Sofia University „St. Kliment Ohridski“

Faculty of Mathematics and Informatics

Web Technologies

Head 2 Head Chess

Group 5

Borislav Gunovski	1MI0700119
Martin Hristov	5MI0700114
Martin Kanchev	0MI0700014
Yasen Uzunov	6MI0700123
Nikola Georgiev	0MI0700198

Project Overview

Head 2 Head Chess is a real-time multiplayer chess application that enables players to engage in live chess matches through a web interface. The application provides a complete chess playing experience with user authentication, game creation, real-time move synchronization, and spectator capabilities. Built with modern web technologies, match history tracking, and customizable user information.

The application follows a client-server architecture where the frontend communicates with a Node.js backend through both RESTful APIs and WebSocket connections. The backend leverages Express.js for HTTP request handling and Socket.IO for real-time bidirectional communication. Data persistence is achieved through PostgreSQL, with Knex.js serving as the query builder and migration tool. The frontend is implemented using vanilla JavaScript, providing a responsive interface that works across desktop and mobile devices.

System Architecture

The system employs a three-tier architecture consisting of a presentation layer, application layer, and data layer. The presentation layer comprises HTML pages, CSS stylesheets, and JavaScript files that handle user interactions and display game state. The application layer, built on Node.js and Express.js, manages business logic, user authentication, game state management, and real-time communication through Socket.IO. The data layer utilizes PostgreSQL for persistent storage of user information and game history.

The root directory contains the main application entry point and configuration files. Route handlers are organized in a dedicated routes directory, handling HTTP endpoints for page serving and API requests. Database models and migrations reside in their respective directories, with TypeScript providing type safety for database operations. Frontend code is organized by feature, with separate directories for different views, reusable components, utility functions, and static assets.

Real-time game synchronization is achieved through WebSocket connections managed by Socket.IO. When a player makes a move, the client emits a move event to the server, which validates the move, updates the game state, and broadcasts the updated state to all connected clients in the same game room. This event-driven architecture ensures low latency and consistent game state across all participants.

Database Design

The database schema centers around two primary entities: users and games. The users table stores account information including authentication credentials, profile data, and chess ratings. Each user has a unique identifier, email address, and username, along with personal information such as name, country, and optional biography. Password security is maintained through bcrypt hashing, and the system tracks when passwords were last changed for security auditing.

The games table captures the complete state and history of each chess match. Each game record includes references to both players, the timestamp of the match, the final result, and a complete move history stored as JSON. The state field indicates the winner or type of draw, while the result field provides more detailed information about how the game ended, such as checkmate, stalemate, or player disconnection.

The relationship between users and games is established through foreign key constraints, with each game referencing two users as white and black players. This design allows for efficient querying of user match history and supports features like displaying recent games on user profiles. The schema is managed through Knex.js migrations, ensuring version control and reproducibility across different environments.

API & Communication Protocols

The application exposes a RESTful API for user management and game operations. User-related endpoints handle registration, authentication, profile updates, and match history retrieval. The registration process validates user input, checks for existing accounts, hashes passwords, and creates new user records. Authentication compares provided credentials against stored hashes and returns user information excluding sensitive data. Profile management endpoints allow users to update their personal information, contact details, and passwords with appropriate validation and authorization checks.

Game management follows a hybrid approach combining traditional HTTP requests with WebSocket communication. Game creation is initiated through a POST request that generates a unique game identifier and redirects to the game page. Once on the game page, all real-time interactions occur through WebSocket events. The client emits events for joining games, making moves, and signaling game completion. The server responds with game state updates, player notifications, and error messages.

The WebSocket protocol defines several event types for game interaction. The joinGame event includes the game identifier, spectator status, and user identification. Move events contain source and destination positions using algebraic notation. Game end events specify the termination reason and winner. The server broadcasts game state updates to all connected

clients in the same game room, ensuring synchronized views of the board position, turn status, and move history.

Frontend Implementation

The frontend architecture emphasizes modularity and reusability through a component-based approach. The navbar component provides consistent navigation across all pages, dynamically updating based on authentication status. This component is loaded asynchronously on each page, demonstrating the application's approach to code reuse and consistent user experience.

Individual pages handle specific functionality within the application. The home page serves as the entry point, offering options to create new games, join existing games by code, or spectate ongoing matches. The game page implements the core chess interface, rendering the board as an HTML table with event listeners for piece selection and movement. Visual feedback includes highlighting legal moves and the last played move, enhancing user understanding of game state.

User account management pages provide comprehensive control over personal information, authentication credentials, and contact details. These pages implement client-side validation with real-time feedback, ensuring data quality before submission. The validation logic checks field requirements, format constraints, and password complexity rules. Modal dialogs handle confirmation workflows and success notifications, providing clear feedback for user actions.

Game Logic & Chess Rules

The chess engine implements standard chess rules including piece movement patterns, capture mechanics, and special moves. Each piece type has defined movement rules that generate potential moves based on the current board position. The system validates moves by checking piece ownership, movement legality, and whether the move would leave the player's king in check.

Special moves receive particular attention in the implementation. Castling is permitted when neither the king nor the relevant rook has moved, no pieces occupy the intermediate squares, and the king does not move through or into check. Pawn promotion automatically converts pawns reaching the opposite end of the board to queens. The current implementation does not include en passant captures, representing a known limitation.

Game termination conditions are continuously evaluated after each move. Checkmate occurs when a player's king is under attack and no legal move can remove the threat. Stalemate results when a player has no legal moves but is not in check. The system also recognizes draws

by insufficient material when neither player has enough pieces to force checkmate. These conditions trigger appropriate game end events and update the database with final results.

Security and Authentication

The authentication system prioritizes security through multiple layers of protection. Passwords undergo bcrypt hashing with a cost factor of 10 before storage, ensuring that even database compromises do not expose user credentials. Session management utilizes browser sessionStorage, providing isolation between tabs while maintaining ease of implementation. All database queries use parameterized statements through Knex.js, preventing SQL injection attacks.

Input validation occurs at multiple levels to ensure data integrity and security. Client-side validation provides immediate feedback for user experience, while server-side validation serves as the authoritative check before data processing. Validation rules enforce minimum lengths, format requirements, and character restrictions appropriate to each field type. The system validates email formats, enforces username uniqueness, and ensures password complexity through requirements for mixed case, numbers, and special characters.

Deployment & Production Considerations

Production deployment requires careful attention to security, performance, and reliability. Environment-specific configuration manages database connections, with separate credentials for development and production environments. HTTPS encryption protects data in transit, while proper CORS configuration controls cross-origin resource access. Rate limiting on API endpoints prevents abuse and ensures fair resource usage across users.

Performance optimization strategies include database connection pooling to manage resource usage efficiently. Static asset compression and browser caching reduce bandwidth requirements and improve load times. Monitoring systems track server health, database performance, active game sessions, and error rates to ensure reliable operation. Regular database backups protect user data and game history, with defined recovery procedures for disaster scenarios.

The development workflow emphasizes code quality and team collaboration. Version control through Git enables parallel development and feature branching. Code style guidelines ensure consistency across the codebase, with ES6+ syntax and meaningful naming conventions. Testing procedures cover user flows, game mechanics, and cross-browser compatibility to maintain application quality as features evolve.

Additional Resources

The complete source code for Head 2 Head Chess is available on GitHub at https://github.com/AJTheClear/head-2-head-chess, where developers can access the latest version, report issues, and contribute to the project. The repository includes all frontend and backend code, database migrations, and configuration files necessary to run the application.