Contents

06 — SDK Design & npm Publishing: A Thinking Guide

Flagline — Feature Flag SaaS Stack: React, Next.js 14+ (App Router), Node.js, TypeScript, PostgreSQL (Prisma), Redis Audience: Backend developer with 5+ years PHP/Laravel experience transitioning to this stack

Note: This document is not a code reference. It is a thinking guide — mental models, rationale, decision frameworks, and reasoning that will help you design and build the Flagline SDK yourself. If you want implementation details, you will write them after internalizing the principles here.

Table of Contents

---

1. SDK Design Philosophy

The SDK is the face of your product

Here is the thing that is easy to miss when you are building a SaaS: most of your customers will never see your dashboard. They will see it once during setup, maybe again when they need to change a targeting rule. But they will interact with your SDK every single day. It is in their codebase. It is in their editor. It is in their build pipeline, their bundle, their production runtime. The SDK is not a secondary artifact of your product. For the developer who integrates Flagline, the SDK *is* your product.

This changes how you think about everything. A clunky dashboard annoys people. A clunky SDK makes them rip out your service and switch to a competitor. The switching cost for a dashboard is high — you would have to recreate all your flags, targeting rules, and team permissions. The switching cost for a bad SDK is low — swap one import for another, the flag keys stay the same, and you are on LaunchDarkly by lunch.

So the first mental model: treat your SDK like a product with its own design principles, its own user experience standards, and its own quality bar. Every public method name, every error message, every TypeScript type is a design decision that affects whether someone keeps paying you.

Coming from Laravel

If you have published a Composer package, you already understand some of this instinctively. You know the difference between a package where you read the README once and everything works, and one where you spend an hour debugging autoload issues or figuring out which service provider to register. The npm ecosystem amplifies this dynamic because JavaScript packages tend to be smaller and more numerous, which means developers are quicker to try alternatives. A PHP developer might tolerate a rough edge on a Composer package because there are only two options for what they need. A JavaScript developer has twelve options and will switch after fifteen minutes of frustration.

The bar for developer experience is higher in the JavaScript ecosystem, and the tolerance for friction is lower. You should treat this as a design constraint, not an annoyance.

The 5-minute test

Before you write a single line of SDK code, write the README first. Not the whole README — just the quick start section. Write out exactly what a developer would type to go from zero to evaluating their first flag. Then time how long it takes to read and execute those steps. If it takes more than five minutes, your SDK is too complicated.

This is not an arbitrary number. Five minutes is roughly the threshold where a developer evaluating your product transitions from “trying it out” to “debugging it.” You want them to see a flag value appear before they hit that threshold. Every additional concept they have to understand before that moment is friction. Every configuration option that is not strictly necessary for a hello-world is friction. Every import beyond the bare minimum is friction.

Think about what Laravel got right with its first-run experience: `composer create-project`, edit `.env`, `php artisan migrate`, and you have a working application. The Flagline SDK should feel the same way. Install, initialize with a key, read a flag. Three steps. No intermediate concepts. No configuration files. No environment-specific setup for the hello-world case.

The practical implication is that your SDK needs sensible defaults for everything. The base URL should default to production. Streaming should default to on. Timeouts should default to reasonable values. The only thing the developer *must* provide is their SDK key and (optionally) a user context. Everything else should work out of the box.

API surface design: small, obvious, hard to misuse

The best APIs are boring. They do exactly what you expect, the method names tell you what they do, and there is only one obvious way to accomplish any given task. This is sometimes called the "principle of least surprise," and it is the single most important design heuristic for a public API.

When you are designing your SDK's public surface, apply these questions to every method, every parameter, every option:

Can I remove this? The most reliable code is code that does not exist. Every method you expose is a method you have to support, document, and avoid breaking in future versions. Start with the absolute minimum surface and add methods only when real users ask for them. It is always possible to add a public method later. It is extremely painful to remove one, because removing a public method is a breaking change that forces a major version bump and breaks every consumer who used it.

Is there only one way to do this? If a developer can accomplish the same thing two different ways through your API, one of them is wrong and you should remove it. Two ways to do the same thing means two code paths to test, two sets of documentation to maintain, and confused users who are not sure which approach is "correct." If you find yourself writing a "best practices" section that recommends one approach over another, that is a signal that you have too many approaches.

What happens if the developer passes the wrong thing? A well-designed API makes it hard to misuse. TypeScript helps enormously here — wrong types get caught at compile time. But think about runtime behavior too. If someone passes `null` where you expect a string, does the SDK blow up with an unhelpful stack trace, or does it degrade gracefully with a meaningful console warning? The answer should always be the latter. Your SDK is a guest in someone else's application. Guests do not throw tantrums.

Would I understand this if I saw it in a code review with no context? This is the readability test. When your SDK call appears in the middle of someone else's application code, it should be immediately clear what it does. `client.getFlag('dark-mode', false)` reads like English — get the flag called "dark-mode," defaulting to false. There is zero ambiguity. Compare that with something like `client.ev('dark-mode', { d: false, t: 'bool' })` — same functionality, completely opaque. Full words. Obvious parameter names. No cleverness.

Zero dependencies

Your SDK should have zero runtime dependencies. This is a strong position, and it is worth understanding the reasoning thoroughly because you will be tempted to pull in a library at some point.

Every dependency you add to your SDK becomes a dependency of every application that uses your SDK. If you depend on `axios`, every consumer now has `axios` in their bundle, whether they want it or not. If you depend on `axios@1.6` and one of your consumer's other dependencies needs `axios@1.4`, now they have a version conflict to resolve. They did not ask for this conflict. They just wanted to check a feature flag.

This is not hypothetical. Dependency conflicts are the most common source of pain in the npm ecosystem. It is so common that established SDKs treat zero dependencies as a hard requirement, not a nice-to-have. Stripe's JS SDK has zero dependencies. LaunchDarkly's client SDK has zero dependencies. Sentry's core package has zero dependencies. PostHog's JS library has

zero dependencies. When you see this pattern across every major developer tools company, pay attention.

The reasoning goes beyond compatibility. There is a supply chain security argument — every dependency is a potential attack vector (the `event-stream` incident taught the ecosystem that lesson). There is a bundle size argument — especially for client-side SDKs where every kilobyte ships to your users' browsers over the network. And there is a maintenance argument — every dependency you take on is a dependency you have to keep updated, monitor for vulnerabilities, and test against new major versions.

For Flagline specifically, the only things the core SDK needs to do over the network are a single HTTP POST request and an SSE connection. Both of these are available natively in every modern JavaScript runtime via `fetch` and `EventSource`. The `fetch` API is available in all browsers, Node.js 18+, Deno, Cloudflare Workers, and every other runtime you would realistically target. There is no reason to pull in `axios`, `node-fetch`, or any HTTP library. For SSE, the browser-native `EventSource` handles connection management, reconnection, and event parsing. The only case where `EventSource` is unavailable is some edge runtimes, and for those you fall back to polling, which is just `fetch` on a timer.

You may also be tempted to pull in an event emitter library. Do not. A minimal event emitter is about 30 lines of code. You do not need `eventemitter3` or Node's built-in `events` module (which does not exist in browser runtimes anyway). Write the 30 lines. They will never break, they will never have a CVE, and they will never conflict with a consumer's dependency tree.

> **Coming from Composer:** In the PHP world, it is common for packages to depend on `guzzlehttp/guzzle` or `illuminate/support`. The tolerance for dependencies is higher in PHP because packages run server-side only and there is no bundle size concern. In JavaScript, especially for a client-side SDK, every dependency is a cost your users pay in their production bundles and a risk in their supply chain. The culture is different — lean packages are expected, not just preferred.

---

2. The Two-Package Architecture

Why not one package?

The instinct when you are building your first SDK is to make one package that does everything. One `npm install`, one import, one set of docs. The simplicity is appealing. But this is a trap, and understanding why will change how you think about SDK architecture for any product.

