Report

Mário Ferreira

2023-01-19

Contents

Report

The chess project was developed with a backend written in Elixir Phoenix and a Postgresql database. The frontend was built using ReactJS and TypeScript.

JSON REST API was used to allow users to communicate with the backend and retrieve information from the database. The chess games were implemented using a websocket connection, allowing real-time updates of the games in progress.

The advantages of using Elixir Phoenix for the backend are its high performance, its ease of handling parallel processes and its ability to handle a large number of concurrent connections. The combination with Postgresql also offers high reliability and advanced data management features.

By using ReactJS and TypeScript for the frontend, we benefited from a clear and modular code structure, as well as real-time type checking to avoid coding errors.

Features

Implemented

- ☒ **User login**: Users can create an account and login to the application to access game features. Authentication is done with a JWT token. The frontend must, therefore, send the received token after the login in the headers of its HTTP requests.
- ☒ **Game against a bot**: Users can choose to play against a bot using Stockfish artificial intelligence to simulate an opponent. They can also choose the bot's level for a personalized game experience.
- ☒ **Real-time move updates**: Moves made during games are updated in real time on the app so users can keep track of games in progress.
- ☒ **Game storage**: All games are stored in the database for users to analyze once they are completed. This allows users to review games to improve their technique and play.

To be implemented

- ☐ **Real-time play against other players**: Users can also play against other players online in real time. This feature is yet to be implemented but will allow users to compete in real time. Users will be able to define whether the game is ranked or not
- ☐ **Game analysis**: Users can analyze games using built-in analysis tools to improve their game. This feature is yet to be implemented but will allow users to review games with tools to improve their playing technique.
- ☐ **Interface customization**: Users can access a settings page where they can change the theme of the pieces and the chessboard (frontend implemented, backend not) and set the duration of the games.
- ☐ **Statistics**: Users can access a dedicated dashboard to view graphs and maps generated based on their game history. These charts and maps are designed to provide a clear and informative visualization of the user's performance data.

Technologies

Frontend

- Forkmik: For form validation, it provides an efficient interface for managing states and form validators to facilitate the development of login and registration forms.
- AnimeJS/React framer morion: To add dynamics to the application, they allow you to create fluid and responsive animations for a more enjoyable user experience.
- Flowbite: To create a modern and responsive user interface, it offers a large number of predefined components to quickly build a modern user interface.
- ChartJS: To create charts for chess game analysis, it offers a wide variety of chart types to visualize data in a clear and informative way.
- React ChessGround: For creating the game board, it offers a simple interface to create an interactive chess board with advanced features such as move scoring and legal move validation.
- React auth-kit: For authentication and protected routes, it provides a simple interface to manage authentication states and protected route redirections.
- Phoenix (JS module): For websocket connection for real time chess games, it provides a simple interface to manage websocket connections and update games in real time.

Backend

- Phoenix: This Elixir-based web framework was used to handle HTTP and Websocket communications through its GenServer and Cowboy protocol stack. It provides a simple interface to manage requests and responses, as well as real-time updates via websocket connections.
- PostgreSQL: This relational database has been used to store information about chess games, users, settings and statistics using tables and views to structure the data. It offers advanced features such as table inheritance via child tables and different data types at column level via extended data type support.
- Stockfish: This chess bot was used to simulate an opponent for users playing against a bot. It uses the Universal Chess Interface (UCI) to communicate with the application to provide information about possible moves and positional evaluations. It is based on an analysis of the game tree to determine the most effective moves.

Problems encountered

Different types of games

During the design of this project, one of the main challenges encountered was the management of the different types of chess games. A chess game can be of two types: player against player (pvp) or player against computer (pvc). Most of the data fields are shared between the two types of games, so I considered implementing a table-level inheritance to manage the data specific to each type of game.

After analyzing the pros and cons of each approach, I decided to store both types of parties in a single table. The fields specific to each type, will be stored in a blob/json field. This choice simplified the database structure and made queries easier, but required additional data management in the code to handle the fields specific to each party type.

Websocket connection

One of the problems encountered when implementing the websocket connection was related to the management of React components. When the chessboard component is displayed, a websocket connection is established to manage the movements of the pieces in real time. The problem was that the component was reloaded multiple times (the useEffect hook was called multiple times) in some situations, resulting in multiple websocket connections being created.

This caused conflicts when animating the parts, as each movement triggered a broadcast on all connections, resulting in redundant animations and synchronization issues. To solve this problem, I had to implement controls to ensure that only one websocket connection is established for each component on the board, and to properly handle the termination of the connection when the component is taken down.

Authentication

Without using any libraries, I had to store the JWT token in the localstorage and manually manage the authentication status of the user in a context to share it with all the components of the application.

However, I encountered context sharing issues between components, which made authentication management complex and error-prone.

To solve this problem, I decided to use the 'react-auth-kit' library which offers simplified authentication management, with features such as JWT token management and authentication status checking, as well as protected routes. This has greatly simplified authentication management in the application.

Timer

Using the setInterval JavaScript function in React can cause problems with hooks, specifically the useEffect hook.

One issue that can arise is that when the component is re-rendered, a new setInterval is created, and the old one is not cleared. This can lead to multiple intervals running at the same time, causing unexpected behavior in the component.

Another issue is that when the component is unmounted, the setInterval callback will continue to execute, even though the component is no longer on the page. This can lead to memory leaks and other performance issues.

To mitigate these issues, it is recommended to use the useEffect hook in combination with the setInterval function to clear the interval when the component is unmounted or the delay changes. Additionally, we can use useRef hook to persist the callback function across re-rendering, so that we don't create new function reference with every re-render.

If you want to know more about this issue, take a look at this article.