Contents

01 — TypeScript Thinking Guide

Audience: Backend developer with 5+ years PHP/Laravel experience transitioning to React, Next.js, Node.js, TypeScript, PostgreSQL (Prisma), and Redis.

Project: Flagline — a feature-flag SaaS.

What this document is: A mental-model guide. It explains *how to think* about TypeScript decisions in this project — not what code to copy-paste. If you understand the reasoning, you can write the code yourself.

Table of Contents

---

1. TypeScript Configuration in a Monorepo

The Mental Model: Inheritance, Not Duplication

> **Coming from PHP/Laravel:** Laravel has one `composer.json` and one PHP runtime. Configuration is centralized — you set `strict_types=1` per file and you are done. There is no concept of "this directory uses PHP 8.2 features but that directory targets PHP 7.4."
>
> TypeScript is different. Each sub-project can have its own `tsconfig.json` with its own compilation target, module system, and strictness settings. In a monorepo with five or six packages, this means five or six config files. Without a strategy, those files drift apart and you spend hours debugging why the same code compiles in one package but not another.

The core idea is borrowed from object-oriented design: define a base configuration with your non-negotiable defaults, then let each package *extend* it and override only what is specific to its runtime environment.

Think of it in three layers:

Layer 1 — The base config (`packages/config-typescript/base.json`). This is where you make project-wide decisions that should never vary: strict mode on, target ES2022, `noUncheckedIndexedAccess` on, source maps on. Every opinion that is about *code quality* rather than *deployment target* belongs here. If a developer creates a new package tomorrow, they extend the base and immediately inherit every safety guarantee.

Layer 2 — Runtime-specific presets (like `node.json`, `nextjs.json`, `react-library.json`). These extend the base and add settings that depend on *where the code runs*. A Next.js app needs DOM types in its `lib` array because it renders in browsers. A Node.js API service does not — including DOM types would let you accidentally reference `window` in server code without a compile error. A React library needs `jsx: "react-jsx"` to transform JSX; the core SDK that has no JSX does not.

The question you should ask when creating a preset is: *"What is true about every package that runs in this environment?"* If the answer is "it needs DOM types" or "it uses JSX," that belongs in the preset. If the answer is "it uses Prisma" — that is package-specific and does not belong in a preset.

Layer 3 — Per-package overrides (each package's own `tsconfig.json`). This is where you set `rootDir`, `outDir`, `include`/`exclude` globs, and path aliases. These are directory-structure concerns that differ between packages. The dashboard needs `"paths": { "@/*": ["./*"] }` for its internal imports. The SDK does not.

The Decision Framework for a New Package

When you add a new package to the monorepo, here is the sequence of questions:

1. **Where does this code run?** Browser, Node.js, or both (isomorphic)? This tells you which preset to extend. If it runs in both, extend the Node preset and avoid using DOM APIs — the browser consumer's bundler will handle the rest.
2. **Does it emit JSX?** If yes, you need a preset with `jsx` configured. If no, you do not want one — having `jsx: "react-jsx"` in a non-React package is not harmful but it is misleading and invites accidental React dependencies.
3. **Is it published to npm or consumed internally only?** Published packages need `declaration: true` so that TypeScript consumers get `.d.ts` files. Internal packages can skip the build step entirely and point their `main` field at raw `.ts` source, because Turborepo and the consuming package's compiler handle resolution.
4. **Does it have special path aliases?** Only add `paths` if the package is large enough to benefit from shortened imports. For a package with ten files in a flat `src/` directory, relative imports are fine. Path aliases add indirection — every developer and every tool (Jest, tsup, ESLint) needs to understand them.

Project References vs. Path Aliases: The Trade-off

These two features solve different problems and are often confused.

Path aliases (`"@/*": ["./*"]`) are about *convenience within a single package*. Instead of `import { db } from "../../../lib/db"`, you write `import { db } from "@/lib/db"`. This is a cosmetic improvement inside the dashboard app. It has nothing to do with cross-package imports.

Project references (`"references": [{ "path": "../../packages/shared-types" }]`) are about *cross-package type checking*. They tell the TypeScript language server: "when I import from `@flagline/types`, here is where the source lives — jump there for go-to-definition." Without project references, Cmd+Click on an imported type might land you on a generated `.d.ts` file (or nowhere), instead of the actual source.

The trade-off with project references is maintenance overhead: every time you add a new internal dependency, you must update the `references` array. For Flagline, where the dependency graph is small and stable (dashboard depends on types and db, API depends on types and db, SDKs depend on types), this is worth it. For a monorepo with fifty packages and frequent dependency changes, it becomes tedious and teams often drop it in favor of letting the bundler resolve everything.

> **Coming from Laravel:** There is no direct equivalent. The closest analogy is PHPStan's multi-package analysis — where you configure PHPStan to understand your internal package boundaries so it can trace types across them. Project references are TypeScript's way of telling the compiler "these packages form a graph; check them together."

When to Change Defaults

The base config should change rarely. Here are the only legitimate reasons:

- **Upgrading the TypeScript version** opens up a new `target` or `lib` option. When Node.js 22 becomes the minimum, you might bump `target` to `ES2023`.
- **A new strict flag ships** in a TypeScript release. You should evaluate it and, if it catches real bugs, add it to the base.
- **The module resolution strategy changes** in the ecosystem. The move from `"node"` to `"bundler"` module resolution happened once and is unlikely to change again soon.

If you find yourself wanting to turn something *off* in the base config, pause and ask why. Almost always, the right move is to fix the code, not relax the config.

---

2. Strict Mode: What Each Flag Protects You From

TypeScript's `"strict": true` is a shorthand that enables a family of individual flags. Rather than memorizing a list, it is more useful to understand what class of bug each one prevents — specifically in the context of Flagline.