Flagline's SDK is consumed in fundamentally different environments:

- A React single-page application running in a browser
- A Next.js application with both server and client components
- A Node.js backend service that checks flags in API route handlers
- A Cloudflare Worker or Vercel Edge Function that runs in a V8 isolate
- A React Native mobile app
- A plain JavaScript frontend with no framework at all

These environments share a common need — evaluate feature flags — but have very different capabilities and constraints. A React app has `useState` and `useEffect` and re-renders when

state changes. A Node.js server has none of that. An edge worker might not have `EventSource`. A React Native app has a different lifecycle than a web app.

If you put everything in one package, one of two things happens. Either you force every consumer to install React as a dependency (meaning a Node.js backend developer has to install React even though they have no UI), or you make React optional and litter your code with conditional imports and runtime environment checks. The first option is wasteful and confusing. The second is fragile and creates a testing nightmare — you are now responsible for ensuring your code works correctly in every possible combination of "React present" and "React absent" across every environment.

The split: @flagline/js and @flagline/react

The solution is to separate the universally-needed logic from the framework-specific integrations. This is the same pattern used by every mature SDK in the ecosystem, and for good reason.

`@flagline/js` is the core. It works everywhere JavaScript runs. It handles initialization, flag evaluation, caching, SSE connections, user identification, and event subscriptions. It has zero dependencies and no opinion about your UI framework. A Node.js developer can use it directly. A vanilla JavaScript developer can use it directly. A Svelte developer, a Vue developer, an Angular developer — they all use the core and write their own thin wrappers if they want framework-specific niceties.

`@flagline/react` is a thin wrapper around the core. It provides a React context provider and hooks that connect the core SDK to React's rendering lifecycle. It depends on `@flagline/js` as a runtime dependency and on `react` as a peer dependency. Its entire job is to make the core SDK feel native in a React application.

"Thin wrapper" is not a vague aspiration here — it is a measurable constraint. The React package should contain almost no logic of its own. The provider initializes the core client and puts it in context. The `useFlag` hook reads from the core client and subscribes to changes. The `useFlags` hook does the same for all flags. That is essentially the entire package. If you find yourself putting evaluation logic, network logic, caching logic, or retry logic in the React package, something has gone wrong. That logic belongs in the core. The React package is plumbing between two systems (your SDK's event model and React's state model), not a second SDK.

A good litmus test: if you deleted the React package entirely and rewrote it from scratch, it should take less than a day. If it would take longer, too much logic has leaked into the framework layer.

How to think about the dependency graph

Picture it as a pyramid. At the bottom is `@flagline/js`, which depends on nothing. On top of it sits `@flagline/react`, which depends on the core. If you later build a Vue wrapper, it would be `@flagline/vue`, also depending on the core. Each framework wrapper is a peer of the others — they never depend on one another — all rooted in the same core.

This architecture gives you three important properties.

First, changes to evaluation logic, caching, or network transport happen in one place (the core) and automatically benefit every framework wrapper. You do not fix the same bug in three packages. You do not implement the same feature three times. The core is the single source of truth for SDK behavior.

Second, consumers only install what they need. A Node.js backend installs one small package. A React frontend installs two. No one is penalized with code they do not use or dependencies they do not want.

Third, you can version and release the packages independently. A bug fix in the React hooks does not require a new release of the core. A new feature in the core does not require updating the React wrapper unless the new feature needs UI-level integration. This reduces release coordination overhead and lets you ship fixes faster.

Coming from Composer: the analogy

This is similar to how Symfony components work in the PHP world. `symfony/http-foundation` is the core that handles HTTP messages. `symfony/http-kernel` builds on it to handle request/response cycles. `laravel/framework` wraps Symfony components with Laravel-specific features like facades and Eloquent. Each layer adds framework opinion on top of framework-agnostic infrastructure.

Or think of `league/flysystem`. The core package defines the filesystem interface and common logic. Then there are adapter packages: `league/flysystem-aws-s3-v3` for S3, `league/flysystem-local` for local disk, `league/flysystem-ftp` for FTP. Each adapter depends on the core but not on the other adapters. Your Flagline React package is conceptually the same thing — it adapts the core SDK to a specific environment (React's rendering model).

Real-world precedents

This is not a pattern Flagline is inventing. It is the industry standard for developer-facing SDKs:

Stripe ships `@stripe/stripe-js` (core, loads Stripe.js) and `@stripe/react-stripe-js` (React Elements wrapper). The React package is a thin layer of hooks and components around the core Stripe.js functionality.

Sentry ships `@sentry/browser` (core browser SDK), `@sentry/react` (React-specific error boundaries, hooks, and profiler integration), and `@sentry/node` (Node.js-specific transport and integrations). All are built on `@sentry/core`, which contains the shared logic.

LaunchDarkly (the most direct competitor to Flagline) ships `launchdarkly-js-client-sdk` (core JavaScript client) and `launchdarkly-react-client-sdk` (React wrapper with hooks and a provider). The React wrapper is thin and delegates all evaluation to the core.

PostHog ships `posthog-js` (core analytics/feature-flags) plus separate wrappers for React, Next.js, and other frameworks.

When you see the same architecture across every major developer tools company, it is not a coincidence. It is a converged solution to a common problem, arrived at independently by teams with different constraints. That convergence is a strong signal that the pattern is correct.

---

3. Public API Design Thinking

The initialize pattern: factory function over constructor

When a developer first uses your SDK, they need to initialize it. There are two common approaches: a class constructor (`new FlaglineClient(config)`) or a factory function (`initialize(config)`). Flagline uses a factory function, and the reasoning is worth understanding deeply because it affects the entire developer experience.

Initialization is inherently asynchronous. The SDK needs to fetch the current flag values from the server before it can evaluate anything meaningfully. A constructor in JavaScript is always synchronous — you cannot `await` inside a constructor. If you use a constructor, the developer creates the client and then has to call a separate `.init()` method, or listen for a `ready` event, or check an `.isReady()` boolean before they can safely use it. This two-step dance is a constant source of bugs. Developers forget the second step, or they call `getFlag` before initialization completes and wonder why they always get default values.

A factory function can be async. `const client = await initialize(config)` does all the setup work — validate config, fetch flags, open the SSE connection — and returns a fully-ready client. The developer `await`s the initialization and gets back an object that is immediately usable. No two-step dance. No "is it ready yet" checks. No race conditions between initialization and flag reads. The `await` itself is the readiness gate.

There is a subtler design benefit too. A factory function gives you control over what is returned. You can return a carefully shaped public API object that exposes only the methods you want consumers to use. With a class, every public method on the class is part of your API surface, and developers with TypeScript's structural typing might reach into properties or methods you considered internal. A factory function lets you maintain a clean separation between public and private without relying on JavaScript's `#private` fields or naming conventions like underscores.

Note that the factory function can still use a class internally. The class manages state and organizes the implementation. The factory creates the class instance, calls its init method, and returns it. Consumers interact with the class's public methods, but they create instances through the factory. This gives you the organizational benefits of a class with the ergonomic benefits of a factory.

> **Coming from PHP:** In Laravel packages, you often see a service provider that binds a singleton into the container, and then a facade that provides static access. `LaunchDarkly::getFlag(...)` is a facade over a client that was initialized in the service provider. The mental model is similar: the factory/service-provider handles the complex setup and gives you back a ready-to-use interface. The difference is that in JavaScript, the asynchronous nature of initialization makes the factory pattern not just convenient but practically necessary.

Why getFlag returns synchronously

This is one of the most important design decisions in the entire SDK, and the one that has the biggest impact on developer experience. It is also the decision that will feel the most counterintuitive if you are coming from a world where "get data" usually means "make a network request."

When a developer checks a flag, they call `client.getFlag('dark-mode', false)`. This call returns the flag value immediately, synchronously, from the in-memory cache. It does not make a

network request. It does not return a Promise. It does not block.

Understanding why this matters requires thinking about how the SDK is used in practice. Flag checks happen everywhere in application code — in render functions, in event handlers, in middleware, in utility functions, in computed properties, in template expressions. If `getFlag` were async, every single call site would need `await`, every function that calls it would need to be `async`, and in React components, you would need loading states and Suspense boundaries for every individual flag check. The cascade effect is brutal. One async call at a leaf component forces the entire ancestor chain to handle asynchrony.

