Ca' Foscari
University
of Venice

Corso di Laurea triennale
in Scienze e Tecnologie Informatiche

Final Thesis

A web application for the game of Chess

Supervisor
Prof. Pietro Ferrara

Graduand
Giovanni Stevanato
Matriculation Number 880077

Academic Year
2022 / 2023

ACKNOWLEDGMENTS

I would like to express my heartfelt gratitude to several people who have been helpful in the completion of this thesis.

Firstly, I would like to express my sincere appreciation to my esteemed Professor Pietro Ferrara for providing me with the opportunity to work on this project. I am truly grateful for their guidance, support, and invaluable insights throughout the entire research process. Their expert feedback and mentorship have played a pivotal role in shaping and refining this work, enabling me to successfully complete it.

I would also like to extend my gratitude to my fellow classmate and friend, Dr. Andrea D'Attero. Collaborating with them on this project has been an enriching and fulfilling experience. Their unwavering passion, dedication, and collaborative spirit have been a constant source of inspiration throughout our work together. I am truly grateful for the opportunity to work alongside them, and their contributions have significantly enhanced the quality of our project.

Furthermore, I would like to express my deepest appreciation to my loving parents Mauro and Martina. Their constant encouragement, belief in my abilities, and unconditional support have propelled me forward during my university journey. Their unwavering presence and encouragement have been a constant source of motivation, and I am truly grateful for their love and support.

ABSTRACT

This thesis presents the research and development process for a university project focused on creating a web platform dedicated to playing and studying chess (source code available at https://github.com/datteroandrea/chess).

The goal was to develop a comprehensive application that seamlessly integrates chess game mechanics, chess engine interactions, online multiplayer functionalities and educational features.

The resulting platform provides an immersive and engaging experience for chess enthusiasts, facilitating both gameplay and learning. The implementation utilized cutting-edge technologies and incorporated innovative features to deliver a user-friendly and interactive chess platform.

INDEX

1 Introduction

1.1 The game of Chess

Chess is a two-player strategy board game played on a square board with 64 squares arranged in an 8x8 grid. Each player begins the game with 16 pieces: one king, one queen, two rooks, two knights, two bishops, and eight pawns.

The objective of the game is to checkmate the opponent's king, which means placing it under attack (in "check") and there being no legal move to escape the attack. The game can also be won if the opponent resigns or if they run out of time.

Each piece moves in a different way, with the king being the most important piece as it must be protected at all times. The queen is the most powerful piece, able to move any number of squares diagonally, horizontally, or vertically. The rooks can move any number of squares horizontally or vertically, while the bishops can move any number of squares diagonally. The knights move in an L-shape, two squares in one direction and then one square perpendicular to that direction.

Pawns move forward one or two squares on their first move, and then one square forward on subsequent moves. They capture diagonally, one square forward and one to the left or right.

There are also special moves such as castling (moving the king and rook simultaneously) and en passant (capturing a pawn that has just moved two squares forward).

Chess is a game of skill, strategy, and tactics, and is considered one of the oldest and most popular board games in the world.

1.2 How computers revolutionized Chess

With the advent of computers and the development of Chess engines and platforms the way people study and play chess has been revolutionized by providing access to a wealth of information, tools, and resources that were previously unavailable or a lot harder to obtain. This has transformed the game from a relatively closed and exclusive community to a more open and democratic one, where anyone can learn and improve their skills.

One of the most significant benefits of chess technologies is the ability to analyze games and positions in real-time. With the help of chess engines, players can quickly identify mistakes, missed opportunities, and better moves, helping them to improve their game and gain a better understanding of chess. This is particularly valuable for beginners, who may not have access to experienced coaches or mentors. Chess engines can also generate tactics puzzles and exercises, helping players to improve their calculation skills and pattern recognition, by providing immediate feedback and letting players know if they have solved the puzzle correctly or not.

Another important feature that chess technologies have brought to the table is the ability to create and manage chess databases. These databases allows to store, organize, and search large collections of chess games. This information can be used for studying and analyzing games played by strong players, researching opponents and developing strategies for upcoming matches or simply for studying specific openings and variations.

Chess platforms also provide an excellent environment for people to play against each other and connect with other chess enthusiasts from all around the world, at any time of the day and without the need for specialized equipment or dedicated spaces.

Nowadays these technologies have become an essential tool for anyone who wants to learn, play, or improve their skills in the game of chess.

1.3 Our idea for the platform

When we decided to take on this project and started developing our own chess platform in the form of a web application we wanted to include as many useful tools as possible to make the app more engaging and attractive to users. Some of the key features we wanted for the platform were:

- Authentication, to identify users and apply the ELO rating system.
- PVP, to provide a great environment for chess players to face each others.
- PVE, to be able to play against the chess engine at various levels of difficulty.
- Chess engine analysis, with a solid user interface that could be as helpful and as intuitive as possible, and that also allows to analyze an entire game and quickly identify the key points of the game.
- Chess rooms, with the addition of webcams and microphones for educational purposes such as chess lessons offered by a chess instructor or for chess enthusiasts to hang out and talk over a chessboard.

2 The background

When developing our chess platform, we conducted extensive research on the current state of existing chess technologies, including websites, engines, interfaces, libraries, and webcam/microphone integrations. We then carefully evaluated each of these components to determine which ones were most suitable for integration into our platform.

2.1 Existing platforms

There are many existing chess platforms that already provide a wide range of features and tools for players of all levels, some are completely free, others have premium features that necessitate a monthly subscription fee from users, some are open source and some are not.

Throughout the project development, we conducted comprehensive research and drew inspiration from these platforms. This diligent study enabled us to incorporate successfully already available features and design others that might be innovative.

2.1.1 chess.com

Chess.com is a widely used website and platform for playing and learning chess, providing a range of features to its users. It was first launched in 2005 and has since become the most popular destination for chess players of all levels.

One of the key features of Chess.com is its online chess playing platform, which allows users to play against opponents from around the world in real-time. The site offers various game modes, including standard time controls, blitz chess, and bullet chess, as well as the ability to create custom games with specific rule sets.

In addition to playing chess, Chess.com also provides a range of tools and resources for learning the game. This includes chess puzzles and tactics training, as well as instructional articles and videos from expert chess players and coaches.

Chess.com also offers a number of social features, including the ability to create and join clubs, participate in forums and discussions, and connect with other players through private messaging and chat. Users can also compete in tournaments and leagues, and track their performance and progress through the site's comprehensive rating and statistics system.

Finally, Chess.com offers various premium features to its users, which can be accessed through a monthly subscription fee. These features include access to advanced training tools, personalized training plans, and the ability to compete in exclusive tournaments and events.

Due to the closed nature of the source code of Chess.com it is not easy to know exactly what technologies are used by the website. However, Based on

observations of Chess.com's behavior and statements from the company itself, it is likely that the website uses a LAMP (Linux, Apache, MySQL, PHP) stack for its backend, and it also probably utilizes a javascript frontend framework to streamline development and make mobile deployment easier.

2.1.2 lichess.org

Lichess.org is a popular free and open-source online chess platform that offers various features to its users. It was launched in 2010 and has since gained a significant following among chess enthusiasts.

Over the years, it has undergone several updates and improvements, adding new features and functionality to its platform. Today, Lichess.org includes many of the same features that Chess.com offers.

One notable feature of Lichess.org is its commitment to open-source software and free access for all users. The platform does not rely on advertising or premium subscriptions, instead relying on donations from its users to cover the costs of development and maintenance.

The open-source nature of Lichess.org let us know that the website is mostly written in Scala, it has a MongoDB database and Typescript is widely used for the frontend.

2.2 Chess Engines

A chess engine is a software program that is designed to play chess automatically, without human intervention. These engines use complex algorithms and search techniques to evaluate chess positions and determine the best moves to make.

Chess engines can be used for various purposes, such as analyzing game positions, playing games against other engines or human players, or assisting in chess training and coaching. They are commonly used by chess players of all levels, from beginners to grandmasters, to improve their skills and study the game.

In the early days of chess engines, they required significant computational power and memory to operate efficiently, which meant they could only run on high-end computers or specialized hardware. However, with advances in technology, the performance of computers and web browsers has improved significantly, making it possible for modern chess engines to run smoothly in a common web browser or even in a mobile application.

2.2.1 Historical Overview:

Chess engines have a long and fascinating history, stretching back to the early days of computer science and artificial intelligence. Here is a brief historical overview of the development of chess engines:

- **1950s - 1960s**: The first attempts to create chess-playing computers were made in the 1950s and 1960s, with early programs like NSS (1951) and Kotok-McCarthy (1956) using brute-force search techniques to evaluate chess positions. However, these programs were still far from competitive with human players.

- **1970s - 1980s**: The development of computer hardware and software led to significant improvements in the performance of chess engines in the 1970s and 1980s. Programs like Chess 4.0 (1975) and Cray Blitz (1983) used specialized hardware and advanced algorithms to become competitive with human players.

- **1990s - 2000s**: The 1990s and 2000s saw the rise of software-based chess engines, which could run on standard personal computers. Programs like Fritz (1991) and Junior (1993) used sophisticated algorithms and machine learning techniques to improve their performance, and they became increasingly popular among both casual and professional players.

- **2010s - Present**: The continued advancement of computer hardware and software, as well as the growth of artificial intelligence and machine learning, has led to further improvements in the performance of chess engines. Today, programs like Stockfish (2008) and AlphaZero (2017) use neural networks and other advanced techniques to evaluate chess positions and play at a level that is competitive with the strongest human players.

Overall, the development of chess engines has been a fascinating journey that has pushed the boundaries of computer science and artificial intelligence, and has helped to advance our understanding of the game of chess itself.

2.2.2 Game Complexity:

The game complexity of chess has made it very difficult for developers to create effective chess engines. One of the main challenges is the sheer number of possible positions that can arise during a game.

The estimated number of possible positions in a game of chess is based on what is known as the Shannon number, named after mathematician Claude Shannon who first estimated the number in the 1950s. The Shannon number is an estimate of the total number of possible positions that can arise in a game of chess, assuming optimal play by both players. The number is estimated to be around 10^120, which is an incredibly large number.

To put this number into perspective, consider that there are estimated to be around 10^80 atoms in the observable universe. This means that there are many more possible positions in a game of chess than there are atoms in the universe. Even with modern computing power and algorithms, it is not feasible to evaluate all of these positions within a reasonable amount of time.

To overcome this challenge, chess engines use various techniques to narrow down the search space and evaluate the most promising moves. These techniques include heuristics, pruning algorithms, and evaluation functions based on machine learning and other artificial intelligence techniques.

2.2.3 Engine Architecture:

As chess engines have advanced, their architecture has become more complex, incorporating multiple components to ensure optimal performance. Modern engines rely on sophisticated search algorithms, evaluation functions, opening books, endgame tables, and caching mechanisms to avoid evaluating the same positions multiple times. This combination of techniques enables the engine to analyze the large number of possible moves and positions in a game of chess more efficiently and accurately. Overall, the development of these components has been critical to the improvement of chess engines over time. In this chapter we are going to take a look at some key components of a modern chess engine.

2.2.3.1 Board Representation

One of the primary requirements in the development of a chess engine is the implementation of an effective board representation scheme, as it is fundamental to the engine's ability to analyze and make decisions based on the current state of the game.

One popular way of representing a chess board is through the use of bitboards, which are 64-bit integers that represent the state of each square on the board. Each bit of the integer corresponds to a square on the board and is set to 1 if that square is occupied by a particular piece, and 0 otherwise. This representation allows for efficient and fast operations, such as bitwise AND and OR, to be used when generating moves or evaluating the board.To represent the state of a chess game using bitboards, typically 12 different bitboards are used, one for each piece type and color (pawn, knight, bishop, rook, queen, king for both white and black). Each bitboard represents the locations of the pieces of a particular type and color on the chess board.

Another common method of board representation is through the use of arrays or matrices. This involves creating a two-dimensional array of squares, where each element of the array contains information about the type and color of the piece occupying that square. This approach is more intuitive and easier to understand, but may be slower than using bitboards.

2.2.3.2 Move Generation

Move generation is the component of a chess engine that is responsible for calculating all the legal moves that can be made by each player based on the current state of the board. Chess engines typically use a combination of algorithms and data structures to perform move generation efficiently and accurately.

The process of move generation can be significantly faster if we use bitboards, as they allow for efficient bitwise operations that can quickly identify the location and movement options of each piece on the board, thereby greatly reducing the computational complexity of move generation algorithms

The engine will typically iterate through each piece on the board, examining its potential moves and evaluating whether they are legal based on the current position of other pieces. For example, a pawn can move one or two squares forward on its first move, or one square forward on subsequent moves, but cannot move forward if the square in front of it is occupied by another piece.

2.2.3.3 Search Algorithm

Search algorithms are a crucial component of chess engines because they can reduce the amount of positions to be explored by intelligently selecting the most promising moves to evaluate, allowing the engine to search deeper and faster while still maintaining a high level of accuracy in its evaluations.

Here is a brief overview of some of the common search algorithms used by chess engines:

Minimax algorithm: This algorithm works by exploring all possible moves and their resulting positions up to a certain depth and evaluating each position using an evaluation function. The evaluation function assigns a score to each position based on various factors such as piece mobility, pawn structure, and king safety, and is used to guide the search towards positions that are more likely to lead to a favorable outcome. The minimax algorithm aims to maximize the score of the engine's own position while minimizing the score of the opponent's position. This algorithm can be computationally expensive, as it requires exploring a large number of positions to find the best move.

Alpha-beta pruning: This technique involves keeping track of lower and upper bounds on the possible scores of each position, and eliminating branches of the search tree that fall outside these bounds. This can significantly reduce the number of positions that need to be evaluated, leading to a faster and more efficient search. Alpha-beta pruning is based on the minimax algorithm and can be used to improve its performance. The algorithm assigns a score to each position and uses it to prune branches of the search tree that cannot lead to a better outcome. By pruning these branches, the algorithm can search more deeply into the most promising lines of play, leading to a faster and more accurate evaluation of the best move.

Principal Variation Search: PVS is a refinement of the alpha-beta pruning algorithm that focuses on exploring the principal variation, or the sequence of moves that is most likely to be played. It works by assuming that the best move in a position is the same as the best move in a similar position that has already been evaluated, and then exploring other moves only if they are likely to lead to a better result. This allows PVS to search more deeply into promising lines of play and ignore unimportant variations, leading to a faster and more efficient search. PVS is often used in conjunction with iterative deepening, which involves performing multiple searches with increasing depth, to further improve its accuracy and efficiency.

Monte Carlo Tree Search: MCTS is a probabilistic search algorithm that uses random simulations to explore the search space. The algorithm builds a tree of possible moves and simulations, and selects the move with the highest win rate based on the simulations. This algorithm has proven to be particularly effective in games with high branching factors and a large number of possible moves, such as Go, and has recently been applied to chess with great success.

2.2.3.4 Evaluation Function

Another key component of a chess engine is the evaluation function, that takes a chess position as input and produces a score that represents the strength of the position for the player to move. The score is typically based on various factors, such as material count, pawn structure, king safety, piece activity, and control of the board.

Modern chess engines, such as Stockfish and AlphaZero, use advanced evaluation functions that are typically based on machine learning techniques, such as neural networks or decision trees. These functions are trained on large datasets of chess positions, with each position assigned a score based on the outcome of the game from that position.

The evaluation function is typically used in conjunction with a search algorithm to determine the best move to make in a given position. The search algorithm generates a tree of possible moves and positions, and the evaluation function is used to assess the strength of each position. The search algorithm then selects the move that leads to the position with the highest evaluated score.

In recent years, AlphaZero has revolutionized the field of computer chess by demonstrating the power of deep reinforcement learning to train an evaluation function. AlphaZero's evaluation function is based on a deep neural network that is trained to predict the outcome of the game from a given position. Unlike traditional chess engines, AlphaZero does not rely on a fixed evaluation function or an opening book. Instead, it learns to play chess from scratch by playing against itself and using reinforcement learning to update its neural network.

2.2.3.5 Opening Book

An opening book is a database of chess openings and their associated moves. It is used by chess engines to guide their moves in the opening phase of the game, where players typically follow established patterns and variations to develop their pieces and control the center of the board.

Opening books can be created by humans or generated automatically using computer algorithms that analyze large databases of chess games. The moves in the opening book are ranked based on their popularity and effectiveness in winning or drawing games.

During gameplay, the chess engine consults the opening book to select its moves in the opening phase. This allows the engine to quickly and efficiently navigate through the opening, avoiding mistakes and staying on a strong and established path. Once the game moves beyond the opening, the engine switches to its own search algorithms and evaluation function to determine the best moves.

While opening books can be helpful for guiding the engine's moves in the opening, they do have limitations. The book may not cover all possible variations or lines, and the opponent may deviate from the expected moves. Additionally, relying too heavily on the opening book can make the engine vulnerable to traps or unexpected tactics.

2.2.3.6 Endgame Tables

Just like opening books, endgame tables are a database used by chess engines to improve their performance in the endgame phase of the game. Endgame tables contain pre-calculated optimal moves for all possible endgame positions with a small number of pieces remaining on the board, usually up to 7 pieces.

When the engine detects that the game has reached an endgame position that matches one in the endgame table, it can consult the table to find the optimal move. This can significantly improve the engine's play in the endgame, as it eliminates the need for the engine to spend time calculating the best move for that specific position.

Endgame tables can be created using computer algorithms that analyze all possible endgame positions with a small number of pieces and calculate the optimal moves for each position. However, the size of the table can grow exponentially as the number of pieces on the board increases, making it impractical to store all possible endgame positions.

Overall, opening books and endgame tables are a useful tool for chess engines to navigate the opening phase of the game, but they are just one component of a larger chess engine architecture.

2.2.3.7 Transposition Table

A transposition table is a cache used by chess engines to store previously analyzed positions and their corresponding evaluation values. The purpose of the transposition table is to avoid redundant analysis of the same position during the search process, thereby improving the efficiency of the engine's search algorithm.

When the engine analyzes a position, it first checks the transposition table to see if it has already evaluated the position before. If it finds the position in the table, it retrieves the stored evaluation value instead of re-analyzing the position. This can save significant computational resources, especially in deep search trees where many positions are analyzed multiple times.

The transposition table works by using a hash function to map the chess position to a unique index in the table. In addition to storing evaluation values, the transposition table can also store other information about the position, such as the best move found so far or the depth at which the position was analyzed. This information can be used to improve the efficiency of the search algorithm further.

2.2.4 Machine Learning Techniques

After examining the architecture of a chess engine, it becomes clear that the evaluation function is the most critical component in creating a strong chess engine. While other features, such as search algorithms, transposition tables and opening/endgame books, are important for efficiency, it is the evaluation function that determines the engine's understanding of the game state and the quality of its play.

One of the most commonly used machine learning techniques in chess engines is supervised learning. This involves training the engine on large datasets of expertly annotated game positions and outcomes, allowing it to learn the patterns and relationships between board positions and their expected values. The engine can then use this knowledge to evaluate new positions based on similar patterns it has learned.

Another technique used in chess engines is reinforcement learning, which involves the engine learning through self-play. Engines like AlphaZero use this technique to train their evaluation functions by playing against themselves repeatedly and adjusting their weights based on the outcomes of these games. This allows the engine to learn more complex patterns and strategies, and it can achieve superhuman performance by training on massive amounts of data.

Engines like Stockfish rely more on heuristics to define their own static evaluation function that has been handcrafted over the course of many years by chess grandmasters and computer scientists. These heuristics are rules of thumb that are programmed to evaluate the importance of various features of the game state, such as material balance, pawn structure, and king safety. The problem with heuristics is that they can be difficult to define and can sometimes lead to incorrect or suboptimal decisions in certain scenarios.

However, while it is true that heuristics can have limitations and may not always be perfect, they are still an important tool for solving complex problems and making decisions in many different domains.

2.2.5 Future Directions

Chess engines have come a long way since their inception, and their future looks bright as they continue to evolve and improve. Currently, the best chess engine on the market is still Stockfish, which has consistently demonstrated its ability to play at a superhuman level and run on commodity hardware.

However, the future of chess engines is moving towards deep neural networks, which are being trained and getting stronger and stronger. Neural network-based engines such as AlphaZero have shown remarkable results in recent years, and they use a fundamentally different approach to evaluate chess positions. Instead of relying on pre-defined rules and heuristics, these engines are trained on massive amounts of data and learn to play chess through trial and error.

One of the advantages of deep neural network-based engines is their ability to generalize and adapt to new situations that were not explicitly programmed into their algorithms. They can also discover new patterns and strategies that were previously unknown to human players.

However, one major challenge of neural network-based engines is their computational requirements. They require massive amounts of computing power and resources to train and run effectively, which means they are not as accessible to the average player as Stockfish and other traditional engines.

In summary, while Stockfish remains the best chess engine available for most players due to its ability to run on commodity hardware and play at a superhuman level, the future of chess engines is moving towards deep neural networks, which have the potential to revolutionize the game of chess and bring new insights and strategies to players of all levels.

3. Our Solution

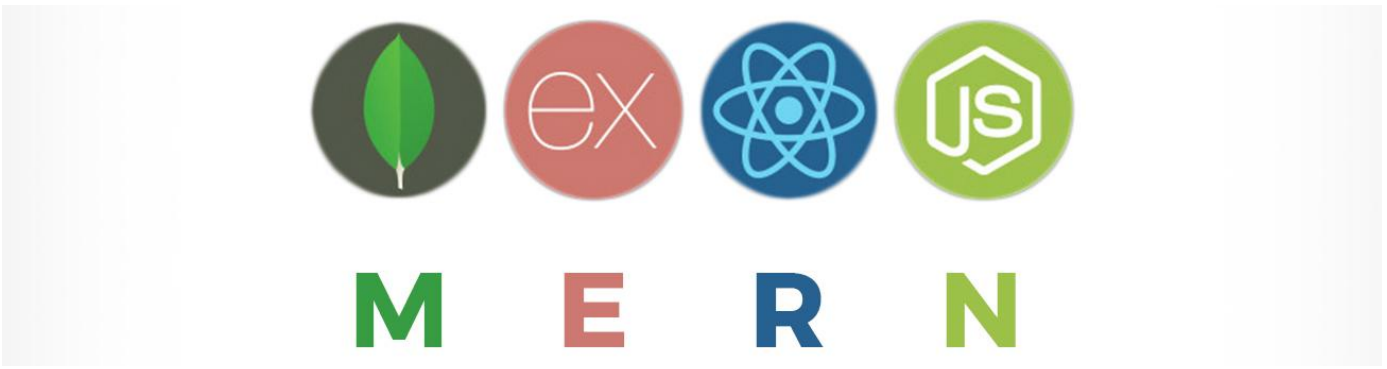
When developing our own chess platform we wanted to ensure using a modern web development stack of technologies such as the MERN stack, which relies on MongoDB, Express, React and Node to ensure performance, scalability and adaptability to different screen sizes. We also wanted to take open source chess technologies such as chess.js and stockfish.js to take advantage of the already existing high quality javascript libraries while creating our own UI and implementing all the desired features such as gameplay, analysis, rooms with webcams and shared boards and much more. For the previously mentioned reasons our engine of choice was stockfish as it is currently the strongest and can easily run on a web browser client-side. After conducting extensive research into chess technologies, we are proud to present our own implementation of a chess platform. The source code for this project is available at https://github.com/datteroandrea/chess.

