SE/COM S 3190 – Construction of User Interfaces

Final Project Technical Documentation

Fall 2025

DraftZone

Team Members

Neel Rajan
neel9033@iastate.edu

Aadi Bhatia
aadi2005@iastate.edu

Iowa State University
Department of Computer Science

Contents

1 Introduction

DraftZone is a web-based fantasy football application designed to help users better understand fantasy football concepts and make informed decisions during the drafting process. The project focuses on simplifying complex fantasy football data and presenting it through an intuitive and visually engaging user interface that supports learning, exploration, and decision-making for new and casual players.

1.1 Project Overview

Fantasy football can be overwhelming for new players due to the large amount of player data, statistics, projections, and scoring rules involved. Beginners often struggle to understand how to evaluate players, compare positions, and make effective draft decisions within a limited time frame.

DraftZone addresses this problem by providing a centralized platform where users can explore NFL players, view fantasy-relevant statistics and projections, and simulate drafting a team in a guided environment. The application presents key information such as rankings, projected points, and player performance trends in a clear and accessible way, reducing the learning curve for new users.

DraftZone is implemented as a full-stack web application that combines interactive user interfaces with real-time data presentation to support fantasy football education and draft preparation.

1.2 Target Users and Use Cases

The primary target users of DraftZone are individuals who are new to fantasy football or have limited experience participating in fantasy leagues. These users benefit from simplified explanations, structured player comparisons, and an interactive drafting interface that helps them understand how fantasy teams are built.

Typical use cases include users browsing player rankings before a draft, searching for specific players or teams, and participating in a simulated draft to practice roster construction. The application allows users to filter players by position, view projections, and make draft selections in real time.

Secondary users include more experienced fantasy football players who want a clean interface for quick player lookup, comparison, or mock drafting. Instructors or peers may also use the application as a demonstration tool to explain fantasy football concepts.

1.3 Core Features

DraftZone includes several core features that support its educational and drafting goals. The application provides a searchable player database where users can filter NFL players by position, team, and ranking. Each player card displays fantasy-relevant statistics, projected points, and recent performance indicators to aid decision-making.

Another key feature is the interactive draft interface, which allows users to build a fantasy roster by selecting players and assigning them to appropriate positions. The system visually tracks drafted players and remaining roster slots, helping users understand positional requirements and draft strategy.

Additionally, DraftZone includes real-time or simulated betting odds and projections to give users exposure to how player performance expectations are calculated. Together, these features simplify complex fantasy football data and directly address the challenges new players face when drafting teams.

1.4 Originality vs Inspiration

The idea for DraftZone is inspired by existing fantasy sports platforms such as ESPN Fantasy, Yahoo Fantasy, and Sleeper. These platforms influenced the overall concept of player rankings, draft interfaces, and statistical comparisons.

However, DraftZone differs by placing a stronger emphasis on clarity, visual simplicity, and beginner-focused design. Rather than targeting experienced fantasy players exclusively, the application prioritizes ease of use, reduced information overload, and an educational drafting experience. The project reimagines fantasy football tools with a learning-first approach while maintaining realistic fantasy mechanics.

2 System Overview and Project Description

This section describes the overall structure of the DraftZone system and explains how the application is organized from both a technical and user perspective. The system is designed as a modular full-stack web application, separating frontend presentation logic from backend data handling to improve maintainability, scalability, and clarity.

2.1 Major Functional Modules

DraftZone is divided into several major functional modules, each responsible for a specific area of the application's behavior and user experience.

The **Home Page module** serves as the landing page for users and introduces the application's purpose, features, and navigation options. It provides entry points to other major areas of the system, such as player search, drafting, and odds.

The **Player Search module** allows users to browse and search NFL players using filters such as position and team. This module displays player statistics and projections retrieved from an external player statistics API, enabling users to compare players and make informed decisions.

The **Drafting module** provides an interactive fantasy draft experience. Users can choose between player-versus-player (PvP) drafts or player-versus-CPU drafts. This module manages draft turns, roster construction, and position requirements while updating the interface dynamically as players are selected.

The **Odds module** displays betting odds and related information using data retrieved from an external odds API. This module helps users understand performance expectations and game projections that can influence fantasy decisions.

The **Authentication module** supports user login functionality, distinguishing between regular users and administrative users. This module controls access to protected features and ensures that user-specific actions are properly authorized.

From a technical perspective, DraftZone uses a **React frontend** with **React Router** to manage navigation between modules. Styling is implemented using **Tailwind CSS and PostCSS** for consistent and responsive design. The backend is built using **Node.js and Express**, exposing RESTful API endpoints that the frontend communicates with using HTTP requests via fetch or axios

2.2 UI and Navigation Flow

A user begins by landing on the DraftZone home page, where they are introduced to the application and can navigate to core features using the main navigation bar. Users may browse player data and view odds without logging in, allowing casual exploration of the system.

When a user selects the player search page, they can search for NFL players, filter by position, and view relevant statistics and projections. Selecting the drafting page allows the user to choose between a player-versus-player draft or a player-versus-CPU draft. During the draft, users select players in turn while the interface updates roster slots and available players in real time.

If a user chooses to log in, they are redirected to the login page, where authentication determines whether the user is a standard user or an administrator. After successful login, the user is returned to the main application with access to features appropriate to their role.

2.3 CRUD Entities

The **User** entity stores account information such as username, email, role (user or admin), and authentication credentials. CRUD operations allow users to create accounts and delete accounts when necessary.

The **Draft** entity represents an active fantasy draft session and includes data such as draft type (player-versus-player or player-versus-CPU), draft order, and current pick. Draft entities are created when a new draft begins, updated as selections are made, and removed when drafts are completed or reset.

The **Team** entity stores a fantasy roster, including selected players and assigned positions. Teams are created during drafts and updated as players are selected.

