Fantasy football

Introduction

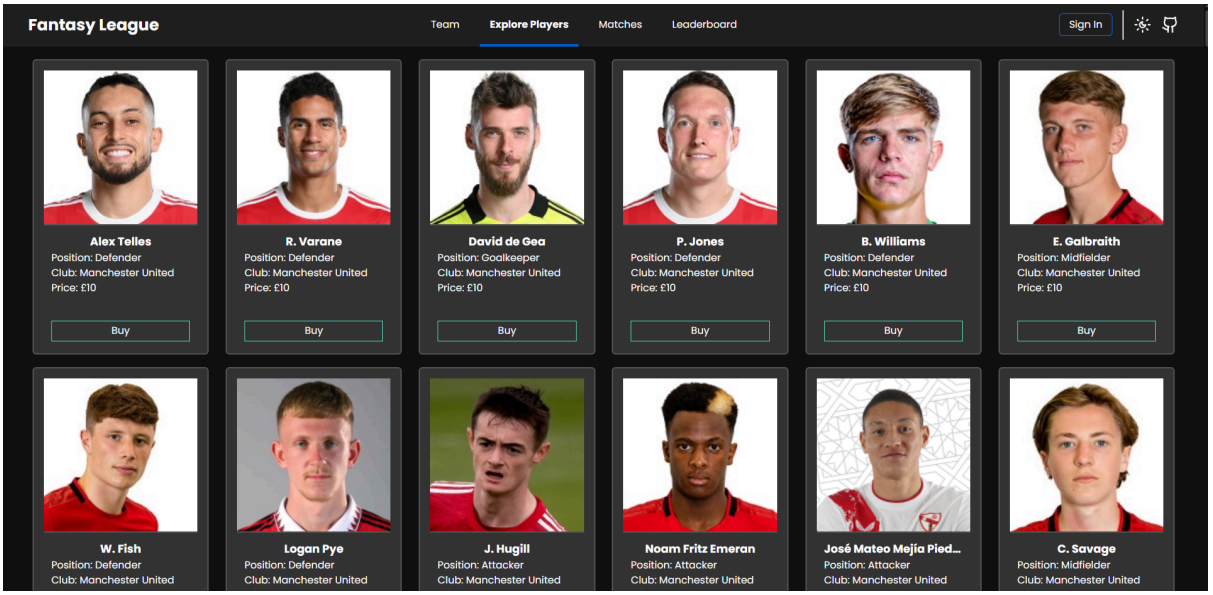
[Github-repo]

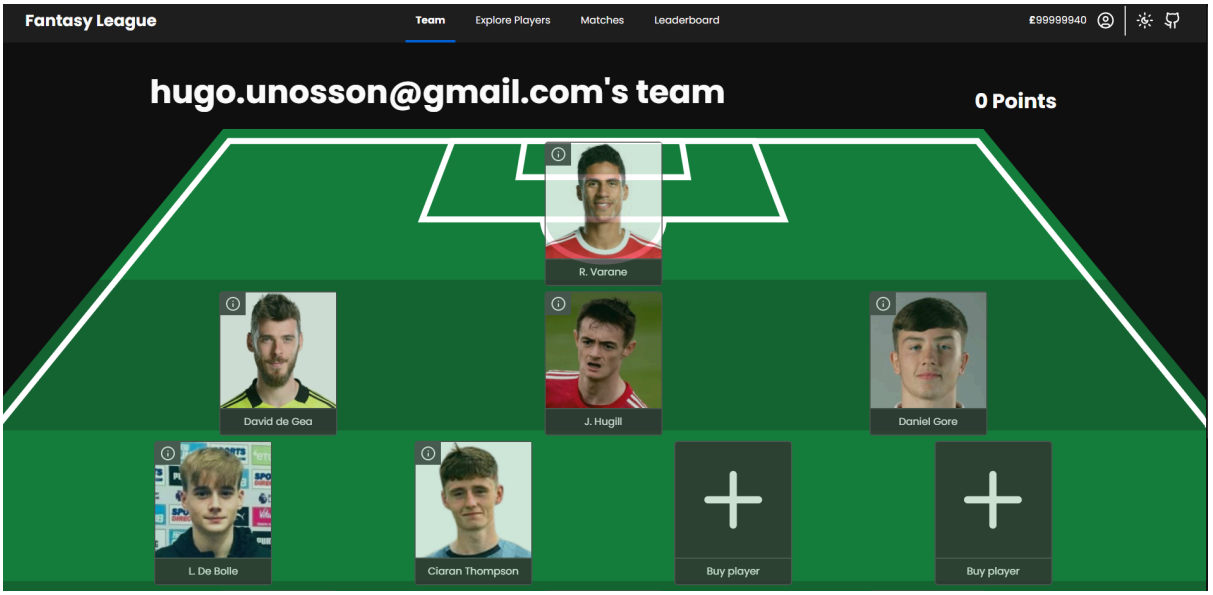
The project is a web-application for fantasy football where users can draft and manage their own virtual football team to compete with other users based on real-world player performances.

