**CHAPTER 1**

**INTRODUCTION**

* 1. **PROBLEM STATEMENT**

Traditional cricket scoring methods rely heavily on manual paper scorebooks and basic digital solutions, leading to numerous challenges in match management and statistical tracking. Scorers face difficulties in maintaining accurate ball-by-ball records, calculating complex statistics, and managing player performances in real-time. Additionally, the growing popularity of fantasy cricket creates a need for immediate statistical updates and point calculations. The current solutions often lack integration between match scoring and fantasy elements, resulting in delayed updates and poor user engagement.

* 1. **OBJECTIVES**

The Fantasy Cricket Scoring System provides comprehensive match management and fantasy cricket capabilities. The system allows match officials to record ball-by-ball scoring with automatic statistical calculations and fantasy point updates. Users can create fantasy teams before matches, selecting players within credit limits and designating captains/vice-captains. The scoring module handles all cricket scenarios including extras, dismissals, while maintaining detailed player statistics. Fantasy points are calculated in real-time based on player performances, with separate tracking for batting, bowling, and fielding achievements. The system ensures proper match state management across innings transitions and maintains data integrity through robust validation mechanisms.

* 1. **SCOPE**

The system encompasses complete match management functionality from team creation to match completion, including detailed statistical tracking and fantasy point calculations. It handles limited-overs cricket matches with support for:

* Real-time ball-by-ball scoring
* Comprehensive player statistics
* Fantasy team creation and management
* Live leaderboard updates
* Detailed match summaries
* Multi-user access control

**CHAPTER 2**

**EXISTING AND PROPOSED MODEL**

**2.1 EXISTING MODEL**

The functional requirement in Food Waste Management System is the collective information about what are the operations available in the system.

* Manual scorekeeping with paper scorebooks
* Basic digital applications without fantasy features
* Separate fantasy platforms without live integration
* Limited statistical tracking capabilities
* Poor data synchronization

**2.2 PROPOSED MODEL**

The non-functional requirement describes about platform and physical resource required for building the student management system.

* Flutter-based Web application with Supabase backend
* Real-time scoring and fantasy point updates
* Comprehensive match and player statistics
* Secure user authentication and role management
* Intuitive user interface with proper validation

**CHAPTER 3**

**SYSTEM DESIGN**

**3.1 SOFTWARE REQUIREMENTS SPECIFICATION**

**3.1.1. Introduction**

**3.1.1.1 Purpose**

The Fantasy Cricket Scoring System aims to revolutionize cricket match management by combining real-time scoring capabilities with fantasy sports features. This system provides a comprehensive platform for cricket enthusiasts to score matches accurately while participating in fantasy cricket competitions.

**3.1.1.2 Scope**

The system encompasses:

* Real-time cricket match scoring and management
* Fantasy team creation and management
* Player performance tracking and statistics
* Live fantasy points calculation
* Comprehensive match analytics
* Multi-user access with role-based permissions

**3.1.1.3 Acronyms, and Abbreviations**

* FCSS: Fantasy Cricket Scoring System
* UI/UX: User Interface/User Experience
* API: Application Programming Interface
* JWT: JSON Web Token
* RLS: Row Level Security
* DB: Database

**3.1.1.4 References**

1. Cricket Laws (MCC): https://www.lords.org/mcc/the-laws-of-cricket

2. Flutter Documentation: https://flutter.dev/docs

3. Supabase Documentation: https://supabase.com/docs

4. Fantasy Sports Guidelines: https://fifs.in/guidelines/

**3.1.1.5 Overview**

This document provides detailed specifications for:

* System architecture and components
* User interface requirements
* Database structure
* Security implementations
* Performance metrics
* Integration requirements

**3.1.2 OVERALL DESCRIPTION**

**3.1.2.1 Product Perspective**

The system operates as a mobile application with:

* Flutter-based frontend for cross-platform compatibility
* Supabase backend for real-time data management
* PostgreSQL database for data persistence
* WebSocket connections for live updates
* Email integration for notifications
* Secure authentication system

**3.1.2.2 Product Functions**

**Match Management**

Real-time cricket scoring takes place through an intuitive interface where officials can record ball-by-ball action. Each delivery can be marked as a regular ball, wide, no-ball, or dead ball. Runs are recorded through quick-tap buttons (0-6), with additional options for extras like byes and leg-byes.

The system automatically tracks overs, maintaining proper sequencing and enforcing bowling restrictions. When wickets fall, officials can select from multiple dismissal types and record relevant fielder and bowler contributions.

**Fantasy Integration**

Users create fantasy teams within a 100-credit budget constraint. Each player has a credit value based on their past performance and current form. Teams must include a valid combination of batsmen, bowlers, wicket-keepers, and all-rounders.

Points are calculated automatically as the match progresses. Batting points accumulate for runs scored, with bonuses for high strike rates and milestones. Bowling points reward wickets and economical spells, while fielding points cover catches, run-outs, and stumpings.

**Statistical Analysis**

The system maintains comprehensive statistics for every player, including batting averages, bowling figures, and fielding records. Match data is analyzed to generate insights like partnership breakdowns, wagon wheels, and Manhattan graphs.

Live projections help teams track required run rates and target probabilities. Historical data feeds into player rankings and helps determine fantasy player valuations for future matches.

**User Experience**

A clean, responsive interface ensures smooth operation during high-pressure match situations. Quick-action buttons and gesture controls speed up common scoring tasks. Real-time validation prevents common scoring errors while helpful tooltips guide new users.

**Reporting and Analytics**

Post-match reports compile key statistics and memorable moments. Fantasy team owners receive detailed breakdowns of their players' performances. League administrators can access comprehensive match and player databases for tournament management.

**3.1.2.3 USER CHARACTERISTICS**

**Match Officials**

Match officials interact with the core scoring functionality. They require:

**Technical Proficiency:**

* Basic smartphone operation skills
* Ability to navigate touch interfaces
* Understanding of menu-driven applications

**Domain Knowledge:**

* Comprehensive cricket rules understanding
* Familiarity with scoring conventions
* Knowledge of different match formats
* Quick decision-making ability

**Fantasy Players**

Fantasy players primarily engage with team creation and points tracking. They need:

**Game Understanding:**

* Basic cricket knowledge
* Strategy development skills
* Understanding of player statistics
* Familiarity with fantasy sports concepts

**Technical Requirements:**

* Email account for registration
* Stable internet connection
* Compatible mobile device
* Basic app navigation skills

**System Administrators**

Administrators manage the overall system operation. They require:

**Technical Skills:**

* Database management knowledge
* User account administration
* System monitoring capabilities
* Performance optimization experience

**3.1.2.4 CONSTRAINTS**

**Technical Constraints**

The system operates within these limitations:

**Hardware Requirements:**

Minimum Device Specifications:

* RAM: 4GB
* Storage: 64GB
* Processor: 1.6 GHz dual-core
* Network: 4G/WiFi
* Screen: 5" HD display

**Software Limitations:**

Platform Requirements:

* Android 6.0 or higher
* iOS 12.0 or higher
* Flutter SDK 3.0+
* Supabase infrastructure limits

**3.1.2.5 ASSUMPTIONS AND DEPENDENCIES**

**ASSUMPTIONS**

**User Assumptions**

* Users have basic knowledge to operate a mobile device
* Users possess valid email accounts for authentication
* Users have stable internet connectivity for login and verification
* Users can understand basic English interface elements
* Users will verify their email addresses when registering

**Technical Assumptions**

* Mobile devices meet these minimum requirements:
* Android 6.0 or iOS equivalent
* 4GB RAM
* Stable internet connection
* Basic touchscreen functionality
* Users maintain current versions of the application
* Device time settings are accurate for proper authentication

**DEPENDENCIES**

**Core Dependencies**

**External Services:**

* Supabase Authentication Service
* Supabase Real-time Database
* Email Service Provider

**Development Framework:**

* Flutter SDK 3.0+
* Dart 2.17+
* Material Design Components

**Required Packages:**

* supabase\_flutter: For authentication
* simple\_animations: For UI animations
* flutter\_material: For UI components

**System Dependencies**

* Active Supabase project instance
* Configured authentication providers
* Email service for verification
* Proper environment variables setup
* SSL certificates for secure connections

**User Interface Dependencies**

* Material Design theme configuration
* Custom animation controllers
* Form validation logic
* Error handling mechanisms
* State management system

**3.1.3 FUNCTIONAL AND NON-FUNCTIONAL REQUIREMENTS**

**3.1.3.1 Functional Requirements**

**Authentication System**

The system shall implement a secure email-based authentication using Supabase integration, including email verification flows, password validation, and session management with proper error handling and user feedback mechanisms.

**Login Form Validation**

The system shall validate all form inputs in real-time, including email format verification with regex patterns, required field validation, and password strength requirements before allowing form submission.

**Error Handling**

The system shall provide comprehensive error handling for authentication failures, network issues, and validation errors, displaying user-friendly error messages through a dedicated error message display area.

**Visual Feedback**

The system shall provide immediate visual feedback for all user interactions, including loading states during authentication, success animations for successful operations, and error indicators for failed actions.

**Session Management**

The system shall maintain secure user sessions using JWT tokens, handle session timeouts appropriately, and provide proper session cleanup on logout or application closure.

**Navigation Control**

The system shall manage secure navigation between authentication states, including redirection to home page after successful login and proper handling of back navigation and state persistence.

**Email Verification**

The system shall enforce email verification before allowing access to protected routes, with functionality to resend verification emails and proper status checking.

**Form State Management**

The system shall maintain form state throughout the authentication process, preserve user inputs during validation, and clear sensitive data upon successful authentication or manual reset.

**Animation System**

The system shall implement a sophisticated animation system including background animations with rotating circles, form transitions, and loading state animations to enhance user experience.

**Social Login Interface**

The system shall provide a structured interface for future social login integrations while clearly communicating current limitations to users.

**3.1.3.2 Non-Functional Requirements**

**Performance**

The system shall maintain smooth animations at 60 FPS, process form submissions within 300ms, and complete authentication requests within 3 seconds under normal network conditions.

**Security**

The system shall implement secure password handling, protect against XSS attacks, sanitize all user inputs, and maintain secure communication channels using SSL/TLS encryption.

**Reliability**

The system shall maintain 99.9% uptime for authentication services, handle network interruptions gracefully, and preserve user data integrity throughout all operations.

**Usability**

The system shall provide clear user feedback, maintain consistent UI elements, offer intuitive navigation, and ensure accessibility standards compliance.

**Responsiveness**

The system shall adapt to different screen sizes, maintain proper layouts from 320px to 2560px width, and handle orientation changes smoothly.

**Resource Efficiency**

The system shall optimize memory usage during animations, properly dispose of controllers and animations, and maintain efficient widget rebuilding strategies.

**Code Quality**

The system shall maintain clean architecture principles, follow Flutter best practices, and implement proper documentation for all complex logic.

**Error Recovery**

The system shall implement automatic retry mechanisms for failed network requests, preserve user inputs during errors, and provide clear recovery paths.

**Compatibility**

The system shall function consistently across Android 6.0+ and iOS 12.0+ devices, maintaining feature parity across platforms.

**Maintainability**

The system shall implement modular code structure, maintain clear separation of concerns, and follow consistent naming conventions for future maintainability.

**3.1.4 GLOSSORY:**

**AnimatedBackgroundPainter :** A custom painting class that draws and manages the animated circular patterns in the login screen background.

**TextEditingController :** A controller class that manages text input state for email and password fields in the login form.

**StatefulWidget :** A Flutter widget class that maintains changeable (mutable) state, used as the base for the login page.

**BuildContext :** A locator that keeps track of where a widget is positioned in the widget tree structure.

**CustomPaint :** A widget that provides a canvas on which custom shapes can be drawn, used for the animated background.

**TickerProviderStateMixin :** A mixin that provides the vsync ticker for animations, ensuring they remain efficient and synchronized with the screen refresh rate.

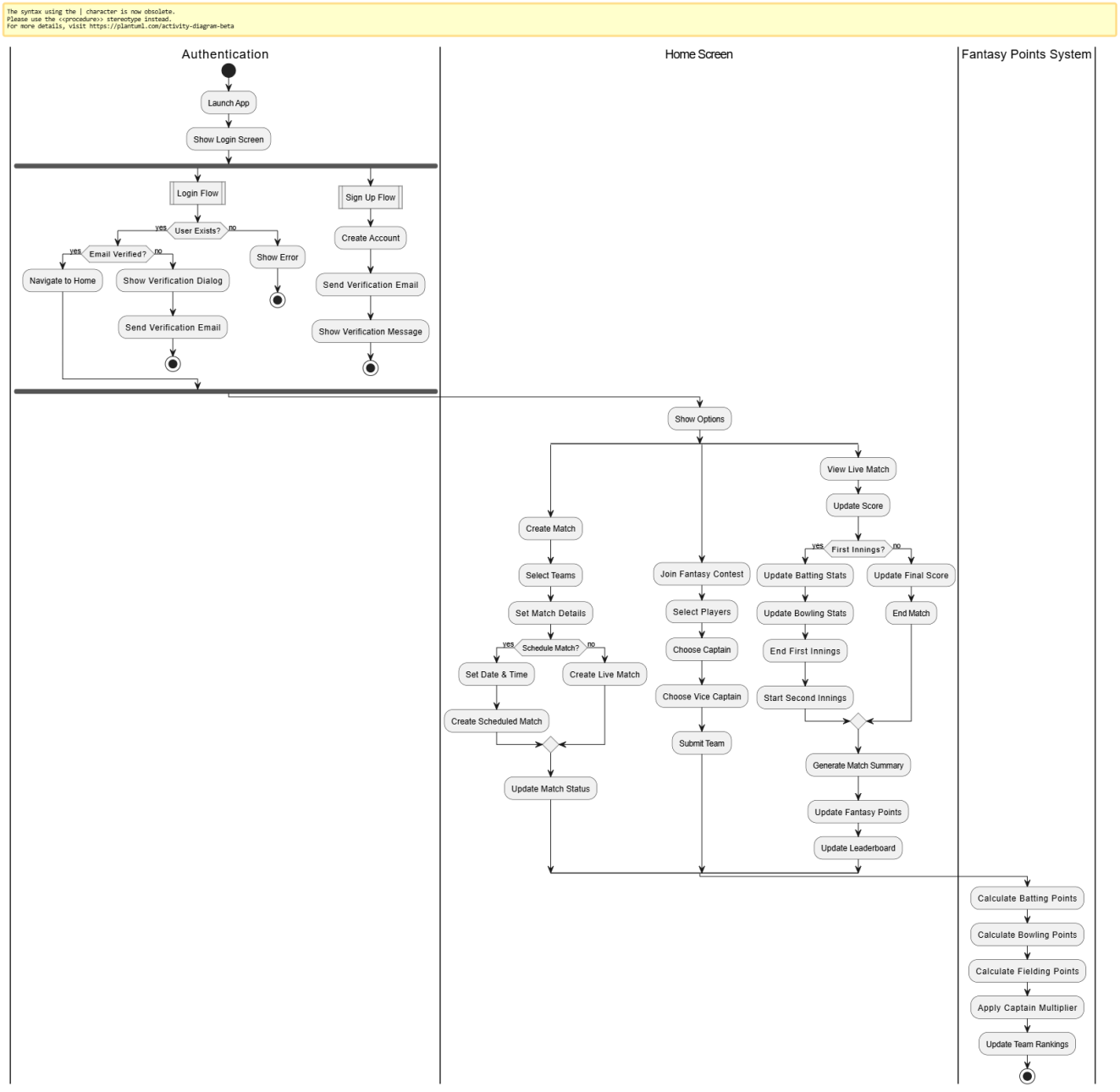
**LinearGradient :** A gradient that interpolates colors along a line, used in the login button's background.