3.1 System architecture

One of the pivotal aspects of the system architecture that we have implemented is the utilization of the MERN stack, which provides a robust and flexible framework for developing web applications. Furthermore, our decision to utilize the MERN stack is driven by the versatility and ubiquity of JavaScript as the primary programming language for the entire full-stack development process. JavaScript allows for seamless integration and communication between the different layers of the application, enabling us to build a cohesive and efficient system.

3.1.1 The MERN stack

The MERN stack is a popular web development stack that comprises four key technologies: MongoDB, Express.js, React, and Node.js. Each of these technologies plays a specific role in the development process and works together to create a comprehensive and efficient system architecture.

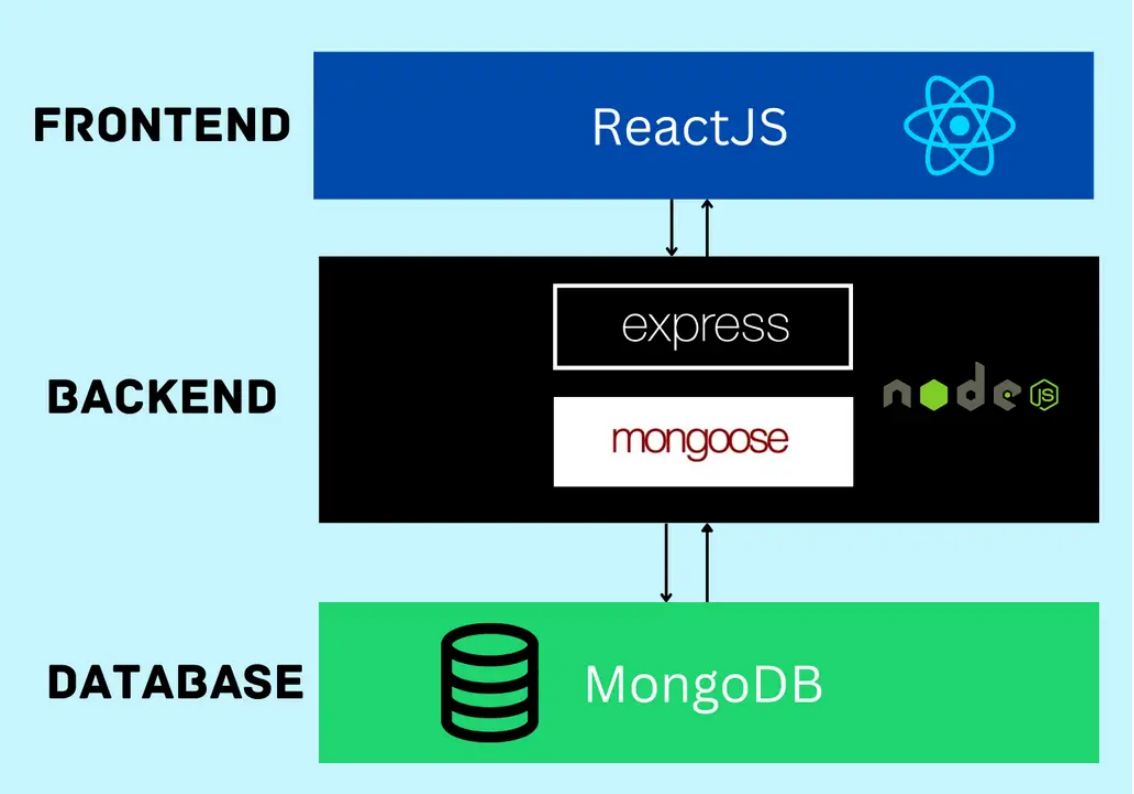


- **MongoDB** is a NoSQL database that stores data in JSON-like documents, which makes it a highly flexible and scalable data storage solution. It is designed to handle large amounts of data and supports sharding and replication, making it suitable for applications with high traffic and data processing needs.

- **Express.js** is a back-end web application framework that runs on top of Node.js. It provides a set of tools and functionalities that allow developers to create and manage web applications with ease. Express.js supports middleware functionality, which allows developers to add additional functionality to the application, such as authentication and validation.

- **React** is a front-end JavaScript library that is used for building user interfaces. It allows developers to create reusable UI components and manage the application's state and user interactions. React uses a virtual DOM, which is a lightweight representation of the actual DOM, to improve performance and efficiency.

- **Node.js** is a server-side JavaScript runtime environment that allows developers to run JavaScript code outside of the browser. It provides a rich set of libraries and tools for building scalable and high-performance web applications.



In a web application built using the MERN stack, these components work together to create a comprehensive and efficient system architecture. React components on the client-side communicate with the server-side Node.js using RESTful APIs and WebSocket connections. The server-side, built using the Express framework, interacts with the MongoDB database using the mongoose library to fetch the necessary data for generating responses to client requests. The combination of these components provides a comprehensive and efficient architecture for building web applications.

3.1.2 JavaScript FullStack development

Another reason why we chose a JavaScript technology stack is that it can be a valuable option for full-stack development projects, particularly for small teams. With a unified language, agile development practices, high performance, large community, and robust ecosystem of libraries and frameworks, it can enable developers to build scalable and reliable web applications efficiently.

- **Unified Language**: By using a single programming language for both the front-end and back-end development, the development team can avoid the need to learn and work with multiple programming languages. This can help maintain code consistency and reduce the need for specialized expertise.

- **Agile Development**: With a JavaScript stack, developers can work on both front-end and back-end tasks, which promotes more efficient collaboration between team members. This can be particularly useful for small teams where developers may need to work on multiple aspects of the application.

- **High Performance**: Node.js, a popular server-side component of many JavaScript stacks, is known for its ability to handle large amounts of data and requests simultaneously, which makes it a great choice for high-performance web applications.

- **Large Community**: JavaScript is one of the most popular programming languages with a large developer community. This means that there are many resources, libraries, and tools available to help with development, reducing development time and cost.

- **Robust Ecosystem**: The JavaScript ecosystem is rich with libraries and frameworks that can be used to build scalable and reliable web applications. For instance, many JavaScript stacks use popular front-end frameworks like React or Angular, which provide efficient and reliable user interface development.

3.2 Frontend technologies

In our project, we utilized various front-end technologies to create a seamless and visually appealing user interface. The project is built using React, a popular and efficient front-end framework that allows for the creation of dynamic and interactive web applications. React was chosen for its ability to handle complex user interfaces, efficient rendering, and component reusability.

To maintain a standardized look and feel throughout the application, we also integrated Bootstrap, a widely used CSS library, which provides an extensive set of pre-built components and styles that can be easily customized. By leveraging the power of Bootstrap, we were able to rapidly prototype and develop UI components, thereby improving the overall development speed and reducing time-to-market.

Additionally, we created custom CSS UI components when necessary, to add a unique touch to the application's visual appearance and enhance the user experience. This approach allowed us to tailor the application to the specific needs and requirements of the project, while maintaining consistency and usability across all elements of the UI.

Together, these front-end technologies allowed us to create a visually appealing and highly functional user interface, which enhances the overall user experience and increases the efficiency of the application.

3.2.1 React

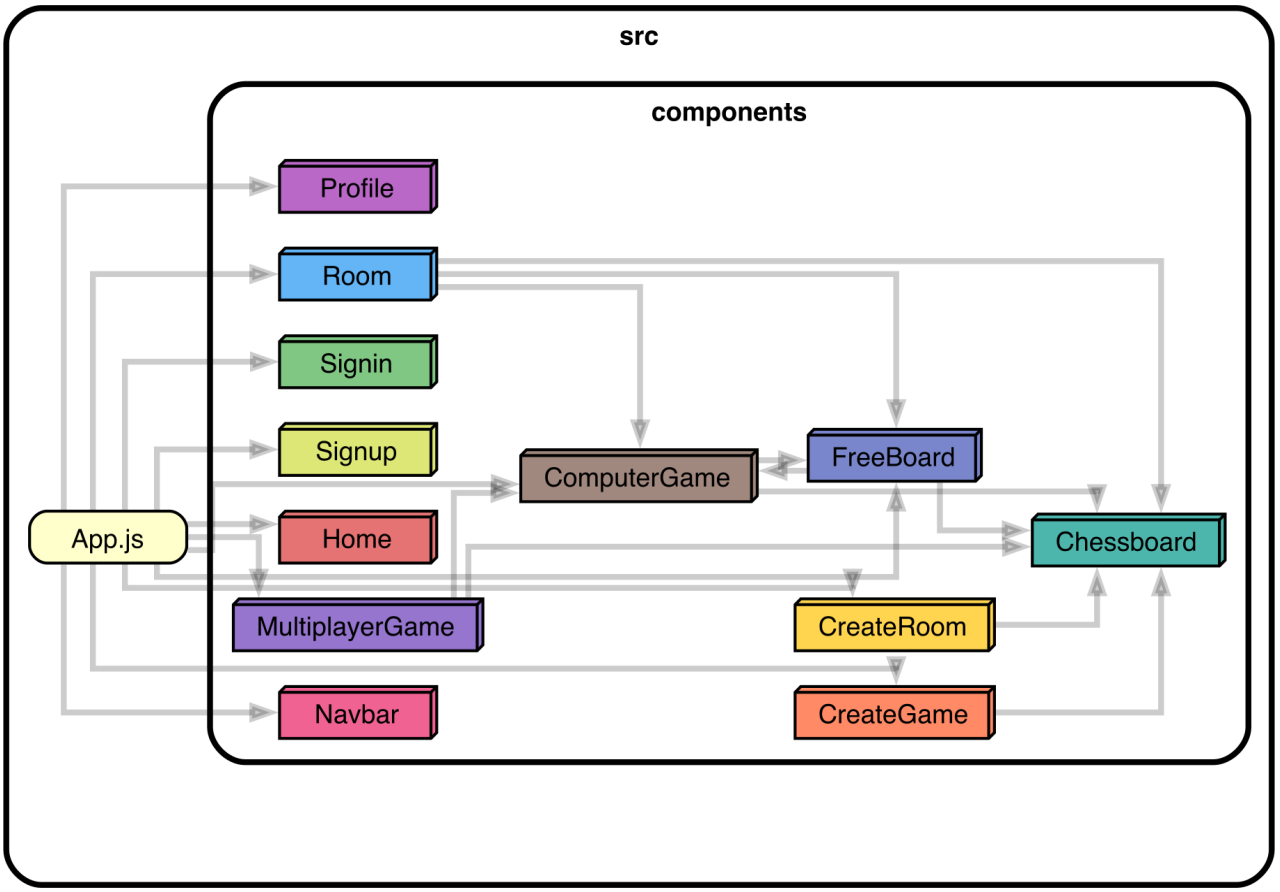
React is a powerful front-end library developed by Facebook that has revolutionized the way web applications are built. It allows developers to create dynamic and interactive user interfaces by efficiently rendering components using a virtual DOM. React's strength lies in its ability to build Single-Page Applications (SPAs), which offer a seamless and fluid user experience.

SPAs, as the name suggests, are web applications that operate within a single HTML page. They dynamically update content without requiring full page reloads, resulting in faster and more responsive applications. With SPAs, users can navigate through different sections and perform actions without experiencing the traditional page flickering or interruptions. This is achieved by asynchronously loading data from the server and updating the content on the same page.

React excels in building SPAs by efficiently managing the virtual DOM. It intelligently determines the minimal changes required to update the actual DOM, resulting in faster rendering times and enhanced performance. React's component-based architecture promotes code reusability, making it easier to develop and maintain complex applications. It also integrates seamlessly with other libraries and frameworks, allowing for additional functionality and flexibility.

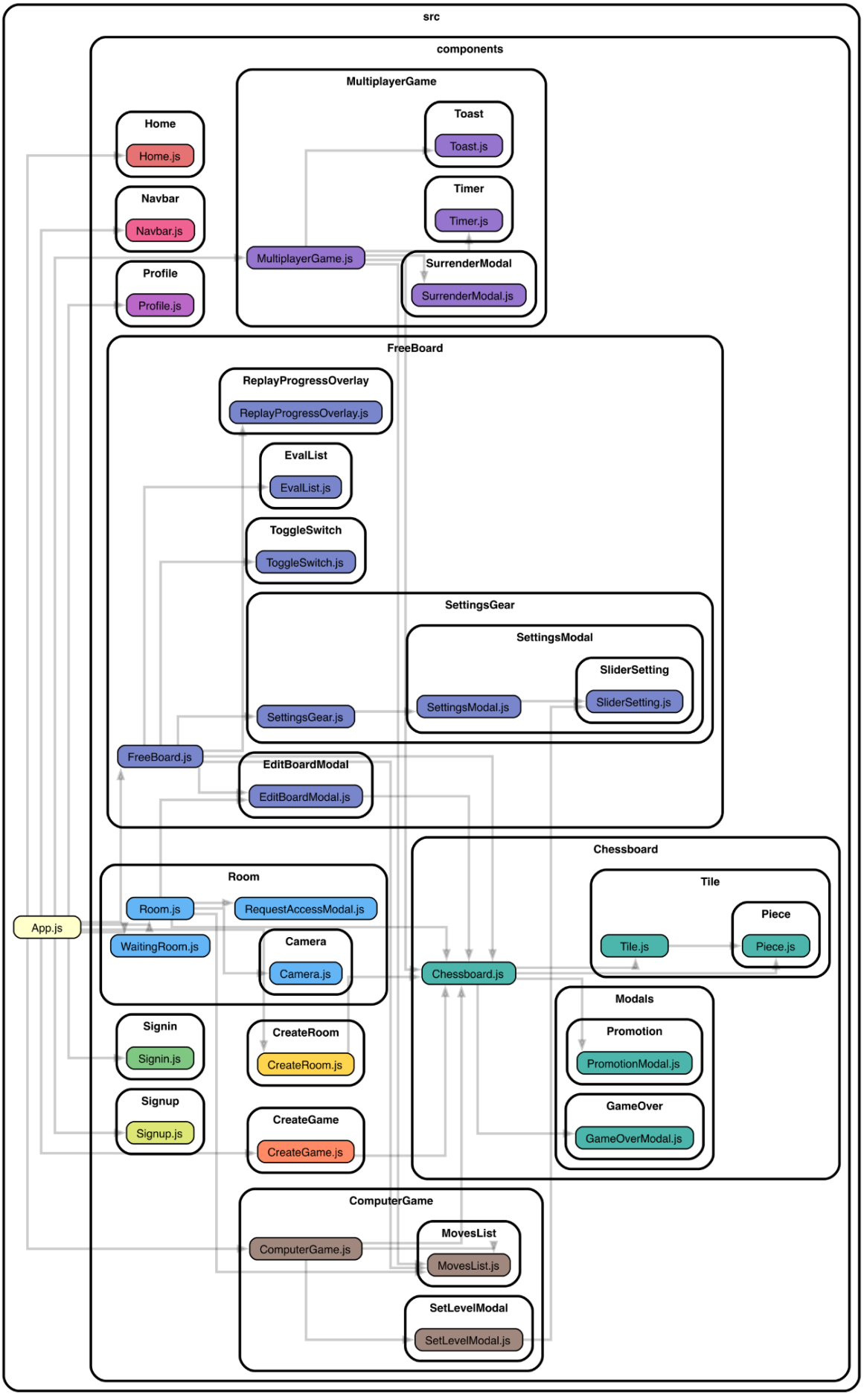
Overall, React's combination of efficient rendering, component reusability, and support for SPAs makes it an excellent web development tool. It enables developers to create highly responsive and interactive applications with smoother navigation, faster load times, and improved user experiences. React has gained widespread adoption in the industry and continues to be a popular choice for building modern web applications.

Next, we will explore the structure and interaction of the components in our project, gaining a deeper understanding of their organization and collaborative dynamics. This analysis will shed light on the relationships and dependencies among the components, providing insights into the overall frontend structure.

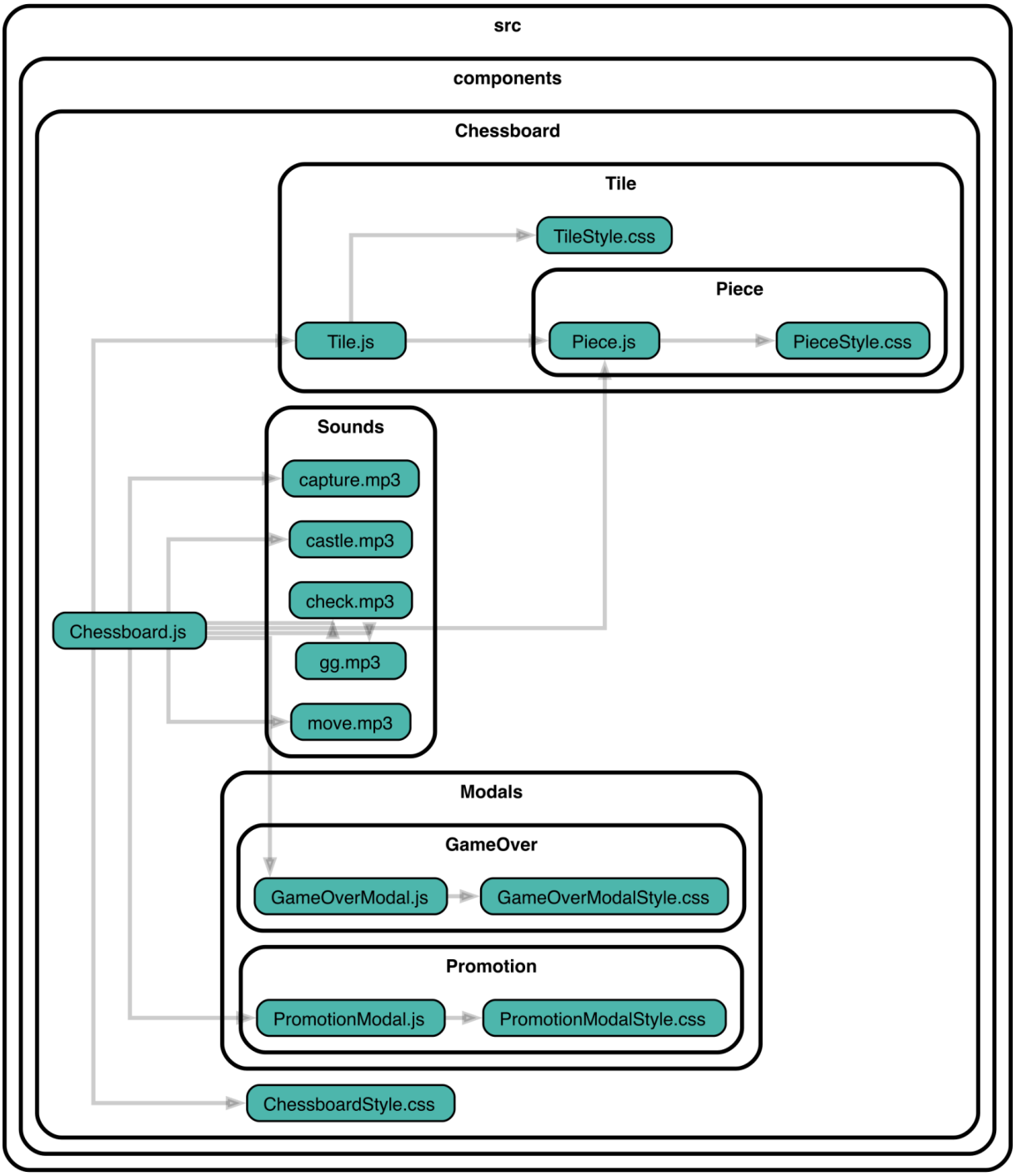


The initial image provides a comprehensive overview of the main components and their relationships. It provides a holistic understanding of the interdependencies and collaborations required to successfully deploy a specific feature in the project. By examining this visual representation, we can identify the key components that need to work together harmoniously to achieve the desired functionality.

- **App.js**: is the entry point and central hub for routing and rendering components in a single-page application
- **Navbar**: is the component responsible for the navbar, it is present in almost every other page and allows users to easily navigate the website.
- **Signin, Signup and Profile**: are components responsible for the user authentication and Administration.
- **Home**: is the homepage of the website.
- **Chessboard**: handles the UI for the Chessboard itself.
- **Freeboard**: is a versatile chessboard component that enables users to freely manipulate pieces, customize the board, load entire games, and utilize Stockfish for engine analysis and study purposes.
- **ComputerGame and MultiplayerGame**: are the components responsible for PvE and PvP.
- **Room**: is the component that handles a room with webcams and microphones with a shared editable chessboard.
- **CreateRoom and CreateGame**: are components used to facilitate the instantiation of Rooms and PvP Games.