On startup, the user will be navigated to the welcome page. The user can access the “Explore Players” tab without logging in. To start playing the game, the user will have to create an account. The application contains four different tabs: “Team”, “Explore Players”, “Matches”, and “Leaderboard”.

- **Team:** Explore your current team, explore additional stats, and sell the once that you do not want.
- **Explore Players:** Explore all the players that exist in the database. Buy the once that you want to add to your team.
- **Matches:** Here you can start the game. The fixtures are played at a certain time each week. After each fixture, points are awarded to each football player. In the current version of the game, points are randomized. Each fixture is played every minute. Users can change their teams between each fixture.
- **Leaderboard:** Displays a list of all the users and their team’s score. The team's score is based on how many points each soccer player in the user's team has received.

The images below showcase the “Team” tab and the “Explore Players” tab.





Use Cases

As a user, I want to be able to view all players available for drafting.

As a user, I want to be able to see more information about an individual player.

As a user, I want to be able to buy players to my team.

As a user, I want to be able to sell players from my team.

As a user, I want to be able to save my favourite players in a team and view them later.

As a user, I want to see my team's points, so that I can see how they perform over a season.

As a user, I want to have a balance for my team, so I know how expensive players I can afford.

As a user, I want to simulate football games, so that I can progress with my team. A real time football season takes a long time, therefore I want a faster way to simulate it.

As a user, I want to be able to see the status of the current season to know what actions I can take. For example if I can join with my team, view the current standings or if the season has ended and I can view my final rank.

As a user, I want to be able to see the top performing players to know how I can improve my team.

As a user, I want to be able to see my and other users' teams current rankings and points on a leaderboard.

User manual

Set up the application

1. To run our web-application you need to clone the repo.
2. Navigate to the root folder and run “npm install” in the terminal.
3. Navigate to the “client” folder by running “cd /client/”.
4. Run “npm run dev” to start the application's client side.
5. Open a new terminal window.
6. Navigate to the “server” folder by running “cd /server/”.
7. Run “npm run dev” to start the application's server side.

Set up environment variables

You will have to create three variables in an .env file to get everything to work. The .env file should be located in “/server/src/.env”. Add the following variables:

1. **SESSION_SECRET** - A string used to hash passwords.
2. **MOCK_PLAYERS** - This can be either set to true or false depending whether you want to use real-world players or not. If you do not want to use real-world players, set this to true.
3. **API_KEY** - An API key used to fetch football players from www.api-football.com. To get your API key, create an account and navigate to your profile where your personal API key can be found. Note! This is only needed if MOCK_PLAYERS=false. If you intend to use mock players, then leave the API_KEY as an empty string.
4. **CONNECTION_STRING** - This is the key to access your database. The one provided in the example file below is a connection string to your database without authorization. If you used a password to the database this is the correct string: ‘postgres://app-db-user:<password>@localhost:5432’. Where app-db-user is your username and <password> is your password.

