DESIGN DOCUMENT

Group 12

MCMASTER CAPSTONE

MONICA BAZINA-GROLINGER
FARZAN YAZDANJOU
ANANT PRAKASH
NIDA NASIR
PHILIP LEE

Table of Contents

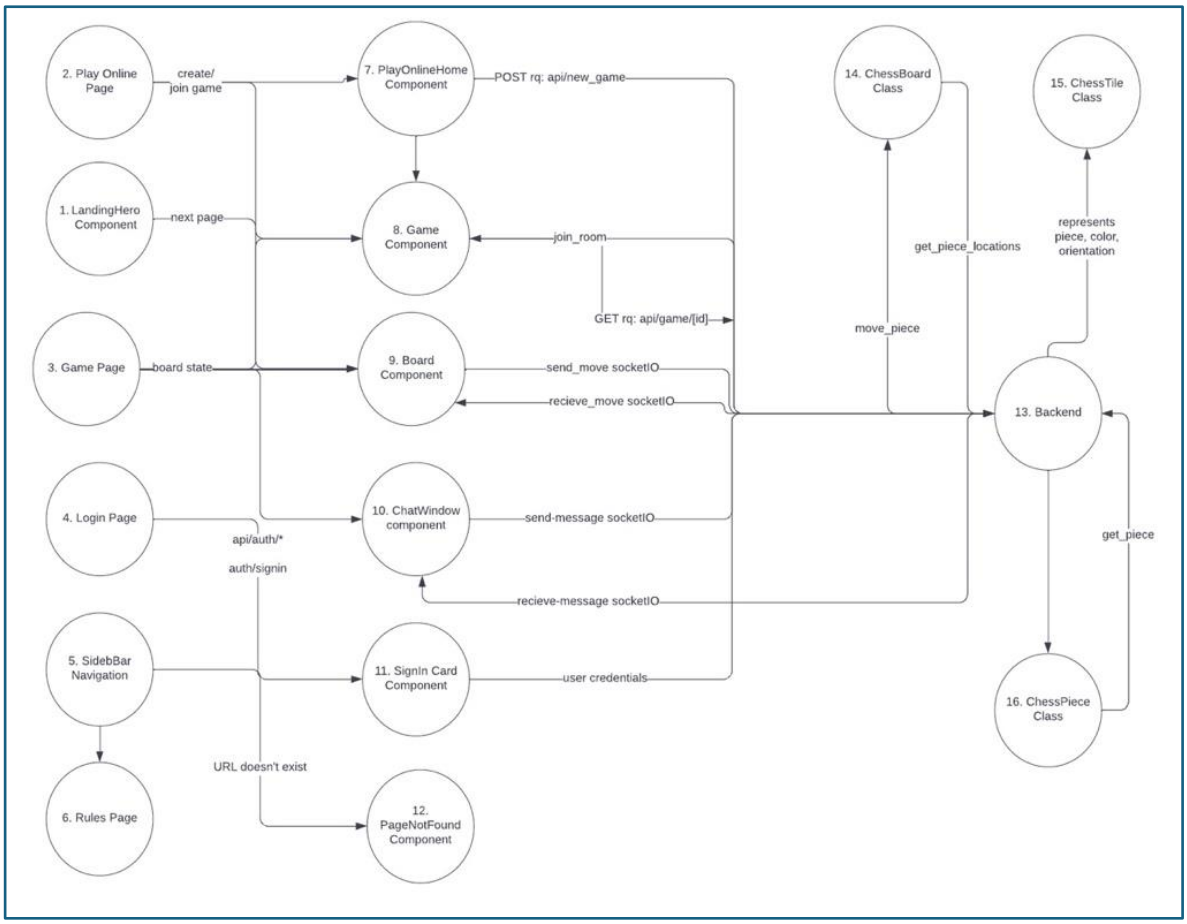
1. Controls / Versions

Date	Version	Developer(s)	Change
January 26, 2024	1.0	Entire Team	Initial Design Doc

2. Purpose Statement

The purpose of our chess variant project is to create an innovative, engaging, and challenging experience by introducing numerous gameplay elements that provide a new twist on the traditional game of chess.