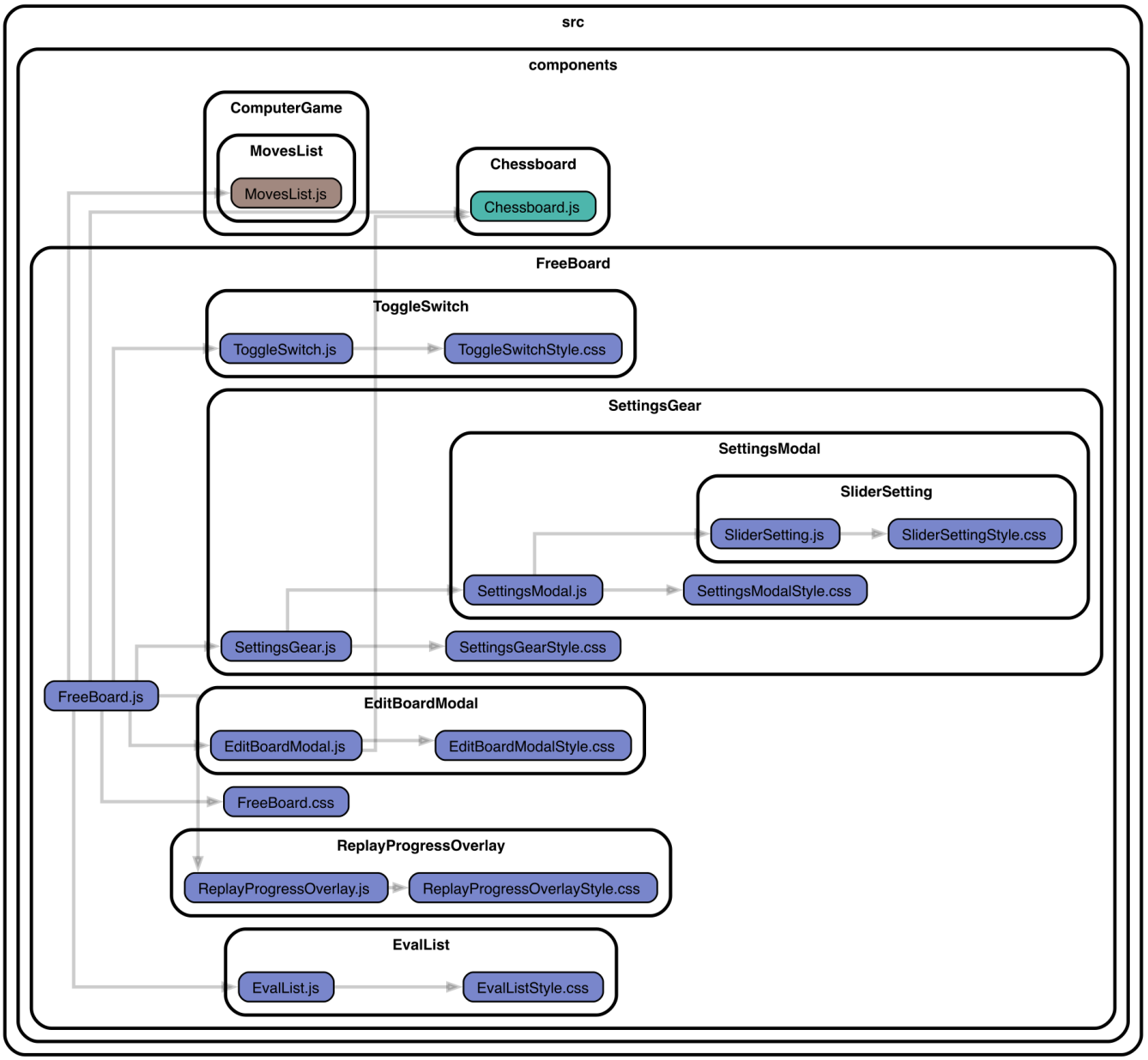


The second image provides us with an overarching view similar to the first one, illustrating the overall structure and relationships among the main components. However, it also includes additional subcomponents, which can potentially introduce complexity and confusion. To mitigate this, it is beneficial to examine each articulated component individually.

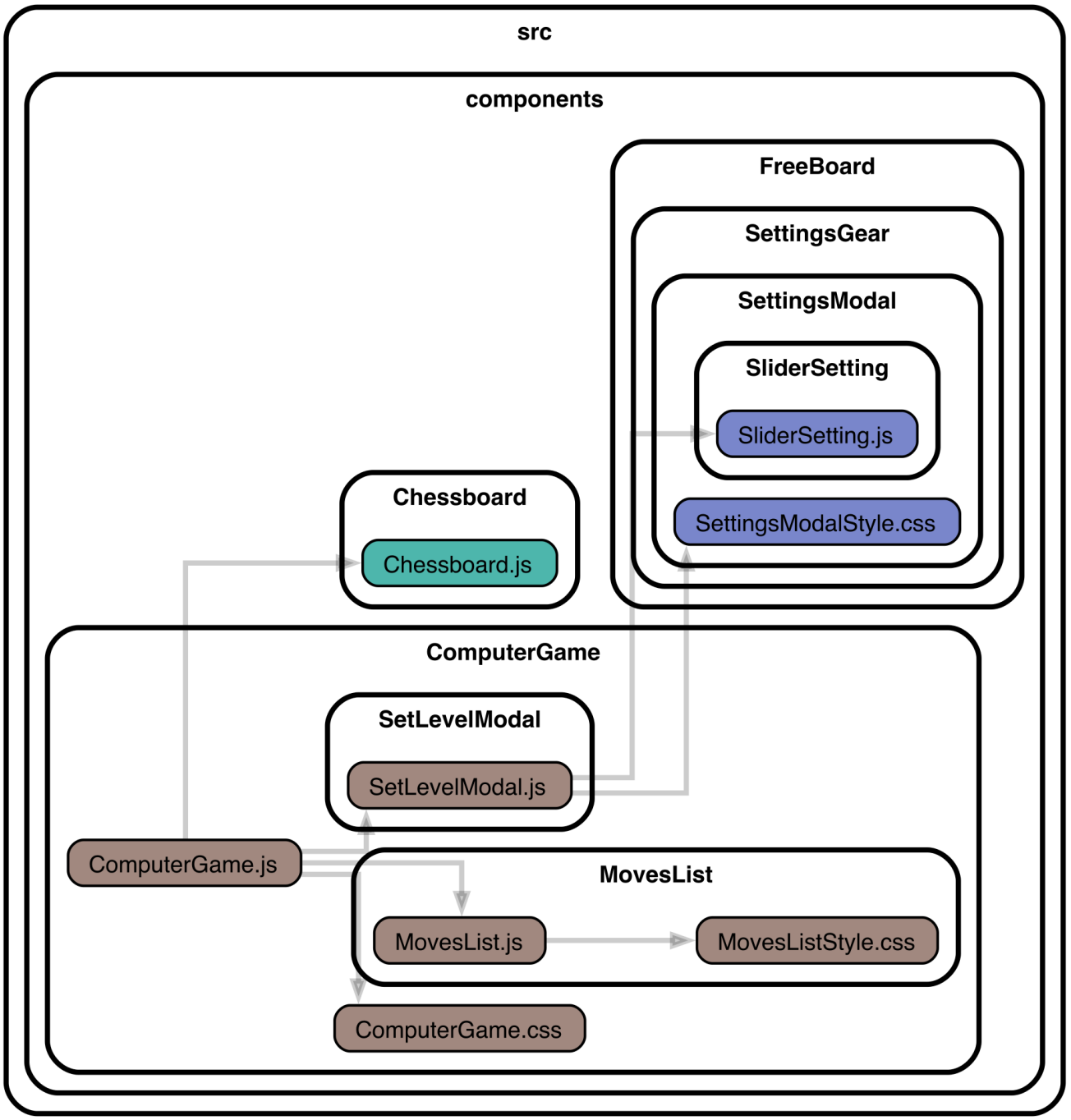


Chessboard is a crucial component within the system architecture. It comprises 64 individual tile components, each capable of containing chess pieces. This component assumes the responsibility of visually rendering the chessboard, handling tasks such as loading pieces onto their respective tiles, facilitating piece movements, managing audio output, handling promotions (when a pawn reaches the last rank), displaying legal moves to the user, and supporting additional features such as drawing arrows or highlighting specific squares on the board.
The encapsulation of this key component enhances its reusability within the system. As evident from the previous images, this chessboard component is utilized by various other components, including Freeboard, ComputerGame, MultiplayerGame and Room. By encapsulating the core functionality and rendering logic of the chessboard within a standalone component, it becomes highly reusable across different parts of the application.

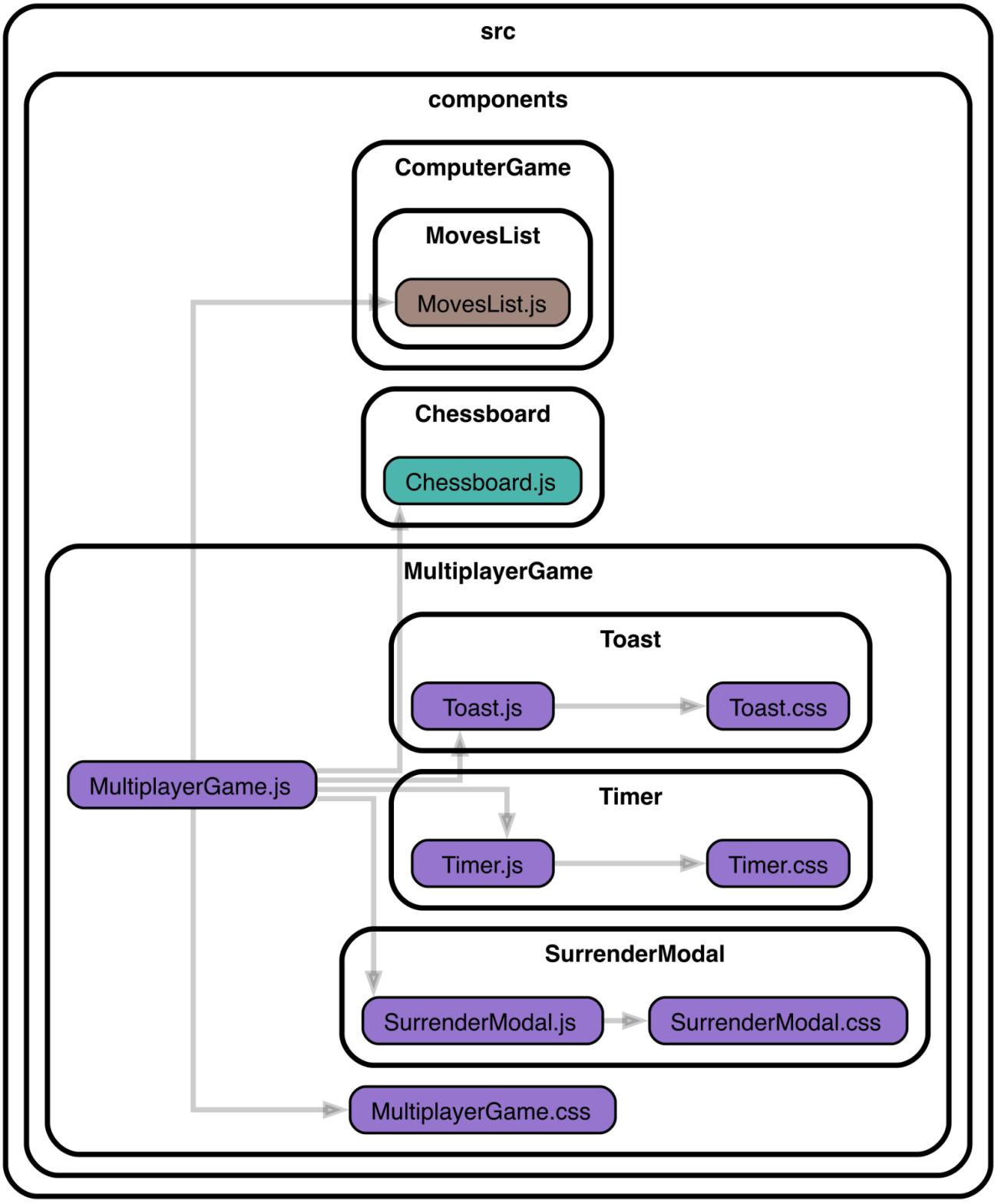
This level of encapsulation allows other components that require a chessboard display to simply import and integrate the chessboard component without having to reimplement the chessboard functionality from scratch. It promotes code reuse, reduces development effort, and ensures consistency in how the chessboard is presented and interacts across different sections of the application and also enables easy maintenance and extensibility. Any updates or enhancements made to the chessboard component can be seamlessly propagated to all the components that rely on it, ensuring uniform behavior and reducing the risk of introducing inconsistencies or bugs.



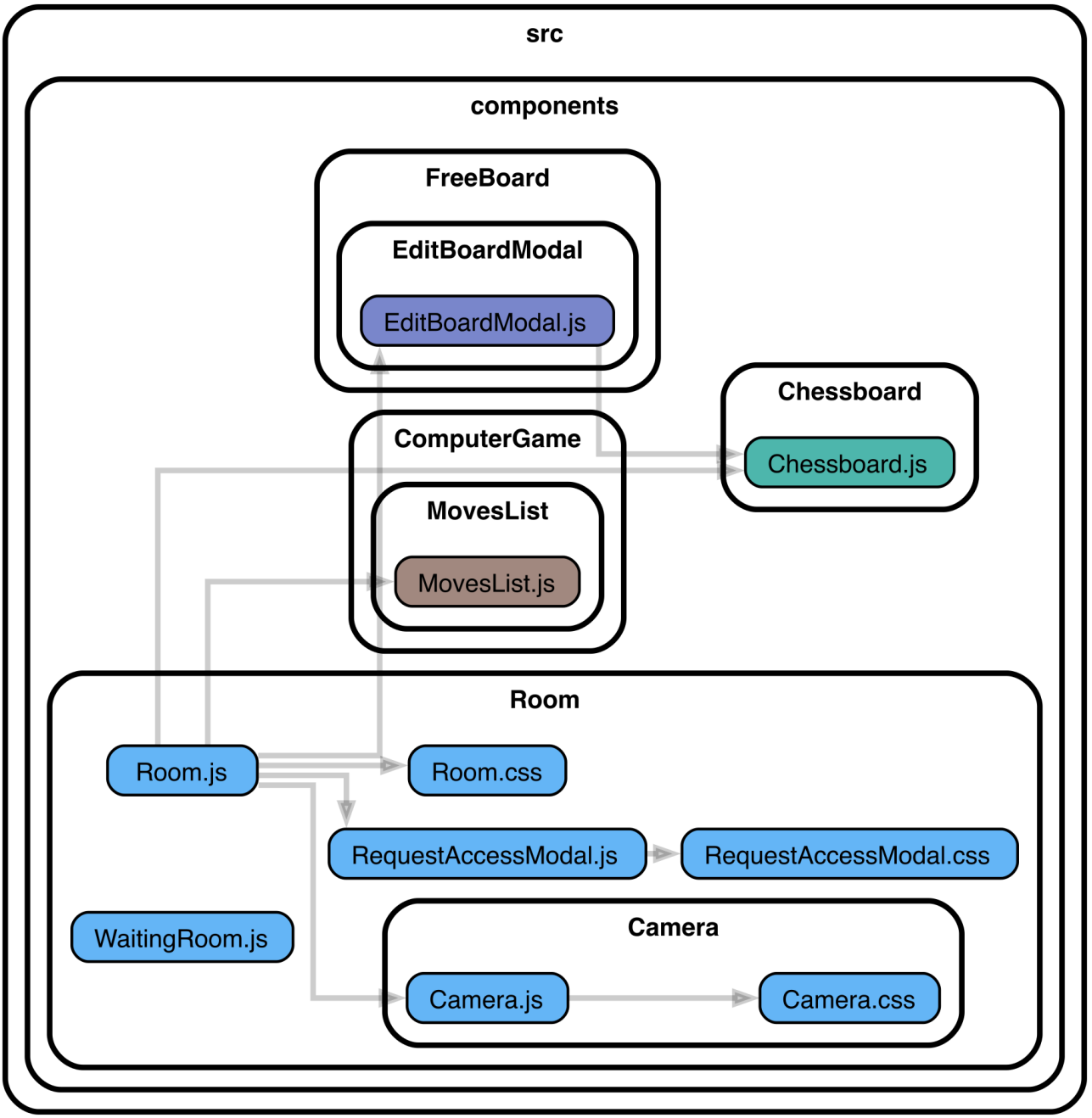
Freeboard is a highly articulated component on the UI side that incorporates several functionalities within a limited space. It utilizes the Chessboard component to visually represent a chessboard, surrounded by a collection of buttons and other views. It enables users to perform various actions on the board, such as moving pieces, setting up positions using FEN strings or by manually setting up pieces with the EditBoardOverlay. Additionally, users can toggle the usage of Stockfish, configure engine parameters, and visualize lines and their evaluations. The component also imports the "MoveList" component, responsible for listing played moves, facilitating navigation through them, and displaying an accuracy level indicator based on the engine evaluation.



ComputerGame is a component that manages player-versus-engine (PvE) gameplay. It utilizes the Chessboard component for visually representing the game board and inherits some of the Stockfish-related features from the FreeBoard component. This component enables players to engage in local matches against the Stockfish engine, allowing them to set the difficulty level on a scale of 1 to 10. It tracks and displays moves on the side, handles the game flow and rules, and offers the ability to load the move list into the FreeBoard component at the end of the game to quickly analyze any mistakes or missed opportunities that occurred during the game. The PvE matches do not comprehend a Timer.



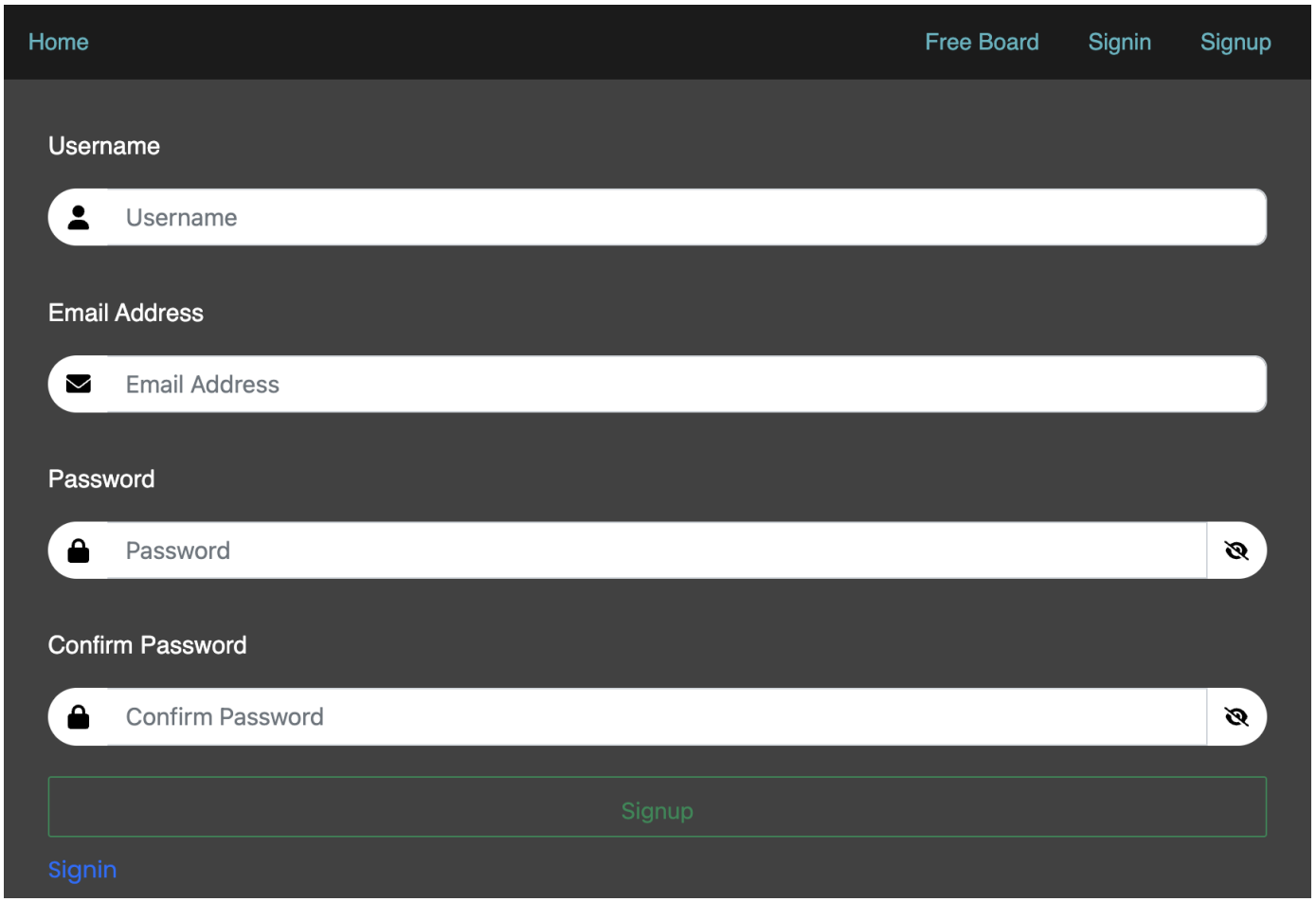
MultiplayerGame is a vital component responsible for facilitating player-versus-player (PvP) gameplay in the game. Once a match is established, this component loads the two players and offers features similar to the "ComputerGame" component such as move list and the possibility to analyze the game at the end. Additionally, "MultiplayerGame" handles the flow of the game and enforces the game rules utilizing websockets in the backend to display moves in real-time, creating an experience similar to a real-world match. The component also incorporates a timer, material count, and buttons for draw offers and surrender.



Room is a component that manages a virtual space where users can interact through webcams and microphones while collaborating on a shared editable chessboard. This component is controlled by an administrator who has special privileges within the room. The admin can manipulate the shared board, mute or kick other users, and grant permission to selected users to edit the board. The room can be either public or private, with the latter requiring admin approval for each user before they can join. The admin also has the ability to initiate a voting process from a specific position on the board. Users can vote on their preferred move, and the results of the vote are displayed at the end of the voting period. These features are thought for online chess lessons or for friends who want to learn and play together. The shared editable chessboard allows for collaborative learning and analysis of positions. The webcams and microphones enable real-time communication, making it possible for participants to interact and discuss moves or strategies. The admin's control over the room, such as muting or kicking users, ensures a managed and organized learning environment. The voting feature adds an interactive element, allowing participants to collectively decide on moves and learn from each other's perspectives.

3.2.2 Bootstrap

For the visual aspect of the website, we utilized a combination of custom handcrafted CSS and the Bootstrap CSS library. While we crafted our own custom CSS to ensure a unique and tailored appearance, Bootstrap proved to be highly valuable for expediting the development process of basic UI components like text boxes, buttons, etc. This was especially beneficial for designing authentication pages and the navigation bar, where Bootstrap's pre-built styles and components provided a solid foundation. By leveraging Bootstrap, we were able to streamline the development of these essential elements, saving time and effort in creating consistent and visually appealing user interfaces. Here is an example showcasing how we utilized Bootstrap to create the registration page for our website.



