Report

Introduction

Our project includes making a gambling/betting website for football. We will involve making our own API for fetching data like players average score, player line up for live matches, plus getting a real API. Moreover, our design will look robust and colourful to entice users to play other than football. But the bigger question is with what do I bet with? we will implement a new system like currency that isn't liquid rather like V-bucks or Robux. What happens when a user bets on a team, and they lost? Well – we have added like a 'back-draw' system where if they betted a certain amount of points, they get half of what they originally betted. I thought of a way where the users get more points everyday by either log-in or well of course betting. We have already implemented that a user who logs in for the first time gets 100 points to bet with.

I know that betting can be addicting, the gambling industry in the UK has been growing exponentially over the years and statistics show that here in the UK the gambling community is an incredibly large part of the economy. So soon, we will add a limit in our website that a user can bet a certain amount of money and bets per day, so it doesn't cause issues to our users. We also implement a Leader board and user stats on how much they won on certain live matches. The layout I will follow will be Designs/Images, Testing, Conclusion, and lastly references.

Designs/Images of our Website

The key features we included:
- Live Betting: Real-time match updates, dynamic odds, live statistics, and in-play betting options. Also, Fake data.

```
type LiveMatch = {
  id: string;
  tournament: string;
  minute: number;
  team1: {
    name: string;
    score: number;
  };
  team2: {
    name: string;
    score: number;
  };
  odds: {
    team1Win: number;
    draw: number;
    team2Win: number;
  };
};
```

- Pre-match Betting: Upcoming matches, tournament predictions, league-specific bets, and special event markets.

```
type Bet = {
  id: string;
  matchId: string;
  match: string;
  selection: string;
  odds: number;
  stake: number;
  potentialWin: number;
  isLive?: boolean;
};
```

- User Account System: Virtual currency, bet history tracking, profile management, and favourites/bookmarks.

- Statistics and Analysis: Match statistics, team performance metrics, player statistics, and league standings.

The platform is built using a modern tech stack including React, and TypeScript. It is deployed on the Vercel platform with edge functions and CDN optimization.

FootballX was designed with simplicity and performance in mind, using data structures that naturally align with the types of information the application handles. The primary structures used include TypeScript objects and arrays, which are ideal for organizing and processing real-time and pre-match data.

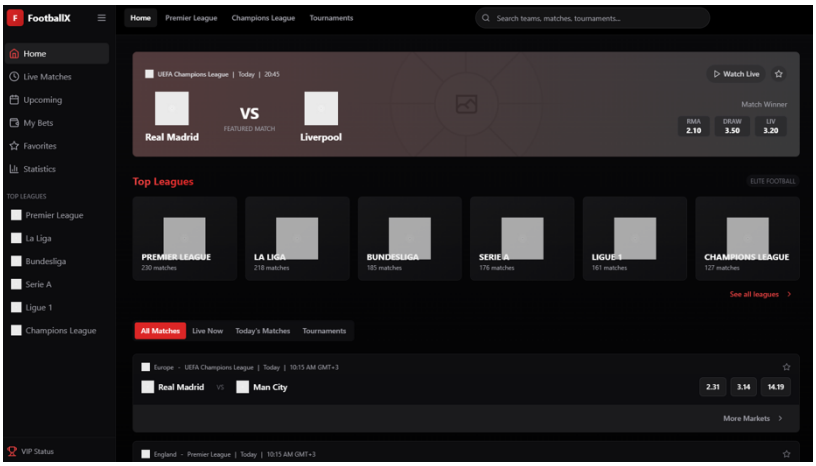
For storing live match data, we defined a ‘LiveMatch’ type that contains structured information such as team names, current scores, odds for different outcomes, and the match's current minute. Each match object also includes an ID and tournament name. This structure allows for quick retrieval and updates of match-specific data, which is essential for real-time performance. Data is typically handled in arrays of these match objects, which can be efficiently looped through, filtered, or mapped when rendering to the frontend.

Betting data is handled using a ‘Bet’ type that stores essential information like the bet ID, the associated match ID, the user's selected outcome, the stake, and the calculated potential win. This structure also supports an optional Boolean ‘isLive’ flag to differentiate between live and pre-match bets. Using a match ID as a reference allows us to link a bet directly to a specific live match, enabling real-time tracking and updates.