3. Diagram of Components



4. Relationship between Components and Requirements

Users can use their existing credentials to log in		**P0**
Related Routes	`/api/auth/*, /auth/signin`	
Related Components	`SignInCard.jsx, AuthProvider.jsx`	
To implement this common requirement, we are using `NextAuth` and `Prisma` to facilitate the authentication process. `NextAuth` provides a framework for session authorization and ensuring that credentials are correct, while `Prisma` is used to store user information in our database. We created a custom page for logging in to our application, using the `SignInCard` component - the user is redirected to this page upon clicking ‘Sign In’ on the app or attempting actions that require an account.		

Users can start a new game			**P0**
Related Routes	`play/online`	Related Components	`CreateGame.jsx`
Related API Endpoints	`/api/new_game`	Related Backend	`create_game(...)`

Implementing this requirement requires full-stack considerations. On the back-end, we have created an API endpoint that generates a new game. On the front-end, we have created a page that allows any logged-in user to create a game.

The API endpoint takes an optional password for the game, the data of the user creating the game along with their chosen colour (black or white). The API then stores a new game with the default state of the game, the creator as `player_1` along with their colour and a temporary entry of `player_2` with the opposite colour. The front-end takes the user's inputs and hits said endpoint, receives the game object, and redirects the user to the game.

Allow users to click on a piece and make a move	**P0**
Related Routes	`/game/{code}`
Related Components	`Board.jsx, Tile.jsx, Piece.jsx`
Related API Endpoints	`/api/game/<game_id>`
Related Backend	`ChessPiece, ChessTile, ChessBoard`

Implementing this requirement requires full-stack considerations. On the back-end, we have created an API endpoint that allows users to make moves in their game. On the front-end, we have implemented logic to allow client-side moving of pieces on the board.

The API post endpoint requires the last and new positions of the piece being moved, and uses this information to manipulate the `Board` object stored in the backend. Once the move is made, the API returns the new state of the board. We will soon implement sophisticated logic to this endpoint to ensure the correct player is making the move and if the move is valid.

Using `onClick` events, the front-end allows users to click individual tiles on the board to select and move pieces between tiles. At the moment, there is no confirmation per move, but we will add a confirmation dialog for moves as required by the supervisor. Once the tiles have been clicked, the front-end makes a call to the backend API as described above. Using the received new game state, the front-end re-renders the `Board`.

Allow users to choose their opponent	**P0**
Related Routes	`/play/online, /game/{code}`
Related Components	`JoinGame.jsx`
Related API Endpoints	`/api/join_game/<game_id>`

Our initial idea was to display a list of all open games on the Play Online page and allow users to select to join any of them. However, we voted against this idea as we foresee many issues with this approach regarding the security and efficiency of the implementation.

Instead, we will allow users to join a game using the game code that is generated upon game creation. This code is displayed in the URL and will be displayed on the page for easy sharing. Users can either directly navigate to the URL or join through the "Play Online" page using the game code. Both methods require that the user is authenticated. If a password is set on the game, the joining user will be prompted to enter it. This logic is handled by the `JoinGame.jsx` component on the front-end, which hits the API endpoint upon form submission.

Display board with the pieces on each side				**P0**
Related Routes	`/game/{code}`			
Related Components	`Board.jsx, Tile.jsx, Piece.jsx, Game.jsx`			
Related API Endpoints	`/api/game/<game_id>`			
Related Backend	`ChessPiece, ChessTile, ChessBoard`			
Implementing the game board requires both front-end and back-end considerations. On the back-end, we came up with a design for storing and manipulating the state of the board. The board is stored as an array of arrays - where the parent array represents the entire board, and each child array represents a column of tiles. As the directions of the tiles change across rows, we thought it'd be best to store and modify them by the column. Each column stores the list of pieces from top to bottom - with an empty string representing an empty tile. Pieces are their own class - storing their colour and name. We have implemented various API endpoints that return the state of the game board. We go over these all in the following sections. However, let's consider the basic `GET /api/game/[id]` endpoint which returns the current state of the board. When loading into a new game, the front-end hits this endpoint. The front-end then takes the game board information and passes it to the `Board` component. The `Board` parses the array of arrays and renders `Tile` components for each string it comes across – along with an image of the piece in the correct colour. The `Board` is also given information about what colour the current user is – and it flips the board accordingly so both players have their side facing themselves.				

