HockeyStatsTs 2.0: Master Engineering Blueprint

Detailed Component Logic Specification

Lead Architect

December 29, 2025

Contents

1 Architectual Foundation **2**

1.1 Core Principles 2

1.2 Directory Structure 2

2 Phase 1: Environment Tooling **3**

2.1 Step 1.1: Initialization 3

2.2 Step 1.2: Strict Configuration 3

3 Phase 2: Domain Layer Design **5**

3.1 Step 2.1: Value Objects 5

3.1.1 2.1.1 ActionType (Enum) 5

3.1.2 2.1.2 Position (Enum) 5

3.1.3 2.1.3 TeamColors (Value Object) 5

3.2 Step 2.2: Entities 6

3.2.1 2.2.1 Player Entity 6

3.2.2 2.2.2 GameAction Entity 6

3.2.3 2.2.3 Game Entity (The Root Aggregate) 7

3.3 Step 2.3: Domain Services 8

3.3.1 2.3.1 StatCalculator Service 8

3.4 Step 2.4: Repository Interfaces 9

4 Phase 3: Infrastructure Layer **10**

4.1 Step 3.1: Mappers 10

4.2 Step 3.2: Firebase Repository Implementation 11

5 Phase 4: Application Use Cases **12**

5.1 Step 4.1: RecordActionUseCase 12

6 Phase 5: Presentation Layer **13**

6.1 Step 5.1: State Management (Zustand) 13

6.2 Step 5.2: Components Refactoring 14

6.2.1 5.2.1 RinkBoard Component (Organism) 14

6.2.2 5.2.2 GamePage (Controller) 14

7 Phase 6: Quality Assurance **16**

7.1 Step 6.1: Unit Testing Domain Entities 16

Chapter 1

Architectural Standards Conventions

1.1 The "Why" of Clean Architecture

We are strictly separating the **Core (Domain)** from the **Tools (React/Firebase)**.

> **Architectural Rationale:** *In the previous version, game logic was mixed inside React components (e.g., calculating scores inside ‘GamePage‘). This makes testing impossible without rendering a UI. By moving logic to the Domain, we can test complex game rules (offside, overtime logic, stats) in milliseconds using pure TypeScript tests.*

1.2 Strict Coding Rules

- **No Logic in UI**: React components must only *display* data and *dispatch* events.
- **No Frameworks in Domain**: The ‘domain/‘ folder must not contain ‘import ... from ’react’‘ or ‘’firebase’‘.
- **One Way Data Flow**: UI → Use Case → Domain → Repository → Store → UI.

Chapter 2

Domain Layer Specification

2.1 Value Objects

Immutable objects that define the "shape" of data.

2.1.1 TeamColor

Path: ‘src/domain/value-objects/TeamColor.ts‘

```
                export class TeamColor {
                        constructor(
                        public readonly primary: string,
                        public readonly secondary: string
                        ) {
                                if (!/^#[0-9A-F]{6}$/i.test(primary)) throw new Error("Invalid Primary Hex");
                                if (!/^#[0-9A-F]{6}$/i.test(secondary)) throw new Error("Invalid Secondary Hex");
                        }
                }

```

Architectural Rationale: *Validating colors on instantiation prevents "broken UI" states where a team has no valid color to render on the scoreboard.*

2.1.2 GameClock

Path: ‘src/domain/value-objects/GameClock.ts‘

```
                export class GameClock {
                        constructor(
                        public readonly period: number,
                        public readonly secondsRemaining: number
                        ) {}

                        get formattedTime(): string {
                                const m = Math.floor(this.secondsRemaining / 60);
                                const s = this.secondsRemaining % 60;
                                return ‘${m}:${s < 10 ? ’0’ : ’’}${s}‘;
                        }
                }

```

2.2 Entities

2.2.1 Game Entity

Path: ‘src/domain/entities/Game.ts‘ The aggregate root. It must enforce the rules of Hockey.

```
    export class Game {
        // ... properties ...

        public recordGoal(playerId: string, assistIds: string[], time: number): Game {
            // Rule: Cannot score if game is over
            if (this.isFinalized) throw new Error("Game is finalized");

            // Rule: A player cannot assist their own goal
            if (assistIds.includes(playerId)) throw new Error("Player cannot assist self");

            // Logic to create action...
            return this.cloneWithNewAction(newAction);
        }
    }

```

Chapter 3

Presentation: Component Catalog

We utilize **Atomic Design** to organize components.