Imagine a React component that checks three flags: one for a feature toggle, one for a copy variant, one for a layout option. If `getFlag` were async, you need three `await`s, or a `Promise.all`, or three separate `useEffect` calls with loading states. The component goes from three lines of flag logic to fifteen lines of async plumbing. Multiply that across every component in the application that reads a flag, and you have a codebase drowning in incidental complexity.

The cache-first approach avoids all of this. The SDK fetches all flag values during initialization (the one async moment, gated by the `await initialize(...)` call). After that, flag reads are pure in-memory lookups — a Map get, nothing more. When flag values change on the server, the SSE connection delivers updates and the SDK updates its in-memory cache in the background. The next time `getFlag` is called, it returns the new value.

This means there is a brief window where a flag value might be stale — between the moment a flag changes on the server and the moment the SSE event arrives (typically tens to hundreds of milliseconds depending on network conditions). For a feature flag system, this is a completely acceptable tradeoff. Feature flags are not financial transactions. A user seeing the old version of a button for an extra 200 milliseconds is a non-issue. A user seeing a loading spinner every time the application checks a flag is a real UX problem.

The second argument to `getFlag` — the default value — serves as the fallback if the flag has never been fetched, if the SDK is still initializing, or if the network is completely down. This guarantees that the call always returns something useful. The application never crashes because a flag is missing. It never throws because the server is unreachable. It gets the developer-specified default and continues running.

Think about what this means for the consuming developer's mental model. They never write `try/catch` around a flag check. They never handle a Promise rejection from flag evaluation. They never display a loading state while waiting for a flag value. They write `if (client.getFlag('dark-mode', false))` and move on. That simplicity is the entire point of the design.

The identify pattern

Feature flags without targeting are just config values stored in a database instead of a file. The real power of a feature flag system is targeting: showing a feature to 10% of users, or only to users on the Pro plan, or only to users in Europe, or only to users who signed up after a certain date. To evaluate targeting rules, the SDK needs to know who the current user is.

The `identify` method tells the SDK about the current user context. It accepts an object with at least a user ID, plus any custom attributes the developer wants to use for targeting — plan tier, country, creation date, company size, whatever their targeting rules reference.

The key design question is: when does identity change, and what should happen when it does?

In a typical web application, identity changes at three moments: login (anonymous user becomes a known user), logout (known user becomes anonymous), and occasionally mid-session (a user upgrades their plan, changes an account setting, or an admin impersonates a user). Each identity change potentially affects which flags are enabled for that user, because targeting rules evaluate against user attributes.

So when `identify` is called, the SDK needs to re-fetch flag values from the server for the new user context. This makes `identify` an async operation — it triggers a network request, waits for the response, and updates the in-memory cache. The key design decision here is what happens to flag reads *during* that re-fetch.

The approach that causes the least surprise is: the previous flag values remain in cache and continue to be returned by `getFlag` until the new values arrive. This avoids a flash where all flags briefly revert to defaults during the transition. From the consuming application's perspective, flag values smoothly transition from the old user's values to the new user's values with no intermediate "unknown" state.

Logout is its own operation — typically `client.reset()` — which clears the user context and reverts to anonymous flag evaluation. Naming it `reset` rather than `logout` is a deliberate choice: the SDK is not an auth library and should not use auth terminology. It knows about user context, and resetting that context is the SDK-level concept. Your auth library handles logout; the SDK handles resetting its targeting context.

One subtle design consideration: what if `identify` is called rapidly? A user typing in a form field, an attribute that changes frequently, a component that re-renders with new props. You do not want to fire a network request on every keystroke. This is where debouncing comes in — the SDK should debounce `identify` calls internally so that rapid successive calls result in a single network request with the final user context. The debounce window should be short (200ms is reasonable) to avoid noticeable delays while preventing request floods.

The subscribe pattern

Some consuming applications need to react when a flag changes in real time. A dashboard might rearrange its layout when an admin toggles a feature. A marketing page might switch copy variants when an A/B test is updated. A settings panel might show or hide options based on the user's current plan.

The SDK provides a subscribe mechanism: call a method with a flag key and a callback, and the callback fires whenever that flag's value changes. The method returns an unsubscribe function — a common JavaScript pattern — that the consumer calls when they no longer want updates.

This is the low-level primitive that the React hooks build on. The `useFlag` hook internally subscribes on mount and unsubscribes on unmount. But the subscribe method is part of the core SDK because non-React consumers need it too. A Svelte developer might use it in a store. A Vue developer might use it in a watcher. A vanilla JS developer might use it to update the DOM directly.

The unsubscribe-function-as-return-value pattern (rather than, say, passing a subscription ID to a separate `unsubscribe` method) is idiomatic JavaScript and mirrors patterns in RxJS, MobX, Redux, and the DOM's own `addEventListener`/`removeEventListener` (though the DOM uses the latter style, the cleanup-function pattern has become the more popular convention in modern

JS). It is concise, impossible to leak (the unsubscribe function captures everything it needs via closure), and works naturally with React's `useEffect` cleanup.

The destroy/cleanup pattern

The SDK opens a persistent SSE connection. It might have pending timers for polling fallback. It holds references to subscriber callbacks and to the in-memory cache. If the consuming application does not clean up these resources, they leak.

In a long-lived Node.js server, this is less of an issue — the process runs until you stop it, and a single SSE connection is fine for the lifetime of the process. In a single-page application, it matters when the app unmounts or when you are running tests that create and destroy many SDK instances. In server-side rendering, it is critical — each incoming request might create a new SDK context, and if those contexts are not destroyed, you accumulate SSE connections until you run out of file descriptors and the process crashes.

The `destroy` method closes the SSE connection, clears all timers, removes all subscribers, and releases all internal references. After calling `destroy`, the client should be inert — calling any method on it either does nothing or returns the default value, rather than throwing an error. This is important: if a component calls `getFlag` after the SDK has been destroyed (a timing issue that happens in complex React trees during unmount), it should get a safe default, not an exception.

> **Coming from PHP:** PHP's request lifecycle handles cleanup automatically — everything is garbage collected when the request ends. There is no persistent SSE connection in a PHP process (each request is a fresh process or thread). JavaScript's long-running processes do not have this luxury. Think of `destroy()` as the equivalent of explicitly closing a database connection in a PHP daemon process or a Laravel queue worker. In a web request, PHP handles it for you. In a long-lived process, you must handle it yourself. The SDK is always running in a "long-lived process" context (the browser tab, the Node.js server), so cleanup is always your responsibility.

---

4. Internal Architecture Thinking

The initialization flow

When a developer calls the `initialize` function, a specific sequence of events happens inside the SDK. Understanding this sequence — and why each step is ordered the way it is — helps you reason about error states, timing edge cases, and resilience.

Step 1: Validate configuration. Check that an SDK key is present, that it matches the expected format (starts with `fl_live_` or `fl_test_`), that any provided options have sane values (polling interval is not set to 100ms, for example). Fail fast with a clear, actionable error message if something is wrong. This is the one place where throwing an error is appropriate — the developer is actively setting up the SDK and needs immediate feedback about misconfiguration. An error here, during development, saves hours of debugging later.

Step 2: Fetch all flags. Make a single HTTP request to the Flagline API to get the current flag values for this environment. One request, not one per flag. This is a critical design decision worth emphasizing. If you have 50 flags, you do not want 50 HTTP requests during initialization. You

want one request that returns everything. The server already knows which flags exist for this environment key — let it do the work of collecting and returning them all. The request includes the user context if one was provided during initialization, so the server evaluates targeting rules server-side and returns resolved values.

Step 3: Populate the in-memory cache. The flag values from the server response go into a plain JavaScript `Map` (or a plain object, but `Map` has better performance characteristics for frequent reads). This is the cache that `getFlag` reads from. It is intentionally simple — no TTLs, no eviction policies, no LRU logic, no serialization to disk. Flag values are small (a boolean, a string, a number, occasionally a small JSON object) and few (most projects have tens or hundreds of flags, not thousands). Keeping them all in memory is trivial. Overengineering the cache is a common mistake — you are caching kilobytes, not gigabytes.