In terms of algorithms, two key areas i’ve analysed: the filtering of live matches and the calculation of live betting outcomes. The filtering algorithm is used when a user wants to view matches only from a specific tournament or league. This operation involves checking each match in the list and returning only those that match the selected criteria. This is a linear-time operation, with time complexity O(n), where n is the number of live matches currently active.

Figure 1

We got this picture as an inspiration from Dribbble.com. We changed pretty much everything but stuck to the same framework as the picture. The thing I was mostly thinking when choosing this design was to not over complicate ourselves, meaning we have some crazy dynamic button that shows the users wallet before betting on a match or create some design that drifts off from our aim.



This is the first draft and final draft (with images rendered) of our website, it has “live matches” (its old matches but we call it live because if its live the tester has to wait until the match begins) which any user can bet on wither the teams or players depending how many goals they score or even assists.

For the front-end, one of our developers will use TypeScript/React because it is vastly used and there’s loads of resources out there that could help them throughout our project. To ensure responsiveness in the live betting and statistics features, functions were written with efficiency in mind. For example, filtering and displaying live matches is implemented using optimized state management in React, with data processing functions having a time complexity of O(n), allowing smooth performance even as the number of live events increases. We added bubble sort to the leader board where people with the highest amount of money are at the top of the leader board and the lowest amount of money at the bottom, so highest to lowest. Then within the back end we again added bubble sort from lowest to highest so we can see the newest users from the oldest users that are logged in.

However, with the backend, our developer (Tomi) did have a hard time using the APIS but in the end Tomi got it to work. Obviously, our back-end developer will use C# and will sort out the connection with our SQL database. We thought what we are going to include in the

database like users(passwords), user's wallet, and if we want to make it more interesting; adding data on how much the user lost or spent. Our backend developer (Tomi) made a Git repo to test how the functionality will work before merging it to the main repo with the front end.

Josh (Team Leader) thought it was a good idea to put the first draft (when it's done) on a free domain so its easier for our tester (Chris) to test the websites capability's. We could implement a monthly payment where the predictions are like a VIP subscription. But we will still work on our own API so it is easier to test rather than Live data API. On our recent meeting we thought that our tester should start testing our final git pushes and found out we had issues with the filtering system with teams so our frontend developer (Rigon) got right to work.

The way we will set the data will be in json format so we can fetch it easier with the front end. We will use algorithms like bubble sort to repetitively go throughout the data to check if they are sorted out correctly. However, when we use bubble sort with saving users' data on our database is decent because it can manually sort out users out depending on their currency and the ones who have just made an account. We have added a sort function so we can filter them using their usernames which is beneficial. We also have switched to SQLite because it is more of a file-based database with no separate database server needed on the side.

The reason we switched databases was because we had some issues with SQL Server where we would have connectivity issues and outdated indexes that we always needed to update. This also gave us the reason to make it File-based storage (Football.db) so it can easily be seen for users. Our User Model has basic user information which also tracks balance management plus authentication fields.

Testing

The way we validated the functionality and stability of the website, a combination of manual testing and automated testing tools was used. Focus areas included real-time data updates, user interactions, responsive design, and backend functionality.

We tested features such as live betting, account management, and statistics display across multiple devices and browsers. Automated tests were written for critical frontend components, while API endpoints were tested using tools like Postman and Swagger.

Performance testing was conducted to assess system behaviour under simulated user load. Additionally, efficiency and responsiveness were monitored during testing. For example, filtering functions used in displaying live match data were optimized to run in linear time complexity (O(n)), ensuring smooth performance as the data set increases.

Rankings

GET `/api/UserRankings/sorted` Gets all users sorted by their coin balance (lowest to highest)

Sample request:

```
GET /api/userrankings/sorted
```

Sample response:

```
[
    {
        "username": "player1",
        "coins": 50
    },
    {
        "username": "player2",
        "coins": 100
    }
]
```

