of a previous check. This makes the middleware easier to reason about and test. If you find yourself building complex conditional logic with nested if-else chains, that is a sign that some of the logic belongs in a server component or layout rather than in middleware.

The middleware function should be short. If it grows beyond 50-60 lines, you are probably putting too much logic at the edge. Extract the complex parts into server components where you have full Node.js capabilities.

> **Coming from Laravel:** The biggest adjustment is going from "many middleware classes composed declaratively" to "one middleware function with conditionals." It feels less elegant at first. In practice, it is easier to reason about because there is no hidden composition order – you read the function top to bottom and you know exactly what happens. The constraint of "one function, keep it short" also prevents the middleware sprawl that happens in mature Laravel applications where you end up with 15 middleware classes and struggle to remember which ones run on which routes.

---

5. The "Two APIs" Architecture

The key architectural decision

Flagline serves two fundamentally different kinds of HTTP traffic, and recognizing this distinction early is one of the most important architectural decisions in the project.

The first kind is dashboard traffic. Humans using the Flagline web app to create flags, edit targeting rules, manage team members, and view audit logs. This traffic is characterized by low volume (tens of requests per second at most), complex rendering (full HTML pages with nested layouts), and a need for the richest possible developer experience (server components, server actions, streaming, Suspense).

The second kind is evaluation traffic. Customer applications calling the Flagline API millions of times per day to check whether a flag is enabled for a given user. This traffic is characterized by extremely high volume, a need for sub-10ms response times, simple JSON payloads, and long-lived connections (Server-Sent Events for real-time flag updates).

Trying to serve both through Next.js would be a mistake, and understanding why is critical to the architecture.

Why separate them?

Scaling characteristics are completely different. Next.js on Vercel runs as serverless functions. Each request spins up a function, does its work, and the function can be recycled. This is perfect for dashboard traffic: infrequent, bursty, complex responses. It is terrible for evaluation traffic: constant, high-volume, latency-sensitive. Serverless cold starts add 50-200ms of latency, which is unacceptable when your SLA is sub-10ms.

The evaluation API runs on a persistent server (Fly.io, Railway, or a VPS). It boots once and stays running. It maintains a connection pool to PostgreSQL. It keeps a local Redis cache of flag configurations. It can serve SSE streams that stay open indefinitely. None of these things work well in a serverless environment.

Deployment independence matters. When you push a change to the dashboard UI – say, redesigning the settings page – you do not want to risk affecting the evaluation API. And when you optimize the evaluation engine, you do not want to trigger a rebuild of the marketing site. Separate deployables mean separate risk profiles.

Resource isolation. A bug in the dashboard that causes runaway memory usage does not take down flag evaluation for your customers' applications. A spike in evaluation traffic does not slow down the dashboard for your internal team.

The thinking process for each request

When designing a feature, ask: "Where does this request come from and where should it go?"

If the request comes from a human using the Flagline web app, it goes through Next.js. The user clicks something, which either navigates to a new page (server component rendering) or calls a server action (mutation). All dashboard-related data fetching and mutation happens within the Next.js application.

If the request comes from a customer's SDK (the `@flagline/js` or `@flagline/react` package), it goes to the Fastify evaluation API. The SDK calls `GET /v1/flags` to fetch all flag configurations, `POST /v1/evaluate` to evaluate a specific flag for a user context, or connects to `GET /v1/flags/stream` for real-time updates via SSE.

If the request comes from an external service like Stripe, it goes to a Next.js route handler at `/api/webhooks/stripe`. Webhooks are infrequent and dashboard-related (they update billing state), so they belong in the Next.js application.

If the evaluation API needs to tell Next.js that cached dashboard views are stale (because a flag was changed via API rather than the UI), it calls an internal Next.js route handler at `/api/internal/revalidate` with a shared secret and a list of cache tags to invalidate. This is the one case where the evaluation API communicates with the Next.js application.