Step 4: Open the SSE connection. Start a Server-Sent Events connection to the Flagline streaming endpoint. This is a long-lived HTTP connection that the server pushes events through when flag values change. The SSE connection is the real-time update channel — it is how the SDK learns about flag changes without polling.

Step 5: Emit "ready" and return. The factory function resolves its promise, returning the client to the developer. At this point, the client has current flag values and a live connection for updates. The developer can immediately start calling `getFlag`.

The ordering of these steps matters. Steps 1-3 are sequential and blocking (the factory function does not resolve until flags are fetched or the timeout expires). Steps 4-5 can overlap — you do not need to wait for the SSE connection to be established before returning the client, because the client already has flag values from step 2. The SSE connection is for *future* updates, not the initial state.

What happens when initialization fails

If step 2 fails — network error, server down, DNS timeout, invalid API key — the SDK does *not* throw. This is a crucial design decision and it will feel wrong the first time you implement it.

Instead, the SDK resolves with a client that has no server-fetched values. It populates the cache with the developer-provided default values (from the `defaultValues` config option) if any exist, or leaves the cache empty. It logs a warning. It starts a retry loop to attempt the initial fetch again. And the consuming application continues to work, just with fallback values for every flag.

Why not throw? Because the SDK initialization typically happens at application startup. If it throws, the application does not start. If the application is a React SPA, the user sees a white screen. If it is a Node.js server, the process exits and no requests are served. All because the feature flag service was briefly unreachable.

Feature flags are an enhancement, not a critical dependency. The application should be able to run without them, falling back to sensible defaults. The SDK's job is to make flags available when possible and to degrade gracefully when the flag service is unavailable. The developer chose their default values specifically for this scenario — let those defaults do their job.

The one exception is an obviously wrong configuration — an invalid or missing API key, for example. That is a developer error, not a transient network issue, and it should be surfaced immediately with a thrown error. The distinction is: programming mistakes are caught loudly; infrastructure failures are handled silently.

The cache-first principle

Everything the SDK does at runtime is governed by one principle: reads are always synchronous from memory; writes happen in the background.

When `getFlag` is called, it looks up the key in the in-memory cache. If the key exists, return the cached value. If not, return the developer-provided default. That is the entire read path. No network, no disk, no async, no error handling, no branching beyond a single map lookup.

When the SSE connection receives a flag change event, the SDK updates the cache and notifies subscribers. When `identify` is called, the SDK fires off a fetch request, and when the response arrives, it updates the cache and notifies subscribers. These are background operations that do not block the calling code.

This model is sometimes called "optimistic" — the SDK optimistically serves whatever it has and updates asynchronously. It is the right model for feature flags because staleness is acceptable (for the milliseconds between a flag change and the SSE event delivery) and blocking is not (you never want a feature flag check to delay a render or a request handler).

The cache also provides an implicit offline mode. If the network drops entirely, the SDK keeps serving the last known flag values. The application does not suddenly break because WiFi went out. It continues operating with its most recent state, which is almost certainly still correct (flag values change infrequently, and a few minutes of staleness is irrelevant).

The SSE connection

Server-Sent Events are a deliberately simple protocol built on HTTP. The client opens a long-lived GET request. The server holds the connection open and sends text events whenever it has data to push. If the connection drops, the browser's built-in `EventSource` API automatically reconnects.

For Flagline's use case, SSE is ideal for several reasons. The communication is one-directional — only the server sends data to the client. The events are small — a flag key and its new value, typically less than 100 bytes. The protocol handles reconnection natively with built-in exponential backoff. And unlike WebSockets, SSE works through most corporate proxies and load balancers without special configuration, because it is just a long-lived HTTP response.

The SDK listens for a few event types on the SSE connection: a flag value changed, a flag was deleted, and a bulk refresh (typically sent after an `identify` call when the server has re-evaluated all flags for the new user). Each event triggers the appropriate cache update and subscriber notification.

One important consideration: `EventSource` in browsers does not support custom headers. You cannot send an `Authorization: Bearer ...` header with a browser-side SSE connection. The standard workaround is to pass the API key as a query parameter. This is acceptable for client-side SDK keys (which are public by design — they are embedded in client-side JavaScript). It would not be acceptable for server-side secret keys, but server-side usage typically uses polling rather than SSE anyway.

When SSE is unavailable (some edge runtimes do not provide `EventSource`, or the connection fails repeatedly after several retries), the SDK falls back to polling — a `setInterval` that re-fetches all flags at a configurable interval. Polling is objectively worse than SSE (higher latency for changes, more bandwidth usage, more load on the server) but it is better than nothing. The

developer can configure the poll interval, or they can disable real-time updates entirely if they prefer to fetch flags only at initialization.

The resilience principle

This is worth stating explicitly because it is the single most important architectural principle for SDK internals, and it is easy to violate incrementally as you add features:

The SDK must never crash the host application.

Not "should not." Must not. The host application is a customer's production system serving their users. They installed your SDK to check feature flags — a non-critical, nice-to-have enhancement. If your SDK throws an uncaught exception that takes down their checkout flow, or their dashboard, or their API server, they will rip it out within the hour, they will not come back, and they will tell other developers about the experience.

This means every network call is wrapped in try/catch. Every callback invocation from subscribers is wrapped in try/catch (because the subscriber's callback might throw, and that is not your fault, but you still must not let it propagate and kill other subscribers or the SDK's internal state). Every timer callback handles errors. Every state transition handles the "unexpected state" case gracefully.

In practice, the resilience principle manifests as a cascade of fallbacks:

1. Try SSE for real-time updates. If SSE fails, fall back to polling.
2. Try polling. If polling fails, serve stale cached values and keep retrying.
3. If the initial fetch fails, serve developer-provided defaults and keep retrying.
4. If the developer did not provide defaults, return the default value passed to `getFlag`.
5. If something truly unexpected happens internally, log a warning and continue. Never throw.

At every level of this cascade, the application keeps working. The quality of service degrades (less fresh data, less real-time), but it never drops to zero. The developer should be able to wrap their entire application in the Flagline SDK and have zero risk of the SDK being the cause of a production incident.

There is a practical testing implication here too: your SDK test suite should include tests for every failure mode. SSE connection refused, SSE connection dropped mid-stream, HTTP 500 from the flag endpoint, malformed JSON in the response, network timeout, `EventSource` constructor throwing, `localStorage` being unavailable, callbacks throwing exceptions. Each of these should result in graceful degradation, not a crash.

> **Coming from PHP:** Think about how Laravel's exception handler works — it catches exceptions and renders an error page rather than dumping a raw stack trace to the browser. Your SDK needs the same philosophy but stricter: you are a guest in someone else's application, and you do not get to decide what their error page looks like. You do not get to decide if an error is fatal. You handle your own problems, silently and gracefully, and you let the host application be in charge of its own error handling.

---

5. React SDK Thinking

Why a React wrapper exists at all

If the core SDK already works in the browser, why do you need a React package? A developer could just call `client.getFlag(...)` directly in their component and be done with it.

The answer is lifecycle management and reactivity. These are the two things React manages that plain JavaScript does not, and they are exactly the two things that make "just calling the client directly" insufficient.

React re-renders components when their state changes. If a flag value changes on the server and the SSE connection updates the SDK's internal cache, React does not know about that change. There is no re-render trigger. The component keeps displaying the stale value until something else causes a re-render (a user interaction, a different state change, a parent re-render). The flag value in the cache is correct, but the UI does not reflect it.

The React wrapper solves this by bridging the SDK's event system with React's state system. When the SDK's cache updates, the wrapper updates React state, which triggers a re-render, which causes the component to read the new value. This is the "reactivity" part.

The second problem is cleanup. When a component unmounts, any subscriptions it created need to be cleaned up. If you manually subscribe to flag changes in a `useEffect`, you need to return the unsubscribe function from the effect's cleanup. Every developer who uses the SDK in React would have to write this same boilerplate. Some would forget the cleanup, leading to memory leaks and stale callbacks. The wrapper handles cleanup automatically.

These two problems — reactivity and cleanup — are why every SDK that targets React ships a React wrapper. It is not about adding features. It is about making the existing features work correctly within React's paradigm.

