Web Development Project 1:

Team Members: Jiangyi Lin, Neha Shewani

Game: Checker

- **Introduction and Game Description:**

We have implemented a two-player game - Checker. It includes features such as: register, login and play checker with another player. A user may create a room and wait for other user or they can join the room created by another player. When the main two players are playing checker, other users can watch their match, but only the two players can move the pieces.

Initially, at the loading of the game, user will see a login screen, where he can enter his user name and login into the game. The initial game state that will be visible to the players will be: each player has twelve pieces, arranged on the diagonals of an 8 x 8 square board. The player can see his/her name and piece color as well as his opponent's. The red-color pieces move first. As the game advances the players will be able to see how many checker pieces they have lost.

The players will get the chance alternatively. The user will be moving the pieces with the help of mouse. The pieces can only move forward one tile diagonally. If one of the piece gets to the opposite side of the board (opponent's side last row), it will turn into King where the king can move in any direction diagonally. They can combine forward and backward on the same turn.

To capture the opponent's piece and to remove it from the board, the player can jump over their piece with the other player's.

The player wins by removing all the opponent's pieces from the board, or if the opponent can' make the move.

- **UI Design**

Login Page: The login page is the initial page of the game when we load our game. This page shows the user the introduction to checkers and has the login details. The user can register to the application in case of new users or login to the game if the user already has created an account on the application.

Rooms Page: After you login into application, you get redirected to the rooms page. This page lists the rooms which are already present (created by other users). The user can join any of the available rooms and start the play. If the user does not wish to join a room, he/she can create a new room where in he/she becomes the red opponent in the game. If the user joins an already existing room, he/she would be black opponent if the he/she is the second player to join or will be an observer if there are already two opponents in the room. The list also gives the information about room name, the ID of the red opponent, the ID of the black opponent and the join button for that room.

Game Page: Once the player joins the game, we notify the information about which player it would be and who is his opponent. If there are two players in the game, we notify that the game can start. In case of only 1 player in the game, he/she is notified with the information to wait for the other opponent to join the game.

On the game page, we can see an 8 x 8 checkerboard and the 24 checker coins (12 red and 12 white) which are placed on the checkerboard. We highlight the coins of the checkers for the player who has the chance to play. On the player's chance, the player sees the highlighted coins which can make a move with the white highlight. Once the user clicks on the coin, the clicked coin is highlighted to yellow, and its valid move is highlighted with green. If there exists a kill move, the move is highlighted with the red circle. There exists a resign button, if the user wants to leave the game in middle, the use is notified with an alert the he/she has lost the game.

- **UI to Server Protocol**

Client uses web socket to communicate with the server. There are two type of channels server handles: global channel and game channel. On browser, the JavaScript code creates socket and channel object, and can send information to server or listen message sent from the server.

Global channel: Global channel is used to get the current games and create new games. If server get request for current games information, it will check information from Supervision tree about how many alive GenServer processes there are. Then it sends back a list of game id, game information like red player and black player (user id or empty), which will be received and used in page of our current rooms. If the server is asked to create a new room, it will check the room name is not used and start a new GenServer for this new game and broadcast all users looking at the room list for this new room. The creator will be the red player and redirected to the game page.

Game channel: Game channel is used to handle with the essential game information between server and two players and viewers. Every user in the room will receive the real time state of the checker board by listening message from the server. When the player is in his/her turn, he/she can make one valid move and update the state of the game. In server, the code of channel can rightly send this request to right GenServer and more details of how to check the validation and change the state is at data structures on server part. The player can also send a resign request if he/she wants to end the game in the middle of the game. Server will handle that message, broadcast the result and end the GenServer.

- **Data structures on server**

Each game relates to one GenServer. The main work in GenServer is check the move request, if it is valid and then change the state, which may lead to one player wins, else will reject the request and send back the reason.

The Structure contains the game id, red play's id, black player's id (nil if not exists), the list of viewers' ids, a turn used string to work as a flag and a list with 32 elements indicating on that board tile there is red solider, black solider, red king, black king or empty.

