Contents

What Is All This Stuff? **1**
- The Tools . . . 1
  - pnpm . . . 1
  - Turborepo (turbo) . . . 1
  - Fastify . . . 2
  - Next.js (App Router) . . . 2
  - Prisma . . . 3
  - tsup . . . 3
  - tsx . . . 3
  - Vitest . . . 4
  - Docker Compose . . . 4
- The Folder Structure . . . 4
- The Config Files . . . 4
- How the Pieces Talk to Each Other . . . 5
- The `workspace:*` Pattern . . . 5
- What Happens When You Run `pnpm dev` . . . 6

What Is All This Stuff?

A plain-English guide to every tool, file, and concept in the Flagline monorepo — written for someone coming from the PHP/Laravel world.

The Tools

pnpm

What it is: A package manager — like Composer, but for JavaScript.

Why not npm? npm works fine for single projects, but in a monorepo (multiple projects in one repo), it falls apart. pnpm has first-class support for workspaces — it lets you say "these 8 folders are all separate packages, but they share dependencies and can import each other." It also uses a content-addressable store, which means if five packages all use React, pnpm stores React once on disk and symlinks it everywhere. npm would download five copies.

The Laravel equivalent: `composer install`, but for the entire monorepo at once.

Key file: `pnpm-workspace.yaml` — this tells pnpm "look in `apps/*` and `packages/*` for my workspace packages."

Turborepo (turbo)

What it is: A build orchestrator for monorepos. It does NOT build your code — it tells other tools (like `tsc`, `tsup`, `next build`) when and in what order to build.

The problem it solves: You have 8 packages. Some depend on others. `@flagline/react` depends on `@flagline/js`, which depends on `@flagline/types`. If you change something in `types`, you need to rebuild `js`, then `react`, then the dashboard and API. Turborepo figures out this order automatically.

The killer feature is caching. If you run `turbo build` and nothing changed in `packages/sdk-js` since last time, Turborepo skips it entirely and replays the cached output. This makes builds fast — especially in CI.

The Laravel equivalent: Imagine if `php artisan` knew about all your packages, ran their tests in dependency order, and skipped packages that hadn't changed. That's Turbo.

Key file: `turbo.json` — defines the "pipeline" (which tasks exist, what they depend on, what they output).

Fastify

What it is: A web framework for Node.js — like Laravel's HTTP layer (routing, middleware, request/response), but stripped down and fast.

Why not Express? Express is the Laravel of Node.js — everyone knows it, huge ecosystem, been around forever. But it's old, has clunky async support, and is slower. Fastify is the modern alternative: built-in TypeScript support, automatic JSON serialization that's 2-3x faster, proper async/await error handling, and a plugin system that encourages clean architecture.

Why is Flagline using Fastify instead of Next.js for the API? The evaluation API needs to handle thousands of requests per second, hold open long-lived SSE connections, and scale independently from the dashboard. Next.js API routes run as serverless functions on Vercel — they spin up, handle one request, and die. That doesn't work for SSE (which needs a persistent connection) and adds cold-start latency to flag evaluations. Fastify runs as a long-lived process on Fly.io, always warm, always ready.

The Laravel equivalent: Think of Fastify as a stripped-down Lumen — lightweight, fast, just the HTTP stuff. The dashboard (Next.js) is like your full Laravel app with Blade views. The evaluation API (Fastify) is like a separate Lumen microservice.

Next.js (App Router)

What it is: A React framework that handles routing, server-side rendering, and deployment. It's the full-stack framework — like Laravel, but for React.

Why "App Router"? Next.js has two routing systems. The old one ("Pages Router") works like `routes/web.php` — you define routes. The new one ("App Router") uses the filesystem: create a file at `app/dashboard/page.tsx` and you get a route at `/dashboard`. It also introduced server components (React components that run on the server, like Blade templates) and server actions (functions you call from forms that run on the server, like Laravel controller methods).

The Laravel equivalent: Next.js = Laravel. App Router = file-based routing + Blade + controllers all rolled into the filesystem convention.

Prisma

What it is: An ORM for TypeScript/Node.js — like Eloquent, but with a fundamentally different philosophy.

The key difference from Eloquent: Eloquent is Active Record — `$flag = Flag::find(1)` gives you a model object with methods. Prisma is a query builder that returns plain objects. You write `prisma.flag.findUnique({ where: { id: '1' } })` and get back a plain `{ id, key, name, ... }` JavaScript object. No model class, no magic methods.

The upside: TypeScript knows the exact shape of what you get back, and it changes based on your query. If you `include: { rules: true }`, the return type includes `rules: Rule[]`. If you don't, it doesn't. This is impossible with Eloquent's magic approach.

Key file: `packages/db/prisma/schema.prisma` — this is your single source of truth for the database. It defines all models, relationships, indexes, and enums. Prisma reads this file and generates both the database migrations AND the TypeScript client.

---

tsup

What it is: A TypeScript bundler — it takes your `.ts` source files and produces `.js` files that other packages can import.