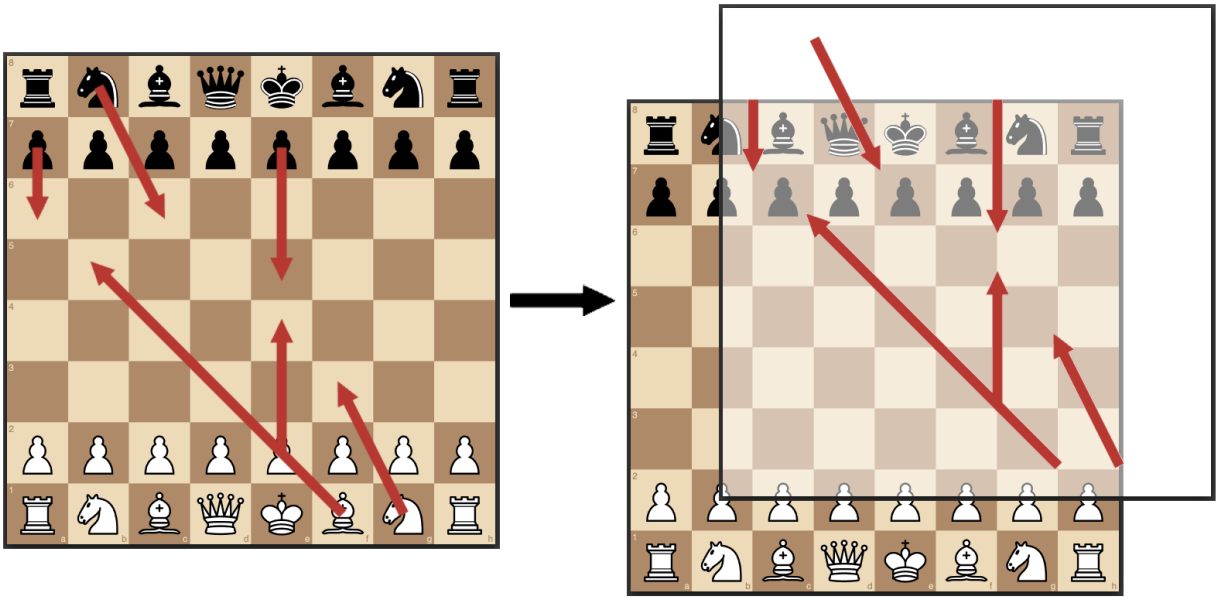
3.2.3 Custom UI

While Bootstrap proved to be an incredibly useful tool, we encountered specific requirements for the feature we wanted to implement that necessitated the creation of custom CSS elements. Meeting the intricate demands of our project and achieving the desired level of specificity and uniqueness was no small task. As a result, we invested a significant amount of time and resources into meticulously crafting custom CSS elements that were tailored to our specific needs. Let's now

explore a few examples of these carefully crafted custom elements that were developed and implemented using a combination of HTML, CSS, and JavaScript

The primary example I'd like to present is the chessboard itself, which was thoughtfully styled to achieve a minimalistic yet functional design. The objective was to provide players with a visually clear and intuitive interface that facilitates a better understanding of the position. The chessboard allows users to select pieces and view their legal moves, facilitating strategic planning. Players can move pieces by either dragging them or clicking first on the piece and then on the target square. Additionally, the chessboard displays the last played move, ensuring players stay updated on the game progression. For enhanced analysis, users can draw arrows or highlight squares by right-clicking and dragging on the board. These interactive features further aid in visualizing and analyzing different scenarios.

One of the most challenging aspects during the development of the chessboard component was implementing the drawing arrow functions. This required overlaying a transparent canvas on top of the chessboard to enable the drawing functionality. However, it was crucial to ensure that the canvas did not block mouse interactions events on the chessboard. Through careful design and programming, we successfully achieved a solution that allowed seamless arrow drawing while preserving the usability of the chessboard, ensuring users could interact with the board without any hindrance.



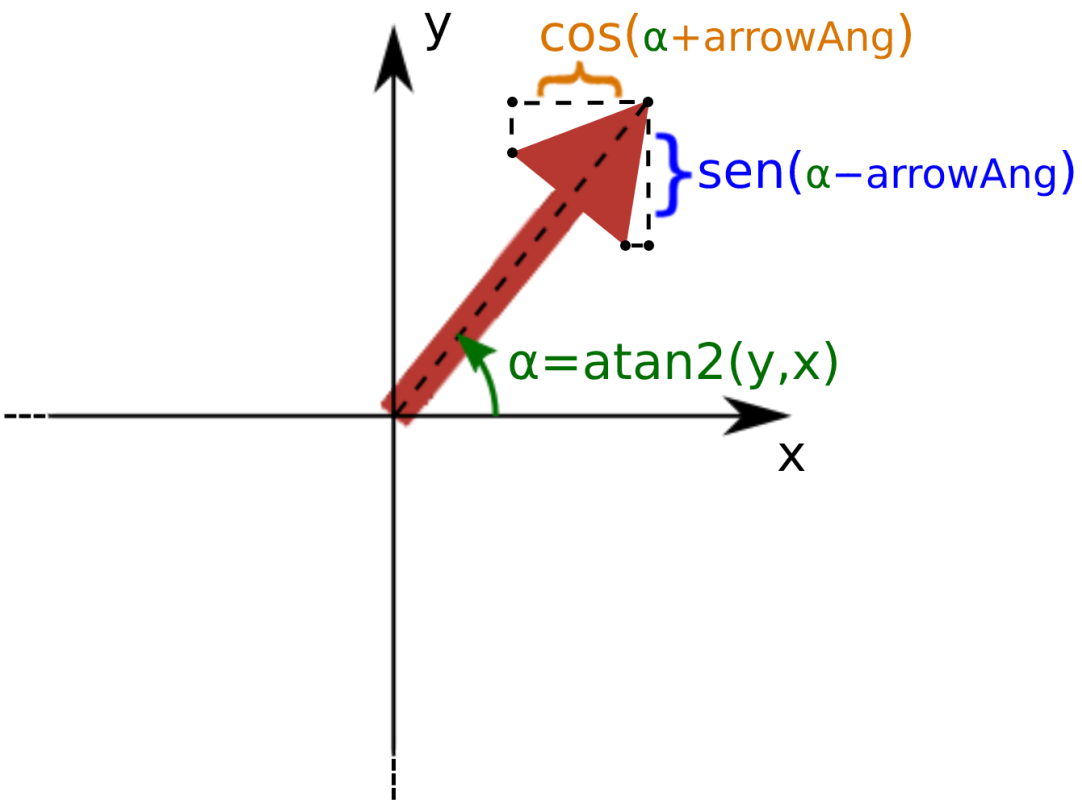
To effectively render the arrow on the canvas, I gathered up my understanding of trigonometry to create the following function:

```
drawArrow(from, to) {
    // Setup
    const canvas = document.getElementById("arrowCanvas");
    const context = canvas.getContext("2d");
    const offset = vmin(5);                          // Offset to center arrows in cells
    const headLen = offset / 4;                      // Length of the arrowhead
    const headAng = Math.PI / 7;                     // Angle of the arrowhead
    const color = "#c62828";                         // Color of the arrow
    context.strokeStyle = color;                     // Stroke color of the arrow
    context.fillStyle = color;                       // Fill color of the arrow
    context.lineWidth = offset / 3;                  // Width of the arrow stroke
    // Get coordinates of 'from' and 'to' squares
    const getCoordinates = element => ({
      x: element.getBoundingClientRect().left - canvas.getBoundingClientRect().left + offset,
      y: element.getBoundingClientRect().top - canvas.getBoundingClientRect().top + offset
    });
    const { x: fromx, y: fromy } = getCoordinates(from);
    const { x: tox, y: toy } = getCoordinates(to);
    // Calculate the angle of the arrow
    const angle = Math.atan2(toy - fromy, tox - fromx);
    // Draw the main line
    context.beginPath();
    context.moveTo(fromx, fromy);
    context.lineTo(tox, toy);
    context.stroke();
    // Draw the arrowhead
    context.beginPath();
    context.moveTo(tox, toy);
    context.lineTo(tox - headLen * Math.cos(angle - headAng), toy - headLen * Math.sin(angle - headAng));
    context.lineTo(tox - headLen * Math.cos(angle + headAng), toy - headLen * Math.sin(angle + headAng));
    context.closePath();
    context.fill();
    context.stroke();
  }
```

The arrow drawing algorithm employs trigonometry concepts and functions to precisely position and orient the arrows on the canvas. Central to this is the Math.atan2() function, which accurately calculates the angle of each arrow. Formally defined as Math.atan2(y, x) is a mathematical function that returns the angle (in radians) between the positive x-axis and the vector from the origin (0, 0) to the point (x, y) in a Cartesian coordinate system. It overcomes the limitations of Math.atan(y / x) by considering the signs of both x and y, thus providing the correct angle in all quadrants of the coordinate plane. The returned angle ranges from -π to π. By supplying the differences in y and x coordinates between the from and to points as the arguments to Math.atan2(), the function calculates the angle that represents the direction of the arrow, accounting for all quadrants.

To draw the arrowhead, Math.sin() and Math.cos() trigonometric functions are utilized. Specifically, it calculates the horizontal offset from the tip of the arrow by multiplying headLength with Math.cos(angle - headAngle). This determines the horizontal distance in the direction of the arrow. Similarly, the vertical offset from the tip is obtained by multiplying headLength with Math.sin(angle - headAngle). This represents the vertical distance perpendicular to the arrow's direction. By subtracting these offsets from the tox and toy coordinates, respectively, the arrowhead's side points are positioned correctly.

In summary, the arrow drawing algorithm utilizes Math.atan2() to accurately determine the angle of each arrow, taking into account the quadrant and direction. Moreover, Math.sin() and Math.cos() are employed to calculate the vertical and horizontal offsets from the tip of the arrow, enabling precise placement of the arrowhead. These mathematical calculations and trigonometric functions work in harmony to ensure that the arrows are correctly positioned and aligned on the canvas.



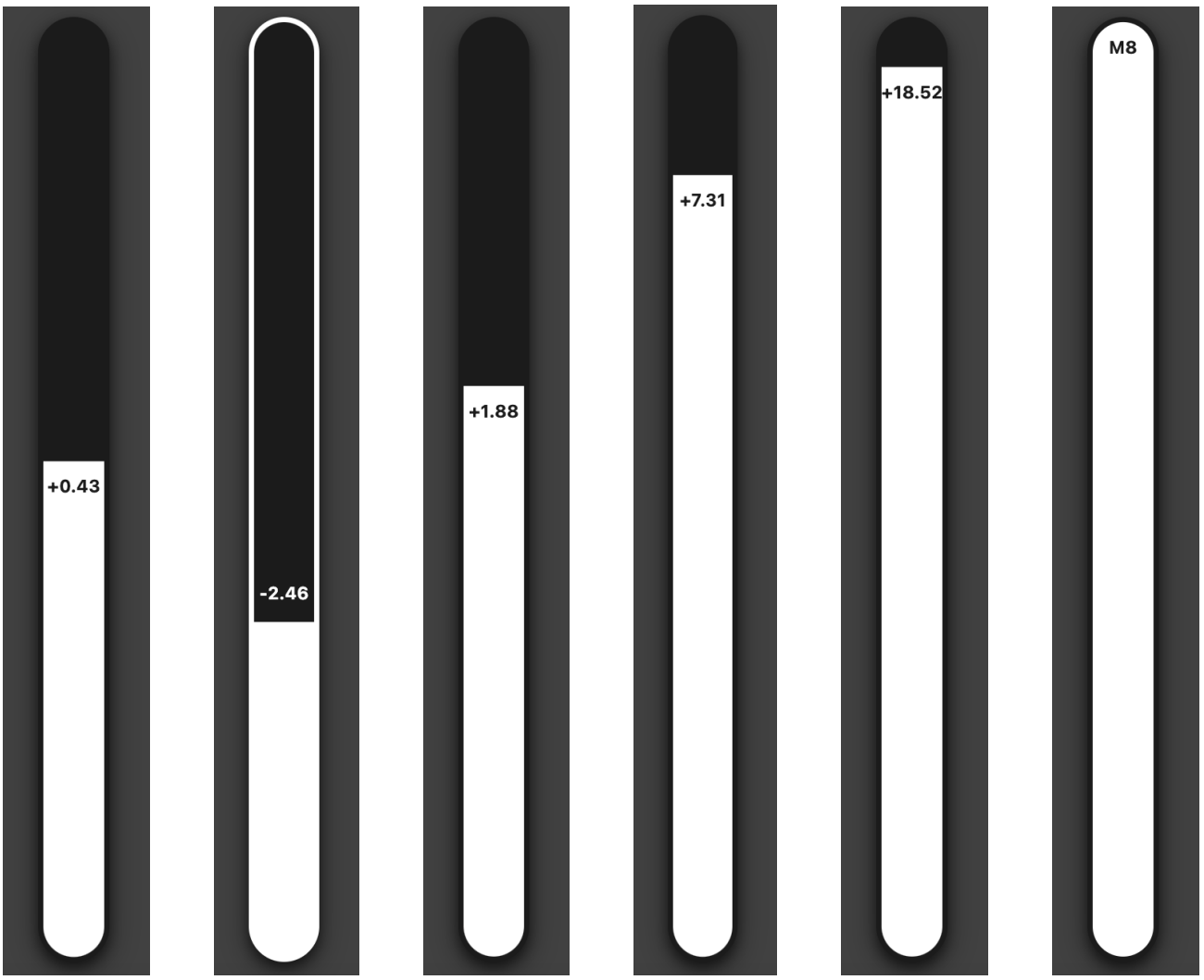
Another noteworthy example is the MoveList, as illustrated in the provided image. The component is displayed as a box on the left side of the interface, enabling users to navigate through the list of played moves. It incorporates a visual representation of moves using five distinct icons and colors. These icons and colors correspond to the evaluation provided by Stockfish, indicating whether a move was excellent, good, inaccurate, a mistake, or a blunder. By hovering the mouse over a move in the MoveList, users can visually see the move on the chessboard. Additionally, the component provides the Stockfish suggestion for each move, offering valuable insights into alternative moves or strategies.

Yet another worthy example is the Evaluation Bar, which presented a unique challenge in its seamless implementation. The Evaluation Bar is designed to display the current game situation based on Stockfish's evaluation, indicating which side is winning. However, it required careful handling to prevent the bar from being completely filled unless a forced checkmate was calculated by the engine.

The Evaluation Bar effectively represents the evaluation as a visual indicator, providing users with an instant understanding of the game's dynamics. It allows players to assess the balance of power and make informed decisions based on Stockfish's analysis. The complexity of this feature lies in its ability to dynamically update and reflect the changing evaluation without overwhelming the visual representation.

By skillfully incorporating HTML, CSS, and JavaScript, we were able to create an Evaluation Bar that harmoniously conveys the game's state while avoiding premature or exaggerated indications of a win or loss. This thoughtful

implementation ensures that the Evaluation Bar serves as a reliable and intuitive tool for players to gauge the progress of the game and make strategic choices accordingly.



To address the requirement of the Evaluation Bar to prevent it from fully filling when the evaluation deviates significantly from zero, a carefully crafted mapping function was implemented. This function converts the evaluation value, which can range from negative infinity to positive infinity, into a value between 0 and 100, representing the percentage of the bar to be filled.

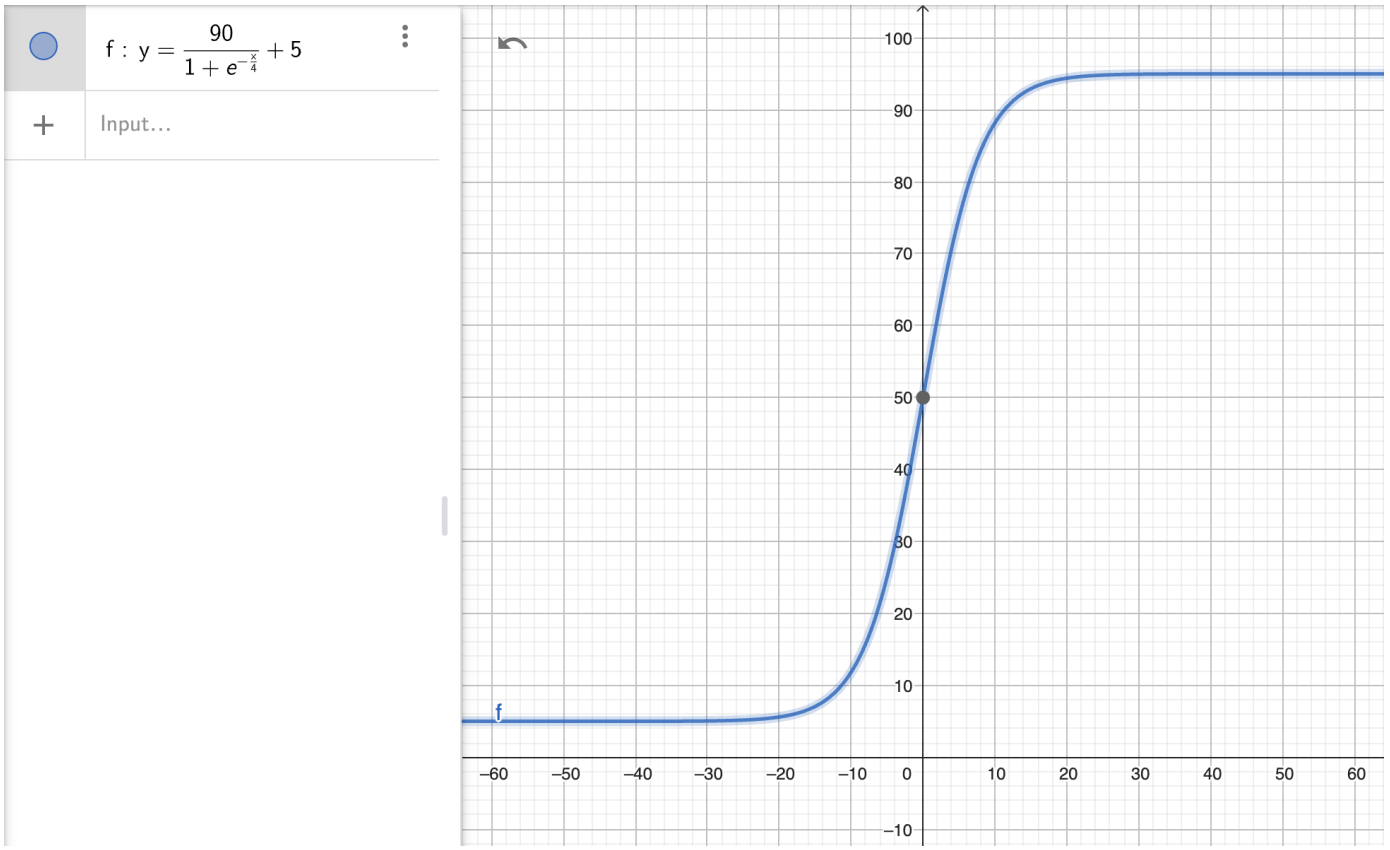
Based on the design decision to leave a margin of 5% to avoid extreme values close to 0% or 100%, a sigmoidal function was formulated. The sigmoidal function takes the evaluation value as its input and applies a mathematical transformation to map it to a value between 5 and 95. Here is the function that was derived:

```
static sigmoidalFunction(x) {
    return (90 / (1 + Math.exp(-x/4)))+5;
}
```

In this function, the input x represents the evaluation value. By applying the sigmoidal transformation using the logistic function, the output is adjusted to fit within the desired range of 5% to 95%. This mapping function ensures that the

Evaluation Bar accurately represents the game's evaluation while maintaining a visually balanced representation on the bar.

By leveraging mathematical concepts and utilizing our calculus knowledge, we were able to create a function that effectively maps evaluations to the appropriate range, allowing the Evaluation Bar to visually and accurately depict the game's current state. Here is a graphical representation of the aforementioned function.



All the components related to Stockfish are designed to provide live updates, continuously reflecting the engine's analysis as it delves deeper into evaluating the position. This seamless and real-time update functionality is achieved through the implementation of asynchronous event-driven programming techniques.

By leveraging asynchronous programming, the application can initiate requests to Stockfish for analysis without blocking the user interface. As Stockfish processes the position and provides updated evaluations, the application captures and handles these events asynchronously. This allows the UI components to be dynamically updated in response to the engine's ongoing analysis.

The use of event-driven programming ensures a smooth and responsive user experience, as the components receive and display the latest information from Stockfish without causing delays or interruptions. This enables players to closely monitor the evolving evaluations, make informed decisions, and gain valuable insights into the current state of the game.

One more remarkable case where custom UI design was necessary was for the PvP and PvE functionalities. This involved the implementation of various features such as a movelist, surrender button, draw offer button, material count display, and timer display. Additionally, an animated pop-up was implemented when the game ended, providing users with the opportunity to review the game with the assistance of Stockfish by hopping into a freeboard. This tailored UI design enhanced the user experience and facilitated post-game analysis and engagement.

Lastly, the implementation of the rooms feature posed a particular challenge as it involved integrating webcams on the left side of the interface along with interaction buttons for muting microphones, hiding cameras, and granting access to the chessboard. Another noteworthy feature we implemented was the ability for the room admin to initiate a voting process from the current position. Each student had the opportunity to play a move and participants could see which move received the highest and lowest number of votes at the end. This interactive voting mechanism added an engaging and collaborative element to the room, allowing students to actively participate and make decisions collectively.

3.3 Backend technologies

When it comes to the backend technologies, I can provide a brief overview rather than delving into intricate implementation details since it wasn't my primary area of focus in the project. However, I should highlight that my colleague Andrea D'Attero took the lead in handling most aspects of the backend. He has extensively covered the subject in his thesis titled "Sicurezza e Performance delle piattaforme online" where he provides a comprehensive explanation of how the backend operates to ensure security and performance. If you're interested in gaining a deeper understanding of the backend functionalities, I recommend reading Andrea's thesis, as it offers valuable insights into the intricacies of the system.

The backend code utilizes Express.js, a popular web application framework for Node.js, to handle HTTP requests and create API endpoints. It uses the HTTPS module to create an HTTPS server, allowing secure communication with clients.

For real-time functionality and bidirectional communication with clients, the code incorporates the WebSocket module. This enables the server to establish WebSocket connections and handle events such as joining games, making moves, and handling game-related actions.

To store and retrieve data, the backend uses MongoDB, a NoSQL database. It establishes a connection to the MongoDB database using Mongoose, an Object-Data Modeling (ODM) library for MongoDB and Node.js. This allows the backend to interact with the database and perform CRUD (Create, Read, Update, Delete) operations.