`strictNullChecks` — The Single Most Important Flag

> **Coming from PHP/Laravel:** PHP 8 added union types and `?string` syntax, but decades of PHP code treats `null` as a valid value for everything. Laravel's Eloquent

> returns `null` when a model is not found, and if you forget to check, you get a "trying to access property on null" error at runtime. You have probably been burned by this.
>
> `strictNullChecks` makes this a *compile-time* error in TypeScript. If a function can return `null`, the type says so, and you must handle it before accessing properties.

In Flagline, flag evaluation is the hot path. A flag might not exist (the customer passed a typo as the key), might exist but be disabled, or might have no rules that match the current user. Without `strictNullChecks`, the evaluator could merrily access properties on `undefined` and return garbage to the SDK without any compiler complaint.

The mental model is simple: **`null` and `undefined` are not members of other types unless you explicitly say so.** A `string` is always a string — never null, never undefined. If it can be null, the type is `string | null`, and you must narrow it before using it. This one flag eliminates an entire category of runtime crashes.

`strictFunctionTypes` — Contravariance for Callbacks

This flag is more subtle. It ensures that function parameters are checked *contravariantly*, which means: if you have a handler that accepts a `ServerEvent`, you cannot pass it where a handler that accepts `FlagCreatedEvent` (a more specific type) is expected. Without this flag, TypeScript allows the unsafe assignment and you discover the mismatch at runtime.

In Flagline, this matters for event handlers and webhook callbacks. The audit system emits different event types (`FLAG_CREATED`, `RULES_UPDATED`, `MEMBER_INVITED`), each with a different payload shape. If you wire up a handler that expects a specific event shape to a generic event bus, `strictFunctionTypes` catches the mismatch before it reaches production.

`noImplicitAny` — No Guessing

When TypeScript cannot infer a type, it defaults to `any` — the "I give up, anything goes" type. With `noImplicitAny`, the compiler refuses to do this silently. Instead, it asks you to be explicit.

This catches a surprisingly common mistake: forgetting to type function parameters. In PHP, if you forget a type hint, the parameter accepts anything and you deal with it at runtime. TypeScript *can* behave the same way — `noImplicitAny` prevents that.

For Flagline, this is particularly important in route handlers and middleware. A Fastify handler receives a request object; if you forget to type the body, TypeScript infers `any`, and you lose all type checking on the most dangerous part of your application — user input.

`noUncheckedIndexedAccess` — The Flag That Changes How You Think

This is not part of `strict: true` — you must enable it separately — and it is arguably the most impactful flag for a feature-flag service.

Here is the scenario it prevents: Flagline stores evaluation results as a `Record<string, EvaluationResult>`. When the SDK looks up a flag by key, the type system (without this flag) says the result is `EvaluationResult` — definitely present, never undefined. But what if the key does not exist in the record? At runtime, you get `undefined`. The type system lied to you.

With `noUncheckedIndexedAccess` enabled, any indexed access on a record or array includes `| undefined` in the result type. Looking up `results[flagKey]` gives you `EvaluationResult | undefined`, forcing you to handle the missing case.

The cost is real: you write more `if` checks, more optional chaining, and more non-null assertions in places where you *know* the value exists. But for a service where the core value proposition is "tell you the right flag value for this user," false confidence about a value being present is worse than verbose code.

> **Coming from PHP/Laravel:** This is like the difference between `$array['key']` (which throws a notice if missing) and `$array['key'] ?? null` (which is explicit). PHP chose to make missing keys a notice, not an error, which leads to silent bugs. TypeScript with this flag treats every array/record access as potentially missing, which is closer to how the real world works.

`strictPropertyInitialization` — Catch Uninitialized Class Properties

If you declare a class property with a type but do not assign it in the constructor, this flag catches it. In Flagline, this matters for service classes that have injected dependencies. If you add a new dependency to a service and forget to initialize it, the compiler tells you immediately rather than letting `undefined` flow through your code until it hits a method call and throws.

The Cost of Strict Mode

Every strict flag makes your code more verbose. You will write more type annotations, more null checks, more narrowing logic. This is the trade-off, and it is worth making explicitly rather than pretending it does not exist.

The benefit is proportional to the distance between where the bug is introduced and where it manifests. In a flag evaluation pipeline — where a value flows from the database through targeting rules through rollout logic through serialization to JSON and across the network to an SDK — a `null` introduced in step one might not crash until step five. Strict mode catches it at step one. The verbosity cost is paid once, at write time. The bug-hunting cost without it is paid repeatedly, at debug time, often at 2 AM.

---

3. Key Type Patterns and When to Reach for Them

This section is about building intuition. For each pattern, the question is not "what is the syntax?" — you can look that up. The question is: "What problem is in front of me, and which tool solves it?"

Generics: "This Works the Same Way for Different Types"

> **Coming from PHP/Laravel:** PHP got generics in PHPStan and Psalm via doc-block annotations (`@template T`), but the runtime does not enforce them. If you have used Laravel's `Collection<int, User>`, you have seen the concept. TypeScript generics work the same way conceptually, but the compiler enforces them — you cannot put a `string` where a `User` was expected.

The signal that you need a generic: You are writing a function or type and you find yourself wanting to say "this works for any type of thing, but once you tell me *which* thing, I should be consistent about it." The classic example: a function that wraps a value in a response envelope. It does not care whether the value is a flag, a project, or a user — but the *output type* should match the *input type*.

Think of it as a function at the type level. `ApiResponse<T>` is a type-level function that takes a type `T` and returns a new type — an object with a `data` field of type `T`. When you write `ApiResponse<FlagConfig>`, you are "calling" that function with `FlagConfig` as the argument.