The **Player** entity is read-only and contains player names, positions, teams, and statistical data retrieved from an external API. Read operations support player search, comparison, and drafting functionality.

3 File and Folder Architecture

This section documents the physical layout of your project. Explain the reasoning behind your folder structure and how it supports good development practices.

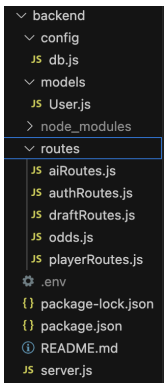
3.1 Folder Structure Overview

The project uses separate backend and frontend directories to separate server-side logic from the user interface. The backend handles API routes, authentication, and database access, while the frontend contains the React components, pages, and styling. This structure was provided to us and clearly defines responsibilities within the application.

This setup improves maintainability and workflow by allowing the backend and frontend to be developed, tested, and debugged independently. An alternative would be a single combined project where the frontend is served from the backend, but the given structure was chosen because it keeps concerns separated and reflects common real-world full-stack development practices.

3.2 Backend Folder Structure

3.2.1 Backend Folder Tree:



3.2.2 Backend Folder Description:

The config folder houses all configuration files that define how the application connects to external services and databases. The primary file, db.js, contains database connection configuration using Mongoose ODM (Object Document Mapper) to connect to MongoDB. The models folder defines database schemas and data structures using Mongoose models, representing the data layer in the MVC (Model-View-Controller) architectural pattern. The User.js file, for example, defines the User schema with fields like email, password, and role (admin or user), along with validations, middleware hooks, and authentication methods.

The routes folder contains all API route definitions that map incoming HTTP requests to their corresponding controller functions. This folder is organized by feature domain, with authRoutes.js

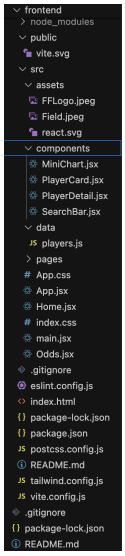
managing authentication endpoints like login and signup, playerRoutes.js handling player search and statistics, draftRoutes.js managing draft room functionality, oddsRoutes.js fetching live NFL betting odds, and aiRoutes.js providing AI-powered features. This routing structure supports good API design through several key principles.

The .env file stores environment variables containing sensitive configuration data that isnt committed to version control. This includes the MongoDB connection string (MONGODB_URI) and the Gemini Api key. The package.json and package-lock.json files manage Node.js dependencies and project metadata.

The server.js file functions as the main entry point and orchestration center for the entire backend application. This file is responsible for initializing the Express server, establishing the database connection to MongoDB through Mongoose, registering middleware such as CORS for cross-origin requests and express.json() for handling JSON payloads, mounting route handlers at appropriate paths (like /api/auth for authentication and /api/players for player data), and finally starting the server listening on the specified port.

3.3 Frontend Folder Structure

3.3.1 Frontend Folder Tree:



3.3.2 Frontend Folder Description:

The frontend is organized using a component-based React structure to keep the codebase simple, readable, and easy to maintain. The src folder contains all application logic and UI code. Reusable UI elements such as PlayerCard, PlayerDetail, MiniChart, and SearchBar are placed in the components folder so they can be shared across multiple pages. Page-level views like Home.jsx and Odds.jsx are kept separate, making it clear which files represent full screens versus smaller UI pieces.

Static assets such as images and logos are stored in the assets folder, while mock data like players.js is centralized in the data folder for easy future replacement with API data. Global styles are managed through App.css and index.css, and the application entry points (App.jsx and main.jsx) handle layout and app initialization. This structure improves readability, encourages reuse, and makes the frontend easier to scale as new features are added.

4 Code Explanation and Logic Flow

This section explains how the application behaves internally, how data flows between components, and how requests are processed from the UI to the database.

4.1 Frontend–Backend Interaction

How the frontend triggers API calls (e.g., using fetch or axios):

The Odds component uses the browser's native **fetch API** to make HTTP requests to the backend. API calls are triggered automatically when the component mounts using the useEffect hook

```
63      useEffect(() => {
64        fetchOdds();
65      }, []);
```

The empty dependency array [] ensures this effect runs only once when the component first renders, initiating the odds data fetch immediately upon page load.

How data is sent to the backend (JSON bodies, query parameters, route parameters):

Data is sent via a GET request with the endpoint URL

```
52      try {
53        const res = await fetch("http://localhost:8080/api/odds");
```

The backend route (/api/odds) receives this request, fetches data from The Odds API (a third-party sports betting API), and returns the processed data to the frontend.

How responses are handled in the React components (state updates, conditional rendering):

The component uses React's useState hook to manage two pieces of state:

```
export default function Odds() {
  const [games, setGames] = useState([]);
  const [loading, setLoading] = useState(true);
```

Response Processing:

```
49    const fetchOdds = async () => {
50      setLoading(true);
51
52      try {
53        const res = await fetch("http://localhost:8080/api/odds");
54        const data = await res.json();
55        setGames(data);
56      } catch (err) {
57        console.error("Error fetching odds:", err);
58      }
59
60      setLoading(false);
61    };
```

Error-handling strategies (try/catch blocks, UI messages, fallback states):

All API calls are wrapped in try/catch blocks to handle network failures, server errors, or malformed responses

4.2 React Component Example

Props Received and Data Flow

The Odds component serves as a dedicated page within the DraftZone application that displays live NFL betting odds fetched from a third-party API. Its primary purpose is to present moneyline, point spread, and over/under totals for upcoming NFL games in an interactive, visually appealing interface that helps users make informed betting decisions.

Hooks Used

This component receives no props as it's a top-level route component. It manages its own data fetching and state internally, with data flowing downward from internal state to child elements. The component uses two useState hooks to manage state. The games state stores the array of NFL game data with betting odds, while the loading state tracks whether data is currently being fetched.