How they share the database and Redis

Both services connect to the same PostgreSQL database. The Next.js application handles writes (creating flags, updating rules, managing users). The evaluation API primarily reads (fetching flag configurations). This is a natural read/write split that could eventually be formalized with PostgreSQL read replicas.

Redis serves a different purpose for each. The evaluation API uses Redis as a primary cache for flag configurations. When an SDK requests flag data, the evaluation API checks Redis first and only falls back to PostgreSQL if the cache misses. This is what achieves sub-10ms response times. The Next.js application uses Redis (via Upstash, for edge compatibility) for rate limiting in middleware and potentially for session storage.

Both services use Redis pub/sub as a communication channel. When the dashboard updates a flag, the server action publishes a message to a Redis channel. The evaluation API subscribes to that channel and invalidates its local cache. This gives near-real-time propagation of flag changes to customer applications.

The data flow for a flag toggle

To make this concrete, trace the full data flow when a Flagline user toggles a flag from disabled to enabled in the dashboard:

1. The user clicks the toggle in the dashboard UI (client component). The client component calls the `toggleFlag` server action.
2. The server action (running in a Vercel serverless function) verifies the session, checks authorization, and updates the flag in PostgreSQL via Prisma.
3. The server action publishes a message to a Redis pub/sub channel: "flag X in project Y environment Z was updated."
4. The server action calls `revalidateTag` to invalidate the dashboard's cached view of the flags list (if caching is enabled).
5. The evaluation API, which subscribes to that Redis channel, receives the message and invalidates its local Redis cache for that flag's configuration.
6. The next time a customer's SDK requests flag data from the evaluation API, it gets the updated value (either from a fresh Redis cache or directly from PostgreSQL).
7. If the customer's SDK is connected via SSE, the evaluation API pushes the update through the SSE stream immediately.

Notice that the Next.js application and the evaluation API never talk to each other directly in this flow. They communicate through shared infrastructure: PostgreSQL (the source of truth) and Redis (the invalidation signal). This loose coupling means either service can be deployed, restarted, or scaled independently without breaking the other.

> **Coming from Laravel:** Think of this as if your Laravel application served the admin panel, and you had a separate Lumen (or Octane) service for the public API. In Laravel,

you might be tempted to serve everything from one application because Forge makes single-app deployment easy. But if your API is handling thousands of requests per second with sub-10ms latency requirements, separating the services is the right call. The same logic applies here.

---

6. Authentication Thinking

The core question: where does the session live?

In Laravel, sessions typically live in the database (or Redis, or files) on the server side. The browser gets a session cookie with an opaque ID. On each request, the framework looks up the session by that ID. This means every request that needs auth requires a database or Redis lookup.

In Next.js with Auth.js (NextAuth.js v5), you have a choice: database sessions or JWT sessions. For Flagline, JWT sessions are the right choice, and here is the reasoning.

A JWT session stores the session data (user ID, email, organization memberships) directly in an encrypted cookie. No database lookup needed to verify authentication. The cookie is signed and encrypted, so it cannot be tampered with. This matters because of how many places in a Next.js application need to check the session:

Middleware checks the session at the edge (before any server component runs). A database session would require the edge runtime to query PostgreSQL, which is not possible without special HTTP-based database drivers. A JWT can be verified by just checking the cryptographic signature, which the edge runtime can do.

Server components check the session when rendering. Layout components check it to decide what to show in the sidebar. Page components check it to verify access to a specific resource. If each of these required a database roundtrip, rendering a single page could trigger multiple session lookups.

Server actions check the session on every mutation. If every toggle-flag call required a database session lookup before it could even verify the user's identity, you would add latency to every interaction.

With JWTs, all of these checks are just "decrypt the cookie and read the data." Fast, no I/O required.

The tradeoff is that JWT data can become stale. If a user changes their name or gets removed from an organization, the JWT still contains the old data until it expires or is refreshed. For Flagline, this is acceptable because organization membership changes are infrequent, and the JWT is refreshed on every page load via the Auth.js session callback.