The decision process for Flagline:

- You are writing the API response wrapper. Every endpoint returns `{ success, data, error }`. The shape is identical, but `data` varies. This is a textbook generic: you define `ApiResponse<T>` once, and each endpoint specifies what T is.
- You are writing the SDK's `getFlag` method. It returns a flag value, but the *type* of that value depends on the flag type (boolean, string, number, JSON). You want the caller to say `getFlag<boolean>("dark-mode")` and get back a `boolean`, not a vague `FlagValue` union. This is a generic constrained by the possible flag value types.
- You are writing a React hook. `useFlag("dark-mode")` should return the evaluated value, but the hook does not know at definition time what type the value is. A generic with a default lets the caller specify it if they want precision, or leave it for a general return type.

When NOT to use generics: If the function only works with one type, do not make it generic for the sake of abstraction. A function that creates a flag always takes `FlagConfig` — making it `createEntity<T>(entity: T)` adds complexity without value. Generics solve the "works for multiple types" problem. If there is only one type, you do not have that problem.

Discriminated Unions: "This Is One of Several Shapes, and I Can Tell Which"

> **Coming from PHP/Laravel:** Think of a polymorphic relationship in Eloquent. A `notifications` table has a `type` column (`"email"`, `"sms"`, `"push"`) and a `data` JSON column whose shape depends on the type. In PHP, you check the type and then cast/validate the data manually. TypeScript discriminated unions do this at the type level — the compiler knows that if `type` is `"email"`, then `data` has `subject` and `body` fields.

The signal that you need a discriminated union: You have a value that can take one of several shapes, and there is a field (the "discriminant") that tells you which shape it is. The key insight is that checking the discriminant *narrows the type automatically*. You do not need a type assertion or a cast — a regular `if` or `switch` statement is enough.

For Flagline, discriminated unions appear everywhere:

- **Evaluation reasons.** An evaluation result has a `reason` field that is one of `"TARGETING_MATCH" | "DEFAULT" | "FLAG_DISABLED" | "FLAG_NOT_FOUND" | "ERROR"`. Each reason implies different available data. A `TARGETING_MATCH` has a `ruleId` and `variationId`. A `FLAG_NOT_FOUND` has neither. By modeling this as a discriminated union (rather than making `ruleId` and `variationId` optional on every result), you make it impossible to access `ruleId` on a `FLAG_NOT_FOUND` result without the compiler complaining.

- **Audit events.** The audit log has an `action` field (`FLAG_CREATED`, `RULES_UPDATED`, `MEMBER_INVITED`, etc.) and `before`/`after` JSONB columns whose shape depends on the action. If you model this as one big type with every field optional, you get no help from the compiler. If you model it as a union discriminated by `action`, you get exhaustive checking — add a new event type and the compiler tells you everywhere that needs to handle it.
- **API errors.** A validation error has field-level details. A permission error has the required role. A rate-limit error has a retry-after timestamp. Same pattern — a `code` field discriminates the shape.

The thinking process: when you find yourself writing a type with several optional fields and a mental note that says "fields X and Y are only present when Z is true," you almost certainly want a discriminated union instead. The mental note becomes a compiler guarantee.

Utility Types: Transforming Types You Already Have

> **Coming from PHP/Laravel:** PHP has no equivalent. The closest analogy is Eloquent's `makeHidden()` / `makeVisible()` / `only()` methods, which transform a model's serialized output at runtime. TypeScript's utility types do the same thing at the type level, at compile time — no runtime cost.

Utility types are type-level functions that transform existing types. The important ones for Flagline:

- **`Pick<T, K>` and `Omit<T, K>`** — Select or remove fields from a type. The signal: you have a complete type (like the full flag model from Prisma) but you need a subset (like the fields you send to the SDK, which should not include `tenantId` or `createdById`). Rather than defining a new type from scratch (and risking drift when you add a field to the original), you derive it: `type SdkFlag = Pick<Flag, "key" | "name" | "enabled">`.
- **`Partial<T>` and `Required<T>`** — Make all fields optional or required. The signal: you need an "update" payload where every field is optional (the user might update the name without touching the description), derived from a "create" payload where every field is required. `Partial<CreateFlagInput>` gives you the update type for free.
- **`Record<K, V>`** — An object type with known key and value types. The signal: you are building a lookup table or map. The batch evaluation response is a `Record<string, EvaluationResult>` — a mapping from flag keys to their results.
- **`Readonly<T>`** — Prevents mutation. The signal: you have a value that should never be changed after creation — like flag configurations fetched from the server that the SDK caches locally. Making them `Readonly` catches accidental mutations at compile time.

The decision is always the same: *"Do I already have a type that is close to what I need, or am I starting from scratch?"* If the answer is "close," reach for a utility type. If the answer is "from scratch," define a new interface.

Type Guards: "I Know More Than the Compiler Does"

> **Coming from PHP/Laravel:** PHP's `instanceof`, `is_string()`, `is_array()` checks are type guards — after the check, PHP (and your IDE) knows the type. TypeScript type guards work identically, but you can define *custom* ones for your own types.

A type guard is a function that returns a boolean and has a special return type annotation (`value is SomeType`) that tells the compiler: "if this function returns true, narrow the type."

The signal that you need a type guard: You have a value whose compile-time type is broad (like `unknown` or a union type) and you need to narrow it to a specific type based on runtime inspection. The two most common scenarios in Flagline:

1. **Validating external input.** An API request body comes in as `unknown`. You need to verify it matches `EvaluationContext` before using it. A type guard that checks for the presence and types of required fields lets you go from `unknown` to `EvaluationContext` with compile-time safety.
2. **Narrowing union types.** You have a `FlagValue` (which is `boolean | string | number | Record<string, unknown>`) and you need to do something specific with the boolean case. `typeof value === "boolean"` is a built-in type guard. But if your union members are objects distinguished by a field (like audit events), you write a custom guard.