Allow users to send messages to each other during the game				**P0**
Related Routes	`/game/{code}`			
Related Components	`ChatWindow.jsx, Game.jsx`	Related Backend	`join_room(…)`	
Related SocketIO	`handle_join_room, handle_send_message`			
Allowing users to communicate within the game is a functionality requirement that we expected. We did not have to reinvent the wheel for this feature and went with a design that the user would be familiar with. We designed a simple chat box that sits in the bottom right corner of the game page. The users can enter messages into the input textbox and hit send. The front-end connects to the back-end with socket.io which allows continuous connections between both clients and the server to exchange instant information - such as messages.				

Manage user authentication		**P0**
Related Routes	`/api/auth/*, /auth/signin`	
Related Components	`SignInCard.jsx, AuthProvider.jsx, NextAuth`	
As briefly covered before, user authentication is managed on the front-end with `NextAuth` - a library for `Next.js`. Using the `getServerSession` or `getSession` hooks, depending on if the component is rendered on the client or server, we have access to authentication information everywhere in our application. This is useful as we block certain actions in the app as they require an account - such as playing online. Furthermore, we pass user information to the backend to properly store and access game information. Eventually, we will implement security measures in the back-end as well, to extend this authentication.		

Store game state		**P0**
Related Components	`Board.jsx, Tile.jsx, Piece.jsx, Game.jsx`	
Related Backend	`ChessPiece, ChessTile, ChessBoard, create_game(...), games: dict[str, dict] = {}`	
As covered before, the game's board is stored in the back-end and rendered on the front-end. However, in addition to the board, we store various information regarding the game that the backend can modify and the front-end uses to provide the game experience. For example, the game state stores information regarding both players such as their user id, name, and game colour - this ensures users make moves for their side and no more than two players join the game. Furthermore, the password along with other optional parameters for the game are stored on the game object. As such, the back-end defines a game as a dictionary - with individual key/value pairs for such information. In various instances, the APIs send back this data to the front-end as well, to ensure correct behavior when rendering and playing the game.		

5. Component Behaviour

5.1 Existing Pages

Landing Page (Page)	
Components	`LandingHero, Default Board`
Normal Behavior	The Landing Page is the first page the user sees when first directed to the website. The buttons on the site will direct the user to the correct next page.
Implement	This page uses the `LandingHero` component and the `Board` component which makes up the webpage. The `Board` component is being used as a visual and its state has been set to disabled.
Potential Undesired	The page components may visually look incorrect when the page window is resized to extremely small sizes or the site may be incompatible with certain web browsers.

Play Online (Page)	
Components	`CreateGame (Password), JoinGame (Code, Password)`
Normal Behavior	Allows the player to create a new game or join an existing game and will direct the user to the login page if they are not already logged in. The buttons will redirect the user to the correct next page and return feedback if any part of the process fails (ex. invalid game code).
Implement	This page uses the `PlayOnlineHome` Component which makes up the webpage.
Potential Undesired	The page components may visually look incorrect when the page window is resized to extremely small sizes or the site may be incompatible with certain web browsers. The page may freeze on a certain event where the backend fails to respond in a reasonable time frame.

Game Page (Page)	
Components	This page consists of the `Board` and `ChatWindow` components, retrieving and passing in the necessary data to both components.

Normal Behavior	Allows the user to interact with the board and shows the real time board game state between two players. When it is the player's turn, they are able to select a piece and move it to locations that are legal according to the game rules. Additionally, the player may communicate with the other player using a chat box by entering text and sending said text.
Potential Undesired	The page components may visually look incorrect when the page window is resized to extremely small sizes or the site may be incompatible with certain web browsers. The game page could potentially desync with the server game board state and cause issues with the connection to the backend.

