Front End Documentation

High-Level Structure

Route-Store-Components Structure: The "route-store-components" folder structure is a common pattern for organizing React projects with separate folders for routes, stores (state management), and components. This structure promotes modularity and separation of concerns.

- **routes**: This folder contains files related to routing in your application. It may include files for defining routes, handling navigation, and configuring the router. You might use a library like React Router to manage your routes.
- **store**: This folder holds the files related to state management using Zustand or other state management libraries. It may include store files where you define your application's state, actions, and selectors. Additionally, you may have separate files for each store or module within your application.
- **components**: The "components" folder contains reusable components that can be shared across different parts of your application. You can organize components based on their functionality or domain. Each component may have its own folder with the component file, styles file, and any related files.

Vite: Vite serves as the build tool and development server for your React project. It provides fast development server capabilities with hot module replacement (HMR) and optimized build output. With Vite, you can quickly scaffold and run your React application.

State Management with Zustand: Zustand is used for state management in your application. You can define your stores and related files within the "store" folder. Each store might include a file for state definition, actions, selectors, and any other necessary logic. Zustand provides hooks that allow you to access and update the state within your components.

Styling with Styled Components: Styled-Components is utilized for styling your components. Each component within the "components" folder may have its own styles file where you define the CSS styles using Styled Components' tagged template literals. This allows you to encapsulate styles within each component.

Functional Components and React Hooks: Your components, whether they are in the "routes" or "components" folder, are built as functional components using React hooks. You can use hooks like useState, useEffect, and others within these functional components to manage state, handle side effects, and more.

Pages Implemented

Homepage: Developed the main landing page with sliders, banners, and interactive elements.

Transaction Page: Implemented financial transaction functionality with forms and transaction history.

Tournaments Page: Created a page to display and manage ongoing/upcoming tournaments, including registration and participation features.

Wagers Page: Enabled users to place bets on events or matches with real-time odds updates.

Privacy Page: Provided information about privacy policies and data handling practices.

Rules Pages: Developed pages explaining guidelines and regulations for different aspects of the application.

Support Page: Designed a page for users to seek assistance or contact customer support.

Games Rule Page: Created pages with detailed rules and instructions for each game available in the application.

User Page: Developed a personalized area for users with user-specific information, settings, and preferences.

Admin Page: Implemented an administrative page with features for managing user accounts, content moderation, and analytics. Not implemented fully

Referee Page: Created a page for referees or officials involved in managing tournaments or matches. Not implemented fully

Shop Page: Implemented a page where users can browse and purchase items or services.

Brackets Page: Not implemented fully

Dispute Page: Not implemented fully

Ladders Page: Not implemented fully

Routes

In our application, we used React Router Dom to handle routing. React Router Dom is a popular library for routing in React applications. It provides a declarative way to define routes and render components based on the current URL.

With React Router Dom, we were able to create a single-page application where the content changes dynamically based on the route. This allows us to provide a smooth and seamless user experience without requiring a full page reload for each navigation using lazy loading.

Promises

In a React web app, asynchronous operations are commonly handled using the async/await syntax and try/catch blocks. The async/await syntax allows you to handle promises in a more readable and concise manner. By marking a function as async, you can use the await keyword to pause the function's execution until the promise is resolved or rejected. Using try/catch blocks helps to catch any errors that may occur during the asynchronous operations, preventing unhandled errors from being thrown. This approach provides better error handling and ensures that the application can gracefully handle asynchronous tasks.

Patterns

The container/presentational components pattern is widely adopted in React projects as an architectural approach. This pattern divides components into two distinct categories: container components and presentational components. Container components are responsible for managing state, data fetching, and interacting with other parts of the application. They encapsulate the logic and provide data to the presentational components. On the other hand, presentational components focus solely on rendering the user interface and receiving data and callbacks from the container components. By employing this pattern, developers achieve better separation of concerns, making components more modular, reusable, and easier to test and maintain.

While React itself doesn't enforce any specific architectural methodologies, the container/presentational components pattern aligns well with the principles of component-based development and functional programming. This pattern allows developers to create more scalable and maintainable codebases by decoupling the logic from the presentation layer. Additionally, it promotes reusability of presentational components and improves testability by isolating business logic in container components. As a result, the container/presentational components pattern has become a recommended practice for structuring React applications, enabling developers to build complex UIs while maintaining code clarity and maintainability.

Responsiveness

In order to ensure responsiveness and cross-browser compatibility in our project, we employed a combination of styled-components and media queries. Styled-components is a popular CSS-in-JS library that allowed us to create reusable and dynamic styles for our components. By encapsulating styles within the component itself, we were able to easily modify and adapt the styles based on different screen sizes and devices.

Media queries played a crucial role in making our application responsive across various devices and browsers. We used media queries to define specific CSS rules and styles that should be applied based on the screen size and resolution. This allowed us to create a fluid and adaptable layout that seamlessly adjusted to different viewport sizes. By testing and fine-tuning our media queries across multiple browsers, we ensured that our application looked and functioned consistently across different browser environments.

Overall, the combination of styled-components and media queries enabled us to handle responsiveness and cross-browser compatibility effectively. These tools allowed us to create responsive designs that seamlessly adjusted to different screen sizes and resolutions. By considering various browsers during the development and testing phase, we ensured that our application was compatible and provided a consistent user experience across different browser platforms.

Challenges

One challenging part of the project was dealing with the brackets. The way we imagined the brackets is similar to a reverse heap tree. By using the formulas following formulas we were able to get the winning teams that moved up a bracket:

Parent: floor((idx - 1) / 2)

Right: (idx * 2) + 1

Left: (idx * 2) + 2

Final Notes

- Moving forward, using Redux Toolkit is a superior idea to using Zustand.
- The project can easily be hosted since the GitHub repo is ready.
- Client-side rendering is used in this project.