PocketSports

A Digital Coaching App

User & Developer Manual

Team Members:

Garrett Gmeiner - ggmeiner2021@my.fit.edu

Tyler Ton - tton2021@my.fit.edu

Parker Cummings - pcummings2021@my.fit.edu

Taylor Carlson - tcarlson2021@my.fit.edu

Faculty Advisor:

Fitzroy Nembhard - fnembhard@fit.edu

Client:

Brad MacArthur - bmacarthur@fit.edu

Florida Institute of Technology

4/8/2025

User Manual

1. Introduction

1.1 Purpose of the Manual

This manual serves as a comprehensive guide for users of the PocketSports Digital Coaching App. It outlines how to install, navigate, and utilize all of the application's core features. The guide is meant to act as a lifeline to support coaches, players, and owners in managing teams and improving performance for all specified sports such as volleyball, lacrosse, and basketball.

1.2 Overview of the Application

PocketSports is a web-based sports team management platform designed for coaches, players, and owners of teams. It allows users to create teams, manage rosters, set goals, track practice plans, and analyze player stats. The app consolidates features typically spread across multiple apps into one streamlined interface for the ease of usage.

1.3 Document Conventions

- Bold headers could indicate important information or major sections.
- Screenshots may be referenced where it is applicable.

2. Getting Started

2.1 System Requirements

Browser: Chrome, Firefox, or Safari (latest versions) or any other browser

Device: PC, Mac, Tablet, Windows

Internet connection required

2.2 Installation/Setup Instructions

- Navigate to the PocketSports web app URL.
- No download is required —-- web-based interface.
- Create a new account with email verification.
- Login if you already have credentials.
- Begin creating/joining player teams.

2.3 Account Setup & Login

- Register using first name, last name, email, password, password confirmation.
- Wait for email confirmation/verification to be sent to your email address.
- Get the 6 digit verification code to finish creating an account.
- Login using registered email and password.
- Upon login, users will be redirected to a personalized dashboard based on your assigned role.

3. Using the Features of the System

3.1 Feature Overview

- Create and manage teams and rosters
- Build and track goals
- Design and execute practice plans
- View calendar events and practice results
- Create and edit drills

3.2 Detailed Feature Walkthroughs

3.2.1 Feature 1: Build and track goal

Coaches and players can create, view, and update goals. Goals can be categorized as individual or team-based. Progress towards the completion of the goals is represented through badges and a progress bar. When a user makes headway or progress towards a specified goal they can then update the specified metric resulting in a new goal summary percentage along with the corresponding achievement badges.

3.2.2 Feature 2: Design and execute practice plans

Coaches and owners of teams can build practice plans to determine what they need to get done. Practice plans include drill selection, duration, and any other details users want to specify in addition. Practice plans are then stored in a drill bank and can be reused. They can also be exported to a pdf format for easy sharing between teams and coaches. Players can then view practice plans created by their coaches to then tailor their training to meet the specified drills and tasks in the practice plan.

3.2.3 Feature 3: Create and manage teams and rosters

Coaches and owners can create new teams and assign user roles such as coach or player. Each team will have a specific/ unique join code where other players can use to join after they create an account. Coaches in addition, can add or remove players from the roster. Each player will

have a detailed player profile criteria such as height, size, weight. Roster information includes player stats. Players' stats can be filtered to only display information desired.

3.2.4 Feature 4: View calendar events and practice results

Each user will have a calendar where they can input upcoming events that they need to look out for. Events are logged and saved for further usage. An event creation consists of date, title, and details. Users can click and traverse between dates on the calendar to view upcoming practices, games, and team events through the interface. Practice results include coach feedback and player-specific stats.

3.2.4 Feature 5: Create and edit drills

Coaches and owners of teams can create and edit drills using a drag and drop interface. The editor acts as an interface to efficiently diagram the necessary drills. There are X, O, and arrows in the editor for the users to choose from. Clear backgrounds in the editors consist of field views from specific sports such as basketball, volleyball, and lacrosse in a clear format. Drills are then stored in a drill bank and can be reused.

3.3 Navigation and Interface Tips

- Use the top navigation bar to switch between Home, Calendar, Goals, Roster, and Practice Plans
- Role-based UI: Coaches see team management tools, players see personal goals and feedback
- Most pages include quick buttons for adding new entries

4. User-Specific Examples

4.1 End Users vs. Administrators

Players: View goals, calendar, stats, and feedback

Coaches: Create/edit practice plans and drills, manage goals and players

Owners: Full admin rights to manage users and teams

4.2 Role-Based Scenarios

4.2.1 Example 1:

A coach logs in to update a team goal, then uploads a practice plan for next week. This new updated team goal is then shown on all player pages. Players can also view the upcoming practice plans that the coach uploads.

4.2.2 Example 2:

A player checks their personalized calendar and updates progress on a shooting goal. Players can create and edit their own goals without coach interference.

5. Troubleshooting and FAQs

5.1 Common Issues

Can’t log in: Check email/password, use "Forgot Password".

Practice plan not saving: Ensure all required fields are filled out.

Goal progress not updating: Check if practice results are linked to correct tags.

5.2 FAQ Section:

Q: Can parents track player goals?

A: Yes, if the coach has enabled that view.

Q: Can I use this on my phone?

A: No, the app is only compatible on web-based applications.

Q: Can players create their own goals?

A: Yes, players can create personal goals visible only to them or shared with coaches.

Q: How do I reset my password?

A: Click “Forgot Password” on the login page to receive a reset link via email.

Q: Can multiple coaches manage the same team?

A: No, teams can only have one coach or owner with privileged access and responsibilities.

Q: How do I join an existing team?

A: After registration, a coach or owner must send you a team invite via email or join code that you can input.

6. Support and Contact Information

6.1 Customer Support

Users can use the contact us form for more information or any additional help.

6.2 Feedback Mechanisms

Users can use the contact us form for recommendations or any additional information.

Developer Manual

1. Introduction

This document serves as a key resource to onboard developers, guide ongoing contributions, and maintain consistency throughout the project's evolution. It explains the underlying system architecture, the code structure, and the procedures for setting up a development environment.

1.1 Purpose of the Developer Manual

The purpose of this manual is to provide a centralized reference. In other words, to equip developers with detailed guidelines, architectural insights, and best practices needed to develop, maintain, and expand the project. Additionally, the goal is to standardize processes and practices and ensure consistency in code quality, documentation, and development workflows across the team. Finally, it is imperative that it is used to facilitate onboarding by offering a clear starting point for new developers to quickly understand project goals, directory layout, and environment setup.

1.2 Audience

This document is intended for developers and engineers including both new and experienced team members responsible for writing and maintaining the code, technical leads and architects overseeing system design and integration, quality assurance (QA) teams to understand the structure and flow of the system for effective testing, devops and release engineers for insights into environment configuration, continuous integration, and deployment procedures, and project stakeholders interested in a technical overview of the system.

1.3 Document History and Versioning:

v1.0.0: [4/13/2025] — Initial version of the developer manual.

2. System Architecture Overview

2.1 Architecture Diagram

Figure 1 provides a high-level visual representation of the entire system. Figures 2 and 3 provide the navigational block diagrams for the different roles in the system.