**CurvedAnimation :** An animation that applies a non-linear curve to transform how the animation progresses over time.

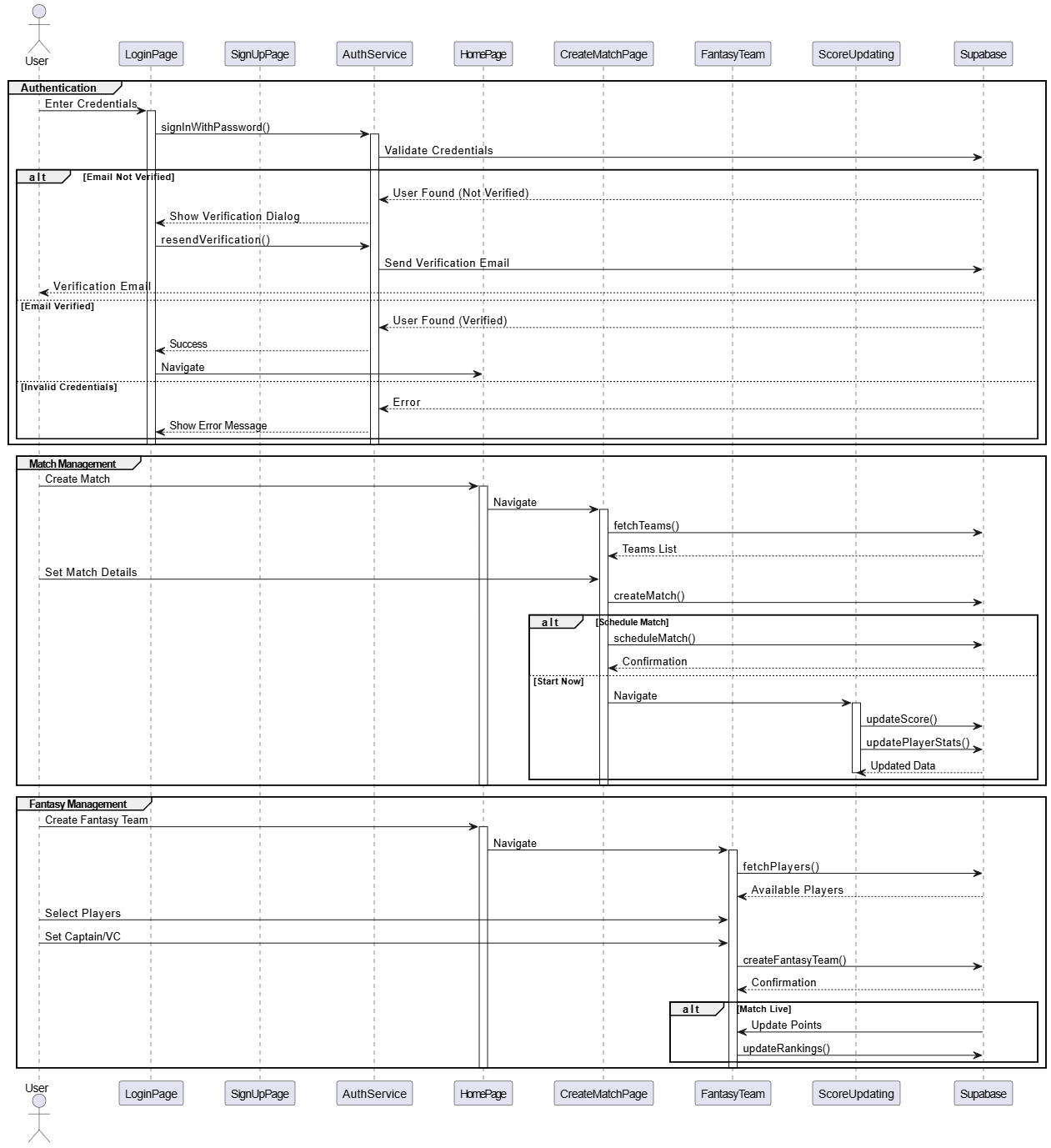
**AuthException :** A Supabase-specific error type that handles authentication-related errors during login attempts.

**MaterialPageRoute :** A modal route that replaces the entire screen with a platform-adaptive transition animation when navigating between pages.

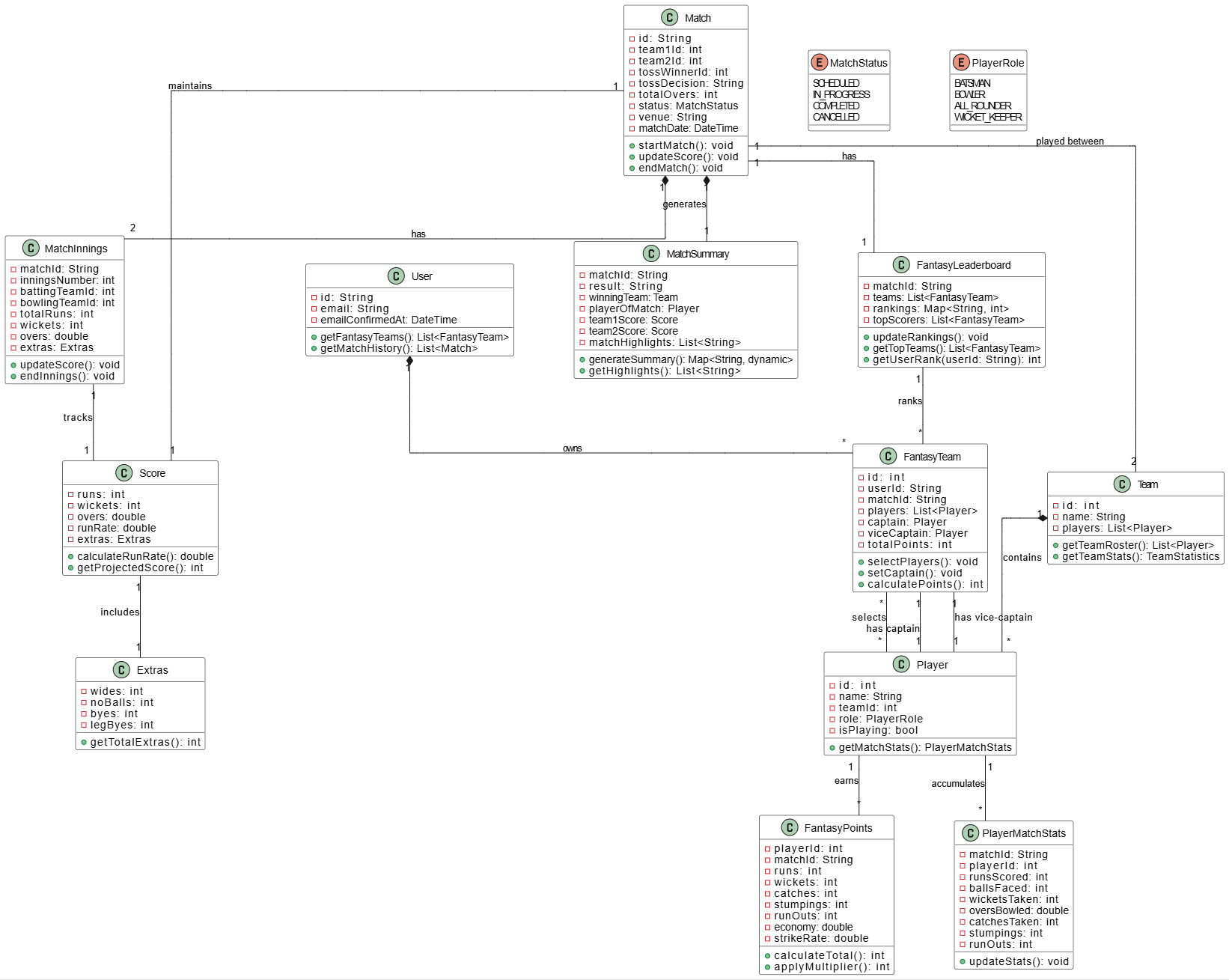
**3.2 UML DIAGRAMS / PIN CONFIGURATION & DESIGN DIAGRAMS**

****

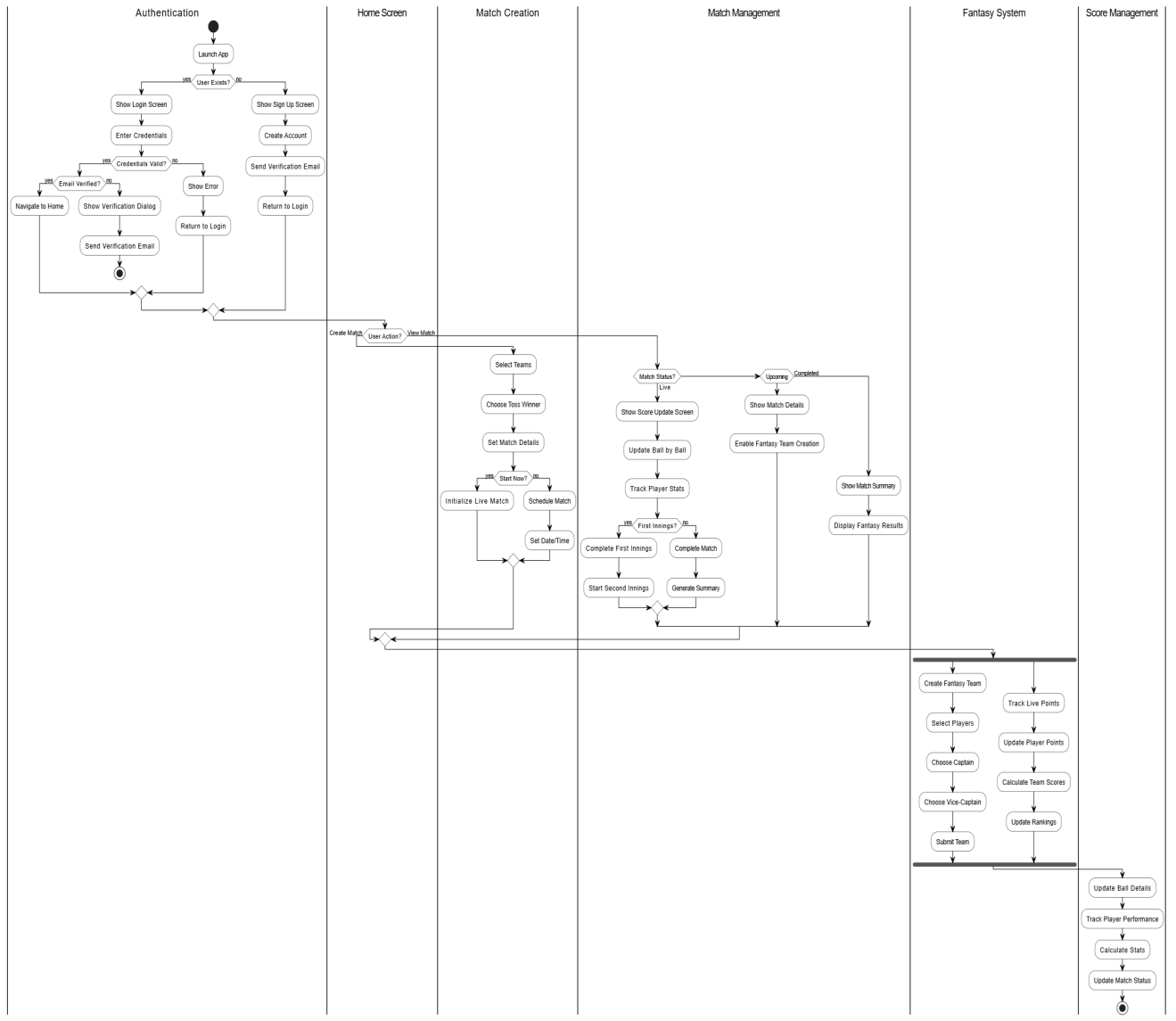
**Figure 3.1: Activity Diagram for Fantasy Cricket Application**



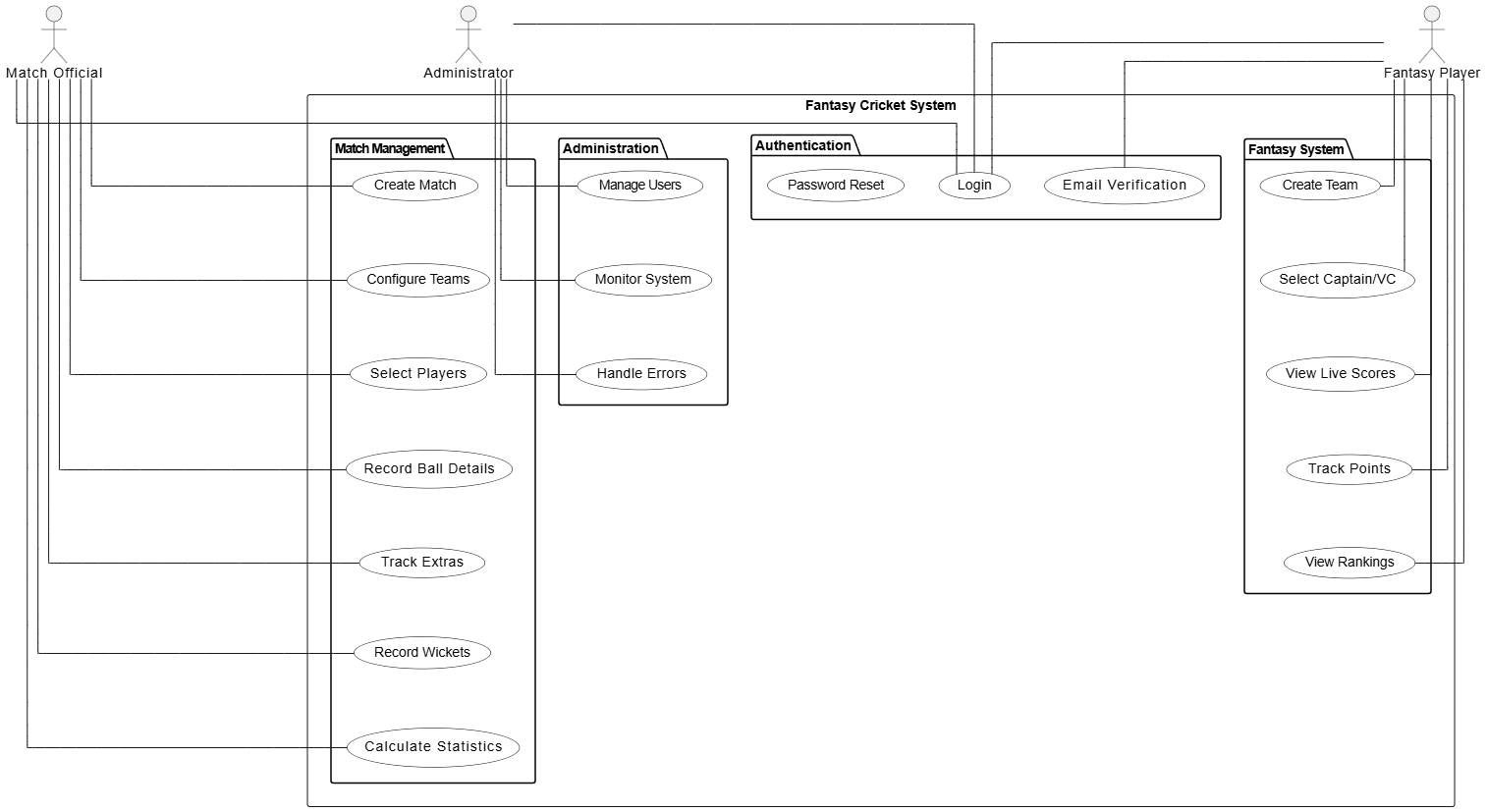
**Figure 3.2: Sequence Diagram for Fantasy Cricket Application**