State Management Patterns

The fetchOdds function coordinates multiple state updates around the asynchronous fetch operation. It sets loading to true before fetching, updates games with the received data, then sets loading to false. This ensures the UI accurately reflects the current data fetching status

Backend Route Interaction

The component makes a GET request to http://localhost:8080/api/odds. The backend route handler receives this request, fetches data from The Odds API, and returns formatted odds data as JSON.

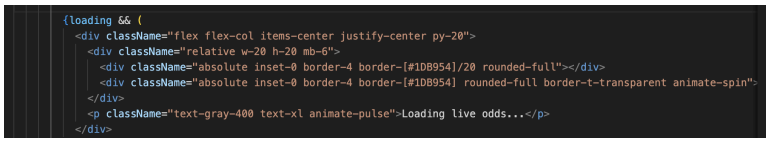
The response is a JSON array of game objects. Each game includes team names, game time, and a bookmakers array with betting markets for moneyline, spreads, and totals.

The component uses optional chaining and the find method to safely extract betting markets from the nested data structure without causing errors if data is missing.

```
141                {games.map((game, index) => {
142                  const markets = game.bookmakers?.[0]?.markets || [];
143                  const moneyline = markets.find(m => m.key === "h2h");
144                  const spreads = markets.find(m => m.key === "spreads");
145                  const totals = markets.find(m => m.key === "totals");
```

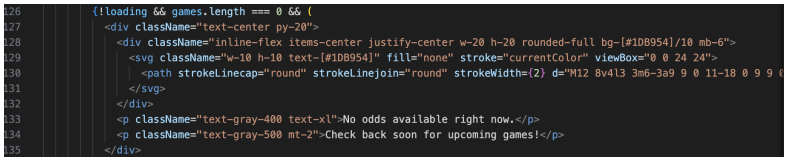
Conditional UI Logic

The component implements three UI states. The loading state displays an animated spinner while data is being fetched, providing visual feedback to the user.



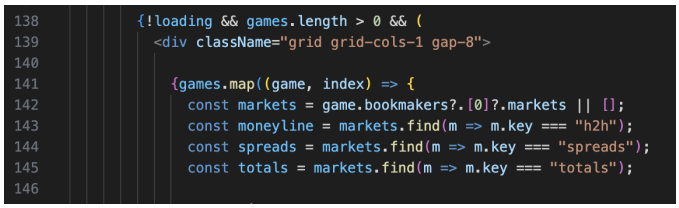
```
{loading && (
  <div className="flex flex-col items-center justify-center py-20">
    <div className="relative w-20 h-20 mb-6">
      <div className="absolute inset-0 border-4 border-[#1DB954]/20 rounded-full"></div>
      <div className="absolute inset-0 border-4 border-[#1DB954] rounded-full border-t-transparent animate-spin">
    </div>
    <p className="text-gray-400 text-xl animate-pulse">Loading live odds...</p>
  </div>
```

The empty state appears when loading completes but no games are available. This displays a friendly message with a clock icon suggesting the user check back later.



```
{!loading && games.length === 0 && (
  <div className="text-center py-20">
    <div className="inline-flex items-center justify-center w-20 h-20 rounded-full bg-[#1DB954]/10 mb-6">
      <svg className="w-10 h-10 text-[#1DB954]" fill="none" stroke="currentColor" viewBox="0 0 24 24">
        <path strokeLinecap="round" strokeLinejoin="round" strokeWidth={2} d="M12 8v4l3 3m6-3a9 9 0 11-18 0 9 9 0
      </svg>
    </div>
    <p className="text-gray-400 text-xl">No odds available right now.</p>
    <p className="text-gray-500 mt-2">Check back soon for upcoming games!</p>
  </div>
```

The success state maps over the games array to create individual game cards displaying team matchups and betting markets.



```
{!loading && games.length > 0 && (
  <div className="grid grid-cols-1 gap-8">

    {games.map((game, index) => {
      const markets = game.bookmakers?.[0]?.markets || [];
      const moneyline = markets.find(m => m.key === "h2h");
      const spreads = markets.find(m => m.key === "spreads");
      const totals = markets.find(m => m.key === "totals");
```

Each betting market is conditionally rendered to handle cases where certain markets might not be available for a game.

```
{/* BETTING MARKETS */}
<div className="grid md:grid-cols-3 gap-6">

  {/* MONEYLINE */}
  {moneyline && (
    <div className="bg-gradient-to-br from-[#0f1f33] to-[#16233b] p-6 rounded-2xl border border-g
      <div className="flex items-center gap-2 mb-4">
        <div className="w-2 h-2 rounded-full bg-[#1DB954]"></div>
        <p className="text-gray-300 font-bold text-lg">Moneyline</p>
      </div>
      {moneyline.outcomes.map((team, idx) => (
        <div
          key={idx}
          className="bg-[#1DB954]/10 hover:bg-[#1DB954]/20 p-4 rounded-xl mb-3 transition-all du
        >
          <div className="flex justify-between items-center mb-2">
            <span className="font-semibold text-white">{team.name}</span>
            <span className="text-2xl font-bold text-[#1DB954]">{team.price > 0 ? '+' : ''}{team.
          </div>
          <div className="flex items-center gap-2 text-sm">
            <span className="text-gray-400">Win Probability:</span>
            <span className="font-semibold text-[#1DB954]">{calcWinProb(team.price)}</span>
          </div>
        </div>
      ))
    </div>
  )}

  {/* SPREADS */}
  {spreads && (
    <div className="bg-gradient-to-br from-[#0f1f33] to-[#16233b] p-6 rounded-2xl border border-g
      <div className="flex items-center gap-2 mb-4">
        <div className="w-2 h-2 rounded-full bg-[#1DB954]"></div>
```

