Open Chess Software Documentation

William Stenberg

2020-05-03
Version 0.2

1 Introduction

This document outlines the software architecture for the web-based application *Open Chess* that offers an interactive chess board through a React Application served with Flask. Using the application (playing a game of chess through mouse interaction), the user is encouraged to explore the early-game strategies of the game of chess. The backend of the application (written in Flask, with a MongoDB database) internally evaluates positions and assigns them scores. The evaluation process is performed by Stockfish, a chess computer engine. Stockfish is used to assign scores to board positions fed to it, representing a move's soundness in the game. The scores for the initial and resulting board states in a move form the weight of the move, taking into account other avaliable moves in the initial position.

The application defines *theory*, which are moves in certain positions of the board that are known to be good, typically generated by parsing a precompiled set of games in the Polyglot format, or by reading in PGN files of Grandmaster games. Moves that are known by the application are shown as arrows color-coded by their score: A good move is green, a bad move is red, and a theory move is blue. Their score as evaluated by Stockfish are available as arrow tooltips. Moves that have been added (through parsing Polyglot or PGN) but not yet evaluated are white.

Making a move, by dragging and dropping the pieces in the frontend client, updates the position with a fetch call to the backend. Making an unknown move will trigger an analysis of the position, showing a loading spinner during the few seconds of analysis time.

The app is intended for use as a training tool, prompting the player to make good moves in pre-determined board positions. This documentation will showcase the features of the software and its modes of usage, along with a technical description of the database and client-server communication. The Open Chess source code is on `https://github.com/WilliamStenberg/open-chess`, and is Free Software.

2 Features

This chapter will give an overview of the user-centric features that are implemented in the software. As such, it will concern the user interface components in the frontend. We will start off with the picture in Figure 1, which shows a large number of features that will be explained further.

Figure 1: A view of the software in a typical position

We will start the discussion by a quick walk-through of the panels: Top-left are the *game modes* where the user selects between exploring and practising (see Section 2.3). Below are the *favorites* where the user can load a prerecorded board position. The right hand panel with title *details* is dependent on the game mode, here it shows a list of the arrow-indicated *suggestions* with options for the focused suggestion arrow of knight to f3. The bottom panel holds buttons to step the game forwards and backwards, analyse the current position, and flip the board (to see it from Black's perspective).

The following details will also give which shortkeys are available, which simulate button presses.

2.1 Basics

The board is used by dragging and dropping pieces. When a piece is moved, the move is sent to the server to be checked for validity. On initialization, the frontend client will receive the SVG objects that form the board, and assign mouse handlers to them.

Whenever a move has been played, either by the player or by the engine, the user can step back (shortkey **left arrow**). When a backing has been made, the move can be replayed by the forward-button (similarly, shortkey **right arrow**). Whenever a new move is player, the forward-stack is emptied and the new move is pushed to the backing stack. The board can at any time be flipped by the bottom-panel button without further change to the functionality. Internally, this only involves repositioning of the SVG elements. Shortkey **f**.

The analysis button will conduct a more thorough analysis and add new moves to the existing position.

2.2 Favorites

The second panel on the left-hand side are the user's *favorites*. These are positions that the user has put on the board, written a name for in the text input field, and saved with the checkbox button. These are meant to be used for positions that the user often is interested in.

The system will reject a name that already exists as a favorite, as well as a position that already exists with another name.

Clicking an item in the favorite list will load the position and populating the backing stack so that the user can load a position but still walk backwards in the move order history.

2.3 Game modes

This version supports two game modes: Explore and Practise. These are switched by clicking the top-left panels and by shortkeys **e** and **p**, respectively. The Explore mode is intended to try out different moves and show what moves are recorded in the database along with their scores (evaluations by the Stockfish engine). The practise move will not show what the database knows about a position, but wait for the user to make a move. It will then check the played move against the database, and only accept moves that are good enough.

2.3.1 Explore mode

In Figure 1, the game is in exploration mode. This means that arrows of available moves are shown and color-coded by their scoring. Green arrows indicate bght etter moves, and blue ones are theory (as introduced in previous chapters). The user can hover over an arrow to show its score (seen with the move Nc3). When an arrow is clicked, or through the list of arrows in the right-hand panel, it is focused and can be unlinked (shortkeys **u/backspace/delete**); removed as a possible move from the board.

If a move is played that is not known (i.e. does not have an arrow), the system will analyse it. It will then be added as a new arrow in the

future. This takes some time, the system limits the analysis to two seconds for analyses made through playing a move in explore mode.