To ensure secure and reliable authentication, we have implemented the industry-standard JWT (JSON Web Token) mechanism. JWT tokens are utilized to authenticate and authorize user access, providing a robust layer of security. By employing JWT, we can verify the authenticity of requests and protect sensitive information throughout the communication process.

Furthermore, we have taken comprehensive measures to enhance security. We have implemented safeguards against CSRF (Cross-Site Request Forgery) attacks, mitigating the risk of unauthorized actions on behalf of users. Additionally, we have employed techniques like digest and salting for password storage, ensuring the confidentiality and integrity of user credentials.

3.3.1 Node.js

The choice of Node.js as the backend technology for our project was not just preferred, but mandatory due to the numerous advantages it offers. Here are the key reasons why Node.js was the perfect fit for our requirements:

- **JavaScript Expertise**: Node.js allows us to utilize our existing JavaScript skills for both frontend and backend development. This eliminates the need for developers to learn a new programming language, resulting in a more efficient and streamlined development process. The ability to share code

between the client and server-side further enhances productivity and code maintainability.

- **Asynchronous and Non-blocking Nature**: Node.js utilizes an asynchronous, non-blocking I/O model, which allows it to handle multiple requests concurrently without blocking other operations. This architecture is highly efficient for applications that require handling a large number of concurrent connections or real-time interactions, ensuring optimal performance and responsiveness.

- **Scalability and Performance**: Node.js is known for its scalability and ability to handle a high volume of concurrent connections. It achieves this through event-driven architecture and lightweight processes, making it well-suited for applications that require horizontal scalability and efficient resource utilization.

- **Vast Ecosystem and Package Manager**: Node.js has a vast ecosystem with a rich collection of open-source libraries and modules available through npm (Node Package Manager). This extensive ecosystem provides ready-made solutions for various functionalities, saving development time and effort. It also encourages code reuse and promotes collaboration within the Node.js community.

- **Microservices and API Development**: Node.js is an excellent choice for building microservices and API-centric architectures. Its lightweight nature, coupled with its ability to handle concurrent requests, makes it ideal for developing scalable and modular services. Node.js's support for creating RESTful APIs simplifies the development of robust and efficient API endpoints.

- **Developer Productivity**: Node.js promotes developer productivity by offering a familiar language and tools. Its event-driven, non-blocking nature allows for rapid prototyping, quick iterations, and faster development cycles. Additionally, the extensive documentation, active community support, and vast selection of tools and frameworks further enhance developer productivity.

3.3.1.1 Express.js

Another mandatory choice was using Express.js as our middleware due to the following reasons:

- **Simplicity and Minimalism**: Express.js is known for its simplicity and minimalist approach. It provides a lightweight framework that allows us to build web applications quickly and efficiently. The straightforward and intuitive API of Express.js makes it easy to learn and use, enabling rapid development and reducing the learning curve for developers.

- **Robust Routing and Middleware**: Express.js offers robust routing capabilities, allowing us to define routes for different HTTP methods and URLs. This flexibility enables us to handle complex routing scenarios and efficiently manage our API endpoints. Additionally, Express.js has excellent support for middleware functions, which can be used for tasks like request/response modification, error handling, authentication, and more.

- **Scalability and Performance**: Express.js is designed to be highly scalable and performant. It allows us to handle a large number of concurrent requests efficiently, making it suitable for applications with high traffic and demanding workloads. Express.js leverages non-blocking I/O and event-driven architecture, enabling better utilization of system resources and delivering excellent performance.

- **Vibrant Ecosystem and Community**: Express.js benefits from a vibrant and active community of developers. It has a large ecosystem of plugins, middleware, and extensions that extend its functionality and provide additional features. The availability of a strong community ensures ongoing support, frequent updates, and the ability to leverage existing solutions and best practices.

- **Integration with Other Node.js Libraries**: Express.js seamlessly integrates with other Node.js libraries and modules, allowing us to leverage the rich Node.js ecosystem. This interoperability enables us to easily integrate database libraries, template engines, authentication frameworks, and other components into our Express.js application, enhancing its capabilities and reducing development time.

Overall, Express.js was a compelling choice as the middleware for our backend due to its simplicity, robustness, scalability, and strong community support. It provides a solid foundation for developing efficient and feature-rich web applications. Please refer to the table below for a detailed description of our HTTP REST API.

Endpoint	Method	Description	Request Body	Response
/profile	GET	Retrieves user profile information.	None	`{ email: string, username: string, elo: number }`
/auth/sign-in	POST	Authenticates user credentials.	`{ email: string, password: string }`	JWT token or `{ error: string }`
/auth/sign-up	POST	Creates a new user account.	User object	`{ message: string }` or `{ error: string }`
/auth/sign-out	POST	Signs out a user by deleting their account.	`{ email: string, password: string }`	`Redirect` or `{ error: string }`
/games	GET	Retrieves a list of games.	None	Array of games
/games/create	POST	Creates a new game.	Game object	Created game object
/games/:gameId/play	POST	Updates game state and manages player time.	None	Updated game object

/games/delete	DELETE	Deletes all games.	None	`{ message: string }`
/rooms	GET	Retrieves a list of public rooms.	None	Array of rooms
/rooms/create	POST	Creates a new room with an admin user.	Room object	Created room object
/rooms/:roomId	GET	Retrieves room details by room ID.	None	Room object
/rooms/:roomId/is-admin	GET	Checks if the user is an admin of the room.	None	`{ isAdmin: boolean }`
/rooms/:roomId/access	GET	Checks if the user has access to the room.	None	`{ access: boolean }`
/rooms/:roomId/position	GET	Retrieves the user's position in the room.	None	`{ position: number }`
/rooms/delete	DELETE	Deletes all rooms.	None	`{ message: string }`

3.3.1.2 Socket.IO

Due to the real-time nature of our project, REST API alone was not sufficient to fulfill all the required features. Therefore, we implemented WebSocket communication using Socket.IO, which has proven to be an excellent choice for several reasons:

- **Real-time Communication**: Socket.IO enables real-time communication between the client and the server. It establishes a persistent connection that allows data to be sent and received instantly, without the need for frequent HTTP requests. This is particularly useful for applications that require instant updates, such as chat applications, collaborative tools, multiplayer games, and live dashboards.

- **Event-based Communication**: Socket.IO uses an event-driven communication model. Both the client and the server can emit events and listen for events. Events can be custom-defined and can carry data payloads. This allows you to create custom event-based actions and triggers for handling various functionalities in your application.

- **Broadcasting and Rooms**: Socket.IO provides broadcasting capabilities, allowing you to send messages to specific clients or groups of clients. You can organize clients into rooms, channels, or namespaces, and broadcast events to all clients within a specific room or channel. This enables targeted communication and efficient data distribution.

- **Fallback Options**: Socket.IO has built-in fallback mechanisms that ensure compatibility with a wide range of clients, including older browsers or clients that do not support WebSockets. It automatically falls back to alternative transport mechanisms, such as long-polling or server-sent events, to maintain a reliable connection in different environments.

- **Scalability and Load Balancing**: Socket.IO can be scaled horizontally by running multiple instances of the server and distributing the client connections across them. Socket.IO provides a built-in adapter that allows you to use a message broker, such as Redis, to enable inter-process communication and share messages across multiple instances. This helps in load balancing and scaling your real-time application.

By utilizing Socket.IO, we were able to enhance our application with real-time features, such as live updates, chat functionality, and collaborative interactions. Please refer to the table below, which showcases the Socket Events utilized in our application.

Event	Description	Parameters
Connection	Triggered when a client establishes a socket connection.	-
Join	Triggered when a client joins a game.	- message (String): JSON string containing the token and gameId information.
Move	Triggered when a player makes a move in the game.	- message (String): JSON string containing the move information.
Surrender	Triggered when a player surrenders the game.	- message (String): JSON string containing the surrender information.
Offer-draw	Triggered when a player offers a draw in the game.	- message (String): JSON string containing the offer-draw information.
Disconnect	Triggered when a client disconnects from the game server.	-
Join-room	Triggered when a client joins a room.	- roomId (String): ID of the room. - userSessionId (String): Session ID of the user. - userId (String): ID of the user.

Toggle-mute	Triggered when a client toggles their mute state.	-
Toggle-camera	Triggered when a client toggles their camera state.	-
Toggle-board	Triggered when a client requests to toggle the board.	- clientId (String): ID of the client requesting the toggle.
Admin-mute	Triggered when an admin mutes a client in the room.	- clientId (String): ID of the client to be muted.
Board-update	Triggered when the board position is updated in the room.	- position (String): The updated board position. - move (String): The move made to update the position.
Toggle-stockfish	Triggered when an admin toggles the Stockfish engine in the room.	-
Comment	Triggered when an admin adds a comment in the room.	- comment (String): The comment added by the admin.
Toggle-move	Triggered when the state of move toggling is changed in the room.	- state (Boolean): The new state of move toggling.

Vote-move	Triggered when a client votes for a move in the room.	- move (String): The move voted by the client.
Move	Triggered when a client makes a move in the room.	- move (String): The move made by the client.
Position	Triggered when a client requests the current board position in the room.	-
Disconnect	Triggered when a client disconnects from the room server.	-
Ask-access	Triggered when a client requests access to a private room.	- roomId (String): ID of the room. - userId (String): ID of the user.
Admin-approved	Triggered when an admin approves a client's access request to a private room.	- roomId (String): ID of the room. - userAccessId (String): Access ID of the user. - adminId (String): ID of the admin.
Ban	Triggered when an admin bans a client from the room.	- clientId (String): ID of the client to be banned.
Remove	Triggered when an admin removes a client from the room.	- clientId (String): ID of the client to be removed.

3.3.2 MongoDB

Utilizing MongoDB as our database was an exceptional and contemporary solution. This decision proved to be highly advantageous for several reasons. Let's explore the key factors that made MongoDB a great choice for our project:

- **Flexibility**: MongoDB is a flexible and schema-less database, allowing you to store and retrieve data without strict predefined schemas. This flexibility is beneficial when dealing with evolving data models or frequent changes in requirements.

- **Scalability**: MongoDB is designed to scale horizontally, allowing you to distribute data across multiple servers or clusters easily. This makes it suitable for handling large amounts of data and high traffic loads.

- **Performance**: MongoDB offers high-performance read and write operations, making it well-suited for real-time applications and high-volume data processing. It supports indexing, sharding, and caching mechanisms to optimize query performance.

- **JSON-like Document Structure**: MongoDB stores data in a JSON-like document format called BSON (Binary JSON), which closely aligns with the structure of many programming languages. This makes it easier to work with and map data between the application and the database.

- **Ad Hoc Queries**: MongoDB supports powerful ad hoc queries, including complex aggregations and real-time analytics. It provides a rich query language with various operators and functions, allowing for flexible data retrieval and manipulation.

- **Horizontal Scaling**: MongoDB's sharding capabilities enable horizontal scaling by distributing data across multiple servers or clusters. This allows for increased storage capacity, improved performance, and fault tolerance.

- **Automatic Replication**: MongoDB provides built-in replication features, allowing for automatic data synchronization across multiple replicas. This ensures high availability and fault tolerance, as well as data durability in case of hardware failures.

- **Community and Ecosystem**: MongoDB has a large and active community, offering extensive documentation, tutorials, and support resources. It also integrates well with popular frameworks, libraries, and tools, providing a robust ecosystem for development and operations.

- **Adoption and Industry Support**: MongoDB is widely adopted in both small startups and large enterprises, with notable companies using it for their data needs. Its widespread usage and industry support ensure ongoing development, updates, and security enhancements.

- **Cloud Compatibility**: MongoDB has native support for cloud environments and is available as a fully managed database service on various cloud platforms. This simplifies deployment, scalability, and management in cloud-based applications.

3.3.2.1 Collections

MongoDB's non-relational nature simplifies the complexity of describing database structures. Instead of intricate schemas and complex queries we can just examine the structure of collections and look at some JSON examples. Here are the collections we used in our project along with their descriptions:

USER COLLECTION

Field	Type	Unique	Required	Description
userId	String	Yes	Yes	Unique identifier for the user
username	String	No	Yes	Username of the user
email	String	Yes	Yes	Email address of the user
password	String	No	No	Hashed password of the user
createdAt	Date	No	No	Timestamp of user creation
updatedAt	Date	No	No	Timestamp of last user update

JSON EXAMPLE:


```
{
        _id : ObjectId('623bfba2fd5653195f25fcd1')
        userId : "1c0281d0-f08f-478f-9f15-370bd05b2b22"
        username : "Giovanni"
        email : "prova@mail.com"
        password : "$2a$10$YWJWwp/AK5Kdw/TK3pRGteHcshS1KggCe9btPI9M3Izo9uftHwQIu"
        createdAt : 2022-03-24T05:03:30.236+00:00
        updatedAt : 2022-03-24T05:03:30.236+00:00
}
```


PROFILE COLLECTION

Field	Type	Unique	Required	Description
userId	String	Yes	Yes	Unique identifier for the profile
elo	Number	No	No	Elo rating of the profile (default: 800)
createdAt	Date	No	No	Timestamp of profile creation
updatedAt	Date	No	No	Timestamp of last profile update

JSON EXAMPLE:


```
{
        _id : ObjectId('623bfba2fd5653195f25fcd2')
        userId : "1c0281d0-f08f-478f-9f15-370bd05b2b22"
        elo : 800
        createdAt : 2022-03-24T05:03:30.236+00:00
        updatedAt : 2022-03-24T05:03:30.236+00:00
}
```

GAME COLLECTION

Field	Type	Unique	Required	Description
gameId	String	Yes	Yes	Unique identifier for the game
whitePlayerId	String	No	No	ID of the white player
blackPlayerId	String	No	No	ID of the black player
moves	Array	No	No	Array of moves made in the game
hasEnded	Boolean	No	No	Indicates if the game has ended
winnerId	String	No	No	ID of the winning player
isStarted	Boolean	Yes	Yes	Indicates if the game has started
isRated	Boolean	Yes	Yes	Indicates if the game is rated
timeLimit	Number	Yes	Yes	Time limit for the game in minutes
timestamps	Array	No	No	Array of timestamps for game events

whitePlayerTime	Number	No	No	Remaining time for the white player
blackPlayerTime	Number	No	No	Remaining time for the black player
turn	String	No	No	Indicates the current turn
timeIncrement	Number	No	No	Time increment in seconds
reason	String	No	No	Reason for game end

JSON EXAMPLE:


```
{
        _id : ObjectId('646682d94798da6e8a8a785b')
        gameId : "029a6db1-2983-4355-b923-4478018fdc5a"
        whitePlayerId : "988c4983-e376-4437-b2ec-7704fdb2b5b1"
        blackPlayerId : "4ff155ec-956c-48d6-9858-efb10ca6d9f1"
        moves : Array
                0 : "e2e4"
                1 : "d7d5"
                2 : "e4d5"
                3 : "d8d5"
                      ⋮
                      ⋮
                22 : "d3f5"
        hasEnded : true
        winnerId : "988c4983-e376-4437-b2ec-7704fdb2b5b1"
        isStarted : true
        isRated : false
        timeLimit : 180
        timestamps : Array
                0 : 2023-05-18T19:56:16.741+00:00
                1 : 2023-05-18T19:56:18.505+00:00
                2 : 2023-05-18T19:56:23.092+00:00
                3 : 2023-05-18T19:56:25.554+00:00
                                ⋮
                                ⋮
                24 : 2023-05-18T20:03:30.091+00:00
        whitePlayerTime : 204
        blackPlayerTime : 0
        turn : "black"
        timeIncrement : 2
        reason : "Timeout"
}
```

ROOM COLLECTION

Field	Type	Unique	Required	Description
roomId	String	Yes	Yes	Unique identifier for the room
isPublic	Boolean	No	No	Indicates if the room is public
admins	Array	No	No	Array of admin IDs
moves	Array	No	No	Array of moves made in the room
timestamps	Array	No	No	Array of timestamps for room events
approved	Array	No	No	Array of approved items
position	String	No	No	Position value for the room

JSON EXAMPLE:


```
{
        _id : ObjectId('626ece4e5ea0311d49048d63')
        roomId : "d90e8364-ab88-4f4a-8599-4134434b9611"
        isPublic : false
        admins : Array
                0 : "60d17d5b-e4f9-48c8-a5bf-9179293a5324"
        moves : Array
        timestamps : Array
        approved : Array
                0 : "a2aea624-a56d-4c0f-bac8-e517d1a4aa63"
        position : "rnbqkbnr/pppppppp/8/8/4P3/8/PPPP1PPP/RNBQKBNR b KQkq e3 0 1"
}
```

3.3.2.2 Mongoose

Mongoose is a robust and feature-rich object modeling tool designed for Node.js and MongoDB. It simplifies the interaction with MongoDB by providing a straightforward API and advanced functionalities. Some of the advantages of using Mongoose include:

- **Schema-based Approach**: Mongoose allows developers to define schemas for their data models, providing a structured and consistent way to organize data.

- **Data Validation**: Mongoose offers built-in validation capabilities, allowing developers to ensure that the data meets specific requirements before saving it to the database.

- **Middleware Support**: Mongoose supports middleware functions, enabling developers to add custom logic before or after specific database operations, such as saving or removing documents.

- **Query Building**: Mongoose provides a rich set of query building methods, making it easy to construct complex queries and perform advanced data retrieval operations.

- **Relationship Management**: Mongoose supports defining relationships between different data models, including one-to-one, one-to-many, and many-to-many relationships.

- **Data Population**: Mongoose allows for easy data population, enabling developers to retrieve related data from other collections in a single query.

- **Plugin System**: Mongoose has a plugin system that allows developers to extend its functionality with custom plugins or leverage existing plugins from the Mongoose ecosystem.

- **Active Community and Documentation**: Mongoose has a large and active community, providing ample support and resources. Its extensive documentation makes it easy to get started and learn the various features and best practices.

3.4 Chess Related technologies

During the development of our project, a critical aspect was exploring the available chess libraries to leverage existing functionalities and avoid reinventing the wheel. We carefully considered the choice of Stockfish as the engine due to its exceptional performance. Fortunately, we discovered a well-established library that provided a JavaScript WebAssembly (WASM) implementation of Stockfish, similar to what lichess utilizes. Additionally, we needed a robust class to handle chess rules and data, and we opted for chess.js. To effectively utilize these libraries, it is important to understand standard chess notations such as FEN (Forsyth-Edwards Notation), UCI (Universal Chess Interface), AN (Algebraic Notation), and their variants.

3.4.1 Forsyth-Edwards Notation (FEN)

FEN (Forsyth-Edwards Notation) is a standard notation system used to describe a particular chess position. It provides a compact and human-readable representation of the board state, including the placement of pieces, the current player to move, castling rights, en passant targets, and the halfmove and fullmove counters.

The FEN notation consists of six fields separated by spaces. Here is an explanation of each field:

- **Piece Placement**: This field represents the position of pieces on the board. Each rank (row) is described from the eighth rank (top of the board) to the first rank (bottom of the board). Each piece is represented by a letter: 'K' for white king, 'Q' for white queen, 'R' for white rook, 'B' for white bishop, 'N' for white knight, and 'P' for white pawn. The lowercase letters represent the black pieces.

- **Active Color**: This field indicates the side to move next. 'w' represents white, and 'b' represents black.

- **Castling Rights**: This field denotes the castling availability for each side. It uses four letters: 'K' for kingside castling by white, 'Q' for queenside castling by white, 'k' for kingside castling by black, and 'q' for queenside castling by black. If a side cannot castle, a hyphen '-' is used.

- **En Passant Target Square**: If there is a possibility for an en passant capture, this field indicates the target square. Otherwise, it is represented by a hyphen '-'.

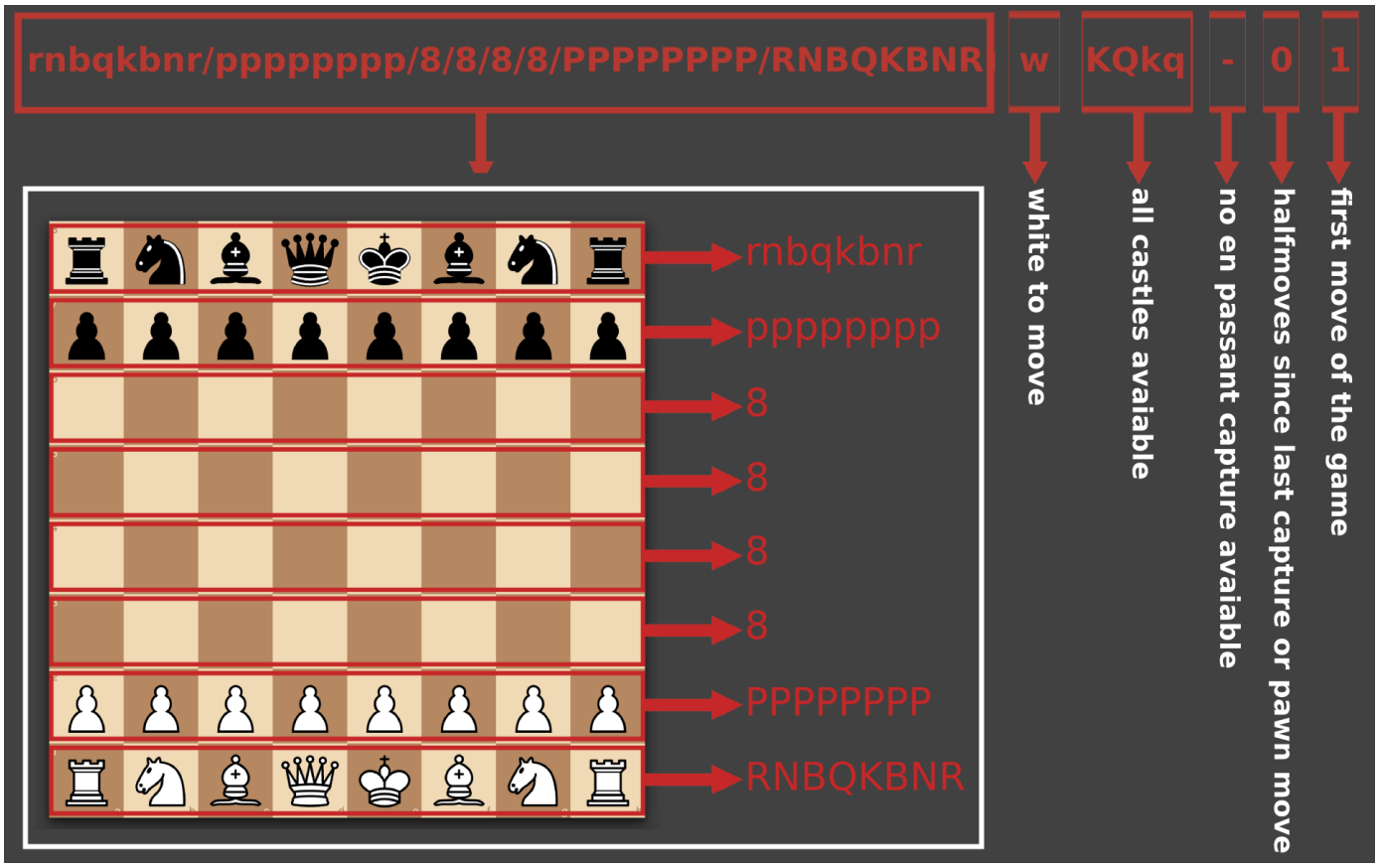
- **Halfmove Clock**: This field represents the number of halfmoves since the last capture or pawn move. It is used for the fifty-move rule in chess.