Responses

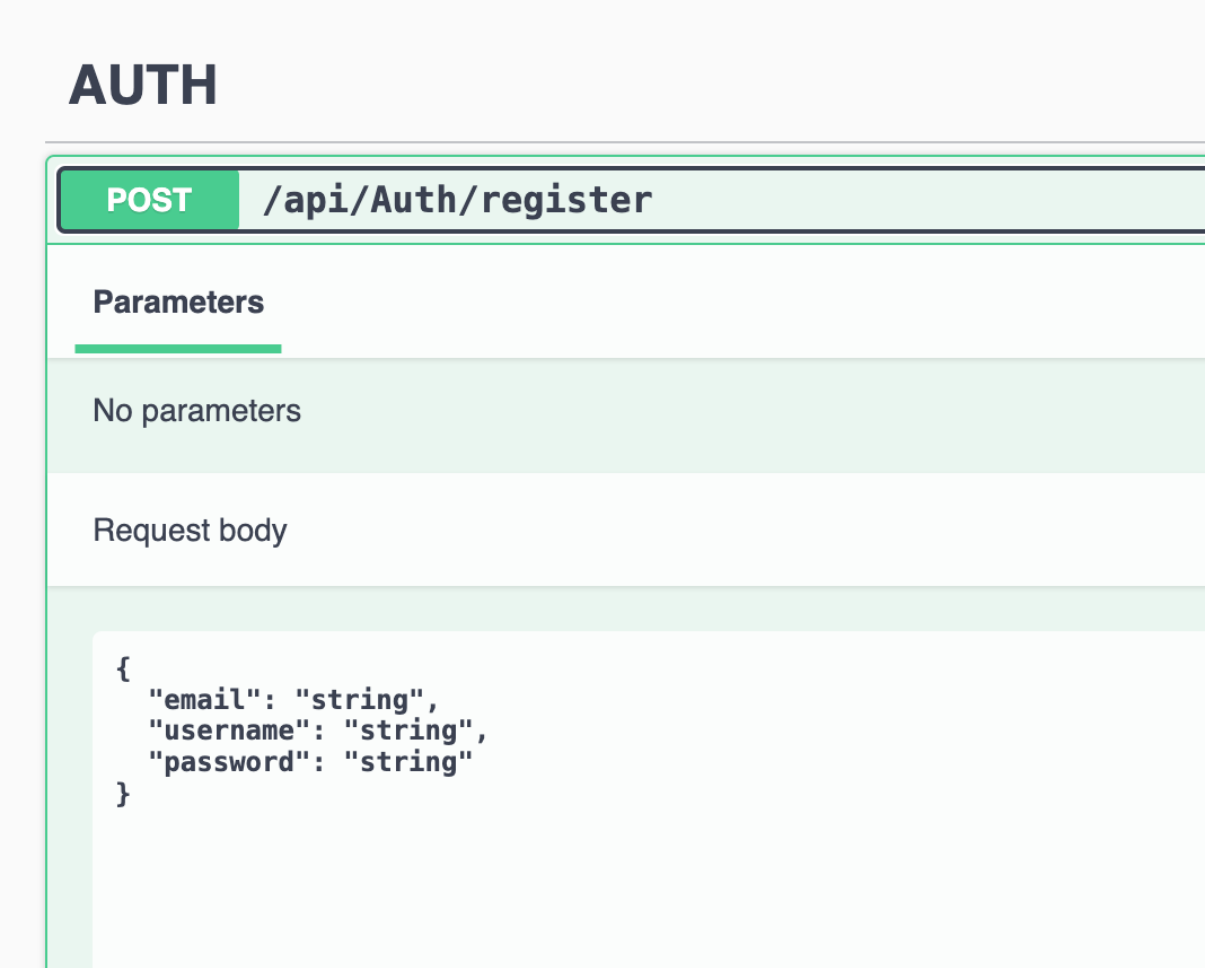
Code	Description
200	Success

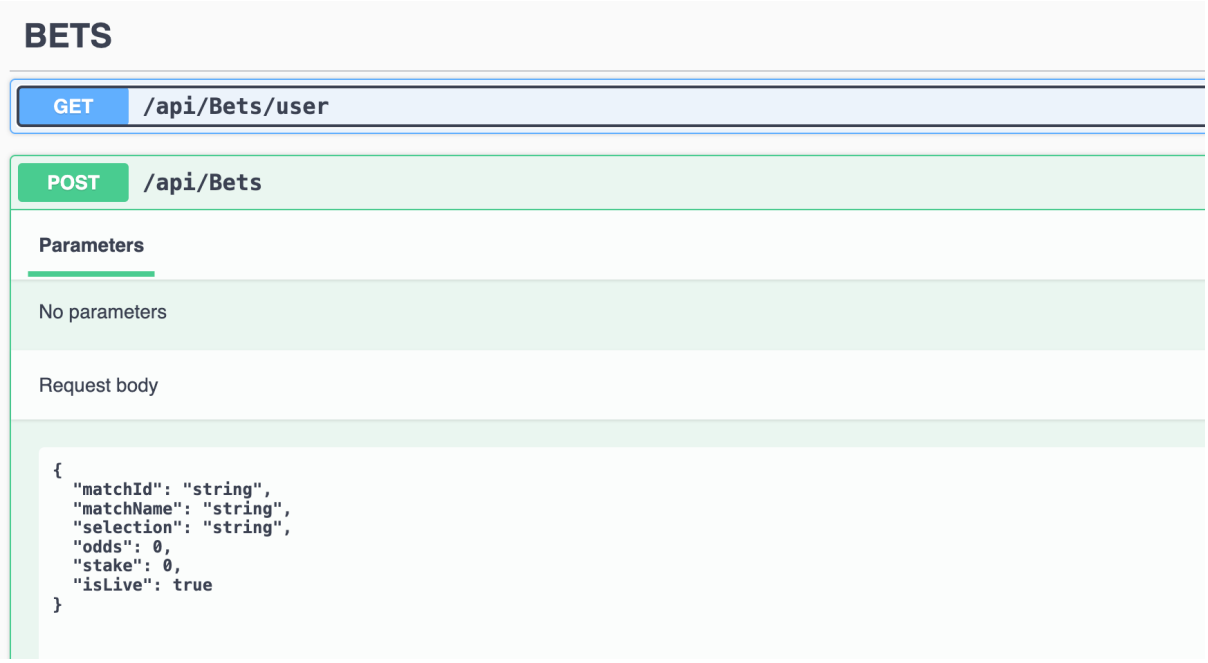
Media type: text/plain

Controls Accept header.

Example Value | Schema

```
{
  "success": true,
  "players": [
    {
      "id": 0,
      "name": "string",
      "commonName": "string",
      "position": "string",
      "nationality": "string",
      "currentTeam": "string",
      "age": 0,
      "imageUrl": "string"
    }
  ]
}
```


AUTH
POST
/api/Auth/register
Parameters
No parameters
Request body
{
"email": "string",
"username": "string",
"password": "string"
}


BETS
GET
/api/Bets/user
POST
/api/Bets
Parameters
No parameters
Request body
{
"matchId": "string",
"matchName": "string",
"selection": "string",
"odds": 0,
"stake": 0,
"isLive": true
}

Conclusion

Overall, we made a working product that hits our criteria, both in terms of design and functionality. We believe our users also have a right to give feedback on our work — their judgment could help us make the website even better in future versions. As every developer knows, there is always room for improvement.

However, there were some constraints throughout the process. One of the major issues we faced was coordinating team meetings, which occasionally hindered development and decision-making. We also found it difficult to fully meet all our initial design goals as a group, mainly due to differences in interpretation and time constraints.

From a critical perspective, one area for improvement would be pushing updates more consistently to our Git repository. This would have helped us better track our progress and collaborate more smoothly. Another challenge we encountered was attempting to set up deployment with Microsoft Azure. Unfortunately, we could not figure it out in time due to it being all our first time trying to use it. In the future, we would devote more time to understanding deployment platforms early in the project cycle to minimize last-minute adjustments.

If we did a comparable project again, we would improve our communication and scheduling as a group. We would also plan more technical learning sessions as a group for tools we don't fully understand, like Azure — so we can avoid getting stuck. By addressing these limitations and reflecting on our process, we are confident we could deliver an even stronger product next time.

Reference

Figure 1:
Kiriakov, F., 2025. *Dynamic Sports Analytics Dashboard.* [Online]
Available at: https://dribbble.com/shots/25514823-Dynamic-Sports-Analytics-Dashboard
Date added: 28/03/2025

Main repo with Both front end and back end made by everyone:
https://github.com/dancer/Frontend
First repo made by Tomi with the back end: https://github.com/iokitpusher/backend

Git Usernames:
dancer: Josh
l0af1: Nicolas
rigonbajrami: Rigon
iokitpusher: Tomi
chris22dg: Cristian