In practice, you will find that most type guards can be replaced by either discriminated unions (if you control the shape) or schema validation libraries like Zod (if you are validating external data). Reach for a custom type guard only when those two options do not fit.

Branded Types: "These Are Both Strings, But They Are Not the Same Thing"

This is a pattern for preventing a specific class of mistake: passing the right *type* but the wrong *value*. In Flagline, a tenant ID, a project ID, a flag ID, and an environment ID are all strings. TypeScript is perfectly happy if you pass a tenant ID where a project ID is expected — they are both `string`. The code compiles, the query runs with the wrong ID, and you get garbage data or a subtle security bug.

A branded type adds a phantom property to a type that exists only at the type level (no runtime cost). `type TenantId = string & { readonly __brand: "TenantId" }`. Now `TenantId` and `ProjectId` are structurally different, and passing one where the other is expected is a compile error.

The signal that you need branded types: You have multiple values of the same primitive type that should not be interchangeable. The more dangerous the mix-up (IDs across security boundaries, amounts in different currencies), the more valuable the branding.

For Flagline, the most valuable place to brand is the multi-tenant boundary. Every database query is scoped by tenant ID. Passing a project ID where a tenant ID is expected would bypass tenant isolation — the most critical security invariant in the system. Branding makes this mistake a compile error.

> **Coming from PHP/Laravel:** PHP has no direct equivalent, but if you have used value objects (a `Money` class instead of a raw `float`, a `UserId` class instead of a raw `int`), you have applied the same principle at the runtime level. Branded types are the compile-time version — zero runtime overhead, same conceptual safety.

The cost is that branded types require explicit construction. You cannot write `const id: TenantId = "abc"` — you need a creation function that brands the string. This is the trade-off: more ceremony for more safety. Use it surgically, at the boundaries that matter most.

`const` Assertions and Literal Types: Precision Without Overhead

TypeScript infers `"production"` as `string` by default. If you want the type to be the *literal* `"production"` — not any string, but specifically that one — you use `as const`. This gives you the same precision as an enum without defining one.

For Flagline, this matters for things like evaluation reasons, flag value types, and plan tiers. A type like `"TARGETING_MATCH" | "DEFAULT" | "FLAG_DISABLED"` is a union of literal types. Each member is a specific string, not "any string." This means typos are compile errors, exhaustive switch statements are enforceable, and autocomplete works in the IDE.

The decision: use literal unions when the set of values is small, known at compile time, and does not need runtime iteration. Use enums when you need to iterate over the values at runtime (like populating a dropdown) or when the values carry meaning that benefits from a named constant. In Flagline, evaluation reasons and flag types are literal unions in the shared types package. Prisma enums (like `Role` and `AuditAction`) are actual enums because Prisma generates them that way.

---

4. Typing the Backend

The Fundamental Mental Shift: Compile Time vs. Runtime

> **Coming from PHP/Laravel:** PHP type checking happens at runtime. If a function declares `function evaluate(FlagConfig $flag): EvaluationResult`, PHP enforces this when the function is called. If you pass the wrong type, you get a `TypeError` at runtime — but at least you get one. The types are *real*.
>
> TypeScript types are *erased* before the code runs. After compilation, there are no types — just JavaScript. This means TypeScript cannot protect you from bad data coming in from the outside world at runtime. It can only protect you from mistakes you make within your own code at compile time.

This distinction creates a clear boundary in how you think about types:

Inside the boundary (your own code): Types are trustworthy. If a function says it returns `EvaluationResult`, it does. If a variable is typed as `FlagConfig`, it is one. The compiler enforces this. You do not need runtime checks within your own typed code unless you are dealing with dynamic data like JSON parsing.

At the boundary (external input): Types are wishes, not facts. An HTTP request body, a database JSON column, an environment variable, a Redis cache value, a webhook payload — these are all `unknown` in the honest sense. The data may or may not match the type you want. You must validate before you trust.

In Flagline, the boundaries are:

1. **API request bodies.** SDK clients send evaluation contexts. Dashboard users send flag configurations. These are untrusted. Validate with Zod (or a similar schema library) and let the validated output become the trusted type.
2. **Database JSON columns.** The `rules` column in `flag_environments` is `Json?` in Prisma, which becomes `JsonValue | null` in TypeScript — essentially `unknown`. You defined the

schema, but the database does not enforce it. Parse and validate when reading.

3. **Redis cache values.** Serialized to JSON strings and back. The deserialized value is `unknown` until you validate it.
4. **Environment variables.** `process.env.DATABASE_URL` is `string | undefined`. Not `string` — it might not be set. Handle the `undefined` case.
5. **Webhook payloads** from Stripe or other services. These have their own type packages, but you should still validate the signature and structure before trusting.

The Zod Pattern: Validate Once, Trust Forever

The practical pattern is: define a Zod schema that mirrors your TypeScript type, validate incoming data against it, and use the output. Zod schemas produce TypeScript types automatically via `z.infer<typeof schema>`, so you can avoid defining the type twice.

The decision of where to put the validation is architectural. In Flagline:

- **Fastify route handlers** validate request bodies at the route level, before the request reaches the service layer. The service layer receives already-validated, correctly-typed data. This is analogous to Laravel's Form Request validation — the controller does not worry about validation because the framework handled it.
- **Next.js Server Actions** validate inputs at the top of the action, before any database call. Server Actions are direct function calls from the client — there is no middleware layer to intercept them.

The mental model is a pipeline: `unknown input` –> `validation` –> `typed value` –> `business logic`. Everything before validation is untrusted. Everything after is trusted. The validation step is the gate.