The Provider pattern

The Provider pattern solves the same problem that Laravel's service container solves: making a shared resource available to any code that needs it without passing it explicitly through every intermediate layer.

In Laravel, you register a singleton in a service provider: `$this->app->singleton(FlaglineClient::class, ...)`. Any controller, middleware, or service can then inject it via the constructor or resolve it with `app(FlaglineClient::class)`. You do not pass the client from the HTTP kernel to the middleware to the controller to the service — the container makes it available everywhere.

In React, the equivalent mechanism is Context. You create a context, wrap your component tree in a Provider that holds the value, and any descendant component can access it via a hook. The Flagline provider wraps the app root (or the relevant subtree), and any component anywhere in the tree can call `useFlag` to access flags without receiving them as props through every intermediate component.

The provider has two responsibilities. First, it initializes the core SDK client once, at mount time. It receives the config as props, calls the factory function, and holds the resulting client in a ref. This means initialization happens once per application lifecycle, not once per component that needs a flag. Second, it manages cleanup — when the provider unmounts, it calls `destroy()` on the

client, preventing resource leaks. In development mode with React Strict Mode (which mounts and unmounts components twice to help you catch cleanup bugs), this is especially important. Your provider needs to handle the mount-unmount-remount cycle gracefully.

The useFlag hook: what it actually does

When a component calls `useFlag('dark-mode', false)`, the following sequence happens, and each step exists for a specific reason:

The hook reads the Flagline client from context. It calls `getFlag` on the client to get the current cached value and stores it in React state via `useState`. It sets up a subscription via `useEffect` that listens for changes to that specific flag key. When the flag changes (due to an SSE event, a polling update, or an `identify` call), the subscriber callback fires and calls `setState` with the new value. React detects the state change and re-renders the component. When the component unmounts, the `useEffect` cleanup function runs the unsubscribe function.

The hook does *not* make network requests. It does *not* manage caching. It does *not* evaluate targeting rules. All of that is in the core SDK. The hook is pure plumbing — read from core, subscribe to changes, update React state when notified, clean up on unmount. This is what "thin wrapper" means in concrete terms.

The hook also returns a loading state (`isReady` or `isLoading`), which tells the component whether the SDK has completed its initial flag fetch. Before initialization completes, the hook returns the default value. After initialization, it returns the server-evaluated value. The consumer can use the loading state to show a skeleton or shimmer if they want, but they do not have to — the default value is always available, so rendering something reasonable is always possible.

Server-side rendering: the genuinely hard problem

This is the part where things get tricky, and it is worth investing significant thinking time because a wrong decision here creates problems that are expensive to fix later.

In a Next.js App Router application, components can render on the server. When a request comes in, Next.js renders the React tree server-side, sends the HTML to the browser, and then "hydrates" it on the client — attaching event listeners and making the static HTML interactive.

The SDK faces a fundamental tension. On the server, there is no SSE connection. There is no long-lived in-memory cache persisting between requests. Each request is essentially a fresh start. You need flag values to render the page correctly (you do not want to render the "old checkout" on the server and then flash to the "new checkout" when the client hydrates). But you also do not want to initialize an entire SDK client for every incoming request, because that means an HTTP round-trip to the flag evaluation API on every page load, adding latency to your server response time.

The way to think about this is in two distinct layers.

Server layer: On the server (in a Server Component or a layout's data-fetching logic), you fetch the flag values directly. This could be a plain `fetch` call to the Flagline API, or it could be a direct Redis read if your API server has access to the same Redis cache that the flag evaluation service uses. The point is: this is server-to-server communication, fast and direct. You pass the resulting flag values down to the client component as props or through a server-side data channel.

Client layer: On the client, the provider initializes with these server-fetched values as its *initial state*. This is the crucial piece. The provider does not start with an empty cache and then fill it from the network — it starts with the server-provided values and renders immediately with them. There is no loading state. There is no flash of wrong content. The page arrives with the correct flag values baked into the HTML.

After hydration is complete, the client-side SDK then opens its SSE connection for real-time updates going forward. If a flag changes after the page has loaded, the SSE event arrives, the cache updates, and the component re-renders.

This is sometimes called the "seed and stream" pattern. The server seeds the initial state; the client streams updates.

The tricky detail is hydration mismatch. If the server renders with one flag value and the client initializes with a different value (because the flag changed in the milliseconds between server render and client hydration), React will log a hydration mismatch warning and potentially re-render the component, causing a visual flash. The solution is to ensure the client-side provider uses the exact same server-provided values for its initial render and only switches to live values *after* hydration is complete. This requires careful ordering in the provider's logic — render with server values first, then check if the client has newer values and update.

None of this is an insurmountable problem, but it requires forethought. If you design the SDK without considering SSR, you will end up retrofitting it later, which is always harder and usually results in a worse API. Think about the server rendering story from day one.

---

6. TypeScript DX Thinking

Types are not just for correctness — they are your primary documentation

If you have worked with TypeScript for even a short time, you know the immediate benefit: the compiler catches mistakes before runtime. But for an SDK, TypeScript types serve a much more important purpose that goes beyond correctness. They are the primary documentation that developers interact with moment to moment.

When a developer types `client.` in their editor and sees the autocomplete dropdown, that *is* your API documentation. The method names, parameter types, return types, and JSDoc comments tell them what the SDK can do and how to use it. When they hover over a method and see a rich tooltip with parameter descriptions and an example, that is your inline help. When they pass the wrong type and get a red squiggle before they even save the file, that is your error handling — catching the mistake five minutes into development instead of five hours into debugging production.

Most developers using your SDK will spend ten times more time reading autocomplete suggestions and hover tooltips than reading your documentation website. This means the investment you make in your TypeScript types has a higher return on developer experience than almost anything else you can spend time on. An hour spent writing good JSDoc comments and precise type signatures will save your users collectively hundreds of hours of documentation-browsing time.

Generic flag values and type inference

The simplest version of a `getFlag` method returns `unknown` or a union type. The SDK does not know at compile time whether `dark-mode` is a boolean, `banner-text` is a string, or `rate-limit` is a number. If the return type is `unknown`, every consumer has to cast the value with `as boolean` at every call site, which is tedious and error-prone (they might cast to the wrong type and TypeScript will not catch it, because `as` assertions override the type checker).

A better approach uses TypeScript generics to infer the return type from the default value. When the method signature is `getFlag<T>(key: string, defaultValue: T): T`, and a developer writes `getFlag('dark-mode', false)`, TypeScript infers that T is `boolean` and the return type is `boolean`. No cast needed. If they later try to use the result as a string, TypeScript catches the mistake immediately with a type error.

This is a small design decision that has an outsized impact on how the SDK feels to use. Every flag check is type-safe with zero effort from the consumer. They do not need to remember what type each flag is. The default value encodes that information, and the type system propagates it.

You can refine this further with constrained generics: `T extends string | number | boolean | Record<string, unknown>`. This limits flag values to the types your system actually supports and prevents nonsensical usage like `getFlag<Date>(...)` or `getFlag<Map<...>>(...)`.

What types to export

Export every type that a consumer might conceivably need to reference. The rule of thumb: if you can imagine a reasonable use case where a developer would need to write `: SomeType` and `SomeType` comes from your SDK, export it. The cost of exporting a type is near zero — types are erased at compile time and add no bundle size. The cost of *not* exporting a type is that the developer resorts to `typeof client`, `ReturnType<typeof initialize>`, or, worst case, `any`.

For a feature flag SDK, a reasonable set of type exports includes: the client class or interface, the configuration options type, the user context type, the flag value union type, the error type, the change event type, and the unsubscribe function type. A developer might need any of these when declaring variables, function parameters, or component props that interact with the SDK.

A common mistake is to export only the “main” types and leave utility types internal. If your error type is not exported, a developer who wants to type an error handler callback has to use `(error: unknown) => void` instead of `(error: FlaglineError) => void`. They lose the ability to access `.code` and `.message` without a type assertion. Export everything. It costs nothing.

Module augmentation for custom user attributes

Your SDK accepts a user context object for targeting. This object has some standard fields — `id`, `email`, `plan` — and potentially many custom attributes that vary per customer. How do you type the custom attributes?