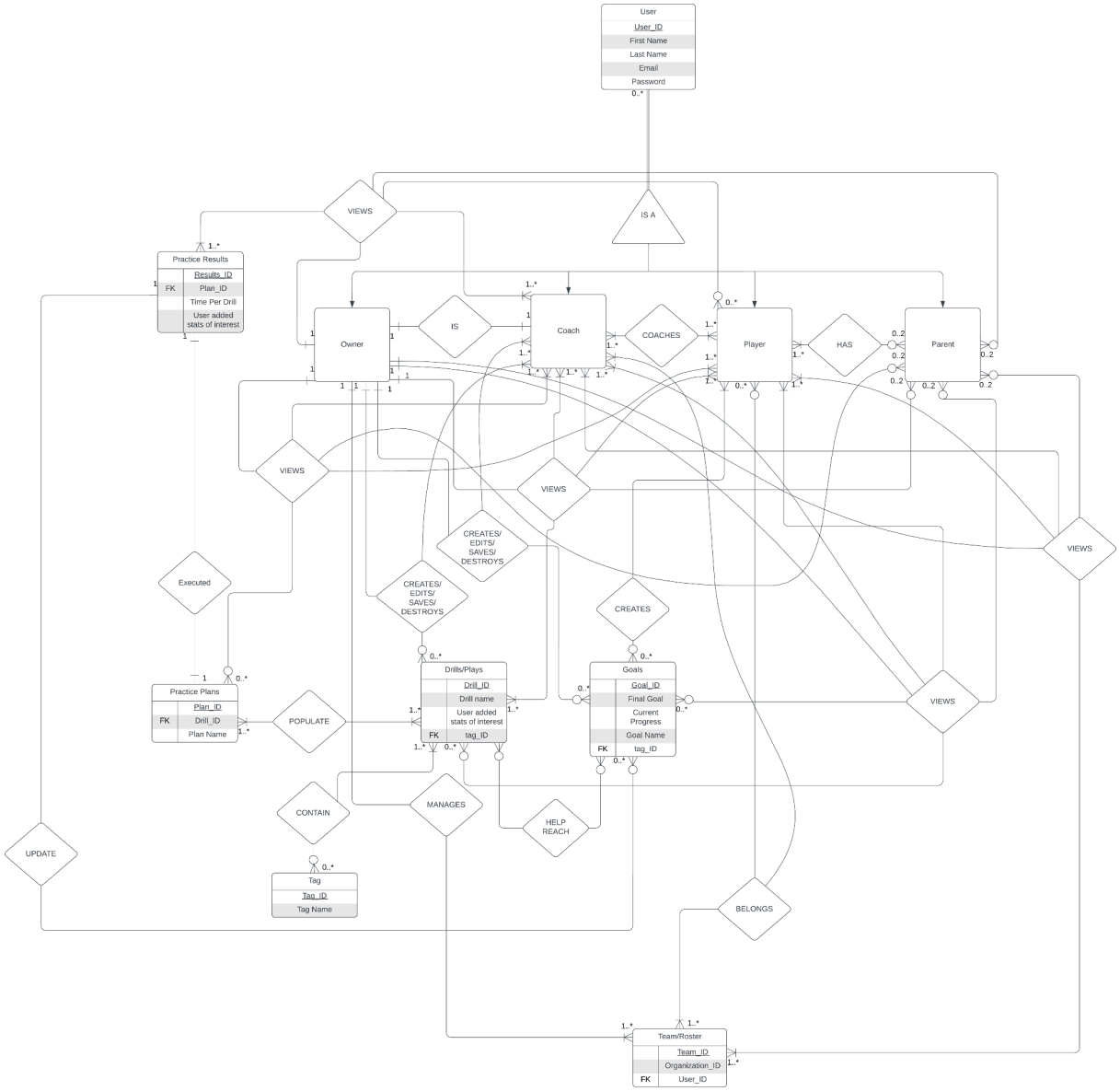

Figure 1: Database design

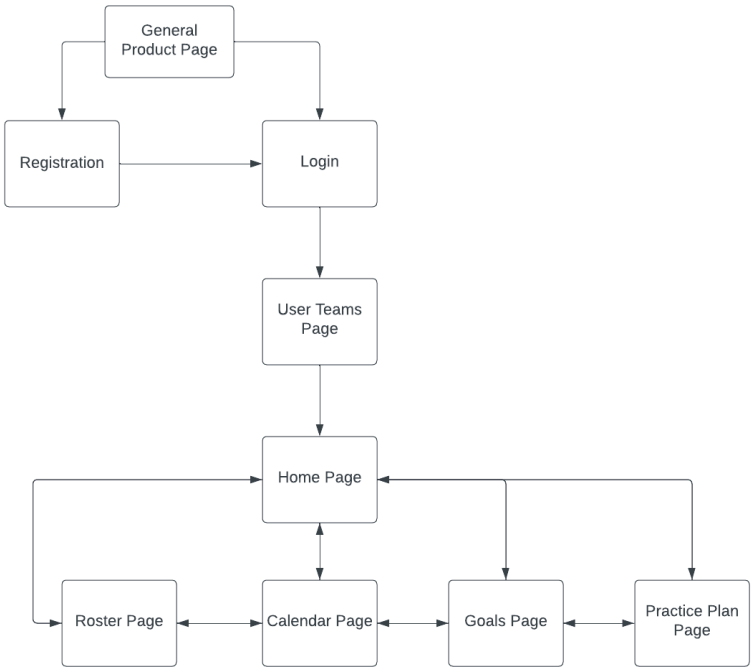

Figure 2: Owner/Coach Navigational Layout

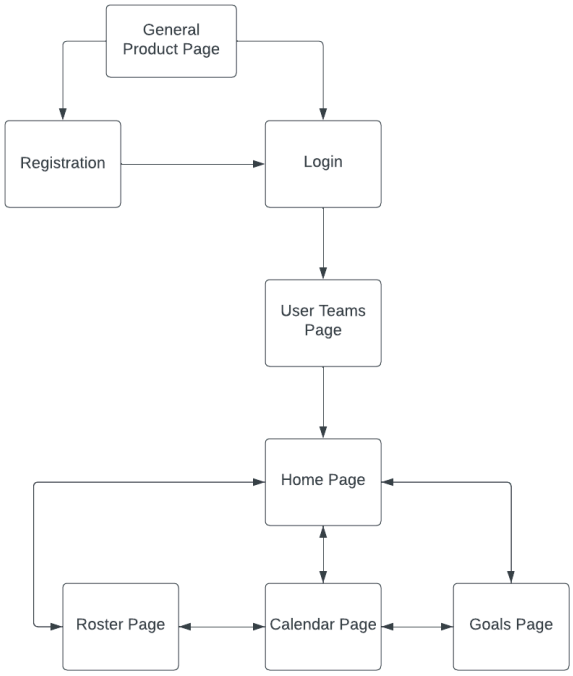

Figure 3: Player/Parent Navigational Layout

2.2 Component Descriptions

2.2.1 Frontend

The role of the frontend is to deliver the user interface, ensuring that users can interact with the system efficiently and intuitively. It is responsible for rendering dynamic content, managing client-side state, and handling user interactions and integrating with backend APIs.

2.2.2 Backend

The role of the backend is to manage the core business logic, data processing, and persistent storage. It is responsible for providing secure and scalable RESTful APIs, handling data validation, authentication, and authorization, and interfacing with databases and external services.

2.2.3 Integration

The role of the integration is to act as the connector between the core system and external systems or microservices. It is responsible for managing API integrations with third-party services (e.g., payment gateways, messaging systems), facilitating internal communications between microservices, and ensuring data consistency and efficient asynchronous processing.

2.3 Data Flow and Interactions

This section describes the lifecycle of data as it moves through the system from the moment a user interacts with the frontend until it is processed, stored, and possibly sent back as a response. Data flows through several key stages in the system, beginning with user input, where data is entered through the frontend interface. This input then triggers API requests, as the frontend sends HTTP/HTTPS calls to the backend. Upon receiving these requests, the backend processes them by applying the necessary business rules and interacting with the database. Once the processing is complete, the results are stored in a persistent data store to ensure that information is reliably maintained. Finally, the processed data is delivered back to the frontend, which updates the user interface accordingly. Throughout this entire process, additional considerations are addressed: robust error handling protocols are implemented to manage and log errors consistently, and security measures such as encryption during data transit and secured endpoints are employed to protect sensitive information.

3. Source Code Structure

3.1 Directory Layout

The directory layout is designed to promote modularity and clarity, making the project easy to navigate.

```
.
├── Field Views/
│   ├── Lacrosse.jpg
│   ├── Basketball.jpeg
│   └── Volleyball.jpeg
├── Backend/
│   ├── models/
│   │   └── Database Schemas
│   ├── node_modules
│   ├── index.js
│   ├── package-lock.json
│   └── package.json
├── Frontend/
│   ├── public/
│   │   ├── Arrows.mp4
│   │   └── FieldViews.jpg
│   ├── src/
│   │   ├── assets/
│   │   │   └── react.svg
│   │   └── Main files (.jsx and .css)
│   ├── vite-project/
│   │   ├── public/
│   │   │   └── vite.svg
│   │   ├── src/
│   │   │   ├── assets/
│   │   │   │   └── react.svg
│   │   │   ├── App.css
│   │   │   ├── App.jsx
│   │   │   ├── index.css
│   │   │   └── main.jsx
│   │   ├── eslint.config.js
│   │   ├── index.html
│   │   ├── package-lock.json
│   │   ├── package.json
│   │   └── vite.config.js
│   ├── eslint.config.js
│   ├── index.html
```

```
│   ├── package-lock.json
│   ├── package.json
│   └── vite.config.js
├── public/
│   └── vite.svg
└── src/
    ├── assets/
    │   └── react.svg
    ├── App.css
    ├── App.jsx
    ├── index.css
    └── main.jsx
```

3.2 Module and File Descriptions

3.2.1 Core Files

These files are the backbone of the application, containing the primary logic and key entry points. Each file has its own corresponding .css file.