2.3.2 Practise mode

In practise mode, the arrows are not shown. After the user plays a move, it will be checked against the list of existing moves. If there are theory moves, the user is required to play one of them - otherwise the move will be rejected as inferior. If there is no theory for the position, the user is required to have played the known move with the highest score.

The user can request a *swap* of the computer's move, prompting it to select another of the known moves for the position. The user may also *reject* the computer's move, which performs a swapwhile also unlinking the move so it will not be played again in the future.

Below in Figure 2 we see the same position with white to play. Making a legal and good move, in this case any of the theory moves seen in Figure 1, would prompt the computer to answer with its own move.

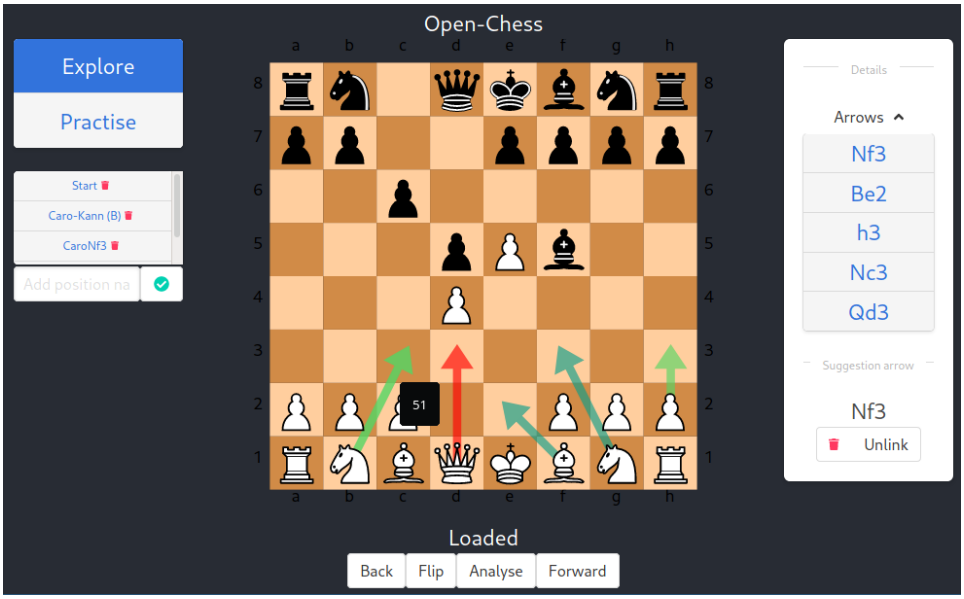

Figure 2: The system in Practise mode

3 Communication

The frontend communicates through `fetch` calls, making RESTful HTTP requests to the backend, typically situated on the same server. The Flask server runs from `backend/server.py`, where all endpoints are defined.

Each call is made through POST, and accepts a dictionary of parameters

as input. The expected inputs are somewhat standardised e.g. when taking a move as input, the endpoints all expect a key named `move` containing a four-letter move UCI: the square to move from, concatenated with the square to move to.

In the cases with more variation, at this point in time we refer to the flask definitions as mentioned.

The backend holds an active board and related context-dependent computations in `backend/motor.py`. Here, the board is kept as well as a cursor to the document in the database that is currently active. Hence, the motor can fetch which moves are known (as theory or other moves) easily. All interactions with the database are however relegated to `backend/database.py` where inserting, fetching and removing are done, typically returning a document (in the form of cursor) or a boolean status.

In addition to the context-holding motor, the `backend/utils.py` allows a developer to troubleshoot the database, add moves and boards from sources such as PGN files or Polyglot files. From here a *crawling evaluation* can be started from a given position, doing a recursive walk with a maximum depth, evaluating every position with Stockfish.

