TRIBHUVAN UNIVERSITY
INSTITUTE OF ENGINEERING

PASCHIMANCHAL CAMPUS
LAMACHAUR, POKHARA

SOFTWARE REQUIREMENT SPECIFICATIONS
On

Chess Engine

BY

SUJAN KHAREL [PAS079BCT043]

TO

Department of Computer and Electronics Engineering
POKHARA, NEPAL
August 4, 2025

Table of Contents

Chapter 1: Introduction

1.1 Background Theory

A chess engine is a program designed to play chess by analyzing positions and calculating the best move using algorithms and evaluation functions. Your system uses **Stockfish**, one of the most powerful open-source chess engines, to simulate intelligent bot opponents. Stockfish evaluates millions of positions per second and supports adjustable difficulty levels, enabling a scalable AI challenge for players.

1.1.2 Real-Time Communication with WebSockets

WebSockets provide full-duplex communication channels over a single TCP connection. This allows real-time interaction between clients and the server—essential for fast-paced applications like multiplayer games and chat. In your system, WebSockets are used to:

- Transmit moves between two players instantly.
- Update the game board in real time.
- Enable live in-game chat.

1.1.3 Multiplayer Game Systems

Multiplayer systems allow multiple users to interact with a shared environment simultaneously. Your chess system implements:

- Matchmaking (finding an opponent).
- Synchronous gameplay (each user sees the same game state).
- Chat (side-channel communication).

1.1.4 User Authentication and Game History

To manage user sessions and keep track of performance or past games, **user registration and login** are implemented. MongoDB stores:

- User credentials (safely hashed).
- Game histories.

- Chat messages.

This ensures users can resume or review games later.

1.1.5 MongoDB for Data Management

MongoDB is a NoSQL database well-suited for applications that require flexible schemas and real-time updates. It's used here to store:

- User profiles
- Game records (moves, results)
- Chat messages
  This enables scalability and easy data retrieval.

1.2 Purpose

The purpose of this project is to design and implement an interactive, web-based chess application that allows users to:

- **Play chess in real time** against other users.
- **Challenge a bot** powered by Stockfish at different difficulty levels.
- **Chat with opponents** during a game.
- **Track their games and progress** through persistent user accounts and saved histories.

The project showcases the integration of modern web technologies (Next.js), real-time data transfer (WebSockets), and artificial intelligence (Stockfish) to create an engaging online gaming experience. It also provides practical exposure to full-stack development, real-time systems, and AI integration in games.

1.3 Scope

This project focuses on developing a **web-based multiplayer chess application** that allows players to compete in real-time either against other human players or a computer-controlled bot. It also supports **real-time chat**, **game history tracking**, and **adjustable bot difficulty**.

1.4 Overview

The development process follows modern full-stack and event-driven architecture, integrating frontend, backend, real-time protocols, and AI components. The system is modular and scalable, allowing easy deployment and further enhancements.

- **Requirements Analysis:** Identified the key use cases — real-time gameplay, chat, bot integration, and user management.
- **Frontend Development:** Built using Next.js with React for UI and server-side capabilities for API handling.
- **Real-Time Engine:** WebSockets used to synchronize board states, moves, and chat messages across clients.
- **Bot Integration:** Stockfish is used as the chess engine to power computer opponent play.
- **Database:** MongoDB stores user information, game data, and chat logs.
- **Authentication:** Implemented secure login/register using hashed passwords and session tokens.
- **Testing:** Covered unit tests, socket event testing, and integration tests for game flow.
- **Deployment:** Ready for cloud deployment using Vercel or Docker-based setup.

Chapter 2: Overall Description

2.1 Product Perspective

The Chess System is a **modular web application** that combines a frontend interface, a backend server with real-time communication, a bot engine, and a persistent database. It can run as a standalone product or be embedded into larger gaming or educational platforms.

It consists of:

- **Next.js frontend/backend** for UI and REST/WebSocket APIs
- **WebSocket-based event system** for multiplayer synchronization
- **Stockfish engine module** for AI gameplay
- **MongoDB database** for storing all game-related data

This modularity allows for future integration of features like ranking systems, tournaments, or leaderboards.