3.1 Atoms (Base UI)

Single-responsibility, dumb components.

3.1.1 Button

Props:

- ‘variant‘: ’primary’ | ’secondary’ | ’danger’
- ‘size‘: ’sm’ | ’md’ | ’lg’
- ‘isLoading‘: boolean
- ‘icon¿: ReactNode

> **Architectural Rationale:** *Centralizing buttons ensures we can change the entire app’s look (e.g., border-radius, shadows) in one file.*

3.1.2 StatBadge

Props: ‘label‘: string, ‘value‘: number, ‘trend¿: ’up’ | ’down’ | ’neutral’. **Usage**: Displays things like "Shots: 24" on the game dashboard.

3.2 Molecules (Composite UI)

Combinations of atoms functioning together.

3.2.1 PlayerListItem

Path: ‘src/presentation/components/molecules/PlayerListItem.tsx‘ **Purpose**: Displays a single row in the roster table. **Props**:

- ‘player‘: Player (Domain Entity)
- ‘isSelected‘: boolean
- ‘onToggle‘: (id: string) => void

Why: We need this reusable component because player selection happens in multiple places: creating a team, setting a game roster, and filtering stats.

3.2.2 ActionLogItem

Path: ‘src/presentation/components/molecules/ActionLogItem.tsx‘ **Purpose**: A single row in the "Play-by-Play" feed. **Props**: ‘action‘: GameAction, ‘homeTeamId‘: string. **Logic**:

- If ‘action.type === ’GOAL’‘, render with gold background.
- If ‘action.teamId === homeTeamId‘, align left. Else, align right.

3.3 Organisms (Complex Business Components)

Distinct sections of an interface.

3.3.1 RinkInteractive

Path: ‘src/presentation/components/organisms/RinkInteractive.tsx‘ **Purpose**: The core input method for game logging. **Props**:

- ‘rinkImageSrc‘: string
- ‘actions‘: GameAction[] (to display markers)
- ‘onMapClick‘: (x: number, y: number) => void
- ‘isReadOnly‘: boolean

> **Architectural Rationale:** *This component must be decoupled from the ‘Game‘ entity. It simply takes coordinates and visual data. It doesn’t "know" who scored, it just renders dots.*

3.3.2 TeamRosterEditor

Path: ‘src/presentation/components/organisms/TeamRosterEditor.tsx‘ **Purpose**: Complex form to add/edit/remove players from a team. **State**: Local state for the "Add Player" modal. **Props**: ‘team‘: Team, ‘onAddPlayer‘: (p: PlayerDTO) => void, ‘onRemovePlayer‘: (id: string) => void.

3.4 Templates (Layouts)

3.4.1 GameDashboardLayout

Structure:

- **Header**: ScoreBoard (Fixed at top)
- **Left Col**: RinkInteractive
- **Right Col**: ActionFeed + StatPanel (Tabs)
- **Footer**: GameControls (Pause, Finalize)

> **Architectural Rationale:** *Hockey games need a specific layout where the Rink is the focal point. The standard ‘MainLayout‘ with a sidebar is too distracting for live game logging.*

Chapter 4

Presentation: Page Specifications

4.1 Page 1: GamePage (The Core)

Route: ‘/game/:gameId‘

4.1.1 Responsibilities

- Hydrate ‘useGameStore‘ with the game ID from URL.
- Handle the "Game Loop" (autosave, clock ticking if needed).
- Coordinate the ‘ActionCreationWizard‘ (the modal flow).

4.1.2 The "Action Creation Wizard" Flow

This page manages a complex state machine for creating an action. It’s not just a form; it’s a multi-step process.