Here is an example of how a file with environment variables could look like:

```
SESSION_SECRET = "5e95ec694d09362a"
API_KEY = "my-secret-key"
MOCK_PLAYERS = "true"
CONNECTION_STRING = 'postgres://postgres@localhost:5432'
```

Set up database

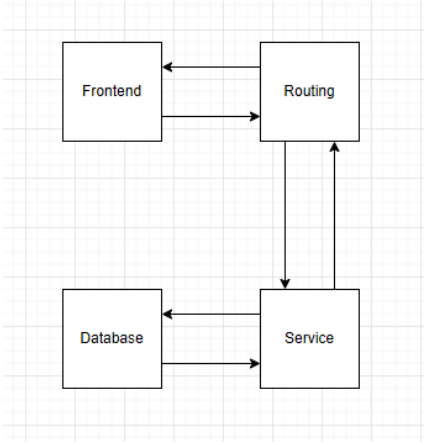
To setup the database without authorization, you need to have docker running on your machine. The only thing you need to do after that is to run the command “docker run --env POSTGRES_HOST_AUTH_METHOD=trust --publish 5432:5432 --name web_apps_db --detach postgres:17” in your terminal/console.

To setup the database with authorization/password, you need to have docker running on your machine and then run the command “run -d --name web_apps_db -e POSTGRES_USER=postgres -e POSTGRES_PASSWORD=securepassword123 -e POSTGRES_DB=my_database -p 5432:5432 postgres:17”. Where POSTGRES_USER is your user, and POSTGRESS_PASSWORD is your password.

Don’t forget to update your connection string accordingly!

Design

The core flow of information or design of our web-application is that our frontend communicates with the backend through a routing layer by API requests. The routing layer then communicates with the service layer, which handles most of the logic as well as communicating with the database. The database returns an answer to the service layer, which returns to the routing layer and finally to the frontend.



The plan was to send requests to an external API to retrieve live player data and matches, but we did not have the time or resources to implement this, so we settled on mockup players and matches. These requests would have been done in the service layer of the backend and stored in the database. The frontend would first update when refreshing the site or switching the route/url, and would display the new ratings and results. The flow of information would be very similar to what we have already implemented.

Besides the base libraries, frameworks and runtime environments introduced in the course such as React, Node, Bootstrap and many more, we also used Skeleton.dev with Tailwindcss. We used Skeleton.dev for functional components and Tailwindcss integration, to spend less time on making the web-application look good.

PlayerCard
PlayerCard is a component that displays the player image as well as some information about the player. It has the following props: player(Player) which is the player.

PlayerCardAdditional
PlayerCardAdditional is PlayerCard with additional information about the player, its purpose is to display the player and that information. It has the following props: id which is the id(number) of a player, onClose which is a function regarding what should happen when you click away the card, and fieldcase which is a boolean used to see if we need to display a close button, depending on if the component is displayed in the shop or on the field. It has the following states: loading(boolean) used to see if the card is ready to be displayed,

player(Player | null) which is the player, teamPlayers(Player[]) which are all players in your team. recentForm(number | null) which is the last 4 average ratings of the player , nextMatchAvailability(boolean | null) which is if the player is available for the upcoming game and finally gameSessionActive which is if you have joined a league.

It calls the backend to retrieve a specific player and all players in your team.

PlayerSlot
PlayerSlot is the slots on the field that the player-cards are displayed in. It has the props: initialPlayer(Player | undefined), setPlayerAvailable: (id: number, available: boolean) => void which sets the players availability and onInfoClick: () => void which is currently only used for displaying PlayerCardAdditional.
It calls the backend to retrieve if there is a game session, to get the current round and to get the rating of the player.

SellButton
SellButton is a button used for selling players. It has the the prop: playerId(number)

BuyButton
Buybutton is a button used for buying players, it has the props: playerId(number) which is the players id, and completed(boolean) which is used to determine if the player purchase went through.

TeamPoints
TeamPoints is used to display your teams current points for this round. It has the state teamPoints which are the teams points. It calls the backend to retrieve your teams points.

ThemeToggle
ThemeToggle is basically a button that switches the theme from dark to light or vice versa. It has the state theme(string) which is either “dark” or “light”.

TopPerformers
TopPerformers presents the top 10 best performers from the last round. It has the prop round(number) and the states players(Player[]), ratings(number | null), loading(boolean). It calls the backend through getTopPerformers to retrieve a descending ordered list of all players from the round based on rating. We then use getRating to retrieve the exact rating of each of those players.

