Multiplayer Othello Game Developed Using React and Phoenix

Team : 318

Yuqing Cheng, Yuan Gao

1 INTRODUCTION AND GAME DESCRIPTION

1.1 Game Introduction

1.1.1 Game rules. Othello, also called Reversi, is a strategy board game for two players, played on an 8*8 uncheckered board. There are sixty-four identical game pieces called disks (Black and White). Players take turns placing disks on the board with their assigned color facing up. During a play, any disks of the opponent's color that are in a straight line and bounded by the disk just placed and another disk of the current player's color are turned over to the current player's color.

Only placement of disk that can flip opponent's disk can be considered as a valid placement, otherwise the player cannot put disk at this position. If in a player's turn, no valid placement exists under the situation, it will automatically becomes opponent's turn. If both players don't have a valid placement, the player with more disks win.

1.1.2 Game object. The object of the game is to have the majority of disks turned to display your color when the last playable empty square is filled.

1.2 Design Introduction

1.2.1 Join/Create/Speculate a game. There are two ways for a logged-in user to join an existing game. He can either click the join game button on the index page and type in the existing game name if he/she knows, or he can enter the game lobby by clicking the game lobby button. There are two kinds of game, one is games already started, which other users can speculate, the other is games still wait for matching, which other users can join to play against others.

1.2.2 Speculators and Players authority. We allow anonymous user (who don't log in) to speculate the game that has started but that user will not be shown in speaculator list in game page. But one user is only be allowed to join or create the game when he/she is logged in.

1.2.3 Priority for current players. When a player creates a new room, the game board is locked, he must wait for an opponent to join to start the game. After second player enters the room, the board will be unlocked, first person enter the room will move first. After the game start, other users can speculate the game. When one of the players leaves the room, the room will still exist, while when both players left the room, the room will be deleted and all the speculators will receive an alert and be redirected to index page . That's why we call the design choice priority for players, as they are the key roles of the game.

2 UI DESIGN

There are four main pages in our game.

2.1 Index Page

Index page contains the entry to log in, create game and enter game lobby.

- When click log in, the app will redirect user to log-in page.
- When click join game button, user can create a new game or join an existing game by entering the game name. When hit enter, the app will redirect user to game page.
- When click game lobby button, the app will redirect user to lobby page.

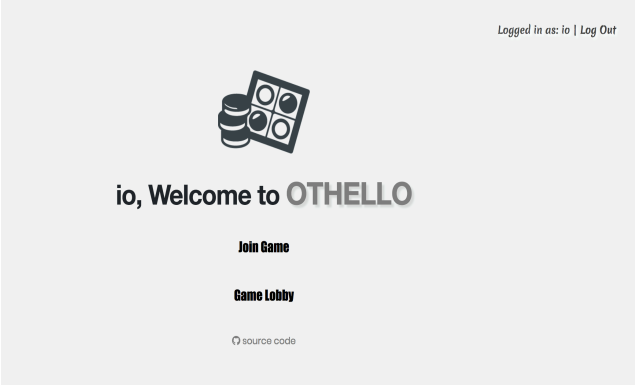

Figure 1: `Index Page`

Figure 2: `Join Game Button`

2.2 Log in Page

Log in part is very basic. User can just enter the name without password to log in.

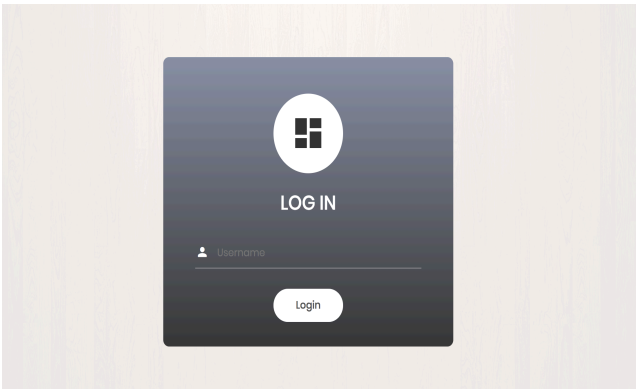

Figure 3: `Log in Page`

2.3 Lobby Page

The information shown on the lobby page including game name, game status(user is waiting for a match/the players currently playing), number of speculators, and a column indicates choice of joining or speculating one game. Notice that when the lobby is empty, the user receive empty lobby alert instead of being directed to empty lobby.