2.2 Product Functions

The major product functionalities include:

- User registration and login
- Multiplayer matchmaking
- Chat system for in-game communication
- Chess game interface (with rules enforced)
- Bot gameplay (Stockfish integration)
- Difficulty selection for bot (e.g., depth control)
- Game state saving (moves, outcome)
- Chat history saving
- Display of previous games

2.3 User Characteristics

The system is designed for users with basic knowledge of:

- Chess rules
- Web usage (mouse, forms, buttons)

Target users:

- Casual online chess players
- Learners who want to practice vs bots
- Friends who want to play and chat in real time

No technical or chess engine expertise is needed.

2.4 Constraints

- Requires an active and stable internet connection for real-time multiplayer and chat
- Stockfish engine must be properly installed or accessible on the backend
- Server must handle concurrent socket connections smoothly
- Data security and authentication must be enforced (e.g., hashed passwords)
- The game must synchronize state accurately between clients in real time

2.5 Assumptions and Dependencies

- The client uses a modern browser (Chrome, Firefox, Edge)
- MongoDB instance is active and accessible
- WebSocket server is running and properly routing events
- Stockfish binary is installed or available via API
- Node.js environment is available to run the server

Chapter 3: Specific Requirements

➢ Functional Requirements

The following are the primary functional requirements of the FormalNet system:

FR1. The system shall allow users to register and log in securely.

FR2. The system shall allow users to play chess against another online user.

FR3.The system shall allow users to play against a bot with selected difficulty.

FR4.The system shall enable real-time updates of game state via WebSocket.

FR5. The system shall support in-game chat using WebSockets.

FR6. The system shall store user data ,game moves ,and chat logs in MongoDB

FR7. The system shall allow user to resign or request rematch

FR8. The system shall handle turn-based logic and validate legal chess moves.

➢ Non-Functional Requirements

These are constraints on how the system performs:

NFR1. The system shall respond to WebSocket events within 300 milliseconds under normal load

NFR2. The application shall support at least 100 concurrent users.

NFR3. The UI shall be responsive across desktop and mobile devices

NFR4. User data shall be securely stored with hashed passwords..

NFR5. Game data and chat logs shall persist after the session ends.

NFR6. The system shall use HTTPS to ensure secure communication.

➢ External Interface Requirements

User Interface

The user interface shall be developed using **Next.js (React)** and will include the following components.

- ❖ **Authentication Pages:**
  - ➢ Login and Register forms with field validation
  - ➢ Error handling (e.g., duplicate email, weak password)
- ❖ **Game Lobby:**
  - ➢ Options to play vs player or vs bot
  - ➢ Difficulty selection dropdown for bot game
- ❖ **Multiplayer Game Room:**
  - ➢ Real-time chessboard view (drag and drop functionality)
  - ➢ Chatbox for player communication
  - ➢ Game control options (Resign, Offer Rematch)
- ❖ **Bot Game Interface:**
  - ➢ Same as multiplayer game board, without chat
  - ➢ Displays AI moves and move suggestions (optional)
- ❖ **Profile/Dashboard Page:**
  - ➢ View game history
  - ➢ Replay past games

        - ➢ View win/loss stats

- ❖ **Responsive Design:**

    - ● Layout adjusts to mobile, tablet, and desktop screen size

API Interface

The system uses REST APIs to handle user registration, login, profile, and game history. When a user signs up or logs in, the frontend sends a request to the backend API, which checks the credentials and sends back a response. Logged-in users can also request their profile information or view past games. All the APIs send and receive data in JSON format. These APIs help the frontend and backend stay connected and keep the user data updated.

In addition to REST APIs, the system also uses WebSockets for real-time gameplay and chat. This allows instant communication between users during a match — moves and chat messages are sent and received without refreshing the page. The WebSocket connection handles all in-game actions like making a move, chatting, resigning, or requesting a rematch.

