AI FOR SOFTWARE ENGINEERING - ISIS4226

BETSMART - PROJECT DOCUMENTATION

System Architecture and design decisions:

For this project, we used a pre-built technology stack based on JavaScript technologies called MERN (Mongo, Express, ReactJS, NodeJS).

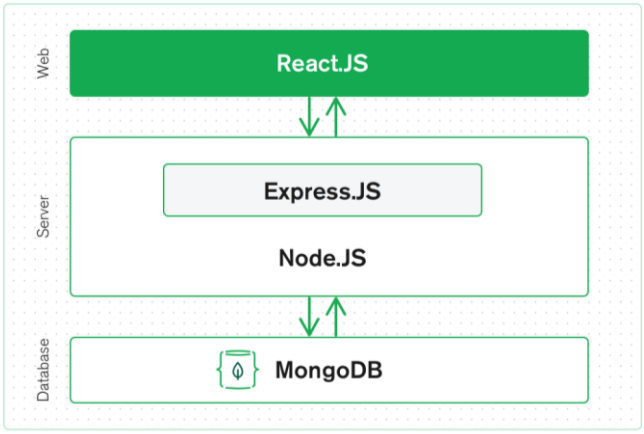
Benefits:

1. **Full-Stack JavaScript:** The MERN stack allows us to use JavaScript for both the front-end and back-end development, creating a uniform environment. This consistency helps in reducing context switching and allows developers to be more versatile, working on both client-side and server-side components.

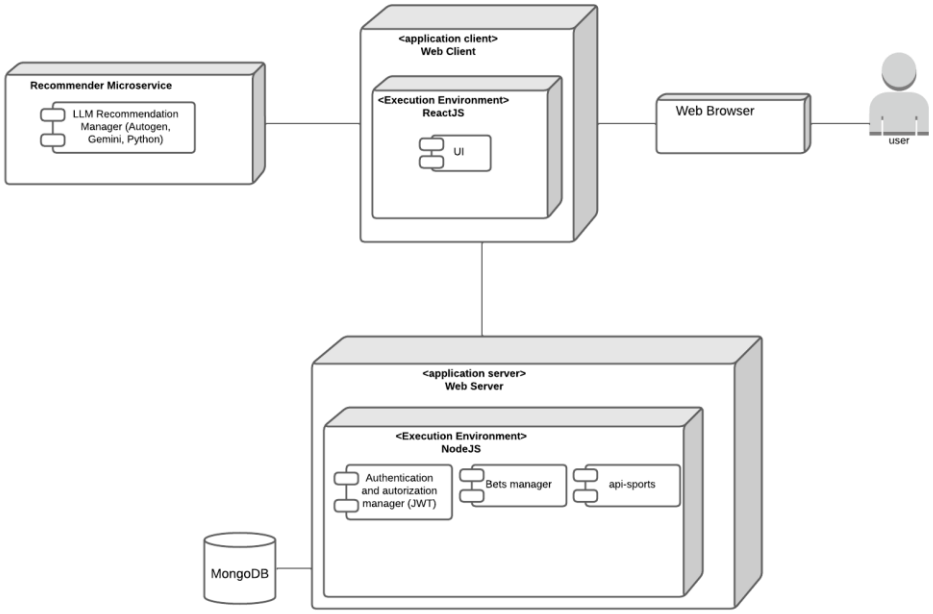
2. **Efficiency and Speed:** Node.js enables high-performance server-side execution, handling multiple requests simultaneously without slowing down. Coupled with MongoDB's NoSQL nature, this leads to faster query processing and efficient data handling, crucial for scalable applications.

3. **Flexibility:** ReactJS provides a component-based architecture, promoting code reusability and easier management of complex user interfaces. This modularity facilitates the development and maintenance of dynamic and interactive web applications.

4. **Scalability:** MongoDB's flexible document structure allows for efficient scaling as data needs grow. This scalability is essential for applications expected to handle increasing amounts of data and users over time.



MERN Stack Explained | MongoDB



Implementation Details:

User Authentication and Account Management (Harold Nicolás Coca):

For BETSMART, we implemented a comprehensive user authentication and account management system using the MERN stack (MongoDB, Express.js, ReactJS, Node.js). This system incorporates JWT for token-based authentication, Nodemailer for password reset functionality, protected routes for authenticated users, UserContext for managing user state on the front end, and Axios for handling HTTP requests.

Backend (Node.js and Express.js)

User Model: We defined a User model using Mongoose to manage user data in our MongoDB database. The model includes fields such as username, email, password, and additional fields for password reset tokens and their expiration times.

JWT Token Generation: For user authentication, we used the jsonwebtoken package to generate JWT tokens. These tokens are created during user login and are included in the response to the client. The token contains user-specific information and an expiration time, ensuring secure and temporary access.

Authentication Routes: We implemented routes for user registration, login, and logout:

Registration Route: Handles new user registration by hashing the password and saving the user details in the database. Upon successful registration, a JWT token is generated and sent to the client.

Login Route: Verifies user credentials, and if they are correct, a JWT token is generated and sent to the client.

Logout Route: Although token invalidation typically happens client-side by simply deleting the token, server-side logout mechanisms can also be employed if needed.

