Brief (what you’re building)

A React web app that evaluates a user-entered FPL squad (11 + bench + bank) without login. It computes a single **next-3-GW score** per player from xG, xA, form, fixture difficulty, and expected minutes; labels each player **perfect / good / fine / bad**; and suggests 1–3 budget/club-cap-compliant replacements for every non-perfect pick. Data is kept fresh via a backend cache that regularly pulls from FPL + advanced stats sources.

Step-by-step development plan

0) Repo & tooling

- Monorepo (or single repo) with `apps/web` (React) and `apps/api` (Node/Next.js API Routes or Express).
- **Stack**: TypeScript everywhere, pnpm, ESLint + Prettier, Vitest/Jest, Playwright.
- Scripts: `dev`, `build`, `test`, `lint`, `typecheck`.
- Env: `.env` with `NODE_ENV`, `REDIS_URL` (or file cache), `CRON_SECRET`.

Acceptance: `pnpm dev` runs both API and web, health endpoint returns 200.

---

1) Data layer (server) — fetching & caching

Implement a cache-first fetcher for:

- **FPL bootstrap** (players, teams, prices, status, form), **fixtures** (includes difficulty).
- **Advanced stats** (xG/xA per player). Start with one source; add adapters later.

Modules

- `lib/cache.ts`: `get(key)`, `set(key, data, ttl)`. (Redis or filesystem JSON for MVP)
- `lib/fetchers/fpl.ts`: `getBootstrap()`, `getFixtures()`.
- `lib/fetchers/advanced.ts`: `getPlayerAdvanced(playerRef)` returning xG/xA rates (per 90 or last N).
- `lib/merge.ts`: joins FPL and advanced stats (see identity resolution below).

Identity resolution (FPL ⇄ advanced stats)

- Normalize: lowercase, strip accents/punctuation.
- Match by `(team, normalizedName)` first; fallback to fuzzy ratio ≥ threshold.
- Keep a `manual_overrides.json` for edge cases.

Fixture ease

- For each player: next 3 fixtures for their team; compute `ease = avg(6 - FDR)` so higher = easier.

Expected minutes

- Rolling last 5 GWs minutes per player (from FPL history if available), cap at 90, discount if flagged:
    - `status: a=1.0, d=0.8, i/s=0.4`.
    - If recent mins < 45 avg, clamp `expMin` down.

Acceptance

- `/api/players` returns a list with `{id, name, team, pos, price, form, status, xg90, xa90, expMin, next3Ease}`.
- `/api/fixtures` returns team fixtures with FDR for at least next 3 GWs.

---

2) Scoring engine

Goal: single value per player within position.

```
// weights can be user-tuned from UI later
const W = { xg: 0.35, xa: 0.20, form: 0.20, fix: 0.15, min: 0.10 };

score(p) = W.xg * n_pos(xg90) +
        W.xa * n_pos(xa90) +
        W.form * n_pos(form) +
        W.fix * n_pos(next3Ease) +
        W.min * n_pos(expMin/90)
all multiplied by availability penalty from status (as above)
```

- `n_pos(.)` = min-max normalize **within position (GK/DEF/MID/FWD)**.
- Persist last computed scores to cache with timestamp.

Acceptance: `/api/score-table` returns `{ id, pos, score }` for all players.

---

3) Squad model & validation

Input (user pastes JSON or uses form):

```
{
  "bank": 1.3,
  "startingXI": [{"id": 1,"pos":"GK","price":4.5}, ...],
  "bench": [{"id": 15,"pos":"FWD","price":4.5}]
}
```

Validation

- 15 players, valid formation in XI (1 GK, 3–5 DEF, 2–5 MID, 1–3 FWD).
- ≤ 3 from any real club.
- Prices match FPL price list (warn if mismatch; trust server prices).

Acceptance: `/api/validate-squad` returns `{ok: true}|{ok:false, errors:[...]}`.

---

4) Labelling players (perfect/good/fine/bad)

Compute:

- **Rank** each player within position by `score`.
- **Best affordable delta** for each owned player (see next section).

Rules (tweakable)

- **Perfect**: rank ≤ 15% **and** best affordable delta < 1.5 (next-3 pts).
- **Good**: rank ≤ 30% **or** delta < 1.5.
- **Fine**: delta 1.5–3.0.
- **Bad**: rank ≥ 70% **or** delta > 3.0 **or** flagged (`i/s`).

Acceptance: `/api/analyze` (POST squad) returns per-slot `{ current:{id,score}, label }`.

---

5) Suggestions engine (per player)

Constraints:

- Same position, price ≤ `slot.price + bank`, obey 3-per-club considering whole squad after swap.

Algorithm:

1. Build `ownedIds`, `clubCounts`.
2. Filter candidates by constraints (skip already owned).
3. Rank by `delta = candidate.score - current.score`.
4. Return top 1–3 with `{id, name, price, delta}` where `delta > 0`. If only one qualifies, return one.

Edge cases

- If 0 candidates: show “no better option within budget/club limits.”
- Tie-breaking: prefer higher expected minutes, then lower price.

Acceptance: `/api/suggestions` (POST squad) returns `{ slotId -> [suggestions] }`.