Chapter 4: Diagrams

4.1 ER Diagram

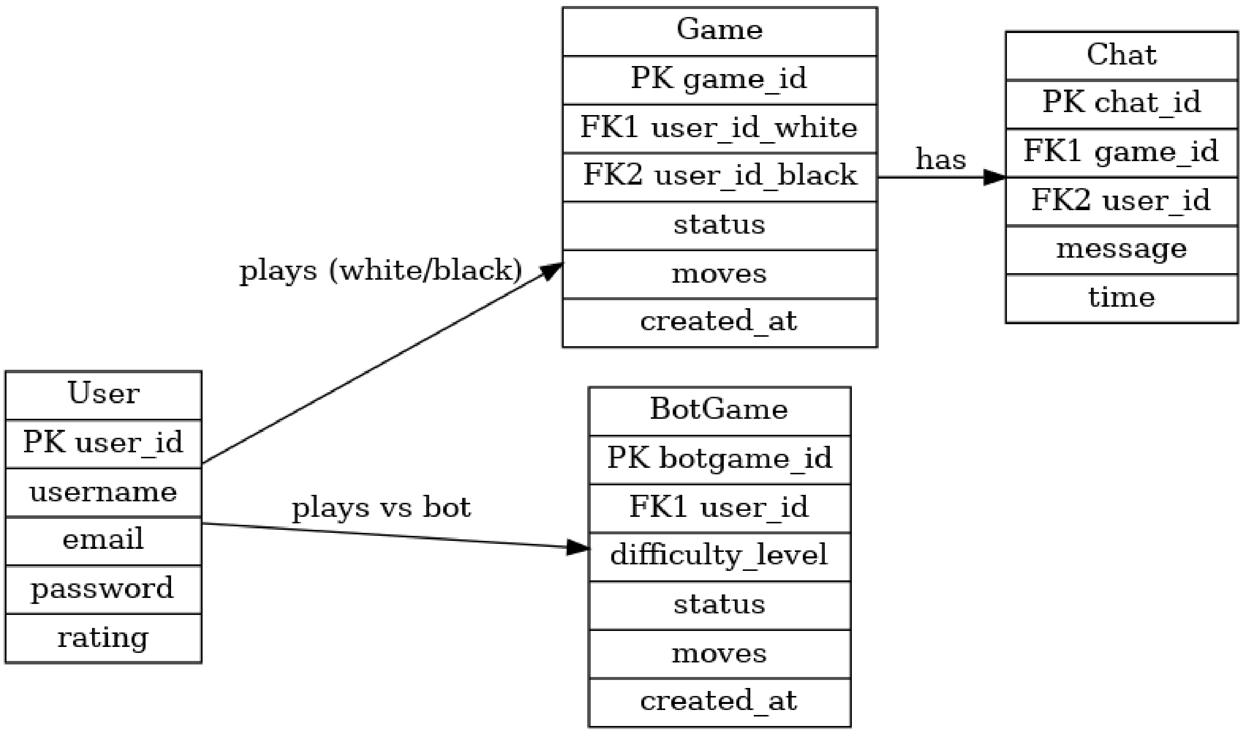

Figure 4.1: Entity-Relationship Diagram of ChessEngine

The Entity-Relationship (ER) Diagram of the chess engine system models the core entities and their relationships in the application. Key entities include **User**, **Game**, **Chat**, and **Move**. A **User** can participate in multiple games (as Player 1 or Player 2), and each **Game** consists of several **Moves**. The **Chat** entity is linked to a game, enabling players to exchange messages during multiplayer matches. The diagram also represents relationships like one-to-many between User and Game, and Game and Move. This ER model is essential for designing the MongoDB schema and ensuring efficient data organization across the system.