- **App.jsx**: Central React component that sets up client-side routing by mapping different URL paths to their respective page components.
- **main.jsx**: Acts as the application's entry point, rendering the primary App component within a router to enable client-side navigation.
- **index.js**: Main backend server built with Express that defines API endpoints for user registration, authentication, team management, events, drills, and various other functionalities.
- **calendar.jsx**: React component that renders an interactive calendar interface, manages event creation, and fetches scheduled events for display.
- **contactPage.jsx**: React component that provides a contact form interface, allowing users to send messages that are processed by the backend.
- **drills.jsx**: React component that offers an interface for creating, editing, and managing sports drills by integrating dynamic canvas elements, draggable components, and drill templates.
- **goals-page.jsx**: React component that manages and displays team goals, enabling users to track progress, create new goals, and view completed achievements.
- landing-page.jsx: serves as the landing page component, showcasing app features through engaging visuals and guiding users to login or register.
- **login.jsx**: Provides the login interface, handling user authentication, password visibility toggling, and redirection upon successful login.
- **registration.jsx**: Offers a user registration interface with input validation, password strength checks, and initiates email verification upon successful sign-up.

- **Roster.jsx**: Displays the team roster by listing player details, managing role assignments, and allowing users to edit personal information.
- **TeamHome.jsx**: Represents the team home page, featuring recent goals, upcoming events, and an overview of team-specific information.
- **Teams.jsx**: Handles team creation and joining, displays a list of teams, and manages user profile updates within the context of team interactions.
- **Verify.jsx**: Implements the email verification process by collecting a six-digit code from the user and verifying it with the backend.

3.2.2 Configuration Files

These files are responsible for defining environment-specific settings and operational parameters.

- package-lock.json
- package.json

3.3 Environment Setup

Installation of a code editor (VSCode), Node and its npm, and MongoDB Compass are required.
Setup Instructions:

1. Clone the Repository:
2. Execute: git clone on the repository
3. Install Dependencies using npm install

Two different thermals should be opened. One should be in the frontend, and the other should be in the backend. Run the following commands in each:

1. Frontend: npm run dev
2. Backend: npm start

4. Code Conventions and Guidelines

4.1 Coding Standards

For our codebase, we use a standardized iteration of 2 spaces, since we are using JavaScript. Additionally, the maximum number of characters on a single line should not exceed 120. All names of files, variables or functions should be meaningful. For example, if we are creating a drill, the function should be called createDrill. Variables should be in camel case. Additionally, they should start lowercase unless a function or class name. Also, functions and classes should be kept short and focused on a single responsibility. Structured error handling should be used for applications where they will not always work.

4.2 Version Control and Branching Strategy

Changes should be tracked systematically, and collaboration should be organized through a clear branching and merging process. Git should be used as the intermediary between the local software and the GitHub repository.There is a Main/Production branch that contains the stable code that is ready for deployment. There is also a development branch that serves as an integration branch for features and bug fixes before merging into the main branch. Developers should pull the latest frequently using “git pull”. Each task should be linked to a Jira ticket, and the branch should be created off of the development branch. The branch name should include the ticket descriptor and should describe the ticket a bit. For example, FIT-167-create-user-dev-manual. When ready to commit, ensure the command “git status” is run. Only add necessary files. Run a “git status” again to ensure the correct files are added in green. Finally use a git commit statement to save the work to the branch. This commit message should be clear and concise that gives the ticket name and explains what was accomplished. For example, git commit -m "FIT-167 Finished the User and Dev Manual". The code must next be pushed from the local machine to the remote branch. If the branch does not exist, one should follow through setting the push upstream. A pull request should be created to merge into the development branch. Additionally, at least two team members should review and test the code before it gets merged.

4.3 Code Commenting and Documentation

Code must be self-explanatory and maintainable by providing necessary context within the code itself, as well as comprehensive external documentation. Use inline comments to explain non-obvious logic within code lines. Keep comments brief and directly above the code they describe unless they are short. Use block comments for providing overviews or explanations for sections of code and complex algorithms. Ensure block comments are kept up-to-date as the code evolves. Comments should add value by explaining “why” something is done, not just “what” is being done. Avoid redundant comments that duplicate the code. With newly created features, please be sure to update this manual and when applicable.

5. Detailed Documentation of Methods and Functions

5.1 API Documentation

The API for the backend uses several endpoints to communicate with the frontend and display features and information to the user.

Authentication & Account Management

- POST /register – Register a new user account.
- POST /login – Authenticate user and return user ID.
- POST /forgot-password – Send password reset email with token.
- POST /reset-password/:token – Reset user password using the token.
- POST /verifyemail – Send a verification code to user's email.
- POST /verifycode – Verify user's email using the code.

User Profile

- GET /registers/:userId – Retrieve a user's basic information (name, email, profile picture).
- POST /upload-profile/:userId – Upload and store a user's profile picture.

Team Management

- POST /teams – Create a new team and assign the creator as the owner.
- GET /teams – Retrieve all teams that a user belongs to.
- GET /teams/:teamId – Get detailed information about a specific team.
- PUT /teams/:teamId/extraInfoVisibility – Update visibility settings for additional team info.
- DELETE /teams/:teamId – Delete a team (only allowed for the creator).

Team Membership

- POST /invite-to-team – Send an email invitation to join a team using its team code.
- POST /joinTeam – Join a team using a team code.

- POST /leaveTeam – Leave a team as a member.
- GET /useronteams – Get all users on a specific team.
- DELETE /useronteams – Remove a user from a team.
- PUT /useronteams/role – Change a user’s role on a team.
- PUT /useronteams/:userId – Update a user's information on a team.

Event Management

- POST /events – Create a new team event (e.g., game, practice).
- GET /events – Get all events for a given team.
- PUT /events/:eventId – Update an event's details.
- DELETE /events/:eventId – Delete an event.
- PUT /events/:eventId/feedback – Add feedback to a specific event for a player.

Feedback

- POST /feedback – Submit feedback for a player on a specific event.
- GET /feedback/:playerId/:eventId – Retrieve feedback given to a player for a specific event.

Goals

- POST /goals – Create a goal for a team or player.
- GET /goals – Get all goals for a team.
- PUT /goals/:goalId – Update goal information such as progress.
- DELETE /goals/:goalId – Delete a goal.

Practice Plans

- POST /practiceplans – Create a new practice plan with drills.
- GET /practiceplans – Get all practice plans for a team (query param: teamId).
- GET /practiceplans/:id – Get a specific practice plan by ID.
- GET /practiceplans/:teamId – Get all practice plans for a team by teamId param.
- DELETE /practiceplans/:planId – Delete a practice plan.

Drill Bank

- POST /drillbank – Save a new drill (PDF + metadata) to the drill bank.
- GET /drillbank/team/:teamId – Get all drills for a team.
- GET /drillbank/:drillName – Download a drill PDF by its name.
- DELETE /drillbank/:drillId – Delete a drill from the drill bank.

Drill Tags

- POST /drilltags – Create a new tag for a team (if not already present).
- GET /drilltags/:tag – Check if a tag with a given name exists.
- GET /drilltags/team/:teamId – Get all tag names for a specific team.

Drill Stats

- POST /drillStats – Create a new drill stat for a team.
- GET /drillStats/:stat – Check if a stat exists by name.
- GET /drillStats/team/:teamId – Get all stats associated with a team.

User Stats

- GET /userStats/:userId – Retrieve stats for a user on a given team (teamId in headers).

Contact

- POST /contact – Submit a message through the contact form.