The base approach is to accept `custom: Record<string, string | number | boolean>`. This works for everyone but provides no type safety on the custom attributes themselves. A developer can pass `{ teamSize: "not a number" }` and TypeScript will not catch it.

TypeScript’s module augmentation (declaration merging) lets power users extend an interface from your package without modifying your code. You define a `FlaglineUserCustomAttributes` in-

terface in your SDK as an empty interface. The consumer, in their own codebase, creates a `.d.ts` file that augments the module and adds their specific fields to that interface. Now TypeScript validates their custom attributes against their declared schema.

This is an advanced feature, and it should absolutely not be required for basic usage. The base type works without augmentation. Module augmentation is a power-user feature for teams that want an extra layer of type safety on their targeting attributes. Document it, but do not make people feel they need to understand it to use your SDK.

JSDoc comments as embedded documentation

Every public method and every exported type should have a JSDoc comment. Not a terse one-liner — include parameter descriptions, return value descriptions, `@example` blocks with realistic usage, and `@default` annotations for optional parameters with defaults.

These comments appear in the developer's IDE when they hover over a method or type. They are the most-read form of your documentation by a wide margin. A developer in flow state will not switch to a browser tab to check your docs website. They will hover over the method, read the tooltip for three seconds, and keep coding. Make sure those three seconds give them everything they need.

The `@example` tag is particularly powerful. A short, realistic code example in the JSDoc comment shows up directly in the hover tooltip. The developer does not even have to leave their current line of code to see how the method is used. For a method like `identify`, showing a realistic example with common attributes (plan, email, custom properties) communicates more than a paragraph of description ever could.

> **Coming from PHP:** This is exactly the same concept as PHPDoc comments (`/** @param string $key */`) that PhpStorm reads for autocomplete and hover tooltips. The tooling works the same way — you annotate your code, and the IDE surfaces those annotations to the consumer. The difference is that TypeScript's type system is structural and much more expressive than PHP's type declarations, so the annotations carry more weight and catch more errors. But the fundamental practice of "write good doc comments on your public API" is identical across both ecosystems.

The litmus test

Here is how to evaluate whether your TypeScript types are doing their job: a developer who has never seen your documentation should be able to install your SDK, type the package name followed by a dot in their editor, and figure out how to initialize the client, check a flag, and identify a user purely from autocomplete suggestions and JSDoc hover tooltips. If they have to open a browser and navigate to your docs website for anything in the basic happy path, your types are not carrying enough information.

This does not mean you do not need external documentation. You do, for advanced use cases, architectural explanations, troubleshooting guides, and migration paths. But the common case — the path that 90% of developers follow 90% of the time — should be fully navigable through the type system and JSDoc alone.

---

7. Building and Bundling Thinking

Why you need a build step at all

If you are coming from PHP, this might seem strange. A Composer package is just PHP files. You publish them, and consumers `require` them. The PHP runtime reads and executes the same source files you wrote.

JavaScript is fundamentally different in a way that trips up many newcomers: the code you write is not the code your consumers run. You write TypeScript — they run JavaScript. You write modern ESM syntax with `import` and `export` — some of their toolchains still expect CommonJS with `require` and `module.exports`. You write one package with many internal source files organized in directories — they want a small number of optimized entry points.

The build step bridges this gap. It compiles TypeScript to JavaScript (removing type annotations and generating separate `.d.ts` declaration files for TypeScript consumers). It bundles your internal modules into clean entry points. It produces the output formats the ecosystem expects. Think of it as the step that transforms your developer-friendly source code into consumer-friendly distribution artifacts.

> **Coming from Composer:** The closest PHP analogy would be if you wrote your package in a language that compiles to PHP (like a hypothetical TypeScript-for-PHP), and you had to run a build step before publishing to Packagist. Consumers would get the compiled PHP files and separate "type stubs" for IDE support. Nobody does this in PHP because PHP is its own source and runtime language. In JavaScript, the source language (TypeScript) and the runtime language (JavaScript) are different, and the runtime has multiple module format standards, so the build step is unavoidable.

Dual format output: ESM and CJS

The JavaScript ecosystem is in a long, painful transition from CommonJS (CJS, the `require()` format originating from Node.js) to ECMAScript Modules (ESM, the `import`/`export` syntax that is the official language standard). As of now, most of the ecosystem supports ESM, but CJS is far from dead. Many Node.js projects, older test runners like Jest (in certain configurations), and legacy build tools still use CJS internally or expect CJS input.

Your SDK needs to ship both formats. Failing to ship CJS means Node.js projects that have not migrated to ESM cannot use your package without additional configuration or wrapper code. Failing to ship ESM means modern bundlers cannot tree-shake your package (more on this below). Either omission results in frustrated developers and GitHub issues.

A dual-format build produces two sets of JavaScript files. The ESM output uses `import` and `export` statements. The CJS output uses `module.exports` and `require()`. Both have corresponding `.d.ts` declaration files so TypeScript consumers get type information regardless of which module format they use.

Build tools like `tsup` handle this with minimal configuration. You specify your entry point and that you want both `esm` and `cjs` formats, and it produces the correct outputs. The complexity is real (there are subtle differences in how ESM and CJS handle default exports, for instance), but it is handled by the tooling rather than by you manually writing two versions of your code.

Tree-shaking and why your build format matters

Tree-shaking is the process by which a consumer's bundler (Vite, webpack, esbuild, Rollup) analyzes your package's exports and removes any code that the consumer does not actually import. If your package exports ten functions and the consumer only imports two, tree-shaking eliminates the other eight from their final bundle.

This only works reliably with ESM. CommonJS's `require()` is dynamic — you can write `require(variable)`, or conditionally require different modules, or access exports by dynamic string keys. A bundler cannot analyze this statically, so it has to include everything just in case. ESM's `import` statements are static and declarative — the bundler knows exactly what is imported at build time and can confidently eliminate unused exports.

This is why the ESM output matters even when you also provide CJS. Modern bundlers will preferentially use the ESM entry point specifically because it enables tree-shaking. Your `package.json` tells bundlers where to find each format through the `exports` field (covered in the publishing section).

For tree-shaking to work well, your code must avoid side effects at the module level. A "side effect" is any code that executes when the module is *imported*, rather than when an *exported function is called*. If your module runs `addEventListener(...)` or modifies a global variable at the top level, the bundler cannot safely remove it — that side effect might be important even if no exports are used. Keep module-level code to pure declarations and exports only. Any setup logic should happen inside functions.

You can declare your package as side-effect-free in `package.json` with `"sideEffects": false`. This is a signal to bundlers that they can aggressively tree-shake any unused exports without worrying about losing important side effects.

Bundle size as a design constraint

For a server-side package, bundle size is largely irrelevant — the code runs on a machine with gigabytes of memory and no network transfer to the end user. For a client-side SDK, bundle size directly affects your customers' users. Every kilobyte of your SDK is a kilobyte that every user of every application that integrates your SDK downloads on every page load (or at least on their first visit before caching kicks in).

This is why established client-side SDKs obsess over bundle size and publish their gzipped size prominently. It is a competitive metric. LaunchDarkly's client SDK is around 20KB gzipped. If Flagline can deliver the same functionality in 5KB gzipped, that is a meaningful selling point for performance-conscious teams.

Establish a bundle size budget early and enforce it in CI. A tool like `size-limit` can fail your CI build if the bundle exceeds your budget. This prevents accidental growth — a single `import { format } from 'date-fns'` at the wrong level can add 20KB to your bundle without anyone noticing until a customer complains about their Lighthouse score dropping.

Zero dependencies is the foundation of a small bundle. Beyond that, be thoughtful about what code you include. Do you need a full-featured event emitter, or do 30 lines of code cover your use case? Do you need a complete URL parser, or does string concatenation work for the two URLs

your SDK constructs? Every utility function, every abstraction, every "nice to have" adds bytes. For an SDK, the discipline of "the smallest code that correctly does the job" pays dividends.