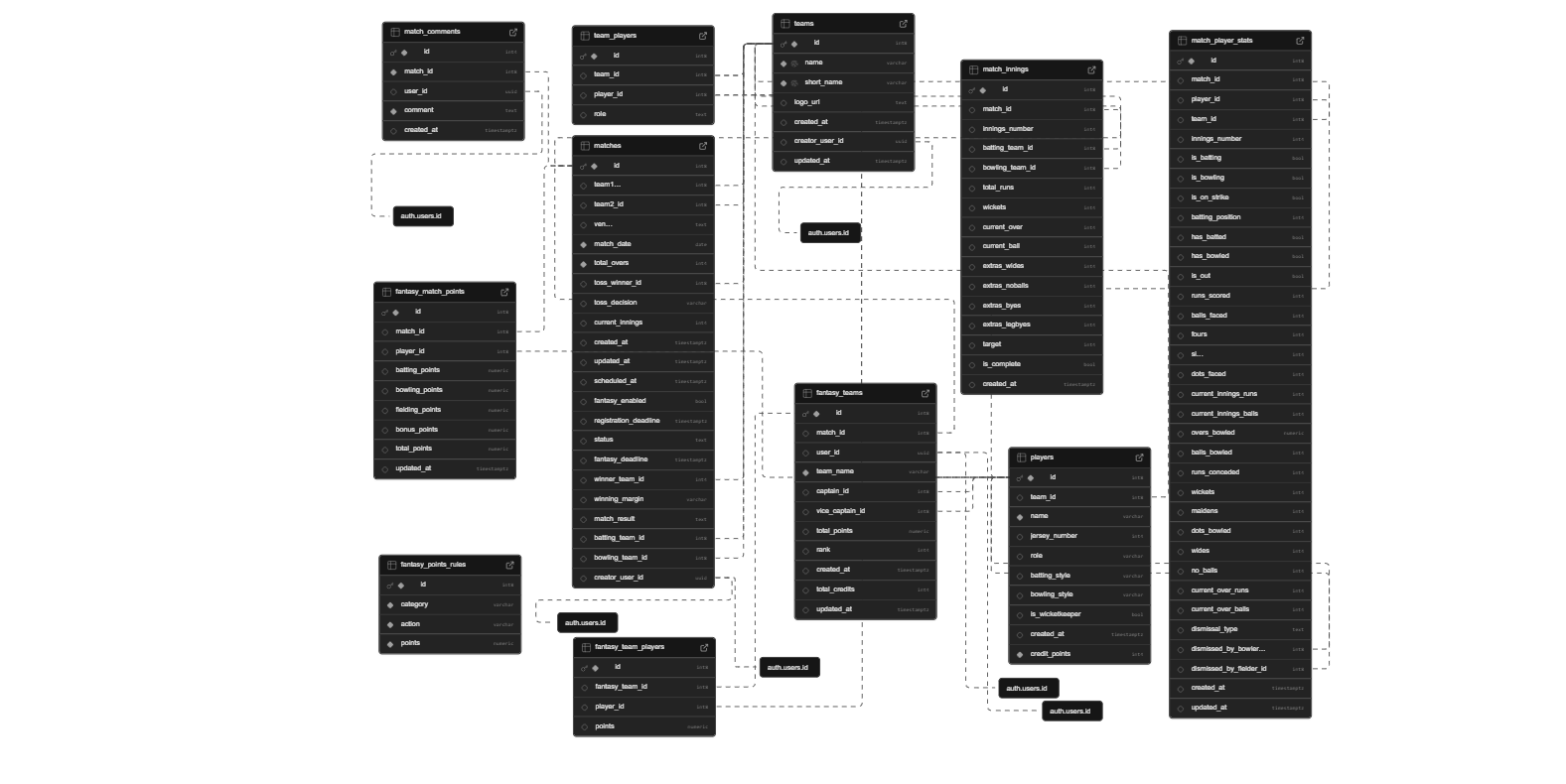
**Figure 3.3: Class Diagram for Fantasy Cricket Application**

****

**Figure 3.4: Flow Diagram for Fantasy Cricket Application**

****

**Figure 3.5: Use Case Diagram for Fantasy Cricket Application**

****

**Figure 3.6: Schema Diagram for Fantasy Cricket Application**

**3.3 DESIGN COMPONENTS**

**3.2.1 Front End:**

The Fantasy Cricket Scoring Application uses the following for developing interactive pages.

Framework: Flutter (v3.0+)

Language: Dart

Key Components:

Material Design widgets

Custom animations

Form validation

State management

**3.2.2 Back End:**

The backend for the application is being handled by,

Platform: Supabase

Services:

Authentication system

PostgreSQL database

Real-time data sync

Row level security

**3.4 DATABASES DESCRIPTION**

Listed below gives a description of database document schemas used for the Fantasy Cricket Scoring Application

**3.4.1 Fantasy Match Points Table**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Attribute Name** | **Width** | **Type** | **Constraints** | **Description** |
| id | 8 | bigserial | PRIMARY KEY | Points record identifier |
| match\_id | 8 | bigint | FOREIGN KEY, UNIQUE composite | Reference to matches table |
| player\_id | 8 | bigint | FOREIGN KEY, UNIQUE composite | Reference to players table |
| batting\_points | 10,2 | numeric | DEFAULT 0 | Points earned from batting |
| bowling\_points | 10,2 | numeric | DEFAULT 0 | Points earned from bowling |
| fielding\_points | 10,2 | numeric | DEFAULT 0 | Points earned from fielding |
| bonus\_points | 10,2 | numeric | DEFAULT 0 | Additional bonus points |
| total\_points | 10,2 | numeric | DEFAULT 0 | Sum of all points earned |
| updated\_at | - | timestamp | DEFAULT now() | Last update timestamp |

**3.4.2 Fantasy Points Rules Table**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Attribute Name** | **Width** | **Type** | **Constraints** | **Description** |
| id | 8 | bigserial | PRIMARY KEY | Rule identifier |
| category | 50 | varchar | NOT NULL, UNIQUE composite | Points category (e.g., batting, bowling) |
| action | 100 | varchar | NOT NULL, UNIQUE composite | Specific scoring action |
| points | 5,2 | numeric | NOT NULL | Points awarded for the action |

**3.4.3 Fantasy Team Players Table**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Attribute Name** | **Width** | **Type** | **Constraints** | **Description** |
| id | 8 | bigserial | PRIMARY KEY | Player entry identifier |
| fantasy\_team\_id | 8 | bigint | FOREIGN KEY, UNIQUE composite, CASCADE on delete | Reference to fantasy team |
| player\_id | 8 | bigint | FOREIGN KEY, UNIQUE composite | Reference to player |
| points | 10,2 | numeric | DEFAULT 0 | Points earned by player |

**3.4.4 Fantasy Teams Table**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Attribute Name** | **Width** | **Type** | **Constraints** | **Description** |
| id | 8 | bigserial | PRIMARY KEY | Fantasy team identifier |
| match\_id | 8 | bigint | FOREIGN KEY | Reference to match |
| user\_id | - | uuid | FOREIGN KEY, UNIQUE composite | Reference to auth.users |
| team\_name | 100 | varchar | NOT NULL, UNIQUE composite | Fantasy team name |
| captain\_id | 8 | bigint | FOREIGN KEY | Reference to captain |
| vice\_captain\_id | 8 | bigint | FOREIGN KEY | Reference to vice-captain |
| total\_points | 10,2 | numeric | DEFAULT 0 | Total team points |
| rank | - | integer | No constraint | Team ranking |
| created\_at | - | timestamp | DEFAULT now() | Creation timestamp |
| total\_credits | - | integer | DEFAULT 0 | Total credits used |
| updated\_at | - | timestamp | DEFAULT CURRENT\_TIMESTAMP | Last update time |

**3.4.5 Match Comments Table**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Attribute Name** | **Width** | **Type** | **Constraints** | **Description** |
| id | 4 | serial | PRIMARY KEY | Comment identifier |
| match\_id | 8 | bigint | FOREIGN KEY, NOT NULL, CASCADE on delete | Reference to match |
| user\_id | - | uuid | FOREIGN KEY, CASCADE on delete | Reference to auth.users |
| comment | - | text | NOT NULL | Comment content |
| created\_at | - | timestamp | DEFAULT CURRENT\_TIMESTAMP | Creation timestamp |

**3.4.7 Match Player Statistics Table**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Attribute Name** | **Width** | **Type** | **Constraints** | **Description** |
| id | 8 | bigserial | PRIMARY KEY | Statistics identifier |
| match\_id | 8 | bigint | FOREIGN KEY, UNIQUE composite | Match reference |
| player\_id | 8 | bigint | FOREIGN KEY, UNIQUE composite | Player reference |
| team\_id | 8 | bigint | FOREIGN KEY | Team reference |
| innings\_number | - | integer | CHECK [1,2], UNIQUE composite | Innings number |
| is\_batting | - | boolean | DEFAULT false | Current batting status |
| is\_bowling | - | boolean | DEFAULT false | Current bowling status |
| is\_on\_strike | - | boolean | DEFAULT false | On strike status |
| batting\_position | - | integer | No constraint | Batting order position |
| has\_batted | - | boolean | DEFAULT false | Has batted flag |
| has\_bowled | - | boolean | DEFAULT false | Has bowled flag |
| is\_out | - | boolean | DEFAULT false | Dismissal status |
| runs\_scored | - | integer | DEFAULT 0 | Total runs scored |
| balls\_faced | - | integer | DEFAULT 0 | Total balls faced |
| fours | - | integer | DEFAULT 0 | Number of boundaries |
| sixes | - | integer | DEFAULT 0 | Number of sixes |
| dots\_faced | - | integer | DEFAULT 0 | Dot balls faced |
| current\_innings\_runs | - | integer | DEFAULT 0 | Current innings runs |
| current\_innings\_balls | - | integer | DEFAULT 0 | Current innings balls |
| overs\_bowled | 4,1 | numeric | DEFAULT 0 | Overs bowled |
| balls\_bowled | - | integer | DEFAULT 0 | Total balls bowled |
| runs\_conceded | - | integer | DEFAULT 0 | Runs given while bowling |
| wickets | - | integer | DEFAULT 0 | Wickets taken |
| maidens | - | integer | DEFAULT 0 | Maiden overs bowled |
| dots\_bowled | - | integer | DEFAULT 0 | Dot balls bowled |
| wides | - | integer | DEFAULT 0 | Wide balls bowled |
| no\_balls | - | integer | DEFAULT 0 | No balls bowled |
| current\_over\_runs | - | integer | DEFAULT 0 | Current over runs |
| current\_over\_balls | - | integer | DEFAULT 0 | Balls in current over |
| dismissal\_type | - | text | No constraint | How player got out |
| dismissed\_by\_bowler\_id | 8 | bigint | FOREIGN KEY | Bowler who took wicket |
| dismissed\_by\_fielder\_id | 8 | bigint | FOREIGN KEY | Fielder who took catch/stumping |
| created\_at | - | timestamp | DEFAULT CURRENT\_TIMESTAMP | Record creation time |
| updated\_at | - | timestamp | DEFAULT CURRENT\_TIMESTAMP | Last update time |

**3.4.8 Matches Table**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Attribute Name** | **Width** | **Type** | **Constraints** | **Description** |
| id | 8 | bigserial | PRIMARY KEY | Match identifier |
| team1\_id | 8 | bigint | FOREIGN KEY | First team reference |
| team2\_id | 8 | bigint | FOREIGN KEY | Second team reference |
| venue | - | text | No constraint | Match venue |
| match\_date | - | date | NOT NULL | Match date |
| total\_overs | - | integer | NOT NULL, CHECK > 0 | Number of overs |
| toss\_winner\_id | 8 | bigint | FOREIGN KEY | Toss winning team |
| toss\_decision | 4 | varchar | CHECK ['bat','bowl'] | Toss winner's choice |
| current\_innings | - | integer | DEFAULT 1, CHECK [1,2] | Current innings number |
| created\_at | - | timestamp | DEFAULT CURRENT\_TIMESTAMP | Creation time |
| updated\_at | - | timestamp | DEFAULT CURRENT\_TIMESTAMP | Last update time |
| scheduled\_at | - | timestamp | No constraint | Match schedule time |
| fantasy\_enabled | - | boolean | DEFAULT true | Fantasy status |
| registration\_deadline | - | timestamp | No constraint | Team registration cutoff |
| status | - | text | DEFAULT 'draft', CHECK ['draft','scheduled','in\_progress','completed','cancelled'] | Match status |
| fantasy\_deadline | - | timestamp | No constraint | Fantasy team creation deadline |
| winner\_team\_id | - | integer | FOREIGN KEY | Winning team reference |
| winning\_margin | 100 | varchar | No constraint | Victory margin details |
| match\_result | - | text | No constraint | Match result description |
| batting\_team\_id | 8 | bigint | FOREIGN KEY | Current batting team |
| bowling\_team\_id | 8 | bigint | FOREIGN KEY | Current bowling team |
| creator\_user\_id | - | uuid | FOREIGN KEY | Match creator reference |

**3.4.9 Players Table**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Attribute Name** | **Width** | **Type** | **Constraints** | **Description** |
| id | 8 | bigserial | PRIMARY KEY | Player identifier |
| team\_id | 8 | bigint | FOREIGN KEY, CASCADE on delete | Team reference |
| name | 100 | varchar | NOT NULL | Player's full name |
| jersey\_number | - | integer | UNIQUE composite with team\_id | Player's jersey number |
| role | 50 | varchar | CHECK ['Batsman','Bowler','All-rounder','Wicket Keeper'] | Player's primary role |
| batting\_style | 50 | varchar | No constraint | Player's batting style |
| bowling\_style | 50 | varchar | No constraint | Player's bowling style |
| is\_wicketkeeper | - | boolean | DEFAULT false | Wicketkeeper indicator |
| created\_at | - | timestamp | DEFAULT CURRENT\_TIMESTAMP | Record creation time |
| credit\_points | - | integer | NOT NULL, DEFAULT 8, CHECK (≥4 AND ≤12) | Fantasy credit value |

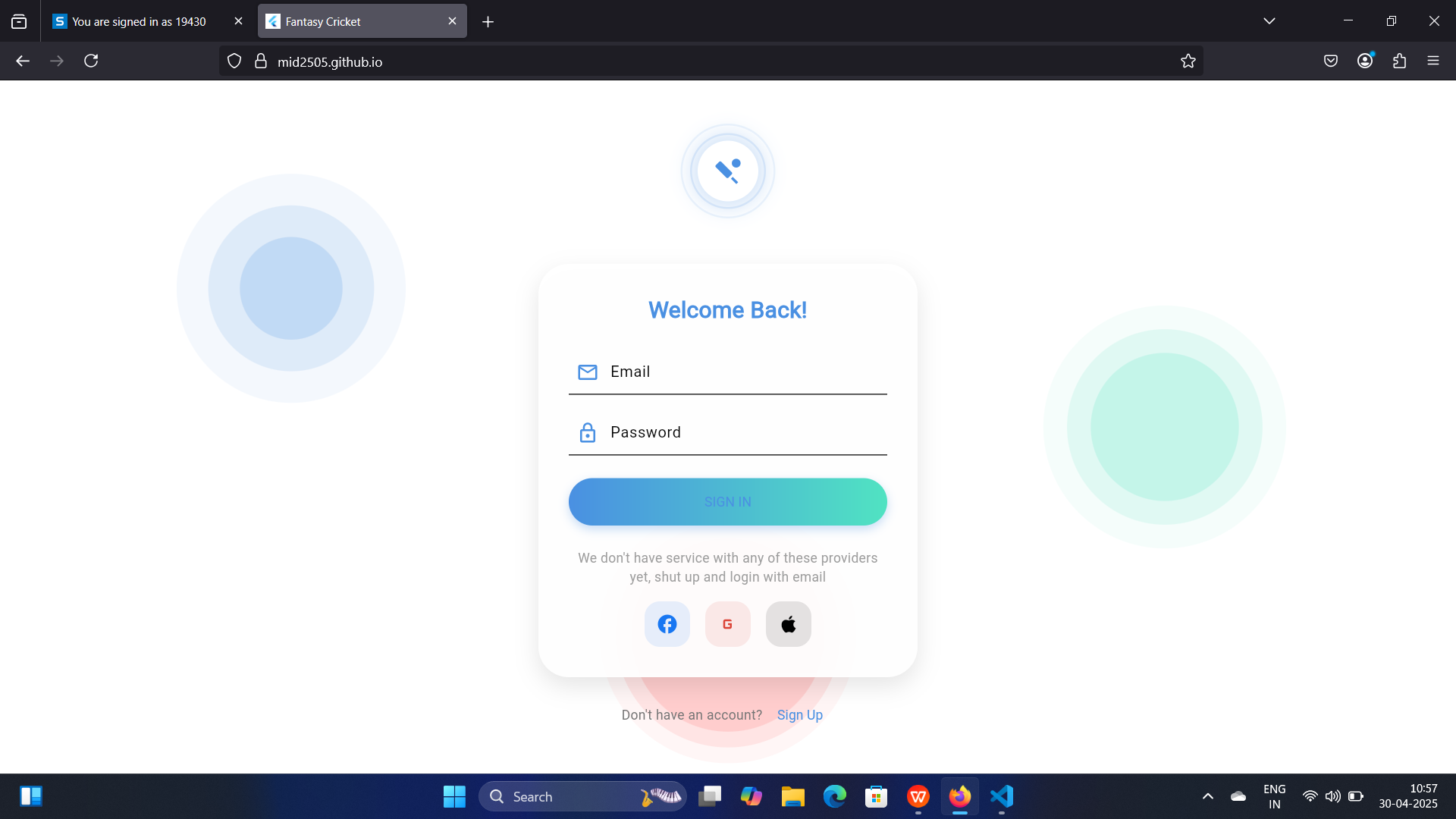
**3.4.10 Team Players Table**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Attribute Name** | **Width** | **Type** | **Constraints** | **Description** |
| id | 8 | bigint | PRIMARY KEY | Team player entry identifier |
| team\_id | 8 | bigint | FOREIGN KEY (2 refs) | Reference to team |
| player\_id | 8 | bigint | FOREIGN KEY (2 refs) | Reference to player |
| role | - | text | No constraint | Player's role in team |