4 Database layout

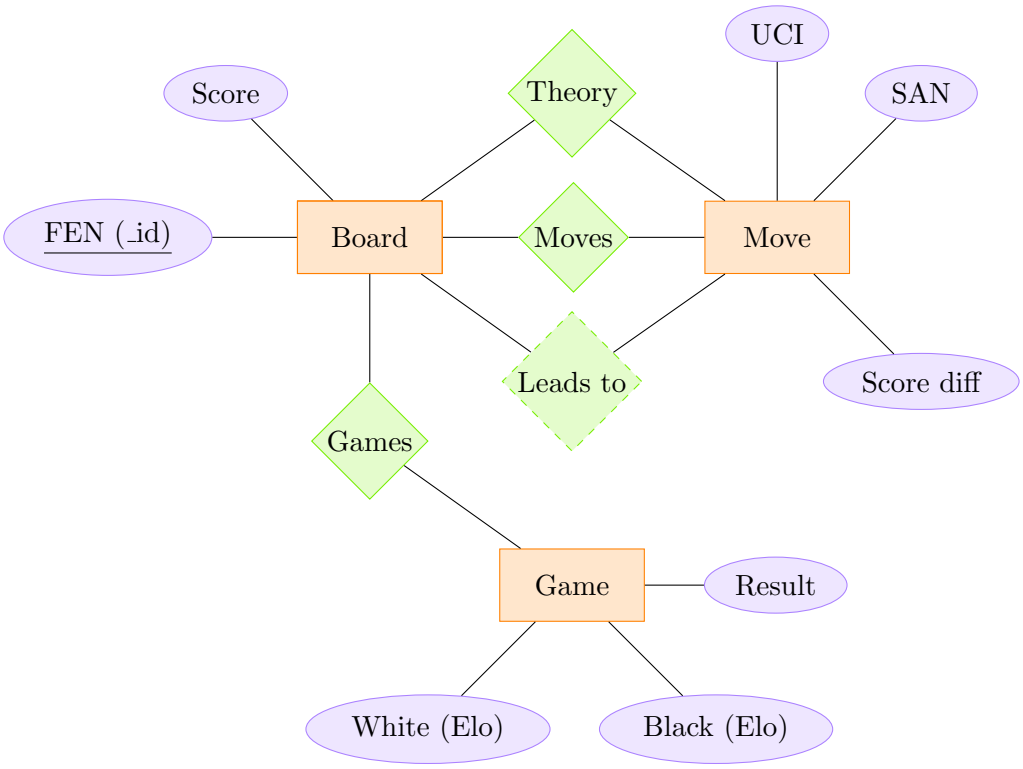

Figure 3: Entity-Relationship (ER) diagram for *open-chess* database

The entity-relationship diagram in Figure 3 outlines the structure of the database, implemented as MongoDB documents. The method of drawing the diagram comes from a paper on TikZ diagrams [1]. Below, each of the elements are explained in terms of their intended use in the application:

- **Board**: Representing a chess board position, i.e. a configuration of pieces. The board is populated with known moves, and when analysed also a score. When adding games by PGN, boards can also hold information about what games they were seen in. This is interesting to those trying to study certain strong players.
  - **FEN** (index): A board is uniquely defined by its FEN string (Forsyth-Edwards Notation) which is computed by taking into account piece positions and encodes which player is to move next. This is used as the index, allowing for quick lookups.
  - **Score**: A board's score is its numeric evaluation by a chess engine - who's winning. A score is computed from a player's perspective

to a centipawn (hundredths of a pawn-up-advantage). A positive score with White to move indicates their advantage, as does a positive score for Black with Black to move.

  - **Moves**: An array of objects representing legal moves from the board position. Contains the move's UCI, SAN, score difference, and the board the move leads to, by FEN.
  - **Theory**: An array of Move objects that are separated to be identified as theory moves, verified by high-level play.
  - **Games**: A list of ObjectIDs to Game objects, see below.

- **Move**: Not in its own collection in the database, these objects are used to transition between boards.
  - **UCI**: The UCI representation of the move, comprised of the square the moved piece starts and lands on, concatenated to a four-letter string, e.g. "e2e4", "b1c3".
  - **SAN**: The SAN representation of the move, which is more commonly used by humans when talking about moves, e.g. "e4", "Nc3".
  - **Score difference**: Centipawn difference between board positions before and after the move. A positive score difference indicates a sound move, while a negative means the move might be a mistake or even a blunder.
  - **Leads to**: FEN of the resulting board after the move is made; The relationship between a move and the board it leads to. This is used to traverse the tree of moves and boards, using lookings on the Board collection by FEN.

- **Game**: A subset of PGN game information. Contains the black and white players and their respective Elo ratings, and the outcome of the game as a string out of "1 - 0", "0 - 1", and "1/2 - 1/2".

References

[1] Claudio Fiandrino, *Drawing ER diagrams with TikZ.* `https://www.guitex.org/home/images/ArsTeXnica/AT015/drawing-ER-diagrams-with-TikZ.pdf`. Downloaded 2020-05-03.