Typing Fastify Handlers

Fastify has built-in support for typing route parameters, query strings, request bodies, and response bodies via generics on the route definition. The mental model: you describe the *shape* of the HTTP contract (what the route accepts and returns), and Fastify ensures your handler function matches that contract.

When you define a route's schema, think of it as defining a contract at two levels simultaneously: the TypeScript type (for compile-time checking within your code) and the JSON Schema (for run-time validation of incoming requests). Fastify can derive one from the other using libraries like `@sinclair/typebox`, or you can use Zod with a Fastify adapter. The key insight is that you want *one source of truth* for the shape, not a TypeScript interface in one file and a JSON Schema in another.

Typing Express Middleware (Dashboard API Routes)

Next.js API route handlers in the App Router receive a `Request` object and return a `Response`. The request body is accessed via `await request.json()`, which returns `any` by default. This is a trust boundary — you must validate.

The pattern is to create thin wrapper functions that validate the request body and return a typed result, or throw a standardized error. Your route handler calls the wrapper and gets back typed data. This keeps validation logic reusable and prevents the handler from becoming a soup of `if` checks.

Typing Middleware and Shared Context

In Fastify, plugins and decorators allow you to attach typed data to the request object — things like the authenticated user, the resolved tenant, or the API key. The mental model is *progressive enrichment*: the raw request enters with minimal type information, and each middleware layer adds more. By the time the request reaches your handler, the request type includes all the data that the middleware layers have attached.

> **Coming from Laravel:** This is exactly what Laravel middleware does — `auth` middleware adds `$request->user()`, tenant middleware adds `$request->tenant()`. The difference is that in Laravel, `$request->user()` can still return `null` if you forget the middleware. In typed Fastify, the type system knows whether the middleware has run or not, because the handler's request type either includes `user` or it does not.

---

5. Typing React

The Mental Shift from Templates to Functions

> **Coming from Blade/Laravel:** In Blade, you pass data from a controller to a view: `return view('flags.show', ['flag' => $flag]).` The template receives `$flag` and uses it. There is no type checking — if the controller passes an array instead of a model, the template crashes at runtime.
>
> In React, a component is a function. Props are its parameters. TypeScript types those parameters. If you pass the wrong shape, the code does not compile. This is the entire mental model: **components are typed functions, props are typed arguments**.

A Blade template is a loosely-typed HTML factory that receives variables by name. A React component is a strictly-typed function that receives a single object (props) whose shape is defined by a TypeScript interface. The compiler checks every call site — every place where the component is used — to ensure the props match.

Props: More Than Just Arguments

The simplest way to think about props is as function parameters, but there are nuances:

Required vs. optional props. A prop without ? is required — omitting it is a compile error. A prop with ? is optional. The decision: make a prop required if the component cannot render meaningfully without it. Make it optional if there is a sensible default. This mirrors PHP function parameters with default values.

Children as props. In Blade, you use `@yield` and `@section` for slot-based composition. In React, `children` is a prop — it is the content between the opening and closing tags of a component. Typing `children` explicitly (as `React.ReactNode` for any renderable content, or as a more specific type for render-prop patterns) documents what kind of content the component accepts.

Callback props. Where Blade uses Alpine.js or Livewire for interactivity, React passes functions as props. A `FlagRow` component might accept an `onToggle` prop — a function called when the user flips the flag. Typing the callback's signature ensures that the parent component passes a function with the right parameters.

Typing Context: The Global State Contract

React Context is how you share data across a component tree without passing props through every level. In Flagline, you would use context for things like the current tenant, the current project, and the authentication state.

> **Coming from Laravel:** Context is conceptually similar to singleton bindings in Laravel's service container. When you `app()->make(TenantResolver::class)` anywhere in your code, you get the same instance. React Context works similarly — you provide a value at the top of the tree, and any descendant can consume it.

The typing concern: when you create a context, you must define its type. The default value (what you get if the context has no provider above you in the tree) creates a tension. If the default is `null` (meaning "no provider"), then every consumer must handle the `null` case. The common pattern is to create a custom hook that throws if the context is null, effectively saying "this hook must be used within a provider." The hook's return type is then the non-null type, which eliminates the null-check boilerplate in every consumer.

This is a design decision. You are choosing between: (a) making the context nullable and forcing consumers to handle it (safer but more verbose), or (b) making the context non-nullable in the custom hook and crashing at runtime if the provider is missing (less verbose but fails hard if misused). For contexts that are always present (like the authenticated user in the dashboard), option (b) is standard. For optional contexts, option (a) is safer.

Generic Components: When the Component Does Not Know the Data Type

Sometimes a component works with data of varying types. A `<DataTable>` might display flags, projects, or users. The columns, the row shape, and the sort logic depend on the data type, but the table chrome (headers, pagination, selection checkboxes) is the same.

This is where generic components earn their keep. A `DataTable<T>` component accepts a type parameter that defines the row shape, and the column definitions, renderers, and event handlers are all typed in terms of `T`. The consumer writes `<DataTable<FlagConfig> data={flags} columns={flagColumns} />` and gets full type safety on the column accessors.

The signal: you are building a reusable UI component that operates on data of an unknown shape, and you want type safety on the data-access parts (column keys, sort fields, filter predicates).

The cost: generic components are more complex to define. If the component is only used with one data type, skip the generic and type it directly.

When Types Get in the Way of UI Code

TypeScript is excellent for data-flow correctness. It is less helpful — and sometimes actively annoying — for highly dynamic UI concerns like animation states, CSS class logic, or imperative DOM manipulation.

The practical advice: type your data rigorously (props, state, context, API responses). For CSS-in-JS values, animation configs, and third-party library interop, accept that the types may be loose and do not fight the compiler to achieve perfect typing on things that provide little safety benefit. A typed component with a loosely-typed className string is fine. A loosely-typed component with a loosely-typed data flow is not.