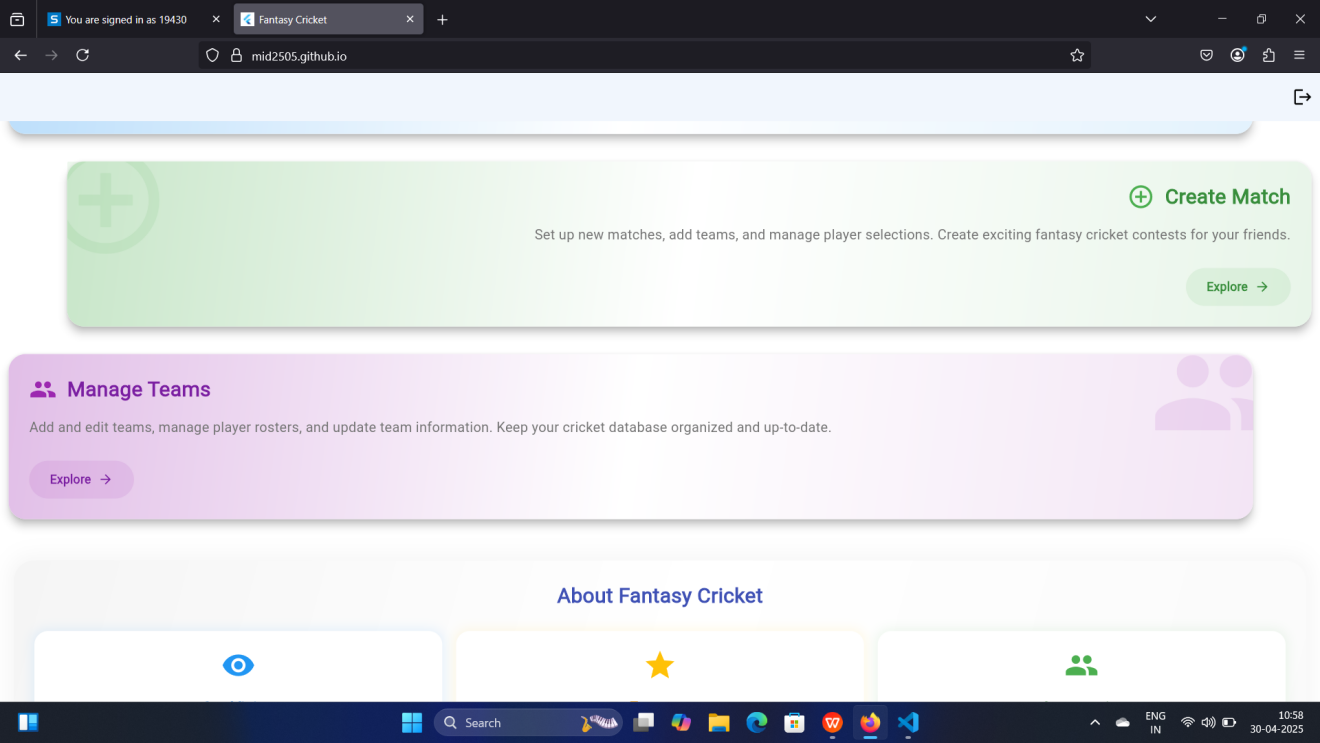
**3.4.11 Teams Table**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Attribute Name** | **Width** | **Type** | **Constraints** | **Description** |
| id | 8 | bigserial | PRIMARY KEY | Team identifier |
| name | 100 | varchar | NOT NULL, UNIQUE | Team full name |
| short\_name | 10 | varchar | NOT NULL, UNIQUE | Team abbreviation |
| logo\_url | - | text | No constraint | Team logo URL |
| created\_at | - | timestamp | DEFAULT CURRENT\_TIMESTAMP | Creation timestamp |
| creator\_user\_id | - | uuid | FOREIGN KEY | Creator reference |
| updated\_at | - | timestamp | DEFAULT CURRENT\_TIMESTAMP | Last update time |

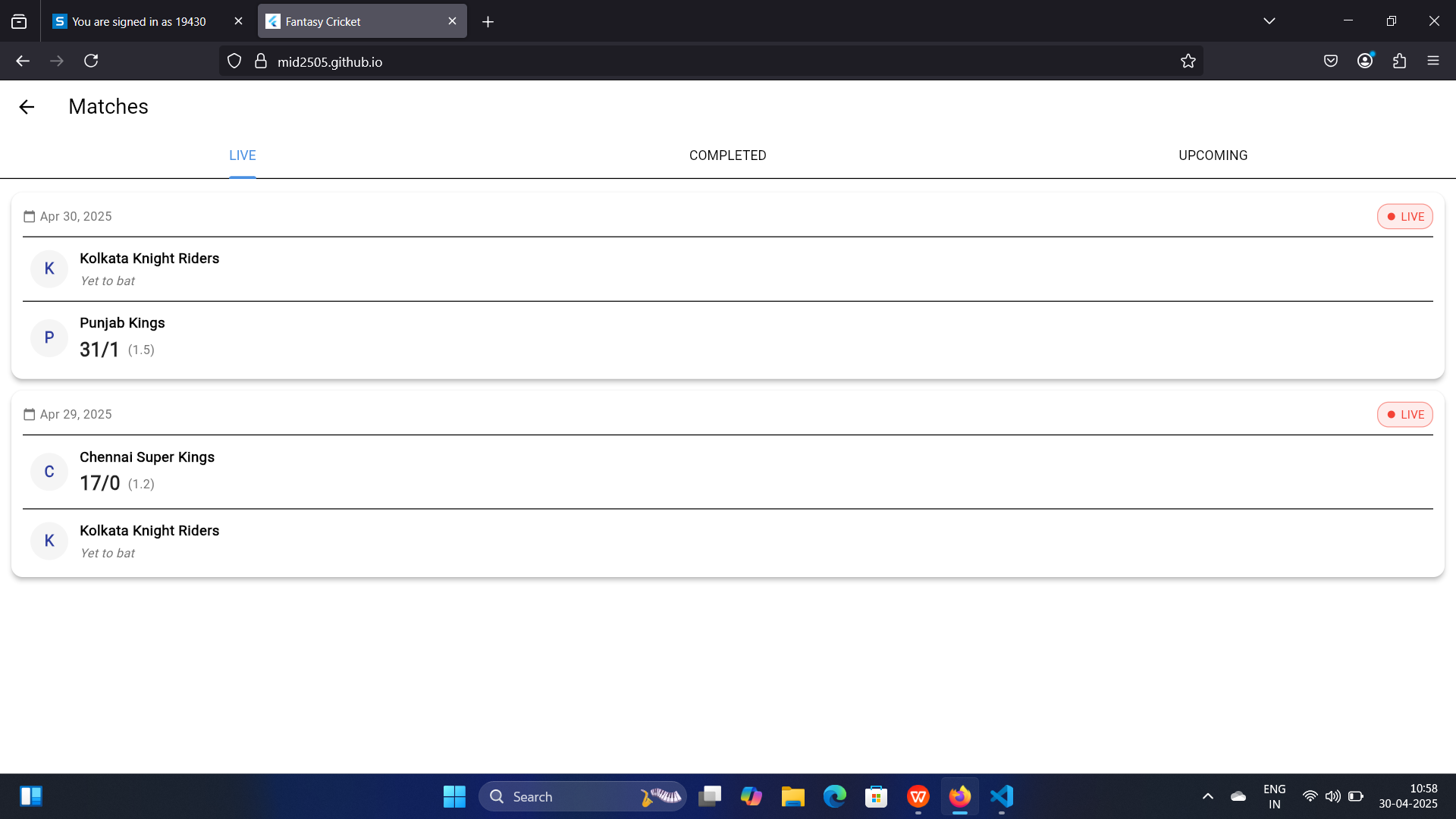
**3.5 USER INTERFACE DESIGN**



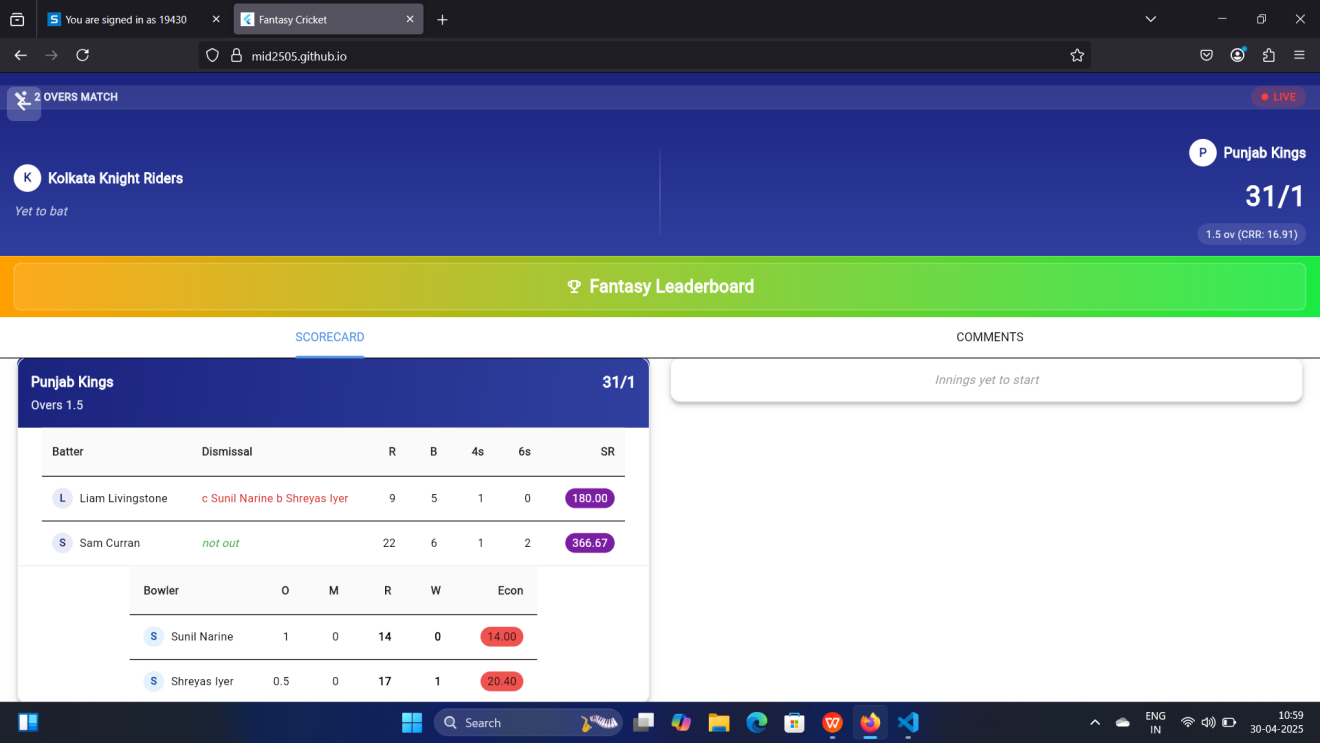
**Figure 3.1: Interface for Login Page**



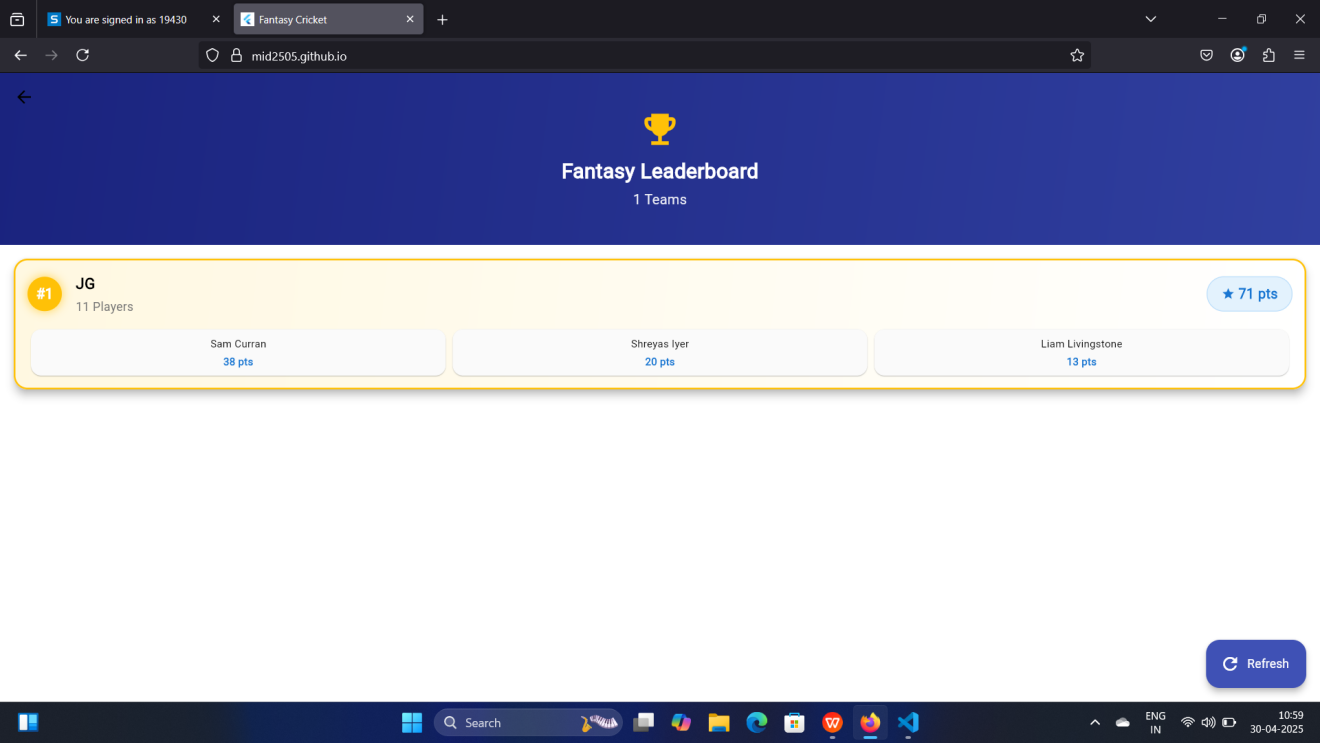
**Figure 3.2: Interface for Home Page**



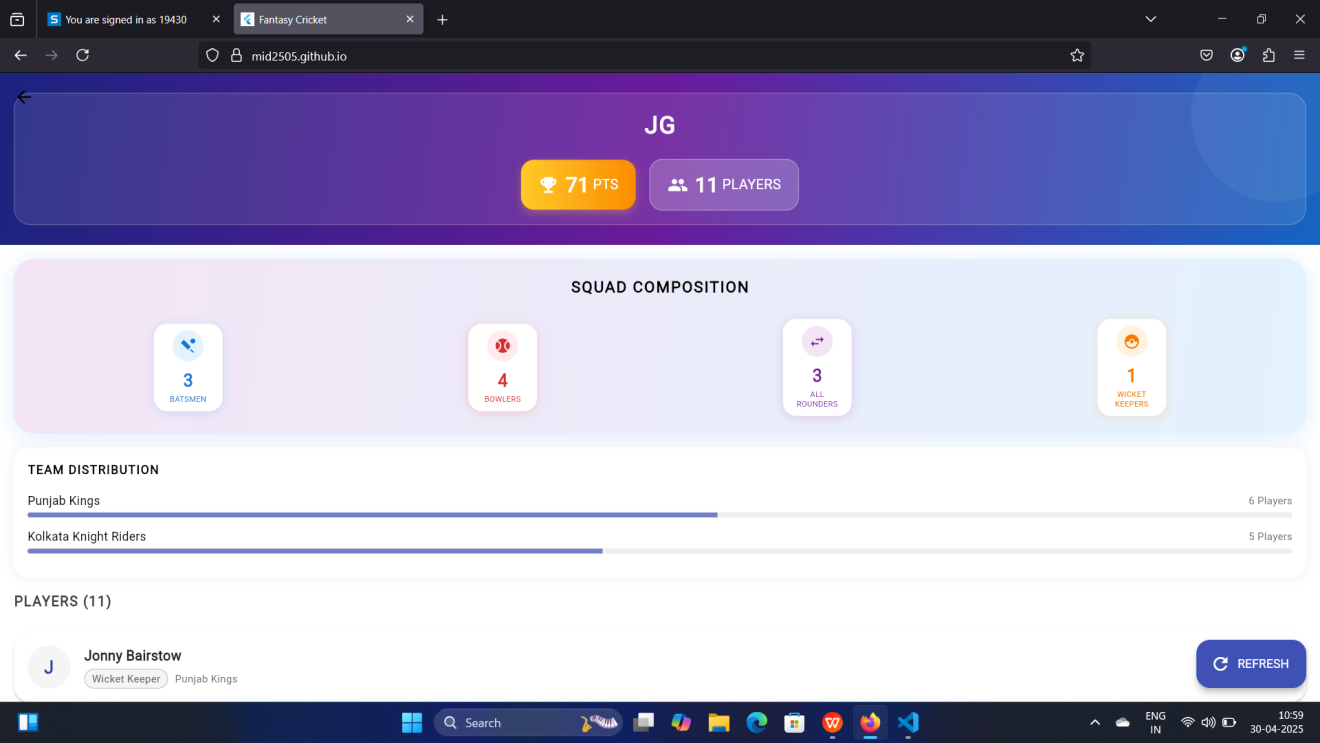
**Figure 3.3 Interface for Viewing matches**