5.2 Class and Function Descriptions

5.2.1 Major Functions

- App()
  - Holds the routing structure for the base of the application
- CalendarPage()
  - Contains the logic for adding events to the calendar, editing them, and displaying them to the user
- Drills()
  - Contains the logic for adding, removing, and updating drills in the drill bank
- GoalsPage()
  - Contains the logic for creating new goals for both a player and a coach.
- LandingPage()
  - Contains the logic for creating and displaying the landing page once a user registers and logs in
- Login()
  - Allows the user to input a username and password and be displayed the information tied with their account
- PracticePlans()
  - Contains the logic to create, remove, and update practice plans from the drills available in the drill bank
- RegistrationPage()
  - Contains the logic to create an account, communicating with the backend to add a user into the MongoDB database
- Roster()
  - Contains the logic to display to the user the players, owners, and coaches on a team
- HomePage()
  - Contains the logic to create the team home page, when a user selects a team to view. This has the logic to display the calendar, goals, and other information to the user

- TeamsPage()
    - Contains the logic to display to the user the list of teams they are members of. A user can create a new team, accept an invite from another team, or leave a team.
- Verify()
    - Contains the logic for verifying email addresses by sending a one-time code from a SMTP server to the user's email.

5.2.2 Methods/Functions

- CalendarPage()
    - getUserDetails
        - Purpose: Retrieve user details from MongoDB given a specific player ID.
        - I/O
            - **Input**: The unique ID of the player.
            - **Output**: A JSON Object of the player's details.
        - Error Handling:
            - Utilizes try-catch statements to log errors and reset states
        - Examples

```
const playerDetails = getUserDetails(player.userId));
```

    - getRoster
        - Purpose: Retrieve current players, coaches, and owners of a PocketSports team
        - I/O
            - **Input**: No parameters.
            - **Output**: Fetches details for each player on the roster.
        - Error Handling:
            - Utilizes try-catch statements to log errors and reset states
        - Examples

```
getRoster();
```

    - getEvents
        - Purpose: Retrieve all events scheduled for a team given a date.
        - I/O
            - **Input**: A calendar date.
            - **Output**: A JSON object of the current events scheduled for the given date
        - Error Handling:
            - Utilizes try-catch statements to log errors and reset states
        - Examples

```
getEvents(new Date('2025-04-16T10:30:00));
```

    - handleDateChange
        - Purpose: Allow user to change date on the calendar to see what events are occurring

        - I/O
            - Input: A Javascript date object.
            - Output: None, this method sets a field for state use
        - Error Handling:
            - Utilizes try-catch statements to log errors and reset states
        - Examples

```
const handleDateChange = (date) => {
  setSelectedDate(date); // Update selected date
};
```

- addEvent
    - Purpose: Bring to the user the screen to specify details and create an event.
    - I/O
        - Input: None
        - Output: None, this method sets a field for state use
    - Error Handling:
        - Utilizes try-catch statements to log errors and reset states
    - Examples

```
const addEvent = () => {
  setShowPopup(true);
};
```

- handleEventClick
    - Purpose: Execute the creation of an event given the “create” button is pressed
    - I/O
        - Input: A JSON object containing information about an event
        - Output: None, this method sets a field for state use.
    - Error Handling:
        - Utilizes try-catch statements to log errors and reset states
    - Examples

```
handleEventClick(event)
```

- handleSubmit
    - Purpose: Handle the submission of an event so it is created
    - I/O
        - Input: an event e, containing the information submitted by the new event creator
        - Output: None, the events are retrieved and updated in the database
    - Error Handling:
        - Utilizes try-catch statements to log errors and reset states
    - Examples

```
<form onSubmit={handleSubmit}>
```

- removeEvent
    - Purpose: Contains logic to remove an event from the database.

        - I/O
            - Input: An integer index pointing to the location of the event
            - Output: Nothing returned, the DB and UI are updated accordingly
        - Error Handling:
            - Utilizes try-catch statements to log errors and reset states
        - Examples

```
<button className="remove-btn" onClick={() =>
removeEvent(index)}>❌</button>
```

- Drills()
    - getSport
        - Purpose: Get the selected sport for a specific PocketSports Team
        - I/O
            - Input: unique ID of a PocketSports Team
            - Output: The selected sport of that team (Lacrosse, Volleyball, Basketball)
        - Error Handling:
            - Utilizes Javascripts' .then and .catch for error handling
        - Examples

```
const sport = getSport(selectedTeam.teamId);
```

    - handleCreateDrill
        - Purpose: Execute the database operations to create a new drill.
        - I/O
            - Input: No parameters for this function
            - Output: No values are returned, a drill is added to the database
        - Error Handling:
            - Utilizes Javascripts .then and .catch operators to log errors and reset state
        - Examples

```
<button className="contactButton1" onClick={handleCreateDrill}>Create
Drill</button>
```

    - saveDrill
        - Purpose: Execute the database operations to save newly created drill
        - I/O
            - Input: No parameters are required
            - Output: No value is returned, a drill is added to the database
        - Error Handling:
            - Utilizes Javascripts .then and .catch operators to log errors and reset state
        - Examples

```
<Button onClick={saveDrill}> Save Drill</Button>
```

    - handleTagSelection
        - Purpose: When a user selects a tag for a drill, save that tag to the drill
        - I/O
            - Input: a tag, a string of characters that describe several drills' theme
            - Output: Returns the tags added to the drill in an array

        - Error Handling:
            - Utilizes try-catch statements to handle errors and reset state
        - Examples

```
<Button onClick={() => {const tagToAdd = dropdownTag || inputTag;
            handleTagSelection(tagToAdd);
            setDropdownTag(""); // Optionally clear the dropdown selection
            setInputTag("");    // Clear the manual input
            }}>Add Drill Tag
          </Button>
```

    - removeTag
        - Purpose: Deletes a tag from the database by sending a DELETE request to the backend
        - I/O
            - Input: tag, a string representing the name of the tag that is assigned to drills
            - Output: Nothing returned, filters the tag to remove from the tags array
        - Error Handling:
            - Utilizes try-catch statements for error handling and state management
        - Examples

```
<span key={index} className="tag" style={{ color: 'black', marginRight: '5px' }}>
               {tag} <button onClick={() => removeTag(tag)}>x</button>
             </span>
```

    - fetchDrills
        - Purpose: Get all the current drills located in the database for a specific team
        - I/O
            - Input: None
            - Output: No values returned, adds drill to database and drill bank
        - Error Handling:
            - .then and .catch are used for error handling and state management
        - Examples

```
if (selectedTeam) {
    fetchDrills();
  }
```

    - fetchDrillPdf
        - Purpose: Retrieve the base64-encoded string of the pdf when a user attempts to download it
        - I/O
            - Input: The name of the drill
            - Output: No values returned, the pdf is then downloaded to the users browser
        - Error Handling:
            - Utilizes .then and .catch for error handling and state management
        - Examples

```
<button onClick={() => fetchDrillPdf(drill.drillName)}>Download</button>
```

- handleDrillBank
  - Purpose: Handle the event in which the user wants to view the Drill Bank
  - I/O
    - Input: None
    - Output: The drill bank is displayed to the user
  - Error Handling:
    - Try-catch statements were used for error handling and state management
  - Examples

```
<button onClick={handleDrillBank}>Go to Drill Bank</button>
```

- deleteSelectedElement
  - Purpose: Delete a draggable element from the Drill creation canvas.
  - I/O
    - Input: None
    - Output: No values returned, the canvas deletes the selected (active) item
  - Error Handling:
    - Utilizes try-catch statements for error handling and state management
  - Examples

```
<Button onClick={deleteSelectedElement} className='deleteElement'>Delete Selected</Button>
```

- addDraggableElement
  - Purpose: Add either an X, O, or arrow to the drill creation canvas.
  - I/O
    - Input: symbol, either an X, O, or arrow, and color, a string determining the color
    - Output: No values returned, draggable element is added to the canvas
  - Error Handling:
    - Uses try-catch statements for error handling and state management
  - Examples

```
<Button onClick={() => addDraggableElement("O", "blue")}>Add O</Button>
```

- deleteDrill
  - Purpose: Perform the necessary database operations to remove a drill from the drill bank and database
  - I/O
    - Input: Unique ID of the drill to be deleted
    - Output: No values returned, the drill is removed from the drill bank and database
  - Error Handling:
    - Utilizes .then and .catch handlers for error handling and state management
  - Examples

```
<button onClick={() => deleteDrill(drill._id)}>Delete</button>
```