- **Fullmove Counter**: This field tracks the number of the current full move. It starts at 1 and is incremented after each move by black.

Here's an example FEN string that represents the starting position of a chess game:

"rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1"

Let's break it down:



By using FEN notation, chess positions can be easily shared, stored, or transmitted. It is widely used in various chess-related applications, including chess databases, game analysis tools, and communication between chess engines and user interfaces.

3.4.2 Algebraic notation (AN)

Chess Algebraic Notation is a standardized system used to record and describe chess moves. It provides a concise and human-readable representation of the moves played during a game. There are different variants of Algebraic Notation, including Standard Algebraic Notation (SAN), Long Algebraic Notation (LAN), and Portable Game Notation (PGN).

Let's examine them and gain a comprehensive understanding of how these notations function:

- **Standard Algebraic Notation (SAN)**:

  SAN is the most widely used variant of Algebraic Notation. It uses a combination of letters and numbers to represent the moves. Here's how it works:

  - Each move starts with the abbreviation of the piece moved (K for king, Q for queen, N for knight, R for rook, B for bishop) or is represented by an empty space for a pawn move.
  - The target square of the move is specified using the file (a-h) and rank (1-8) coordinates.
  - When a piece is captured the lowercase "x" is placed between the piece abbreviation and the arrival square (**e.g.** "Qxd5" for White Queen captures on d5).
  - Additional annotations can be added, such as "+" for check or "#" for checkmate.
  - Castling is represented by "O-O" for kingside and "O-O-O" for queenside.
  - Piece promotion is indicated by appending the desired piece's abbreviation after the move (**e.g.** "e8Q" for promoting to a Queen on e8).
  - If multiple pieces of the same type can move to the same square, the originating piece's file is used to distinguish between them (**e.g.** "Nbd4" and "Ngd4" for knights on b1 and g1 moving to d4). Rarely, both the rank and file of the originating piece need to be specified to avoid ambiguity (**e.g.** "R3f4" and "R5f4" for white rooks on the third and fifth ranks moving to f4).

    - **Example**: The move where a pawn moves from e2 to e4 is represented as "**e4**" in SAN.

- **Long Algebraic Notation (LAN)**:

  LAN is a more detailed variant of Algebraic Notation that includes the starting square of the moved piece. It is often used in chess software and analysis.

  - By providing these additional details it minimizes ambiguity.
  - This notation is what we primarily adopted for storing chess moves in our project. The avoidance of ambiguity is particularly useful for accurately representing the move.

    - **Example**: The move where a pawn moves from e2 to e4 is represented as "**e2e4**" in LAN.

- **Portable Game Notation (PGN)**:
  PGN is a standard format for recording chess games, including move information, game metadata, and annotations. It can store the moves in SAN or LAN format, along with additional game-related information.

    - It can be very useful for tournament reporting, building chess databases, chess studies and lessons.

        - **Example**:

```
[Event "Fool's Mate"]
[Site "Unknown"]
[Date "2023.05.20"]
[Round "1"]
[White "Player1"]
[Black "Player2"]
[Result "1-0"]

1. f3 e5
2. g4 Qh4#
```

Each variant of Algebraic Notation has its advantages and use cases. SAN is popular among players and widely used in notation books, LAN provides more detailed move information, and PGN is a comprehensive format for storing complete game records.

3.4.2 Universal Chess Interface (UCI)

The Universal Chess Interface (UCI) is a communication protocol that allows a chess graphical user interface (GUI) to interact with a chess engine. It establishes a standardized way for the GUI to send commands to the engine and receive responses, enabling seamless integration and compatibility between different chess software components. UCI has become the de facto standard for communication between GUIs and engines in the chess software ecosystem.

UCI is primarily used for playing and analyzing chess games. It allows users to utilize powerful chess engines to analyze positions, find the best moves, and play against the engine at various difficulty levels. The GUI acts as the intermediary between the user and the engine, handling the user's inputs, displaying the board, and relaying the engine's analysis and moves.

To interact with a chess engine using UCI, these are the most useful commands:

GUI to ENGINE:

- **uci**: This command is used to initiate the communication with the engine. The engine should respond with its identification details and capabilities.

- **setoption name <option> [value <x>]**: This command is used to set various engine-specific options, such as hash table size or search depth. Here are some commonly used options:

    - **Hash**: Specifies the size of the hash table used for storing positions and their evaluations during the search.
        - **Example**: "setoption name Hash value 128"

    - **Threads**: Sets the number of threads or CPU cores the engine can utilize for searching.
        - **Example**: "setoption name Threads value 4"

    - **MultiPV**: Sets the number of principal variations (best moves) to be returned by the engine.
        - **Example**: "setoption name MultiPV value 2"

    - **Skill Level**: Adjusts the playing strength or skill level of the engine. Higher values make the engine play stronger.
        - **Example**: "setoption name Skill Level value 20"

    - **Contempt**: Controls the engine's attitude towards draws. Positive values make the engine more likely to play for a win, while negative values make it more willing to accept a draw.
        - **Example**: "setoption name Contempt value 50"

    - **Ponder**: Enables or disables the engine's ability to think on the opponent's time. When enabled, the engine will continue analyzing during the opponent's turn.
        - **Example**: "setoption name Ponder value true"

    - **UCI_AnalyseMode**: Activates or deactivates the engine's analysis mode, allowing it to provide analysis and evaluations during a game.
        - **Example**: "setoption name UCI_AnalyseMode value true"

    - **SyzygyPath**: Specifies the path to the tablebases files for endgame positions.
        - **Example**: "setoption name SyzygyPath value C:\Tablebases"

- **position [fen <fenstring> | startpos] [moves <move 1> .... <move n>]**: This command is used to set up the board position for analysis or play. You can provide the position using Forsyth-Edwards Notation (FEN) or specify the standard starting position ("startpos"). You can also include a list of moves to reach the desired position.
    - **Example**:
      "position startpos moves e2e4 e7e5 g1f3"
      or
      "position fen rnbqkbnr/pppp1ppp/8/4p3/4P3/5N2/PPPP1PPP/RNBQKB1R b KQkq - 1 2"

- **go [search_parameters]**: This command instructs the engine to start searching for the best move. Search parameters may include options like search depth, time limits, or other criteria. Here are some common search_parameters:

    - **depth <x>**: Specifies the maximum depth of the search tree in ply (half-move) units. The engine will search to a depth of <x> plies.
        - **Example**: "go depth 10"

    - **nodes <x>**: Sets the maximum number of nodes the engine should search. A node represents a position in the search tree. The engine will stop searching when it reaches <x> nodes.
        - **Example**: "go nodes 1000000"

    - **movetime <x>**: Sets the maximum amount of time, in milliseconds, that the engine should spend on the search. The engine will stop searching after <x> milliseconds.
        - **Example**: "go movetime 5000"

    - **infinite**: Tells the engine to search until it receives a "stop" command. It will keep searching indefinitely until instructed otherwise.
        - **Example**: "go infinite"

    - **wtime <x> btime <x>**: Specifies the remaining time for White and Black in milliseconds. These parameters are used in conjunction with the "movetime" or "depth" parameters to control the engine's search time allocation.
        - **Example**: "go wtime 120000 btime 180000"

    - **winc <x> binc <x>**: Specifies the time increment for White and Black in milliseconds. If these parameters are set, the engine will add the specified increment to the remaining time after each move.
        - **Example**: "go winc 2000 binc 3000"

    - **searchmoves <move 1> ... <move n>**: Limits the search to a specific set of moves. The engine will only consider the specified moves when searching for the best move. This parameter can be used to explore specific lines or variations.
        - **Example**: "go searchmoves e2e4 d2d4"

    - **ponder**: Instructs the engine to ponder (think during the opponent's time) after it has returned a move. The engine will continue searching and analyzing the position while waiting for the opponent to make a move.
        - **Example**: "go ponder"

    - **ponderhit**: Indicates that the opponent's move received after a pondering search is the same as the move the engine was pondering. It confirms that the engine's predicted move was correct.
        - **Example**: "go ponderhit"

- **isready**: This command is used to synchronize the engine with the GUI. When the GUI has sent a command or multiple commands that can take some time to complete, this command can be used to wait for the engine to be ready again.

- **stop**: This command is used to instruct the engine to stop searching and provide the best move found so far.

- **quit**: This command terminates the engine.

ENGINE to GUI

- **uciok**: Sent to indicate that the engine has finished sending all necessary information and is ready to operate in UCI mode.

- **readyok**: Sent in response to the GUI's "isready" command, indicating that the engine has processed all input and is ready to accept new commands.

- **bestmove <move1> [ponder <move2>]**: Sent when the engine has finished searching and found the best move for the current position. Optionally followed by a "ponder" move, indicating the move the engine is considering while the opponent is thinking.
    - **Example**: "bestmove d2d4 ponder d7d5"

- **info**: By far the most useful answer that can provide various types of information during the search process. Includes details such as search depth, time elapsed, number of nodes searched, principal variation (best line), evaluation scores, hash usage, and more. Allows the GUI to display real-time search progress and analysis information to the user.

    - **depth <x>**: Specifies the current search depth in plies (half-moves).
        - **Example**: "info depth 12"

    - **seldepth <x>**: Represents the selective search depth, which is the deepest the engine has calculated.
        - **Example**: "info seldepth 20"

    - **score**: Provides the evaluation score from the engine's perspective.

        - **cp <x>**: The score in centipawns. Positive values favor White, and negative values favor Black.
            - **Example**: "info score cp 50"

        - **mate <y>**: Indicates a checkmate is expected within y moves. Positive values indicate the number of moves until checkmate, and negative values indicate that the engine is being mated.
            - **Example**: "info score mate -3"

    - **pv <move 1> ... <move n>**: Represents the principal variation, which is the sequence of moves the engine considers the best for both players.
        - **Example**: "info pv e2e4 c7c5 g1f3"

    - **multipv <num>**: Used in multi-pv mode to indicate the number of the current variation.
        - **Example**: "info multipv 1 pv e2e4 e7e5
          info multipv 2 pv d2d4 d7d5"

- **currmove <move>**: Indicates the move the engine is currently analyzing or searching.
  - **Example**: "info currmove d2d4"

- **currmovenumber <x>**: Indicates the move number the engine is currently analyzing or searching.
  - **Example**: "info currmovenumber 4"

- **time <x>**: Indicates the time (in milliseconds) the engine has spent on the search so far.
  - **Example**: "info time 5000"

- **nodes <x>**: Indicates the number of nodes (positions) the engine has evaluated so far.
  - **Example**: "info nodes 500000"

- **hashfull <x>**:Represents the hash table usage as a percentage.
  - **Example**: "info hashfull 50"

- **nps <x>**: Represents the number of nodes per second the engine is evaluating.
  - **Example**: "info nps 100000"

- **tbhits <x>**: Represents the number of positions that were found in the endgame tablebases.
  - **Example**: "info tbhits 2"

These are some of the basic commands used to interact with a chess engine through UCI. The specific capabilities and supported commands may vary between different engines.

- **Example**:

GUI to ENGINE:

```
position startpos
go depth 3
```

ENGINE to GUI:

```
info depth 1 seldepth 0 time 10 nodes 15 nps 1500 score cp 10 pv e2e4
info depth 2 seldepth 0 time 20 nodes 30 nps 1500 score cp 20 pv e7e5 g1f3
info depth 3 seldepth 1 time 30 nodes 50 nps 1667 score cp 30 pv c7c5 b1c3 d7d6
bestmove d7d5
```

3.4.3 Chess.js library

When it comes to managing move generation, validation, piece placement, movement, and check/checkmate/stalemate detection, using an established library like chess.js is a smart choice. chess.js is a popular JavaScript library that provides chess-related functionality and it is commonly used for building chess applications, engines, and interactive chessboards. The library offers a comprehensive set of features for managing chess games, including:

- **Board Representation**: chess.js provides a data structure to represent the chessboard and its pieces. It offers methods to initialize the board, set piece positions, and retrieve information about specific squares.

- **Move Generation and Validation**: The library supports generating all legal moves for a given position. It includes functions to validate whether a move is legal or not based on the current board state, including checks, captures, castling, en passant, and promotions.

- **Game State Management**: chess.js allows you to manage the state of a chess game. You can make moves, undo moves, get the current position, and track the game history using standard algebraic notation (SAN) or coordinate notation.

- **Check, Checkmate, and Stalemate Detection**: The library includes functions to detect if the king is in check, checkmate, or stalemate. This enables you to determine the game's outcome and implement appropriate game logic.

- **PGN (Portable Game Notation) Support**: chess.js supports parsing and generating PGN, a standard format for recording chess games. You can import and export games in PGN format, making it easy to analyze, share, and replay games.

The chess.js library is extensively documented, offering detailed explanations, usage examples, and information about its methods and properties. To explore and access the library and its documentation, you can visit the official GitHub repository at https://github.com/jhlywa/chess.js.

By incorporating the chess.js library into our project, we have experienced significant benefits, such as reduced development time and minimized risk of errors. Let's take a look at an example that showcases the capabilities of chess.js:

```
const Chess = require('chess.js');

// Initiate a chess object
const chess = new Chess();

// Make a few moves
chess.move('e4');
chess.move('e5');
chess.move('Nf3');
chess.move('Nc6');

// Load position from FEN
chess.load('r1bqkbnr/pppp1ppp/2n5/4p3/4P3/5N2/PPPP1PPP/RNBQKB1R b KQkq - 3 3');

// Get all legal moves
chess.moves();
// Output: An array of legal moves

// Get legal moves from e7
chess.moves({ square: 'e7' });
// Output: ['e6', 'e5'] (Legal moves for the piece on e7)

// Make a move with specific square as starting position
chess.move({ from: 'e7', to: 'e5' });

// Get the current FEN position
chess.fen();
// Output: r1bqkbnr/pppp2pp/2n5/4pp2/4P3/5N2/PPPP1PPP/RNBQKB1R w KQkq - 0 4

// Get the current turn (side to move)
chess.turn();
// Output: 'w' (it's White's turn to move)

// Undo the last move
chess.undo();

// Check if the game has ended
chess.game_over();
// Output: false (the game is not over)

// Check if a player is in check
chess.in_check();
// Output: false (no player is currently in check)

// Check if a player is in checkmate
chess.in_checkmate();
// Output: false (no player is currently in checkmate)
```

3.4.4 Stockfish.js library

For our engine of choice, we have decided to use Stockfish due to its unparalleled power, performance, and availability as an open-source solution. Fortunately, we found a well-built JavaScript port of Stockfish called stockfish.js, which can be efficiently run in a modern browser as a Web Worker. This allows it to calculate moves without blocking the UI and provides constant updates as the calculations progress. If you would like to learn more about stockfish.js, you can visit the GitHub repository at https://github.com/nmrugg/stockfish.js/.

Here is an example that demonstrates basic interaction with the stockfish.js library, which implements the UCI (Universal Chess Interface) standard we discussed earlier:

- **Loading the engine**:

```
loadStockfishEngine(){
        //initiate web worker
        this.stockfish = new Worker("stockfish/src/stockfish.js");
        //set up the callback for stockfish output
        this.stockfish.onmessage = (e) => {
            this.updateStockfishOutPut(e.data);
        };
        //set the number of lines to evaluate
        this.stockfish.postMessage("setoption name MultiPV value " + this.lines)
        //set the starting position
        this.stockfish.postMessage("position startpos");
        //start evaluating with a depth parameter
        this.stockfish.postMessage("go depth " + this.depth);
    }
```

- **The onFenUpdate Event**:

```
<Chessboard ref={this.board}
      onFenUpdate = {(fen) => {
            if(this.stockfishON){
                  //stop current evaluation
                  this.stockfish.postMessage("stop");
                  //update FEN
                  this.stockfish.postMessage("position fen " + fen);
                  //start evaluating the new position
                  this.stockfish.postMessage("go depth " + this.depth);
            }
      }
}/>
```

- **The updateStockfishOutPut Callback**:

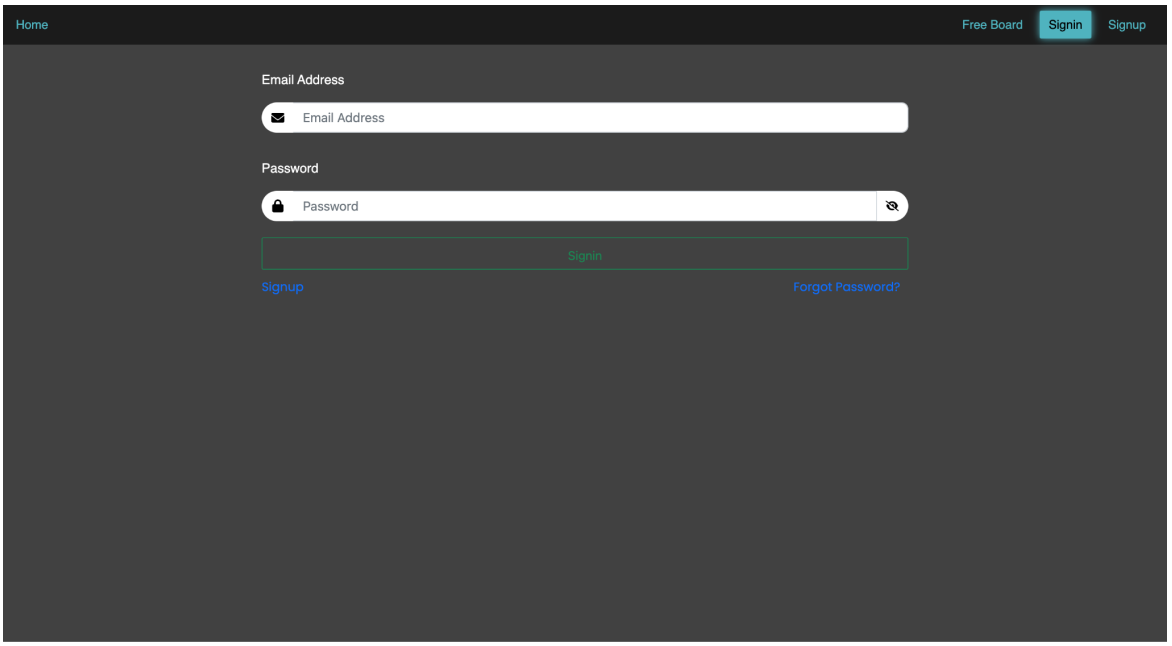
```
updateStockfishOutPut(msg) {
   if (msg.startsWith('bestmove')) {
     const bestMove = msg.split(' ')[1];
     // Process best move
   } else if (msg.startsWith('info')) {
     const evaluation = msg.match(/score cp (-?\d+)/)?.[1];
     const forcedMate = msg.match(/mate (\d+)/)?.[1];
     const multipv = msg.match(/multipv (\d+)/)?.[1];
     const pv = msg.match(/pv (.+)/)?.[1];
     // Process evaluation, forced mates and Principal Variations (best lines)
   }
}
```

3.5 Features implemented

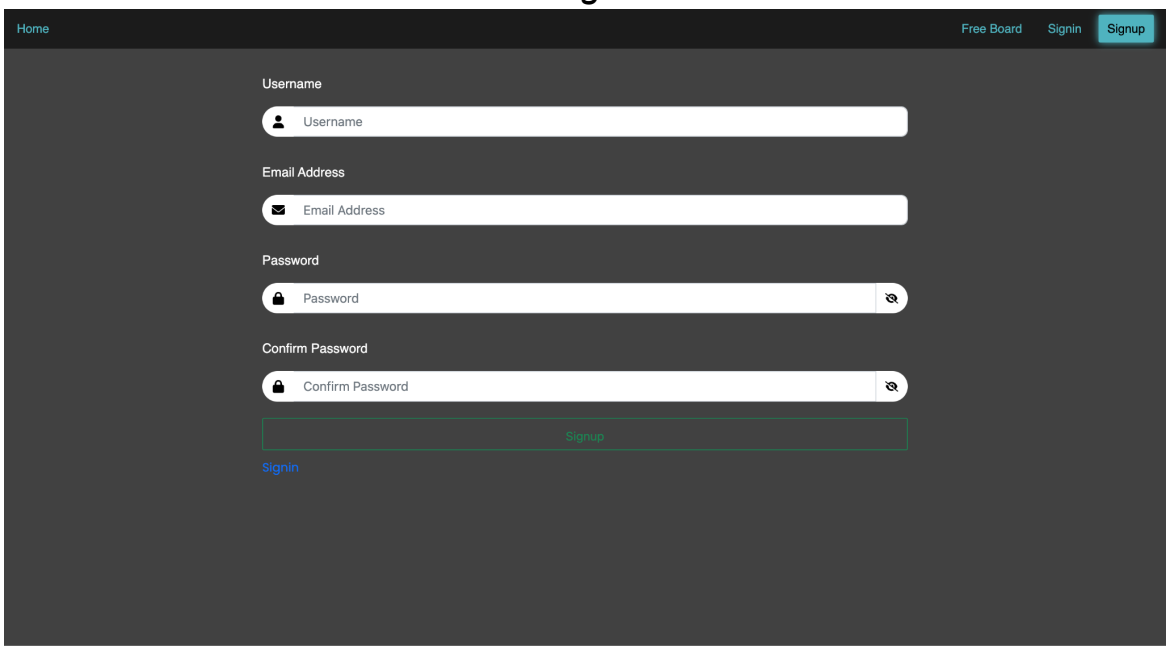
In this final chapter, we are excited to showcase some of the key features of our web application. Take a look at the following screenshots that highlight the implementation of authentication, a user-friendly freeboard, engaging gameplay, a comprehensive postgame Stockfish analysis, and dedicated rooms for chess lessons. These features come together to provide an enjoyable and educational experience.