Component FeaturesDynamic

Team Logo Display

A constant object maps NFL team names to logo URLs hosted on ESPN's CDN. The component displays these logos alongside team names for visual identification.

```
const TEAM_LOGOS = {
  "Arizona Cardinals": "https://a.espncdn.com/i/teamlogos/nfl/500/ari.png",
  "Atlanta Falcons": "https://a.espncdn.com/i/teamlogos/nfl/500/atl.png",
  "Baltimore Ravens": "https://a.espncdn.com/i/teamlogos/nfl/500/bal.png",
  "Buffalo Bills": "https://a.espncdn.com/i/teamlogos/nfl/500/buf.png",
  "Carolina Panthers": "https://a.espncdn.com/i/teamlogos/nfl/500/car.png",
  "Chicago Bears": "https://a.espncdn.com/i/teamlogos/nfl/500/chi.png",
  "Cincinnati Bengals": "https://a.espncdn.com/i/teamlogos/nfl/500/cin.png",
  "Cleveland Browns": "https://a.espncdn.com/i/teamlogos/nfl/500/cle.png",
```

Win Probability Calculator

The calcWinProb function converts American odds to implied win probability percentages, making odds easier to interpret.

```
  const calcWinProb = (decimal) => {
    if (!decimal || decimal <= 0) return "–";
    return (100 / decimal).toFixed(1) + "%";
  };
```

Responsive Design and User Experience

The component uses Tailwind CSS for responsive design that adapts across device sizes. The grid layout switches from a single column on mobile to three columns on a desktop. Hover effects and smooth transitions enhance interactivity and provide visual feedback.

4.3 API Route Examples

GET Route: Fetch NFL Betting Odds

The GET /api/odds endpoint retrieves live NFL betting odds from The Odds API and returns them to the frontend. This endpoint serves as a proxy between the frontend and the external sports betting API, handling authentication and data formatting. The route fetches current moneyline, point spread, and over/under totals for all upcoming NFL games.

This endpoint accepts a GET request with no required parameters or request body. The API key and query parameters are handled internally by the backend.

Validation Steps

The backend validates the response from The Odds API before sending it to the frontend. If the external API returns an error or invalid data, the route catches the error and returns an appropriate error response. The validation ensures the data structure matches what the frontend expects.

Middleware Used

This route uses Express's built-in middleware. The express.json() middleware parses JSON request bodies, and cors() middleware enables cross-origin requests from the frontend running on a different port. No authentication middleware is used for this public endpoint.

On success, the endpoint returns a 200 status code with a JSON array of game objects. Each game contains team information, game times, and nested bookmaker data with betting markets. On failure, the endpoint returns a 500 status code with an error message explaining what went wrong.

Error Handling

The route wraps the external API call in a try-catch block to handle various error scenarios. Network errors, API rate limit errors, and invalid responses are caught and logged. The catch block returns a 500 status code with a JSON object containing an error message and details about what went wrong.

POST Route: User Login

This endpoint is to authenticate an existing user by validating their email and password. If the credentials are valid, the API returns basic user information (email, role, and name) that can be used by the frontend to establish a logged-in session.

The login endpoint expects a JSON request body containing an email address and a password. No URL parameters or query parameters are used

Validation Steps

Once the request is received, the server extracts the email and password fields from the request body. The email is converted to lowercase to ensure consistent matching regardless of user input casing. The route then checks whether a user exists whose email and password match the provided credentials. If no matching user is found, authentication fails.

Middleware Used

This route relies on Express's built-in JSON parsing middleware to interpret the incoming request body. No authentication middleware is applied because the purpose of this endpoint is to authenticate the user itself.

Database Operations Triggered

The login route performs a read-only operation against a mock in-memory database represented by the USERS array. The route does not write, update, or delete any records. The .find() method is used to locate a user record that matches the provided credentials.

Error Handling

Errors are handled using conditional logic rather than exceptions. When invalid credentials are detected, the route logs a failure message to the server console and returns a structured JSON response with a 401 status code. This allows the frontend to display a clear authentication error message without exposing internal implementation details.

PUT Route: Draft Page

This endpoint updates the status of an existing draft to mark it as completed. This endpoint finalizes a draft session by recording the winner and final scores, changing the draft status from active to completed, and storing the completion timestamp. This allows the application to maintain a historical record of completed drafts while preventing further modifications to finished games.

This endpoint accepts a PUT request with the draft ID as a URL parameter and completion details in the request body. The ID parameter must be a valid MongoDB ObjectId format.

Validation Steps Performed

The route performs validation on the draft ID before attempting to update the database. It uses MongoDB's ObjectId.isValid() method to verify the ID parameter is in the correct format. If the ID is invalid, the route immediately returns a 400 Bad Request error without querying the database.

Middleware Used

This route uses Express's standard middleware stack. The express.json() middleware parses the JSON request body and makes it available through req.body. The cors() middleware enables cross-origin requests from the frontend.

Database Operations Triggered

The route performs a single database update operation using MongoDB's updateOne method. It locates the draft document by its ObjectId and updates multiple fields atomically. The $set operator updates the status field to COMPLETED, records the current timestamp in completedAt, and stores the results object containing the winner and both scores. This operation uses MongoDB's atomic update guarantees, ensuring all fields are updated together or not at all. If the draft ID doesn't exist in the database, the updateOne operation completes successfully but modifies zero documents

Error Handling

The route wraps all operations in a try-catch block to handle unexpected errors. The catch block logs the full error details to the server console for debugging while returning a sanitized error message to the client. The error handling covers database connection failures, invalid update operations, and any other unexpected runtime errors.

DELETE Route: Draft Page

The DELETE endpoint permanently removes a specific draft from the database. This endpoint allows users to clean up unwanted or test drafts, freeing up database space and removing drafts

from the user's draft history. This endpoint accepts a DELETE request with the draft ID as a URL parameter.