- handleDrillLayoutClick

        - Purpose: Sets the current modal to the image of the drill template the user is creating a drill with
        - I/O
            - Input: imageSrc, which is the source of the drill template being used
            - Output: No values returned, the modal appears to the user
        - Error Handling:
            - Uses try-catch statements for error handling and state management
        - Examples

```
<img src="/Volleyball.jpg" alt="Volleyball Field"
className="field-image" onClick={() =>
handleDrillLayoutClick("/Volleyball.jpg")} />
```

    - closeModal
        - Purpose: Remove the drill creation modal and return the user back to the drills page.
        - I/O
            - Input: None
            - Output: No values returned, the modal is removed from the user's view
        - Error Handling:
            - Utilizes try-catch statements for error handling and state management
        - Examples

```
<Button variant='secondary' onClick={closeModal}>Close</Button>
```

- GoalsPage()
    - fetchGoals
        - Purpose: Retrieve the goals for the current team and players from the database.
        - I/O
            - Input: None
            - Output: No values returned, useState() for the goals is set to the response from the backend
        - Error Handling:
            - Utilizes try-catch statements for error handling and state management
        - Examples

```
setNewGoal({ title: "", description: "",  targetNumber: 1 }); // Clear
input fields
      fetchGoals(); // Fetch updated goal list from the database
```

    - createGoal
        - Purpose: Given user parameters in the dropdowns, create a new goal and add it to the goals page as well as the database.
        - I/O
            - Input: No parameters given to function, useState() is used to get the values
            - Output: No values returned, the goal is added to the database and goal page
        - Error Handling:

        - Utilizes try-catch statements for error handling and state management
    - Examples

```
<button onClick={createGoal} style={{backgroundColor:
selectedTeam?.teamColors?.[0], color: 'white'}} >Save Goal</button>
```

- deleteGoal
    - Purpose: Remove a goal from the goals page and database.
    - I/O
        - Input: goalID, the unique identifier for the goal
        - Output: No values returned, the goal is removed from the database
    - Error Handling:
        - Utilizes try-catch statements for error handling and state management
    - Examples
        -
```
<button onClick={deleteGoal} style={{backgroundColor:
selectedTeam?.teamColors?.[0], color: 'white'}} >Remove
Goal</button>
```
- handleGoalClick
    - Purpose: Brings up the selected goal and sets the new parameters given to edit
    - I/O
        - Input: goal, an object containing several fields like id, title, description, progress
        - Output: No values returned, goal is edited according to fields of input object
    - Error Handling:
        - Utilizes try-catch statements for error handling and state management
    - Examples
        -
```
<button onClick={handleGoalClick(goal)}>Confirm Goal
Edit</button>
```
- handleSaveClick
    - Purpose: After a new goal is created, allow a user to save it to the database.
    - I/O
        - Input: goalID, a unique ID for a goal
        - Output: No values returned, the goal is saved into the database
    - Error Handling:
        - Uses try-catch statements for error handling and state management
    - Examples
        -
```
<button onClick={handleSaveClick(goal._id)}>Save Goal</button>
```
- updateGoalProgress
    - Purpose: Allow a user to edit the progress of a current goal.
    - I/O
        - Input: None, useState() is used to track the new progress value
        - Output: The current goal is changed via a PUT request to the backend,
    - Error Handling:

            - Uses try-catch statements for asynchronous operations
        - Examples

```
<button onClick={updateGoalProgress}>Save</button>
```

- LandingPage()
    - register
        - Purpose: Navigate the user to the registration page when the button is pressed
        - I/O
            - Input: None
            - Output: No values returned, user is redirected to another page
        - Error Handling:
            - None
        - Examples

```
<button onClick={register} className="featuresButtons">Register Now!/>
```

    - login
        - Purpose: Navigate the user to the login page when the button is pressed
        - I/O
            - Input: None
            - Output: No values returned, user is redirected to another page
        - Error Handling:
            - None
        - Examples

```
<button onClick={register} className="featuresButtons">Login/>
```

- Login()
    - landing
        - Purpose: Navigate the user to the landing page when the button is pressed
        - I/O
            - Input: None
            - Output: No values returned, user is redirected to another page
        - Error Handling:
            - None
        - Examples

```
<button onClick={landing} className="featuresButtons">Home Page/>
```

        -
    - teams
        - Purpose: Navigate the user to the teams page after they login with a correct email and password.
        - I/O
            - Input: None
            - Output: No values returned, user is redirected to another page
        - Error Handling:
            - None
        - Examples

```
<button onClick={teams} className="featuresButtons">Login/>
```

    - toggleShowPassword

        - Purpose: Reveal the password to the user typing it in upon selection of the button.
        - I/O
            - Input: None
            - Output: No values returned, the useState() value for showing the password is set to true
        - Error Handling:
            - None
        - Examples

```
const toggleShowPassword = () => setShowPassword(!showPassword);
```

    - handleSubmit
        - Purpose: Handle the event in which the user pressed “Login” with an email and password typed into the text boxes
        - I/O
            - Input: e, and event received from the user pressing the button
            - Output: No items returned, if the credentials are correct the user is logged in
        - Error Handling:
            - Utilizes Javascripts .then and .catch handlers for error handling and state management
        - Examples

```
<form onSubmit={handleSubmit} className="registration-form">
```

    - goToRegister
        - Purpose: Redirect user back to the registration page.
        - I/O
            - Input: None
            - Output: No values returned, user is redirected to another page
        - Error Handling:
            - None
        - Examples

```
<button type="button" className="register-button"
onClick={goToRegister}>
          Register
        </button>
```

    - handleClose
        - Purpose: Allow user to close modal that pops up when they successfully login.
        - I/O
            - Input: None
            - Output: No values returned, the login modal no longer appears to the user
        - Error Handling:
            - None
        - Examples

```
<Modal show={showModal} onHide={handleClose}>
```

- PracticePlans()

- fetchPracticePlans
  - Purpose: Retrieve all of the current practice plans for a team
  - I/O
    - Input: None
    - Output: No values returned, the response is saved through useState()
  - Error Handling:
    - Utilizes try-catch statements for error handling and state management
  - Examples

```
useEffect(() => {
      if (selectedTeam && selectedTeam._id) {
          fetchPracticePlans();
      }
   }, [selectedTeam]);
```

- fetchDrills
  - Purpose: Retrieve all of the current drills for a team to allow practice plans to be created with them.
  - I/O
    - Input: None
    - Output: No values returned. the state variable for the current drills is set
  - Error Handling:
    - Utilizes try-catch statements for error handling and state management
  - Examples

```
useEffect(() => {
      if (selectedTeam && selectedTeam._id) {
          fetchDrills();
      }
   }, [selectedTeam]);
```

- handleCreatePlan
  - Purpose: Allows a user to save a practice plan in the database.
  - I/O
    - Input: None, parameters to the POST request are retrieved through useState()
    - Output: No values returned, the practice plan is saved in the DB
  - Error Handling:
    - Utilizes try-catch statements for error handling and state management
  - Examples

```
<Button variant="primary" onClick={handleCreatePlan}>Create
Plan</Button>
```

- handleDeletePlan
  - Purpose: Allows a user to remove a practice plan from the database
  - I/O
    - Input: planId, the unique ID of the plan to be deleted

- Output: No values returned, but the practice plan is removed from the list of available plans
- Error Handling:
  - Utilizes try-catch statements for error handling and state management
- Examples

```
Button variant="danger" onClick={() =>
handleDeletePlan(plan._id)}>Delete</Button>
```

- handleExecutePlan
  - Allows a user to execute a practice plan, with all included drills and timers.
  - I/O
    - Input: drills, an array of javascript objects containing drill information
    - Output: No values returned, the current drills and modal are set to display the practice plan to the user
  - Error Handling:
    - Utilizes try-catch statements for error handling and state management
  - Examples

```
<Button variant="primary" onClick={() =>
handleExecutePlan(plan.drills)}>Execute</Button>
```

- RegistrationPage()
  - toggleShowPassword
    - Purpose: Allow a user to see or not see the password they are inputting for the first password input
    - I/O
      - Input: None
      - Output: No values returned, state is set for the show/hide password
    - Error Handling:
      - None
    - Examples

```
const toggleShowPassword = () => setShowPassword(!showPassword);
```