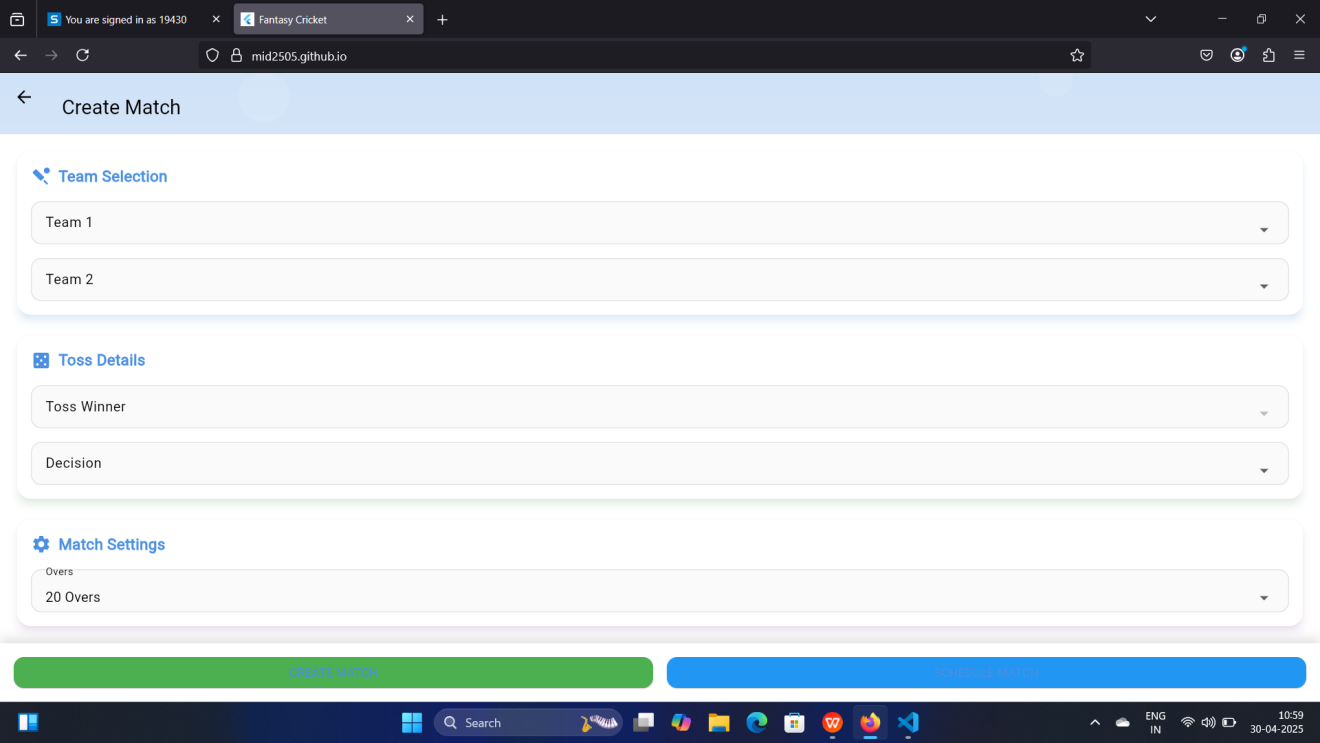
**Figure 3.4: Interface for Live score updation**



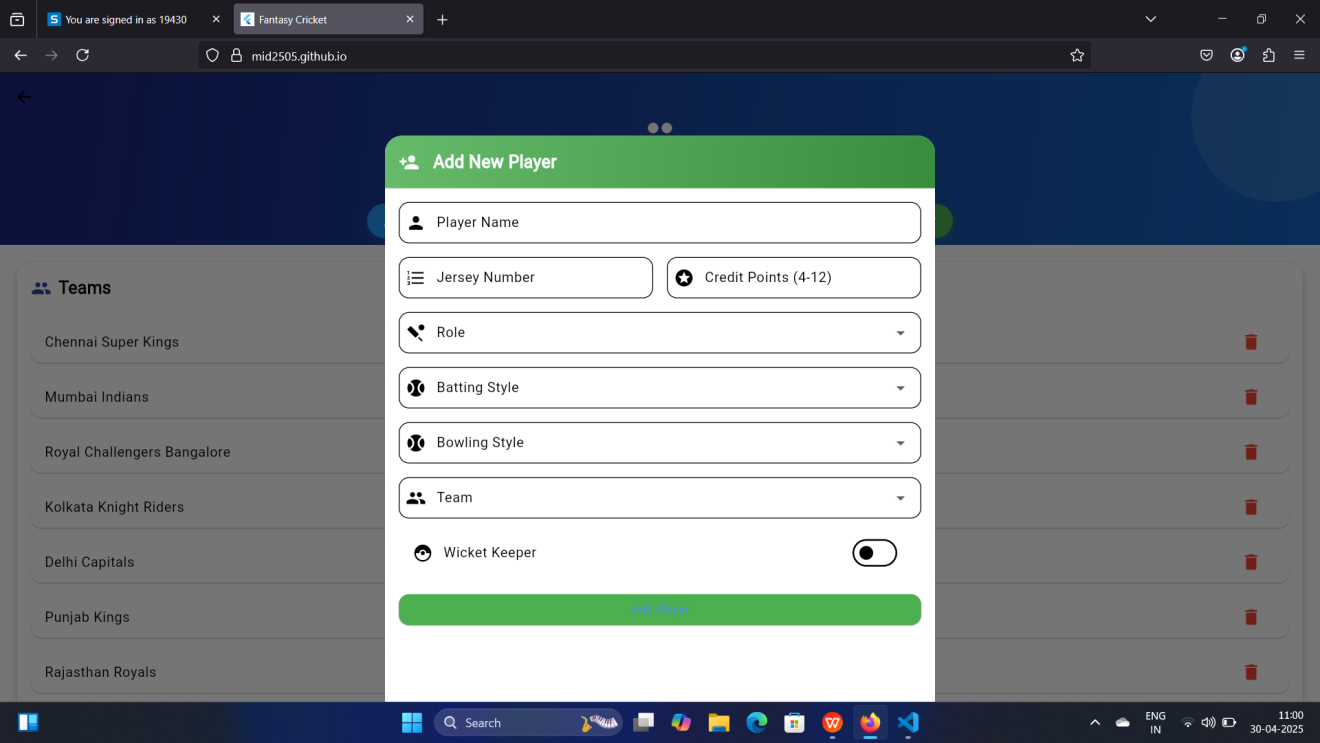
**Figure 3.5: Interface for Fantasy leaderboard**



**Figure 3.6: Interface for viewing fantasy team stats**



**Figure 3.7: Interface for Creating/Scheduling match**



**Figure 3.8: Interface for Team/Players management**

**CHAPTER 4**

**SYSTEM IMPLEMENTATION**

**4.1 LOGIN IMPLEMENTATION**

The login system uses Supabase authentication with email verification.

GET email, password from form

IF validation passes

CHECK Supabase authentication

IF user verified

NAVIGATE to HomePage

ELSE

SHOW verification dialog

**4.2 MATCH CREATION IMPLEMENTATION**

Match officials can create new cricket matches with team details.

GET match details:

Team1, Team2 selection

Match date and venue

Overs limit

Fantasy deadline

IF all fields valid

CREATE match in database

ENABLE fantasy team creation

SET match status as 'scheduled'

**4.3 MATCH SCORING IMPLEMENTATION**

The ball-by-ball scoring system handles various cricket scenarios.

GET current innings state

FOR each ball

GET delivery type

IF normal delivery

GET runs scored (0-6)

UPDATE batting statistics:

Batsman runs

Strike rate

Boundaries count

UPDATE bowling figures:

Balls bowled

Runs conceded

Economy rate

IF extras

GET extra type:

Wide: +1 run

No ball: +1 run

Byes/Leg byes: input runs

ADD to extras total

UPDATE team score

IF wicket falls

GET dismissal type:

Bowled

Caught

LBW

Run out

Stumped

UPDATE batting card

UPDATE bowling figures

GET new batsman

UPDATE match state:

Current score

Wickets fallen

Overs completed

Required rate

**4.4 BATTING STATISTICS IMPLEMENTATION**

Real-time tracking of batting performance metrics.

FOR each batsman

INITIALIZE stats:

Runs = 0

Balls = 0

Fours = 0

Sixes = 0

Dot balls = 0

ON each ball faced

IF runs scored

ADD to total runs

IF four runs

INCREMENT fours

IF six runs

INCREMENT sixes

ELSE

INCREMENT dot balls

INCREMENT balls faced

CALCULATE:

Strike rate = (runs/balls)\*100

Control % = ((balls-misses)/balls)\*100

Boundary % = ((fours+sixes)/balls)\*100

ON dismissal

RECORD:

Final score

Time batted

Dismissal type

Bowler name

**4.5 FANTASY POINTS IMPLEMENTATION**

Real-time fantasy points calculation.

FOR each player action

IF batting

CALCULATE points:

Runs \* 1

Boundaries bonus

Strike rate bonus

IF bowling

CALCULATE points:

Wickets \* 25

Maiden over \* 8

Economy bonus

UPDATE player total

UPDATE team rankings

**4.6 STATISTICS IMPLEMENTATION**

Player statistics tracking system.

FOR each innings

TRACK batting stats:

Runs, balls, SR

4s, 6s count

TRACK bowling stats:

Overs, maidens

Wickets, economy

UPDATE match summary

GENERATE player rankings

**4.7 MATCH STATE IMPLEMENTATION**

Match progress tracking system.

INITIALIZE match state

WHILE match in progress

UPDATE current innings

TRACK over progress

CHECK innings completion

VALIDATE match rules

UPDATE required rate

IF match complete

DECLARE winner

FINALIZE statistics

**4.8 FANTASY TEAM IMPLEMENTATION**

Fantasy team creation and validation.

GET user selection

VALIDATE team composition:

11 players total

1-4 wicketkeepers

3-6 batsmen

3-6 bowlers

1-4 all-rounders

CHECK credit limit (100)

ASSIGN captain (2x)

ASSIGN vice-captain (1.5x)

**4.9 LEADERBOARD IMPLEMENTATION**

Real-time fantasy rankings system.

FOR each fantasy team

CALCULATE total points

SORT by points DESC

ASSIGN ranks

UPDATE leaderboard

DISPLAY top performers

SHOW point breakdowns

**4.10 MATCH SUMMARY IMPLEMENTATION**

Match statistics compilation.

GET match details

COMPILE statistics:

Team scores

Player performances

Fantasy points

GENERATE match report

UPDATE match status

**CHAPTER 5**

**RESULTS AND DISCUSSION**

**5.1 TEST CASES AND RESULTS**

**5.1.1 Score Recording Test Cases**

The Tables 5.1 and 5.2 show test cases for ball-by-ball scoring functionality and validation.

**Table 5.1: Valid Ball Recording Test Case**

|  |  |
| --- | --- |
| **Test Case ID** | TC1 |
| **Test Case Description** | Test recording a valid delivery with runs |
| **Test Data** | Delivery type: Normal, Runs: 4, Batsman: Smith, Bowler: Johnson |
| **Expected Output** | Runs added, statistics updated |
| **Result** | PASS |

**Table 5.2: Extras Recording Test Case**

|  |  |
| --- | --- |
| **Test Case ID** | TC2 |
| **Test Case Description** | Test recording wide ball with additional runs |
| **Test Data** | Delivery type: Wide, Extra runs: 2 |
| **Expected Output** | 3 runs added (1 wide + 2 runs), no ball counted |
| **Result** | PASS |

**5.1.2 Fantasy Team Creation Test Cases**

**Table 5.3: Valid Team Creation**

|  |  |
| --- | --- |
| **Test Case ID** | TC3 |
| **Test Case Description** | Test creating team within credit limit |
| **Test Data** | 11 players selected, Total credits: 98.5, Valid roles distribution |
| **Expected Output** | Team created successfully |
| **Result** | PASS |

**Table 5.4: Invalid Team Composition**

|  |  |
| --- | --- |
| **Test Case ID** | TC4 |
| **Test Case Description** | Test team with invalid role distribution |
| **Test Data** | 7 batsmen, 4 bowlers, no wicketkeeper |
| **Expected Output** | Error: "Must select 1-4 wicketkeepers" |
| **Result** | PASS |

**5.1.3 Match Statistics Test Cases**

**Table 5.5: Partnership Calculation**

|  |  |
| --- | --- |
| **Test Case ID** | TC5 |
| **Test Case Description** | Test current partnership calculation |
| **Test Data** | Batsman1: 45(30), Batsman2: 28(20), Partnership: 73 runs |
| **Expected Output** | Partnership details updated correctly |
| **Result** | PASS |

**Table 5.6: Required Rate Calculation**

|  |  |
| --- | --- |
| **Test Case ID** | TC6 |
| **Test Case Description** | Test required run rate calculation |
| **Test Data** | Target: 180, Current: 120/2 in 15 overs |
| **Expected Output** | Required Rate: 12.00 |
| **Result** | PASS |

**5.1.4 Real-time Match Progress Test Cases**

**Table 5.7: Live Score Update**

|  |  |
| --- | --- |
| **Test Case ID** | TC7 |
| **Test Case Description** | Test real-time score update after each ball |
| **Test Data** | Current Score: 45/1, Ball: 4 runs, Extras: 0 |
| **Expected Output** | Score updates to 49/1, Batsman stats increment |
| **Result** | PASS |

**Table 5.8: Over Completion**

|  |  |
| --- | --- |
| **Test Case ID** | TC8 |
| **Test Case Description** | Test over completion and bowler change |
| **Test Data** | Over: 5.6, Bowler: Johnson |
| **Expected Output** | Over becomes 6.0, Prompt for new bowler |
| **Result** | PASS |

**5.1.5 Fantasy Points Calculation Test Cases**

**Table 5.9: Batting Points**

|  |  |
| --- | --- |
| **Test Case ID** | TC9 |
| **Test Case Description** | Test fantasy points for century |
| **Test Data** | Batsman score: 102(65), 8 fours, 5 sixes |
| **Expected Output** | Points: 102 + 16 + 25 + 8 = 151 points |
| **Result** | PASS |

T**able 5.10: Bowling Points**

|  |  |
| --- | --- |
| **Test Case ID** | TC10 |
| **Test Case Description** | Test fantasy points for 5-wicket haul |
| **Test Data** | Bowling: 4-0-25-5 |
| **Expected Output** | Points: 125 (wickets) + 4 (economy) = 129 points |
| **Result** | PASS |

**5.1.6 Match State Test Cases**

**Table 5.11: Innings Break**

|  |  |
| --- | --- |
| **Test Case ID** | TC11 |
| **Test Case Description** | Test innings transition handling |
| **Test Data** | First innings complete, Score: 185/8 |
| **Expected Output** | Second innings initiated, Target set: 186 |
| **Result** | PASS |

**Table 5.12: Match Completion**

|  |  |
| --- | --- |
| **Test Case ID** | TC12 |
| **Test Case Description** | Test match end conditions |
| **Test Data** | Target: 186, Score: 187/4 in 18.3 overs |
| **Expected Output** | Match ended, Winner declared, Stats finalized |
| **Result** | PASS |

**5.1.7 Statistics Update Test Cases**

**Table 5.13: Partnership Record**

|  |  |
| --- | --- |
| **Test Case ID** | TC13 |
| **Test Case Description** | Test partnership calculation accuracy |
| **Test Data** | Batsman1: 28(20), Batsman2: 35(25), Total: 65(45) |
| **Expected Output** | Partnership details match individual scores |
| **Result** | PASS |

**Table 5.14: Economy Rate**

|  |  |
| --- | --- |
| **Test Case ID** | TC14 |
| **Test Case Description** | Test bowler economy rate calculation |
| **Test Data** | Overs: 3.2, Runs: 24, Wickets: 2 |
| **Expected Output** | Economy Rate: 7.20 |
| **Result** | PASS |

**CHAPTER 6**

**CONCLUSION AND FUTURE ENHANCEMENT(S)**

The Fantasy Cricket Scoring System is a comprehensive application designed to revolutionize cricket match management and fantasy sports experience. The system successfully implements real-time match scoring capabilities alongside fantasy team management features, providing an engaging platform for both match officials and cricket enthusiasts. The application's user-friendly interface, enhanced by smooth animations and intuitive controls, makes it accessible for scorers to record ball-by-ball action while fantasy players can simultaneously track their teams' performance. Future enhancements could include advanced statistical analytics, automated highlight generation, real-time player valuation adjustments, and integration with live cricket feeds, further enriching the cricket scoring and fantasy gaming experience.