Rules (Page)	
Normal Behavior	A regular HTML page that contains the text which explains all the Official rules for the game RhombiChess.
Implement	Raw HTML code containing all the relevant text needed for the rules page was included. All piece images are imported from the `public` directory.
Potential Undesired	The page may visually look incorrect when the page window is resized to extremely small sizes or the site may be incompatible with certain web browsers.

Login (Page)	
Components	`SignInCard`
Normal Behavior	Allows a user to sign into their account using their email and password. Additionally provides the user with the ability to be logged in and redirected to another page if they do not know their password. Once the credentials have been filled in, the authentication service verifies the credentials.
Implement	This page uses the `SignInCard` component as well as some additional page layout CSS for various visual effects.
Potential Undesired	The page may visually look incorrect when the page window is resized to extremely small sizes or the site may be incompatible with certain web browsers. The authentication service may fail which might mean that the user gets stuck and is not able to proceed as they will not be able to access game play.

5.2 Existing Components

`Board.jsx`	
Components	`Piece, Tile (Board consists of Tiles, Tile contains Piece)`
Normal Behavior	This component contains the implementation of the board with the specific pieces and sets up any requests that need to be sent to the backend to fetch updates based on the boards' changed state on each players turn.
Implement	The `Board` component takes in props `initialBoard`, `gameCode`, `disabled`, `socket` and `color`. The `useState` is used to track the current selected piece on the board as well as the board state. A hook has been used to set up a listener for the `recieve_move` event on the socket and will update the board state when received. The component sends a `POST` request to the server containing the old and new positions of a moved piece and emits a `send_move` event to inform

	other connected clients of the move made. When a piece on the board is in a selected state, the next selected tile is the location the piece will attempt to move to. Otherwise, it does nothing until a board piece is selected. This is made under the assumption that it is the player's turn to make their move.
Potential Undesired	Network latency or disconnection can lead to desynchronized game states between different players or between the client and the server. Continuous state updates and re-renders in a game can lead to performance issues making the interface laggy, particularly for lower-end devices.

`ChatWindow.jsx`	
Normal Behavior	This component allows for communication between two players. When someone sends a message in the chat, the message feeds of both player's messages are updated.
Implement	The `ChatWindow` takes props `gameCode` and `socket`. The socket listens for any `receive_message` updates and when triggered, updates the chat history for both players. When a player sends a message, the `send_message` function is triggered and sends an update via the web-socket.
Potential Undesired	If the web-socket connection fails, this feature will fail to function.

`LandingHero.jsx`	
Normal Behavior	Landing page component that provides the user with buttons to navigate to other pages and give a short introduction to the site.
Implement	The `LandingHero` Component uses raw HTML to implement all of the needed information as well as CSS for any relevant designs. The `PlayOnlineButton` and `PlayLocalButton` Components are used to abstract the logic needed for the buttons.
Potential Undesired	The component may visually look incorrect when the page window is resized to extremely small sizes or the site may be incompatible with certain web browsers.

`PlayOnlineHome.jsx`	
Components	`CreateGame, JoinGame`
Normal Behavior	Component that allows the user to create a new game or join an existing game by entering in a game code. The site will redirect the user to login if they have not already done so.
Implement	The `CreateGame` and `JoinGame` components are combined to build this.
Potential Undesired	The component may visually look incorrect when the page window is resized to extremely small sizes or the site may be incompatible with certain web browsers.

`SideBar Navigation`	
Normal Behavior	Available on all pages. Component that allows for sidebar navigation of the site. It provides a list of pages that the user can navigate to.

Implement	The sidebar navigation is used to wrap the entire application and is therefore displayed on all pages. The 'account' dropdown menu visually updates with the account name that is currently logged in. This component displays the game logo at the top, as well. Basic react component.
Potential Undesired	The component may visually look incorrect when the page window is resized to extremely small sizes or the site may be incompatible with certain web browsers.

`SignInCard.jsx`	
Normal Behavior	Component that allows for a user to authenticate and sign into the site. It is expected to handle the events where a user submits their login information or redirect them to recover their account in the event they forget their password.
Implement	The `SignInCard` component consists of fields for email and password. There are three event handlers which deal with the email text being changed, password text being changed and action to take when the form is submitted. The `handleSubmit` function asynchronously verifies that the given information is correct (using the `NextAuth signIn` function) and will otherwise inform the user that their email or password is incorrect.
Potential Undesired	The component may visually look incorrect when the page window is resized to extremely small sizes or the site may be incompatible with certain web browsers. The authentication service may not respond, meaning the user is unable to login and therefore cannot play.