---

6. Typing Prisma

The Mental Model: "The Type Depends on What You Selected"

> **Coming from Laravel:** Eloquent always gives you a full model. `User::find(1)` returns a `User` with all columns populated (unless you use `select()`). The model class is the type, always.
>
> Prisma is different. The type of the result depends on the query — specifically, on which fields you `select` and which relations you `include`. This is powerful but requires a different mental model.

When you write `prisma.flag.findMany()`, Prisma returns the full `Flag` type with all scalar fields. When you write `prisma.flag.findMany({ select: { key: true, name: true } })`, Prisma returns `{ key: string; name: string }` — a narrower type. When you add `include: { flagEnvironments: true }`, the return type expands to include the related records.

The type is computed from the query. This is not something you see in the PHP world, where ORMs return the same model class regardless of which columns you fetched (with the unselected columns silently set to `null` or throwing on access).

Why This Matters for Flagline

The flag evaluation hot path needs only a few fields: key, enabled, rules, rollout percentage, default value. The dashboard flag list needs different fields: name, description, created date, created-by user name. The flag detail page needs everything plus relations.

If you write a single "get flag" function that returns the full model with all relations, you are over-fetching for most use cases. If you write three separate functions with different `select` clauses, each returns a different type, and the types are correct — Prisma generates them from the query shape.

The thinking process: define your queries based on what the consumer needs, and let Prisma generate the precise return type. Do not fight this by casting everything to a single "Flag" type, because that either requires over-fetching or leads to types that claim fields are present when they are not.

The Gap Between Database Types and API Types

Prisma types reflect the database. API response types reflect what the client sees. These are not the same, and the gap between them is where bugs hide.

A `Flag` from Prisma has `tenantId`, `createdById`, `isArchived`, internal IDs, and timestamps in `Date` format. An API response has a curated subset: the flag key, name, enabled state, and human-readable metadata — with timestamps as ISO strings, without internal IDs, and without tenant information (because the tenant is implicit from the API key).

> **Coming from Laravel:** This is exactly why Laravel has API Resources (the `JsonResource` class). You transform the Eloquent model into a JSON-safe shape, hiding fields, renaming fields, and adding computed fields. The same need exists in TypeScript — the solution is a mapper function with typed input and output.

The pattern: define a mapping function that takes the Prisma type (or a subset via `select`) and returns the API response type. The function signature enforces the contract: `function toFlagResponse(flag: PrismaFlagWithEnvs): FlagResponse`. If the Prisma query changes (adding or removing a `select` field), the mapper either still compiles (because it does not use the removed field) or breaks at compile time (because it needs a field that is no longer selected). This is the safety net.

When to Create DTOs

A DTO (Data Transfer Object) is a type that represents data in transit — between layers, between services, or between your server and a client. The question "when to create DTOs" is really "when does the shape of the data change?"

For Flagline, the shape changes at these boundaries:

1. **Database to service layer.** Sometimes. If the service layer works directly with Prisma types and the query selects exactly what it needs, an extra DTO is redundant. If the service layer combines data from multiple queries or adds computed fields, a DTO clarifies the shape.
2. **Service layer to API response.** Almost always. The API response should be a curated, stable shape that does not leak database internals. Even if it looks identical to the Prisma type today, having a separate type means you can change the database schema without changing the API contract.
3. **API to SDK types.** Always. The SDK types are your public API. They must be deliberately designed, versioned, and documented. They should not reference Prisma, Fastify, or any server-side concern. The `@flagline/types` package exists for this purpose.

The rule of thumb: if two layers could evolve independently (database schema and API contract, API contract and SDK types), give them separate types and a mapper between them. If they always change together (an internal helper function and its caller within the same service), a shared type is fine.

---

7. Avoiding `any` and Common Pitfalls

The Temptation of `any`

Every TypeScript developer reaches for `any` when they are stuck. The compiler is complaining, the deadline is approaching, and `any` makes the red squiggles disappear. This is the TypeScript equivalent of `// @phpstan-ignore-next-line` — it silences the checker without solving the problem.

> **Coming from PHP/Laravel:** In PHP, everything is implicitly `any` unless you add type hints. You have spent five years working without a safety net. The temptation in TypeScript is to recreate that comfort by using `any` liberally. Resist this. The whole reason you moved to TypeScript is the safety net. Using `any` is cutting it.

The Decision Tree When You Are Tempted

When you want to write `any`, ask yourself these questions in order:

1. "Do I know the type but the compiler does not?" This usually means you need to help the compiler with a type annotation or a generic parameter. For example, `JSON.parse()` returns `any` — but you know the shape of the parsed data. The fix is not `any` on your variable; it is parsing through Zod or adding a type assertion *with* a comment explaining why it is safe.

2. "Is this data from outside my application?" External data should be typed as `unknown`, not `any`. The difference is crucial: `any` disables type checking on everything it touches (it is viral — assign `any` to a typed variable and the typed variable loses its type). `unknown` forces you to narrow before using. `unknown` is honest. `any` is a lie.

3. "Am I fighting a third-party library's types?" Sometimes a library's type definitions are wrong or incomplete. In this case, use `any` locally with a `// TODO` comment explaining the problem and linking to the library issue. This is the legitimate use of `any`: a temporary bridge over a gap in someone else's types.

4. "Is the type genuinely too complex to express?" This is rare but real. Some metaprogramming patterns (like Prisma's query-dependent return types) are extremely complex to type. Prisma's own types handle this, but if you are building something similarly dynamic, you may hit the limits of what TypeScript can express. In this case, `any` with a justification comment is acceptable, but consider whether a simpler design would make the types expressible.