Peer dependencies vs bundled code

When your React package imports from `react`, that import should not be resolved at build time and bundled into your output. If it were, consumers would end up with two copies of React — the one they installed and the one baked into your package. React specifically cannot run as two separate instances in the same application (it uses shared module-level state), so this causes crashes with confusing error messages like "Invalid hook call."

Instead, React is declared as a *peer dependency*. A peer dependency is a contract that says: "I need this to work, but I expect the consumer to provide it. Do not install it on my behalf." The consumer's copy of React is used at runtime. Your package just imports from `react` and trusts it will be available in the consumer's `node_modules`.

In your build configuration, this means React must be "externalized" — the bundler sees `import { useState } from 'react'` and leaves it as-is rather than following the import and inlining React's source. The same applies to `react-dom` and to `@flagline/js` when building the React wrapper. Anything that the consumer provides should be external.

The rule of thumb: if your consumer is expected to have this dependency installed independently, it is a peer dependency and should be externalized. If it is an internal implementation detail that the consumer has no reason to install or know about, it should be bundled (or, ideally, written inline to avoid dependencies entirely).

> **Coming from Composer:** Peer dependencies are conceptually similar to Composer's `require` entries in a library that assume the host application provides the dependency. When a Laravel package lists `illuminate/support` in its `require` section, it expects the host Laravel application to provide it. The npm ecosystem makes this relationship explicit with a separate `peerDependencies` field, and the package manager warns (or errors) if the peer dependency is not satisfied. It is a stricter, more formal version of the same concept.

---

8. npm Publishing Thinking

How npm compares to Composer/Packagist

At a structural level, npm and Composer are similar enough that your existing mental model carries over with a few adjustments. Composer has Packagist as its default registry; npm has npmjs.com. Composer has `composer.json`; npm has `package.json`. Composer has `composer.lock`; npm has `package-lock.json` (or `pnpm-lock.yaml`). Composer installs to `vendor/`; npm installs to `node_modules/`.

The differences are in the details. Composer resolves a flat dependency tree — one version of each package, period. npm historically used a nested tree where multiple versions of the same package can coexist side by side in different parts of `node_modules`. Modern package managers like pnpm use a content-addressable store with symlinks to approximate a flat structure, but the possibility of version duplication still exists.

The biggest mental model shift is in how publishing works. In the PHP world, you push a Git tag, and Packagist detects the tag, clones your repository, and makes the package available. Packagist always serves your actual source code from Git. In npm, publishing is an explicit action: you run `npm publish`, which creates a tarball of the files you specify and uploads it to the registry. The registry stores that tarball. It does not clone your Git repository. It does not track your source code. It stores a snapshot of your build output.

This means what you publish to npm is not necessarily what is in your Git repo. Your published package contains only the files listed in the `files` field of `package.json` — typically your `dist/` directory (compiled JavaScript and type declarations), your `README.md`, and your `LICENSE`. Your source TypeScript, your tests, your build configuration, your CI workflows — none of that goes to npm. It stays in your repository.

The package.json fields that matter for publishing

Not every field in `package.json` affects your published package's behavior. These are the ones that directly determine whether consumers can use your package correctly:

`name` is what consumers type in `npm install`. For Flagline, this is `@flagline/js` and `@flagline/react`.

`version` follows semver. The registry rejects duplicate versions — once `1.0.0` is published, it is immutable. You cannot overwrite it, only unpublish it (within a limited time window and with restrictions).

`main` tells Node.js and older tools which file to load when someone calls `require('@flagline/js')`. It points to your CJS build output.

`module` tells ESM-aware bundlers (Vite, webpack, Rollup) which file to load when someone writes `import ... from '@flagline/js'`. It points to your ESM build output. This is a de facto standard, not part of the Node.js specification, but it is understood by every major bundler.

`types` tells TypeScript where to find type declarations. Without this, TypeScript consumers get no autocomplete, no type checking, no hover tooltips — your SDK becomes effectively untyped for them.

`exports` is the modern replacement for `main`/`module`/`types`. It is a conditional exports map that says: "when someone imports my package with ESM, serve this file; when they require it with CJS, serve this other file; and here are the type declarations for each." The `exports` field takes priority over the legacy fields in modern tooling. You should include both the `exports` map and the legacy fields for maximum compatibility across the range of tools your consumers might use.

One important detail in the `exports` map: the `types` condition must come before `default` in each entry. TypeScript resolves conditions in order and stops at the first match. If `default` (which points to a `.js` or `.mjs` file) comes before `types` (which points to `.d.ts`), TypeScript will find the JavaScript file, not the type declarations, and type checking will silently fail.

`files` is an allowlist of files and directories to include in the published tarball. It is the npm equivalent of a `.gitignore` in reverse — instead of listing what to exclude, you list what to include. Typically `["dist", "README.md", "LICENSE"]`. Without a `files` field, npm includes almost everything in your directory, which means source code, test files, configuration files, and potentially sensitive files like `.env` end up in your published package where anyone can see them.

`peerDependencies` lists packages that the consumer must provide. For `@flagline/react`, this includes `react` and `@flagline/js`.

`sideEffects` tells bundlers whether your package code can be safely tree-shaken. Set to `false` to enable maximum tree-shaking.

Scoped packages and why they matter

In npm, a scope is a namespace prefix: `@flagline/js` instead of `flagline-js`. Using a scope is a deliberate choice with several benefits.

Scopes give you a protected namespace. Once you claim the `@flagline` scope (by creating an npm organization named `flagline`), nobody else can publish packages under that scope. You can publish `@flagline/js`, `@flagline/react`, `@flagline/node`, and `@flagline/vue` knowing they will never collide with someone else's package.

Scopes communicate provenance. When a developer sees `@flagline/react` in their `package.json`, they know it is an official Flagline package. When they see `flagline-react`, it could be anyone — an official package, a community wrapper, or a malicious typosquat. The scope is a trust signal.

Scopes group packages visually. In the npm registry UI, in `node_modules`, in `package.json` — all official Flagline packages appear together under the `@flagline/` prefix, making it easy to identify the Flagline-related parts of a project's dependency tree.

One practical detail: scoped packages on npm default to private access (you need a paid org plan to publish private packages). For a public SDK, you need to either pass `--access public` on the first publish or set `"publishConfig": { "access": "public" }` in your `package.json`. After the first publish, npm remembers the access level.

> **Coming from Composer:** Composer's two-part naming convention (`vendor/package`) is effectively the same concept. `laravel/framework` on Packagist is analogous to `@laravel/framework` on npm. The `@scope/name` format is npm's equivalent of the Composer vendor namespace.

Semver: what constitutes a breaking change for an SDK

Semantic versioning is the versioning contract in both npm and Composer. Major.Minor.Patch (e.g., `1.3.7`). Patch for bug fixes, minor for backward-compatible new features, major for breaking changes.

For an SDK specifically, "breaking change" has a precise definition: any change that would cause existing consumer code to fail, error, or behave differently after updating. This includes:

- Removing a public method or exported type
- Changing a method's parameter signature (adding a required parameter, changing a parameter's type)
- Changing a method's return type
- Renaming a method, type, or configuration option
- Changing the shape of the configuration object in a way that invalidates existing configs
- Changing the default behavior of an existing feature (e.g., SSE was on by default and now it is off)

Things that are *not* breaking changes:

- Adding a new optional parameter to an existing method (existing calls still work)
- Adding a new method (nothing existing breaks)
- Adding a new exported type (nothing existing breaks)
- Adding a new optional field to the configuration object
- Fixing a bug (even if someone depended on the buggy behavior — documented behavior is the contract, not incidental behavior)
- Improving performance without changing observable behavior
- Internal refactoring that does not affect the public API

The semver contract matters enormously for an SDK because consumers typically pin their dependency with a caret range: `^1.3.0` means "any version >= 1.3.0 and < 2.0.0." They are trusting that every release within that range is backward-compatible. Violating that trust — shipping a breaking change in a minor or patch version — is one of the fastest ways to lose credibility in the developer ecosystem. A single broken semver contract generates GitHub issues, angry tweets, and a reputation that takes years to recover.

When in doubt about whether a change is breaking, treat it as breaking. Bump the major version. It is always better to do an unnecessary major bump than to accidentally break your consumers' production applications.