Session vs JWT: when to pick which

The JWT-vs-database-session decision is worth dwelling on because it affects how you architect auth checks throughout the app.

Database sessions store session data server-side and give the browser an opaque session ID cookie. Every time you need session data, you look it up in the database. The advantage is that sessions can be invalidated instantly (delete the row, and the session is dead). The disadvantage is the database lookup on every request.

JWT sessions store session data in an encrypted cookie on the client. Every time you need session data, you decrypt the cookie. The advantage is zero I/O – no database call needed. The disadvantage is that JWTs cannot be instantly invalidated. If you want to revoke a session (user was compromised, employee was terminated), you cannot delete the JWT from the user's browser. You have to wait for it to expire, or maintain a revocation list that you check on each request (which brings back the database lookup).

For Flagline, JWT sessions are the right default because the edge middleware needs to check auth, and the edge runtime cannot easily make database calls. If you need instant session revocation (a feature you might add later for enterprise customers), you can add a lightweight revocation check in server components without abandoning JWTs – check the JWT first (fast, no I/O), and then if the request reaches a server component, verify the user has not been revoked (one fast database query).

How auth is checked at different layers

Think of authentication and authorization as a series of increasingly specific checks, each at a different layer.

Middleware (the gate). The broadest check: "Is there a session at all?" If someone tries to access `/dashboard/anything` without a session cookie, middleware redirects them to `/login` before any page code runs. This is cheap (just checking if a cookie exists and is valid) and catches the most common case (unauthenticated users). It runs at the edge, so the redirect happens with minimal latency.

Layout (the organizational check). The dashboard layout verifies that the session is valid and extracts user data. The project layout checks that the user is a member of the organization that owns the requested project. If not, it renders a 404 (not a 403, because revealing that a resource exists to unauthorized users is an information leak). This runs in the Node.js runtime and can query the database.

Page (the contextual check). Specific pages may have additional requirements. The settings page checks that the user has an admin or owner role. The billing page checks that the user is the organization owner.

Server action (the definitive check). Every server action must re-verify authentication and authorization independently, regardless of what the layout or page already checked. This is because server actions are callable endpoints – a malicious client could call them directly with fabricated data. Never assume that because the page rendered successfully, the server action is being called by an authorized user. The server action is the final line of defense, just like a Laravel controller method should validate authorization even though middleware already ran.

This layered approach means that most unauthorized requests are caught early (at middleware or layout level) and only pay the cost of deeper authorization checks when they actually reach a mutation.

The multi-tenancy model

Flagline's authorization model is hierarchical: a user belongs to one or more organizations, each organization has one or more projects, and each project has multiple environments. A user's role is scoped to the organization level: OWNER, ADMIN, or MEMBER.

The thinking process for protecting a resource is to trace the ownership chain. A flag belongs to an environment, which belongs to a project, which belongs to an organization. To check if a user can toggle a flag, you verify that the user is a member of the organization that owns the project that contains the environment that contains the flag, and that their role is sufficient for the operation (toggling requires ADMIN or OWNER).

In Laravel, you would express this with policies: `$this->authorize('update', $flag)`, and the `FlagPolicy@update` method would trace the ownership chain. In Next.js server actions, you build a similar helper – a function like `requireProjectRole(projectId, 'ADMIN')` that checks the session, queries the membership, and throws if the check fails. The conceptual model is identical; the implementation style is just more explicit.

Session data vs database lookups: the freshness tradeoff

One question that comes up frequently: how much data should you store in the JWT session, and when should you query the database instead?

The JWT session should contain data that is needed on almost every request and changes rarely: user ID, email, name, and their organization memberships with roles. This avoids a database query for basic identity and authorization checks.

The JWT session should NOT contain data that changes frequently or is large: the list of all projects, feature flag configurations, or billing details. These should be fetched from the database when needed, in the specific server component or server action that needs them.