Server will make sure every player obeys the rules. The player can only make move when it's his/her turn. Red player can only move red pieces and black player can only move black pieces. The piece an only move to empty position and jump through an enemy to an empty position. Based on the original position and board state with nice Elixir library like Enum, server can validate the request fast. For normal situation, the turn will flip, the board will be changed to new state rightly. All situations like normal piece only move forward, king can move forward or backward, eaten piece will be removed and solider will be lifted to king when it reaches the innermost row are handled by well-designed functions which only use the board data in the server, so the player can only defeat his/her opponent by his wisdom on checker instead of trying to cheat. If server find opposite player loses all pieces, it will open a new thread to handle the result and broadcast to everyone connect to this game room through channel. Besides, a player may get the right to make a multi-jump, which means when the piece jump through an enemy, if it can have another valid jump, the user can choose to do it or not, if it use this right, then the piece will check this multi-jump rule again, until the piece cannot make more multi-jump or the user decide to stop here. Then the turn, instead of only can indicate red turn or black turn, will have another value to show it's a continue turn, and the piece's position so that player can only move that piece to jump instead of making other moves to break the rules. Using turn as a flag makes server part's code clear

- **Implementation of game rules**

Board State: We are maintaining a 1D array of 32 items since these are the only positions where a checker coin can move. Each user is assigned a role of player or viewer. The first two users to enter a room become the players whereas rest of the users to join the same room get a role of viewers who can just observe the game.

Room: users can create game rooms or join an existing one. Based on their role they can either be a player in a room or a viewer. There are no restrictions on how many rooms a user can join.

Moves:

a. Valid Moves: Each coin has a valid move of diagonal cells. Black can move forward from row 1 to row 8 and red can move from row 8 to row 1. The moves are dependent on the row being even or odd. Also, while computing a valid move, we check the row numbers. For black the row number should be +1 than current tow. For red it should be - 1 than current row. For queens we check the valid moves should be either +1 or - 1but not the same rows.

b. Kill moves: each kill moves have to be in +2 row from the current row. Write the extra logic.

Each move is assigned an index from the 1D array. Once the user selects a move on the board, the ultimate change is reflected in the 1D array. When a coin is moved from one position to another, a request is sent to the server which will validate and give the final result back. If the moves is valid, then the change is reflected on the board.

User can select any of the white highlighted coins. And then its corresponding moves will be reflected.

- **Challenges and Solutions**

First problem is thinking a way to have many thread for games and the server can handle each game thread through some unique id. However, Phoenix is a great framework with a powerful language Elixir, so some modules like GenServer and Supervisor is really helpful. We can send message to GenServer to change its state or make it shutdown, which is just the process of game state change and ending the game. A simple one for one strategy fits our needs of dynamic games. Then the project's server architecture is clear. Building Phoenix Battleship gave us a nice example to learn.

How to render a board on the browser is our second problem. There are many well-designed checker games. Our project learns its CSS and the resources (of course, we add attributions for all of them). Then the main part is write the real code for JS and Elixir.

We need to know nearly everything to make a great web application, however, to start the first step, only a clear brain and patience in debugging and reading related document is needed. To finish this project, the documentation for JavaScript and Elixir's syntax and convenient libraries is always read. Using the knowledge of how to write beautiful code learnt from PDP and other beneficial experience, the project is something has a definite purpose, with well-designed and easy-to-add architecture, left many small and clear functions to write one by one, finished bottom to up. There will be bugs, of course, and trying to debug in communicating between server and client, of some special action in the middle of the game is something most troublesome. Luckily, they are all resolved at last, and the experience in coding and debugging become helpful when new but similar bugs come out in the future.

Last but not the least, using GitHub and git to collaborate is also a beneficial practice. Commit helps us to manage the schedule, and we do not need to save a zip package at somewhere and then add new features. Using branch and merge is also interesting and worth to try. Solving the conflict is also learned during this project.