High-Level Design: Premier Roster (Expo POC)

Premier Roster is a lightweight mobile app built with React Native using Expo. It allows users to view English Premier League teams and detailed player information for the 2024/25 season. The core goal was to create a lightweight, cleanly structured mobile app that showcases the ability to integrate with a third-party API, manage local data efficiently, and provide a solid user experience without over-engineering.

Why Expo Was Chosen

Expo was selected because it offers faster project setup and makes configuring continuous deployment with EAS simple and efficient — ideal for delivering a quick and clean POC.

Project Structure & Navigation

The app is intentionally simple — it has two main screens:

• /teams – displays all Premier League teams.

• /team – displays the squad for a selected team. Takes params teamName, teamId, league and season to fetch and display the correct data.

Given the minimal shared state between them, I decided not to include global state libraries like Redux or Zustand — simple useState and useEffect were sufficient.

The project uses expo-router for file-based navigation, making the routing structure clean and easily scalable.

API Integration with Type Safety

To interact with the API-Football service, the project uses openapi-typescript-codegen. This tool generates a fully typed client from a local OpenAPI YAML spec (api-football.yaml) and eliminates manual endpoint handling. This ensures:

• Compile-time safety when calling APIs.

• Auto-generated service methods and model types under src/api/generated.

• Easy maintainability if the API evolves.

The client is generated using the script:

npm run api:gen

It uses the native fetch transport to remain compatible with Expo and requires no native modules.

Local Storage Strategy with MMKV

The football API is rate-limited and paid, so avoiding unnecessary requests is critical. I implemented a manual caching strategy using MMKV that avoids the complexity of redux-persist and gives full control over how and when data is stored.

Teams List:

• Fetched once per season.

• Saved as teams-${league}-${season}.

• Never refreshed during the season, since participating clubs are fixed.

Team Squad:

• Fetched per team on demand.

• Saved as players-${teamId}-${league}-${season}.

• A timestamp is saved alongside each squad (timestamp-${teamId}-${league}-${season}).

• On each request, the timestamp is checked. If older than 30 days, the data is refreshed automatically.

This approach minimizes API usage, keeps the experience fast, and future-proofs the app to support multiple leagues/seasons with predictable and clean cache keys.

UI Layer & Theming

The interface is built using Gluestack UI, a component library that provides styled components out-of-the-box. Since this is a POC without any formal design requirements, Gluestack helped me:

• Quickly scaffold clean, responsive UI components.

• Maintain consistent layout and styling.

• Focus on functionality without getting blocked on UI design.

CI/CD, Environment, and Secrets

The app uses EAS (Expo Application Services) for both CI and deployment. Build and deployment environments are managed using eas.json with support for:

• Local and cloud secrets via eas env:create (e.g., API_FOOTBALL_KEY, FOOTBALL_API_BASE_URL).

• Production, preview, and development build profiles.

• Automatic version bumping for production builds.

• Secure injection of environment variables into app.config.js.

Secrets are not hardcoded but passed using EAS environments for safe, flexible builds.

Code Quality & Development Tooling

Linting:

• Uses ESLint with eslint-config-expo, customized via eslint.config.js.

• Ignores generated code (src/api/generated/**) and build output (dist/).

• Linting can be executed via expo lint and is part of pre-commit hooks and CI.

Formatting:

• Uses Prettier for consistent code formatting.

• Scripts:

– npm run format – formats all files.

– npm run format:check – checks format without applying changes.

• Integrated with ESLint via eslint-config-prettier and eslint-plugin-prettier.

Testing:

• Jest (jest-expo) with React Native Testing Library and jest-native.

• Scripts:

– npm run test

– npm run test:watch

– npm run test:verbose

• Config in jest.config.js and jest.setup.js.

Pre-commit Hooks:

• Uses Husky + lint-staged.
• Automatically runs ESLint and Prettier to ensure clean commits.

CI Flow

Based on eas.json and the current structure, the app supports these flows:

Continuous Integration:

• Code quality checks: ESLint, Prettier, TypeScript validation.
• Automated tests: Unit and component-level via Jest.
• Build validation: EAS prebuild and development client testing.

Continuous Deployment:

• Production builds triggered from the main branch when a PR is merged.
• All builds use secrets injected via --env to avoid leaking keys.