The reasoning is about the staleness window. A JWT is refreshed when the user's session is validated (typically on page loads), but between refreshes, it serves stale data. If you store "user is an ADMIN of organization X" in the JWT and someone changes their role to MEMBER, the JWT will not reflect that change until the next refresh. For role changes (which are rare and significant), this staleness window is acceptable. For billing state (which might change via a webhook at any time), it is not – you should query the database for current billing state rather than trusting the JWT.

Think of the JWT as a cache that you control the invalidation of via the Auth.js session callbacks. Put in it what you want cached. Leave out what needs to be fresh.

> **Coming from Laravel:** The biggest shift is that there is no `Auth` facade that magically knows the current user everywhere. Instead, you call `await auth()` explicitly wherever you need the session. This is more verbose but clearer – you always know exactly where the session is being checked and what is being read from it. There is no hidden global state. The JWT-vs-database question is similar to the decision in Laravel of whether to use `Auth::user()` (which caches the user for the request) or to re-query `User::find($id)` for fresh data.

---

7. Caching Mental Model

Why caching is confusing in Next.js

Caching is where most developers coming from Laravel get tripped up, and for good reason. In Laravel, caching is explicit. You choose to cache something by calling `Cache::remember()`. If you do not call it, nothing is cached. The mental model is straightforward: I decide what to cache, for how long, and when to invalidate it.

Next.js caching is more complex because there are multiple layers, some of which cache things by default. Understanding these layers and knowing when each one applies is essential to building an application that behaves predictably.

The good news is that recent versions of Next.js (15+) have simplified the defaults significantly. `fetch()` requests are no longer cached by default. Pages that use dynamic functions like `cookies()` or `headers()` are automatically treated as dynamic. The framework has moved toward "uncached by default, opt in to caching" rather than the earlier "cached by default, opt out of caching." This makes the mental model closer to Laravel's approach, but you still need to understand the layers because ISR and static generation are powerful tools for marketing pages that have no equivalent in a traditional Laravel deployment.

The key question: how fresh does this data need to be?

Before reaching for any caching mechanism, ask this question about each piece of data your page displays. The answer determines your entire approach.

"It can be hours or days stale." This is your marketing site. The landing page, pricing page, blog posts, documentation. This content changes rarely. Use static generation with periodic revalidation. Set `export const revalidate = 3600` (one hour) on the page, and Next.js will serve a prebuilt HTML page from the CDN. Every hour, the next request triggers a background rebuild. Users always get a fast response because they are hitting the CDN, and the content is at most one revalidation period stale.

"It should be fresh, but a few seconds of staleness is acceptable." This is the Flagline dashboard's flag list, project settings, member list. The data changes when a user performs an action, and after that action, it should update. Use server components with `noStore()` to skip the server-side cache and always query the database directly. Alternatively, use `unstable_cache` with cache tags and invalidate the tags when a mutation happens. The flags list shows fresh data because the server action that toggles a flag calls `revalidateTag('project:abc:flags')` after the mutation.

"It must be real-time." This is flag evaluation for customer SDKs. A developer toggles a flag in the dashboard, and within seconds, the change should be reflected in their application. This is not a Next.js caching concern at all – it is handled by the evaluation API's Redis cache and the pub/sub invalidation channel. Next.js is not in this path.

The layers, and when each applies

Request memoization. If your page makes the same `fetch()` call multiple times during a single render (because multiple components need the same data), Next.js deduplicates those calls automatically. You do not need to think about this unless you are confused about why a data function is only called once despite appearing in multiple components. This is per-request and has no persistence between requests. Think of it as automatic `Cache::remember()` that expires when the request ends.

The data cache. This is the server-side cache for `fetch()` results and `unstable_cache` results. It persists across requests. If you call `fetch('https://api.stripe.com/v1/products', {`

`next: { revalidate: 3600 } })`, the result is cached for an hour. This is the closest equivalent to Laravel's `Cache::remember('stripe-products', 3600, fn() => ...)`.