Field
Field presents your team on a football field with 11 PlayerSlots. It has the props: numDefenders, numMidfielders and numAttackers which are all numbers representing the amount of players on each position.

LoginOrBalance

LoginOrBalance presents either a “sign in” button or the balance of the team/user depending on if you are signed in or not.

LuicideCircleUser
This is just an icon used to display the navigation to login in the navigation bar.

ProtectedComponent
This is used in navigation, when a routing should not be accessed without being logged in. This components purpose is to protect children components from unauthorized users. If the user is not logged in, they will be navigated to /login. If the user is logged in, the user will see the children component of “ProtectedComponent”.

BuyView
This view displays a table format with all the players in the database. It can be used to buy and sell players.

FieldView
This is the view of the users team on the field. The purpose of it is to display the current players as PlayerCards. It retrieves the team's player through the backend.

LeaderboardView
This view displays all users sorted by their team’s current points.

LoginView
This is the view for the login function. It handles the input from your email/username and password through a form. This is then sent to the backend through a POST request to locate your specified user in the database.

MatchesView
See status for season, either if the season hasn’t started and you can join, the season is in progress or the season is over and you can view the leaderboard.

PlayerView
This is the view that showcases all available players. There are 2 views available of the players, the default view shows 6 players per row, and unlimited rows until all players have been shown. The other sorts them based on their position. It retrieves all players from the backend through a GET request.

RegisterView
This view is for the register function. It handles the input from your email/username and password through a form. This is then sent to the backend through a POST request for hashing and then storing in the database.

StartPageView
This is the home page for the web app. The purpose is to welcome the user and link them to the register page. The page does not have any props or states and does not make any calls to the backend.

API Specification

/player
This endpoint accepts GET as a verb and no request body. 200 OK Returns an array of players. 500 Internal Server Error - Returns an error message.

/player/:id
This endpoint accepts GET as a verb and no request body. 200 OK Returns player data for given ID. 400 Bad Request, “Missing id param” or “id number must be a non-negative integer”. 404 Not Found, “Player ${req.params.id} not found”. 500 Internal Server Error.

/player/:id/rating/:round
This endpoint accepts GET as a verb and no request body. 200 OK Returns the players rating for the given round. 400 Bad Request, “Missing id param”, “Missing round param”, “round number must be a non-negative integer between 0 and 38” or “id number must be a non-negative integer”. 404 Not Found, “Rating not found for player: ${id} in round: ${round}”. 500 Internal Server Error.

/player/:id/availability/:round
This endpoint accepts GET as a verb and no request body. 200 OK Returns true or false for player availability. 400 Bad Request, “Missing id param”, “Missing round param”, “round number must be a non-negative integer between 0 and 38” or “id number must be a non-negative integer”. 404 Not Found, “Rating not found for player: ${id} in round: ${round}”. 500 Internal Server Error.

/player/:id/form/:round
This endpoint accepts GET as a verb and no request body. 200 OK Returns the players form for the given round. 400 Bad Request, “Missing id param”, “Missing round param”, “round number must be a non-negative integer between 0 and 38” or “id number must be a non-negative integer”. 404 Not Found, “Rating not found for player: ${id} in round: ${round}”. 500 Internal Server Error.

/player/performance/:round
This endpoint accepts GET as a verb and no request body. 200 OK Returns an array of top performers for the given round. 400 Bad Request, “Missing round param” or “round number must be a non-negative integer between 0 and 38”. 404 Not Found, “Rating not found for players in round: ${round}”. 500 Internal Server Error.

/team/players
This endpoint accepts GET as a verb and no request body. 200 OK Returns an array of players in the logged-in users team. 401 Unauthorized, “Not logged in”. 500 Internal Server Error.

/team/balance
This endpoint accepts GET as a verb and no request body. 200 OK Returns the current balance of the logged-in users team. 401 Unauthorized, “Not logged in”. 500 Internal Server Error.

/team/points
This endpoint accepts GET as a verb and no request body. 200 OK Returns the current points of the logged-in users team. 401 Unauthorized, “Not logged in”. 500 Internal Server Error.