  - toggleShowPassword2
    - Purpose: Allow a user to see or not see the password they are inputting for the second password confirmation
    - I/O
      - Input: None
      - Output: No values returned, state is set for the show/hide password
    - Error Handling:
      - None
    - Examples

```
const toggleShowPassword2 = () => setShowPassword2(!showPassword);
```

  - landing
    - Purpose: Navigate the user to the landing page when the button is pressed
    - I/O
      - Input: None
      - Output: No values returned, user is redirected to another page

        - Error Handling:
            - None
        - Examples

```
<button onClick={landing} className="featuresButtons">Home Page/>
```

    - login
        - Purpose: Navigate the user to the login page when the button is pressed
        - I/O
            - Input: None
            - Output: No values returned, user is redirected to another page
        - Error Handling:
            - None
        - Examples

```
<button onClick={register} className="featuresButtons">Login/>
```

    - verifyEmail
        - Purpose: Verify a user's email address is correct through a one-time 6 digit code.
        - I/O
            - Input: email, a string containing the user's email address
            - Output: No values returned, backend will send email to user's address
        - Error Handling:
            - Uses Javascript's .then and .catch handlers to resolve errors
        - Examples

```
axios.post('http://localhost:3001/register', { fname, lname, email, password, password2 })
    .then(result => {
      console.log('Registration successful!');
      verifyEmail(email);
    })
```

    - handleSubmit
        - Purpose: Handle a user submission to register for an account
        - I/O
            - Input: e, an event received from the form used to take user input
            - Output: No values returned, user is registered in the backend and DB
        - Error Handling:
            - Utilizes .then and .catch handlers to resolve errors and failure states
        - Examples

```
<form onSubmit={handleSubmit} className="registration-form">
```

- Roster()
    - loadTeamSettings
        - Purpose: Retrieve the current information for settings for the team
        - I/O
            - Input: None
            - Output: No values returned, to show position, height, and weight are set with useState()
        - Error Handling:
            - Utilizes try-catch statements for error handling and state management
        - Examples

```
useEffect(() => {
   if (selectedTeam && selectedTeam._id) {
     loadTeamSettings();
   }
 }, [selectedTeam]);
```

- updateTeamSettings
  - Purpose: Allows a user to change the team settings via the UI.
  - I/O
    - Input: newSettings, and array of strings to show the position, height, and weight of the players to the user
    - Output: No values returned, the state variables are set
  - Error Handling:
    - Uses try-catch statements for error handling and state management
  - Examples

```
<input
          type="checkbox"
          checked={showPosition}
          onChange={(e) => {
            const newVal = e.target.checked;
            setShowPosition(newVal);
            updateTeamSettings({
              showPosition: newVal,
              showHeight,
              showWeight
            });
          }}
        />
```

- getUserDetails
  - Purpose: Retrieve the position, height, and weight of a certain player.
  - I/O
    - Input: userId, a unique identifier for a member of a team
    - Output: Returns the position, height, and weight of a certain player
  - Error Handling:
    - Uses try-catch statements for error handling
  - Examples

```
const detailPromises = rosterData.map((p) =>
getUserDetails(p.userId));
```

- getRoster
  - Purpose: Retrieve the full list of players for a team.
  - I/O
    - Input: None
    - Output: No values returned, a state variable containing players is set
  - Error Handling:
    - Try-catch statements are used for error handling

- Examples

```
useEffect(() => {
   const storedTeamString = localStorage.getItem('selectedTeam');
   if (storedTeamString) {
     const team = JSON.parse(storedTeamString);
     setSelectedTeam(team);
   }
   getRoster();
 }, []);
```

- removeUser
  - Purpose: Removes a user from the roster and team, as well as the backend.
  - I/O
    - Input: userId, a unique identifier for the user to be removed
    - Output: No values returned, roster is update to remove player
  - Error Handling:
    - Utilizes try-catch statements for error handling and state management
  - Examples

```
<Dropdown.Item onClick={() => removeUser(player.userId)}>
  Remove User
</Dropdown.Item>
```

- openChangeRoleModal
  - Purpose: Opens the modal where a user can change a role for a member of the team
  - I/O
    - Input: userId, a unique identifier, and currentRole, the current role of a member (coach, owner, player)
    - Output: No values returned, updates the new role for the player
  - Error Handling:
    - Utilizes try-catch statements for error handling and state management
  - Examples

```
<Dropdown.Item onClick={() => openChangeRoleModal(player.userId,
player.role)}>
  Change Role
</Dropdown.Item>
```

- handleChangeRoleSubmit
  - Purpose: Handle the event in which a user submits a new role they want to assign to another user
  - I/O
    - Input: None
    - Output: No values returned, update to the player's role is sent to the backend
  - Error Handling:

        - Utilizes try-catch statements for error handling and state management
    - Examples

```
<Button variant="primary" onClick={handleChangeRoleSubmit}>
  Save
</Button>
```

- openEditUserModal
    - Purpose: Opens the modal that allows a user to edit a certain player's information
    - I/O
        - Input: userId, a unique identifier for a user
        - Output: No values returned, the edit data is retrieved from the UI
    - Error Handling:
        - Utilizes try-catch statements for error handling and state management
    - Examples

```
<Dropdown.Item onClick={() => openEditUserModal(player.userId)}>
                    Edit User Info
                  </Dropdown.Item>
```

- handleEditUserSubmit
    - Purpose: Allows a user to save their changes to a player in the backend.
    - I/O
        - Input: None
        - Output: No values returned, the player information is updated in the backend through a PUT request
    - Error Handling:
        - Utilizes try-catch statements for error handling and state management
    - Examples

```
<Button variant="primary" onClick={handleEditUserSubmit}>
           Save Changes
         </Button>
```

- renderRosterCards
    - Purpose: Create and return a card for each of the players containing their information (position, height, and weight)
    - I/O
        - Input: None
        - Output: Returns several React Card components for each player on the roster
    - Error Handling:
        - Uses try-catch statements for error handling and state management
    - Examples

```
<div style={{ backgroundColor: 'whitesmoke' }}>
       {loading ? <p>Loading roster...</p> : renderRosterCards()}
     </div>
```

- HomePage()
    - getUserDetails

        - Purpose: Retrieve the details of a specific user (name and email)
        - I/O
            - Input: None
            - Output: No values returned, but the state variable userDetails is set to the values returned from the GET request to the backend
        - Error Handling:
            - .then and .catch are used for error handling
        - Examples

```
// On mount, load team info, user details
 useEffect(() => {
   if (storedTeamString) {
     setTeamName(storedTeamString);
     setSelectedTeam(JSON.parse(storedTeamString));
   }
   getUserDetails();
 }, []);
```

    - getEvents
        - Purpose: Retrieve all the upcoming events for a certain team.
        - I/O
            - Input: None, state variable is used
            - Output: No values returned, the state variable holding events is set
        - Error Handling:
            - Utilizes try-catch statements for error handling and state management
        - Examples

```
// On mount, load team info, events
 useEffect(() => {
   if (storedTeamString) {
     setTeamName(storedTeamString);
     setSelectedTeam(JSON.parse(storedTeamString));
   }
   getEvents();
 }, []);
```

    - fetchGoals
        - Purpose: Retrieve all of the current goals for a player or their team.
        - I/O
            - Input: None, state variables used
            - Output: No values returned, the goals are set in a state variable
        - Error Handling:
            - Utilizes try-catch statements for error handling and state management
        - Examples

```
useEffect(() => {
    if (selectedTeam && selectedTeam._id) {
        fetchGoals();
```

```
        getPracticePlans();
    }
}, [selectedTeam]);
```

- renderGoalCards
  - Purpose: Show the user each of their goals through a Card.
  - I/O
    - Input: None
    - Output: Returns Card components for each goal of a player
  - Error Handling:
    - Utilizes try-catch statements for error handling and state management
  - Examples

```
{localStorage.getItem('role') === 'Player' && (
        <>
          <strong className='homepage-headers'>Top Goals</strong>
          {renderGoalCards()}
        </>
      )}
```

- renderEventCards
  - Purpose: Show the user each of their upcoming events through a Card
  - I/O
    - Input: None
    - Output: Returns Card components for each upcoming event for a player
  - Error Handling:
    - Utilizes try-catch statements for error handling and state management
  - Examples

```
<strong className='homepage-headers'>Upcoming Events</strong>
      {renderEventCards()}
```