For Prisma queries, there is no automatic data caching because Prisma does not use `fetch()`. If you want to cache a Prisma query result, you wrap it in `unstable_cache()` with a cache key and tags. If you do not wrap it, the query runs fresh every time. This is an important distinction: Prisma queries are uncached by default. This is actually what you want for dashboard pages where freshness matters.

The full route cache. When Next.js builds your application, it pre-renders static pages into HTML. These pre-rendered pages are served from the CDN without invoking any server-side code. This only applies to pages that have no dynamic data dependencies. Your marketing pages (if they do not fetch from a database) are statically generated. Your dashboard pages, which depend on session data and database queries, are not – they are rendered dynamically on every request.

If a page uses `export const revalidate = 3600`, it gets ISR (Incremental Static Regeneration): it is pre-rendered at build time, served from cache, and regenerated in the background at the specified interval. This is perfect for pages like blog posts or documentation that change occasionally but do not need to be real-time.

The client router cache. When a user navigates between pages in the browser, Next.js caches the RSC (React Server Component) payload of visited pages in the browser's memory. This is what makes back/forward navigation instant – the framework does not need to re-fetch the page from the server. This cache has a short TTL (30 seconds for dynamic pages, 5 minutes for static pages in recent versions) and is cleared on navigation in certain cases.

You generally do not need to think about this cache unless a user performs a mutation and the UI does not update. In that case, `router.refresh()` forces the client to re-fetch the current page's server component data.

The Flagline caching strategy

For Flagline specifically, the approach is:

Marketing pages use ISR. Set a revalidation interval and forget about them. If you update pricing, call `revalidateTag('pricing')` from the admin action. The next visitor sees the updated page.

Dashboard pages opt out of caching entirely. Call `noStore()` at the top of each dashboard page, or set `export const dynamic = 'force-dynamic'` in the dashboard layout. These pages always render fresh from the database. The overhead of one or two Prisma queries per page load is negligible for a dashboard that handles tens of requests per second.

The evaluation API handles its own caching via Redis, entirely outside of Next.js. Flag configurations are cached in Redis with pub/sub-based invalidation. This has nothing to do with Next.js caching layers.

The revalidation mental model: tags vs paths

When a mutation happens and cached data is stale, you have two invalidation strategies. Understanding when to use each saves you from debugging mysterious stale-data issues.

Path-based revalidation (`revalidatePath('/dashboard/my-app/production/flags')`) says: “Everything on this exact page is stale. Re-render it from scratch on the next request.” This is the nuclear option for a specific URL. It is simple to understand and easy to get right. The downside is that it is URL-specific – if the same data appears on multiple pages, you need to invalidate each path individually.

Tag-based revalidation (`revalidateTag('project:abc:flags')`) says: “Every piece of cached data tagged with this label is stale.” This is more surgical and more powerful. If multiple pages display data tagged with `project:abc:flags` – the flags list page, the project overview page, a summary widget in the dashboard – all of them are invalidated with a single call. The downside is that you must set up the tagging scheme consistently. Every `unstable_cache` call and every `fetch` call that should be tag-invalidatable needs the `tags` option.

For Flagline, a good mental framework is: use path revalidation when you know exactly which page was affected, and use tag revalidation when the change could affect multiple views. Creating a flag affects the flags list page (path revalidation works) but also any summary counts or overview widgets (tag revalidation is better). Toggling a flag’s enabled state might affect the flags list page and the flag detail panel (tag revalidation handles both).

A practical tagging convention is to namespace tags by entity and scope: `project:{id}:flags`, `project:{id}:members`, `env:{id}:audit-log`, `flag:{id}:detail`. When you mutate a flag, invalidate `project:{projectId}:flags` and `flag:{flagId}:detail`. This is analogous to Laravel’s cache tags but applied to rendered pages and data fetches rather than arbitrary cache entries.