Protected Routes: We created middleware to protect certain routes, ensuring that only authenticated users can access them. This middleware checks for the presence of a valid JWT token in the request headers and verifies it. If the token is valid, the request proceeds; otherwise, it is denied.

Password Reset: For password reset functionality, we used Nodemailer to send password reset emails:

Generate Token: When a user requests a password reset, a unique token is generated and stored in the user model along with an expiration time.

Send Email: Nodemailer sends an email containing a link with the reset token to the user's email address.

Verify Token: When the user clicks the link, the token is verified, and if valid, the user can reset their password.

Frontend (ReactJS)

UserContext: We used React's Context API to create a UserContext, which maintains the user's authentication state across the application. This context provides an easy way to access and update user information, ensuring that the user state is consistent and readily available throughout the app.

Axios: For handling HTTP requests, we used Axios. Axios simplifies making API calls to our backend for actions like registration, login, and password reset. We also configured Axios to include the JWT token in the request headers for protected routes, ensuring that the server can authenticate the user for those requests.

Authentication Components:

Login and Registration Forms: These components handle user input and interact with the backend API to authenticate users and manage their accounts.

Protected Routes: We created components that wrap around certain routes to ensure that only authenticated users can access them. These components check the user's authentication state from UserContext and redirect to the login page if the user is not authenticated.

Password Reset:

Request Reset Form: This form allows users to request a password reset by entering their email address. It triggers an API call to generate a reset token and send an email.

Reset Password Form: This form is accessed via the link sent in the password reset email. It allows users to enter a new password, which is then sent to the backend for updating the user’s password after verifying the reset token.

We also used hashing for the passwords in the database, making it more secure and reliable.

Using API:

For the listing of upcoming events and live match information, we utilized the API provided by RapidAPI to consume and extract the necessary information for implementing these features. The API is called API-Sports and exposes data for various sports. While reading the documentation provided by the developers, we realized that each sport had a different API. Due to this, we decided to implement the football (soccer) API first, with the intention of implementing other sports if time allows.

Once we had an idea of how to establish communication between the different components of the project, we followed the steps indicated in the documentation to make API calls. Although the documentation was quite extensive, it didn't provide sufficient relevant information to know exactly how the parameters introduced within the API call would influence the results obtained. Consequently, a large part of the development process involved trial and error until we obtained functional results. While this trial-and-error methodology helped us understand the API's functionality, it was also a disadvantage because our Free Tier account only allowed 100 API calls per day. This slowed down the development process, as we had to use our calls wisely and carefully study the results to make informed decisions and necessary code changes for subsequent calls.

Regarding the implementation and consumption of the API within the project, we added two routes in the backend (corresponding to the two completed features). Each route made an API call to a specific endpoint. As an example, we'll describe the implementation for Live Scores:

First, we added a .js file in the backend's routes folder. This file made an API call using the request module. The URL called was 'https://v3.football.api-sports.io/fixtures', with 'fixtures' being the endpoint in this case. To obtain live matches, the documentation indicated that adding the parameter qs: {live: 'all'} would retrieve currently ongoing matches. Once the call was processed and results returned, the JSON was exposed on the 'api/fixtures/live' route. This JSON was then read by a React component, which extracted the important information and displayed it to the user in a more user-friendly manner.

The same procedure was followed for Upcoming Events, with the only changes being the endpoint used and the parameter for retrieving a specific number of upcoming events. Additionally, we added variables to manage filters within this feature.

Recommender system:

In the backend, we focused on enhancing the bet analysis functionality. We updated the analyze_betting_patterns function in bet_recommender.py to accurately compare the betChoice (team name) with the match result. This involved parsing the fixture to separate home and away teams and determining wins and losses based on the score. The function get_all_betting_history was used to retrieve all betting data from the MongoDB collection bets, ensuring that the analysis has access to the complete betting history. The generate_recommendations function utilized these analyzed betting patterns to create a prompt for the Gemini API, generating content recommendations based on the betting history.

To handle recommendation requests, we set up a Flask server in app.py. This server includes an endpoint /recommend that triggers the recommendation generation process and returns the result. We enabled CORS in Flask to allow requests from the frontend, ensuring seamless communication between the backend and the frontend. The Flask server processes the recommendation logic and communicates with the Gemini API to fetch the recommendations.

In the frontend, we integrated the recommendation functionality by adding the getRecommendations function in api.js. This function sends a request to the Flask server's /recommend endpoint and retrieves the recommendations. We updated BetsPage.jsx to include a button for fetching recommendations, which displays the results in a styled div. This ensures that the fetched recommendations are presented in a user-friendly manner.

To enhance the readability of the recommendation results, we added a .recommendation-result CSS class. This class styles the recommendation results with a dark background and white text, ensuring that the content is easily visible and readable. This styling adjustment helps in presenting the recommendations clearly to the users.

By combining these components, we successfully implemented a recommendation system that analyzes historical betting data, generates recommendations using the Gemini API, and displays the results in the frontend. This system leverages existing data and integrates it with an advanced language model to provide valuable betting insights to the users.

Testing: Due to time constraints and team management issues, we couldn’t implement any type of test in our application.