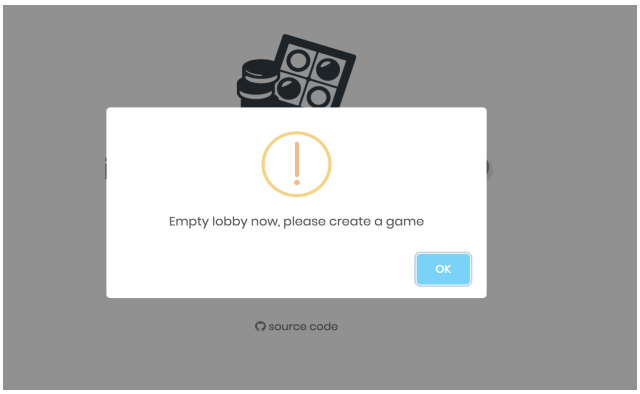

Figure 4: `Empty Lobby`

2.4 Game Page

In the game page, there are three main sections.

- Header shows the game info, including game name, and game status(waiting for a match or players name who are currently playing).
- Side bar shows all the speculators' names and has one exit button for a player to exit a game.
- Game board uses react-konva component. The disc uses Circle, the board uses Rect, and board line uses Line. On right side of board shows the ownership of current move, and number of white and black discs.

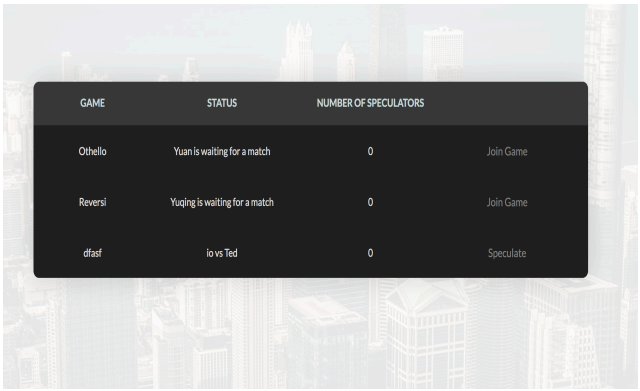

Figure 5: `Lobby Page`

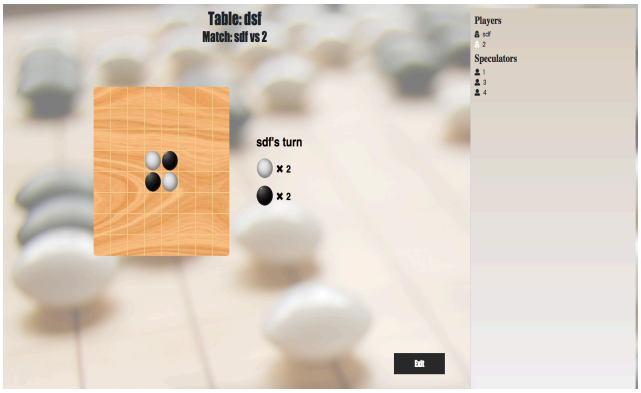

Figure 6: `Game Page`

3 UI TO SERVER PROTOCAL

In this project, there are two types of connection from UI to Server. For page rendering and user login, the connection is supported by Phoenix controllers invoked from the router in response to HTTP requests (as used in Tasktracker homework). For game joining, game state update and broadcasting, the connection is supported by websocket implemented by Phoenix channel, which provides a bidirectional communication from UI to server.

3.1 HTTP request and Controllers

The paths under router.ex are shown as following:

```
get "/", PageController, :index
get "/login", PageController, :login
get "/lobby", PageController, :lobby
get "/game/:game", PageController, :game

post "/session", SessionController, :create
delete "/session", SessionController, :delete
```

PageController handles all page requests, and SessionController only handles login/logout operations. PageController is invoked as corresponding function calls by Router, and corresponding templates are rendered.

SessionController is invoked in login template and uses put_session to assign username to window session. In router.ex, we also added a plug :get_current_user, by which the username can be retrieved from session in any template. Based on this, we can use <script> in game template to assign session username to window, by which a socket can obtain session username for later use in channel.

3.2 Websocket and Channel

In user_socket.ex, we defined channel as "gamechannel:*", where the "*" stands for specific name of game table. Therefore a instance of channel stands for a specific game table.

When a user joins a game (no matter as a player or a speculator), the user just provides the channel name that stands for the name of game table, as well as the socket that contains username originally from session. The join function in game_channel.ex will invoke module in game.ex to handle the real joining logic, e.g. whether the game exists or needs to be created in Agent, or whether the user joined as a player or speculator...