- renderPracticePlanCards
  - Purpose: If the user's role is Owner, display the current created practice plans as Cards
  - I/O
    - Input: None
    - Output: Returns Card components for each practice plan
  - Error Handling:
    - Utilizes try-catch statements for error handling and state management
  - Examples

```
<strong className='homepage-headers'>Your Practice Plans</strong>
          {renderPracticePlanCards()}
```

- renderRosterCards
  - Purpose: Render the 3 top performers of the roster in UI through Cards
  - I/O
    - Input: None
    - Output: Returns each of the members of the roster as a Card

        - Error Handling:
            - Utilizes try-catch statements for error handling and state management
        - Examples

```
<strong className='homepage-headers'>Your Top Performers</strong>
          {renderRosterCards()}
```

- TeamsPage()
    - landing
        - Purpose: Navigate the user to the landing page when the button is pressed
        - I/O
            - Input: None
            - Output: No values returned, user is redirected to another page
        - Error Handling:
            - None
        - Examples

```
<button onClick={landing} className="featuresButtons">Home Page/>
```

    - login
        - Purpose: Navigate the user to the login page when the button is pressed
        - I/O
            - Input: None
            - Output: No values returned, user is redirected to another page
        - Error Handling:
            - None
        - Examples

```
<button onClick={register} className="featuresButtons">Login/>
```

    - goToTeamPage
        - Purpose: Redirects the user back to the team homepage
        - I/O
            - Input: team, a JS object containing several fields for the team such as name, color, sport, etc.
            - Output: No values returned, user is redirected to team homepage
        - Error Handling:
            - Utilizes try-catch statements for error handling and state management
        - Examples

```
<li key={index} onClick={() => goToTeamPage(team)}
className="team-card">
```

    - handleProfileClick
        - Purpose: Brings the modal for the profile into the user's view.
        - I/O
            - Input: None
            - Output: No values returned, modal is displayed to user
        - Error Handling:
            - None
        - Examples

```
<button onClick={handleProfileClick}
className="contactButton">Profile</button>
```

- closeProfileModal
  - Purpose: Removes Profile modal from user's view when they click on close button.
  - I/O
    - Input: None
    - Output: No values returned, modal is removed from user's view
  - Error Handling:
    - None
  - Examples

```
<button onClick={closeProfileModal} className="close-btn">Save</button>
```

- handleFileChange
  - Purpose: Handles the event that a user uploads a profile picture
  - I/O
    - Input: e, and event received from the file input
    - Output: No values returned, updates the profile picture in the backend
  - Error Handling:
    - Utilizes try-catch statements for error handling and state management
  - Examples

```
<input
  type="file"
  onChange={handleFileChange}
  accept="image/*"
  ref={fileInputRef}
  className="file-input"
/>
```

- logout
  - Purpose: Handle a click on the "logout" button and end a user's session.
  - I/O
    - Input: None
    - Output: No values returned, removes session variables and navigates user back to login page
  - Error Handling:
    - None
  - Examples

```
<button onClick={logout} className="coachButton">Logout</button>
```

- handleColorClick
  - Purpose: Set the state variable for the team color once an option is selected
  - I/O
    - Input: color, a hex string of the RGB color
    - Output: No values returned, state variable for color is set
  - Error Handling:

        - Utilizes try-catch statements for error handling and state management
    - Examples

```
{colors.map((color) => (
<div
  key={color}
  onClick={() => handleColorClick(color)}
></div>
))}
```

- getTeams
    - Purpose: Retrieve all teams a user is a member of (Coach, player, or owner)
    - I/O
        - Input: None
        - Output: No values returned, state variable for current teams is set
    - Error Handling:
        - Utilizes try-catch statements for error handling and state management
    - Examples

```
if (storedUserId) {
    setUserId(storedUserId); // Set userId state from localStorage
    getTeams();
```

- getUserDetails
    - Purpose: Retrieve user details from MongoDB given a specific player ID.
    - I/O
        - **Input**: The unique ID of the player.
        - **Output**: A JSON Object of the player's details.
    - Error Handling:
        - Utilizes try-catch statements to log errors and reset states
    - Examples

```
const playerDetails = getUserDetails(player.userId));
```

- handleSubmit
    - Purpose: Handle the submission of details to create a new team.
    - I/O
        - Input: e, an event received from the submission of the team creation form
        - Output: No values returned, team is created in backend
    - Error Handling:
        - Utilizes try-catch statements for error handling and state management
    - Examples

```
<form onSubmit={handleSubmit}>
```

- handleJoinSubmit
    - Purpose: Handle the event in which a user requests to join a team via a 4-digit team code.
    - I/O
        - Input: e, an event received by the submission of the join form
        - Output: No values returned, the user is added to the team in the backend

        - Error Handling:
            - Utilizes try-catch statements for error handling and state management
        - Examples

```
{showJoinPopup && (
        <div className="popup-top">
        <div className="popup-content">
          <form onSubmit={handleJoinSubmit}>
```

    - handleCreateTeam
        - Purpose: Handle the event in which a user clicks on the “Create Team” Button
        - I/O
            - Input: None
            - Output: No values returned, the create team modal is displayed to the user
        - Error Handling:
            - None
        - Examples

```
<button className="topButtons" onClick={handleCreateTeam}>Create Team
+</button>
```

    - handleClosePopup
        - Purpose: Handle the event in which a user closes the Create Team modal
        - I/O
            - Input: None
            - Output: No values returned, the modal is no longer displayed to the user
        - Error Handling:
            - None
        - Examples

```
<button className="topButtons" type="button"
onClick={handleClosePopup}>Cancel</button>
```

    - handleJoinTeam
        - Purpose: Handle the event in which a user submits a 4-digit team code to join a team
        - I/O
            - Input: e, an event received by the submission of the join team code
            - Output: No values returned, the user is added to the team in the backend
        - Error Handling:
            - Output:
        - Examples

```
<button className="topButtons" type='submit'
onClick={handleJoinSubmit}>Submit Team Code</button>
```

    - handleCloseJoinPopup
        - Purpose: Remove the join team popup from the user’s view
        - I/O
            - Input: None

        - Output: No values returned, the join team modal is removed from the user's view
    - Error Handling:
        - None
    - Examples

```
<button className="topButtons" type="button"
onClick={handleCloseJoinPopup}>Cancel</button>
```

- Verify()
    - handleChange
        - Purpose: Adjust UI for entering a 6-digit verification code so every time a number is entered, the cursor moves to the next box
        - I/O
            - Input: e, an event received from inserting a new digit into the box, and index, which tracks the current index of the user's cursor
            - Output: No values returned, only adjust UI with email verification
        - Error Handling:
            - Uses regex and length checks to determine only single digits are entered
        - Examples

```
{code.map((digit, index) => (
          <input
            key={index}
            id={`code-${index}`}
            className="code-input"
            maxLength="1"
            value={digit}
            onChange={(e) => handleChange(e, index)}
          />
        ))}
```

    - landing
        - Purpose: Navigate the user to the landing page when the button is pressed
        - I/O
            - Input: None
            - Output: No values returned, user is redirected to another page
        - Error Handling:
            - None
        - Examples

```
<button onClick={landing} className="featuresButtons">Home Page/>
```

        -
    - login
        - Purpose: Navigate the user to the login page when the button is pressed
        - I/O
            - Input: None
            - Output: No values returned, user is redirected to another page

        - Error Handling:
            - None
        - Examples

```
<button onClick={register} className="featuresButtons">Login/>
```

    - handleSubmit
        - Purpose: Handles the event in which a user submits an email verification code.
        - I/O
            - Input: e, an event received from pressing the button to submit the verification code
            - Output: No values returned, the code is sent to the backend to be compared with the expected value
        - Error Handling:
            - .then and .catch used for error handling
        - Examples

```
<button onClick={handleSubmit} type="submit"
className="code-submit-button">Submit Verification Code</button>
```

    - handleClose
        - Purpose: Handle event where user successfully verifies their email and needs to be returned to the login page
        - I/O
            - Input: None
            - Output: No values returned, the verification modal is removed from the user's view and the user is redirected to the login page
        - Error Handling:
            - None
        - Examples

```
<Button variant="primary" onClick={handleClose}>
            OK
          </Button>
```

5.3 Utility and Helper Functions

- useEffect()
    - Used for side effects for functions, allowing code to execute when components or input changes
- useState()
    - Also used for side effects to store variables needed for computation during component changes
- useNavigate()
    - Used to supply navigation between the webpages
- axios
    - .get(url): sends a GET request to retrieve resources
    - .put(url): sends a PUT request to update resources

    - .post(url): sends a POST request to create resources
    - .delete(url): sends a DELETE request to destroy resources
- try-catch blocks
    - Used for error handling and state management
- canvas.add()
    - Adds draggable objects to drill canvas.
- canvas.getActiveObject()
    - Retrieves selected object on the drill canvas.
- localStorage.getItem/setItem()
    - Stores user session or selected team.
- Array.map/filter/reduce()
    - Common array utilities used in rendering or calculations.