Validation Steps Performed

The route performs two levels of validation before deleting data. First, it validates the format of the draft ID using MongoDB's ObjectId.isValid() method. After the delete operation completes, the route checks whether any documents were actually deleted by examining the deletedCount property of the result. If deletedCount is zero, it means no draft with that ID existed in the database, and the route returns a 404 Not Found error.

Middleware Used

This route uses Express's standard middleware configured in server.js. The cors() middleware enables cross-origin requests from the frontend running on a different port. Since DELETE requests don't typically include a body, the express.json() middleware is not strictly necessary for this endpoint but remains active as part of the global middleware stack.

Database Operations Triggered

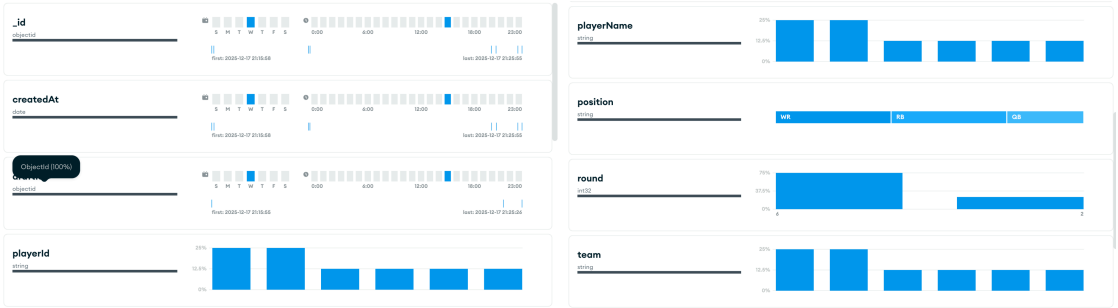
The route performs a single database delete operation using MongoDB's deleteOne method. This method locates the draft document by its ObjectId and removes it from the drafts collection.The deleteOne method returns a result object containing information about the operation, including how many documents were deleted.The route captures this result to determine whether the draft actually existed. Using deleteOne instead of deleteMany ensures only a single draft is removed even if multiple drafts somehow shared the same ID, though MongoDB's unique ObjectId constraint makes this scenario impossible in practice.

Error Handling

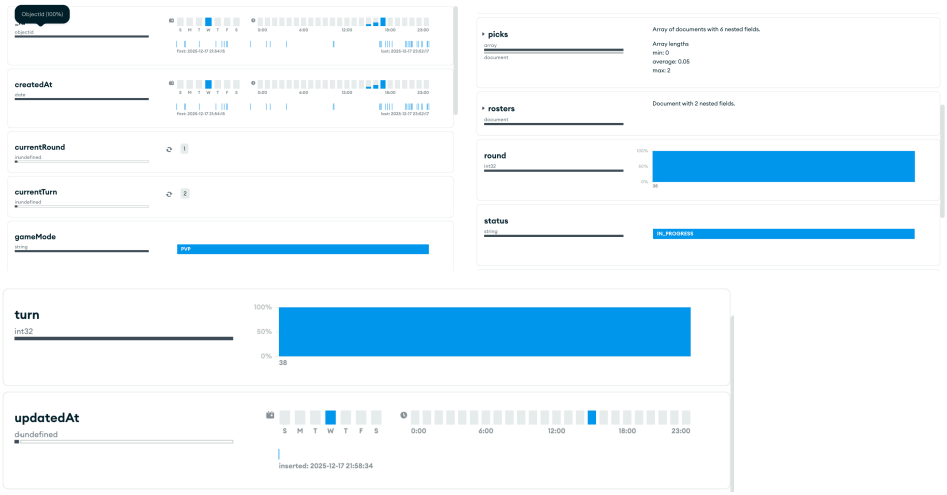
The route implements comprehensive error handling with a try-catch block that captures all unexpected errors.The error handling strategy provides three distinct response types based on the error condition. Invalid ID formats receive a 400 status, non-existent drafts receive a 404 status, and unexpected errors receive a 500 status.

4.4 Database Interaction

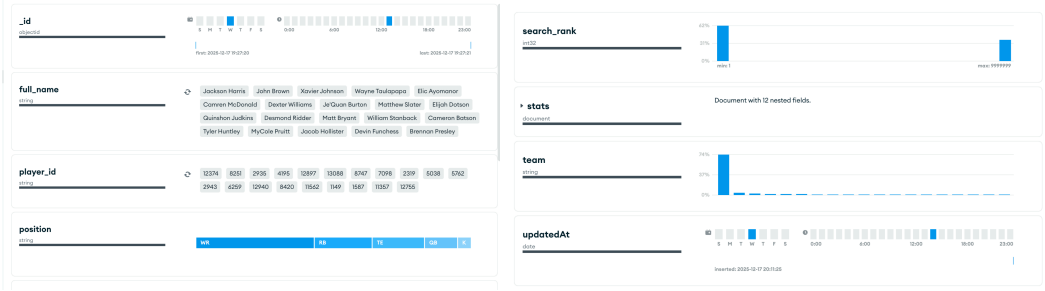
- draftPicks



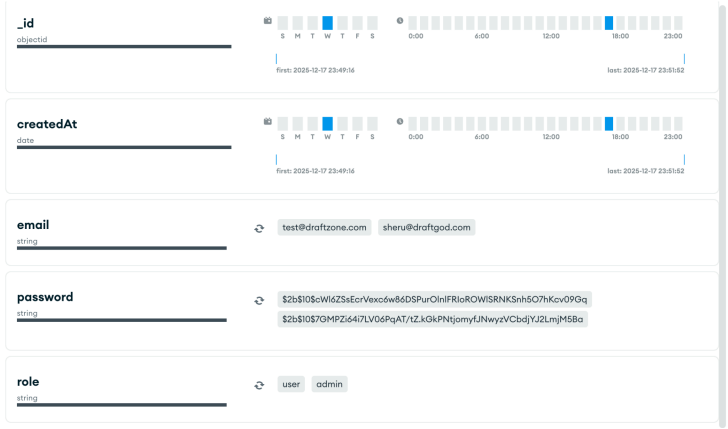
- **Drafts**



- **Players**



- **Users**



- **draftPicks Explanation:**
    - The draftPicks collection stores individual draft picks, including an ID and timestamp, the draft it belongs to, the player selected, the round and turn of the pick, the team, and the user who made the selection. All fields are required and type-checked to keep the data consistent. The draftId field links each pick to a