The `unknown` –> Narrow –> Use Pattern

This is the core mental model for handling untyped data in TypeScript:

1. **Start with `unknown`.** The data's type is genuinely unknown. Do not pretend otherwise.
2. **Narrow.** Use type guards, schema validation, `typeof` checks, or discriminant checks to determine the actual type.
3. **Use.** After narrowing, the compiler knows the type and you can use it safely.

In Flagline, this pattern appears every time you read from Redis (the cached value is a JSON string that you parse to `unknown`, then validate), every time you read a JSONB column from PostgreSQL (Prisma gives you `JsonValue`), and every time you process a webhook payload.

The pattern is mechanical. Once you internalize it, you stop fighting the compiler and start working with it.

Type Assertions: When You Are Lying to the Compiler

A type assertion (`value as FlagConfig`) tells the compiler: "Trust me, I know this is a `FlagConfig`." The compiler believes you without checking. If you are wrong, you get a runtime error later.

This is fundamentally different from type narrowing (which the compiler verifies) and schema validation (which happens at runtime). A type assertion is an escape hatch — a promise you make that the compiler cannot verify.

When type assertions are justified:

- After runtime validation that the compiler cannot trace. If you validated with Zod and the schema matches the TypeScript type, asserting the validated output is safe. (Better: use `z.infer<typeof schema>` so Zod produces the type directly.)
- In test code, where you are constructing partial mock objects and the full type is inconvenient. Tests are not production code — the bar for type safety is lower.
- When a library's types are overly broad and you know the specific case. For example, `document.getElementById()` returns `HTMLElement | null`, but you know the specific element is an `HTMLInputElement`. Assert with a comment.

When type assertions are dangerous:

- Asserting `as any as DesiredType` — the double assertion is a code smell. It means the types are fundamentally incompatible and you are forcing the compiler to accept it. This almost always indicates a design problem.
- Asserting the type of external data without validation. `(await response.json()) as User` is not type safety — it is a prayer.

The `satisfies` Operator: Check Without Widening

`satisfies` is a relatively recent addition to TypeScript that solves a specific problem: you want to verify that a value matches a type *without* changing the inferred type to the wider one.

The scenario: you define a configuration object for flag evaluation. You want to ensure it matches a `Config` interface (so you do not misspell a key), but you also want TypeScript to infer the *specific literal types* of the values (so that you get autocomplete and narrowing downstream).

Without `satisfies`, you have two options: (a) annotate the variable as `Config`, which widens the value types to what `Config` declares, or (b) skip the annotation, which gives you precise types but no validation that the object matches `Config`.

`satisfies` gives you both: validation against the type and preservation of the precise inferred types. Think of it as a compile-time assertion that does not lose information.

For Flagline, this is useful for configuration objects, lookup tables (like the mapping from plan tier to plan limits), and constant definitions where you want both precision and validation.

---

8. Practical Thinking for This Project

This section walks through the decision process for several of Flagline's core type design challenges. The focus is on the thinking, not the final types — those you can derive once the thinking is clear.

Designing the Flag Evaluation Type System

The flag evaluation pipeline is the core of the product. A request comes in with a context (who is the user, what are their attributes), and a response goes out with the evaluated flag values (what features should they see). The type system should model every possible outcome.

Step 1: Enumerate the outcomes. Before you write any types, list every possible result of evaluating a single flag:

- The flag exists, is enabled, a targeting rule matches, and the corresponding variation's value is returned.
- The flag exists, is enabled, no targeting rules match, the rollout percentage matches, and the rollout value is returned.
- The flag exists, is enabled, no rules or rollout match, and the default value is returned.
- The flag exists but is disabled. The "off" value is returned.
- The flag does not exist. A default/fallback is returned.
- An error occurs during evaluation. A default/fallback is returned.

Each of these outcomes needs to be distinguishable by the consumer. An SDK user who asks "is dark mode enabled?" needs the boolean answer, but they also might want to know *why* — for debugging, for analytics, for understanding unexpected behavior.

Step 2: Choose the discriminant. The `reason` field is the natural discriminant: `"TARGETING_MATCH"`, `"ROLLOUT"`, `"DEFAULT"`, `"FLAG_DISABLED"`, `"FLAG_NOT_FOUND"`, `"ERROR"`. This becomes a discriminated union where each reason carries different metadata.

Step 3: Decide what metadata varies. A `TARGETING_MATCH` knows which rule and which variation matched. A `DEFAULT` does not. If you make `ruleId` and `variationId` optional on a single flat type, the compiler cannot help you — accessing `result.ruleId` after checking `result.reason === "FLAG_NOT_FOUND"` would not be an error, even though `ruleId` is meaningless in that case.

The discriminated union approach means each variant carries exactly the metadata that makes sense for it. The compiler enforces this, and consumers get precise types after a `switch` on the reason.

Step 4: Decide on the value type. Flagline supports boolean, string, number, and JSON flags. The `value` field in the evaluation result could be typed as the union `boolean | string | number | Record<string, unknown>` (which is `FlagValue`), or it could be generic — `EvaluationResult<T extends FlagValue>` — so that the caller can specify "I expect a boolean from this flag."

The generic approach is better for the SDK public API (where the caller knows their flag types) and overkill for the server-side evaluation (where the evaluator processes all flag types uniformly).

Designing API Response Types

Every API endpoint returns either a success or an error. The question is: how do you model this so that consumers can distinguish the two at the type level?

Option A: A single type with optional fields. `{ success: boolean; data?: T; error?: ApiError }`. This is simple but imprecise — the type allows `{ success: true, error: { ... } }`, which is nonsensical.

Option B: A discriminated union. `{ success: true; data: T } | { success: false; error: ApiError }`. This is precise — when `success` is `true`, `data` is present and `error` is not. When `success` is `false`, `error` is present and `data` is not. The compiler enforces this after you check `success`.