Why you need it: TypeScript isn't JavaScript. Node.js and browsers can't run `.ts` files directly. tsup compiles your TypeScript to JavaScript and generates `.d.ts` type declaration files (so consumers get autocomplete and type checking). It outputs both ESM (`import`) and CJS (`require()`) formats for compatibility.

The Laravel equivalent: There isn't one — PHP doesn't have a compile step. The closest analogy is like running `composer dump-autoload`, but instead of just generating an autoloader, it's actually translating your code from one language to another.

Where it's used: Every package that gets imported by other packages: `shared-types`, `db`, `sdk-js`, `sdk-react`, and the `api` build.

---

tsx

What it is: A tool that runs TypeScript files directly — no build step needed. Used for development only.

Why? During development, you don't want to rebuild after every change. `tsx watch src/index.ts` runs your TypeScript file directly and restarts when you save. It's like `php artisan serve` — fast feedback loop for development.

Where it's used: `apps/api` dev script and running the Prisma seed script.

Vitest

What it is: A test runner — like PHPUnit, but for TypeScript/JavaScript. It's fast, understands ESM, and has a Jest-compatible API. If you've used Jest before, Vitest is the same API but faster (powered by Vite's transform pipeline).

Where it's used: `sdk-js`, `sdk-react`, and `api` for unit and integration tests.

Docker Compose

What it is: You probably know this one. It runs PostgreSQL and Redis locally in containers so you don't have to install them on your machine.

The Laravel equivalent: Laravel Sail is literally Docker Compose with a wrapper script. Our `docker-compose.yml` is Sail without the wrapper — just Postgres and Redis, nothing else.

The Folder Structure

```
Flagline/
├── apps/
│   ├── dashboard/        → The Next.js SaaS dashboard (deployed to Vercel)
│   └── api/              → The Fastify evaluation API (deployed to Fly.io)
├── packages/
│   ├── shared-types/     → TypeScript types shared across everything
│   ├── db/               → Prisma schema, client, and migrations
│   ├── sdk-js/           → @flagline/js — the core SDK (published to npm)
│   ├── sdk-react/        → @flagline/react — React wrapper (published to npm)
│   ├── config-eslint/    → Shared linting rules
│   └── config-typescript/→ Shared tsconfig base files
├── docs/                 → The reference documents you're reading
└── .github/workflows/    → CI/CD pipelines
```

`apps/` vs `packages/`: Apps are things that get deployed (a website, an API server). Packages are things that get imported by apps or by other packages. Apps are entry points. Packages are libraries.

The Config Files

File	What it does
`package.json` (root)	Defines workspace scripts (`pnpm dev`, `pnpm build`) and root dev dependencies (Prettier, Turbo)
`pnpm-workspace.yaml`	Tells pnpm where to find workspace packages

File	What it does
`turbo.json`	Defines the build pipeline — task ordering, caching, outputs
`.nvmrc`	Pins the Node.js version (20) so everyone uses the same one
`.prettierrc`	Code formatting rules — like Laravel Pint but for JS/TS
`.gitignore`	Keeps `node_modules/`, `.next/`, `dist/`, `.env` out of git
`.env.example`	Template for environment variables — copy to `.env.local`
`docker-compose.yml`	Runs Postgres and Redis locally

How the Pieces Talk to Each Other

```
Browser (your customer's app)
  └─ @flagline/js SDK
       └─ HTTP requests + SSE connection
            └─ Fastify Evaluation API (apps/api)
                 ├─ Redis (cache + pub/sub)
                 └─ PostgreSQL (source of truth)

Browser (your dashboard users)
  └─ Next.js Dashboard (apps/dashboard)
       ├─ PostgreSQL (via Prisma directly — no API needed)
       ├─ Redis (for publishing flag changes)
       └─ Stripe (billing webhooks)
```

The dashboard talks directly to the database (server components can call Prisma). The evaluation API also talks to the database, but primarily reads from Redis cache. When someone toggles a flag in the dashboard, the dashboard writes to Postgres and publishes a message to Redis. The evaluation API subscribes to that Redis channel and pushes the change to all connected SDKs via SSE.

The `workspace:*` Pattern

In each package's `package.json`, you'll see dependencies like:

```
"@flagline/db": "workspace:*"
```

This is pnpm's way of saying "import this from the monorepo, not from npm." It's like a Composer path repository (`"repositories": [{"type": "path", "url": "../packages/db"}]`) but built into the package manager. When you publish to npm, `workspace:*` gets replaced with the actual version number.

What Happens When You Run `pnpm dev`

1. Turborepo reads `turbo.json` and sees `dev` is a persistent task with no cache.
2. It starts all workspace `dev` scripts in parallel:
   - `apps/dashboard`: `next dev` — starts the Next.js dev server on port 3000
   - `apps/api`: `tsx watch src/index.ts` — starts the Fastify server on port 4000
   - `packages/shared-types`: `tsup --watch` — watches for type changes, rebuilds
   - `packages/db`: `tsup --watch` — watches for changes, rebuilds
3. You open `http://localhost:3000` for the dashboard, and the SDK points to `http://localhost:4000` for flag evaluation.

That's it. One command, everything runs.