**APPENDIX – A**

**SYSTEM REQUIREMENTS**

**HARDWARE REQUIREMENTS:**

|  |  |
| --- | --- |
| Processor : | Any |
| RAM : | 500MB |
| HDD : | 2GB |

**SOFTWARE REQUIREMENTS:**

|  |  |
| --- | --- |
| Operating System : | Any |
| DBMS : | Supabase |
| IDE used : | Visual Studio Code |
| Flutter SDK Version : | 3.29.2 and above |

**APPENDIX – B**

**SOURCE CODE**

import 'package:flutter/material.dart';

import 'package:login/match\_summary\_page.dart';

import 'package:supabase\_flutter/supabase\_flutter.dart';

// Add these imports at the top of the file

import 'dialogs/wide\_dialog.dart';

import 'dialogs/no\_ball\_dialog.dart';

import 'dialogs/wicket\_dialog.dart';

// Add at the top of the file after imports

enum DismissalType { bowled, caught, lbw, runOut, stumped, hitWicket }

class ScoreUpdatingPage extends StatefulWidget {

final int matchId;

final int team1Id;

final int team2Id;

final int battingTeamId;

final int bowlingTeamId;

final int maxOvers;

final bool isFirstInnings;

// Add these new parameters

final int tossWinnerId;

final String tossChoice;

const ScoreUpdatingPage({

Key? key,

required this.matchId,

required this.team1Id,

required this.team2Id,

required this.battingTeamId,

required this.bowlingTeamId,

required this.maxOvers,

required this.isFirstInnings,

required this.tossWinnerId, // Add this

required this.tossChoice, // Add this

}) : super(key: key);

@override

State<ScoreUpdatingPage> createState() => \_ScoreUpdatingPageState();

}

class \_ScoreUpdatingPageState extends State<ScoreUpdatingPage> {

// Add in \_ScoreUpdatingPageState class, with other state variables

List<Map<String, dynamic>> \_playerStats = [];

void \_showError(String message) {

ScaffoldMessenger.of(context).showSnackBar(

SnackBar(content: Text(message)),

);

}

// Basic match state

bool \_isLoading = true;

Map<String, dynamic>? \_innings;

int \_totalRuns = 0;

int \_wickets = 0;

int \_currentOver = 0;

int \_currentBall = 0;

int? \_target;

// Add at the top of the \_ScoreUpdatingPageState class

// Players

String? \_striker;

String? \_nonStriker;

String? \_currentBowler;

List<Map<String, dynamic>> \_battingTeam = [];

List<Map<String, dynamic>> \_bowlingTeam = [];

// Current over tracking

List<String> \_currentOverBalls = [];

int \_currentOverRuns = 0;

Map<String, List<int>> \_currentOverExtras = {

'wide\_runs': [],

'noball\_runs': [],

};

@override

void initState() {

super.initState();

// Initialize without setting state directly

\_initialize();

}

// New method to handle initialization

Future<void> \_initialize() async {

if (!mounted) return;

try {

// First update match status

await Supabase.instance.client

.from('matches')

.update({

'status': 'in\_progress',

'updated\_at': DateTime.now().toIso8601String(),

})

.eq('id', widget.matchId)

.execute();

// Then initialize match

await \_initializeMatch();

} catch (error) {

if (mounted) {

\_showError('Failed to initialize: $error');

}

}

}

@override

void dispose() {

// Save state before disposing

\_saveCurrentState().then((\_) {

// Only proceed if still mounted

if (mounted) {

super.dispose();

}

});

}

Future<void> \_saveCurrentState() async {

try {

await Supabase.instance.client

.from('matches')

.update({

'status': 'in\_progress',

'updated\_at': DateTime.now().toIso8601String(),

})

.eq('id', widget.matchId)

.execute();

} catch (error) {

debugPrint('Error saving match state: $error');

}

}

Future<void> \_loadPlayers() async {

try {

// Load batting team players

final battingTeamResponse = await Supabase.instance.client

.from('players')

.select()

.eq('team\_id', widget.battingTeamId)

.execute();

// Load bowling team players

final bowlingTeamResponse = await Supabase.instance.client

.from('players')

.select()

.eq('team\_id', widget.bowlingTeamId)

.execute();

setState(() {

\_battingTeam =

List<Map<String, dynamic>>.from(battingTeamResponse.data ?? []);

\_bowlingTeam =

List<Map<String, dynamic>>.from(bowlingTeamResponse.data ?? []);

});

} catch (error) {

\_showError('Failed to load players: $error');

}

}

Future<void> \_initializeMatch() async {

try {

// 1. Load teams and players

await \_loadPlayers();

// 2. Initialize or load innings

await \_initializeInnings();

// 3. Initialize player stats

await \_initializePlayerStats();

// 4. Show player selection dialog

if (\_striker == null) {

await \_showPlayerSelectionDialog();

}

setState(() => \_isLoading = false);

} catch (error) {

\_showError('Failed to initialize match: $error');

}

}

Future<void> \_initializeInnings() async {

try {

final response = await Supabase.instance.client

.from('match\_innings')

.select()

.eq('match\_id', widget.matchId)

.eq('innings\_number', widget.isFirstInnings ? 1 : 2)

.single()

.execute();

setState(() {

\_innings = response.data;

\_totalRuns = response.data['total\_runs'] ?? 0;

\_wickets = response.data['wickets'] ?? 0;

\_currentOver = response.data['current\_over'] ?? 0;

\_currentBall = response.data['current\_ball'] ?? 0;

\_target = widget.isFirstInnings ? null : response.data['target'];

});

if (!widget.isFirstInnings) {

// Show target dialog for second innings

if (mounted) {

await showDialog(

context: context,

barrierDismissible: false,

builder: (context) => AlertDialog(

title: const Text('Second Innings'),

content: Column(

mainAxisSize: MainAxisSize.min,

children: [

Text('${\_getTeamName(widget.battingTeamId)} needs'),

Text(

'${\_target} runs to win',

style: const TextStyle(

fontSize: 24,

fontWeight: FontWeight.bold,

),

),

Text('from ${widget.maxOvers} overs'),

],

),

actions: [

ElevatedButton(

onPressed: () {

Navigator.pop(context);

\_showPlayerSelectionDialog();

},

child: const Text('Start Batting'),

),

],

),

);

}

} else {

await \_showPlayerSelectionDialog();

}

} catch (error) {

\_showError('Failed to initialize innings: $error');

print('Error details: $error');

}

}

Future<void> \_initializePlayerStats() async {

try {

final response = await Supabase.instance.client

.from('match\_player\_stats')

.select()

.eq('match\_id', widget.matchId)

.eq('innings\_number', widget.isFirstInnings ? 1 : 2)

.execute();

setState(() {

\_playerStats = List<Map<String, dynamic>>.from(response.data ?? []);

// Set current batsmen if they exist

final batsmen =

\_playerStats.where((p) => p['is\_batting'] == true).toList();

if (batsmen.length >= 2) {

\_striker = batsmen[0]['player\_id'].toString();

\_nonStriker = batsmen[1]['player\_id'].toString();

}

// Set current bowler if exists

final bowler = \_playerStats.firstWhere(

(p) => p['is\_bowling'] == true,

orElse: () => {},

);

if (bowler.isNotEmpty) {

\_currentBowler = bowler['player\_id'].toString();

}

});

} catch (error) {

\_showError('Failed to initialize player stats: $error');

}

}

Future<void> \_showPlayerSelectionDialog() async {

if (\_striker != null || \_nonStriker != null || \_currentBowler != null) {

// Players already selected, skip selection

return;

}

try {

// Show striker selection

final striker = await showDialog<String>(

context: context,

barrierDismissible: false,

builder: (context) => AlertDialog(

title: const Text('Select Striker'),

content: SingleChildScrollView(

child: Column(

mainAxisSize: MainAxisSize.min,

children: \_battingTeam

.where((p) => !\_playerStats.any((s) =>

s['player\_id'].toString() == p['id'].toString() &&

s['innings\_number'] == (widget.isFirstInnings ? 1 : 2)))

.map((player) => ListTile(

title: Text(player['name']),

onTap: () =>

Navigator.pop(context, player['id'].toString()),

))

.toList(),

),

),

),

);

if (striker == null) return;

// Show non-striker selection

final nonStriker = await showDialog<String>(

context: context,

barrierDismissible: false,

builder: (context) => AlertDialog(

title: const Text('Select Non-Striker'),

content: SingleChildScrollView(

child: Column(

mainAxisSize: MainAxisSize.min,

children: \_battingTeam

.where((p) =>

p['id'].toString() != striker &&

!\_playerStats.any((s) =>

s['player\_id'].toString() == p['id'].toString() &&

s['innings\_number'] ==

(widget.isFirstInnings ? 1 : 2)))

.map((player) => ListTile(

title: Text(player['name']),

onTap: () =>

Navigator.pop(context, player['id'].toString()),

))

.toList(),

),

),

),

);

if (nonStriker == null) return;

// Check if stats already exist for these players

final strikerStats = await \_checkExistingStats(striker);

final nonStrikerStats = await \_checkExistingStats(nonStriker);

// Initialize or update striker stats

if (!strikerStats) {

await Supabase.instance.client.from('match\_player\_stats').insert({

'match\_id': widget.matchId,

'innings\_number': widget.isFirstInnings ? 1 : 2,

'player\_id': striker,

'team\_id': widget.battingTeamId,

'is\_batting': true,

'is\_on\_strike': true,

'has\_batted': true,

'runs\_scored': 0,

'balls\_faced': 0,

'fours': 0,

'sixes': 0,

}).execute();

}

// Initialize or update non-striker stats

if (!nonStrikerStats) {

await Supabase.instance.client.from('match\_player\_stats').insert({

'match\_id': widget.matchId,

'innings\_number': widget.isFirstInnings ? 1 : 2,

'player\_id': nonStriker,

'team\_id': widget.battingTeamId,

'is\_batting': true,

'is\_on\_strike': false,

'has\_batted': true,

'runs\_scored': 0,

'balls\_faced': 0,

'fours': 0,

'sixes': 0,

}).execute();

}

setState(() {

\_striker = striker;

\_nonStriker = nonStriker;

});

// Now show bowler selection

await \_showBowlerSelectionDialog();

} catch (error) {

\_showError('Failed to select players: $error');

print('Error details: $error');

}

}

// Add this helper method to check for existing stats

Future<bool> \_checkExistingStats(String playerId) async {

final response = await Supabase.instance.client

.from('match\_player\_stats')

.select()

.eq('match\_id', widget.matchId)

.eq('innings\_number', widget.isFirstInnings ? 1 : 2)

.eq('player\_id', playerId)

.maybeSingle()

.execute();

return response.data != null;

}

Future<void> \_showBowlerSelectionDialog() async {

try {

final bowler = await showDialog<String>(

context: context,

barrierDismissible: false,

builder: (context) => AlertDialog(

title: const Text('Select Bowler'),

content: Column(

mainAxisSize: MainAxisSize.min,

children: \_bowlingTeam

.map((player) => ListTile(

title: Text(player['name']),

onTap: () =>

Navigator.pop(context, player['id'].toString()),

))

.toList(),

),

),

);

if (bowler == null) return;

// Initialize stats for selected bowler

await \_initializeBowler(bowler);

setState(() {

\_currentBowler = bowler;

});

} catch (error) {

\_showError('Failed to select bowler: $error');

}

}

Future<void> \_initializeBowler(String playerId) async {

await Supabase.instance.client.from('match\_player\_stats').insert({

'match\_id': widget.matchId,

'player\_id': playerId,

'team\_id': widget.bowlingTeamId,

'innings\_number': widget.isFirstInnings ? 1 : 2,

'is\_bowling': true,

'has\_bowled': true,

}).execute();

}

String \_getBatsmanName(String playerId) {

final player = \_battingTeam.firstWhere(

(p) => p['id'].toString() == playerId,

orElse: () => {'name': 'Unknown'},

);

return player['name'] ?? 'Unknown';

}

String \_getBowlerName(String playerId) {

final player = \_bowlingTeam.firstWhere(

(p) => p['id'].toString() == playerId,

orElse: () => {'name': 'Unknown'},

);

return player['name'] ?? 'Unknown';

}

String \_getBatsmanScore(String playerId) {

final stats = \_playerStats.firstWhere(

(s) => s['player\_id'].toString() == playerId,

orElse: () => {},

);

return '${stats['runs\_scored'] ?? 0}(${stats['balls\_faced'] ?? 0})';

}

String \_getBowlerFigures(String playerId) {

final stats = \_playerStats.firstWhere(

(s) => s['player\_id'].toString() == playerId,

orElse: () => {},

);

final overs = (stats['balls\_bowled'] ?? 0) ~/ 6;

final balls = (stats['balls\_bowled'] ?? 0) % 6;

return '${stats['wickets'] ?? 0}-${stats['runs\_conceded'] ?? 0} ($overs.$balls)';

}

String \_getTeamName(int teamId) {

if (teamId == widget.team1Id) {

return 'Team 1'; // Replace with actual team names from database

} else if (teamId == widget.team2Id) {

return 'Team 2';

}

return 'Unknown Team';

}

Color \_getRunButtonColor(int runs) {

switch (runs) {

case 0:

return Colors.grey;

case 4:

return Colors.green;

case 6:

return Colors.blue;

default:

return Colors.blue.shade700;

}

}

@override

Widget build(BuildContext context) {

if (\_isLoading) {

return const Scaffold(

body: Center(child: CircularProgressIndicator()),

);

}

return Scaffold(

appBar: AppBar(

title: Text('Live Score'),

actions: [

IconButton(

icon: Icon(Icons.refresh),

onPressed: \_refreshStats,

),

],

),

body: Column(

children: [

\_buildScoreCard(),

\_buildCurrentOver(),

Expanded(

child: DefaultTabController(

length: 2,

child: Column(

children: [

TabBar(

tabs: [

Tab(text: 'Batting'),

Tab(text: 'Bowling'),

],

),

Expanded(

child: TabBarView(

children: [

\_buildBattingStats(),

\_buildBowlingStats(),

],

),

),

],

),

),

),

\_buildScoringPanel(),

],

),

);

}

Widget \_buildScoreCard() {

return Card(

child: Padding(

padding: const EdgeInsets.all(16.0),

child: Column(

children: [

Text(

'${\_getTeamName(widget.battingTeamId)} vs ${\_getTeamName(widget.bowlingTeamId)}',

style: const TextStyle(fontSize: 18, fontWeight: FontWeight.bold),

),

const SizedBox(height: 8),

Text(

'$\_totalRuns/$\_wickets',

style: const TextStyle(fontSize: 24, fontWeight: FontWeight.bold),

),

Text('Overs: $\_currentOver.${\_currentBall}'),

if (!widget.isFirstInnings && \_target != null) ...[

const SizedBox(height: 8),

Text(

'Target: $\_target',

style: const TextStyle(fontWeight: FontWeight.bold),

),

Text(

'Need ${\_target! - \_totalRuns} from ${(widget.maxOvers \* 6) - (\_currentOver \* 6 + \_currentBall)} balls',