/team/:id
This endpoint accepts POST as a verb and request body: { “action”: “buy” } or { “action”: “sell” }. 201 Created Returns the bought or sold players details. 400 Bad Request, “Missing id param”, “Field 'action' has type ${typeof (req.body.action)}”, “id number must be a non-negative integer” or “Invalid action: ${action}”. 401 Unauthorized, “Not logged in”. 404 Not Found, “Player ${id} unavailable, too expensive, already bought, or not found” or “Player not found: ${id}”. 500 Internal Server Error.

/user
This endpoint accepts POST as a verb. The body should contain: { username: string, password: string }. 201 Created means that a new user was successfully registered. 400 Bad Request, “Missing username or password” or “Username and password must be strings”. 409 Conflict, “Username already exist”. 500 Internal Server Error.

/user/login
This endpoint accepts POST as a verb. The body should contain: { username: string, password: string }. 200 OK means a user was successfully logged in, “Logged in as ${username}”. 400 Bad Request, “Missing username or password”, “You are already logged in”, or “Username and password must be strings”. 401 Unauthorized, “Invalid username or password”. 500 Internal Server Error.

/user/logout
This endpoint accepts POST as a verb and no body. 200 OK means a user was successfully logged out, “Logged out successfully.”. 500 Internal Server Error, “Failed to log out.”.

/user/check-session
This endpoint accepts GET as a verb and no body. This checks if a user is logged in or not. If the user is logged it returns { loggedIn: true, user: req.session.user }, and if not it returns { loggedIn: false }.

/gamesession
This endpoint accepts POST and GET as a verb and no request body.
For POST:
200 OK true if games session started successfully, false otherwise. 401 Unauthorized with message “Not logged in”, 500 Internal Server Error with error message.
For GET:
200 OK true if the user is in an active game session, false otherwise. 401 Unauthorized with message “Not logged in”, 500 Internal Server Error with error message.

/gamesession/finished
This endpoint accepts GET as a verb and no request body. 200 OK true if game session is finished, false otherwise. 401 Unauthorized with message “Not logged in”, 404 Not Found with message “No game session found for user ${user}”, 500 Internal Server Error with error message.