`CreateGame.jsx`	
Normal Behavior	Handles the logistics for when a logged-in user creates a new game. Allows you to choose your side and set a game password – both optional.
Implement	The `CreateGame` component contains a simple form for creating a new game – this includes fields for the game password, and the creator's chosen side. The `handleCreateGame` asynchronous event handler facilitates the creation of a new game by submitting a `POST` request with the requested game data to the `/api/new_game` endpoint.
Potential Undesired	The component may look incorrect when the page window is too small. The game creation event may not respond, meaning the user will not be able to create the game and will have to retry.

`JoinGame.jsx`	
Normal Behavior	Component that allows for a user to enter a game code and join an existing game. If the game does not exist, a prompt will inform the user of so.
Implement	This component consists of 2 fields – the game code and password. Upon form submission, the `handleJoinGame` event submits a `GET` request to the `/api/game` endpoint with the game code to check if the game exists in the backend. If the game requires a password, the password box will show up and request the user enter the password for the game. If the correct password is submitted, the user is redirected to the game – otherwise, error messages convey the issue.
Potential Undesired	The component may look incorrect when the page window is very small. The game service may not respond which means the user cannot join the game and will have to retry the action.

5.3 Backend: board (Piece, Tile, functions)

`Class ChessPiece`	
Normal Behavior	This class represents the chess piece. It is expected to hold the name and colour of a piece.
Implement	It has one getter `get_piece` which returns the piece along with its colour in the from `{piece}-{colour}` as a string. It does not expect any inputs. When creating a piece, we initialize the piece with a name (a `string`) and a colour (as an `int`, 0 is black and 1 is white).
Potential Undesired	If non-string or non-integer values are passed to the constructor, it could lead to unexpected behaviour. The `get_piece` method assumes that colour is either 0 or not 0, which could be problematic if other values are used.

`Class ChessTile`	
Normal Behavior	This class represents a single tile on the chess board. It holds a `ChessPiece` instance that represents the piece on that tile, the colour of the tile, as well as the orientation of the tile.
Implement	The constructor takes a `ChessPiece` object, a colour (`int`, each distinct integer represents a different colour), and an orientation (`int`). The class has getters for each property, and an implementation for the python `__str__` method, which allows us to print the class in the format: `Piece:{piece}, Color: {color}, Orientation: {orientation}`

`Class ChessBoard`	
Normal Behavior	This class represents a chess board. It initializes an empty board and adds default pieces to it. It also provides methods to get the board, get piece locations, create a tile column, add default pieces, and move a piece. This class is meant to represent a single chess game and maintains the state of the game in a 2D array of `ChessTile` objects.
Implement	When initialized, the constructor takes no arguments but calls two functions; `create_board` and `add_default_peices`. These functions create the representation of the board in a 2D array of `ChessTile` objects, and then add the default pieces to the tiles to create the initial board state. The class contains a `move_piece` function that moves a piece from one location to another on the board and updates the board array. Another function `get_piece_locations` returns a 2D array consisting of the piece name and colour in `{piece}-{color}` format, to make it easier for the front-end to process a board state.
Potential Undesired	If the `move_piece` method is called with an end position that already has a piece, the method will return without moving the piece.The `add_default_pieces` method manually adds pieces to the board. If the board size or the number of pieces changes, this method would need to be updated. The `create_tile_column` method assumes that the colour list has three elements. If a different size list is passed, it could lead to unexpected behaviour.

5.4 API Endpoints

`/api/new_game`
• **Normal Behaviour:** This endpoint creates a new game with a unique game ID when a `POST` request is made by a user. • A `POST` request is made to the backend at `/api/new_game`, this endpoint does not expect any data, and returns a JSON with the game ID as a key. • It is a Flask route that generates a unique game ID, creates a new game, and returns the game ID in a JSON response.