Option B is more work to define but pays for itself every time a consumer processes a response. `if (response.success) { /* response.data is T here */ }` — the narrowing is automatic and complete.

For Flagline, the batch evaluation endpoint is particularly important. It returns results for multiple flags, and some might succeed while others fail (a flag might be missing or errored while others evaluated correctly). The per-flag result type should model success and failure. The top-level response type should model the overall request success/failure (like an authentication failure or rate limit).

Designing SDK Public Types

The SDK types are the most consequential type design decision in the project. They are your public API. Once published to npm, changing them is a breaking change that affects every customer.

Principle 1: Minimize the surface area. Export only what consumers need. Internal types (like the structure of the SSE transport layer, or the cache implementation details) should not be in the public type exports. The less you expose, the more freedom you have to change internals.

Principle 2: Prefer simple types. SDK consumers are not TypeScript experts. If your SDK requires the caller to understand generics, discriminated unions, and branded types just to call `getFlag("dark-mode")`, you have over-designed. The simplest possible call signature with the most useful default behavior is the goal.

Principle 3: Separate the essential from the advanced. The basic usage should require no type arguments: `client.getFlag("dark-mode")` returns `FlagValue`. Advanced usage allows a type argument: `client.getFlag<boolean>("dark-mode")` returns `boolean`. The default is permissive, the override is precise.

Principle 4: Do not expose server-side concerns. The SDK types should not import from Prisma, Fastify, or any server-side package. They should be self-contained in `@flagline/types`. If the SDK type file has an import from `@prisma/client`, something has gone wrong architecturally.

Designing Audit Event Types

The audit log records every significant action in Flagline. Each action has a different shape of “before” and “after” data. A `FLAG_CREATED` event has no “before” (the flag did not exist) and its "after" is a flag configuration. A `RULES_UPDATED` event has a "before" (old rules) and "after" (new rules) that are both targeting rule arrays. A `MEMBER_ROLE_CHANGED` event has "before" and "after" that are role strings.

The thinking process:

1. **Can I model this as a discriminated union?** Yes — the `action` field is the discriminant. Each action variant specifies the types of `before` and `after`.
2. **Is the union going to be huge?** Yes — Flagline has 15+ audit actions. A discriminated union with 15 members is large but manageable. The compiler handles it fine. The readability cost is offset by the exhaustive checking benefit.
3. **Do I need this precision everywhere?** No. The audit log *display* in the dashboard probably just shows the JSON diff generically. The audit log *processing* (for webhooks, for alerting) might care about specific event shapes. Model the full union, but use the general shape (with `before: unknown` and `after: unknown`) in the generic display code.
4. **How does this interact with the database?** The `before` and `after` columns are JSONB in PostgreSQL, which means Prisma types them as `JsonValue | null`. You need to validate/assert when reading. The discriminated union type lives in the application layer, not in the Prisma types. The mapper from Prisma type to application type is where the validation happens.

The Shared Types Package: What Goes Where

The `@flagline/types` package should contain types that cross package boundaries. The decision for each type is: "Is this type used by more than one package?"

- **`FlagConfig`, `EvaluationContext`, `EvaluationResult`**: Used by the API, the dashboard, and both SDKs. Belongs in shared types.
- **`ApiResponse<T>`, `ApiError`**: Used by the API (to construct responses) and the dashboard (to consume responses via server actions or API calls). Belongs in shared types.
- **`PlanLimits`, `SubscriptionInfo`**: Used by the API (for enforcement) and the dashboard (for display). Belongs in shared types.
- **Fastify route schemas, Prisma query helpers, React component props**: Used by one package only. Do NOT put these in shared types. They belong in the package that uses them.

The shared types package should be the thinnest possible contract between packages. Every type you add to it creates a coupling point. Keep it lean.

Thinking About Type Evolution

Types change over time. A flag gains a new field. An API response adds pagination. A new evaluation reason is added. How you structure your types determines how painful these changes are.

Additive changes are safe. Adding an optional field to an interface does not break existing code. This is why optional fields are the default for "might be present in the future" data.

Union expansion is safe for producers, breaking for consumers. Adding a new `EvaluationReason` variant does not break code that *produces* evaluation results. But it breaks exhaustive `switch` statements in code that *consumes* them — which is the point. You want the compiler to tell you "there is a new reason and you have not handled it."

Field removal and type narrowing are breaking. Removing a field or changing `string` to `"active" | "inactive"` breaks existing code. These require a major version bump in published packages (the SDKs).

The practical advice: design types for the common case, add optional fields generously, use literal unions instead of broad strings when the set of values is known, and lean on discriminated unions so that new variants force exhaustive handling.

Summary: The Type System as a Design Tool

The biggest mental shift from PHP to TypeScript is not syntax or semantics — it is the realization that types are a design tool, not just documentation.

In PHP, types mostly serve as runtime assertions ("this parameter must be a string") and as IDE hints. The code works without them. You add them for safety and clarity, but the architecture does not depend on them.

In TypeScript, types shape the architecture. A discriminated union does not just document the possible states of a flag evaluation — it *enforces* exhaustive handling. A generic API response type does not just describe the response shape — it *prevents* a handler from returning the wrong data type. A branded ID type does not just label the parameter — it *makes it impossible* to pass a project ID where a tenant ID is expected.

When you design a feature in Flagline, start with the types. Ask: "What are the possible states? What should be impossible to represent? What should the compiler enforce?" Then write the types that answer those questions. The implementation follows naturally.

This is not how most PHP developers think, and that is fine — it is a learned skill. The payoff is that entire categories of bugs become structurally impossible. Not caught by tests, not caught by code review — impossible to write in the first place. For a SaaS where a bug in flag evaluation affects every customer's users simultaneously, that level of confidence is worth the investment.