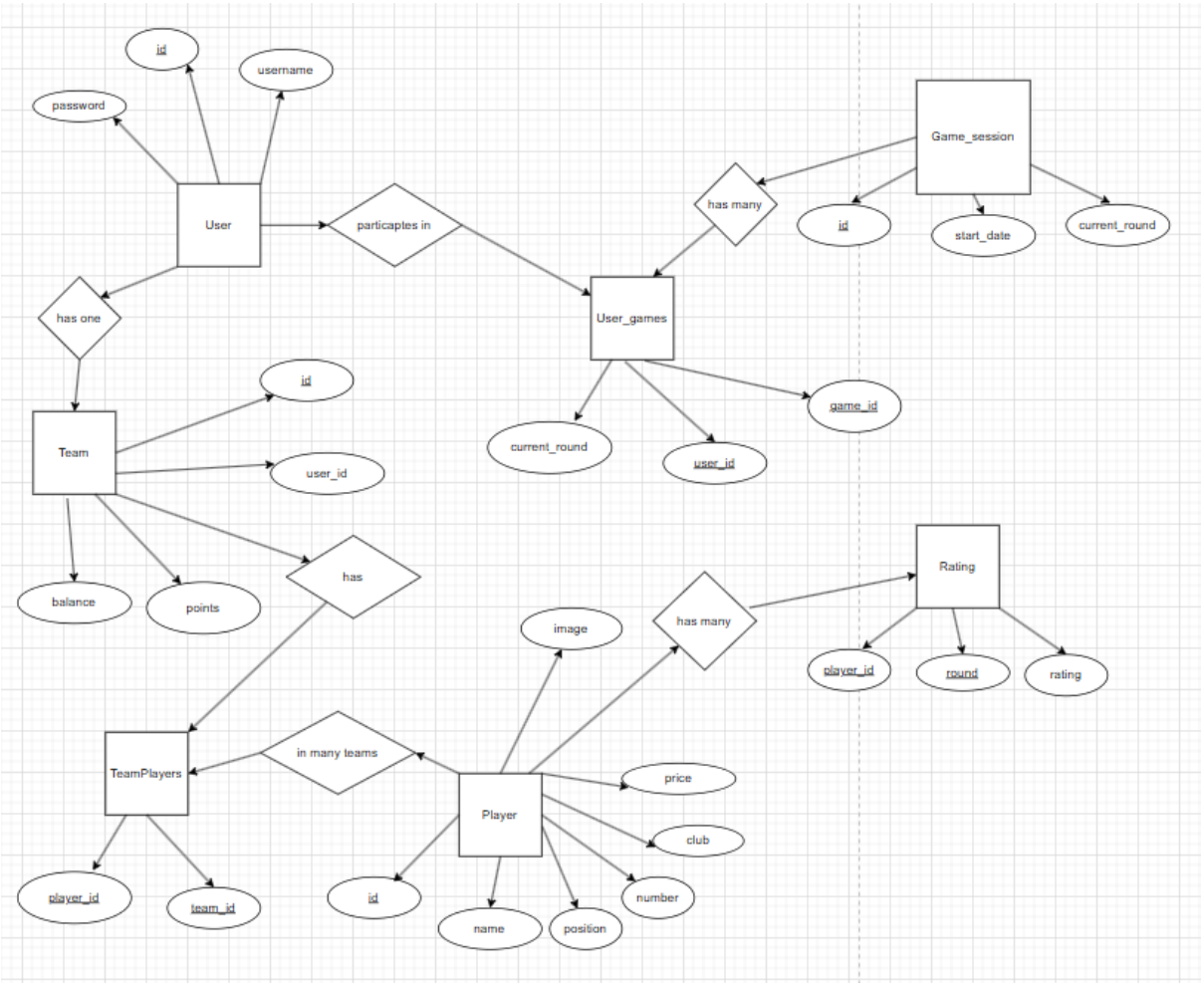
/gamesession/matchesActive
This endpoint accepts GET as a verb and no request body. 200 OK true if matches are in progress, false otherwise. 401 Unauthorized with message “Not logged in”, 404 Not Found with message “No game session found for user ${user}”, 500 Internal Server Error with error message.

/gamesession/round
This endpoint accepts GET as a verb and no request body. 200 OK current round number. 401 Unauthorized with message “Not logged in”, 404 Not Found with message “No game session found for user ${user}”, 500 Internal Server Error with error message.

/gamesession/state
This endpoint accepts PATCH as a verb and no request body. 200 OK “Game session state updated”. 401 Unauthorized with message “Not logged in”, 404 Not Found with message “No game session found for user ${user}”, 500 Internal Server Error with error message.

/gamesession/leaderboard
This endpoint accepts GET as a verb and no request body. 200 OK Returns an array of leaderboard data ([username1, 100]). 401 Unauthorized with message “Not logged in”, 404 Not Found with message “Couldn’t get leaderboard for user ${user}”, 500 Internal Server Error with error message.

ER Diagram of Database



Responsibilities

Hugo:
I have created the backend for user authentication. I have created the integration with the external football api, where we fetch real players and add them to the database. I have created the protected component, which is used to protect frontend from non-authenticated users.

Andreas:
I was mostly responsible for the frontend. I styled most of the website, for example the list in player view, the field/team view, appbar, leaderboard view and match view. I also made the authentication information be available everywhere in a context through authcontext with useauth that we used throughout the app and that I needed when I made the balance/sign in button for the header so it would update. I wrote some tests and fixed some error handling. I also contributed small stuff here and there when helping and working with the others in the group.

Ludwig:
I made PlayerView, PlayerAdditionalCard and logic regarding those. Frontend for LoginView and RegisterView and StartPage. Wrote tests for playerDB as well as for userDB in the service layer, as well as the test for player in the router layer. I contributed to small things in a lot of places, but I don't remember exactly what.

Marco - The database tables, the backend: in the service layer: playerSevice, teamService, userService, gameSessionService and the pointSystemService. The router layer: game_session, player and team. Also written tests for the backend on the service and router layers for most of the classes. On the frontend: the actionButton component and buy/sell buttons.