draft, even though MongoDB does not enforce foreign keys. Common operations include retrieving all picks for a draft, sorting them by round and turn, and adding new picks as the draft runs. MongoDB automatically indexes _id, and indexing draftId and the round and turn fields helps improve query performance.

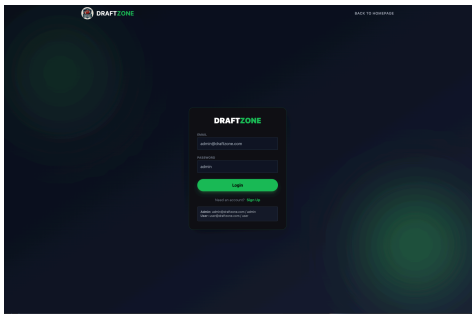
- **Draft Explanation**:
  - The draft collection stores all data for a fantasy draft in one place. It includes IDs and timestamps, basic draft state like game mode, status, current round, and turn, plus embedded arrays and objects for picks and user rosters. Picks record which player was taken, by whom, and when, while rosters track each user's selected players and bench. All fields are type-checked and required where needed to keep the data consistent. Common operations include loading a draft by ID, updating the round or turn, and adding new picks. MongoDB automatically indexes _id, and indexing active draft fields like status can help with performance.
- Players Explanation:
  - The players collection stores player data used for drafting and scoring. It includes an ID, the player's name and external player ID, position, team, search rank, and an embedded stats object with fantasy-relevant statistics such as points, yards, and touchdowns, plus an updatedAt timestamp. All required fields are type-checked to keep the data consistent. This collection does not depend on other collections and is mostly read-only during drafts. Common operations include finding players by position, sorting by search rank, and reading stats for scoring or display. MongoDB automatically indexes _id, and indexing fields like player_id, position, and search_rank helps improve lookup and sorting performance.
- Users Explanation:
  - The users collection stores account and authentication information. It includes an ID and creation timestamp, along with the user's email, hashed password, and role. All fields are required and type-checked to ensure valid user records. This collection is self-contained and does not directly reference other collections. Common operations include finding a user by email during login, creating new user accounts, and updating user roles. MongoDB automatically indexes _id, and indexing the email field improves lookup speed and prevents duplicate accounts.

5 Screenshots and Web Views

The Home page is the main landing page for DraftZone and is designed to introduce users to the platform and clearly communicate its value. It highlights DraftZone as an AI-powered fantasy

football tool through a large hero section with the headline “Master Your Fantasy Draft,” supported by a short description and eye-catching visuals. The navigation bar at the top allows users to easily move to key areas such as Players, Draft, Odds, and Login & Signup, while prominent call-to-action buttons like “Get Started” and “Start Drafting Now” guide users toward deeper engagement with the app. Feature highlights and credibility metrics help build trust and quickly explain what the platform offers.

Users can interact with the Home page by clicking navigation links or call-to-action buttons, which smoothly route them to other pages using client-side navigation without reloading the site. Scrolling down the page reveals feature cards that explain core functionality such as real-time rankings, advanced analytics, and the AI draft assistant, followed by a final call-to-action encouraging users to begin drafting. Visual effects like background animations, parallax scrolling, and navbar transitions respond dynamically to user scrolling, making the experience feel modern and engaging while keeping the focus on guiding users to start using the application.

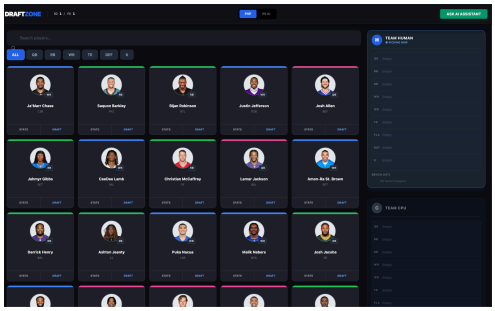


The Login page allows users to authenticate and access protected areas of the DraftZone application. Its main purpose is to collect a user’s email and password and send those credentials to the backend authentication API for validation. The page presents a simple, focused UI with the DraftZone logo, email and password input fields, a primary Login button, and a toggle that lets users switch between login and signup modes. An error message is displayed if authentication fails, and example credentials for admin and regular users are shown to help with testing and demonstration.

When a user submits the form, the page sends a POST request to the backend authentication endpoint using Axios. If the login is successful, the application stores the returned user information and automatically navigates the user to the appropriate page, such as the draft screen. If the credentials are invalid, an error message is shown without reloading the page. This page demonstrates how the frontend handles form submission, basic validation, error feedback, and client-side routing, while delegating all authentication logic to the backend API (separate from routes like playerRoutes.js, which handle player data).

The Player Search page allows users to explore and discover NFL players using an interactive search and filtering interface. Its main purpose is to help fantasy managers quickly find players by name, team, or position while viewing key information such as rankings and projected points. The page includes a prominent search bar, position filter buttons (QB, RB, WR, TE, etc.), and sorting controls that let users reorder results by rank, name, or projected points. Each player is displayed using a reusable player card component, making the layout clean and easy to scan.