`/api/game/[id]`
• **Normal Behaviour:** This endpoint returns the board state of a game when a `GET` request is made, and moves a piece when a `POST` request is made. • `GET` and `POST` requests. For `GET` requests, it returns a JSON object with a 'board' key, containing the most recent game state in the back-end. For `POST` requests, it expects a JSON object with `old_pos` and `new_pos` keys and returns a JSON object with a `'board'` key once the piece has been moved. • It is a Flask route that checks if the game exists, then either returns the board state or moves a piece based on the request method.

`join_room`
This is a web socket event that adds the client to a room when emitted – using built-in Flask functions.

`send_message`
• This is a web socket event that adds a message to a room's message list on the backend as a string array and emits a `'receive_message'` event with the updated message list. • It expects a JSON object with `'room'` and `'message'` keys. It is a Flask-SocketIO event that adds a message to a room's message list and emits a `'receive_message'` event.

`send_move`
• **Normal Behaviour:** This web socket event allows us to update the board state for the opponent once the player has moved a piece on the board. It gets called when the player has made a move and lets the opponents' session know that the board has updated and returns the new game state. It expects a JSON object with `'room'` and `'game_id'` keys. • **Potential Undesired Behaviours:** If the game doesn't exist or the `'game_id'` key is not provided, this event could fail to emit the `'receive_move'` event.

6. User Interface

Throughout our UI we have utilized two main colour pallets. Our primary colour pallet consists of several shades of grey, bright green, and blue. The shades of grey are used for backgrounds, so the user is not distracted or overwhelmed. The bright green and blue are utilized for our call-to-action buttons, so they are more identifiable. The secondary pallet used consists of bright

pink, blue, and yellow. These are used for the tiles of the board and the game logo. The choice of bright colours is for these elements to create a contract against the neutral gray background.

Throughout the application we use two fonts, the main font being “Inter” and the secondary font being “Jockey One”. Our primary font is used for all text except our RhombiChess logo (uses our secondary font), which allows the logo to stand out more. Our text is also structured in hierarchies. Large fonts are used for important headings and smaller fonts are used for body text.

For the placement and general flow of our application, we took some inspiration from similar applications such as chess.com. We decided to include a fixed side panel to help the discoverability of features. On each page, we have displayed the most important information in the center of the page. Below we discuss each page design from our Figma Board.

Landing - Not Signed In
This is the landing page users would be directed to upon opening the application. The navigation panel on the left allows users to choose from options; “Play”, “Rules”, “Account”, “Settings”, and “Help”. Displayed in the center of the page is an enticing description, a link to the Rules page, and two prominent call-to-action buttons: “Play Online” and “Play Local”. Since the user is not signed in these buttons will redirect to the Sign In page. To the right of the main content is a render of the game board, this serves to familiarize users and create intrigue.
Landing - Signed In
Page looks the same as the one described above (in Landing - Not Signed In) with the key difference being the inclusion of the users’ username in the navigation panel indicating they are signed in. The button now links to the Play Online - Landing and Play Local pages respectively.
Sign In: This page allows the user to input their email and password to sign in and play online.
Rules
On the rules page, an image of each chess piece is displayed alongside a description detailing its movements within the game. The side navigation panel is present, retaining the same functionality as outlined previously (in Landing - Not Signed In).
Play Local: This page contains a board where two players can play on the same computer.
Play Online - Landing & Play Online - Game Password Needed
This page allows users to either start a new game or join an existing one. To start a new game, users have the option of what colour they will be, and whether to set a password for their game. Once the “Create Game” button is selected a new game is made and the user gets directed to the Play Online page. Alternatively, to join a game, users must enter a game code and click the “Join Game” button. If the game is not password protected, the user gets redirected to the Play Online page. If there is a password, a “Game Password” input field will appear. After entering the correct password and clicking the “Join Game” button, the user is then taken to the Play Online page. On this page, the side panel remains accessible with the same functionalities.
Play Online
This page is where the user plays the game. The navigation panel remains visible. The main content area allows the user to focus on the board. To the right of the board is an area where game statistics will be kept. Below these statistics will be a live chat.