4.2 Dataflow Diagram

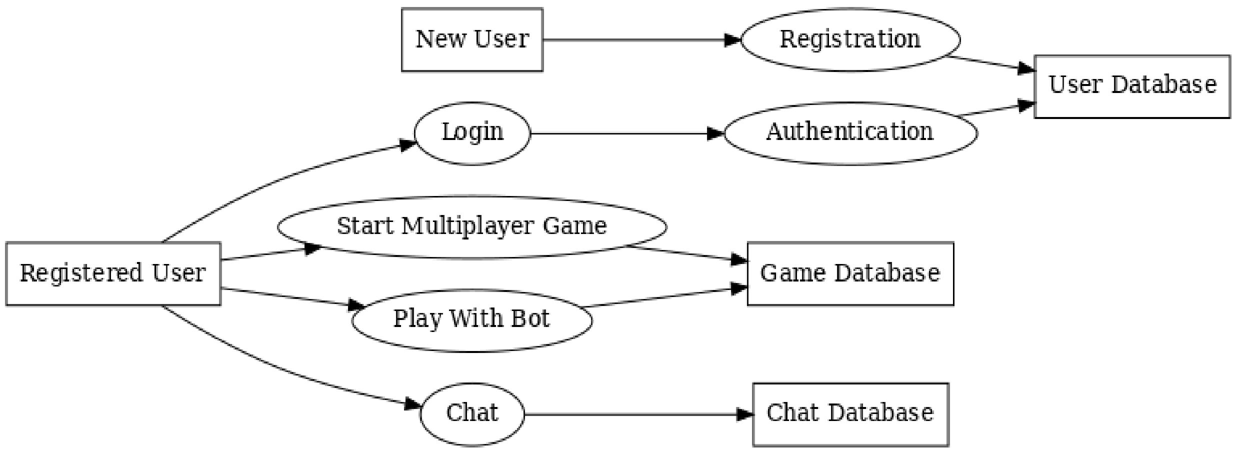

Figure 4.2: Dataflow Diagram of ChessEngine

The Data Flow Diagram (DFD) for the multiplayer chess system illustrates the movement of data between users, system processes, and data stores. It provides a high-level overview of how data flows through the system during user interactions, such as registration, gameplay, and chatting.

The main processes include:

- **User Registration**: New users submit their information, which is stored in the user database.
- **Login and Authentication**: Registered users log in and are authenticated using the stored credentials.
- **Start Multiplayer Game / Play With Bot**: Users initiate real-time gameplay either against other players or against the Stockfish bot.
- **Chat During Game**: Players can exchange messages during multiplayer games.
- **Data Storage**: User information, game data, and chat messages are stored in the respective databases (User DB, Game DB, Chat DB).

This DFD provides a clear visualization of how real-time interactions (via WebSocket) are connected with backend processes and persistent storage (via MongoDB).