3.5.1 Authentication

The authentication feature provides standard user sign-up functionality, allowing users to create an account by providing a username, email, and password. Once registered, users can log in using their credentials to access the application.



Login



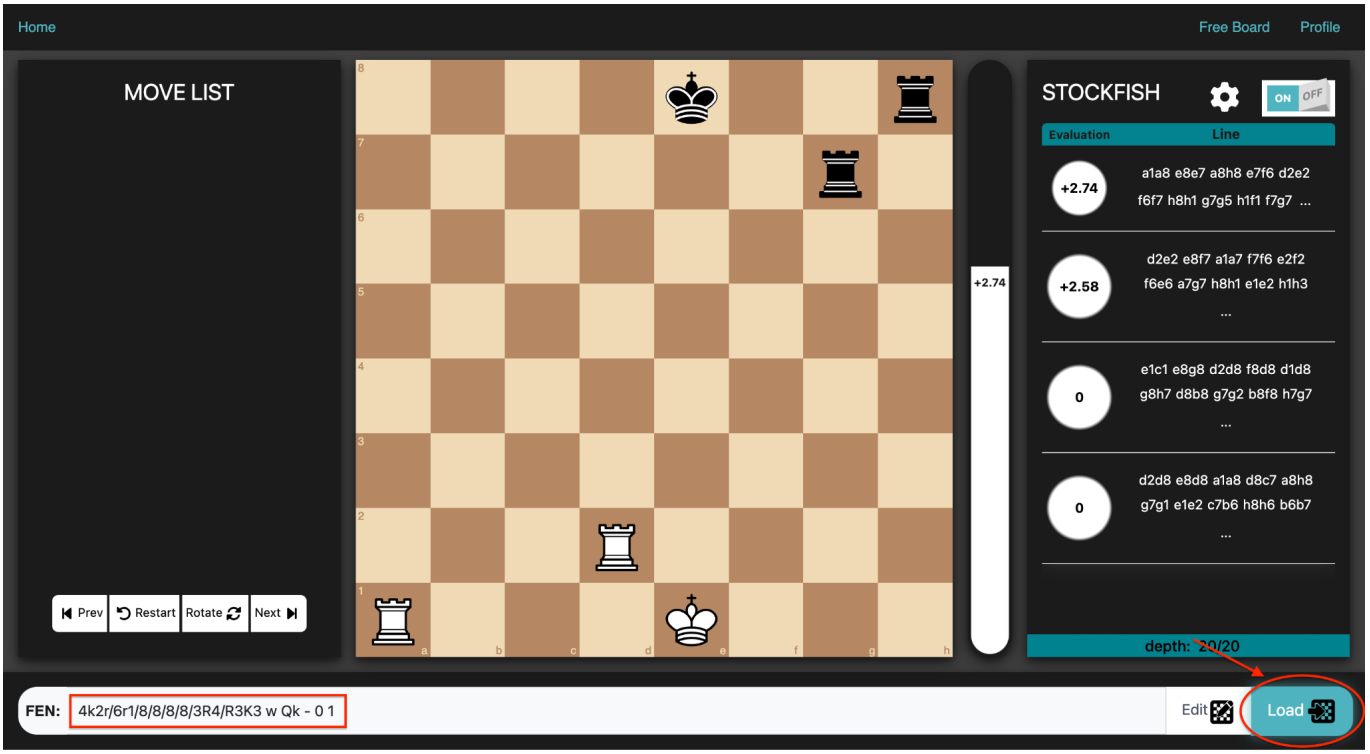
Signup

3.5.2 Freeboard

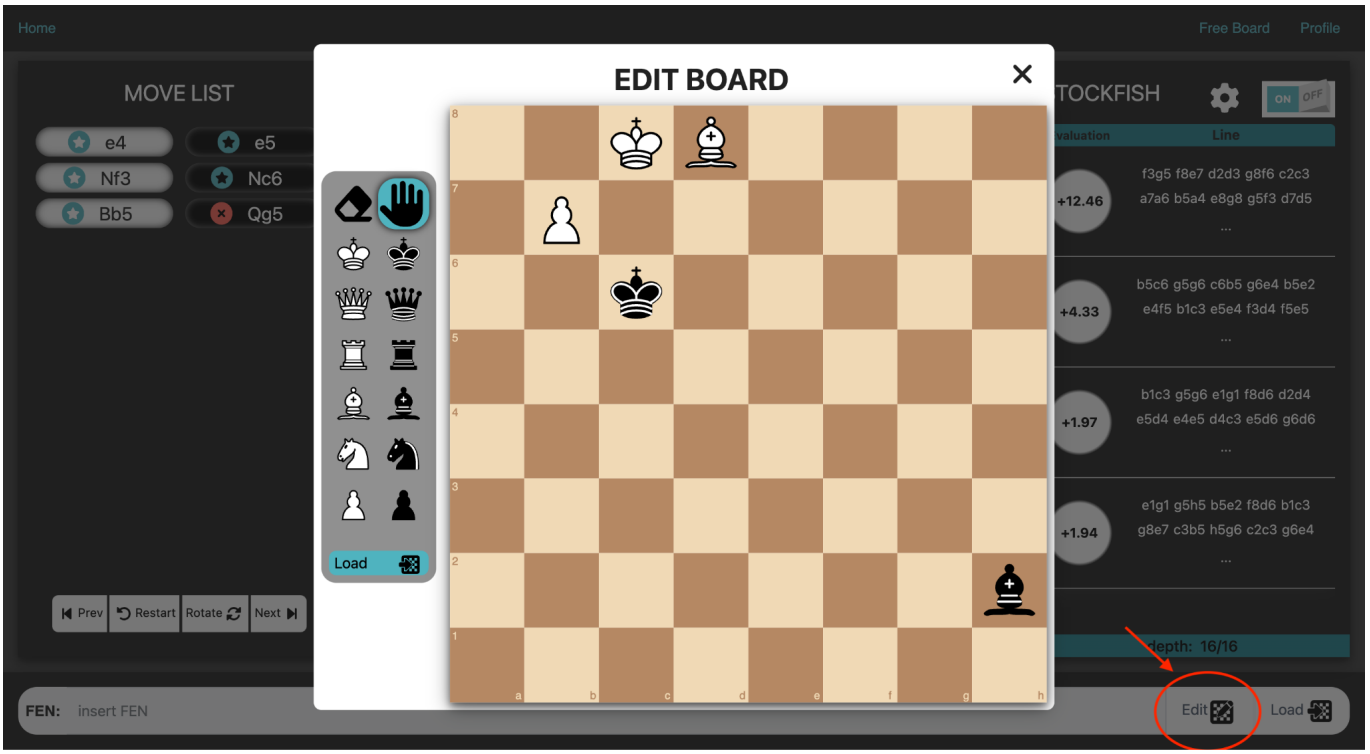
The Freeboard stands out as a visually appealing and user-friendly feature. It offers a fully interactive chessboard that allows users to freely navigate through moves, rotate the board, make their own moves, and explore Stockfish's suggestions at each step. The board also provides the flexibility to input a FEN string, customize the board using the edit button or load entire games as PGN. Additionally, users have control over Stockfish's behavior, with the ability to toggle it on or off and adjust parameters such as depth and the number of lines displayed.

Making a few moves

Rotating the baord



Loading a FEN string



Creating a custom position

Adjusting stockfish parameters

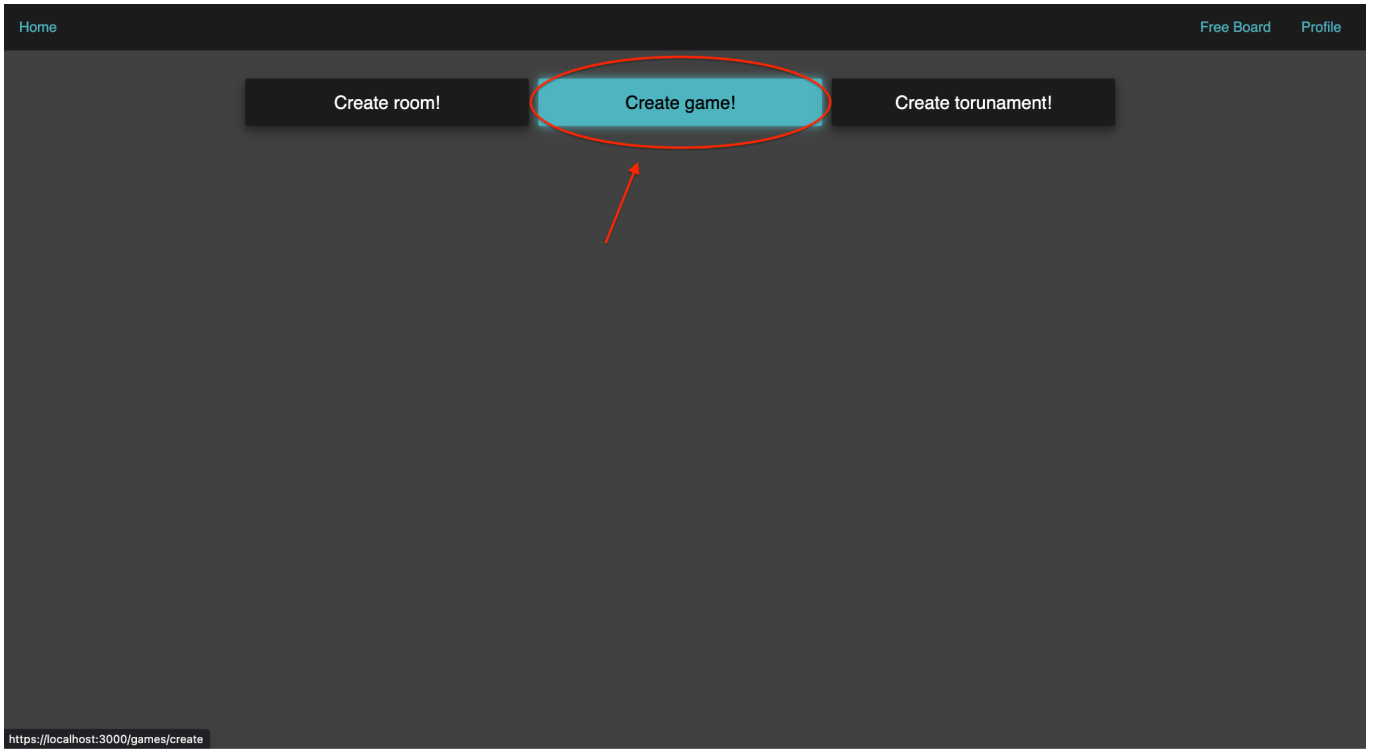
Toggling stockfish

Checking stockfish depth progress

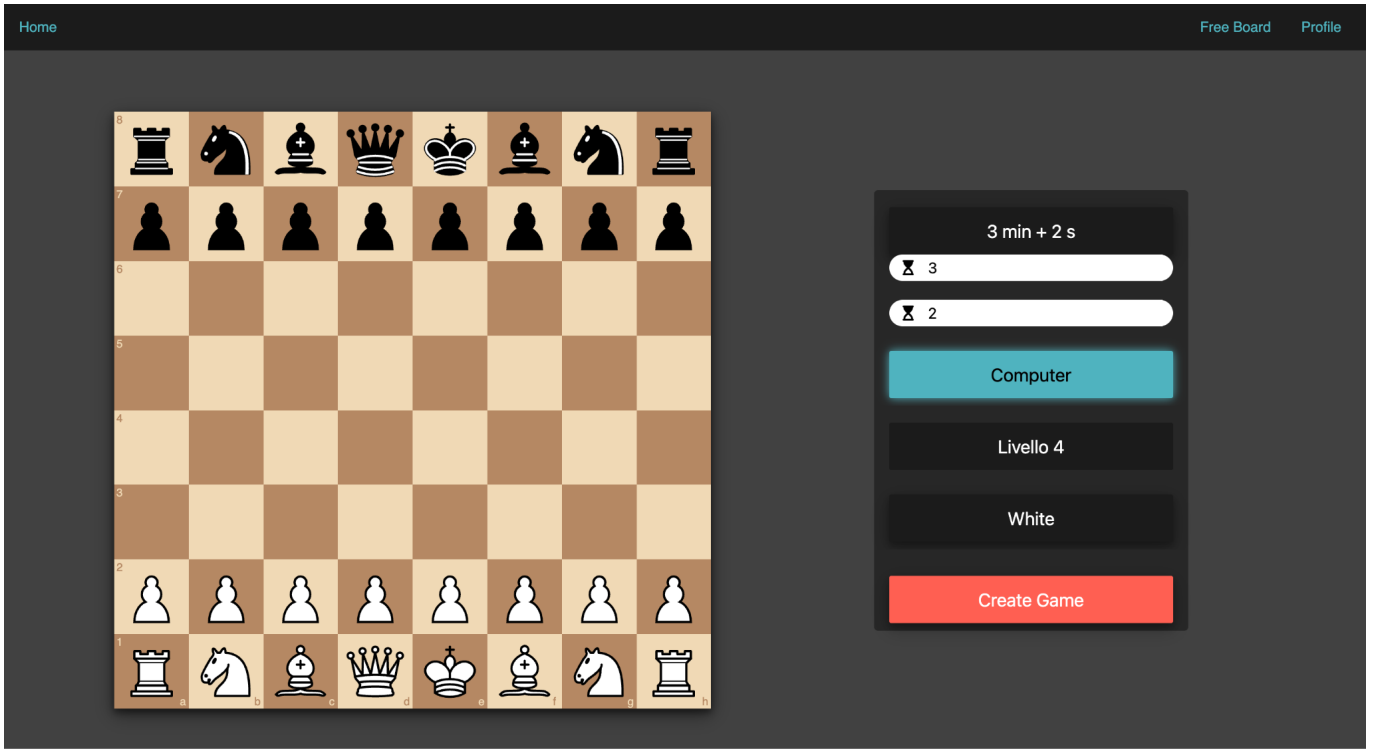
Hovering on a move to see stockfish suggestion

3.5.3 Gameplay

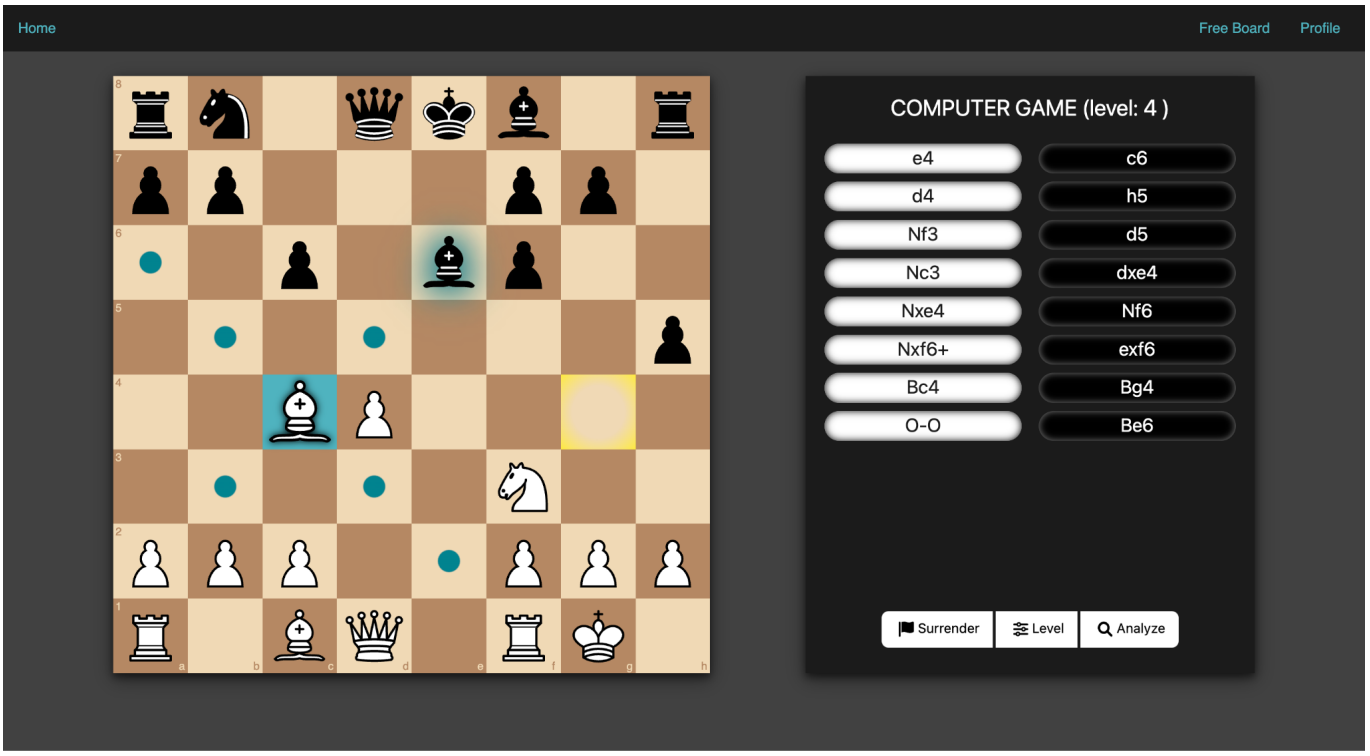
The gameplay features of the application cater to both player-versus-player (PvP) and player-versus-engine (PvE) scenarios. Users can engage in chess games with other players or challenge stockfish. The gameplay interface provides essential features such as a move list, allowing players to keep track of the moves made during the game. Additionally, users have the ability to surrender or propose a draw, providing options for game resolution.