),

],

],

),

),

);

}

Widget \_buildCurrentOver() {

return Container(

padding: const EdgeInsets.all(8),

color: Colors.grey.shade200,

child: Column(

crossAxisAlignment: CrossAxisAlignment.start,

children: [

const Text('This Over:'),

\_buildThisOverDetails(),

],

),

);

}

Widget \_buildThisOverDetails() {

return Container(

padding: const EdgeInsets.all(8),

decoration: BoxDecoration(

color: Colors.grey.shade100,

borderRadius: BorderRadius.circular(8),

),

child: Row(

mainAxisSize: MainAxisSize.min,

children: \_currentOverBalls.asMap().entries.map((entry) {

Color ballColor;

String displayText = entry.value;

final index = entry.key;

// Handle different ball types

switch (entry.value) {

case 'Nb':

// Extract runs from no ball if any

final nbRuns =

(\_currentOverExtras['noball\_runs']?.length ?? 0) > index

? \_currentOverExtras['noball\_runs']![index]

: 1;

displayText = nbRuns > 1 ? 'Nb+${nbRuns - 1}' : 'Nb';

ballColor = Colors.purple;

break;

case 'Wd':

// Extract runs from wide if any

final wideRuns =

(\_currentOverExtras['wide\_runs']?.length ?? 0) > index

? \_currentOverExtras['wide\_runs']![index]

: 1;

displayText = wideRuns > 1 ? 'Wd+${wideRuns - 1}' : 'Wd';

ballColor = Colors.orange;

break;

case 'W':

ballColor = Colors.red;

break;

case '4':

ballColor = Colors.green;

break;

case '6':

ballColor = Colors.blue;

break;

default:

ballColor = Colors.black87;

}

return Container(

margin: const EdgeInsets.symmetric(horizontal: 4),

padding: const EdgeInsets.all(8),

decoration: BoxDecoration(

color: Colors.white,

border: Border.all(color: ballColor),

borderRadius: BorderRadius.circular(4),

),

child: Text(

displayText,

style: TextStyle(color: ballColor, fontWeight: FontWeight.bold),

),

);

}).toList(),

),

);

}

Widget \_buildBattingStats() {

return SingleChildScrollView(

child: DataTable(

columns: const [

DataColumn(label: Text('Batter')),

DataColumn(label: Text('R')),

DataColumn(label: Text('B')),

DataColumn(label: Text('4s')),

DataColumn(label: Text('6s')),

DataColumn(label: Text('SR')),

DataColumn(label: Text('Dismissal')), // Add this column

],

rows: \_playerStats

.where((p) =>

p['team\_id'] == widget.battingTeamId &&

(p['is\_batting'] == true || p['has\_batted'] == true))

.map((stats) {

final player = \_battingTeam.firstWhere(

(p) => p['id'].toString() == stats['player\_id'].toString(),

orElse: () => {'name': 'Unknown'},

);

final strikeRate = stats['balls\_faced'] > 0

? ((stats['runs\_scored'] ?? 0) \*

100.0 /

(stats['balls\_faced'] ?? 1))

.toStringAsFixed(1)

: '0.0';

// Add dismissal info

String dismissalInfo = '';

if (stats['is\_out'] == true) {

dismissalInfo = stats['dismissal\_type']?.toUpperCase() ?? '';

if (stats['dismissed\_by\_bowler\_id'] != null) {

final bowler = \_bowlingTeam.firstWhere(

(p) =>

p['id'].toString() ==

stats['dismissed\_by\_bowler\_id'].toString(),

orElse: () => {'name': 'Unknown'},

);

dismissalInfo += ' b ${bowler['name']}';

}

if (stats['dismissed\_by\_fielder\_id'] != null) {

final fielder = \_bowlingTeam.firstWhere(

(p) =>

p['id'].toString() ==

stats['dismissed\_by\_fielder\_id'].toString(),

orElse: () => {'name': 'Unknown'},

);

dismissalInfo += ' c ${fielder['name']}';

}

}

return DataRow(

selected: stats['player\_id'].toString() == \_striker ||

stats['player\_id'].toString() == \_nonStriker,

cells: [

DataCell(Text(player['name'] ?? 'Unknown')),

DataCell(Text('${stats['runs\_scored'] ?? 0}')),

DataCell(Text('${stats['balls\_faced'] ?? 0}')),

DataCell(Text('${stats['fours'] ?? 0}')),

DataCell(Text('${stats['sixes'] ?? 0}')),

DataCell(Text(strikeRate)),

DataCell(Text(dismissalInfo)), // Add this cell

],

);

}).toList(),

),

);

}

Widget \_buildBowlingStats() {

return SingleChildScrollView(

child: DataTable(

columns: const [

DataColumn(label: Text('Bowler')),

DataColumn(label: Text('O')),

DataColumn(label: Text('M')),

DataColumn(label: Text('R')),

DataColumn(label: Text('W')),

DataColumn(label: Text('Eco')),

],

rows: \_playerStats

.where((p) =>

p['team\_id'] == widget.bowlingTeamId &&

(p['is\_bowling'] == true || p['has\_bowled'] == true))

.map((stats) {

final player = \_bowlingTeam.firstWhere(

(p) => p['id'].toString() == stats['player\_id'].toString(),

orElse: () => {'name': 'Unknown'},

);

final overs = (stats['balls\_bowled'] ?? 0) ~/ 6;

final balls = (stats['balls\_bowled'] ?? 0) % 6;

final economy = overs > 0

? ((stats['runs\_conceded'] ?? 0) / overs).toStringAsFixed(1)

: '0.0';

return DataRow(

selected: stats['player\_id'].toString() == \_currentBowler,

cells: [

DataCell(Text(player['name'] ?? 'Unknown')),

DataCell(Text('$overs.$balls')),

DataCell(Text('${stats['maidens'] ?? 0}')),

DataCell(Text('${stats['runs\_conceded'] ?? 0}')),

DataCell(Text('${stats['wickets'] ?? 0}')),

DataCell(Text(economy)),

],

);

}).toList(),

),

);

}

Widget \_buildScoringPanel() {

return Container(

padding: const EdgeInsets.all(16),

decoration: BoxDecoration(

gradient: LinearGradient(

colors: [Colors.black87, Colors.blue.shade900],

begin: Alignment.topLeft,

end: Alignment.bottomRight,

),

borderRadius: BorderRadius.circular(16),

boxShadow: [

BoxShadow(

color: Colors.blue.withOpacity(0.3),

blurRadius: 8,

spreadRadius: 2,

),

],

),

child: Column(

children: [

// Runs buttons

Row(

mainAxisAlignment: MainAxisAlignment.spaceEvenly,

children: [0, 1, 2, 3, 4, 5, 6].map((runs) {

return Container(

decoration: BoxDecoration(

shape: BoxShape.circle,

gradient: LinearGradient(

colors: [

\_getRunButtonColor(runs),

\_getRunButtonColor(runs).withOpacity(0.7),

],

begin: Alignment.topLeft,

end: Alignment.bottomRight,

),

boxShadow: [

BoxShadow(

color: \_getRunButtonColor(runs).withOpacity(0.5),

blurRadius: 4,

spreadRadius: 1,

),

],

),

child: ElevatedButton(

onPressed: () => \_handleRunScored(runs),

style: ElevatedButton.styleFrom(

backgroundColor: Colors.transparent,

foregroundColor: Colors.white,

shadowColor: Colors.transparent,

minimumSize: const Size(50, 50),

shape: const CircleBorder(),

),

child: Text(

'$runs',

style: const TextStyle(

fontSize: 20,

fontWeight: FontWeight.bold,

),

),

),

);

}).toList(),

),

const SizedBox(height: 16),

// Extras buttons

Row(

mainAxisAlignment: MainAxisAlignment.spaceEvenly,

children: [

\_buildExtraButton('Wide', Colors.orange),

\_buildExtraButton('No Ball', Colors.red),

\_buildExtraButton('Bye', Colors.teal),

\_buildExtraButton('Leg Bye', Colors.indigo),

\_buildExtraButton(

'Penalty', Colors.purple), // Added penalty button

],

),

const SizedBox(height: 16),

// Wicket button

Container(

width: double.infinity,

height: 50,

decoration: BoxDecoration(

gradient: LinearGradient(

colors: [Colors.red.shade900, Colors.red.shade700],

begin: Alignment.topLeft,

end: Alignment.bottomRight,

),

borderRadius: BorderRadius.circular(25),

boxShadow: [

BoxShadow(

color: Colors.red.withOpacity(0.3),

blurRadius: 8,

spreadRadius: 2,

),

],

),

child: ElevatedButton.icon(

onPressed: () => \_handleWicket(),

icon: const Icon(Icons.sports\_cricket, size: 28),

label: const Text(

'WICKET',

style: TextStyle(fontSize: 18, fontWeight: FontWeight.bold),

),

style: ElevatedButton.styleFrom(

backgroundColor: Colors.transparent,

foregroundColor: Colors.white,

shadowColor: Colors.transparent,

shape: RoundedRectangleBorder(

borderRadius: BorderRadius.circular(25),

),

),

),

),

],

),

);

}

Widget \_buildExtraButton(String label, Color color) {

return Container(

decoration: BoxDecoration(

gradient: LinearGradient(

colors: [color, color.withOpacity(0.7)],

begin: Alignment.topLeft,

end: Alignment.bottomRight,

),

borderRadius: BorderRadius.circular(12),

boxShadow: [

BoxShadow(

color: color.withOpacity(0.3),

blurRadius: 4,

spreadRadius: 1,

),

],

),

child: ElevatedButton(

onPressed: () {

switch (label) {

case 'Wide':

\_handleWide();

break;

case 'No Ball':

\_handleNoBall();

break;

case 'Bye':

\_handleBye();

break;

case 'Leg Bye':

\_handleLegBye();

break;

case 'Penalty':

\_handlePenalty();

break;

}

},

style: ElevatedButton.styleFrom(

backgroundColor: Colors.transparent,

foregroundColor: Colors.white,

shadowColor: Colors.transparent,

padding: const EdgeInsets.symmetric(horizontal: 12, vertical: 12),

shape: RoundedRectangleBorder(

borderRadius: BorderRadius.circular(12),

),

),

child: Text(

label,

style: const TextStyle(

fontSize: 14,

fontWeight: FontWeight.bold,

),

),

),

);

}

Future<void> \_updateStats({

required int runsScored,

required bool isExtra,

required String extraType,

required bool countAsBall,

bool updateWides = false,

bool updateNoBalls = false,

int batsmanRuns = 0,

bool penaltyToBowlingTeam = false,

bool countBatsmanBall = false,

}) async {

try {

// Update total runs before checking target

final newTotalRuns = \_totalRuns + runsScored;

// Check if this will exceed target in second innings

if (!widget.isFirstInnings &&

\_target != null &&

newTotalRuns >= \_target!) {

// Update stats first

await \_updateMatchAndPlayerStats(

runsScored,

isExtra,

extraType,

countAsBall,

updateWides,

updateNoBalls,

batsmanRuns,

penaltyToBowlingTeam,

countBatsmanBall);

// End innings immediately

await \_endInnings();

return;

}

// Continue with normal stats update if target not reached

await \_updateMatchAndPlayerStats(

runsScored,

isExtra,

extraType,

countAsBall,

updateWides,

updateNoBalls,

batsmanRuns,

penaltyToBowlingTeam,

countBatsmanBall);

// Check other innings completion conditions

if (\_shouldEndInnings()) {

await \_endInnings();

} else if (\_currentBall >= 6) {

await \_handleOverComplete();

}

} catch (error) {

\_showError('Failed to update stats: $error');

print('Error details: $error');

}

}

Future<void> \_updateMatchAndPlayerStats(

int runsScored,

bool isExtra,

String extraType,

bool countAsBall,

bool updateWides,

bool updateNoBalls,

int batsmanRuns,

bool penaltyToBowlingTeam,

bool countBatsmanBall) async {

try {

// 1. Update match innings

await Supabase.instance.client

.from('match\_innings')

.update({

'total\_runs': \_totalRuns + runsScored,

'current\_over': \_currentOver,

'current\_ball': countAsBall ? \_currentBall + 1 : \_currentBall,

})

.eq('match\_id', widget.matchId)

.eq('innings\_number', widget.isFirstInnings ? 1 : 2)

.execute();

// Replace the batsman stats section in \_updateStats

if (!isExtra || batsmanRuns > 0 || countBatsmanBall) {

final batsmanStats = await Supabase.instance.client

.from('match\_player\_stats')

.select()

.eq('match\_id', widget.matchId)

.eq('innings\_number', widget.isFirstInnings ? 1 : 2)

.eq('player\_id', \_striker)

.single()

.execute();

await Supabase.instance.client

.from('match\_player\_stats')

.update({

'runs\_scored': (batsmanStats.data['runs\_scored'] ?? 0) +

(isExtra ? batsmanRuns : runsScored),

'balls\_faced': (batsmanStats.data['balls\_faced'] ?? 0) +

((countAsBall || countBatsmanBall) ? 1 : 0),

'fours': (batsmanStats.data['fours'] ?? 0) +

((batsmanRuns == 4 || (!isExtra && runsScored == 4)) ? 1 : 0),

'sixes': (batsmanStats.data['sixes'] ?? 0) +

((batsmanRuns == 6 || (!isExtra && runsScored == 6)) ? 1 : 0),

'dots\_faced': (batsmanStats.data['dots\_faced'] ?? 0) +

((batsmanRuns == 0 && countAsBall) ? 1 : 0),

})

.eq('match\_id', widget.matchId)

.eq('innings\_number', widget.isFirstInnings ? 1 : 2)

.eq('player\_id', \_striker)

.execute();

}

// 3. Rest of the method (bowler stats, state updates, etc.) remains the same...

if (\_striker == null || \_currentBowler == null) return;

try {

// Get current stats first

final bowlerStats = await Supabase.instance.client

.from('match\_player\_stats')

.select()

.eq('match\_id', widget.matchId)

.eq('innings\_number', widget.isFirstInnings ? 1 : 2)

.eq('player\_id', \_currentBowler)

.maybeSingle()

.execute();

// Update bowler stats

if (!penaltyToBowlingTeam) {

await Supabase.instance.client

.from('match\_player\_stats')

.update({

'runs\_conceded':

(bowlerStats.data?['runs\_conceded'] ?? 0) + runsScored,

'balls\_bowled': (bowlerStats.data?['balls\_bowled'] ?? 0) +

(countAsBall ? 1 : 0),

'dots\_bowled': (bowlerStats.data?['dots\_bowled'] ?? 0) +

(runsScored == 0 && countAsBall ? 1 : 0),

'wides':

(bowlerStats.data?['wides'] ?? 0) + (updateWides ? 1 : 0),

'no\_balls': (bowlerStats.data?['no\_balls'] ?? 0) +

(updateNoBalls ? 1 : 0),

})

.eq('match\_id', widget.matchId)

.eq('innings\_number', widget.isFirstInnings ? 1 : 2)

.eq('player\_id', \_currentBowler)

.execute();

}

// Update state

setState(() {

\_totalRuns += runsScored;

if (countAsBall) {

\_currentBall++;

\_currentOverBalls.add(

isExtra ? extraType[0].toUpperCase() : runsScored.toString());

} else {

\_currentOverBalls.add(extraType == 'wide' ? 'Wd' : 'Nb');

}

if (!isExtra && runsScored % 2 == 1) {

final temp = \_striker;

\_striker = \_nonStriker;

\_nonStriker = temp;

}

});

// Refresh stats

await \_refreshStats();

// Check for over completion

if (\_currentBall >= 6) {

await \_handleOverComplete();

} else if (\_currentOver >= widget.maxOvers) {

// If somehow we reach max overs without completing current over

await \_endInnings();

}

// Also check for innings completion conditions

if (\_shouldEndInnings()) {

await \_endInnings();

}

} catch (error) {

\_showError('Failed to update stats: $error');

print('Error details: $error');

}

} catch (error) {

\_showError('Failed to update stats: $error');

print('Error details: $error');