4.3 State Diagram

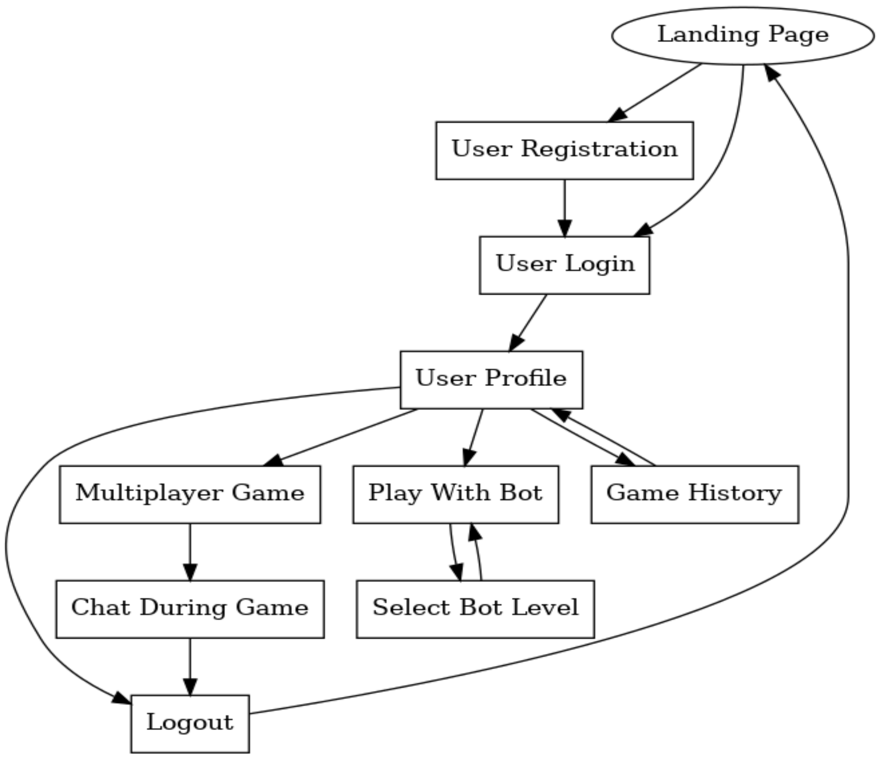

Figure 4.3: State Diagram of ChessEngine

The State Diagram of the multiplayer chess system illustrates the different states a user may experience during their interaction with the application. It begins at the landing page, where users can either register or log in. After successful login, the user is directed to their profile from where they can choose to play a multiplayer game, play against the bot, or view their game history. Bot gameplay includes a state for selecting the difficulty level. During multiplayer games, users can also engage in real-time chat with their opponents. The user can log out at any time, returning the system to the initial state. This diagram helps visualize system behavior and transitions triggered by user actions.

4.4 Use Case Diagram

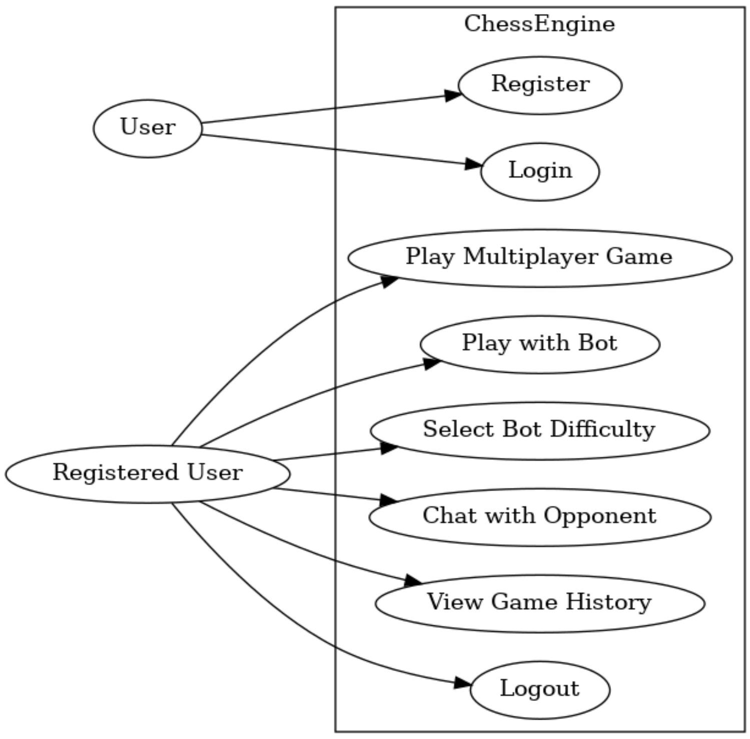

Figure 4.4: Use Case Diagram of FormalNet

The Use Case Diagram of the chess engine system represents the main interactions between users and the system. It identifies two types of actors: the general user and the registered user. General users can register or log in, while registered users have access to core features like playing multiplayer games, playing against the bot, selecting bot difficulty, chatting with opponents, viewing game history, and logging out. Each use case represents a functional goal the user wants to achieve through the system. The diagram helps in understanding system boundaries and ensuring all expected functionalities are addressed during development.

Refresences

[1] Next.js Documentation, “Next.js - The React Framework,” 2024. [Online]. Available: https://nextjs.org/docs

[2] MongoDB Inc., “MongoDB Manual,” 2024. [Online]. Available: https://docs.mongodb.com/manual/

[3] M. Müller, “Integrating Stockfish Chess Engine into Web Applications,” *Journal of Open Source Software*, vol. 4, no. 39, 2019.

[4] Fette and A. Melnikov, “The WebSocket Protocol,” RFC 6455, 2011. [Online]. Available: https://tools.ietf.org/html/rfc6455