6. Development and Debugging Practices

6.1 How to Add New Features

If new features are wanted when a user selects a team, create a new .jsx file to start and copy over the header and footer involved with the other pages to keep a uniform layout. Depending on the feature functionality, methods to add information about a user, edit, and delete are already outlined in current features. The new feature can use the same logic as the current ones, but the information must be changed. For the backend of the feature, GET, POST, and PUT are used in current features, so a new database would need to be created for the new information, but can edit existing commands to get the desired functionality.

6.2 Maintenance Guidelines

Code should be commented on, describing what a method is accomplishing, and have clear names to easily describe to a developer what the variable is used for. When modifying features, ensure functionality still works correctly after changes are made. Duplicated code should be removed, and efficiency should be improved. Documentation should include any changes made to the code, and errors should be identified and solved when they are found.

6.3 Debugging Procedures

Developers should use the console to print out any possible errors trying to be debugged. The error must be traced to the method that is incorrectly causing the error. For backend errors, use the server logs to find the errors. Once the error is found, fix it and re-run multiple tests to ensure it is completely solved. Fixes should be committed and pushed to review by teammates.

6.4 Testing

All new or edited features need to be tested to ensure correct functionality. Unit tests can be written for each new feature component, and integration tests can be made as well. Before pushing changes, run tests locally to ensure that errors will not be pushed to production.

7. Build, Deployment, and CI/CD Processes

7.1 Build Process

- Create Dockerfile for both frontend and backend
    - Specify the proper active directories and necessary dependencies
- Create a docker-compose.yml containing necessary build information
- Build containers using a command like `docker build`

7.2 Deployment Instructions

- Ensure you have an AWS account with proper permissions set up
- Tag and push the docker images to AWS ECR (Elastic Container Registry)
- Go to App Runner and choose the image from ECR
- Configure settings for the service, such as environment variables, port mappings, and auto-scaling options
- Deploy the application with AWS App Runner

7.3 Continuous Integration/Continuous Deployment (CI/CD)

- First ensure the workflow is defined in the github repository by adding a YAML file
- Make necessary changes to source code
- Push to repository (`git add`, `git commit`, `git push`)
- AWS App Runner monitors the Github repo and automatically rebuild and deploys the changes

8. Appendices and Additional Resources

8.1 Glossary of Terms

API (Application Programming Interface): A set of defined endpoints and protocols used for communication between the frontend and backend systems.

App Runner: An AWS service used to deploy containerized applications directly from a source code repository or container registry.

Axios: A promise-based HTTP client used in the frontend to communicate with backend APIs.

Backend: The server-side portion of the application responsible for handling database interactions, authentication, and serving API endpoints.

Button: An element of a webpage where a user can click to create some side effect.

CI/CD (Continuous Integration / Continuous Deployment): Development practice that allows automated testing and deployment of code upon changes being pushed to the codebase.

Canvas: A visual HTML element used to render and manipulate 2D graphics, leveraged for creating interactive drills using libraries like Fabric.js.

Component: A modular part of a React application responsible for rendering part of the UI and managing associated logic.

Component-Based Architecture: A design pattern in React where the UI is broken into isolated, reusable pieces.

CRUD (Create, Read, Update, Delete): The four basic operations for managing data in a web application.

Database: An organized collection of data stored digitally.

Database Schema: The structure of a database described in a formal language, including tables, fields, and relationships.

Drill Bank: A repository within the app where custom drills are saved, managed, and retrieved for future use.

Drill Stats: Statistical metrics associated with specific drills to track performance and evaluation.

Drill Tags: Keywords or labels used to categorize and filter drills in the drill bank.

Email Verification: A security process in which a 6-digit code is sent to the user's email to confirm identity during registration.

Event: A scheduled item such as a game, practice, or meeting, stored in a team’s calendar and optionally containing feedback and stats.

Fabric.js: A JavaScript library used to render and manage the drill creation canvas, including drag-and-drop elements.

Field View: Background images used in the drill editor to visually represent sport-specific layouts (e.g., volleyball, basketball courts).

Frontend: The client-side portion of the application that includes the visual interface and handles user interactions.

Hook: A special function in React (e.g., `useState`, `useEffect`) that allows a user to "hook into" React features from function components.

HTML2Canvas: A JavaScript library that captures visual DOM elements as images, used in exporting drills to PDF.

HTTP (HyperText Transfer Protocol): The foundational protocol for data communication on the web.

Invite Code / Join Code: A 4-digit code that allows users to join a team after registration.

jsPDF: A library for generating PDF documents on the client side, used for exporting practice plans.

jsx: A variation of JavaScript that allows for HTML templating, giving users the ability to directly inject HTML code into their function-based components.

Landing Page: The first page a user sees upon visiting the site, designed to introduce features and offer login or registration options.

LocalStorage: A browser feature used to store data persistently on the client side, such as session tokens or selected teams.

Modal: A pop-up window overlay component often used for user prompts like profile editing or creating new practice plans.

MongoDB: A NoSQL database used to store all user, team, goal, and drill information in JSON-like documents.

Nodemailer: A Node.js library used for sending emails, such as invitations and verification codes.

PDF (Portable Document Format): A file format used for exporting and sharing practice plans and drills in a readable and printable layout.

Practice Plan: A list of drills and training activities that coaches can create, view, execute, and export.

Props: Short for "properties," these are inputs passed to React components to configure behavior or display.

React: A JavaScript library used for building the user interface with a component-based architecture.

React Router: A library used to manage navigation and page views in a React application.

REST (Representational State Transfer): An architectural style for designing networked APIs, usually accessed over HTTP.

Role-Based Access: A UI pattern where features shown to the user depend on their role (e.g., player, coach, owner).

Roster: A list of players and associated metadata (e.g., height, weight, position) for a team.

Session: The time span during which a user is logged into the app, typically tracked via cookies or local storage.

SMTP (Simple Mail Transfer Protocol): A protocol used for sending email messages from one server to another.

State Management: In React, the practice of managing and updating dynamic data within a component using hooks like `useState`.

Swiper: A JavaScript library used for implementing a sliding carousel UI for displaying drill previews or other content.

Team Code: A unique identifier used for users to join a specific team within PocketSports.

UI (User Interface): The visual part of the application that users interact with.

useEffect / useState: React hooks used to handle component lifecycle events and local state within a component.

Validation: The process of checking data (e.g., registration input or drill creation fields) to ensure it is complete and correct before submission.

Verification Code: A 6-digit numeric code sent via email to confirm a user's identity during account creation.

Vite: A frontend build tool used to bundle and serve the React application quickly during development.

8.2 External Resources

- Vite - https://vite.dev/
    - Used for creation of frontend web pages. Allowed for HTML templating in Javascript
- MongoDB - https://www.mongodb.com/
    - Used for storing all user data
- React - https://react.dev/
    - Used for app structure, routing, etc.

- Nodemailer - https://www.nodemailer.com/
    - Used for email communication for verification and invitation to a team
- Swiper - https://swiperjs.com/
    - Used to create sliding-window card style components
- jsPDF - https://parall.ax/products/jspdf
    - Used for dealing with PDFs for drills
- Html2canvas - https://html2canvas.hertzen.com/
    - Used for editing items and creating PDFs in the browser window
- Fabric.js - https://fabricjs.com/
    - Used for creating drills and adding draggable elements
- FontAwesome - https://docs.fontawesome.com/v5/web/use-with/react
    - Used for fonts for the UI
- Axios - https://axios-http.com/docs/intro
    - Used for all HTTP communication