When to opt out entirely

There is a temptation to cache everything for performance. Resist it for dashboard pages. The complexity of managing cache invalidation (making sure every mutation invalidates every affected cache entry) is rarely worth the performance gain for a dashboard application. Your users are authenticated humans making infrequent requests, not bots hammering your server. A direct Prisma query takes 5-50ms depending on complexity. That is fast enough.

Cache aggressively for public-facing pages where every millisecond of response time matters and the data does not change often. Cache not at all for authenticated dashboard pages where data freshness matters more than speed.

The most common caching mistake in a Next.js application is not over-caching or under-caching – it is inconsistent caching. If some dashboard pages are cached and others are not, and your invalidation logic covers some mutations but not others, you get a dashboard where some pages show stale data unpredictably. It is better to either commit to a full caching strategy with disciplined invalidation, or to opt out entirely for all dashboard pages and only cache the public-facing content.

For Flagline, the recommendation is to start with no caching for the dashboard (`dynamic = 'force-dynamic'` in the dashboard layout), cache marketing pages with ISR, and add dashboard caching later only if you have a measured performance problem. Premature caching optimization is a worse trap in Next.js than in most frameworks because the invalidation surface area is larger.

> **Coming from Laravel:** In Laravel, caching is opt-in and explicit. In Next.js, some caching happens by default (route cache for static pages, fetch cache). The most important thing to learn is how to opt OUT of caching for dynamic pages. `noStore()` and

`dynamic = 'force-dynamic'` are your tools for this. Think of them as the equivalent of never calling `Cache::remember()` in your Laravel controllers – just querying the database directly every time. For a dashboard, that is the right default.

---

8. Deployment Thinking

The Vercel model vs the Forge model

If you have deployed Laravel applications on Forge (or Envoyer, or manually via SSH), you are used to a deployment that looks like: push to `main`, CI runs, artifacts are built, code is copied to a server, `php artisan migrate` runs, `php artisan config:cache` runs, the queue is restarted, and the new version is live.

Vercel's deployment model is fundamentally different in a way that takes some getting used to.

When you push to `main`, Vercel builds your Next.js application. During the build, it analyzes every page to determine whether it is static (can be pre-rendered at build time) or dynamic (must be rendered per-request). Static pages become HTML files on a CDN. Dynamic pages become serverless functions that are deployed to Vercel's global edge network.

There is no single server running your application. There are potentially hundreds of serverless function instances across the globe, spun up on demand to handle requests. When no one is making requests, there are zero instances running. When traffic spikes, new instances are created automatically.

This is the same model as Laravel Vapor (which runs Laravel on AWS Lambda), so if you have used Vapor, the concepts transfer directly. If you have not, the key implications are:

Cold starts. The first request to a serverless function after a period of inactivity incurs a startup cost (typically 100-500ms). Subsequent requests are fast because the function instance stays "warm" for a few minutes. For a dashboard, this is mostly irrelevant because the function stays warm as long as you have active users. For the marketing site, it means the first visitor after a quiet period might experience slightly slower page loads.

No persistent state. You cannot store anything in memory between requests (well, you can, but there is no guarantee the same function instance handles the next request). This is why session data lives in cookies (JWTs) or a database, not in server memory. This is why the evaluation API is separate – it needs persistent in-memory state (connection pools, SSE streams) that serverless functions cannot maintain.

Automatic scaling. You never think about "how many servers do I need?" or "should I increase the instance count?" Vercel handles this. If your marketing site goes viral and gets 100,000 visitors in an hour, the infrastructure scales automatically.

Environment variable scoping

Vercel allows you to set different values for each environment: Production, Preview, and Development. This is enormously useful.

Production environment variables point to your production database, production Stripe keys, and production third-party services. Preview environment variables point to a staging database, test Stripe keys, and sandbox services.

This means every pull request gets a preview deployment with its own URL (`flagline-git-feature-xyz.vercel.app`) that connects to the staging database and uses test Stripe keys. A reviewer can click the preview link, test the feature, and see it working with real (but non-production) data. This is like having Forge create a new server with a staging database for every PR – except it is automatic and costs nothing extra.