1. **Step 1: Click Rink** → Save (x,y) → Open Modal.
2. **Step 2: Select Action Type** (Shot, Goal, Hit).
3. **Step 3: Select Player** (Filtered by Team).
4. **Step 4: (If Goal) Select Assists** (Multi-select, excluding scorer).
5. **Step 5: Confirm**.

Code Structure for Wizard:

```
// Inside GamePage.tsx
const [wizardStep, setWizardStep] = useState<WizardStep>(’IDLE’);
const [draftAction, setDraftAction] = useState<Partial<GameAction>>({});

const handleRinkClick = (coords) => {
        setDraftAction({ x: coords.x, y: coords.y });
        setWizardStep(’SELECT_TYPE’);
};

const handleTypeSelect = (type) => {
        setDraftAction(prev => ({ ...prev, type }));
        setWizardStep(’SELECT_PLAYER’);
};

```

> **Architectural Rationale:** *We use a local state machine for the wizard because this data is ephemeral. It only becomes "Domain Data" when the user hits Confirm.*

4.2 Page 2: TeamManagementPage

Route: ‘/teams/:teamId/manage‘

4.2.1 Components Used

- ‘TeamHeader‘ (Logo, Name, Colors)
- ‘TeamRosterEditor‘ (Organism)
- ‘TeamStatsSummary‘ (Molecule)

4.2.2 Data Requirements

Must load the Team entity AND all Players associated with it. **Why**: We need to edit player details (Jersey) in the context of the team.

Chapter 5

Application Layer: Use Cases

5.1 Game Use Cases

5.1.1 InitializeGameUseCase

Input: ‘homeTeamId‘, ‘awayTeamId‘, ‘season‘, ‘gameType‘. **Logic**:

1. Fetch Home and Away ‘Team‘ entities to ensure they exist.
2. Create a new ‘Game‘ entity with a UUID.
3. Set initial state (0-0 score, period 1).
4. Save to Repository.

5.1.2 ProcessActionUseCase

Input: ‘gameId‘, ‘RawActionDTO‘ (from UI). **Logic**:

1. Load Game.
2. Call ‘game.validateAction(rawAction)‘.
3. Call ‘game.addAction(rawAction)‘.
4. **Side Effect**: Check if the action triggers a stoppage (optional rule).
5. Save Game.

5.2 Team Use Cases

5.2.1 CreateTeamUseCase

Logic:

1. specific validation: Check if name already exists in DB (async check).
2. validation: Check if colors contrast sufficiently (accessibility).
3. Save.

Chapter 6

State Management

We split the store to avoid re-renders.

6.1 useGameStore

Path: ‘src/presentation/store/useGameStore.ts‘

```
                interface GameStore {
                        // State
                        activeGame: Game | null;
                        isSyncing: boolean;
                        lastSaved: Date | null;

                        // Actions
                        loadGame: (id: string) => Promise<void>;
                        dispatchAction: (actionParams: ActionParams) => void;
                        undoLastAction: () => void;
                }
```

6.2 useReferenceDataStore

Purpose: Caches "static" data like Teams and Seasons so we don’t refetch them on every page navigation.

> **Architectural Rationale:** *Teams rarely change. Fetching them once at app launch (or lazily) and keeping them in memory improves navigation speed significantly.*

Chapter 7

Infrastructure Implementation

7.1 Firestore Schema

We use a sub-collection strategy for Actions to avoid hitting the 1MB document limit for very long games.

- ‘teams/teamId‘
- ‘games/gameId‘
  - ‘actions/actionId‘ (Sub-collection)
- ‘players/playerId‘

7.2 Repository Implementation Details

7.2.1 FirebaseGameRepository.ts

When ‘findById(id)‘ is called:

1. Fetch ‘games/id‘ document.
2. **Parallel Fetch**: Fetch ‘games/id/actions‘ subcollection.
3. Combine the Game metadata and the list of Actions into the Domain ‘Game‘ entity.
4. Return the entity.

> **Architectural Rationale:** *Separating actions into a subcollection allows us to load the "Game List" (Schedule) very quickly without downloading thousands of shot coordinates for every game.*