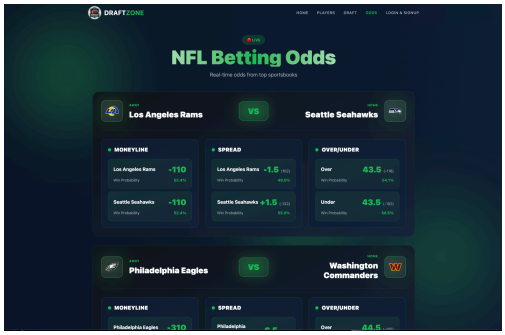
Users can type into the search bar to dynamically filter players, click position pills to narrow results, or change the sort option to compare players in different ways. Selecting a player opens a detailed view with more in-depth information, while a back action returns the user to the results list. All filtering, sorting, and selection happens instantly on the client side without page reloads, creating a smooth and responsive experience. Although this page currently uses local player data, it is designed to work seamlessly with backend routes (such as player APIs) to support real-time data in future versions.



The Draft Simulator page is the core interactive experience of the DraftZone application, allowing users to simulate a full fantasy football draft in real time. Its purpose is to let users search and browse available players, view basic stats, and draft players into their roster while tracking rounds and picks. The page presents a large player grid with position-based color cues, search and filter controls (QB, RB, WR, TE, DEF, K), and clear “Stats” and “Draft” actions on each player card. On the right side, live rosters for Team Human and Team CPU are displayed, showing which positions are filled, whose turn it is, and overall draft progress.

Users can interact with this page by searching for players, filtering by position, opening a detailed stats modal, or drafting players directly into their roster. Drafting a player immediately updates the roster, removes the player from the available pool, advances the turn and round, and visually highlights the active team. In VS AI mode, the CPU automatically drafts players using backend AI suggestion routes, while users can also request AI advice during their turn. Once

both rosters are full, the page transitions to a draft recap view that compares total projected points and declares a winner. All of this happens dynamically without page reloads, making the simulator feel fast, immersive, and game-like.



The Odds page displays live NFL betting odds pulled from an external sportsbook API and presents them in a clear, game-by-game layout. Its main purpose is to allow users to view real-time moneyline, spread, and over/under odds for upcoming NFL matchups in a visually intuitive format. Key UI elements include a live status indicator, team logos, matchup headers, and separate cards for each betting market. Win probabilities are calculated and displayed alongside each line, helping users quickly interpret the odds. A loading state and fallback message are shown when data is still being fetched or unavailable.

When the page loads, it automatically sends a GET request to the backend odds endpoint, which retrieves live data from a third-party odds API and returns it to the frontend. Once the data is received, the page dynamically renders each matchup and updates the UI without requiring a page refresh. Users can scroll through games to compare odds across matchups, while the animated background and hover effects enhance the overall experience. This page demonstrates full frontend-to-backend integration, combining asynchronous API calls, dynamic rendering, and real-time data presentation in a single view

.

6 Installation and Setup Instructions

This section describes how to install, configure, and run the DraftZone application locally. The project is divided into separate frontend and backend folders, each of which must be set up independently.

6.1 Frontend Setup

The frontend of DraftZone is built using React and Vite. To run the frontend locally, Node.js must be installed. A recent Node.js LTS version (Node 18 or later) is recommended.

First, navigate to the frontend directory and install the required dependencies using npm. Once installation is complete, start the development server using the provided Vite command. Then the application will be available at the local development URL provided by Vite

6.2 Backend Setup

The backend of DraftZone is built using Node.js and Express and is responsible for handling API requests, external API communication and database logic.

The backend uses Express for routing, Axios for API requests, CORS for cross-origin communication, and dotenv for environment variable support. Nodemon is included as a development dependency to allow automatic server restarts during development if desired.

First navigate to backend directory then run npm install and then run npm start

6.3 Database Setup

This project uses a locally hosted MongoDB instance running on the developer's machine. MongoDB is started locally, and the application connects to it using a connection string pointing to localhost. The database and collections are created automatically when the server first runs and Mongoose inserts data. Seed data, such as player information, is added by running a script or API route that fetches and saves data into the database. The MongoDB connection string is stored in the .env file as MONGO_URI and is loaded at runtime to establish the database connection. Since the database is local, no special network access or external permissions are required beyond running MongoDB on the machine.

6.4 Environment Variables

The backend supports environment variables through the use of dotenv. These variables may include values such as the server port number or API keys required for external data sources.

7 Contribution Overview

7.1 Member 1 Contributions - (Neel Rajan)

I developed three major pages for the DraftZone application, focusing on creating modern, responsive user interfaces with consistent design patterns. The Odds page displays live NFL betting odds fetched from The Odds API, featuring a card-based layout with moneyline, point spread, and over/under totals for each game. I implemented conditional rendering for loading states, empty states, and error handling to ensure a smooth user experience regardless of data availability. The page includes dynamic team logos, win probability calculations, and hover effects that enhance interactivity.

I developed the complete backend infrastructure for the Odds featureThis included creating the /api/odds GET route in the odds.js file that serves as a proxy between the frontend and The Odds API. The route handles API authentication, constructs requests with appropriate parameters for NFL games, and returns formatted data to the frontend. I implemented error handling with try-catch blocks that log errors to the console while returning user-friendly error messages. The route includes validation to check for valid API responses and uses axios to make external API calls with query parameters for regions, markets, and odds format.

I was responsible for debugging and fixing multiple issues throughout development. The most significant issue was resolving port conflicts where the backend server was already running on port 8080. I debugged routing issues where the odds route was returning 404 errors. The problem was that the route was defined as router.get("/odds", ...) while being mounted at /api/odds in server.js, creating a path of /api/odds/odds. I fixed this by changing the route definition to router.get("/", ...) so it correctly responds at /api/odds. I also resolved import path errors in the Login component where the DraftLogo asset couldn't be found and fixed navigation links.