}

}

bool \_shouldEndInnings() {

// Check if innings should end based on various conditions

if (!widget.isFirstInnings && \_target != null) {

// Second innings conditions

if (\_target != null && \_totalRuns >= \_target!) {

// Target achieved

return true;

}

if (\_wickets >= 10) {

// All out

return true;

}

if (\_currentOver >= widget.maxOvers) {

// Overs completed

return true;

}

} else {

// First innings conditions

if (\_wickets >= 10 || \_currentOver >= widget.maxOvers) {

return true;

}

}

return false;

}

Future<void> \_handleRunScored(int runs) async {

if (\_striker == null || \_currentBowler == null) return;

try {

await \_updateStats(

runsScored: runs,

isExtra: false,

extraType: '',

countAsBall: true,

);

} catch (error) {

\_showError('Failed to update score: $error');

}

}

Future<void> \_refreshStats() async {

try {

// Fetch updated match stats

final matchStats = await Supabase.instance.client

.from('match\_innings')

.select()

.eq('match\_id', widget.matchId)

.eq('innings\_number', widget.isFirstInnings ? 1 : 2)

.single()

.execute();

// Fetch all player stats for this innings

final playerStats = await Supabase.instance.client

.from('match\_player\_stats') // Updated table name

.select()

.eq('match\_id', widget.matchId)

.eq('innings\_number', widget.isFirstInnings ? 1 : 2)

.execute();

if (mounted) {

setState(() {

// Update match stats

\_innings = matchStats.data;

\_totalRuns = matchStats.data['total\_runs'] ?? 0;

\_wickets = matchStats.data['wickets'] ?? 0;

// Update player stats

\_playerStats =

List<Map<String, dynamic>>.from(playerStats.data ?? []);

});

}

} catch (error) {

\_showError('Failed to refresh stats: $error');

}

}

Future<void> \_handleWide() async {

try {

final result = await showDialog<Map<String, dynamic>>(

context: context,

builder: (context) => WideDialog(allowedRuns: [0, 1, 2, 3, 4]),

);

if (result != null) {

final additionalRuns = result['runs'] as int;

final isWicket = result['isWicket'] as bool;

await \_updateStats(

runsScored: 1 + additionalRuns,

isExtra: true,

extraType: 'wide',

countAsBall: false,

updateWides: true,

);

if (isWicket) {

await \_handleWicket(allowedDismissals: [DismissalType.stumped]);

}

}

} catch (error) {

\_showError('Failed to process wide: $error');

}

}

Future<void> \_handleNoBall() async {

try {

final result = await showDialog<Map<String, dynamic>>(

context: context,

builder: (context) => NoBallDialog(allowedRuns: [0, 1, 2, 3, 4, 6]),

);

if (result != null) {

final runsScored = result['runs'] as int;

final isWicket = result['isWicket'] as bool;

// Update extras tracking

setState(() {

\_currentOverExtras['noball\_runs']?.add(runsScored + 1);

});

// Update stats with runs counting for batsman

await \_updateStats(

runsScored: 1 + runsScored, // 1 for no ball + runs scored

isExtra: true,

extraType: 'noball',

countAsBall: false, // Don't count in bowler's overs

updateNoBalls: true,

batsmanRuns: runsScored, // Credit runs to batsman

countBatsmanBall: true, // Add this to count the ball for batsman

);

if (isWicket) {

await \_handleWicket(

allowedDismissals: [DismissalType.runOut, DismissalType.hitWicket],

);

}

}

} catch (error) {

\_showError('Failed to process no ball: $error');

}

}

Future<void> \_handleWideWicket() async {

// Only allow stumping on wide

await \_handleWicket(allowedDismissals: [DismissalType.stumped]);

}

Future<void> \_handleNoBallWicket() async {

// Only allow run out and hit wicket on no ball

await \_handleWicket(

allowedDismissals: [DismissalType.runOut, DismissalType.hitWicket],

);

}

Future<void> \_handleBye() async {

try {

int? runs;

await showDialog(

context: context,

builder: (context) => AlertDialog(

title: const Text('Bye Runs'),

content: Row(

mainAxisSize: MainAxisSize.min,

children: [1, 2, 3, 4].map((run) {

return Padding(

padding: const EdgeInsets.symmetric(horizontal: 4),

child: ElevatedButton(

onPressed: () {

runs = run;

Navigator.pop(context);

},

child: Text('$run'),

),

);

}).toList(),

),

),

);

if (runs != null) {

await \_updateStats(

runsScored: runs!,

isExtra: true,

extraType: 'bye',

countAsBall: true,

);

}

} catch (error) {

\_showError('Failed to process bye: $error');

}

}

Future<void> \_handleLegBye() async {

try {

int? runs;

await showDialog(

context: context,

builder: (context) => AlertDialog(

title: const Text('Leg Bye Runs'),

content: Row(

mainAxisSize: MainAxisSize.min,

children: [1, 2, 3, 4].map((run) {

return Padding(

padding: const EdgeInsets.symmetric(horizontal: 4),

child: ElevatedButton(

onPressed: () {

runs = run;

Navigator.pop(context);

},

child: Text('$run'),

),

);

}).toList(),

),

),

);

if (runs != null) {

await \_updateStats(

runsScored: runs!,

isExtra: true,

extraType: 'legbye',

countAsBall: true,

);

}

} catch (error) {

\_showError('Failed to process leg bye: $error');

}

}

Future<void> \_handleOverComplete() async {

try {

// Check for maiden over

final isMaiden = \_currentOverBalls.every((ball) => ball == '0');

// Get current bowler stats

final bowlerStats = await Supabase.instance.client

.from('match\_player\_stats')

.select()

.eq('match\_id', widget.matchId)

.eq('innings\_number', widget.isFirstInnings ? 1 : 2)

.eq('player\_id', \_currentBowler)

.maybeSingle()

.execute();

if (bowlerStats.data != null) {

// Update existing bowler stats

await Supabase.instance.client

.from('match\_player\_stats')

.update({

'is\_bowling': false,

'maidens': isMaiden

? (bowlerStats.data['maidens'] ?? 0) + 1

: bowlerStats.data['maidens'] ?? 0,

'balls\_bowled':

(bowlerStats.data['balls\_bowled'] ?? 0) + (6 - \_currentBall),

'overs\_bowled': \_currentOver + 1,

})

.eq('match\_id', widget.matchId)

.eq('innings\_number', widget.isFirstInnings ? 1 : 2)

.eq('player\_id', \_currentBowler)

.execute();

}

// Show new bowler dialog

final newBowler = await showDialog<String>(

context: context,

barrierDismissible: false,

builder: (context) => AlertDialog(

title: const Text('Select New Bowler'),

content: SingleChildScrollView(

child: Column(

mainAxisSize: MainAxisSize.min,

children: \_bowlingTeam

.where((player) =>

player['id'].toString() != \_currentBowler &&

!\_playerStats.any((stats) =>

stats['player\_id'].toString() ==

player['id'].toString() &&

stats['is\_bowling'] == true))

.map((player) => ListTile(

title: Text(player['name']),

onTap: () =>

Navigator.pop(context, player['id'].toString()),

))

.toList(),

),

),

),

);

if (newBowler != null) {

// Initialize or update new bowler stats

final newBowlerStats = await Supabase.instance.client

.from('match\_player\_stats')

.select()

.eq('match\_id', widget.matchId)

.eq('innings\_number', widget.isFirstInnings ? 1 : 2)

.eq('player\_id', newBowler)

.maybeSingle()

.execute();

if (newBowlerStats.data != null) {

await Supabase.instance.client

.from('match\_player\_stats')

.update({

'is\_bowling': true,

'has\_bowled': true,

})

.eq('match\_id', widget.matchId)

.eq('innings\_number', widget.isFirstInnings ? 1 : 2)

.eq('player\_id', newBowler)

.execute();

} else {

await Supabase.instance.client.from('match\_player\_stats').insert({

'match\_id': widget.matchId,

'innings\_number': widget.isFirstInnings ? 1 : 2,

'player\_id': newBowler,

'team\_id': widget.bowlingTeamId,

'is\_bowling': true,

'has\_bowled': true,

'balls\_bowled': 0,

'runs\_conceded': 0,

'maidens': 0,

'wickets': 0,

'overs\_bowled': 0,

}).execute();

}

setState(() {

\_currentBowler = newBowler;

\_currentOver++;

\_currentBall = 0;

\_currentOverRuns = 0;

\_currentOverBalls.clear();

\_currentOverExtras.clear();

});

}

} catch (error) {

\_showError('Failed to complete over: $error');

print('Error details: $error');

}

}

Future<void> \_handlePenalty() async {

try {

int? runs;

String? team;

await showDialog(

context: context,

builder: (context) => AlertDialog(

title: const Text('Penalty Runs'),

content: Column(

mainAxisSize: MainAxisSize.min,

children: [

DropdownButtonFormField<String>(

decoration: const InputDecoration(labelText: 'Award to'),

items: [

DropdownMenuItem(

value: 'batting', child: Text('Batting Team')),

DropdownMenuItem(

value: 'bowling', child: Text('Bowling Team')),

],

onChanged: (value) => team = value,

),

const SizedBox(height: 8),

Row(

mainAxisAlignment: MainAxisAlignment.spaceEvenly,

children: [5].map((run) {

return ElevatedButton(

onPressed: () {

runs = run;

Navigator.pop(context);

},

child: Text('$run'),

);

}).toList(),

),

],

),

),

);

if (runs != null && team != null) {

await \_updateStats(

runsScored: runs!,

isExtra: true,

extraType: 'penalty',

countAsBall: false,

penaltyToBowlingTeam: team == 'bowling',

);

}

} catch (error) {

\_showError('Failed to process penalty: $error');

}

}

Future<void> \_handleWicket({List<DismissalType>? allowedDismissals}) async {

try {

final result = await showDialog<Map<String, dynamic>>(

context: context,

builder: (context) => WicketDialog(

bowlingTeam: \_bowlingTeam,

allowedDismissals: allowedDismissals,

currentBowler: \_currentBowler,

),

);

if (result != null) {

final dismissalType = result['type'] as String;

final fielderId = result['fielder'] as String?;

// Don't credit wicket to bowler for run outs

final shouldCreditBowler = dismissalType != 'runOut';

// Get current bowler stats

final bowlerStats = await Supabase.instance.client

.from('match\_player\_stats')

.select()

.eq('match\_id', widget.matchId)

.eq('innings\_number', widget.isFirstInnings ? 1 : 2)

.eq('player\_id', \_currentBowler)

.maybeSingle()

.execute();

// Update bowler's wickets if applicable

if (shouldCreditBowler) {

await Supabase.instance.client

.from('match\_player\_stats')

.update({

'wickets': (bowlerStats.data?['wickets'] ?? 0) + 1,

})

.eq('match\_id', widget.matchId)

.eq('innings\_number', widget.isFirstInnings ? 1 : 2)

.eq('player\_id', \_currentBowler)

.execute();

}

// Update batsman dismissal

await Supabase.instance.client

.from('match\_player\_stats')

.update({

'is\_out': true,

'is\_batting': false,

'dismissal\_type': dismissalType,

'dismissed\_by\_bowler\_id':

shouldCreditBowler ? \_currentBowler : null,

'dismissed\_by\_fielder\_id': fielderId,

})

.eq('match\_id', widget.matchId)

.eq('innings\_number', widget.isFirstInnings ? 1 : 2)

.eq('player\_id', \_striker)

.execute();

setState(() {

\_wickets++;

\_currentOverBalls.add('W');

});

// Show new batsman dialog if wickets < 10

if (\_wickets < 10) {

await \_showNewBatsmanDialog();

} else {

await \_endInnings();

}

}

} catch (error) {

\_showError('Failed to process wicket: $error');

}

}

// Add this helper method to check if innings exists

Future<bool> \_checkInningsExists(int inningsNumber) async {

final response = await Supabase.instance.client

.from('match\_innings')

.select()

.eq('match\_id', widget.matchId)

.eq('innings\_number', inningsNumber)

.execute();

return response.data != null && (response.data as List).isNotEmpty;

}

Future<void> \_endInnings() async {

try {

// Step 1: Update current innings

await Supabase.instance.client

.from('match\_innings')

.update({

'is\_complete': true,

'total\_runs': \_totalRuns,

'wickets': \_wickets,

'current\_over': \_currentOver,

'current\_ball': \_currentBall,

})

.eq('match\_id', widget.matchId)

.eq('innings\_number', widget.isFirstInnings ? 1 : 2)

.execute();

if (widget.isFirstInnings) {

// Check if second innings already exists

final secondInningsExists = await \_checkInningsExists(2);

if (!secondInningsExists) {

// Create second innings only if it doesn't exist

await Supabase.instance.client.from('match\_innings').insert({

'match\_id': widget.matchId,

'innings\_number': 2,

'batting\_team\_id': widget.bowlingTeamId,

'bowling\_team\_id': widget.battingTeamId,

'total\_runs': 0,

'wickets': 0,

'current\_over': 0,

'current\_ball': 0,

'is\_complete': false,

'target': \_totalRuns + 1,

}).execute();

}

if (mounted) {

await showDialog(

context: context,

barrierDismissible: false,

builder: (context) => AlertDialog(

title: const Text('First Innings Complete'),

content: Column(

mainAxisSize: MainAxisSize.min,

children: [

Text('${\_getTeamName(widget.battingTeamId)} Innings'),

Text('Total: $\_totalRuns/$\_wickets'),

Text('Overs: $\_currentOver.${\_currentBall}'),

const SizedBox(height: 16),

Text(

'${\_getTeamName(widget.bowlingTeamId)} needs ${\_totalRuns + 1} to win',

style: const TextStyle(fontWeight: FontWeight.bold),

),

],

),

actions: [

ElevatedButton(

onPressed: () {

Navigator.pushReplacement(

context,

MaterialPageRoute(

builder: (context) => ScoreUpdatingPage(

matchId: widget.matchId,

team1Id: widget.team1Id,

team2Id: widget.team2Id,

battingTeamId: widget.bowlingTeamId,

bowlingTeamId: widget.battingTeamId,

maxOvers: widget.maxOvers,

isFirstInnings: false,

tossWinnerId: widget.tossWinnerId,

tossChoice: widget.tossChoice,

),

),

);

},

child: const Text('Start Second Innings'),

),

],

),

);

}

} else {

// Match is complete

await Supabase.instance.client

.from('matches')

.update({

'status': 'completed',

'updated\_at': DateTime.now().toIso8601String(),

})

.eq('id', widget.matchId)

.execute();

if (mounted) {

Navigator.pushReplacement(

context,

MaterialPageRoute(

builder: (context) => MatchSummaryPage(

matchId: widget.matchId,

team1Id: widget.team1Id,

team2Id: widget.team2Id,

),

),

);

}

}

} catch (error) {

\_showError('Failed to end innings: $error');

print('Error details: $error');

}

}

Future<void> \_showNewBatsmanDialog() async {

try {

final newBatsman = await showDialog<String>(

context: context,

barrierDismissible: false,

builder: (context) => AlertDialog(

title: const Text('Select New Batsman'),

content: SingleChildScrollView(

child: Column(

mainAxisSize: MainAxisSize.min,

children: \_battingTeam

.where((player) =>

player['id'].toString() != \_nonStriker &&

!\_playerStats.any((stats) =>

stats['player\_id'].toString() ==

player['id'].toString() &&

stats['has\_batted'] == true))

.map((player) => ListTile(

title: Text(player['name']),

onTap: () =>

Navigator.pop(context, player['id'].toString()),

))

.toList(),

),

),

),

);

if (newBatsman != null) {

// Initialize new batsman stats directly

await Supabase.instance.client.from('match\_player\_stats').insert({

'match\_id': widget.matchId,

'innings\_number': widget.isFirstInnings ? 1 : 2,

'player\_id': newBatsman,

'team\_id': widget.battingTeamId,

'is\_batting': true,

'is\_on\_strike': true,

'has\_batted': true,

'runs\_scored': 0,

'balls\_faced': 0,

'fours': 0,

'sixes': 0,

}).execute();

setState(() {

\_striker = newBatsman;

});

}

} catch (error) {

\_showError('Failed to select new batsman: $error');

}

}

}

**REFERENCES**

1. https://developer.android.com/guide/topics/ui/layout/linear

2. <https://firebase.google.com/docs/database/unity/retrieve-data>

3. <https://developer.android.com/guide/topics/ui/layout/cardview>