The mental model for environment variables in Next.js has one critical rule: any variable that needs to be accessible in client-side JavaScript must be prefixed with `NEXT_PUBLIC_`. Variables without this prefix are only available in server-side code (server components, server actions, route handlers, middleware). This is a security feature enforced at the build level. In Laravel, you access all environment variables everywhere via `env()`. In Next.js, the build system literally does not include non-prefixed variables in the client bundle. If you forget the prefix, the variable is `undefined` in the browser. If you add the prefix to a secret, it is exposed to the browser.

Preview deployments as a workflow tool

This is one area where Vercel's developer experience significantly exceeds traditional Laravel deployment. Every pull request automatically gets a deployment. This means:

Your QA process changes. Instead of "merge to staging, deploy to staging, test on staging," you test on the PR's preview deployment directly. Each PR is isolated – PR A's preview deployment is independent from PR B's.

Your demo process changes. Need to show a stakeholder a feature in progress? Share the preview URL. No staging server contention.

Your rollback process changes. If a production deployment causes issues, you redeploy the previous commit. Vercel keeps the build artifacts for recent deployments, so rollbacks are near-instant.

How this compares to deploying the full Flagline stack

Vercel handles the Next.js dashboard application. But Flagline has other components that Vercel does not manage: the Fastify evaluation API, the PostgreSQL database, and Redis.

The evaluation API deploys to a platform like Fly.io or Railway, which provide persistent servers (not serverless) with global distribution. The deployment is more traditional: push to a branch, CI builds a Docker image, the image is deployed to the server, health checks pass, traffic is routed to the new instance.

PostgreSQL runs on a managed service like Neon (which has a serverless/branching model that pairs well with Vercel's preview deployments) or Supabase or RDS. Database migrations run as part of the build or via a CI step, not via the deployment platform.

Redis runs on Upstash (for edge compatibility in middleware) or a managed Redis service like Redis Cloud.

The full deployment picture for a Flagline release is: push to `main`, Vercel builds and deploys the dashboard, CI builds and deploys the evaluation API to Fly.io, and if there are database migrations, they run as a CI step before the application deployments. This is more complex than a single `forge deploy` command, but each piece scales independently and fails independently.

The build step: what happens and why it matters

When Vercel builds a Next.js application, the build process does more than just compile TypeScript. It analyzes every route to determine its rendering strategy.

Pages that have no dynamic data dependencies (no `cookies()`, no `headers()`, no uncached `fetch`, no direct database calls) are pre-rendered into static HTML at build time. These pages are served from the CDN with no server-side processing. Your marketing landing page, if it uses no dynamic data, becomes a static HTML file that loads instantly from the nearest CDN edge.

Pages that depend on request-time data (they read cookies for session, or they call `noStore()`, or they use `dynamic = 'force-dynamic'`) become serverless functions. Each request to these pages invokes a function that renders the page on demand.

This analysis happens automatically. You do not declare which pages are static and which are dynamic – the framework infers it from your code. But you need to be aware of it because an accidental dependency on request-time data (like reading a cookie in a marketing page layout) can convert a static page into a dynamic one, losing the CDN performance benefit.

The build output tells you which pages were statically generated and which are dynamic. Pay attention to it. If your marketing pages show up as dynamic and you expected them to be static, trace through the component tree to find what is pulling in request-time data.

Thinking about database migrations in deployment

In Laravel with Forge, you often run `php artisan migrate` as part of your deploy script. The deploy process is: pull code, install dependencies, run migrations, clear caches, restart workers.

With Vercel, there is no server to SSH into and run migrations on. Database migrations need to run separately from the application deployment. The common approach is to run migrations in a CI/CD step (GitHub Actions) before the Vercel deployment triggers. This ensures the database schema is updated before the new application code that depends on it goes live.