We will cover in more details about the data structure on server in later part of the report, but to illustrate the payload structure, we just put our server-side data structure here:

```
%{
 colors: [],        # a list of numbers of length 64
 turn: 0,           # a number to indicate who's turn, the
                      number is the index in players
 players: [],       # a list of string that stands for
                      players
 winner: nil,       # a number that stands for the winner,
                      nil if no winner
 speculators: [],   # a list of string that stands for
                      speculators
 online_players: 2  # number of players remained online in
                      this table
 }
```

The above data structure stands for a state of game in a game table, which has uniform structure through UI to Server, and we will call it "state" or "game state" in later parts for simplicity. After a join function is called, it wraps and returns a response through socket in a structure like this:

```
%{"state" => state, "msg" => "message to be displayed",
     "type" => "optional type"}
```

where state is the current game state, which has a structure we discussed above. The "msg" stands for the message to be displayed at front-end to notify a new user/speculator has joined/left. "type" stands for what kind of notification or dialog(success, warning or error) should be displayed.

Apart from a direct return, the join function also triggers handle_info (:after_join, resp, socket) to be invoked after joining, where resp is identical with the reponse passed to socket. In this handle_info function, we broadcast resp to every socket that subsribed to this channel (this game table). In this way, every player and speculator will be notified that a new player or new speculator has joined this room/table, or a player comes back to this room/table. This broadcast has a label "new:user", and the callback in front-end's channel.on("new:user") will be called to determine how the notification will be executed according to the resp["msg"] and resp["type"].

When a player clicks a position on game board, the UI pushes "move" through the socket to server by channel.push("move", index: index), where index is the index in the 1-D array of colors calculated by x and y coordinates. Therefore, the index in 1-D array is the only variable passed to server in order to make a move. In game_channel.ex, handle_in("move", %"index" => i, socket) will be invoked and passed i to Game.simulate_next_state(curr_name, i) in game.ex, curr_name is the current table name that can be retrieved from socket. The Game.simulate_next_state/2 function returns a tuple boolean, resp, where the boolean indicates whether this move is a valid move, and resp has a structure like this:

```
%{"state" => state}
```

where state is the current game state. If current move is a valid move, resp will be broadcasted to every socket subscribed, and :ok, resp will be returned. If move is invalid, resp will not be broadcasted, and :error, resp will be returned. In client-side, if channel detects "error" in call-back, the player will be notified that the current move is invalid.

When any user leaves the game table (close browser, log out or go to another page), terminate(reason, socket) will be called. The terminate/2 determines whether the game table should persist and what kind of message should be broadcasted, the broadcasted payload has a structure like this:

```
 %{"msg" => msg, "type" => type}
```

4 DATA STRUCTURES ON SERVER

As discussed before, the game state on server has a structure like this:

```
%{
  colors: [],      # a list of numbers of length 64
  turn: 0 or 1,    # a number to indicate who's turn, the
                     number is the index in players
  players: [],     # a list of string that stands for
                     players
  winner: nil,     # a number that stands for the winner,
                     nil if no winner
  speculators: [], # a list of string that stands for
                     speculators
  online_players: 2 # number of players remained online in
                     this table
  }
```

More specifically, each element in the color list stands for the occupation or color on this position, 0 represents unoccupied, 1 represents black disc, and 2 represents white disc. The list of players and list of speculators are both lists of usernames. The "turn" attribute is the index in players, the username of owner of this turn is players[turn]. The "winner" attribute uses the similar design, players[winner] is the username of the winner, but apart from 0 and

1, "winner" attribute can also be 2, which stands for a tie situation when game ends. The online_players stands for the number of online players remained in this table, when online_players becomes 0, the game table will be closed and deleted as introduced in our design part.

Game states all persist on Agent in server. The Agent in our project has a structure like this:

```
%Agent{games: %{game_name: state}}
```

Under this design, Agent.get(:games, &(&1)) will get all game states in a form of map, and this map can get a specific game state using the game_name as key. The reason why we chose not to directy use game_name => state as key-value pair under Agent is that, in order to implement game lobby, we want the whole list of game states, and Agent.get(:games, &(&1)) can simply achieve this. The trade-off is, this makes us need to get all game states before retrieving a specific state, and each game state has O(n) complexity, where n is number of users (players and speculators list), if there are m rooms, the space complexity can be O(mn). However, as this is a course project, even if we have 1000 users and 1000 rooms, the worst-case memory used is just O(1000,000), which can actually use only MBs of memory, therefore making our design totally acceptable.

5 IMPLEMENTATION OF GAME RULES

All game rules are implemented under a single module in game.ex. Apart from game rules, game.ex also handles how game state would change, whether a user should be a player or a speculator in a joining or leaving game event, as well as winner check. In general, it handles all behind-the-scene app logic/intelligence, which is the "model" portion of a MVC pattern.