The publishing workflow

The process of getting code from your repository to the npm registry follows a predictable sequence:

Build. Compile TypeScript, generate declaration files, produce ESM and CJS outputs. This should run in CI on every push so the build is always known to work.

Test. Run unit tests, integration tests, type-checking, and bundle size checks. Everything must pass before a release is considered.

Version. This is where Changesets shines. Instead of manually editing version numbers in `package.json` files, developers add "changeset" files during development — small markdown files that describe what changed and whether it is a patch, minor, or major change. At release time, the Changesets tool collects all pending changesets, determines the correct version bump for each package, updates the version numbers in `package.json`, and generates changelog entries. In a monorepo, this is particularly valuable because a change to the core SDK might also require a version bump in the React wrapper (if the React wrapper depends on a specific core version range), and Changesets handles this dependency chain automatically.

Publish. `npm publish` (or `pnpm publish`, or `changeset publish` for monorepo multi-package publishing) pushes the built tarball to the registry. In an automated workflow, this is triggered by merging a "Version Packages" pull request that the Changesets GitHub Action creates when pending changesets exist on the main branch.

The typical CI flow is: a developer merges a PR to main, the CI workflow detects pending changesets, it creates a "Version Packages" PR that bumps versions and updates changelogs, and when that PR is merged, the workflow publishes the new versions to npm. This two-PR flow ensures that version bumps and changelogs are reviewable by humans before publishing.

Pre-releases for early validation

Before committing to a `1.0.0` release, you want real users to try your SDK in real applications. npm supports pre-release versions: `1.0.0-beta.1`, `1.0.0-rc.1`, and so on. These follow semver's pre-release convention and are not installed by default — a developer must explicitly request them.

You publish pre-releases on a separate npm dist-tag (like `beta` or `next`). Dist-tags are mutable pointers to specific versions. The `latest` tag always points to the most recent stable release. The `beta` tag points to the most recent pre-release. `npm install @flagline/js` gives the `latest` tag. `npm install @flagline/js@beta` gives the `beta` tag.

Use pre-releases to validate your API design with early adopters before locking it in. The transition from `0.x` or beta to `1.0.0` is significant — once you publish a stable `1.0.0`, you are bound by semver to maintain backward compatibility until `2.0.0`. Pre-releases are your chance to iterate freely on the API, discover awkward patterns, and fix them without worrying about breaking changes.

---

9. Documentation as Product

The README is a landing page

When a developer finds your package on npm — through a Google search, a recommendation from a colleague, or a link in a blog post — the first thing they see is the README. Not your marketing site. Not your docs portal. The README, rendered directly on the npmjs.com package page.

More developers will read your README than will visit your documentation website, attend your conference talk, watch your video tutorial, or read your blog post combined. Treat it with the same seriousness you would treat a product landing page, because that is exactly what it is.

The structure of an effective SDK README follows a consistent pattern that has been refined across thousands of successful developer tools:

Opening line. One sentence that explains what this is and who it is for. Not a mission statement. Not a paragraph. One sentence. "The official Flagline SDK for JavaScript. Evaluate feature flags in any JS runtime with zero dependencies."

Badges. npm version, bundle size, build status. These communicate professionalism and health at a glance. A developer who sees a green build badge and a "3.2KB gzipped" badge has already formed a positive impression before reading a word of documentation.

Installation. The `npm install` command. Copy-pasteable. Nothing else in this section. No explanatory paragraphs. No "you might also need." Just the install command.

Quick start. The complete happy path in the minimum number of lines. Initialize, check a flag, done. This should be a single code block that the developer can copy into a file, replace the SDK key with theirs, and run. Not a simplified example. Not a pseudocode sketch. A real, working, copy-pasteable example.

API reference. Every public method with its signature, parameters, and a brief description. This can be compact — a couple of lines per method. Detailed explanations and edge cases go in full documentation, not the README.

Framework links. If someone found `@flagline/js` but actually wants the React wrapper, make it one click to get there. "Using React? See `@flagline/react`."

Every code example must be copy-pasteable

This cannot be overstated, because it is the single most common failure mode in SDK documentation: code examples that do not work when copied and pasted.

Every example should include its import statements. Do not assume the reader knows where `FlaglineProvider` is imported from — write the import line. Every example should use realistic but obviously fake values for API keys — `fl_live_your_key_here`, not `YOUR_KEY` (which looks like a placeholder the developer is supposed to leave as-is) and not a real key (which is a security problem). Every example should be self-contained. If it depends on setup from a previous example, repeat the setup or link to it explicitly.

If an example requires running code that was shown earlier (initializing the client, for instance), either include the initialization in the example or clearly state what the prerequisite is. "Assumes you have initialized `client` as shown in Quick Start above" is acceptable. No mention of the prerequisite at all is not.

A useful habit: after writing a documentation example, open a fresh project, paste the example, and run it. If it does not work without modification, fix the example. Your documentation is a promise to the developer: "do this, and it will work." Breaking that promise erodes trust faster than almost anything else.

Framework-specific examples

Different frameworks have different integration patterns, and a developer evaluating your SDK wants to see that it works in *their* specific setup. A React developer expects to see a provider and hooks. A Next.js developer needs to see the server-rendering story. A Node.js developer expects plain function calls with no framework magic. A Vite developer wants `import.meta.env` instead of `process.env`.

Each framework gets its own example that shows the idiomatic integration pattern for that framework. The React example uses `useFlag` inside a functional component. The Next.js example shows server-side flag fetching in a Server Component and client-side hooks in a Client Component. The Node.js example shows initialization at server startup and flag checks in an Express request handler.

These examples serve a second purpose beyond instruction: they validate your API design. If the Next.js example is awkward or requires too much boilerplate, that is a signal that your SDK does not fit well with Next.js patterns, and you should consider API adjustments. If the Node.js example requires the developer to understand React concepts, something has leaked across your package boundaries.

Write the framework examples before you finalize the SDK's API. The examples will tell you whether your design is right. If you cannot write a clean, short example for a given framework, the API probably needs work.

> **Coming from PHP:** Laravel's documentation is widely considered the best in the PHP ecosystem, and a significant reason is that every code example is complete, correct,

and works when copied. Taylor Otwell famously writes documentation before (or alongside) implementation, using the examples as a specification for how the feature should work. This is exactly the same principle. Your documentation examples are both user-facing docs and a design specification. If the example looks good, the API is probably right. If the example is ugly, the API might be wrong.

Documentation is not an afterthought — it is a product feature

Documentation is not something you write after the SDK is "done." It is a core feature of the SDK, with the same standing as type safety or error handling. A developer who cannot figure out how to use your SDK because the docs are incomplete will have exactly the same experience as a developer whose code does not work because of a bug. From their perspective, the result is identical: they cannot get the thing to work.

Budget time for documentation the same way you budget time for writing tests. Write docs alongside the code, not after. Review docs in pull requests the same way you review code — does this make sense? Is anything ambiguous? Would a developer with no prior context understand this paragraph?

And keep docs updated. Stale documentation is actively worse than no documentation, because it is misleading. When you change a method signature, update the docs in the same pull request. When you add a configuration option, document it before merging. Treat doc updates as a required, non-optional part of the code review checklist — the PR is not mergeable if the docs do not reflect the changes.

Closing Thought

Building an SDK is a different discipline than building an application. An application serves end users who interact through a visual interface. An SDK serves developers who interact through code, types, and documentation. The feedback loop is different (SDK bugs are reported as GitHub issues, not support tickets). The quality bar is different (a bug in your SDK is a bug in *every application* that uses it). And the design principles are different (simplicity and predictability matter more than feature richness).

The single most important thing to internalize is empathy for your consumer. Every design decision — the API shape, the error handling, the type signatures, the documentation, the bundle size, the default values — should be evaluated from the perspective of the developer who will use your SDK in their project at 11 PM while trying to ship a feature by morning. They do not care about your internal architecture. They do not care about your clever abstractions. They care about whether they can install your package, check a flag, and get back to building their own product.

Make that as easy, as reliable, and as invisible as possible. Everything else follows from that.