This ordering matters. If you deploy application code that references a new database column before the migration runs, you get runtime errors. In Laravel on Forge, the deploy script runs sequentially, so you naturally get the right order. With Vercel, you need to make this ordering explicit in your CI/CD pipeline.

For Flagline, the pattern is: GitHub Actions runs `prisma migrate deploy` against the production database, waits for success, and then either Vercel picks up the deployment automatically (if triggered by the same push) or the CI pipeline triggers it. For preview deployments, you either run migrations against a staging database or use Neon's database branching feature, which creates an isolated database branch for each PR.

> **Coming from Laravel:** Vercel is to Next.js what Vapor is to Laravel – a serverless deployment platform that abstracts away servers. The key difference is that Vercel is the default, expected deployment target for Next.js, whereas Vapor is an alternative to Forge. Vercel's integration with GitHub (automatic preview deployments, environment variable scoping) makes the developer experience smoother than anything in the

Laravel ecosystem, at the cost of being locked into their platform for the serverless deployment model. The migration story is the one area where the Laravel/Forge experience is simpler – having migrations as a built-in deploy step rather than a separate CI concern is a genuine convenience that you give up in the serverless world.

Bringing It All Together

The overarching mental model for building Flagline with Next.js is this: the framework does more for you than Laravel does, but in exchange, it requires you to think differently about where code runs.

In Laravel, all your PHP code runs in one place: the server. The question "where does this code execute?" never arises. In Next.js, code can run in four different environments: the build process (static generation), the edge runtime (middleware, edge functions), the Node.js server runtime (server components, server actions, route handlers), and the browser (client components). Every file you write, every function you define, runs in one of these environments, and the capabilities available in each differ.

The App Router's file-based routing means your directory structure is your architecture. It is not just organization – it is load-bearing. How you nest directories determines your layout hierarchy, your code-splitting boundaries, and your caching behavior.

Server components as the default means you think about what needs to be on the client, not what needs to be on the server. This is an inversion from the traditional React model and from most frontend frameworks. It is closer to how you think in Laravel: server-rendered by default, with islands of interactivity.

Server actions mean most mutations do not need API endpoints. The form-to-function pipeline eliminates an entire category of boilerplate. But you must still validate, authenticate, and authorize in every action – the directness of the call does not absolve you of security responsibilities.

The caching system means you need to think about data freshness at every level. Not in the Laravel sense of "should I cache this?" but in the Next.js sense of "is this being cached without me asking for it, and should I opt out?"

And the deployment model means your application is not one server but a distributed system of serverless functions, edge functions, and CDN-cached static assets. This is more powerful and more complex than a single Laravel application on a single server.

The good news is that once you internalize these mental models, building with Next.js is remarkably productive. The framework handles a tremendous amount of complexity – routing, code splitting, streaming, caching, deployment – that you would otherwise have to solve yourself or through third-party packages. The trick is understanding what it is doing and why, so you can work with it rather than against it.

The adjustment period

If you have been writing Laravel for five or more years, expect an adjustment period of two to four weeks before the Next.js way of thinking feels natural. The concepts that will click first are routing (file-based is intuitive once you see it) and server actions (they map cleanly to the controller pattern).

The concepts that take longer are the server/client component boundary (it requires rethinking how you compose UI) and the caching model (it requires understanding layers that Laravel does not have).

The single best way to accelerate the transition is to build something. Not to read more documentation, not to watch more tutorials, but to sit down with the Flagline codebase and build a feature from scratch: add a new page, fetch data, handle a form submission, protect a route, deal with caching. The mental models described in this guide will crystallize through application, not through memorization.

When you get stuck – and you will get stuck, particularly on caching behavior and the serialization boundary – come back to the core questions. “Where does this code run?” “Does this component need interactivity?” “How fresh does this data need to be?” “Where should this request go?” These questions, applied consistently, will guide you to the right architectural decision in almost every case.

This document is part of the Flagline engineering reference. See also: `05-database-prisma-reference.md`, `07-stripe-billing-reference.md`, `08-architecture-monorepo-reference.md`.