Repository Structure Summary

The Track & Field Training Platform repository follows a modern React application structure, divided into web and mobile applications. The web directory contains the primary codebase, structured around a Vite-React application with TypeScript. It employs a component-based architecture organized by technical function rather than business domain, with distinct directories for pages, components, services, hooks, and utilities. Key files are quite substantial, with several components exceeding 500 lines of code. The API layer is centralized in a large service file, while UI components are primarily built with Chakra UI. Database operations are managed through Supabase, with migration scripts stored in a dedicated directory. While the architecture provides clear separation of concerns at a high level, it shows signs of needing refactoring to modularize larger components and transition to a more domain-driven structure.

Cleanup Tasks for Files Over 300 Lines

Profile.tsx (1297 lines)

[ ] Extract avatar upload functionality into a separate component

[ ] Create a separate ProfileForm component with its own state management

[ ] Move debug-related functionality into a utility module

[ ] Extract validation logic into a standalone utility

[ ] Split view and edit modes into separate components

[ ] Create dedicated components for profile sections (personal info, athletic details, etc.)

Dashboard.tsx (1029 lines)

[ ] Extract card components into reusable components

[ ] Move data fetching logic into custom hooks

[ ] Create separate components for different dashboard sections (metrics, workouts, progress)

[ ] Extract chart-related code into specialized visualization components

[ ] Move utility functions to a dedicated utilities file

Workouts.tsx (988 lines)

[ ] Extract workout form into a standalone component

[ ] Create a separate component for workout listings

[ ] Split workout detail view into its own component

[ ] Move validation and transformation logic to utility functions

[ ] Create a custom hook for workout data management

api.ts (856 lines)

[ ] Separate into domain-specific modules (workouts, profile, teams, etc.)

[ ] Create type definition files for API responses

[ ] Extract error handling into a utility function

[ ] Create a base API client with common functionality

[ ] Add documentation for each API function

Navigation.tsx (399 lines)

[ ] Extract mobile navigation into its own component

[ ] Create a separate component for user menu

[ ] Extract navigation items into a configuration file

[ ] Create role-specific navigation components

[ ] Move authentication-related logic to a custom hook

WorkoutModal.tsx (381 lines)

[ ] Split into smaller form components

[ ] Extract exercise management into a dedicated component

[ ] Move validation logic to utility functions

[ ] Create reusable form field components

[ ] Extract state management into a custom hook

Events.tsx (574 lines)

[ ] Extract event listings into a separate component

[ ] Create dedicated components for filters and search

[ ] Move data transformation logic to utilities

[ ] Create a component for event details

[ ] Extract form handling into its own component

NPM Scripts

# NPM Scripts

| Script | Description |

|--------|-------------|

| `dev` | Starts the development server using Vite with hot reloading enabled |

| `build` | Compiles and bundles the application for production deployment |

| `lint` | Runs ESLint to check for code quality issues |

| `preview` | Serves the production build locally for testing before deployment |

Updated NPM Scripts Documentation

# NPM Scripts

| Script | Description |

|--------|-------------|

| `dev` | Starts the development server using Vite with hot reloading enabled |

| `build` | Compiles and bundles the application for production deployment |

| `lint` | Runs ESLint to check for code quality issues |

| `preview` | Serves the production build locally for testing before deployment |

```## Updated NPM Scripts Documentation

```markdown

# NPM Scripts

| Script | Description | NPM Scripts

| Script | Description |-----|-------------|

| `dev`rver with | Starts the Vite development server with replacement (HMR) |unslation and builds the production TypeScript compilation and builds the productioneck for code quality issues Runs ESLint to check for code quality issues coding standards |tion buil

| `preview` | Serves the production buil

```

##d locally for testing before deployment |e patterns observed in the repository, here are 5 meaningful ESLint rules to add:

1. \*\*`max-lines-per-function: ['error', { max: 50 }]`\*\*: Enforces a reasonable function size limit to encourage breaking down large components into smaller, more manageable pieces.

2. \*\*`react-hooks/exhaustive-deps: 'warn'`\*\*: Ensures useEffect hook dependencies are correctly specified to prevent unexpected behavior and memory leaks.

3. \*\*`no-unused-vars: ['error', { argsIgnorePattern: '^\_', varsIgnorePattern: '^\_' }]`\*\*: Prevents unused variables but allows intentionally unused variables when prefixed with underscore.

4. \*\*`complexity: ['error', 10]`\*\*: Limits cyclomatic complexity to encourage simpler, more maintainable functions with fewer decision points.

5. \*\*`import/order: ['error', { groups: ['builtin', 'external', 'internal', 'parent', 'sibling', 'index', 'object'] }]`\*\*: Enforces consistent import ordering to improve code readability and organization.

These rules address the primary issues identified in the codebase: overly large components, potential hook dependency issues, and code organization. Implementing these would significantly improve code quality and maintainability.

Suggested ESLint Rules

Based on the code patterns observed in the repository, here are 5 meaningful ESLint rules to add:

max-lines-per-function: ['error', { max: 50 }]: Enforces a reasonable function size limit to encourage breaking down large components into smaller, more manageable pieces.

react-hooks/exhaustive-deps: 'warn': Ensures useEffect hook dependencies are correctly specified to prevent unexpected behavior and memory leaks.

no-unused-vars: ['error', { argsIgnorePattern: '^\_', varsIgnorePattern: '^\_' }]: Prevents unused variables but allows intentionally unused variables when prefixed with underscore.

complexity: ['error', 10]: Limits cyclomatic complexity to encourage simpler, more maintainable functions with fewer decision points.

import/order: ['error', { groups: ['builtin', 'external', 'internal', 'parent', 'sibling', 'index', 'object'] }]: Enforces consistent import ordering to improve code readability and organization.

These rules address the primary issues identified in the codebase: overly large components, potential hook dependency issues, and code organization. Implementing these would significantly improve code quality and maintainability.

- [ ] Extract helpers and split \*\*Profile.tsx\*\* (1,297 LOC) into smaller subcomponents

- [ ] Break \*\*Dashboard.tsx\*\* (1,029 LOC) into isolated layout and feature components

- [ ] Refactor \*\*Workouts.tsx\*\* (988 LOC) to separate presentation from logic (e.g., custom hook, context)

- [ ] Split \*\*services/api.ts\*\* (856 LOC) into domain modules (`workouts.ts`, `profile.ts`, `teams.ts`, etc.)

- [ ] Simplify \*\*Events.tsx\*\* (574 LOC) by extracting header, list, and filter components

- [ ] Refactor \*\*Navigation.tsx\*\* (399 LOC) to extract menu and sidebar subcomponents

- [ ] Decompose \*\*WorkoutModal.tsx\*\* (381 LOC) into form, validation, and UI pieces