As for game rules, imagine we have a game board, each time a player clicks a position on this board, the algorithm would check in 8 directions (up, down, left, right, up-left, up-right, down-left, down-right) from this position to determine whether there exists a disc in the same color so that all opponent's discs along the path could flip. Since Elixir is a functional language, we implemented the above algorithm using recursion in the function check_swaps/4. Besides check_swaps/4, there is in_bound/2 function working with it to make sure not going out of bound. Also, valid_next/2 function is used to filter the cases when the next adjacent disc is in the same color, i.e. a valid move should flip at least one opponent's disc.

It's worth noticing that in Othello, in some cases, in one player's turn, the player cannot find any valid move in order to flip opponent's discs. According to many examples of Othello games, they handle this problem by simply giving the current turn to this player's opponent, and we adopted this game rule in our design. Futhermore, even if the opponent take over the turn, the opponent might also not have a valid move, in this case, as both players don't have a valid move, we will start counting number of discs to produce and announce a winner. In our algorithm, a valid next move check will be done after one player successfully makes a move. In the valid next move check, every spare place on board is checked to determine whether the opponent can make a valid move here, if no such place exists, the turn of game state will remain the same and this will be notified at front-end. If current player cannot move, either, a winner will be determined and announced. The time complexity of this process might sound high, i.e. if total positions on board is N, our algorithm is approximately $O(N\hat{2})$, but there are totally only 64 positions on board, that is, the input size of our algorithm is very small in nature, so the server would definitely afford an "inefficient" algorithm, not to mention it's still poly-nomial.

Check winner is also an important part of game rules. In most cases, players fill up all spaces on board, and player with more discs on the board becomes winner. Besides, as discussed above, if both players cannot make a valid move, the player with more disc on the board becomes winner. Moreover, a special case of the second case is that when all colors of discs on board become identical, this should be combined in second case. These logics are implemented with get_num_colors/3 function to count the number of discs in each color. Similar to check_swaps/4, get_num_colors/3 is also implemented using recursion.

6 PROBLEMS/CHALLENGES AND SOLUTIONS

6.1 Server-side update on every client

Every time the server side has been updated, every client connect to the same game should see the change. This can be done with broadcast in Phoenix.Channel.

6.2 Speculator number increase when refresh

When we implement speculators function, the number of speculators will increase when the speculator refresh the page even the actual number of speculators remain the same. That's because at first we just append the user_name to speculators when the user_name exists. Then we add a judgement that if the speculator is already in current_game speculators, we just move on and do nothing.

6.3 Game page login style missed

Sometimes when user log in from lobby page, the style is incomplete, this can be fixed by changing all the url in index page to absolute path. Like **href="/images/icons/favicon.ico"** instead of **href="images/icons/favicon.ico"**.

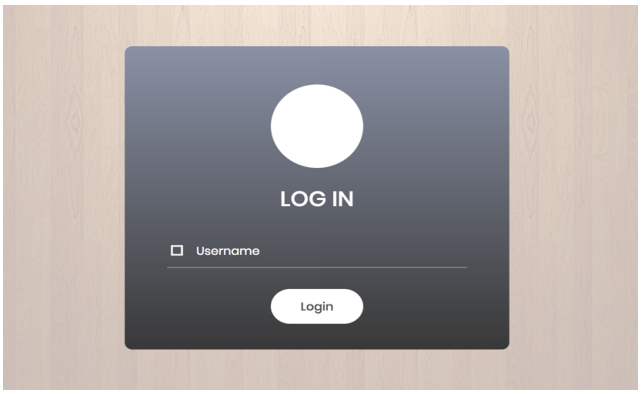

Figure 7: `Missing style`

6.4 Empty lobby

We redirect user to index page when lobby is empty. While when another user created a new table, the user who has been redirected to index page has to refresh before clicking into the lobby page. As @gamenum we count on the backend is only loaded for once in index page. Solution is that the judgement of gameNum can be moved to lobby page (.html.eex), if in lobby page we find gameNum is 0, just render index page.

6.5 Submit form when hit enter

Index part is written in react, and we want to submit form by hit enter instead of click on the button. We need to write the form onSubmit function and change the button type to submit.

```
<form onSubmit={function(e) {
  e.preventDefault();
  window.location=link;
  }}>
  <div className="input">
    <input onKeyUp={this.handleChange} placeholder={'Enter
         game name'}/>
      <button id="enter" type="submit" color='success'>Join
           Game</button>
  </div>
</form>
```