Creating a game

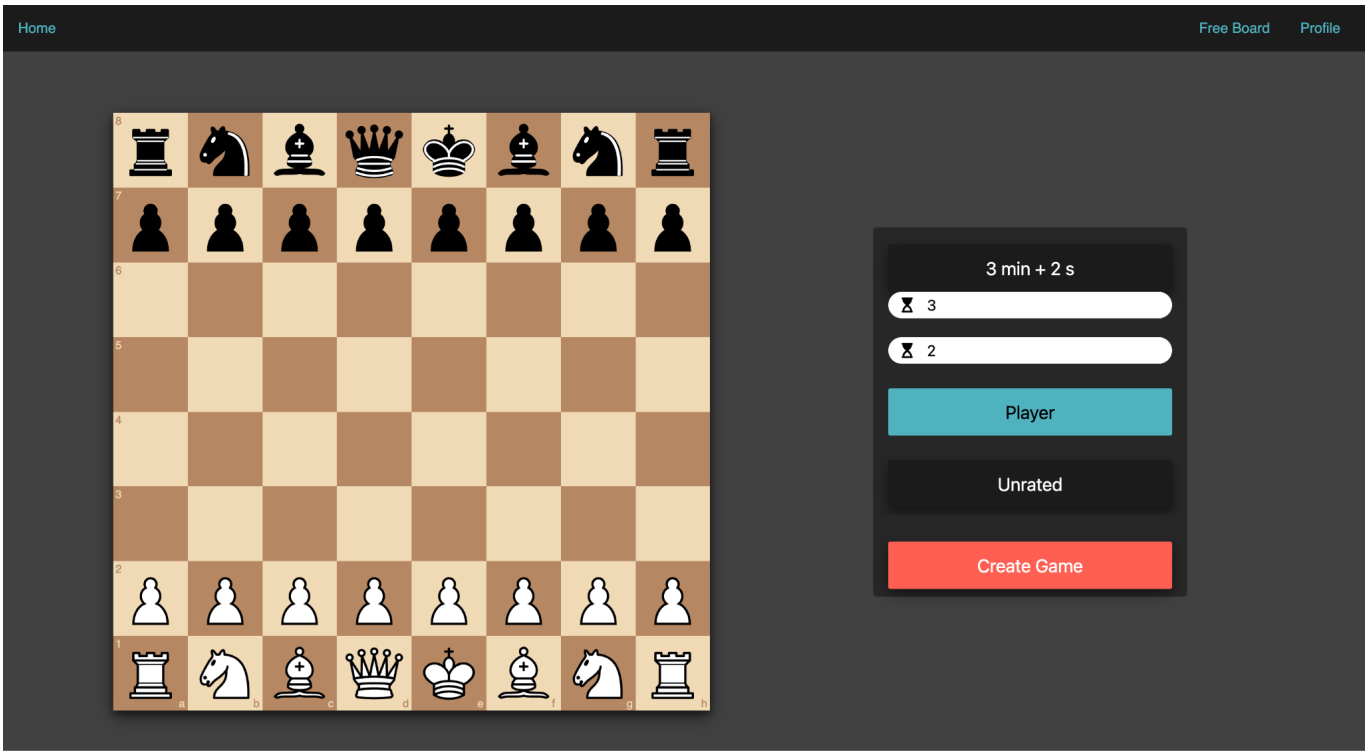


Creating a computer game

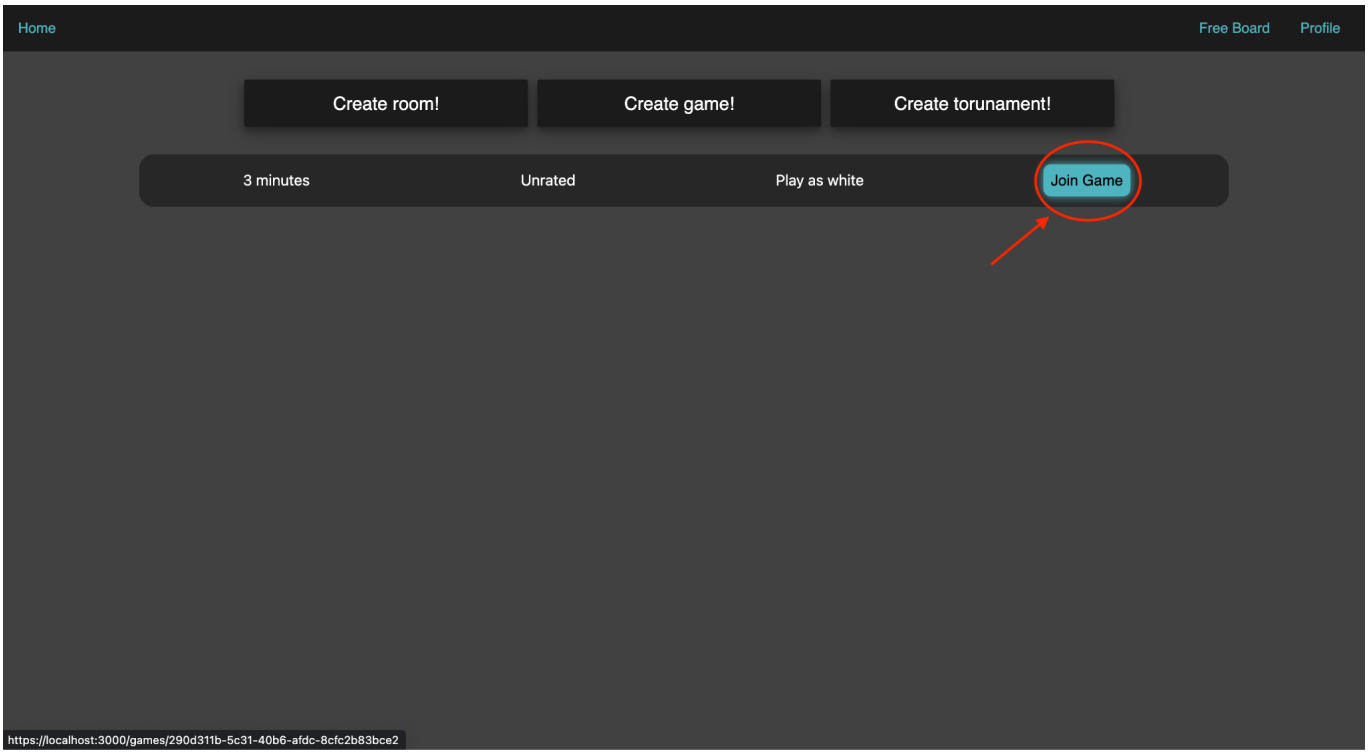


Playing a game against the computer

Surrending



Creating a player game



Joining a game

Playing a game againt a player

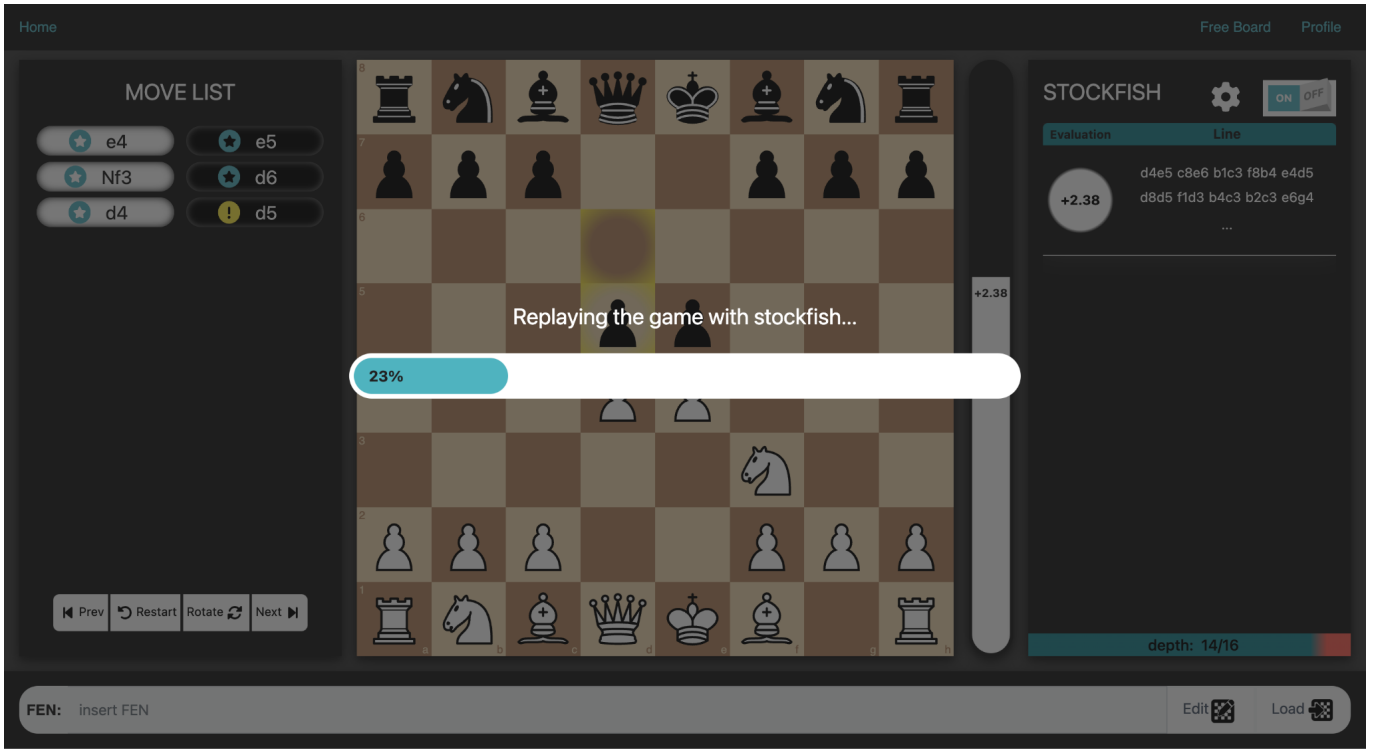
Offering Draw

Accepting draw offer

Game ended

3.5.4 Post game Analysis

One notable feature is the "analyze" button, which becomes available at the end of a game. Clicking this button enables users to load the game's move list into the freeboard which becomes a powerful analysis tool, allowing users to explore and review the game in detail, making use of Stockfish's suggestions and insights. This analysis feature enhances the learning and improvement aspects of the application, providing users with valuable post-game analysis capabilities.



Loading a game into the freeboard

Game Loaded

Undestanding mistakes

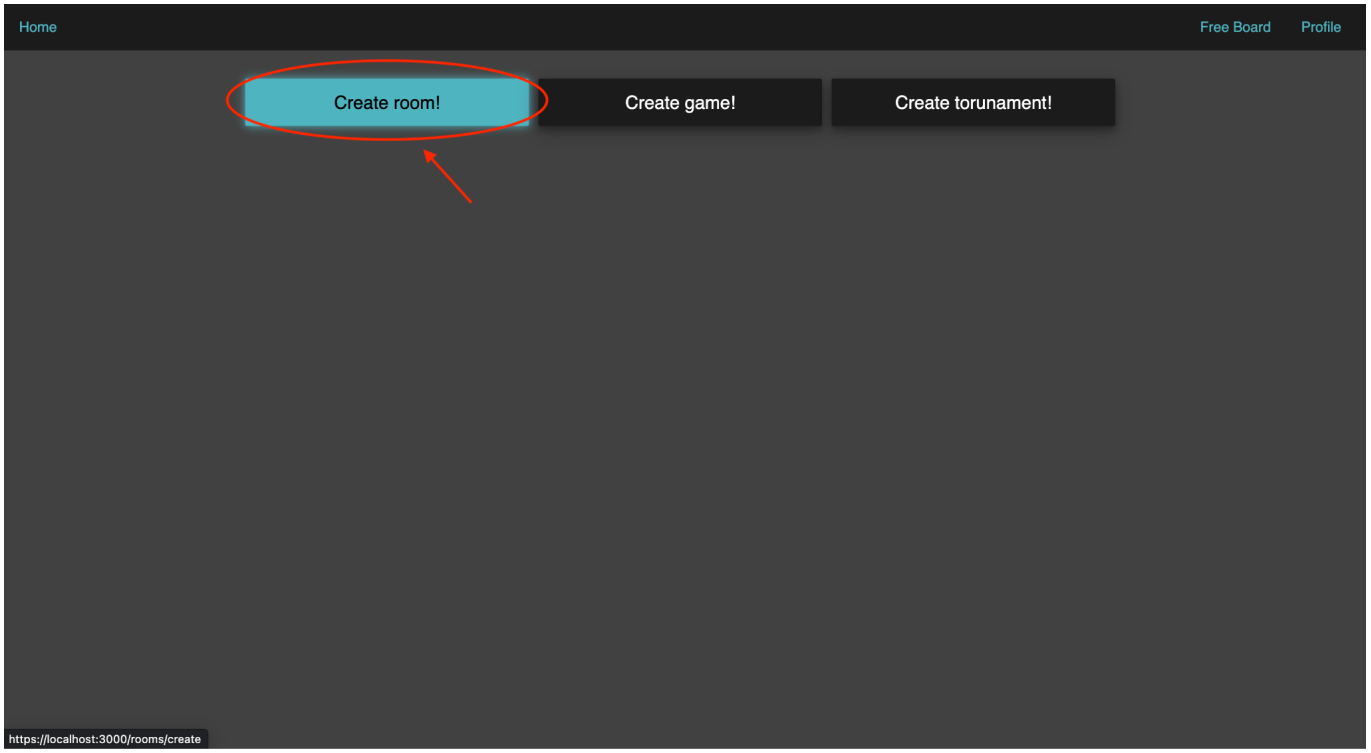
and looking at optimal moves

3.5.5 Rooms

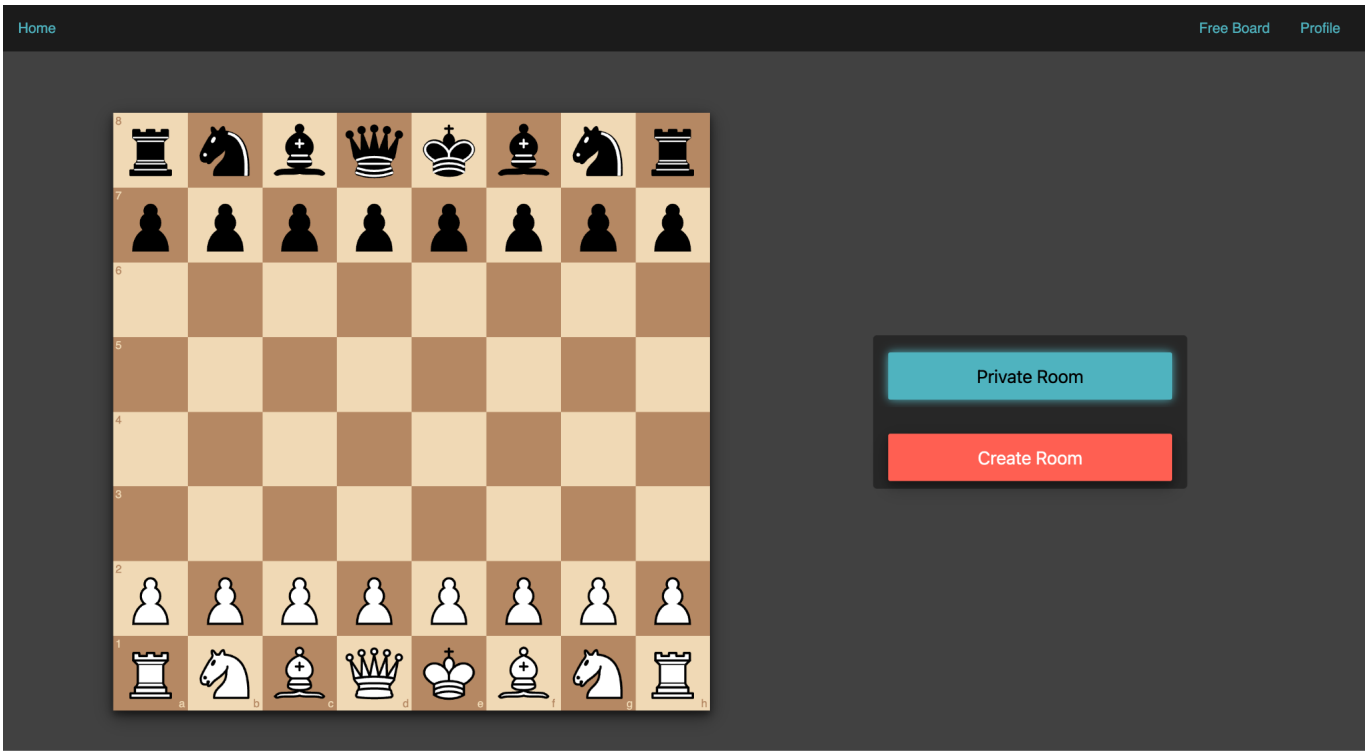
The room feature of the application provides a platform for users to create and join virtual meetings centered around chess discussions and lessons. This feature is particularly valuable for facilitating interactive learning experiences. Users can create both public and private rooms, with the room creator acting as the administrator. In private rooms, the administrator has the authority to accept or reject requests from individuals seeking to join.

Furthermore, the room administrator holds additional privileges, including the ability to mute audio and video for all participants and grant permission for users to make moves on the shared chessboard. The administrator can also load FEN strings or set up custom positions, similar to the functionality offered by the freeboard feature.

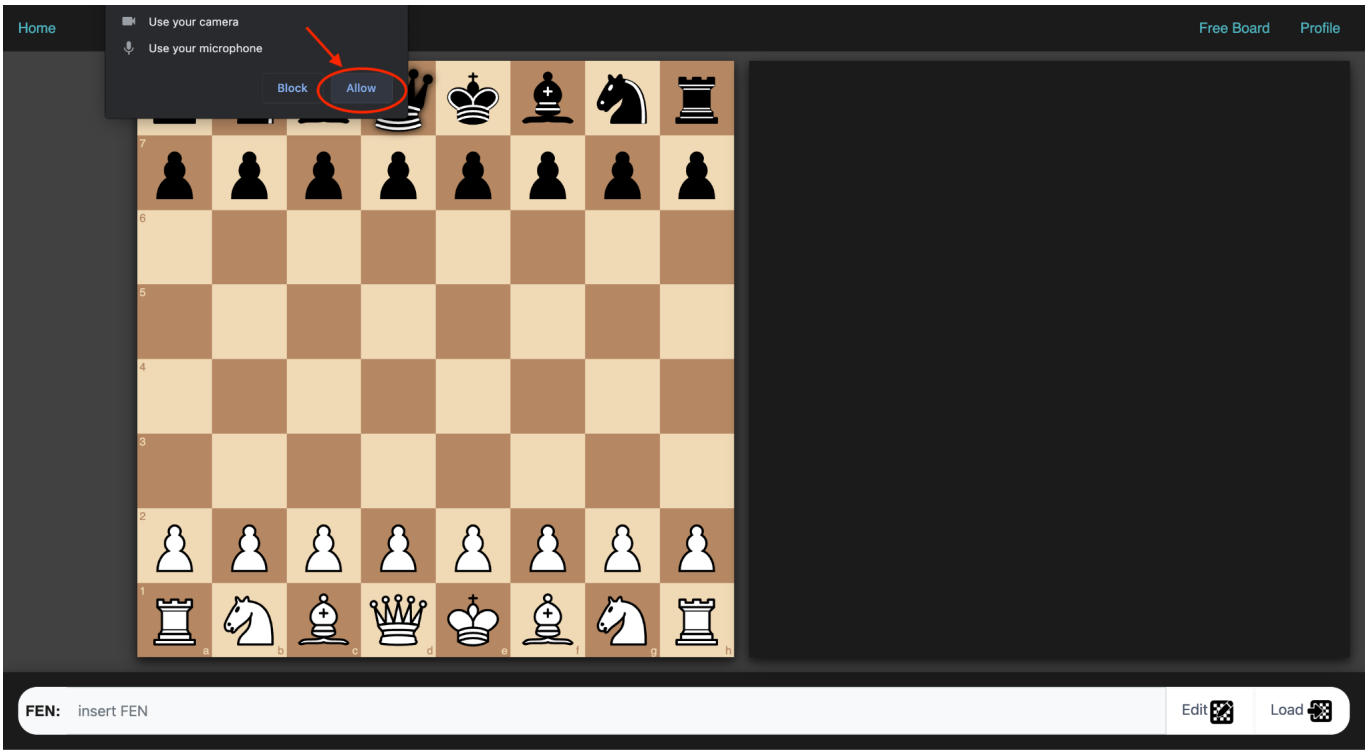
One notable aspect is the voting system, which allows the administrator to initiate a voting session from the current chess position. Each user participating in the room can contribute a single move to the vote. Once the voting concludes, the results are displayed, providing visibility into the number of votes each move received. This feature promotes engagement and collaboration among participants, fostering a sense of community within the chess room environment.



Creating a Room

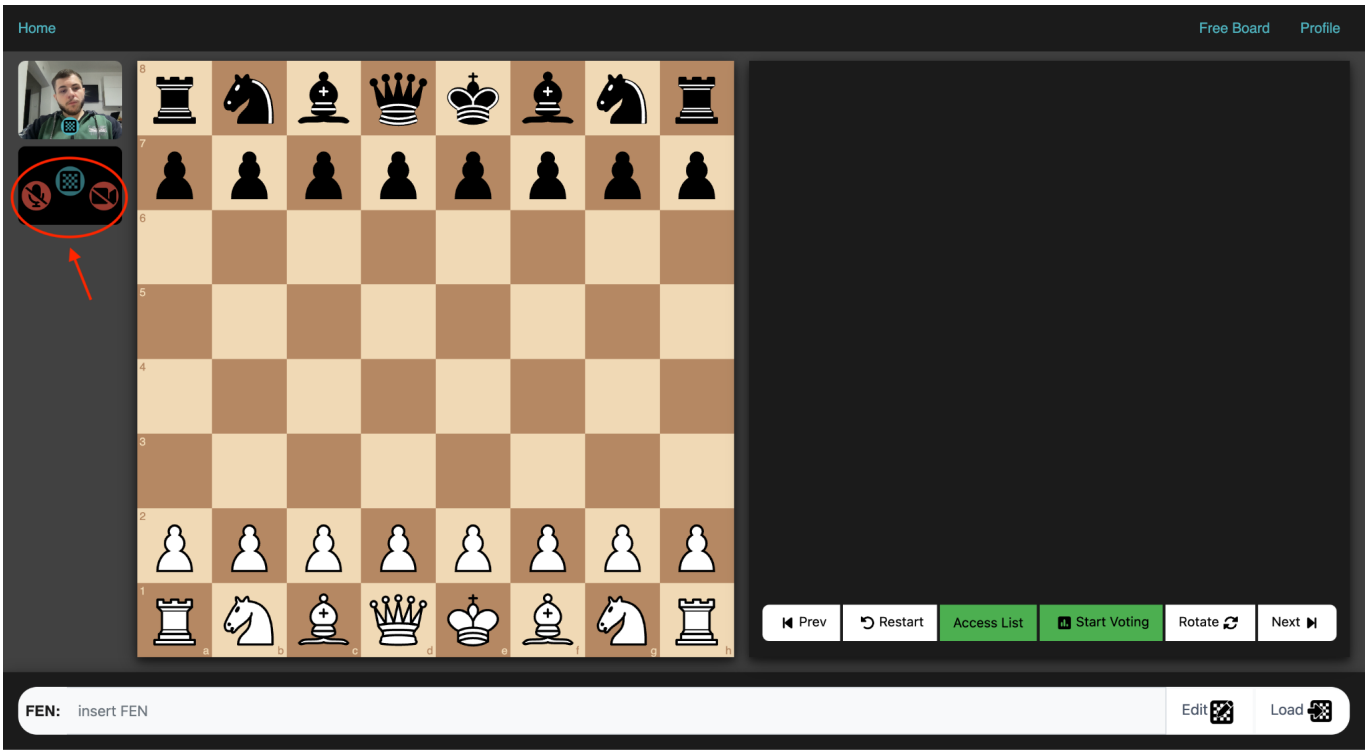


Creating a private Room

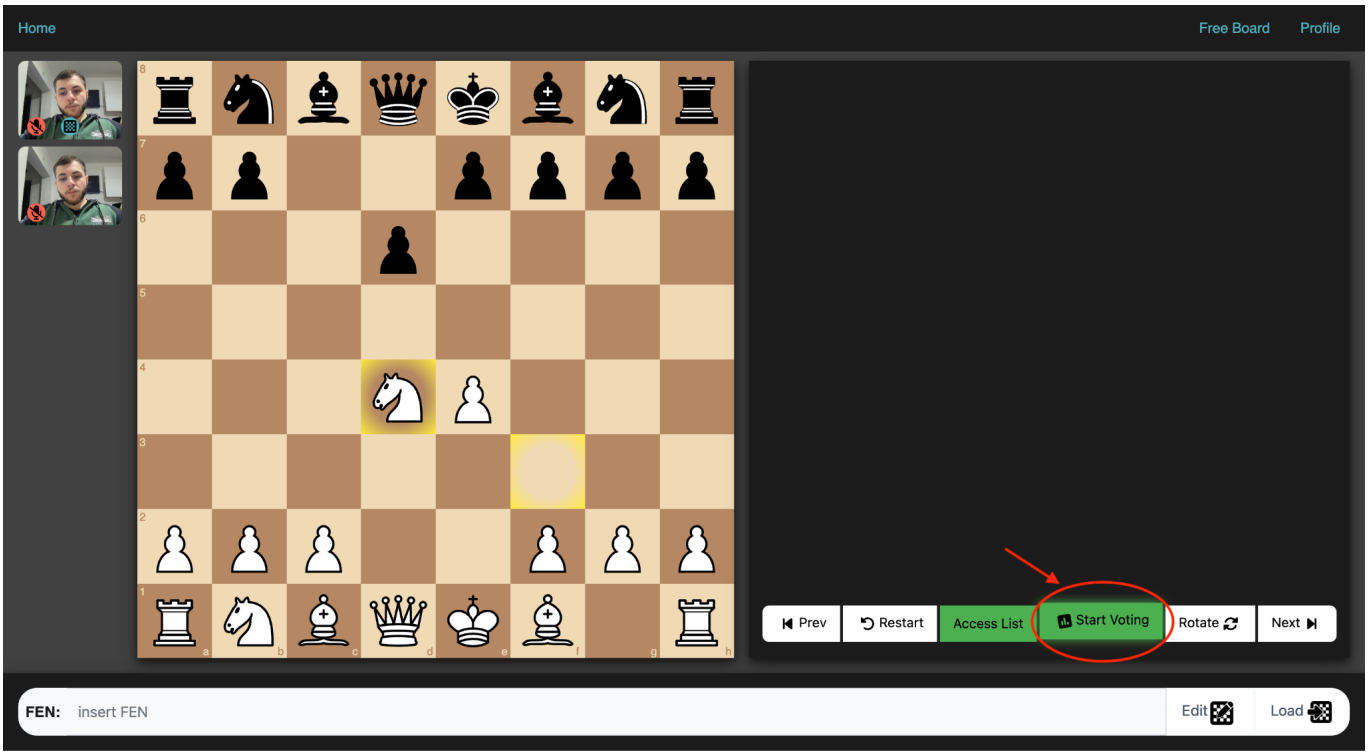


allowing browser to access microphone and camera

Granting access to another user



Managing permissions



Start voting

End voting