> **Stretch (v2)**: 1–2 transfer bundle optimization via ILP to maximize total XI score, optional -4 hit cost.

---

6) API design (server)

- `GET /api/health` → `{ok:true}`
- `GET /api/players` → enriched players (cached)
- `GET /api/fixtures` → fixtures (cached)
- `GET /api/score-table` → `{id, pos, score}`
- `POST /api/validate-squad` → validation result
- `POST /api/analyze` → labels + scores for each of the 15
- `POST /api/suggestions` → suggestions per non-perfect slot
- `POST /api/analyze-and-suggest` → single call returning {labels, suggestions, meta}

Errors: structured `{error:{code,message,details}}`.

---

7) Data freshness (critical)

- **Cron**: hourly during active weeks, every 6–12h otherwise.
- **Invalidations**: force refresh at GW deadline + after matchdays.
- **Manual refresh**: protected route `POST /api/admin/refresh?token=…`.
- **TTL**: players 1h, fixtures 6h, scores recompute after any underlying update.

Acceptance: logs show scheduled refreshes; `updatedAt` fields change.

---

8) Web UI (React + TS)

Pages

- `/` Input squad:
  - Searchable player picker (server returns name → id), or paste JSON.
  - Bank input, weight sliders (persist to localStorage).
  - "Analyze" button → calls `/api/analyze-and-suggest`.
- `/analyze` Results:
  - Summary header: average score per position, flagged players count, bank left.
  - Table of 15 players:
    - Columns: Name, Team, Pos, Price, Score, Label (colored pill).
    - Row details (accordion): if not perfect → show 1–3 suggestions with delta, add-to-clipboard.
    - Fixture chips: next-3 FDR shown as small badges.

UX details

- Loading skeletons, toasts on errors, sticky "Bank" & "Weights" panel.
- Copyable "OUT → IN" suggestions.

Acceptance: With mock data, shows labels & suggestions, handles empty states.

---

9) Testing

- **Unit**: scoring normalization, label thresholds, suggestion filtering (budget/club caps).
- **Integration**: API endpoints with fixture player pools (mock fetchers).
- **E2E** (Playwright): paste sample squad → see consistent labels & suggestions.
- **Property tests**: never suggest already-owned players; never exceed club caps; price never > `slot.price+bank`.

Acceptance: CI passes `pnpm test` and e2e smoke.

---

10) Performance & reliability

- Server cache layer in front of external fetches; exponential backoff on failures.
- Rate-limit public endpoints; CORS allowlist for your domain.
- Request validation with Zod/Yup.
- Observability: simple request logging; optionally Sentry.

Acceptance: Cold start < 1s for cached reads; analyze request p95 < 300ms on 10k player pool.

---

11) Deployment

- **API**: Vercel/Render/Fly; add Cron (Vercel Cron or GitHub Action + webhook).
- **Web**: Vercel/Netlify static.
- Secrets in platform UI; production/staging environments.

Acceptance: Public URL with working analysis; admin refresh secured.

---

12) Docs & admin

- `README.md`: setup, run, env, API docs, data update policy.
- `docs/weights.md`: what each weight means, suggested defaults.
- `docs/mapping.md`: player name matching rules + overrides.

---

13) Roadmap (post-MVP)

- Multi-GW planning & transfer sequencing (+ hit cost).
- Chip strategy (Wildcard, Free Hit, BB, TC).
- Fixture difficulty from multiple models; opponent-specific xG/xA conceded.
- Personalized risk profiles; captaincy recommendation.
- “What-if” mode (simulate 1–2 transfers and compare XI score).

Suggested file structure (Cursor-friendly)

```
/apps
  /api
    /src
      lib/cache.ts
      lib/fetchers/{fpl.ts,advanced.ts}
      lib/merge.ts
      lib/scoring.ts
      lib/squad.ts
      routes/{players.ts,fixtures.ts,analyze.ts,suggestions.ts}
  /web
    src/
      pages/{index.tsx,analyze.tsx}
      components/{SquadForm.tsx,PlayerRow.tsx,WeightsPanel.tsx,FixtureChips.tsx}
      lib/{api.ts,types.ts,format.ts}
```

```
state/{useSquad.ts,useWeights.ts}
```

Sample contracts (to copy into types)

```
// shared types
export type Pos = 'GK'|'DEF'|'MID'|'FWD';

export type EnrichedPlayer = {
  id:number; name:string; teamId:number; teamShort:string; pos:Pos; price:number;
  form:number; status:'a'|'d'|'i'|'s'; xg90:number; xa90:number;
  expMin:number; next3Ease:number; score?:number;
};

export type SquadSlot = { id:number; pos:Pos; price:number };
export type Squad = { startingXI:SquadSlot[]; bench:SquadSlot[]; bank:number };

export type Suggestion = { id:number; name:string; price:number; delta:number };
```

“Definition of Done” for MVP

- User can input squad (or paste JSON), set bank, click Analyze.
- App displays per-player **score + label** and **1–3 suggestions** per non-perfect slot.
- Data reflects last sync (< 1–2h old) and can be manually refreshed.
- All constraints respected (budget, formation, club caps), with tests proving it.