I authored the complete Section 4 of the technical documentation, which covers Frontend-Backend Interaction and React Component Examples. I also created Section 4.3 API Route Examples, providing comprehensive documentation for four CRUD operations.

Additionally, I created a utility function calcWinProb() in the Odds component that converts American odds format to implied win probability percentages.

7.2 Member 2 Contributions - (Aadi Bhatia)

I was responsible for implementing several core frontend features and key backend systems for the project. I built the sign-up and login system, handling form state, validation, role selection (user vs. admin), authentication requests, error handling, and role-based routing after login. On the backend, I implemented the authentication routes, supporting both predefined accounts and MongoDB users, using bcrypt for secure password hashing and handling edge cases such as duplicate users and invalid credentials.

I implemented the Draft Simulator, which manages the full drafting experience including player search and filtering, roster construction, turn and round tracking, enforcing roster and bench limits, and displaying user-facing error messages. I added modals for player stats, visual indicators for active turns, and a draft recap screen that compares final team performance. I also developed the Player Search page, allowing users to browse, filter, sort, and view detailed player information with a focus on usability and smooth UI interactions.

On the backend, I implemented the playerRoutes, which fetch live NFL player data and season statistics from the Sleeper API. These routes merge player metadata with statistical data, normalize and format it for frontend use, apply ranking logic to prioritize relevant players, and return a sorted player pool used by both the Draft Simulator and Player Search page. This ensured consistent, up-to-date player data across the application. I also implemented the draftRoutes, which handle creating new drafts, saving individual draft picks, updating rosters, tracking round and turn state, finishing drafts with final results, fetching draft history, and deleting drafts. These routes directly manage draft state in MongoDB and keep the frontend synchronized with the database. In addition, I built the Admin Panel, which uses these routes to display system-wide draft statistics, draft history, and allows administrators to refresh data and permanently delete drafts. For the AI feature, I implemented both the frontend integration and the backend AI routes using Google’s Gemini API. I created an AI suggestion endpoint that analyzes roster needs, available players, and the current draft round to recommend optimal picks, and added strict JSON parsing, context limits, and fallback logic to ensure the draft flow never stalls. This logic also powers CPU auto-drafting in player-versus-AI mode.

Throughout development, I performed extensive testing and debugging to ensure reliable authentication, correct draft state transitions, stable AI behavior, accurate player data, and smooth interaction between the frontend, backend, and MongoDB database.

8 Challenges Faced

Each member must document the technical challenges they personally faced.

8.1 Member 1 Challenges - (Neel Rajan)

One major challenge I faced during the project was merging JSON data files when multiple sources had overlapping fields and slightly different structures. This caused unexpected behavior when rendering player and odds data in the frontend, such as missing values or incorrect mappings. I diagnosed the issue by logging the incoming JSON objects to the console and comparing their schemas side by side to identify where keys conflicted or were overwritten. I also gained a deeper understanding of git commands in the terminal to help me undo and get around merge issues.

Another challenge involved issues with the API GET call for fetching odds data, where requests were failing or returning unexpected responses. I debugged this by checking browser network logs, printing API responses, and reviewing error messages returned by the backend. I discovered that the issue was related to endpoint configuration and how the response data was being accessed on the frontend. After correcting the API route usage and adjusting how the response object was parsed, the odds data began loading correctly. From this experience, I learned how critical careful API debugging is, especially when dealing with asynchronous requests and gained confidence in using logs and network tools to trace and fix data flow issues between the frontend and backend.

8.2 Member 2 Challenges - (Aadi Bhatia)

One challenge I faced was keeping the Draft Simulator's frontend state synchronized with the backend and database while handling turn order, roster updates, and AI-driven picks. I debugged this by adding logging, testing edge cases such as invalid picks and AI timeouts, and verifying draft data directly in MongoDB. I resolved the issue by tightening turn validation, centralizing state updates, and adding fallback logic so the draft could continue reliably. This taught me the importance of backend-validated state and defensive programming.

A second challenge involved implementing secure and flexible authentication for both admin and user accounts. Issues arose around handling duplicate users, invalid credentials, and role-based routing after login. I debugged this by testing different login and signup scenarios and reviewing authentication responses from the backend. I fixed the problem by improving validation, clearly separating admin and user roles, and standardizing response formats. This reinforced the value of clear authentication logic and thorough testing of edge cases.

9 Final Reflections

Each team member reflects individually.

9.1 Member 1 Reflection - (Neel Rajan)

Through this project, I became more confident working with React and backend APIs. Building reusable components, managing state, and connecting frontend pages to API endpoints helped me better understand full-stack application flow. The most rewarding part was UI development, as it was satisfying to design clean, aesthetic pages and see real data, such as player information and odds, render correctly in the interface.

If I were to start over, I would redesign the bottom section of the homepage to improve layout balance and usability. Overall, I gained valuable skills that I can apply to future projects, including planning frontend components around backend data, handling asynchronous API calls, debugging integrations, and writing more scalable, maintainable code.

9.2 Member 2 Reflection - (Aadi Bhatia)

Working on this project helped me grow as both a frontend and backend developer. On the frontend, I became more comfortable managing complex React states, building reusable components, and designing responsive interfaces with Tailwind CSS. On the backend, I gained hands-on experience building REST APIs with Express, integrating MongoDB, and implementing authentication, validation, and error handling. I also improved my debugging skills by tracing issues across the full stack, using logging and testing edge cases to identify and fix problems efficiently.

Through this project, I learned what it takes to build a complete full-stack application where the frontend, backend, and database must work together seamlessly. If I were to continue improving the project, I would add real-time draft updates, enhance the AI drafting logic, and strengthen security with token-based authentication and role-based access control. Overall, this experience taught me the importance of clean API design, reliable state management, and thoughtful planning when developing scalable applications.